Universidad Politécnica de Cartagena
Dpto. de Tecnologías de la Información y de la Comunicación.

Evaluación y desarrollo incremental de una arquitectura software de referencia para sistemas de teleoperación utilizando métodos formales.

Tesis Doctoral

Juan Ángel Pastor Franco

2002

Directores:
Dña Bárbara Álvarez Torres
D. Andrés Iborra García.

Abstract

Teleoperation systems cover a wide range of missions and mechanisms, and each must satisfy a number of specific requirements. The thesis deals with the evaluation and redesign of a reference architecture for robot tele-operation systems, using respectively the methods ATAM (Architecture Tradeoff Analysis Method) and ABD (Architecture Based Design). Among the main contributions of the thesis may be included:

- Characterization of quality attributes of the teleoperation systems.
- Assessment of a reference architecture using the ATAM method.
- A functional decomposition for the considered systems
- A requirement specification for a component model applicable to teleoperation systems.

The main design drivers for the development of software architectures in the teleoperation are to provide designers with strategies that separate the patterns of interaction between components from their functionality. These strategies should be the basis to define a development framework to define specific architectures aimed to fulfill the specific requirements of the different systems. It is much more interesting to develop such a framework that to attempt to define a single architecture for the entire domain.

Los sistemas de teleoperación cubren una amplísima gama de misiones y mecanismos, y cada uno de ellos debe satisfacer una serie de requisitos muy específicos. Este trabajo de tesis se ocupa de la evaluación y re-diseño de una Arquitectura de Referencia para sistemas de teleoperación de robots de servicios, utilizando respectivamente los métodos ATAM (Architecture Tradeoff Analysis Method) y ABD (Architecture Based Design).

Entre las principales aportaciones de la tesis pueden mencionarse:

- *La caracterización de los atributos de calidad de los sistemas considerados.*
- *La evaluación de una arquitectura de referencia utilizando el método ATAM.*
- *La propuesta de una nueva descomposición funcional para los sistemas considerados.*
- *La especificación de requisitos para un modelo de componentes aplicable a este tipo de sistemas.*

Las principales conclusiones del trabajo de tesis pueden resumirse en.

- *Deben adoptarse estrategias de diseño e implementación que permitan separar por un lado los patrones de interacción entre componentes de su funcionalidad y por otro los diferentes aspectos de dicha funcionalidad.*
- *Es mucho más interesante definir un marco de desarrollo en el que puedan definirse diferentes arquitecturas que intentar definir una sola arquitectura para todo el dominio.*

Directores:
Dña Bárbara Álvarez Torres
D. Andrés Iborra García.

Universidad:
UPCT

Fecha de lectura:
8 de julio de 2002

Tribunal:
Isidro Ramos Salavert (presidente)
Juan Antonio de la Puente Alfaro
Alejandro Alonso Muñoz
Francisco José Rodríguez Urbano
José María Fernández Meroño.

Dirección y datos de contacto del interesado:
Juan Ángel Pastor Franco
C/Jorge Juan 22,
30204 Cartagena, España

Palabras clave:
Arquitecturas de referencia, evaluación de arquitecturas, ATAM, sistemas de teleoperación.

Agradecimientos

En primer lugar, deseo expresar mi agradecimiento a Bárbara Álvarez y Andrés Iborra, directores de esta tesis, por sus innumerables contribuciones en mi formación como investigador y por la confianza que han depositado en mí. Esta tesis es el resultado de un trabajo realizado codo a codo con ellos.

Con la misma intensidad extiendo mi agradecimiento a las muchas personas que de un modo u otro han contribuido a hacer posible este trabajo de tesis. En especial:

- ✓ A mi mujer, Pilar, y a mis hijas, Cristina y Sara, por su apoyo incondicional e infinita paciencia.
- ✓ A Pedro Sánchez Palma. Siempre dispuesto a echar una mano y a sacrificar su tiempo, sus muchas sugerencias han contribuido decisivamente a la realización de esta tesis.
- ✓ A Carlos Fernández, Paco Ortiz y Alejandro Martínez por su apoyo y entusiasmo.
- ✓ A toda la gente del Área de Lenguajes y Sistemas de esta Universidad por las múltiples ayudas que he recibido de ellos.
- ✓ A todo el personal de ENWESA Operaciones S.A., y, muy especialmente, a Lidia Alcedo y Antonio Benito, con los que he pasado casi cinco años desarrollando sistemas de teleoperación. Ha sido un auténtico placer y un privilegio haber trabajado con ellos.
- ✓ A mis padres, por tantas y tantas cosas.

A mis padres.

A mi mujer, Pilar

A mis hijas, Cristina y Sara

Contenido

1 Planteamiento y Objetivos de la Tesis.

1.1 Introducción

El diseño de cualquier sistema de cierta complejidad requiere conocer y seguir disciplinadamente una serie de reglas bien definidas que indiquen como ensamblar y conectar los diferentes elementos del sistema y cómo gestionar el desarrollo y evolución del mismo. Sin dichas reglas el diseño se convierte en un proceso caótico que en el caso del software se agrava por estar sujeto a modificaciones a lo largo de su ciclo de vida. Por esta razón, la Arquitectura Software se ha convertido en una de las disciplinas más importantes de la Ingeniería Software, particularmente en el ámbito del desarrollo de grandes sistemas. A pesar de ello, no existe un consenso general acerca del alcance y significado de la Arquitectura Software. Una definición que, por razones que se explicarán más adelante, se adapta especialmente bien a los propósitos de este trabajo de tesis, es la ofrecida en [Bass et al 1998]. Dicha definición dice así:

"La arquitectura software de un programa o sistema de computación es la estructura o estructuras del sistema, incluyendo sus componentes software, las propiedades externamente visibles de dichos componentes y las relaciones entre los mismos".

Estructura, componentes, relaciones. Estas son las tres palabras clave de la mayoría de las definiciones de Arquitectura Software. Garlan [Garlan et al 1995b] añade:

"*…y los principios que gobiernan su evolución a lo largo del tiempo",*

subrayando el hecho de que un sistema software es un *ente* vivo, sujeto a cambios y modificaciones a lo largo de su vida operativa. La Arquitectura Software de un sistema define sus estructuras y establece a través de ellas las líneas básicas de su diseño y de su proceso constructivo, determinando la mayor parte de los *atributos de calidad* que pueden esperarse del mismo. La definición de la arquitectura de un sistema es un proceso muy arduo, que debe tener en cuenta factores técnicos, sociales y de negocio. Sin embargo, una vez definida, la Arquitectura ofrece una representación global del sistema que puede ser entendida con relativa facilidad por todas las partes interesadas en el mismo.

Diferentes sistemas exigen diferentes arquitecturas. La arquitectura de un sistema puede verse como el resultado de una serie de compromisos de diseño que intentan conciliar requisitos muy diferentes y a menudo contradictorios. Una arquitectura será mejor o peor en función de su capacidad para conseguir el equilibrio adecuado entre todos los atributos de calidad que se le piden al sistema [Boehm 1996]. Cada arquitectura adopta un conjunto de estilos o patrones en función de los atributos de calidad que desee optimizar. Durante la década de los 90, se han

documentado una gran cantidad de estos patrones y estilos que ponen a disposición del arquitecto software soluciones probadas a problemas comunes, reduciendo el grado de incertidumbre que siempre lleva asociado el diseño de un nuevo sistema. Ahí están los trabajos de Gamma [Gamma et al 1995], Buschmann [Buschmann et al 1996], Klein [Klein et al 1999a, 1999b], Douglas [Douglas 1999] y otros muchos que se citan a lo largo de esta tesis.

Pero si por un lado es cierto que cada sistema tiene sus propias particularidades, también lo es que muchos sistemas comparten un gran número de características y deben satisfacer una serie de atributos de calidad comunes. Diversos autores proponen el estudio y clasificación de las arquitecturas software existentes para una familia de sistemas para identificar una arquitectura que sirva de referencia para todos ellos (*Domain Specific Software Architecture*). La *Arquitectura de Referencia* define la infraestructura común a todos los sistemas del dominio considerado, los componentes o subsistemas que incluyen y las interfaces que deben ofrecer dichos componentes y subsistemas. Una vez definida, la arquitectura de referencia puede ser ejemplarizada para crear arquitecturas de sistemas específicos. Disponer de tal arquitectura facilita enormemente el desarrollo de nuevas aplicaciones, pues permite la reutilización de modelos y componentes, cuyo desarrollo se hace posible por la existencia de tal arquitectura[1].

Este trabajo de tesis se ocupa precisamente de una *Arquitectura de Referencia*, en concreto la definida por Alvarez en [Alvarez 1997] para un subdominio de los sistemas de teleoperación. Los sistemas de teleoperación cubren una amplísima gama de misiones y mecanismos, y cada uno de ellos debe satisfacer una serie de requisitos muy específicos. Sin embargo, más allá de estas diferencias, todos ellos comparten muchas características que agrupadas definen un dominio sobre el cual es posible describir una arquitectura software de referencia.

Los sistemas de teleoperación se utilizan para teleoperar mecanismos (robots, vehículos y herramientas) que llevan a cabo actividades de inspección y mantenimiento en entornos cuyas condiciones de habitabilidad o riesgo desaconsejan o impiden la presencia de operarios humanos. En general, los trabajos que realizan tienen un grado de complejidad que no permite dotarles de una total autonomía. Sus actividades, incluyendo las tareas automáticas y las secuencias preprogramadas, deben ser monitorizadas por un operador humano situado en zonas seguras que, en todo momento, debe tener al sistema bajo su control. Ejemplos típicos son los robots utilizados en trabajos de mantenimiento en centrales nucleares y en plantas químicas y, por poner un ejemplo más popular, los robots utilizados para desactivar artefactos explosivos.

La teleoperación, a lomos de los avances de la robótica, es una disciplina en auge. Trabajos que anteriormente no se realizaban o se llevaban a cabo en condiciones penosas son ahora posibles mediante el uso de mecanismos que pueden ser operados remotamente desde zonas seguras. La preocupación por las condiciones de trabajo de los operarios y por la seguridad de las instalaciones son cada día mayores y en consecuencia aumenta el número de posibles aplicaciones. Sin embargo, aún queda casi todo por hacer. Contrariamente a la imagen que tiene el gran público sobre la robótica, más cercana a la ciencia ficción que a la realidad, los robots sólo pueden ejecutar de forma segura y eficiente labores relativamente sencillas, como la ejecución de ciertos movimientos previamente definidos utilizando sistemas de referencia perfectamente calibrados. De ahí el éxito de los robots industriales utilizados en procesos de fabricación. Pero cuando se trata de realizar trabajos de cierta complejidad o el robot debe trabajar en entornos no estructurados, donde la calibración es difícil o imposible, las cosas cambian radicalmente. Existen muy pocos sistemas de este tipo que hayan probado su eficacia y ninguno de ellos se libra de la existencia de un operador humano que supervise su funcionamiento y corrija sus errores.

[1] La relación también puede definirse en sentido contrario. La disponibilidad de ciertos componentes puede forzar o condicionar el diseño de la arquitectura.

En este trabajo de tesis se aborda la evaluación, corrección y refinamiento de la arquitectura de referencia descrita en [Alvarez 1997] con los siguientes propósitos:

- Comprobar su conformidad con los requisitos de calidad originales.
- Comprobar si es factible su utilización para el desarrollo de sistemas de teleoperación en entornos no estructurados, que requieren la incorporación de utilidades y servicios no contemplados en la formulación de la arquitectura.
- Comprobar si puede utilizarse para la definición de sistemas cooperativos, en los que diferentes sistemas teleoperados deben realizar conjuntamente ciertos trabajos con cierto grado de autonomía.
- Incorporar a la arquitectura paradigmas y principios propios del desarrollo software basado en componentes y proporcionar un marco de desarrollo de componentes software y hardware reutilizables que puedan servir de base para el desarrollo de nuevas aplicaciones o para la mejora o modificación de las existentes.

En lo que resta de capítulo se presentarán:

- La motivación o justificación de los trabajos.
- Los objetivos de la tesis.
- Las metodologías que se proponen para conseguirlos.
- La estructura de la tesis.

1.2 Motivación

La arquitectura definida en [Alvarez 1997] ha permitido el rápido desarrollo de sistemas de teleoperación para entornos estructurados, demostrando todas las ventajas que supone disponer de una arquitectura de referencia. Dicha arquitectura ha sido empleada con éxito en la implementación de varias aplicaciones en el marco de diferentes proyectos. En concreto:

- *Proyecto "Aplicaciones Automáticas para la Reducción de Dosis"* [AAA 1996a].
  - ✓ Sistema de teleoperación para el robot ROSA III de Westinghouse, utilizado para la reparación de generadores de vapor de centrales nucleares de agua a presión. Se sustituyó el software de teleoperación existente y se incluyeron nuevos procesos y herramientas [Alonso et al 1997], [Alvarez et al 2000a, b, c], [Pastor et al 1996], [Iborra et al 2002].
  - ✓ Sistema de teleoperación para el vehículo IRV para la inspección y recuperación de objetos de toberas y tuberías inundadas [Pastor et al 1998].
- *Proyecto "TRON"* [TRON 1996, 1997a, 1997b, 1998].
  - ✓ Sistema de teleoperación de una pértiga articulada para la inspección y recuperación de objetos del fondo de los internos inferiores de las vasijas de centrales nucleares de agua a presión [Alvarez et al 2000a, b, c], [Iborra et al 2000, 2002], [Pastor et al 2000].
- *proyecto "GOYA"* [GOYA 1998a]
  - ✓ Sistema de teleoperación para un conjunto de mecanismos y herramientas utilizados para la limpieza de finos de embarcaciones de grandes dimensiones [Ortiz et al 2000].

Cada una de las aplicaciones arriba mencionadas pudo implementarse con relativa rapidez y sencillez. Fué posible emplear componentes software genéricos configurándolos con los

parámetros específicos de cada mecanismo, reutilizándose en algunos casos subsistemas completos. Siendo esto así, es inmediato preguntarse las razones por las que hay que corregir y reformar una arquitectura que ha demostrado su validez al ser utilizada en el desarrollo de sistemas de muy diverso tipo. Estas razones pueden agruparse en tres grandes grupos, que se describen a continuación:

1. Debilidades de la arquitectura detectadas durante el desarrollo de los sistemas.
2. Formulación de los requisitos de partida.
3. Formulación de nuevos requisitos, enfocados a ampliar el dominio de la arquitectura para contemplar entornos no estructurados y a posibilitar la cooperación de diferentes sistemas teleoperados con cierto grado de autonomía para llevar a cabo ciertos trabajos.

1.2.1 Debilidades de la Arquitectura

Aunque la arquitectura definida en [Alvarez 1997] ha servido de marco para el desarrollo de distintos sistemas, también ha mostrado una serie de debilidades o carencias, de las que cabe destacar las siguientes:

- En aquellos sistemas muy exigentes en cuanto a cálculos cinemáticos y representaciones gráficas [AAA 1996a], los componentes encargados de realizarlos se encontraban en ocasiones desbordados, no presentando un comportamiento temporal aceptable y propagando su mal comportamiento al resto del sistema. En el caso de los gráficos, el problema puede *resolverse* empeorando la calidad de los mismos o actualizando la imagen a intervalos más largos. Sin embargo, esta *solución* no puede aplicarse a los cálculos cinemáticos, que se utilizan para comparar los estados real y esperado y para detectar o prevenir colisiones. El problema puede tener su origen en la calidad de los algoritmos empleados, en el rendimiento de los componentes comerciales utilizados o en la capacidad de procesamiento de las plataformas, pero también puede ser intrínseco a la arquitectura, lo cual implicaría una revisión de los estilos empleados en la misma.
- Los componentes genéricos son de grano muy grueso, lo cual dificulta su reutilización entre sistemas muy diferentes.
- La adición de nuevas operaciones que requieren sincronismo entre el brazo y la herramienta acoplada a su extremo, o la modificación de las ya existentes, implican una cascada de cambios en los controladores que gobiernan o monitorizan sus comportamientos.

Ninguna de estas deficiencias supuso un gran impacto en el desarrollo de dichos sistemas por diversas razones. En el proyecto *Aplicaciones Automáticas* [AAA 1996a], todos los brazos eran muy parecidos (diferentes versiones del ROSA III) y los parámetros del controlador genérico representaban fielmente las características de todos ellos. Sin embargo, en algunos de los siguientes desarrollos, aunque la arquitectura seguía ofreciendo un modelo válido, dichos controladores no pudieron reutilizarse, no ya su implementación, ni siquiera su interfaz, dadas las notables diferencias entre los mecanismos considerados en unos y otros proyectos. Esto no supuso un gran inconveniente ya que en dichos proyectos la mayor parte del control se realizaba en el controlador local y la implementación del sistema de teleoperación era relativamente sencilla. Por último, la rigidez de la arquitectura ante cambios en las misiones tuvo un impacto limitado ya que, por un lado, en todos los proyectos existía una definición muy precisa de las

operaciones que debían llevarse a cabo, y por otro, el sincronismo entre brazo y herramienta era en todos los casos muy sencillo. Brazo y herramienta no trabajaban simultáneamente, limitándose la sincronización a una cesión de control, en la que un mecanismo se detenía cuando debía actuar el otro.

Sin embargo, estos aspectos suponen riesgos potenciales para el uso de la arquitectura, ya que no puede suponerse que en todos los sistemas la reimplementación de los controladores de la plataforma de teleoperación sea sencilla, ni que se conozcan a priori y con detalle todas las operaciones que deben realizar los sistemas, ni que el sincronismo brazo-herramienta responda a patrones de interacción tan sencillos.

1.2.2 Formulación de los requisitos de partida y nuevos requisitos.

La arquitectura de un sistema o de una familia de sistemas se basa en sus requisitos. Si los requisitos cambian o aparecen otros nuevos, la arquitectura debe adaptarse a dichos cambios. Dependiendo del alcance de las modificaciones y del grado en que estuvieran previstas en el diseño original, la arquitectura será más o menos capaz de admitir los cambios, pero ningún diseño es lo suficientemente flexible como para adaptarse a *cualquier* modificación. En relación con la Arquitectura objeto de este trabajo de tesis, habría que destacar varios aspectos:

- Al tratarse de una arquitectura de referencia, los requisitos de partida están descritos con un alto grado de generalidad. Sin embargo, la experiencia acumulada a lo largo de los proyectos arriba mencionados y los trabajos de Kazman [Kazman et al 1994, 2000] y Bass [Bass et al 1998, 2000] sobre la caracterización de los atributos de calidad de arquitecturas software, permiten en algunos casos una formulación más precisa de los mismos, que puede dar lugar a modificaciones en la arquitectura.
- Se definen casos de uso con relación a los requisitos funcionales [Alvarez et al 1998], pero no para los no funcionales. Se hace necesaria la definición de escenarios que describan casos de uso asociados a estos últimos.
- Han surgido nuevos requisitos, como:
    1. La extensión de la arquitectura para la teleoperación en entornos no estructurados.
    2. La incorporación al sistema de utilidades de visión artificial.
    3. La capacidad de teleoperar a varios sistemas conjuntamente, y
    4. La posibilidad de dotar a los mecanismos teleoperados de mayor autonomía.

Los dos primeros (muy relacionados) ya se pusieron de manifiesto en el desarrollo de los sistemas TRON e IRV. Los restantes se contemplan a medio plazo para el sistema GOYA. Ninguno de ellos está, sin embargo, convenientemente caracterizado. Es necesario traducir estos requisitos a atributos de calidad específicos y analizar si pueden alcanzarse con la arquitectura propuesta. Si no es así, la arquitectura deberá ser modificada.

1.3 Objetivos y Planteamiento de la Tesis.

Los objetivos que se proponen para este trabajo de tesis son los siguientes:

1. Caracterización de los atributos de calidad aplicables a la arquitectura de referencia [Álvarez 1997], incluyendo una definición más precisa de los requisitos originales e incorporando los nuevos requisitos.
2. Evaluación de la arquitectura software de referencia para sistemas de teleoperación de robots descrita en [Alvarez 1997] para determinar su conformidad con dichos atributos de calidad.
3. Aplicar los resultados de las actividades anteriores para la definición de la arquitectura del sistema GOYA [GOYA 1998b].

Para alcanzar dichos objetivos es necesario:

- Utilizar un método de evaluación que permita determinar los lugares de la arquitectura que son críticos para la obtención de los atributos, considerándolos tanto de forma aislada como en su conjunto. Dicho método es ATAM (*Architecture Trade-off Analysis Method* [Kazman et al 2000]).
- Utilizar una metodología de diseño aplicable a arquitecturas de referencia en la cual puedan incluirse de forma natural los resultados de ATAM. La metodología de diseño adoptada es el "Diseño Basado en Arquitectura", en adelante ABD, (*Architecture-Based Design Method)* descrito en [Bass et al 1999].

Los procesos de evaluación y diseño deben realimentarse entre sí sin excesivo esfuerzo. Los resultados de cada evaluación deben constituir las entradas de un nuevo ciclo de diseño, que resultará en una nueva versión de la arquitectura, susceptible de ser nuevamente evaluada. Tanto el ABD como ATAM han sido desarrollados por el SEI (*Software Engineering Institute, Carnegie Mellon University* [SEI 2002]) y ambos se inscriben dentro de un mismo proyecto que tiene como fin el desarrollo y evaluación de arquitecturas software [SEI-ata 2002]. El ABD es un método iterativo que incluye la realización de un *mini*-proceso ATAM en cada iteración. Si una arquitectura ha sido desarrollada desde un principio utilizando el ABD no hay ninguna razón metodológica para aplicarle ATAM[2]. Tanto ATAM como el ABD necesitan para su ejecución de una caracterización muy precisa de los atributos de calidad. La arquitectura descrita en [Alvarez 1997] no ha sido desarrollada con el ABD y, como veremos, algunos de sus atributos no están suficientemente caracterizados. Por ello, los primeros pasos deben ser la caracterización de atributos y la realización de ATAM. A partir de ahí podrá utilizarse el ABD.

Sin embargo, el ABD tiene sus limitaciones. El ABD es un método de diseño de arquitecturas, no de sistemas. El ABD define las pautas generales para dividir el sistema en subsistemas, para asignar responsabilidades a los mismos y para identificar plantillas de comportamiento y de relación con la infraestructura aplicables a todos los subsistemas. Sin embargo, no proporciona criterios para la definición de componentes concretos, para la estructuración de los subsistemas en clases y tareas, ni para el despliegue de los mismos en nodos físicos. Para llenar las lagunas

[2] No obstante, los creadores de ambos métodos aconsejan la realización de ATAM por un equipo externo incluso en una arquitectura desarrollada con el ABD.

del ABD, se utilizarán criterios y patrones de ingeniería software basada en componentes, en adelante CBD (de sus siglas en inglés *Component Based Development*).

La aplicación sobre las que se pretende implantar la nueva arquitectura es el sistema GOYA. El sistema GOYA, es un mecanismo para la limpieza de los finos de embarcaciones de diverso tipo, cuyos requisitos, como se verá en los capítulos 7 y 8, difieren mucho de los requisitos para los que se planteó la arquitectura definida en [Alvarez 1997]

1.4 Métodologías, notación y herramientas.

Las tres piezas clave para el diseño y evaluación de una arquitectura son: metodologías, notaciones formales y herramientas. El objetivo de este apartado es presentar las que se han utilizado en este trabajo de tesis y justificar brevemente su elección.

- Como herramienta de diseño se utilizará Rational ROSE'2000 [Rational 2002].
- La notación empleada ha sido UML (*Unified Modeling Language* [Rumbaugh et al 1998]), en concreto la versión 1.4 [UML v1.4 2000]).
- Las metodologías empleadas han sido ATAM para la evaluación y el ABD para el diseño. Como ya se ha dicho, para llenar las lagunas del ABD, se han utilizado criterios y patrones de la ingeniería software basada en componentes.

Razones para utilizar UML.

- UML incorpora y unifica lo mejor de las notaciones de Booch [Booch 1994], Rumbaugh [Rumbaugh et al 1991], Jacobsson [Jacobson et al 1992] y Harel [Harel et al 1990] y tiene riqueza semántica suficiente para describir todas las vistas de la arquitectura.
- El uso de una notación única ayuda a mantener la coherencia entre las diferentes vistas del sistema y facilita su trazabilidad.
- UML es un estándar de facto, que incorporan la mayoría de las herramientas de diseño, entre ellas la que se usa en el trabajo de tesis, Rational ROSE'2000 [RationalRose 2000]. Las que aún no lo soportan es previsible que no tarden mucho en incorporarlo.

No obstante, UML tiene algunas carencia importantes para su uso como lenguaje de descripción de arquitecturas. Dichas carencias, algunas de las cuales pueden soslayarse mediante los mecanismos de extensión del lenguaje, se comentan en el capítulo 2.

Razones para utilizar ATAM.

- Aunque es de reciente factura (las primeras referencias se remontan al año 97 [Barbacci et al 1997]) existen ejemplos de aplicación, algunos incluso sobre arquitecturas de referencia [Gallagher 2000], y está patrocinado por una institución, el SEI (*Software Engineering Institute*) de la CMU (*Carnigie Mellon University*) de reconocido prestigio.
- Puede utilizarse de forma natural en el marco de un ciclo de vida de desarrollo en espiral. Está dirigido por riesgos y admite una evolución iterativa e incremental.
- Permite incorporar los resultados de análisis formales sobre los atributos de calidad al proceso de evaluación. Cuando no existe el modelo formal correspondiente al atributo a evaluar se recurre a escenarios y casos de uso.

- Considera explícitamente la relación entre los atributos de calidad y los estilos o patrones usados en la arquitectura a través de ABAS (*Attribute Based Architectural Styles*) definidos por Klein [Klein 1999a, b] y de las *primitivas de diseño basadas en atributo* [Bass et al 2000].
- Tiene muy en cuenta las interacciones entre los distintos atributos de calidad y permite identificar aquellas partes del sistema que son claves para la obtención de los mismos.
- ATAM está incluido en el método de diseño que se utilizará para modificar la arquitectura (el ABD).
- ATAM proporciona criterios para caracterizar los atributos de calidad.

Razones para utilizar el ABD.

- Define un proceso de diseño en el que la evaluación de la arquitectura es un paso explícitamente considerado. Incluye la realización de un *mini*-proceso ATAM en cada iteración.
- Es un método enfocado al diseño de arquitecturas para líneas de producto y sistemas de larga vida operativa, por lo que se adapta mejor a los objetivos de esta tesis que otras metodologías orientadas al diseño de sistemas específicos.
- Establece criterios claros para la estructuración del sistema en subsistemas y para la asignación de responsabilidades a los mismos.

Razones para utilizar Rational Rose:

- Rational ROSE'2000 [RationalRose 2000] es una herramienta de diseño y modelado visual que se inscribe dentro de un grupo de herramientas software desarrollado por Rational que proporcionan soporte para un desarrollo iterativo y basado en componentes.
- Permite la inclusión de nuevas utilidades como la herramienta UML-Mast [Drake et al 2000] desarrollada por la Universidad de Cantabria para el modelado, análisis y diseño de sistemas distribuidos de tiempo real.
- Permite enlazar fácilmente los diferentes componentes y conectores de la arquitectura con otros documentos adicionales.
- Incorpora la notación UML.

1.5 Estructura y Contenidos de la Tesis

La presente tesis está dividida en 10 capítulos y 3 anexos. El primer capítulo presenta los objetivos y el planteamiento de la tesis, así como las metodologías que se emplean en la misma. Los capítulos 2, 3 y 4 describen el estado de la técnica en lo referente a este trabajo de tesis. Por razones de espacio es imposible tratar con detalle los temas que se abordan en estos capítulos (arquitectura software, patrones, vistas y estilos arquitecturales, desarrollo software basado en componentes, atributos de calidad, metodologías de evaluación y diseño, etc.). Se ha tratado de dar una panorámica de los mismos, explicando con mayor detalle aquellos aspectos que son fundamentales para la comprensión del trabajo de tesis y aportando las referencias necesarias para el lector interesado en profundizar en temas concretos. Los capítulos 5, 6 y 7 se corresponden con la aplicación del método de evaluación (ATAM) y en ellos se describen los requisitos de partida, la arquitectura a evaluar, el proceso de análisis y los resultados del mismo.

En el capítulo 8 se propone una nueva descomposición funcional de la arquitectura y los requisitos de un modelo de componentes para el desarrollo del sistema GOYA. En el capítulo 9 se resumen las conclusiones, aportaciones y trabajos futuros y, finalmente, en el capítulo 10 se listan las referencias bibliográficas.

Adicionalmente, este trabajo de tesis incluye 3 anexos, cuyo objetivo es separar aquella información que por su grado de detalle puede dificultar la lectura de la tesis, pero que debe incluirse en la misma. El anexo I resume los resultados del análisis del dominio de sistemas de teleoperación realizado en [Álvarez 1997] y de las técnicas empleadas para el mismo, incluyéndose algunas nuevas referencias que actualizan dicho estudio. En el anexo II se describen una serie de plantillas que caracterizan de forma precisa los requisitos de los sistemas del dominio considerado. En el anexo III se describe la realización de los escenarios de evaluación.

El contenido de cada uno de los capítulos es el siguiente:

- ***Capítulo 1: Planteamiento y Objetivos de la Tesis.***

  Se describen de forma general los objetivos de la tesis y su motivación y se realiza una presentación preliminar de las metodologías, herramientas y notaciones que van a utilizarse. Por último, se presenta el contenido y estructura de este trabajo de tesis.

- ***Capítulo 2: Arquitectura Software.***

  Se revisa el significado del término *Arquitectura* aplicado a sistemas software y se justifica su importancia en el proceso de desarrollo. Se describen brevemente los términos estilo arquitectónico y patrón de diseño y se presentan los lenguajes de descripción de arquitecturas. Se describen brevemente UML, Rational Rose y MAST. Asimismo se presentan los *frameworks* orientados a objeto, las líneas de producto y algunos conceptos fundamentales de la ingeniería software basada en componentes. Para terminar se resume el ciclo de negocio de la arquitectura.

- ***Capítulo 3: Arquitectura y Atributos de Calidad del Software.***

  Se describen y clasifican los atributos de calidad y se revisa su relación con la arquitectura y con los patrones y estilos presentados en el capítulo anterior. Se presentan los ABAS (Estilos Arquitecturales Específicos de Atributo) y las primitivas de diseño basadas en atributo.

- ***Capítulo 4: Metodologías de evaluación y diseño de arquitecturas software.***

  Se presentan conceptos básicos relacionados con los parámetros de calidad del software y se presentan las metodologías de evaluación y diseño que se utilizan en el trabajo de tesis (ATAM y ABD). Finalmente se describe la forma en que dichas metodologías se aplican en el trabajo de tesis.

- ***Capítulo 5: Caracterización de los atributos de calidad.***

  Se caracterizan los atributos de calidad, tanto los originales como los nuevos, según las pautas establecidas por ATAM y ABD. Se describen las directrices de la arquitectura y los los requisitos funcionales y no funcionales que servirán de guía para los procesos de evaluación y diseño. Los contenidos de este capítulo se completan de forma más exhaustiva en el anexo II.

- ***Capítulo 6: La Arquitectura de Referencia.***

  Se presenta la arquitectura de referencia descrita en [Alvarez 1997]. Se refinan sus vistas y se expresan en UML. Se describen los estilos arquitecturales empleados en la misma y su relación con los atributos de calidad caracterizados en el capítulo anterior.

- ***Capítulo 7: Evaluación de la Arquitectura.***

  A partir de los atributos caracterizados en el capítulo 5 y de la descripción de la arquitectura ofrecida por el capítulo 6 se realiza la evaluación de la misma y se describen las conclusiones del análisis. La realización de los escenarios presentados en este capítulo se describe en el anexo III.

- ***Capítulo 8: Requisitos de un modelo de componentes para el sistema GOYA.***

  Se describe el enfoque que debe darse al desarrollo de nuevos sistemas de teleoperación, basado en líneas de producto y en un proceso de desarrollo basado en componentes software. Se definen los requisitos que debe cumplir el modelo de componentes que se adopte o defina.

- ***Capítulo 9: Conclusiones y Trabajos Futuros.***

  Se exponen los resultados obtenidos en este trabajo de tesis y se presentan las líneas de investigación a seguir como extensión de la misma.

- ***Anexo I: Análisis del dominio de teleoperación.***

  Se resumen los trabajos de [Alvarez 1997] en relación con el análisis y descripción del dominio de teleoperación de robots.

- ***Anexo II: Plantillas de Atributo.***

  Se describen con detalle las plantillas de atributo presentadas en al capítulo 5.

- ***Anexo III: Escenarios de Evaluación.***

  Se describen los escenarios y casos de uso utilizados para la evaluación de la arquitectura.

2 Arquitectura Software.

2.1 Introducción

La asociación de los términos Arquitectura y Software es relativamente nueva. El término Arquitectura siempre ha estado ligado a la construcción de edificios. Dentro de tal ámbito todos entendemos y apreciamos la función de la Arquitectura y, aunque sea informalmente, estamos familiarizados con ella. Reconocemos su necesidad, valoramos su importancia, distinguimos gran parte de su jerga y sabemos con que materiales trabaja. Sin embargo, y a pesar de que el software es ya para la mayoría de nosotros algo tan omnipresente como los edificios en que habitamos, las cosas cambian radicalmente cuando el término Arquitectura se aplica a estos sistemas. Aquí, la Arquitectura es un concepto relativamente nuevo, dentro de una disciplina nueva, la Informática, con muy pocas décadas de existencia[3].

Hasta los años 80, los diseñadores software se centraban en el código, donde el diseño involucra algoritmos y estructuras de datos, los componentes con los que trabajar son las primitivas de los lenguajes de programación y los mecanismos de composición incluyen variables, registros, cierta lógica de control, funciones y procedimientos. Pero, intentar construir sistemas software grandes y complejos a partir de estos elementos, aun sabiamente administrados, viene a ser como intentar construir un rascacielos con poco más que unos buenos conocimientos de albañilería. A partir de los años 80 se popularizan paradigmas como la programación modular, la organización del software en paquetes, la programación orientada a objetos y la programación concurrente cuya formulación se remonta en algunos casos 20 años atrás. La aplicación de estos paradigmas hace posible la puesta en práctica de principios y reglas de diseño cuya realización se hacía anteriormente muy difícil, pero siguen moviéndose en el nivel del código. Hacía falta algo más. El diseño del software, como el de cualquier sistema complejo, requiere conocer y seguir disciplinadamente una serie de reglas bien definidas que indiquen como ensamblar y conectar los diferentes elementos del sistema y cómo gestionar el desarrollo y evolución del mismo. Sin dichas reglas, el diseño se convierte en un proceso caótico, que en el caso del software se agrava al estar sujeto a modificaciones a lo largo de su ciclo de vida. *Pero ¿Qué reglas seguir?* Y dada la diversidad de sistemas posibles *¿Dónde y cómo aplicarlas?* La historia del software de los últimos 30 años es en buena parte la historia de cómo se ha intentado responder a estas preguntas.

En la década de los 70 y como consecuencia de los problemas que surgieron a la hora de desarrollar software a gran escala, y que dieron lugar a la crisis del software, muchos ingenieros

[3] El lector interesado en la génesis de la asociación de los términos software y arquitectura puede empezar por [Clements 1996a] y [Coplien 1999].

e investigadores centraron su atención en los problemas de diseño del software. Surge la idea de separar el diseño de la implementación y se aprecia la necesidad de utilizar notaciones y técnicas especiales, entre las que cabe destacar el modelo entidad-relación de Chen [Chen 1976] y el diseño estructurado [Yourdon 1977].

La década de los 80 supuso grandes avances en la capacidad de desarrollar, describir y analizar grandes sistemas software. En esta década se desarrollan metodologías y lenguajes de notación formal para expresar las relaciones estructurales entre los diversos componentes de un sistema software y racionalizar su proceso de diseño. También durante esta década Boehm propone su ciclo de vida en espiral [Boehm 1988], de plena vigencia en la actualidad, y del que han surgido múltiples variantes. Continuando está línea de investigaciones, la década de los 90 supuso el verdadero nacimiento de la Arquitectura Software. Las investigaciones sobre metodologías y notaciones, que arrancan en la década anterior, cristalizan en un aluvión de publicaciones sobre el tema ([Shlaer et al 1988], [Booch et al 1994], [Coad 1991], [Rumbaugh 1991], [Coleman et al 1993] y [Jacobson et al 1992] por citar las más representativas). Se definen y catalogan estilos arquitecturales [Klein et al 1999a, 1999b] y patrones de diseño [Gamma et al 1995], [Buschmann et al 1996], indicando además su ámbito de aplicación. Las metodologías de diseño se enfocan hacia la modularización, el crecimiento incremental y el prototipado evolutivo y adoptan decididamente el modelo de ciclo de vida en espiral. Aparecen las primeras herramientas que incorporan los resultados de estas investigaciones y que automatizan su aplicación en el proceso de desarrollo. Al principio en el marco de laboratorios y Universidades, pero ya a finales de los 90 como productos comerciales. Empieza a hablarse en serio de reutilización del software. Los entornos de programación, ya claramente orientados a objetos, incorporan *toolkits* y *frameworks* muy elaborados. Las preguntas que nos hacíamos en los párrafos anteriores respecto del proceso de diseño de sistemas software complejos (*¿qué reglas seguir?, ¿dónde y cómo aplicarlas?*) empiezan a ser contestadas. Las metodologías, los lenguajes de notación formal y las herramientas que engloban a ambos, y que surgen durante esta década, constituyen las tres piedras angulares con las que cuenta el arquitecto software para desarrollar su trabajo. La Arquitectura Software se convierte así en una disciplina en auge que se ofrece, por un lado, como un marco de trabajo para satisfacer los requisitos técnicos de un sistema y, por otro como una base técnica eficaz para analizar, diseñar y reutilizar un sistema.

Dada su relación con este trabajo de tesis, es necesario destacar la aparición de los primeros estudios acerca de arquitecturas específicas de dominio [Lubars 1988] [Arango 1989] [Kang et al 1990] hacia finales de los años 80. Diversos autores proponen el estudio formal de las características y estructuras de una familia de sistemas. Dicho estudio permite obtener una **arquitectura de referencia** para todas las aplicaciones de esa familia mediante la identificación de partes comunes. Dentro de esta línea es de mencionar el método FODA (*Featured Oriented Domain Analysis* [Kang et al 1990]) y en general los trabajos del SEI de la CMU (*Software Engineering Institute, Carnigie Mellon University*). La obtención de una arquitectura específica de dominio conlleva la definición de nuevos estilos arquitectónicos a partir de patrones conocidos y documentados, promoviendo de esta manera la **reutilización del software**. No obstante, es necesario apuntar que la idea de distinguir los puntos comunes de *una familia de productos software* con vistas a la reutilización de componentes es bastante anterior. De hecho fue propuesta por Parnas en [Parnas 1976].

En resumen, es importante destacar que, con estos antecedentes, las ideas clave en relación con este trabajo de tesis son las siguientes.

1. La arquitectura de un sistema define las estructuras de sus componentes, sus relaciones o comportamiento externamente visible y los principios que gobiernan su evolución a lo largo del tiempo.
2. No es posible abordar la construcción adecuada de sistemas de cierto tamaño y complejidad sin un diseño cuidadoso de su arquitectura.
3. El diseño de una arquitectura requiere de notaciones formales, de metodologías y de herramientas que permitan automatizar el proceso de diseño y simular comportamientos.
4. La arquitectura debe ser evaluada, ya que las decisiones que hace patentes son precisamente las que más van a condicionar el desarrollo del sistema.
5. Los atributos de calidad sólo tienen sentido dentro del contexto definido por las especificaciones del sistema y no como *entes* abstractos.
6. Los atributos de calidad no son independientes, sino que interaccionan entre sí a través de las relaciones estructurales impuestas por la arquitectura. La mejora de unos frecuentemente solo puede lograrse a costa de empeorar otros. Por ello es necesario llegar a compromisos de diseño en los que también hay que tener en cuenta factores sociales y de negocio.
7. El estudio y la clasificación de las arquitecturas software existentes para una familia de sistemas, permite identificar partes comunes, con el fin de obtener una *arquitectura de referencia* que sirva para todas las aplicaciones de esa familia.

2.2 Definiciones de Arquitectura Software

Existen muchas definiciones de Arquitectura Software, la mayoría muy parecidas, diferenciándose entre sí por el énfasis que ponen sobre los aspectos que quieren resaltar. De todas ellas nos quedaremos con la propuesta por Bass [Bass et al 1998][4]:

"La arquitectura de un sistema software es la estructura o estructuras del sistema, incluyendo sus componentes software, las propiedades externamente visibles de dichos componentes y las relaciones entre ellos".

Esta definición es la adoptada por los creadores de los métodos de evaluación de arquitecturas SAAM (*Software Architecture Analysis Method* [Kazman et al 1994] [Bass et al 1998]) y ATAM (*Architecture Tradeoff Analysis Method* [Kazman et al 2000]) e implica dos condiciones que debe cumplir una descripción del sistema para poder ser considerada una descripción de su arquitectura:

1. Asume que un sistema puede incluir más de una estructura *("...es la estructura o estructuras del sistema...")*. En consecuencia, la descripción de la arquitectura debe incluir la descripción de todas estas estructuras, sin que pueda primarse en principio alguna de ellas sobre las demás. En este sentido la definición es coherente con el modelo 4+1 descrito por Krutchen en [Krutchen 1995], que ha asumido el consorcio OMG [OMG 2000] para el desarrollo de UML y Rational en su *Proceso Unificado de Desarrollo de Software* [Jacobson et al 1999].
2. La arquitectura incluye las propiedades *externamente visibles* de los componentes del sistema. Esto significa dos cosas:

[4] Existen otras buenas definiciones de Arquitectura Software. En especial la propuesta por Garlan [Garlan et al 1995b], que es la que toma como guía [Alvarez 1997]: *"La arquitectura de un sistema software es la estructura de sus componentes, sus relaciones y los principios que gobiernan su evolución a lo largo del tiempo".*

- El comportamiento de un componente es parte de la arquitectura sólo si dicho comportamiento puede ser observado por otro componente del sistema o por un actor externo que interacciona con él.
- Debe proporcionarse una descripción clara de las interfaces que cada componente ofrece al resto y al exterior.

Estas condiciones son fundamentales para poder evaluar la arquitectura y por tanto ésta es la definición que mejor se adapta a los propósitos de esta tesis.

La definición también asume que todo sistema tiene una arquitectura, ya que todo sistema *puede* ser descrito en función de sus componentes y de las relaciones entre ellos. Incluso en el caso de un sistema monolítico hay una arquitectura compuesta por un solo componente. La definición revela la diferencia entre la arquitectura de un sistema, que existe porque existe el sistema, y la descripción de dicha arquitectura que puede o no existir[5]. Por supuesto, una arquitectura sólo puede ser evaluada si existe una descripción adecuada de la misma.

2.3 Aportaciones de la arquitectura al proceso de desarrollo software

La arquitectura software constituye un modelo inteligible de la estructura del sistema y de las relaciones entre sus componentes. El nivel de abstracción necesario para describirla se sitúa aún muy lejos de los detalles de implementación, pero exhibe ya toda una serie de características que van a condicionar tanto el producto final como su proceso de desarrollo. Por ello, si la arquitectura está correctamente descrita, permite o favorece [Garlan et al 1994]:

1. La documentación de la estructura y propiedades del sistema en un lenguaje inteligible por todas las partes implicadas en el desarrollo del mismo.
2. La evaluación de las decisiones de diseño desde las etapas más tempranas posibles del proceso de desarrollo y durante la evolución del mismo.
3. La reutilización de modelos y componentes software y el uso de componentes software comerciales.
4. El prototipado evolutivo y el crecimiento incremental.

1. Documentación de la estructura y de las propiedades del sistema. Un camino hacia el consenso.

La arquitectura es un modelo de muy alto nivel, que puede ser entendido tanto por los técnicos como por los clientes y usuarios finales. Como dice Garlan [Garlan et al 1994]:

"La arquitectura software simplifica la comprensión de grandes sistemas mediante su presentación en un nivel de abstracción en el cual es posible entender el sistema como un todo."

Al constituir la arquitectura un modelo *fácil* de entender, proporciona un lenguaje común en el que las necesidades y prioridades de cada parte pueden ser expresadas, negociadas y resueltas. Si la arquitectura está convenientemente descrita, constituye un excelente medio de comunicación para llegar a un consenso acerca de las propiedades y estructuras del sistema.

Este consenso es fundamental para realizar una evaluación realista de la arquitectura de un sistema. Las arquitecturas se evalúan respecto de una serie de atributos de calidad cuyo alcance debe estar claramente

[5] Esta es la posición de [Bass et al 1998]. Bachmann en [Bachmann et al 2000a] afirma que la arquitectura sólo existe si existe una descripción de la misma. Ambas posiciones son razonables, simplemente ven el problema desde diferentes puntos de vista.

definido, pues cada una de las partes implicada en el sistema tiene sus propias necesidades y prioridades, a menudo en conflicto con las del resto.

2. Evaluación precoz del sistema.

Al mostrar todas las características importantes que tendrá el sistema una vez construido, la arquitectura software es la primera representación completa de las decisiones de diseño y, por tanto, la primera oportunidad seria de evaluar dichas decisiones. Oportunidad que, además, se ofrece antes de que el sistema haya empezado a construirse, y en un formato que, como se acaba de explicar, puede ser entendido por clientes y usuarios finales que de esta manera pueden sumarse a la discusión y al consenso.

3. Reutilización de modelos. Hacia un mercado de componentes software.

Al alejarse de los detalles de implementación, la arquitectura se convierte en un artefacto transferible que puede ser aplicado a otros sistemas que compartan requisitos similares. Reutilizar una arquitectura es una decisión crítica y de muy largo alcance, pues supone adoptar unas decisiones de diseño que van a afectar a todo el proceso de desarrollo. Los beneficios de la reutilización son tanto mayores cuanto antes se apliquen en el ciclo de desarrollo del sistema, pero también los riesgos. Puesto que la arquitectura define las propiedades externamente visibles de los componentes, todos los sistemas que compartan una misma arquitectura pueden, en principio, utilizar componentes que exhiban un mismo comportamiento, abriéndose una puerta para el uso de componentes comerciales.

Ejemplos de reutilización de arquitecturas son las líneas de producto [Brownsword et al 1996], [Gannod et al 2000] [Bosch 2000] y las arquitecturas de referencia específicas de un dominio. En las líneas de producto, aunque cada uno de los productos posea ciertas características propias, todos comparten una misma arquitectura y por tanto utilizan en gran medida los mismos componentes. En los dominios en los que se ha alcanzado una gran madurez y existe un consenso acerca de la división funcional del sistema es posible la creación de un mercado de componentes software y la construcción de aplicaciones como mecanos, a base de ensamblar componentes. Tal vez la aplicación obtenida no sea tan buena como una construida a medida[6], pero cierta pérdida de excelencia es un pequeño precio a pagar por los beneficios que proporciona la reutilización de componentes.

No obstante, aunque existen *algunos* buenos ejemplos de líneas de producto y arquitecturas específicas de un dominio donde las premisas arriba expuestas se cumplen, los dos párrafos anteriores muestran el mejor de los mundos posibles. La reutilización no ya de modelos, sino de componentes, es la excepción y no la regla. El mercado de componentes comerciales debería basarse en el principio de intercambiabilidad. Sin embargo, los *fabricantes* de componentes tienden a hacer sus propias *suposiciones*[7] sobre cómo dichos componentes deben ensamblarse y sobre la infraestructura que debe soportarlos [Garlan et al 1995a]. Estas suposiciones son decisiones arquitecturales que pueden entrar en conflicto con la arquitectura en la que dichos componentes deben integrarse, haciendo imposible o extremadamente costosa su inclusión en ella [Boehm 1999a] [Garlan et al 1995a]. Boehm describe en varios artículos los riesgos que conlleva el uso de componentes software comerciales y las precauciones que deben tomarse si se opta por ellos. El más representativo de ellos, [Boehm 1999b], lleva el expresivo titulo de *"COTs*[8] *Components: Plug and Pray?"* ("Componentes Software Comerciales: ¿Enchufar y Rezar?")[9]. Tal y como afirma Shaw en [Shaw et al 1996], *a la hora de utilizar un componente software no sólo es necesario conocer los servicios que proporciona, sino también los servicios que necesita* [Meyer 1992].

A pesar de la situación descrita en el párrafo anterior, existe un interés muy fuerte en la industria del software por definir arquitecturas y estándares que sirvan de marco para la definición e integración de

[6] Probablemente sea mejor que una construida a medida, ya que utiliza componentes cuyo comportamiento ha sido validado en otras aplicaciones.
[7] Del inglés *assumptions*.
[8] *COTs: Commercial of the shell*
[9] El lector interesado en los COTS y en su relación con el proceso de diseño arquitectural puede consultar [SEI-cbs 2002] y [USC 2000]

componentes software, tanto de propósito general como específicos de dominio. En este sentido cabe destacar iniciativas como CORBA[10] [OMG 2000] que entre otros objetivos pretende conseguir una portabilidad real de las implementaciones de objetos servidores y clientes. CORBA ofrece una arquitectura muy general que puede ser adoptada a la hora de definir arquitecturas más específicas. Sin embargo, y a pesar de la creciente aceptación de CORBA, existen todavía relativamente pocos componentes a disposición del diseñador.

Tal vez la clave de la reutilización esté en la granularidad con la que se definen los componentes. Un componente tiene tanta mayor probabilidad de ser reutilizado cuanto más general es el trabajo que realiza y más concretas sus responsabilidades. Así, es más probable que sea reutilizado un componente que *realice*[11] una lista enlazada, que uno que implemente un *encoder* y más probable que sea reutilizado el *encoder* que el controlador de una articulación. Cuanto más específico sea el dominio de aplicación del componente menos usuarios potenciales habrá. Cuanto más responsabilidades tenga un componente mayor será su necesidad de servicios procedentes de otros componentes y de la infraestructura, mayores sus suposiciones arquitectónicas y, por tanto, mayores sus probabilidades de entrar en conflicto con la arquitectura en la que se pretende integrar. Al mismo tiempo, si el componente tiene que parametrizarse para responder a diferentes situaciones, dicha parametrización será tanto más difícil cuanto mayor sea el número de sus parámetros y el rango de sus posibles valores. Y volvemos al principio, sólo en los dominios en los que se ha alcanzado una gran madurez y existe un consenso acerca de la división funcional del sistema es posible la creación de un mercado de componentes software.

4. El prototipado evolutivo y el crecimiento incremental.

Una vez que la arquitectura ha sido definida, puede ser analizada y prototipada como un modelo ejecutable. Esto tiene dos benéficas consecuencias. En primer lugar permite la simulación del sistema y la detección de errores de diseño antes de iniciarse la fase de implementación. En segundo lugar, los componentes simulados pueden ir sustituyéndose por versiones cada vez más refinadas de los componentes finales, favoreciendo de esta manera el crecimiento incremental y la adopción de ciclos de vida en espiral dirigidos por riesgos.

2.4 Necesidad de evaluar la Arquitectura Software de un sistema

Las decisiones de diseño que hace patentes la arquitectura son precisamente las que más van a condicionar el desarrollo del sistema, las de más alcance y por tanto las más difíciles de corregir en caso de error. Así:

- **La arquitectura no fuerza una implementación, pero la condiciona**, ya que deberá adaptarse al tipo de componentes y a las relaciones que entre ellos se hayan definido en la arquitectura.
- **La arquitectura condiciona la organización y gestión del proyecto**, y, si el proyecto es lo suficientemente importante, por los recursos que utiliza o las expectativas creadas, puede incluso condicionar la estructura y gestión de la empresa o institución que lo realiza (considérese una línea de producto). La arquitectura define una descomposición estructural del sistema en subsistemas con interfaces bien definidas. Esto hace posible la división del proyecto en paquetes de trabajo, que se asignarán a diferentes equipos de desarrollo, facilitando, pero también condicionando, la planificación y la asignación de recursos. Así, la estructura organizativa de un proyecto tiende a ser

[10] *CORBA: Common Object Request Broker Architecture*. Los propósitos de CORBA van mucho más allá de los mencionados en el párrafo. El lector interesado puede consultar directamente la Web del OMG [OMG 2000].
[11] *Realizar* en el sentido de *realizar* una interfaz, es decir ofrecer unos servicios a través de unas operaciones perfectamente definidas.

un reflejo de su arquitectura[12]. El reverso de la moneda es que una modificación de la arquitectura puede producir un enorme impacto sobre la gestión del proyecto, pues supone la reestructuración de los grupos de trabajo y la reasignación de recursos y responsabilidades.

- **La arquitectura influye en los atributos de calidad del sistema, mejorando unos a costa de empeorar otros**. Los atributos de calidad no existen de forma independiente, sino que interaccionan entre sí [Kazman et al 2000] [Boehm 1996]. La arquitectura adoptada puede verse como el consenso final sobre los atributos de calidad del sistema al que llegan todas las partes implicadas. Las arquitecturas no pueden evaluarse sobre atributos de calidad abstractos, sino sobre los requisitos concretos que debe cumplir la aplicación. Requisitos que deben haber sido previamente definidos, analizados y consensuados.

- **Las decisiones arquitecturales ocurren en las primeras etapas de la vida de un proyecto y afectan a la estructura misma del sistema, condicionando todo el posterior proceso de desarrollo**. La detección precoz de errores es uno de los caballos de batalla de la informática. El coste de corregir un error se hace tanto mayor cuanto más tarde se detecta en el ciclo de vida de un proyecto. Por ello es necesario disponer de métodos fiables para evaluar la arquitectura y aplicarlos tan pronto como sea posible. En este sentido han ido encaminados los trabajos de, entre otros, Boehm (WinWin y QARCC [Boehm 1996]) y del SEI (Métodos de análisis de arquitecturas SAAM [Bass et al 1998] y ATAM [Kazman et al 2000]).

Obviamente, la consecución de los atributos de calidad de un sistema no depende sólo de la arquitectura. Es también función de otras muchas variables [Voa 1999]. La gestión del proyecto, la elección de lenguajes de programación y de algoritmos apropiados y la calidad de la implementación influyen también de forma decisiva en el resultado final. Pero aunque una buena arquitectura no sea condición suficiente para asegurar la calidad del sistema, es sin ninguna duda una condición necesaria.

2.5 Descripción de la arquitectura software

Una arquitectura sólo puede ser evaluada si:

- Los atributos de calidad con los que deben confrontarse están bien definidos.
- Se dispone de una buena descripción de la misma, un conjunto de herramientas adecuado y una metodología que guíe el proceso.

Como se ha explicado, una buena descripción de la arquitectura debe representar a todas las estructuras del sistema. Pero, *¿Cuáles son esas estructuras?* y *¿cómo representarlas?*.

2.5.1 Estructuras del sistema.

Existen diversas clasificaciones de las estructuras de un sistema, que dependen de los *puntos de vista* desde los cuales se contempla. Según este enfoque, cada estructura de la arquitectura representa una *vista* de la misma. Más formalmente, Selic [Selic 2000] define una *vista* como una representación del sistema global desde la perspectiva de un conjunto de conceptos relacionados y *punto de vista* como una especificación de las convenciones utilizadas para construir las vistas. Aun cuando existe un consenso general acerca de la necesidad de representar la arquitectura utilizando diferentes vistas, tal consenso desaparece cuando hay que definir cuales son esas vistas. El estándar IEEE-1471-2000 [IEEE-1471-2000] define de alguna manera los puntos de vista que deben tenerse en consideración para describir la

[12] Algunos autores proponen lo contrario: La arquitectura es un reflejo de las estructuras de la organización que la desarrolla [Conway 1996]. En realidad la influencia es mutua, como explican Bass y otros en [Bass et al 1998].

arquitectura, pero a un nivel de abstracción tan alto que las vistas pueden seleccionarse o definirse siguiendo criterios muy diferentes. Ejemplos de modelos que definen puntos de vista explícitos son el modelo de 4+1 vistas de Krutchen [Krutchen 1995], los modelos de Sowa y Zachman [Sowa et al 1992] y el modelo propuesto por Soni y otros en [Soni et al 1995]. UML por su parte no define puntos de vista explícitos, pero éstos pueden deducirse de la semántica de sus diagramas. De todos estos modelos, el que ha conseguido mayor aceptación es el modelo de 4 + 1 vistas de Krutchen. El modelo de Krutchen merece un apartado en esta tesis por varias razones:

- Es un clásico de la literatura software, de gran influencia en publicaciones posteriores, como demuestran la gran cantidad de referencias al mismo.
- Es el que se ha usado para describir la arquitectura de referencia a evaluar [Alvarez 1997].
- Muchos otros modelos se han inspirado en él.
- Es el adoptado por Rational [Rational 2002], cuya herramienta, Rational Rose [RationalRose 2002] se ha utilizado en este trabajo tesis[13].

El modelo 4+1 (figura 2.1) define cuatro estructuras o *vistas* que deben ser consideradas en la documentación de todo sistema:

- **Vista Física**. Describe la relación entre el software y el hardware y sus aspectos de distribución. Básicamente es la descripción de cómo el software se ha *trasladado* al hardware del sistema. En esta vista se identifican los procesadores del sistema, su interconexión y los componentes software que debe ejecutar cada uno de ellos.
- **Vista Lógica**. Describe los requisitos funcionales del sistema. En un sistema orientado a objetos esta vista sería la descomposición del sistema en clases y objetos, la descripción de sus relaciones y la definición de las interfaces proporcionadas por los mismos. De forma más general, esta vista describe los servicios que deben proporcionar los componentes del sistema, bien a otros componentes, bien a un usuario externo.
- **Vista de Procesos**. Describe el comportamiento dinámico del sistema y sus procesos. Cubre aspectos tales como el paralelismo, la concurrencia y la sincronización de tareas. Krutchen enfatiza que estos aspectos capturan los requisitos no funcionales del sistema y menciona explícitamente el rendimiento, que considera no funcional, la disponibilidad, la integridad y la tolerancia a fallos. Este enfoque no es del todo compartido por otros autores [Bass et al 1998], que argumentan que dichos requisitos están presentes en todas las vistas y que ciertos requisitos no funcionales están más directamente asociados a otras vistas. De hecho, Krutchen en el mismo artículo [Krutchen 1995], asocia la reusabilidad y la portabilidad a la vista de desarrollo.
- **Vista de Desarrollo.** Describe la organización estática del software en su entorno de desarrollo. El software se agrupa en módulos o subsistemas, organizados en diferentes librerías software, y cada módulo ofrece a los demás una interfaz bien definida. A menudo la vista de desarrollo se estructura jerárquicamente en niveles, donde cada nivel ofrece una interfaz bien definida al inmediatamente superior. En esta vista se definen las relaciones de uso (importación y exportación de servicios) entre los diferentes módulos del sistema. La vista de desarrollo tiene una extraordinaria importancia en la planificación del proyecto, ya que es la más habitualmente usada para la asignación de trabajo a los diferentes equipos de desarrollo.
- **Vista de escenarios**. Los escenarios, representan casos de uso significativos del sistema y son la argamasa que une a las diferentes vistas del sistema, ya que muestran al sistema como un todo. La vista de escenarios es redundante o, si se prefiere complementaria, respecto del resto (de ahí el "+1") y tiene dos propósitos:

    1. Servir de herramienta para descubrir elementos arquitectónicos (componentes y relaciones).

[13] Aunque se ha cambiado la terminología respecto de la propuesta por Krutchen.

2. Validar la arquitectura. Los mismos casos de uso que se han usado para el diseño, aunque más elaborados, sirven de trazas para validar el sistema en sus diferentes fases de desarrollo hasta llegar al producto final.

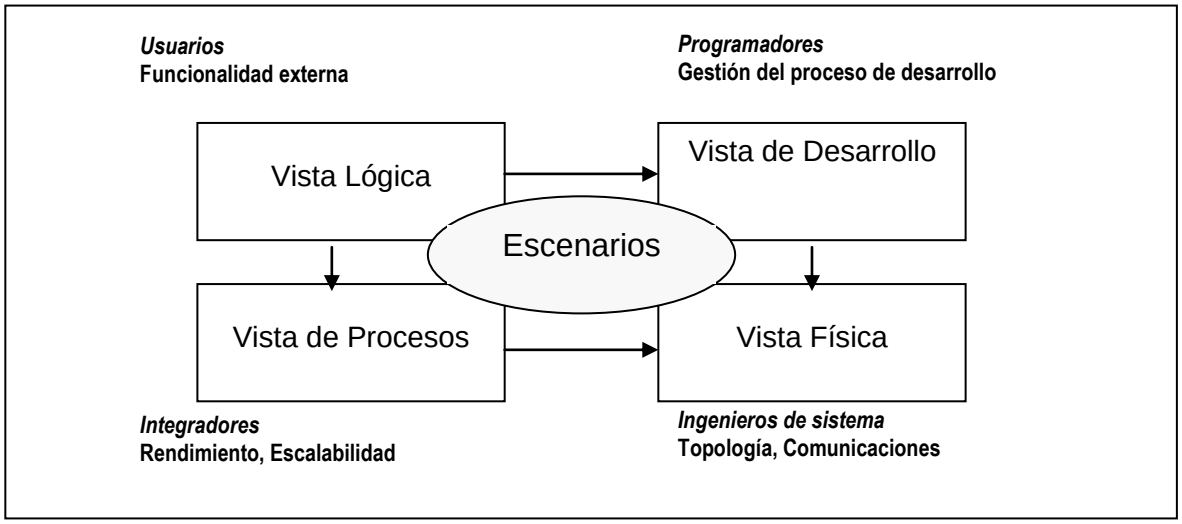


Figura 2.1: El modelo 4+1 [Krutchen 1995]

El modelo 4+1 es muy general. Normalmente cada una de las vistas debe ser descrita con varios diagramas de diferentes tipos. Una clasificación alternativa es la propuesta en [Bass et al 1998], cuyas estructuras se describen y ponen en relación con el modelo 4 + 1 en la figura 2.2.

Como se observa en la figura 2.2, la clasificación de [Bass et al 1998] sigue el modelo 4+1 de Krutchen, pero añadiendo algunas estructuras más. El enfoque es más abierto y pragmático que el de Krutchen (admiten la inclusión de otras estructuras si fueran útiles para describir el sistema). Sin embargo, es fácil observar que las 4 primeras estructuras son las de Krutchen y que el resto pueden encajarse en alguna de ellas sin grandes problemas.

Como ya se ha mencionado, a pesar de su influencia, el modelo de Krutchen no es ni mucho menos el único. El estándar ISO 1996 [ISO 1996] define el modelo de referencia para sistemas abiertos distribuidos RM-ODP (*Reference Model-Open Distributed Processing*). RM-ODP define un marco de referencia para el desarrollo de sistemas distribuidos e identifica cinco puntos de vista: de negocio, de información, de computación, de ingeniería y de tecnología. RM-ODP ofrece además una guía para organizar los modelos de acuerdo con dichos puntos de vista y resolver las relaciones entre los mismos. Por su parte, el Departamento de Defensa de los Estados Unidos define otra serie de modelos estandarizados incluidos en los marcos de trabajo TAFIM [TAFIM 1997] y C4ISR [C4ISR 1997], cuyo alcance se limita por el momento a sistemas de información relacionados con aplicaciones militares. También merece la pena mencionar las estructuras propuestas en [Soni et al 1995] y [Ellis et al 1996] para el desarrollo de aplicaciones industriales, la clasificación de [Ran 1998] que distingue explícitamente las estructuras en tiempo de compilación, carga y ejecución y los modelos de Sowa y Zachman [Sowa et al 1992].

La descripción del sistema mediante diferentes vistas facilita su diseño y análisis, ya que proporciona una forma muy útil de abordar su complejidad. Sin embargo introduce un nuevo problema. Puesto que cada vista es una representación del sistema global, las vistas no son completamente independientes. Las modificaciones realizadas sobre una vista pueden implicar modificaciones en el resto de las vistas. En los sistemas más pequeños la consistencia entre las vistas puede realizarse manualmente con la debida disciplina. Sin embargo, este enfoque se convierte en completamente inviable a medida que el tamaño o la complejidad del sistema aumentan. Por ello, aunque cada vista impone sus propias notaciones y convenciones, éstas deben alinearse formalmente con objeto de automatizar el mantenimiento del sistema. La integración de las diferentes vistas es un problema no resuelto por la comunidad software y es objeto de múltiples trabajos de investigación. El lector interesado puede consultar los trabajos de Egyed [Egyed et al 1999, 2000] [Egyed 2000], centrados en la integración de diferentes vistas utilizando UML, en los cuales encontrará además referencias muy recientes relacionadas con este tema.

ESTRUCTURAS ARQUITECTÓNICAS			
Estructura	**Descripción**	**Unidades/ *Relaciones entre unidades***	**Observaciones**
De módulos	Describe las relaciones entre módulos del sistema. El término módulo tiene un significado muy amplio, pudiendo referirse a interfaces, paquetes software, librerías, niveles, etc.	Módulos del sistema/ ***Ser submódulo de***	La estructura de módulos se utiliza asignar trabajo y recursos para el desarrollo del proyecto Se corresponde parcialmente con la vista de desarrollo del modelo 4+1
Conceptual o lógica	Describe los requisitos funcionales del sistema.	Abstracciones de los requisitos funcionales del sistema/ ***Comparte datos con***	Un ejemplo puede ser un módelo de referencia. Esta vista es útil para entender las interacciones entre entidades en el espacio del problema. Se corresponde parcialmente con la vista lógica del módelo 4+1
De coordinación o procesos	Describe los aspectos dinámicos del sistema, sincronización y concurrencia	Procesos o threads/ ***Relaciones de sincronización***	Esta vista es ortogonal a las vistas de módulos y conceptual. Se corresponde con la vista de procesos del módelo 4+1
Física	Describe la asignación de componentes software a procesadores (mapeo software/hardware)	Procesadores y enlaces de comunicaciones/ ***Se comunica con***	De especial interés en sistemas de procesamiento paralelo y distribuido. Muy relacionada con los atributos de rendimiento, disponibilidad y seguridad. Se corresponde con la vista de física del módelo 4+1
De uso	Describe las relaciones de uso entre los módulos y funciones del sistema	Procedimientos y módulos/ ***Asume la existencia de***	Especialmente relacionada con los atributos de extensibilidad, reusabilidad. Muy de tener en cuenta si se opta por un modelo de crecimiento incremental. Puede considerarse parte de la vista de desarrollo del módelo 4 +1
De llamadas	Describe la estructura de llamadas entre funciones o procedimientos	Procedimientos/ *Invoca a*	Útil para el seguimiento del flujo de ejecución del programa. Puede considerarse un refinamiento de la anterior.
De flujo de datos	Describe el flujo de datos entre los componentes del sistema	Programas, procesos, módulos/ ***Envía datos a***	Muy útil para la trazabilidad de requisitos. Puede considerarse parte de la vista lógica del módelo 4 +1.
De flujo de control	Describe el comportamiento funcional del sistema y su comportamiento temporal	Programas, módulos, procesos, estados del sistema/ ***Se activa después de***	Si el único mecanismo de transferencia de control es la invocación de métodos, esta estructura coincide con la de llamadas Puede considerarse parte de la vista de procesos del módelo 4 +1.
De clases	Describe la estructura de las clases y objetos y sus relaciones.	Clases y objetos/ *Hereda de,* ***Es instancia de*** *Es parte de, etc*	Imprescindible si se utiliza una metodología orientada a objetos. Puede considerarse parte de la vista lógica del módelo 4 +1.

Figura 2.2: Estructuras Arquitectónicas [Bass et al 1998]

2.5.2 Patrones, Estilos de Arquitectura, Modelos de Referencia y Arquitecturas de Referencia.

Asociados al concepto de Arquitectura Software, hay una serie de términos de uso común. Como ocurre con la propia Arquitectura no hay un consenso definitivo sobre su alcance. Sin embargo, aunque estos términos carezcan de definiciones precisas, permiten a los diseñadores razonar sobre sistemas complejos usando abstracciones que los hacen inteligibles. Los patrones y estilos arquitectónicos y los modelos y arquitecturas de referencia proporcionan el contenido semántico necesario para describir las propiedades del sistema, su evolución en el tiempo, sus paradigmas computacionales y sus relaciones con otros sistemas similares. Las descripciones que aparecen en esta sección se han confeccionado a partir de las dadas en [Gamma et al 1995], [Buschmann et al 1996], [Klein 1999a, 1999b] y [Bass et al 1998].

Patrones y estilos arquitectónicos

Un patrón es una solución recurrente a un problema estándar. Los patrones de diseño capturan las estructuras estáticas y dinámicas de soluciones que funcionan de forma satisfactoria dentro de ciertos contextos o aplicaciones. Así, cada patrón define ciertos componentes y las reglas mediante las cuales pueden relacionarse con objeto de resolver un problema software específico.

Obsérvese que los patrones son definidos por inducción. Los patrones software describen soluciones que han sido probadas con éxito al aplicarlas una y otra sobre los mismos problemas. Los patrones se documentan usando un formato en el que se describe no sólo el patrón, sino su nivel de abstracción, su ámbito de aplicación y las consecuencias de su uso, entre ellas su relación con ciertos atributos de calidad. La documentación de patrones pone a disposición de los diseñadores soluciones probadas para problemas comunes, favorece el uso de esquemas de reconocida eficacia y constituye un paso más hacia la reutilización del software.

La catalogación de patrones se remonta a los años 90, en los que se publican los trabajos de Coad [Coad 1992], Gamma [Gamma et al 1995] y Buschmann [Buschmann et al 1996], que describen una serie de patrones para el diseño orientado a objetos, aunque su aplicabilidad no se reduce necesariamente a ese ámbito. [Klein et al 1999a, 1999b] propone una serie de estilos arquitecturales (*ABAS: Attribute-Based Architectural Styles*), que relaciona con atributos de calidad específicos. Otros ejemplos pueden encontrarse en [Aarsten et al 1996] (patrones para el diseño de sistemas de control concurrentes y distribuidos), [Islam et al 1996] (sistemas tolerantes a fallos) y [Mckenney 1996] (programación paralela), por citar sólo algunos.

A medida que aumenta la experiencia en el manejo de tales patrones, los diseñadores pueden integrar grupos de patrones relacionados para formar *lenguajes de patrones*. Los lenguajes de patrones definen estilos arquitectónicos que guían a los diseñadores a utilizar los patrones para construir sistemas completos. Un lenguaje de patrones puede generar un sistema software, o puede guiar su construcción, incluyendo su organización, sus procesos, sus interfaces de usuario, su diseño y su aprendizaje.

Como se ha dicho, el estudio y documentación de patrones es un campo en auge desde mediados de los 90, sin embargo aún queda mucho trabajo por hacer, en especial en lo referente a la asociación de patrones con atributos de calidad [Klein 1999a, 1995b], a la clasificación de los patrones y a la definición de lenguajes de descripción de patrones [Coplien 1995][14].

Un estilo de arquitectura es una descripción de un conjunto de componentes y un patrón o patrones que describen la transferencia de control y datos entre los mismos. Un estilo define una serie de restricciones respecto al tipo de componentes que pueden utilizarse y a las formas en que dichos componentes pueden interaccionar. La frontera entre patrón y estilo no está clara en la literatura, ni tampoco en las

[14] El lector puede encontrar una excelente base de datos sobre patrones en [Brad 2000], en donde se ofrecen gran cantidad de enlaces a publicaciones sobre el tema, incluyendo todos los clásicos.

descripciones aquí ofrecidas. La construcción cliente/servidor ¿es patrón o estilo?. Desde luego es una solución probada a un problema concreto. Sin embargo pueden adoptarse diferentes patrones para su adecuación a un problema particular. Los patrones pueden ser vistos como una *micro-arquitectura* que define un estilo *puro* o primario. Un estilo de arquitectura viene generalmente definido por una combinación de patrones. Puede así entenderse el estilo como una abstracción de nivel más alto que los patrones, que define una codificación particular de elementos de diseño y combinaciones formales de los mismos, limitando su número y el tipo de sus posibles combinaciones y mecanismos de interacción. Esta distinción no es seguida por todos los autores, y es habitual que estilos arquitectónicos como la descomposición del sistema en niveles, los sistemas cliente/servidor, los diseños *pipe-filter*, las máquinas virtuales, las pizarras, etc., sean definidos como patrones en la literatura referente al tema.

Modelos de Referencia y Arquitecturas de Referencia

Un modelo de referencia es una descomposición estándar de un problema conocido en unidades funcionales que trabajan cooperativamente para resolverlo. Un modelo de referencia no es una arquitectura, sino una descripción de la división de la funcionalidad de un sistema software en diferentes piezas y de la forma en que fluyen los datos a través de ellas. Los modelos de referencia son típicos de dominios muy maduros, en los que existe un consenso entre los diseñadores acerca de la descomposición funcional del sistema.

Ejemplos de modelos de referencia son la torre OSI (*Open System Interconnection*) y los modelos de referencia para el diseño de compiladores**.**

Una arquitectura de referencia es una correspondencia entre un modelo de referencia y los componentes software que implementan la funcionalidad definida en el mismo [Bass et al 1998]. La correspondencia entre las unidades funcionales definidas en el modelo y los componentes de la arquitectura de referencia no es necesariamente uno a uno.

Una arquitectura de referencia define la infraestructura común a todos los sistemas del dominio considerado, los componentes o subsistemas que incluyen y las interfaces que deben ofrecer dichos componentes o subsistemas. Disponer de tal arquitectura facilita enormemente el desarrollo de nuevas aplicaciones, pues permite por un lado la reutilización de modelos y componentes y por otro ofrece un marco para el desarrollo de los mismos.

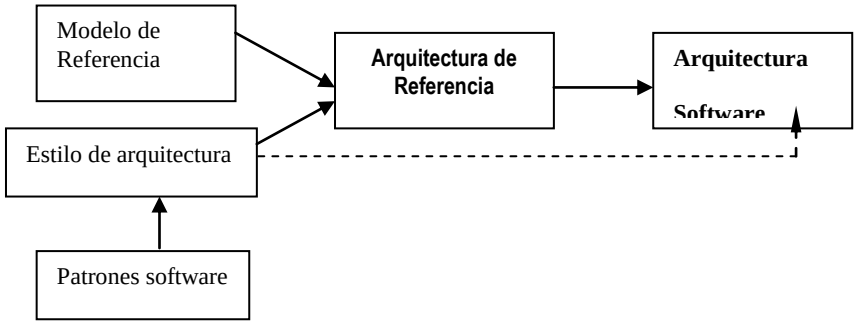


Figura 2.3: Relación entre patrones, estilos, modelos de referencia y arquitecturas.

Los patrones, estilos, modelos y arquitecturas de referencia constituyen los pasos que hay que seguir hasta definir la arquitectura del sistema (figura 2.3). Las arquitecturas de referencia se construyen a partir de un modelo de referencia en el que las unidades funcionales y sus relaciones se definen siguiendo un determinado estilo o patrón. La arquitectura de referencia aunque ya define pautas claras de implementación, sigue siendo un artefacto bastante general que tendrá que especializarse para definir *una* arquitectura software. En aquellos dominios donde existen modelos y arquitecturas de referencia, el arquitecto software puede aprovechar las soluciones que éstas le proporcionan.

2.5.3 Lenguajes de descripción de arquitecturas.

Iniciábamos este capítulo afirmando que una arquitectura sólo puede ser evaluada si se dispone de *una buena descripción de la misma*, un conjunto de herramientas adecuado y una metodología que guíe el proceso. Un lenguaje de descripción de arquitecturas o ADL (de sus siglas en inglés ADL: *Architecture Description Languaje*) debe al menos [Clements et al 1996]:

- Ser capaz de describir todas las estructuras del sistema, tanto las estáticas como las dinámicas.
- Estar libre de ambigüedades y disponer de un conjunto de símbolos, componentes y conectores, con una semántica lo suficientemente rica como para capturar todas las propiedades externamente visibles de los componentes de cada estructura.
- Debe permitir la descripción de la arquitectura en diferentes niveles de detalle, representando la información con distintos grados de granularidad.
- Debe incluir reglas que aseguren la consistencia y completitud de la arquitectura para servir de soporte a los procesos de diseño, refinado y validación de la misma.
- Debe ser capaz de representar, directa o indirectamente, diferentes estilos arquitecturales.
- Debe constituir una base sólida para la posterior implementación del sistema, pero sin comprometer una implementación específica del mismo.
  - Debe incluir capacidades para *mapear* distintas implementaciones sobre una misma arquitectura.
  - Debe dar la posibilidad de generar prototipos y código a partir de la arquitectura.

Un ADL que cumpla estas características puede integrarse en un entorno de desarrollo. Dentro de este entorno el ADL permite:

- Describir el sistema con diferentes grados de detalle, utilizando notaciones gráficas y textuales. Los subsistemas son descritos de acuerdo con la información que aceptan o producen. Puede asociarse a cada componente la descripción de su comportamiento externamente visible, incluyendo los eventos que produce y a los que responde, los mensajes que envía y recibe, su comportamiento temporal y los recursos que utiliza.
- Analizar, también con diferente grado de detalle, el comportamiento del sistema a través de escenarios y casos de uso o, si se dispone de las herramientas adecuadas, mediante análisis formales.
- Refinar incrementalmente el diseño de cada componente de acuerdo con los resultados de los análisis, combinando componentes descritos con diferente detalle. En cualquier momento es posible inspeccionar cualquier componente y la información relativa a los resultados de los análisis que le conciernen.
- Obtener una implementación de los componentes. Una vez que los análisis demuestran un comportamiento satisfactorio de los componentes, puede generarse el código correspondiente a los mismos o al menos una plantilla o esqueleto de sus interfaces.

Todo esto da una idea de la importancia de los ADLs en todo el proceso de análisis y diseño. Esta importancia se ve magnificada si pensamos que ninguno de los puntos anteriores podría lograse sin disponer de ellos. Sin embargo, es inmediato preguntarse si existen realmente ADLs que cumplan las características que acaban de describirse. Clemens, en [Clements 1996b], pasa revista a los ADLs

existentes en aquel momento, concluyendo que ninguno de ellos satisfacía por completo estas expectativas. El estudio de Clemens es excelente y de obligada lectura. Sin embargo, desde el año 1996 a esta parte los ADLs han evolucionado y ha aparecido UML. Aunque UML no soluciona todos los aspectos con los que puede encontrarse un diseñador de sistemas, y probablemente sufrirá revisiones de alguna importancia, se ha convertido en un punto de referencia para la descripción de sistemas software. Un estudio mucho más reciente sobre ADLs es el realizado por Medvidovic y Taylor en [Medvidovic et al 2000]. Ambos autores definen lo que a su juicio es un ADL, describen sus características y realizan una comparativa bastante exhaustiva de los ADLs existentes en la actualidad. Aunque algunos de sus criterios no son compartidos por otros autores (p.e. no consideran a UML un verdadero ADL[15]) el artículo realiza un excelente estado de arte sobre ADLs y cita todas las referencias significativas sobre el tema.

2.5.4 Herramientas de diseño

Existen ya en el mercado una buena cantidad de herramientas de diseño que permiten al ingeniero o programador definir el modelo conceptual asociado a un problema y generar a partir de él la estructura o incluso el código de la aplicación[16]. Es posible distinguir tres aproximaciones a esta línea:

- **Herramientas de enfoque estructural.** Proporcionan generación de código a partir de las estructuras estáticas definidas (clases, atributos y tipos) y de las relaciones entre ellas. Entre ellas cabe destacar *System Architect* [Popkin 1997] y *Rational Rose* [RationalRose 2000], ésta última incorporando UML, si bien con algunas restricciones sobre el lenguaje definido en [OMG 2000]. Estas herramientas no generan código relativo al comportamiento, por lo que los desarrolladores deben completar manualmente el código generado.
- **Herramientas con enfoque de comportamiento**. Incorporan la generación de código a partir de máquinas de estados y de la especificación asociada a sus acciones. Una aplicación clara de este enfoque es la simulación y depuración de sistemas. Dentro de esta línea pueden mencionarse *Rhapsody* de *i-Logix* [Ilogix 2000] y *ObjectTime* [ObjectTime 2000], basada en el método *ROOM* [Selic et al 1998] (*Real Time Object Oriented Method*).
- **Enfoque de traducción.** En este enfoque se incorpora un modelo de arquitectura (patrones, plantillas, etc.) que marca las reglas en el proceso de obtención de código (estrategias de implementación). Las herramientas que encajan en este enfoque proporcionan arquitecturas por defecto que pueden ser adaptadas. De esta forma, el modelo de arquitectura es independiente de la aplicación. La calidad del código obtenido depende de la definición del modelo de arquitectura. Dentro de esta línea se encuentra *BridgePoint* [Bridge 2000] de *Project Technology*, basada en el método de Shlaer-Mellor [Shlaer et al 1988], aunque incorpora UML en las últimas versiones

2.5.5 Rational Rose, Mast y UML

Puesto que Rational Rose 2000 con la extensión UML-Mast son las herramientas empleadas en este trabajo de tesis y UML el ADL utilizado, es necesario comentar, aunque sea brevemente, algunas de sus características. Rational ROSE'2000 [RationalRose 2000] es una herramienta de

[15] No son los únicos autores que opinan así. Los argumentos aportados en el artículo respecto de este punto son muy consistentes. Además, apuntan las extensiones que deben realizarse en UML para que pueda ser considerado como un verdadero ADL.

[16] La base de datos más completa que sobre herramientas de ingeniería software que ha encontrado el autor está en [CASE 2000]

diseño y modelado visual que se inscribe dentro de un grupo de herramientas software desarrollado por Rational que proporcionan soporte para un desarrollo iterativo y basado en componentes. Aunque estos dos aspectos son conceptualmente independientes, su uso combinado es bastante natural. Rational ROSE'2000 utiliza la notación UML, soportando los 9 tipos de diagramas definidos por éste. La correspondencia entre los diagramas de UML y las vistas de Krutchen se muestra en la figura 2.4.

Rational ROSE'2000 es una herramienta válida para realizar la descripción de una arquitectura, sin embargo no incorpora utilidades de simulación que en el caso que nos ocupa son importantes para determinar el comportamiento temporal de la arquitectura. Por ello se utilizará una extensión a Rational ROSE'2000 que permite realizar el análisis temporal del sistema. La herramienta UML-Mast [Drake et al 2000] es una utilidad desarrollada por la Universidad de Cantabria para el modelado, análisis y diseño de sistemas distribuidos de tiempo real. UML-Mast consiste en un *framework/profile* desarrollado para la herramienta ROSE'2000 de Rational que permite formular una vista complementaria en la descripción del sistema (*Mast RT View*) la cual constituye un modelo de tiempo real de su comportamiento. La modularidad de la *Mast RT View* coincide con la modularidad de la vista lógica lo cual permite disponer tanto del modelo de tiempo real del sistema como del modelo de cada clase lógica e incluso del modelo de cada método de su interfaz. Con ello, la herramienta pretende conseguir que los modelos de tiempo real sean reutilizables y puedan constituir parte de la especificación de componentes de tiempo real.

Aunque Rational Rose 2000, UML-Mast y UML proporcionan un marco de desarrollo suficiente para desarrollar este trabajo de tesis, presentan también algunos inconvenientes. Desde el punto de vista de esta tesis y sin pretender ser exhaustivo ni completo en la crítica, estos inconvenientes son los siguientes:

Respecto de UML y Rational Rose:

- Las reglas de visibilidad de UML 1.3 son similares a las reglas de Ada 95 para paquetes y en general a las reglas de visibilidad de cualquier lenguaje de programación estructurado en bloques[17]. Estas reglas establecen que los paquetes hijos pueden ver los elementos de sus progenitores, pero no al revés. Este enfoque tiene la ventaja de acercar el lenguaje de diseño a las prácticas comunes de los lenguajes de programación estructurados, sin embargo desde el punto de vista de la ingeniería software tiene un grave inconveniente: La realización de un sistema software depende de la realización de sus subsistemas, pero no al revés. Puesto que la organización de un sistema en subsistemas se expresa mediante paquetes, las reglas de visibilidad de UML 1.3 establecen unas relaciones poco naturales entre los mismos[18].
- Las reglas de visibilidad de UML 1.4 son las mismas que las de UML 1.3. Sin embargo, UML 1.4 amplia los conceptos de modelo y subsistema. Al igual que en UML 1.3 los subsistemas definen un espacio de nombres, pero a diferencia de UML 1.3, los subsistemas no son simples paquetes, sino que constituyen una unidad de comportamiento del sistema físico que ofrece interfaces y tiene operaciones. Sus contenidos pueden particionarse en elementos de especificación y realización y son instanciables bajo ciertas reglas. Aunque los problemas de visibilidad siguen presentes, estas características facilitan la definición de subsistemas. Desgraciadamente Rational Rose 2000 no las soporta.

[17] No así en UML 1.1, en el que las reglas de visibilidad se definen justamente al revés: Los paquetes padres tiene visibilidad sobre los elementos públicos definidos en sus hijos.
[18] El lector interesado puede encontrar una disertación completa sobre este tema en [Schürr et al 2000].

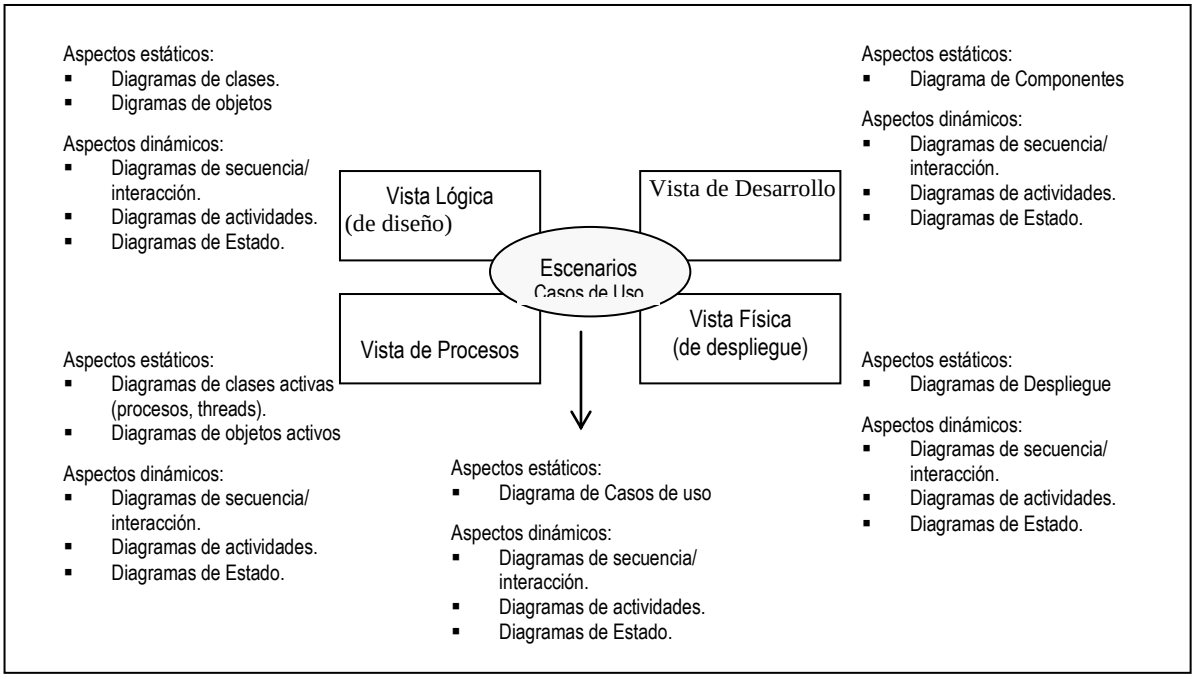


Figura 2.4: Diagramas UML y el modelo 4+1 vistas

- Como ya se ha comentado, un buen ADL debe permitir la descripción del sistema en diversos grados de detalle y combinar las partes del sistema más elaboradas con otras que necesitan mayores refinamientos. UML define tal relación de refinamiento, pero a una escala insuficiente.

- La descripción de las redes de comunicaciones en UML es claramente insuficiente. La descripción de sistemas distribuidos no puede realizarse correctamente sin la inclusión de la infraestructura de comunicaciones en el modelo. Aunque en los sistemas de teleoperación la presencia de enlaces de comunicaciones está limitada a unos pocos enlaces que se usan con ciertas restricciones, se hace difícil su modelado con UML.

- Uno de los principales requisitos de la arquitectura objeto de esta tesis es su adaptabilidad frente a cierto tipo de cambios y la posibilidad de definir sistemas con elementos opcionales y alternativos. La definición de multiplicidades en las relaciones ayuda a caracterizar opciones y alternativas, pero no es suficiente para definir explícitamente los *puntos de variación* de la arquitectura, es decir aquellos componentes o relaciones cuya presencia y características dependen de las opciones y alternativas elegidas para el sistema final. En este sentido Bachmann y Bass en [Bachmann et al 2001] definen una serie de criterios que podrían incorporarse al modelo como etiquetas o restricciones.

- Finalmente, UML no considera (al menos no explícitamente) a los conectores como entidades de primera clase. Aunque este aspecto será tratado con mayor extensión en capítulos posteriores, su repercusión sobre el rediseño de la arquitectura justifica dedicarle aquí algunas líneas. Recordemos, (1) la arquitectura de un sistema define sus componentes y las relaciones entre los mismos, (2) las arquitecturas de referencia deben capturar la variabilidad entre los diferentes sistemas del dominio considerado. Pero *¿y si esa variabilidad se manifiesta precisamente en la necesidad de adoptar diferentes patrones de*

interacción entre componentes?. Puesto que las relaciones entre componentes están representadas precisamente por los conectores que los unen, los conectores deben ser elementos parametrizables de la arquitectura.

Consideremos el siguiente ejemplo. Supongamos que la arquitectura tiene dos componentes, *A* y *B*, que se relacionan mediante el conector *ab*. Supongamos ahora que en algunos sistemas de la familia el patrón de interacción más conveniente entre A y B es un cliente/servidor con llamadas síncronas de *A* a *B*. En ese caso la semántica de *ab* debe representar dicha relación cliente/servidor. Supongamos ahora que en otro sistema con otros requisitos de rendimiento o "reactividad" el patrón de interacción más adecuado es un patrón observador. En este caso la semántica de *ab* debe representar al patrón observador. Aun cuando es posible parametrizar este tipo de relaciones en UML a través de los mecanismos de extensión del lenguaje, el uso de tales mecanismos de una forma consistente supone una carga de trabajo mayor de lo que podría parecer a primera vista[19].

Este inconveniente está relacionado con las carencias de la relación de refinamiento definidas en el actual estándar UML (es necesario relacionar claramente los niveles de representación en los que no es relevante el patrón de interacción concreto entre A y B con aquellos en los que se refina) y con la dificultad de representar elementos opcionales en los diagramas UML (representación de las opciones y posibilidad de optar a cierto nivel de abstracción por un patrón de interacción u otro).

Respecto de UML-MAST:

- No soporta el análisis de transacciones con eventos múltiples. Aunque dichas transacciones pueden ser modeladas y se generan los correspondientes ficheros para el análisis, éste no es soportado por la herramienta. Este inconveniente puede soslayarse particionando la transacción original en tantas transacciones como eventos considerados.
- Puesto que UML-Mast está pensado para integrarse en Rational Rose 2000, se ve limitado por las características de dicha herramienta. Así, aunque los diagramas de objetos están especificados en UML, no son soportados completamente por Rational Rose 2000. En particular, Rational Rose 2000 no permite introducir los valores de los atributos de los objetos. Esto obliga a UML-Mast a definir los diagramas de objetos como diagramas de clases, representando los objetos a través del símbolo de la clase.
- Finalmente, UML-Mast no garantiza la consistencia entre la vista lógica y la vista de tiempo real. Dichas vistas se definen en paralelo, pero son independientes. Cualquier cambio en una de ellas no se refleja en la otra. La consistencia debe gestionarse *a mano*.

2.6 Frameworks orientados a Objeto

2.6.1 Definición y Clasificación

Según [Johnson 1988] un *framework* es una aplicación semi-completa y reutilizable que puede especializarse, aplicando ciertos mecanismos de extensión o de configuración, para producir

[19] Y difícilmente justificable si los perfiles que se definan no van a reutilizarse en otros desarrollos.

aplicaciones específicas, y que incluye a menudo una infraestructura de ejecución. Según [Fayad et al 1997] los *frameworks* pueden clasificarse por su alcance o ámbito de aplicación y por sus mecanismos de extensión. Así:

Tipos de frameworks según su alcance:

- *Frameworks* de Infraestructura: Orientados al desarrollo de infraestructura software, sistemas operativos, software de comunicaciones, interfaces gráficas de usuario, etc.
- *Frameworks Middleware*: Orientados a la integración de componentes en sistemas distribuidos. Han adquirido un gran protagonismo en los últimos años. CORBA, DCOM y .NET entran dentro de esta categoría.
- *Frameworks* de Aplicación: Orientados a un dominio de aplicaciones. Son los más difíciles de desarrollar y amortizar. Su desarrollo requiere de un conocimiento exhaustivo del dominio y sus posibilidades de aplicación se ven limitadas por el mismo. A día de hoy son los menos frecuentes.

Tipos de frameworks por mecanismos de extensión.

- *Frameworks* de caja blanca: La extensibilidad se apoya en las características de los lenguajes OO: Herencia y enlace dinámico. Su uso requiere de un conocimiento muy profundo de sus características y de su estructura interna.
- *Frameworks* de caja negra: La extensibilidad se apoya en la definición de interfaces a través de las cuales los componentes pueden integrarse mediante composición de objetos. Son más difíciles de desarrollar puesto que requieren un diseño muy preciso y refinado de las interfaces.

Los principales beneficios de los frameworks son:

- Modularidad: Los frameworks favorecen la modularidad mediante la encapsulación de los detalles de implementación detrás de interfaces estables.
- Reusabilidad: Mediante la definición de componentes genéricos que pueden ser utilizados para crear nuevas aplicaciones. La reutilización de los componentes de un framework puede mejorar sustancialmente la productividad y los atributos de calidad del software.
- Extensibilidad: Proporcionando mecanismos explícitos que permitan a las aplicaciones extender sus interfaces estables (*hook methods*). La extensibilidad del framework es esencial para asegurar el rápido desarrollo de nuevas aplicaciones, características y servicios.
- Inversión del control: La arquitectura *run-time* de un framework se caracteriza por una inversión del control. Cuando ocurre un evento, el *dispatcher* del framework reacciona invocando *hook methods* de manejadores pre-registrados, que realizan un procesamiento del evento específico de la aplicación. La inversión del control permite al *framework* (en lugar de a las aplicaciones) determinar el conjunto de métodos específicos de la aplicación que deben invocarse cuando ocurre un evento.

2.6.2 Problemas de los frameworks

Los frameworks son el máximo exponente de la reutilización en las tecnologías orientadas a objetos. Sin embargo, presentan también una serie de problemas o inconvenientes.

Los frameworks requieren un gran esfuerzo de desarrollo. Como se afirma en [Roberts et al 1996] el desarrollo de frameworks reutilizables no ocurre simplemente sentándose a pensar acerca del dominio del problema. Nadie tiene la capacidad suficiente para dar con las abstracciones correctas. Los *frameworks* son el resultado de muchos años de trabajo, pues en ellos se resume la experiencia adquirida en muchos desarrollos.

Los desarrolladores de *frameworks* se enfrentan a muchos compromisos de diseño. Uno de los más cruciales es determinar que componentes del framework deben ser variables y cuáles deben ser estables. Una insuficiente variabililidad dificulta la configuración o extensión del framework impidiendo que pueda acomodarse a los requisitos de diversas aplicaciones. De la misma manera, una insuficiente estabilidad hace difícil a los usuarios entender el framework. Un *framework* debe ser lo suficientemente sencillo para ser aprendido y utilizado, y al mismo tiempo debe proporcionar las suficientes características y prestaciones para ser usado eficientemente. En cualquier caso, es necesario un conocimiento exhaustivo del dominio para el que está orientado el *framework*, el cual debe incluir una teoría del dominio del problema y es siempre el resultado del análisis del dominio bien explícito y formal, bien implícito e informal.

La mantenibilidad de las aplicaciones se dificulta en algunos casos:

- ✓ Como los frameworks inevitablemente evolucionan, las aplicaciones que los usan deben evolucionar con ellos.
- ✓ La validación y la corrección de defectos se hace más difícil. Los componentes genéricos son difíciles de evaluar en abstracto. La inversión del control y la falta de un flujo de control explícito dificultan la trazabilidad de la aplicación y puede ser difícil distinguir entre los errores introducidos por el programador y los debidos al *framework*.
- ✓ Por último, los distintos frameworks suelen ser incompatibles entre sí, de forma que optar por uno significa ligarse a un cierto tipo de tecnología.

2.6.3 Frameworks y Patrones de Diseño.

Los *frameworks* y los patrones de diseño están estrechamente relacionados, pero son conceptos diferentes. Mientras los patrones están enfocados a la reutilización de diseños abstractos o microarquitecturas, los *frameworks* están enfocados a la reutilización de diseños, algoritmos e implementaciones concretas realizadas en un lenguaje de programación concreto. Algunos autores [Schmidt 1997] ven los *frameworks* como reificaciones concretas de familias de patrones de diseño enfocados a un dominio de aplicación determinado.

Patrones y frameworks son conceptos sinérgicos, pero no subordinados. La reutilización sistemática de componentes software complejos requiere el conocimiento de los patrones de diseño fundamentales en los que se basan los mismos. Los patrones asisten el desarrollo de software reutilizable porque expresan la estructura y colaboración de los componentes a un nivel de abstracción mayor que el código fuente y los modelos de diseño orientados a objeto. Por consiguiente, los patrones facilitan la reutilización de la arquitectura software incluso cuando otras formas de reutilización no son posibles (debido a diferencias fundamentales entre los mecanismos de los sistemas operativos o entre las características de los diferentes lenguajes de programación).

2.7 Líneas de Producto.

En la programación orientada a objetos la unidad básica de reutilización es la clase[20]. Sin embargo, este enfoque excluye la reutilización de elementos software de mayor grano, que pueden corresponderse con partes fundamentales del sistema a construir. Por otro lado, la reutilización del software para ser efectiva requiere:

1. Una planificación cuidadosa. La reutilización oportunista no es efectiva en la práctica y produce sistemas inconsistentes.
2. Un buen conocimiento del dominio del problema que permita identificar los componentes claves de los sistemas.
3. De técnicas de análisis y diseño que promuevan la reutilización.

Dentro de este marco, uno de los enfoques más prometedores para conseguir la reutilización de componentes software son las líneas de producto. Las líneas de producto pueden definirse como un conjunto de productos estrechamente relacionados por su funcionalidad y cuyo objetivo es satisfacer las necesidades de un cierto segmento del mercado. La estrategia de las líneas de producto es capturar las partes comunes entre productos y tratarlas como entidades de primera clase. En la terminología técnica de las líneas de producto a estas partes comunes se las denomina *core assets* o simplemente *assets*[21]. Al contrario que en el desarrollo de un único producto, el trabajo se divide entre la ingeniería de dominio (*core assets*) y la ingeniería de aplicación (productos individuales). Los *core-assets* se desarrollan tomando en consideración todos los sistemas. La ingeniería de aplicación consiste en la configuración y ensamblado de los *assets* y en el desarrollo e integración de componentes específicos de producto. Los *assets* principales son la *arquitectura base* de la línea de producto, el conjunto de componentes reutilizables y los productos resultantes.

Algunas de las definiciones propuestas para las líneas de producto son las siguientes:

*"Una línea de productos consiste en una arquitectura de línea de productos y en un conjunto de elementos software reutilizables que han sido diseñados para su incorporación en dicha arquitectura. Adicionalmente, la línea de productos incluye los productos que han sido desarrollados utilizando los "*assets" *mencionados." [Bosch 2000]*.

*"Una línea de productos software es un conjunto de sistemas intensivos en software que comparten un conjunto común y gestionado de características que satisfacen las necesidades específicas de un particular segmento de mercado o misión y que son desarrollados a partir de un conjunto de "*assets" *comunes de una forma determinada." [Clements et al 2000]*.

"Conjunto de productos que comparten un conjunto común de requisitos, pero que exhiben una variabilidad significativa respecto de los mismos" [Griss 2000].

Los componentes de una línea de producto se desarrollan considerando todos los productos de la línea. La motivación es compartir los costes de desarrollo y mantenimiento entre todos los productos. *La restricción es que la evolución de los productos individuales debe ser conforme con la evolución de la arquitectura y viceversa.* Las líneas de producto exigen por tanto una gran capacidad organizativa para gestionar su desarrollo, uso y mantenimiento. En cualquier

[20] Para la reutilización de la implementación.
[21] Dada la dificultad de encontrar un término castellano para *asset* que sea comúnmente aceptado, se utilizará el término en inglés.

caso, las líneas de producto no aparecen de forma accidental, sino que requieren un esfuerzo considerable por parte de la organización interesada.

Como se ha dicho, las líneas de producto son uno de los enfoques más prometedores para conseguir la reutilización del código. Las explicaciones incluidas en este apartado constituyen una breve introducción al tema. El lector interesado puede consultar [Bosch 2000] y [Clements et al 1998] y las páginas web del SEI dedicadas a líneas de producto [SEI-lpd]. En dichas referencias pueden encontrarse estudios exhaustivos sobre el tema y referencias a otras muchas publicaciones.

2.8 Desarrollo software basado en componentes.

El desarrollo basado en componentes o CBD (de sus siglas en inglés: *Component Based Development*) ofrece una forma flexible para el desarrollo eficiente de soluciones software a partir de componentes reutilizables. En el marco de trabajo que ofrece el CBD las aplicaciones se diseñan a partir de unos componentes que cumplen una serie de especificaciones predefinidas que pueden ensamblarse para dar lugar a aplicaciones completas. En [D'Souza et al 1999] se define el CBD como:

"Un enfoque de desarrollo del software en el que todos los elementos software (desde el código ejecutable a las especificaciones de las interfaces, arquitecturas y modelo de negocio) y a todas las escalas (desde aplicaciones y sistemas completos hasta las partes más pequeñas) pueden ser construidas ensamblando, adaptando y conectando componentes existentes en una variedad de configuraciones"

El CBD toma muchas de los conceptos de la programación orientada a objetos, pero carece del grado de madurez que ha alcanzado ésta. Para empezar, mientras que toda la comunidad software está de acuerdo en el significado de *objeto*, existen muchas definiciones de componente. Algunas de ellas son:

"Un componente software es una unidad de composición que ofrece interfaces explícitamente especificadas, requiere de unas interfaces e incluye unas interfaces de configuración, incluyendo además atributos de calidad" [Bosch 2000].

"Un componente es una implementación opaca de funcionalidad, utilizable por terceras partes para la composición de sistemas y que cumple un modelo de componentes" [Bachmann et al 2000]

"Un componente es una parte modular, distribuible e intercambiable de un sistema, que encapsula su implementación y presenta un conjunto de interfaces. Un componente está especificado por uno o más clasificadores (clases de implementación) que residen en él y puede ser implementado por uno o más elementos software (ficheros binarios, ejecutables o de script)" [OMG 2001]

"Un componente es una unidad de composición de aplicaciones software, que posee un conjunto de interfaces y un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes de forma independiente, en espacio y en tiempo" [Szyperski et al 1997].

Puesto que los componentes pueden usarse para componer otros componentes un aspecto clave del CBD es la posibilidad de deducir las propiedades de un componente a partir de las

propiedades de los componentes que lo constituyen. Sin embargo, este es un extremo que está aún muy lejos de conseguirse [Bachmann et al 2000].

2.8.1 Modelos de componentes.

De acuerdo con éstas definiciones, los componentes constituyen unidades de composición reemplazables y configurables a partir de los cuales pueden construirse sistemas como mecanos. La pregunta es, *¿Quién o qué determina como definir, implementar, ensamblar e instalar dichos componentes?* La respuesta la ofrece la definición de [Bachmann et al 2000]: El modelo de componentes.

"*Un **componente** es una implementación software que puede ser ejecutada en un dispositivo lógico o físico. Un componente implementa una o más **interfaces**. Estas interfaces reflejan las obligaciones del componente que se describen como **contratos**. Las obligaciones contractuales de los componentes aseguran que los mismos podrán ser desarrollados de forma independiente, pues obedecen ciertas reglas que determinan cómo interaccionan y cómo pueden ser distribuidos (*integrados*) en entornos estándares.*

Un sistema basado en componentes se basa en un pequeño número de tipos de componente, cada uno de los cuales desempeña un papel especializado en el sistema y está descrito, como ya se ha dicho, por una interfaz.

*Un **modelo de componentes** es un conjunto de tipos de componente, sus interfaces y, adicionalmente, una especificación del patrón o patrones de interacción entre tipos de componentes.*

*Un **framework de componentes** ofrece una serie de servicios en tiempo de ejecución para dar soporte y reforzar el modelo de componentes. En muchos aspectos, los frameworks son como sistemas operativos de propósito especial, aunque operando a un nivel de mucha mayor abstracción."* [Bachmann et al 2000]

Y no hay mucho más que añadir, salvo que algunos autores no distinguen entre modelo de componentes y *framework* de componentes. Lo meten todo en el mismo saco.

Puesto que actualmente hay varios modelos de componentes, realizar una programación basada en componentes significa tener que optar por uno de dichos modelos. Cuanto más estandarizado y estable sea el modelo, mayores posibilidades habrá para desarrollar componentes para terceros o para utilizar componentes desarrollados por terceros. Dada la falta de estándares comunes, la adopción de un modelo de componentes supone hoy por hoy ligarse a una arquitectura y a una tecnología. Todo componente que sea conforme con el modelo elegido podrá integrarse siguiendo procedimientos estándar definidos en la especificación del modelo. Todo el que no lo siga necesitará de procedimientos de integración específicos, cuya dificultad dependerá del grado de compatibilidad entre los modelos[22]. Entre los modelos de componentes actualmente disponibles cabe destacar los siguientes:

- Los modelos de componentes de Microsoft:
  - ✓ El modelo COM (Component Object Model) [Microsoft 1995].

[22] La interoperabilidad entre aplicaciones Java-RMI y CORBA es posible en muchos casos. La especificación del modelo de componentes CORBA (CCM) tiene como requisito explícito la interoperabilidad con EEJB. Además, el OMG está trabajando para conseguir interoperabilidad con aplicaciones basadas en tecnología DCOM de Microsoft.

- ✓ El modelo DCOM (Distributed Component Object Model) [Thai et al 1999].
- ✓ El modelo .NET [Microsoft 2001].

- El modelo CCM (Corba Component Model) [CCM 1999].
- El modelo EJB (Enterprise JavaBeans) [Sun 2000].

2.8.2 Interfaces de los componentes. Componentes y objetos.

El concepto de *componente* es mucho más amplio que el de *objeto* en el sentido de que un componente no tiene porqué ser un objeto, ni implementarse como tal, pero es mucho más restrictivo en el sentido de que un componente debe cumplir muchas más condiciones que un objeto[23]. Recordemos la definición de componente proporcionada en [Bosch 2000]

"Un componente software es una unidad de composición que ***ofrece interfaces*** *explícitamente especificadas,* ***requiere*** *de unas* ***interfaces*** *e* ***incluye unas interfaces de configuración****,* ***incluyendo*** *además* ***atributos de calidad****"*

Ningún lenguaje de programación incorpora abstracciones suficientes para soportar el concepto de componente, ni siquiera los lenguajes orientados a objetos[24]. En particular, la capacidad de los lenguajes de programación para definir interfaces es muy limitada.

Un objeto es la encapsulación de un conjunto de datos y un conjunto de métodos que realizan operaciones sobre esos datos o proporcionan algún servicio a otros objetos a través de las invocaciones de sus métodos.

Es decir, los servicios que proporciona un objeto se definen explícitamente, pero no los servicios que a su vez necesita para proporcionarlos. Algunas características importantes de los componentes como la calidad de servicio ofrecida o su comportamiento temporal no pueden expresarse de forma directa utilizando objetos. Esta falta de correspondencia entre objetos y componentes obscurece la implementación de los componentes, ya que no expresa de forma directa los conceptos de diseño. Mientras no se definan lenguajes de programación que soporten estos conceptos es posible acudir a patrones de diseño [Andrade et al 1999].

Algunos lenguajes formales incorporan algunos de los aspectos mencionados en el párrafo anterior, pero presentan otros inconvenientes, como el grado de destreza que requiere su uso y la necesidad de herramientas para obtener código fuente o ejecutables a partir de ellos. La programación orientada a aspectos [Kickzales et al 1997] ofrece algunas claves para modelar e incorporar (o eliminar) incrementalmente algunos de estos requisitos, pero también necesita tiempo para madurar y, sobre todo, para ser incorporada de forma masiva en la producción de software.

2.9 El Ciclo de negocio de la arquitectura

El desarrollo de un producto software de cierta envergadura es un proceso complejo en el que están implicados factores y actores de muy diversa índole, técnicos, sociales y de negocio. Los requisitos

[23] Para una discusión en profundidad sobre las diferencias entre objeto y componente véase [Szyperski 2000]
[24] Eiffiel [Meyer 1998] incorpora muy buenos paradigmas en ese sentido, pero no es un lenguaje de uso común en la industria.

técnicos, por su parte, no se reducen a una mera descripción de la funcionalidad, sino que contemplan cuestiones muy diversas, tales como la mantenibilidad, fiabilidad, seguridad, facilidad de uso, generación de documentación y un largo etcétera.

Los diferentes modelos de ciclo de vida que se han usado para gestionar el desarrollo de los proyectos, desde el modelo en cascada, hasta los más modernos, y hoy en voga, modelos en espiral, proponen diversos enfoques para abordar esta complejidad. Pero, *¿de qué manera se manifiestan las dependencias entre los requisitos técnicos y los factores sociales y de negocio?* En [Bass et al 1998] se propone el modelo de la figura 2.5: el Ciclo de Negocio de la Arquitectura o ABC (de sus siglas en inglés, *Architecture Business Cycle*).

A grandes rasgos, el ABC propone el siguiente esquema. El sistema final es el resultado de su arquitectura, la cual viene dada por el conjunto de decisiones de diseño adoptadas por los ingenieros software. Para llegar a tales decisiones, los ingenieros software se ven influenciados por una serie de factores e intereses que tienen su origen en:

- Los requisitos definidos por los clientes, los usuarios y por la propia organización que desarrolla el proyecto. Estos requisitos definen el conjunto de atributos de calidad que se esperan del sistema final.
- El estado actual de la técnica (*Technical Environment* en la terminología empleada en [Bass et al 1998]).
- Las habilidades y la experiencia del equipo de desarrollo.

El éxito o fracaso del sistema influye a su vez en todos estos factores cerrándose el ciclo.

Los usuarios y los clientes finales, sean o no los mismos, tienen diferentes intereses. El cliente paga el sistema, el usuario lo usa. La organización que desarrolla el sistema, coincida o no con el cliente o con el usuario, tiene como tal otros intereses, relacionados con sus planes de negocio a corto y largo plazo y con su propia estructura. Así, la organización puede estar interesada en obtener beneficios a corto plazo, por lo que no asumirá grandes costes de desarrollo, o puede ver en el sistema la oportunidad que estaba esperando para invertir en una nueva línea de producto. Por otro lado, la organización puede o no tener por sí misma la capacidad de desarrollar el sistema completo o puede estar interesada en subcontratar parte de él. En este caso, estará interesada en una arquitectura más modular posible, que facilite la división del trabajo entre equipos o subcontratistas.

Finalmente, el estado actual de la técnica y las habilidades y la experiencia del equipo de desarrollo influyen decisivamente en la solución adoptada. El equipo de desarrollo tenderá a usar las técnicas y métodos que mejor conoce o que mejor resultado le ha proporcionado en el pasado, condicionando el diseño a su propia experiencia. Por supuesto, aquí también influyen los intereses de la organización desarrolladora, que puede decidir formar a su personal en nuevas técnicas o contratar personal ya adiestrado en ellas.

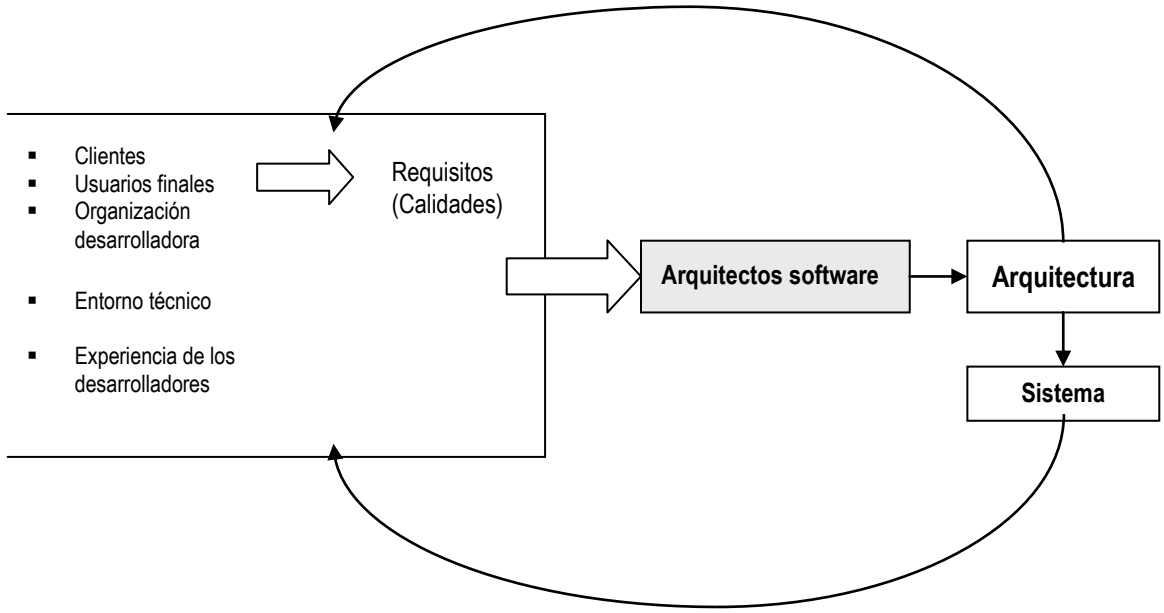


Figura 2.5: El ciclo de negocio de la arquitectura.

La arquitectura adoptada influye a su vez en todos los factores y actores que previamente han influido en ella, bien sea directamente, bien a través del éxito o fracaso del sistema en el que se implementa. Influye en los clientes, que pueden modificar o relajar parte de sus requisitos para adaptarse a un producto ya disponible, con los consiguientes ahorros de tiempo y dinero. Influye en los usuarios que tienden a expresar sus requisitos en función de las costumbres adquiridas. Influye en el equipo de desarrollo, que tenderá a reutilizar estilos, patrones y modelos que les hayan proporcionado buenos resultados, descartando los demás. Influye en los objetivos de la organización desarrolladora, que a la vista del éxito de un producto puede decidir el desarrollo de sistemas similares basados en la misma arquitectura. Los recursos asignados al desarrollo de un sistema tienden a organizarse siguiendo las mismas pautas que la arquitectura del sistema, por lo que la arquitectura influye también sobre la estructura de la organización desarrolladora. Por último, unas pocas arquitecturas tienen el poder de influir en el entorno técnico, por los nuevos conceptos que aportan al estado de la técnica y por el éxito de los productos que siguen sus líneas.

3 Arquitectura y Atributos de Calidad del Software.

3.1 Introducción.

Las arquitecturas se evalúan con relación a los requisitos o atributos de calidad que debe poseer el sistema. Aunque existen definiciones normalizadas de cada uno de estos atributos[25], tales definiciones son demasiado abstractas como para servir de guía en el análisis de *una* arquitectura. La arquitectura es siempre la arquitectura de un sistema, aunque pueda reutilizarse para otros, y un sistema siempre es una solución a un problema concreto o, a lo sumo, la solución a un conjunto de problemas relacionados. Los problemas concretos tienen requisitos concretos, a menudo cuantificables o, al menos, susceptibles de ser descritos con cierta precisión. Los atributos de calidad no existen como conceptos aislados, sino que tienen sentido dentro de un contexto dado ([Boehm 1996], [Bass et al 98], [Kazman et al 2000]).

Por otro lado, los atributos de calidad no son independientes, sino que interaccionan entre sí a través de las relaciones estructurales impuestas por la arquitectura. La mejora de unos, frecuentemente solo puede lograrse a costa de empeorar otros. Por ello es necesario llegar a *compromisos de diseño* (del inglés *tradeoffs*). Hay que optar por unos atributos a costa de otros. Como dice Boehm [Boehm 1996]

"...muchos proyectos, a pesar de tener la funcionalidad e interfaces bien especificadas, fracasan por no tener bien definidos los requisitos de sus atributos de calidad. Encontrar el balance adecuado entre todos los atributos de calidad es un paso importante para lograr el éxito del producto. Para ello, es necesario identificar los conflictos entre los atributos deseados y alcanzar un balance satisfactorio...".

El método ATAM es la respuesta del SEI para alcanzar tal balance. *Pero ¿quién decide los atributos que debe tener un sistema?*

El consenso acerca de los atributos de un sistema involucra a todas las partes implicadas. Clientes, usuarios, ingenieros, todos ellos deben participar en la especificación de los requisitos del sistema. Obviamente, los intereses de cada parte son diferentes, a menudo contradictorios, e incluyen factores sociales y de negocio que se unen a los técnicos. Así, será necesario considerar factores como el coste, los tiempos de vida y de comercialización del producto, el uso de patentes, las estrategias empresariales, etc. Según [Bass et al 1998] y [Boehm 1995, 1996] estos atributos tienen al menos la misma importancia que los técnicos a la hora de

[25] [IEEE-610.12], [IEEE-1061], [ISO-9126]

alcanzar el *balance* que mencionaba Boehm. Las preguntas que debemos hacernos ahora son del estilo: *¿qué es más deseable, un sistema plenamente portable o comercializar el producto antes de 6 meses?*. Las relaciones entre los factores técnicos, sociales y de negocio son complejas y el consenso no es una meta fácil. En el proceso de negociación cada parte implicada tendrá que renunciar a una parte de sus requisitos en aras del éxito del sistema como un todo. En [Bass et al 1998] se define el ABC (*Architecture Business Cycle*), ya descrito anteriormente, para explicar estas interacciones. Boehm propone el sistema QARCC [Boehm 1996] como una herramienta basada en conocimiento (un sistema experto) para ayudar a usuarios y desarrolladores a expresar sus requisitos, identificar los conflictos y llegar a un consenso final sobre las *calidades* del sistema. Por último, ATAM [Kazman et al 2000] trata de establecer un marco de razonamiento en el que dichas interacciones puedan ser descubiertas, puestas en relación con los componentes y conectores definidos en la arquitectura y finalmente resueltas.

3.2 Clasificación de los atributos de calidad

Los requisitos o atributos de calidad pueden clasificarse según la siguiente taxonomía:

- **Requisitos técnicos del sistema**, impuestos por las especificaciones que debe cumplir el sistema final.
- **Requisitos de negocio**, impuestos por la estrategia empresarial en relación con el producto que se pretende desarrollar.
- **Requisitos propios de la arquitectura** o requisitos que debe cumplir toda arquitectura, independientemente de los requisitos técnicos y de negocio.

Los requisitos propios de la arquitectura son la integridad conceptual, la corrección, la compleción y la capacidad de realización. La integridad conceptual está relacionada con la coherencia del diseño. Podría definirse como el esquema general o *leiv motif* que relaciona todas las estructuras del sistema y las hace comprensibles en su conjunto. La arquitectura debe resolver problemas similares de formas similares, evitando características, que aunque puedan considerarse en algún aspecto mejoras, introducen anomalías que las alejan de las líneas básicas del diseño. La corrección es la capacidad de la arquitectura para cumplir sus requisitos. La compleción es la capacidad de cubrir todos los requisitos y la capacidad de realización se refiere a la dificultad de implementación del sistema que supone utilizar una determinada arquitectura. Finalmente, la arquitectura debe dar lugar a sistemas que puedan realizarse mediante las técnicas de implementación disponibles empleando un conjunto de recursos razonable, dentro de los costes y plazos impuestos por los requisitos de negocio.

Tradicionalmente, los atributos o requisitos técnicos de un sistema se han clasificado en dos categorías: requisitos funcionales y requisitos no funcionales. Aunque ésta es la clasificación que sigue este trabajo de tesis, algunos autores la rechazan por dos razones. En primer lugar porque es ambigua (¿dónde está la frontera entre lo funcional y lo no funcional?). Y en segundo lugar, porque asume implícitamente que ambos tipos de atributos pueden abordarse y alcanzarse por separado, relegando el cumplimiento de los requisitos no funcionales a etapas demasiado tardías del ciclo de desarrollo. En [Bass et al 1998] se propone una clasificación alternativa de los atributos del sistema entre: atributos observables en tiempo de ejecución y atributos no observables en tiempo de ejecución. Los atributos observables en tiempo de ejecución se corresponden aproximadamente con los requisitos funcionales y los no observables con los requisitos no funcionales, pero ahora el argumento de clasificación no asume que ambos puedan abordarse por separado, sino simplemente diferentes métodos de medida. El cumplimiento de

los atributos observables en tiempo de ejecución no implica que se cumplan los no observables, y viceversa, pero están estrechamente relacionados. Los atributos de calidad, funcionales o no, dependen de las relaciones estructurales impuestas por la arquitectura. La elección de un determinado patrón o estilo arquitectónico favorece o perjudica la obtención de ciertos atributos. Es decir, los atributos interaccionan entre sí a través de las estructuras del sistema, de forma que la optimización de uno a menudo sólo se logra a costa de empeorar otros.

Los atributos de calidad deben considerarse tanto en las fases de diseño como en las de implementación del sistema, sin embargo no todas las calidades se manifiestan ni se consiguen de igual manera en cada una de estas fases. La arquitectura es crítica para la obtención de la mayoría de estos atributos, que por tanto deben ser evaluados en este nivel, mientras que otros dependen más de los algoritmos utilizados y del estilo y calidad de la implementación. Por supuesto, muchos de los atributos dependen de ambas cosas.

En este apartado se presentaran los atributos de calidad más importantes, profundizando sobre sus relaciones con la arquitectura y la forma de abordarlas en el resto del capítulo. No se pretende dar definiciones rigurosas, que pueden encontrarse en [IEEE-610.12], [IEEE-1061] e [ISO-9126], sino de explicar su relación con la arquitectura y apuntar algunas de sus interacciones. Tanto la clasificación de los atributos que se ofrece como sus definiciones siguen las líneas definidas por el SEI. Dicha clasificación se adapta especialmente bien a los contenidos de esta tesis porque es coherente con las metodologías de evaluación y diseño que se emplean en la misma. Así, las referencias que se han utilizado para la elaboración del siguiente apartado son básicamente [Barbacci 1995, 1996] y [Bass et al 1998]. La figura 3.1 resume las relaciones entre arquitectura y atributos de calidad que identifican estos autores. El lector puede encontrar otras definiciones y clasificaciones en textos clásicos de la ingeniería del software. Especialmente interesante es el apartado dedicado en [Meyer 1997] a discutir los diferentes aspectos de la calidad del software. En [Plessel 1998] se ofrecen definiciones alternativas de los atributos de calidad basadas en las de Meyer.

3.2.1 Requisitos Funcionales. Atributos del sistema observables en tiempo de ejecución

Rendimiento

Número de transacciones por unidad de tiempo o tiempo que tarda el sistema en completar una transacción.

El rendimiento mide la capacidad de respuesta del sistema ya que es una medida del tiempo que éste tarda en responder a un determinado estímulo o evento. El rendimiento depende fundamentalmente de la carga de comunicaciones del sistema. Ciertamente, también depende de otros muchos factores, como la selección de algoritmos adecuados, pero la influencia de las comunicaciones es determinante, especialmente en los sistemas distribuidos, en los que las operaciones de transmisión de datos suelen ser las que más tiempo consumen. Pero las comunicaciones no son sólo *esas* comunicaciones. Cualquier tipo de interacción entre componentes, tales como el paso de mensajes, la sincronización entre tareas o la invocación de métodos son también comunicaciones. Puesto que la arquitectura define la relación entre los componentes, el rendimiento es un atributo estrechamente vinculado a la arquitectura.

Seguridad (*security*)

La seguridad mide la capacidad del sistema para resistir usos y usuarios no autorizados o intentos de dañar los servicios que provee el sistema. La seguridad se clasifica atendiendo al tipo de amenazas a las que debe hacer frente el sistema, clasificándose éstas en dos categorías:

- Daño al servicio
- Acceso no autorizado

La prevención, detección y respuesta a un ataque requiere la adopción de las siguientes estrategias:

- Servicio de autentificación externo, independiente de los usuarios y de los proveedores de los servicios.
- Monitores de red para la inspección y el registro de los eventos de comunicaciones.
- Sistema intermediario (*proxy server*) que canalice las comunicaciones entre usuarios y aisle a los mismos de las comunicaciones.
- Construcción del sistema en la cima de un núcleo (*kernel*) que proporcione los servicios de seguridad necesarios.

Todas estas estrategias implican la adición de componentes especiales que definen reglas muy estrictas a las posibles interacciones entre los componentes y a la forma en que pueden llevarse a cabo. La seguridad, como la eficiencia o rendimiento, es un atributo estrechamente ligado a la arquitectura. Desgraciadamente son atributos contradictorios.

Disponibilidad y Fiabilidad

La disponibilidad mide la proporción de tiempo que el sistema está funcionando. Para caracterizarla se utilizan dos parámetros:

- Tiempo entre fallos (fiabilidad).
- Tiempo de recuperación después de un fallo.

La unión de estas dos medidas en una sola fórmula nos proporciona una medida de la disponibilidad del sistema.

$$\textbf{Disponibilidad} = \frac{\text{Tiempo medio entre fallos}}{\text{Tiempo medio entre fallos + Tiempo medio de reparación}}$$

Estrechamente relacionado con el concepto de disponibilidad está el concepto de **fiabilidad** de un sistema, que suele medirse como el tiempo medio entre fallos.

La optimización de la disponibilidad requiere de la instalación de *componentes redundantes* que sustituyan a los que fallen, de monitores que supervisen el funcionamiento del sistema y de manejadores de errores y servicios de reporte que recuperen al sistema e informen a los usuarios de la ocurrencia de fallos. Nuevos componentes y nuevas interacciones. La disponibilidad, como la eficiencia y la seguridad, es un atributo ligado a la arquitectura.

La fiabilidad está relacionada con la *tolerancia a fallos del sistema*, es decir a la capacidad del sistema a seguir funcionando aunque falle alguno de sus componentes. La **tolerancia a fallos** se

consigue como la disponibilidad mediante redundancia, replicando los componentes más críticos del sistema y sus canales de comunicaciones.

La disponibilidad y la fiabilidad se logran a costa de empeorar el rendimiento y la seguridad. Mayor disponibilidad y fiabilidad, mayor redundancia. Mayor redundancia, mayor carga para el sistema y más número de componentes e interacciones. Mayor número de componentes e interacciones, mayor número de lugares sensibles a un ataque. Sin embargo, los principios de diseño que deben aplicarse para conseguirlas, principalmente una clara división de conceptos, favorecen de paso el logro de otros atributos deseables, tales como la integrabilidad, la testeabililidad, la modificabilidad y la mantenibilidad[26] del sistema, que veremos en la sección siguiente.

Seguridad (*safety*)

La seguridad o capacidad del sistema para evitar situaciones que pudieran producir daños a bienes o personas está relacionada con el contexto en el que se desenvuelve el sistema y está íntimamente ligada al factor humano y al cumplimiento de otros atributos, en especial [Douglas 1999]:

- La usabilidad: La interfaz del sistema debe prevenir contra el uso incorrecto del mismo, evitando que se produzcan situaciones peligrosas.
- La seguridad (*security*): Deben evitarse amenazas que degraden el funcionamiento del sistema.
- La disponibilidad, la fiabilidad y la tolerancia a fallos[27].
- En sistemas de tiempo, real el cumplimiento de los plazos.

Puesto que estos atributos dependen de la arquitectura, la seguridad también depende de la arquitectura.

Funcionalidad

La funcionalidad es la habilidad del sistema para realizar las tareas que se le han encomendado. La funcionalidad está relacionada con una buena distribución de la carga de trabajo entre los componentes del sistema y con la buena coordinación de los mismos.

La funcionalidad, sin embargo, es ortogonal a la estructura y por tanto es en gran medida un atributo no arquitectónico. Muchas estructuras distintas pueden proporcionar una buena funcionalidad. Aún así, la arquitectura puede afectar a la funcionalidad, ya que la división de la funcionalidad entre los componentes del sistema puede estar condicionada por la obtención de otros atributos.

Usabilidad

La usabilidad describe la facilidad y comodidad de uso del sistema. Es un concepto muy vago, que involucra muchos aspectos, algunos arquitectónicos y otros no:

---

[26] La mayor parte de los nombres de los atributos de calidad son anglicismos (*integrability, modifiability, maintainability*, etc.) que no tienen equivalente directo al castellano. Para evitar perífrasis se utilizarán los anglicismos.

[27] [Douglas 1999] ofrece una excelente introducción a los aspectos de seguridad (*safety*) y disponibilidad del sistema, estableciendo relaciones entre ambas y proponiendo patrones arquitectónicos para abordarlas.

- Facilidad para aprender el manejo de la interfaz de usuario.
- Tiempo que tarda el sistema en responder a las peticiones de los usuarios.
- Capacidad de los usuarios para recordar como funciona el sistema.
- Capacidad del sistema para prevenir y corregir los errores del usuario.
- Manejo de errores: Capacidad del sistema para asistir al usuario en la recuperación del sistema.
- Satisfacción de los usuarios: Percepción que tienen los usuarios sobre la dificultad de manejo del sistema.

3.2.2 Requisitos no Funcionales. Atributos del sistema no observables en tiempo de ejecución

Modificabilidad/Mantenibilidad

La modificabilidad se refiere a la capacidad de realizar cambios en el sistema, en sus componentes o relaciones, sin que los efectos de dichos cambios se propaguen por la estructura del sistema ni supongan un coste desproporcionado con relación al beneficio obtenido con los mismos.

La modificabilidad es función del alcance de los cambios necesarios para obtener el comportamiento deseado. Cuanto más locales puedan ser dichos cambios más modificable es la arquitectura. **La modificabilidad es el atributo de calidad más estrechamente alineado con la arquitectura** [Bass et al 1998]. Puesto que la arquitectura define los componentes y sus relaciones, define también la forma en que dichos componentes pueden ser modificados y las condiciones bajo las cuales tales modificaciones pueden llevarse a cabo. Desde un punto de vista arquitectónico los cambios pueden clasificarse, según su alcance, en tres categorías: aquellos que afectan a un solo componente, aquellos que afectan a varios componentes y aquellos cuyos efectos se propagan por toda la estructura del sistema obligando a cambiar la arquitectura.

Los cambios pueden venir motivados por [Bass et al 1998]:

- *Extensión o cambio de las capacidades del sistema*. Adición de nueva funcionalidad o mejora de la existente. La adición de nuevas características recibe el nombre de ***extensibilidad***.
- *Eliminación de capacidades innecesarias*. Simplificación de la funcionalidad, probablemente con objeto de poner en el mercado versiones más sencillas, pero también más baratas, del sistema.
- *Adaptación a nuevos entornos operativos*. Adaptación del sistema para trabajar con otros sistemas operativos y en otras plataformas. Este tipo de modificabilidad recibe el nombre de ***portabilidad***.
- *Reestructuración*. Racionalización de los servicios que proporciona el sistema, modularización, optimización, creación de componentes reutilizables, etc.

Bachmann y Bass en [Bachmann et al 2000] refinan esta clasificación añadiendo dos nuevas fuentes de cambios: los cambios tecnológicos y los cambios en los requisitos de calidad del sistema.

Portabilidad

La portabilidad es la capacidad del sistema para ejecutarse en diferentes entornos de computación. Estos entornos están constituidos por el software (sistema operativo y otros programas de infraestructura) y por el hardware. Un sistema es portable si todas las *suposiciones* acerca de un entorno de computación específico están confinadas en un solo componente o en el peor de los casos en un número pequeño de componentes fácilmente sustituibles o modificables [Garlan et al 1995a].

La encapsulación de las características específicas de una plataforma suele realizarse definiendo una capa o nivel de portabilidad. Esta capa suministra un conjunto de servicios que proporcionan a la aplicación una interfaz abstracta con su entorno. Esta interfaz es independiente de la plataforma y permanece constante aunque cambie el entorno de computación o la implementación de los servicios del nivel de portabilidad. Para que este esquema funcione debe seguirse sin excepciones. Cualquier acceso al entorno de computación debe hacerse a través de los servicios de la capa de portabilidad. El precio que se paga es cierta pérdida de rendimiento, tanto mayor cuanto mayor sea el número de llamadas a los servicios que proporciona esta capa.

Reusabilidad

La reusabilidad es la propiedad mediante la cual los componentes del sistema o su arquitectura pueden ser reutilizados en el desarrollo de otros sistemas. Diseñar con componentes reutilizables significa que el sistema ha sido estructurado de manera que sus componentes puedan ser elegidos de productos previamente construidos, en cuyo caso *reusabilidad* es sinónimo de ***integrabilidad***.

La reusabilidad está ligada a la arquitectura en el sentido de que las unidades de reutilización son los componentes definidos en la misma y la reusabilidad de un componente depende de su grado de acoplamiento con el resto. Para reutilizar un componente en otro sistema es necesario reutilizar también todos los componentes a los que éste se encuentra ligado, es decir todos los componentes que le proporcionan algún tipo de servicio. Cuanto menor sea el grado de acoplamiento de un componente, menos hipotecas tendrá su posible reutilización.

Integrabilidad

La integrabilidad es la capacidad del sistema para que en el mismo trabajen de forma correcta y coordinada componentes desarrollados por separado. La integrabilidad depende de la complejidad externa de los componentes a integrar, de sus mecanismos y protocolos de interacción y de la separación de responsabilidades entre los mismos. Con otras palabras, la integrabilidad depende de las interfaces provistas por los componentes y de la funcionalidad asignada a cada uno de ellos.

Un caso especial de la integrabilidad es la ***interoperabilidad***. La ***integrabilidad*** mide la capacidad que diferentes partes de *un* sistema tienen para trabajar juntas. La ***interoperabilidad*** mide la capacidad que un grupo de componentes de *un* sistema tienen para trabajar con los componentes de *otro* sistema.

Testeabilidad

La *testeabilidad* es la capacidad de encontrar los fallos del sistema a través de un procedimiento de prueba. La *testeabilidad* se refiere a la probabilidad de que, asumiendo que el sistema tiene al

menos un fallo, dicho fallo sea encontrado en la siguiente ejecución del test de prueba. La *testeabilidad* está relacionada con los conceptos de ***observabilidad*** y ***controlabilidad***. Un sistema es *testeable* si es posible *controlar* las entradas y el estado interno de cada componente y *observar* sus salidas.

La testeabilidad está relacionada con aspectos arquitectónicoes tales como la separación de conceptos, el grado de ocultación de la información, la documentación de las estructuras del sistema. El desarrollo incremental favorece la testeabilidad del sistema.

Atributo de Calidad	Arquitectónico	Aspectos arquitectónicos
Observables en tiempo de ejecución (funcionales)		
Eficiencia	Sí	Comunicación entre componentes. División de la funcionalidad. Concurrencia y paralelismo.
Seguridad	Sí	Componentes especializados. Kernels de seguridad, servicios de autentificación.
Disponibilidad	Sí	Tolerancia a fallos con componentes redundantes. Control de la interacción entre componentes. Monitorización de funcionamiento. Manejadores de errores.
Funcionalidad	No	Interacciona con el resto de atributos.
Usabilidad	Parcialmente	Relacionada con el resto de los atributos.
No observables en tiempo de ejecución (no funcionales)		
Modificabilidad	Sí	Modularización, encapsulación de componentes.
Portabilidad	Sí	Nivel de portabilidad.
Reusabilidad	Sí	Bajo acoplamiento y alta cohesión de componentes. Suposiciones arquitectónicoes sobre infraestructuras.
Integrabilidad	Sí	Mecanismos de interconexión compatibles. Interfaces consistentes y bien definidas. Suposiciones arquitectónicoes sobre infraestructuras.
Testeabilidad	Sí	Modularización, encapsulación de componentes.

Figura 3.1: Aspectos arquitectónicos y atributos de calidad [Bass et al 1998].

3.2.3 Atributos de negocio

Además de los atributos de calidad que se aplican directamente al sistema, hay que considerar los requisitos de negocio que se pretenden conseguir con el sistema en el cual se va a implementar la arquitectura. Dichos requisitos influyen sobre la arquitectura al menos en igual medida que los técnicos. De entre ellos, cabe destacar:

- **El tiempo de comercialización** o tiempo disponible para obtener un producto comercializable. Un tiempo de comercialización escaso favorece la adopción de arquitecturas que usen componentes ya existentes, reutilizando los ya desarrollados en sistemas previos[28] o adquiriendo componentes comerciales.
- **El coste del producto.** El desarrollo de un sistema tiene que hacerse dentro de los límites fijados por un presupuesto. Diferentes arquitecturas conducen a diferentes costes a corto y largo plazo.
- **El tiempo de vida del sistema** o tiempo que se pretende que el sistema esté operativo y en funcionamiento. Si se prevé que el sistema va a tener una vida larga, los requisitos de

[28] *Legacy code* o código heredado.

modificabilidad/mantenibilidad y portabilidad son muy importantes. Sin embargo, el cumplimiento de estos requisitos encarece el desarrollo del sistema y dilata el tiempo necesario para obtener un producto comercializable. La organización desarrolladora debe llegar a un compromiso entre los atributos de calidad del sistema final y los recursos que está dispuesta a asignar a su desarrollo.

- **El mercado de destino** condiciona el diseño de muy variadas formas. En el desarrollo de software de propósito general, dirigido al gran público, la plataforma en la que se ejecuta el sistema y su conjunto de características determina el tamaño del mercado potencial. Bajo estas condiciones, la portabilidad y la funcionalidad son fundamentales para conseguir cuota de mercado. Si el producto va dirigido a un mercado grande, pero específico, la estrategia puede ser conseguir una línea de producto a partir de un núcleo común, alrededor del cual se van añadiendo capas que proporcionan servicios específicos.
- **Integración con código ya existente.** Si el sistema debe integrarse con otros ya existentes, deben definirse los mecanismos de integración apropiados. La integración de un sistema con otros ya en funcionamiento es una decisión de negocio que tiene fuertes implicaciones arquitectónicos.

3.3 Arquitectura y Atributos de Calidad

Como se ha comentado, los atributos de calidad dependen en gran medida de la arquitectura. Sin embargo, aunque hay bastante literatura a este respecto, no existe todavía ningún manual que documente de forma completa y sistemática las relaciones entre arquitectura software y atributos de calidad. La ausencia de este manual se debe a las siguientes razones [Bass et al 2000]:

1. La falta de una definición precisa de muchos de los atributos de calidad. El rendimiento, la disponibilidad o la fiabilidad disponen de dichas definiciones e incluso de modelos formales que los caracterizan. Sin embargo, no ocurre lo mismo con otros atributos como la modificabilidad, la usabilidad o la seguridad.
2. Los atributos se solapan e interaccionan, lo cual hace difícil su definición y su clasificación (La portabilidad, ¿es un atributo por derecho propio o un aspecto de la modificabilidad?).
3. El análisis de los atributos no se presta a estandarización. Existen innumerables patrones a diferentes niveles de granularidad y de abstracción. En estas condiciones no es fácil determinar a que nivel debe realizarse el análisis.
4. Las técnicas de análisis son específicas de atributo y no revelan la forma en que los atributos interaccionan entre sí.

Por si esto fuera poco, incluso cuando existen modelos asociados a un atributo (modelos de Markov, teoría de colas, *rate-monotonic analysis*, etc), su utilización para evaluar la arquitectura software se ve limitada por las siguientes razones:

- Los modelos analíticos son modelos generales y su aplicación a problemas específicos requiere de un alto grado de conocimiento de los mismos [Klein et al 1999a]. A menudo es posible establecer una correspondencia entre alguna de las vistas de la arquitectura y un modelo analítico que permita caracterizar su comportamiento respecto de cierto atributo de

calidad[29]. Sin embargo, el esfuerzo que ello implica ha supuesto un gran obstáculo para la aplicación de modelos formales en la evaluación de las arquitecturas.

- Cada modelo específico de atributo impone su particular punto de vista sobre el sistema, que poco tienen que ver con el de los demás. Además, los bloques constructivos manejados por dichos modelos no suelen coincidir con los utilizados para representar la arquitectura[30].
- Los atributos de calidad no son independientes, sino que interaccionan entre sí a través de las relaciones estructurales impuestas por la arquitectura. Cada estructura tiende a favorecer el cumplimiento de ciertos atributos, perjudicando a otros. Boehm en un artículo del año 78 [Boehm 1978] dudaba de la utilidad de realizar medidas sobre atributos individuales, ya que cualquier modificación encaminada a mejorar un atributo específico tendía a producir efectos indeseados sobre los demás. Barbacci en un artículo del año 99 [Barbacci 1999], en el que describe las interacciones entre la seguridad (*safety*) y otros atributos, corrobora las afirmaciones de [Boehm 1978].

Superar estas dificultades es fundamental, ya que sólo caracterizando de forma precisa el efecto concreto que cada patrón y estilo tienen sobre los atributos de calidad, es posible determinar hasta que punto son apropiados para conseguirlos [Klein et al 1999a, 1999b] [Kazman et al 2000].

Todos los catálogos de patrones describen junto a los mismos su influencia en ciertos atributos, aconsejando o desaconsejando su uso para la solución de ciertos problemas. La figura 3.2 muestra un ejemplo del tipo de reglas y recomendaciones que suelen adjuntarse a la definición de los patrones y estilos. El ejemplo resume las reglas de selección que propone [Bass et al 1998] sobre una serie de estilos descritos en [Shaw 1997]. Aunque estas reglas proporcionan criterios válidos para la selección de estilos y patrones, su utilidad se ve limitada por las siguientes razones:

1. Las combinaciones de estilos y atributos que pueden darse en la práctica son incontables. Las relaciones entre estilos y atributos no son fáciles de identificar, en especial si consideramos el diseño de grandes sistemas que incluyen una gran variedad de estilos y a los que se exige el cumplimiento de muchos atributos.
2. En general, no se basan en la aplicación de modelos de medida, sino en razonamientos heurísticos. Por tanto, aunque son útiles para establecer criterios generales, no permiten determinar con precisión sobre que interfaces, componentes y conectores debe actuarse para conseguir los atributos de calidad deseados.
3. No realizan un recorrido sistemático sobre la influencia que un determinado patrón o estilo tiene sobre *todos* los atributos de calidad. Habitualmente se limitan a exponer los casos límite: cuando es más apropiado y cuando es menos o nada apropiado.

Para hacer frente a estos problemas Klein [Klein 1999a] introduce la noción de Estilo Arquitectónico Basado en Atributo (*ABAS: Attribute Based Architectural Style*). El propósito de los ABAS es convertir a los estilos arquitectónicos en una base de conocimiento aplicable al diseño y evaluación de arquitecturas. El objetivo es crear lo que Klein denomina *marcos de razonamiento* asociados a cada estilo arquitectónico. Estos marcos de razonamiento están basados en establecer correspondencias entre los modelos cuantitativos o cualitativos que caracterizan a los atributos de calidad y los estilos arquitectónicos. La idea de Klein es aplicar

[29] Siempre que la arquitectura esté representada con un ADL libre de ambigüedades y con la suficiente riqueza semántica para representar los modelos de medida.

[30] Barbacci en [Barbacci 99] argumenta que una de las razones de la falta de consenso acerca de las estructuras del sistema es precisamente la discrepancia entre los modelos que se utilizan para medir los distintos parámetros de calidad.

modelos de medida ya existentes y de probada eficacia sobre los estilos arquitectónicos. Por ahora, los ABAS definidos en [Kle99b] son específicos de atributo (sólo asocian un atributo a cada estilo) pero pueden extenderse para tener a otros en cuenta

Estilo [Sha97]	Reglas de Selección [Bas98]
Flujo de Datos	La salida es función de una serie de transformaciones secuenciales sobre los datos de entrada. Facilita la integrabilidad (intefaces simples entre etapas consecutivas).
Proceso por lotes secuencial	Existe una única operación de salida sobre una única entrada sobre la que se realizan transformaciones en secuencia
Red de Flujo de datos	La entrada y salida son series recurrentes. Existe una correlación muy alta entre los miembros de cada serie.
Acíclica	No se realimentan resultados a etapas anteriores
Fanout	No se realimentan resultados a etapas anteriores y una entrada se corresponde con más de una salida.
Pipeline	Las transformaciones se aplican sobre flujos de entrada continuos
Tuberías y filtros	Las transformaciones son incrementales.
Control en bucle cerrado	Monitorización continua de la entrada y la salida. Sistemas empotrados que responden a eventos externos.
Invocación y respuesta	Orden de procesamiento fijo. El invocador no puede progresar hasta recibir una respuesta del invocado.
Objetos y ADTs(1)	Favorece la modificabilidad y la integrabilidad (interfaces bien definidas)
ADTs	La representación interna de los tipos de datos puede cambiar
Objetos	Se mejoran los tiempos de desarrollo y prueba mediante la explotación de la herencia y el polimorfismo.
Cliente/Servidor (síncrona)	Modificabilidad, escalabilidad
Niveles	La funcionalidad puede ser dividida entre la específica de la aplicación y la genérica para muchas aplicaciones, pero específica del entorno de computación. Portabilidad, Reutilización de infraestructura.
Componentes Independientes	Plataforma multiprocesador. Componentes débilmente acoplados que pueden progresar de forma (relativamente) independiente. Ajuste del rendimiento (Reasignación de tareas a procesos y reasignación de procesos a procesadores)
Paso de mensajes	No se requiere otro mecanismo de interacción que el paso de mensajes
Procesos Ligeros	Es necesario compartir datos por razones de rendimiento.
Objetos Distribuidos	Paso de mensajes, ocultación de la información, herencia, polimorfismo.
Redes de Filtros	Paso de mensajes, transformaciones secuenciales sobre los datos.
Cliente/Servidor (asíncrono)	Las tareas pueden ser divididas entre productores y consumidores de datos. El cliente no debe ser bloqueado por el servidor.
Heartbeat	El estado del sistema debe ser monitorizado periódicamente. La disponibilidad es un atributo importante.
Broadcast	Los componentes deben sincronizarse a intervalos regulares o no. Disponibilidad.
Paso de testigo	Todas las tareas se comunican entre sí. El estado del sistema debe monitorizarse periódicamnete, pero las tareas son asíncronas. Tolerancia a fallos.
Servidores descentralizados	Disponibilidad y Tolerancia a Fallos. Los datos y servicios proporcionados por los servidores son críticos.
Sistemas de eventos	Desacoplamiento entre productores y manejadores de eventos. Escalabilidad (adición de procesos)
Sistemas de datos centralizados	Los aspectos fundamentales son el almacenamiento, representación, procesamiento y gestión de grandes cantidades de datos.
Bases de datos transaccionales/ Almacen (Repository)	El orden de ejecución de los componentes está determinado por el flujo de peticiones de acceso a los datos. Los datos están muy estructurados. Existen bases de datos comerciales que se adaptan a los requisitos del sistema.
Pizarra	Escalabilidad. Adición de consumidores sin afectar a los productores. Modificabilidad. Cambios en los productores y en los consumidores
Máquina Virtual/Intérprete	Diseño de sistemas de computación independientes de plataforma.

Figura 3.2: Clasificación de Estílos [Shaw 1997] y reglas de selección [Bass et al 1998]

En la misma línea que los ABAS[31], aunque con un enfoque algo distinto, el mismo Klein, Bass y Bachmann [Bass et al 2000] definen una serie de *Primitivas de Diseño de los Atributos de Calidad*, cuyo objetivo es proporcionar una caracterización de los atributos de calidad que muestre una relación explícita entre los mismos y los *mecanismos arquitectónicos* más adecuados para conseguirlos. Para ello, estos autores definen unas plantillas en las que se establecen correspondencias sistemáticas entre los mecanismos arquitectónicos y los atributos de calidad: *las plantillas de atributo y las plantillas de mecanismo arquitectónico*.

En los siguientes apartados se explicarán con más detalle las plantillas definidas en [Bass et al 2000] y los ABAS de Klein. Las plantillas, con algunas modificaciones, se utilizarán en el trabajo de tesis ya que permiten realizar una descripción sistemática de los atributos de calidad que debe cumplir la arquitectura a evaluar. Los ABAS no se utilizarán directamente, pero aún así se les dedicará un pequeño apartado, ya que ofrecen una caracterización de los atributos de calidad que es utilizada por la metodología ATAM de evaluación de arquitecturas.

3.3.1 Mecanismos Arquitectónicos y Atributos de Calidad.

Booch [Booch 1994] define un mecanismo arquitectónico como:

"Una estructura mediante la cual los objetos colaboran para proporcionar un comportamiento que satisface un requisito del problema."

A partir de esta definición Bass, Klein y Bachmann construyen un marco de razonamiento en el que es posible relacionar mecanismos arquitectónicos y atributos de calidad [Bachmann et al 2000]. Estos autores conciben el proceso de diseño arquitectónico como un conjunto de decisiones acerca de cómo separar y clasificar las responsabilidades del sistema, de cómo asignar dichas responsabilidades a distintos componentes y de cómo organizar las interacciones entre tales componentes para que cooperen y trabajen de forma coordinada. Examinando estas decisiones es posible determinar el comportamiento que mostrará la arquitectura en relación con la mayoría de los atributos de calidad. Este comportamiento, concluyen los autores, es prácticamente independiente de la funcionalidad, por lo que el diseño de la arquitectura puede realizarse con muy poco conocimiento de la misma. Y ponen el siguiente ejemplo: el rendimiento depende de los procesos, de su asignación a distintos procesadores y de las características de los canales de comunicación entre los mismos, la disponibilidad implica el uso de estrategias de redundancia, la modificabilidad, separación de conceptos. Todas estas decisiones no requieren un conocimiento detallado de la funcionalidad, basta con una descripción de muy alto nivel de la misma.

Cada vez que se toma una decisión arquitectónica que afecta a los atributos de calidad es necesario:

1. Proporcionar un razonamiento que explique por qué tal decisión ayuda a obtener un determinado atributo de calidad. O dicho de otra manera: qué atributos de calidad han causado que dicha decisión se tome.
2. Explicar el efecto que dicha decisión puede causar en la obtención del resto de los atributos de calidad (efectos colaterales).

Estos razonamientos son importantes no sólo porque expliquen las causas de una determinada decisión, sino también porque ayudan a entender las consecuencias de un posible cambio en la misma. Con objeto de organizar estos razonamientos, se proponen las siguientes herramientas:

[31] Todos estos trabajos se inscriben dentro de la iniciativa ATA (*Architecture Trade-off Analysis*) patrocinada por el SEI, dentro de la cual también se incluyen los métodos ATAM y ABD.

- Los escenarios generales,
- Las plantillas de atributo y
- Las plantillas de mecanismos.

3.3.1.1 Atributos de Calidad y Escenarios Generales.

Cada escenario general consiste de [Bachmann et al 2000]:

- Los estímulos a los que la arquitectura debe responder.
- El tipo de componentes involucrados en la respuesta.
- Las métricas utilizadas para caracterizar la respuesta.

Los escenarios generales son completamente abstractos, es decir son independientes de cualquier funcionalidad concreta y, por tanto, pueden ejemplarizarse sobre cualquier sistema en el que tengan relevancia. Por ejemplo, es posible definir un escenario de cambio de plataforma sin ningún conocimiento concreto del sistema, porque todo sistema se ejecuta sobre una plataforma.

Cada vez que se define un escenario general, es necesario describir los mecanismos arquitectónicos que contribuyen a realizarlo. De la misma manera, cada vez que se identifica un nuevo mecanismo, es necesario descubrir a qué escenarios contribuye y asignárselos. En unas ocasiones los mecanismos pueden usarse conjuntamente, en otras definen distintas alternativas para abordar el problema. En cualquier caso, la lista de mecanismos asociada a un determinado escenario y la de escenarios asociados a un mecanismo debe ser tan exhaustiva como sea posible e ir acompañada de los razonamientos pertinentes.

Pero dada la variedad de mecanismos existentes y sus diferentes niveles de abstracción y granularidad, *¿qué mecanismos son relevantes a nivel arquitectónico? ¿y cómo clasificarlos?*. Por ejemplo, la separación de conceptos es un mecanismo muy abstracto que divide el problema en partes coherentes. A un nivel más bajo este mecanismo puede representar la separación entre datos y funciones, la separación de las funciones y la separación de los datos en diferentes niveles y jerarquías. Incrementando el nivel de detalle se incrementa la precisión del análisis. Sin embargo, puesto que el objetivo es determinar primitivas de diseño de aplicación general, es necesario utilizar el nivel de abstracción más alto que siga proporcionando algún significado que sirva de base para razonamientos posteriores. A medida que los escenarios generales se vayan ejemplarizando para arquitecturas más concretas, de un dominio o de un sistema, podrá refinarse su significado y los mecanismos hacerse más concretos.

Establecidas las correspondencias entre escenarios y mecanismos, queda aún un problema por resolver, *¿cómo afecta la elección de un mecanismo para un cierto escenario al resto de los escenarios?* Cada mecanismo debe ir acompañado de un razonamiento que explique su influencia en *cada uno* de los atributos de calidad y dado un conjunto de escenarios y mecanismos deben describirse las interacciones entre los mismos. La información asociada a escenarios y mecanismos es diversa y abundante. Para organizarla se proponen las plantillas de atributo y de mecanismo que se describen a continuación.

3.3.1.2 Plantillas de Atributo.

Cada plantilla de atributo[32] presenta los aspectos más importantes del atributo al que concierne y los mecanismos que contribuyen a alcanzarlos. Cada plantilla consta de las siguientes entradas:

- **Conceptos del atributo.** Descripción del atributo, incluyendo estímulos relevantes y métricas de respuesta típicas.

[32] En [Bass et al 2000] se denomina a estas plantillas *Attribute Story Templates*

- **Escenarios generales del atributo.** Descripción del mayor número posible de escenarios independientes de sistema. Estos escenarios generales, de muy alto nivel y completamente abstractos, pueden aplicarse a cualquier sistema.
- **Estrategias para abordar el atributo.** Descripción de las estrategias básicas que contribuyen a realizar los escenarios generales. Esta descripción establece las bases de la siguiente entradas de la tabla.
- **Enumeración de los mecanismos arquitectónicos para realizar el atributo.** En esta entrada se enumeran y describen todos los mecanismos que ayuden a realizar el atributo. Cada mecanismo debe tener su propia descripción.

La figura 3.4 resume las plantillas que se definen para la modificabilidad.

3.3.1.3 Plantillas de Mecanismo.

Una plantilla de mecanismo[33] describe la relación de un mecanismo arquitectónico con la modificabilidad, el rendimiento, la disponibilidad, la seguridad y la usabilidad. Cada plantilla consta de las siguientes entradas:

- **Descripción del mecanismo.** Descripción del mecanismo y presentación de los objetivos del mismo, es decir de los atributos de calidad pretende optimizar.
- **Propósito y análisis del mecanismo.** Descripción de las estrategias que utiliza el mecanismo para alcanzar sus objetivos y de los escenarios que soporta. En esta sección se aborda el atributo al que está principalmente asociado el atributo. El resto de las entradas describen sus efectos laterales en otros atributos.
- **Escenarios generales de la modificabilidad.** Esta sección interpreta los escenarios generales de la modificabilidad desde el punto de vista del mecanismo.
- **Escenarios generales del rendimiento.** Esta sección interpreta los escenarios generales del rendimiento desde el punto de vista del mecanismo.
- **Escenarios generales de la disponibilidad/fiabilidad.** Esta sección interpreta los escenarios generales de la disponibilidad/fiabilidad desde el punto de vista del mecanismo.
- **Escenarios generales de la seguridad.** Esta sección interpreta los escenarios generales de la seguridad desde el punto de vista del mecanismo.
- **Escenarios generales de la usabilidad.** Esta sección interpreta los escenarios generales de la usabilidad desde el punto de vista del mecanismo.

La clasificación de atributos que implícitamente realizan las plantillas es discutible. Sin embargo, éste no es el aspecto esencial de las mismas, sino la posibilidad de capturar y organizar los efectos que cada mecanismo tiene sobre los diferentes atributos de calidad. Las plantillas siguen siendo útiles aunque se aplique otra taxonomía. A modo de ejemplo, las figura 3.5 resume la plantilla que se define para el mecanismo *"distribuidor de datos"*.

[33] En [Bass et al 2000] se denomina a estas plantillas *Mechanism Story Templates*

Modificabilidad

Conceptos de la Modificabilidad

La modificabilidad de un sistema es su capacidad para soportar cambios una vez que ha sido construido.

Los estímulos son los cambios requeridos y la respuesta el número de modificaciones, la dificultad de las mismas y el tiempo que ocupan.

Los estímulos pueden reflejar cambios en la funcionalidad, la plataforma y el entorno operativo y pueden definirse en tiempo de compilación, carga o ejecución.

Escenarios generales de la modificabilidad.

- Debe modificarse la funcionalidad del sistema.
  El cambio puede consistir en añadir o eliminar funcionalidad o en modificar la existente.
- Debe modificarse la plataforma en la que se ejecuta sistema.
  Cambios en el hardware, en el sistema operativo o en las herramientas y COTS sobre los que se apoya.
- Debe modificarse el entorno operativo del sistema.
  El sistema debe trabajar con otros sistemas no considerados en principio, debe reaccionar a cambios en las características de sus usuarios (personas u otros sistemas), debe operar en solitario o debe descubrir dinámicamente los servicios disponibles.
- Debe modificarse el comportamiento del sistema respecto de algún otro atributo.
  Modificación del rendimiento, disponibilidad, usabilidad o seguridad requeridas.

  Forma general del escenario:
  Tiempo de modificación: Compilación | Carga | Ejecución.
  Tipo de modificación: Adición | Eliminación | Modificación.
  Objeto de la modificación: Funcionalidad | Plataforma | Entorno | Atributo.

Estrategias.

En tiempo de compilación:

- **Indirección**: Inserción de módulos intermedios de desacoplo (mediadores, adaptadores, etc) que permitan la variación de una determinada característica, mediante la reducción del conocimiento e interacciones mutuas entre componentes.
- **Separación de Conceptos**: Encapsular de forma separada datos y funciones relacionadas con ciertos aspectos del problema. De esta forma la modificación de alguno de estos aspectos no supone la modificación del resto.
- **Codificación de la funcionalidad en intérpretes**: Codificando parte de la funcionalidad en datos y proporcionando un mecanismo para interpretar dichos datos se consigue simplificar las modificaciones que afectan a los parámetros de los mismos.

En tiempo de ejecución:

- **Reconfiguración dinámica:** Detección de los recursos o funciones dinámicamente disponibles y acceso a los mismos.

Mecanismos:

- **Distribuidor de datos:** mecanismo intermediario que direcciona datos entre productores y consumidores. Requiere del uso de otro mecanismo que permita identificar a los mismos.
- **Almacén de datos:** Almacena datos para su posterior uso.
- **Máquina virtual:** Mecanismo intermediario entre usuarios y suministradores de una cierta funcionalidad.
- Intérprete: Mecanismo que incluye la codificación de funciones en parámetros y descripciones abstractas que permiten su modificación. Muy relacionado con el anterior.
- **Cliente/Servidor:** Proporciona una colección de servicios que pueden ser usados por otros componentes según las reglas definidas en un protocolo fijo.
- …

Figura 3.4: Plantilla de la Modificabilidad

Distribuidor de Datos

Descripción del mecanismo.

Este mecanismo (también llamado mediador, enrutador y bus de datos) implementa una estrategia de indirección.

Consiste en insertar un mediador entre productores y consumidores de datos. El productor debe saber que los datos deben ser enviados al mediador y el consumidor debe saber que debe recogerlos del mismo. Este mecanismo debe utilizarse con otro que establezca como unos y otros pueden reconocerse.

En ocasiones, el mediador puede realizar conversiones de datos.

Propósito y análisis del mecanismo.

El propósito de este mecanismo es desacoplar productores y consumidores, simplificando la adición de nuevos productores y consumidores.
Escenarios generales:

- Estímulo: Adición | Eliminación | Modificación de Productores | Consumidores de datos de un determinado tipo.
  Respuestas: Los nuevos productores | consumidores deben ser registrados (debe proporcionarse un mecanismo para ello).
  Si el productor eliminado era el único que producía el dato, los consumidores ya no lo recibirán.
- Estímulo: Producción por parte de un productor de un nuevo tipo de dato.
  Respuestas: El resto de los productores y los consumidores ya existentes no se ven afectados.
  El mediador debe modificarse para tener en cuenta el nuevo dato.
- Estímulo: Producción por parte de un productor de un nuevo tipo de dato.
  Respuestas: El resto de los productores y los consumidores ya existentes no se ven afectados.
  El mediador debe modificarse para tener en cuenta el nuevo dato.
- Estímulo: Se añade una nueva conversión de datos al mediador.
  Respuestas: El mediador debe modificarse para tener en cuenta el nuevo dato.
- Estímulo: Se modifica el mediador.
  Respuestas: Depende del tipo de modificación.
  La adición de un nuevo tipo de datos no afecta a los tipos existentes.
  La modificación de la interfaz puede afectar, según su alcance, a todos o a una parte de los productores y consumidores.
- …

Escenarios generales del rendimiento.

- ¿Cuántos datos pasan a través del mediador?
- ¿Cuántos mensajes recibe el mediador simultáneamente?
- ¿Cuántos productores | consumidores están conectados al mediador?
- ¿Cuál es el número máximo de consumidores de un dato determinado?
- ¿Qué retardo introduce el mediador?
- ¿Cuál es el régimen de llegada de los mensajes (periódico, esporádico, estocástico)?
- ¿Se ejecutan los productores | consumidores en un mismo procesador?
- Si se ejecutan en diferentes procesadores ¿Cuánto tiempo tardan en entregarse los mensajes? ¿Qué recursos se utilizan?
- ¿Es necesario sincronizar a productores y consumidores? Si lo es ¿Cómo se sincronizan?
- …

Escenarios generales de la seguridad.

- ¿Puede un agente externo escuchar las comunicaciones entre el mediador y los productores | consumidores?
- ¿Es posible que un proceso no autorizado acceda al mediador?
- …

Escenarios generales de la disponibilidad/fiabilidad.

- ¿Cuál es la probabilidad de que un dato producido en un productor no llegue al consumidor? ¿Y el impacto?
- ¿Cuál es la probabilidad de que un dato no se produzca a tiempo? ¿Y su impacto?
- …

Figura 3.5: Plantilla del distribuidor de datos

3.3.2 Los ABAS: Estilos arquitectónicos basados en atributo.

3.3.2.1 Estructura de los ABAS

Klein en [Klein 1999a] define un ABAS como una tupla de tres elementos:

1. La topología de los componentes y una descripción de los patrones de interacción entre los mismos, expresados tanto en términos de flujos de datos como de control.
2. Un modelo específico de atributo que proporcione un marco formal de razonamiento acerca del comportamiento del estilo en relación con el atributo caracterizado por el modelo. Estos modelos permitirán determinar con precisión hasta que punto los componentes y patrones de interacción definidos en el estilo son apropiados para conseguir el atributo de calidad.
3. Los resultados de aplicar el modelo sobre el patrón.

Por ejemplo, el ABAS *pipe_and_filter/Rendimiento* incluiría:

1. Una descripción del estilo *pipe and filter*, incluyendo la definición de los componentes (tuberías y filtros) y de sus patrones de interacción (flujo de datos y control).
2. Un modelo de colas aplicado al estilo, incluyendo las reglas de instanciación del modelo.
3. Los resultados de aplicar el modelo, junto con los razonamientos que se estimen pertinentes acerca de la suposiciones o hipótesis que se hayan asumido para su aplicación y resolución.

La estructura de los ABAS refleja con claridad su objetivo: asociar con cada estilo un marco de razonamiento, formal si es posible, que permita determinar su comportamiento respecto a los atributos de calidad. Las abstracciones arquitectónicoes deben relacionarse con las abstracciones impuestas por el modelo. Por ejemplo [Klein et al 1999a], si el objetivo es razonar acerca de la fiabilidad, las características de la arquitectura que tengan relación con la misma (componentes replicados, redundancia) deben traducirse a modelos formales, tales como modelos de Markov. Siguiendo este esquema, los comportamientos que revelan los modelos pueden ser comparados con los comportamientos deseados.

3.3.2.2 Caracterización de atributos.

Como se ha explicado, la aplicación de modelos formales requiere establecer una correspondencia entre las abstracciones arquitectónicoes y las abstracciones del modelo. Esta correspondencia debe establecerse también entre los requisitos de la arquitectura y los parámetros de entrada y salida del modelo. Para ello es necesario relacionar los parámetros del modelo con las estructuras arquitectónicoes y con las entradas o *estímulos* que previsiblemente vaya a admitir el sistema. [Klein et al 1999a] distingue entre medidas de los atributos de calidad (*quality attribute measures*) y parámetros de los atributos de calidad *(quality attribute parameters*). Aunque los nombres parecen referirse a lo mismo, en realidad definen conceptos muy diferentes:

- *Las medidas de los atributos de calidad* son los requisitos que deben cumplir los atributos de calidad, expresados en términos mensurables o al menos observables, y de forma que se correspondan o al menos puedan relacionarse con los parámetros de entrada y salida de los modelos que caracterizan los atributos.

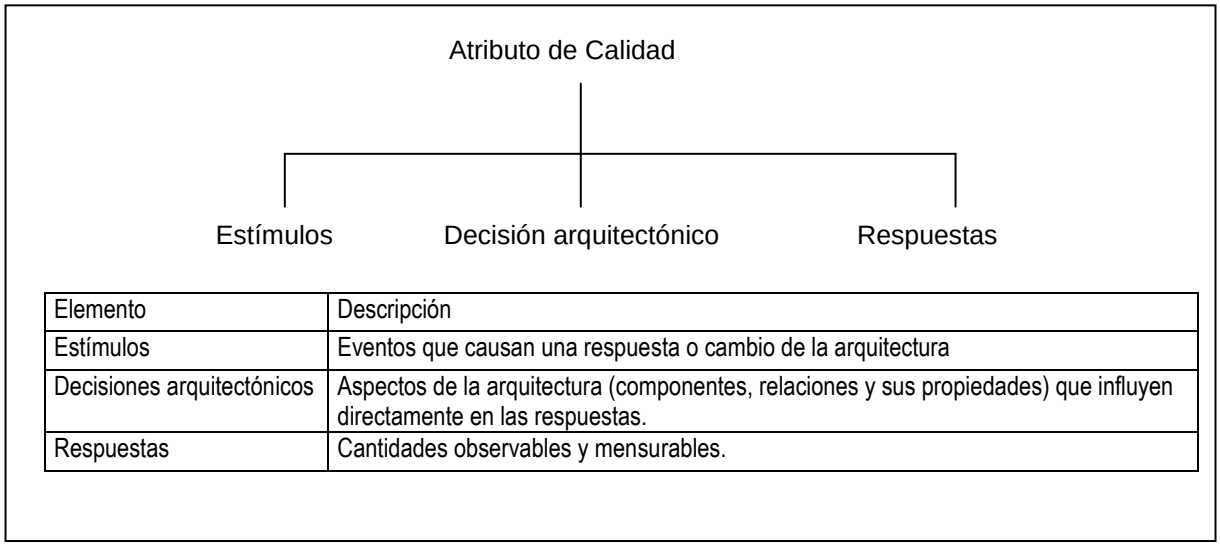
- *Los parámetros de los atributos de calidad* son las propiedades de la arquitectura (componentes y patrones de interacción) que pueden parametrizarse para alcanzar el comportamiento deseado.

De acuerdo con lo dicho arriba, [Klein et al 1999b] organiza los conceptos presentados en el párrafo anterior clasificando la información relevante para caracterizar el atributo en tres categorías: estímulos externos, decisiones arquitectónicos y respuestas (figura 3.6).

- *Los estímulos externos* son los eventos que causan una respuesta o cambio en la arquitectura, expresados en términos mensurables u observables. Se corresponden con las medidas de los atributos de calidad definidas en [Klein et al 1999a].
- *Las decisiones arquitectónicos* son aquellos aspectos de la arquitectura (sus componentes y conectores y las propiedades de los mismos) que tienen una influencia directa en las *respuestas* que ofrece la arquitectura. Se corresponden con los parámetros de los atributos de calidad definidos en [Klein et al 1999a].
- *Las respuestas* describen el comportamiento de la arquitectura ante los estímulos externos. Al igual que los estímulos deben estar expresados en términos mensurables u observables y deben corresponderse con los resultados del modelo que caracteriza al atributo.

Obsérvese la correspondencia con los escenarios generales descritos en el apartado anterior.

Las figuras 3.7 a 3.9 muestran está información para el rendimiento, la modificabilidad y la disponibilidad. Aunque no se incluyen aquí, también están definidos los árboles de caracterización correspondientes a la seguridad (*security*) y a la interoperabilidad. La figura 3.10 resume los ABAS definidos por Klein [Klein et al 1999b] hasta la fecha.



Elemento	Descripción
Estímulos	Eventos que causan una respuesta o cambio de la arquitectura
Decisiones arquitectónicos	Aspectos de la arquitectura (componentes, relaciones y sus propiedades) que influyen directamente en las respuestas.
Respuestas	Cantidades observables y mensurables.

Figura 3.6: ABAS: Caracterización de atributos.

3.3.2.3 Descripción de un ABAS

La descripción de un ABAS consta de cuatro partes:

- Descripción del problema que trata de resolverse con la aplicación del estilo, incluyendo los atributos de interés, sus requisitos y restricciones relevantes y su contexto de uso.
- Medidas de los atributos de calidad, incluyendo los estímulos de entrada y las respuestas generadas por el ABAS.
- Descripción del estilo, incluyendo su topología (componentes y patrones de conexión) y sus restricciones.
- Análisis o descripción de cómo los modelos que caracterizan a los atributos están formalmente relacionados con el estilo y conclusiones acerca del comportamiento del mismo en función de dichos atributos.

3.3.2.4 Mecanismos Arquitectónicos y ABAS

Existe una relación evidente entre los escenarios generales y los mecanismos arquitectónicos presentados en el apartado anterior y los ABAS descritos en éste. Sin embargo, los conceptos de mecanismo arquitectónico y ABAS no son completamente equivalentes.

Los ABAS asocian estilos arquitectónicos más o menos complejos con un marco de análisis específico de atributo. Los mecanismos arquitectónicos representan primitivas de diseño de los estilos arquitectónicos. Aunque es posible que un ABAS sólo incorpore uno de esos mecanismos (en cuyo caso ambos conceptos son equivalentes), lo más usual es que presente una combinación de varios. Los mecanismos son de aplicación general, mientras que los ABAS tienen un ámbito de uso más concreto. Finalmente, aunque sólo se hay descrito un pequeño conjunto de ABAS, su número es potencialmente mucho más grande que el de mecanismos, ya que existen muchas maneras en que los mecanismos pueden combinarse para formar ABAS.

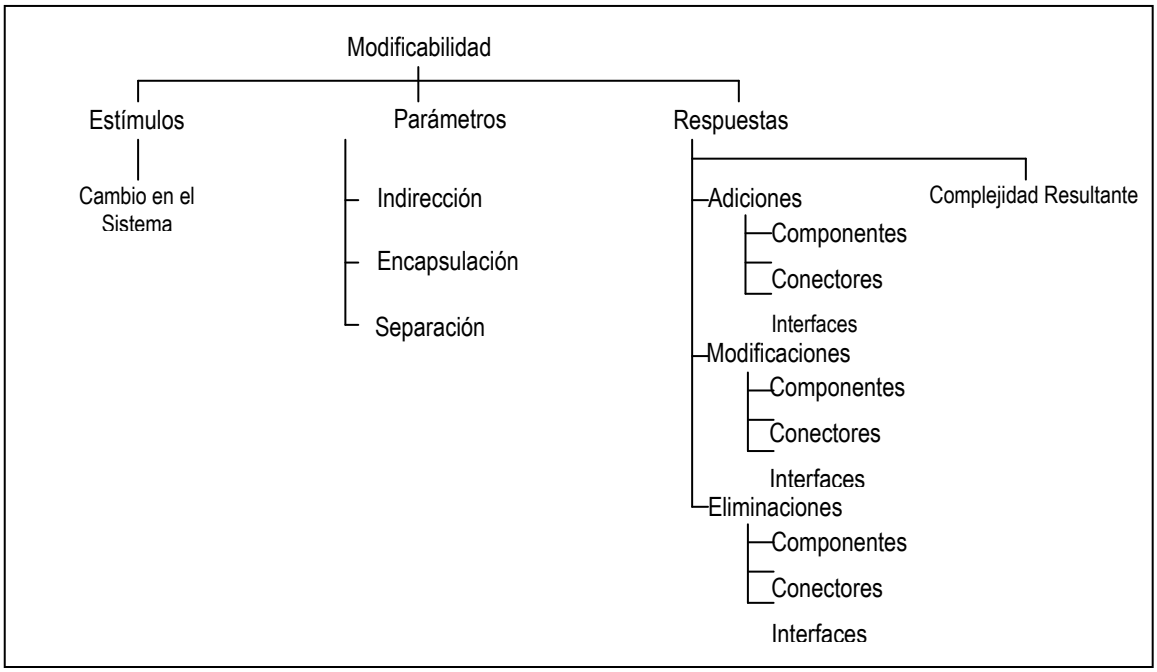


Figura 3.7: Caracterización de la Modificabilidad

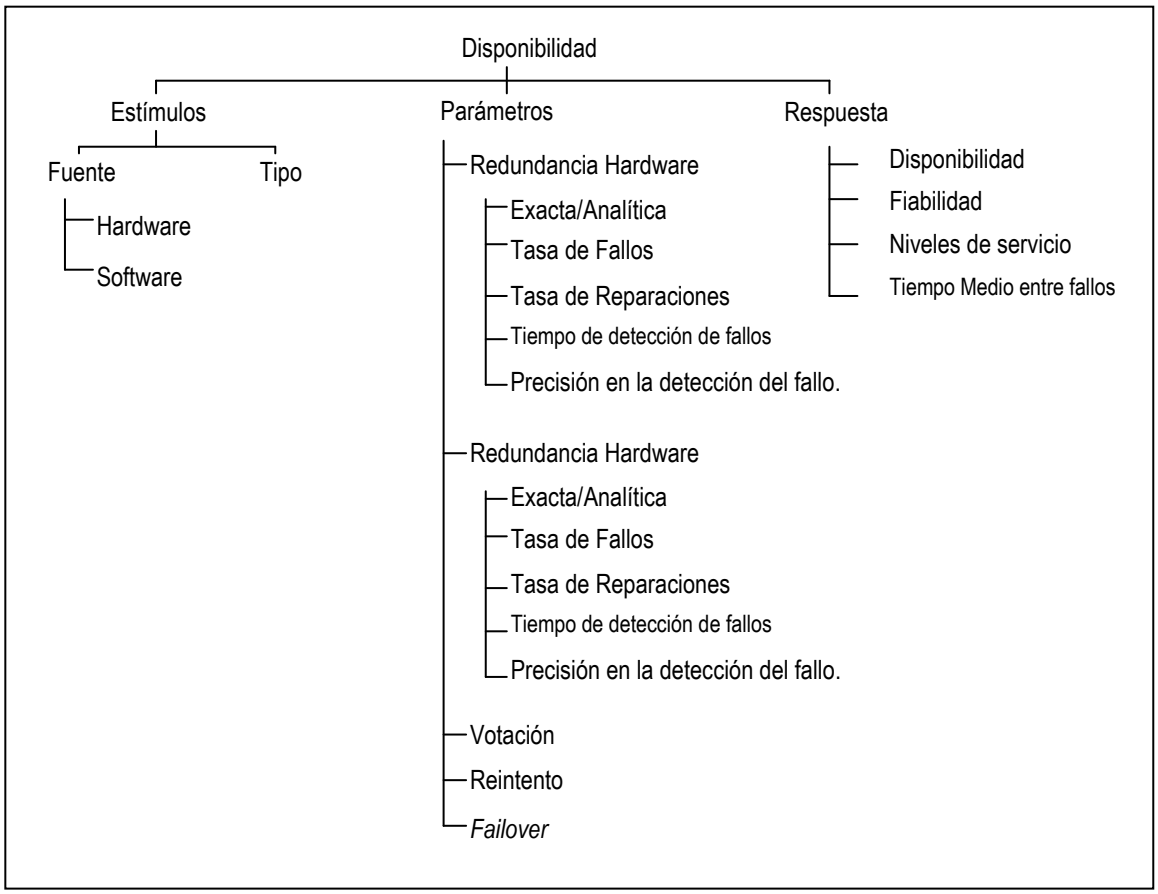


Figura 3.8: Caracterización de la Disponibilidad

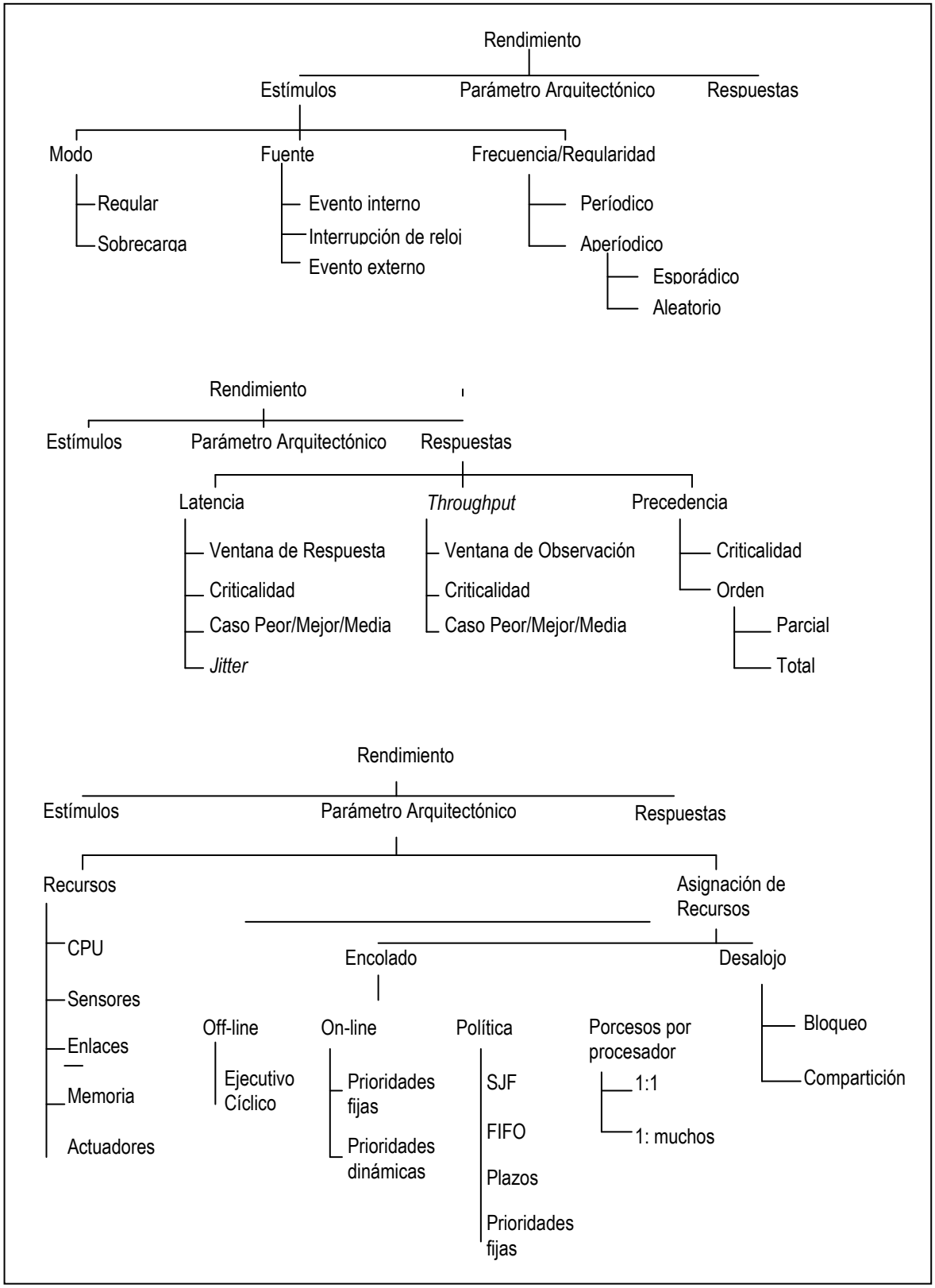


Figura 3.9: Caracterización del Rendimiento

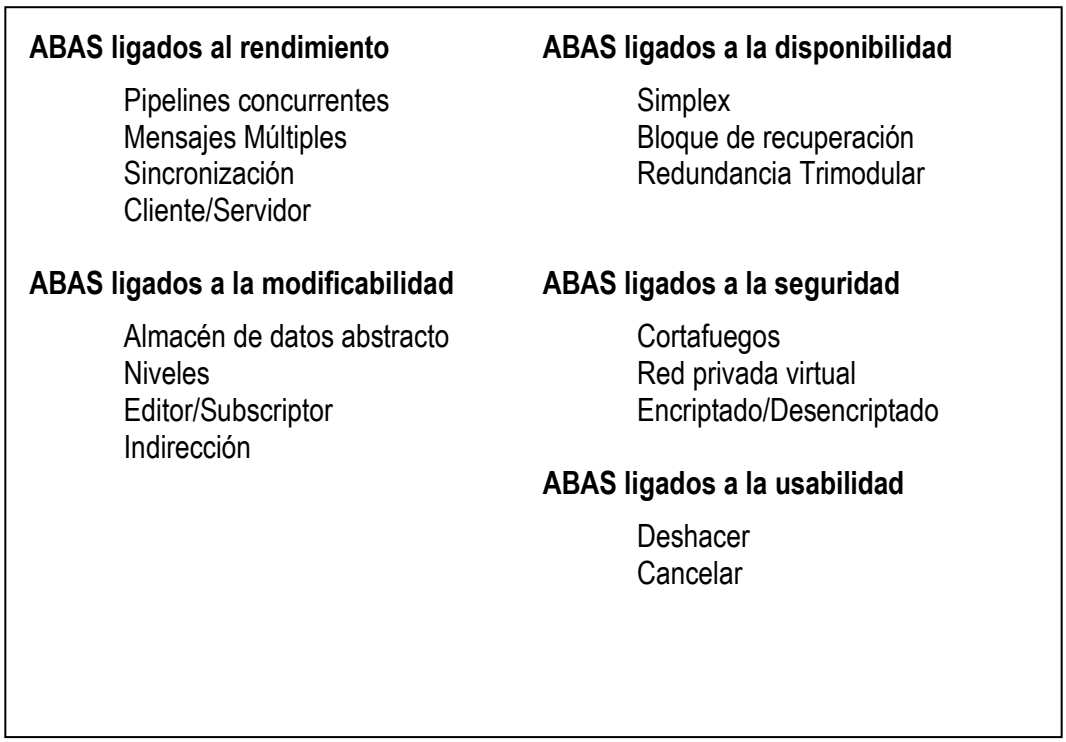


Figura 3.10: Colección de ABAS [Klein et al 1999]

4 Metodologías de Evaluación y Diseño de la Arquitectura: el ATAM y el ABD.

4.1 Introducción.

Existe una gran variedad de metodologías de análisis y diseño, desde las basadas en los principios de la programación estructurada [Yourdon 79] hasta las más actuales orientadas a objetos, entre las que destacan las definidas por Booch [Booch 94], Jacobson [Jacobson et al 92], Rumbaugh (OMT) [Rumbaugh 1996], Shlaer-Mellor [Shlaer et al 1988], Catalysis [D'Souza et al 1999], y más recientemente y orientadas al diseño de sistemas empotrados las definidas por Gomaa, método COMET [Gomma 2000], y Douglas, método ROPES [Douglas 1999]. Estas dos últimas se basan en las reglas y principios de diseño definidas por las anteriores (especialmente en [Jacobson et al 1992]) e incorporan nuevos conceptos orientados al diseño de sistemas reactivos de tiempo real.

Simplificando mucho puede decirse que el enfoque de las metodologías de diseño orientadas a objeto se basa en la identificación de los objetos más representativos del dominio considerado, su agrupación en diferentes clases y la definición de los mensajes o servicios intercambiados entre los mismos. La identificación de los objetos más representativos y sus patrones de interacción requiere de una fase previa de análisis en la que se estudian las características del dominio del problema. Para la realización de este análisis, la utilización de los casos de uso propuestos por Jacobson [Jacobson et al 1992] se ha ido imponiendo con fuerza, y hoy en día se hace difícil hablar análisis orientado a objetos prescindiendo de los mismos. Los casos de uso describen las interacciones de los usuarios finales con el sistema y se utilizan, por tanto, para descubrir su funcionalidad básica. A partir de los casos de uso se construye un modelo de análisis que se va elaborando progresivamente hasta obtener modelos de diseño cada vez más refinados que en última instancia dan lugar a un modelo implementable en un lenguaje de programación. Si las metodologías están soportadas por herramientas lo suficientemente potentes, este enfoque elaborativo puede sustituirse por un enfoque traductivo, en el que es posible generar código a partir de modelos más o menos refinados. Para la definición de los modelos de diseño se aportan heurísticos y criterios que permiten identificar clases y objetos, determinar sus dependencias, definir relaciones de herencia y composición y seleccionar los patrones de interacción que mejor se adapten a las *calidades* exigibles al sistema. Sea cual sea el enfoque, el proceso está dirigido por casos de uso y el diseño sigue un desarrollo iterativo e incremental en el que los objetivos de cada ciclo se establecen en función de los riesgos a resolver y de los casos de uso considerados. En resumen, metodologías dirigidas por riesgos y casos de uso que utilizan los conceptos del diseño orientado a objetos.

Aunque todas estas metodologías aportan enfoques útiles y cubren con mejor o peor fortuna todas las fases del proceso de desarrollo de un sistema software, presentan un grave inconveniente para su empleo en este trabajo de tesis: en general, consideran el diseño de un sistema concreto y no el de una familia de sistemas. En los sistemas concretos es relativamente fácil seleccionar un conjunto reducido, pero lo suficientemente representativo, de casos de uso. Los requisitos pueden especificarse con concreción y las líneas de evolución del sistema pueden acotarse con un alto grado de confianza. Sin embargo, cuando se aborda el diseño de una arquitectura de referencia para una familia de sistemas o para una línea de producto las premisas anteriores dejan de ser ciertas. Los casos de uso pueden variar mucho entre sistemas. Algunos casos de uso esenciales en un sistema pueden no tener relevancia en otros. Un mismo caso de uso puede desarrollarse de formas diferentes dependiendo del sistema. Lo mismo ocurre con los requisitos. A menudo, sólo se conocen de forma general y su especificación varía entre sistemas. Sus líneas de evolución futura son, en el mejor de los casos, el sumatorio de las líneas de evolución de los diferentes sistemas incluidos en la familia. El diseño de una arquitectura de referencia requiere de una metodología que tenga explícitamente en cuenta la falta de concreción de los requisitos y maneje la variabilidad entre productos o sistemas.

Finalmente, como ya se ha dicho, las decisiones arquitectónicas ocurren en las primeras etapas del diseño y afectan a la estructura misma del sistema, condicionando todo el posterior proceso de desarrollo. Es evidente que las arquitecturas deben evaluarse, confrontándolas con los atributos de calidad que dependen de ellas. Pero *¿cómo medir la calidad de una arquitectura?* Existe una gran cantidad de metodologías que miden la complejidad y la calidad del software utilizando diversas métricas, tales como los puntos de función [Marciniak 1994], la medida de la complejidad de Halstead [Halstead 1977], la complejidad ciclomática [McCabe 1989] y el índice de mantenibilidad [Oman 1994]. Sin embargo, ninguna de ellas está específicamente pensada para evaluar arquitecturas software y muchas de ellas se aplican cuando el código ya está escrito. Aunque algunas son bastante utilizadas en la industria, especialmente los puntos de función, no existe un consenso definitivo sobre su utilidad [Jones 1994] y su utilización requiere tiempo, entrenamiento y por lo general herramientas. Por ello, aunque estas técnicas suelen incluirse en todos los libros, manuales y cursos sobre ingeniería software, ninguna de ellas se ha utilizado en este trabajo de tesis.

Más específicamente, en el campo del diseño y evaluación de arquitecturas cabe destacar los trabajos de la Universidad del Sur de California (USC), principalmente a través de los trabajos de Bohem, Eyged y Medvidovic y sobre todo del SEI a través de los métodos de evaluación de arquitecturas SAAM [Kazman et al 1994] y ATAM [Kazman et al 2000] y otros trabajos incluidos en la iniciativa ATA [SEI-ata 2002] (*Architecture Tradeoff Analysis*)[34].

El objetivo de este capítulo es presentar las metodologías de evaluación, ATAM, y de diseño, el ABD, que se han empleado para evaluar y re-diseñar la arquitectura de referencia para sistemas de teleoperación descrita por Álvarez en [Álvarez 1997], así como la forma en que se han aplicado. El último apartado de este capítulo ofrece una guía para la lectura y comprensión del resto de la tesis, explicando los artefactos que se han generado y las relaciones entre los mismos. Adicionalmente, se describe de forma breve SAAM. La inclusión de SAAM obedece a razones históricas: es el antecedente de ATAM y ofrece alguna de las claves que han llevado a su definición.

[34] Obviamente los trabajos sobre arquitecturas software no se acaban en la USC y el SEI. Sin embargo, constituyen con diferencia las primeras fuentes de información de este trabajo de tesis.

4.2 El método SAAM

4.2.1 Introducción. Características generales.

SAAM ([Kazman et al 1994, 1996], [Bass et al 1998]) son las siglas de *Software Architecture Analysis Method* (Método de Análisis de Arquitectura Software). El SAAM es un método de evaluación de arquitecturas software desarrollado entre principios y mediados de los 90 por el SEI de la CMU que se ha utilizado tanto para validar arquitecturas en desarrollo como para evaluar sistemas software ya implementados. Pueden encontrarse ejemplos de su aplicación en [Kazman et al 1994, 1996], [Abowd et al 1996], [Eickelmann et al 1996] y [Bass et al 1998]. Aunque el SAAM no se ha empleado en este trabajo de tesis, merece la pena dedicarle un pequeño apartado por dos razones. Primera, SAAM es el predecesor de ATAM. Segunda, SAAM es un buen método para evaluar arquitecturas respecto de la modificabilidad.

Las principales características del método SAAM son las siguientes:

1. SAAM no evalúa la arquitectura respecto de atributos de calidad abstractos, sino respecto de requisitos concretos. La evaluación se realiza en un contexto muy específico, asumiendo que los atributos de calidad no existen como conceptos aislados, sino que tienen sentido dentro de un contexto.
2. El método debe consumir pocos recursos y debe poder aplicarse antes de que el sistema este implementado, aun cuando no se conozcan todos los detalles del sistema final.
3. La aplicación del método requiere de una descripción clara, pero en general no muy detallada de la arquitectura.
4. SAAM es un método conducido por escenarios. Si la calidad de la arquitectura depende del contexto en el que se evalúa, es necesario definir de alguna manera ese contexto.

4.2.2 Descripción del método

El método SAAM se desarrolla en seis etapas (véase figura 4.1): Desarrollo de escenarios, descripción de las arquitecturas candidatas, clasificación de escenarios, realización de escenarios, interacción entre escenarios y evaluación global.

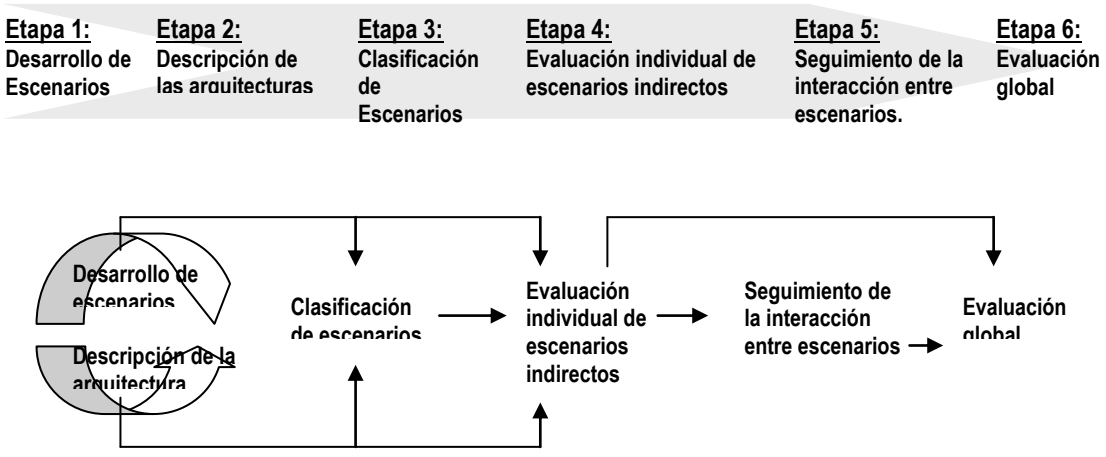


Figura 4.1: SAAM: Actividades del método y sus dependencias

1.- Desarrollo de escenarios.

Los escenarios deben ilustrar el tipo de actividades que deben ser soportadas por el sistema y los cambios que previsiblemente puedan realizarse en el mismo. Los escenarios deben considerar todos los usos y usuarios importantes del sistema, así como los atributos de calidad que el sistema debe satisfacer. La obtención de escenarios requiere de cierta experiencia y de buena voluntad entre las partes implicadas para alcanzar un consenso que evite una explosión de escenarios.

2.- Descripción de la arquitectura.

El SAAM no necesita una descripción exhaustiva de todas las estructuras del sistema. La importancia de una estructura viene dada por la importancia del atributo en el que influye. Cuanto más importante es un atributo más exhaustiva debe ser la representación de las estructuras implicadas. SAAM pretende ser un método de bajo coste, aplicable en las primeras etapas de diseño y sus resultados son indicaciones cualitativas. Aún así, la experiencia demuestra que en la mayoría de los casos no existen descripciones apropiadas para ejecutar el método y han debido realizarse sobre la marcha. Por esta razón los inventores del método han decidido incluir este paso en el mismo[35].

3.- Clasificación de escenarios. Escenarios directos e indirectos.

Puesto que los escenarios representan casos de uso concretos, su número puede ser muy grande. Sin embargo, los escenarios pueden clasificarse en grupos, de forma que un solo escenario sirva para representar a todos los escenarios del grupo al que pertenece. SAAM contabiliza el número de escenarios que causan cambios en un determinado componente de la arquitectura. Si un componente sólo se ve afectado por un determinado grupo de escenarios significa que la descomposición funcional del sistema es buena. Si, por el contrario, un grupo de escenarios similares afecta a muchos componentes o un mismo componente es afectado por diferentes grupos de escenarios, significa que el sistema está mal estructurado, ya que los efectos de los escenarios se propagan por toda su estructura.

Se dice que una arquitectura puede *soportar directamente un escenario* si no es necesario modificar el sistema para que lo realice. Si un escenario no se soporta directamente, es decir no puede ser realizado con la definición actual de la arquitectura, se dice que tal *escenario es indirecto*. Para que un escenario indirecto se convierta en directo, será necesario definir una serie de cambios en los componentes del sistema y/o en sus relaciones.

En principio, una arquitectura que soporte directamente más escenarios es más apropiada que otra que requiera modificaciones y recibirá una mejor puntuación al ejecutar el SAAM. Sin embargo, el número de escenarios soportados directamente no es un indicador completamente fiable de la bondad de la arquitectura. Una arquitectura puede soportar un escenario directa, pero pobremente. Otra, puede no soportarlo, pero el alcance de los cambios necesarios para que lo haga de forma completamente satisfactoria puede ser pequeño y abordable. Por ello, al ejecutar el método, es necesario describir el grado de modificación necesario para soportar un escenario indirecto.

4.- Evaluación individual de escenarios indirectos

[35] En la mayoría de las ocasiones existe documentación sobre los requisitos y estructuras del sistema, pero en un formato no apropiado. Gráficos donde la semántica de conectores y componentes no está definida. Descripciones no entendibles por personal no especializado, etc.

Una vez determinados los escenarios indirectos, es necesario describir el efecto que tienen sobre la arquitectura los cambios necesarios para conseguir que el escenario se soporte directamente. Para ello se construye una tabla en la que se listan:

- Los escenarios indirectos.
- Las modificaciones necesarias para que la arquitectura lo soporte directamente.
- Una estimación del coste de cada una de las modificaciones.

Una modificación en la arquitectura supone, o bien el cambio de los componentes y conectores ya existentes, o bien la introducción de otros nuevos. Puede suponer incluso un cambio en alguna de las especificaciones. El alcance de los cambios puede ser muy variado y presentar grados de dificultad muy distintos. Por ello, es necesario ponderar el grado de dificultad que supone cada cambio.

5.- Seguimiento de la interacción entre escenarios

La mera enumeración y descripción de los escenarios indirectos no es suficiente para evaluar la arquitectura. Una arquitectura recibiría tanta mejor puntuación cuanto menos modular fuera. El estudio de la interacción entre escenarios evita este extremo.

Cuando dos o más escenarios indirectos implican cambios en un mismo componente se dice que interaccionan en dicho componente.

La interacción de escenarios pone de manifiesto la distribución de la funcionalidad en la arquitectura. Los componentes de la arquitectura con un alto grado de interacción de escenarios revelan una pobre separación de conceptos en los mismos. El grado de interacción de escenarios está relacionado con la complejidad estructural, la cohesión y el acoplamiento entre los componentes de la arquitectura. Un alto grado de interacción entre escenarios diferentes se corresponde con una baja cohesión e indica una gran complejidad estructural. Por el contrario, un alto grado de interacción entre escenarios semánticamente relacionados sugiere una alta cohesión del componente con el que interaccionan.

6.- Evaluación global. Ponderación de los escenarios.

Aunque el SAAM no proporcione resultados cuantitativos, proporciona el siguiente mecanismo para comparar dos arquitecturas[36].

Una vez definidos los escenarios y descubiertas sus interacciones, se pondera la importancia relativa de dichos escenarios e interacciones. Para ello se les asignan pesos.

El propósito de los pesos es resolver situaciones en las que una de las arquitecturas candidatas puntúa mejor en un conjunto de escenarios y otra en otro conjunto. La asignación de pesos es un proceso subjetivo, sujeto a la negociación y al consenso entre todas las partes implicadas. El SAAM no define las métricas, son los usuarios del método los que deben decidir qué escenarios e interacciones son más importantes, y darles un peso en consecuencia.

4.2.3 Conclusiones y críticas al método

SAAM es uno de los primeros intentos de sistematizar el proceso de evaluación de las arquitecturas software que cuenta con ejemplos de aplicación prácticos. El SAAM es además el

[36] Comparación que se hace considerando un mismo sistema, un mismo conjunto de requisitos y por tanto un mismo conjunto de escenarios. No conviene olvidar que SAAM asume un contexto específico para la evaluación.

punto de partida de ATAM. Sin embargo, aunque el análisis que proporciona SAAM es útil para detectar enfoques incorrectos, es claramente insuficiente para guiar un proceso de diseño. Entre las limitaciones de SAAM, podrían citarse las siguientes:

- A pesar de basarse únicamente en los escenarios para caracterizar a los atributos de calidad, no establece ninguna metodología para la obtención de los mismos, salvo el criterio de las partes implicadas.
- Necesita de una descripción detallada de los atributos de calidad, pero no define, ni siquiera sugiere, una forma de caracterizarlos.
- No establece ninguna asociación formal entre estilos arquitectónicos y atributos de calidad, fiándolo todo a la experiencia y los conocimientos de los evaluadores. Aunque en [Bass et al 2000] se describen una serie de *operaciones de diseño unitarias* para la obtención de los atributos de calidad, no se explica convenientemente como utilizarlas durante la ejecución de SAAM.
- Está tan enfocado a la medida de la modificabilidad del sistema, que algunos autores [Boehm 1996] no lo consideran aplicable para evaluar el resto de los atributos de la arquitectura.
- Aunque el método reconoce la importancia de que las partes implicadas lleguen a compromisos respecto de los atributos del sistema, es extremadamente optimista en este aspecto. Supone que las partes implicadas renunciarán a parte de sus exigencias sin una demostración seria de lo que implican sus requisitos en el coste del sistema (coste no sólo monetario, sino en términos de lo que se ven perjudicados otros atributos de calidad)[37].

4.3 El método ATAM

4.3.1 Introducción. Características generales.

ATAM son las siglas de *Architecture Tradeoff Analysis Method*. Como SAAM, ATAM ha sido desarrollado por el SEI de la CMU, si bien ATAM es un poco posterior y constituye una evolución del método SAAM para solucionar sus carencias. La versión más reciente del método ATAM está descrita en [Kazman et al 2000], pudiendo consultarse ejemplos de aplicación en [Barbacci et al 97] [Kazman et al 1998a, 1998b], [Woods 1999] y por supuesto en el SEI [SEI-ata 2002].

El ATAM puede usarse para crear definiciones preliminares de la arquitectura, analizar decisiones arquitectónicos en ausencia de código, analizar diversas arquitecturas candidatas para el desarrollo de un sistema y analizar sistemas ya existentes. Su propósito principal es [Kazman et al 2000]:

Descubrir las consecuencias de las decisiones arquitectónicas a la luz de los requisitos de los atributos de calidad.

Asumiendo que:

[37] El lector interesado en el proceso de negociación de requisitos puede consultar [Boehm et al 1995, 1996, 1998a y 1998b]. En estas referencias Bohem y otros describen el sistema WinWin y la herramienta QARCC (*Quality Attribute Risk and Conflict Consultant*) que constituyen un sistema experto para determinar estrategias de diseño a partir de compromisos (*tradeoffs*) alcanzados entre atributos de calidad. Este trabajo está relacionado con otros desarrollos atribuibles a Bohem como la herramienta COCOMO de estimación de costes.

Los análisis de atributos de calidad específicos deben realizarse teniendo en cuenta su dependencia con el resto de los atributos del sistema. Los atributos de calidad no son independientes, sino que interaccionan entre sí a través de las relaciones estructurales impuestas por la arquitectura.

Las ideas básicas en las que se sustenta el método son las siguientes:

- Es un método conducido por escenarios.
- No evalúa la arquitectura respecto de atributos de calidad abstractos, sino respecto de requisitos concretos.
- Requiere de una descripción de las estructuras del sistema tanto más elaborada cuanto mayor sea su influencia en los atributos de calidad que se pretenden evaluar.
- Su ejecución debe consumir pocos recursos y realizarse en un lapso de tiempo relativamente corto.
- Toma en consideración tanto los requisitos técnicos, como aspectos sociales y de negocio.

Hasta aquí no hay diferencias con el SAAM, pero en el caso del ATAM estos aspectos se refinan y se incorporan dos elementos nuevos:

- La caracterización de los atributos de calidad.
- La noción de estilo arquitectónico.

Es importante destacar que ATAM necesita una caracterización precisa de los atributos de calidad. Para ello, utiliza la caracterización de atributos definida por Klein [Klein et al 1999a, 1999b] (véanse figuras 3.6 a 3.9), racionaliza el proceso de obtención de escenarios, introduce los árboles de utilidad (*utility trees*) para obtener las especificaciones de los atributos y relaciona los mismos con estilos arquitectónicos a través de los ABAS y de los escenarios generales ya descritos en el capítulo 3.

El proceso de evaluación permitirá descubrir los riesgos, puntos sensibles (*sensitivity points*) y puntos de compromiso (*tradeoff points*) asociados a las decisiones arquitectónicos:

- **Los riesgos** se producen cuando hay que elegir entre varias decisiones de diseño arquitectónico o cuando, una vez tomada dicha decisión, no pueden determinarse claramente los efectos que produce sobre alguno de los atributos de calidad. También hay riesgos asociados a la gestión del proyecto y a la comunicación entre las partes implicadas.
- **Los puntos sensibles** de una arquitectura son aquellas propiedades de sus componentes y relaciones altamente correladas con el cumplimiento de un determinado atributo de calidad y que por tanto son críticas para el cumplimiento del mismo.
- **Los puntos de compromiso** son los puntos sensibles que involucran a más de un atributo de calidad. Los puntos de compromiso son los lugares donde se producen los conflictos entre atributos, o dicho de otra manera, los lugares en los cuales los atributos interaccionan y no pueden considerarse por separado.

Con objeto de identificar los riesgos, puntos sensibles y puntos de compromiso, ATAM utiliza tres elementos**:** los escenarios de alta prioridad, las cuestiones específicas de atributo y los estilos arquitectónicos. Durante la evaluación, el arquitecto realiza los escenarios a través de las estructuras definidas en la arquitectura. Para ello deben identificarse los componentes y conectores involucrados en su realización y plantearse las cuestiones necesarias para identificar

los riesgos, los puntos sensibles y los puntos de compromiso. La figura 4.2 resume las entradas y salidas del método.

El propósito del método no es producir análisis precisos de los atributos de calidad, sino identificar tendencias que permitan determinar si el enfoque arquitectónico adoptado es el correcto [Kazman et al 2000]. El objetivo no es caracterizar de forma precisa el comportamiento de los atributos, sino descubrir en qué lugares y de que manera les afectan las decisiones de diseño arquitectónicos. Una vez encontrados los puntos sensibles y de compromiso se puede determinar la necesidad de modelar con mayor exactitud los componentes y las relaciones involucrados en los mismos y analizar su comportamiento en relación con los atributos de calidad. Los resultados de dichos análisis pueden incorporarse al método, pero los análisis en sí están fuera del mismo.

En las siguientes secciones se describirán los pasos definidos en el método y sus características más significativas.

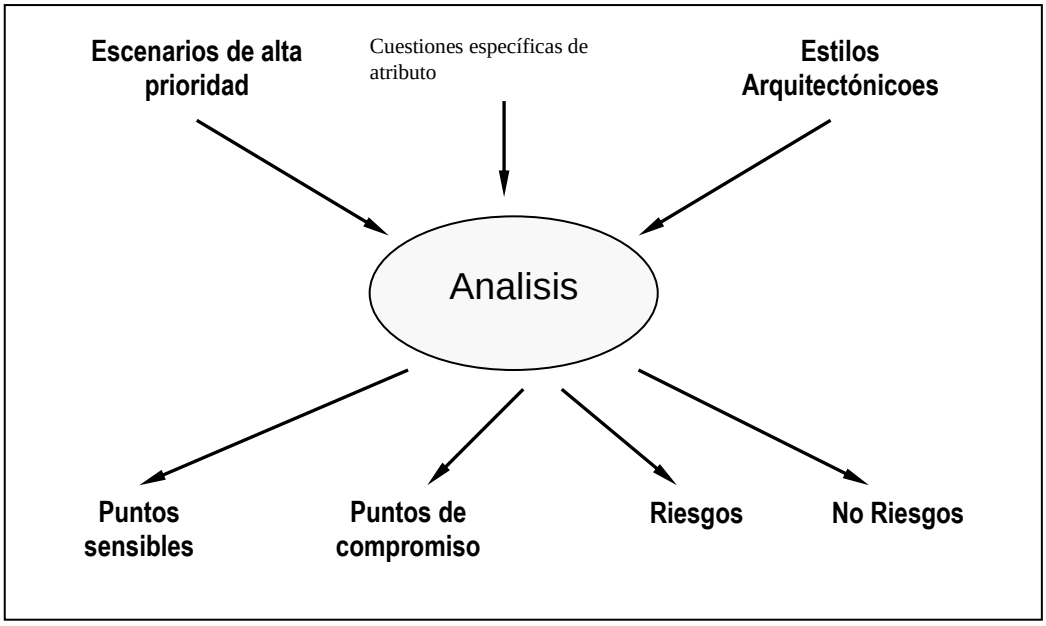


Figura 4.2: ATAM: Entradas/Salidas.

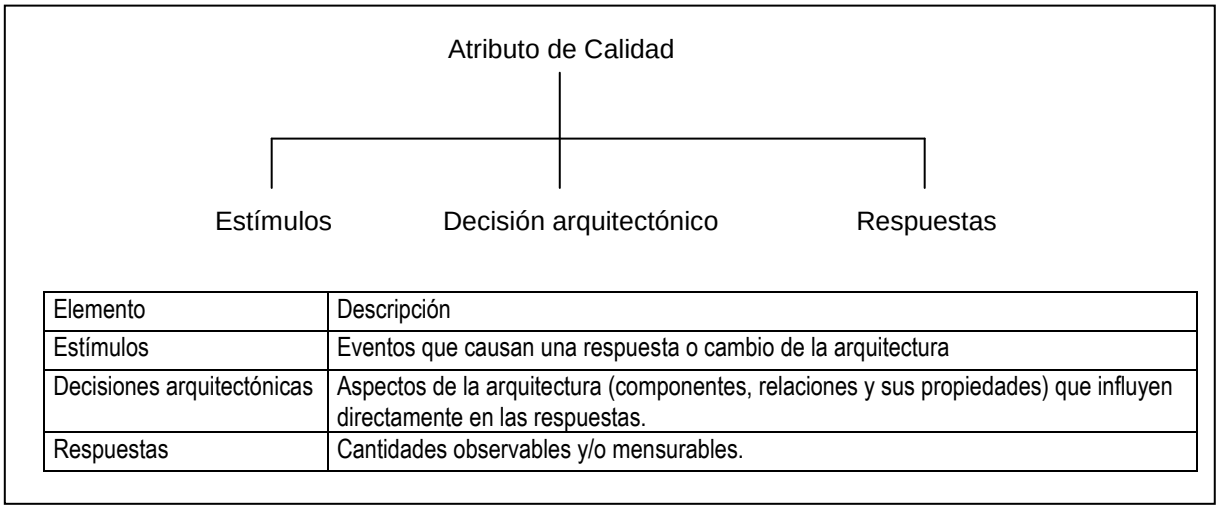
4.3.2 Caracterización de atributos. Escenarios y Árboles de utilidad.

ATAM caracteriza los atributos de calidad en función de tres parámetros: estímulos externos, decisiones arquitectónicas y respuestas (figuras 3.6 a 3.9. La figura 3.6 se repite en la 4.3 para la comodidad del lector).

Los estímulos externos son aquellos eventos que producen una respuesta por parte de la arquitectura. Para analizar hasta que punto la arquitectura cumple los requisitos de los atributos

de calidad, tales requisitos deben expresarse de forma concreta, para que puedan ser observados y medidos o, al menos, estimados. Estos parámetros observables constituyen las respuestas asociadas a la caracterización del atributo. Las decisiones arquitectónicas son aquellos aspectos de la arquitectura (componentes, conectores y sus propiedades) que influyen directamente en el logro de las respuestas.

Las herramientas que utiliza ATAM para analizar los atributos son las cuestiones específicas de atributo, los estilos arquitectónicos y los escenarios de alta prioridad. Las cuestiones específicas de atributo proporcionan el marco para razonar acerca de los requisitos de los atributos de calidad. Los escenarios permiten trasladar ese razonamiento a casos de estudio concretos. Los estilos arquitectónicos proporcionan el conjunto de soluciones cuya validez debe confrontarse, mediante la realización de los escenarios, con los requisitos que se han formulado.



Elemento	Descripción
Estímulos	Eventos que causan una respuesta o cambio de la arquitectura
Decisiones arquitectónicas	Aspectos de la arquitectura (componentes, relaciones y sus propiedades) que influyen directamente en las respuestas.
Respuestas	Cantidades observables y/o mensurables.

Figura 4.3: ATAM: Caracterización de atributos.

Escenarios

Un escenario puede ser una breve descripción de la interacción de un usuario con el sistema (casos de uso tradicionales [Jacobson et al 1992]) o la descripción de alguna característica importante del sistema desde el punto de vista de todas las partes implicadas. ATAM distingue tres tipos de escenarios:

- **Escenarios de casos de uso.** Describen la interacción de los usuarios con el sistema en ejecución.
- **Escenarios de crecimiento.** Representan probables futuros usos del sistema. Este tipo de escenarios está estrechamente ligado a las características de modificabilidad del sistema, pero sus efectos se extienden por todos los atributos.
- **Escenarios exploratorios.** Representan situaciones extremas. Su objetivo es establecer los límites del diseño y revelar las suposiciones que implícitamente puedan hacerse las diferentes partes implicadas sobre el mismo.

Una vez obtenidos, los escenarios deben priorizarse en un proceso en el que tienen cabida todas las partes implicadas. Para obtener y priorizar los escenarios, ATAM utiliza dos mecanismos: los árboles de utilidad (*utility trees*) y la tormenta de ideas estructurada[38] (véase figura 4.4).

Árboles de utilidad

Los árboles de utilidad proporcionan un marco de razonamiento deductivo para traducir los requisitos del sistema en escenarios concretos asociados a los diferentes atributos de calidad. El objetivo de los árboles de utilidad es definir requisitos concretos sobre los atributos de calidad. La figura 4.5 muestra un ejemplo de tales árboles (adaptado de [Kazmann et al 2000]).

Los atributos de calidad se sitúan en el primer nivel del árbol. Su nivel de abstracción es demasiado alto para guiar el proceso de evaluación, por ello su significado se va concretando en las sucesivas ramas del árbol. Debajo de cada uno de los atributos de calidad, se definen sub-atributos que reflejan los sucesivos refinamientos que se van aplicando a sus raíces. Los nodos hojas son lo suficientemente específicos para caracterizar los atributos y organizar los mismos según su orden de importancia. La asignación de prioridades a los atributos-hoja tiene dos componentes que vienen dadas por la importancia relativa del nodo y por el grado de riesgo que implica la obtención del atributo. Las hojas del árbol de utilidad definen por sí mismas los primeros escenarios

	Árboles de Utilidad	**Tormenta de ideas**
Partes implicadas (stakeholders)	Arquitectos y líder del proyecto	Todas las partes implicadas
Tamaño típico del grupo	2 evaluadores 2 ó 3 desarrolladores	4 ó 5 evaluadores 5 a 10 partes implicadas
Objetivos primarios	Obtener, concretar y priorizar los requisitos de los atributos de calidad. Proporcionar las bases para el resto de la evaluación.	Recoger los puntos de vista de las partes implicadas con el fin de validar los requisitos obtenidos con el árbol de utilidad.
Enfoque	Deductivo (De lo general a lo específico)	Inductivo (De lo específico a lo general)

Figura 4.4: ATAM: Obtención de escenarios: Árboles de utilidad y Tormenta de ideas.

[38] Del inglés *Structured brainstorming*

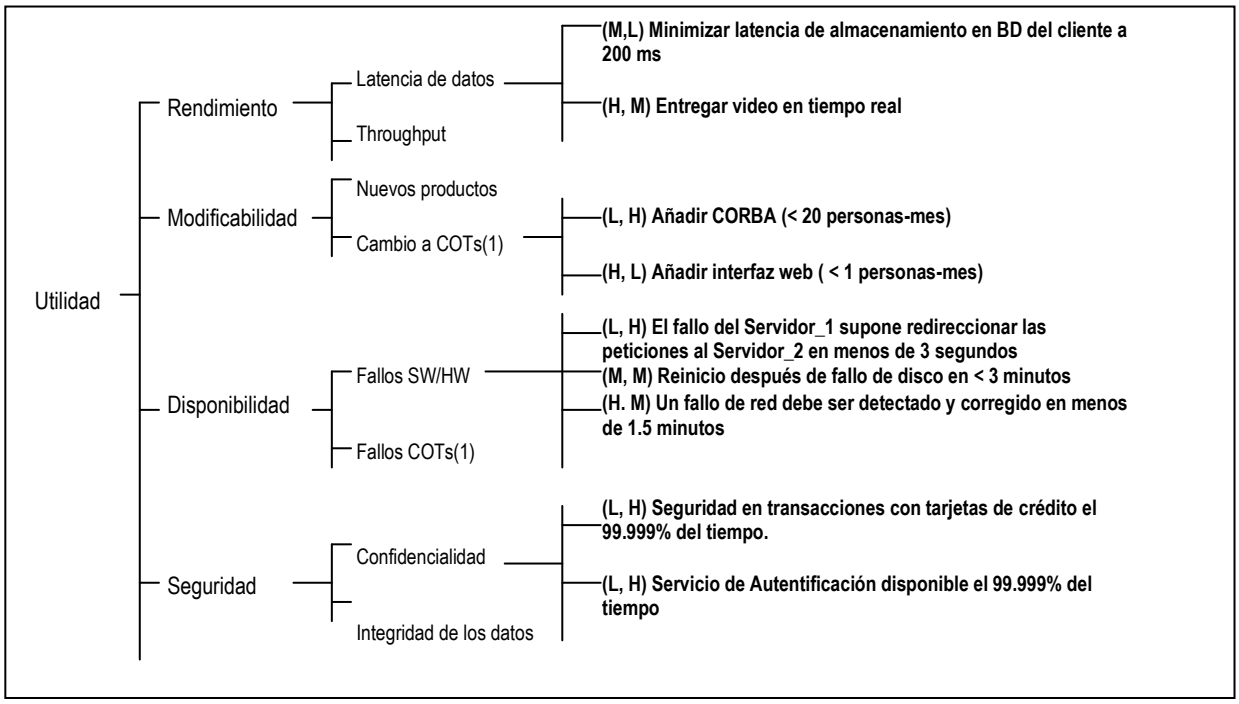


Figura 4.5: Ejemplo de árbol de utilidad (fragmento).

4.3.3 Los pasos de ATAM

El método ATAM consta de 9 pasos. Aunque dichos pasos están numerados y sugieren un desarrollo lineal del método, su orden puede variar de acuerdo con las necesidades del proyecto y puede ser necesario realizar varias iteraciones de los mismos. La importancia relativa de cada paso y el tiempo que puede consumir dependen de las características de cada proyecto. Estos pasos, resumidos en la figura 4.6, son los siguientes:

Paso	Actividad	Partes implicadas
1	Presentación de ATAM	Equipo de Evaluación Equipo de desarrollo Representación cualificada del cliente
2	Presentación de los factores de negocio	Idem anterior
3	Presentación de la arquitectura	Idem anterior
4	Identificación de los estilos arquitectónicoes	Idem anterior
5	Generación del árbol de utilidad	Idem anterior
6	Análisis de los estilos arquitectónicoes	Idem anterior
7	Tormenta de ideas y priorización de escenarios	Todas
8	Análisis de los estilos arquitectónicoes	Equipo de Evaluación Equipo de desarrollo Representación cualificada del cliente
9	Presentación de resultados	Todas

Figura 4.6: ATAM: Pasos del método y partes implicadas en los mismos.

1. Presentación del método.

Se explica el método a todas las partes implicadas, sus fundamentos y pasos, las técnicas para la obtención y refinado de escenarios, el significado de los riesgos, los puntos sensibles y los puntos de compromiso, etc. El objetivo es que todas las partes implicadas entiendan el proceso y se sientan involucradas en él.

2. Descripción de los factores de negocio.

El sistema en cuyo desarrollo se va a utilizar la arquitectura debe ser perfectamente entendido por todas las partes. El cliente describe el sistema desde la perspectiva de negocio. La figura 4.7 muestra algunas de las cuestiones que debe abordar el cliente.

Descripción de los factores de negocio:

- **Descripción del entorno de negocio:**
  - ✓ Historia.
  - ✓ Signos diferenciadores del producto.
  - ✓ Requisitos de negocio
  - ✓ Necesidades de la clientela y cómo se pretende satisfacerla.
  - ✓ ...
- **Restricciones de negocio:**
  - ✓ Tiempo de comercialización.
  - ✓ Demandas del cliente.
  - ✓ Estándares.
  - ✓ Coste
  - ✓ Tiempo de vida del sistema.
  - ✓ ...
- **Atributos de calidad deseados:**
  - ✓ Rendimiento.
  - ✓ Disponibilidad.
  - ✓ Seguridad.
  - ✓ Usabilidad.
  - ✓ ...

y de qué manera se derivan de los factores de negocio.

- **Restricciones técnicas:**
  - ✓ Uso de componentes comerciales.
  - ✓ Interacción/integración con software existente
  - ✓ Plataformas Hardware y SOs disponibles.
  - ✓ ....

Figura 4.7: ATAM: Cuestiones del cliente.

3. Presentación de la arquitectura.

El equipo de desarrollo presenta la arquitectura describiendo el enfoque adoptado para el logro de los atributos de calidad definidos en el paso anterior. Esta descripción debe realizarse en un nivel de abstracción inteligible para todas las partes implicadas, pero en todo caso suficiente para determinar la conformidad de la arquitectura con los atributos que se pretende evaluar. La figura 4.8 muestra algunas de las características que deben describirse durante la presentación de la arquitectura.

Presentación de la arquitectura
Explicación de los requisitos arquitectónicos y de las propiedades mensurables asociadas a dichos requisitos. Presentación de estándares, modelos y estilos arquitectónicos que pueden usarse para el logro de tales requisitos.
Descripción de alto nivel de las diferentes vistas o estructuras de la arquitectura.
Vista funcional (vista lógica): Funciones, abstracciones del sistema, modelo conceptual, elementos del dominio y sus dependencias, flujos de datos, etc.
Vista de módulos/niveles/subsistemas (vista de desarrollo): Presentación de los subsistemas, niveles y modules que describen la descomposición funcional del sistema. Descripción de los objetos, procedimientos y funciones de cada uno de ellos (los públicos) y de la forma en que se relacionan (llamadas a procedimiento, invocación de métodos, *callback*, citas, etc)
Vista de procesos: Descripción de los procesos y de sus formas de sincronización, flujos de datos y eventos.
Descripción del hardware (vista física): CPUs, dispositivos de almacenamiento primario y secundario, dispositivos y sensores, así como los buses y redes de comunicaciones que los conectan.
Descripción de los estilos arquitectónicos empleados, explicando razonadamente los atributos de calidad que se ven favorecidos o perjudicados por su adopción.
Descripción de los componentes software comerciales que se contemplan y sus criterios de selección e integración.
Realización sobre la arquitectura de los escenarios de casos de uso más importantes (de 1 a 3), si es posible incluyendo una estimación de los recursos que consumen en tiempo de ejecución.
Realización sobre la arquitectura de los escenarios de cambio más importantes (de 1 a 3), si es describiendo el impacto del cambio en términos de su dificultad y de los componentes, conectores e interfaces afectados.
Descripción de los riesgos detectados en relación con el cumplimiento de los atributos.

Figura 4.8: ATAM: Presentación de las características de la arquitectura.

4. Identificación de estilos arquitectónicos.

Se identifican claramente los estilos arquitectónicos usados en la arquitectura. Dichos estilos representan los medios mediante los cuales la arquitectura puede alcanzar sus atributos de calidad y definen la forma en que un sistema puede crecer, modificarse, integrarse con otros sistemas y responder a estímulos externos.

5. Generación del árbol de utilidad.

A partir de los factores de negocio expuestos por el cliente (paso 2), se genera un árbol de utilidad (figura 4.5) que traduzca sus expectativas a requisitos específicos de los atributos de calidad. Se describen escenarios en términos de los estímulos que recibirá el sistema y de las respuestas esperadas y se priorizan en función tanto de su importancia para el éxito del proyecto como de los riesgos que implican.

El árbol de utilidad sirve de guía para los siguientes pasos del método, indicando las áreas hacia las que el evaluador debe dirigir sus esfuerzos. Recuérdese que cada una de las hojas del árbol de utilidad define por sí misma un escenario.

6. Análisis de los estilos arquitectónicos.

Una vez que el alcance de la evaluación ha sido definido mediante el árbol de utilidad (paso 5) y los estilos arquitectónicos han sido identificados (paso 4), el equipo de evaluación puede empezar a analizar la arquitectura. El primer paso consiste en asociar los escenarios más prioritarios del árbol de utilidad con los estilos arquitectónicos empleados en la arquitectura. Para ello, los equipos de desarrollo y el de evaluación[39] deben realizar las siguientes tareas.

1. Realización de los escenarios más prioritarios (en este punto, y si es la primera iteración del método, los escenarios vienen determinados exclusivamente por las hojas del árbol de utilidad).
2. Identificación de cada uno de los componentes, conectores, configuraciones y restricciones que están directamente relacionados con los atributos de calidad más importantes (hojas más prioritarias del árbol de utilidad).
3. Formular y responder una serie de cuestiones *específicas* sobre los atributos requeridos y los estilos seleccionados. Estas cuestiones, que deben relacionar los atributos con los elementos de la arquitectura, pueden obtenerse de muy diversas fuentes. Los ABAS [Klein et al 1999b] proporcionan un guía para la formulación de cuestiones específicas de atributo, pero también se puede echar mano de otras fuentes. [Gamma et al 1995], [Buschmann et al 1996], [Shaw et al 1997] o [Bass et al 1998] proporcionan criterios de selección de estilos y patrones. La propia experiencia del equipo de desarrollo, de los evaluadores y de algunas de las partes implicadas en el sistema pueden ser también fuentes útiles de cuestiones.

   Estas cuestiones enfrentan los requisitos de los atributos de calidad con los mecanismos o estilos arquitectónicos utilizados por lo que su respuesta es el punto de partida para determinar riesgos y no-riesgos, puntos sensibles y puntos de compromiso.

Al final de este paso, el equipo evaluador debe tener ya una idea general de los aspectos más importantes de la arquitectura. A partir de aquí el objetivo es refinar este conocimiento y llegar a conclusiones acerca de la adecuación de la arquitectura con sus objetivos.

[39] Equipo de desarrollo y el de evaluación pueden ser el mismo, sin embargo se aconseja que sean distintos [Kazman et al 2000][Bachman et al 2000a].

7. Generación de escenarios y asignación de prioridades a los mismos.

En este paso el objetivo es generar escenarios y asignarles diferentes prioridades. Mientras que la obtención del árbol de utilidad implica al personal técnico, la obtención de escenarios involucra a todas las partes implicadas y sigue la técnica del *brainstorming*. La priorización de escenarios se realiza mediante votación de *todas* las partes implicadas. Si el tiempo y los recursos disponibles para la evaluación son limitados el equipo de evaluación puede considerar sólo los escenarios más prioritarios.

Una vez definidos y priorizados los escenarios se contrastan con las hojas del árbol de utilidad. Cualquier discrepancia entre ambos conjuntos de escenarios supone una discrepancia entre los puntos de vista del personal técnico que ha elaborado el árbol de utilidad y el resto de las partes implicadas. Estas discrepancias debe ser resueltas o al menos explicadas.

8. Análisis de los estilos arquitectónicos.

Este paso es una repetición del paso 6, pero tomando en consideración los nuevos escenarios definidos en el paso 7. Si el árbol de utilidad ha sido bien definido este paso debe ser muy breve, pues los escenarios definidos por las hojas del árbol de utilidad deben ser básicamente los mismos que los definidos en el paso 7. Si no es así, debe volverse al paso 4, redefinir el árbol de utilidad y repetir los pasos 5, 6 y 7.

9. Presentación de resultados.

Para terminar, la información generada durante la ejecución del método se organiza, resume y presenta a todas las partes implicadas. Esta presentación debe incluir:

- Documentación de los estilos arquitectónicos.
- Conjunto de escenarios y su orden de prioridad.
- El conjunto de cuestiones específicas de atributo formuladas y sus respuestas.
- El árbol de utilidad.
- Los riesgos y no-riesgos descubiertos.
- Los puntos sensibles y los puntos de compromiso.

Es posible incluir recomendaciones sobre las estrategias a seguir para mitigar los riesgos y resolver los conflictos entre atributos. Estas recomendaciones pueden referirse a aspectos de gestión (proceso de desarrollo, equipos de desarrollo, estrategias de gestión y comunicación, etc) y técnicos. Pero las recomendaciones no forman parte del método, el cual está centrado en la detección de riesgos, puntos sensibles y puntos de compromiso.

4.3.4 Conclusiones y críticas al método

Aunque ATAM puede utilizarse para evaluar sistemas ya implementados, el método está concebido para servir de guía en el proceso de diseño de la arquitectura. En un ciclo de vida en espiral, ATAM sirve para identificar riesgos y conducir los procesos de análisis de atributos y de toma de decisiones arquitectónicas.

Las dos mayores innovaciones que diferencian a ATAM de SAAM son, como se ha dicho:

- *La caracterización de los atributos de calidad.*

- *La noción de estilo arquitectónico.*

La caracterización de atributos permite la definición precisa de los mismos en términos de estímulos, decisiones arquitectónicas y respuestas mensurables. La noción de estilo arquitectónico relaciona el cumplimiento de dichos atributos con las decisiones de diseño adoptadas por la arquitectura. Los puntos sensibles y de compromiso tienen el propósito de identificar claramente las áreas de la arquitectura donde se producen los riesgos y las interacciones entre atributos, apuntando hacia los lugares en los que el equipo de desarrollo debe centrar sus esfuerzos.

Sin embargo, el método no está exento de críticas, algunas de ellas realizadas por el propio SEI [Lopez 2000]. Se intentará resumir aquí las más importantes.

Si bien el esquema utilizado para caracterizar los atributos puede ser válido, los criterios adoptados para tal caracterización no están suficientemente explicados. Es necesario acudir a fuentes alternativas para descubrir los modelos y propiedades asociados a cada atributo (p.e.: *rate monotonis analysis* para el rendimiento, modelos de Markov y teoría de colas para la disponibilidad, etc). Aunque es cierto que tales modelos están fuera del método, son las herramientas con las que se cuenta para evaluar cada atributo por separado. Los modelos sobre los que se realizan dichas evaluaciones no son en general los mismos que los utilizados para describir la arquitectura, por lo que habría que establecer una correspondencia entre los mismos. Los estilos arquitectónicos de [Klein 1999b] son una guía en este sentido, pero sólo parcial, ya que son específicos de atributo.

Los criterios de evaluación se obtienen de dos fuentes distintas: el árbol de utilidad definido por expertos y los escenarios definidos por las partes implicadas. Las discrepancias entre ambas fuentes están contempladas, pero no se sugiere más forma de resolverlas que repetir la ejecución del método.

Aunque se describen algunas pautas para la obtención de las cuestiones específicas de atributo, no se define ninguna guía para saber hasta que punto tales cuestiones representan completamente los intereses de las partes implicadas.

ATAM puede usarse para crear definiciones preliminares de la arquitectura, analizar decisiones arquitectónicas en ausencia de código, analizar diversas arquitecturas candidatas y analizar sistemas ya existentes. Aunque las características de cada una de estas tareas pueden ser muy diferentes, el método no define vías alternativas para cada una de ellas.

Los pasos del método están definidos de forma muy general y no se explican convenientemente las relaciones entre los mismos. En [Kazman et al 2000] se afirma que la ejecución del método no tiene que ser necesariamente lineal, pero no explica convenientemente cuando retroceder o avanzar.

Como el SAAM, el ATAM no establece ningún mecanismo para guiar la negociación entre las partes implicadas para alcanzar compromisos respecto de los atributos del sistema.

4.4 Diseño de Arquitecturas de Referencia. El Método de Diseño Basado en la Arquitectura.

4.4.1 Introducción

El Método de Diseño Basado en la Arquitectura o ABD (de sus siglas en inglés *Architecture Based Design Method*) es un método para el diseño de arquitecturas software para líneas de producto y sistemas de larga vida operativa, desarrollado por el SEI de la CMU. El método se basa en tres pilares:

- Descomposición funcional del problema, basada en los criterios de bajo acoplamiento y alta cohesión.
- Realización de los requisitos de calidad y de negocio a través de la elección de los estilos arquitectónicos adecuados.
- El uso de *plantillas software*. Las plantillas software en el contexto del ABD se refieren a los patrones de interacción de los elementos de la arquitectura entre sí y con la infraestructura que utilizan y al conjunto de responsabilidades comunes que dichos elementos deben asumir.

Las entradas del método están constituidas por la lista de requisitos y restricciones que debe satisfacer la arquitectura. Los requisitos se clasifican dentro de tres categorías: requisitos funcionales, requisitos de calidad y requisitos de negocio e, independientemente de la categoría a la que pertenezcan, se dividen en abstractos y concretos. Los requisitos abstractos se utilizan para generar la arquitectura, los concretos para validarla.

El ABD considera que cada sistema se compone de dos partes, una correspondiente a la aplicación y otra correspondiente a la infraestructura que la soporta y sobre la cual se ejecuta (sistema operativo, *middleware*, servidores externos, bases de datos pre-existentes, etc). El ABD considera ambas partes y ambas contribuyen en la descomposición del sistema y en la definición de la arquitectura.

El método ABD descompone el sistema en subsistemas recursivamente. Es decir, las mismas reglas que se aplican para descomponer el sistema en susbsistemas se aplican a su vez para descomponer dichos subsistemas en otros más simples. Los requisitos funcionales se logran asignando una parte de la funcionalidad del sistema a cada uno de los subsistemas o componentes resultantes de las sucesivas descomposiciones del mismo. Los requisitos no funcionales se alcanzan mediante la selección del estilo arquitectónico que mejor se adapte a los mismos. Cada descomposición se examina por un lado desde las perspectivas de las vistas lógica, de concurrencia y de despliegue y por otro desde el punto de vista de las plantillas software. En los siguientes apartados se presentarán los fundamentos del método y su terminología. Una vez descritos estos aspectos se pasará a la descripción del método.

4.4.2 Elementos de Diseño y Vistas Arquitectónicos

El ADB considera elementos de diseño al *sistema*, a cada uno de los *subsistemas* que resultan de las sucesivas descomposiciones del mismo y a los *componentes conceptuales* que pertenecen a

los subsistemas finales (aquellos que no se han dividido en otros subsistemas). Cada uno de estos elementos tiene asignada una parte de la funcionalidad que se corresponde con las responsabilidades que debe asumir. Asimismo, todos los elementos de diseño tienen una *interfaz conceptual* que describe las entradas que reciben y las salidas que producen.

La figura 4.9 muestra la relación entre los distintos elementos de diseño. Al nivel más alto se encuentra el *sistema*, que se descompone en dos o más *subsistemas conceptuales*. Puesto que el método es recursivo estos subsistemas pueden descomponerse en otros de acuerdo con las mismas reglas que rigen la descomposición del sistema. En el último nivel se encuentran los *componentes conceptuales*. Los *componentes concretos* que aparecen en la parte inferior de la figura 4.9 son componentes para los cuales se ha alcanzado un compromiso de diseño acerca de los objetos y procesos que contienen y de la estructura de los mismos. El ABD no se ocupa de estos componentes, que deben ser diseñados a partir de los componentes conceptuales utilizando otros métodos. En cada uno de los niveles se definen *plantillas software* que describirán más adelante.

Las vistas de la arquitectura que utiliza el ABD se basan en el modelo 4+1 de Krutchen, si bien con algunas restricciones. Así, el ABD no utiliza la vista de desarrollo y la vista de procesos, que el ABD denomina de concurrencia, se limita a examinar aspectos relacionados con la existencia de múltiples usuarios, el acceso a recursos compartidos, el arranque del sistema y la identificación de las actividades que pueden ejecutarse en paralelo. La vista de casos de uso se completa con la definición de escenarios que describan los requisitos no funcionales con el mayor grado de detalle posible.

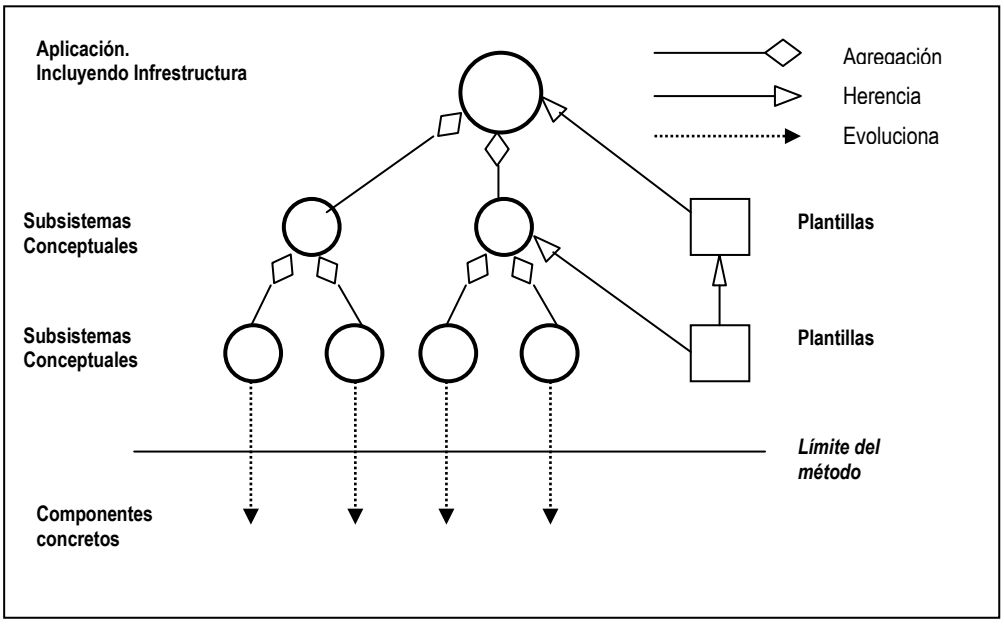


Figura 4.9: Descomposición del sistema en elementos de diseño.

4.4.3 Plantillas Software e Infraestructura del sistema.

El concepto de plantilla software en el ABD describe una taxonomía de los elementos de diseño basada en sus patrones de interacción y en las responsabilidades que los diferentes elementos de diseño deben asumir. Los criterios que definen dicha taxonomía y determinan el contenido de las plantillas son los siguientes:

1. **Interacción con los servicios.** Patrones que describen como un tipo determinado de elementos de diseño interacciona con los servicios provistos por el resto de los elementos de diseño. Por ejemplo, si hay un servicio de diagnóstico, todos los elementos que proporcionan datos a dicho servicio deben usar el mismo protocolo.
2. **Interacción con la infraestructura.** Patrones que describen como un tipo determinado de elementos de diseño interacciona con la infraestructura. Por ejemplo, se determina utilizar sólo los servicios del sistema operativo definidos en el estándar POSIX.
3. **Responsabilidades comunes[40].** Responsabilidades que aunque no estén relacionadas con la interacción con los servicios o con la infraestructura son comunes a todos los elementos de diseño de un tipo determinado. Por ejemplo, la forma en que un determinado tipo de elementos debe llevar a cabo el tratamiento de las excepciones.

Las plantillas software sirven para:

1. Facilitar la integración de componentes y subsistemas. Todos los elementos de diseño que compartan un determinado patrón de interacción pueden integrarse de la misma manera.
2. Identificar los patrones de comportamiento comunes a diferentes elementos de diseño y que por tanto son susceptibles de implementarse en componentes reutilizables en todos esos elementos.
3. Construir un esqueleto del sistema lo más independiente posible de la funcionalidad asignada a los elementos de diseño. Si el esqueleto del sistema no depende de la funcionalidad se facilita el crecimiento incremental de las aplicaciones. El uso de plantillas constituye en este sentido una forma de abordar las variantes de funcionalidad.
4. Facilitar el uso de modelos formales, los cuales se construyen a partir de los patrones de interacción entre los elementos y no de su funcionalidad específica.

4.4.4 Directrices de la Arquitectura, estilos arquitectónicos y división de la funcionalidad.

Las directrices de la arquitectura (del inglés *architectural drivers*) son aquellos requisitos que conforman la arquitectura. Las directrices de la arquitectura se determinan a partir del propósito del sistema y de los factores de negocio más críticos. Por ejemplo, la finalidad de un sistema de simulación de vuelo es entrenar pilotos (propósito), lo cual requiere una alta fidelidad en la simulación y la ejecución de las maniobras en tiempo real. La empresa desea desarrollar una línea de producto (factor de negocio), lo que requiere una alta adaptabilidad del sistema a las características de diferentes aeronaves.

[40] En [Bachmann et al 2000b] se denomina a esas responsabilidades "*citizenship responsabilities*", responsabilidades de ciudadanía u obligaciones del buen ciudadano

Las directrices de la arquitectura no dependen de los detalles de los requisitos funcionales, sino de la abstracción de los mismos. En el ejemplo de arriba, el hecho de que las aeronaves a simular tengan uno, dos o cuatro motores, no es determinante, sino la capacidad de procesar grandes cantidades de datos en tiempo real. Diferentes directrices implican diferentes arquitecturas. Uno de los pasos más importantes del ABD es la selección de un estilo o conjunto de estilos arquitectónicos dominantes, bajo el criterio de que el estilo seleccionado debe ser el que mejor se adapte a las directrices de la arquitectura. Como se ha explicado en el capítulo 3, un estilo arquitectónico consiste en una colección de componentes acompañada por la descripción de los patrones de interacción entre los mismos. Pero los componentes a los que se refiere esta definición son conceptuales y no se les supone ninguna funcionalidad específica. El estilo seleccionado no soporta funcionalidad alguna hasta que la funcionalidad del sistema sea repartida y asignada a sus componentes. Y esta asignación de funcionalidad nuevamente depende de las directrices de la arquitectura. En el ejemplo del simulador de vuelo la adaptabilidad es una *directriz* que *dicta* que los cálculos correspondientes a cada motor se hagan en una unidad de computación separada. En el ABD el término *división de funcionalidad* se refiere al reparto de los requisitos funcionales entre los elementos de diseño en función de los criterios que determinen las directrices arquitectónicas.

4.4.5 El ABD en el ciclo de vida del sistema. Entradas y salidas del método.

El ABD puede inscribirse dentro del Proceso de Desarrollo Basado en la Arquitectura (*Architecture Based Development Process*) definido por Bass y Kazman en [Bass et al 1999]. Dicho proceso se resume en la figura 4.10. En ella puede observarse que el proceso de diseño se encuentra entre la definición de requisitos y el análisis de la arquitectura. El proceso puede repetirse de forma iterativa, resolviendo en cada iteración una parte del diseño.

4.4.5.1 Entradas del ABD

Las entradas del método están constituidas por la lista de requisitos y restricciones que debe satisfacer la arquitectura. Los requisitos se clasifican dentro de tres categorías: requisitos funcionales, requisitos de calidad y requisitos de negocio e, independientemente de la categoría a la que pertenezcan, se dividen en abstractos y concretos. **Los requisitos abstractos se utilizan para generar la arquitectura, los concretos para validarla.**

Diseñar una arquitectura para una familia de sistemas implica ser capaz de prever las variaciones entre los distintos miembros de la familia de forma que la arquitectura pueda adaptarse a las mismas. Las *variantes* son aspectos que cambian de un sistema a otro y pueden referirse a cambios en la funcionalidad, en la plataforma o en el entorno. Los *invariantes* son aquellos aspectos comunes a todos los sistemas del dominio, línea de producto o familia de sistemas considerada.

4.4.5.2 Requisitos funcionales abstractos.

Las arquitecturas de referencia deben ser aplicables a todos los sistemas de una familia o dominio, por tanto los requisitos que debe cumplir son distintos que los que debe satisfacer la arquitectura de un sistema específico. Un ejemplo:

Arquitectura de un Sistema de Teleoperación concreto:

> *Las alarmas deben ser procesadas antes de que transcurran 50 ms desde su activación.*

Arquitectura de referencia:

> *Las alarmas deben ser procesadas antes de que transcurra un intervalo de tiempo determinado.*

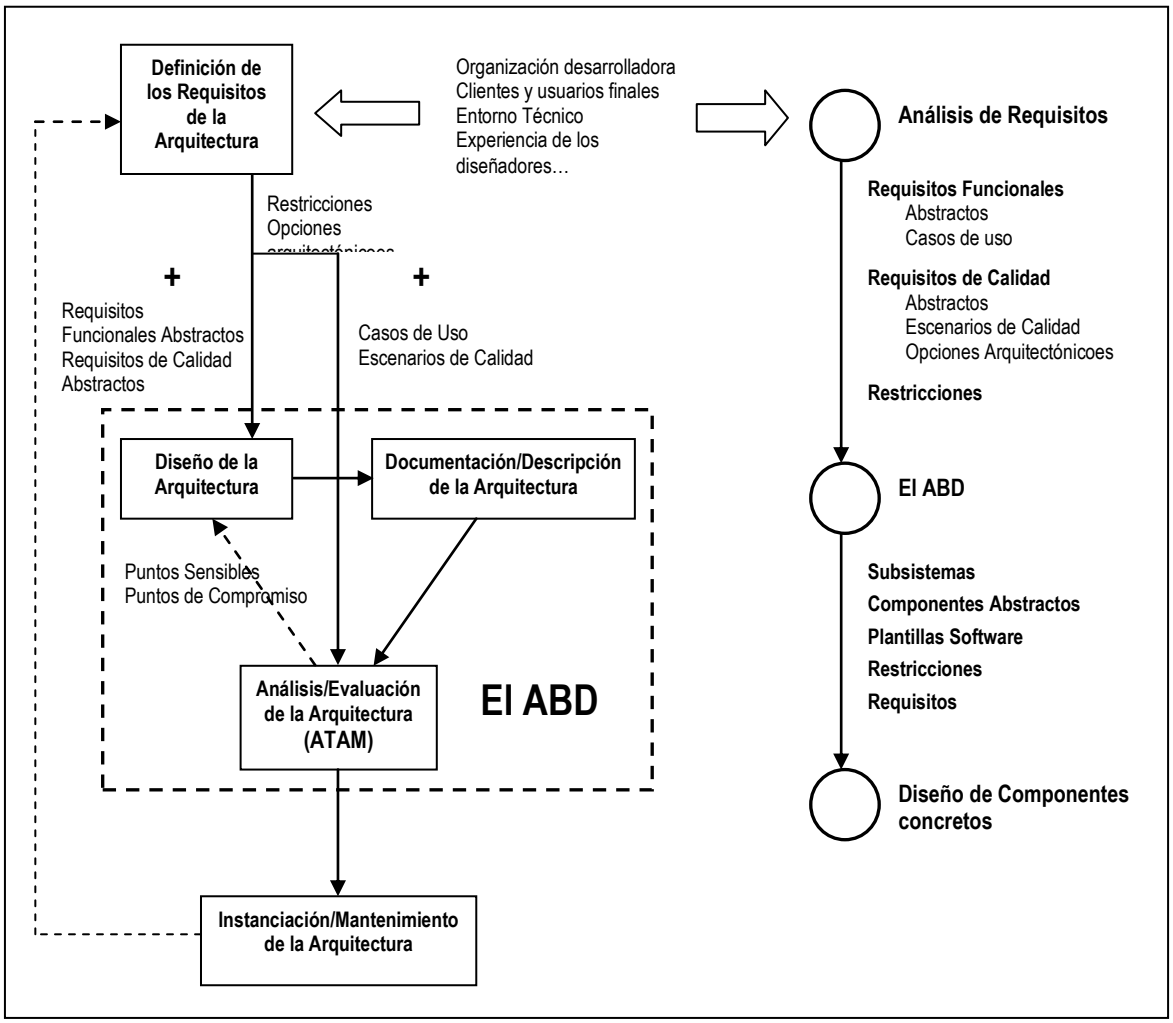


Figura 4.10: El método de desarrollo basado en arquitectura y el ABD [Bass et al 1999]

Los requisitos de una arquitectura de referencia se caracterizan por su falta de especificidad. Esta falta de especificidad es necesaria para capturar las variantes de funcionalidad. En ocasiones el rango de las variantes se conoce, al menos en uno de sus extremos y el requisito de arriba puede hacerse más específico:

> *Las alarmas deben ser procesadas antes de que transcurra un intervalo que en los sistemas más críticos es de 40 ms.*
>
> O, si se conocen ambos extremos,

> *Las alarmas deben ser procesadas antes de que transcurra un intervalo de tiempo que oscila, según los sistemas, entre 40 y 500 ms.*

En otras, no es posible hacer suposiciones de este estilo, el requisito queda completamente indeterminado y sólo puede formularse de una manera *abstracta*, mediante sentencias como las del ejemplo anterior. Una arquitectura de referencia debe incluir mecanismos que aseguren que los requisitos *abstractos* pueden alcanzarse *en general*. En el ejemplo anterior la arquitectura debería incluir algún mecanismo que permita planificar las tareas y asignarles prioridades. La especificación de los requisitos en un alto nivel de abstracción es necesaria tanto para capturar las variantes de funcionalidad como por el hecho de que dichas variantes no son siempre conocidas con antelación.

4.4.5.3 Casos de Uso.

Un caso de uso es la *descripción concreta* de una interacción entre uno o más actores y el sistema. Los casos de uso representan un caso específico de interacción entre los actores y un sistema de la familia, en el que los requisitos funcionales pueden expresarse de forma precisa. Los casos de uso concretos pueden ser instancias específicas de los abstractos. Los casos de uso son fáciles de generar y su número crece rápidamente. Es necesario clasificarlos y priorizarlos de forma que sea posible seleccionar un número manejable de los mismos suficientemente representativo del comportamiento del sistema.

4.4.5.4 Calidades Abstractas y Requisitos de Negocio.

El ABD denomina *calidades* a los requisitos no funcionales o, según la terminología de Bass [Bass et al 1998], a los *atributos de calidad no observables en tiempo de ejecución*. Al igual, que los requisitos funcionales, las calidades deben estar expresadas de forma abstracta, para que abarquen a todos los sistemas de la familia. Pero incluso, dentro de ese nivel de abstracción hay que caracterizar las calidades de la forma más precisa posible. Decir que la arquitectura de teleoperación objeto de este trabajo de tesis debe ser modificable es lo mismo que no decir nada. Decir que debe ser modificable con respecto al tipo de robots y herramientas es decir algo. Decir que el sistema debe ser modificable respecto a determinados cambios en el robot y las herramientas, especificando hasta donde sea posible la naturaleza de dichos cambios, es empezar a entender el problema.

El ABD no distingue entre calidades y requisitos de negocio, ya que estos últimos deben traducirse a atributos de calidad concretos.

4.4.5.5 Escenarios de Calidad.

Al igual que los casos de uso hacen concretos los requisitos funcionales abstractos, los escenarios de calidad hacen concretas las *calidades* abstractas. Por ejemplo:

Calidad abstracta:

- *El sistema debe a adaptarse a robots de distintas características.*

Escenarios de calidad:

- *El sistema debe adoptarse a robots con diferente número de grados de libertad.*

- *El sistema debe admitir el uso de diferentes algoritmos para el cálculo de la transformada cinemática inversa.*
- *El sistema debe adaptarse a diferentes tipos de accionadores eléctricos, neumáticos o hidráulicos.*

Al igual que los casos de uso los escenarios de calidad son fáciles de generar y su número crece rápidamente. Nuevamente es necesario clasificarlos y priorizarlos en función de las directrices de la arquitectura, con objeto de examinar sólo los más relevantes.

4.4.5.6 Restricciones.

Las restricciones son decisiones de diseño preestablecidas. La mayor parte de estas decisiones vienen determinadas por los factores de negocio (inversiones previas en componentes software comerciales, interoperabilidad con sistemas ya existentes, uso de las plataformas disponibles, utilización de un determinado sistema operativo, etc), aunque en ocasiones responden a motivos técnicos (un arquitecto basándose en su experiencia puede imponer un sistema operativo determinado, aun cuando existan otros que ofrezcan servicios similares).

4.4.5.7 Opciones arquitectónicas.

Para cada uno de los requisitos funcionales y de calidad existe un estilo o conjunto de estilos que permiten su cumplimiento. Algunos requisitos pueden admitir varias opciones, otros sólo una. Las opciones deben enumerarse para proceder más adelante a la selección de las más apropiadas. La enumeración de estas opciones es parte del proceso de diseño, sin embargo a menudo esta enumeración se realiza durante la fase de especificación de requisitos, constituyendo en este caso una entrada más del ABD.

4.4.6 Actividades del ABD. Ejecución del Método.

4.4.6.1 Los pasos del método

De acuerdo con el ABD, cada elemento de diseño se caracteriza por el conjunto de requisitos que debe satisfacer (funcionales y de calidad), la plantilla asociada al mismo y una serie de restricciones de diseño. Cuando un elemento se descompone el resultado son dos o más elementos hijos, cada uno con sus propios requisitos, plantillas y restricciones. La figura 4.11 muestra la secuencia de descomposición de un elemento de diseño que, como puede observarse, comienza por la definición de la vista lógica, continúa con la definición de las vistas de concurrencia y de despliegue, prosigue con la verificación de las mismas de acuerdo con los requisitos de partida y acaba con un bucle de realimentación entre la verificación y la definición de la vista lógica. Aunque no se muestran en la figura se contempla la posibilidad de otros bucles de realimentación entre la verificación y las vistas de concurrencia y despliegue y entre las diferentes vistas entre sí. La definiciones de las diferentes vistas no son procesos aislados. Por ejemplo, al definirse la vista de concurrencia puede identificarse funcionalidad adicional que debe incorporarse a la vista lógica.

La figura 4.9 muestra el árbol de elementos de diseño que resulta después de la ejecución del método. El ABD no impone ningún orden de generación del mismo, aunque establece algunos

criterios. Por ejemplo, si el dominio está convenientemente caracterizado es más fácil recorrer el árbol en horizontal pues se tiene una idea bastante precisa de los subsistemas principales. Si se detectan riesgos que deben abordarse en etapas tempranas puede ser necesaria la implementación de un prototipo y el recorrido *en profundidad* de los subsistemas implicados.

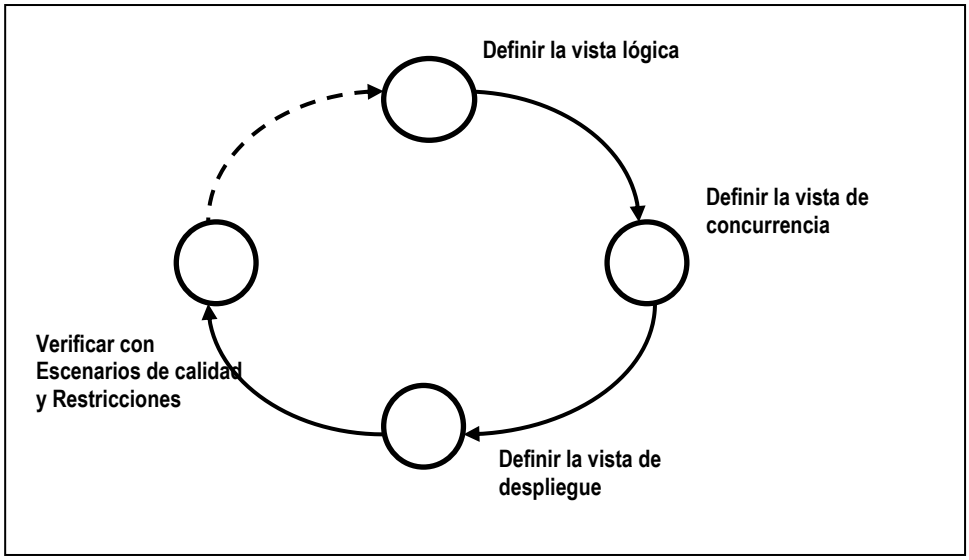


Figura 4.11: Pasos para la descomposición de un elemento de diseño

4.4.6.2 Definición de la vista lógica.

La figura 4.12 muestra los pasos a seguir para la definición de la vista lógica: descomposición de la funcionalidad, selección del estilo arquitectónico básico, asignación de funcionalidad a los componentes definidos en el estilo, refinamiento de las plantillas y verificación con casos de uso y escenarios de cambio. La división de la funcionalidad permite descomponer la misma en diferentes grupos según los criterios que dicten las directrices de la arquitectura, las cuales determinan además el estilo arquitectónico mediante el que deben organizarse los elementos resultantes de la descomposición. Dicho estilo arquitectónico define una serie de componentes a los cuales debe asignárseles cada uno de los grupos de funcionalidad resultantes de la descomposición.

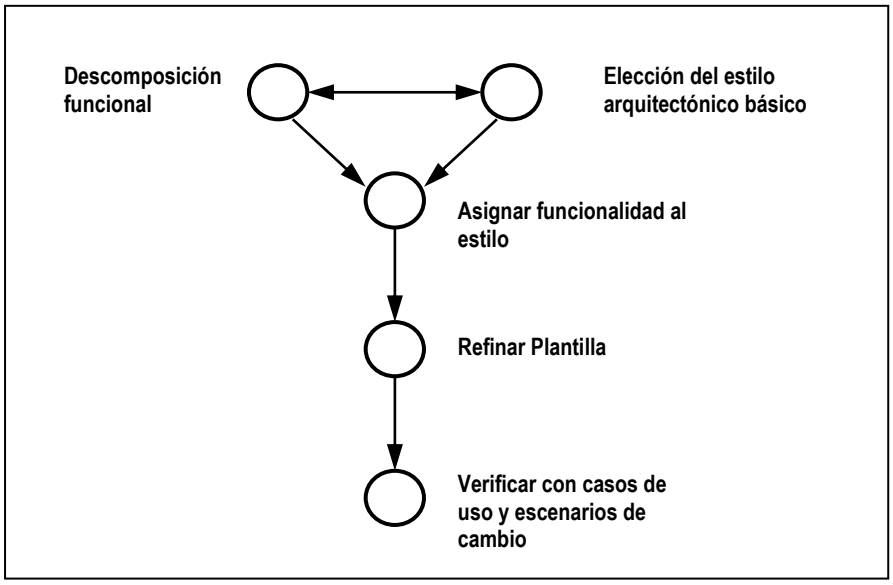


Figura 4.12: Definición de la vista lógica

1.- División de la funcionalidad.

Cada elemento de diseño tiene asignadas unas responsabilidades para satisfacer sus requisitos de diseño. Estas responsabilidades, que están constituidas por la funcionalidad que realiza el elemento y los atributos de calidad que le son propios, pueden dividirse en distintos grupos que pueden ser asignados a los elementos de diseño resultantes de su descomposición. Los criterios de división vienen determinados por los atributos de calidad que debe satisfacer el elemento de diseño.

Si la modificabilidad es el atributo más importante, los grupos de responsabilidades se establecen según los siguientes criterios:

1. Coherencia Funcional. Bajo acoplamiento y alta cohesión.
2. Aquellas responsabilidades que exhiben patrones de datos y comportamiento similares deben agruparse juntas.
3. Los diferentes grupos deben definirse en niveles de abstracción similares.
4. Localización de las responsabilidades. Aquellas responsabilidades que proporcionan servicios a otros servicios no deben ser agrupadas con las responsabilidades puramente locales.

Si el atributo clave es el rendimiento la división debe basarse en:

1. La minimización del flujo de datos entre los elementos.
2. La minimización de la frecuencia de los cálculos.

El objetivo de la descomposición es generar grupos de funcionalidad con una granularidad tal que permita:

1. Representar una descomposición de las responsabilidades asignadas al elemento de diseño.
2. Mantener el número de grupos dentro de un rango manejable. (El método aconseja entre 3 y 15).

Cada uno de los grupos generados lleva asociados:

1. Una caracterización de sus variantes y dependencias.
2. Una descripción de sus flujos de entrada y salida de datos.
3. Una relación de las entidades externas con las que interacciona.

2.- Elección del estilo arquitectónico.

Cada uno de los elementos de diseño tiene asociado un estilo arquitectónico o patrón dominante, que define la forma en que el elemento de diseño realiza sus responsabilidades. La selección de dicho estilo se basa en las directrices arquitectónicas correspondientes a dicho elemento. Una vez que el estilo arquitectónico ha sido elegido, debe adaptarse a los requisitos de calidad del elemento de diseño y a las opciones arquitectónicas[41] identificadas para satisfacer dichos requisitos. Para ello, se examina cada uno de los requisitos de calidad y se determina si tiene alguna relevancia en el elemento que va a ser descompuesto. Si es así, se elige alguna de las opciones arquitectónicas apropiadas para conseguirlos y se aplica al estilo arquitectónico seleccionado. Es aconsejable utilizar la misma opción para un mismo atributo siempre que sea posible.

El resultado de elegir y refinar un estilo arquitectónico es una colección de *tipos* de componentes. Ejemplos de tipos de componentes son clientes, servidores, dispositivos virtuales,

[41] En ocasiones esas opciones determinan el estilo dominante, en otros casos lo modulan o refinan.

etc que aunque en principio están vacíos de funcionalidad sugieren un determinado tipo de comportamiento o de interacción.

3.- Asignación de Funcionalidad al Estilo y Definición de Interfaces Conceptuales.

La elección de un determinado estilo arquitectónico implica el uso de ciertos tipos de componentes. Los grupos de funcionalidad resultantes de la descomposición de la misma deben ser asignados a los componentes determinados por el estilo. Esto implica determinar cuantas instancias deben existir de cada uno de los tipos de componentes y que funcionalidad debe incluir cada uno. Los componentes resultantes de esta asignación son los elementos de diseño en los que va a descomponerse el elemento de diseño considerado. El siguiente paso es identificar las interfaces conceptuales de cada uno de estos elementos de diseño.

Cada uno de los elementos resultantes de la descomposición tiene asociadas una colección de plantillas heredadas de sus padres en la jerarquía de elementos que define ABD. El refinado de las plantillas consiste en ir añadiéndoles responsabilidades a medida que avanza el método. Estas responsabilidades que, recordemos, se refieren a los patrones de interacción del elemento con los servicios y la infraestructura y a patrones de comportamiento comunes a diversos elementos, deben implementarse mediante componentes concretos en algún momento del proceso de diseño.

4.4.6.3 Definición de la Vista de Concurrencia.

El propósito de describir una vista de concurrencia es determinar aquellas actividades del sistema que deban ejecutarse en paralelo, con objeto de descubrir los puntos de sincronización y de acceso a recursos a compartidos.

El examen de la vista de concurrencia se realiza en términos de *threads virtuales*. Cada *thread* virtual es un camino de ejecución secuencial dentro del programa, un modelo dinámico o alguna otra representación de un flujo de control. No se consideran *threads* virtuales a aquellos definidos por el sistema operativo. Los *threads* virtuales se utilizan para describir diferentes secuencias de actividades y sus puntos de sincronización y de acceso a recursos compartidos. Estos aspectos pueden implicar la adición de nueva funcionalidad en la forma de gestores de recursos compartidos que gestionen el acceso a los mismos, que deben incorporarse a la vista lógica asignándolos a los elementos de diseño o a las plantillas correspondientes. La utilización de casos de uso que describan los efectos que producen varios usuarios accediendo simultáneamente al sistema o que examinen los efectos del arranque y parada del sistema es especialmente útil durante el desarrollo de esta vista.

4.4.6.4 Definición de la vista de despliegue.

Si el sistema incluye más de un procesador es necesario describir como pueden asignarse elementos de diseño a los mismos. Un *thread* virtual no tiene una ubicación determinada y puede trasladarse a través de una red de un procesador a otro. ABD denomina *thread físico* a aquel que se ejecuta en un procesador determinado.

4.4.7 Conclusiones y críticas al método.

El ABD está pensado para el diseño de arquitecturas software para líneas de producto y tiene en consideración la falta de especificidad de los requisitos y la variabilidad entre los diferentes sistemas de un determinado dominio. Asimismo, el ABD tiene muy en cuenta los atributos de calidad del sistema y sus interacciones, los cuales son determinantes para elegir los estilos arquitectónicos más apropiados. Los requisitos abstractos y las opciones arquitectónicas pueden expresarse usando los escenarios generales y las plantillas de mecanismo y de atributo descritas en el capítulo 3. Además, el ABD define las pautas generales para dividir el sistema en subsistemas, para asignar responsabilidades a los mismos y para identificar plantillas de comportamiento y de relación con la infraestructura aplicables a todos los subsistemas. Finalmente, como se muestra en la figura 4.10, el ABD define un proceso de diseño en el que la evaluación de la arquitectura es un paso explícitamente considerado e incluye la realización de un *mini*-proceso ATAM en cada iteración.

Sin embargo, como puede observarse en la figura 4.9, la ejecución del ABD termina una vez que se han definido los componentes conceptuales de más bajo nivel. Dichos componentes siguen teniendo un grado de generalidad muy alto. El ABD no proporciona criterios para la estructuración de los mismos en clases y tareas, ni para su despliegue en nodos físicos, ni para la traducción de sus interfaces conceptuales en interfaces concretas.

Aunque una descripción de la arquitectura al nivel de componentes conceptuales es por sí misma un artefacto útil, pues define las características generales de los elementos del sistema y sus patrones de interacción, sentando las bases para sucesivos refinamientos, no es suficiente para la implementación de un sistema, ni para definir componentes software reutilizables. Tales componentes deben ofrecer y recibir servicios a través de interfaces concretas con signaturas bien definidas. La definición de tales interfaces requiere la existencia de un modelo de componentes que defina las reglas mediante las cuales puede realizarse dicha definición. El ABD permitirá definir las líneas generales de la arquitectura. Del modelo de componentes se hablará en el capítulo 8.

4.5 Enfoque Metodológico de la tesis.

En este capítulo se han presentado con cierto detalle las metodologías de análisis y diseño utilizadas para el desarrollo de la tesis, básicamente ATAM y el ABD, enriquecidas con las *primitivas* de diseño basadas en atributos de calidad y los escenarios generales descritos en el capítulo 3.

La definición de las primitivas de diseño y los escenarios de calidad es posterior a ATAM, pero su integración en el método está descrita en [Bass et al 2001a, 2001b] por lo que su uso no presenta grandes problemas. La ausencia de los criterios para definir componentes concretos se abordará en el capítulo 8, en el que se definirán los requisitos que debe cumplir dicho modelo.

En este apartado se pretende explicar la forma en que se han aplicado el ATAM y el ABD de forma que sirva de guía en la lectura de los siguientes capítulos. En la tabla 4.1 se resumen las relaciones entre los conceptos manejados, los *artefactos*[42] utilizados para describirlos y se referencian los apartados de esta tesis donde se elaboran y explican. La figura 4.13 relaciona

[42] *Artefacto* es una traducción directa del inglés *artifact*. Un artefacto es una pieza de información utilizada o producida en un proceso de desarrollo software. Un artefacto puede ser un modelo, una descripción o un software [Rumbaugh et al 1998]

dichos artefactos con el *Proceso de Desarrollo Basado en arquitectura*. La mayor parte de estos artefactos son tablas y diagramas, que resumen las entradas y salidas de cada una de las etapas de este trabajo de tesis. Los resultados de cada etapa constituyen las entradas para la etapa siguiente. La figura 4.13 describe un proceso iterativo, con realimentaciones entre cualquiera de las etapas y etapas anteriores, en las que los artefactos se van refinando y corrigiendo progresivamente. A continuación se explican las figuras y tablas que acaban de mencionarse.

4.5.1 Ejecución del ATAM y del ABD

El ABD y ATAM parten de la caracterización de los atributos de calidad del sistema. Dicha caracterización se basa en las directrices de la arquitectura, las cuáles, como ya se ha explicado, se obtienen a partir del propósito y características de los sistemas en los que va a emplearse la arquitectura. El propósito de los sistemas lo describen sus clientes y usuarios. Las características de los sistemas se obtienen de su análisis. El modelo de análisis que se ofrece en [Alvarez 1997] es el de FODA [Kang et al 1990]. Dicho modelo captura una gran cantidad de información en diferentes diagramas (véase anexo I) y puede reutilizarse en este trabajo de tesis.

Caracterización de los atributos de calidad.

Como ya se ha visto, las entradas del ABD están constituidas por las directrices de la arquitectura, los requisitos funcionales abstractos, los atributos de calidad abstractos y (opcionalmente) una lista de los mecanismos arquitectónicos que permiten alcanzarlos. Estos elementos se presentan en los siguientes apartados:

- Las directrices del sistema se describen en el apartado 5.6.
- Los requisitos funcionales abstractos se describen en el apartado 5.8 y se resumen en la tabla 5.1.
- Los atributos abstractos se caracterizan mediante las plantillas de atributo descritas en el apartado 5.7 y en al anexo II. Dichas plantillas se resumen para facilitar su consulta en la tabla 5.2. Las plantillas de atributo incluyen sus escenarios generales y una lista de mecanismos arquitectónicos asociadas a los mismos.

ATAM, por su lado, parte también de las directrices de la arquitectura, de la caracterización de sus atributos de calidad y de una descripción de la arquitectura que permita identificar claramente los estilos y patrones utilizados en la misma. Estos elementos se presentan en los siguientes apartados:

- Los atributos de calidad se caracterizan mediante el árbol de utilidad, que se describe en el apartado 7.2 y en las tablas 7.1, 7.2 y 7.3.
- La realización de los escenarios se resumen en el apartado 7.2 y se describe con detalle en el Anexo III. Se ha preferido incluir la realización de los escenarios en un anexo para no distraer al lector con excesivos detalles.
- La arquitectura se describe en el capítulo 6. Los artefactos que se utilizan para describir la arquitectura son diagramas de UML y diversas tablas. Los diagramas muestran los aspectos más relevantes de la estructura estática y del comportamiento dinámico de los componentes de la arquitectura. El propósito de las tablas es doble. Por un lado, tal y como prescribe el ABD, permiten describir y consultar rápidamente las responsibilidades de cada subsistema (tabla 6.1) y por otro los mecanismos arquitectónicos que emplean para alcanzar sus atributos de calidad (tabla 6.2). Por otro lado, permiten mediante su comparación con las tablas 5.1 y 5.2 detectar incompletitudes e inconsistencias y facilitan el análisis de los mecanismos utilizados en la arquitectura.

Aunque el ABD y ATAM necesiten una caracterización diferente de los atributos de calidad y utilicen diferentes artefactos para describirlos, existe una transición natural entre ambas caracterizaciones y los artefactos que utilizan, ya que:

- En ambos casos, los atributos se caracterizan en función de los estímulos a los que se somete a la arquitectura y de las respuestas que ésta ofrece de acuerdo con los patrones o estilos arquitectónicos que emplea, aunque con un propósito diferente. ATAM trata de evaluar la validez de tales mecanismos en función de las respuestas obtenidas. El ABD pretende seleccionar mecanismos arquitectónicos en función de las respuestas deseadas.
- Puede definirse una relación de herencia entre los artefactos del ABD (los escenarios generales) y los de ATAM (árboles de utilidad). Los escenarios generales tal y como se definen en [Bass et al 2000] son completamente abstractos. Su empleo para definir los atributos de calidad abstractos de una arquitectura supone una primera especialización. La obtención de los escenarios de calidad concretos que constituyen las hojas del árbol de utilidad no es más que una segunda especialización, en este caso de los requisitos de calidad abstractos utilizados para el diseño de la arquitectura.

Resultados del análisis y Rediseño de la Arquitectura.

La ejecución de ATAM debe dar como resultado un conjunto de puntos sensibles, de puntos de compromiso y de riesgos arquitectónicos. Sin embargo, como se explica en el capítulo 7 la amplitud del dominio impide definir con concreción los puntos sensibles y de compromiso. Los resultados de la evaluación se exponen en el apartado 7.3.

Por último, en el capítulo 8 se propone una nueva división del sistema en subsistemas (apartado 8.4, figura 8.2) y se proponen los requisitos del modelo de componentes que debe soportar la definición de la arquitectura.

Tabla 4.1: Artefactos Descriptivos

Concepto	**Artefactos**		**Apartado**
Caracterización de los sistemas	*Propósito de los sistemas*	Descripción	5.2
	Características de los sistemas	Descripción	5.3
	Mercado Potencial y Objetivos de la Empresa	Descripción	5.4, 5.5
	Directrices de la arquitectura	Descripción	5.6
	Modelo de Análisis	Diagramas Análisis [Alvarez 1997]	Anexo I
Caracterización de Atributos	*Requisitos Funcionales Abstractos*	Tabla responsabilidades	Tabla 5.1
	Atributos de Calidad Abstractos	Descripción. Plantillas de Atributo	5.9 Anexo II
		Tablas Resumen Atributos	Tabla 5.2
	Escenarios de Análisis	Árbol de utilidad	7.2
		Escenarios ATAM	7.2 Anexo III
		Tabla resumen escenarios	Tablas 7.1, 7.2, 7.3
Arquitectura	*Descripción de la arquitectura*	Descripción + Diagramas UML	Capítulo 6
		Tabla Responsabilidad/Subsistema	Tabla 6.1
		Tablas Atributo/Subsistema/Mecanismo	Tabla 6.2
Resultados Análisis	*Resultados Análisis*	Descripción	Capítulo 7 Tablas 7.1, 7.2, 7.3

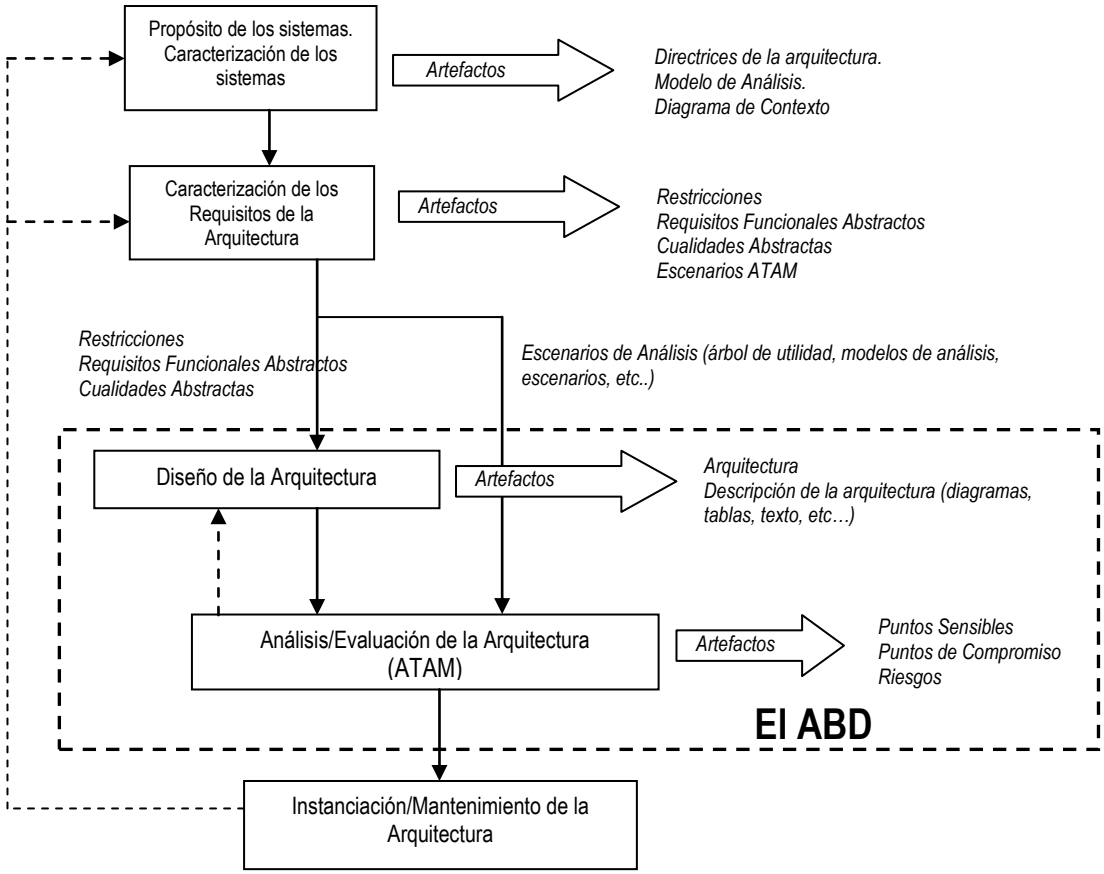


Figura 4.13: Ciclo de desarrollo.

5 Caracterización de los Sistemas de Teleoperación

5.1 Introducción.

Los sistemas de teleoperación son sistemas de control que dependen del software para realizar sus operaciones. Su diseño es una tarea muy compleja que debe integrar elementos mecánicos, eléctricos y electrónicos y componentes software en un mismo sistema. El software de control debe ejecutar diferentes algoritmos y monitorizar el funcionamiento del sistema, proporcionando determinismo lógico y temporal. Los sistemas de teleoperación son en gran medida sistemas reactivos de tiempo real en los que el software debe mostrar al menos la misma fiabilidad que el resto de los componentes. Además, la naturaleza de las misiones que desempeñan obliga a considerar aspectos de seguridad y tolerancia a fallos en distintos lugares del sistema. Tradicionalmente, sus requisitos se han centrado en la funcionalidad y el rendimiento. Sin embargo, a medida que estos sistemas se van haciendo más complejos y se van acortando los tiempos disponibles para su desarrollo, otros requisitos cobran al menos la misma importancia. Diferentes aspectos de la modificabilidad, la portabilidad y la usabilidad son igualmente relevantes tanto para el desarrollo de sistemas capaces de evolucionar como para reducir los costes y plazos de tales desarrollos. Exceptuando ciertos aspectos de la funcionalidad y la usabilidad todos estos requisitos dependen, como se ha explicado en capítulos anteriores, de la arquitectura del sistema.

El primer y más obvio paso a la hora de definir la *arquitectura de un sistema o de una familia de sistemas* es determinar aquellos requisitos que tienen una importancia determinante en el éxito o fracaso de la misma. Estos requisitos, que el ABD denomina *directrices de la arquitectura (architectural drivers)* y ATAM *directrices de negocio (business drivers)* se deducen a partir del propósito y características de los sistemas y de los factores de negocio más críticos. Obsérvese que las directrices no pretenden ser una descripción técnica precisa de los atributos de calidad, sino establecer una serie de principios a partir de los cuales dicha descripción puede llevarse a cabo. Por ello, en general, es posible identificar un número relativamente reducido de directrices.

El propósito de los sistemas y sus características sirven para acotar el dominio de sistemas considerado. La arquitectura no pretende cubrir todos los sistemas de teleoperación posibles, sino un subconjunto de los mismos. Los factores de negocio cubren una amplia gama de aspectos, que pueden clasificarse según diferentes criterios. Cualquier clasificación es buena siempre que muestre claramente [Kazman et al 1998a]:

- La necesidad, el propósito y las características de los sistemas.

- Las características del mercado potencial y el tipo de producto que demanda.
- Los objetivos de negocio que se pretenden.
- Las posibles restricciones técnicas y de negocio.

A partir de los factores de negocio, de las características y propósito de los sistemas y de las directrices que se derivan de los mismos es posible caracterizar los atributos de calidad de la arquitectura. Puesto que dichos requisitos se definen para una familia de sistemas, su grado de concreción es menor cuanto mayor sea el rango de sistemas considerados. Como se ha explicado en el capítulo anterior, esta variabilidad de los requisitos está contemplada en el ABD a través de los requisitos funcionales abstractos y de los requisitos de calidad abstractos. No obstante, se ha tratado de ser lo más preciso posible dentro de la generalidad necesaria para cubrir el dominio.

Este capítulo se organiza siguiendo las pautas descritas en los párrafos anteriores. En primer lugar se presentan el propósito y características de los sistemas considerados. Después se describen sus usos y usuarios más relevantes y los propósitos de la organización desarrolladora. A continuación se presentan las directrices que se derivan de los puntos anteriores. Y para finalizar, se caracterizan los requisitos de los sistemas.

El lector puede echar en falta en el capítulo un modelo de análisis de los sistemas y un estudio sobre las arquitecturas de teleoperación que se han empleado hasta el momento. Tanto el uno como el otro están convenientemente descritos en [Alvarez 1997] y se ofrece un resumen actualizado de los mismos en el anexo I, en el cual se han incluido algunas referencias más recientes.

5.2 Propósito de los Sistemas de Teleoperación.

5.2.1 Justificación.

Existen ciertos trabajos que más allá de cualquier otra circunstancia comparten dos características que les hacen pertenecer a una misma categoría. La primera es el riesgo que implican para quien los lleva a cabo. La segunda es la necesidad ineludible de realizarlos para evitar riesgos aún mayores. El propósito de los sistemas de teleoperación es sustituir a los operarios que realizan este tipo de actividades por mecanismos teleoperados (robots, vehículos y herramientas) que puedan ser controlados desde zonas seguras, situadas a una distancia segura del entorno de operación.

Sin embargo, si bien la primera y más importante razón que justifica el uso de sistemas teleoperados es evitar a los operarios situaciones de peligro o condiciones de trabajo penosas, no es ni mucho menos la única. Iborra en [Iborra et al 2002] enumera una serie de razones para el uso de sistemas teleoperados en centrales nucleares que son válidas para cualquier otro entorno:

1. El riesgo al que se expone a los operarios es muy alto. Incluso ciertas actividades que aisladamente consideradas suponen un riesgo bajo o moderado, se convierten en actividades de alto riesgo cuando se realizan de forma continuada. A veces el riesgo es tan alto o las condiciones de trabajo tan penosas que es imposible asegurar la integridad del personal que las realiza.

2. La cualificación del personal que realiza los trabajos *in situ* es un proceso largo y caro, implicando en algunos casos años de experiencia y costosos cursos de entrenamiento.
3. El número de intervenciones que puede realizar el personal cualificado en zonas de riesgo o exposición a sustancias tóxicas o radiaciones está limitado por diferentes legislaciones, y la tendencia es que dichas legislaciones sean cada vez más exigentes.
4. Los receptores de los servicios prefieren el uso de medios que minimicen los riesgos del personal propio y ajeno, siempre que ello no suponga incurrir en costes desproporcionados[43].
5. En las condiciones impuestas por los apartados anteriores, se hace muy difícil una planificación racional de los recursos humanos, en especial si hay que responder a situaciones de emergencia o contingencias.

Y habría que añadir al menos una más: simplemente, cierto tipo de trabajos los realizan mejor las máquinas que las personas.

5.2.2 Sistemas de Teleoperación considerados

Las consideraciones del apartado anterior son aplicables a cualquier sistema de teleoperación. Sin embargo, no se trata de considerar todo el dominio, sino un subconjunto del mismo. Para definir dicho subconjunto probablemente sea más útil renunciar a definiciones, siempre discutibles, y acotar el dominio mediante:

- Una serie de condiciones que deben exhibir los sistemas.
- Algunos ejemplos de los sistemas considerados.
- Algunos ejemplos de los sistemas no considerados.

Así, los sistemas que se consideran comparten en mayor o menor medida las siguientes características:

1. Realizan una actividad o un pequeño conjunto de actividades en un entorno determinado. Las actividades realizadas y el entorno en el que se realizan determinan sus características morfológicas y su complejidad.

   Quedan en principio excluidos sistemas de propósito general o adaptables a entornos muy diferentes.

   Ejemplo: Sistema Robotizado de Mantenimiento de Generadores de Vapor.

   Actividades: Taponado (mecánico y soldadura).
   Taladrado.
   Inspección de tubos.

   Ejemplo: Sistema Robotizado para el acceso a los internos inferiores de vasijas de agua a presión.

   Actividades: Inspección visual mediante cámaras.
   Recogida de pequeños objetos (diámetro < 80 mm, peso < 500 g).

[43] En el sector nuclear existe el principio ALARA respecto de las dosis recibidas por los operarios: *"As Less As **Reasonably** Possible",* Tan poco como **razonablemente** posible.

Contraejemplo: Robot escalador de superficies casi-verticales de diferentes características capaz de realizar diversas actividades (definidas y a definir).

2. Las actividades suelen ser trabajos de mantenimiento o inspección habituales en la industria, que se realizan según procedimientos perfectamente definidos y poco sujetos a circunstancias excepcionales. En general, aunque no siempre, son actividades realizadas anteriormente de forma manual por operarios humanos. Sus características no permiten una completa automatización mediante robots industriales.

   Ejemplo: Sistema Robotizado para la limpieza de finos de embarcaciones

   Contraejemplo: Vehículo submarino para la localización y desactivación de minas.

3. Las características del entorno, estructurado o no, son perfectamente conocidas y no están sujetas a grandes modificaciones.

   Ejemplo: Vehículo sumergible dotado de garra o cazoleta para la recogida de objetos de las toberas de primario en las proximidades del generador de vapor.

   Contraejemplo: Vehículo para la limpieza de toberas en presencia de agua o no, depósitos de combustible, superficies al aire libre, etc[44]

4. El grado de autonomía es limitado, los mecanismos no tienen que tomar decisiones complejas, pero pueden definirse ciertas tareas y acciones automáticas que implican un comportamiento reactivo frente a cambios en el entorno. Estos cambios pueden ser producidos por la propia actividad del sistema.

   Ejemplo: Sistema Teleoperado de soldadura: brazo + herramienta sensorizada.

   Soldadura Automática: Aporte de material y avance de la herramienta durante una operación de soldadura.

   Ejemplo: Sistema Teleoperado de limpieza de superficies: grúa posicionadora + cabezal sensorizado.

   Limpieza automática de un paño: Avance de cabezal de limpieza en función de la calidad del acabado (inspección en línea).

   Detección de obstáculos (protuberancias, vigas, oquedades, etc.) y parada automática.

   Ejemplo: Detección de colisiones y detención automática de los mecanismos.

   Contraejemplo: Vehículo capaz de orientarse en el entorno, detectar obstáculos y evitarlos sin ayuda del operador (el Mars Rovers).

Aunque puede haber sistemas que se sitúen en una franja fronteriza en la que no sea fácil determinar si pertenecen o no al dominio considerado, las condiciones y ejemplos de los párrafos anteriores dan una idea del tipo de sistemas para los que se define la arquitectura y de su grado de complejidad. En el siguiente apartado se tratará de aportar características adicionales que ayuden a acotar el dominio con mayor precisión.

[44] Es posible que un vehículo pensado para una actividad específica en un entorno específico pueda ser reutilizado para otras actividades parecidas en entornos muy diferentes, pero la clave del ejemplo es que para su definición y diseño no se considera esa posibilidad.

5.3 Características de los Sistemas de Teleoperación del dominio considerado.

A pesar de las condiciones definidas en el apartado anterior, la amplitud del dominio sigue siendo muy grande y en consecuencia también lo es el número de características que hay que tener en cuenta. Sin embargo, las realmente relevantes para la definición de la arquitectura son las que se enumeran a continuación:

- **Requisitos de seguridad por lo general muy estrictos**, pues suelen estar en juego la integridad de personas e instalaciones.

- **Alto grado de especialización**. Cada sistema se diseña para realizar una actividad determinada en un entorno muy concreto. Aunque en ocasiones es posible utilizar vehículos, robots o herramientas comerciales, son más frecuentes los diseños específicos, perfectamente adaptados al entorno de trabajo y especializados en la realización de tareas muy concretas.

- **Gran dispersión de características de los mecanismos teleoperados**. Como consecuencia de la especialización, la complejidad y características de los vehículos, brazos y herramientas utilizados varían enormemente de unas aplicaciones a otras. Pueden encontrarse desde dispositivos muy sencillos que podrían operarse con una botonera, hasta mecanismos extraordinariamente complejos y sofisticados.

- **Gran variedad de plataformas**, consecuencia también de la especialización.

- **Distintas configuraciones de mecanismos**. El mecanismo o mecanismos a teleoperar suelen ser una herramienta, un brazo con una herramienta ensamblada en su extremo o un vehículo provisto de un brazo dotado con la correspondiente herramienta. Otras configuraciones son más raras, pero no imposibles: vehículos sin ningún brazo o herramienta, utilizados exclusivamente para labores de inspección, vehículos que portan más de un brazo, etc.

- **Gran variedad de interfaces de usuario**, adaptadas a la características de los mecanismos, a la naturaleza del trabajo a realizar y a las preferencias de los operadores. La mayoría de los sistemas incluyen interfaces gráficas, que en ocasiones se complementan con *joysticks* más o menos sofisticados[45]. La realimentación de pares y fuerzas hacia el operador empieza a tener relevancia en algunas actividades muy concretas y *todavía* minoritarias. En cualquier caso la interfaz debe ofrecer al operador una imagen clara y consistente del estado del entorno y de los dispositivos, para que éste pueda supervisar las operaciones[46].

- **Diferentes grados de autonomía y automatismos.** Habitualmente el operador ordena cada una de las acciones que deben realizar los mecanismos, limitándose las acciones automáticas a aquellas que tienen que ver con la detección de fallos y la parada segura. Pero no siempre es así, en ocasiones es posible planificar una determinada secuencia de

[45] En este tipo de sistema no es probable encontrarse con interfaces de usuario que sean reproducciones a escala de los mecanismos a operar.

[46] El uso de cámaras es casi siempre imprescindible, pero salvo que el sistema incorpore utilidades de visión artificial se considera estas no pertenecen al sistema pues no están bajo el control de la aplicación de teleoperación ni le suministran ningún dato.

operaciones, que los dispositivos ejecutan de una vez, limitándose el operador a iniciar y supervisar el proceso.

- **Diferentes patrones de reactividad.** Como consecuencia del punto anterior, algunos sistemas deben ser capaces de reaccionar ante cambios en el entorno o como consecuencia del desarrollo de las actividades que realizan. En otros el comportamiento viene dictado casi exclusivamente por las órdenes del operador.

- **Carácter distribuido.** En la mayoría de las aplicaciones hay al menos dos nodos, uno correspondiente a la plataforma de teleoperación en el que se ejecuta la interfaz de usuario y los procesos de control de más alto nivel y otro local a los dispositivos, en ocasiones embarcado en los mismos, que suele ocuparse de las tareas de control de bajo nivel (sensores y actuadores). Sin embargo, a veces todos los procesos se ejecutan en un mismo nodo.

- **Presencia de enlaces de comunicaciones**. Como consecuencia del punto anterior, casi siempre existe un enlace de comunicaciones, que varía mucho de unas aplicaciones a otras (Ethernet, CAN-bus, diversos buses de campo, más raramente comunicaciones inalámbricas, etc.)

- **Requisitos de tiempo real,** mucho más estrictos cuanto más bajo es el nivel de los procesos de control. Aunque es más probable que estos requisitos afecten a los procesos del nodo local, pueden aparecer en cualquier lugar dependiendo de las características del sistema. Piénsese por ejemplo en alguna de las interfaces de usuario enumeradas más arriba o en la utilidades que se mencionan en el siguiente párrafo.

- **Incorporación de herramientas software muy diversas** como utilidades de simulación, que en ocasiones se utilizan *on-line* y otras únicamente para el entrenamiento, planificación o test previo de las operaciones, sistemas de navegación, mapas del entorno, visión artificial, etc.

- **Vida operativa muy variable.** Algunos sistemas están pensados para trabajar durante años, en otros casos se definen para operaciones muy puntuales impuestas por algún tipo de contingencia.

5.4 Mercado Potencial. Actividades Demandadas.

Uno de los factores de negocio más importantes es el mercado al que está destinado el producto. El mercado potencial de los sistemas de teleoperación considerados está constituido por:

- Empresas u organizaciones que de forma regular o esporádica deben realizar actividades que suponen riesgos para los operarios (centrales nucleares, plantas químicas, refinerías de petróleo, plataformas petrolíferas, astilleros, policías y cuerpos de seguridad del estado, etc).

  Es bastante habitual que se trate de grandes empresas, que subcontratan la mayor parte de los trabajos de mantenimiento a empresas de servicios, reservándose únicamente el control de calidad. Si participan en algún proyecto relacionado con un sistema de teleoperación utilizable en sus instalaciones su contribución suele limitarse a la especificación de requisitos.

- Empresas de servicios a las cuáles se subcontratan la mayoría de los trabajos y que son, por las razones ya explicadas, las primeras interesadas en sustituir a sus operarios por mecanismos teleoperados. Es frecuente que estas empresas aborden por sí mismas el diseño de sistemas de teleoperación si cuentan con el nivel tecnológico suficiente.

Los dos párrafos anteriores presentan un escenario general que conoce, no obstante, bastantes excepciones. Los trabajos de mantenimiento de instalaciones no son las únicas aplicaciones posibles de los sistemas de teleoperación, pero constituyen lo que podríamos llamar el grueso del negocio y en ellas no centraremos en los párrafos siguientes.

La clasificación de los trabajos que demandan las empresas según su planificación permite razonar acerca del tiempo disponible para el desarrollo de los sistemas y de su impacto económico, en términos de coste/beneficio. Según esta clasificación las actividades pueden dividirse en:

- Actividades rutinarias, previamente planificadas y ejecutadas habitualmente de forma periódica.
- Actividades esporádicas, no planificadas pero previsibles.
- Actividades imprevistas.

Las actividades rutinarias y previamente planificadas proporcionan a las empresas de servicios su fuente regular de ingresos. Su forma de ejecución está por lo general descrita en procedimientos de obligado cumplimiento definidos por la empresa suministradora del servicio, la propiedad y el fabricante de los equipos. La mayor parte de las operaciones de inspección y mantenimiento de las centrales nucleares y plantas químicas caen dentro de esta categoría. Dada la frecuencia con la que se realizan este tipo de actividades, la sustitución de operarios por máquinas incrementa espectacularmente la seguridad de las operaciones, disminuye en gran medida las necesidades de personal cualificado[47] y favorece una planificación racional de los recursos humanos.

Las actividades espóradicas, no previamente planificadas, pero previsibles, responden al suceso de incidencias durante las operaciones de mantenimiento planificadas o durante el régimen de funcionamiento de las instalaciones. La propiedad suele exigir la definición de medidas de contingencia para este tipo de incidentes si estima que la probabilidad de que se produzcan es elevada, si han ocurrido en el pasado, aun cuando sean muy poco probables, si existe alguna normativa o recomendación al respecto o si alguna empresa las ofrece como parte de sus servicios. Una vez que el cliente exige tales medidas, éstas pasan a formar parte del conjunto de actividades rutinarias, tanto si se realizan como si no. La empresa que esté en condiciones de ofrecer las mejores medidas de contingencia está en una posición ventajosa respecto de las demás.

Las actividades imprevistas responden al suceso de incidencias y accidentes que o bien son imprevisibles o bien su probabilidad de ocurrencia es tan pequeña que no se implementan métodos de contingencia para los mismos. Cuando ocurren es necesaria una gran capacidad de reacción. La resolución de este tipo de incidentes se realiza sobre la marcha, aprovechando los recursos existentes o implementando unos nuevos a toda prisa. La empresa que sea capaz de dar una solución a corto plazo se coloca en una situación privilegiada respecto de las demás y respecto de la propiedad.

Otro aspecto a considerar es el perfil de los operadores de los sistemas de teleoperación. Contrariamente a lo que pueda pensarse no suele ser personal con un gran bagaje técnico, al menos no en lo que al software y al uso de ordenadores se refiere. La mayoría de las empresas

[47] El personal debe ser igualmente cualificado, pero no se *quema* en unas pocas intervenciones, pues realiza las mismas desde zonas seguras.

no contratan a personal especializado, sino que reciclan al personal que realiza las operaciones manualmente. Este tipo de usuario demanda una interfaz fácil de usar, que se adapte en la medida de lo posible a sus habilidades y conocimientos y que le asista en la ejecución de las operaciones.

5.5 Objetivos de la empresa.

En este caso la organización que define la arquitectura no es una empresa, sino una institución de enseñanza e investigación, la Universidad Politécnica de Cartagena, a través del Grupo de Investigación DSIE, que engloba personal de los Departamentos de Automática e Ingeniería de Sistemas, Tecnología Electrónica y Tecnologías de la Información y la Comunicación.

El producto final no es un sistema de teleoperación concreto, sino una arquitectura de referencia. Los objetivos que se pretenden alcanzar con el desarrollo de tal arquitectura se inscriben dentro de la línea de investigación iniciada por Álvarez [Álvarez 1997] y suponen una continuación de la misma.

Tales objetivos pueden resumirse en los siguientes puntos:

1. Validación de la arquitectura descrita en [Álvarez 1997] para comprobar su grado de adecuación a los nuevos requisitos. Como se ha explicado en capítulos anteriores, la arquitectura propuesta por Álvarez se refiere a un dominio mucho más concreto. Se trata de determinar si es posible su uso o adaptación en el dominio que se describe en este capítulo.
2. Obtención de una nueva arquitectura que tenga en cuenta:
   - ✓ Tanto los requisitos de calidad ya contemplados en [Álvarez 1997] como los que se añaden en este capítulo.
   - ✓ Una caracterización precisa de los atributos de calidad.
   - ✓ Los avances que sobre notaciones formales y metodologías de diseño y evaluación de arquitecturas se han producido en estos últimos años.

   Y que permita en la medida de lo posible:

   - ✓ La definición de arquitecturas específicas para sistemas concretos.
   - ✓ El desarrollo rápido de aplicaciones.
   - ✓ La generación de un conjunto de componentes software reutilizables en las diferentes aplicaciones de la familia.
   - ✓ La integración de componentes hardware y software comerciales.
3. Formar a personal investigador en el uso de técnicas formales o semi-formales de descripción, diseño y evaluación de arquitecturas.
4. Crear un marco en el que puedan llevarse a cabo líneas de investigación más especializadas, relacionadas con los subsistemas descritos en la arquitectura o con dominios de teleoperación más reducidos.
5. Establecer lazos de cooperación con la industria e impulsar líneas de investigación que puedan aplicarse en la misma a través de la correspondiente transferencia de tecnología.

5.6 Directrices de la Arquitectura.

La definición de las directrices de la arquitectura a partir de los factores de negocio es un proceso iterativo. Algunas directrices se expresan de forma explícita en los factores de negocio o son tan evidentes que pueden deducirse directamente. Así ocurre en nuestro caso con la modificabilidad y la seguridad. Otras requieren de un análisis cuidadoso de los factores de negocio y de las características de los sistemas. Durante dicho análisis se descubren otros factores que dan lugar a la definición de nuevas directrices y así sucesivamente. Para la arquitectura objeto de este trabajo de tesis se proponen las siguientes directrices, clasificadas en 6 categorías, según el atributo de calidad al que conciernen.

1. Seguridad.

   Los sistemas de teleoperación se emplean en zonas de riesgo o para trabajos de riesgo y esto implica el uso de sistemas seguros que no provoquen daños a los operarios ni al entorno.

   - *Las operaciones sólo deben ejecutarse si existe una confianza razonable en que el resultado de las mismas no va a poner en peligro la integridad de bienes y equipos.*
   - *Los sistemas deben incorporar mecanismos de parada segura que se activen por orden del operador o de forma autómatica en respuesta a ciertos eventos.*
   - *Debe proporcionarse al operador información fiable respecto del estado del entorno y de los dispositivos.*
   - *Deben definirse alarmas, mensajes y advertencias que prevengan al operador de cualquier mal funcionamiento del sistema. Si los errores son graves debe activarse automáticamente los mecanismos de parada segura.*

2. Modificabilidad.

   - *La arquitectura debe ser válida para el mayor número posible de sistemas de teleoperación que cumplan las características descritas en los apartados anteriores.*
   - *Debe, asimismo, constituir un marco adecuado para la definición de arquitecturas más específicas y para el desarrollo de componentes software reutilizables*
   - *Los sistemas deben poder ejecutarse sobre diferentes sistemas operativos y plataformas (portabilidad).*
   - *El software debe adaptarse con facilidad al empleo de nuevos mecanismos (brazos, vehículos y herramientas) o a la modificación de los existentes.*
   - *Los sistemas deben incorporar fácilmente nuevas misiones y permitir cambios en las ya existentes.*
   - *La arquitectura debe contemplar la posibilidad de usar componentes comerciales cuando se estime que ofrecen servicios interesantes para el sistema, bien directamente (herramientas software, middleware), bien indirectamente (a través de bibliotecas software).*

3. Rendimiento.

   - *Tanto los comandos como la información de estado deben procesarse dentro de unos plazos que garanticen el correcto funcionamiento de los dispositivos y la validez de la información que se ofrece al operador y que manejan los diferentes subsistemas.*

- *Si se utilizan enlaces de comunicaciones, éstos no deben suponer una pérdida de rendimiento que impida el cumplimiento del punto anterior.*

4. Fiabilidad/Disponibilidad.
   - *Posibilidad de funcionamiento básico degradado si falla algún sistema, al menos el indispensable para ejecutar la parada segura y rearrancar el sistema.*
   - *Si se utilizan enlaces de comunicaciones, los fallos en las mismas deben ser detectados y si es posible corregidos. Debe contemplarse la posibilidad de utilizar redundancia en las comunicaciones.*
   - *Mecanismos de fallo-seguro.*

5. Usabilidad.
   - *La interfaz de usuario debe poder adaptarse a las necesidades, habilidades y prácticas comunes de los operadores.*
   - *Debe existir la posibilidad de incluir representaciones gráficas del entorno y de los mecanismos teleoperados y de incorporar utilidades de visión artificial y/o sistemas de navegación más o menos complejos.*
   - *Asimismo debe existir la posibilidad de incorporar utilidades de ayuda en línea que asistan al operador durante el transcurso de las operaciones.*
   - *Los comandos inseguros deben inhabilitarse y los mecanismos de parada segura deben estar siempre disponibles y ser fácilmente identificables y accesibles por el operador.*

6. Interoperabilidad.
   - *Posibilidad de repartir las diferentes actividades de una misión entre diferentes sistemas de teleoperación. O dicho de otra manera, posibilidad de descomponer una misión en misiones más simples asignando cada una de ellas a un sistema de teleoperación determinado.*

5.7 Caracterización de las Responsabilidades y Requisitos de Calidad Abstractos.

Como se explicó en el capítulo 4, los requisitos de una arquitectura de referencia rara vez pueden expresarse concretamente. La especificación de requisitos en un alto nivel de abstracción se hace necesaria por dos razones:

1. La necesidad de capturar las variantes de funcionalidad entre todos los sistemas de la familia.
2. La imposibilidad de conocer con antelación los requisitos específicos de cada sistema concreto.

Pero aún en este nivel de abstracción es necesario realizar una caracterización de los atributos tan precisa como sea posible. El rango en el que se mueven las variantes debe describirse

siempre que se conozca y es aconsejable[48] enumerar los mecanismos[49] arquitectónicos más adecuados para la obtención de los atributos de calidad.

El enfoque de este apartado es el siguiente: partiendo de las directrices definidas en el apartado anterior y de las características de los sistemas de teleoperación se desglosará cada atributo general en sus diferentes aspectos. Cada uno de estos aspectos puede ser considerado como un atributo de calidad *abstracto* que debe caracterizarse en la medida de lo posible. Para ello, se utilizará la plantilla mostrada en la figura 5.1, inspirada en la propuesta en [Bass et al 2000] para la caracterización de atributos de calidad completamente abstractos (no ligados a ningún sistema). El propósito de cada una de las entradas es el siguiente:

- La descripción permite definir en unas pocas líneas el aspecto del atributo de calidad que se considera.
- Los escenarios generales permiten dotar de contenido especifico a los atributos de calidad, y serán muy útiles a la hora de generar el árbol de utilidad y los escenarios específicos que emplea ATAM.
- Los mecanismos definen los patrones y estilos arquitectónicos más apropiados para satisfacer el atributo de calidad. Su enumeración explícita facilitará la identificación de aquellos que han sido utilizados en la arquitectura a evaluar y su comparación con otras alternativas. La enumeración de estos mecanismos es además una de las entradas del ABD.
- Las relaciones con otros atributos permite razonar sobre las influencias mutuas e identificar con ayuda de la segunda y tercera entradas de la tabla mecanismos arquitectónicos compatibles o contradictorios.

[48] La identificación de mecanismos arquitectónicos es realmente parte del proceso de diseño, sin embargo el ABD recomienda enumerarlos al mismo tiempo que se especifican los requisitos. Esta enumeración no compromete ninguna decisión, pero puede ayudar a tomarla.

[49] Mecanismo: Estructura por medio de la cual los componentes colaboran para proporcionar un comportamiento que satisfaga los requisitos del problema [Booch 1994].

Número Atributo: **Nombre del atributo**

Descripción general.
Se describe en unas pocas líneas el propósito del atributo.

Escenarios Abstractos.
Se definen escenarios abstractos que ayuden a determinar el alcance de los aspectos descritos en el punto anterior.

Los escenarios deben describirse en términos de los estimulos proporcionados al sistema y de las respuestas que debe proporcionar el mismo. Si existen modelos formales asociados al atributo, es aconsejable que dichos estímulos y respuestas se correspondan con los parámetros de entrada y salida de alguno de ellos. Esta forma de proceder facilita la selección de estrategias y mecanismos arquitecturales y el posterior análisis de la arquitectura.

Estrategias y Mecanismos arquitecturales.
En este apartado se describen estrategias y mecanismos arquitecturales que pueden emplearse para la realización de los escenarios generales.

Las estrategias se refieren a paradigmas y técnicas de programación generales, mientras que los mecanismos apuntan a soluciones más concretas. Por ejemplo, estrategias son el encapsulado, la separación de conceptos o proponer el uso de alguna técnica de asignación de recursos. Los Mecanismos definen patrones arquitectónicos básicos (adaptador[Gamma et al 19995]) o complejos (cliente/servidor[Shaw 1996]) y describen técnicas concretas para implementar las estrategias arquitecturales (planificación mediante prioridades fijas, ejecutivo cíclico, etc).

Relación con otras directrices.
Se enumeran las directrices con las que se relaciona y se explica brevemente el motivo de la relación (interacciona con..a través de…, mecanismos incompatibles con.., se deriva de…, condiciona la…, favore a…, es favorecido por…, etc)

Relación con directrices de la arquitectura.
Se explica brevemente como se deriva el requisito de los factores de negocio y del propósito y características de los sistemas.

Observaciones.
Cualquier información que se estime relevante no incluida en los apartados anteriores.

Figura 5.1: Plantillas de atributo.

- La relación con las directrices justifica la definición del atributo.
- Las observaciones permiten incluir razonamientos y detalles que o bien no encajan en ninguno de los otros apartados o bien son aplicables a varios de ellos.

La elaboración de estas plantillas debe proporcionar:

- Una organización jerárquica de los requisitos. En el primer nivel, los requisitos de las plantillas son completamente abstractos. A medida que se avanza en profundidad la especificación se va haciendo más concreta. Las hojas del árbol se corresponden con los requisitos de sistemas específicos.

- Una base de conocimiento para la selección de estilos arquitectónicos y patrones de diseño. Este extremo se consigue mediante la inclusión en las plantillas de los mecanismos de diseño candidatos para la obtención de un atributo.

La información incluida en las plantillas es muy extensa, por lo que deben proporcionarse utilidades que permitan *navegar* a través del árbol de requisitos. Una primera aproximación puede ser definir las plantillas mediante hipertexto. Más adelante puede ser necesario el empleo de herramientas que permitan mantener las plantillas y ponerlas en relación con entornos de diseño.

Con objeto de no agobiar al lector con excesivos detalles, las plantillas elaboradas se incluyen en el anexo II, comentándose sus aspectos más relevantes en los siguientes apartados e incluyéndose un resumen de las mismas en la tabla 5.3.

De la misma manera, las responsabilidades funcionales de los sistemas se resumen en la tabla 5.2. Nuevamente, esta tabla se corresponde con las responsabilidades abstractas, que deberán sufrir refinamientos sucesivos hasta dar lugar a requisitos funcionales concretos aplicables a sistemas específicos.

5.8 Atributos de Calidad Abstractos

Siguiendo el esquema propuesto en [Bass et al 2000] los atributos de calidad abstractas se clasifican en las siguientes categorías:

- Aspectos de la Modificabilidad.
- Aspectos del Rendimiento.
- Aspectos de la Seguridad.
- Aspectos de la Fiabilidad/Disponibilidad.
- Aspectos de la Usabilidad.
- Aspectos de la Interoperabilidad.

5.8.1 Aspectos de la Modificabilidad.

Los diferentes aspectos de la modificabilidad se refieren a la portabilidad, a la capacidad de distribución, a la integración de servicios y utilidades, a la adaptabilidad frente a cambios en las interfaces de usuario y en las unidades de control local, a las modificaciones relacionadas con cambios en el entorno operativo, en los mecanismos y en las misiones y finalmente, con la posibilidad de integrar herramientas, bibliotecas y componentes software comerciales.

La portabilidad se refiere a la necesidad de los sistemas a adaptarse a diferentes infraestructuras, entendiéndose éstas como los servicios que proporcionan el sistema operativo, los enlaces de comunicaciones y el *middleware*. La independencia de la infraestructura es una directriz de diseño que se deriva directamente de las características de los sistemas. Aun cuando la portabilidad puede no ser un requisito de todos sistemas del dominio, diferentes sistemas pueden requerir diferentes infraestructuras. La necesidad de utilizar hardware de control específico, que será diferente en cada sistema, refuerza aún más este requisito. La definición de

una interfaz abstracta de acceso a la infraestructura y la adopción de estándares son claves para lograr la portabilidad.

La capacidad de distribución se refiere a la posibilidad de asignar los diferentes subsistemas a diferentes nodos del sistema. La capacidad de distribución viene motivada por el carácter distribuido de los sistemas, que habitualmente constan de dos o más nodos, por la necesidad de ubicar las tareas más críticas o más exigentes en términos de computación en nodos específicos y de definir estrategias de recuperación y de tolerancia a fallos (si un nodo cae sus responsabilidades pueden ser asumidas por otro). La capacidad de distribución exige ser muy cuidadoso en la división de la funcionalidad y en la definición de los patrones de interacción entre susbsistemas que pueden ejecutarse en diferentes nodos (sólo debe admitirse el paso de mensajes).

La integración de servicios o utilidades es una directriz de diseño. Es muy habitual que los sistemas de teleoperación necesiten utilidades de simulación, representación gráfica, visión artificial, navegación, realimentación de pares y fuerzas, etc. Estos servicios pueden emplearse en línea proporcionando información al resto de los subsistemas o fuera de línea para actividades de entrenamiento o de definición de misiones. Las aplicaciones más sencillas sólo demandan alguna de ellas (a veces ninguna), las más complejas pueden necesitar de muchas.

La integración de servicios está estrechamente relacionada con la posibilidad de integrar hardware y software comercial en los sistemas, en especial para implementar servicios de representación gráfica, de cálculos cinemáticos y de navegación.

La adaptabilidad a distintas interfaces de usuario es también una directriz de diseño. Los sistemas deben adaptarse a interfaces de usuario muy diferentes, que incluyen dispositivos específicos como *joysticks*, botoneras, reproducciones a escala de los dispositivos, utilidades de visión artificial, etc. y con muy distintas necesidades de representación gráfica y disposición de las ventanas. Las interfaces de usuario son uno de los componentes del sistema más sujetos a sufrir variaciones tanto entre sistemas como durante la vida operativa de un sistema. La separación de la interfaz de usuario del resto de la aplicación es, por tanto, fundamental. La adaptabilidad a distintas interfaces de usuario puede considerarse un caso particular de la adaptación a nuevos servicios y está estrechamente relacionada con la posibilidad de utilizar herramientas y componentes software comerciales.

La adaptabilidad a diferentes entornos operativos puede considerarse desde dos puntos de vista:

- Posibilidad de construir aplicaciones que operan en diferentes entornos.
- Posibilidad de adaptar un sistema dado para que pueda trabajar en diferentes entornos.

Puesto que la adaptación a diferentes entornos se basa (ver estrategias) en el uso de ciertos servicios, este atributo puede verse como un caso particular de la incorporación de nuevos servicios, y como aquel está relacionado con la posibilidad de utilizar herramientas y componentes software comerciales.

En el segundo caso la posibilidad de adaptar al sistema para trabajar en diferentes entornos está estrechamente relacionada con la posibilidad de adaptarlo a diferentes mecanismos y a diferentes misiones. Cada mecanismo tiene unas propiedades (mecánicas, eléctricas, estructurales, etc.) que le permiten realizar ciertas actividades en ciertos entornos. Si el cambio entre entornos implica a dos entornos estructurados normalmente es suficiente con disponer de

utilidades de modelado y representación gráfica. Si el entorno cambia, basta con modificar el modelo. En entornos no estructurados la utilidad de estas herramientas es limitada y puede ser necesaria la incorporación de un sistema de visión artificial u otras utilidades que permitan al sistema recoger información de su entorno y actuar en consecuencia.

La adaptabilidad a diferentes misiones está relacionada con:

- La especialización de los sistemas: Cada sistema debe realizar un conjunto de misiones, en general muy específicas.
- La mantenibilidad del sistema: La ejecución de una misión determinada está sujeta a modificaciones. Durante la vida de un sistema es posible que surjan nuevas actividades a realizar.

Una misión es la ejecución por los mecanismos de una serie de actividades, realizadas en secuencia o según la lógica impuesta por una máquina de estados. En ocasiones una misión compleja no es más que la composición de otras más simples. Una misión es sobre todo una *pieza de funcionalidad*, que implica la ejecución de ciertos procedimientos y el acceso a diferentes servicios. Es necesario separar los aspectos ligados a la misión del resto.

Puesto que las misiones se realizan en un determinado entorno y utilizando determinados mecanismos, la adaptabilidad a nuevas misiones está ligada a la adaptabilidad a diferentes entornos, mecanismos y configuraciones (combinaciones vehículo-brazo-herramienta). En ocasiones la misión sólo implica a un mecanismo (realización de un proceso de herramienta o ejecución de un movimiento o una secuencia de movimientos). En otras implica a varios. Para que varios mecanismos colaboren en una misión es necesario que sincronicen sus actividades e intercambien mensajes. Esto revela una nueva dimensión de la adaptabilidad a nuevas misiones: la necesidad de establecer medios de comunicación flexibles entre los controladores de los mecanismos.

Puesto que la realización de ciertas misiones necesita de ciertos servicios, la adaptabilidad a nuevas misiones está relacionada con la incorporación de nuevos servicios, y a través de ésta con la posibilidad de utilizar herramientas y componentes software comerciales. Aún más: A menudo las misiones deben seguir un procedimiento más o menos estricto que debe ser seguido por el operador. La aplicación debe guiar la ejecución de las misiones, razón por la cual la adaptabilidad a nuevas misiones está relacionada con la adaptabilidad a nuevas interfaces y a la usabilidad.

La adaptabilidad a diferentes mecanismos (brazos, vehículos y herramientas) viene motivada por:

- La especialización de los sistemas: Cada sistema controla unos mecanismos específicos.
- La posibilidad de que un mismo sistema realice diferentes trabajos. Para ello pueden ser necesarias diferentes combinaciones de mecanismos (p.e: un mismo brazo robotizado debe portar diferentes herramientas según el tipo de operación que debe realizar).

La adición, sustitución o modificación de los mecanismos que constituyen el sistema es una actividad bastante habitual, que a menudo debe realizarse con el sistema en funcionamiento. El logro de este requisito implica un modelado muy cuidadoso de las características estructurales y dinámicas de los mecanismos, la definición de interfaces abstractas de acceso a los mismos y la definición de controladores genéricos. La mayor parte de estas actividades se realizan en [Alvarez 1997]

Finalmente, **la posibilidad de integrar componentes o herramientas software comerciales** viene motivada por la necesidad de cumplir el resto de los requisitos en un plazo de tiempo razonable. Debe existir la posibilidad de usar componentes comerciales que ofrezcan servicios interesantes para el sistema, bien directamente (herramientas software), bien indirectamente (a través de bibliotecas software). Aunque puede considerarse un caso particular de la integración de servicios, presenta sin embargo diferencias importantes:

- Los COTS pueden utilizarse para proporcionar dichos servicios directamente o para implementarlos.
- El uso de COTS tiene su problemática propia:
  - ✓ Compatibilidad entre diferentes versiones.
  - ✓ Compatibilidad entre COTS.
  - ✓ Suposiciones arquitectónicas de los COTS.
  - ✓ Sustitución de unos COTS por otros.

5.8.2 Aspectos del Rendimiento.

Dado lo amplio del dominio, el rendimiento sólo puede caracterizarse con un muy alto grado de abstracción, ya que se desconocen los eventos concretos a los que deben responder los diferentes sistemas, así como los plazos en que los mismos deben ser procesados. No obstante, es posible caracterizar algunos aspectos generales del comportamiento temporal que podrán ser refinados más adelante. Desde el punto de vista arquitectónico conviene clasificar las tareas según su nivel de criticidad, el tipo de eventos a los que deben responder y la forma en que el sistema debe responder a dichos eventos. En general, aunque en los sistemas de teleoperación pueden existir tareas que respondan a cualquier combinación de criticidad, tipo de eventos y características de la respuesta, es posible distinguir dos categorías principales:

- **Tareas críticas** que deben realizarse dentro de unos plazos estrictos. Suelen estar relacionadas con procesos de control de los mecanismos (vehículo-brazo-herramienta) cercanos al hardware, tales como el control del bucle de realimentación de los servos, la lectura de sensores y el accionamiento de los actuadores y procesos o servicios relacionados con la parada segura del sistema.
- **Tareas menos críticas.** Son en general procesos de control de alto nivel no directamente relacionados con el funcionamiento seguro del sistema. La planificación y secuenciación de las misiones, la captura de datos que pueden procesarse en diferido y dicho procesamiento, ciertos procesos de diagnosis, la actualización de la interfaz de usuario, el suministro de ciertos servicios, etc. entran habitualmente dentro de esta categoría.

El rendimiento interacciona con el resto de los atributos de calidad, normalmente entrando en conflicto con ellos como se muestra en la tabla 5.1.

Tabla 5.1: Interacciones del rendimiento	
Portabilidad	▪ Los mecanismos que facilitan la portabilidad (nivel de acceso) pueden suponer un empeoramiento del rendimiento a través de las llamadas entre módulos. ▪ No todos los sistemas operativos ofrecen los servicios que pueden ser necesarios (procesos ligeros, planificación con prioridades, predictabilidad, etc). ▪ No todos los enlaces de comunicaciones tienen el ancho de banda adecuado, ni son determinísticos. ▪ Por otro lado, la posibilidad de utilizar buses de alto rendimiento y sistemas operativos de tiempo real favorecen el rendimiento.
Integración de servicios:	Si las tareas críticas necesitan de ciertos servicios, el acceso a los mismos y su procesamiento se convierten asimismo en tareas críticas. Los mecanismos que facilitan la integración de servicios (mediadores, módulos de desacoplo, etc.) pueden suponer un empeoramiento del rendimiento a través de las llamadas entre módulos.
Uso de COTS.	Deben proporcionar el rendimiento adecuado y no hacer suposiciones arquitecturales que afecten o entren en conflicto con las estrategias seleccionadas. La posibilidad de integrar hardware específico puede mejorar mucho el rendimiento.
Flexibilidad en el despliegue:	Puede mejorar el rendimiento, pues facilita la asignación de recursos a diferentes nodos y favorece la definición de nodos *autosuficientes*.(Minimización de las comunicaciones).
Otros aspectos de la modificabilidad	En general la modificabilidad se consigue aumentando los niveles de indirección, que suponen una degradación del rendimiento.
Seguridad:	La seguridad depende del rendimiento. Si las tareas críticas no cumplen sus plazos el funcionamiento puede degradarse hasta límites inaceptables.
Usabilidad:	El rendimiento favorece la usabilidad. La información que se ofrece al operador debe reflejar el estado actual del sistema y los comandos deben activarse *inmediatamente*, dentro de los límites de la percepción humana.

5.8.3 Aspectos de la Seguridad.

La seguridad y la prevención de accidentes es un aspecto crítico de los sistemas de teleoperación. Aparte del riesgo inherente al uso de los propios mecanismos, éstos trabajan en zonas de riesgo donde la probabilidad de inicidencias y accidentes es de por sí más alta de lo habitual. Los accidentes se deben a muchos factores (en ocasiones hace falta que confluyan varios para que se produzca el accidente), algunos de los cuáles tienen poco que ver con el software. De entre ellos, se pueden destacar [Alvarez 1997]:

- Mal funcionamiento del sistema de control (hardware y software).
- Acceso indebido del personal a la zona de trabajo de los mecanismos.
- Errores humanos de los operarios.
- Roturas de partes mecánicas.
- Liberación de energía almacenada.
- Sobrecarga de los mecanismos.
- Medio ambiente o herramientas peligrosas.

Ciertos factores dependen exclusivamente de las medidas de seguridad de las instalaciones y de las que puedan definirse en los procedimientos de ejecución de las operaciones. Otros dependen del software. En los sistemas de teleoperación es necesario supervisar constantemente el funcionamiento del sistema, gestionar la ejecución de los comandos y proporcionar mecanismos de parada segura que se activen por orden del operador o cuando se detecten situaciones de riesgo o errores graves de funcionamiento.

La gestión de los comandos es necesaria para impedir que se ejecuten comandos inseguros y asegurar que los comandos se ejecutan según el plan previsto. Puede considerarse un caso particular de la monitorización del estado, sin embargo la ejecución de comandos introduce ciertas condiciones particulares. Una parte del estado del sistema debe supervisarse siempre, haya o no haya un comando en curso, otras partes son relevantes precisamente por el comando que se está ejecutando. Es necesario separar los aspectos específicos de cada comando.

La seguridad influye fuertemente en el diseño del sistema, ya que sus componentes deben pensarse desde un principio para ser seguros. Los componentes críticos deben proporcionar interfaces a través de las cuales un servicio de diagnóstico o una tarea de supervisión pueda detectar fallos en su funcionamiento. Asimismo, la seguridad está estrechamente relacionada con el rendimiento y la disponibilidad. Por un lado, los requisitos temporales deben ser garantizados para asegurar que la información disponible sobre el sistema refleja fielmente su estado, pero por otro las tareas encargadas de monitorizar el funcionamiento del sistema suponen una sobrecarga del mismo. En general los mecanismos y estrategias arquitecturales favorecen la disponibilidad son los mismos que favorecen la seguridad.

5.8.4 Aspectos de la Disponibilidad/Fiabilidad.

La disponibilidad se mide en términos de la tasa de fallos (número de fallos por unidad de tiempo) y del tiempo de reparación (número de reparaciones por unidad de tiempo o porcentaje de tiempo que el sistema está de baja debido a reparaciones). Sin embargo, estos son aspectos muy dependientes del sistema y deberán abordarse cuando se emplee la arquitectura en los sistemas concretos. Por el momento es suficiente con caracterizar la disponibilidad mediante la identificación de los servicios o funciones más críticos de los sistemas, y enumerar las estrategias y mecanismos que debe proporcionar la arquitectura para que dichos servicios estén disponibles y muestren un comportamiento fiable. En general, dichas estrategias y mecanismos se basan en aplicar diversas formas de redundancia homogénea o heterogénea. Los aspectos a destacar de este atributo son:

- Definición de modos degradados de funcionamiento.
- La inexistencia de modos comunes en los que un único fallo puede provocar la caída de todo el sistema.
- La disponibilidad y fiabilidad de los enlaces de comunicaciones: Los fallos en los enlaces de comunicaciones entre nodos deben al menos detectarse y si es posible corregirse.
- La disponibilidad y fiabilidad de las interfaces de usuario: El operador debe disponer de forma permanente de una interfaz de usuario mínima que le permita ejecutar al menos la parada segura del sistema y el rearranque del mismo. Asimismo, debe existir una interfaz reducida mediante la cual puedan ejecutarse los comandos más básicos de los mecanismos.
- La disponibilidad y fiabilidad de los mecanismos de parada segura. Debe existir al menos un mecanismo de parada segura accesible por el operador y cuyo funcionamiento no dependa del comportamiento de la aplicación.

Cuando se aborda el estudio de la disponibilidad/fiabilidad suelen tenerse en cuenta dos aspectos importantes relacionados con el funcionamiento seguro del sistema:

- La existencia de mecanismos de fallo seguro (*fail-safe*)
- La posibilidad de admitir modos de funcionamiento degradado.

Fallo seguro significa que cuando ocurre alguna circunstancia que impide el funcionamiento normal del sistema (el fallo de un componente hardware o software, el fallo de las comunicaciones, un corte de energía eléctrica, etc.) el sistema debe pasar a un estado conocido y seguro. El significado de lo que es seguro depende de cada sistema. Algunos ejemplos son:

- Parada de emergencia. El sistema se detiene.
- Modo *Hold:* El sistema no se detiene, pero se ejecutan inmediatamente ciertas acciones de seguridad.
- Modo Manual. Toda la responsabilidad pasa al operador.
- Modo Degradado. Se renuncia a la parte de funcionalidad que falla, hasta que consigue repararse.
- Rearranque del sistema. Posiblemente la medida de recuperación más popular (por desgracia) de todos los sistemas informáticos.

Para cada sistema hay que determinar la política más adecuada, que puede ser alguna de las anteriores o una combinación de varias.

5.8.5 Aspectos de la Usabilidad.

La usabilidad es un atributo difícil de caracterizar incluso en sistemas concretos, pues depende de muchos factores, algunos de ellos ligados a la arquitectura y otros no. Por otro lado, la casuística que puede generarse a través de los casos de uso y los escenarios es infinita. Por ello, en este caso, en lugar de presentar plantillas, es más conveniente destacar aquellos atributos arquitectónicos que tienen una incidencia directa para conseguir que un sistema concreto sea cómodo y fácil de usar. Entre ellos cabe destacar:

- Adaptabilidad a distintas interfaces de usuario. Cada una de ellas con las características más apropiadas para el tipo de operador y el tipo de trabajo a realizar.
- Rendimiento: Actualización rápida de la interfaz y respuesta rápida de la misma a las peticiones del operador.
- Botón o botones de parada segura fácilmente accesibles e identificables.
- Posibilidad de integrar servicios y utilidades que asistan al operador en el entrenamiento y ejecución de sus trabajos.

Todos estos aspectos han sido ya tratados en el apartado correspondiente, por lo que no se insistirá más sobre ellos.

5.8.6 Aspectos de la interoperabilidad.

Como ya se ha comentado, la interoperabilidad define la capacidad de un sistema para trabajar con otros sistemas. El grupo de trabajo de arquitecturas C4ISR[50] [C4ISR 1998] define en el modelo LISI[51] cuatro atributos para la interoperabilidad: procedimientos, aplicaciones, infraestructura y datos. Aunque el modelo LISI es un modelo orientado a sistemas de información (que no es el caso de la arquitectura objeto de esta tesis), la definición que hace del atributo *aplicaciones* revela algunos aspectos que, interpretados desde la óptica de los sistemas de teleoperación, pueden ser útiles para plantear el problema:

"El atributo aplicaciones abarca el propósito y función fundamental para el que cualquier sistema es construido: su misión. Para que la interoperabilidad ocurra de forma efectiva los sistemas deben poseer capacidades similares y debe existir una forma común de entender la información entre los mismos; de otra manera, no existe un marco de referencia común que haga posible la cooperación[52]".

La interoperabilidad entre sistemas heterógeneos es un atributo muy díficil de conseguir. Sus misiones pueden ser diferentes y su interpretación de la información puede no tener nada que ver. Sin embargo, en el caso que nos ocupa, la interoperabilidad se plantea entre sistemas que comparten la *misma* arquitectura, por tanto:

- Todos los sistemas comparten un mismo propósito. Aunque cada sistema pueda estar especializado en la realización de un pequeño conjunto de actividades, todos ellos están pensados para permitir el control a distancia de una serie de mecanismos.
- Las interfaces de acceso a los diferentes sistemas son (deben ser) bastante homogéneas, pues todas responden a las reglas de interacción definidas en la arquitectura.
- Todos los sistemas manejan información que a un cierto nivel de abstracción puede considerarse equivalente y que interpretan de formas similares (alarmas, eventos, estado de los dispositivos, comandos básicos comunes, etc.).

Además, el nivel de colaboración que se plantea es limitado:

- Los sistemas mantienen un alto grado de independencia. Aunque todos ellos colaboran en una misión más amplia, las responsabilidades de cada sistema están perfectamente definidas y desde su perspectiva no necesitan saber si la información que les llega procede de otro sistema.
- No es necesario un gran intercambio de información entre los mismos, aunque sí el paso de ciertos mensajes. Los estrictamente necesarios para coordinar sus acciones.

Con todo ello, la interoperabilidad se plantea como la posibilidad de descomponer una misión en misiones más simples asignando cada una de ellas a un sistema de teleoperación determinado y realizándose todas ellas de forma simultánea y coordinada. La figura 5.2 describe un ejemplo del tipo de interoperabilidad que se persigue.

---

[50] *Command, Control, Communication, Computer, Intelligence, Surveillance and Reconnaisance Architecture Framework*

[51] Modelo LISI (*Levels of Information System interoperability*) [CI4SR 1997]

[52] La última parte de la frase no es una traducción literal del atributo. En el original en inglés era*: "otherwise, users have no common frame of reference"*. Sin embargo, tal como queda es más apropiada para los propósitos del apartado.

El mantenimiento de los cascos de embarcaciones incluye diversas tareas, incluyendo reparaciones, soldaduras, la eliminación de la pintura vieja y el repintado. Algunas de estas operaciones presentan riesgos para los operarios, ya que se realizan sobre andamios y en una atmósfera plagada de polvo y emanaciones tóxicas.

Las operaciones de eliminación de pintura y repintado son relativamente sencillas y pueden realizarse de forma semiautomática, mediante mecanismos muy económicos, requiriéndose la intervención del operador sólo para iniciar los trabajos y en ciertas situaciones muy específicas. Ahora bien, los cascos de ciertos buques son muy grandes y es difícil que un solo robot se adapte a todos los contornos. Por otro lado, esas mismas dimensiones hacen posible simultanear operaciones de varios mecanismos siempre que se tomen ciertas precauciones. Se trata básicamente de:

- *Poner a varios mecanismos a eliminar pintura de forma simultánea sobre diferentes paños.*
- *Poner a otros mecanismos a repintar los paños tan pronto como los mecanismos de eliminación de pintura se hayan alejado lo suficiente.*
- *Sincronizar las tareas de todos los mecanismos de modo que no se solapen los paños y que el repintado no empiece hasta que los mecanismos de eliminación de pintura se encuentren lo bastante lejos.*

La automatización completa de las operaciones no es económicamente viable (tal vez tampoco técnicamente), pero el papel del operario puede reducirse a:

- *Introducir las misiones y repartir el trabajo.*
- *Iniciar el proceso.*
- *Resolver situaciones de bloqueo y atender a las alarmas.*
- *Reorganizar las misiones.*

Una vez introducidas las misiones y repartido el trabajo, los mecanismos pueden llevarlas a cabo con un alto grado de autonomía. Las misiones que introduce el operador son de muy alto nivel. No se trata de ordenarle a un mecanismo que se mueva de un punto a otro o de activar una herramienta, sino de que ejecute una misión compleja en la que pueden producirse muchos eventos

Figura 5.2: Características del Sistema GOYA (extracto de [GOYA 1998])

Tabla 5.1: Requisitos Funcionales Abstractos

1. Modos de Operación	1. Operacional: El operador gobierna los mecanismos. (En todos los sistemas). 1.1. Modo Normal (Sólo están disponibles los comandos seguros). 1.2. Modo Manual (Están disponibles todos los comandos). 1.2.1. Programación (Definición y Registro de Misiones) 1.2.2. Modos de Test y Calibración. 2. Simulación: El operador interacciona con mecanismos simulados. (En sistemas más complejos y en aquellos que integran servicios de simulación y entrenamiento). 3. Los modos deben ser cambiados mediante una acción explícita del operador.
2. Chequeo de Viabilidad de los comandos de operador	1. Chequeo del rango de los parámetros. 2. En modo normal sólo están disponibles los comandos seguros. Un comando es seguro si está permitido en el estado operacional actual y su ejecución deja al sistema en un estado conocido y seguro.
3. Monitorización constante del estado del sistema.	1. Monitorización constante del estado de los mecanismos (posición, estado activación, alarmas, etc,..) 2. Monitorización constante de la ejecución de los comandos. 2.1. Control de Movimiento 2.1.1. Los movimientos deben realizarse con la precisión y velocidad requeridas. 2.1.2. Detección de colisiones (en todos los sistemas). 2.1.3. Prevención de colisiones (en algunos sistemas). 2.2. Condiciones de seguridad que se vean afectadas por la ejecución de los comandos. 3. Monitorización constante del funcionamiento de los componentes software y hardware. 4. El operador debe ser informado de cualquier mal funcionamiento. 5. El sistema debe ser *safe-failure*.
4. Acceso a los mecanismos	1. Acceso remoto a los mecanismos. 2. Enlaces de comunicaciones eficientes y fiables. (Algunos sistemas pueden requerir protocolos de comunicaciones deterministas). 3. Si el sistema incluye más de un mecanismo (p.e: vehículo-brazo-herramienta), cada mecanismo debe poder ser accedido y gobernado individualmente.
5. Secuenciación de misiones. Coordinación y sincronización de mecanismos	1. Debe ser posible programar y realizar secuencias de operaciones (en la mayoría de los sistemas) y automatizar ciertas acciones y misiones (en algunos sistemas). 2. Si el sistema incluye más de un mecanismo debe ser posible coordinar y sincronizar sus acciones para la realización de misiones que requieran su funcionamiento simultáneo (en algunos sistemas) 3. En los sistemas más simples el operador puede encargarse de la coordinación entre mecanismos, en los más complejos, la coordinación debe ser automática, de acuerdo con el desarrollo de la misión en curso.
6. Configuración de la Aplicación	1. El sistema debe poder configurarse de acuerdo a: 1.1. Los mecanismos a controlar (diferentes tipos de brazos, vehículos y herramientas). 1.2. El entorno de trabajo (Especialmente importante en aquellos sistemas que usan representaciones gráficas del entorno y los mecanismos y realizan simulaciones de las misiones). 1.3. La misión o misiones a realizar 1. Las configuraciones 1.1 y 1.2 pueden hacerse en tiempo de carga. La 1.3 en tiempo de ejecución.
7. Servicios Especiales	1. Representación gráfica del entorno y de los mecanismos (en muchos sistemas). (9.1) 2. Simulación de misiones (en algunos sistemas). 3. Cálculos cinemáticos (en muchos sistemas) 4. Cálculo de trayectorias y detección de colisiones (En algunos sistemas) 5. Capacidades de visión artificial y sistemas de navegación (En algunos sistemas). 6. Interfaces de usuario especiales: 6.1. Botoneras especiales (en muchos sistemas) 6.2. *Joysticks* (en algunos sistemas). 6.3. Reproducciones a escala de los mecanismos teleoperados y realimentación de pares y fuerzas (en algunos sistemas).(9.6)
8. Seguridad y Logging	1. Acceso al sistema a través de *loggin* y *password* (En algunos sistemas puede haber niveles de acceso diferentes para operadores, administradores y programadores). 2. Los eventos importantes y las principales acciones del operador deben registrarse en un *logbook* (en algunos sistemas).
9. Interfaz de Usuario	1. Representación gráfica del entorno y de los mecanismos (en muchos sistemas)..(7.1) 2. Ventanas de alarmas y advertencias. 3. Parada-Segura fácilmente accesible e identificable. 4. Actualización del estado del sistema en tiempo real. 5. Envío inmediato de los comandos. 6. Interfaces de usuario especiales (7.4) (En algunos sistemas).

Tabla 5.2 : Atributos de Calidad Abstractos

1. Caracterización de la Modificabilidad

1.1. Portabilidad:
- 1.1.1. Posibilidad de cambio de plataforma y sistema operativo.
- 1.1.2. Posibilidad de cambio de enlaces y protocolos de comunicación.

1.2. Capacidad de distribución:

Posibilidad de migrar procesos o subsistemas de un nodo a otro (en tiempos de compilación o carga)

1.3. Integración de Servicios:
- 1.3.1. Posibilidad de incluir/excluir servicios especiales (gráficos, simulaciones, cálculos cinemáticos, visión artificial, etc).
- 1.3.2. Adición/Eliminación/Modificación de servicios ofrecidos por un servidor.
- 1.3.3. Sustitución de un servidor por otro.

1.4. Adaptabilidad a cambios en la Unidad de Control Local:
- 1.4.1. Incremento/Decremento de la funcionalidad de la unidad de control local.
- 1.4.2. Adición/Eliminación/Modificación de comandos de la unidad de control local.
- 1.4.3. Incremento/Decremento/Modificación de la información de estado ofrecida.

1.5. Modificaciones relacionadas con cambios en los mecanismos y en las misiones.
- 1.5.1. Adición/Eliminación/Modificación de mecanismos (brazos, vehículos, herramientas).
- 1.5.2. Cambios en la estructura y la cinemática de los mecanismos.
- 1.5.3. Adición/Eliminación/Modificación de controladores de mecanismos.
  - *1.5.3.1. Adición/Eliminación/Modificación de comandos, alarmas e información de estado.*
  - *1.5.3.2. Cambios en la máquina de estados de los mecanismos.*
  - *1.5.3.3. Adición/Eliminación/Modificación de Eventos.*
- 1.5.4. Cambios en el entorno operativo.
  - *1.5.4.1. Adaptabilidad a entornos estructurados.*
  - *1.5.4.2. Adaptabilidad a entornos no estructurados.*
- 1.5.5. Adición/Eliminación de Misiones.
- 1.5.6. Cambios en las misiones.
  - *1.5.6.1. Modificación de los pasos de las misiones (Adición/Elimin./Modificación de paso, cambio de orden).*
  - *1.5.6.2. Modificación de la sincronización (Adición/Elimin./Modificación de eventos o señales de sincronismo).*
- 1.5.7. Dada una combinación vehículo-brazo-herramienta: Cambiar el vehículo, el brazo o la herramienta.
- 1.5.8. Adaptabilidad a cambios en los requisitos de rendimiento.
  - *1.5.8.1. Cambios en el tiempo de ejecución, plazos y períodos de las tareas.*
  - *1.5.8.2. Cambios en el número de recursos requeridos por las tareas.*
  - *1.5.8.3. Cambios en la capacidad de los enlaces de comunicaciones y en las plataformas de ejecución.*

2. Caracterización del Rendimiento

2.1. Rendimiento de la Interfaz de Usuario:
- 2.1.1. La interfaz debe actualizarse a un ritmo que refleje al operador el estado actual de los sistemas de teleoperación y local y de los mecanismos.
- 2.1.2. Los comandos deben enviarse al controlador inmediatamente.

2.2. Rendimiento de la Monitorización:

El estado del sistema y de los mecanismos debe monitorizarse a una tasa y en unos plazos que garanticen el funcionamiento correcto y seguro del sistema.

2.3. Rendimiento de los Servicios:
- 2.3.1. **Gráficos:** Los gráficos deben actualizarse para reflejar el estado actual de los mecanismos y el entorno.
- 2.3.2. **Cálculos Cinemáticos/Detección de Colisiones**: Dentro de los plazos que se determinen.
- 2.3.3. **Visión Artificial:** Procesado de imágenes a un ritmo que refleje el estado actual de los mecanismos y el entorno.
- **2.3.4.** **Otros:** Dependedientes de la aplicación.

2.4. Rendimiento de las comunicaciones
- 2.4.1. Los comandos deben entregarse a la unidad de control local dentro de unos plazos determinados.
- 2.4.2. El estado de los mecanismos debe llegar al sistema de teleoperación dentro de unos plazos determinados.

Tabla 5.2: Atributos de Calidad Abstractos (Continuación)

3. Caracterización de la Seguridad

3.1. El sistema de teleoperación debe evitar situaciones que pongan en peligro a personas y bienes.

3.2. Supervisión del estado del sistema y de la ejecución de los comandos:

3.2.1. El operador debe ser informado dentro de unos plazos de cualquier mal funcionamiento del sistema. (Detección de errores graves y reporte automático al operador según políticas de información que razonablemente aseguren que éste no va a ignorar las alarmas)

3.2.2. Se debe activar automáticamente una parada segura si se detectan condiciones que pueden poner en peligro a bienes o personas.

3.2.3. Deben existir medios independientes para que el operador active una parada segura, fácilmente accesibles e identificables.

3.2.4. Deben establecerse mecanismos para evitar situaciones de riesgo, en especial las relacionadas con el acceso a las zonas de operación, las fuentes de alimentación (pérdida o sobrecarga), la sobrecarga de motores y el fallo de los enlaces de comunicación.

3.3. Fail safe. (Definición dependendiente de la aplicación).

3.3.1. En caso de fallo grave de los sistemas local y de teleoperación deben pasar a un estado conocido y seguro, según los sistemas, esto puede suponer:

3.3.1.1. Parada de emergencia. (En todos los sistemas).

3.3.1.2. Parada de protección. (En muchos sistemas).

3.3.2. En caso de fallo menos grave del sistema de teleoperación el sistema debe pasar a un estado conocido y seguro, según los sistemas esto puede suponer.

3.3.2.1. Parada de emergencia. (En los sistemas más simples)

3.3.2.2. Parada de protección.

3.3.2.3. Paso a modo degradado o manual.

3.3.3. Evitar modos comunes (evitar que los fallos se propaguen por el sistema afectando a los módulos encargados de la supervisión de la seguridad)

4. Caracterización de la Fiabilidad/Disponibilidad

4.1. Comunicaciones Fiables

4.1.1. Los enlaces de comunicación deben ser fiables.

4.1.2. Los enlaces deben ser recuperables.

4.2. Modos de operación degradados

4.2.1. En caso de fallo en los controladores, de la interfaz de usuario o de los servicios debe habilitarse un modo de operación degradado que permita el acceso a los comandos más básicos.

4.2.2. Deben existir algún mecanismo para efectuar una parada segura en caso de fallo de cualquiera de los subsistemas.

4.3. Parada Segura siempre disponible.

4.3.1. En el peor de los casos debe existir una forma de efectuar una parada segura.

4.3.2. Evitar modos comunes (evitar que los fallos se propaguen por el sistema afectando a los módulos encargados de la supervisión de la seguridad)

5. Caracterización de la Usabilidad

5.1. La interfaz de usuario debe adaptarse a las habilidades y prácticas comunes de los operadores.

5.2. La interfaz de usuario debe representar el estado actual de los mecanismos, el entorno y el sistema.

5.3. Suelen ser necesarias representaciones gráficas del entorno y los mecanismos.

5.4. En ocasiones son necesarias utilidades de visión artificial.

5.5. Safe Stop Button.

5.6. Asistencia al operador (ayuda en línea, deshabilitación de comandos inseguros, diferentes modos de operación, etc)

6. Caracterización de la Interoperabilidad

Descomponer una misión en misiones más simples asignando cada una de ellas a un sistema de teleoperación determinado y realizándose todas ellas de forma simultánea y coordinada.

6 La Arquitectura de Referencia

6.1 Introducción

Una vez caracterizados los atributos de calidad, el siguiente paso es disponer de una descripción de la arquitectura que haga posible su análisis mediante ATAM. Tal descripción debe:

1. Proporcionar una identificación precisa de los estilos o patrones arquitectónicos utilizados en la arquitectura, acompañada de los razonamientos que justifiquen su empleo.
2. Proporcionar información suficiente para evaluar todos los atributos de calidad que se consideren importantes, en un grado de detalle que permita la ejecución de los escenarios de calidad. En algunos casos puede ser necesario proporcionar modelos formales que permitan analizar y simular el comportamiento de la arquitectura en relación con un determinado atributo [Kazman et al 1998a].
3. Identificar claramente las responsabilidades y los atributos de calidad de cada subsistema de forma que, siguiendo las pautas marcadas por el ABD, puedan detectarse inconsistencias e faltas de compleción al confrontarlas con las responsabilidades y los atributos de calidad previamente definidos.

El objetivo de este capítulo es proporcionar dicha descripción. Para ello, se partirá de la descripción ofrecida por Álvarez en [Alvarez 1997][53]. Dicha descripción está estructurada según el modelo 4+1 vistas propuesto por Krutchen [Krutchen 1995], ya explicado en el capítulo 2, y proporciona una buena caracterización de la arquitectura, pero no se adapta lo suficientemente bien a los requisitos del ATAM y del ABD, por lo que será necesario realizar en la misma algunas modificaciones y refinamientos[54]. Los principales inconvenientes de la descripción ofrecida por Álvarez son los siguientes:

1. Utiliza diferentes notaciones para describir las diferentes vistas de la arquitectura, haciéndose difícil en ocasiones establecer relaciones entre las mismas.
2. Gran parte de la información no está expresada de forma gráfica, sino textual. Los diagramas ofrecidos apoyan al texto, pero por sí mismos sólo proporcionan una descripción parcial de la arquitectura.

[53] Asimismo, existen ejemplos de utilización de la misma descritos en [Alonso et al 1997], [Alvarez et al 2000a, b, c], [Iborra et al 2000] y [Pastor et al 1996, 1998, 2000].

[54] Las modificaciones y refinamientos propuestos en este capítulo se refieren a la *descripción* de la arquitectura, no a la arquitectura misma.

3. La vista lógica no está suficientemente desarrollada, si bien puede ser suficiente a efectos de evaluar la arquitectura.
4. En la vista lógica se mezclan aspectos realmente conceptuales con otros más propios de la infraestructura, que probablemente estarían mejor ubicados en diagramas de componentes o de despliegue.

Así, el enfoque propuesto para este capítulo es el siguiente:

1. Se revisará la descripción ofrecida en [Alvarez 1997], traduciendo a UML los diagramas proporcionados, reorganizando algunos de ellos y añadiendo otros nuevos cuando sea necesario.
2. Se relacionarán tabularmente los requisitos definidos en el capítulo 5 con los subsistemas definidos en la arquitectura.

Con todo ello, el capítulo se estructura de la siguiente manera. En primer lugar se describe la jerarquía de subsistemas, explicando las relaciones estructurales y dinámicas entre los mismos. Esto cubre la descripción de la vista lógica. A continuación se presentan las vistas de concurrencia, de despliegue y desarrollo. Por último, se presentan las tablas responsabilidad/subsistema y calidad/subsistema que se acaban de mencionar en el párrafo anterior.

Cómo ya se ha mencionado, la descripción de la vista lógica adolece de ciertas carencias, pero con la información disponible no pueden proporcionarse más detalles. Cuando se aborde la evaluación de la arquitectura en el siguiente capítulo, estas carencias se pondrán explícitamente de manifiesto. En este capítulo el objetivo es obtener una descripción suficiente para la evaluación[55], no redefinir o completar la arquitectura.

6.2 Subsistemas Básicos

En [Alvarez 1997] se identifican cinco subsistemas o componentes básicos que definen un diseño genérico de alto nivel, en el que cada uno de los subsistemas puede verse como una "caja negra" reconfigurable que se comunica con los demás a través de interfaces bien definidas. La figura 6.1 muestra el diagrama de subsistemas proporcionado en [Alvarez 1997]. Los subsistemas identificados son:

- Representación gráfica.
- Módulo de Cinemática, Simulación y Detección de Colisiones.
- Interfaz de Usuario.
- Comunicaciones.
- Controlador de Robot.

[55] En realidad, para la evaluación se ha manejado información más precisa y diagramas más elaborados que los presentados en el capítulo. Sin embargo, existe un limite razonable para la extensión del capítulo. Tal vez sería más correcto decir que el objetivo del capítulo es proporcionar suficiente información para *entender* la evaluación.

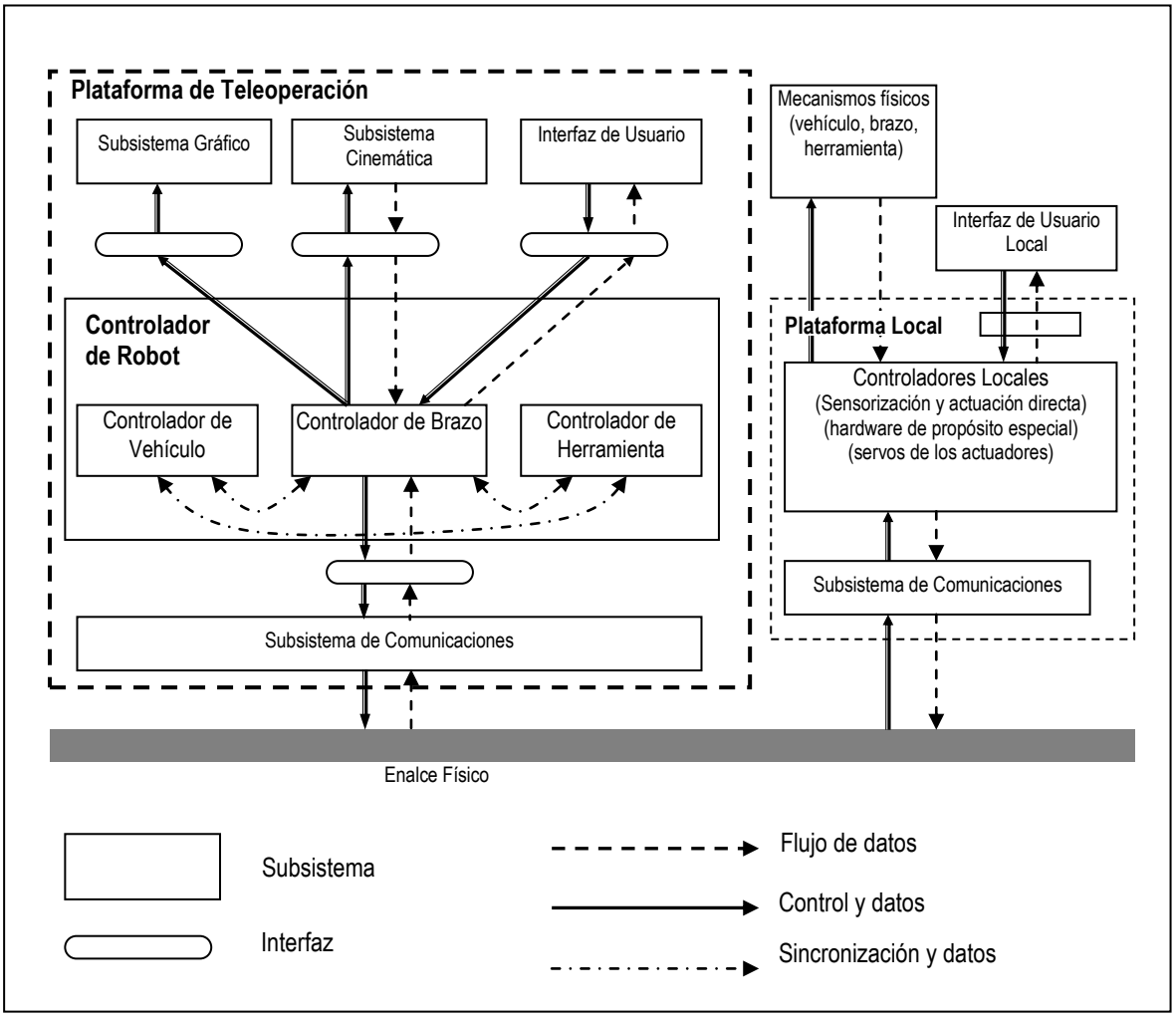


Figura 6.1: Descripción arquitectural de alto nivel [Alvarez 97].

Todos los controladores de la plataforma de teleoperación interaccionan de la misma forma con la interfaz de usuario y los subsistemas gráfico, cinemático y de comunicaciones. En la figura 6.1 sólo se han representado las interacciones del controlador del brazo. Aunque la figura muestra la unidad de control local, la arquitectura definida en [Alvarez 1997] se refiere únicamente a la plataforma de teleoperación, no imponiendo ninguna condición al diseño de la unidad de control local. La figura 6.2 resume las responsabilidades de cada uno de los subsistemas identificados en la figura 6.1.

En [Alvarez 1997] se modelan las relaciones entre los subsistemas básicos según patrones o estilos definidos en [Shaw et al 1997]. La selección de los estilos adecuados es fundamental para alcanzar los atributos de calidad exigidos a la arquitectura. Cada estilo o patrón define la forma en que se ensamblan e interaccionan los componentes del sistema y su documentación sirve de guía tanto para el correcto empleo de la arquitectura como para su evaluación. Sin embargo, el diagrama de la figura 6.1 no sirve para mostrar con suficiente claridad los estilos utilizados en la arquitectura. Por ello, antes de pasar revisión a los mismos es conveniente proporcionar en los siguientes apartados una descripción más adecuada.

Subsistema	Responsabilidades
Subsistema Gráfico: **`GrapServer`**	**Está encargado de representar gráficamente el robot y el entorno de trabajo. Debe incluir funcionalidad suficiente para:** ✓ Recibir la posición actual del robot por parte del controlador. ✓ Actualizar en pantalla dicha posición. Enviar al controlador el resultado de dicha operación.
Subsistema Cinemático : **`CinServer`**	**Calcula trayectorias y detecta colisiones. Debe incluir funcionalidad para:** ✓ Recibir parámetros del movimiento (posición destino, velocidad, aceleración, etc). ✓ Calcular la cinemática del robot. ✓ Simular movimientos para comprobar que no existen colisiones. ✓ Detectar colisiones. ✓ Calcular tiempo de movimiento. Enviar el resultado de sus operaciones al controlador.
Interfaz de Usuario: **`RemoteUI`**	**Interacciona con el operador mostrándole el estado del robot y permitiéndole el acceso a los distintos comandos. De forma general, se le puede asignar la siguiente funcionalidad:** ✓ Recepción de comandos por parte del operador. ✓ Análisis léxico y sintáctico de los comandos recibidos. ✓ Envío del comando al controlador si es correcto. ✓ Recepción de mensajes de estado o avisos al operador enviados por el controlador. Actualización en pantalla de los mensajes recibidos.
Subsistema de Comunicaciones **`CommSys`**	**Engloba el protocolo de comunicaciones con la unidad de control del robot, permitiendo el envío de comandos y la recepción de mensajes de estado.** Este subsistema debe incorporar un tratamiento de fallos que garantice la máxima seguridad. Deberá intentar el reenvío de mensajes en caso de fallo y el restablecimiento de las comunicaciones en caso de una pérdida de las mismas.
Controlador de Robot **`RemoteCtler`**	**Maneja los comandos de usuario, comprueba su viabilidad y ordena y monitoriza su ejecución.** Para ello, recibe mensajes de estado del robot e interacciona con el resto de subsistemas para asegurar la viabilidad de los comandos y monitorizar su ejecución. Se le puede asignar la siguiente funcionalidad: ✓ Recibir comandos desde la Interfaz de Usuario. ✓ Comprobar la viabilidad de su ejecución en función del estado del sistema. ✓ Si se trata de un comando de movimiento, simular antes su ejecución, haciendo uso de los servicios del Módulo Cinemático. ✓ Si el comando es viable, debe enviarlo a la unidad de control del robot. ✓ Cuando recibe un mensaje de estado de la unidad de control, debe enviarlo a la Interfaz de Usuario y al subsistema de Representación Gráfica. ✓ Sincroniza las acciones de los controladores que incluye (brazo, herramienta, vehículo y otros que pudieran definirse) En cualquier momento, el controlador puede decidir la ejecución de un comando automático, en función del estado del sistema, por ejemplo puede ordenar la parada del robot.

Figura 6.2: Subsistemas Principales. Responsabilidades [Álvarez 1997].

6.3 Relaciones entre subsistemas.

Como se dijo en el capítulo 2, en UML 1.4 existen los conceptos de *subsistema* y *modelo*. Al igual que los paquetes, los subsistemas definen un espacio de nombres, pero a diferencia de estos constituyen una unidad de comportamiento del sistema físico [UML1.4 1999]. Un subsistema ofrece interfaces y tiene operaciones. Modelos y subsistemas pueden combinarse en una jerarquía paquete/subsistema en la que la cima representa la frontera del sistema físico. Por ello, el diagrama de la figura 6.1 puede redefinirse y organizarse utilizando el concepto de subsistema tal y como se define en UML 1.4.

6.3.1 Jerarquía de subsistemas.

La figura 6.3 define la jerarquía de subsistemas del sistema de teleoperación y la figura 6.4 las interfaces que realizan y a las que llaman. Los nombres de los subsistemas se corresponden con los nombres escritos en el tipo de letra **`Courier`** de la figura 6.2. El prefijo **`Remote`** de la interfaz de usuario (**`RemoteUI`**) y del controlador de robot (**`RemoteCtler`**) enfatiza el hecho de que ambos son subsistemas del sistema de teleoperación (**`TeleopSys`**) y sugiere la existencia de un sistema local conectado directamente a los mecanismos teleoperados.

A partir de la figura 6.1 y de la división de responsabilidades entre subsistemas (figura 6.2) se deduce inmediatamente el carácter jerárquico de la arquitectura, que se estructura en cuatro niveles. El inferior, representado por **`LocalSys`** y **`CommSys`**, se ocupa de la interfaz con el hardware, de los servos de los motores, de los dispositivos periféricos y de los enlaces de comunicaciones. El inmediatamente superior está constituido por los servidores (**`CinServer`**, **`GrpServer`** y otros que pudieran añadirse). A continuación se sitúa el subsistema controlador **`RemoteCtlr`** que constituye lo que algunos autores [Kapoor et al 1996] denominan nivel operacional y, por último, en el nivel superior se encuentra la interfaz de usuario, **`RemoteUI`**.

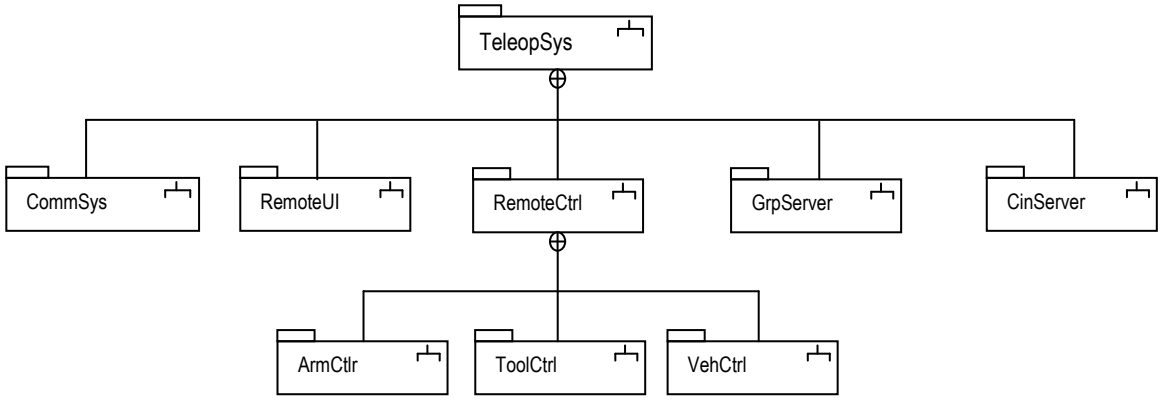


Figura 6.3: Sistema de Teleoperación. Jerarquía de subsistemas

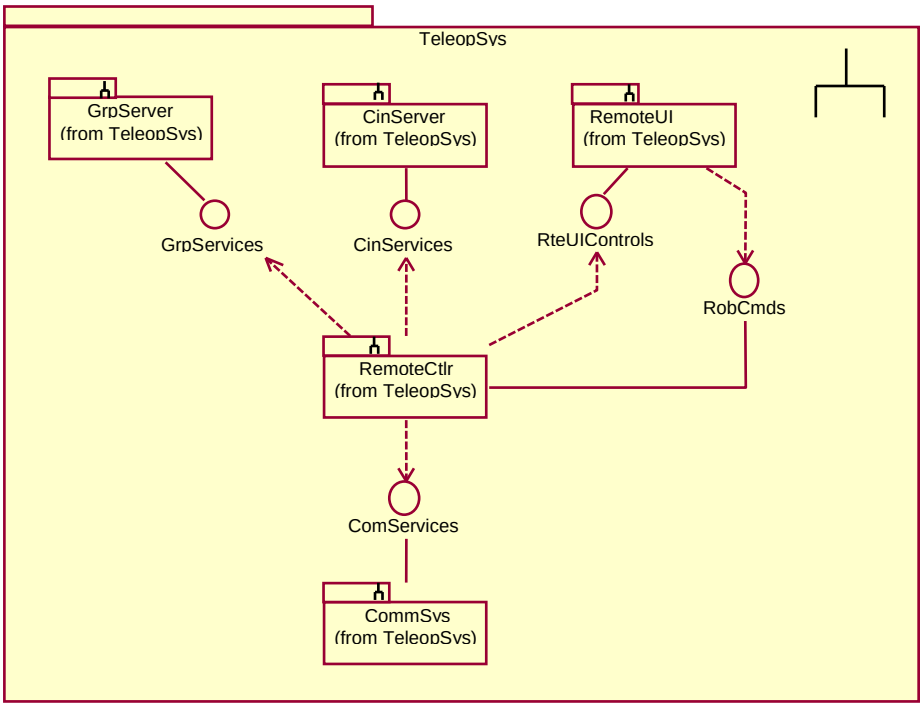


Figura 6.4a: Sistema de Teleoperación. Relaciones de los subsistemas principales

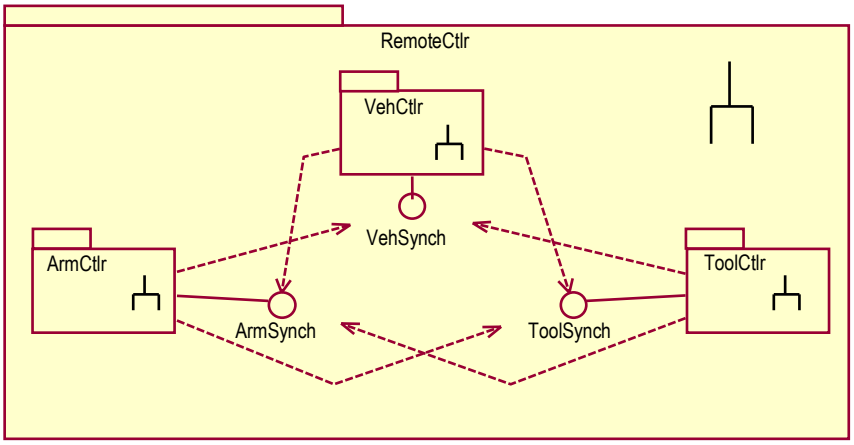


Figura 6.4b: Relaciones entre subsistemas de **`RemoteCtlr`**

Figura 6.4: Relaciones estructurales entre subsistemas

Dentro de esta organización, el *corazón* de la arquitectura está constituido por **`RemoteCtlr`**. Este subsistema asume la mayor parte de las responsabilidades, toma la práctica totalidad de las iniciativas de control y a través de él pasa o se genera toda la información que fluye por el sistema.

Las figuras 6.3 y 6.4 describen las relaciones estructurales de alto nivel entre subsistemas. No obstante, es necesario realizar las siguientes observaciones respecto de los diagramas que aparecen en dichas figuras:

1. En la figura 6.3. De todos los subsistemas que dependen directamente de **`TeleopSys`**, los únicos que son estrictamente necesarios en todos los sistemas son **`RemoteUI`** y **`RemoteCtler`**. El resto existirán o no dependiendo de las necesidades específicas de cada aplicación.
2. Obsérvese nuevamente en la figura 6.3. que **`RemoteCtler`** es un agregado de varios subsistemas. La presencia o ausencia de los mismos depende de la aplicación. Por ejemplo, si los mecanismos a teleoperar son un brazo y una herramienta, sólo existirán **`ArmCtlr`** y **`ToolCtlr`**. La figura representa el caso más general, que contempla la existencia de un vehículo (**`VehCtlr`**) portando un brazo (**`ArmCtler`**) con una herramienta ensamblada en su extremo (**`ToolCtlr`**)
3. Los subsistemas de **`RemoteCtler`** se relacionan a través de interfaces de sincronismo. No existe ningún coordinador que sincronice sus acciones. Cada subsistema controlador (**`ArmCtlr`**, **`ToolCtlr`** y **`VehCtlr`**) es responsable de llamar correctamente a las interfaces de sincronismo del resto.
4. La relación entre el controlador de robot (**`RemoteCtlr`**) y el resto de los subsistemas, excepto la interfaz de usuario (**`RemoteUI`**), sigue un patrón **cliente/servidor**, donde el rol de cliente es desempeñado por el controlador. El patrón cliente/servidor permite desacoplar los módulos servidores (servidor de gráficos, **`GrapServer`**, servidor de cinemática, **`CinServer`** y subsistema de comunicaciones, **`CommSys`**) del controlador de robot, **`RemoteCtlr`**.
5. La relación entre la interfaz de usuario, **`RemoteUI`**, y el controlador sigue un esquema diferente. En una relación cliente/servidor, el servidor no necesita conocer a sus clientes. Entre la interfaz de usuario y el controlador de robot existe una relación más simétrica. La interfaz de usuario invoca los comandos del controlador a través de la interfaz ***`RobCmds`*** y el controlador actualiza la interfaz de usuario a través de la interfaz ***`RteUIControls`***.

6.3.2 Clases de desacoplo de RemoteCtlr. Patrones de interacción entre subsistemas.

En la figura 6.5 se muestran una serie de objetos cuyos nombres acaban en **`Decoupler`** o **`Listener`**. Son los módulos de desacoplo, cuya responsabilidad es hacer de mediadores entre el controlador de robot y el resto de subsistemas.. Los módulos de desacoplo no se incluyen en las figuras 6.3 y 6.4 por dos razones:

1. No son auténticos subsistemas, sino **mediadores** que definen los patrones de interacción entre **`RemoteCtlr`** y el resto de los subsistemas.
2. No se definen de forma independiente sino, como se muestra en la figura 6.5, formando parte del subsistema **`RemoteCtlr`**.

La figura 6.5 describe las relaciones estructurales entre **`Controller`** y las clases de desacoplo. Las figuras 6.6, 6.7 y 6.8 muestran sus relaciones dinámicas. Todas juntas definen el **modelo de procesos** de **`RemoteCtlr`** y, por separado, cada una de ellas es arquetípica, en el sentido de que definen las tres formas de interacción que considera la arquitectura para conectar **`RemoteCtler`** con otro subsistema. Como **`Controller`** puede ser a su vez un agregado de controladores más específicos (**`ArmCtlr`**. **`ToolCtlr`**, **`VehCtlr`**), podrían definirse módulos de desacoplo correspondientes a cada uno de ellos. A continuación se describirán las responsabilidades de dichos módulos de desacoplo y el tipo de interacciones que realizan.

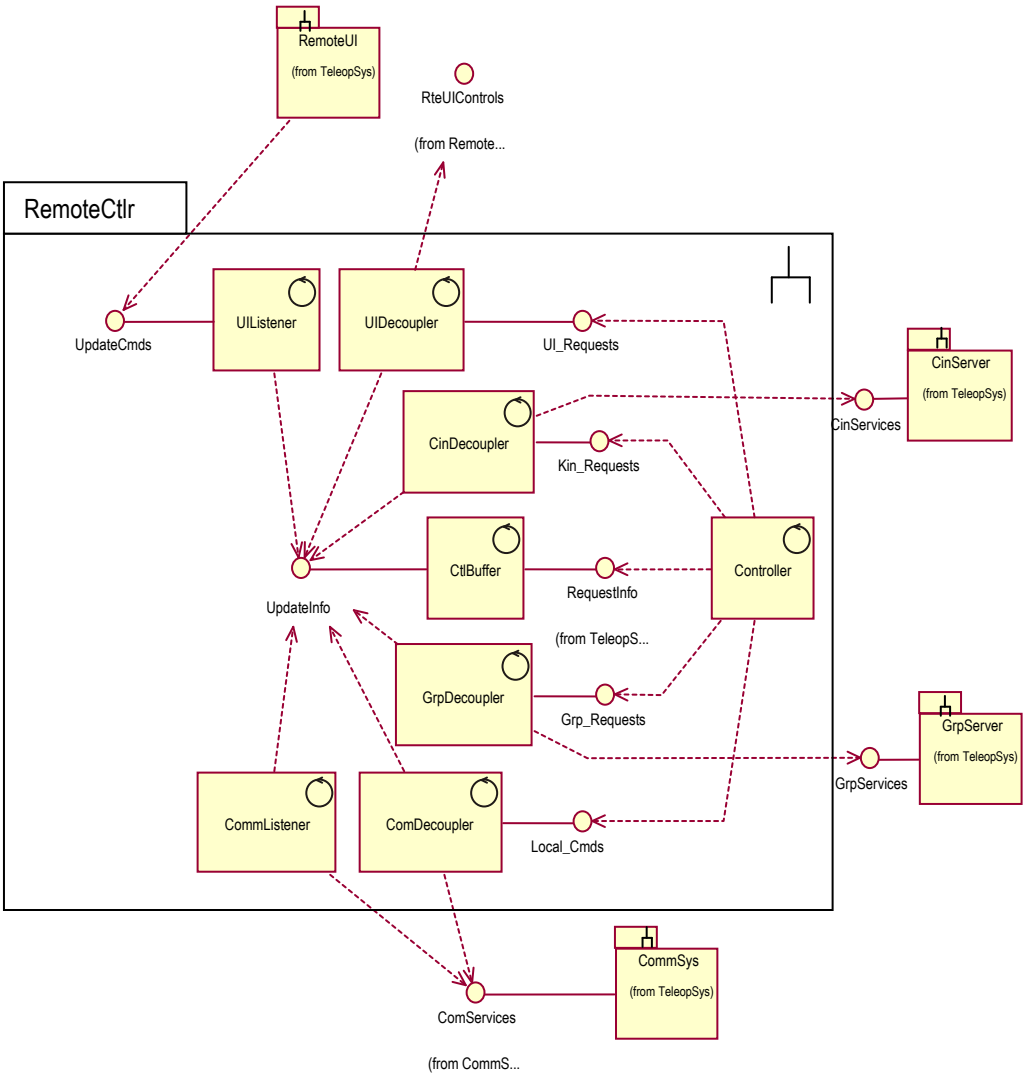


Figura 6.5. Subsistema `RemoteCtlr`: Módulos de desacoplo.

Cometidos de las clases de desacoplo:

1. Proporcionar un mecanismo de comunicación asíncrono entre **`Controller`** y los subsistemas **`CinServer`**, **`GrapServer`**, **`CommSys`** y **`RemoteUI`**.

   El controlador no puede esperar por el resto de los subsistemas, ni quedar bloqueado por un mal funcionamiento de los mismos.

2. Proporcionar a **`Controller`** una interfaz estándar inmutable para el acceso al resto de los subsistemas. De esta manera aíslan al controlador de cambios en sus interfaces. En este sentido, las clases de desacoplo realizan la función de **adaptadores**.

3. Ocultar al controlador aspectos de distribución. Algunos de los subsistemas podrían ejecutarse en máquinas diferentes (**`CinServer`** y **`GrpServer`** pueden ser extremadamente exigentes en términos de recursos de computación). Las clases de desacoplo pueden implementar o incluir un **proxy** de los subsistemas remotos, ocultando los aspectos de distribución al controlador.

Desacoplo entre Controller y CinServer

El desacoplo entre el controlador y el servidor de gráficos se realiza a través de **`CinDecoupler`** y **`BufferCtlr`**. **`CinDecoupler`** ofrece al controlador la interfaz ***`Cin_Requests`***.

Todas las llamadas que **`Controller`** hace a **`CinDecoupler`** son asíncronas (véase figura 6.6).

`CinDecoupler` invoca los servicios que **`CinServer`** ofrece a través de la interfaz ***`CinServices`***. Estas llamadas pueden ser síncronas o asíncronas dependiendo de las características de ***`CinServices`***. ***`CinServices`*** puede cambiar, en especial si se implementa a partir de COTS o es una herramienta gráfica integrada en el sistema, que puede variar de una aplicación a otra. ***`Cin_Requests`*** es una interfaz inmutable.

Cuando **`CinDecoupler`** recibe de **`CinServer`** la respuesta a su petición de servicio, la reenvía síncronamente a **`BufferCtler`**, de donde la recogerá **`Controller`** cuando lo estime oportuno. En general, **`Controller`** no espera por los resultados. Por ello, en la figura 6.6, el mensaje se ha estereotipado como *balking*[56]. Es responsabilidad de **`Controller`** tomar las medidas adecuadas si la respuesta no llega en el plazo esperado.

Desacoplo entre Controller y GrapServer

Puede aplicarse exactamente lo mismo que en el caso anterior, con solo cambiar **`CinServer`** por **`GrapServer`**, **`CinDecoupler`** por **`GrpDecoupler`**, ***`CinServices`*** por ***`GrpServices`*** y ***`Cin_Requests`*** por ***`Grp_Requests`***.

Desacoplo entre Controller y RemoteUI

1. **Actualización de la interfaz de usuario.**

   `Controller` actualiza la interfaz de usuario siguiendo un esquema parecido a la forma en la que invoca los servicios de **`CinServer`** y **`GrpServer`**, con la diferencia de que en este caso no espera ninguna respuesta (véase figura 6.7). No obstante, aunque los mecanismos

[56] El cliente no solicita el servicio si el servidor no puede atenderle inmediatamente.

de interacción son parecidos, la semántica de las relaciones es muy diferente. **`Controller`** necesita los servicios de **`CinServer`** y **`GrpServer`**, pero no necesita los servicios de **`RemoteUI`**. En este caso, si consideráramos las relaciones entre ambos según un patrón cliente/servidor[57], el rol de servidor le correspondería al controlador.

Todas las llamadas que **`Controller`** hace a **`UIDecoupler`** son asíncronas (véase figura 6.7). **`UIDecoupler`** invoca los métodos que **`RemoteUI`** ofrece a través de la interfaz ***`RteUIControls`***. Estas llamadas pueden ser síncronas o asíncronas dependiendo de las características de ***`RteUIControls`***. ***`RteUIControls`*** puede cambiar, en especial si se implementa a partir de COTS o de *toolkits* comerciales. ***`UI_Requests`*** es una interfaz inmutable.

2. Envío al controlador de comandos de usuario.

El envío y recepción de comandos se realiza siguiendo un esquema que puede asimilarse a un patrón **observador simplificado**[58].

`UIListener` se registra en la interfaz de usuario (**`RemoteUI`**) durante la compilación. Cada vez que el operador introduce un nuevo comando, la interfaz lo envía a **`UIListener`** invocando algún método de la interfaz ***`RobCmds`***. **`UIListener`** reenvía los comandos recibidos a **`BufferCtlr`**. Es reponsabilidad de **`Controller`** inspeccionar el buffer para obtener los comandos introducidos por el usuario. **`Controller`** no espera por los comandos. Por ello, en la figura 6.7, el correspondiente mensaje se ha estereotipado como *balking*.

Desacoplo entre Controller y CommSys

1. Envío de Comandos a la Unidad de Control Local.

`Controller` envía comandos a la unidad de control local a través de la interfaz ***`Local_Cmds`*** de **`ComDecoupler`**. Todas las llamadas que **`Controller`** hace a **`ComDecoupler`** son asíncronas (véase figura 6.8).

`ComDecoupler` invoca los servicios que **`CommSys`** ofrece a través de la interfaz ***`ComServices`***. Estas llamadas pueden ser síncronas o asíncronas dependiendo de las características de ***`ComServices`***. ***`ComServices`*** puede cambiar, dependiendo del *middleware* y de los protocolos de comunicaciones subyacentes. ***`Com_Requests`*** es una interfaz inmutable.

Las comunicaciones deben ser seguras y fiables, de forma que **`ComDecoupler`** debe informar a **`Controller`** si la transmisión del comando se ha realizado satisfactoriamente. Para ello, **`ComDecoupler`** inserta en **`BufferCtler`** el resultado de la transmisión. **`Controller`** lo inspeccionará cuando lo estime oportuno. En general, **`Controller`** no espera por los resultados. Por ello, en la figura 6.8, el mensaje se ha estereotipado como *balking*. Es responsabilidad de **`Controller`** tomar las medidas adecuadas si ha habido problemas con las comunicaciones.

2. Recepción del estado de la Unidad de Control Local

[57] [Alvarez 1997] no lo hace.
[58] No se contemplan en principio servicios de suscripción que puedan invocarse en tiempo de ejecución, ni la interfaz de usuario mantiene lista alguna de observadores.

La tarea **`ComListener`** invoca periódicamente los métodos de recepción de datos que ofrece **`CommSys`** a través de su interfaz **`ComServices`**. Cada vez que se reciben nuevos datos, **`ComListener`** los procesa y, si lo estima oportuno, los introduce en **`BufferCtlr`**, de donde los extraerá **`Controller`** cuando lo estime oportuno. **`Controller`** no espera por los resultados. Por ello, en la figura 6.6c, el mensaje se ha estereotipado como *balking*. Es responsabilidad de **`Controller`** tomar las medidas adecuadas si no recibe el estado de la unidad de control local.

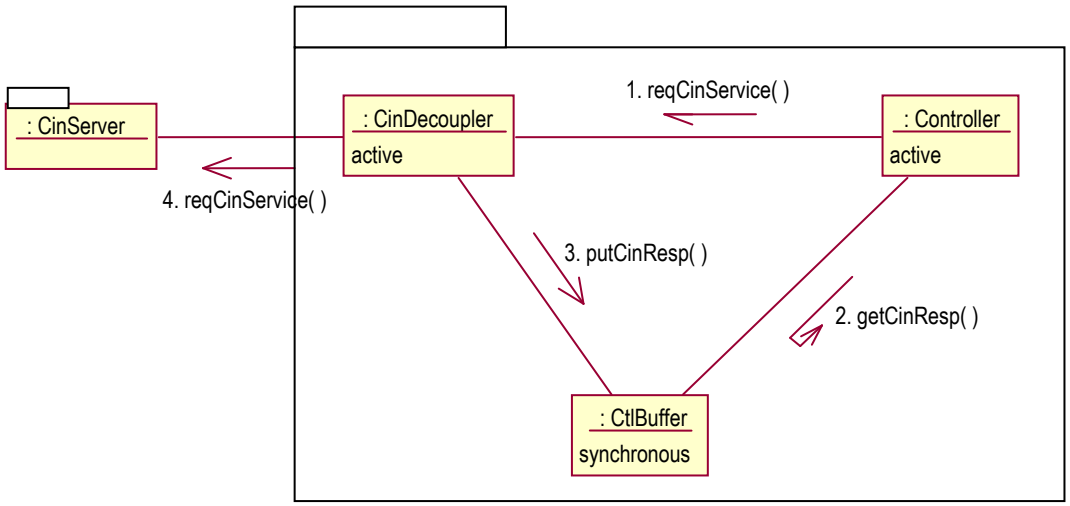


Figura 6.6: Relaciones dinámicas entre el controlador y sus módulos de desacoplo.

Esquema de interacción entre el controlador y los subsistemas servidores (`CinServer`, `GrpServer`)

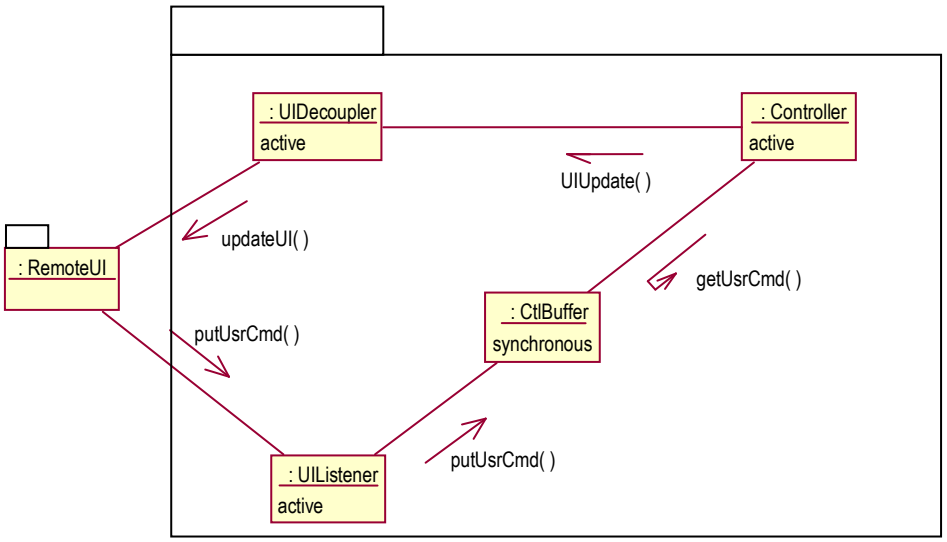


Figura 6.7: Relaciones dinámicas entre el controlador y sus módulos de desacoplo.

Esquema de interacción entre el controlador y la interfaz de usuario (`RemoteUI`)

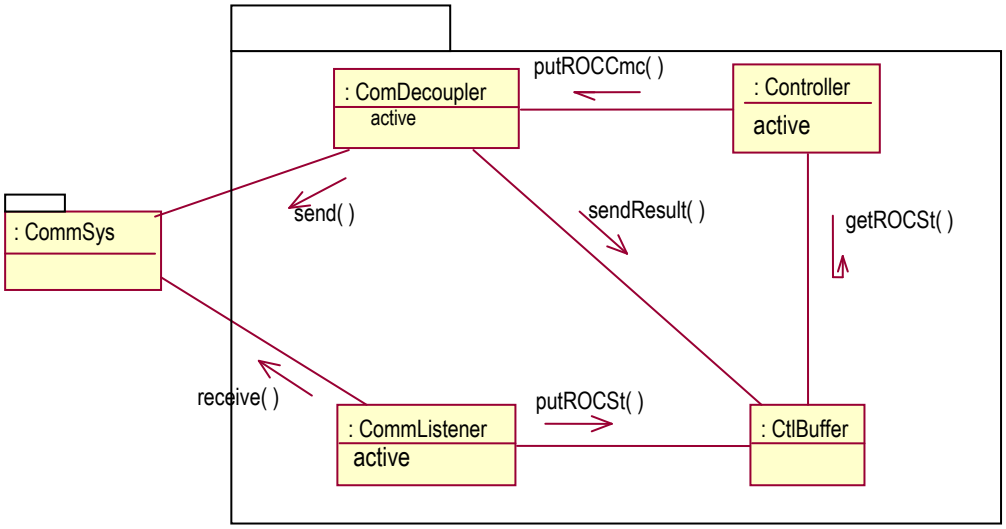


Figura 6.8: Relaciones dinámicas entre el controlador y sus módulos de desacoplo.

Esquema de interacción entre el controlador y el subsistema de comunicaciones (`CommSys`)

6.4 Modelo de procesos

Puesto que cada subsistema puede potencialmente ejecutarse sobre su propia plataforma, la arquitectura asume que cada uno de los subsistemas, excepto **`CommSys`**, tiene asociado al menos un proceso con sus propios recursos y espacio de direcciones. Cada uno de estos procesos puede incluir a su vez un conjunto de *threads* o procesos ligeros, sin que puedan en principio hacerse suposiciones sobre el número de *threads* incluidos en cada proceso. Esto es así porque la implementación de los subsistemas puede variar mucho de un sistema a otro. De hecho, pueden ser herramientas comerciales integradas en la aplicación cuyas características de concurrencia son a priori desconocidas. **`CommSys`** ha de verse como una biblioteca de servicios de acceso a la infraestructura de comunicaciones que utliza **`RemoteCtlr`**.

Los *threads* que pueden incluirse en **`RemoteCtlr`** están descritos en las figuras 6.6, 6.7 y 6.8. Sin embargo, la arquitectura admite que los *threads* correspondientes a los módulos de desacoplo (todos, excepto el de **`Controller`**) puedan agruparse atendiendo a criterios de rendimiento y planificación, según las necesidades de cada sistema. En [Alvarez 1997] se proporciona un marco para el análisis del comportamiento temporal de la arquitectura, estudiando las secuencias de tiempos para las operaciones y llamadas entre módulos, que se completa en [Alvarez et al 1998]. En el capítulo 7 se propondrá un modelo parcial encaminado a realizar escenarios de evaluación asociados al rendimiento no contemplados en dichas publicaciones.

6.5 Modelo Físico

En este modelo (véase figura 6.9) se identifican los procesadores del sistema, su interconexión y los componentes que se ejecutan en cada uno de ellos. La arquitectura admite diferentes despliegues. La figura 6.9 muestra el despliegue más habitual, con un nodo de teleoperación donde se ejecutan los procesos correspondientes a los diferentes subsistemas y un nodo local donde se ejecuta el sistema de control local. El uso del modelo cliente/servidor para la comunicación del controlador con los subsistemas de Gráficos, **`GrpServer`**, y Cinemático, **`CinServer`**, permite su asignación a distintas máquinas (figura 6.10a), posibilitando la utilización de un sistema operativo de tiempo real para las operaciones críticas. También es posible que los sistemas de teleoperación y de control local se ejecuten ambos en un mismo nodo (figura 6.10b).

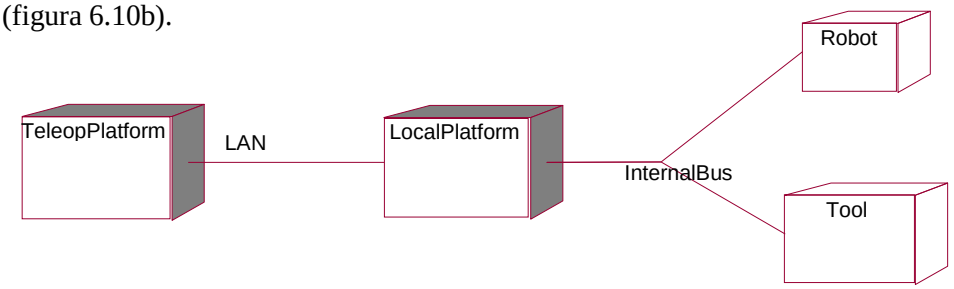


Figura 6.9: Despliegue típico

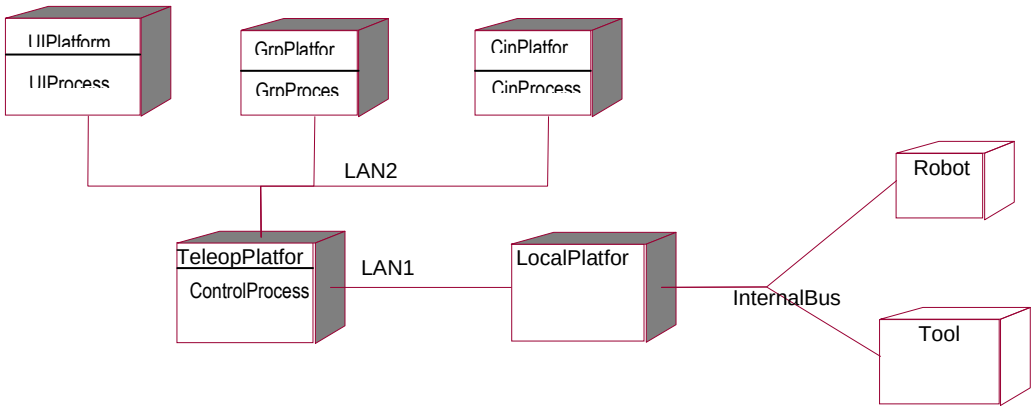


Figura 6.10a: Despliegue con todos los subsistemas activos distribuidos

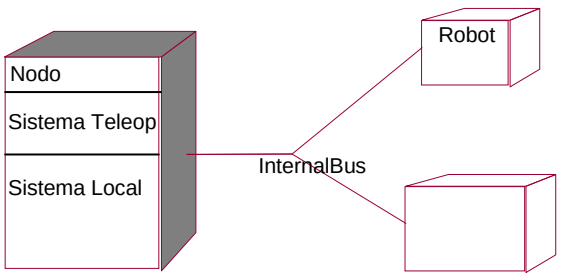


Figura 6.10b: Despliegue en un único nodo

Figura 6.10: Despliegues alternativos

6.6 Modelo de Desarrollo

En este modelo se describe la organización estática del software en su entorno de desarrollo. [Alvarez 1997] organiza el software agrupándolo en subsistemas, organizados en una jerarquía de niveles donde cada uno ofrece una interfaz bien definida. Los niveles más bajos soportan mecanismos independientes del dominio de aplicación, sistemas operativos, sistemas de gestión de base de datos, mecanismos de comunicación, etc.

El siguiente nivel añade características de una arquitectura software específica del dominio de teleoperación, y cada nivel va añadiendo más funcionalidad, siendo el nivel superior el que muestra la interfaz de usuario. La figura 6.11 muestra el modelo de desarrollo de la arquitectura objeto de este trabajo de tesis.

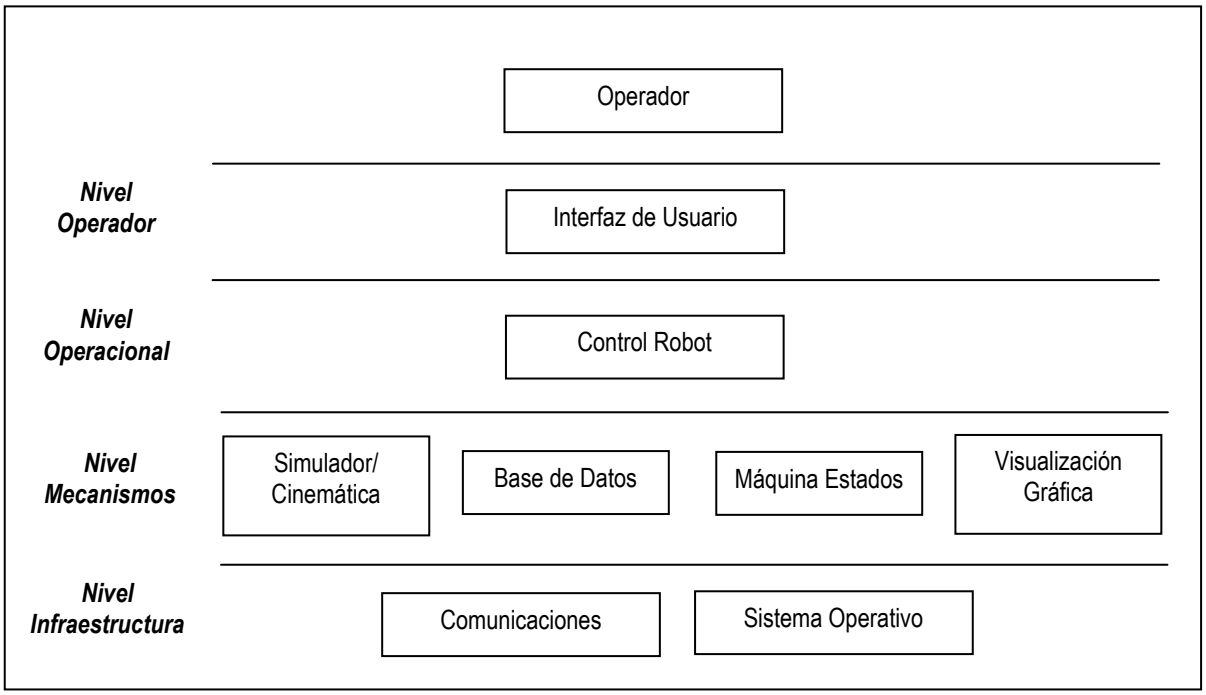


Figura 6.11: Modelo de desarrollo del sistema.

En la vista de desarrollo se ha considerado que el módulo de Control de Robot, **`RemoteCtler`** según la terminología empleada, ocupa el lugar central de la arquitectura y se tiene en cuenta que asociados con el robot hay un conjunto de comandos que el controlador del mismo debe aceptar y una máquina de estados que representa a aquél y permite tomar una decisión acerca de los comandos que pueden ejecutarse en cada momento. Los niveles definidos en la figura 6.11 tienen en cuenta los pasos que sigue un comando desde que es introducido por el usuario hasta que finaliza su ejecución (test de viabilidad, simulación previa de los comandos de movimiento, envío del comando al robot, monitorización y actualización de la máquina de estados). Según esto los niveles identificados son los siguientes:

Nivel de Operador: Es necesario permitir al operador introducir comandos y recibir mensajes acerca del estado de la máquina. Será la interfaz de usuario la que permita estas operaciones.

Nivel Operacional: Este nivel, constituido por el controlador de robot, es el núcleo del sistema. Se encarga de procesar la información del robot y del envío de comandos al mismo.

Nivel de Mecanismos: Incluye un conjunto de componentes que proveen servicios para operar el robot.

Nivel de Infraestructura: Incluye los servicios del sistema operativo y de las bibliotecas que ofrecen servicios para la comunicación entre procesos.

Cada nivel ofrece servicios a los niveles superiores a través de interfaces, diseñadas de forma que sea posible la máxima reutilización de componentes a la hora de desarrollar otra aplicación para otro robot o para incorporar nuevas misiones al mismo. Los componentes e interfaces de cada nivel están descritos en [Alvarez 1997]. Baste decir aquí que se ha tenido un especial cuidado para asegurar la independencia de operaciones entre los subsistemas que se conectan.

6.7 Responsabilidades, atributos de calidad e interfaces abstractas de los subsistemas.

Como se ha explicado en el capítulo 4, el ABD propone que todas las responsabilidades que debe asumir el sistema sean asignadas a los diferentes subsistemas, de forma que no quede ninguna sin considerar. La descripción ofrecida en los apartados anteriores ha tratado de dar una panorámica breve, pero lo más completa posible de la organización del sistema en subsistemas y de los mecanismos arquitectónicos que gobiernan sus interacciones. Llegados a este punto y siguiendo las pautas del ABD se va proceder a asignar las responsabilidades resumidas en la tabla 5.2 del capítulo 5 a los subsistemas identificados en este capítulo. Siguiendo este esquema, la tabla 6.1 resume las responsabilidades de cada subsistema, así como la información que producen y consumen. La tabla 6.2, por su parte, resume los atributos de calidad de cada subsistema y los mecanismos que proporciona la arquitectura para alcanzarlos.

La información mostrada en las tablas 6.1 y 6.2 debe relacionarse con los requisitos funcionales abstractos y los atributos de calidad identificados las tablas 5.2 y 5.3. Si alguna de las responsabilidades definidas en la figura 5.2 no está claramente asignada a ningún subsistema en la tabla 6.1, este hecho revela falta de compleción. De la misma manera, si en la tabla 6.2 alguno de los subsistemas no dispone de un mecanismo arquitectónico que le permita alcanzar sus atributos de calidad, la arquitectura debe ser modificada o extendida para proporcionarlo. Mediante la inspección de las tablas que se ofrecen a continuación, es ya posible detectar inconsistencias y faltas en la arquitectura. Los errores que se detecten en esta fase pueden servir de guía para un futuro refinamiento de la arquitectura y como fuente de cuestiones y escenarios para la evaluación.

La inspección de las tablas es árida, pero no están pensadas para su lectura completa, sino como una guía de referencia rápida de las características de la arquitectura. Las tablas condensan una gran cantidad de información que, expresada de otro modo, extendería en exceso el capítulo y permiten identificar y ubicar rápidamente los patrones de diseño propuestos en la arquitectura.

Tabla 6.1: Responsabilidades e Interfaces Abstractas de los Subsistemas

Subsistema	Responsabilidades (Funcionalidad)	Info. Consumida	Info. Producida
GrpServer	▪ Representación gráfica de los mecanismos teleoperados y del entorno de operación.	Peticiones de Servicio de **RemoteCtlr**	Resultado del servicio.
CinServer	▪ Cálculo de la cinemática inversa (si existe) y del tiempo de movimiento. ▪ Cálculo de trayectorias libres de colisiones (Prevención de colisiones) ▪ Detección de colisiones. ▪ Programación y Simulación de movimientos.	Peticiones de Servicio de **RemoteCtlr**	Flag de Colisión (T, F) Cinemática Inversa Trayectorias (*path planning*) Tiempo de movimiento. Resultado del servicio.

RemoteUI	▪ Proporcionar acceso remoto a los mecanismos.
	  ▪ Interacción con el operador remoto.
	  ▪ Chequeo del rango de los parámetros y análisis sintáctico de los comandos.
	  ▪ Deshabilitación de comandos no seguros.
	  ▪ Actualización en tiempo real del estado del sistema (posiciones de los mecanismos, alarmas, advertencias, etc).
	  ▪ Soportar diferentes modos de funcionamiento
	  ▪ Solicitud de login y password.
		Infomación de estado del sistema propocionada por **RemoteCtlr**	Comandos para **RemoteCtlr**

CommSys	▪ Proporcionar comunicaciones seguras y eficientes entre nodos.	Comandos de **RemoteCtlr**. Información de estado del Sistema Local	Información del estado del enlace. Resultado de los servicios de comunicaciones.

RemoteCtlr	▪ Soportar los diferentes modos de operación.
	▪ Secuenciación y coordinación de misiones.
	▪ Chequeo de viabilidad de los comandos de usuario (incluyendo detección de colisiones).
	▪ Monitorización del estado del sistema incluyendo:
	  ▪ El resto de subsistemas.
	  ▪ Detección de fallos.
	  ▪ Ejecución de acciones de recuperación.
	▪ Monitorización del movimiento, incluyendo:
	  ▪ Petición de trayectorias a CinServer (*path planning*)
	  ▪ Petición de chequeo de colisiones.
	  ▪ Monitorización de alto nivel del movimiento (timeouts, condiciones de parada, etc ..)
		Comandos de usuario
		Respuestas de **CinServer**
		Respuestas de **GrpServer**
		Estado de los subsistemas.
		Estado de las comunicaciones.
		Estado de la unidad de control local.
		Estado de los mecanismos teleoperados.
		.
			Peticiones de servicio a **CinServer** y **GrpServer**
			Infromación de estado para **RemoteUI**
			Comandos para la Unidad de Control Local (gobierno de los mecanismos teleoperados)

Tabla 6.2: Subsistemas, Atributos de Calidad y Mecanismos Arquitectónicos.

`RemoteUI`: Atributos de Calidad y Mecanismos Arquitectónicos.

Atributo de Calidad	Mecanismo Arquitectónico
Modificabilidad	
▪ Portabilidad: *Cambio de plataforma y/o Sistema Operativo*	▪ Nivel de Portabilidad. ▪ Interfaz Abstracta de acceso a los servicios del S.0. ▪ Uso de estándares (POSIX)
▪ Capacidad de Distribución: *Ejecución de la IU en su propia plataforma*	Módulos de desacoplo. ▪ Si **`RemoteCtlr`** se ejecuta en otra CPU, **`UIDecoupler`** puede implementar o contener un proxy de **`RemoteUI`**. ▪ Es necesario registrar en **`RemoteUI`** la dirección de **`RemoteCtlr`**
▪ Adición de hardware/software de propósito especial *Botoneras específicas* *Joysticks especiales, reproducciones a escala de los dispositivos, botoneras específicas, etc* *Dispositivos con realimentación de pares y fuerzas*	▪ No se definen mecanismos explícitos. ▪ Dada la separación entre **`RemoteUI`** y el resto de subsistemas las modificaciones tienen un alcance local a **`RemoteUI`**.
▪ Modificaciones en misiones y mecanismos. *Adición/Modific./Eliminación de ventanas y botones.* *Cambio del layout de la IU.*	▪ El uso de toolkits y APIs comerciales facilita enormemente la definición de interfaces gráficas y su reorganización. ▪ Dada la separación entre **`RemoteUI`** y el resto de subsistemas las modificaciones tienen un alcance local a **`RemoteUI`**.
Rendimiento	
Las actualizaciones deben realizarse a un ritmo que asegure que la información presentada al operador representa el estado actual del sistema. *Los comandos deben ser enviados dentro de un plazo, por lo general muy pequeño, a* **`RemoteCtlr`**.	▪ **`UIDecoupler`** y **`UIListener`** puede soportar colas de mensajes organizadas por prioridades. ▪ Esquema de planificación por prioridades fijas (RMS). ▪ Pueden asignarse prioridades y plazos a **`UIDecoupler`** y **`UIListener`** y analizar el comportamiento temporal con RMA ▪ Si **`RemoteUI`** se ejecuta en su propia plataforma se considera el uso de enlaces de comunicaciones deterministas.
Seguridad	
Botones de parada segura fácilmente accesibles e identificables.	▪ Requisito No arquitectónico
Envío del comando de parada segura dentro de su plazo.	▪ Idem Rendimiento
Inhabilitación de comandos inseguros. *Modos de operación especiales (con acceso a diferentes conjuntos de comandos)*	▪ El uso de toolkits y APIs comerciales facilita enormemente la definición de interfaces gráficas y su reorganización.
Fiabilidad/Disponibilidad	

En caso de fallo debe estar disponible una interfaz degradada que proporcione acceso a los comandos más básicos.	▪ No hay un mecanismo claro que informe a **`RemoteCtlr`** de fallos en **`RemoteUI`**.
	▪ No hay un mecanismo que permita cambiar en tiempo de ejecución la interfaz a la que está ligado **`RemoteCtlr`**
	▪ El desacoplo entre **`RemoteUI`** y **`RemoteCtlr`** faclita, no obstante:
	  ▪ La definición de dichos mecanismos.
	  ▪ El reemplazo de una interfaz por otra

Si falla la interfaz degradada debe existir (siempre) una forma alternativa de realizar una parada segura.	▪ No se define ningún mecanismo explícito. ▪ Se considera el uso de un *Hard Stop Button* cableado directamente a la unidad de control local o a los accionadores de los mecanismos teleoperados.
Usabilidadad	
Aplican las mismas consideraciones que en la Seguridad y el Rendimiento, y además:	
Ayuda en línea. *Asistencia en la ejecución de misiones.*	No se definen mecanismos explícitos, no obstante: ▪ El uso de toolkits y APIs comerciales facilita enormemente la definición de interfaces gráficas y su reorganización. ▪ Dada la separación entre **`RemoteUI`** y el resto de subsistemas las modificaciones tienen un alcance local a **`RemoteUI`**.

Tabla 6.2 (Continuación)

`GrpServer`: Atributos de Calidad y Mecanismos Arquitectónicos.

Atributo de Calidad	Mecanismo Arquitectónico
Modificabilidad	
▪ Portabilidad: *Cambio de plataforma y/o Sistema Operativo*	▪ Nivel de portabilidad para servidores de gráficos hechos a medida. ▪ Si se implementan con COTS la portabilidad depende de las suposiciones arquitecturales que impliquen y de la infraestructura que necesiten.
▪ Capacidad de Distribución: *Ejecución de la* **`GrpServer`** *en su propia plataforma*	Módulos de desacoplo. ▪ Si **`RemoteCtlr`** se ejecuta en otra CPU, **`GrpDecoupler`** puede implementar o contener un proxy de **`GrpServer`**.
▪ Modificaciones en misiones y mecanismos. *Adición/Modificación/Eliminación de mecanismos (brazos, herramientas, vehículos)* *Modificación del entorno de operación.* *Adición/Eliminación de entornos de operación.*	▪ Servicios *Off-Line*: Modelado de entornos y mecanismos y adición de los mismos a la base de datos de **`GrpServer`**. ▪ *Servicios On-line:* Servicios de Configuración (carga y descarga de entornos y mecanismos y de ubicación de mecanismos en el entorno).
Rendimiento	
Las actualizaciones gráficas deben tener una calidad aceptable y realizarse a un ritmo que asegure que la información presentada al operador representa el estado actual del sistema.	▪ Esquema de planificación por prioridades fijas (RMS). ▪ Pueden asignarse prioridades y plazos a **`GrpDecoupler`** y analizar el comportamiento temporal con RMA ▪ Ejecución de **`GrpServer`** en su propia plataforma. En este caso se considera el uso de enlaces de comunicaciones deterministas.
Fiabilidad/Disponibilidad	
Muy dependiente de la aplicación.	**`GrpServer`** informa a **`RemoteCtlr`** del resultado del servicio. No se definen mecanismos específicos de recuperación en caso de fallo.

`CinServer`: Atributos de Calidad y Mecanismos Arquitectónicos.

Atributo de Calidad	Mecanismo Arquitectónico
Modificabilidad	
▪ Portabilidad: *Cambio de plataforma y/o Sistema Operativo*	▪ Nivel de portabilidad para servidores de gráficos hechos a medida. ▪ Si se implementan con COTS la portabilidad depende de las suposiciones arquitecturales que impliquen y de la infraestructura que necesiten.
▪ Capacidad de Distribución: *Ejecución de la* **`CinServer`** *en su propia plataforma*	Módulos de desacoplo. ▪ Si **`RemoteCtlr`** se ejecuta en otra CPU, **`CinDecoupler`** puede implementar o contener un proxy de **`CinServer`**.
▪ Modificaciones en misiones y mecanismos. *Adición/Modificación/Eliminación de mecanismos (brazos, herramientas, vehículos)* *Modificación del entorno de operación.* *Adición/Eliminación de entornos de operación.*	▪ Servicios *Off-Line*: Modelado de entornos y mecanismos y adición de los mismos a la base de datos de **`CinServer`**. ▪ *Servicios On-line:* Servicios de Configuración (carga y descarga de entornos y mecanismos y de ubicación de mecanismos en el entorno).
Rendimiento	
Los cálculos cinématicos y la detección de colisiones deben hacerse dentro de un plazo, que puede ser muy exigente en algunas aplicaciones. *Los resultados de* **`CinServer`** *deben estar disponibles para* **`RemoteCtlr`** *dentro de un plazo.*	▪ Esquema de planificación por prioridades fijas (RMS). ▪ Pueden asignarse prioridades y plazos a **`CinDecoupler`** y analizar el comportamiento temporal con RMA ▪ Ejecución de **`CinServer`** en su propia plataforma. En este caso se considera el uso de enlaces de comunicaciones deterministas.
Fiabilidad/Disponibilidad	
Muy dependiente de la aplicación.	**`CinServer`** informa a **`RemoteCtlr`** del resultado del servicio. No se definen mecanismos específicos de recuperación en caso de fallo.

Tabla 6.2 (continuación)

`CommSys`: Atributos de Calidad y Mecanismos Arquitectónicos.

Atributo de Calidad	Mecanismo Arquitectónico
Modificabilidad	
▪ Portabilidad: *Cambio de plataforma y/o Sistema Operativo* *Cambio de enlace y protocolos*	▪ Nivel de portabilidad. Si se implementan con COTS o Middleware la portabilidad depende de las suposiciones arquitecturales que impliquen y de la infraestructura que necesiten.
▪ Modificaciones en misiones y mecanismos. *Adición/Modificación/Eliminación de mensajes*	▪ Interfaces Abstractas. ▪ Los cambios en **`CommSys`** son locales, no afectando al resto de componentes.
Rendimiento	
Los mensajes deben entregarse dentro de un plazo. *Las comunicaciones deben ser predecibles y planificables*	Se considera el uso de protocolos de comunicaciones deterministas
Fiabilidad/Disponibilidad	
No pueden perderse mensajes silenciosamente *Los fallos en las comunicaciones deben ser detectados.* *Las comunicaciones deben ser recuperables*	Números de secuencia en los mensajes Redundancia en los mensajes (CRCs, complemento a 1, etc…) Chequeos periódicos del enlace Uso de protocolos orientados a conexión.

`RemoteCtlr`: Atributos de Calidad y Mecanismos Arquitectónicos.

Atributo de Calidad	Mecanismo Arquitectónico
Modificabilidad	
▪ **Portabilidad** *Cambio de plataforma y/o Sistema Operativo.*	▪ Nivel de Portabilidad. ▪ Uso de estándares (Se recomienda POSIX).
▪ Modificación de las Comunicaciones. *Cambio de los enlaces y/o protocolos de comunicaciones (incluyendo cambios en los mensajes).*	Módulos de desacoplo. Mientras se mantenga ***`Comm_Request`*** los cambios quedan localizados en **`ComDecoupler`** y **`ComListener`** .
▪ Modificaciones en la Unidad de Control Local *Cambios en los comandos de la Unidad de Control Local*	No se definen mecanismos específicos. ▪ El alcance de los cambios depende de cómo se estructure Controller y de las interfaces que puedan definirse.La división de RemoteCtlr en subsistemas puede contribuir a la localización de los cambios. ▪ Aunque los servicios de comunicaciones no cambien, habrá que implementar nuevos mensajes.
Modificaciones en la funcionalidad de la unidad de control local: *Incremento de la funcionalidad* *Disminución de la funcionalidad*	Cambios en **`Controller`**. ▪ Controller debe asumir las responsabilidades que ya no asume la unidad de control local. ▪ Eliminación de la funcionalidad que asume la unidad de control local.
▪ Capacidad de distribución: *RemoteCtlr se ejecuta en una plataforma distinta al resto de subsistemas*	Las clases de desacoplo ocultan a **`Controller`** los aspectos de distribución.
▪ Cambio en los requisitos de rendimiento.	Planificación por prioridades fijas (RMS)
▪ Modificaciones en misiones y mecanismos.	
Cambios en la estructura y cinemática de los mecanismos	Quedan confinados en **`GrpServer`** y **`CinServer`**.
Adición/Eliminación/Modificación de mecanismos (brazos, herramientas, vehículos)	Controladores (**`Controller`**) genéricos. Interfaces abstractas y parametrizables.
Adición/Eliminación de controladores *Modificación de los controladores* *Añadir/Eliminar/Modificar comandos, estatus, alarmas.* *Modificaciones en la máquina de estados del controlador.*	Controladores (**`Controller`**) genéricos. Interfaces abstractas y parametrizables. Se modifica la implementación de **`Controller`**, se mantiene la interfaz.
Cambios en los pasos de las misiones. *Cambios en la sioncronización de las misiones.*	Se modifica la implementación de **`Controller`**, se mantiene la interfaz.
Dada una combinación brazo-vehículo-herramienta: *Cambio del brazo portado por el vehículo.* *Cambio de la herramienta ensamblada al brazo*	El brazo, la herramienta y el vehículo tienen sus propios controladores (**`ArmCtlr`**, **`ToolCtlr`** y **`VehCtlr`**). Simplemente se sustituye un controlador por otro.

Tabla 6.2 (continuación)

`RemoteCtlr`: (continuación)

Disponibilidad/Fiabilidad	
Los controladores no deben verse bloqueados por el resto de subsistemas	Comunicaciones asíncronas. Módulos de desacoplo.
En caso de fallo en los controladores debe haber una forma de control alternativa	Acceso directo a la unidad de control local
Modos degradados de funcionamiento	No hay mecanismos claros
Rendimiento/Seguridad	
Los controladores deben realizar sus tareas dentro de un plazo.	Estructuración del controlador en tareas y RMS.

7 Evaluación de la Arquitectura

7.1 Introducción

En el capítulo 5 se han descrito los atributos de calidad de un subdominio muy amplio de los sistemas de teleoperación. En el capítulo 6 se han resumido las características más relevantes de la arquitectura de referencia definida en [Alvarez 1997]. Se trata ahora de evaluar hasta que punto la arquitectura cumple dichos atributos de calidad. Sin embargo, antes de entrar en materia hay que realizar dos observaciones, la segunda de ellas muy importante a la hora de juzgar la arquitectura descrita en [Alvarez 1997]:

- Los requisitos descritos en el capítulo 5, aunque no representan a todo el dominio de teleoperación, definen un subdominio muy amplio.
- Los requisitos originales de la arquitectura a evaluar no se corresponden con todos los requisitos definidos en el capítulo 5, sino con un subconjunto más bien reducido de los mismos.

Es decir, no se trata de evaluar la arquitectura respecto de sus requisitos originales, sino respecto de éstos y un nuevo conjunto de requisitos que se han ido añadiendo a medida que el grupo de investigación DSIE de la UPCT ha ido abordando nuevos desarrollos.

El método utilizado para la evaluación es el ATAM, ya descrito en el capítulo 4. Algunos de sus pasos ya han sido ejecutados en los capítulos anteriores y otros han debido adaptarse a las circunstancias en las que se ha desarrollado este trabajo de tesis. Por ello, conviene recapitular y comentar brevemente como se ha ejecutado el método. Los pasos de ATAM son:

1. Presentación del método.
2. Descripción de los factores de negocio.
3. Presentación de la arquitectura.
4. Identificación de los estilos arquitectónicos.
5. Generación del árbol de utilidad.
6. Análisis de los estilos arquitectónicos mediante la realización de los escenarios definidos en el árbol de utilidad.
7. Generación de escenarios adicionales.

8. Análisis de los estilos arquitectónicos mediante la realización de los nuevos escenarios.
9. Presentación de resultados.

Aunque los equipos de evaluación y desarrollo pueden ser el mismo, los pasos del método se definen considerando que ambos equipos van a ser diferentes y que todas las partes implicadas (equipo de desarrollo, clientes, usuarios, etc.), van a estar disponibles para resolver las dudas de los evaluadores. Ni una ni otra circunstancia se dan en este trabajo de tesis, en el que:

- El evaluador ha sido y es parte activa tanto en el proceso de desarrollo como en el de evaluación.
- La disponibilidad de los clientes y usuarios finales ha sido limitada, no habiendo sido posible reunirlos para realizar la evaluación. Sin embargo, se ha dispuesto de una gran cantidad de información procedente de los mismos, obtenida durante el desarrollo de cada una de las aplicaciones para las que se ha empleado o podría emplearse la arquitectura [SARPA 1995] [AAA 1995, 1996a, 1996b][TRON 1997b, 1997c], [Pastor et al 1996, 2000], [GOYA 1998a, 1998b], [EFCoR 2001].

Por todo ello, la ejecución de ATAM se ha centrado en:

1. La descripción de los factores de negocio (capítulo 5).
2. La descripción de la arquitectura, identificando los estilos arquitectónicos utilizados (capítulo 6).
3. La generación del árbol de utilidad (capítulo 7).
4. El análisis de los estilos arquitectónicos mediante las tablas de requisitos definidas en los capítulos 5 y 6 y la realización de los escenarios definidos en el árbol de utilidad.

El paso 1 (presentación del método) se realizó en el capítulo 4. Los pasos 7 y 8 no tienen sentido, puesto que requieren la contribución activa y conjunta de partes implicadas que no han podido reunirse. Esta ausencia ha sido paliada aprovechando la información obtenida en el desarrollo de los proyectos arriba mencionados e incorporándola al árbol de utilidad.

Según ATAM, estos resultados están constituidos por la identificación de los puntos sensibles y de compromiso y por la enumeración de las incertidumbres o riesgos asociados a las decisiones de diseño. Sin embargo, para el caso que nos ocupa, esta forma de presentar los resultados no es la más adecuada. Dada la amplitud del dominio, la cantidad de escenarios posibles es tan grande y variada que prácticamente todas las estructuras y conectores pueden considerarse puntos de compromiso. En estas circunstancias, es más útil seguir un enfoque más general y juzgar la arquitectura en función de su compleción, de la corrección de la división funcional que propone y de los estilos arquitectónicos empleados en la misma. En este sentido, un aspecto esencial apuntado por el ABD es la forma en que la arquitectura resuelve *las responsabilidades comunes* de sus componentes o subsistemas.

Algo parecido puede decirse del árbol de utilidad. Como se explicó en el capítulo 4, el árbol de utilidad define una taxonomía de los atributos de calidad en la que éstos se van refinando en las sucesivas ramas del árbol hasta llegar a las hojas, las cuales constituyen escenarios de evaluación concretos. La importancia relativa de cada escenario viene dada por su prioridad, la cual es función de dos componentes: la importancia relativa del escenario y el riesgo que implica su realización.

La ordenación de los escenarios según su prioridad es importante, ya que permite que el proceso de evaluación se centre en los aspectos claves de la arquitectura. Sin embargo, la asignación de prioridades no tiene sentido en este caso ya que, como consecuencia de la amplitud del dominio que se considera, todos los escenarios del árbol de utilidad son importantes. Dicho de otra manera, para cada uno de los escenarios que se proponen es fácil encontrar ejemplos de aplicación en los que tal escenario es crítico. Lo contrario también es cierto. Escenarios que son importantes para un sistema dado, carecen de toda relevancia en otros. Sin embargo, puesto que la arquitectura se propone para todos los sistemas de la familia, basta con que un escenario sea relevante para alguno de ellos.

Con todo ello, el capítulo se organiza de la siguiente manera:

- En el apartado 7.2 se define el árbol de utilidad. Por no recargar el capítulo con excesivos detalles, la realización de la mayor parte de los escenarios se describe con mayor profundidad en el anexo III, limitándose el apartado a destacar los aspectos más relevantes.
- En el apartado 7.3 se ofrece una evaluación global de la arquitectura, comentando su división funcional y su compleción y analizando los estilos arquitectónicos empleados.
- En el apartado 7.4 se evalúa de nuevo la arquitectura, pero en este caso no frente a los requisitos definidos en el capítulo 5, sino frente a sus requisitos originales y se describe el proceso que ha llevado a la definición de los nuevos requisitos.
- En el apartado 7.5 se analiza el comportamiento temporal de la arquitectura cuando el controlador debe invocar los servicios del subsistema de cálculos cinemáticos en línea para monitorizar la ejecución de un movimiento. Esta funcionalidad es necesaria en los sistemas que deban trabajar en entornos no estructurados.
- Por último, en el apartado 7.6 se presentan las conclusiones del capítulo, tanto en lo referente a la arquitectura, como al método, ATAM, en el que se ha basado la evaluación.

7.2 Árbol de Utilidad.

El árbol de utilidad definido para la evaluación de la arquitectura es el que se muestra en las tablas 7.1 a 7.3. Respecto de los escenarios que en él se definen conviene hacer algunas observaciones previas:

1. La clasificación de los escenarios es coherente con la clasificación de los atributos de calidad realizada en el capítulo 6. Todas las clasificaciones son discutibles, sin embargo para la evaluación lo realmente importante es el consenso acerca de los escenarios que han de considerarse y no tanto su clasificación, que puede realizarse según diferentes taxonomías.
2. En el caso de los escenarios de modificabilidad es difícil concretar las respuestas. Se supone que el escenario se realiza satisfactoriamente si los cambios a realizar no se propagan por la arquitectura y no conllevan una gran complejidad.

Tabla 7.1: Árbol de Utilidad: Modificabilidad			
Aspecto	**Escenarios**	**Respuestas Esperadas**	**Evaluación del escenario**
1.1- Portabilidad	**P1:** Cambio de WorkStation sobre UNIX a PC sobre W2K.	*Idealmente debe bastar con recompilar.* *La complejidad de los cambios debe ser pequeña y su número limitado.*	Satisfactoria
	P2: Cambio de Ethernet por CAN-BUS	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Satisfactoria
	P3: Inclusión de Middleware: Utilización de CORBA	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Insatisfactoria
1.2- Capacidad de Distribución	**CD1:** Ejecutar **CinServer** en otra plataforma.	*Recompilar sistema.*	Mediocre
	CD2: Ejecutar **LocalSys** y **TeleopSys** en lamisma plataforma.	*Recompilar sistema.*	Mediocre
1.3- Integración de Servicios	**IS1:** Adición de un nuevo servidor	*Definición de interfaz abstracta de acceso a los servicios.*	Satisfactoria
	IS2: Eliminación de **CinServer**	*Ningún efecto. Los sistemas deben poder construirse con los servicios que necesitan.*	Satisfactoria
	IS3: Sustitución de un servidor por otro (p.e: **CinServer** basado en ROBCAD por **CinServer** basado en GRASP)	*Ningún efecto.*	Satidfactoria
1.4- Adaptabilidad a cambios en la unidad de control local (LocalSys)	**ACL1**: **LocalSys** produce información de estado adicional.	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Mediocre
	ACL2: **LocalSys** produce menos información	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Mediocre
	ACL3: Aumenta la funcionalidad de **LocalSys**.	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Insatisfactoria
	ACL4: Disminuye la funcionalidad de **LocalSys**.	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Insatisfactoria
	ACL5: Cambia el formato de la información producida por **LocalSys,** pero no su semántica.	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Satisfactoria
	ACL6: Cambia el formato de los comandos admitidos por **LocalSys,** pero no su semántica.	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Satisfactoria
1.5- Adaptabilidad a cambios en los mecanismos y en las misiones	**AMM1**: Adición/eliminación/modificación de señales de sincronismo de **ArmController**	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Insatisfactoria
	AMM2: Adición/eliminación/modificación de información de estado producida por **ArmController**	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Insatisfactoria
	AMM3: Adición de un nuevo comando de **ArmController**	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Insatisfactoria
	AMM4: Adición de una nueva misión, implicando varios mecanismos.	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Insatisfactoria
	AMM5: Adición de una nueva herramienta.	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Satisfactoria
	AMM6: cambiar una herramienta por otra en tiempo de ejecución.	*Inmediato.*	Satisfactoria
	AMM7: cambia la cinemática y/o la estructura del brazo	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Satisfactoria
	AMM8: cambia del entorno operativo, de estructurado a estructurado	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Satisfactoria
	AMM9: cambia del entorno operativo, de estructurado a no estructurado	*La complejidad de los cambios debe ser pequeña y su número limitado.*	Insatisfactoria
1.6- cambios en el rendimiento	**AR1**: Se dobla el número de cálculos necesarios para obtener la transformada cinemática inversa.	*El sistema puede adaptarse a los cambios:*	Satisfactoria
	AR2: Se duplica la tasa de envío de estados de la unidad de control local	*El sistema puede adaptarse a los cambios:*	Satisfactoria

Tabla 7.2: Árbol de Utilidad: Rendimiento			
Aspecto	**Escenarios**	**Respuestas Esperadas**	**Evaluación del escenario**
Rendimiento	**R1**: Actualización de gráficos.	*Los gráficos se actualizan en tiempo real*	Satisfactoria
	R2: Actualización de interfaz de usuario.	*La interfaz de usuario se actualiza en tiempo real.*	Satisfactoria
	R3: Uso de **CinServer** para monitorizar en línea la ejecución de movimientos y detectar colisiones .	*El sistema es capaz de procesar toda la información que recibe.*	Satisfactoria
	R4: Entrega de comando a la unidad de control local.	*El comando se entrega en plazo.*	Insatisfactoria

Tabla 7.3: Árbol de Utilidad: Fiabilidad/Disponibilidada			
Aspecto	**Escenarios**	**Respuestas Esperadas**	**Evaluación del escenario**
Fiabilidad de los subsistemas. ▪ **Detección de fallos.** ▪ **Modos de operación degradados.** ▪ **Recuperación de fallos**	**FD1**: Falla un enlace de comunicaciones.	*El fallo se detecta. Se avisa al operador y* *a) el enlace se recupera automáticaticamente o* *b) el operador puede reestablecer el enlace.*	Satisfactoria
	FD2: Falla la interfaz de usuario.	*El fallo se detecta y hay una interfaz alternativa desde la que el operador puede ejecutar los comandos imprescindibles y rearrancar una nueva interfaz.*	Mediocre
	FD3: Falla la unidad de control local	*El fallo se detecta. Se avisa al operador y si es necesario se pasa al sistema a un estado de operación seguro.*	Mediocre
	FD4: Falla **RemoteCtlr**	*El fallo se detecta. Se avisa al operador y si es necesario se pasa al sistema a un estado de operación seguro.*	Mediocre
	FD5: Falla un servidor (**CinServer** o **GrpServer**).	*El fallo se detecta. Se avisa al operador y si es posible se prescinde del servidor. Si no, se pasa al sistema a un estado de operación seguro.*	Insatisfactoria

7.2.1 Escenarios de la Portabilidad: P1 .. P3.

P1: Cambio de WorkStation sobre UNIX a PC sobre Windows 98.
P2: Cambio de Ethernet por CAN-BUS
P3: Inclusión de Middleware: Utilización de CORBA

Mecanismos Arquitectónicos:

- Nivel de portabilidad: Interfaces abstractas de acceso al sistema operativo y a la infraestructura de comunicaciones.
- Se recomienda el uso de POSIX

Realización de los escenarios.
Satisfactoria, los cambios se localizan en el nivel de acceso a la infraestructura, no propagándose al resto de componentes.

No obstante, si los servicios que se integran en el sistema (p.e **`CinServer`** o **`GrpServer`**) son herramientas comerciales o se implementan por medio de COTS, la portabilidad depende de los recursos del sistema que dichas herramientas y componentes necesiten. No se identifican mecanismos para hacer frente a los problemas derivados del uso de COTS.

Observaciones:

El escenario P3 se comenta más extensamente en *Escenarios de la Integrabilidad*.

7.2.2 Escenarios de la Capacidad de Distribución: CD1, CD2.

CD1: Ejecutar **`CinServer`** en otra plataforma.
CD2: Ejecutar **`LocalSys`** y **`TeleopSys`** en la misma plataforma.

Mecanismos Arquitectónicos:

- Módulos de desacoplo entre subsistemas que pueden encapsular los aspectos de distribución (patrón proxy). Sin embargo, la arquitectura no explica la forma en que deben encapsularse los servicios de comunicaciones en los módulos de desacoplo.

Realización de los escenarios
La realización de los escenarios es mediocre, principalmente porque los mecanismos de distribución no se definen con claridad.

La arquitectura describe en un mismo nivel de abstracción la infraestructura de comunicaciones y los susbsistemas que pueden hacer o no hacer uso de la misma, dependiendo de su distribución. Así, en la figura 6.5 se establece una relación explícita entre el controlador remoto (**`RemoteCtrl`**) y la infraestructura de comunicaciones (**`CommSys`**) a través de la interfaz ***`LocalCmds`***. Este enfoque lejos de clarificar la relación de los subsistemas con la infraestructura de comunicaciones, la obscurece aún más, ya que:

1. Asume explícitamente que los controladores **`RemoteCtrl`** y **`LocalCtrl`** están en diferentes plataformas, lo cual es frecuente, pero no estrictamente necesario.

2. No se establece la misma relación con el resto de subsistemas, lo cual es asumir implícitamente que los subsistemas servidores (**`CinServer`** y **`GrpServer`**) y **`RemoteCtlr`** se ejecutan siempre en la misma plataforma (lo cual no es cierto) o que los aspectos de distribución de los servidores se resuelven mediante otra infraestructura de comunicaciones (que no está descrita).

Observaciones.

A pesar de todo, la encapsulación de los aspectos de comunicación en los módulos de desacoplo hace posible la definición de tales mecanismos sin afectar al resto de la arquitectura.

Otros aspectos del problema (uso de middleware) se comentarán en los *Escenarios de la Integrabilidad*.

Estos escenarios se refieren a la capacidad de distribución de subsistemas completos. Otros aspectos más refinados de la capacidad de distribución se discuten en *Escenarios de Adaptabilidad a cambios de la Unidad de Control Local*.

7.2.3 Escenarios de la Integrabilidad: IS1 .. IS4.

IS1: Adición de un nuevo servidor.
IS2: Eliminación de **`CinServer`**
IS3: Sustitución de un servidor por otro (**`CinServer1`** por **`CinServer2`**)
IS4: Inclusión de Middleware: Utilización de CORBA.

Mecanismos Arquitectónicos:

- Módulos de desacoplo entre subsistemas que ofrecen interfaces abstractas de acceso a los servicios.

Realización de los escenarios.

IS1, IS2 e IS3: Satisfactoria, siempre y cuando la interfaz se defina correctamente. Si hubiera que modificar las interfaces los cambios se propagarían por todo el sistema.

Insatisfactoria para IS4, debido como en el caso de los escenarios de distribución a la poca claridad con la que se define la relación de los subsistemas con la infraestructura de comunicaciones. CORBA asumiría las responsabilidades de los módulos de desacoplo y del subsistema de comunicaciones. Pero como los mecanismos internos de tales módulos no están bien definidos, no es fácil determinar el número de modificaciones necesarias ni su complejidad.

Observaciones:

Otro asunto es determinar si las actuales implementaciones de CORBA cumplen con los requisitos de calidad exigibles a los sistemas de teleoperación. Pero no es eso lo que se juzga.

7.2.4 Escenarios de adaptabilidad a cambios en la Unidad de Control Local: ACL1 .. ACL5.

ACL1: **`LocalSys`** produce información de estado adicional.

ACL2: **`LocalSys`** produce menos información de estado.
ACL3: Aumenta la funcionalidad de **`LocalSys`**.
ACL4: Disminuye la funcionalidad de **`LocalSys`**.
ACL5: Cambia el formato de la información producida por **`LocalSys`,** pero no su semántica.
ACL6: Cambia el formato de los comandos admitidos por **`LocalSys`,** pero no su semántica.

Mecanismos Arquitectónicos:

- Interfaz abstracta ***`LocalCmds`*** de acceso a los comandos de ***`LocalSys`***.
- Controladores genéricos, con interfaces abstractas y parametrizables.
- No se definen mecanismos específicos para extender o recortar la funcionalidad de los controladores. No se refinan suficientemente los diversos aspectos de la funcionalidad.
- No se definen mecanismos específicos para trasladar funcionalidad de **`RemoteCtrl`** a **`LocalCtrl`** o viceversa.

Realización de los escenarios.

ACL1, ACL2: Mediocre.
ACL3, ACL4: No satisfactoria.
ACL5, ACL6: Satisfactoria.

Los escenarios ACL1 a ACL4 dan lugar a cambios que se propagan por toda la estructura del sistema, siendo necesario modificar un gran número de interfaces e implementaciones (véanse detalles en el anexo III). Sin embargo, ACL1 y ACL2 por una parte y ACL3 y ACL4 por otra se refieren a conceptos diferentes y puntúan de forma diferente.

ACL1 y ACL2 no implican una modificación de las responsabilidades de la unidad de control local, sino de la complejidad inherente a dichas responsabilidades. La unidad de control local correspondiente a un robot con 6 grados de libertad y cinemática inversa será mucho más compleja que la correspondiente a una mesa cruzada con dos ejes. Sin embargo, las responsabilidades que la arquitectura asigna a ambas unidades son conceptualmente las mismas: actuación directa y sensorización. Estos escenarios están relacionados con los cambios que pueden sufrir los mecanismos durante su desarrollo y su vida operativa y con la posibilidad de reutilizar el software de la unidad de control o alguna de sus partes en diferentes sistemas.

La puntuación es mediocre. Los controladores de la unidad de teleoperación (**`ArmCtlr`**, **`ToolCtlr`** y **`VehCtrl`**) son instancias de un controlador genérico, uno de cuyos parámetros es el tipo de datos que reciben de **`LocalSys`**. Sin embargo, no se definen mecanismos para extender o recortar la funcionalidad de los controladores y **`BufCtrl`** es sensible a los cambios, eliminaciones y adiciones de datos.

Los escenarios ACL3 y ACL4 tienen que ver con la división de responsabilidades entre las unidades de teleoperación y de control local. La arquitectura asume implícitamente que la funcionalidad de la unidad de control local es la misma para todos los sistemas de la familia y se reduce a tareas de muy bajo nivel como el control de los servos (actuación) y la lectura de los sensores. Al partir de esta premisa, la arquitectura no define mecanismos para trasladar funcionalidad de una unidad a otra en función de las características de los sistemas. Sin embargo, esta suposición es falsa ya que:

1. En algunos sistemas, la información del entorno da lugar a un comportamiento reactivo que puede, y a veces debe, resolverse en la unidad de control local. Las máquinas de estados que modelan este comportamiento no son siempre triviales.

p.e: sincronización de movimiento del robot y funcionamiento de una herramienta de soldadura.

p.e: limpieza automática de un paño de casco de buque (sistema GOYA [GOYA 1998]). Una vez iniciada la operación los movimientos del robot y la activación de la lanza de chorreo no dependen de las ordenes del operador, sino que vienen determinadas por una máquina de estados cuyas entradas están constituidas por los valores proporcionados por los sensores.

2. La funcionalidad puede asignarse a una u otra unidad en función de sus requisitos temporales, del patrón de comportamiento (reactivo o no reactivo) y de las características de las plataformas de teleoperación y local. Todos estos aspectos pueden evolucionar durante la vida operativa de un sistema.

Observaciones:

El número de escenarios relacionados con cambios en la unidad de control local es proporcionalmente muy elevado debido a que las suposiciones arquitectónicas referentes a la misma no son completamente ciertas. Por ello, los aspectos de distribución entre ambas plataformas deben estudiarse con cierto detalle.

7.2.5 Escenarios de Adaptabilidad a cambios en los mecanismos y en las misiones: AMM1 .. AMM9.

AMM1: Adición/eliminación/modificación de señales de sincronismo de **`ArmController`**
AMM2: Adición/eliminación/modificación de información de estado producida por **`ArmController`**.
AMM3: Adición de un nuevo comando de **`ArmController`**.
AMM4: Adición de una nueva misión, implicando varios mecanismos.
AMM5: Adición de una nueva herramienta.
AMM6: Cambio de una herramienta por otra en tiempo de ejecución.
AMM7: Cambia la cinemática y/o la estructura del brazo.
AMM8: Cambia del entorno operativo, de estructurado a estructurado.
AMM9: Cambia del entorno operativo, de estructurado a no estructurado.

Mecanismos Arquitectónicos:

- Encapsulación de las propiedades estructurales y cinemáticas en **`CinServer`** y **`GrpServer`**.
- Controladores genéricos con interfaces abstractas y parametrizables.
- No se definen mecanismos específicos para trasladar funcionalidad de **`RemoteCtrl`** a **`LocalCtrl`** o viceversa.
- Separación de los controladores correspondientes a cada uno de los mecanismos (brazo, herramienta y vehículo).
- Interfaces abstractas de sincronismo: ***`ArmSynch`***, ***`ToolSynch`***, ***`VehSync`***.
- No se definen componentes que resuelvan globalmente los aspectos de sincronización.

Realización de los escenarios.

AMM1, AMM2, AMM3, AMM4: No satisfactoria.
AMM5, AMM6, AMM7, AMM8: Satisfactoria.

AMM9: No satisfactoria.

Cualquier modificación que pueda implicar un cambio de los patrones de sincronización entre los diferentes controladores de **`RemoteCtlr`** es soportada de forma muy deficiente por la arquitectura, ya que implica una cascada de cambios (Véase anexo III), que implican tanto a las interfaces de sincronismo de los controladores como a su implementación (máquinas de estados). Este escenario se considera explícitamente en AMM1, pero puede reproducirse cada vez que se añade un nuevo comando (escenario AMM3) o se realizan cambios en las misiones (AMM4).

Los escenarios que suponen cambios en la información de estado recibida por cualquiera de los controladores (escenario AMM2) o modificaciones en sus comandos (escenario AMM3) tampoco se realizan satisfactoriamente. La definición de controladores genéricos con interfaces parametrizables no es suficiente, ya que:

1. Protege a los controladores de cambios en su interfaz, pero no al resto de los subsistemas.

2. No es fácil (si es que es posible) diseñar máquinas de estados tan genéricas que puedan adaptarse a cualquier cambio de comportamiento, ya que la complejidad de dichas máquinas es tanto mayor cuanto mayor es el número de sus entradas y tipos de comportamiento frente a los que debe parametrizarse. El mecanismo de los genéricos es bueno en teoría, pero no es fácil de llevar a la práctica cuando se consideran unidades tan grandes como los controladores.

Los escenarios AMM5 a AMM8 se realizan razonablemente bien (detalles en anexo III). En general la arquitectura se comporta mal frente a la modificación de los mecanismos existentes o de sus patrones de interacción y bien frente a la adición de nuevos mecanismos siempre que estos no impliquen cambios en los patrones de interacción. De la misma manera, la adición o modificación de misiones es fácil o difícil según implique o no cambios de sincronismo.

El escenario AMM9 está estrechamente relacionado con:

- ✓ La utilización de ciertos servicios en línea, como por ejemplo utilidades de visión artificial (Escenario R4).
- ✓ Con la sensorización de los dispositivos y por tanto, aunque indirectamente, con un incremento de los patrones de interacción reactivos (Escenario AMM1, Escenarios ACL3 y ACL4).

Puesto que la arquitectura no se comporta bien frente a ninguno de estos aspectos tampoco es fácilmente adaptable a entornos no estucturados.

Observaciones:

El concepto de comando (básico, compuesto o automático) está bien caracterizado en la arquitectura, pero no el de misión. En consecuencia, y puesto que las misiones pueden involucrar a más de un controlador, tampoco están bien resueltos los problemas de sincronismo. Si eliminamos los sistemas que presentan comportamiento reactivo y restringimos los posibles patrones de interacción entre controladores a dos o tres casos sencillos, la arquitectura podría realizar de forma satisfactoria todos los escenarios.

7.2.6 Escenarios de Adaptabilidad a cambios en los requisitos de rendimiento: AR1, AR2.

AR1: Se dobla el número de cálculos necesarios para obtener la transformada cinemática inversa.
AR2: Se duplica la tasa de envío de estados de la unidad de control local

Mecanismos Arquitectónicos:

- Estructuración de la aplicación en procesos independientes (uno por subsistema).
- Estructuración de los procesos en threads que pueden agruparse o separarse según las necesidades.
- Asignación de prioridades fijas. Planificación RMS (*Rate Monotonic Scheduling*).
- Posibilidad de utilizar enlaces de comunicaciones dedicados, de alta capacidad y determinísticos.

Realización de los escenarios.

Satisfactoria en ambos casos. Los escenarios pueden realizarse reestructurando tareas, y aumentando la capacidad de las plataformas y de los enlaces de comunicaciones. Además, la política de planificación (RMS) puede analizarse manualmente o con herramientas (análisis RMA), si se utilizan enlaces deterministas permite modelar la transmisión de mensajes como una tarea más y es estable (si el sistema no tiene capacidad suficiente se pierden los plazos de la tareas menos prioritarias). En [Alvarez et al 1998] se realiza un estudio sistemático de estos aspectos.

Observaciones:

La planificación mediante RMA tiene sin embargo algunos inconvenientes:

1. No optimiza recursos.
2. Supone la existencia de un sistema operativo que soporte prioridades y desalojo.
3. Requiere conocer cómo se *mapean* los *threads* de la aplicación en *threads* del sistema operativo.
4. En sistemas operativos que no sean de tiempo real es muy difícil estimar el tiempo de ejecución de las tareas.
5. No es el más apropiado para los procesos de la plataforma de teleoperación, donde la pérdida eventual de algún plazo no suele suponer grandes problemas.

Parece más apropiado restringir el uso de RMS a las tareas de más bajo nivel, que suelen ejecutarse en la unidad de control local, dotando a ésta de un sistema operativo de tiempo real.

7.2.7 Escenarios del Rendimiento: R1 .. R4.

R1: Actualización de gráficos.
R2: Actualización de interfaz de usuario.
R3: Entrega de comando a la unidad de control local.
R4: Uso de `CinServer` para monitorizar en línea la ejecución de movimientos y detectar colisiones.

Mecanismos Arquitectónicos:

- Estructuración de la aplicación en procesos independientes (uno por subsistema).
- Estructuración de los procesos en threads que pueden agruparse o separarse según las necesidades.
- Asignación de prioridades fijas. Planificación RMS (*Rate Monotonic Scheduling*)
- Posibilidad de utilizar enlaces de comunicaciones dedicados, de alta capacidad y determinísticos.

Realización de los escenarios.
R1, R2, R3: Satisfactoria.
R4: Mediocre-Insatisfactoria.

Respecto de R1, R2 y R3 pueden hacerse los mismos comentarios que para los *Escenarios de Adaptabilidad a Cambios en los requisitos de Rendimiento*.

El escenario R4 supone un gran trasiego de mensajes a través de la arquitectura, ya que todos ellos pasan a través de **`RemoteCtrl`**, y una gran sobrecarga de la plataforma en la que se ejecute **`CinServer`**. En general, la concordancia entre la posición real del robot real enviada desde la unidad de control local y la posición esperada puede realizarse a través de tablas precalculadas, proporcionadas por **`CinServer`**, si éste está disponible en el sistema.

Sin embargo, la generalización de este escenario representa todos los casos en los que **`RemoteCtlr`** debe invocar servicios en línea. No en todos los casos es posible precalcular los estados esperados. Imaginemos una variación de este escenario, en la que el servidor considerado es un sistema de visión artificial que calcula la siguiente consigna de movimiento en función de su interpretación del entorno. El escenario tiene sentido, ya que los entornos no estructurados están explícitamente considerados en los requisitos. En este caso no es posible precalcular nada, ya que los resultados deben obtenerse a partir de la información actual.

El escenario R4 se realiza en el apartado 7.5 a través del estudio de dos diagramas de colaboración, uno representando el esquema de interacción propuesto por la arquitectura y el otro un esquema de interacción alternativo.

Además, la planificación de las tareas de los servidores es difícil de realizar si éstos están constituidos por herramientas comerciales, sobre las cuáles se tiene muy poco control, si es que se tiene alguno. A menudo, la planificación debe basarse en estimaciones dudosas y en mediciones previas, cuyas condiciones de ejecución no son fácilmente repetibles.

7.2.8 Escenarios de la Disponibilidad: FD1 .. FD5.

FD1: Falla un enlace de comunicaciones.
FD2: Falla un servidor (**`CinServer`** o **`GrpServer`**)
FD3: Falla la interfaz de usuario.
FD4: Falla la unidad de control local.
FD5: Falla **`RemoteCtlr`**.

Mecanismos Arquitectónicos:

- Redundancia en los mensajes (números de secuencia, CRC).
- Chequeo periódico de los enlaces de comunicaciones.
- Uso de protocolos de comunicaciones orientados a conexión.

- Los servidores (**`GrpServer`**, **`CinServer`** u otros que pudieran añadirse) informan a **`RemoteCtlr`** del resultado del servicio. No se definen mecanismos de recuperación en caso de fallo.
- No se identifican mecanismos explícitos para detectar y recuperar los fallos de la interfaz de usuario.
- Todos los servicios de monitorización están centralizados en **`RemoteCtlr`**. Pero no se identifican mecanismos para detectar o recuperar fallos en el mismo.

Realización de los escenarios.
FD1: Satisfactoria.
FD2, **FD3, FD4**: Mediocre.
FD5: Insatisfactoria.

El único escenario que se realiza de forma satisfactoria es FD1 (Fiabilidad de las comunicaciones). Se proporcionan mecanismos para detectar el fallo y pueden habilitarse estrategias de recuperación en dos niveles, localmente en **`CommSys`** y globalmente en **`RemoteCtlr`**.

La realización de FD2 (fallos en servidores) y FD3 (fallos en la interfaz de usuario) y FD4 (fallos en unidad de control local) es mediocre. Se proporcionan algunos mecanismos para detectar los fallos, pero no de recuperación. Los únicos fallos de la unidad de control local que pueden detectarse son errores en los cálculos cinemáticos, mediante la comparación del estado enviado desde la Unidad de Control Local con los resultados de los cálculos de **`CinServer`**. Sin embargo, no se proporcionan mecanismos para distinguir cual de los datos es correcto. Los fallos en la interfaz de usuario pueden pasar desapercibidos (véase discusión en anexo III) y no se identifica un mecanismo claro para la utilización simultánea de una interfaz básica alternativa o para su arranque en caso de fallo de la principal. La definición de la interfaz como un subsistema independiente facilita, no obstante, la definición de tales mecanismos.

La realización del escenario FD5 (fallos de **`RemoteCtlr`**) es claramente insatisfactoria. Todos los servicios de monitorización se centralizan en dicho susbsistema, sin embargo no se definen explícitamente mecanismos para la detección y recuperación de fallos en el mismo. **`RemoteCtlr`** debe ser refinado en este aspecto.

Observaciones:

Las relaciones entre los diversos controladores de **`RemoteCtlr`** (**`ArmCtlr`**, **`ToolCtlr`** y **`VehCtlr`**) son de alguna ayuda en este aspecto. Cada controlador puede percibir fallos en los demás a través de las invocaciones a sus interfaces de sincronismo. Sin embargo, no se define nada explícito en este aspecto.

7.3 Resultados de la Evaluación.

Este apartado se divide en varias secciones, en las cuales se abordan respectivamente la división funcional propuesta por la arquitectura, su forma de abordar las responsabilidades comunes y los estilos arquitectónicos empleados.

7.3.1 División funcional, compleción y responsabilidades comunes.

Los objetivos del presente apartado son:

1. Verificar la compleción de la arquitectura[59], comprobando que todos los requisitos funcionales abstractos definidos en el capítulo 5 y resumidos en la tabla 5.3 han sido asignados a algún subsistema. Esta verificación puede realizarse comparando la mencionada tabla 5.3 con la tabla 6.2 en la que se resumen las responsabilidades asignadas a los diferentes subsistemas definidos por la arquitectura.
2. Razonar sobre la distribución de la funcionalidad en la arquitectura considerando:
   - La independencia funcional de los módulos y susbsistemas (principios de alta cohesión y bajo acoplamiento).
   - Los servicios que proporcionan y solicitan.

   La simple inspección de la descripción de más alto nivel de la arquitectura (figuras 6.1, 6.3 y 6.4) y de las responsabilidades e interfaces abstractas de sus subsistemas (tabla 6.1) permite razonar eficientemente sobre estos extremos. No obstante, en algunos casos ha sido necesaria la definición y realización de escenarios de evaluación.
3. Analizar como se resuelven las responsabilidades comunes. Estas responsabilidades tienen que ver con las plantillas definidas en el ABD y tal y como se explicó en el capítulo 4 se refieren a la forma en que los componentes se relacionan entre sí y con la infraestructura y a la forma en que se abordan aquellos *aspectos* que son comunes a todos los componentes.

Como se explicó en el capítulo 6 y se deduce inmediatamente de las interfaces de los subsistemas y de la división de responsabilidades entre los mismos (tablas 6.1 y 7.4), la arquitectura propone una organización fuertemente jerárquica en la que **`RemoteCtlr`** asume la mayor parte de las responsabilidades, toma la práctica totalidad de las iniciativas de control y canaliza o genera toda la información que fluye por el sistema. El control fluye desde los niveles más altos hasta los más bajos, siendo inexistentes las interacciones horizontales entre componentes o subsistemas de un mismo nivel.

[59] Recuérdese: No respecto de sus requisitos originales, sino respecto de los definidos en el capítulo 5 y en el anexo II.

El comportamiento de este tipo de arquitecturas está descrito en la literatura [Coste et al 2000] [Shaw 1996a] y los escenarios realizados confirman lo que ésta dice. Su punto fuerte es que facilitan el razonamiento de alto nivel y el seguimiento de los flujos de datos y de control a través de sus estructuras. Facilita asimismo la definición de interfaces y la composición de subsistemas complejos a partir de otros más simples. Por el contrario, son arquitecturas poco flexibles que se adaptan con mucha dificultad a comportamientos reactivos.

La tabla 7.4 se ha construido a partir de las responsabilidades abstractas definidas en la tabla 5.2 y de la asignación de responsabilidades a subsistemas descrita en la tabla 6.1. En esta tabla puede observarse que la mayor parte de las responsabilidades *propias del dominio* han sido explícitamente consideradas. En cualquier caso, la división funcional es bastante consistente y si surgiera algún nuevo requisito perteneciente a lo que podría denominarse *funcionalidad básica del dominio* puede determinarse el subsistema al que debe ser asignado. Las interfaces de los sistemas son coherentes con la división funcional propuesta y sus detalles de funcionamiento, resumidos en el capítulo 6, están suficientemente descritos en [Alvarez 1997].

No obstante, se nota la ausencia de subsistemas importantes. Ninguno de los subsistemas definidos proporciona los servicios necesarios para configurar la aplicación, arrancar los otros subsistemas y pasar a los mismos la información necesaria para que trabajen de acuerdo con la configuración elegida. En principio, parece que el propio controlador, **`RemoteCtlr`**, es el subsistema que asume estas funciones, pero la arquitectura es ambigua en este aspecto. Puesto que **`RemoteCtlr`** conoce y maneja la información procedente del resto de los subsistemas, parece que es responsabilidad suya. Sin embargo, la funcionalidad asociada a la configuración y arranque de la aplicación es un aspecto alejado de las responsabilidades básicas de **`RemoteCtlr`**. Entre las interfaces abstractas que se definen en la tabla 6.1 no aparece ninguna que pueda asociarse a la *configurabilidad*, salvo para los controladores, pero sin definir ningún subsistema que acceda a la misma. Tampoco se explica convenientemente la secuencia de arranque de la aplicación, que de alguna manera esta relacionada con estos aspectos, los cuales parece que se dejan a la libre elección de los usuarios de la arquitectura (lo que no deja de ser una contradicción).

Obsérvese que por el momento no se han tenido en cuenta lo que podríamos llamar *responsabilidades transversales*, es decir, aquellas que no tienen tanto que ver con lo que hacen los sistemas como con cómo lo hacen. Tal es el caso de la fiabilidad y de la testeabilidad. Este tipo de responsabilidades o *aspectos* están presentes en todos los subsistemas y son de hecho parte de las *responsabilidades comunes* de las que se hablaba al comienzo de este apartado. Tampoco se ha tenido en cuenta si está distribución es la más adecuada para alcanzar los atributos de calidad esperados.

Tal vez la mayor debilidad de la arquitectura sea su falta de mecanismos para tratar aquellos aspectos de la funcionalidad relacionados con las *responsabilidades comunes*. Así:

- No se definen interfaces de diagnóstico que permitan chequear el funcionamiento de los componentes, lo cual compromete la testeabilidad y la integrabilidad de nuevos componentes.
- No se definen interfaces para la instalación o configuración de subsistemas o componentes (excepto para los controladores), aunque la substitución de los mismos o parte de ellos se considera explícitamente entre los requisitos (también en los originales).
- No se define un mecanismo de tratamiento de excepciones, aunque se recomienda su empleo. La organización de la arquitectura sugiere un tratamiento centralizado en **RemoteCtlr**, pero no lo prescribe.

- No se describe convenientemente la relación con la infraestructura. Las recomendaciones sobre el uso de POSIX deberían refinarse más. La portabilidad respecto de diferentes sistemas operativos es importante, sin embargo no todos soportan el estándar. Tampoco se describe como configurar los subsistemas frente a diferentes infraestructuras, ni se explica como extender el nivel de portabilidad.

Tabla 7.4: Asignación de responsabilidades a subsistemas.

Requisitos funcionales abstractos (Tabla 5.2)		**Asignación de responsabilidades a subsistemas (Tabla 6.1)**
Requisito	**Subrequisito**	**Subsistema Responsable**
Soportar modos de operación.	Todos	**RemoteCtlr**
Chequeo Viabilidad Comandos de Operador.	Rango parámetros	**RemoteUI**
	Deshabilitación comandos no seguros	**RemoteCtlr**
Monitorización constante del estado del sistema	Estado de los mecanismos	A bajo nivel (sensorización) : **LocalCtlr** A alto nivel: **RemoteCtrl**
	Control movimiento	A bajo nivel (bucles servos): **LocalCtlr** A alto nivel: **RemoteCtrl**
	Detección de colisiones	A bajo nivel (sensores): **LocalCtlr** A alto nivel: **RemoteCtlr**, a través de los servicios de **CinServer**.
	Prevención de colisiones	**RemoteCtlr**, a través de los servicios de **CinServer**.
	Funcionamiento componentes hardware y software.	A bajo nivel (sensorización) : **LocalCtlr** A alto nivel: **RemoteCtrl** De las comunicaciones: **CommSys**, **RemoteCtlr**
	Información al operador estado sistema	**RemoteCtlr**
Acceso a los mecanismos	Acceso remoto a los mecanismos	**RemoteCtlr** **CommSys**
	Enlaces de comunicaciones eficientes y fiables	**CommSys**
Secuenciación de misiones. Coordinación y Sincronización de mecanismos	Programación y realización de secuencias de operaciones y operaciones automáticas (simples o compuestas)	**RemoteCtlr**
	Sincronización/Coordinación de mecanismos	**RemoteCtlr**
Configuración de la aplicación	Todos	No está claro. En principio, **RemoteCtlr**, pero **RemoteCtlr** debe arrancar con alguna configuración.
Servicios comunes	Representación gráfica	**RemoteIU** y/o **GrpServer**
	Cálculos cinemáticos	**LocalCtlr** y/o **CinServer**
	Cálculo de trayectorias, detección de colisiones.	**CinServer**
	Simulación de misiones	**CinServer**, **GrpServer**.
	Visión artficial/sistema de navegación	El servidor correspondiente.
Seguridad y logging	Todos	**RemoteUI**
Relación con operador	Todos	**RemoteUI**
Testeabilidad	Todos	Todos, pero no se explican mecanismos.

- Puesto que **`LocalSys`** y **`RemoteCtlr`** comparten a diferente nivel las mismas responsabilidades es importante que ambos tengan una visión coherente de los mecanismos que controlan. Para que esto sea posible es necesario que el modelo de diseño establezca relaciones entre las estructuras de datos que ambos sistemas manejan. Por ejemplo (pero no exclusivamente):
  - ✓ Modelos cinemático y dinámico (ambos sistemas deben considerar las mismas transformadas).
  - ✓ Modelo de comandos. Es necesario describir modelos que permitan estrategias para generar los comandos de **`LocalSys`** a partir de los comandos de **`RemoteCtlr`**.

Aunque en el modelo de análisis ([Alvarez 1997], resumido en anexo I), se consideran estos aspectos, no se trasladan suficientemente al modelo de diseño.

- Este mismo problema puede generalizarse a las relaciones entre **`RemoteCtlr`** y los subsistemas servidores y la interfaz de usuario.
- Los módulos de desacoplo pueden asumir más o menos responsabilidades, desde simples adaptadores a proxys, y pueden implementar al menos tres patrones de interacción distintos (figuras 6.6, 6.7 y 6.8), sin embargo tampoco se definen mecanismos o interfaces para configurarlos. Además, como los módulos de desacoplo se definen internamente a los controladores cualquier cambio en los patrones de interacción se propaga a los mismos.
- No se describe en que forma pueden integrarse componentes comerciales, salvo que estos sean subsistemas completos independientes del resto de la aplicación (p.e: **`CinServer`** y **`GrpServer`**).

En resumen, no se definen (al menos no suficientemente) mecanismos, estrategias y subsistemas para gestionar o configurar la mayor parte de las responsabilidades comunes. Como por su propia naturaleza, estas responsabilidades se esparcen por todo el código y afectan a todos los susbsistemas se hace muy difícil la definición de componentes reutilizables. Si estos aspectos se dejan abiertos, pueden resolverse de forma diferente en cada sistema haciendo imposible que dos sistemas de la familia compartan componentes con la misma *funcionalidad básica*.

7.3.2 Evaluación de los estilos arquitectónicos

La simple inspección de la tabla 6.2 permite detectar la ausencia de mecanismos arquitectónicos asociados a ciertos atributos de calidad. Algunas de las ausencias están relacionadas con la falta de estrategias para resolver las responsabilidades comunes comentadas en la sección anterior. Así:

- No se definen mecanismos ni estrategias para el tratamiento de excepciones.
- No se definen mecanismos para proporcionar tolerancia a fallos.
- No se definen mecanismos claros para el diagnóstico de componentes.
- No se definen mecanismos para la integración de COTs.
- No se definen mecanismos ni estrategias asociados a la interoperabilidad.

No obstante, la organización jerárquica de la arquitectura puede en algunos casos facilitar la definición de tales mecanismos. Algunos de los mecanismos no se definen con el suficiente detalle, en parte debido a la ausencia de los anteriores. Así:

- Los mecanismos para la detección y recuperación de fallos no son suficientes (Véanse escenarios FD1 a FD5).
- Los mecanismos de configuración son insuficientes:
  - ✓ No hay un subsistema encargado de gestionar los cambios de configuración.
  - ✓ Los componentes genéricos se refieren a componentes muy grandes, que pueden asumir muchas responsabilidades (los controladores) y por tanto muy difíciles de implementar y parametrizar en la práctica con el grado de generalidad requerido (Escenarios AMM1 a AMM4).

- La relación con la infraestructura de comunicaciones no está convenientemente descrita, lo que limita las posibilidades de los módulos de desacoplo para proporcionar capacidad de distribución y las posibilidades de integrar *middleware* comercial (Escenario IS4).

En otros casos, los estilos y/o mecanismos arquitectónicos empleados no realizan satisfactoriamente los escenarios. Así:

- Es muy difícil incrementar la funcionalidad del sistema de control local, ya que no se contempla la posibilidad de que éste asuma más responsabilidades que las que originalmente le asigna la arquitectura (Escenarios ACL1 a ACL4). Esta suposición no es coherente con la amplitud del dominio considerado ni con la posibilidad de incluir comportamiento reactivo.
- El esquema de sincronización entre los controladores dificulta la adaptabilidad a nuevos comandos o misiones ya que los cambios de los patrones de sincronización se propagan por todos los controladores (Escenarios AMM1 a AMM4).
- La relación entre **`RemoteCtlr`** y los servidores puede suponer en algunos casos la sobrecarga del sistema, comprometiendo el cumplimiento de los requisitos temporales (Escenario R4).

7.4 Evaluación de la arquitectura respecto de sus requisitos originales.

Los resultados presentados en los párrafos anteriores describen un panorama bastante desolador. Sin embargo, como se ha venido comentado a lo largo de la tesis, la arquitectura ha sido utilizada con resultados satisfactorios en el desarrollo de varios sistemas, habiendo sido posible reutilizar una buena parte del código. ¿Cómo se resuelve la paradoja? Analizando las características de los sistemas para los que se ha empleado la arquitectura.

En los apartados anteriores se han descrito las carencias de la arquitectura y se ha puesto todo el énfasis en los escenarios que no han podido realizarse correctamente. Sin embargo, no se han comentado otros escenarios para los cuales la arquitectura se comporta de forma satisfactoria. Las características y requisitos descritos en el capítulo 5 dan como resultado un dominio muy grande, en el que hay que dar satisfacción a un gran número de atributos, cuyo logro implica el uso de estilos y mecanismos arquitectónicos a menudo contradictorios. Si la arquitectura se ha comportado bien en los sistemas para los que ha sido empleada, este hecho significa que los requisitos de tales sistemas constituyen el subconjunto de requisitos correspondiente a los escenarios que se realizan satisfactoriamente. Significa también que algunas de las carencias detectadas en la arquitectura han sido corregidas durante la implementación de los sistemas, pero no han sido incorporadas a la descripción de la misma. Merece la pena dedicar algunos párrafos a discutir estos aspectos con mayor profundidad. Razonar acerca de ellos permite acotar el dominio de aplicabilidad de la arquitectura y ofrece un marco para comprender el porqué de algunos escenarios.

7.4.1 Aplicaciones de mantenimiento de Generadores de vapor.

En los primeros proyectos en los que se empleo la arquitectura y que dieron origen a la misma ([SARPA 1995] y [AAA 1996a]) se abordaba la sustitución del sistema de teleoperación del robot ROSA III de Westinghouse, con objeto de permitir su uso en los nuevos generadores de

vapor de las centrales nucleares españolas de agua a presión[60]. Las características de los sistemas considerados están ampliamente descritas en [Alonso et al 1997], [Pastor et al 1996, 1998], [Iborra et al 2002] y, por supuesto, en [Alvarez 1997].

Aunque en [Alvarez 1997] se realiza un estudio genérico de los sistemas de teleoperación, los requisitos de estas aplicaciones pesaron mucho sobre el diseño de la arquitectura, por ser los de los sistemas en los que ésta debía utilizarse de forma inmediata. Este hecho no es negativo visto con la suficiente perspectiva. De hecho, el sistema de teleoperación del brazo ROSA III representa fielmente las características de muchos sistemas de teleoperación y es una aplicación muy exigente, en la que están en juego la integridad de los operarios y grandes intereses económicos[61]. Las principales características de tales sistemas pueden resumirse así:

- El comportamiento reactivo se reduce a paradas de seguridad que se gestionan en la unidad de control local (**`LocalSys`** en la terminología de la tesis). Todas las acciones son iniciadas y supervisadas por el operador, incluso las secuencias de operaciones preprogramadas.
- El código de la unidad de control local debía respetarse, así cómo su protocolo de comunicaciones con la plataforma de teleoperación. Este requisito venía impuesto por tres razones:
  1. Dicho código funcionaba correctamente.
  2. Las mejoras que se proponían para los nuevos sistemas no tenían que ver con la funcionalidad que dicho código proporcionaba.
  3. No había tiempo para desarrollar un código nuevo.
- El entorno de operación es completamente estructurado. El brazo ROSA III opera anclado en la placa-tubo del generador de vapor (figura 7.1), dónde existen lugares en los que se puede realizar una calibración muy precisa del punto de herramienta del brazo. Bajo estas condiciones los módulos de representación gráfica, de cálculos cinemáticos y de detección de colisiones son muy fiables, pues representan fielmente al robot en su entorno. De hecho, los operarios, una vez calibrado el brazo, prescindían en muchos casos de las cámaras.
- La adaptación de la aplicación a nuevos generadores de vapor consistía en proporcionar a los sistemas de gráficos y de detección de colisiones los modelos de la placa-tubo y de la caja de aguas (figura 7.1). Un dibujante experimentado en herramientas CAD podía llevar a cabo este trabajo en un par de días.
- Todas las herramientas mostraban patrones de interacción con el robot muy parecidos y muy sencillos. Simplemente, hasta que el robot no situaba la herramienta correctamente en el punto de trabajo, la herramienta no podía actuar. Y viceversa, mientras la herramienta estaba en funcionamiento el brazo no podía moverse. En estas condiciones la definición de un controlador genérico de herramienta era relativamente sencilla. Una vez definido, sólo hacia falta instanciar dichos genéricos con las características de las nuevas herramientas y proporcionar los modelos estructurales de las mismas a los módulos gráfico y de detección de colisiones.

[60] El sistema se planteó para las centrales españolas, pero existen docenas de reactores en todo el mundo donde puede ser utilizado.

[61] Para hacernos una idea. Cinco minutos en un generador de vapor pueden suponerle a un operario la dosis de radiaciación anual permitida por la ley. En una central nuclear cada día de parada no programada puede suponer un quebranto de unos 100 millones de pesetas [AAA96].

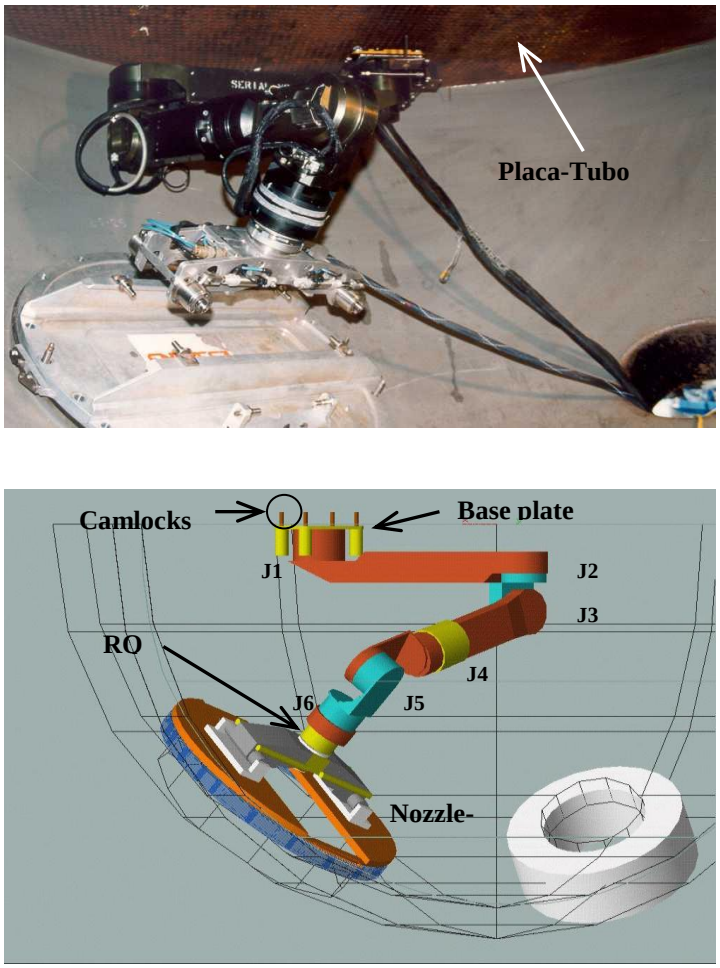


Figura 7.1: Brazo ROSA III en generador de vapor. Actuación y Simulación.

- Todos los brazos que trabajan en los generadores de vapor de las centrales nucleares de agua a presión son conceptualmente muy parecidos. En realidad, el brazo ROSA III es el más completo y complejo y también el más robusto y flexible. La definición de un controlador genérico para el brazo fue una tarea complicada, pero posible. Dicho controlador se configuraba con las características de las diferentes versiones del brazo ROSA (ROSA II, III y IV) o con las características de otros brazos que, como se acaba de decir, eran conceptualmente muy parecidos.
- No hay restricciones sobre la plataforma de teleoperación. Puede ser tan potente y cara como sea necesario. Los PCs ni siquiera se consideraron. La remuneración obtenida por las operaciones de mantenimiento permite amortizar rápidamente los gastos. Lo importante es llegar antes que la competencia, por tanto, es mucho más importante optimizar el tiempo de desarrollo que las inversiones en equipos o herramientas[62].
- Si ocurre un error en tiempo de ejecución, lo importante es detectarlo, parar la aplicación dejando al robot en un estado seguro y, si es necesario, rearrancar el sistema. La disponibilidad y la seguridad son mucho más importantes que la fiabilidad. Si el tiempo de recuperación del sistema es pequeño (del orden de minutos) pueden admitirse unos pocos fallos (p.e.: 1 por turno, trabajando a tres turnos de 8 horas) si éstos no ponen en peligro a equipos, instalaciones y operarios.

Dados estos requisitos, los escenarios se reducen a aquellos para los cuales la arquitectura presenta en general un buen comportamiento. Así:

- **Portabilidad:**
  El escenario P1 se restringe a compatibilidad entre diversas versiones de UNIX y los escenarios P2 y P3 no tienen sentido. Puesto que no se consideró el uso de COTS, salvo para los servidores de gráficos y cinemática, tampoco son pertinentes las observaciones que se anexan a la realización de tales escenarios.
- **Capacidad de distribución:**
  El escenario CD2 no tiene sentido y el CD1 aunque se realiza de forma mediocre no es muy probable.
- **Integrabilidad:**
  El escenario IS4 no tiene sentido, pues no se consideró el empleo de *middleware*.
- **Cambios en la unidad de control local:**
  ACL1 y ACL2 se realizan de forma mediocre, pero no son muy frecuentes.
  ACL3 y ACL4 no tienen sentido.
- **Adaptabilidad a cambios en mecanismos y misiones:**
  El escenario AMM1 no tiene sentido.
  Los escenarios AMM2, AMM3 y AMM4 se realizan satisfactoriamente en este caso, ya que los controladores genéricos modelan bien las características de los patrones de comportamiento de los controladores de los brazos y herramientas considerados.
  El escenario AMM9 no tiene sentido.
- **Adaptabilidad a cambios en los requisitos de rendimiento:**
  Se realizan correctamente.
- **Escenarios del Rendimiento:**

[62] A modo de ejemplo, las licencias de ROBCAD [ROBCAD 2000] utilizadas para desarrollar los servidores gráfico y cinemático costaron alrededor de 90.000 euros (15 millones de pesetas, del año 1996).

Todos los escenarios tienen sentido. Los escenarios R1, R2 y R3 se realizan correctamente, no así el R4, que considera la posibilidad de usar los servidores en línea para monitorizar la ejecución de los comandos. Sin embargo, aunque éste comportamiento es muy deseable puede soslayarse de forma relativamente fácil:

- ✓ Precalculando trayectorias libres de colisiones (incluido en la funcionalidad de **`CinServer`**).
- ✓ Almacenando puntos intermedios y comparando las tablas de posiciones precalculadas (estado esperado) con el estado enviado por la Unidad de Control Local (estado real).
- ✓ Ejecutando una simulación simultánea a la realización de los movimientos, para dar al operador la ilusión de actualización en tiempo real.

- **Escenarios de la fiabilidad/disponibilidad**
  Estos escenarios se plantean de forma mucha más relajada. Como se ha dicho, lo importante es detectar el fallo y detener la aplicación. No se plantean mecanismos de tolerancia a fallos, salvo en las comunicaciones y de una forma muy laxa. No obstante, la arquitectura en sí no define mecanismos adecuados

Los mecanismos no descritos, asociados fundamentalmente a la configurabilidad y a la disponibilidad, se definieron en las implementaciones (realizadas en Ada 83). Así, se utilizó el mecanismo de excepciones proporcionado por el lenguaje y se definieron manejadores de excepciones en diferentes niveles. Asimismo, se implementó una pequeña interfaz textual, mediante la cual se podían enviar comandos básicos a la unidad de control local en caso de fallo general de los procesos de la plataforma de teleoperación, completamente independiente del resto de la aplicación. Sin embargo, esta información no llegó a incorporarse a la descripción de la arquitectura. Igualmente, se definió un pequeño susbsistema encargado de arrancar los procesos de acuerdo con la configuración elegida, que tampoco se incorporó a la descripción de la arquitectura.

Resumiendo, la arquitectura realiza bastante bien los escenarios que se corresponden con sus requisitos de partida, excepto el R4, que puede soslayarse, y los mecanismos no descritos en la misma se definieron en las implementaciones. La organización jerárquica y el estilo cliente servidor permitió integrar fácilmente los servidores de gráficos y cinemático a costa de la flexibilidad del sistema, flexibilidad que, dados los requisitos originales, era un aspecto secundario.

7.4.2 Inspección y recuperación de objetos de la vasija del reactor. Sistema TRON.

El segundo proyecto para el que se utilizó la arquitectura fue el sistema TRON [TRON 1996, 1997a, 1998]. En este caso se trataba de teleoperar una pértiga articulada capaz de inspeccionar los internos inferiores de las vasijas de las centrales nucleares de agua a presión y de recuperar objetos del interior de las mismas [Pastor et al 2000]. El TRON comparte muchas características del primer proyecto, pero también presenta diferencias sustanciales:

- El TRON es un sistema completamente nuevo. No hay que integrar código o equipos ya existentes.
- El entorno de trabajo es no estructurado. La pértiga se ancla por su base a la placa de soporte del núcleo (a 10 metros de profundidad bajo el agua y a 25 metros de la estación de

teleoperación) y debe ser capaz de acceder a través de la estructura de los internos hasta el fondo de la vasija (figuras 7.2 y 7.3).

La aplicación parece una réplica de la anterior, en la que el brazo se ha sustituido por una pértiga y el generador de vapor por la vasija. En realidad es muy diferente. Aunque la pértiga está fijada a su entorno en un lugar conocido, el propio entorno, la vasija, no está bien definido. Los planos disponibles de la vasija no se corresponden *exactamente* con las estructuras de la misma y, aunque así fuera, son de un grado de complejidad tal que no es razonable modelarlos. En segundo lugar, los objetos a recuperar no están en lugares predeterminados sino que pueden estar situados en cualquiera de las superficies de los internos. Por último, las condiciones de visibilidad en el fondo de la vasija son todo lo malas que cabría esperar a 10 metros bajo el agua y sin apenas luz ambiental.

En tales condiciones los servidores gráfico y cinemático no proporcionan una representación fidedigna de los mecanismos y su entorno y la imagen ofrecida por las cámaras no es de buena calidad. Por tanto, es necesario el empleo de un sistema de visión artificial que proporcione utilidades de visión estereoscópica y de reconocimiento de formas. Tales utilidades fueron efectivamente incorporadas en el sistema, pero siguiendo un esquema de interacción (flujo de datos y control) distinto al utilizado para los subsistemas gráfico y cinemático [Pastor et al 2000].

- Las operaciones de mantenimiento de los generadores de vapor son actividades regulares que se ejecutan en cada ciclo de mantenimiento de la central. El TRON es un dispositivo para contingencias. La recuperación de las inversiones depende de la ocurrencia de los incidentes para los que está diseñado. Como éstos podrían no darse nunca, el presupuesto se limita a lo estrictamente necesario. Así se estableció que:
  - ✓ Las plataformas debían ser PCs con Windows 95 ó 98. De hecho, se consideraba el uso de una única plataforma.
  - ✓ No era necesario integrar herramientas gráficas y de simulación potentes.

En estas circunstancias, algunos de los escenarios que no tenían sentido en el caso anterior son importantes en este. Otros adquieren mayor relevancia. Así, los escenarios P1, CD2, AMM9 y R4 que no tenían importancia o podían soslayarse (R4) en el caso anterior, deben ser cuidadosamente considerados en éste.

Debido a que R4 y AMM9 no se realizan satisfactoriamente, la arquitectura no puede adaptarse fácilmente para la realización de trabajos en entornos no estructurados, por su incapacidad para integrar de forma efectiva utilidades de visión artificial. Los esquemas de interacción entre **`RemoteCtlr`** y el resto de subsistemas representan un cuello de botella que no puede soslayarse mediante otras técnicas ni, dado lo limitado de los recursos, mediante el empleo de plataformas o servidores más potentes. El sistema es operativo a costa de comprometer la usabilidad e incrementar los períodos de adiestramiento de los operarios, que deben entrenarse para interpretar las imágenes proporcionadas por el sistema de visión artificial.

Figura 7.3: Sistema TRON. Pruebas en maqueta durante el desarrollo del sistema.

7.4.3 El sistema IRV.

La tercera aplicación fue el sistema IRV (*Inspection Robot Vehicle*), un pequeño vehículo cuya misión era la inspección y recuperación de objetos de toberas y tuberías inundadas [Pastor et al 1998] (figura 7.4). Para ello, el vehículo portaba una pinza, luces y cámaras. En las tuberías no hay referencias e incluso con buenas cámaras e iluminación adecuada sólo es posible ver a unos pocos metros de distancia. Más allá, la obscuridad y la nada. Nuevamente fue necesario incorporar un sistema de visión artificial, si bien más sencillo que en el caso anterior ya que inicialmente no se consideró la visión estereoscópica. Y nuevamente dicho sistema se integró siguiendo un esquema distinto al propuesto en la arquitectura para los otros subsistemas.

Las condiciones de operación del TRON y del IRV no son excepcionales. La arquitectura puede utilizarse en estos dos sistemas porque la complejidad del entorno y de las operaciones a realizar permiten adiestrar a los operadores en la interpretación de las imágenes. La información del sistema de visión es realimentada al sistema a través de los ojos y las manos del operador humano, quedando la utilidad de los módulos gráfico y cinemático reducida a trabajos de simulación y entrenamiento. Sin embargo, no es previsible que este adiestramiento sea siempre posible, ni que la degradación de la usabilidad que conlleva sea aceptable en todos los casos.

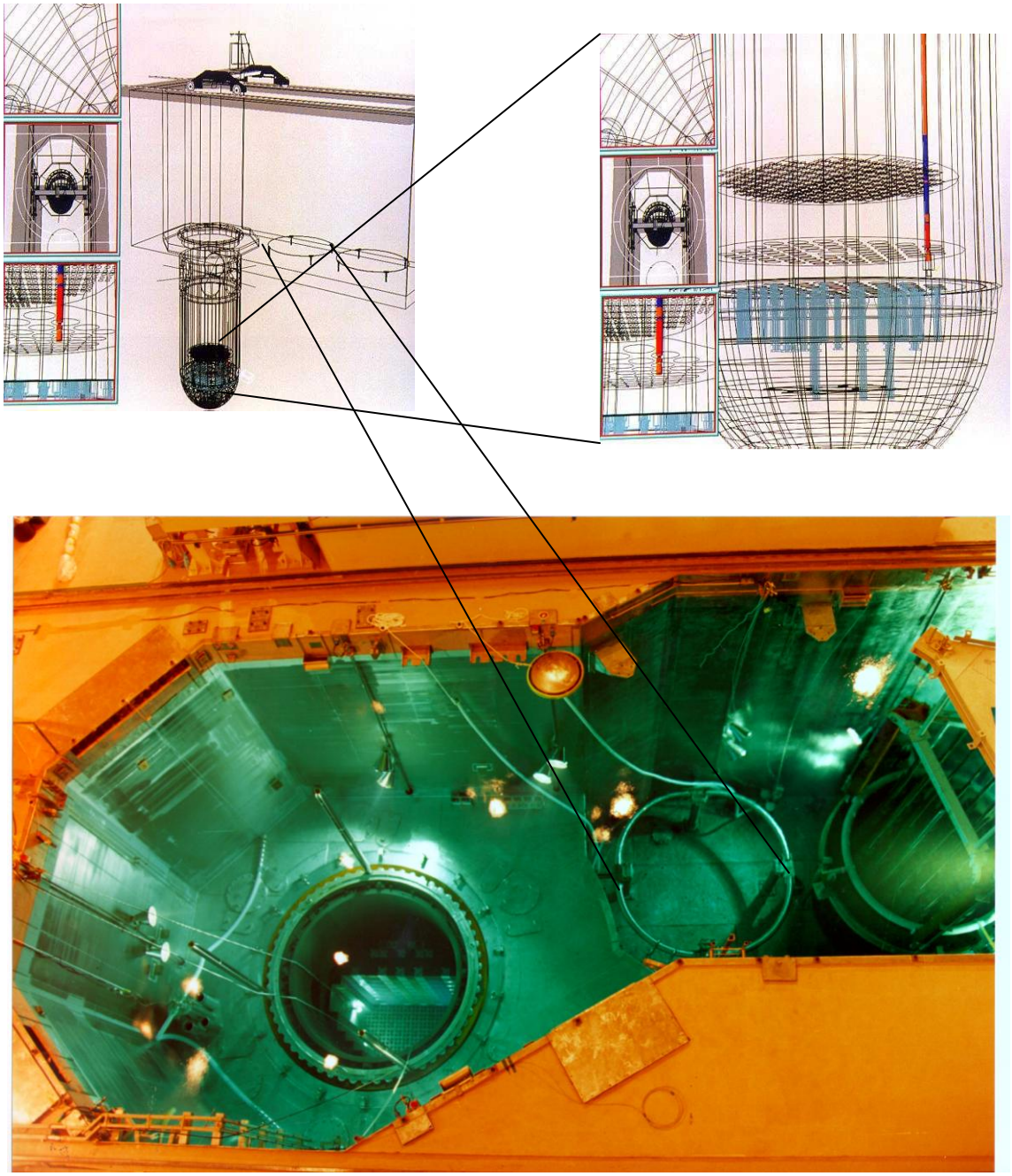


Figura 7.3. TRON: Entorno de Operación.

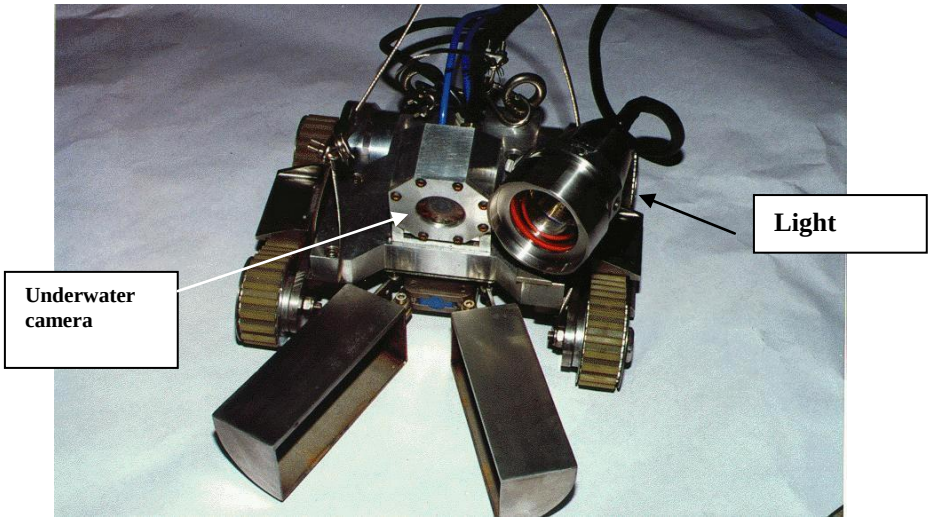


Figura 7.4. Vehículo IRV

7.4.4 Limpieza e inspección de finos de embarcaciones. Sistema GOYA.

Por último, la arquitectura se ha utilizado para el desarrollo de un prototipo de un sistema teleoperado de limpieza de finos de embarcaciones de grandes dimensiones (proyecto GOYA [GOYA 1998], figura 7.5). Las características de este sistema difieren bastante de las de los anteriores:

- Por vez primera, el comportamiento reactivo predomina en diversas fases del funcionamiento sobre el deliberativo (órdenes del operador). El sistema debe ser capaz de limpiar paños enteros del casco de un buque sin la asistencia o supervisión constante del operador.
  - ✓ El avance del cabezal de limpieza y la presión del chorreo dependen de los acabados que se van obteniendo.
  - ✓ Los movimientos deben adaptarse al contorno del casco y a las singularidades de las superficies. La información sobre estos aspectos se obtiene a partir de diferentes formas de sensorización.
  - ✓ El funcionamiento del cabezal de limpieza y del posicionador deben sincronizarse de acuerdo con el desarrollo de la operación.
- Puede haber varios mecanismos realizando operaciones de mantenimiento simultáneas en diversas partes del casco. Hay que evitar solapamientos y colisiones.
- El entorno es en gran medida no estructurado. Las modificaciones necesarias para hacerlo estructurado no son económicamente viables.

Figura 7.5. Prototipo sistema GOYA durante pruebas en astillero.

- La fiabilidad es tan importante como la disponibilidad. No es admisible la tasa de fallos que podía tolerarse en los sistemas anteriores. Es importante que el sistema sea tolerante a cierto tipo de fallos y que proporcione modos de funcionamiento degradados, que permitan realizar las operaciones aunque falle parte del sistema.
- La seguridad sigue siendo un requisito de capital importancia.
- La precisión de posicionamiento y la complejidad de la cinemática es mucho menos exigente que en los sistemas anteriores, pero:
  - ✓ Este requisito puede cambiar si se consideran otras operaciones de mantenimiento.
  - ✓ Aspectos relacionados con el comportamiento dinámico y con la flexión de los posicionadores, que podían despreciarse en otros sistemas, pueden ser relevantes en éste.
- Puesto que el sistema puede extenderse para considerar otras operaciones de mantenimiento del casco, siguen siendo importantes los requisitos de adaptabilidad a nuevos mecanismos y misiones.
- El coste de los equipos debe adaptarse al presupuesto de los astilleros y el coste total de las operaciones, incluyendo la amortización de los equipos, no debe exceder al actual.

El sistema GOYA está a medio camino entre los robots industriales y los robots puramente teleoperados. Por un lado debe ser capaz de actuar con cierta autonomía, por otro no puede liberarse de una cierta supervisión del operador. Las funciones de éste se reducen a definir las misiones (secuencia de paños a limpiar) y sus parámetros y a resolver las alarmas y las *dudas* del sistema de limpieza. Los cascos de los barcos tienen superficies relativamente fáciles de

recorrer y otras muy complicadas, bien por la presencia de abultamientos, agujeros, salientes y ventanas, bien por su geometría, plana en unas zonas y muy curvada en otras. Cuanto más capaces de detectar y librar estas zonas sean los mecanismos, menos trabajo de teleoperación.

En este sistema, a pesar de que los dispositivos teleoperados presentan, en primera instancia, menos complejidad que en los anteriores, prácticamente todos los escenarios descritos tienen alguna relevancia. Por ello, dados los resultados de la evaluación, no queda más remedio que plantearse modificaciones en la misma o su completa sustitución.

7.5 Caracterización del Rendimiento. Escenario R4: Uso de servidores en línea.

Como se ha explicado, una de las consecuencias de la adaptabilidad a nuevos entornos es la posibilidad de considerar entornos no estructurados. En este tipo de entornos la información sobre los mismos debe ser obtenida *sobre la marcha* a partir de diferentes tipos de sensorización. El escenario R4 (Uso de los servicios de **`CinServer`** en línea para detectar colisiones) y el escenario IS1 (adición de nuevos servidores) responden en buena parte a esta necesidad.

En este apartado se consideran los efectos que tiene sobre el rendimiento del sistema el uso en línea de los servicios de **`CinServer`**. Aunque el tipo de servicios que proporciona **`CinServer`** no están pensados para su uso en entornos no estructurados, los razonamientos que se realizan en este apartado pueden aplicarse a otro tipo de servicios que sí lo están, por ejemplo los proporcionados por un sistema de visión artificial.

En [Álvarez 1997] y con mayor extensión en [Álvarez et al 1998] se realiza una caracterización del comportamiento temporal del sistema en función del tipo de eventos que reciben los subsistemas y sus plazos de respuesta. Dichos estudios son muy exhaustivos y describen con gran detalle todos los posibles eventos y sus patrones de llegada, proporcionando un marco de razonamiento muy completo para deducir el comportamiento temporal de los sistemas y planificar las diferentes tareas. Sin embargo (como es lógico) no tienen en cuenta patrones de interacción no previstos en los sistemas originales, como es el caso del uso en línea de los servicios de **`CinServer`**. Con objeto de ampliar el marco de estudio temporal se ha extendido el modelo para considerar nuevos tipos de interacciones. Para ello, se ha utilizado la herramienta UML-Mast [Drake et al 1999], mediante la cual se han modelado nuevos escenarios.

Las figuras 7.6 y 7.7 muestran dos diagramas de colaboración en los que se consideran el uso en línea de los servicios de **`CinServer`** tal y como propone la arquitectura (figura 7.6) y mediante un patrón de suscripción (figura 7.7). Los nombres de los controladores se han cambiado respecto de anteriores diagramas, así el controlador local (**`LocalSys`**) recibe ahora el nombre de **`LowLevelController`** y **`RemoteCtlr`** el de **`HighLevelController`**, pero sus responsabilidades siguen siendo exactamente las mismas. El cambio de nombre responde al hecho de que en los análisis se ha contemplado la ejecución de todas las tareas en un solo nodo y los prefijos **`Local`** y **`Remote`** podrían dar lugar a confusión.

En el diagrama de la figura 7.7 **`CinServer`** recibe la información directamente de **`LowlevelController`**. Cuando **`CinServer`** detecta peligro de colisión envía un evento a una tarea muy prioritaria de **`HighLevelController`**, que inmediatamente ordena a **`LowLevelController`** que detenga a los mecanismos. En todas las simulaciones se han considerado los mismos tiempos de ejecución para cada una de las operaciones realizadas y se

han considerado las mismas transacciones, salvo en los casos en los que se indica explícitamente.

Los escenarios que se proponen no tienen por objeto realizar un estudio exhaustivo del rendimiento del sistema, sino comparar comportamientos frente a diferentes patrones de interacción. Por ello, se han realizado algunas simplificaciones, que se comentan al final del apartado, una vez descrito el modelo de análisis.

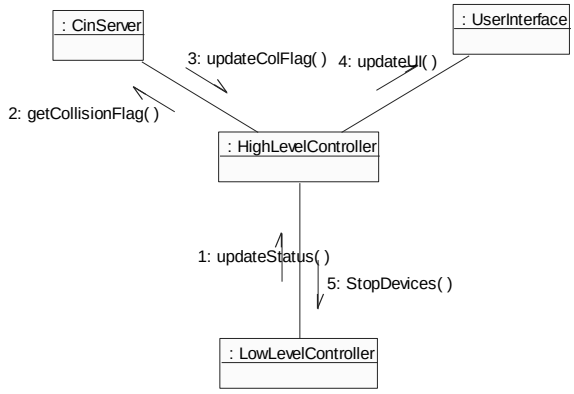


Figura 7.6: Servicios en línea de CinServer. Esquema Original

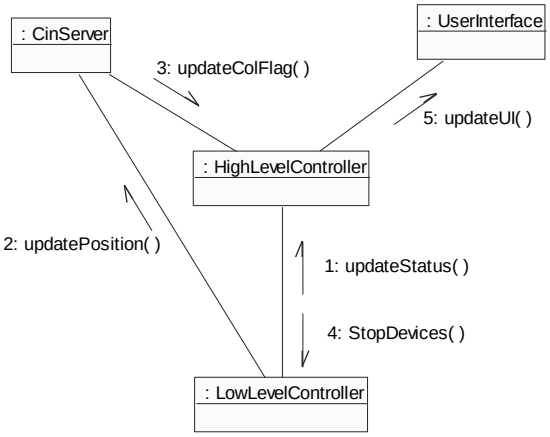


Figura 7.7: Servicios en línea de CinServer. Patrón observador.

Para la definición de la vista de tiempo real, UML-Mast necesita de tres sub-vistas complementarias: el modelo de plataforma, el modelo de componentes lógicos y el modelo de escenarios o transacciones.

El **modelo de plataforma** modela la capacidad de procesamiento de los recursos hardware y software que se declaran en el sistema y que ejecutan las actividades que constituyen el sistema que se modela. Sus componentes básicos son los *Scheduling Servers* que modelan los procesos y las políticas de planificación de las actividades que se les asignan y los *Processing Resources* que modelan los componentes hardware (incluyendo enlaces de comunicaciones) y la infraestructura software (S.O., drivers, etc). En este caso, el modelo de plataforma es el mostrado en la figura 7.8. Por simplicidad, se ha considerado un único nodo, y todas las tareas (*Scheduling Servers*) se planifican según una política de prioridades fijas RMS (*Rate Monotonic Scheduling*).

El **modelo de componentes lógicos** modela los requisitos de procesado que exige la ejecución de las operaciones definidas en los componentes lógicos (métodos, procedimientos, funciones, etc). Asimismo, en este modelo se definen los recursos compartidos que deben accederse en régimen de exclusión mutua. Este modelo se define con una modularidad paralela a la que existe en el diseño lógico del sistema, pero no se genera de forma automática por lo que debe construirse y mantenerse a mano. Las figuras 7.9 a 7.13 describen las operaciones y recursos

que se han modelado para este caso. Estas operaciones son invocadas en las tareas definidas en el modelo de plataforma, como se mostrará en las transacciones definidas en los modelo de escenarios.

Por último, en **el modelo de escenarios** se modelan las transacciones que describen las secuencias de eventos y actividades que deben ser analizados frente a los requisitos de tiempo real. Este modelo incluye la descripción de los patrones de llegada de los eventos y de los plazos que debe cumplir el sistema. Mientras que los anteriores modelos son comunes a los diagramas de colaboración de las figuras 7.6 y 7.7, se han definido tres modelos de escenarios. El primero correspondiente al diagrama de colaboración la figura 7.6 (patrones de interacción definidos en la arquitectura) y el segundo y el tercero correspondientes a dos variantes del patrón de interacción mostrado en el diagrama de la figura 7.7. En el siguiente apartado se describen las transacciones definidas en dichos escenarios y los resultados de su simulación.

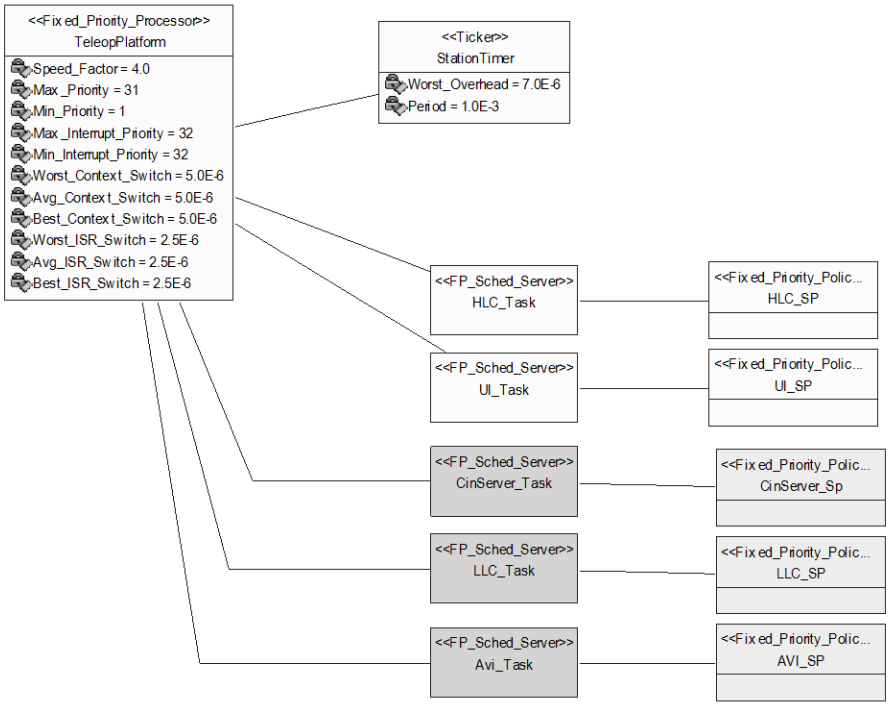


Figura 7.8: Modelo de plataforma.

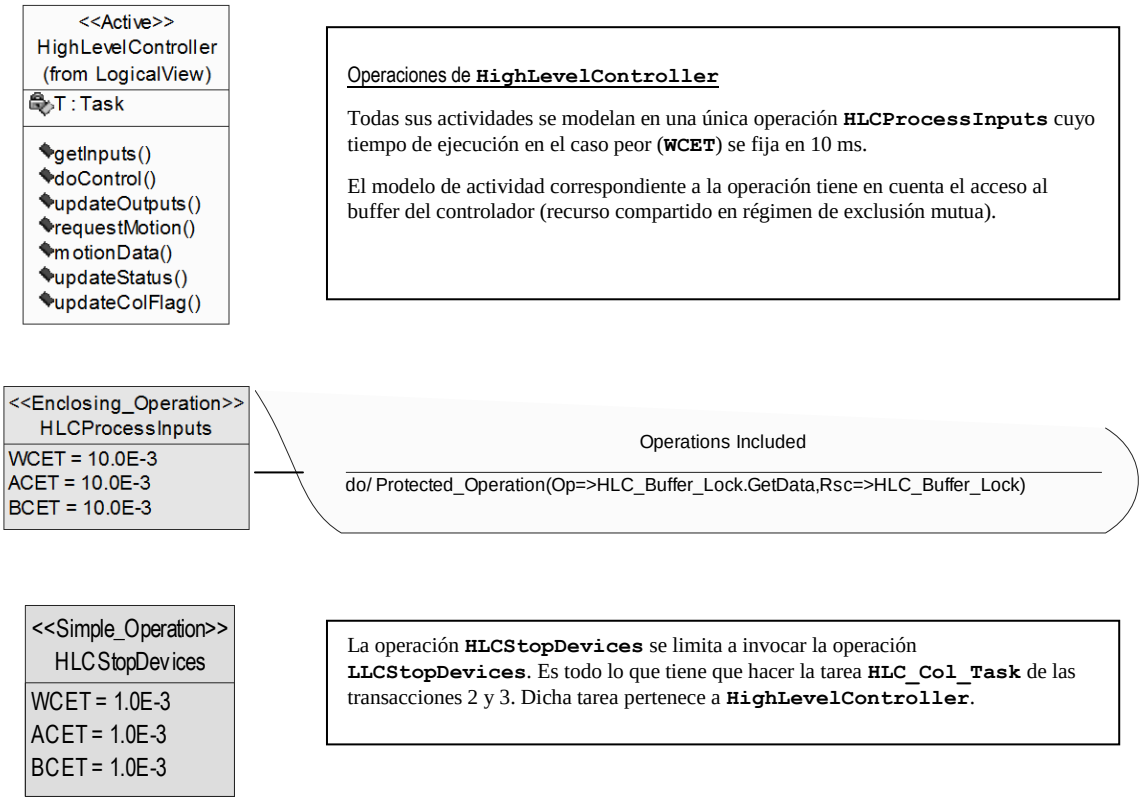


Figura 7.9: Modelo de Componentes Lógicos.
Modelado de las actividades de `HighLevelController` (`RemoteCltr`).

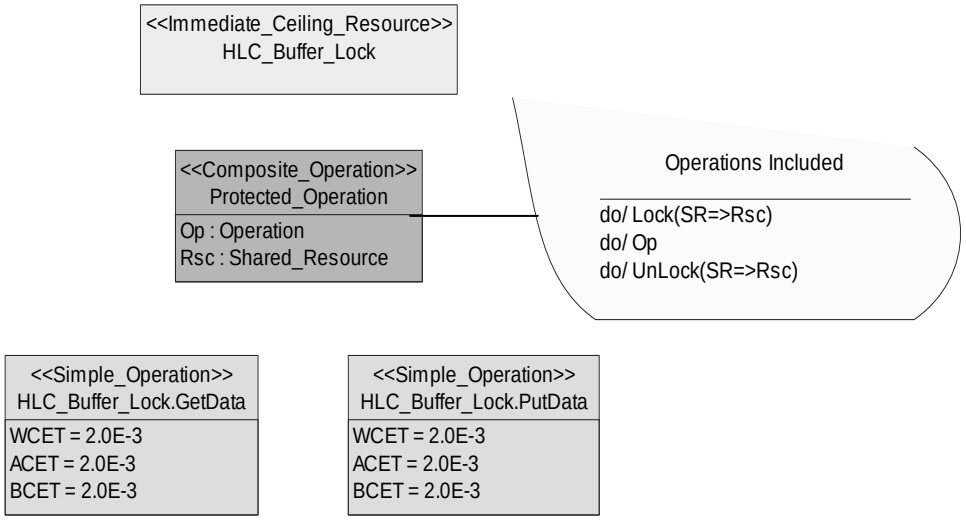


Figura 10: Modelo de Componentes Lógicos.
Buffer del controlador

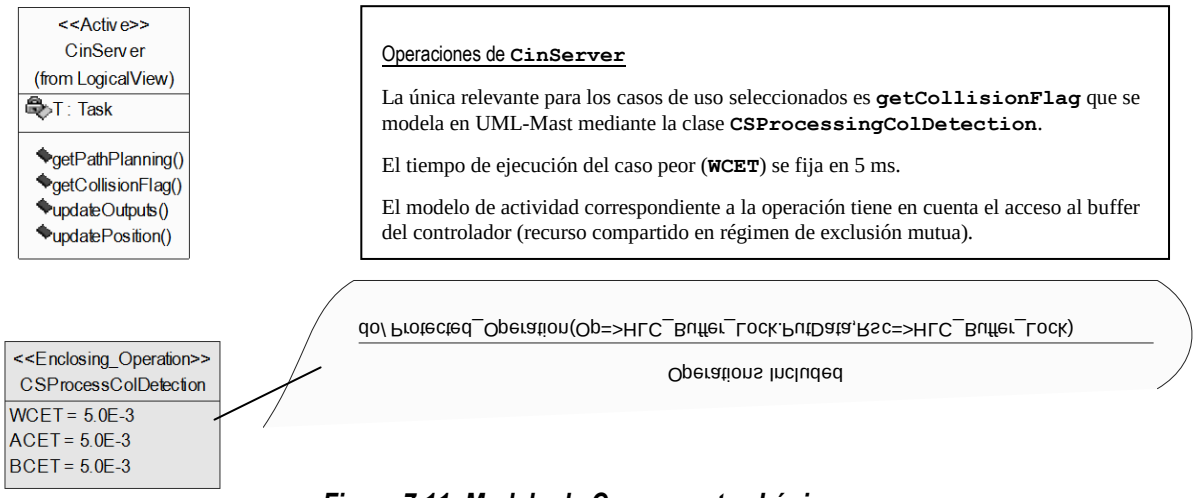


Figura 7.11: Modelo de Componentes Lógicos. Modelado del procesamiento de colisiones.

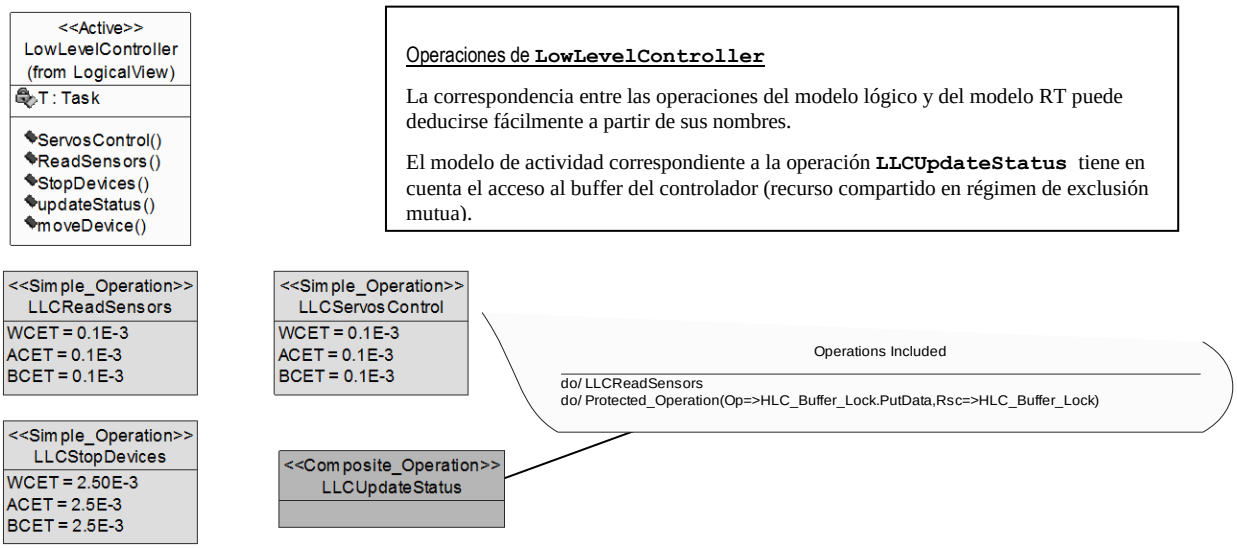


Figura 12: Modelo de Componentes Lógicos. `LowLevelController` (`LocalCtlr`)

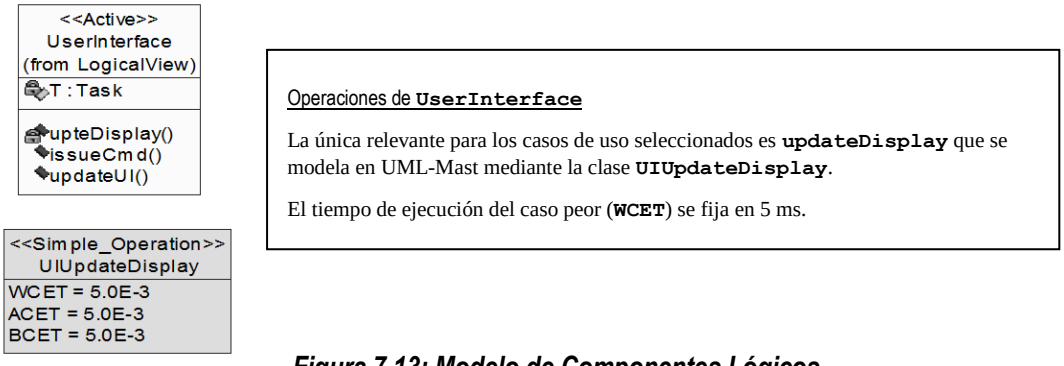


Figura 7.13: Modelo de Componentes Lógicos. Interfaz de usuario

7.5.1 Modelos de escenarios

Los modelos de escenarios modelan las transacciones que deben tenerse en cuenta para realizar el análisis temporal. Así, cada modelo de escenarios consta de un conjunto de transacciones, cada una de las cuales se caracteriza por un diagrama de actividad, un patrón de eventos de llegada a los que debe atender y sus requisitos temporales. Para caracterizar las interacciones de las figuras 7.6 y 7.7 se han elaborado tres modelos de escenarios. El primero de ellos (modelo 1, figura 7.14) se corresponde con el diagrama de colaboración de la figura 7.6. Las transacciones de este modelo responden al tipo de interacciones definidas por la arquitectura. Toda la información pasa a través del controlador **`HighLevelController`**. Todos los intercambios de datos entre dicho controlador y **`CinServer`** y **`LowLevelServer`** se hacen a través del buffer del controlador (**`HLC_Buffer_Lock`**) que es un recurso compartido en régimen de exclusión mutua. Los servicios de **`CinServer`** son solicitados de forma explícita por **`HighLevelController`** cada vez que recibe nuevos datos de **`LowLevelController`**.

El caso de uso correspondiente al diagrama de colaboración 7.7 tiene asociados dos modelos de escenarios. En el primero de ellos (figura 7.15) sólo se han considerado las interacciones relacionados con el procesamiento de la detección de colisiones. Por ello, este modelo no es una buena referencia de comparación con el modelo de escenarios 1, ya que no tiene en cuenta el resto de actividades que periódicamente realiza **`HighLevelController`**. Para soslayar este problema se define el modelo de escenarios 3 (figura 7,16) en el que se ha incluido una nueva transacción (**`MonitorStatus`**) que incluye dichas actividades.

Cada una de las transacciones que aparecen en los modelos de escenarios de las figuras 7.14 a 7.16 tiene asociado un diagrama de actividad. Los diagramas de actividad correspondientes a los diferentes modelos de escenarios se muestran en las figuras 7.17 a 7.21. Los diagramas de las figuras 7.17 y 7.18 se corresponden con las transacciones **`ServosControl`** y **`UpdateDisStatus`** que aparecen en los tres modelos de escenarios. **`ServosControl`** representa el control de los servos de los mecanismos teleoperados y **`UpdateDisStatus`** la actualización en la interfaz de usuario del estado de dichos mecanismos. El diagrama de la figura 7.20 se corresponde con la transacción **`CollisionControl`** que representa la forma en que se maneja la detección de colisiones en el modelo de escenarios 1, correspondiente al diagrama de colaboración de la figura 7.6, en tanto que el diagrama de actividad de la figura 7.21 se corresponde con la transacción **`CollisionControl2`**, que representa la forma en que se maneja la detección de colisiones en el diagrama de colaboración de la figura 7.7. Dicha transacción es común a los modelos 2 y 3. Por último, el diagrama de actividades mostrado en la figura 7.19 es exclusivo del modelo de escenarios 3 y se corresponde con la transacción **`MonitorStatus`** que representa las actividades del modelo 1 no tenidas en cuenta en el 2.

Los tiempos de respuesta asociados a cada una de las transacciones se muestran en las figuras 7.14 a 7.16 y se resumen para facilitar su comparación en la tabla 7.5.

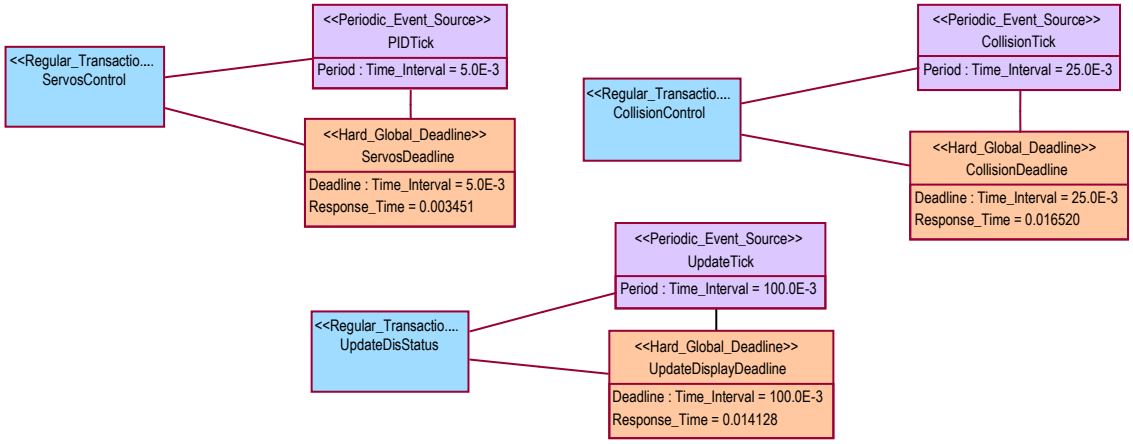


Figura 7.14: Modelo de escenarios 1

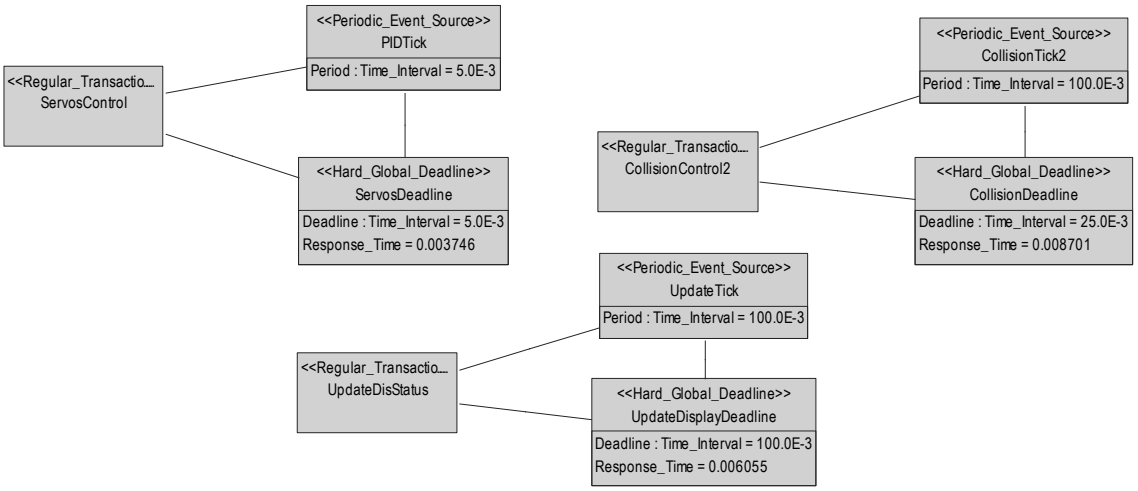


Figura 7.15: Modelo de escenarios 2

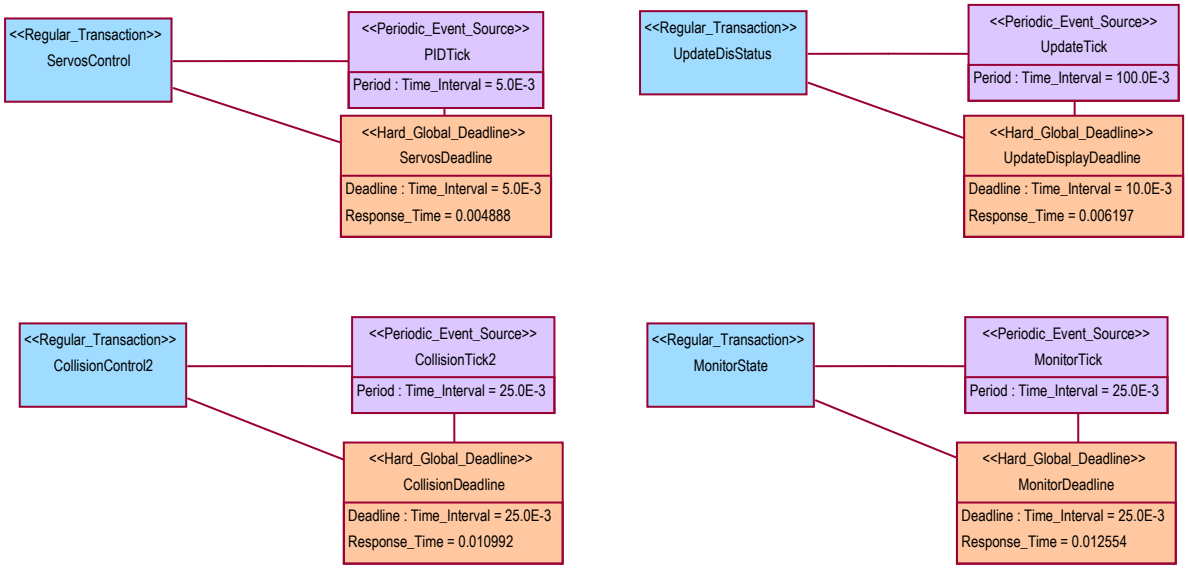


Figura 7.16: Modelo de escenarios 3

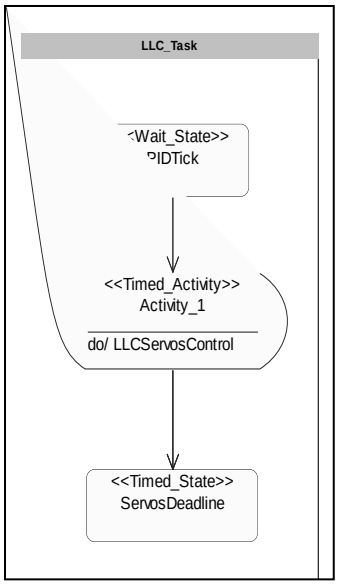


Figura 17: Modelo de escenarios 1, 2, 3:

Transacción **`ServosControl`**

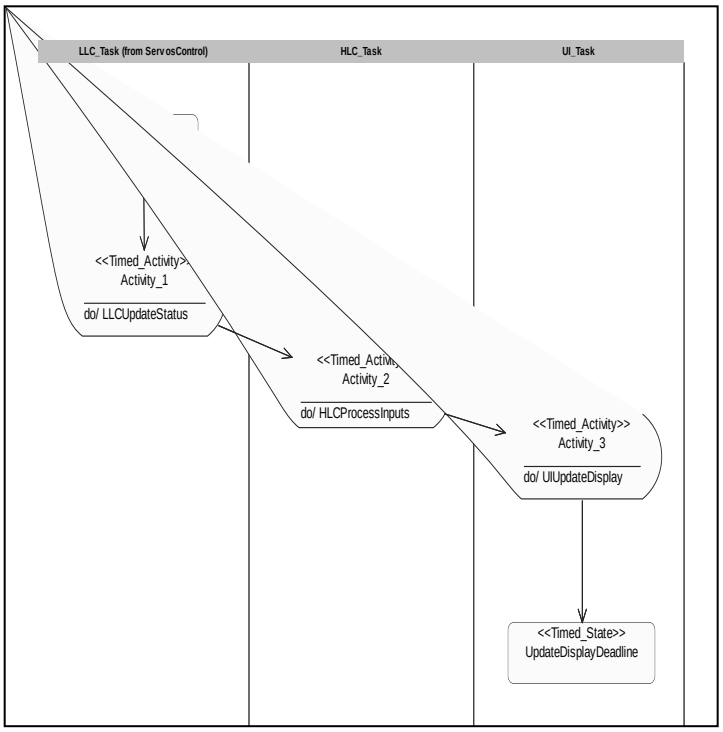


Figura 7.18: Modelo de escenarios 1, 2 y 3

Transacción **`UpdateDisStatus`**

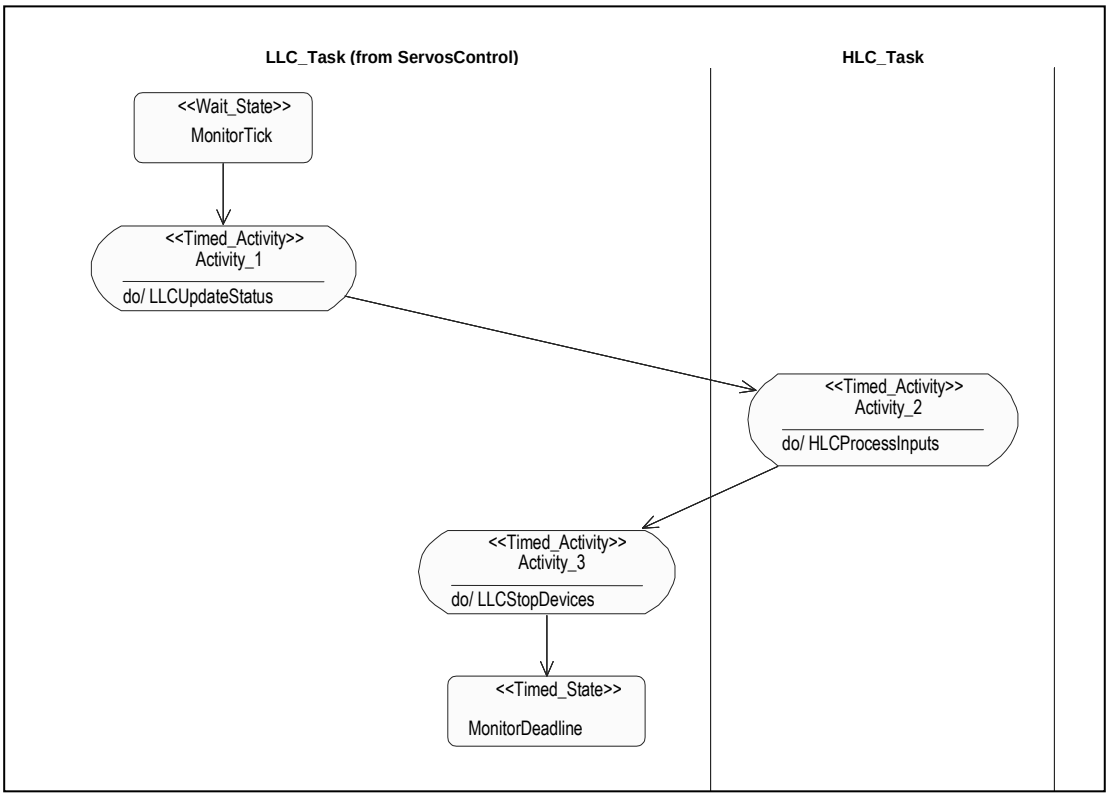


Figura 19: Modelo de escenarios 3

Transacción **`MonitorStatus`**

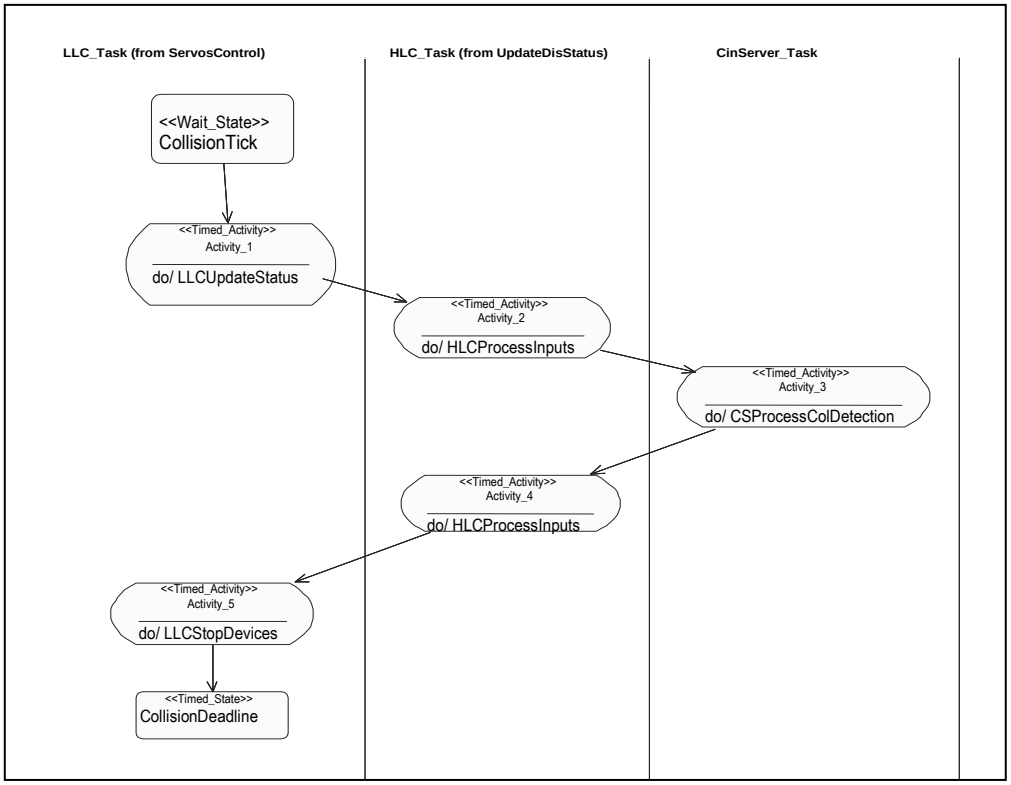


Figura 7.20: Modelo de escenarios 1

Transacción **`CollisionControl`**

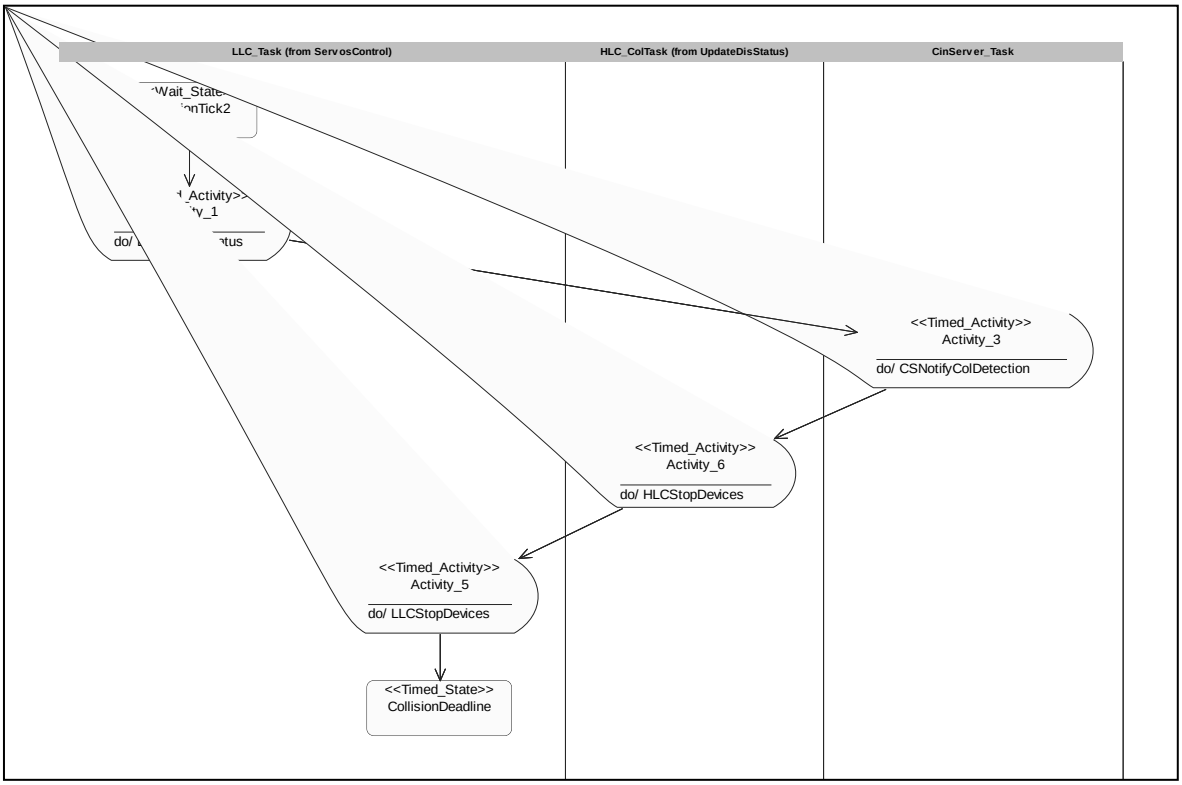


Figura 7.21: Modelo de escenarios 2 y 3

Transacción **`CollisionControl2`**

7.5.2 Resultados de la simulación.

Los resultados de los análisis correspondientes a los tres modelos de escenarios se muestran en la tabla 75. La asignación de prioridades es realizada de forma automática por la herramienta UML-Mast según el algoritmo RMS (*Rate Monotonic Scheduling*). El método de análisis empleado ha sido un RMA (*Rate Monotonic Analysis*) clásico. Obsérvese que los tiempos de respuesta asociados a la detección de colisiones y a la actualización de la interfaz de usuario son el doble en el modelo de escenarios 1 que en los modelos 2 y 3. El tiempo de respuesta de la transacción **`ServosControl`** de los escenarios 2 y 3 son ligeramente superiores a la correspondiente del escenario 1. Dicha transacción es independiente de los patrones de interacción de **`HighLevelController`** por lo que no se beneficia de los cambios en los patrones de interacción y sin embargo se ve perjudicada por la existencia de una tarea adicional.

Los modelos descritos en este apartado contienen algunas simplificaciones, sin embargo es razonable pensar que ninguna de ellas desvirtúa el análisis ya que los escenarios que se proponen no tienen por objeto realizar un estudio exhaustivo del rendimiento del sistema, sino comparar comportamientos frente a diferentes patrones de interacción.. Dichas simplificaciones son:

- No se han tenido en cuenta las tareas asociadas a los módulos de desacoplo.
- Todos los procesos se ejecutan en un nodo. No se considera la influencia de los enlaces de comunicaciones. En el despliegue más habitual **`LowLevelController`** se ejecuta en su propia unidad de procesamiento o se implementa directamente en hardware.
- La política de planificación de la tarea suscrita a la detección de colisiones (**`HLC_Col_Task`**) no es seguramente la más apropiada. En realidad la ocurrencia de colisiones es más bien un evento esporádico que se serviría mejor con un servidor esporádico (*Sporadic Server Scheduling*).
- En la simulación mostrada en la tabla 7.5 la capacidad del procesador se ha ajustado para que independientemente del modelo todas las transacciones cumplieran sus plazos. La capacidad asignada ha quedado justo por encima de la necesaria para que **`ServosControl`** pudiera planificarse en el modelo de escenarios 3.
- Sólo se ha considerado un servidor con lo que el efecto de la exclusión mutua en el buffer del controlador probablemente este subestimado.
- La cantidad de datos que se leen o escriben en dicho buffer es muy pequeña, el tiempo de acceso al mismo sería bastante más grande si se considerase el empleo de un servidor que requiriese grandes cantidades de datos (p.e: un sistema de visión artificial)

Todas las simplificaciones realizadas favorecen el esquema de interacción propuesto por la arquitectura y aun así los tiempos de respuesta son peores que utilizando el patrón alternativo propuesto en la figura 7.7.

Tabla 7.5: Resultados de la simulación								
Transacción	**Patrón llegada de eventos**	**Plazo**	**Escenario 1 System Slack: 76 %**		**Escenario 2 System Slack: 46 %**		**Escenario 3 System Slack: 4 %**	
			Trespuesta	**Slack**	**Trespuesta**	**Slack**	**Trespuesta**	**Slack**
`ServosControl`	Periódico T = 5 ms	5 ms	3,5 ms	5574 %	3,8 ms	4987 %	4,8 ms	447 %
`UpdateDisStatus`	Periódico T = 100 ms	100 ms	17 ms	87 %	6,0 ms	69 %	6,1 ms	7 %
`CollisionControl`	Periódico T = 25 ms	25 ms	14 ms	79 %				
`CollisionControl2`	Periódico T = 25 ms	25 ms			9 ms	89 %	11 ms	6 %
`MonitorStatus`	Periódico T = 25 ms	25 ms					13 ms	4 %

7.6 Conclusiones

La evaluación de la arquitectura frente a los requisitos definidos en el capítulo 5 es negativa. Sin embargo, ésta es quizás la conclusión menos importante. Probablemente ninguna arquitectura evaluaría bien frente a tales requisitos. La evaluación de la arquitectura frente a sus requisitos originales arroja resultados satisfactorios, que van empeorando a media que se amplía el número, o mejor dicho, las características, de los requisitos. El problema es la amplitud del dominio, que convierte las actividades de análisis y diseño en un cúmulo de problemas de imposible solución dentro del marco de una única arquitectura.

La primera conclusión es que no es realista definir arquitecturas de referencia para dominios o líneas de producto en los que los sistemas presentan (demasiados) requisitos contradictorios. Como afirman Coste y Simmons en un breve y excelente artículo [Coste et al 2000] en el que repasan las principales arquitecturas propuestas para sistemas robotizados:

"...la principal lección es que diferentes aplicaciones necesitan descomponer los problemas de formas diferentes, y las arquitecturas necesitan ser lo suficientemente flexibles para acomodarse a las diferentes estrategias de descomposición."[63]

Sin embargo, si la flexibilidad se lleva demasiado lejos se entra en contradicción con el propio concepto de arquitectura software y con sus objetivos, ya que:

1. Por su propia definición una arquitectura no puede acomodarse a cualquier estrategia, porque ella misma las define.

[63] Coste y Simmons comparan estrategias de descomposición basadas en abstracciones temporales o espaciales, pero la lección puede generalizarse sin contradecir el espíritu del artículo.

2. La definición de componentes exige la existencia de un marco de desarrollo estable y bien definido.

La arquitectura descompone la funcionalidad de los sistemas en componentes según determinadas estrategias. Si la estrategia cambia, las unidades funcionales cambian y por tanto cambia la arquitectura. Pero la arquitectura no define sólo las estrategias de descomposición funcional, define también las estrategias de interacción. Aunque la funcionalidad de un componente sea exactamente la que se espera de él, dicho componente no puede integrarse en el sistema si sus patrones de interacción con el resto no siguen las reglas definidas por la arquitectura. Nuevamente, si las estrategias de interacción cambian, lo que cambia es precisamente la arquitectura. Puesto que los componentes del sistema se definen según los patrones de descomposición e interacción definidos por la arquitectura, si tales patrones cambian el componente deja de ser integrable.

La segunda conclusión es que la separación entre la funcionalidad de un componente y sus patrones de interacción con el resto es fundamental para la adaptabilidad de la arquitectura. Para que ello sea posible es necesario que las relaciones entre componentes, es decir, sus patrones de interacción, sean considerados en el modelo como entidades de primera clase. Este enfoque permite definir fácilmente estrategias de interacción reemplazables. El patrón política o estrategia se utiliza con gran profusión para sustituir unos patrones de comportamiento por otros, utilizando un sencillo mecanismo de delegación. Si la funcionalidad y los patrones de interacción se definen por separado puede ser posible generalizar el patrón para considerar diversos tipos de interacción. El escenario R4 que modela la capacidad de requerir servicios en línea fracasa no porque la división de la funcionalidad sea incorrecta, sino por la forma en que el controlador solicita y obtiene los servicios. Los módulos de desacoplo propuestos en la arquitectura son una aproximación a este enfoque, pero al incluirlos en el interior del controlador se pierden la mayoría de sus beneficios.

La tercera conclusión nace al hilo de la segunda. Si las relaciones entre componentes y su funcionalidad se definen por separado, es posible reutilizar los mismos componentes en arquitecturas muy diferentes. Aunque el dominio considerado es demasiado grande para definir una arquitectura de referencia, todos los sistemas incluidos en el mismo comparten una gran parte de la funcionalidad. Este hecho es tanto más cierto cuanto más bajo es el nivel de la funcionalidad considerada y más pequeñas y generales son las responsabilidades de los componentes (casi todos los sistemas tienen PIDs), pero puede darse a cualquier nivel de abstracción y de granularidad (**`CinServer`** puede ser adecuado en muchos sistemas y los controladores genéricos definidos por la arquitectura también pueden ejemplarizarse en muchos sistemas).

Desde el punto de vista de las expectativas del grupo de investigación DSIE de la UPCT (véase capítulo 5) estas tres conclusiones son mucho más determinantes para enfocar las líneas de futuros esfuerzos que los resultados de la evaluación de la arquitectura. No se trata tanto de definir arquitecturas genéricas, como de establecer principios que permitan definir, desarrollar o adquirir marcos de desarrollo (*frameworks*) y herramientas mediante los cuales sea posible definir distintas arquitecturas y diseñar componentes que puedan utilizarse en diferentes sistemas, compartan o no la misma arquitectura. Estos aspectos serán tratados con mayor profundidad en el siguiente capítulo.

Respecto al propio proceso de evaluación es necesario justificar algunas decisiones. Dos puntos clave de ATAM no han podido ser realizados tal y como éste los define: la priorización de escenarios y la definición de puntos sensibles y de compromiso. Nuevamente el motivo es la amplitud del dominio. Todos los escenarios definidos son importantes para algún tipo de sistema y puesto que no puede determinarse qué sistemas son más importantes que otros,

tampoco es posible determinar que escenarios son más importantes. Esta profusión de escenarios conduce a que todas las estructuras y componentes de la arquitectura sean puntos sensibles o de compromiso, pues al final siempre existe un escenario que afecta a alguna de estas estructuras. La caracterización de los atributos de calidad mediante la tupla (estímulo, escenario, mecanismo) ha sido muy útil para razonar acerca de los requisitos de los sistemas y los mecanismos arquitectónicos asociados a su cumplimiento. Sin embargo, la información generada es muy extensa y difícil de mantener sin la ayuda de herramientas. Es necesario organizar y almacenar la información de forma que pueda completarse, corregirse, refinarse y si es posible formalizarse. Sólo de esta manera podrá ser una base de conocimiento útil para el grupo de investigación y la comunidad técnica y científica.

8 Requisitos de un modelo de componentes para el sistema GOYA.

8.1 Introducción.

En el capítulo anterior, tras evaluar la arquitectura de referencia propuesta por Álvarez frente a los requisitos descritos en el capítulo 5, se llegaba a las siguientes conclusiones:

1. No es realista plantear el diseño de arquitecturas de referencia para dominios excesivamente amplios, pues pueden existir tantos requisitos contradictorios entre sistemas que no sea posible alcanzar una solución que satisfaga a todos.
2. La separación de la funcionalidad del componente y sus patrones de interacción con el resto es fundamental para conseguir la adaptabilidad de la arquitectura.
3. Si tal separación se lleva a cabo es posible, por un lado, reutilizar los mismos componentes en arquitecturas muy diferentes y, por otro, adaptar los patrones de interacción a las necesidades del momento.

Estas conclusiones deben determinar el enfoque con el que se aborden los nuevos desarrollos, entre los que se incluye a más corto plazo el GOYA. Sus consecuencias más inmediatas son:

1. Es necesario reducir la amplitud del dominio a unas dimensiones tales que el problema sea abordable y que su solución siga ofreciendo un marco en el que sea posible reutilizar el software desarrollado en un conjunto significativo de sistemas.
2. Deben adoptarse unas estrategias de diseño e implementación que permitan separar, por un lado, los patrones de interacción entre componentes de su funcionalidad y, por otro, los diferentes *aspectos* de dicha funcionalidad.
3. Es mucho más práctico definir un marco de desarrollo en el que puedan definirse diferentes arquitecturas en las que se reutilicen componentes comunes que intentar desarrollar arquitecturas que deban satisfacer demasiados requisitos contradictorios.

La solución a la primera cuestión puede venir dada enfocando el problema como *una línea de producto*. La solución a las dos últimas por la adopción de un *modelo de desarrollo basado en componentes*. Este capítulo se centra en las características que deben exhibir los componentes de los sistemas. La discusión sobre líneas de producto se limita a justificar el enfoque y a

proporcionar la información sobre el GOYA necesaria para justificar el modelo de componentes que se propone. Respecto de dicho modelo, habría que hacer la siguiente precisión:

- Si la simple elección de un modelo de componentes no es un problema trivial, su definición es un trabajo de enorme envergadura. Los objetivos de esta tesis son más modestos. *No se trata de definir el modelo, sino de establecer una serie de requisitos para el modelo que finalmente se adopte o defina*[64]. Algunos de los requisitos podrán generalizarse a otros sistemas, pero otros no.

Incluso así, hay que relacionar información de muy diverso tipo. En el apartado 8.2 se justifica el enfoque como línea de producto. En el apartado 8.3 se describen los propósitos, características y requisitos más relevantes del sistema GOYA. En el apartado 8.4, y aplicando los principios de diseño del ABD explicados en al capítulo 4, se identifican los subsistemas principales del GOYA. Por último, en el apartado 8.5 se proponen una serie de requisitos para el modelo de componentes que se adopte o defina. Los apartados 8.2, 8.3 y 8.4 proporcionan los criterios a partir de los cuales se definen los requisitos del modelo de componentes, que se describen en el apartado 8.5 y elementos de juicio para su crítica y refinamiento. Por dos razones:

1. Proporcionan al lector los elementos de juicio que han llevado al autor a alcanzar sus conclusiones. Estas conclusiones son el resultado de una interpretación personal (no puede ser de otra manera en un trabajo de tesis) de las características y requisitos descritos en los apartados 8.2, 8.3 y 8.4, que pueden corregirse y mejorarse con las aportaciones de terceros.
2. Muestran de forma explícita los criterios que se han tenido en cuenta para la elaboración de los requisitos *y de forma implícita aquellos que no se han tenido en cuenta*. Los apartados 8.2, 8.3 y 8.4 pueden completarse con nuevas aportaciones, que pueden derivar en una corrección o un refinamiento de los requisitos del modelo de componentes.

8.2 Reducción del dominio. Un enfoque como línea de producto.

El primer problema a resolver es reducir la amplitud del dominio hasta unos límites en los que sea posible:

- Definir con exactitud los posibles usos y usuarios del sistema.
- Acotar la variabilidad de los sistemas tanto en su dimensión *espacial* (conjunto de sistemas diferentes que comparten la misma arquitectura y ciertos componentes) y *temporal* (cómo pueden evolucionar los sistemas en el tiempo).

Como se acaba de comentar en la introducción, la solución pueden proporcionarla las *líneas de producto*, ya comentadas en el capítulo 4. Llegados a este punto, es conveniente recordar las definiciones propuestas por Bosch [Bosch 2000] y Clements [Clements et al 1998]:

"Una línea de productos consiste en una arquitectura de línea de productos y en un conjunto de elementos software reutilizables que han sido diseñados para su incorporación en dicha

64 Aunque existen diferentes modelos de componentes actualmente disponibles (CCM, .NET, .COM, DCOM, Java Beans, EJB) está por ver hasta que punto se adaptan a las características del GOYA.

arquitectura. Adicionalmente, la línea de productos incluye los productos que han sido desarrollados utilizando los assets[65] *mencionados." [Bosch 2000].*

"Una línea de productos software es un conjunto de sistemas intensivos en software que comparten un conjunto común y gestionado de características que satisfacen las necesidades específicas de un particular segmento de mercado o misión y que son desarrollados a partir de un conjunto de assets *comunes de una forma determinada." [Clements et al 1998].*

El enfoque que sigue esta tesis para definir la línea de producto del GOYA puede considerarse una combinación de ambas definiciones, donde se tiene en cuenta la especificidad dada por Clements:

…necesidades específicas de un particular segmento de mercado o misión...

y la clasificación de elementos (*assets*) dada por Bosch[66]:

...arquitectura, elementos software reutilizables,…productos desarrollados.

Los elementos software reutilizables a los que se refiere Bosch incluyen la arquitectura de la línea de producto, los productos desarrollados y sus componentes. Pero no se reducen a estos. Elementos reutilizables son también la especificación de requisitos realizada en el capítulo 5 y el modelo de análisis definido por Álvarez en [Alvarez 1997]. Retomando el hilo principal del discurso, el enfoque del GOYA como línea de producto debe:

- Simplificar el problema de forma que pueda resolverse con un esfuerzo *razonable*.
- Proporcionar un marco lo suficientemente amplio para la reutilización del software.

Empezando por el segundo de estos puntos, en el GOYA:

- Se consideran varios sistemas teleoperados trabajando coordinadamente, cada uno de ellos mostrando diferentes grados de variabilidad en cuanto a sus prestaciones, opciones y precios.
- Existen diversas combinaciones posibles en cuanto al conjunto de mecanismos teleoperados que pueden formar parte del sistema.
- Se considera un abanico relativamente amplio de futuros usos de los sistemas que pueden dar lugar a nuevos productos o a nuevas versiones de los existentes.

Es decir, no estamos hablando de un único producto, sino de una familia de productos relacionados que comparten un conjunto común de requisitos, pero que al mismo tiempo exhiben una variabilidad significativa respecto de los mismos. Existe, por tanto, un conjunto de sistemas lo suficientemente amplio como para diseñar de cara a la reutilización. Pero *¿No será tan amplio que nuevamente impida la adopción de una solución general?* El desarrollo del GOYA presenta una serie de características que simplifican la gestión de la variabilidad:

---

[65] Como se indicó en el capítulo 2, en la terminología técnica de las líneas de producto, los *assets* son aquellos elementos comunes que pueden reutilizarse en la línea de producto.

[66] En realidad en [Bosch 2000] también se considera la especificidad de los productos y [Clements 2000] clasifica los *assets* de manera análoga. Sin embargo, las definiciones que proporcionan enfatizan aspectos diferentes.

- Dicha variabilidad está inscrita dentro de unos propósitos bien definidos (los sistemas realizan misiones muy concretas) que puede caracterizarse con un alto grado de confianza, siendo posible identificar los diferentes productos de la familia.
- Todos los sistemas de la familia están estrechamente relacionados por su funcionalidad y en caso de duda, todas las partes implicadas pueden ser reunidas y consultadas.
- Aunque la mayoría de los escenarios descritos en el capítulo 5 son aplicables al GOYA, las características y propósito del sistema permiten concretarlos en gran medida, restringiendo su alcance y resolviendo ambigüedades.

Así, el enfoque del problema como una línea de producto simplifica su resolución, pero manteniendo un grado de generalidad suficiente. Esta simplificación se hace a costa de restringir la solución a una familia de productos concreta, pero no hay nada que impida que el conocimiento y la experiencia obtenidos durante el desarrollo del GOYA puedan ser utilizados posteriormente en otros proyectos.

Plantear la resolución del GOYA como una línea de producto permite aprovechar las técnicas de análisis de la ingeniería de dominio para satisfacer ciertas necesidades o expectativas del mercado, casi siempre muy concretas. Si consideramos que una familia de productos define un dominio, la arquitectura de dicha familia de productos es precisamente la arquitectura de referencia definida para dicho dominio. Este enfoque, aunque es coherente con la definición de dominio *("cuerpo especializado de conocimiento, un área de experiencia o una colección de funcionalidad relacionada"* [Clements et al 1998]) se utiliza, sin embargo, de manera oportunista. El dominio no se define en función de su coherencia funcional, sino en función de los requisitos de las misiones a realizar y de las expectativas del segmento de mercado al que van dirigidos.

Las ventajas de enfocar el GOYA como una línea de producto pueden resumirse en los siguientes puntos:

1. Reduce considerablemente el dominio, pero mantiene el suficiente grado de generalidad para que la solución sea aplicable a un conjunto de productos.
2. Se conoce las variaciones y extensiones de los productos de la línea a corto y medio plazo. Los requisitos pueden definirse sin ambigüedades. Todas las partes implicadas están disponibles para definir escenarios y resolver dudas.
3. De cara a la evaluación de la arquitectura, los escenarios pueden priorizarse. En el caso de las líneas de producto los escenarios más importantes son aquellos relacionados con los puntos de variación de la arquitectura, y estos puntos de variación pueden determinarse.
4. Puede aprovecharse la experiencia adquirida en trabajos previos, en particular:
   - El modelo de análisis de Álvarez [Álvarez 1997], que puede particularizarse para el GOYA.
   - Los escenarios descritos en el capítulo 5, ejemplarizándolos con las características propias del GOYA.
5. A medio o largo plazo la solución alcanzada puede generalizarse para considerar un dominio más amplio. Es previsible que algunos de los *assets* desarrollados puedan reutilizarse en sistemas muy distintos al GOYA. Para que ello sea posible es necesario definir las estrategias de diseño que permitan definir o escoger un modelo de componentes que promueva la reutilización de los mismos.

8.3 Características del sistema GOYA.

8.3.1 Justificación y propósito del sistema:

A corto plazo, el propósito del sistema GOYA es proporcionar los medios para una limpieza semiautomática y respetuosa con el medio ambiente de los cascos de embarcaciones de gran tamaño [GOYA 1998b]. Esta limpieza implica la eliminación de pinturas con componentes altamente contaminantes y cancerígenos que incluyen en su composición metales pesados. Los medios de limpieza actuales consisten en chorrear el casco con diferentes granallas que se proyectan sobre el mismo a alta presión. Estas operaciones, que se realizan manualmente sobre andamios, implican riesgos para los operarios y provocan la dispersión de las granallas y los materiales de desecho en extensas áreas alrededor de los astilleros. La mayor parte de los países de la Unión Europea han prohibido la realización de estas operaciones en tanto no se resuelvan los problemas que acaban de mencionarse. La imposibilidad de realizar estas tareas supone una importante pérdida de competitividad para los astilleros, ya que los armadores buscan servicios de mantenimiento integrales. No poder realizar la limpieza del casco supone casi siempre la pérdida de un cliente. Así, los principales objetivos del GOYA son:

- Alejar a los operarios de la zona de limpieza.
- Confinar los residuos resultantes.
- Mantener o reducir el tiempo y los costes de las operaciones de limpieza.

A más largo plazo, el sistema GOYA debe proporcionar una plataforma para la realización de trabajos de mantenimiento del casco más complicados (pintado, soldadura, reparaciones de diverso tipo, etc.) que actualmente son, como la limpieza, llevados a cabo de forma manual por operarios que trabajan sobre grúas o andamios.

8.3.2 Elementos principales del sistema.

Los componentes principales del sistema GOYA son:

- Un vehículo-grúa que porta al resto de los mecanismos y cuyo objetivo es posicionar al sistema en el paño del casco donde se debe realizar la limpieza.
- Mecanismos posicionadores que sitúan al cabezal de limpieza sobre diferentes áreas del paño.
- El cabezal de limpieza, cuyas características varían en función de la tecnología empleada (láser, agua o granalla), de las granallas utilizadas y de los parámetros de calidad exigibles a la limpieza.
- Un sistema de inspección de la calidad de la limpieza obtenida, que funciona en línea con el cabezal.
- Un sistema de sensorización que puede incluir utilidades de visión artificial.
- El sistema de confinamiento de residuos y de reciclado de la granalla.

La figura 8.1 (repetición de la 7.5 para comodidad del lector) muestra un prototipo del GOYA en el que pueden observarse algunos de estos componentes. La figura 8.1 es bastante útil a efectos ilustrativos, aunque el sistema final probablemente se parezca poco al que muestra

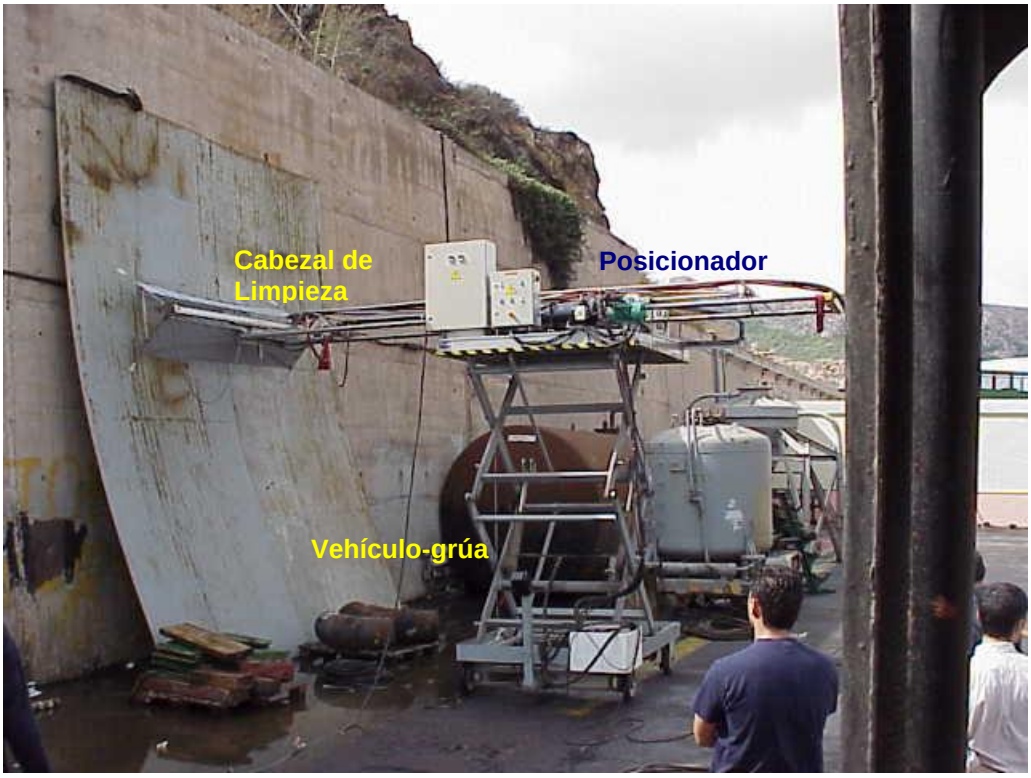


Figura 8.1: Prototipo del sistema GOYA durante pruebas en astillero

8.3.3 Características y requisitos del sistema.

Los requisitos del sistema GOYA están descritos con todo detalle en [GOYA 1998b] y [EFTCoR 2001]. No se trata de agobiar aquí al lector con una enumeración detallada de los mismos, sino de resumir algunas características que permitan entender el tipo de sistemas que se pretende desarrollar y relacionar sus propósitos, componentes y requisitos con:

- El enfoque como línea de producto que se proponía al inicio de esta sección.
- Los requisitos del modelo de componentes que se describen en las secciones siguientes.

En primer lugar, se pretende que el proceso de limpieza se realice no ya de forma teleoperada, sino semiautomática. El sistema GOYA está a medio camino entre los robots industriales y los robots teleoperados. Por un lado, el sistema debe ser capaz de actuar de forma autónoma, limpiando paños enteros del casco de un buque sin la asistencia o supervisión constante del operador. Por otro, dadas las características del entorno de operación, no puede liberarse de una cierta supervisión. En general, durante la mayor parte del tiempo, el comportamiento reactivo

predomina sobre el deliberativo (órdenes del operador), pero cuando éste ocurre tiene prioridad sobre aquél.

Los movimientos deben adaptarse al contorno del casco y a las singularidades de su superficie. El entorno es en gran medida no estructurado, de forma que la información sobre dicho entorno debe obtenerse a partir de diferentes formas de sensorización. Los cascos de los barcos tienen superficies relativamente fáciles de recorrer y otras muy complicadas, bien por la presencia de abultamientos, agujeros, salientes y ventanas, bien por su geometría, plana en unas zonas y muy curvada en otras.

La calidad del acabado es un parámetro importante, que depende del avance del cabezal de limpieza, de la presión del chorreo y de la granalla empleada[67]. El funcionamiento del cabezal de limpieza y del posicionador deben sincronizarse de acuerdo con el desarrollo de la operación. La posición del vehículo-grúa depende del paño y se mantiene constante mientras éste no cambia.

Dadas las dimensiones de los cascos, puede haber varios mecanismos realizando operaciones de mantenimiento simultáneas en diversas partes del mismo. Hay que evitar solapamientos y colisiones. Deben existir servicios que permitan definir las *misiones* a realizar y repartirlas entre los diferentes sistemas.

La fiabilidad, la disponibilidad y especialmente la seguridad son requisitos importantes. En los entornos industriales el tiempo es dinero. La tasa de fallos debe ser muy pequeña y el tiempo de recuperación mínimo. En ningún caso el tiempo de operación puede ser superior al empleado con las técnicas actuales y deben proporcionarse modos de funcionamiento degradados que permitan realizar las operaciones en caso de un fallo parcial del sistema. En particular:

- Si falla el modo de funcionamiento automático, el sistema debe poder trabajar en un modo puramente teleoperado.
- En caso de fallo general del sistema de control los dispositivos deben poder accionarse directamente desde *botoneras* provistas a tal efecto.

A corto plazo, las principales fuentes de variabilidad entre sistemas están constituidas por:

- El empleo de diferentes tecnologías de limpieza: chorreo con granalla, chorreo de agua a alta presión y láser.
- La posibilidad de utilizar vehículos-grúa diferentes, adaptados a diferentes tamaños y formas del casco.
- La posibilidad de utilizar diferentes posicionadores del cabezal, adaptados, como en el caso anterior, a diferentes tamaños y formas del casco.

El vehículo-grúa debe ser un mecanismo comercial, en tanto que los cabezales y posicionadores serán en general diseños específicos, que sin embargo estarán en su mayor parte constituidos por componentes mecánicos, eléctricos, neumáticos e hidráulicos comerciales. Hay una necesidad muy fuerte de poder integrar dispositivos hardware específicos asociados a tales componentes y habitualmente suministrados con los mismos.

A medio plazo la variabilidad viene determinada por la ampliación de las operaciones de mantenimiento a realizar en el casco del buque. Puesto que el sistema puede extenderse para

[67] Si se utilizan tecnologías láser o agua a presión los parámetros son evidentemente otros. Las técnicas de chorreo son las más empleadas.

considerar otras operaciones de mantenimiento, los requisitos de adaptabilidad a nuevos mecanismos y misiones son muy importantes. Operaciones como el pintado apenas suponen nuevos requisitos, pero la soldadura o el mecanizado son muy exigentes en términos de precisión del posicionamiento. Así:

- La precisión de posicionamiento y la complejidad de la cinemática tenderán a hacerse mucho más exigentes.
- Aspectos relacionados con el comportamiento dinámico de los mecanismos y con la flexión de los posicionadores, que pueden despreciarse para la limpieza, tendrán que tomarse en consideración.
- El sincronismo entre posicionador y herramienta se irá haciendo cada vez más complejo.
- Los servicios de definición y reparto de misiones deberán ampliarse para considerar las nuevas operaciones.
- Los requisitos de las nuevas operaciones pueden forzar a emplear brazos industriales comerciales que proporcionen las suficientes prestaciones.

Otras fuentes importantes de variabilidad tanto a corto como a largo plazo son:

- ✓ Puede cambiar hardware específico asociado a los accionamientos y a la sensorización.
- ✓ Según las capacidades exigibles a los sistemas, parte de los componentes pueden implementarse en hardware o en software.
- ✓ Pueden cambiar la infraestructura de comunicaciones. Existe un interés muy fuerte por parte de los astilleros en el uso a medio plazo de tecnologías de comunicaciones inalámbricas
- ✓ Pueden cambiar las plataformas de ejecución dentro de ciertos límites.

8.4 Identificación de los Subsistemas principales.

En el capítulo 4 se presentó el ABD o Método de Diseño Basado en Arquitectura. Como allí se explicó, dicho método está específicamente pensado para el desarrollo de líneas de producto y arquitecturas de referencia y se basa en:

- La descomposición funcional del problema y la realización de los requisitos de calidad y de negocio a través de la elección de los estilos arquitecturales adecuados.
- El uso de *plantillas software* que modelan, por un lado, los patrones de interacción de los componentes del sistema entre sí y con la infraestructura y, por otro, el conjunto de responsabilidades comunes que dichos elementos deben asumir.

Las entradas del método están constituidas por la lista de requisitos y restricciones que debe satisfacer la arquitectura. En el caso del GOYA dichos requisitos y restricciones son los que se han resumido en la sección anterior. Se está por tanto en condiciones de comenzar a ejecutar el método.

Como se explicó en el capítulo 4, el método ABD descompone el sistema en subsistemas recursivamente. Es decir, las mismas reglas que se aplican para descomponer el sistema en susbsistemas se aplican a su vez para descomponer dichos subsistemas en otros más simples. Los requisitos funcionales se logran asignando una parte de la funcionalidad del sistema a cada

uno de los subsistemas o componentes resultantes de las sucesivas descomposiciones del mismo. Los requisitos no funcionales se alcanzan mediante la selección del estilo arquitectural que mejor se adapte a los mismos. La experiencia acumulada en el desarrollo de otros sistemas teleoperados permite identificar los subsistemas siguiendo un proceso de diseño descendente, en el que todos los subsistemas resultantes de la primera descomposición funcional pueden ser identificados.

Las plantillas software definen:

- ✓ La **interacción con los servicios:** Patrones que describen como un tipo determinado de elementos de diseño interacciona con los servicios provistos por el resto de los elementos de diseño.
- ✓ La **interacción con la infraestructura.** Patrones que describen como un tipo determinado de elementos de diseño interacciona con la infraestructura.
- ✓ Las **responsabilidades comunes.** Responsabilidades que son comunes a todos los elementos de diseño de un tipo determinado. Por ejemplo, la forma en que un determinado tipo de elementos debe llevar a cabo el tratamiento de las excepciones.

En las tres secciones siguientes se presentarán respectivamente:

- ✓ Los subsistemas principales.
- ✓ Una primera definición de las plantillas software.
- ✓ Una primera propuesta de componentes para el sistema.

8.4.1 Subsistemas principales

En el caso del GOYA el proceso está en sus inicios, pero se cuenta con la experiencia del desarrollo de sistemas de teleoperación anteriores. Dicha experiencia permite identificar los subsistemas principales y seguir un proceso de diseño descendente, en el que dichos subsistemas tendrán que ser refinados progresivamente. En primera instancia, pero sin descartar que más adelante puedan definirse otros, en todos los sistemas de la línea existirán (o podrán existir) al menos los siguientes subsistemas:

- Controladores de bajo nivel de los mecanismos físicos.
- Sistemas de seguimiento e inspección y seguimiento de las operaciones.
- Sistema de navegación.
- Sistema de diagnóstico, monitorización de recuperación.
- Gestor de Aplicación.
- Sistema de configuración e instalación de las aplicaciones.
- Sistema de definición de misiones.
- El sistema ejecutor de misiones.
- Interfaces de usuario.

La figura 8.2 muestra un resumen de sus responsabilidades y sus interfaces conceptuales.

Controladores de bajo nivel.

La responsabilidad de los controladores de los mecanismos de más bajo nivel es la sensorización directa y la actuación y típicamente incluyen PIDs y tarjetas de entrada/salida a las que se conectan los dispositivos de actuación y sensorización. Es muy probable que estén implementados, al menos parcialmente, en hardware y casi siempre deben satisfacer requisitos temporales estrictos. Puesto que cualquiera de estos elementos puede cambiar de un sistema a otro, en orden a facilitar su sustitución o adaptación es necesario separar los controladores de cada mecanismo, definiéndose controladores para:

- El vehículo-grúa.
- El posicionador
- El cabezal-herramienta.

Que deben

- Ser completamente independientes.
- Disponer de una interfaz de configuración que permita configurar controladores genéricos.
- Disponer de una interfaz de diagnóstico que permita testar su funcionamiento.
- Ofrecer una interfaz uniforme al resto de componentes o subsistemas.

Subsistemas de Inspección y seguimiento de las operaciones.

La responsabilidad fundamental de los sistemas de inspección y seguimiento es comprobar que la operación a realizar (en principio la limpieza, pero más adelante también otro tipo de operaciones) se realiza de acuerdo con una serie de parámetros preestablecidos. La principal salida de estos sistemas es el estado de la operación en curso, que para el caso del sistema de limpieza es la calidad de la limpieza obtenida.

Una característica importante de estos sistemas es que deben funcionar en línea con los controladores y que sus salidas pueden utilizarse para generar consignas de movimiento para el posicionador o de activación para las herramientas, en función de los acabados obtenidos (sistema de limpieza) o del estado de la operación (p.e: aporte de material en operaciones de soldadura).

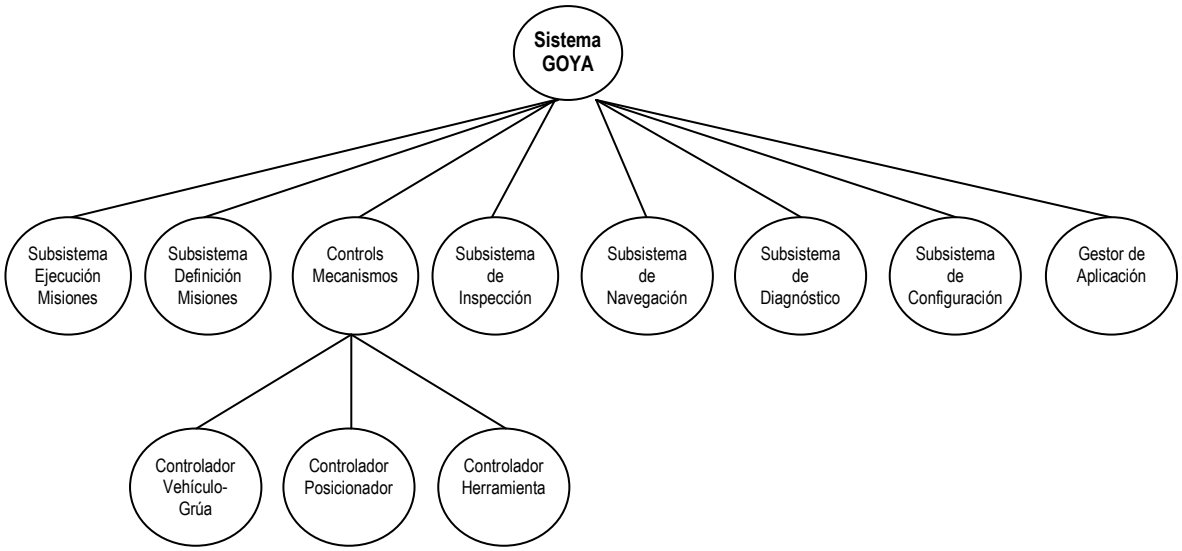
Obsérvese que estos componentes se definen en plural *"subsistemas"*. Así, pueden utilizarse diferentes subsistemas de inspección y seguimiento según los sistemas (p.e.: sistemas basados en visión artificial, en ultrasonidos o en palpadores) y un subsistema de inspección dado puede consistir en la agregación de varios subsistemas más pequeños, tomándose las decisiones en función de las salidas de todos ellos.

A pesar de que la principal responsabilidad de estos sistemas es comprobar que la operación se realiza de acuerdo con una serie de parámetros preestablecidos, habitualmente se implementan a partir de utilidades que tienen capacidad suficiente para asumir otras responsabilidades. Por ejemplo, un sistema de visión artificial puede:

- Determinar la calidad del acabado.
- Detectar obstáculos y si es lo bastante inteligente determinar rutas alternativas.

En este caso ambos tipos de servicios deben ofrecerse a través de interfaces claramente diferenciadas, pues bien podrían realizarse por componentes diferentes. Los servicios de navegación (detección de obstáculos y determinación de rutas seguras u óptimas) y los de

inspección son lógicamente diferentes y su realización puede ser realizada por componentes diferentes. Estos sistemas constituyen un punto de variación del sistema a medio plazo. A medida que se vayan incorporando nuevas operaciones de mantenimiento deberán incorporarse sistemas de seguimiento específicos de las mismas.



Subsistema	Responsabilidades	Entradas	Salidas
Controladores Mecaqnismos: Vehículo-Grúa. Posicionador. Herramienta	Actuación Directa Sensorización. (control de bajo nivel)	Comandos de movimiento. Accionamiento. Parámetros de Configuración	Sensorización Estado Información de diagnóstico
Subsistema Inspección.	Seguimiento proceso de herramienta. En primera instancia control de la calidad del acabado del proceso de limpieza	Sensorización. (Imágenes, rugosidad, textura, etc.) Parámetros de Configuración	Estado del proceso. (calidad del acabado) Info. Diagnóstico
Subsistema Navegación	Evitar colisiones. Determinar ruta de avance	Sensorización. (Imágenes, detectores de proximidad, etc.) Parámetros de Configuración	Rutas Flag de colisiones. Info. Diagnóstico
Subsistema Definición de Misiones	Definición de misiones. Reparto de misiones.	Parámetros misión. Sistemas implicados.	Misiones
Subsistema Ejecución de Misiones	Ejecutar misiones. Coordinar mecanismos.	Misión. Estado Mecanismos. Estado Proceso (De sist insp). Rutas, flag colisiones. Parámetros de Configuración	Comandos para mecanismos. Estado misión. Info. Diagnóstico
Subsistema Monitorización y Diagnóstico	Monitorizar estado y funcionamiento del resto de los subsistemas	Info diagnóstico. Parámetros de Configuración	Info diagnóstico.
Interfaces de Usuario	Acceso Comandos Mostrar Estado del sistema	Estado sistema Parámetros de Configuración	Comandos Info diagnóstico.
Subsistema de Configuración	Configurar resto de subsistemas. Comprobar consistencia de las configuraciones e interacciones entre subsistemas.	Parámetros de configuración	Info Diagnóstico
Gestor de Aplicación	Arranque y parada del sistema. Activación/Desactivación de subsistemas Gestión acciones de recuperación de alto nivel (Políticas de tolerancia a fallos).	Parámetros de configuración	Creación/desrtrucción/activación/ Desactivación de subsistemas.

Figura 8.2: Resumen de responsabilidades de subsistemas.

Sistema de Navegación.

Las responsabilidades más inmediatas de este componente son:

- Detectar obstáculos o situaciones en las que el sistema no puede progresar.
- Asistir al tele-operador proporcionándole unos datos o unas imágenes que le permitan mover de forma segura los dispositivos.

Sin embargo, es uno de los sistemas cuya funcionalidad es más susceptible de ser incrementada para:

- Determinar rutas libres de obstáculos.
- Optimizar rutas y secuencias de movimiento.

Estos servicios podrán utilizarse tanto en línea como fuera de línea. Fuera de línea para pre-programar una secuencia de movimientos óptima, en línea para sortear un obstáculo inesperado. Aunque este tipo de servicios no se consideran para las primeras versiones de los sistemas son una ampliación natural de los mismos, en orden a conseguir un proceso tan automático como sea posible.

Nuevamente, si el sistema de navegación se implementa a partir de utilidades de visión artificial existe la tentación de mezclar estos servicios con los de inspección y seguimiento. Dicha tentación, por las razones que acaban de explicarse, debe ser evitada.

Subsistema de monitorización y diagnóstico.

El objetivo de estos componentes es separar las actividades de control de las de monitorización. La separación de estas dos actividades, junto con la existencia de interfaces de diagnóstico permite definir estrategias de tratamiento de fallos jerárquicas e implementar estrategias de recuperación a diferentes niveles. La definición de este tipo de componentes debe ser posterior a la de los componentes a monitorizar. Sin embargo, aunque todavía no se pueda decir mucho de ellos, su mera existencia impone condiciones al resto de componentes. Así:

- Los componentes deben ofrecer interfaces de diagnóstico.
- Deben diseñarse pensando que deben ofrecerlas.
- Todos los componentes de un tipo determinado deben ofrecer la misma interfaz de diagnóstico o al menos un conjunto de servicios de diagnóstico comunes.

Cuanto más regulares sean las interfaces de diagnóstico más posibilidades habrá de que los componentes sean realmente intercambiables.

Sistema de configuración e instalación de las aplicaciones.

Las responsabilidades de este sistema son:

- Ofrecer una interfaz para la configuración de los componentes.
- Ofrecer una interfaz para la instalación y desinstalación de componentes y para su conexión y desconexión con otros componentes.
- Comprobar que los componentes se configuran e instalan de forma coherente.

La existencia de este subsistema se justifica por la necesidad de configurar los sistemas en función de los mecanismos que incluyan y de las capacidades que deban poseer. Por ejemplo:

- Diferentes sistemas pueden incluir diferentes vehículos, posicionadores y cabezales. Cada uno de estos mecanismos tiene asociado un controlador de bajo nivel. Debe existir una

forma de configurar los controladores según el mecanismo empleado o de sustituir unos controladores por otros.

- Diferentes sistemas pueden necesitar diferentes componentes de inspección y seguimiento. Nuevamente, debe existir una forma de configurar estos componentes o de sustituirlos por otros.

Como en el caso anterior, es prematuro definir los servicios concretos que debe ofrecer, pero (también como en el caso anterior) su mera existencia impone condiciones al resto de componentes que:

- Deben ofrecer una interfaz de configuración.
- Deben diseñarse pensando en que deben tenerla.
- Deben ofrecer y requerir servicios a través de las mismas interfaces (para que sea posible su sustitución).
- Todos los componentes de un mismo tipo deben instalarse de la misma manera.

El sistema de configuración e instalación proporciona una forma explícita de manejar la variabilidad entre productos, aunque no pueda (ni deba) hacerse cargo de toda ella. En particular, cuando la línea del GOYA se amplíe para considerar nuevas operaciones de mantenimiento es muy probable que haya que realizar ingeniería software de aplicación e incluso modificar o ampliar la arquitectura base. Sin embargo, este sistema resulta imprescindible para poder configurar un producto dado y manejar su evolución a lo largo del tiempo.

El sistema ejecutor de misiones.

La responsabilidad de este sistema es coordinar los mecanismos del GOYA para que realicen la misión que les encomiende el operador.

Los mecanismos del GOYA, con la posible excepción en los primeros sistemas del vehículo-grúa, deben trabajar simultáneamente y de forma autónoma para realizar su *misión* durante largos periodos de tiempo. Inicialmente dicha misión consiste en limpiar una superficie del casco del barco (un paño) con una calidad de acabado dada.

Las entradas de este sistema están constituidas por:

- Los parámetros de la misión, principalmente los límites de la zona a limpiar y la calidad del acabado, pero posiblemente también otros como los parámetros de la herramienta (p.e: presión de chorreo máxima) y secuencia de movimientos óptima del posicionador[68].
- La información suministrada por el sistema de inspección y seguimiento (calidades de acabado que se van obteniendo).
- La información suministrada por el sistema de navegación (presencia de obstáculos, rutas de libramiento, etc).
- La información suministrada por los controladores de los mecanismos (nivel de presión del cabezal, activación de finales de carrera, esfuerzo de los motores, activación de alarmas, etc).
- Las órdenes del operador.

[68] Obviamente si el sistema se amplía para realizar otras operaciones los parámetros serán otros.

Las salidas son órdenes para los controladores del posicionador (movimientos) y del cabezal (activación y desactivación y regulación)

La coordinación de los mecanismos puede modelarse mediante máquinas de estado que implementen *estrategias* de sincronización más o menos complejas. Independientemente de su complejidad, estas estrategias están destinadas a cambiar y a refinarse, especialmente durante la fase de desarrollo del sistema, hasta que se dé con los algoritmos óptimos. Por ello, deben ser fácilmente intercambiables.

Hay que evitar asignar a este sistema responsabilidades que no son suyas. En concreto, dado que este sistema maneja información procedente de muchos otros subsistemas existe la tentación de asignarle tareas de monitorización que no debe asumir. La monitorización del comportamiento de los subsistemas debe realizarse independientemente de que se esté realizando o no una misión y del estado de la misión en curso. Obviamente, la misión debe suspenderse si los sistemas no funcionan correctamente, pero este hecho lo que determina es la necesidad de una nueva entrada para este sistema, no que deba asumir dichas monitorizaciones.

La intervención del operador supone la interrupción de la misión en curso. Acabada dicha intervención la misión puede proseguir o no, con los mismos o con diferentes parámetros. La implementación de este tipo de comportamiento puede facilitarse si las misiones admiten comandos del tipo **start()**, **stop()**, **suspend()**, **resume()** y **abort()**.

Otro aspecto a tener en cuenta es la posibilidad de ejecutar una misma misión con una configuración de mecanismos diferente, por ejemplo con otro posicionador o con un cabezal distinto. Para que ello sea posible es necesario que todos los componentes de un determinado tipo ofrezcan la misma interfaz.

Sistema de definición de misiones.

Las operaciones de limpieza (u otras que pudieran definirse) no se realizan siempre de la misma manera ni sobre el mismo tipo de superficies. Cada misión tiene sus propios parámetros y particularidades. Es necesario proporcionar servicios para que el operador pueda:

- Definir operaciones de mantenimiento.
- Asignar dichas operaciones a un sistema determinado (recuérdese que puede haber varios sistemas funcionando de forma simultánea).

Interfaces de usuario

Su responsabilidad es proporcionar al operador acceso a los servicios del sistema y mostrarle de forma fidedigna el estado del mismo.

Las interfaces de usuario pueden ser muchas y acceder a cualquiera del resto de los subsistemas. Las restricciones respecto a su número y organización debe imponerlas la arquitectura de cada sistema.

Las interfaces de usuario son uno de los subsistemas más proclives a sufrir variaciones tanto durante el desarrollo del sistema como a lo largo de su vida operativa. Es imposible evitar que las modificaciones de los subsistemas a los que acceden se propaguen a las mismas. Si se amplían los servicios que proporciona un subsistema habrá que añadir el control correspondiente. Si se produce nueva información de estado habrá que representarla de alguna manera. Sin embargo, sí puede evitarse fácilmente que las modificaciones en las interfaces de usuario se propaguen al resto de los subsistemas y con un diseño cuidadoso de las mismas es posible añadir y suprimir fácilmente elementos de las mismas.

Gestor de Aplicación (Application Manager)

Finalmente, debe existir un susbsistema encargado:

- ✓ De monitorizar el sistema en su conjunto.
- ✓ De arrancar los sistemas según una secuencia correcta.
- ✓ De ubicar los diferentes componentes de la aplicación.
- ✓ De implementar estrategias de tolerancia a fallos.

8.4.2 Plantillas software.

Las plantillas software se refieren por un lado a los patrones de interacción entre los componentes y entre los componentes y la infraestructura y, por otro, a las responsabilidades comunes de los componentes.

Aunque la subdivisión del GOYA en subsistemas que se propone en el apartado anterior es aún de muy alto nivel y debe ser refinada en todos sus extremos, es ya posible identificar las primeras plantillas software, que deberán ir refinándose al mismo ritmo que los propios susbsistemas. En su definición actual, las plantillas constituyen más bien requisitos que decisiones de diseño. Las plantillas tienen una influencia directa sobre las interfaces que debe ofrecer un subsistema y muy especialmente sobre la forma en que pueden o deben definirse dichas interfaces.

- ✓ Todos los subsistemas de un tipo dado deben *ofrecer y requerir* servicios de la misma manera. Obsérvese que no tienen que ofrecer y requerir necesariamente los mismos servicios, sino requerirlos y ofrecerlos de la misma manera.

  Por ejemplo, considérense dos subsistemas de control de bajo nivel correspondientes a dos posicionadores del cabezal. Uno de ellos es un sencillo mecanismo sin transformada cinemática inversa que sólo admite comandos de movimiento en modo *jog* (movimientos angulares). El otro es un auténtico robot con transformada cinemática inversa, que admite comandos de movimiento en modos *jog* y *joystick* (movimientos cartesianos). Aunque el segundo controlador proporciona más comandos, debe proporcionar el mismo acceso a comandos en modo *jog* que el primero.

  Considérense ahora dos subsistemas de ejecución de misión o el mismo subsistema configurado para realizar dos misiones diferentes. La ejecución de la primera misión requiere los servicios del sistema de inspección y del sistema de navegación. La segunda es más sencilla y sólo requiere alguno de los servicios del sistema de inspección. Ambos sistemas deben requerir los servicios del sistema de inspección de la misma manera.

- ✓ Todos los susbsistemas de un tipo dado deben configurarse de la misma manera. Nuevamente, no todos los subsistemas deben ofrecer las mismas capacidades de configuración, pero las que ofrezcan deben ofrecerlas de manera uniforme.

  Por ejemplo, condiderénse dos subsistemas de control de bajo nivel correspondientes a dos posicionadores del cabezal. Uno de ellos admite la parametrización de los parámetros de control proporcional (P) e integral (I) de los ejes que controla, el otro sólo admite la configuración del término (P). La configuración del término proporcional debe hacerse igual en ambos controladores.

- ✓ Los subsistemas deben ser diagnósticables:

Las condiciones de error detectadas por un subsistema durante su despliegue, activación o funcionamiento deben ser:

- O bien publicadas para que el sistema de monitorización y diagnóstico pueda subscribirse a las mismas.
- O bien ofrecidas a través del acceso a un servicio.

Todos los subsistemas de un tipo dado deben informar de sus errores de la misma manera. Los errores comunes a varios tipos de sistemas deben ser reportados por todos ellos de la misma manera.

✓ Relaciones con la infraestructura:

El acceso a los servicios del sistema operativo o a la infraestructura de comunicaciones debe realizarse a través de interfaces abstractas.

Los servicios del sistema operativo deben limitarse a aquellos definidos en la norma POSIX.

Puesto que hay una necesidad muy fuerte de integrar hardware específico desarrollado por terceros, es necesario acceder a sus drivers a través de interfaces abstractas y definir procedimientos de instalación para dichos dispositivos.

8.4.3 De subsistemas a componentes.

Siguiendo las pautas del ABD, tal y como se explicó en el tema 4, una vez establecidos los susbsistemas principales, éstos se dividen a su vez en subsistemas más pequeños, hasta llegar a los componentes conceptuales, cuya realización debe dar lugar a componentes concretos. Estos componentes constituirían en nuestro caso los *componentes base* resultantes de aplicar la ingeniería de dominio, susceptibles de ser reutilizados (después de alguna configuración) en todos los sistemas de la línea.

Sin embargo, en este caso, es conveniente que la mayor parte de los subsistemas identificados en el apartado anterior se correspondan directamente con componentes base. En concreto, pueden considerarse *componentes base*:

- Los controladores de bajo nivel de los dispositivos.
- Las interfaces de usuario.
- El susbsistema de seguimiento
- El subsistema de navegación.
- El subsistema de ejecución de misiones.
- El subsistema de definición de misiones.

La promoción de estos subsistemas a componentes base viene motivada por las siguientes razones:

- Estan presentes en todos los sistemas.
- Sus responsabilidades (funcionalidad) están bien definidas y pueden separarse claramente del resto.
- Se corresponden con partes del sistema directamente relacionadas con sus puntos de variación. Son unidades a configurar o a sustituir.

- No son meros subsistemas lógicos, sino auténticos susbsistemas físicos que, en el caso de los controladore,s pueden ser implementados en hardware y que en todos casos se corresponden o pueden corresponderse con procesos independientes.
- Constituyen unidades de distribución del sistema, ya que pensados para ejecutarse íntegramente en un nodo:
  - ✓ *Pueden* diseñarse e implementarse ignorando aspectos de distribución
  - ✓ Sus interfaces *deben* definirse pensando en que van a estar distribuidos en distintas máquinas (sólo paso de mensajes), aun cuando en un producto determinado puedan ejecutarse en un mismo nodo o desplegarse de diferentes formas.

Además, la definición de componentes base de grano grueso presenta en las fases iniciales de desarrollo una clara ventaja respecto a la definición de componentes base de grano fino: permite diferir las decisiones de diseño e implementación de los subsistemas sin comprometerse prematuramente con un enfoque determinado.

Los componentes base son las unidades de composición del sistema. Puesto que estas unidades están pensadas para su reutilización, la consideración de un componente como componente base exige que:

(1) Sea previsible su reutilización para la construcción de otros sistemas y

(2) Tenga unas interfaces y un comportamiento (funcionalidad) estable.

Definir componentes base de grano fino requiere un conocimiento del sistema y de su forma de evolución que aún no se tiene. A medida que avance el desarrollo de los sistemas, ciertos componentes de grano fino podrán promocionarse a la categoría de *componentes base*.

No obstante, algunos de los subsistemas identificados en el apartado anterior no pueden considerarse, al menos no todavía, como componentes base. Así:

El subsistema de monitorización y diagnóstico es un subsistema lógico, cuyos componentes pueden estar distribuidos por todo el sistema. Cada uno de estos componentes estará necesariamente especializado en la monitorización de un cierto tipo de componentes. Además, las políticas de monitorización y diagnóstico pueden variar durante el proceso de desarrollo, hasta dar con las más adecuadas. Finalmente, diferentes sistemas pueden requerir políticas de monitorización distintas. Por todo ello, es conveniente diferir la definición de estos componentes y de las políticas de monitorización a la fase de ingeniería de aplicación. Más adelante y en función de la experiencia obtenida durante el desarrollo de los sistemas puede ser posible definir *componentes base* de monitorización.

El subsistema de configuración e instalación debe ser definido con mayor precisión y tal vez dividido en subsistemas más pequeños. Sin embargo, sus responsabilidades son complejas y es preferible diferir las decisiones sobre su diseño a fases más avanzadas del proceso de desarrollo. Además, todavía es necesario decidir qué partes del sistema son configurables y qué modificaciones dan lugar a un producto diferente de la línea. En principio, se puede definir sistemas de configuración dependientes del sistema (ingeniería de aplicación) y en función de los resultados y de la experiencia adquirida generalizar la solución al resto de los sistemas. Lo mismo puede decirse del Gestor de Aplicación.

8.5 Estrategias para el diseño. Un modelo de desarrollo basado en componentes.

8.5.1 Introducción

Aunque en el capítulo 4 ya se habló de componentes y modelos de componentes no está de más comenzar esta sección recordando algunos de los conceptos que allí se explicaban poniéndolos en relación con los objetivos que se pretenden. Al inicio de este capítulo se establecía la necesidad de adoptar estrategias de diseño e implementación que permitieran separar:

- Los patrones de interacción entre componentes de su funcionalidad.
- Los diferentes *aspectos* de dicha funcionalidad.

Y se proclamaba la necesidad de definir un marco de desarrollo que promoviera la reutilización de componentes en diferentes arquitecturas.

La separación de los patrones de interacción de los componentes de su funcionalidad exige, como han demostrado diversos autores [Medvidivic et al 1997a, 1997b] [Metha et al 2000] [Shaw et al 1996] [Andrade et al 1998, 2001] [Loques et al 2000], considerar a los conectores como entidades de diseño de primera clase, al mismo nivel que los componentes que relacionan. Por su parte, la separación de los diferentes aspectos de la funcionalidad exige la existencia de un mecanismo para extender o reducir tanto la funcionalidad de un componente como sus interfaces. En este sentido existen diversas propuestas, como la programación orientada a aspectos [Kiczales et al 1997], enfoques basados en el principio de superposición [Bosch 2000b] que pueden aplicarse también a los conectores (superposición de patrones de interacción) o el concepto prisma propuesto por Ramos y Lorenzo en METAOASIS [Lorenzo et al 2002]. Aunque todas estas propuestas son relativamente recientes y se encuentran en el límite del estado de la técnica, pueden llevarse a la práctica utilizando diversos patrones de diseño, como los que se proponen en [Andrade el al 1999] y [Loques at al 2000].

La ingeniería del software basada en componentes incorpora o puede incorporar la mayor parte de las ideas vertidas en el párrafo anterior[69] y además puede proporcionar el marco para la reutilización de componentes que se pretende. Para dar una idea de las posibilidades que implica el desarrollo basado en componentes (y también de sus dificultades) basta recordar las definiciones de Bachmann [Bachmann et al 2000]:

"Un componente es una implementación opaca de funcionalidad, utilizable por terceras partes para la composición de sistemas y que cumple un modelo de componentes"

*"Un componente implementa una o más **interfaces**. Estas interfaces reflejan las obligaciones del componente que se describen como **contratos**. Las obligaciones contractuales de los componentes aseguran que los mismos podrán ser desarrollados de forma independiente, pues obedecen ciertas reglas que determinan cómo interaccionan y cómo pueden ser distribuidos en entornos estándares."*

*"Un **modelo de componentes** es un conjunto de tipos de componente, sus interfaces y, adicionalmente, una especificación del patrón o patrones de interacción entre tipos de componentes"*

[69] Aunque no siempre de forma explícita.

Obsérvese que de acuerdo con estas definiciones:

1. No es posible adoptar un modelo de desarrollo software basado en componentes sin definir o adoptar un modelo de componentes.
2. Los modelos de componentes se apoyan sobre el concepto de contrato[70] entre componentes, que se expresan por medio de las interfaces que dichos componentes exponen. El concepto de contrato definido por Meyer [Meyer 92] se magnifica aun más si cabe en la ingeniería software basada en componentes.

La definición de Bachmann aunque es útil para introducir los conceptos de contrato y modelo de componentes deja una importante cuestión en el aire. Recordemos ahora la definición de Bosch [Bosch 2000]:

"Un componente software es una unidad de composición que ofrece interfaces explícitamente especificadas, requiere de unas interfaces e incluye unas interfaces de configuración, incluyendo además atributos de calidad" [Bosch 2000].

Es decir, los componentes deben exponer *de forma explícita* en sus interfaces no sólo los servicios que proporcionan, sino también los que requieren[71]. Los contratos involucran al menos a dos partes (una que proporciona servicios y otra que los requiere). De estas definiciones pueden deducirse de forma casi inmediata otras dos condiciones que debe cumplir el modelo de componentes:

1. El modelo de componentes debe proporcionar las reglas y mecanismos para establecer y modificar (ampliar o reducir) los contratos entre componentes.
2. El modelo de componentes debe proporcionar mecanismos o reglas de composición de componentes y mecanismos o reglas para la extensión o configuración de los componentes.

Inmediatamente surgen dos preguntas *¿existe algún modelo de componentes que se adapte a los requisitos del GOYA?* Y si no existe *¿cómo definir el modelo de componentes?*. La adopción de un modelo de componentes ya existente tiene una clara ventaja sobre la definición de uno nuevo: Gran parte de los beneficios de un modelo de componentes se basan en el consenso sobre su uso. Adoptar un modelo de componentes significa ligarse a una arquitectura y a una tecnología, pero también significa la posibilidad de integrar los componentes desarrollados por aquellos que han optado por el mismo modelo de componentes. Sin embargo, hay pocos modelos de componentes para el desarrollo de mecanismos robotizados [Albus et al 1987] [Bruyninckx et al 2002] [Hayward et al 1986] [Kapoor et al 1996] [Miller 1991] [Schull et al 2000][72] y la adecuación de modelos de componentes generales a las necesidades del GOYA está por determinar. Como se dijo en la introducción, ***el objetivo de este apartado no es definir el modelo, sino establecer una serie de requisitos que debe satisfacer el modelo que finalmente se adopte o defina.***

Finalmente, es necesario recordar que aunque la ingeniería software basada en componentes es una de las ramas más prometedoras de la ingeniería software, es también una apuesta arriesgada. Ni el CBD ni los modelos de componentes han alcanzado un nivel de madurez que permita

[70] El concepto de contrato software en el que se apoya el CBD fue definido por Meyer [Meyer 92] dice así:
[71] [Bachmann et al 2000] dedica un capítulo entero a comentar este extremo, pero no lo pone de forma explícita en la definición que ofrece.
[72] La mayoría de ellos orientados al desarrollo de controladores, pero no de aplicaciones completas.

considerarlos un marco de desarrollo estable. Actualmente existen diversos modelos de componentes cuya compatibilidad es más que discutible y que se encuentran aún en plena evolución. Ni siquiera el propio concepto de componente está claramente establecido, existiendo diferentes escuelas e interpretaciones [CCM 1999],[Microsoft 2000], [Szyperski 1998a], [Bachmann et al 2000], [Meyer 1999], [D'Souza et al 1999], [Bosch 2000][73] que aunque coinciden casi siempre en lo fundamental se diferencian en matices importantes. Los lenguajes de programación no incorporan mecanismos que soporten los conceptos de la CBD, en especial, su capacidad para definir interfaces es muy limitada, y los ADLs aun no han alcanzado madurez suficiente. En particular, como se comentó en el capítulo 3, UML tiene grandes carencias en este aspecto.

En lo que resta de capítulo se intentará, como se ha dicho, definir los requisitos mínimos que debe cumplir el modelo de componentes que se adopte o defina para el desarrollo del GOYA. Para ello, se comenzará proponiendo una serie de requisitos generales cuyo alcance para el caso del GOYA se concretará en las secciones siguientes, en las que se describirán el tipo de interfaces que deben exponer los componentes, el tipo de operaciones que deben soportar dichas interfaces, los mecanismos de composición, configuración y extensión requeridos y el tipo de interacciones que deben soportarse.

8.5.2 Requisitos generales del modelo

Como requisitos generales del modelo pueden establecerse los siguientes:

- Los conectores deben ser entidades de primera clase, al mismo nivel que los componentes.
- El modelo de componentes debe proporcionar las reglas y mecanismos para establecer y modificar (ampliar o reducir) los contratos entre componentes.
- El modelo de componentes debe proporcionar mecanismos o reglas de composición de componentes y mecanismos o reglas para la extensión o configuración de los componentes y de sus interfaces. En particular, debe proporcionar reglas para:
  - ✓ Construir componentes complejos mediante el ensamblado de componentes más sencillos.
  - ✓ Añadir y eliminar funcionalidad del componente.
  - ✓ Añadir y configurar interfaces.
- Deben poder definirse y cambiarse con facilidad los patrones de interacción entre componentes.
- Debe poder *mapearse* sobre diferentes lenguajes de programación y sobre diferentes infraestructuras. Es decir, aunque el diseño de los componentes y de sus interacciones sea independiente de la implementación deben existir reglas o mecanismos que permitan obtener una implementación a partir del diseño. Sería conveniente que el modelo proporcionara una forma de crear, ensamblar, instalar y activar los componentes.
- Las reglas de composición entre componentes deben ser simples y estar en consonancia con los patrones de interacción seleccionados. Los componentes se unen a través de los patrones de interacción, por tanto éstos son elementos clave para definir las reglas de composición.

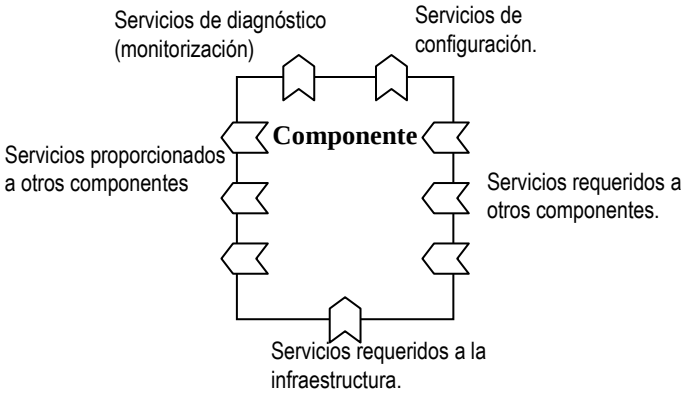
[73] No obstante, a pesar de la variedad y de las discusiones, existen más puntos de similitud que de discrepancia en la mayoría de las definiciones.

- Las reglas de composición deben poder ser verificadas formalmente y las propiedades de los compuestos deducidas a partir de las propiedades de sus componentes [Bachman et al, 2000]. Este requisito exige:
  - ✓ Que el modelo de componentes admita el empleo de lenguajes formales en las que sea posible expresar la semántica de las relaciones entre componentes (MetaOASIS, CSPs, UML-OCL, etc).
  - ✓ Que a partir de las relaciones entre componentes sea posible obtener modelos formales de comportamiento (p.e: modelo RMA correspondiente a un diseño dado)
- Debe proporcionar servicios para llevar a cabo la ingeniería de aplicación, en particular debe contar con mecanismos de extensibilidad tanto de caja blanca como de caja negra (herencia, composición, plantillas o genéricos, etc)

8.5.3 Interfaces de los componentes

Los componentes del sistema GOYA deben exponer los siguientes tipos de interfaces:

- Interfaz de servicios proporcionados.
- Interfaz de servicios requeridos.
- Interfaz de configuración.
- Interfaz de diagnóstico.
- Interfaz con la infraestructura.



La interfaz de servicios proporcionados está constituida por el conjunto de operaciones que pueden invocar los clientes del componente.

La interfaz de servicios requeridos está constituida por el conjunto de operaciones que el componente necesita de otros componentes para poder ofrecer sus servicios.

Este tipo de interfaces no pueden declararse de forma explícita en los lenguajes de programación actuales. El modelo de componentes debe proporcionar las reglas que permitan establecer las correspondencias necesarias con los lenguajes de programación o proporcionar herramientas que realicen la traducción.

La interfaz de configuración está constituida por el conjunto de operaciones que permiten configurar al componente para que trabaje de una forma determinada o para adaptar sus interfaces para que puedan ensamblarse con un determinado tipo de componentes. En general,

en esta interfaz habría que incluir todos aquellos aspectos que pueden variar frecuentemente bien entre sistemas, bien durante el funcionamiento del sistema.

Esta interfaz normalmente no es accedida por los clientes del componente (al menos no en su totalidad), sino por una entidad de más alto nivel, por ejemplo un controlador o supervisor que determine, en función de las necesidades del sistema, la forma en que los componentes deben funcionar o configurarse.

Es fácil darse cuenta de que esta interfaz está estrechamente relacionada con la capacidad de adaptación de los componentes a diferentes sistemas y situaciones y por tanto con la capacidad de usar los mismos componentes en diferentes sistemas.

La interfaz de diagnóstico puede considerarse un caso particular de la interfaz de servicios proporcionados, dónde los servicios que se ofrecen son de diagnóstico (p.e: el componente informa de su modo de funcionamiento o envía una alarma a un observador). Entonces, *¿Por qué considerarla separadamente de aquella?*

- Porque es conveniente separar la funcionalidad normal del componente de sus anomalías.
- Porque el componente que recibe el servicio no siempre puede detectar que el servidor está funcionando mal.
- Porque permite añadir componentes de monitorización de los servidores independientes de los clientes.
- Porque permite definir estrategias de tratamiento de fallos jerárquicas e implementar estrategias de recuperación a diferentes niveles.

La interfaz con la infraestructura puede considerarse un caso particular de la interfaz de servicios requeridos, donde los servicios que se requieren son los de la infraestructura (sistema operativo, middleware, utilidades, etc). Entonces, *¿Por qué considerarla separadamente de aquella?*

- Porque representa un tipo de contrato diferente.
- Porque si se separan los aspectos propios de la funcionalidad del componente de sus relaciones con la infraestructura los cambios en ésta no se propagarán al resto del componente. La interfaz con la infraestructura puede considerarse como un nivel de acceso a la misma.

Esta interfaz está relacionada con la capacidad de utilizar los componentes sobre diferentes infraestructuras. La presencia de esta interfaz no elimina los trabajos de adaptación, pero evita que los efectos de un cambio de infraestructura se propaguen por todo el componente. Tampoco significa que nuestro componente pueda ejecutarse sobre cualquier infraestructura, sino sobre aquellas que le proporcionen los servicios suficientes, aunque lo hagan de diferentes maneras.

8.5.4 Definición de las interfaces. Puertos y semántica de los puertos.

En el apartado anterior se han descrito las interfaces que debe tener un componente de los simuladores, en este se describirá cómo deben ser. En primer lugar, habría que decir que las interfaces de los componentes del GOYA no se van a definir como conjuntos de métodos, sino como conjuntos de puertos. La definición de interfaces como conjunto de métodos es propia de la programación orientada o objetos, pero presenta ciertos problemas en el caso de los componentes (véase discusión en la figura 8.3).

Cada puerto identifica un punto de interacción entre los componentes y su entorno, que esta constituido por otros componentes de diferentes tipos. En el modelo que se propone cada puerto debe ser un punto de interacción simple de entrada o de salida, que expone un servicio, un punto de conexión a un servicio o una entrada o salida de datos.

- Un puerto de salida puede utilizarse para:
    - Pedir un servicio.
    - Proporcionar datos.
- Un puerto de entrada puede utilizarse para:
    - Proporcionar un servicio.
    - Recibir datos.

Los puertos de salida asociados a la petición de un servicio se caracterizan por:

- Su semántica, que puede ser:
    - ✓ Petición/respuesta síncrona[74]
    - ✓ Petición/respuesta asíncrona,
    - ✓ Petición/sin respuesta asíncrona.
    - ✓ Subscripción a eventos o flujos de datos.
- La signatura del servicio que solicitan.
- Si la semántica del puerto es *petición/respuesta asíncrona*, el puerto o los puertos de entrada por los que se reciben los resultados del servicio.
- Si la semántica del puerto es *subscripción a eventos o flujos de datos*: el puerto de entrada por el que se reciben los datos o eventos, que debe ser compatible con el patrón de llegada de los mismos.

Los puertos de entrada asociados al suministro de un servicio se caracterizan por:

- Su semántica, que puede ser:
    - ✓ Petición/respuesta síncrona[75]
    - ✓ Petición/respuesta asíncrona,
    - ✓ Petición/sin respuesta asíncrona.
    - ✓ Subscripción a eventos o flujos de datos.
- La signatura del servicio que proporcionan.
- Si la semántica del puerto es *petición/respuesta asíncrona*, el puerto o los puertos de salida por los que se proporcionan los resultados del servicio.
- Si la semántica del puerto es *subscripción a eventos o flujos de datos*: el puerto de salida por el que se suministran los datos o eventos. Los eventos o datos pueden proporcionarse:
    - Cuando se produce un evento (por ejemplo cambia el valor de un dato).

[74] No es una semántica en las peticiones de servicio entre componentes del GOYA, pero no puede descartarse por completo.
[75] No es una semántica en las peticiones de servicio entre componentes del GOYA, pero no puede descartarse por completo.

        - De forma continua, en este caso pueden ser flujos de datos temporizados (las tramas incluyen información temporal) o envío periódico de datos, donde el período es un parámetro del servicio.
    - Su cardinalidad.

Obsérvese que aunque los servicios se requieren o proporcionan a través de un puerto, pueden implicar la conexión de otros puertos adicionales entre cliente y servidor. El modelo debe proporcionar mecanismos para conectar todos los puertos asociados a un servicio y para comprobar que las conexiones entre puertos son consistentes.

Independientemente del propósito de la interfaz, está puede ofrecer o solicitar servicios a través de puertos con diferentes semánticas. Las posibles formas de interacción vienen en buena medida determinadas por las semánticas que podamos asociar a los puertos. Por ello, el modelo de componentes debe proporcionar la posibilidad de configurar o definir la semántica de los mismos[76]. En una misma interfaz pueden existir puertos que ofrezcan o requieran los mismos o distintos servicios con la misma o con diferente semánticas. Las reglas de conexión de los puertos deben ser sencillas y no ir más allá de reglas como las siguientes:

- Los puertos de salida de un componente (servicios provistos) se conectan con los puertos de entrada de otro (servicios requeridos).
- Los tipos de datos de los puertos de entrada y salida deben ser compatibles.
- Los patrones de interacción de los puertos de entrada y salida deben ser complementarios.

Cualquier otra información concerniente al protocolo de petición de servicios o las condiciones que deben cumplir los clientes para poder solicitar un determinado servicio deben incluirse en los conectores.

Restringir aún más el conocimiento que un componente tiene de sus patrones de interacción, por ejemplo ocultándole la semántica de sus puertos podría ser contraproducente. La semántica de los puertos de un componente está estrechamente relacionada con su funcionalidad básica y con el contrato que se compromete a cumplir, tanto si es cliente como si es servidor. En los contratos puede especificarse no sólo el servicio que se proporciona, sino cómo se proporciona. Y viceversa, no sólo el servicio que se requiere, sino cómo se requiere. Aunque parte del contrato puede embeberse en los conectores, sustraer a un componente el conocimiento de la semántica de sus interfaces puede ser en algunos casos ir demasiado lejos. La parte del contrato que debe incluirse en los conectores es aquella más susceptible de ser cambiada. Además, tan malo es incluir en el componente la descripción de los patrones de interacción como incluir en el patrón de interacción información que es propia del componente. Es malo no tanto porque estamos asignando al conector responsabilidades que no son suyas, sino sobre todo porque estamos extrayendo del componente información y responsabilidades que lo definen. No obstante, si la semántica del puerto no es parte inherente del contrato del componente sería conveniente separarla, aunque esta capacidad no se impone como requisito del modelo de componentes.

[76] Recuérdese que en la sección anterior se establecía la necesidad de que el modelo de componentes tuviera asociado o pudiera asociarse con un lenguaje formal. La definición de una semántica para los puertos requiere de la existencia de este tipo de formalismos.

Interfaces de un objeto: Conjunto de métodos.

El CBD es deudor en muchos sentidos de la programación orientada a objetos, de forma que cuando se habla de interfaz o interfaces de un componente se tiende a pensar en las mismas como un conjunto de métodos. Es más, se tiende a pensar en los componentes como si fueran objetos con algunas características adicionales. Sin embargo, los patrones de interacción entre objetos no son, en general, nada apropiados para los componentes, o al menos no lo son para los componentes del GOYA.

El patrón de interacción típico de las relaciones entre objetos es la invocación de métodos que responde a una semántica petición/respuesta síncrona con bloqueo del cliente. De hecho, no es posible definir de forma directa llamadas asíncronas entre objetos. Aunque se proclame que la comunicación entre objetos es el envío de mensajes, en la práctica no es más que una llamada a procedimiento como las que se realizan en los lenguajes imperativos.

Las llamadas a procedimiento no implican la existencia de puertos de entrada y salida tan habituales en los ADLs, porque no están pensados para comunicar dos entidades diferentes, sino para implementar una abstracción funcional que implica el salto del puntero del programa de una posición de memoria a otra y el retorno a la posición donde se realizó la llamada. Es decir, responde a la necesidad de organizar el programa en funciones, no la necesidad de comunicar entidades independientes.

La invocación de métodos es más de lo mismo. Se incrementa la posibilidad de llamar a diferentes métodos a través de invocaciones en las que el receptor del mensaje puede diferirse hasta el tiempo de ejecución. Pero en realidad, aunque equiparar las invocaciones a métodos con el envío de mensajes es una buena metáfora para entender el polimorfismo, sigue siendo más de lo mismo, un salto de una posición de memoria a otra, en la que la dirección de destino se obtiene de forma dinámica.

Por ello, no es de extrañar que con los lenguajes OO disponibles hasta el momento:

- ✓ *No sea posible definir explícitamente las interfaces que requiere un objeto. ¿Para qué?, si lo que se pretende es extender la abstracción funcional con la capacidad del polimorfismo y no comunicar entidades realmente independientes, el mecanismo request_reply síncrono es más que suficiente.*
- ✓ *No puedan obtenerse referencias a servicios concretos, sino a interfaces enteras. Nuevamente, esto es así porque las interfaces, a pesar de su nombre, no están relacionadas con la oferta o petición de servicios entre entidades independientes, sino con el concepto de tipo abstracto de datos.*

  Un tipo abstracto de datos oculta la representación de la información y proporciona un conjunto de operaciones que actúan sobre dichos datos. Obviamente cuando se maneja una variable de un cierto tipo, por lo general se está interesado en realizar todas las operaciones posibles sobre la misma

Figura 8.3: Interfaces de los objetos.

8.5.5 Interacciones entre componentes. Responsabilidades de los conectores.

Los patrones de interacción arriba descritos son muy sencillos y suelen estar directamente soportados por cualquier modelo de componentes. El patrón observador es algo más complejo, pero en general los modelos de componentes y los entornos de programación proporcionan los medios para realizarlo de forma bastante directa. Los patrones de interacción *que conocen* los componentes no deben ir más allá de este pequeño conjunto de interacciones básicas. El resto debe ser manejado por los conectores.

A menudo, los componentes tienen que interaccionar de formas complejas, de acuerdo con protocolos más o menos complicados en los que pueden intervenir más de dos componentes. En este caso, ***es necesario separar la funcionalidad básica del componente de sus patrones de interacción***. Los conectores deben gestionar los protocolos de interacción entre componentes, definiendo sus reglas y los roles que cada componente debe desempeñar en la interacción. Los cambios de protocolo son relativamente frecuentes (especialmente si hay que integrar componentes desarrollados por terceros). Si se incluyen los detalles de los protocolos de interacción dentro de los componentes cualquier cambio en los protocolos supone un cambio en los componentes. Si se separan, es posible definir colecciones de patrones de interacción intercambiables sin que los componentes se vean afectados. Tal y como se dice en [Shaw et al 1996]:

Los conectores median en las interacciones entre componentes, estableciendo las reglas que gobiernan la interacción entre componentes y proporcionando mecanismos auxiliares.

Si los protocolos de interacción se embeben en los conectores es posible, por un lado, reutilizar los mismos componentes en arquitecturas muy diferentes y, por otro, adaptar los patrones de interacción a las necesidades del sistema, tanto de forma estática como dinámica. Los servicios auxiliares a los que se refiere la definición de Shaw pueden ser filtros de distintos tipos, que realicen conversiones de los tipos de datos de los puertos o de las signaturas de los servicios (adaptadores), bufferes para acompasar las velocidades relativas de clientes y servidores, servicios de encriptado o desencriptado de la información o de compresión y descompresión de los datos, etc. Las interacciones pueden variar:

- ✓ Durante los procesos de desarrollo y mantenimiento.
- ✓ Durante el funcionamiento del sistema.
- ✓ Entre sistemas de la línea.

Además de gestionar la lógica del protocolo entre componentes, los conectores pueden manejar los aspectos relacionados con su distribución. Puesto que el despliegue de los componentes puede variar de unos sistemas a otros o incluso dentro de un sistema dado, es necesario que los componentes no necesiten conocer su ubicación ni la ubicación de los componentes con los que interaccionan.

Queda por determinar la manera en que los conectores deben conectar los puertos de los componentes. Para ello habría que responder primero como se conectan ellos mismos a los componentes. Puesto que la única forma de conectarse a un componente es a través de sus puertos, los conectores deben tener a su vez puertos que sean compatibles con los de los componentes. Es inmediato inferir que los conectores son a su vez un tipo de componente que ofrece los mismos tipos de puertos que cualquier otro y cuya responsabilidad específica es

gestionar protocolos y reglas de negocio. Pero es un componente especial, ya que desde el punto de vista de los componentes que enlaza no existe. Si los componentes percibieran la existencia de un mediador dependerían de alguna forma de él.

Para terminar, el conector no sólo añade nuevas condiciones al contrato, sino que además debe asegurar la parte del contrato entre componentes que es inherente a los mismos. En realidad se trata de aplicar a las interacciones un principio ya expresado por Meyer y Liskov:

Las precondiciones de la interacción deben ser más débiles que las de los componentes que enlaza y las poscondiciones más fuertes.

Por supuesto estás precondiciones y poscondiciones se refieren a las definidas en el interior de los componentes, no a las que añadan los protocolos y reglas de negocio embebidas en el conector.

9 Conclusiones y Trabajos Futuros

9.1 Conclusiones y Aportaciones de esta Tesis

Los principales objetivos a la hora de desarrollar software son mejorar el tiempo de vida y la calidad del producto y facilitar su mantenimiento. Una arquitectura de referencia describe la infraestructura común a todos los sistemas del dominio de aplicaciones para los que se define y determina tanto sus componentes como los patrones de interacción entre los mismos. Una vez definida, la arquitectura de referencia puede ser ejemplarizada para crear arquitecturas de sistemas específicos. Disponer de tal arquitectura facilita enormemente el desarrollo de nuevas aplicaciones dentro del dominio considerado, pues permite la reutilización de modelos y componentes. Sin embargo, el diseño de una arquitectura de referencia es un trabajo muy arduo que requiere un conocimiento exhaustivo del dominio e implica modelar tanto los aspectos comunes a todos los sistemas como las posibles fuentes de variabilidad. Dicha variabilidad viene dada tanto por las diferencias entre sistemas como por la evolución de los mismos a lo largo del tiempo. Así, una de las principales características de una arquitectura de referencia debe ser su flexibilidad.

Pero por muy flexible que sea una arquitectura software, existe un límite en su capacidad de adaptación frente a nuevos requisitos. La arquitectura que se ha evaluado en este trabajo de tesis no constituye una excepción a este respecto. Diferentes sistemas exigen diferentes arquitecturas. Como se ha explicado en los capítulos anteriores, la arquitectura de un sistema puede verse como el resultado de una serie de compromisos de diseño en los que se opta por primar el cumplimiento de ciertos requisitos en detrimento de otros. Por muchas similitudes que compartan dos sistemas, si difieren demasiado en sus requisitos, los compromisos de diseño tendrán que ser diferentes y darán lugar a arquitecturas diferentes. Y, sin embargo, las similitudes siguen ahí y con ellas la posibilidad de utilizar componentes comunes.

Las principales conclusiones que pueden extraerse de la evaluación de la arquitectura propuesta en [Alvarez 1997] no son los resultados de la evaluación en sí. Dicha evaluación arroja resultados aceptables cuando se la enfrenta con sus requisitos originales y negativos cuando se la evalúa frente a los nuevos requisitos.

Las principales conclusiones de dicha evaluación son:

4. No es realista plantear el diseño de arquitecturas de referencia para dominios excesivamente amplios, en los que los diferentes sistemas demandan distintos, y a veces incompatibles, compromisos de diseño.
5. Aunque la arquitectura software define tanto los componentes del sistema como sus interacciones, en el caso de los sistemas de teleoperación, la mayor parte de los compromisos de diseño están relacionados con los patrones de interacción entre componentes.

Una de las características fundamentales de las arquitecturas de referencia es que deben poder utilizarse para generar arquitecturas específicas para sistemas concretos y proporcionar un marco en el que sea posible definir componentes reutilizables. Para ello, las arquitecturas de referencia deben:

1. Ser lo suficientemente estables. La arquitectura debe reducir el espacio de soluciones, definiendo una serie de restricciones o reglas de diseño que son óptimas para el dominio considerado. Una arquitectura en continua evolución, en la que dichas restricciones y reglas cambian constantemente no permite la definición de componentes reutilizables.
2. Proporcionar un conjunto de mecanismos de configuración y extensión que permitan producir arquitecturas para sistemas específicos.

Por ello, el aspecto más crucial a la hora de definir arquitecturas de referencia es determinar que partes de la misma deben ser estables y cuáles deben ser variables, asociando a estas últimas mecanismos de configuración o extensión. En el caso de los sistemas de teleoperación estudiados en esta tesis la fuente de variabilidad más importante viene dada por los patrones de interacción. Con estos antecedentes, es inmediato deducir que:

4. Deben adoptarse unas estrategias de diseño e implementación que permitan separar, por un lado, los patrones de interacción entre componentes de su funcionalidad y, por otro, los diferentes *aspectos* de dicha funcionalidad.
5. Es mucho más práctico definir o adoptar un marco de desarrollo en el que puedan definirse diferentes arquitecturas en las que se reutilicen componentes comunes que intentar desarrollar arquitecturas que deban satisfacer demasiados requisitos contradictorios.
6. Es necesario, o bien reducir la amplitud del dominio, considerando sólo aquellos sistemas cuyos requisitos permitan adoptar los mismos compromisos de diseño, o bien definir mecanismos de configuración que permitan diferir las decisiones de diseño hasta el momento en que deba definirse la arquitectura de un sistema específico.

La separación de los patrones de interacción entre componentes de la funcionalidad de dichos componentes exige la consideración de los conectores como entidades de primera clase, al mismo nivel que los propios componentes. De esta manera, como han demostrado diferentes autores, pueden definirse estrategias de interacción intercambiables que embeban los roles que cada componente debe desempeñar en la arquitectura y las reglas de interacción más susceptibles de ser cambiadas. Aplicando los patrones de reflexión y reificación dichos conectores pueden configurarse incluso en tiempo de ejecución.

La construcción de software a partir de componentes ya existentes depende de su existencia y de su capacidad para combinarse. Pero dicha capacidad es más fácil de proclamar que de conseguir, pues exige que una compatibilidad total no sólo entre las interfaces que determinan

las relaciones entre componentes, en términos de los servicios que requieren y solicitan, sino también entre las interfaces que determinan las relaciones de dichos componentes con la infraestructura que los soporta. En un modelo puramente abstracto basta con que los componentes tengan interfaces compatibles para que dicha combinación sea posible. Sin embargo, la reutilización que se persigue es la reutilización de implementaciones y los componentes que deben combinarse son unidades de despliegue que se ejecutan sobre una determinada infraestructura. Es decir, los *contratos* de los componentes no involucran sólo a los componentes, sino también a la infraestructura que los soporta. La capacidad de combinar componentes y por consiguiente la posibilidad de reutilizarlos en diferentes sistemas requiere de forma ineludible que dichos componentes sean conformes a un mismo modelo de componentes o que pertenezcan a modelos de componentes compatibles.

Un enfoque de desarrollo basado en componentes (CBD) tiene en cuenta la mayor parte de las consideraciones realizadas en los párrafos precedentes. Su mayor inconveniente es su falta de madurez. Actualmente existen diversos modelos de componentes cuya compatibilidad es más que discutible y que se encuentran aún en plena evolución. Ni siquiera el propio concepto de componente está claramente establecido, existiendo, como se ha visto diferentes escuelas e interpretaciones Los lenguajes de programación no incorporan mecanismos que soporten los conceptos del CBD, y como se ha comentado, su capacidad para definir interfaces es muy limitada. Aun así el CBD ofrece un marco de desarrollo muy atractivo que se adapta perfectamente a los propósitos de reutilización del software que se persiguen.

La principal aportación de esta tesis es la caracterización precisa de los atributos de calidad de un subdominio muy amplio de los sistemas de teleoperación. Dicho subdominio engloba la mayor parte de los dispositivos teleoperados potencialmente utilizables en aplicaciones industriales de mantenimiento de instalaciones y equipos.

Las plantillas de atributo definidas en el capítulo 5 y en el anexo II constituyen una especie de prontuario que puede utilizarse para la generación de requisitos de sistemas concretos y una base de conocimiento aplicable al desarrollo de nuevos sistemas. Dichas plantillas deberán refinarse y formalizarse en la medida de lo posible para facilitar su empleo en el proceso de desarrollo y su mantenimiento necesitará sin duda el empleo de herramientas, pero incluso en su forma actual son una herramienta útil para el diseñador.

Otras aportaciones adicionales son:

- La evaluación de la arquitectura mediante el empleo del método ATAM. Aunque existen ejemplos de utilización de ATAM, ninguno de ellos considera un conjunto de atributos de calidad tan grande. La mayor dificultad para aplicar el método ha sido, como se ha explicado en el capítulo 7, la imposibilidad de priorizar los escenarios, dada la amplitud del dominio. Sin embargo, dichos escenarios están ahí y pueden ser reutilizados cuando convenga para la evaluación de las arquitecturas y sistemas que se vayan desarrollando.
- La propuesta, siguiendo las líneas definidas por el ABD, de una nueva descomposición funcional de los sistemas del dominio, que completa la definida en [Alvarez 1997] y resuelve (así lo cree el autor de esta tesis) los problemas detectados durante la evaluación.
- La definición de los requisitos que debe cumplir un modelo de componentes aplicable al desarrollo de aplicaciones en el dominio caracterizado y, finalmente,
- Las propias conclusiones de la evaluación, ya comentadas más arriba, y el enfoque que se propone para los nuevos desarrollos basado en las líneas de producto y en el desarrollo basado en componentes.

9.2 Trabajos Futuros

Las actividades futuras derivables de esta tesis son:

- A corto plazo:
  - ✓ La aplicación de la descomposición funcional propuesta en el capítulo 8 y de los requisitos definidos para el modelo de componentes en el desarrollo de un nuevo prototipo del sistema GOYA (véanse capítulos 7 y 8).
  - ✓ La formalización, cuando sea posible, de los requisitos descritos en las plantillas de atributo definidas en el capítulo 5 y en el anexo II y su transformación para que puedan ser mantenidas con ayuda de herramientas.
  - ✓ La definición de los patrones de interacción entre los componentes del sistema GOYA y el refinamiento de algunos de los subsistemas propuestos, en especial el de monitorización y diagnóstico, el de configuración y el Gestor de Aplicación. Aunque dichos trabajos, como se ha explicado en el capítulo 8, pertenecen a la ingeniería de aplicación, sus resultados pueden generalizarse para dar lugar a componentes o patrones de interacción reutilizables.

- A medio y largo plazo:
  - ✓ Estudio de patrones de diseño que permitan definir patrones de interacción intercambiables.
  - ✓ Refinamiento de los requisitos propuestos para el modelo de componentes y adopción o definición de un modelo de componentes conforme con dichos requisitos. Habrá que estudiar la adecuación de los modelos de componentes, tanto generales, como específicos del dominio, a los requisitos que se proponen.
  - ✓ Estudio y selección de notaciones formales que permitan describir sin ambigüedades las propiedades de los componentes y de sus patrones de interacción y que permitan asimismo deducir las propiedades del sistema a partir de las propiedades de sus componentes.
  - ✓ Estudio de técnicas que permitan añadir y eliminar funcionalidad e interfaces a los componentes en la línea descrita por [Andrade et al 1999] y [Bosch 2000].

Estas actividades no son ortogonales, existiendo muchos puntos en común entre todas ellas. Sin embargo cada una pone el énfasis en un aspecto distinto y todas ellas tienen entidad suficiente por sí solas de definir una línea de investigación. Finalmente, sólo cabe apuntar que el tema de la tesis es lo suficientemente amplio como para suscitar un gran número de trabajos futuros, los que se citan aquí responden, por un lado a las necesidades más inmediatas del grupo de investigación en el que el autor desempeña su labor y, por otro, a sus preferencias personales.

Bibliografía

[AAA 1995] PROYECTO APLICACIONES AUTOMÁTICAS PARA LA REDUCCIÓN DE DOSIS, Programa de Investigación Electrotécnico del MINER (PIE-041049). 1995

[AAA 1996a] "Aplicaciones Automáticas para la Reducción de Dosis: Plan de Calidad", PC-AAA, Endesa, AN Ascó, AN Vandellós, CN Almaraz, CN Cofrentes, Dtn, ENWESA, PIE-041049, 1996

[AAA 1996b] "Aplicaciones Automáticas para la Reducción de Dosis: Especificación Técnica del Software", ET-AAA, Endesa, AN Ascó, AN Vandellós, CN Almaraz, CN Cofrentes, Dtn, ENWESA, PIE-041049, 1996

[Aarsten et al 1996] A. Aarsten et at, "Designing Concurrent and Distributed Control Systems", Communications of the ACM, October-1996

[Abowd et al 1996] G. Abowd, R. Kazman, "Analyzing Differences Between Internet Information System Software Architectures", Proceedings of the IEEE ICC'96, págs. 203-207, Dallas (TX), June 96

[Albus 1987] J.S.Albus, "A Control System Architecture for Multiple Autonomous Vehicles", Proceedings of the 5th International Symposium of Unmmaned, June 1987.

[Albus et al 1987] Albus J.S., Lumia, H. McClain, "NASREM: Standard Reference Model for Telerobot Control", Technical Report, NASA Conference on Space Applications of AI Robotics, 1987.

[Albus et al 1990] J.S. Albus, R. Quintero, "Concepts for a Reference Model for Real Time Intelligent Control Systems", Technical Report, National Institute Standards and Technology, 1990.

[Alonso et al 1997] A. Alonso, B. Alvarez, J.A. Pastor, J.A. de la Puente, "Software Architecture for a Robot Teleoperation System", 4th IFAC Workshop on Algorithems and Architectures for Real Time Control, April 1997, ISBN 0-08-042930-0

[Alvarez 1997] B. Alvarez, "Arquitectura Software de Referencia para Sistemas de Teleoperación", Tesis Doctoral 1997, Departamento de Ingeniería de Sistemas Telemáticos, UPM.

[Alvarez et al 2000a] B. Alvarez, A. Iborra, A. Alonso, J.A. de la Puente, J.A. Pastor, "Developing Multi-Application remote systems", Nuclear Engineering International, March 2000, pág. 24-28, ISSN 0029-5507, vol 45, no.548 (SCI)

[Alvarez et al 1996] B. Alvarez, J.A Pastor, S. Organai, "Arquitecturas software para robots de mantenimiento en Centrales Nucleares", XXII Reunión de la Sociedad Nuclear Española, Santander 1996.

[Alvarez et al 1998] B. Alvarez, A. Alonso, J.A. de la Puente, "Timing Analysis of a Generic Robot Teleoperation Software Architecture", Control Engineering Practice, ISSN 0967-0661 (SCI), vol 6, no.6, pp 409-416, June 98

[Alvarez et al 2000b] B. Alvarez, F. Marín, J.A. Pastor, A. Alonso, J.A. de la Puente, "Arquitectura Software Genérica para Teleoperación: Aplicación a Robots de Servicios en Centrales Nucleares", Automática e Instrumentación, Nov-2000, ISSN 0213-3113, no.312

[Alvarez et al 2000c] B. Alvarez, A. Iborra, A. Alonso, J.A. de la Puente, "Reference Architecture for Robot Teleoperation: Development Details and Practical Use", Control Engineering Practice, ISSN 0967-0661 (SCI)

[Andrade et al 1999] L.F. Andrade, J.L. Fiadeiro, "Interconnecting Objects via Contracts", UML'99 - Beyond the Standard, R. France and B. Rumpe (eds), LNCS 1723, Springer Verlag 1999, 566-583

[Andrade et al 2001] L.F. Andrade, J.L. Fiadeiro, "Coordination Technologies for Managing Information System Evolution", CaiSE 2001, LNCS 2068, pp. 374-387, 2001, Springer-Verlag, Berlín.

[Arango 1989] G. Arango, "Domain Analysis: From Art to Engeeniering Discipline", In Proceedings 5th International WorkShop Specification and Design. IEEE Computer Society Press, May 89

[Arkin et al 1992] R.C. Arkin, "Aura: A Hibrib Reactive/Hierarchical Robot Architecture", Workshop on Architectures for Intelligent Control Systems", May 1992

[Bachmann et al 2000] F. Bachmann et al, "Volume II: technical Concepts of Component Based Software Engineering", Technical Report CMU/SEI-2000-TR-008, 2000

[Bachmann et al 2000a] F. Bachmann et al, "Software Architecture Documentation in Practice: Documenting Architectural Layers", Special Report CMU/SEI-2000-SR-004, March 2000

[Bachmann et al 2000b] F. Bachmann et al, "The Architecture Based Design Method", Technical Report CMU/SEI-2000-001, January 2000

[Bachmann et al 2000c] F. Bachmann, Len Bass, Mark Klein, "An Application of the Architecture-Based Design Method to the Electronic House", Special Report, CMU/SEI-2000-SR-009, September 2000

[Bachmann et al 2001] F. Bachmann, L. Bass, "Managing Variability in Software Architectures", Proceedings of the SSR'2001, 126-132, Toronto (Canada), ACM Press.

[Barbacci 99] M. Barbacci, "Analyzing Quality Attributes", news@sei interactive 2, I (March 1999)

[Barbacci et al 1995] M. Barbacci, M. Klein, T. Longstaff, "Quality Attributes", CMU/SEI-95-TR-021, 1995

[Barbacci et al 1996] M. Barbacci, M. Klein, C. Weinstock, "*Principles for Evaluating the Quality Attributes of Software Architecture*", CMU/SEI-96-TR-036, 1996

[Barbacci et al 1997] M. Barbacci, S.J. Carriere, R. Kazman, M. Klein, H. Lipson, T. Longstaff, C. Weinstock, "*Steps in an Architecture Tradeoff Analysis Method: Quality Attribute Models and Analysis*", CMU/SEI-97-TR-029, 1997

[Barkmeyer et al 88] E.J. Barkmeyer, "Distributed Data Interfaces- The Lessons of Imdas", Proceedings of CIM Databases'88 Conference, March 1988

[Bass et al 1998] Len Bass, Paul Clements and Rick Kazman.*"Software Architecture in Practice". Addison- Wesley 1998, ISBN 0-201-19930-0*

[Bass et al 1999] L. Bass, R. Kazman, "Architecture Based Development", Technical Report CMU/SEI-99-TR-007, April 99

[Bass et al 2000] Len Bass, M. Klein, F. Bachmann, "Quality Attribute Design Primitives", Technical Report, CMU/SEI-2000-TR-017

[Bass et al 2001a] L. Bass et al, "Quality Attribute Design Primitives and the Attribute Driven Design Method", Proceedings of the PFE 2001, Bilbao (Spain)

[Bass et al 2001b] L. Bass et al, "Applicability of General Scenarios to the Architecture Tradeoff Analysis Method", Technical Report, CMU/SEI-2001-TR-014

[Boe78] B.W. Boehm et al, "Characteristics of Software Quality", New York: American Elsevier, 1978

[Boehm 1988] B. W. Boehm, "A Spiral Model of Software Development and Enhancemnet",IEEE-Computer, May-88, v 21, no 5, pp 61-72

[Boehm 1989] B. Boehm, R. Ross, "Theory W Software Project Management: Principles and Examples", IEEE Transac. Software Engineering, July 1989, pp 902-916

[Boehm 1995] B. W. Boehm, P.Bose, E. Horowitz, M.Lee, "Software Requirements Negotiation and Renegotiation Aids: A Theory-W Based Spiral Approach", ICSE-17, IEEE Computer Society Press, Seattle, April 1995

[Boehm 1996] B. W. Boehm, Hoh in, "Identifying Quality Requirements Conflicts", *IEEE Software March 1996*

[Boehm 1998a] *B. W. Boehm, A. Egyed, "Software Requirements Negotiation: Some Lessons Learned", IEEE Software 1998*

[Boehm 1998b] B. Boehm et al, "Using the Win-Win Spiral Model: A Case Study", Computer, July 98, pp 33-34

[Boehm 1999a]. *B. Bohem, Dan Port. "When Models Collide: Lessons from Software System Analysis", ". IEEE Software (IT-Pro), January 1999*

[Boehm 1999b] B. Boehm, C. Abts, "COTS Integration: Plug and Pray?", IEEE Computer, January 1999

[Boissiere 1989] B.T. Boissiere, R.W. Harrigan, "An Alternative Approach to Telerobotics: Using a World Model and Sensor Feedback". 3rd Topical Meeting on Robotics and Remote Systems, 1989.

[Booch 1994] G. Booch, "Object Oriented Analysis and Design with Applications", Redwood City, CA, Benjamin/Cummings, 1994

[Booch 1999] I. Jacobson, G. Booch, J. Rumbaugh, "The Unified Software development Process". *Addison-Wesley 1999, ISBN 0-201-57169-2*

[Bosch 1999] Jan Bosch, "Superimposition: A component adaptation technique", Information and Software Technology 1999.

[Bosch 2000] Jan Bosch, "Product Line Architectures", ObjectiveView, Object Component Architecture Series, (4):13-18, http://www.ratio.co.uk

[Brad 2000] Brad Appleton, http://www.enteract.com/~bradapp/docs/patterns-intro.html

[Bridge 2000] Project Technology Bridge, http://www.projtech.com

[Brooks 1986] R.A. Brooks. "A Robust Layered Control System for a Mobile Robot", April 1986

[Brown 1999] A. Brown, "Building Systems from Pieces with Component-Based Software Engineering", MacMillan Technical Publishing, ISBN 1578701473, Sterling Software, white paper.

[Brownsword et al 1996] L. Brownsword, P. Clements, *"A Case Study in Successful Product Line Development"*, Technical Report, CMU/SEI-1996-TR-016, 1996.

[Bruin et al 2002] H. Bruin et al, "Quality Driven Software Architecture Composition", Proceedings of the 9th IEEE CBSE ECBS'2002, Lund (Sweden)

[Bruyninckx et al 2002] H. Bruyninckx et al, "A software framework for Advanced Motion Control", Red temática OROCOS, http://www.mec.kuleuven.ac.be/~bruyninc

[Buschmann et al 1996] F. Buschmann, R. Meunier et al "Pattern Oriented Software Architecture Vol1: A system of patterns: Pattern languages of program design", John Wiley & Son Ltd, 1996

[C4ISR 1997] DoD Architectures Working Group (1997). C4ISR Architecture Framework, Version 2.

[CASE 2000] http://www.qucis.queensu.ca/Software-Engineering/tools.html

[CCM 1999] OMG, "Corba Components - Volume I", http://www.omg.org/cgi-bin/doc?orbos/99-07-01, 1999

[Chen 1976] P.P. Chen. "The entity-relationship model toward a unified view of data", ACM Transactions on DataBase Systems, 1(1), March 1976

[Clements 1996b] P. Clemens, "A Survey of Architecture Description Languages", 8th International Workshop on Software Specification and Design, Germany, March 1996

[Clements et al 1998] P. Clements et al, " A Framework for Software Product Line Practice - Version 1.0", Product Line System Program, SEI-CMU.

[Clemments 1996a] P. Clemens et al, "Software Architecture: An Executive Overview", Technical Report, CMU/SEI-96-TR-003, February 1996.

[Coad 1991] P. Coad, E. Yourdon, E. Yourdan, "Object Oriented Design", Yourdon Press Computing Series, Prentice Hall, 1991,

[Coad 1992] P. Coad, "Object Oriented Patterns", Communications of the ACM, 35 (9). 1992

[COCOTS 2000] *"Constructive COTS Integration (COCOTS)"*, http://sunset.usc.edu.COCOTS/cocots.html.

[Coleman et al 1993] D. Coleman, B. Bodoff, "Object Oriented Development: The Fusion Metod", Prentice Hall, 1993

[Conway 1968] M.E. Conway, "HowDo Committees invent?" Datamation, vol 14, No 4, Apr. 1968, pp 28-31

[Coplien 1999] J.O. Coplien, *"Reevaluating the Architectural Metaphor: Toward Piecemeal Growth"*, IEEE Software, Sept/Oct 99

[Coplien et al 1995] J.O. Coplien, D.C. Smith, "Pattern Languages of Program Design", Addison Wesley 1995

[Coste et al, 2000] E. Coste-Manière et al, "Architecture, the Backbone of Robotic Systems", Proceedings of the 2000 IEEE International Conference on Robotics and Automation, San Francisco, CA, April 1999, págs 67-72

[Dornier 1991] G. Dornier, "Baseline a&r Control Development Methodology Definition Report", Technical Report, European Space Agency Contract Report, 1991.

[Drake et al 2000]] J. M. Drake, M. González Harbour, J.S. Medina, "Mast Real Time View: Graphic UML Tool for Modeling Object Oriented Real Time Systems", Grupo de Computación y Sistemas de Tiempo Real de la Universidad de Cantabria. Internal Report 2000

[D'Souza et al 1999] D. F. D'Souza et al, "Objects, Components and Frameworks with UML. The Catalysis Approach", Object Technology Series, Addison Wesley.

[Egyed 2000] A. Egyed, "Using Patterns to Integrate UML Views", University of Southern California, Technical Report USC-CSE-1999-515

[Egyed et al, 1999] A. Egyed, N. Medvidovic, "Extending Architectural representation in UML with View Integration", Proceedings of the 2nd International Conference on the Unified Modeling Language (UML), Fort Collins, CO, October 1999, págs 2-16

[Egyed et al, 2000] A. Egyed et al, "Refinement and Evolution Issues in Bridging Requirements and Architectures", University of Southern California, Technical Report USC-CSE-2000-515

[Eickelmann et al 1996] N. S. Eickelmann, D. J: Richardson, "An Evaluatin of Software Test Environment Architectures,", Proceedings of ICSE-18, págs 353-363, IEEE-1996

[Ellis et al 1996] Ellis W.J. et al, "Toward a Recommended Practice for Architectural Description", Proceedings of the 2nd IEEE International Conference on Engineering of Complex Components Systems, Montreal, Quebec, Canada, Oct-96

[Fayad et al 1997] M. Fayad, D.C. Schmidt, "Object-Oriented Application Frameworks", Communications of the ACM, Vol. 40, No. 10, October 1997

[Fong 2001] T. Fong et al, "A Safeguarded Teleoperation Controller", IEEE International Conference on Advanced Robotics 2001, August 2001, Budapest, Hungary.

[Gallagher 2000] Brian P. Gallagher, "Using the Tradeoff Analysis Method to Evaluate a Reference Architecture. A Case Study", Technical Note, CMU/SEI-2000-TN-007

[Gamma et al 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *"Design Patterns: Elements of Reusable Object Oriented Software"*, Addison Wesley, Reading Mass. 1995

[Gannod et al 2000] G.C. Gannod, R.R. Lutz, "An approach to Architectural Analysis of Product Lines", Proceed. of the IEEE 2000 International Conference on Software Engineering, 548-557

[Garlan et al 1994] D. Garlan, D. Perry, "Software Architecture: Practice, Potentials and Pitfalls", Proceed. of the 16th Conference on Software Engineering IEEE ICSE-16, 1994, 149-153

[Garlan et al 1995a] D. Garlan, R. Allen, J. Ockerbloom. "Architectural Mismatch: Why Reuse is so hard", *IEEE Software Nov-1995*

[Garlan et al 1995b] D. Garlan and D. Perry. "Introduction to the special issue of software architecture". *IEEE*

Transactions on Software Engineering, 21(4), April 1995

[Gomaa 2000] Hasan Gomaa, "Designing Concurrent, Distributed and Real Time Applications with UML", Addison-Wesley 2000, ISBN 0-201-65793-7

[GOYA 1998a] Proyecto FEDER (UPCT, Izar Carenas) nro 1FEDER IFD97 - 0823 (TAP)

Robot Escalador para la limpieza de cascos de buques, respetuoso con el medio ambiente.

[GOYA 1998b] "Especificación de Requisitos del Sistema Goya", Documento Interno, Grupo de Investigación DISIE de la UPCT.

[Griesmeyer 1992] M.J. Griesmeyer, "Generic Intelligent System Control (GISC)", Technical Report, Sandia National Laboratories, 1992

[Griss 2000] M.L. Griss, "Implementing Product-Line Features by Composing Component Aspects", Proceedings of the First Software Product Line Conference, Denver (CO) USA, August 2000.

[Halstead 1977] Halstead, Maurice H. *Elements of Software Science, Operating, and Programming Systems Series* Volume 7. New York, NY: Elsevier, 1977.

[Harel 1990] D. Harel, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems.", IEEE Transactions on Software Engineering, 16(4), April 90

[Harel et al 1996] D. Harel, E. Gery, "Executable Object Modelling with StateCharts", IEEE Computer, vol 30, num 7, pages 31-42, July 1996.

[Hayward et al 1986] V. Hayward et al, "Robot Manipulator Control under Unix: A Robot Control C Library", International Journal of Robotics Research, 1986

[Iborra et al 2000] A. Iborra, B. Alvarez, P. Navarro, J.A. Pastor, J.M. Fdez. Meroño, "Robotized System for Retrieven Fallen Objects within the Reactor Vessel of a Nuclear Power Plant (PWR)", IEEE International Symposium of Industrial Electronics, ISIE'2000, Puebla (Mexico)

[Iborra et al 2002] A. Iborra, J.A. Pastor, B. Álvarez, C. Fernández , J.M. Fernández, " Operational Experiences using Robotics during Maintenance Services in PWR Nuclear Power Plants", IEEE Robotics & Automation Magazine, ISSN 1070-9932 (de próxima aparición, julio o agosto de 2002).

[IEEE-1061] IEEE Standards 1061-1992, Standards for a Software Quality Metrics Methodology, New York, NY, IEEE-1992

[IEEE-1471-2000] The **IEEE 1471-2000** Standard - Architecture Views and Viewpoints

[IEEE-610.12] IEEE Standard 610.12-1990, IEEE Standard Glossary of Software Engineering Terminology. New York, NY, IEEE-1990

[Ilogix 2000] i-logix, http://www.ilogix.com

[In 1999] *Hoh In, A.E. Flores-Mendoza, "Initial Design of the "Plug-n-Analyze" Framework for Architecture Tradeoff Analysis, IEEE-1999*

[Islam et al 1996] N. Islam, M. Deravakova, "An Essential Design Pattern for Fault Tolerant Distributed State Sharing", Communications of the ACM, October-1996

[ISO-9126] ISO Standard 9126, Information Technology-Software Product Evaluation, Quality Characteristics and Guidelines For Their Use, Geneva, Switzertland, ISO 1991

[Jacobson et al 1992] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, *"Object Oriented Software Engineering: A Use-Case Driven Approach"*, Reading, MA, Addison-Wesley, 1992

[Jacobson et al 1997] I. Jacobson et al, *"Software Reuse: Architecture, Process and Organization for Business Success"*, Addison-Wesley, New York, 1997

[Jacobson et al 1999] I. Jacobson, G. Booch, J. Rumbaugh, "The Unified Software Development Process", Addison-Wesley, 1999.

[Johnson et al 1990] T.L. Johnson, A.D. Barker, "Trends in shop floor control: Modularity, hierarchy and decentralization". 1990

[Jones 1994] Jones, Capers. "Software Metrics: Good, Bad, and Missing." *Computer 27, 9* (September 1994): 98-100.

[Jones et al 1986] A. T. Jones, C.R. McLean. "A proposed hierarchical control model for automated manufacturing systems", International Journal of Manufacturing Systems, 5 (1), 1986.

[Kai 2001] K. Kai, "Towards Architecture-oriented programming environments", Position Paper, Edmonton. August 24-25, 2001, Software systems Laboratory, Tampere University of Technology, Findland.

[Kang et al 1990] K.C. Kang, S. Cohen, "Feature Oriented Domain Analysis (FODA) Feasibility Study", Technical Report CMU/SEI, 1990

[Kapoor et al 1996] C. Kapoor et al, "A Reusable Operational Software Architecture for Advanced Robotics", PhD Dissertation, The University of Texas, Austin, 1996

[Kapoor et al 1998] C. Kapoor et al, "A Software Architecture for Multi-Criteria Decision Making for Advanced Robotics", Technical Article, University of Texas, NIST_IS98

[Kazman et al 1994] R. Kazman, L. Bass, G. Abowd, M. Webb, "SAAM: A Method for Analyzing the Properties of Software Architectures", Proceedings of 16th International Conference on Software Engineering, págs 81-90, IEEE-1994

[Kazman et al 1996] R. Kazman, G. Abowd, L. Bass, P. clements, "Scenario-Based Analysis of Software Architecture", IEEE Software Nov-96

[Kazman et al 1998a] *R. Kazman, M. Klein, M. Barbacci, T. Longdtaff, H. Lipson, J. Carriere, "The Architecture Tradeoff Analysis Method", 4th Conference of Engineering of Complex Computer Systems (IECCS98), Aug 98*

[Kazman et al 1998b] R. Kazman, M. Barbacci, M. Klein, S.J. Carriere, S. J. Woods "Experience with Performing Tradeoff Analysis Method", Proceed. of the IEEE International Conference on Software Engineering, 1999, 54-63

[Kazman et al 2000] R. Kazman, M. Klein, P. Clemens, "*ATAM SM*: Method for Architecture Evaluation", Technical Report, CMU/SEI-2000-TR-004, 2000

[Kiczales et al 1997] G. Kickzales et al, "Aspect Oriented Programming", Proceeedings of the European Conference on Object Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997

[Klein et al 1993] M. Klein, T. Rayla, B. Pollack, R. Obenza, M. Gonzalez-Harbour, "A practitioners Handbook for Real Time Analysis: Guide to Rate Monotonic Analysis for Real Time Systems", Kluwer Academic Publishers, 1993

[Klein et al 1999b] M. Klein, R. Kazman, *"Attribute- Based Architectural Styles",* Technical Report, CMU/SEI-2000-TR-022, ADA371802, 1999.

[Klein et al19 99a] M. Klein, R. Kazman, Len Bass, J. Carriere, M Barbacci, H. Lipson, *"Attribute- Based Architectural Styles"*, Proceedings of the 1st Working IFIP Conference on Software Architecture (WICSA1), San Antonio, TX, Feb 99, 225-243.

[Kramer et al 1993] J. Kramer, M. K. Senehi, "Feasability Study: Reference Architecture for Machine Control Systems Integration", Technical Report, National Institute Standard and Technology, 1993

[Kruchten 1995] P. Krutchen. *"Architectural blue-prints, the 4+1 view of software architecture". IEEE Software, 21(4), April 1995*

[Kruchten 1998] P. Krutchen, "The Rational Unified Process: An Introduction", Reading M.A: Addison Wesley, 1998

[Krut 1993] R.W. Krut, "Teleoperting 001 tool support into the Featured Oriented Domain Analysis Methodology", Technical Report SEI/CMU, 1993.

[Lopez 2000] Marta López, *"An Evaluation Theory Perspective of the Architecture Tradeoff Analysis Method SM (ATAM SM)"*, Technical Report, CMU/SEI-2000-TR-012, ESC-TR-2000-012, Sept 2000.

[Loques et al 2000] O. Loques et al, "On the Integration of Configuration and Meta-Level Programming Approaches", Papers from OORaSE 1999, 1st OOPSLA

Workshop on Reflection and Software Engineering, Denver, CO, USA, November 1999. Lecture Notes in Computer Science 1826 Springer 2000, ISBN 3-540-67761-5

[Lorenzo et al 2002] A.Lorenzo, I. Ramos, J.A. Carsí, J. Ferrer, "METAOASIS: An Architecture Description Language for Open Distributed Systems Modeling". Proceedings of the 2nd WSEAS, International Conference on SCIENTIFIC COMPUTATION andSOFT COMPUTING (publicación en Julio 2002)

[Lubars 1988] M.D. Lubars. "Domain Analysis and Domain Engineering in Idea", Technical Report, Microelectronics and Computer Technology Corporation (Austin) 1988

[Maimon 1987] O.Z. Maimon. "Real time Operational Control of Flexible Manufacturing Systems, International Journal of Manufacturing Systems, 6 (2), 1987

[Marciniak 1994] Marciniak, John J., ed. *Encyclopedia of Software Engineering*, 131-165. New York, NY: John Wiley & Sons, 1994.

[McCabe 1989] McCabe, Thomas J. & Butler, Charles W. "Design Complexity Measurement and Testing." *Communications of the ACM 32*, 12 (December 1989): 1415-1425.

[Mckenney 1996] P.E. Mckenney, "Selecting Locking Primitives for Parallel Programming", Communications of the ACM, October-1996

[Medvidovic et al 1997] N. Medvidovic et al, "A Classification and Comparison Framework for Software Architecture Description Languages", University of California Irvine, Technical Report, TR-UCI-ICS-97-02

[Medvidovic et al 1997] N. Medvidovic et al, "Modeling Software Architectures in the Unified Modeling Language", University of Southern California, Technical Report, USC-CSE-2000-512

[Metha et al 2000] N. Metha et al, "Towards a Taxonomy of Software Connectors", Proceedings of the ICSE 2000, Limerick (Ireland), ACM Press.

[Meyer 1992] Bertrand Meyer: *Applying "Design by Contract*, in *Computer (IEEE)*, vol. 25, no. 10, October 1992, pages 40-51.

[Meyer 1992] Bertrand Meyer, "Applying Design by Contract", IEEE Computer, Oct. 1992, 40-51

[Meyer 1998] B. Meyer, http://www.elj.com/eiffel/bm/

[Microsoft 1995] Microsoft, "The Component Object Model Specification", version 0.9, Octubre 1995.

[Microsoft 2001] Microsoft, ".NET Framework developers Guide", http://msdn.microsoft.com/.

[Miller 1991] D.J. Miller, "An Object Oriented Environment for Robot Systems Architectures", IEEE Control Systems, 11 (2), February 1991.

[ObjectTime 2000] http://www.objectime.com

[Oman 1991] Oman, P. *HP-MAS: A Tool for Software Maintainability, Software Engineering* (#91-08-TR). Moscow, ID: Test Laboratory, University of Idaho, 1991.

[OMG 2000] OMG Unified Modeling Language Specification. Version 1.4 beta R1, Nov 2000, http://cgi.omg.org

[Ortiz et al 2000] F. Ortiz, A. Iborra, F. Marín, B. Alvarez, J.M. Fdez. Meroño, "GOYA: A teleoperated system for blasting applied to ship maintenance", 3rd International Conference on Climbing and Walking Robots, CLAWAR'2000, Oct-2000

[Parnas 1972] D.L. Parnas, "On the Criteria of Decomposing Systems into Modules", Communications of the ACM 15, 12 (Dec-72): 1053-1058

[Parnas 1976] D.L. Parnas, "On the Design and development of Software Families", IEEE transactions on Software Engineering, 2(1), january 1996

[Pastor et al 1996] J.A. Pastor, S. Organai, B. Alvarez, "Actividades de desarrollo de herramientas para labores de mantenimiento teleoperadas en generadores de vapor", XII Reunión Anual Sociedad Nuclear, Octubre 1996

[Pastor et al 1998] J.A. Pastor, b. Alvarez, A. Iborra, Meroño, " Un underwater teleoperated vehicle for inspection and retrieving", In VRMech'98, 1st International Symposium- CLAWAR'98, Brussels, Nov-98

[Pastor et al 2000] J.A. Pastor, L. Alcedo, A. Iborra, B. Alvarez, J. Jimenez, "TRON: A teleoperated System for the maintanance of the lower internals of the PWRs reactor vessels", EUREL European Advanced Robotic Systems Masterclass and Conference Robotics 2000, Salford, UK, April 2000

[Plessel 1998] T. Plessel, "Design by Contract: A Missing Link In the Test for Quality Software", Lockheed Martin/US EPA, August 1998

[Popkin 1997] Popkin Software & Systems. "The User Guide for the System Architect Family of Tools", 1997

[Prieto 1991] R. Prieto Díaz, "Reuse Library Process Model", Technical Report, 03041-002, STARTS 1991

[Ran 1998] A. Ran, "Architectural Structures and Views", ISAW'98, Proceedings of the 3rd International Workshop on Software Architectures, Nov 1-2, 1998, Orlando, FL

[Rational 2002] Rational Software Corporation, www.rational.com

[RationalRose 2000] Rational ROSE'2000, "Manual of Rational Rose 2000".

[Roberts et al 1996] D. Roberts et al, "Evolving Frameworks. A pattern Language for Developing Object-Oriented Frameworks", Proceedings of the PloP-3, 1996

[ROOM 1994] B. Selic et al, "Real Time Object Oriented Modeling", Weley, New York (NY), 1994, http://www.objecttime.on.ca

[Rumbaugh et al 1098] J. Rumbaugh, I. Jacobson, G. Booch, *"The Unified Modeling Language Reference Manual"*, Reading M.A: Addison Wesley, 1998

[Rumbaugh et al 1991] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, "Object-Oriented Modeling and Design", Englewood Cliffs, NJ: Prentice Hall, 1991

[Sanz 1990] R. Sanz, "Arquitectura de Control Inteligente de Procesos", PhD thesis, Departamento de Automática, Ingeniería Electrónica e Informática Industrial (UPM), 1990.

[SARPA 1995] PROYECTO SARPA: Software con Arquitectura Reconfigurable para Robots de Mantenimiento de Centrales Nucleares, Expediente MINER ATYCA 753/95

[Schmid 1997] H.A. Schmid, "Systematic Framework Design by Generalization", Communications of the ACM, 40(10): 48-51, 1997

[Schmidt 1996] D. C. Schmidt, "Using Design Patterns to Develop Reusable Object-Oriented Software", ACM Computing Surveys, December 1996, http://www.acm.org/surveys/1996/

[Scholl et al] K.U. Scholl et al, "MCA - An Expandable Modular Controller Architecture", 3rd Real-Time Linux WorkShop.

[SEI-ata 2002] *"The Architecture Tradeoff Analysis Method"*, WWW:URL:http://www.sei.cmu.edu/ata/ATAM/index.htm (2000)

[SEI-cbs 2002] *"COTS Based Systems", WWW:URL:http://www.sei.cmu.edu/cbs/cbs_description.html (2000)*

[SEI-lpd 2002] www.sei.cmu.edu/plp/plp_init.html

[Selic et al 1998] B. Selic, J. Rumbaugh: *Using UML for Modeling Complex Real-Time Systems*, Rational Whitepaper, 1998

[Shaw 2001] M. Shaw, "The Coming-of-Age of Software Architecture Research", Proceedings of the 23rd International Conference on Software Engineering, págs 656-666ª, Toronto (Canada), IEEE Computer Society, 2001.

[Shaw et al 1996] M. Shaw, D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline", Prentice Hall, 1996

[Shaw et al 1997] M. Shaw, P. Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems", Proceedings of COMPSAC, Washington DC, August 1997

[Shlaer et al 1988] S.Shlaer, J. Mellor, J. Stephen, "Object Oriented Analysis-Modelling, the Word in Date", Englewood Cliffs, Prentice-Hall, 1988

[Simmons 1994] R. Simmons, "Structured Control for Autonomous Robots", IEEE Transactions on Robotics & Automation, 10(1):34-43, Feb. 1994

[Smolander et al 2001] K. Smolander et al, "Required and Optional Viewpoints. What is included in Software Architecture", Telecom Business Research Center, Lappeeranta University of Technology, http://www.lut.fi/TBRC,

[Sollbach 1989] E.M. Sollbach, "Real Time Control of Robots: Strategies for Hardware and Software Development", Robotics and Computer-Integrated Manufacturing, 6, 1989.

[Soni et al 1995] D. Soni et al, "Software Architecture in Industrial Applications", Proceedinds of the International Conference on Software Engineering, 196-210, Seattle, April-95

[Stewart et al 1990] D.B. Steward, D.E. Schmitz, "Implementing Real Time Robotics Systems Using Chimera ii", IEEE International Conference on Robotics and Automation, 1990.

[Stewart et al 1997] D.B. Steward, et al "Design of Dinamically Reconfigurable Real-Time Software Using Port-Based Objects", IEEE Trans. On Software Engineering, Vol 23, No. 12, Dec 1997, págs 759-776

[Sun 2002] http://developer.java.sun.com/developer/infodocs

[Svahnberg et al 2000] M. Svahnberg et al, "Designing Components for Variability", University of Karlskrona/Ronneby, Department of Software Engineering and Computer, CBSE Reports.

[Szyperski 1998a] Clements Szyperski, "Component Software: Beyond Object Oriented Programming", Addison-Wesley Longman, Harlow, UK, 1998.

[Szyperski 1998b] Clements Szyperski, "Components versus Objects", ObjectView Issue 5, Object/Component Architecture Series, www.ratio.co.uk , 1998.

[Szyperski et al 1997] C. Szyperski et al, "Component Oriented Programming", M. Muhlhauser, editor, Special Issues in Object Oriented Programming - ECOOP96 Workshop Reader. Dpunkt Verlag, Heidelberg.

[TAFIM 1996] Defense Information System Agengy (1996). Department of Defense Technical Architecture Framework for Information Management.

[Thai et al 1999] T. L. Thai et al, "Learning DCOM", O'Reilly & Associates.

[TRON 1996] PROYECTO EUREKA-TRON (EU-1565), Expediente MINER ATYCA 163/96

[TRON 1997a] PROYECTO EUREKA-TRON (EU-1565), Expediente MINER ATYCA P194/1997

[TRON 1997b] "Proyecto TRON: Plan de Calidad", ENWESA ATR-C-001, Sep 97

[TRON 1997c] "Proyecto TRON: Especificación Técnica del Proyecto TRON", ENWESA ATR-E-001, Jul 97

[TRON 1998] PROYECTO EUREKA-TRON (EU-1565), Expediente MINER ATYCA P107/1998, Contrato CDTI: 980204

[USC 2002] www.usc.edu

[Voa 1999] J. Voa, "Software Quality's Eigth Greatest Myths", IEEE Software Sep/Oct 99

[Woods 1999] S.G.Woods, M.R. Barbacci, "Architectural Evaluation of Collaborative Agent_Based Systems", Technical Report, CMU/SEI-99-TR-025, ESC-TR-99-025, Oct-1999

[Yourdon 1979] Yourdon, Ed and Larry L. Constantine. Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design. (1979), Prentice Hall

Anexo I

Análisis del dominio de Teleoperación.

La obtención de una arquitectura de referencia implica un análisis de su dominio de aplicación, que permita identificar, coleccionar, organizar y representar la información relevante para el dominio, basándose en el estudio de sistemas existentes y sus historias de desarrollo [Kang et al 1990]. Este estudio da como resultado la identificación de las partes comunes a todas las aplicaciones del dominio y es la base sobre la que se sustentan el diseño de una arquitectura genérica de dominio y el desarrollo de componentes software reutilizables para la implementación de la misma. El proceso completo se muestra en la figura A1.1

[Alvarez 1997] realiza el análisis y diseño del dominio de teleoperación, cuyo resultado es la arquitectura de referencia de la que se ocupa este trabajo de tesis, y que, muy resumidamente, se describen en las siguientes líneas.

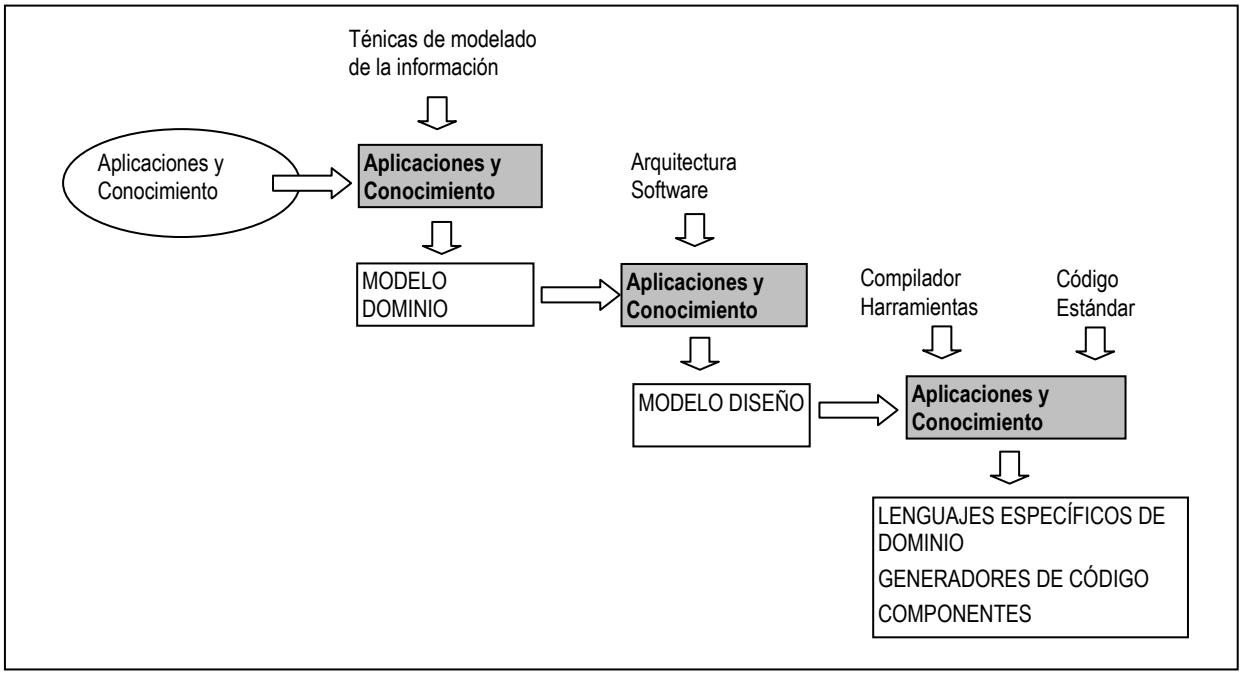


Figura A1.1: Actividades de la Ingeniería de Dominio

A1.1 Análisis de dominio

El análisis de dominio realizado en [Alvarez 1997] sigue el **método FODA** (*Featured-Oriented Domain Analysis*) desarrollado por el SEI de la CMU y descrito en [Kang et al 1990].

El primer paso del método consiste en realizar un **análisis de contexto**, que define el ámbito del dominio de aplicación, establece sus límites, determina sus sub-dominios e identifica las estructuras y flujos de datos comunes de cada una de las aplicaciones identificadas en el dominio. El resultado del análisis de contexto es un **modelo de contexto**, definido por un **diagrama estructural** que sitúa el dominio en relación con otros de nivel superior e inferior, y un **diagrama de flujo de datos**, que muestra el flujo de datos entre el dominio que se analiza y las entidades de otros dominios con las que se comunica.

Una vez obtenido el modelo de contexto, se realiza un análisis funcional, donde se identifican las similitudes y diferencias que caracterizan al conjunto de aplicaciones del dominio, dando como resultado un **modelo de dominio**, que consta a su vez de tres modelos, **el modelo de características**, **el modelo de información** y **el modelo funcional**.

El modelo de características recoge el conocimiento que los usuarios finales tienen sobre las capacidades esenciales de la aplicación para resolver sus necesidades, los servicios que proveen y los entornos operativos en los que se ejecutan. Las características se clasifican, por un lado, entre esenciales y opcionales y, por otro, entre características en tiempo de compilación, en tiempo de carga y en tiempo de ejecución. Esta taxonomía servirá de base para una posterior parametrización de la arquitectura.

El modelo de información o modelo de relaciones entre entidades, es una adaptación del modelo entidad-relación de Chen [Chen 1976] al que se han añadido los conceptos de generalización y agregación. Por último, **el modelo funcional u operacional** identifica, utilizando como guía los dos modelos anteriores, las similitudes y diferencias entre las aplicaciones identificadas en el dominio, clasificándolas en dos categorías, especificación de funciones y especificación de comportamiento. La primera describe aspectos estructurales en términos de entradas, salidas, actividades, datos internos y flujos de datos. La segunda describe el comportamiento de la aplicación en términos de eventos, entradas, estados, condiciones y transiciones, empleando para ello las *activity charts* y *state charts* de Harel [Harel 1990].

[Alvarez 1997] desarrolla todas estas actividades y define cada uno de los modelos arriba mencionados. Para ello, lleva a cabo una revisión de las arquitecturas y aplicaciones representativas del dominio considerado (en la figura A1.2 se muestran las más importantes), las cuales analiza según los niveles de abstracción propuestos por [Kramer et al 1993]. [Alvarez 1997] parte del estudio de las **arquitecturas de control** descritas en la literatura (primer nivel de abstracción), obteniendo una información de control genérica e independiente de dominio y por tanto aplicable a robots, vehículos, maquinaría, etc. La **teleoperación** (segundo nivel de abstracción) surge con la necesidad de llevar a cabo tareas no repetitivas de cierta complejidad, que no pueden ser programadas de antemano y que a menudo se desarrollan en ambientes hostiles o de difícil acceso. Un sistema teleoperado debe responder tanto a las órdenes del operador como a estímulos externos, integrando control en tiempo real. Por último, y con objeto de limitar el dominio a los sistemas de teleoperación y reducir la cantidad de información adicional necesaria para la implementación (cuarto nivel de abstracción), [Alvarez 1997] define el tipo de aplicaciones para las que se va a utilizar la arquitectura (tercer nivel de abstracción).

Las figuras A1.3, A1.4, A1.5 y A1.6 muestran algunos de los diagramas arriba mencionados para la arquitectura de referencia, tal y como se definen en [Alvarez 1997]. En la figura se han

abreviado las referencias por razones de espacio, pero son fácilmente identificables en la bibliografía.

Arquitecturas de Control		
Arquitectura	Descripción/Características	Referencias
Arquitecturas de control centralizadas.	Un solo controlador realiza todas las tareas de control. Suelen incluir una base de datos centralizada con protocolos de acceso simples. No necesita mecanismos de comunicación entre controladores. Muy específicas. Falta de modularidad. Dificil reutilización de componentes para otros sistemas.	✓ CIMPLICITY [Joh90] ✓ [Mai87]
Arquitecturas de control jerárquicas.	Estructura en forma de árbol, con un controlador superior y varios subordinados, que interaccionan entre sí a través de un protocolo comando-estado. Pueden ser sistemas distribuidos. Modularidad (cada controlador es un módulo) y Extensibilidad. Degradación controlada. Estructuración en niveles.	✓ AMRF (1) [Jon86] ✓ IMDAS (2) [Bark88] ✓ [Dor91] ✓ GISC (3) [Gri92]
Arquitecturas de control con técnicas de inteligencia artificial	Arquitecturas de control reactivo para trabajar en ambientes dinámicos y no estructurados, haciendo uso de reglas heurísticas, redes neuronales y sistemas borrosos.	✓ [Bro86] ✓ [Ark92] ✓ [San90]
Arquitecturas de Control en Tiempo Real [KS93]		
MAUV (4)	Arquitectura en tiempo real para vehículos submarinos.	✓ [Alb87]
ARTICS (5)	Modelo de referencia para sistemas de timepo real que incluye componentes software, herramientas y protocolos de comunicación.	✓ [Alb90]
NASREM (6)	Modelo de referencia adoptado por la NASA para el control de robots en estaciones espaciales.	✓ [Albu87]
Arquitecturas de Teleoperación		
Controladores genéricos e interfaces de alto nivel para controladores existentes	Sistemas enfocados al control de robots. Gran énfasis en la planificación y programación de tareas. Sin embargo, no proveen un entrono reconfigurable para la teleoperación.	✓ [Hay86], [Sol89], [Ste90] ✓ [Mil91] ✓ [Boi89], [Bro86], [Albu87]

(1): AMRF: *Automated Manufacturing Research Facilities*.
(2): IMDAS: *Integrated Manufacturing Data Administration System*
(3): GISC: *Generic Intelligent System Control*
(4): MAUV: *Multiple Autonomous Vehicles*
(5): ARTICS: *Architecture for Real Time Intelligent Control Systems*
(6): NASREM: *NASA Standard Reference Model*.

Figura A1.2: Arquitecturas y aplicaciones analizadas en [Alvarez 1997]

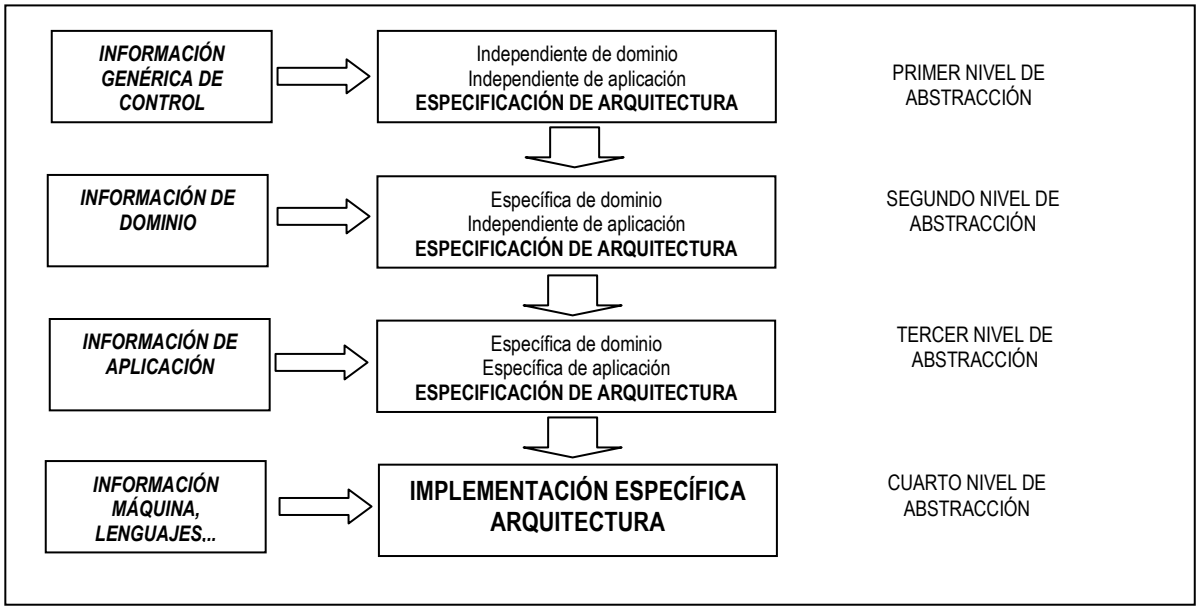


Figura A1.3: Niveles de abstracción para la definición de arquitecturas software [Kra93]

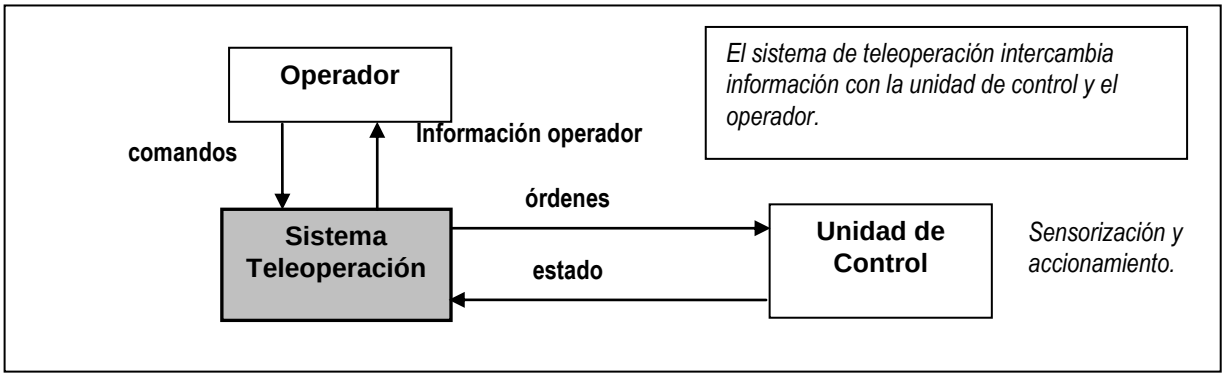


Figura A1.4: Diagrama de contexto.

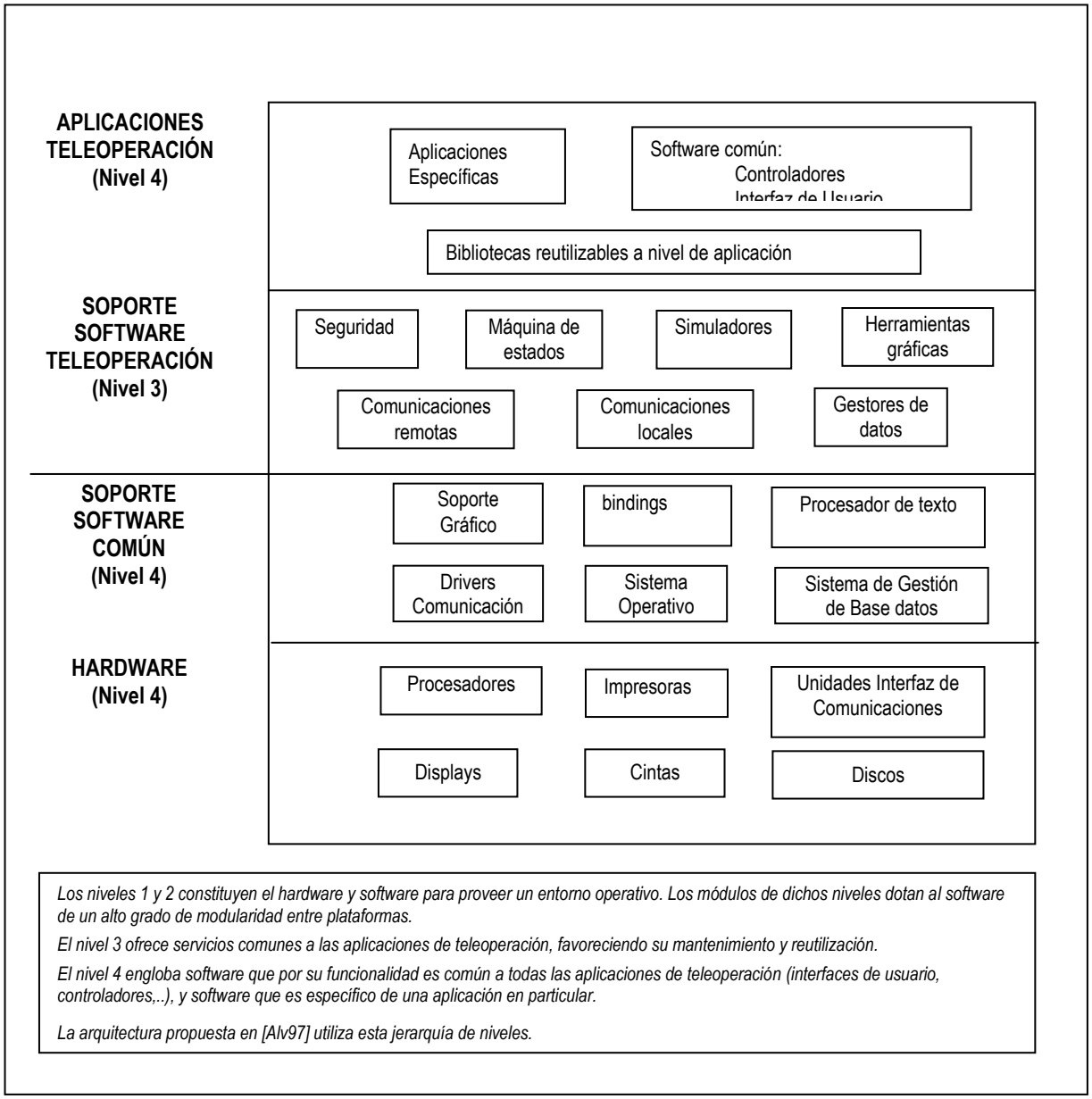
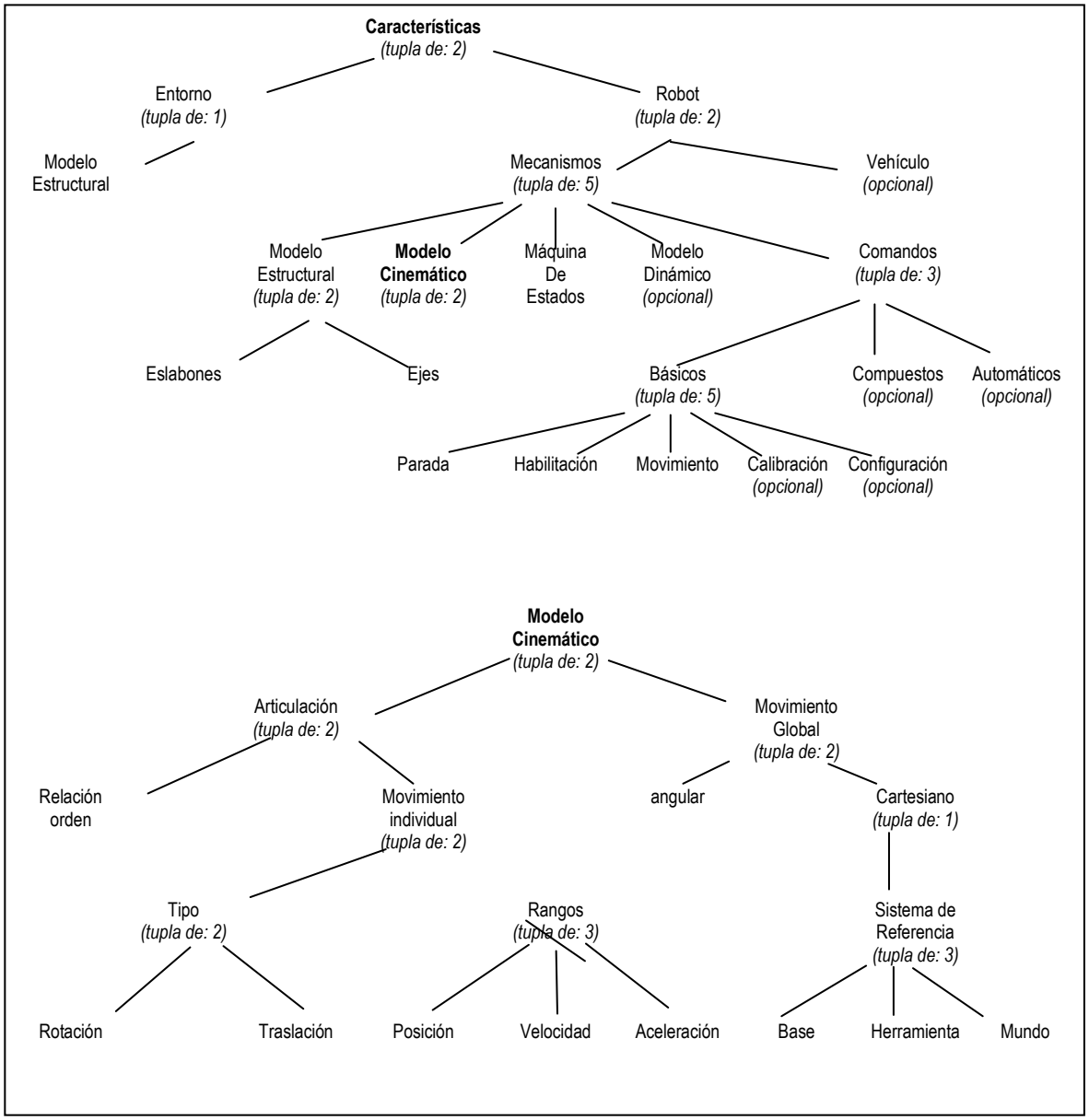


Figura A1.5: Diagrama Estructural del Dominio "Sistemas de Teleoperación".



El diagrama se representa utilizando la notación gráfica definida en [Krut93], y muestra una agrupación lógica de características generales y otras que son optativas y opcionales.

Las características generales u obligatorias se representan por su nombre y la notación "tupla de: x", donde x es el número de carcaterísticas en las que se descompone.

Las características alternativas se representan por su nombre y la notación "una tupla de: x", donde x es el número de alternativas.

Si una característica es opcional se expresa explícitamente después del nombre.

Todas las características están descritas en [Alv97]

Figura A1.6: Diagrama de Características.

Anexo II

Plantillas de Atributo.

Se consideran las diguientes plantillas:

Modificabilidad:

M1: Independencia del sistema operativo y de la plataforma.

M2: Capacidad de distribución.

M3: Independencia de la infraestructura de comunicaciones.

M4: Integración de servicios y utilidades.

M5: Adaptabilidad a diferentes interfaces de usuario.

M6: Adaptabilidad a diferentes entornos de operación.

M7: Adaptabilidad a diferentes misiones.

M8: Adaptabilidad a diferentes mecanismos (vehículos, brazos o herramientas).

M9: Adaptabilidad a diferentes configuraciones de mecanismos.

M10: Integración de componentes software comerciales.

Rendimiento:

R1: Tareas Críticas.

R2: Tareas no críticas.

Seguridad:

S1: Supervisión del estado del sistema.

S2: Gestión de Comandos.

S3: Mecanismos de Parada Segura.

S4: Acceso a la Interfaz de Usuario.

Disponibilidad/Fiabilidad

D1: Disponibilidad de las Comunicaciones.

D2: Disponibilidad de las Interfaces de Usuario.

D3: Disponibilidad de los Mecanismos de Parada Segura.

Interoperabilidad

I1: Escenarios de la interoperabilidad.

P1: Independencia del sistema operativo y de la plataforma.

Descripción:
Los sistemas deben adaptarse a cambios de plataforma y sistemas operativo.

Escenarios Abstractos:
Estímulo: Cambio del sistema operativo en alguno de los nodos incluidos en el sistema.
Respuesta: El número de cambios debe ser limitado y no debe propagarse por el sistema.

Estrategias:
Separación de conceptos.
Uso de estándares (p.e: POSIX)

Mecanismos:
Nivel de acceso a servicios del sistema operativo que proporcione una interfaz abstracta de acceso a los mismos.

Atributos relacionados:
Rendimiento ↓ : Aumenta el número de llamadas.
Flexibilidad en el despliegue físico ↑: No dependencia del S.O. de las plataformas.

Directrices y características de las que se deriva:
Los sistemas deben poder ejecutarse sobre diferentes sistemas operativos y plataformas.

- Especialización de los sistemas: No todas las aplicaciones de teleoperación demandan los mismos servicios del sistema operativo.
- Caracter distribuido de los sistemas: Los procesos asignados a los diferentes nodos pueden demandar servicios diferentes (pe: algunos nodos necesitan S.O. en tiempo real y otros no).
- Mantenibilidad de los sistemas: No todas las partes del sistema evolucionan al mismo ritmo. Puede ser necesario cambiar el S.O., manteniendo el resto de la aplicación.

M2: Adaptabilidad a diferentes despliegues

Descripción:
Diferentes procesos, controladores, servidores y susbsistemas deben poder asignarse a distintos nodos, al menos en tiempo de compilación y carga.

Escenarios Abstractos:
Estímulo: Asignación de uno o varios procesos a diferentes nodos.
Estímulo: Asignación de diferentes subsistemas a diferentes nodos.
Respuesta: El sistema debe seguir ofreciendo el mismo o similar comportamiento.

Estrategias:
Separación de conceptos. Indirección.
Alta cohesión, bajo acoplamiento.
Mecanismos de comunicación, preferiblemente asíncronos, basados en el paso de mensajes.
Criterios de estructuración del sistema en subsistemas (COMET, [Gomma 2000]).
Crierios de despliegue (COMET, [Gomma 2000]).
Uso de Middleware (RPC, DCOM, CORBA, DCE, Java RMI, etc.)

Mecanismos:
- Proxy/Stub.
- Broker
- Criterios de estructuración del sistema en subsistemas:
  - ✓ Agregación: Las entidades que formen parte del mismo agregado deben estar en el mismo subsistema.
  - ✓ Localización: Si dos entidades pueden ser potencialmente ubicadas en nodos diferentes deben asignarse a diferentes subsistemas.
  - ✓ Servicios:Clientes y servidores deben estar en diferentes subsistemas.
  - ✓ Control: Un controlador y las entidades con las que interacciona *directamente* deben estar en un mismo subsistema.
- Criterios de despliegue.
  - ✓ Proximidad a los dispositivos físicos: Proximidad de los componentes a los dispositivos físicos que controlan o de los que reciben datos
  - ✓ Autonomía: Incrementar autonomía de los nodos. Asignar a cada nodo responsabilidades concretas lo menos acopladas posible con las del resto.
  - ✓ Rendimiento: Asignar tareas más críticas a nodos específicos.
  - ✓ Interfaces de usuario y servidores pueden asignarse a diferentes nodos.

Atributos relacionados:
Rendimiento ↑ : Se favorece el paralelismo. Asignación de tareas críticas a nodos específicos.
Disponibilidad ↑ : En caso de que un nodo deje de funcionar, pueden asignarse a otro sus responsabilidades.

Directrices y características de las que se deriva:
La arquitectura debe ser válida para el mayor rango posible de sistemas de teleoperación (idealmente para todos), y constituir un marco adecuado para la definición de arquitecturas más específicas y para el desarrollo de componentes software reutilizables

- Carácter distribuido de los sistemas. Habitualmente los sistemas constan de dos nodos, pero puede haber más o uno sólo.
- Posibilidad de incluir utilidades de simulación, visión artificial, sistemas de navegación, etc., que puedan actuar como proveedores de servicios (servidores) ubicados en diferentes nodos.
- Rendimiento. Posibilidad de ubicar las tareas más críticas o más exigentes en términos de computación en nodos específicos.

M3: Independencia de la infraestructura de comunicaciones.

Descripción:
Los sistemas deben adaptarse al cambio de los enlaces de comunicaciones entre los diferentes nodos físicos del sistema.

Escenarios Abstractos:
Estímulo: Cambio del enlace de comunicaciones entre dos nodos dados.
Estímulo: Cambio del protocolo de comunicaciones subyacente.
Estímulo: Adición/Eliminación de mensajes.
Estímulo: Modificación del formato de los mensajes.
Respuesta: El número de cambios debe ser limitado y no debe propagarse por el sistema.

Estrategias:
Separación de conceptos. Indirección.
Uso de Middleware

Mecanismos:
Nivel de acceso a servicios de comunicaciones que proporcione una interfaz abstracta de acceso a los mismos.

Atributos relacionados:
Rendimiento ↓ : Aumenta el número de llamadas.
Uso de COTS ↑: Empleo de *middleware*.
Flexibilidad en el despliegue físico ↑

Directrices y características de las que se deriva:
Los sistemas deben poder ejecutarse sobre diferentes sistemas operativos y plataformas.

- Especialización de los sistemas: No todas las aplicaciones de teleoperación demandan los mismos enlaces de comunicaiones.
- Carácter distribuido de los sistemas.
- Mantenibilidad de los sistemas: No todas las partes del sistema evolucionan al mismo ritmo. Puede ser necesario cambiar el enlace de comunicaciones manteniendo el resto de la aplicación.

Observaciones:
Puede considerarse un caso particular de la adaptabilidad a diferentes plataformas y sistemas operativos, sin embargo las comunicaciones tienen suficiente entidad en los sistemas de teleoperación como para ser consideradas por separado.

M4: Integración de servicios y utilidades.

Descripción:
Diferentes servicios o utilidades deben poder integrarse en el sistema de teleoperación.

Escenarios Abstractos:
Estímulo: Incorporación de una nueva utilidad o suministrador de servicios al sistema.
Estímulo: Eliminación de una utilidad o servicio existente en el sistema.
Estímulo: Modificación de los servicios provistos por una determinada utilidad del sistema.
Respuesta: El número de cambios debe ser limitado y no debe propagarse por el sistema.

Estrategias:
Separación de conceptos.
Encapsulación.
Módulos de desacoplo.

Mecanismos:
Cliente-servidor.
Mediador.
Editor-Subscriptor.
Almacén abstracto de datos.
Comunicaciones Asíncronas.

Atributos relacionados:
Rendimiento: ↓ Se incrementa el número de llamadas entre módulos (mediadores).
↑ Incorporación de servicios encargados de tareas específicas.
Adaptación a diferentes interfaces de usuario ↑: Posibilidad de incorporar herramientas software comerciales.
Utlilización de COTS: .↑ Posibilidad de incorporar herramientas software comerciales.
Desarrollo rápido de aplicaciones.
Usabilidad ↑: Incorporación de servicios útiles para el operador.
Disponibilidad ↑: Posibilidad de replicar servicios.

Directrices y características de las que se deriva:

La arquitectura debe contemplar la posibilidad de usar componentes comerciales cuando se estime que ofrecen servicios interesantes para el sistema, bien directamente (herramientas software), bien indirectamente (a través de bibliotecas software).

- Especialización de los sistemas: No todas las aplicaciones de teleoperación demandan los mismos servicios o utilidades.
- Es habitual que se necesiten utilidades de simulación, representación gráfica, visión artificial, navegación, realimentación de pares y fuerzas, etc. Las aplicaciones más sencillas sólo demandan alguna de ellas (a veces ninguna), las más complejas pueden necesitar de muchas.
- Mantenibilidad de los sistemas: Posibilidad de añadir/eliminar utilidades o servicios según se vayan necesitando.

M5: Adaptabilidad a diferentes interfaces de usuario.

Descripción:
Los sistemas deben adaptarse a interfaces de usuario muy diferentes, que incluyen dispositivos específicos como joysticks, botoneras, reproducciones a escala de los dispositivos, utilidades de visión artificial, etc. y con muy distintas necesidades de representación gráfica y disposición de las ventanas.

Escenarios Abstractos:
Estímulo: Adición, eliminación, modificación de las ventanas o de los elementos que contienen.
Estímulo: Adición de nuevas interfaces (botoneras, joysticks, reproducciones de los dispositivos, etc) o sustitución de unas interfaces por otras (ratón por joystick, interfaz gráfica por textual, etc).
Respuesta: El resto del sistema no debe verse afectado.

Estrategias:
Separación de la interfaz de usuario del resto de la aplicación.
Uso de bibliotecas y utilidades comerciales (Motif, Visual C++, Java swing, etc.)

Mecanismos:
Módulos de desacoplo.
Modelo-Vista-Controlador.
Editor-Subscriptor.

Atributos relacionados:
Rendimiento: ↓ Se incrementa el número de llamadas entre módulos (mediadores, niveles).
↑ La interfaz puede ejecutarse en nodos específicos, liberando al resto.
Utlilización de COTS: .↑ Posibilidad de incorporar utilidades software comerciales.
Usabilidad ↑: Incorporación de servicios útiles para el operador.
Disponibilidad ↑: Posibilidad de replicar interfaces.

Directrices y características de las que se deriva:
La arquitectura debe ser válida para el mayor rango posible de sistemas de teleoperación (idealmente para todos).

Debe, asimismo, constituir un marco adecuado para la definición de arquitecturas más específicas y para el desarrollo de componentes software reutilizables

La arquitectura debe contemplar la posibilidad de usar componentes comerciales cuando se estime que ofrecen servicios interesantes para el sistema, bien directamente (herramientas software), bien indirectamente (a través de bibliotecas software).

- Especialización de los sistemas: No todas las aplicaciones de teleoperación demandan las mismas interfaces de usuario.
- Es habitual que se necesiten utilidades de simulación, representación gráfica, visión artificial, navegación, realimentación de pares y fuerzas, etc. Las aplicaciones más sencillas sólo demandan alguna de ellas (a veces ninguna), las más complejas pueden necesitar de muchas.

Observaciones:
Puede considerarse un caso particular de la adaptación a nuevos servicios. Está estrechamente relacionada con la posibilidad de utilizar herramientas y componentes software comerciales.

M6: Adaptabilidad a diferentes entornos de operación.

Descripción:
Los sistemas deben adaptarse para trabajar en diferentes entornos operativos.

Escenarios Abstractos:
Estímulo: Cambio en la geometría del entorno (entornos estructurados).
Estímulo: Cambio de entorno estructurado a no estructurado.
Estímulo: Cambio de las condiciones ambientales del entorno.
Respuesta: El número de cambios debe ser limitado y no debe propagarse por el sistema.

Estrategias:
Separación de conceptos.
Modelado del entorno y de los dispositivos.
Incorporación de utilidades de visión artificial.

Atributos relacionados:
Usabilidad ↑: Incorporación de servicios útiles para el operador.
Uso de COTS ↑: Incorporación de servicios útiles para el operador.
Adaptabilidad a diferentes misiones y mecanismos.

Directrices y características de las que se deriva:
La arquitectura debe ser válida para el mayor rango posible de sistemas de teleoperación (idealmente para todos).

- Especialización de los sistemas: Cada sistema de teleoperación trabaja en un entorno determinado.

Observaciones:
Puesto que la adaptación a diferentes entornos se basa (ver estrategias) en el uso de ciertos servicios, este atributo puede verse como un caso particular de la incorporación de nuevos servicios, y como aquel está relacionado con la posibilidad de utilizar herramientas y componentes software comerciales.

Es necesario considerar este atributo desde dos puntos de vista:

- Posibilidad de construir aplicaciones que operan en diferentes entornos.
- Posibilidad de adaptar un sistema dado para que pueda trabajar en diferentes entornos.

En el segundo caso la posibilidad de adaptar al sistema para trabajar en diferentes entornos está estrechamente relacionada con la posibilidad de adaptarlo a diferentes mecanismos y a diferentes misiones. Cada mecanismo tiene unas propiedades (mecánicas, eléctricas, estructurales, etc.) que le permiten realizar ciertas actividades en ciertos entornos.

Si el cambio entre entornos implica a dos entornos estructurados normalmente es suficiente con disponer de utilidades de modelado y representación gráfica. Si el entorno cambia, basta con modificar el modelo. En entornos no estructurados la utilidad de estas herramientas es limitada y puede ser necesaria la incorporación de un sistema de visión artificial.

M7: Adaptabilidad a diferentes misiones.

Descripción:
Posibilidad de adaptar al sistema para la realización de nuevas misiones.

Escenarios Abstractos:
Estímulo: Modificación misión existente:
Adición/Eliminación de pasos de la misión.
Modificación del orden de los pasos de la misión.
Modificación de la *máquina de estados* de la misión.
Estímulo: Adición de nuevas misiones.
Estímulo: Eliminación de misiones.
Estímulo: Construcción de nuevas misiones a partir de las existentes.
Respuesta: Las modificaciones deben ser limitadas y no propagarse por el sistema.

Estrategias:
Separación de conceptos.

Mecanismos:
Asignar la ejecución de las misiones a componentes software específicos (controladores de misión).
Encapsular los conceptos propios de cada misión en componentes separados.

Atributos relacionados:
Adaptabilidad a diferentes entornos y mecanismos.
Incorporación de servicios.
Comunicaciones (sincronismo entre mecanismos).

Directrices y características de las que se deriva:
La arquitectura debe ser válida para el mayor rango posible de sistemas de teleoperación (idealmente para todos).

Debe, asimismo, constituir un marco adecuado para la definición de arquitecturas más específicas y para el desarrollo de componentes software reutilizables.

Los sistemas deben incorporar fácilmente nuevas misiones y permitir cambios en las ya existentes.

- Especialización de los sistemas: Cada sistema debe realizar un conjunto de misiones, en general muy específicas.
- Mantenibilidad: La ejecución de una misión determinada está sujeta a modificaciones. Durante la vida de un sistema es posible que surjan nuevas actividades a realizar.

Observaciones:
Una misión es la ejecución por los mecanismos de una serie de actividades, realizadas en secuencia o según la lógica impuesta por una máquina de estados. En ocasiones una misión compleja no es más que la composición de otras más simples. Una misión es sobre todo una *pieza de funcionalidad*, que implica la ejecución de ciertos procedimientos y el acceso a diferentes servicios. Sin embargo, la adaptabilidad a nuevas misiones no depende de las características particulares de cada misión, su funcionalidad, sino de la forma de organizar dicha funcionalidad. Es necesario separar los aspectos ligados a la misión del resto.

Puesto que las misiones se realizan en un determinado entorno y utilizando determinados mecanismos, la adaptabilidad a nuevas misiones está ligada a la adaptabilidad a diferentes entornos, mecanismos y configuraciones (combinaciones vehículo-brazo-herramienta).

En ocasiones la misión sólo implica a un mecanismo (realización de un proceso de herramienta o ejecución de un movimiento o una secuencia de movimientos). En otras implica a varios. Para que varios mecanismos colaboren en una misión es necesario que sincronicen sus actividades e intercambien mensajes. Esto revela una nueva dimensión de la adaptabilidad a nuevas misiones: la necesidad de establecer medios de comunicación flexibles entre los controladores de los mecanismos.

Puesto que la realización de ciertas misiones necesita de ciertos servicios, la adaptabilidad a nuevas misiones está relacionada con la incorporación de nuevos servicios, y a través de ésta con la posibilidad de utilizar herramientas y componentes software comerciales.

Aún más: A menudo las misiones deben seguir un procedimiento más o menos estricto que debe ser seguido por el operador. La aplicación debe guiar la ejecución de las misiones, razón por la cual la adaptabilidad a nuevas misiones está relacionada con la adaptabilidad a nuevas interfaces y a la usabilidad.

M8: Adaptabilidad a diferentes mecanismos (vehículos, brazos o herramientas).

Descripción:
Adaptación de los sistemas a diferentes mecanismos (brazos, vehículos y herramientas).

Escenarios Abstractos:
Estímulo: Modificación del mecanismo:
- Modificación de la estructura.
- Modificación de la cinemática.
- Modificación de los dispositivos asociados.
  - Modificación/Adición/Eliminación accionadores.
  - Modificación/Adición/Eliminación sensores.
  - Modificación/Adición/Eliminación dispositivos E/S
- Modificación del control
  - Modicación/Adición/Eliminación de comandos o funcionalidad.
  - Modificación máquina de estados.
  - Modificación/Adición/Eliminación de mecanismos de sincronismo (eventos, mensajes…)

Adición de un nuevo mecanismo.
Eliminación de un mecanismo.
Sustitución de un mecanismo por otro.

Estrategias:
Separación de conceptos.
Alta cohesión/bajo acoplamiento
Encapsulación, tipos abstractos de datos.
Herencia y composición.

Mecanismos:
Separación de aspectos estructurales, cinemáticos y de control.
Modelado incremental de la funcionalidad (implementación de interfaces abstractas).
Definición de mecanismos (brazos, vehículos, herramientas) abstractos.
Uso de herencia e interfaces (clases abstractas)
Uso de plantillas y genéricos.
Definición de una interfaz abstracta de acceso a los mecanismos.

Atributos relacionados:
Adaptabilidad a nuevas misiones
Adaptabilidad a nuevos entornos

Directrices y características de las que se deriva:
La arquitectura debe ser válida para el mayor rango posible de sistemas de teleoperación.

Debe, asimismo, constituir un marco adecuado para la definición de arquitecturas más específicas y para el desarrollo de componentes software reutilizables

Los sistemas deben adaptarse con facilidad al empleo de nuevos mecanismos (brazos, vehículos y herramientas) o a la modificación de los existentes.

- Especialización de los sistemas: Cada sistema debe realizar un conjunto de misiones, en general muy específicas.
- Gran dispersión de características de los mecanismos teleoperados. Como consecuencia de la especialización, la complejidad y características de los vehículos, brazos y herramientas utilizados varían enormemente de unas aplicaciones a otras.
- Mantenibilidad: Los mecanismos están sujetos a continuas mejoras. En ocasiones se estima oportuno sustituir un mecanismo por otro, pero manteniendo el resto de las características del sistema (cambio del brazo o la herramienta por otros más modernos, modificaciones en los mismos, etc.)

M9: Adaptabilidad a diferentes configuraciones de mecanismos.

Descripción:
Adaptación a diferentes combinaciones de robots, vehículos y herramientas.

Escenarios Abstractos:
Estímulo: Cambio/Adición/Eliminación de vehículo.
Estímulo: Cambio/Adición/Eliminación de brazo.
Estímulo: Cambio/Adición/Eliminación de herramienta.
Respuesta: Las modificaciones deben ser limitadas y no propagarse por el sistema.

Estrategias:
Separación de conceptos.
Mecanismos de sincronización.

Mecanismos:
Controlador global (por encima de los controladores de los dispositivos)
Interfaces abstractas para el sincronismo (definición de una interfaz abstracta de acceso a los mecanismos).

Atributos relacionados:
Extensibilidad.
Adaptabilidad a nuevos mecanismos.
Adaptabilidad a nuevas misiones.

Directrices y características de las que se deriva:
La arquitectura debe ser válida para el mayor rango posible de sistemas de teleoperación (idealmente para todos).

Debe, asimismo, constituir un marco adecuado para la definición de arquitecturas más específicas y para el desarrollo de componentes software reutilizables.

- Distintas configuraciones de mecanismos. Cada sistema debe realizar un conjunto de misiones, en general muy específicas, que requieren el uso de diferentes combinaciones de mecanismos.

M10: Integración de componentes software comerciales.

Descripción:
Debe existir la posibilidad de usar componentes comerciales que ofrezcan servicios interesantes para el sistema, bien directamente (herramientas software), bien indirectamente (a través de bibliotecas software).

Escenarios Abstractos:
Estímulo: Uso de COTS para la implementación de susbsistemas.
Respuesta: Los cambios deben ser limitados y no propagarse por el sistema.

Estrategias:
Separación de conceptos.
Encapsulación.
Módulos de desacoplo.

Mecanismos:
Cliente-servidor.
Adaptadores (Wrappers)
Mediador.

Atributos relacionados:
Rendimiento: ↓ Se incrementa el número de llamadas entre módulos (mediadores).
↑ Incorporación de servicios encargados de tareas específicas.
Adaptación a diferentes interfaces de usuario ↑: Posibilidad de incorporar herramientas software comerciales.
Usabilidad ↑: Incorporación de servicios útiles para el operador.
Disponibilidad ↑: Posibilidad de replicar servicios.
Rapidez de desarrollo

Directrices y características de las que se deriva:
La arquitectura debe contemplar la posibilidad de usar componentes comerciales cuando se estime que ofrecen servicios interesantes para el sistema, bien directamente (herramientas software), bien indirectamente (a través de bibliotecas software).

- Desarrollo rápido de aplicaciones.

Observaciones:
Puede considerarse un caso particular de la adaptación a nuevos servicios, sin embargo hay algunas diferencias:

- El uso de los COTS no se limita a la incorporación o implementación de servicios.
- Los COTS tienen su problemática propia:
  - ✓ Compatibilidad entre diferentes versiones.
  - ✓ Compatibilidad entre COTS.
  - ✓ Suposiciones arquitecturales de los COTS.
  - ✓ Sustitución de unos COTS por otros.

Este atributo está expresado con una gran generalidad, que es a estas alturas inevitable. Hasta que no se plantee el diseño de la arquitectura no se podrá determinar cuáles son los componentes susceptibles de ser implementados por COTS. En general, los mismos mecanismos y estrategias que favorecen el resto de las *adaptabilidades* favorecen también el uso de COTS.

R1: Tareas críticas

Suelen estar relacionadas con procesos de control de los mecanismos (vehículo-brazo-herramienta) cercanos al hardware. Por ejemplo:

- ✓ Control del bucle de realimentación de los servos.
- ✓ Lectura de sensores y accionamiento de actuadores.
- ✓ Procesamiento de alarmas relacionadas con el funcionamiento del hardware.
- ✓ Procesos o servicios relacionados con la parada segura del sistema.

En este tipo de tareas, los eventos pueden llegar al sistema de forma períodica (muestreo) o esporádica (alarmas e interrupciones), deben procesarse dentro de unos plazos determinados, sin excepciones, y habitualmente en un orden determinado. Las comunicaciones entre nodos entran dentro de esta categoría si las tareas que se ocupan de estos aspectos están distribuidas por el sistema. También entran dentro de ella los procesos servidores si los servicios que suministran son necesarios para llevar a cabo estas tareas.

Comportamiento temporal de tareas críticas

Descripción:

Las tareas críticas deben realizarse dentro de un plazo determinado desde su activación por el evento correspondiente. El comportamiento del sistema debe ser predecible.

Patrón de llegada de eventos: periódico, esporádico.

Respuesta: Dentro de un plazo.

Escenarios Abstractos:

Estímulos:

Variación del período de activación de las tareas.
Variación del tiempo de procesamiento de las tareas.
Variación del número de tareas.
Variación de los plazos de las tareas.
Ejecución de tareas en distintos nodos o en un solo nodo (carga de las comunicaciones y características de los enlaces de comunicaciones).

Respuesta: Procesamiento de las tareas dentro de sus plazos.

Estrategias:

- Minimización del intercambio de datos (las comunicaciones, ya sea a través de enlaces físicos, de paso de mensajes o de llamadas a procedimiento son el principal factor de carga del sistema).
- Buses y protocolos de comunicaciones determinísticos.
- Minimización del número de transformaciones de datos y del número de cálculos.
- Asignación de recursos: Estrategias para repartir los recursos disponibles.
- Arbitraje y uso de recursos: Estrategias para resolver conflictos de acceso a recursos (prioridades, desalojos,etc.) y optimizar el uso de los mismos.

Mecanismos:

- Asignación de recursos:
  - ✓ Balance de carga: Repartir la carga entre los procesadores.
  - ✓ *Bin packing*: Medida de la capacidad disponible y asignación en función de las necesidades de las tareas.
- Arbitraje y uso de recursos:
  - ✓ Planificación con desalojo (*Preemtive*)
    - Planificación por prioridades dinámicas (*earliest deadline first scheduling*).
    - Planificación por prioridades fijas (*rate monotonic analysis, deadline monotonic*).
    - Estrategias de servicio aperiódico (planificación de procesos aperiódicos en un contexto periódico, *slack stealing*).
  - ✓ Exclusión mutua, sincronización.
    - Zonas críticas, semáforos, monitores, *rendezvous*, etc.
  - ✓ Otros.
    - Caché, tablas pre-calculadas, hardware específico, etc.

R1: Tareas críticas (Continuación)

Atributos relacionados:

Modificabilidad:

- *Portabilidad (plataformas, sistemas operativos, enlaces de comunicaciones).*

  Los mecanismos que facilitan la portabilidad (nivel de acceso) pueden suponer un empeoramiento del rendimiento a través de las llamadas entre módulos.

  No todos los sistemas operativos ofrecen los servicios que pueden ser necesarios (procesos ligeros, planificación con prioridades, predictabilidad, etc).

  No todos los enlaces de comunicaciones tienen el ancho de banda adecuado, ni son determinísticos.

  Por otro lado, la posibilidad de utilizar buses de alto rendimiento y sistemas operativos de tiempo real favorecen el rendimiento.

- *Integración de servicios:*

  Si las tareas críticas necesitan de ciertos servicios, el acceso a los mismos y su procesamiento se convierten asimismo en tareas críticas. Los mecanismos que facilitan la integración de servicios (mediadores, módulos de desacoplo, etc.) pueden suponer un empeoramiento del rendimiento a través de las llamadas entre módulos.

- *Uso de COTS.*
  Deben proporcionar el rendimiento adecuado y no hacer suposiciones arquitecturales que afecten o entren en conflicto con las estrategias seleccionadas. La posibilidad de integrar hardware específico puede mejorar mucho el rendimiento.

- *Flexibilidad en el despliegue:*
  Puede mejorar el rendimiento, pues facilita la asignación de recursos a diferentes nodos y favorece la definición de nodos *autosuficientes*.(Minimización de las comunicaciones).

- *Seguridad:*
  La seguridad depende del rendimiento. Si las tareas críticas no cumplen sus plazos el funcionamiento puede degradarse hasta límites inaceptables.

- *Usabilidad:*
  El rendimiento favorece la usabilidad. La información que se ofrece al operador debe reflejar el estado actual del sistema y los comandos deben activarse *inmediatamente*, dentro de los límites de la percepción humana.

Directrices y características de las que se deriva:
Tanto los comandos como la información de estado deben procesarse dentro de unos plazos que garanticen el correcto funcionamiento de los dispositivos y la validez de la información que se ofrece al operador y que manejan los diferentes subsistemas.

Si se utilizan enlaces de comunicaciones, éstos no deben suponer una pérdida de rendimiento que impida el cumplimiento del punto anterior.

Los sistemas deben incorporar mecanismos de parada segura que se activen por orden del operador o de forma autómatica en respuesta a ciertos eventos.

- Requisitos de tiempo real.

Observaciones:
Entre los mecanismos que se han enumerado para gestionar el comportamiento temporal de las tareas críticas no se han incluido ni el ejecutivo cíclico ni algoritmos de colas. El ejecutivo cíclico, aunque puede ser la solución optima para *un* sistema, no es capaz de adaptarse a las variantes de la familia. La teoría de colas modela muy bien el comportamiento medio del sistema (throughput, caso peor, factor de utilización, etc.), pero los algoritmos de planificación basados en prioridades son más adecuados para gestionar tareas con plazos fijos.

R1: Tareas no críticas

Tareas menos críticas
Son en general procesos de control de alto nivel no directamente relacionados con el funcionamiento seguro del sistema. La planificación y secuenciación de las misiones, la captura de datos que pueden procesarse en diferido y dicho procesamiento, ciertos procesos de diagnosis, la actualización de la interfaz de usuario, el suministro de ciertos servicios, etc. entran habitualmente dentro de esta categoría.

Descripción:
Las tareas menos críticas deben realizarse a una tasa (throughput) que proporcione un funcionamiento del sistema que pueda considerarse aceptable. El comportamiento del sistema debe ser predecible.

Patrón de llegada de eventos: periódico, esporádico.

Respuesta: Throughput (tareas realizadas o eventos servidos por unidad de tiempo).

Escenarios Abstractos:
Estímulos:

Variación del período de activación de las tareas.
Variación del tiempo de procesamiento de las tareas.
Variación del número de tareas.
Variación del througput requerido.
Ejecución de tareas en distintos nodos o en un solo nodo (carga de las comunicaciones y características de los enlaces de comunicaciones).

Respuesta:

Throughput.
Pueden definirse plazos y número de pérdidas de plazo (faltas) admisible, además de otros parámetros (jitter o máxima variación admisible en el intervalo de tiempo de respuesta entre el procesamiento de dos eventos, respuesta media y caso peor, etc)

Estrategias:
- Minimización del intercambio de datos (las comunicaciones, ya sea a través de enlaces físicos, de paso de mensajes o de llamadas a procedimiento son el principal factor de carga del sistema).
- Buses y protocolos de comunicaciones con suficiente ancho de banda.
- Minimización del número de transformaciones de datos y del número de cálculos.
- Asignación de recursos: Estrategias para repartir los recursos disponibles.
- Arbitraje y uso de recursos: Estrategias para resolver conflictos de acceso a recursos (prioridades, desalojos,etc.) y optimizar el uso de los mismos.

Mecanismos:
- Asignación de recursos:
  - ✓ Balance de carga: Repartir la carga entre los procesadores.
- Arbitraje y uso de recursos:
  - ✓ Planificación con desalojo (*Preemtive*)
    - Planificación por prioridades fijas (*rate monotonic analysis, deadline monotonic*).
    - Modelos de colas.
  - ✓ Exclusión mutua, sincronización.
    - Zonas críticas, semáforos, monitores, *rendezvous*, etc.
  - ✓ Otros.
    - Caché.

Atributos relacionados:
Pueden hacerse los mismos comentarios que en el caso de las tareas críticas.

Directrices y características de las que se deriva:
Tanto los comandos como la información de estado deben procesarse dentro de unos plazos que garanticen el correcto funcionamiento de los dispositivos y la validez de la información que se ofrece al operador y que manejan los diferentes subsistemas.

Si se utilizan enlaces de comunicaciones, éstos no deben suponer una pérdida de rendimiento que impida el cumplimiento del punto anterior.

Los sistemas deben incorporar mecanismos de parada segura que se activen por orden del operador o de forma automática en respuesta a ciertos eventos.

La actualización de la interfaz de usuario y la respuesta ofrecida por los servicios que se integren en el sistema caen habitualmente, aunque no siempre, dentro de esta categoría. Las comunicaciones son más o menos críticas dependiendo de la distribución de las responsabilidades entre nodos y del grado de autonomía de los mismos.

S1: Supervisión del estado del sistema.

Descripción:
Debe supervisarse el correcto comportamiento de los mecanismos y de todos aquellos subsistemas, tanto hardware (accionadores, motores, tarjetas de control, tarjetas E/S, enlaces de comunicaciones, etc) como software que tengan alguna influencia en la seguridad del sistema.

Escenarios Abstractos:
Estímulos:
a) Se produce una sobrecarga de motores o accionadores.
b) Se corta el suministro de energía.
c) Se produce un mal funcionamiento en los bucles de control de los mecanismos.
d) Se produce un mal funcionamiento de algún módulo hardware.
e) Se produce un mal funcionamiento de algún módulo software.
f) Se cortan las comunicaciones entre nodos.
g) Se activa una determinada alarma.

Respuestas:
a) Se detecta la sobrecarga y se desactivan los accionadores.
b) Se detecta el corte de alimentación y se activa el mecanismo de parada segura.
c) Se detectan las anomalías de los bucles de control y se detiene el proceso en curso (si son los servos, se detiene el movimiento, si es el control del proceso de herramienta se detiene dicho proceso) hasta que el control pueda reestablecerse.
d) Se detectan los fallos de los módulos y si es posible se prescinde de ellos o se sustituyen por otros.
e) Se detectan los fallos de los módulos y si es posible se prescinde de ellos o se sustituyen por otros.
f) Se detectan los fallos en las comunicaciones y si es posible se reestablecen.
g) Las alarmas deben procesarse en un plazo determinado que asegure que las causas que las producen son procesada antes de que se originen situaciones peligrosas para la integridad de personas y equipos.

En todos los casos se debe advertir al operador y si se estima que los errores son lo suficientemente graves se activa automáticamente la parada segura del sistema.

Estrategias y Mecanismos:
- Separación de conceptos.
- Bajo acoplamiento (evita la propagación de fallos), Alta cohesión (Facilita el tratamiento local y precoz de los errores).
- Detección de fallos.
- Tolerancia a fallos: Duplicación de servicios críticos (redundancia). Recuperación de errores, etc.
- Servicios de diagnosis y componentes diseñados para proporcionar información a dicho servicio.
- Auto-test de los componentes.
- Tratamiento de excepciones.
- Tareas específicas que supervisen y procesen periódicamente el estado del sistema.
- Tareas que se activen de forma automática ante la ocurrencia de ciertos eventos (pe: ciertas alarmas).
- Simulación on-line de los comandos.

Atributos relacionados:
- *Rendimiento:*
  Los requisitos temporales deben ser garantizados para asegurar que la información disponible sobre el sistema refleja fielmente su estado.

  Las tareas encargadas de la seguridad suponen una sobrecarga del sistema.

  A través del rendimiento se ven afectados otros atributos.

- *Modificabilidad:*
  Los componentes deben pensarse desde un principio para ser seguros. Los componentes críticos deben proporcionar interfaces a través de las cuales un servicio de diagnóstico o una tarea de supervisión pueda detectar fallos en su funcionamiento.

  Los principios de bajo acoplamiento y alta cohesión que facilitan el tratamiento de errores favorecen la modificabilidad en todos sus aspectos.

- *Disponibilidad:*
  Los mecanismos y estrategias arquitecturales que permiten la obtención de la disponibilidad son precisamente los mismos que pueden garantizar este aspecto de la seguridad.

S1: Supervisión del estado del sistema (Continuación).

Directrices y características de las que se deriva:

La arquitectura debe ser válida para el mayor rango posible de sistemas de teleoperación (idealmente para todos).

Deben definirse alarmas, mensajes y advertencias que prevengan al operador de cualquier mal funcionamiento del sistema. Si los errores son graves debe activarse automáticamente los mecanismos de parada segura.

Observaciones:

A menudo se olvida que las fuentes de error no sólo tienen su origen en los mecanismos y en el hardware asociado, sino también en la propia aplicación. Plazos que se pierden, subsistemas que se bloquean, excepciones que no se tratan adecuadamente, etc., pueden ser la causa de errores que pongan en peligro la seguridad de los operarios y comprometan la integridad de los equipos.

Las medidas software deben ser complementadas con otras de diferente índole. Por ejemplo, de poco vale implementar mecanismos software de parada segura si los dispositivos mecánicos no incorporan frenos o sistemas de enclavamiento que puedan entrar en funcionamiento cuando se corta la alimentación de los accionadores.

S2: Gestión de Comandos.

Descripción:
Gestión de los comandos para:

- Impedir que se ejecuten comandos inseguros.
- Asegurar que los comandos se ejecutan según el plan previsto.

Escenarios Abstractos:
Estímulos:
a) El sistema entra en un estado en el que no es seguro ejecutar ciertos comandos.
b) La ejecución del comando es compatible con el estado, pero no puede determinarse si su ejecución es segura.
c) El comando no se ejecuta según el plan previsto.

Respuestas:
a) Inhabilitación de comandos no compatibles con el estado del sistema.
b) Comprobar la viabilidad de un comando antes de su ejecución (un comando es viable si su ejecución deja al sistema en un estado conocido y seguro).
c) Monitorizar la ejecución de los comandos comprobando que se realizan según el plan previsto (no se observan discrepancias intolerables entre los estados real y esperado, no vencen timeouts, etc.).

Estrategias y Mecanismos:
- Gestión de estados y modelado de comandos:
  - ✓ Cada estado tiene asociados una serie de comandos que pueden ejecutarse en el mismo.
  - ✓ Cada comando puede ejecutarse cuando el sistema se encuentra en ciertos estados.
- Separación de conceptos y encapsulación:
  - ✓ Encapsular con cada estado los comandos que se pueden ejecutar en el mismo.
  - ✓ Encapsular con cada comando los estados en los que se puede ejecutar.
- Controladores para los comandos.
- Controladores para los mecanismos.
- Simulación previa de los comandos.

Atributos relacionados:
- *Rendimiento:*
  Los requisitos temporales deben ser garantizados para asegurar que la información disponible sobre el sistema refleja fielmente su estado. Los comandos deben ser monitorizados en tiempo real.
  A través del rendimiento se ven afectados otros atributos.
- *Integración de servicios y uso de COTS:*
  La simulación de comandos puede requerir del uso de herramientas muy potentes.
- *Modificabilidad*:
  Es posible que las medidas de seguridad se hagan más estrictas y que comandos que antes estaban habilitados en un estado tengan que deshabilitarse, o que las tolerancias entre estados real y esperado se hagan más pequeñas. También podría ocurrir lo contrario.

Directrices y características de las que se deriva:
Tanto los comandos como la información de estado deben procesarse dentro de unos plazos que garanticen el correcto funcionamiento de los dispositivos y la validez de la información que se ofrece al operador y que manejan los diferentes subsistemas.

Las operaciones sólo deben ejecutarse si existe una confianza razonable en que el resultado de las mismas no va a poner en peligro la integridad de bienes y equipos.

La arquitectura debe ser válida para el mayor rango posible de sistemas de teleoperación (idealmente para todos).

Observaciones:
Puede considerarse un caso particular de la monitorización del estado, sin embargo la ejecución de comandos introduce ciertas condiciones particulares. Una parte del estado del sistema debe supervisarse siempre, haya o no haya un comando en curso, otras partes son relevantes precisamente por el comando que se está ejecutando. Es necesario separar los aspectos específicos de cada comando.

La gestión de los comandos tiene que ver ante todo con la funcionalidad del sistema, que no es un atributo específicamente arquitectural. Sin embargo, el cumplimiento de este requisito representa una sobrecarga de actividades que puede afectar bastante al cumplimiento de otros atributos *más arquitectónicos*:

El estudio de la viabilidad de los comandos puede requerir del uso de ciertos servicios. La relación comandos/estados debe tenerse en cuenta a la hora de diseñar los módulos de control y su comportamiento dinámico (el estado del sistema puede cambiar durante la ejecución del comando, por causas achacables al mismo o no). La supervisión de la ejecución tiene importantes consecuencias en el rendimiento y a través de él en otros atributos del sistema.

La simulación on-line de los comandos puede suponer una fuerte sobrecarga del sistema. Un esquema alternativo, pero bastante útil, es aprovechar los datos de una simulación off-line para generar los estados esperados, que se van comparando periódicamente con los reales.

S3: Mecanismos de Parada Segura.

Descripción:
Deben disponerse mecanismos de parada de emergencia que se activen por orden del operador o cuando se detecten situaciones de riesgo o errores graves de funcionamiento.

Todas las interfaces del sistema deben disponer de un dispositivo de seguridad que permita detener inmediatamente a los mecanismos.

Escenarios Abstractos:
Estímulos:
a) El operador acciona el *botón* de parada.
b) Se detectan errores graves en el sistema.

Respuestas:
- Los mecanismos se detienen inmediatamente y entran en un estado conocido y seguro.

Estrategias y Mecanismos:
- Servicios de diagnosis.
- Tratamiento de excepciones.
- Tareas específicas que supervisen y procesen periódicamente el estado del sistema.
- Tareas que se activen de forma automática ante la ocurrencia de ciertos eventos (pe: ciertas alarmas).
- Simulación on-line de los comandos.
- Botón de parada segura.

Atributos relacionados:
Pueden hacerse los mismos comentarios que en los dos casos anteriores y además:

- *Usabilidad:*
  Los dispositivos de parada segura deben ser fácilmente identificables y accesibles por el operador. Su funcionamiento no debe depender del funcionamiento de la aplicación. Un icono en la interfaz de usuario no es suficiente. Deben proporcionarse dispositivos hardware que accedan directamente a los dispositivos.

  En ocasiones (actividades de prueba, programación y mantenimiento) puede ser necesario que el usuario (operador, programador o mantenedor) deba realizar una acción específica (p.e. mantener pulsado un botón) para accionar los dispositivos. En estas situaciones la parada segura está accionada por defecto.

Directrices y características de las que se deriva:
La arquitectura debe ser válida para el mayor rango posible de sistemas de teleoperación (idealmente para todos).

Los sistemas deben incorporar mecanismos de parada segura que se activen por orden del operador o de forma autómatica en respuesta a ciertos eventos.

S4: Acceso a la Interfaz de Usuario.

Descripción:
El acceso al software de control y arranque debe estar limitado mediante códigos de seguridad.

Escenarios Abstractos:
Estímulos: Un usuario no autorizado intenta acceder al sistema.
Respuestas: Se piden login y password.

Estrategias y Mecanismos:
- Contraseña.

Directrices y características de las que se deriva:
Las operaciones sólo deben ejecutarse si existe una confianza razonable en que el resultado de las mismas no va a poner en peligro la integridad de bienes y equipos.

No debe permitirse que personal no entrenado y debidamente cualificado tenga acceso a los mecanismos.

Observaciones
Puede ser aconsejable registrar las acciones realizadas por el operador para descubrir las causas de posibles errores.

D1: Disponibilidad de las Comunicaciones.

Descripción:
Los fallos en los enlaces de comunicaciones entre nodos deben al menos detectarse y si es posible corregirse.

Escenarios Abstractos:
Estímulos:
a) Un mensaje es enviado/recibido incorrectamente.
b) Un nodo no responde o no envía información a la tasa requerida.
c) No es posible reestablecer las comunicaciones con un nodo.
d) Excesivo tráfico. Pérdida de rendimiento.

Respuestas:
a) Al menos el nodo emisor detecta el fallo. Reintenta. Si tras un número determinado de reintentos persiste el fallo se establece un nuevo enlace de comunicaciones.
b) Los nodos receptores detectan el fallo. Se intenta restablecer las comunicaciones. Si tras un número determinado de reintentos el fallo persiste, existen las siguientes alternativas (los sistemas deben implementar al menos una de ellas):
   - ✓ Asignar las responsabilidades del nodo silencioso a otro nodo.
   - ✓ Si es posible trabajar sin los servicios que proporciona el nodo, se prescinde del mismo (modo de operación degradado)
   - ✓ Si es imposible trabajar sin los servicios del nodo, se activa parada segura.
c) Igual que el anterior.
d) Se detecta la sobrecarga. Existen las siguientes alternativas (los sistemas deben implementar al menos una de ellas):
   - ✓ Reconfigurar el despliegue físico para minimizar las comunicaciones.
   - ✓ Si las condiciones de seguridad lo permiten se disminuye el tráfico (modo de operación degradado). Si no, se activa parada segura.

En todos los casos el operador debe ser informado y en la medida de lo posible (dependiendo de la gravedad del fallo) se le debe dar la opción de continuar o detener la operación.

La reconfiguración del despliegue puede realizarse en tiempo de carga (parada del sistema y rearranque con nueva configuración).

Estrategias y Mecanismos:
- Tolerancia a Fallos(Medidas dinámicas, en tiempo de ejecución, de prevención, detección y corrección de fallos).
  - ✓ Reenvío de mensajes.
  - ✓ Número de secuencia en los mensajes.
  - ✓ Utilización de protocolo subyacente orientado a conexión.
  - ✓ Duplicado, replicado de enlaces de comunicaciones.
  - ✓ Adaptabilidad a diferentes despliegues (en tiempo de carga).
  - ✓ Control de tráfico.

Atributos relacionados:
- *Modificabilidad: Adaptabilidad a diferentes enlaces de comunicaciones:*
  Favorece la implementación de los mecanismos arriba mencionados, al *independizar* las comunicaciones del resto de la aplicación..
- *Modificabilidad: Adaptabilidad a diferentes despliegues físicos:*
  Favorece la implementación de los mecanismos arriba mencionados.
- *Seguridad: Supervisión del Estado del Sistema.*
  Puede considerarse como el mismo atributo visto desde otro punto de vista.

Directrices y características de las que se deriva:

Posibilidad de funcionamiento básico degradado si falla algún sistema. Al menos el indispensable para ejecutar la parada segura y rearrancar el sistema.

Si se utilizan enlaces de comunicaciones, los fallos en las mismas deben ser detectados y si es posible corregidos. Debe contemplarse la posibilidad de utilizar redundancia en las comunicaciones.

Observaciones:
Aunque los mecanismos de tolerancia a fallos se definen para el tiempo de ejecución, no es necesario que los sistemas reconfiguren su despliegue en este tiempo, siempre que:

- ✓ La parada y rearranque del sistema puedan realizarse en un tiempo razonable y
- ✓ Los mecanismos puedan mantenerse en un estado seguro mientras se reconfigura el sistema:

Dada la forma en que habitualmente se realizan los trabajos de teleoperación, la flexibilidad que proporciona la reconfiguración del despliegue en tiempo de ejecución no compensa la complejidad que implica.

D2: Disponibilidad de las Interfaces de Usuario.

Descripción:
El operador debe disponer de forma permanente de una interfaz de usuario mínima que le permita ejecutar al menos la parada segura del sistema y el rearranque del mismo.

Debe existir una interfaz reducida mediante la cual puedan ejecutarse los comandos más básicos de los mecanismos.

La probabilidad de que la interfaz de usuario se bloquee y obligue a utilizar la interfaz mínima debe ser pequeña, debiendo establecerse el valor de los parámetros de fiabilidad/disponibilidad de cada sistema concreto.

Escenarios Abstractos:
Estímulos:
a) La interfaz de usuario se bloquea.
b) La interfaz no se recupera pese a repetidos intentos por rearrancarla y los mecanismos se encuentran en una situación comprometida (es *necesario* moverlos).
c) La interfaz parece funcionar, pero los mecanismos no responden.
d) Una parte de las ventanas de la interfaz no aparecen o no se ejecutan correctamente.
e) El teclado, el ratón, el joystick, etc no responden.

Respuestas:
a) Existe una interfaz mínima alternativa mediante la cual se puede ejecutar la parada segura y rearrancar el sistema.
b) Existe una interfaz reducida mediante la cuál se pueden ejecutar comandos básicos.
c) Existe una forma de control alternativa (botonera, acceso directo a controladores de bajo nivel, etc) que permite operar los mecanismos de forma básica.
d) Igual que el anterior.
e) Igual que el anterior.

Estrategias y Mecanismos:
- Interfaces alternativas que:
  - ✓ Accedan al resto de la aplicación como la interfaz original, aunque implementando sólo una parte de la funcionalidad de aquella (p.e.: sustitución de la interfaz gráfica por interfaces textuales que permitan introducir los comandos más básicos)
  - ✓ Puenteen al resto de la aplicación (p.e: setas de seguridad, botoneras cableadas directamente a los accionadores de los dispositivos, etc).

Atributos relacionados:
- *Modificabilidad: Adaptabilidad a diferentes interfaces de usuario:*
  Favorece la implementación de los mecanismos arriba mencionados, al *independizar* la interfaz de usuario del resto de la aplicación..
- *Seguridad: Parada Segura:*
  Puede considerarse un aspecto del mismo atributo, pero considerado desde otro punto de vista.

Directrices y características de las que se deriva:

Posibilidad de funcionamiento básico degradado si falla algún sistema. Al menos el indispensable para ejecutar la parada segura y rearrancar el sistema.

D3: Disponibilidad de los Mecanismos de Parada Segura.

Descripción:
Debe existir al menos un mecanismo de parada segura accesible por el operador y cuyo funcionamiento no dependa del comportamiento de la aplicación.

Escenarios Abstractos:
Estímulos:
- La aplicación se bloquea.

Respuestas:
- El operador puede ejecutar una parada segura.

Estrategias y Mecanismos:
- Botón de parada segura directamente cableado a los dispositivos.

Atributos relacionados:
- *Seguridad: Parada Segura:*

Directrices y características de las que se deriva:

Posibilidad de funcionamiento básico degradado si falla algún sistema. Al menos el indispensable para ejecutar la parada segura y rearrancar el sistema.

I1: Escenarios de la interoperabilidad.

Descripción:

- Posibilidad de repartir las diferentes actividades de una misión entre diferentes sistemas de teleoperación. O dicho de otra manera, posibilidad de descomponer una misión en misiones más simples asignando cada una de ellas a un sistema de teleoperación determinado.
- Establecer mecanismos de sincronismo y coordinación de forma que los diferentes sistemas puedan acompasar el ritmo de sus trabajos.
- Establecer mecanismos que permitan un control global de todos los sistemas de teleoperación compatible con los sistemas de control propios de cada sistema individual.

Escenarios Abstractos:

Estímulos:

a) Se quiere repartir una misión compleja entre diferentes sistemas de teleoperación, de forma que cada uno realice una parte del trabajo.
b) Uno de los subsistemas ha terminado una parte de su trabajo, pero no puede continuar hasta que otro sistema termine una parte del suyo.
c) El operador de los sistemas desea consultar el estado global del sistema o de un subconjunto de los subsistemas.
d) El operador de los sistemas desea realizar una acción que afecta a todos los subsistemas o a una parte de los subsistemas.
e) El operador de los subsistemas desea reestructurar el reparto de tareas.
f) Se detectan problemas que deben ser resueltos a nivel de sistema individual.

Respuestas:

a) Existe una interfaz a través de la cual puede repertirse la misión entre los diferentes sistemas.
b) Existen mecanismos de coordinación y sincronismo entre los dispositivos que les permiten acompasar el ritmo de sus trabajos o detenerse si es necesario.
c) Existe una interfaz a través de la cual puede inspeccionarse el estado de cada subsistema, que ofrece además información global.
d) Existe una interfaz a través de la cual puede accederse a una parte de los comandos de cada subsistema. Puede realizarse un broadcast.
e) Idem a)
f) Puede delegarse el control a la interfaz de operador de cada sistema individual.

Anexo III

Escenarios de Evaluación.

Relación de Escenarios:

Escenario P1: Cambio de Sistema Operativo y/o Plataforma (pe: W2K por Linux, PC por WorkStation).

Escenario P2: Cambio de enlaces y protocolos de comunicaciones (pe: Etherner por CAN Bus, uso de tecnología *wireless*, etc).

Escenario P3: Inclusión de Middleware: Utilización de CORBA.

Escenario CD1: : Ejecutar **CinServer (GrpServer)** en otra plataforma

Escenario CD2: Ejecutar **LocalSys** y **TeleopSys** en la misma plataforma

Escenario IS1: Adición de un nuevo servidor.

Escenario IS2: Eliminación de un servidor (p.e: **CinServer** o **GrpServer**).

Escenario IS3: Sustitución de un servidor por otro (p.e: **CinServer** por **CinServer2**).

Escenario ACL1: : LocalSys produce información de estado adicional (nuevos datos, alarmas o eventos).

Escenario ACL2: LocalSys produce menos información de estado (se eliminan datos, alarmas o eventos).

Escenario ACL3: Aumenta la funcionalidad de **LocalSys. LocalSys** admite más comandos (o comandos con más parámetros)

Escenario ACL4 : Disminuye la funcionalidad de **LocalSys. LocalSys** admite menos comandos (o comandos con menos parámetros)

Escenario ACL5: Cambia el formato de la información producida por **LocalSys**, pero no su semántica.

Escenario ACL6: Cambia el formato de los comandos admitidos por **LocalSys**, pero no su semántica.

Escenario AMM1: Adición/eliminación/modificación de eventos y señales de sincronismo de **ArmController**

Escenario AMM2: Adición/eliminación/modificación de información de estado suministrada por **ArmController**

Escenario AMM3: Adición de un nuevo comando de **ArmController**

Escenario AMM4: Adición de una nueva misión, implicando varios mecanismos.

Escenario AMM5: Adición/Eliminación/Modificación de mecanismos. P.e.: Definición de una nueva herramienta.

Escenario AMM6: Dada una combinación vehículo-brazo-herramienta, cambiar herramienta en tiempo de ejecución.

Escenario AMM7: Cambia la cinemática y estructura del brazo.

Escenario AMM9: Cambio de un entorno operativo. De estructurado a NO estructurado.

Escenario FD1: Fallo del enlace de comunicaciones. Se recupera el enlace o se informa al controlador para que tome las medidas definidas en la especificación del sistema.

Escenario FD2: Falla la interfaz de Usuario. Está disponible una interfaz degradada con comandos básicos.

Escenario FD3: Falla la unidad de control local.

Escenario FD4: Falla **RemoteCtlr**.

Escenario FD5: : Falla un servidor

Escenario P1: Cambio de Sistema Operativo y/o Plataforma (pe: W2K por Linux, PC por WorkStation).

Atributos de Calidad Abstractos: 1.1: Portabilidad

Total Modificaciones:

- Implementaciones: nivel de acceso a los servicios del S.O.

Mecanismos Arquitecturales:

nivel de acceso a los servicios del S.O.

Realización:

- Reimplementar nivel de acceso a los servicios del S.O.

Observaciones:

La nueva plataforma y S.O. deben proporcionar unas prestaciones suficientes al resto de los componentes.
Si se considera la posibilidad de ejecutar LocalSys y TeleopSys en una misma plataforma es necesario tener en cuenta las necesidades de LocalSys cuando se elija el S.O. y la plataforma.
La arquitectura recomienda el uso de POSIX para minimizar el impacto del cambio del S.O:

Escenario P2: Cambio de enlaces y protocolos de comunicaciones (pe: Etherner por CAN Bus, uso de tecnología *wireless*, etc).

Atributos de Calidad Abstractos: 1.1: Portabilidad

Total Modificaciones:

- Implementaciones: nivel de acceso a los servicios de comunicaciones. CommSys

Mecanismos Arquitecturales:

nivel de acceso a los servicios del S.O. CommSys

Realización:

- Reimplementar nivel de acceso a los servicios de comunicaciones.

Observaciones:

Los nuevos enlaces y protocolos deben proporcionar unas prestaciones suficientes.
Es un escenario muy parecido al anterior (cambio de infraestructura), sin embargo la arquitectura trata a la infraestructura de comunicaciones de una forma especial, ya que la pone al mismo nivel que los subsistemas conceptuales.

Escenario CD1: Ejecutar **`CinServer (GrpServer)`** en otra plataforma

Atributos de Calidad Abstractos: 1.2: Capacidad de Distribución

Total Modificaciones:

- Implementaciones: **`CinDecoupler`**, **`ComDecoupler`**, **`ComListener`**, **`CommSys`**.

Mecanismos Arquitecturales:

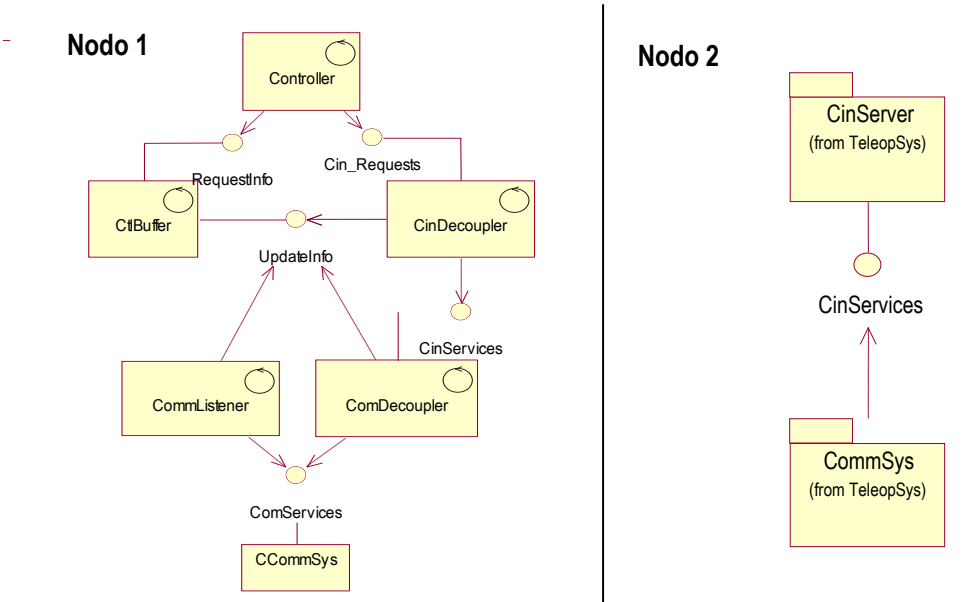
Interfaz abstracta **`Cin_Requests`**
Encapsulación de comunicaciones en **`CinDecoupler`**

Suposiciones:

La arquitectura no explica la forma en que se encapsulan en **`CinDecoupler`** las llamadas a la infraestructura de comunicaciones (representada por **`CommSys`**), aunque sugiere un patrón proxy. Para realizar el escenario supondremos que la arquitectura propone para todos los módulos de desacoplo la misma relación con la infraestructura de comunicaciones.

Realización:

1. Implementar la interfaz **`CinServices`** en **`ComDecoupler`**. (Al igual que se implementa la interfaz **`LocalCmds`** para proporcionar acceso al sistema de control local). De esta manera **`ComDecoupler`** hace las veces de representante local o proxy de **`CinServer`**.
2. Extender **`ComListener`** de forma que recoja las respuestas de **`CinServer`**.
3. Implementar en el nodo donde se ubique **`CinServer`** una infraestructura de comunicaciones análoga a la del nodo de **`TeleopSys`** (definiendo un nuevo **`CommSys`** para el nodo de **`CinServer`** o modificando el actual).
   Esta infraestructura deberá traducir los mensajes recibidos a peticiones de servicio a la interfaz **`CinServices`** de **`CinServer`**, recoger las respuestas y enviarlas a través de la red hasta **`CinListener`**, atravesando la infraestructura de comunicaciones definida en ambos nodos.



Observaciones:

Con la realización propuesta **`ComDecoupler`** y **`ComListener`** cambian cada vez que cambien **`LocalCmds`** o **`CinServices`** (o **`GrpServices`** ya que el esquema sería el mismo). Este extremo depende, no obstante de la organización de **`ComDecoupler`** y **`ComListener`**, cuestión que no se define explícitamente en la arquitectura.

El problema es que en la arquitectura describe en un mismo nivel las llamadas a la infraestructura y los subsistemas que pueden (o no) comunicarse usando dicha infraestructura.

Escenario CD2: Ejecutar **`LocalSys`** y **`TeleopSys`** en la misma plataforma

Atributos de Calidad Abstractos: 1.2: Capacidad de Distribución

Total Modificaciones Realización 1:

- Implementaciones: **`ninguna`**, pero pueden producirse serios problemas de rendimiento.

Total Modificaciones Realización 2:

- Implementaciones: **`ComDecoupler`**, **`ComListener`**, **`LocalSys, TeleopSys`**.
- Otros: Reestructuración de tareas. Nuevos mecanismos de sincronismo.

Mecanismos Arquitecturales:

Interfaz abstracta ***`LocalCmds`***
Encapsulación de comunicaciones en **`ComDecoupler y ComListener`**

Realización1:

1. Cambiar la dirección de **`LocalSys`**. (Tanto **`LocalSys`** como **`TeleopSys`** se ejecutan en la misma máquina).

Observaciones:

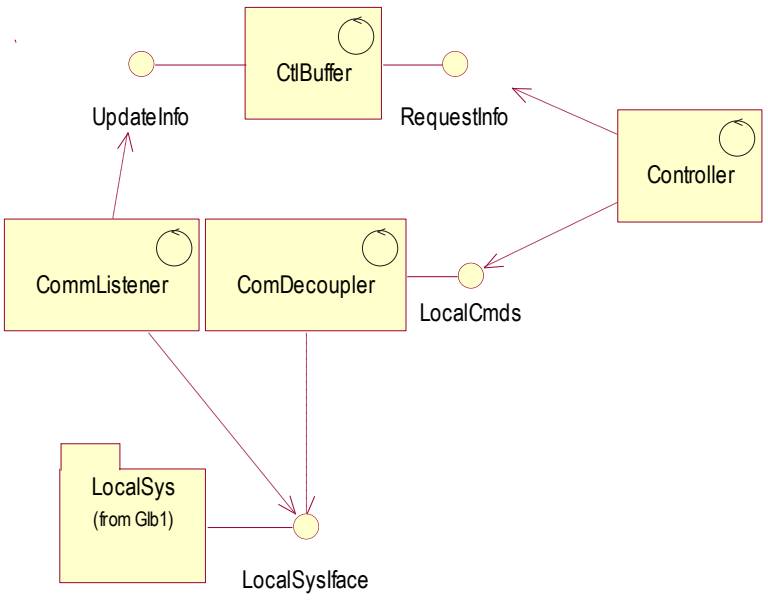
Desde el punto de vista de la modificabilidad, esta solución no tiene coste alguno, sin embargo puede introducir serios problemas de rendimiento teniendo en cuenta que:

- **`LocalSys`** y **`TeleopSys`** son procesos pesados.
- El paso de mansajes entre **`LocalSys`** y **TeleopSys** debe realizarse según una política de plazos.

Además, **`LocalSys`** tiene unos requisitos temporales mucho más estrictos que **`TeleopSys`** y no puede ejecutarse en cualquier plataforma ni con cualquier sistema operativo.

Realización2:

1. Eliminar llamadas a **`CommSys`**, reimplementando **`ComDecoupler`** y **`ComListener`** para que interaccionen directamente con **`LocalSys`**.
2. Unificar los hilos de **`LocalSys`** y **`TeleopSys`** en un solo proceso, definiendo los mecanismos de sincronización de acceso a recursos compartidos y una política de prioridades adecuada.



Escenario IS1: Adición de un nuevo servidor.

Atributos de Calidad Abstractos: 1.3: Integración de Servicios

Total Modificaciones:

- Nuevos Componentes: **`ServiceDecoupler`**.
- Nuevas Interfaces: **`ServicesInterface`**

Mecanismos Arquitecturales:

Interfaz abstracta de acceso a los servicios
Módulos de desacoplo.

Realización:

- Definir la interfaz abstracta y el módulo de desacoplo correspondientes.

Observaciones:

Si los servidores se implementan con COTS o se basan en la integración de herramientas comerciales es necesario estudiar la infraestructura que necesitan.

Escenario IS2: Eliminación de un servidor (p.e: **`CinServer`** o **`GrpServer`**).

Atributos de Calidad Abstractos: 1.3: Integración de Servicios

Total Modificaciones:

- Implementaciones: Eliminar el módulo de desacoplo correspondiente.

Realización:

- Eliminar el módulo de desacoplo correspondiente.

Escenario IS3: Sustitución de un servidor por otro (p.e: **`CinServer`** por **`CinServer2`**).

Atributos de Calidad Abstractos: 1.3: Integración de Servicios

Total Modificaciones:

No hay modificaciones.

Mecanismos Arquitecturales:

Interfaz abstracta **`Cin_Requests`**
Encapsulación de comunicaciones en **`CinDecoupler`**

Realización:

- Simplemente sustituir un servidor por otro.

Observaciones:

Si los servidores se implementan con COTS o se basan en la integración de herramientas comerciales es necesario estudiar la infraestructura que necesitan.

Escenario ACL1 : **`LocalSys`** produce información de estado adicional (nuevos datos, alarmas o eventos).

Total Modificaciones:

- Interfaces: ***`UpdateInfo`***, ***`RequestInfo`***, ***`UIControls`***, ***`UIRequests`***.
- Implementaciones: **`CtlrBuffer`**, **`CommListener`**, **`Controller`**, **`RemoteUI`**.

Razonamientos:

- **`CtlrBuffer`** debe modificarse para albergar los nuevos datos, y sus interfaces ***`UpdateInfo`*** y ***`RequestInfo`*** para hacerlos accesibles.
- Todos los componentes que dependan de dicha interfaces deben ser modificados, de forma que ***`CommListener`*** y **`Controller`** también deben ser modificados.
- La nueva información de estado debe hacerse accesible al operador remoto. Es necesario modificar también: **`RemoteUI`** y ***`UIControls`*** y a través de ellas **`UIDecoupler`**, ***`UIRequest`*** y nuevamente **`Controller`**.

Observaciones:

- **`Controller`** puede necesitar de varias modificaciones. A parte de la definición y manejo de nuevos tipos de datos, su máquina de estados puede tener que modificarse para tener en cuenta la nueva información.
- Las modificaciones de **`RemoteUI`** pueden ser más o menos costosas dependiendo de su organización (granularidad de sus ventanas de estado). La arquitectura no dice nada acerca de la organización de la interfaz de usuario.
- **`CommSys`** no necesita ser modificado en primera instancia ya que los servicios que ofrece ***`CommServices`*** son del estilo **`connect/disconnect`**, **`send/receive`** y no necesitan saber nada de la semántica de los datos.

Escenario ACL2: **`LocalSys`** produce menos información de estado (se eliminan datos, alarmas o eventos).

Total Modificaciones:

- Interfaces: ***`UpdateInfo`***, ***`RequestInfo`***, ***`UIControls`***, ***`UIRequests`***.
- Implementaciones: **`CtlrBuffer`**, **`CommListener`**, **`Controller`**, **`RemoteUI`**.

Razonamientos:

- **`CtlrBuffer`** debe modificarse para eliminar los datos, y sus interfaces ***`UpdateInfo`*** y ***`RequestInfo`*** para no hacerlos accesibles.
- Todos los componentes que dependan de dicha interfaces deben ser modificados, de forma que **`CommListener`** y **`Controller`** también deben ser modificados.
- La nueva información de estado que ya no es significativa debe eliminarse de la interfaz de usuario. Es necesario modificar también: **`RemoteUI`** y ***`UIControls`*** y a través de ellas **`UIDecoupler`**, ***`UIRequest`*** y nuevamente **`Controller`**.

Observaciones:

- **`Controller`** puede necesitar de varias modificaciones. Si la información eliminada incluye eventos, puede ser necesario reorganizar su máquina de estados.
- Las modificaciones de **`RemoteUI`** pueden ser más o menos costosas dependiendo de su organización (granularidad de sus ventanas de estado). La arquitectura no dice nada acerca de la organización de la interfaz de usuario.
- **`CommSys`** no necesita ser modificado en primera instancia ya que los servicios que ofrece ***`CommServices`*** son del estilo **`connect/disconnect`**, **`send/receive`** y no necesitan saber nada de la semántica de los datos.
- Un enfoque alternativo es no modificar nada. Sin embargo, este enfoque tiene dos serios inconvenientes de cara a la mantenibilidad y operabilidad del sistema:
  1. Se mantienen interfaces que carecen de significado y .**`BufferCtlr`** mantiene estructuras de datos sin significado.
  2. Se ofrece al operador información sin significado.
  3. La máquina de estados de **`Controller`** puede ser inconsistente.

Escenario ACL3: : Aumenta la funcionalidad de **`LocalSys. LocalSys`** admite más comandos (o comandos con más parámetros)

Atributos de Calidad Abstractos: 1.4: Adaptabilidad a cambios en la Unidad de Control Local

Total Modificaciones:

- Interfaces: ***`LocalCmds, RobCmds, UpdateInfo`***, ***`RequestInfo`***.
- Implementaciones: **`RemoteUI, UIListener, CtlrBuffer`**, **`ComDecoupler`**, **`Controller`**.

Razonamientos:

- ***`LocalCmds`*** debe modificarse o extenderse. para ofrecer los nuevos comandos, **`Controller`** para invocarlos y **`CommDecoupler`** para implementarlos.
- La nueva funcionalidad debe hacerse accesible al operador remoto. Dependiendo del alcance de los cambios (p.e: adición de movimientos cartesianos) es necesario modificar también: **`RemoteUI`** y ***`RobCmds`*** y a través de ellas **`UIListener`** .
- ***`UpdateInfo`*** debe modificarse para insertar el nuevo comando, **`BufferCtlr`** para albergarlos y ***`RequestInfo`*** para hacerlos accesibles a **`Controller`**, el cual debe ser a su vez modificado.

Observaciones:

- **`Controller`** puede necesitar de varias modificaciones. A parte de la definición y manejo de nuevos tipos de datos, su máquina de estados puede tener que modificarse para tener en cuenta los nuevos comandos.
- Las modificaciones de **`RemoteUI`** pueden ser más o menos costosas dependiendo de su organización (granularidad de sus ventanas de estado). La arquitectura no dice nada acerca de la organización de la interfaz de usuario.
- **`CommSys`** no necesita ser modificado en primera instancia ya que los servicios que ofrece ***`CommServices`*** son del estilo **`connect/disconnect`**, **`send/receive`** y no necesitan saber nada de la semántica de los datos.

Escenario ACL4: : Disminuye la funcionalidad de **`LocalSys. LocalSys`** admite menos comandos (o comandos con menos parámetros)

Atributos de Calidad Abstractos: 1.4: Adaptabilidad a cambios en la Unidad de Control Local

Total Modificaciones:

- Interfaces: ***`LocalCmds, RobCmds, UpdateInfo`***, ***`RequestInfo`***.
- Implementaciones: **`RemoteUI, UIListener, CtlrBuffer`**, **`ComDecoupler`**, **`Controller`**.

Razonamientos:

- ***`LocalCmds`*** debe modificarse para eliminar comandos, y **`Controller`** y **`CommDecoupler`** para ignorarlos.
- Debe eliminarse la funcionalidad accesible al operador. Dependiendo del alcance de los cambios (p.e: eliminación de movimientos cartesianos) es necesario modificar también: **`RemoteUI`** y ***`RobCmds`*** y a través de ellas **`UIListener`** .
- ***`UpdateInfo`*** , **`BufferCtlr`** y ***`RequestInfo`*** deben modificarse para eliminar los comandos, propagándose los cambios hasta **`Controller`**, el cual debe ser a su vez modificado.

Observaciones:

- **`Controller`** puede necesitar de varias modificaciones. A parte de la eliminación de tipos de datos, su máquina de estados *puede* tener que modificarse para ignorar los comandos eliminados.
- Las modificaciones de **`RemoteUI`** pueden ser más o menos costosas dependiendo de su organización (granularidad de sus ventanas de estado). La arquitectura no dice nada acerca de la organización de la interfaz de usuario.
- **`CommSys`** no necesita ser modificado en primera instancia ya que los servicios que ofrece ***`CommServices`*** son del estilo **`connect/disconnect`**, **`send/receive`** y no necesitan saber nada de la semántica de los datos.

Escenario ACL5: Cambia el formato de la información producida por **`LocalSys`**, pero no su semántica.

Atributos de Calidad Abstractos: 1.4: Adaptabilidad a cambios en la Unidad de Control Local

Total Modificaciones:

- Implementaciones: **`CommListener`**.

Razonamientos:

- Puesto que **`CommSys`** es inmune a los formatos y **`BufferCtlr`** ofrece una interfaz abstracta, todos los cambios pueden localizarse en **`CommListener`**

Escenario ACL6: Cambia el formato de los comandos admitidos por **`LocalSys`**, pero no su semántica.

Atributos de Calidad Abstractos: 1.4: Adaptabilidad a cambios en la Unidad de Control Local

Total Modificaciones:

- Implementaciones: **`CommDecoupler`**.

Razonamientos:

- Puesto que **`CommSys`** es inmune a los formatos y **`BufferCtlr`** ofrece una interfaz abstracta, todos los cambios pueden localizarse en **`CommDecoupler`**

Escenario AMM1 : Adición/eliminación/modificación de eventos y señales de sincronismo de **`ArmController`**

Atributos de Calidad Abstractos: 1.5: Cambios en mecanismos y misiones: Modificación funcionalidad de `RemoteCtlr`

Total Modificaciones:

- Interfaces: **`ArmSynch`**
- Implementaciones: **`ArmCtlr`**, **`ToolCtlr`**, **`VehCtlr`**

Mecanismos Arquitecturales:

Separación de los controladores de los diferentes mecanismos **`ArmCtlr`**, **`ToolCtlr`**, **`VehCtlr`**
Interfaces abstracta de sincronismo: ***`ArmSynch, ToolSynch, VehSynch`***

Realización (véase figura []):

1. Modificación de **`ArmController`** (debe cambiar su máquina de estados) y de su interfaz **`ArmSynch`**.
2. Modificación de los controladores que necesiten invocar la señal de sincronismo (incluyendo posiblemente un cambio de sus máquinas de estados).

Observaciones:

Al no definirse explícitamente ningún módulo que sincronice las acciones de los controladores, cada vez que cambia la interfaz de sincronismo de uno de los controladores, puede ser necesario (dependiendo del alcance de los cambios) modificar las implementaciones de todos los controladores.
El alcance de estas modificaciones es mayor de lo que refleja el escenario a primera vista. Imaginemos un controlador de robot que puede usarse con dos herramientas diferentes, pero que pueden ejecutar algunas misiones comunes (p.e: dos tipos de garras).

El cambio de sincronismo de `ArmCtler` obliga a modificar los controladores de todas las herramientas que invocan `ArmSynch`.

Escenario AMM2: Adición/eliminación/modificación de información de estado suministrada por **`ArmController`**

Atributos de Calidad Abstractos: 1.5: Cambios en mecanismos y misiones: Modificación funcionalidad de `RemoteCtlr`

Total Modificaciones:

- Interfaces: **`ArmSynch`**
- Implementaciones: **`ArmCtlr`**, **`ToolCtlr`**, **`VehCtlr`**

Mecanismos Arquitecturales:

Separación de los controladores de los diferentes mecanismos **`ArmCtlr`**, **`ToolCtlr`**, **`VehCtlr`**
Interfaces abstractas: ***`UIRequest, UIControls`***

Realización (véase figura []):

1. Modificación de **`ArmController`**.
2. Modificación de ***`UIRequest`***, **`UIDecoupler`**, ***`UIControls`*** y **`RemoteUI`**.
3. Si el nuevo comando implica la adición de nuevos eventos de sincronismo hay que realizar el escenario [].

Observaciones:

- Las modificaciones en ***`UIRequest`***, **`UIDecoupler`**, ***`UIControls`*** y **`RemoteUI`**, pueden ser más o menos costosas dependiendo de la organización de estos componentes e interfaces. La arquitectura sugiere el particionado de estos componentes de acuerdo con los mecanismos presentes en el sistema (igual que partciona **`RemoteCtlr`** en **`ArmCtlr`**, **`ToolCtlr`** y **`VehCtlr`**).

Escenario AMM3 : Adición de un nuevo comando de **`ArmController`**

Atributos de Calidad Abstractos: 1.5: Cambios en mecanismos y misiones: Modificación funcionalidad de `RemoteCtlr`

Total Modificaciones:

- Interfaces: ***`RobCmds, UpdateInfo, RequestInfo`*** y dependiendo de las características del comando ***`UIControls`***, ***`ArmSynch`***, ***`ToolSynch`***, ***`VehSynch`***.
- Implementaciones: **`ArmCtlr`**, **`BufferCtlr`**, **`UIListener, RemoteUI`** y dependiendo de las características del comando **`UIDecoupler, ToolCtlr y VehCtlr`**

Mecanismos Arquitecturales:

Separación de los controladores de los diferentes mecanismos **`ArmCtlr`**, **`ToolCtlr`**, **`VehCtlr`**
Interfaces abstracta: ***`RobCmds`***

Realización (véase figura []):

1. Ampliación de ***`RobCmds`*** y modificación de los componentes que la llaman (**`RemoteUI`**) e implementan (**`UIListener`**).
2. Ampliación de **`BufferCtlr`** y sus interfaces (***`UpdateInfo`*** y ***`RequestInfo`***).
3. Ampliación de **`ArmController`**.
4. Si el nuevo comando implica la adición de nuevos eventos de sincronismo hay que realizar el escenario [].
5. Si el nuevo comando implica nueva información de estado hay que realizar el escenario [].
6. Modificación de **`RemoteUI`** para incluir el nuevo comando.

Observaciones:

- Las modificaciones en ***`RobCmds`***, **`UIDecoupler`**, **`UIListener`**, ***`UpdateInfo`***, ***`RequestInfo`*** y **`BufferCtlr`**, pueden ser más o menos costosas dependiendo de la organización de estos componentes e interfaces. La arquitectura sugiere el particionado de estos componentes de acuerdo con los mecanismos presentes en el sistema (igual que partciona **`RemoteCtlr`** en **`ArmCtlr`**, **`ToolCtlr`** y **`VehCtlr`**).
- Obsérvese que si el comando implica la definición de nuevos eventos e información de estado, los cambios inducidos por este escenario se propagan espectacularmente.

EscenarioAMM4 : Adición de una nueva misión implicando varios mecanismos (p.e.: brazo y herramienta).

Atributos de Calidad Abstractos: 1.5: Cambios en mecanismos y misiones: Modificación funcionalidad de `RemoteCtlr`

Total Modificaciones:

- Interfaces: Una combinación de las modificaciones definidas en los escenarios [], [] y[].
- Implementaciones: Una combinación de las modificaciones definidas en los escenarios [], [] y[].

Mecanismos Arquitecturales:

Separación de los controladores de los diferentes mecanismos **`ArmCtlr`**, **`ToolCtlr`**, **`VehCtlr.`**
Interfaces abstracta de sincronismo: ***`ArmSynch, ToolSynch, VehSynch`***

Realización (véase figura []):

1. Si la misión implica nuevos comandos, realizar escenario []
2. Si la misión implica nueva información de estado, realizar escenario []
3. Si la misión implica nuevos eventos de sincronismo, realizar escenario [].

Observaciones:

- Las modificaciones dependen de la complejidad de la misión. Si la misión no implica nuevos comandos, información de estado o eventos, las modificaciones pueden ser muy pocas. Sin embargo, en general, la inclusión de una nueva misión de mediana complejidad puede implicar la modificación de las implementaciones de todos o una parte de los controladores y de sus interfaces de sincronismo.

Escenario AMM5: Adición/Eliminación/Modificación de mecanismos. P.e.: Definición de una nueva herramienta.

Atributos de Calidad Abstractos: 1.5: Cambios en mecanismos y misiones: Adición/Elimin./Modificación de mecanismos.

Total Modificaciones:

- Nuevos componentes: Tool_i_Ctlr].
- Otros: Definición de la estructura y cinemática de Tool_i y actualización de los mismos (si procede) en las bases de datos de los servidores gráfico (GrpServer) y cinemático (CinServer) y otros que pudiera haber en el sistema.

Mecanismos Arquitecturales:

Separación de los controladores de los diferentes mecanismos **`ArmCtlr`**, **`ToolCtlr`**, **`VehCtlr.`**
Definición de controladores genéricos.
Interfaces abstracta de sincronismo: ***`ArmSynch, ToolSynch, VehSynch`***

Realización (véase figura []):

1. Implementar el nuevo controlador.
2. Actualizar bases de datos de la aplicación.

Observaciones:

- Los controladores se cargan y descargan de la aplicación durante una operación de configuración, que en este caso debe poder realizarse en tiempo de ejecución. Los genéricos son una forma de proporcionar polimorfismo estático. Su uso dinámico presenta algunos problemas, pero no es imposible.

Escenario AMM6: Dada una combinación vehículo-brazo-herramienta, cambiar herramienta en tiempo de ejecución.

Atributos de Calidad Abstractos: 1.5: Cambios en mecanismos y misiones: Cambio de mecanismo en tiempo de ejecución.

Total Modificaciones: Ninguna.

Mecanismos Arquitecturales:

Separación de los controladores de los diferentes mecanismos **`ArmCtlr`**, **`ToolCtlr`**, **`VehCtlr.`**
Definición de controladores genéricos.
Interfaces abstractas de sincronismo: ***`ArmSynch, ToolSynch, VehSynch`***

Realización: Se sustituye un controlador por otro.

Observaciones:

- Los controladores se cargan y descargan de la aplicación durante una operación de configuración, que en este caso debe poder realizarse en tiempo de ejecución. Los genéricos son una forma de proporcionar polimorfismo estático. Su uso dinámico presenta algunos problemas, pero no es imposible, ya que puede definirse una reconfiguración de la aplicación (que incluya carga y descarga de procesos) para cambiar un controlador por otro.

Escenario AMM7 : Cambia la cinemática y estructura del brazo.

Atributos de Calidad Abstractos: 1.5: Cambios en mecanismos y misiones: Cambio de estruct y cinem. de los mecan.ismos

Total Modificaciones: Ninguna.

Mecanismos Arquitecturales:

Encapsulación de las propiedades estructurales y cinemáticas en **`CinServer`** y **`GrpServer`**

Realización: Actualizar bases de datos de la aplicación.

Escenario AMM8 : Cambio de un entorno operativo. De estructurado a estructurado.

Atributos de Calidad Abstractos: 1.5: Cambios en el entorno operativo: Cambio de un entorno estructurado a otro entorno estructurado

Total Modificaciones: Ninguna.

Mecanismos Arquitecturales:

Encapsulación de las propiedades estructurales del entorno en **`CinServer`** y **`GrpServer`**

Realización: Actualizar bases de datos de la aplicación.

Escenario AMM9 : Cambio de un entorno operativo. De estructurado a NO estructurado.

Atributos de Calidad Abstractos: 1.5: Cambios en el entorno operativo: Cambio de un entorno estructurado a otro entorno NO estructurado

Total Modificaciones: Ninguna.

Mecanismos Arquitecturales:

Encapsulación de las propiedades estructurales del entorno en **`CinServer`** y **`GrpServer`** u otros servidores que pudieran definirse.

Realización: Actualizar bases de datos de la aplicación.

Observaciones:

Los servidores gráficos y cinemáticos pueden representar fielmente a los mecanismos en su entorno siempre que dicho entorno sea estructurado. (Casi siempre basta con alguna operación de calibración para ajustar sus respuestas al entorno real)

Para que sus servicios sigan siendo útiles en un entorno no estructurado, es necesario completarlos con otro tipo de utilidades, por ejemplo de visión artificial, que permitan reconocer el entorno on-line.

Dichas utilidades podrían incluirse integrándolas en **`CinServer`** y/o **`GrpServer`** o de forma independiente, sustituyendo en algunos casos a estos últimos. En ambos casos, la arquitectura puede incluirlos sin grandes dificultades (Ver escenario []).

Sin embargo, hay algunos aspectos que habría que responder para analizar el impacto de su inclusión desde otros puntos de vista, en especial respecto de su influencia en el rendimiento. Por ejemplo, y centrándonos en el caso de las utilidades de visión artificial, la principal pregunta a responder sería:

- ¿Se pretende que estas utilidades sean únicamente una ayuda para el operador o deben proporcionar servicios a **`RemoteCtlr`**?

Escenario FD1 : Fallo del enlace de comunicaciones. Se recupera el enlace o se informa al controlador para que tome las medidas definidas en la especificación del sistema.

Atributos de Calidad Abstractos: Supervisión Estado Sistema/ Fail Safe.

Mecanismos Arquitecturales:

- Servicios de Comunicación encapsulados en **`CommSys`**.
- Uso de protocolos orientados a conexión.
- Redundancia en los mensajes (paridad, CRCs, etc)
- En última instancia es **`RemoteCtlr`** el encargado de detectar y recuperar el fallo. Si **`RemoteCtlr`** no es capaz de hacerlo la responsabilidad pasa al operador.

Realización:

- La realización de este escenario depende de la organización, capacidades e implementación de CommSys.
- Si CommSys falla se realiza el escenario D[]

Escenario FD2: Falla la interfaz de Usuario. Está disponible una interfaz degradada con comandos básicos.

Atributos de Calidad Abstractos: 4.2: Modos de operación degradados.

Mecanismos Arquitecturales:

- En última instancia es **`RemoteCtlr`** el encargado de detectar y recuperar el fallo. Si **`RemoteCtlr`** no es capaz de hacerlo la responsabilidad pasa al operador.
- No hay un mecanismo claro para detectar fallos en **`RemoteUI`** ni para informar a **`RemoteCtlr`** o al usuario:
- No hay un mecanismo claro para cambiar una interfaz por otra en tiempo de ejecución.
- El desacoplo entre **`RemoteUI`** y **`RemoteCtlr`** facilita, no obstante la definición de dichos mecanismos.
  - Pueden definirse dos interfaces, simultáneamente activas, utilizando los mismos o (preferiblemente) distintos componentes de desacoplo.
  - Aunque no está definido explícitamente en la arquitectura es fácil incluir en las interfaces de los subsistemas métodos que permitan tests periódicos de los mismos.

Realización:

Si falla la interfaz de usuario, los únicos actores que pueden darse cuenta son **`RemoteCtlr`** y el usuario operador.

- **Operador:**
  - Según el tipo de fallo de la interfaz, el operador puede no percatarse del mismo hasta que intente introducir un nuevo comando.
  - Si el fallo no se localiza en la introducción de comandos, sino en la actualización del estado, el usuario podría no percatarse nunca, o demasiado tarde (no ha sido informado a tiempo de una alarma o de una condición potencialmente peligrosa).
  - Es necesario que algún componente detecte el fallo de la interfaz e informe al usuario a través de una interfaz alternativa (si la actual falla no es un medio seguro para informar al usuario).
- **`RemoteCtlr`**
  - **`RemoteCtlr`** puede detectar y tratar aquellos fallos de la interfaz de usuario que tengan como consecuencia un error en la invocación de los métodos de la interfaz **`UIControls`** (a través de las excepciones lanzadas por dichas operaciones, de sus valores de retorno o de *timeouts* asociados a la respuesta).
    - El lanzamiento de excepciones está explícitamente contemplado en la arquitectura. Se supone que todas las excepciones que no pueden tratarse en ningún otro componente son recogidas por **`RemoteCtlr`**.
  - Para que **`RemoteCtlr`** pudiera informar al usuario, o bien tendría que existir en tiempo de ejecución una interfaz alternativa o bien tendría que poder crear él mismo una ventana de advertencia.
    - La interfaz alternativa no se contempla explícitamente en la arquitectura, pero tampoco se descarta.
    - La capacidad de crear ventanas es un detalle de implementación del controlador (No obstante la dificultad que implique tal implementación sí depende de aspectos arquitecturales).

Concluyendo:

1. La detección de fallos en la intefaz de Usuario no está completamente resuelta.
2. No se definen mecanismos explícitos para la puesta en funcionamiento de una interfaz degradada en tiempo de ejecución.
   No es difícil ejecutar simultáneamente dos interfaces, la original y una degradada de reserva, aunque no se definen los mecanismos de interacción de esta interfaz con **`RemoteCtlr`** (¿utilizarían los mismos o diferentes componentes de desacoplo?, ¿Se actualizaría siempre o sólo cuando falla la original? ¿Intercambiaría periódicamente información con **`RemoteCtlr`** para indicarle que está activa? etc)
3. La centralización en **`RemoteCtlr`** del tratamiento de los fallos favorece la aparición de modos comunes.
4. Puesto que no hay mecanismos claros, no es posible hacer un estudio formal de la disponibilidad.

Escenario FD4: Falla el controlador. Se activa parada segura.

Atributos de Calidad Abstractos: Supervisión Estado Sistema/ Fail Safe.

Mecanismos Arquitecturales:

- En última instancia es **`RemoteCtlr`** el encargado de detectar y recuperar los fallos.
- No hay ningún mecanismo explícito que permita detectar o recuperar fallos de **`RemoteCtlr.`**
- No se definen mecanismos para incluir un controlador con una funcionalidad degradada.

Realización:

Si falla **`RemoteCtlr`** el único actor que puede detectarlo es el operador.

- Aunque el operador puede detectar fallos en el controlador, no siempre podrá distinguirlos de fallos en la interfaz de usuario.
- Si el operador detecta un mal funcionamiento puede pulsar la seta de emergencia. Este comportamiento puede ser aceptable para algunos sistemas, pero no en todos.
- No se define un controlador alternativo (modo degradado) ni la forma de incluirlo.

Concluyendo:

1. La detección de fallos en el controlador no está resuelta.
2. No se definen mecanismos explícitos para la puesta en funcionamiento de un controlador degradado en tiempo de ejecución.
3. La centralización en **`RemoteCtlr`** del tratamiento de los fallos favorece la aparición de modos comunes.

Escenario FD5: : Falla un subsistema distinto del controlador. Se detecta el fallo y se toman las medidas definidas en la especificación delsistema.

Atributos de Calidad Abstractos: Supervisión Estado Sistema/ Fail Safe.

Mecanismos Arquitecturales:

- En última instancia es **`RemoteCtlr`** el encargado de detectar y recuperar los fallos.
- Todos los subsistemas interaccionan de alguna manera con **`RemoteCtlr`**.
- Aunque no está definido explícitamente en la arquitectura es fácil incluir en las interfaces de los subsistemas métodos que permitan tests periódicos de los mismos.

Realización:

- **`RemoteCtlr`** puede detectar y tratar aquellos fallos que tengan como consecuencia un error en la invocación de los métodos de las interfaces que llama. (a través de las excepciones lanzadas por dichas operaciones, de sus valores de retorno o de *timeouts* asociados a la respuesta).

  El lanzamiento de excepciones está explícitamente contemplado en la arquitectura. Se supone que todas las excepciones que no pueden tratarse en ningún otro componente son recogidas por **`RemoteCtlr`**.
- Pueden incluirse métodos en las interfaces de los subsistemas que permitan su test periódico (habría que ver que consecuencias tiene en el rendimiento).
- Puede establecerse algún protocolo de intercambio de información que permita detectar fallos dentro de un plazo.

Concluyendo:

1. **`RemoteCtlr`** puede detectar fallos en cualquiera de los subsistemas. Aunque no se definen explícitamente en la arquitectura es fácil añadir mecanismos de testeo periódico de los subsistemas, cuya influencia en el rendimiento debe ser estudiada.
2. No se definen mecanismos explícitos de recuperación o de paso a modo degradado.
3. La centralización en **`RemoteCtlr`** del tratamiento de los fallos favorece la aparición de modos comunes.