Tema 7. Metodología para el desarrollo de sistemas en tiempo real

Introducción

Los sistemas en tiempo real son aplicaciones dedicadas, es decir, se realiza un desarrollo específico para cada aplicación. Este desarrollo debe satisfacer una serie de objetivos:

- Cumplir las necesidades del cliente que vienen dadas en forma de especificaciones.
- Respetar los plazos y los costos.
- Alcanzar determinados niveles de calidad.

Pero a lo largo del proceso surgen una serie de dificultades que es preciso superar. En las primeras fases es necesario adquirir una serie de conocimientos y dominar determinadas técnicas relacionadas fundamentalmente con la electrónica y la informática. Además el diseñador necesita conocer exactamente el problema y especificar al máximo su funcionalidad, su alcance y las restricciones que puedan existir. Por otro lado el proyecto será realizado por un conjunto de personas, lo que origina problemas de comunicación, administración de recursos, etc. El resultado final dependerá de la conjunción de estos factores. La gestión y el desarrollo se hacen basándose en la experiencia del director de proyecto y de los componentes del mismo aplicando alguna metodología.

Las metodologías son métodos formales que permiten pasar eficazmente del problema a la definición de la solución. Estas tratan de garantizar la obtención del resultado, determinar la factibilidad de la aplicación (tanto en coste económico y temporal como técnicamente), aumentar la productividad de los diseñadores, conseguir una mayor calidad del producto y permitir una gestión y organización eficiente del conjunto del proyecto.

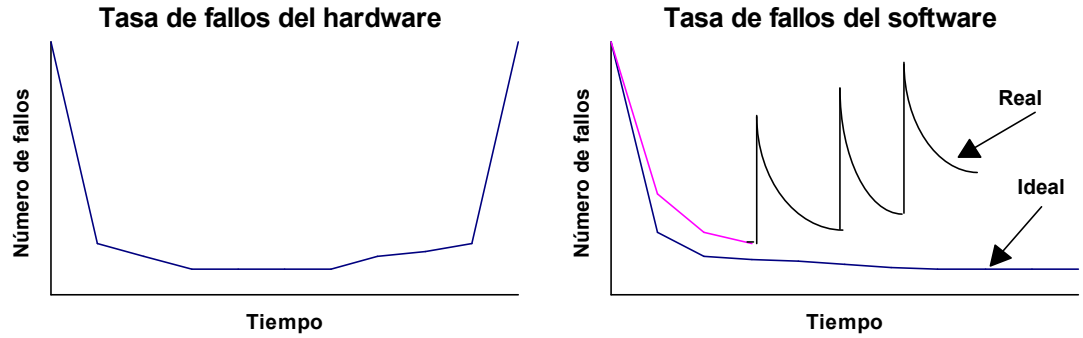
Metodologías y herramientas

En los primeros días de desarrollo del software los gestores del proyecto se centraban exclusivamente en el hardware. Este era el elemento más costoso del proyecto, por lo que se crearon técnicas de desarrollo y control formales, así como estándares técnicos. El software solo era un añadido que se veía en cierto modo como un arte. Esta falta de planificación originaba considerables retrasos en finalizar el software y los costes finales eran más elevados de lo previsto. Por otro lado una gran parte de los errores no se detectaban antes de la entrega a los clientes con la consiguiente insatisfacción de éstos. La calidad del software resultante era normalmente muy cuestionable, y éste era muy difícil de mantener. Todo ello resultaba fruto de una planificación imprecisa o inexistente, unida por otro lado, a una baja productividad de los programadores.

A esto hay que añadir la particular naturaleza del software. Este es un elemento lógico en vez de físico, otras características que lo diferencian del hardware en cuanto a su construcción son:

- El software se desarrolla, no se fabrica. Ambas actividades son similares y tratan de conseguir un producto con una calidad aceptable, pero el coste del software se centra en la ingeniería mientras que el del hardware se acentúa en la producción. Evidentemente el precio de la realización de copias de un programa ya construido es despreciable frente al coste de desarrollo. En el caso del hardware los costes de fabricación suelen superar con creces a los de concepción.
- El software no se deteriora físicamente, no se rompe. En el desarrollo y vida del hardware la tasa de fallos disminuye con el tiempo a medida que estos se van corrigiendo. Finalmente se alcanza una fase terminal donde por los efectos acumulativos de la suciedad, vibraciones, etc. o como resultas de un accidente, o sencillamente porque se hace obsoleto, el hardware se hace inservible. En el software, igualmente hacen su

aparición los fallos a lo largo del tiempo. La corrección de estos o las reformas para dotar de nuevas funciones a los programas introducen a su vez nuevos fallos que serán subsanados posteriormente. Este proceso repetitivo origina una serie de picos en la cantidad de errores a lo largo del tiempo, pero lejos de disminuir el nivel medio tiende a incrementarse, fundamentalmente por el aumento de complejidad del programa, hasta que, como el hardware, se hace inservible. Por otro lado cuando un componente hardware se estropea se puede sustituir con una pieza de repuesto. Esta situación no es posible en el software, evidentemente no hay pieza de repuesto para él.



- La mayor parte del software se construye a medida en vez de ensamblar componentes existentes. El uso de estos componentes con una interfaz bien definido, que normalmente se ajusta a un estándar, y con funciones también completamente definidas simplifica notablemente la tarea de concepción. Sin embargo no existe algo similar a estos componentes para el desarrollo del software, aunque esta situación está cambiando rápidamente desde la aparición de técnicas de programación orientadas a objetos.

El origen de los problemas anteriormente citados está en la falta de un plan de trabajo racional. Con la creación de las metodologías de desarrollo se intentó poner freno a esta situación. Una metodología es algo más que una simple distribución del trabajo en etapas. Es, según el Software Engineering Institute, un conjunto de métodos integrados para alcanzar una meta común, en nuestro caso el desarrollo de un sistema en tiempo real.

Esto nos lleva a la definición de método. Según el diccionario de la lengua española un método es un "modo de decir o hacer con orden una cosa" o un "modo de obrar o proceder" o también un "procedimiento, analítico o sintético, usado para razonar". Nosotros entenderemos por método un conjunto de reglas bien definidas que partiendo de un problema conducen a una solución concreta. Los métodos usados en el desarrollo de hardware y software tienen dos componentes:

- Un modelo, que será la forma de representar el problema. Este es por tanto una interpretación explícita de la comprensión de la situación que tiene el diseñador. Normalmente se representa gráficamente por medio de entidades y relaciones entre ellas, aunque también se emplean símbolos, palabras o notaciones matemáticas.
- Un conjunto de técnicas que indican cómo crear los modelos y transformarlos en otros o en una realización física.

Una herramienta, para el desarrollo de software, es un programa informático o un conjunto de ellos que permiten automatizar uno o varios métodos. Estas tienen su origen en otras herramientas de automatización como las de CAD, CAM y CAE ampliamente usadas en otros sectores de la industria. CASE, acrónimo de Computer Aided Software Engineering, es el nombre con que se conocen estas herramientas. Habitualmente se dividen en dos grandes grupos: UPPERCASE, para las primeras etapas del desarrollo como el análisis y el diseño, y LOWERCASE, para las etapas finales como la generación de código. Habitualmente se encuentran integradas en un mismo paquete al que se llama TOOLSET o TOOLKIT.

El número y funciones de estas herramientas es bastante elevado, no obstante no suelen estar disponibles para el usuario normal, sino que su comercialización se realiza casi exclusivamente en el ámbito empresarial o gubernativo. La división que se expone a continuación no pretende ser exhaustiva, sino que trata de ser una forma más o menos ordenada de presentar algunas de estas herramientas. Así, dividiéndolas según su función, tenemos:

- Herramientas de soporte
- Herramientas de documentación. Fundamentalmente sistemas de autoedición. Son muy importantes ya que el 20 o el 30 por ciento del tiempo del proyecto se dedica a la creación de la documentación.

- Herramientas para software de sistemas. El desarrollo de un proyecto implica a un grupo más o menos numeroso de personas, lo que origina necesidades de comunicación. Los programas de correo electrónico, boletines electrónicos y otros de comunicación vía computador pertenecen a esta categoría.
- Herramientas de control de calidad. Basan sus medidas fundamentalmente en la inspección y análisis del código fuente.
- Herramientas de bases de datos.
- Herramientas de análisis y diseño. Sirven para crear modelos del sistema que se pretende construir. En él se representan los datos, su flujo entre los distintos procesos y su transformación. Además permiten la evaluación de la calidad de dichos modelos.
- Herramientas de análisis y diseño propiamente dichas
- Herramientas de creación de prototipos y simulación
- Herramientas para el diseño y desarrollo de interfaces. Simplifican bastante esta tarea y aumentan considerablemente la productividad puesto que entre el 50 y el 80 por ciento del código generado en la aplicación final se dedica a la gestión y mantenimiento de la interfaz hombre-máquina. Estas herramientas están adquiriendo aun más importancia con la imposición de estándares para la interfaz con el usuario.
- Herramientas de programación
- Herramientas de codificación convencionales. Estas fueron las únicas disponibles durante casi 30 años. A esta categoría pertenecen los compiladores, editores y depuradores.
- Herramientas de codificación de cuarta generación. Permiten un nivel más alto de abstracción que las anteriores. Entre ellas destacamos los lenguajes de cuarta generación, los generadores automáticos de código y los generadores de aplicaciones.
- Herramientas de programación orientadas a objetos.
- Herramientas de integración y prueba
- Herramientas de adquisición de datos. Estos datos serán usados durante las pruebas.
- Herramientas de medida estática. Analizan el código sin ejecutar los casos de prueba.
- Herramientas de medida dinámica. Analizan el código fuente durante la ejecución.
- Herramientas de simulación.
- Herramientas de mantenimiento
- Herramientas de ingeniería inversa. Toman el código fuente como entrada y generan modelos de análisis y diseño.
- Herramientas de reingeniería. Realizan reestructuraciones del código fuente o de los datos.

Estas herramientas suelen estar dotadas de mecanismos que permiten su intercomunicación, facilitando el desarrollo enormemente en todas sus fases. No obstante solo son una ayuda, es necesario conocer los métodos y saber aplicarlos.

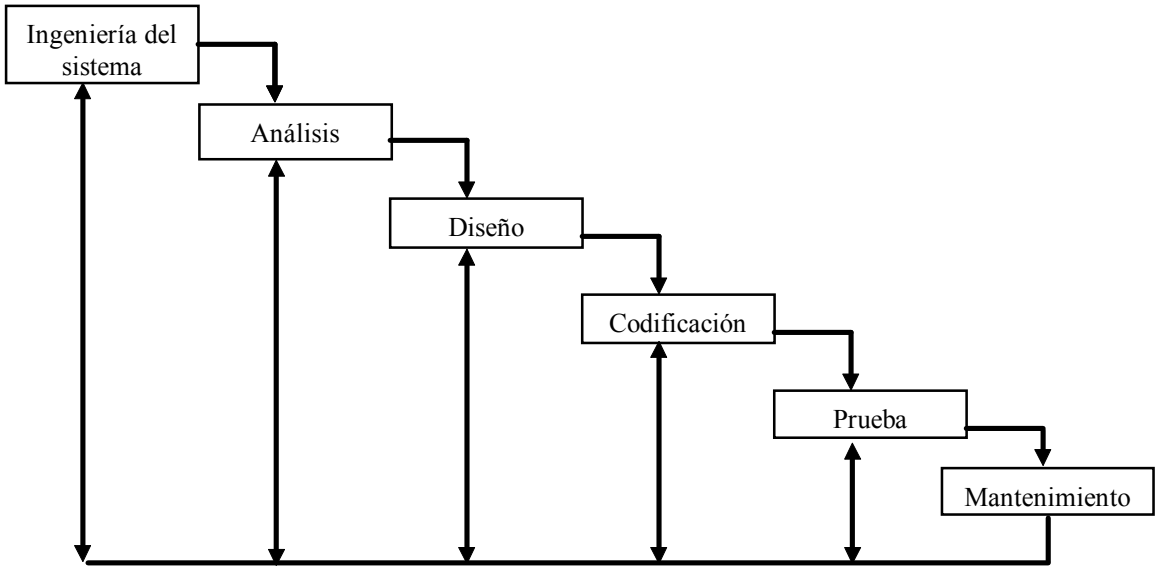
Paradigmas de la ingeniería de software

Además de una modelización del problema y una serie de técnicas para la realización, las metodologías suelen prescribir una división del trabajo en etapas, facilitando la organización de los recursos y un control más fácil del desarrollo. De los muchos modelos que se han propuesto, a continuación describimos los más característicos.

Ciclo de vida clásico

El ciclo de vida clásico o modelo en cascada es el más antiguo y empleado, tiene sus orígenes en las primeras metodología para el desarrollo de hardware. Es un enfoque sistemático y secuencial del desarrollo, que lo divide en una serie de etapas. Según Boehm estas son: especificación, concepción preliminar, concepción detallada y

realización, que concluyen con la explotación y el mantenimiento. Enfocando este esquema al desarrollo del software Pressman propone las siguientes:



- Ingeniería del sistema. Se establecen los requisitos del sistema, asignando posteriormente algún subconjunto de estos al software (y al hardware en los sistemas de tiempo real). La obtención de requisitos se realiza interrogando al cliente hasta llegar a comprender cual es la situación actual y lo se que espera del sistema que se desarrollará. Esta tarea no es tan trivial como puede parecer. Pueden surgir multitud de problemas, desde un sistema complejo con una tecnología que en principio no domina el analista, hasta que el cliente no sabe con exactitud lo que realmente quiere.
- Análisis. Se estudian los requisitos obtenidos anteriormente y se determinan las necesidades del software y hardware. Estos requisitos se documentan y se revisan con el cliente para obtener su aprobación. Esta etapa es de gran importancia puesto que en ella se basa todo el desarrollo posterior. Además la expresión formal de las especificaciones suele tener carácter contractual.
- Diseño. Se crean las estructuras de datos, la arquitectura del sistema, las interfaces y los procedimientos de que constara el sistema. Es pues una traducción de los requisitos a una representación software y hardware realizable.
- Codificación. Se traducen a un lenguaje de programación comprensible por la máquina. Si el diseño, en la etapa anterior, es lo suficientemente detallado, esta tarea es prácticamente automática. En el desarrollo de sistemas hardware o sistemas en tiempo real se adquiere el hardware seleccionado o realizan físicamente los prototipos en esta etapa.
- Prueba. Destinada a detectar errores; puede suponer la vuelta atrás para cambiar algunos detalles de etapas precedentes.
- Mantenimiento. Normalmente los sistemas de software, y en menor medida los de hardware, sufrirán modificaciones después de la entrega al cliente. Estas tendrán su origen en la aparición de fallos no detectados en la fase de pruebas o en ampliaciones de la funcionalidad o del rendimiento que el cliente pueda solicitar. Durante el mantenimiento se aplicarán todas las fases anteriores pero partiendo de un sistema que ya está realizado.

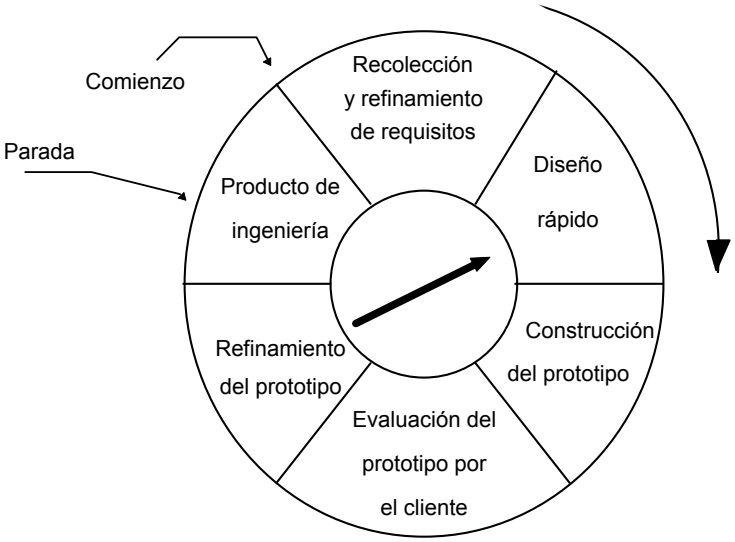
Este modelo es ampliamente criticado por varios motivos:

- Los proyectos reales raramente son tan secuenciales.
- Todos los requisitos deben determinarse en la primera fase, pero a veces es difícil para el cliente establecerlos explícitamente al principio o para el analista comprenderlos exactamente.
- El cliente debe tener paciencia, hasta las etapas finales no se tiene una versión operativa del sistema. La prueba de final por parte del cliente se realiza muy tarde con lo que cualquier deficiencia o error detectado en esta fase puede ser catastrófico y obligar a un replanteamiento completo de todo el sistema.

A pesar estas debilidades este paradigma es el más usado y sus etapas están presentes en mayor o en menor medida en los paradigmas que presentaremos a continuación.

Construcción de prototipos

La construcción de prototipos facilita al desarrollador la creación de un modelo, que puede realizarse teóricamente (en papel) o en un computador (a través de un simulador para el caso de sistemas hardware). Dicho modelo no tiene porqué incorporar toda la funcionalidad deseada o incluso puede emplearse un programa ya existente de capacidades parecidas. Este prototipo se construye después de haber hecho un diseño rápido, que toma como base los objetivos y requisitos aportados por el cliente. Sobre el prototipo y junto con el cliente se refinarán dichos requisitos que servirán de base para otro prototipo. También se usa para determinar la validez de las técnicas de implementación empleadas o para contrastar el rendimiento del sistema. El proceso se detiene cuando se cumplen los requisitos deseados por el cliente.



Dos son las mayores objeciones a este método. Por un lado el cliente ve funcionando el prototipo que toma como primera versión del sistema. Al enterarse de que éste ha de desecharse para iniciar la construcción desde el principio pide los arreglos que sean necesarios para completar el sistema de acuerdo al prototipo. Si se hace así el sistema resultante será de baja calidad. Por otro lado el diseñador, al hacer el prototipo, tomó unas decisiones que si bien eran óptimas para el prototipo (sobre todo con vistas a acortar el tiempo de construcción) no son las mejores para el sistema final.

Estas decisiones, como pueden ser el lenguaje de programación o el modelo de la base de datos, se olvidan con mucha frecuencia debido a la familiaridad que adquiere con ellas el diseñador durante la vida del prototipo. Además el simple hecho de concebir algo que va ha ser desechado produce diseños cuya calidad no es la deseable.

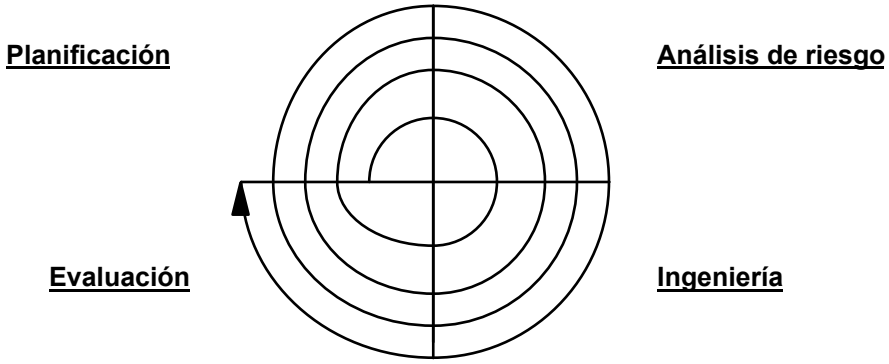
El modelo en espiral

Éste engloba características de los dos paradigmas anteriores y añade un nuevo factor: el análisis de riesgo. Esto lo hace especialmente adecuado para sistemas a gran escala con una fuerte componente de innovación. El mayor handicap de este enfoque es la necesaria habilidad para valorar correctamente el riesgo.

La vida se reparte en cuatro macroetapas que se repiten sucesivamente a lo largo del tiempo. Estas son:

- Planificación. Se determinan los objetivos, técnicas y restricciones.
- Análisis de riesgo y evaluación de alternativas.

- Ingeniería. Esta etapa comprende todas las actividades de un ciclo de vida, como puede ser el clásico o la creación de prototipos, hasta conseguir el producto sea acorde con las especificaciones. Si el análisis de riesgo indica que hay incertidumbre es aconsejable construir prototipos para determinarlos más exactamente.
- Evaluación del cliente. Si éste está de acuerdo con los resultados obtenidos y se verifican todos los requisitos, marcará el final del proyecto. De no ser así se repiten la cuatro etapas.



Hay que hacer notar que en cada vuelta de la espiral la complejidad del sistema y por tanto el tiempo de desarrollo aumentan, especialmente en la parte de ingeniería.

Técnicas de cuarta generación

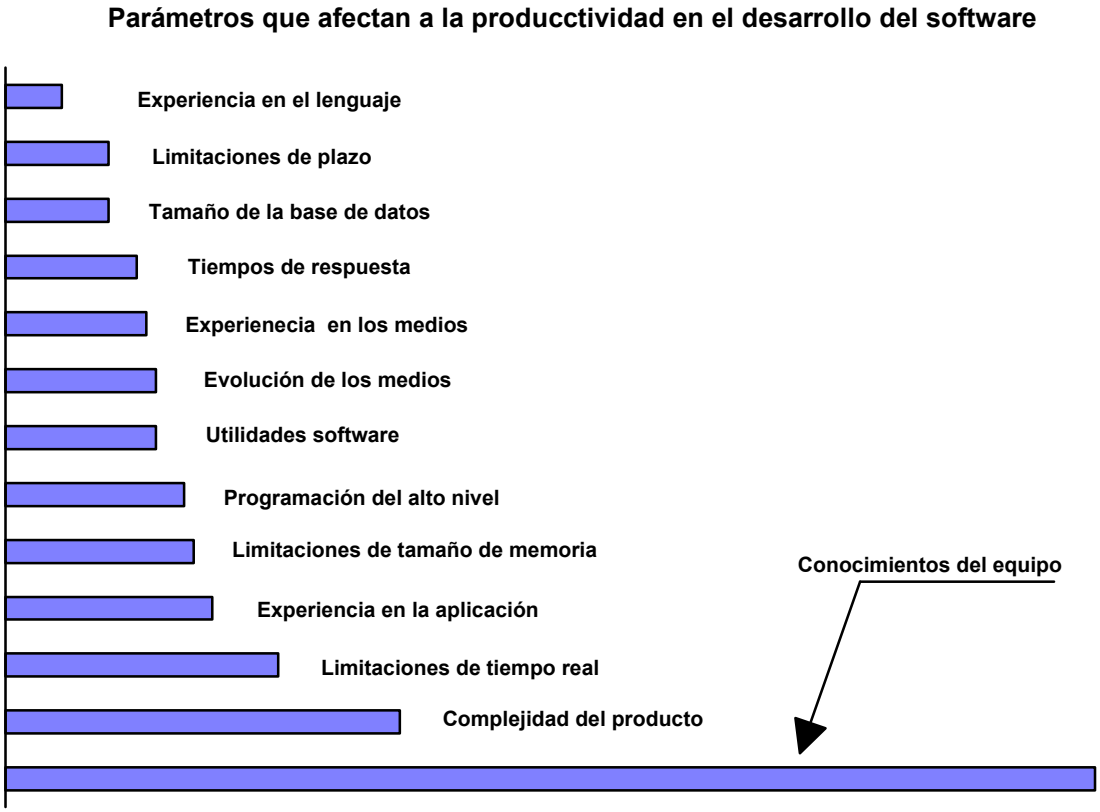
Se usan para producir software basándose en el empleo de varias herramientas que facilitan la generación de código a partir de unas especificaciones muy detalladas. El objetivo de estas incipientes técnicas es especificar el sistema a un nivel próximo al lenguaje natural, objetivo que hoy en día esta muy lejos de ser una realidad. Este paradigma es apropiado para aplicaciones pequeñas pero su uso en proyectos de gran envergadura no es muy apropiado, puesto que se invierte más tiempo en la recolección y presentación exhaustiva de requisitos que el ahorrado en la generación de código. Además la calidad de éste no es todo lo buena que cabría esperar y suele ser bastante difícil el mantenimiento de estos sistemas. Por contra, empleadas adecuadamente estas técnicas reducen notablemente el tiempo de desarrollo y aumentan la productividad del equipo.

Consideraciones adicionales sobre las metodologías y sus ciclos de vida

Los ciclos de vida precedentes describen un conjunto de fases bien definidas y separadas. Esta división en el tiempo rada vez se da, habitualmente existe un solapamiento de las distintas etapas. Es común, por ejemplo, una vez en la fase de diseño volver atrás a retocar una especificación incompleta o ambigua. Otro motivo que origina un salto atrás se debe a que las especificaciones o necesidades del cliente cambian ligeramente a lo largo del proyecto. También es de vital importancia para conseguir un producto de alta calidad comenzar la fase de pruebas lo antes posible, realizando chequeos de cada módulo por separado hasta llegar a las pruebas de integración de todo el sistema.

Es evidente que estos retrocesos suponen un incremento en los plazos de ejecución y consiguientemente en un aumento de los costes. El problema se agrava cuando el retroceso se debe a la aparición de errores no detectados en fases anteriores. Cuanto mayor sea el salto atrás mayor coste originará. Por otro lado, cuanto más avancemos en el desarrollo más se distancian las fases iniciales, con lo que el gravamen temporal y económico se magnifica. Es por tanto muy recomendable la ejecución de cada etapa con la mayor atención y precisión, verificando sus resultados antes de iniciar la siguiente fase.

La productividad es otro factor de gran importancia. Con un valor de esta fijado de antemano y junto con otros indicadores, se calculan los plazos para cada fase. Evidentemente un descenso de esta producirá una demora no esperada en la conclusión del proyecto. Entre los factores que afectan a la productividad, según estudios de Boehm, destaca la competencia del equipo, que está determinada por la formación y experiencia del personal y por la metodología empleada. El dominio y la observación de la metodología permitirá obtener un producto de calidad y fiable, dentro de los plazos y costos previstos. Destaquemos de entre los demás parámetros que aparecen en la lista algunos que tienen una mayor incidencia en los sistemas en tiempo real como los tiempos de respuesta, las limitaciones en el tamaño de la memoria y otras limitaciones de los sistemas en tiempo real.



A lo largo del ciclo de vida del proyecto son necesarios una serie de recursos, tanto humanos como materiales. En un proyecto de software puro los segundos no suelen plantear problemas puesto que las computadoras suelen estar disponibles continuamente. Si se trata de un sistema en tiempo real o de un sistema hardware éstos pueden no estar disponibles (por ejemplo porque son compartidos con otros proyectos) así que será necesaria una planificación de su uso. Los recursos humanos por contra son necesarios a lo largo de todo el ciclo de vida. Para su estimación se emplean las más diversas técnicas. Es muy común asignar el personal para la realización de un producto software mediante la curva de Rayleigh, alcanzando el máximo en el instante de la primera entrega del producto con funcionalidad completa. La distribución y gestión de estos recursos se ve facilitada cuando se sigue una metodología. Tradicionalmente se siguen dos enfoques distintos. El primero, el más tradicional, concentra el máximo nivel de recursos durante la realización física en detrimento de las etapas previas de análisis y diseño, ya que al ser menos visibles son consideradas como improductivas. Por contra si el esfuerzo se maximiza en estas primeras etapas conseguiremos reducir el tiempo de las posteriores, mejorando, además, la calidad del producto final. Actualmente este segundo enfoque se considera el más acertado.

Principales tipos de metodologías

Todas las metodologías persiguen el mismo fin: transformar una necesidad o una idea en un producto (un paquete de software o un sistema en tiempo real en nuestro caso), pero aplican distintas técnicas (fundamentalmente en la fase de análisis) para la consecución del objetivo. Estos diferentes enfoques tiene puntos en común, por

ejemplo la mayor parte se realizan de forma descendente (Top-Down o de lo general a lo particular), pero tienen diferencias que nos permiten clasificarlas en varias grupos. De entre estos, los tres de mayor importancia se describen someramente a continuación.

Desarrollo estructurado

A este grupo pertenece el mayor número de metodologías y las más empleadas en la actualidad. Es el enfoque de desarrollo más antiguo y estudiado y ha servido como base para las demás. Se cimenta en la identificación del origen y el destino de los datos. Determinan también la estructura de estos y sus transformaciones, creando módulos que se encargan de llevar a cabo estas. Estas metodologías inicialmente no contemplaban el desarrollo de sistemas en tiempo real por lo que se realizaron ampliaciones para dar cabida a estos. La metodología que se presenta con más profundidad en secciones posteriores pertenece a esta clase, por lo que no nos extenderemos aquí en la descripción de sus características.

Desarrollo orientado a objetos

En el desarrollo estructurado las distintas áreas funcionales se agrupan juntas en unidades de transformación de datos. Desde un punto de vista orientado a objetos los datos y las funciones que se aplican sobre ellos se agrupan en unidades denominadas objetos. Los objetos se intentan encapsular minimizando las dependencias entre ellos permitiendo que sean fáciles de depurar por separado, igualmente fáciles de mantener y hacer posible su reutilización en otras aplicaciones (nos acercamos a una realización del software similar a la del hardware, basta con unir componentes discretos). Estos son los principales argumentos a favor de esta metodología, pero en la práctica la escasez de arquitecturas (tanto de software como de hardware) orientadas a objetos provocan que los resultados obtenidos con esta técnica en la mayoría de los casos tengan que ser rehechos para su implantación. Aun existen muchas lagunas en las metodologías orientadas a objetos. Por ejemplo no hay un método claro que permita pasar un conjunto de especificaciones a un conjunto de objetos, realizándose en la práctica respetando un conjunto mínimo de reglas y dejando el resto a la intuición y experiencia del desarrollador. De todos modos prolifera este tipo de metodologías intentando dotarlas de elementos que permitan la concepción de sistemas en tiempo real.

Técnicas formales

Estos métodos basan el modelado del sistema en términos de matemáticas y lógica, en vez de en diagramas. Estos pueden ser verificados rigurosamente para probar su corrección y consistencia durante los procesos de análisis y diseño, lo cual permite una generación de código (o una realización física en su caso) extremadamente exacta. Los dos métodos formales más conocidos son el Z y el VDM (Vienna Development Method). Pero estas ventajas tienen su contrapartida en algunas dificultades adicionales:

- Las técnicas formales no son fáciles de usar. Empleando una expresión típica de la jerga informática, no son "amigables con el usuario". No es razonable esperar que el cliente entienda estas representaciones cuando se le entreguen las especificaciones para solicitar su conformidad. Incluso muchos técnicos encuentran dificultad para producir y comprender las especificaciones formales.
- No existe un método detallado para generar especificaciones formales partiendo de los requisitos del cliente. Por otra parte una vez formalizadas estas especificaciones, se traducen directamente a código refinándolas allí donde sea necesario sin tomar en consideración la particular arquitectura del software o del hardware.

Las técnicas formales al igual que las orientadas a objetos aun carecen de un método para modelar el control, un aspecto que resulta crucial en los sistemas de tiempo real.

En este tema nos centraremos en primer lugar en una metodología de diseño, HRT-HOOD, orientada a la construcción de str. Veremos también un método de análisis asociado, HOORA.

7.1 HRT-HOOD (Hard Real-Time Hierarchical Object Oriented Design)

Método de diseño desarrollado en la Universidad de York, orientado a la construcción de sistemas de tiempo real, en particular a los llamados críticos (*hard*). Éstos son sistemas que tienen componentes que, si no producen una respuesta dentro de un intervalo de tiempo prefijado, pueden ocasionar daños importantes.

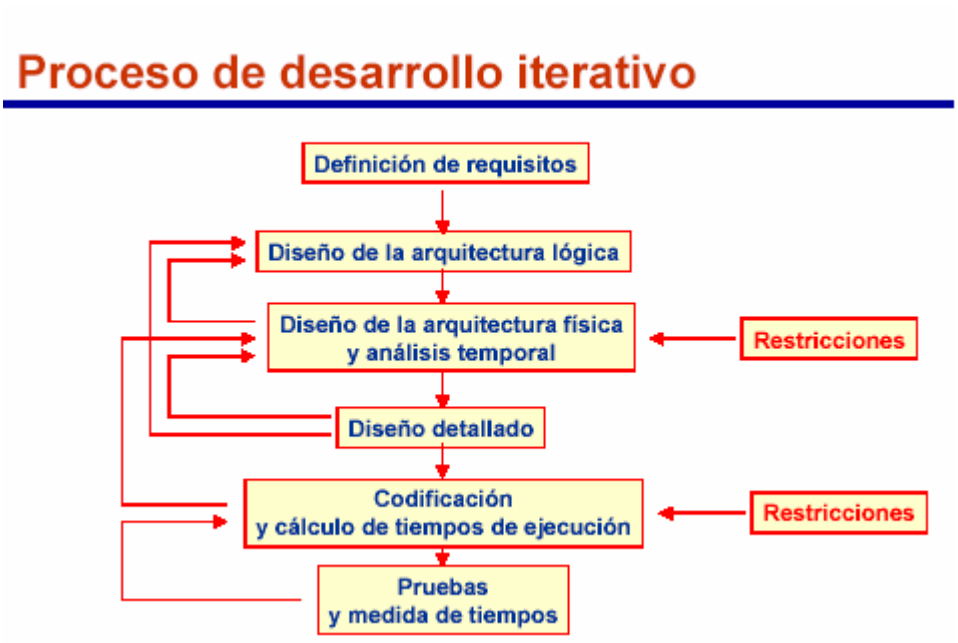
La mayoría de los métodos tradicionales de desarrollo de software utilizan un proceso secuencial. Parten de la definición de requisitos funcionales, siguen con la creación de un modelo arquitectónico para el sistema, luego el diseño detallado, la codificación y finalmente las pruebas del sistema resultante. Los requisitos no funcionales (requisitos de tiempos de respuesta, seguridad, fiabilidad) se comprueban en las etapas finales del desarrollo. Este modo de trabajo presenta un claro inconveniente: muchos errores se encuentran sólo al final, y volver atrás es muy costoso. En HRT-HOOD se utiliza un proceso de desarrollo iterativo.

El diseño se realiza progresivamente a través de la especificación de más responsabilidades cada vez. Las responsabilidades definen las propiedades del sistema con las que opera el diseñador y que éste no tiene libertad de cambiar.

El proceso de refinamiento del diseño, que consiste en transformar las obligaciones (aspectos del diseño que no están sujetos a responsabilidades) en responsabilidades, a menudo se encuentra sujeto a restricciones impuestas por el entorno de ejecución, es decir, el conjunto de componentes hardware y software (procesadores, planificadores, controladores de dispositivos) sobre el que se construye el sistema. Impone restricciones de recursos como la velocidad del procesador, el ancho de banda para las comunicaciones; y restricciones de los mecanismos como la prioridad de las interrupciones, la planificación de tareas y el bloqueo de los datos.

Un método de diseño adecuado para tiempo real debe proporcionar:

- Reconocimiento explícito de los tipos de actividades/objetos que se pueden encontrar en sistemas críticos (por ejemplo, actividades cíclicas y esporádicas).
- Integración de los parámetros apropiados de planificación en el proceso de diseño.
- Definición explícita de los requisitos temporales de cada objeto de la aplicación.
- Definición de la importancia relativa de cada objeto para el funcionamiento correcto de la aplicación.
- Facilidades para realizar el análisis de la planificabilidad del sistema.



Diseño de la arquitectura lógica: Fase destinada fundamentalmente a la satisfacción de los requisitos funcionales. El resultado de esta etapa es un conjunto de objetos terminales (que no requieren mayor descomposición).

Objetivos de la arquitectura física:

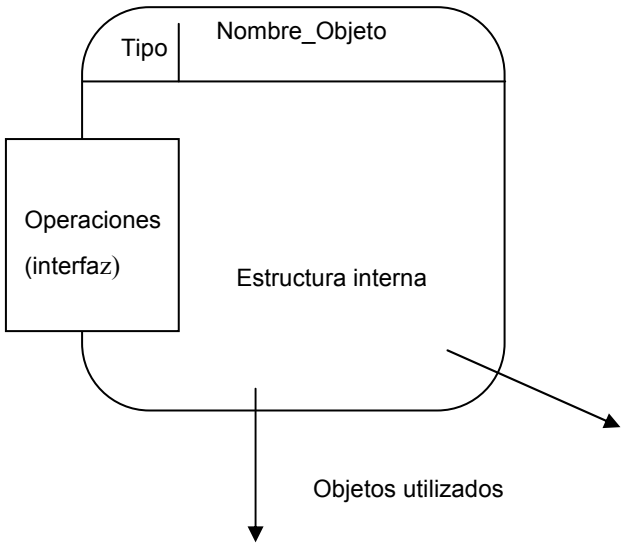
- Relacionar la arquitectura lógica con los recursos de ejecución.
- Asignar atributos temporales a los objetos y asegurar el cumplimiento de los requisitos no funcionales:
- Análisis de plazos de respuesta
- Fiabilidad y seguridad.

En el diseño de la arquitectura física se consideran principalmente 4 actividades:

- Asociación de los objetos resultantes de la arquitectura lógica a los recursos hardware (procesadores) disponibles.
- Planificación de la red. Si el sistema está distribuido, hay que planificar las comunicaciones entre los nodos, de forma que el tiempo de retardo en las comunicaciones esté acotado.
- Planificación de los procesadores. Determinar la planificación, estática o dinámica, que asegure que todas las tareas del sistema cumplan con los plazos prefijados.
- Análisis de la fiabilidad: determinar si es conveniente emplear técnicas de tolerancia de fallos, para aumentar la fiabilidad del sistema.

El resultado del modelo físico es un sistema de objeto terminales con atributos temporales. Aunque la arquitectura física es un refinamiento de la arquitectura lógica, el desarrollo de ambas etapas será iterativo y concurrente.

7.2 TIPOS DE OBJETOS EN HRT-HOOD



Tipo: P (pasivo), Pr (protegido), A (activo), C (cíclico), S (esporádico).

PASIVOS

- No controlan cuándo se ejecutan sus operaciones. Cuando se invoca una operación, el control se transfiere inmediatamente a esa operación.
- No invocan operaciones de otros objetos espontáneamente.
- Sólo contienen código secuencial.

PROTEGIDOS

- Pueden controlar cuándo se ejecutan sus operaciones.
- No invocan operaciones en otros objetos espontáneamente.
- Se utilizan para controlar el acceso a recursos compartidos por varios objetos activos.

ACTIVOS

- Pueden controlar cuándo se ejecutan sus operaciones.
- Pueden invocar operaciones en otros objetos espontáneamente.

CÍCLICOS

- Representan actividades periódicas.
- Sólo pueden incluir operaciones que demanden atención inmediata (**ATC**: transferencias asíncronas de control). Muchos objetos cíclicos no tienen operaciones

ESPORÁDICOS

- Representan actividades esporádicas. Incluyen UNA operación START y pueden incluir una o más ATC.
- Los objetos cíclicos y esporádicos tienen **1 thread** (actividad concurrente) que se ejecuta independientemente de las operaciones.
- Las ATC se utilizan para obtener una atención inmediata del thread.
- **OBCS** (object control structure). Controla la ejecución de las operaciones del objeto..

Los objetos pasivos no tienen OBCS ni thread

Los objetos protegidos tienen 1 OBCS

Los objetos cíclicos tienen 1 thread, y, si tienen operaciones, 1 OBCS

Los objetos esporádicos tienen 1 thread y 1 OBCS

7.3 ATRIBUTOS DE TIEMPO REAL

Algunos de los principales atributos de un objeto son:

- DEADLINE. Cada objeto cíclico y esporádico puede tener un tiempo límite definido para la ejecución de su thread.
- OPERATION_WCET. Cada operación visible debe tener definido un tiempo de ejecución máximo.
- THREAD_WCET. Cada objeto cíclico y esporádico visible debe tener definido un tiempo de ejecución máximo para su thread.
- PERIOD. Cada objeto cíclico debe tener definido un periodo de ejecución.
- OFFSET. Cada objeto cíclico puede tener definido un *offset* que indica el tiempo que el thread debe esperar antes de iniciar sus operaciones cíclicas.
- MINIMUM_ARRIVAL_TIME o MAXIMUM_ARRIVAL_FREQUENCY. Cada objeto esporádico debe tener definido uno de estos dos atributos, que indican el tiempo mínimo o la frecuencia máxima entre dos peticiones de activación.
- PRIORITY. Cada objeto cíclico y esporádico puede tener una prioridad definida para su thread.

7.4 OPERACIONES DE LOS OBJETOS HRT-HOOD

7.4.1 TIPOS DE OPERACIONES DE UN OBJETO PROTEGIDO

Un objeto protegido tiene un OBCS. Las operaciones acceden a los datos internos con exclusión mutua. 2 tipos de restricciones en las operaciones de un objeto protegido:

- PSER. *Protected synchronous execution request*. La actividad del objeto cliente se interrumpe hasta que se completa la ejecución de la operación protegida.
- PAER. *Protected asynchronous execution request*. La actividad del cliente se interrumpe sólo hasta que la petición de la operación ha sido recibida por el objeto protegido..

También pueden tener operaciones sin restricciones, similares a las de los objetos pasivos.

7.4.2 TIPOS DE OPERACIONES DE UN OBJETO ACTIVO

- CONSTRAINED (Con restricciones). Se ejecutan bajo el control del OBCS.
- UNCONSTRAINED (Sin restricciones). Se ejecutan tan pronto son invocadas, similares a las de los pasivos.

Se establecen 2 tipos de restricciones:

a) Funcionales. Restricciones en función del estado interno del objeto. Operación abierta: el estado interno del objeto permite la ejecución de la operación. Ejemplo: objeto BUFFER. Operación PUT abierta si el estado NO es LLENO.

b) Según tipo de invocación. Indican el efecto sobre el objeto que invoca la operación:

- ASER. Asynchronous Execution Request. Cuando se llama a una operación ASER, el objeto que llama no se bloquea.
- LSER. Loosely Synchronous Execution Request. El llamador se bloquea hasta que el objeto llamado está listo para ejecutar la operación.
- HSER. Highly Synchronous Execution Request. El llamador se bloquea hasta que el objeto llamado ha atendido la petición (ha ejecutado la operación).

  Las operaciones LSER y HSER pueden tener asociado un *timeout*.

7.4.3 OPERACIONES DE UN OBJETO CÍCLICO

En general, los objetos cíclicos se comunicarán y sincronizarán con otros threads mediante llamadas a operaciones de objetos protegidos. Sin embargo, puede ser necesario definir algunas operaciones restringidas sobre objetos cíclicos, porque otros objetos pueden necesitar señalar algún cambio de modo o alguna condición de error al objeto cíclico. Podría conseguirse haciendo que el objeto cíclico haga un *polling* a una operación 'notificadora del cambio de modo' o 'notificadora de error' de un objeto protegido, pero esta solución es ineficiente si el tiempo de respuesta del objeto cíclico es pequeño.

Las operaciones restringidas que se definan en el objeto cíclico requerirán una respuesta inmediata del *thread* del objeto. El OBCS del objeto interaccionará con el *thread* para forzar una ATC. Estas operaciones restringidas suelen denominarse ASATC (*Asynchronous asynchronous transfer of control request*), similares a las ASER de los objetos activos, excepto que demandan que el objeto cíclico reaccione de manera inmediata.

7.4.4 OPERACIONES DE UN OBJETO ESPORÁDICO

Cada objeto esporádico tiene una operación restringida, llamada habitualmente START, que se llama para activar la ejecución del *thread*. Es una operación ASER, y puede ser invocada por una interrupción (ASER_BY_IT).

Un objeto esporádico puede además tener otras operaciones restringidas ASATC, iguales a las mencionadas en los objetos cíclicos.

7.5 REGLAS DE DESCOMPOSICIÓN DE LOS OBJETOS

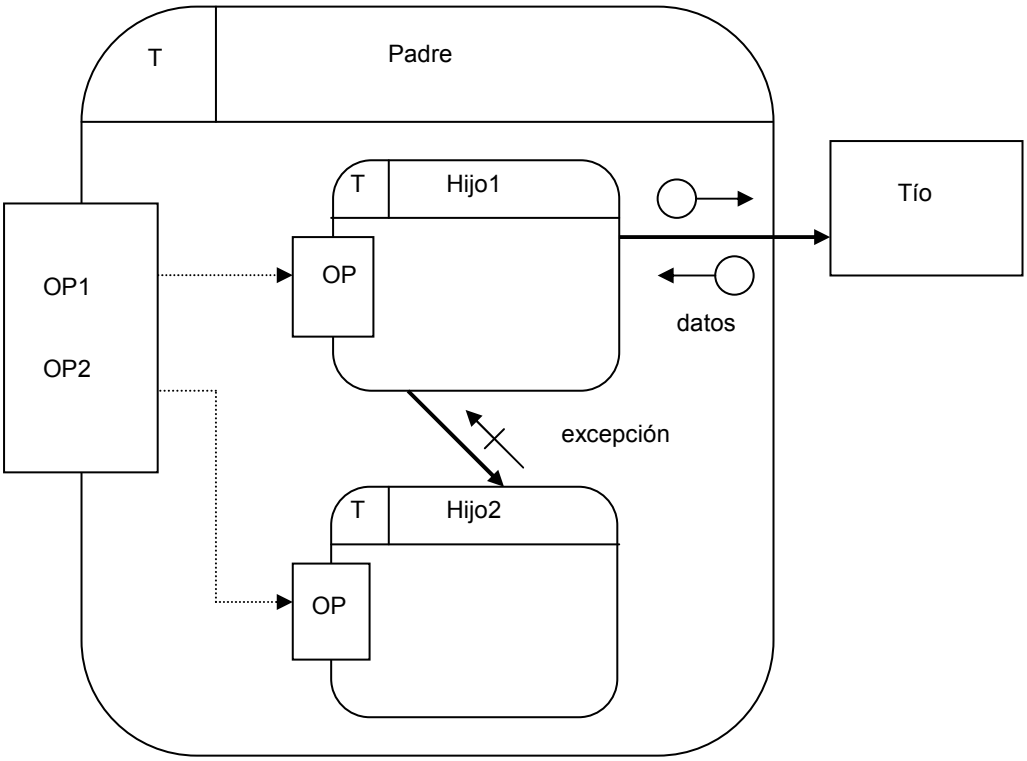
Un objeto (padre) puede incluir uno o más objetos (hijos).

- Un objeto activo puede incluir cualquier objeto. Un objeto activo no puede ser un objeto terminal en HRT-HOOD.
- Un objeto pasivo sólo puede incluir otros objetos pasivos.
- Un objeto protegido puede incluir objetos pasivos y un objeto protegido.
- Un objeto esporádico puede incluir objetos esporádicos, cíclicos, pasivos y protegidos.
- Un objeto cíclico puede incluir objetos cíclicos, esporádicos, pasivos y protegidos.

Una operación restringida de un objeto padre puede ser implementada por una operación restringida de un hijo.

7.6 REGLAS DE USO

- Los objetos pasivos no pueden utilizar (invocar) operaciones restringidas de otros objetos.
- Objetos cíclicos y esporádicos pueden utilizar operaciones de objetos pasivos, protegidos, cíclicos y esporádicos. Pueden utilizar operaciones ASER de objetos activos.
- Los objetos protegidos pueden utilizar operaciones de objetos pasivos y protegidos, operaciones ASATC de objetos cíclicos, operaciones START y ASATC de objetos esporádicos y operaciones ASER de objetos activos.



El objeto padre se descompone en Hijo1 e Hijo2

Padre usa Tío

Hijo1 usa Hijo2 y Tío (Hijo1 es cliente de Hijo2. Hijo2 es servidor de Hijo1)

Hijo1.OP implementa Padre.OP1

Hijo2.OP implementa Padre.OP2

7.7 CODIFICACIÓN EN ADA95

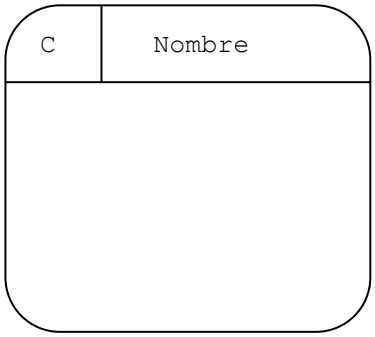
- Cada objeto se realiza mediante un paquete.
- Las relaciones de uso se representan mediante cláusulas with.
- Los objetos hijos se realizan mediante paquetes hijos privados.
- Los objetos cíclicos y esporádicos tienen una tarea, y posiblemente (los esporádicos obligatoriamente) un objeto protegido de sincronización (para el OBCS).
- Objeto protegido: objeto protegido de Ada95.
- Los atributos temporales de cada objeto se definen en un paquete auxiliar.

```
with Tio;

package Padre is
   procedure OP1(…);
   procedure OP2(…);
end Padre;

private package Padre.Hijo1 is
   procedure OP;
end Padre.Hijo1;

private package Padre.Hijo2 is
   procedure OP;
end Padre.Hijo2;

with Padre.Hijo1, Padre.Hijo2;
package body Padre is
   procedure OP1(…) renames Padre.Hijo1.OP;
   procedure OP2(…)  renames Padre.Hijo2.OP;
end Padre;
```

Ejemplo de paquete con atributos temporales:

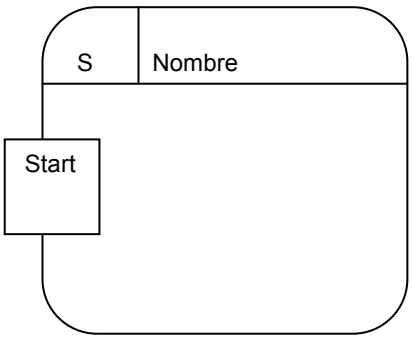
```
with System; use System;
with Ada.real_Time; use Ada.Real_Time;
package Real_Time_Attributes is
   type Operation_Attributes is tagged
   record
      WCET: Time_Span;
   end record;

   type Thread_Attributes is tagged
   record
      P: priority;
      WCET: Time_Span;
      DEADLINE: Time_Span;
   end record;

   type Periodic_Thread_Attributes is
       new Thread_Attributes with
   record
      PERIOD: Time_Span;
      OFFSET: Time_Span;
   end record;
-- otras definiciones
end Real_Time_Attributes;
```

7.7.1 Ejemplo de codificación de un objeto cíclico



```
with Nombre_Atributos;
-- incluye prioridad y periodo
-- del thread (tarea cíclica)
package Nombre is
   ...
   pragma elaborate_body;
end Nombre;

package body Nombre is
   procedure OP_Ciclica(...) is
   begin
     -- acciones periodicas
   end OP_Ciclica;


   task Thread is
      pragma priority(...);
   end Thread;

   task body Thread is
   -- variables locales;
   begin
      ...
      loop
         delay until T;
        OP_Ciclica(..);
       T:=T+Periodo;
   end loop;
   end Thread;
end Nombre;
```

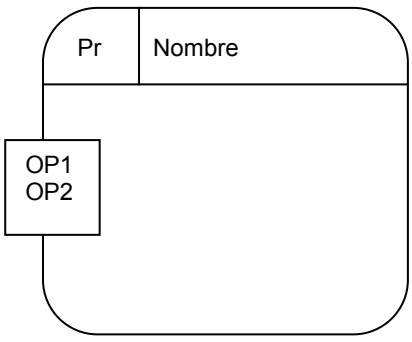
7.7.2 Ejemplo de codificación de un objeto esporádico:




```
with Nombre_Atributos;
   -- incluye prioridad e
   -- intervalo mínimo
   -- del thread
package Nombre is
   ...
   procedure Start(...);
end Nombre;
```

```
package body Nombre is

   procedure OP_Esporadica(...) is
   -- accion esporadica
   task Thread is
      pragma priority(...)
   end Thread;

   protected OBCS is
      entry Wait(...);
      procedure Signal(...);
   private
      Barrera: boolean:=false;
   end OBCS;

   task body Thread is
   -- variables locales;
   begin
      ...
      loop
        OBCS.Wait(...);
        OP_Esporadica(...);
     end loop;
   end Thread;

   protected body OBCS is
   -- variables locales;
     entry Wait(...) when Barrera is
     begin
           ...
           Barrera:=false;
      end Wait;
      procedure Signal(...) is
      begin
         ...
         Barrera:=true;
      end Signal;
   end OBCS;

   procedure Start(...) is
   begin
      OBCS.Signal(...);
   end Start;
end Nombre;
```

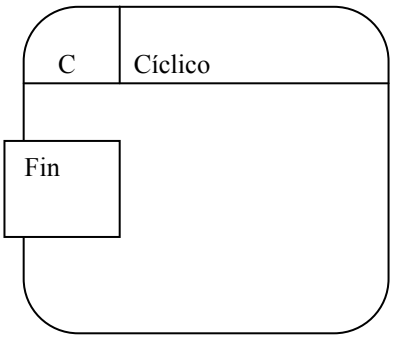
7.7.3 Ejemplo de codificación de un objeto protegido:




```
with Nombre_Atributos;
```

```
package Nombre is
  ...
  procedure OP1(...);
  procedure OP2(...);
end Nombre;

package body Nombre is

   protected Agent is
      pragma priority(...);
      procedure Operacion1;
      procedure Operacion2;
   private
     ...
   end Agent;

   procedure OP1(...) is
   begin
      Agent.Operacion1;
   end OP1;

  procedure OP2(...) is
  begin
    Agent.Operacion2;
  end OP2;


   protected body Agent is
       procedure Operacion1 is
       begin
         -- operacion protegida 1
       end Operacion1;

       procedure Operacion2 is
       begin
         -- operacion protegida 2
       end Operacion2;
   end Agent;
end Nombre;
```

7.7.4 Ejemplo de codificación de un objeto cíclico con una operación (ASATC) visible:

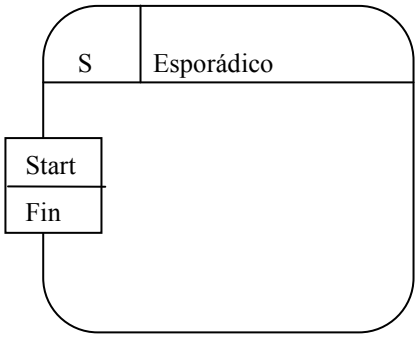



```
-- especificación del paquete con los atributos del objeto cíclico
-- esto vendría a ser el OBCS del objeto (object control structure)
-- ya que los objetos cíclicos tienen un thread y si tienen operaciones,
-- como en este caso, también tienen OBCS
-- ESTE PAQUETE NO TIENE CUERPO, SOLO INCORPORA VALORES DE VARIABLES
with System; use System;
with Ada.Calendar; use Ada.Calendar;
package Atrib_Cic is
```

```
   Prioridad: Priority:=18;
   Periodo: Duration:=5.0;
   Prioridad_OBCS: Priority:=23;
end Atrib_Cic;

-- paquete que incorpora el objeto cíclico
-- usará el paquete que contiene sus atributos

with Atrib_Cic; use Atrib_Cic;
package Ciclico is
  procedure Fin;     -- ésta es la única operación visible del objeto cíclico
                     -- debe ser de tipo ATC, transferencia asíncrona de control
                     -- pues son las únicas toleradas en este tipo de objetos
end Ciclico;


with Ada.Text_IO, Ada.Calendar;
use Ada.Text_IO, Ada.Calendar;
-- este es el cuerpo del paquete que define el objeto cíclico
-- estos objetos tienen un thread y las operaciones que implementan
-- deben ser de tipo ATC, esta será la operación Fin
package body Ciclico is
   procedure OP_Ciclica is   -- esta es un operación interna, no tiene porque ser ATC
   begin
      put_line("tarea cíclica");
   end OP_Ciclica;

   task Thread is
      pragma priority(Atrib_Cic.Prioridad);  -- prioridad está en el paquete OBCS
   end Thread;

   protected OBCS is --mediante esta estructura de control se determinará
   -- bajo que condiciones se ejecutará la tarea ciclica
   -- esta estructura de control no es visible fuera del paquete
   -- solo el objeto cíclico podrá utilizarlo
      entry ACABO;  -- para determinar el final
      procedure FIN;
      pragma Priority(Atrib_Cic.Prioridad_OBCS);
   private
      ACABAR: boolean:=False;  -- llevará el estado
   end OBCS;


   task body Thread is --los objetos cíclicos tienen un thread -> crear tarea
     T:time:=clock;     -- para trabajar con el periodo
   begin
      loop              -- ciclo que se repite hasta acabar
         select         -- ATC
            OBCS.ACABO;
            exit;   --solo se ejecutará cuando llamen a ACABO y se acepte
         then abort     -- parte abortable
            delay until T; -- Estas tres sentencias se irán ejecutando
            OP_Ciclica;  --mientras no se llame a ACABO, cuando se produzca
            T:=T+Atrib_Cic.Periodo; --esta parte se abortará de inmediato
         end select;
      end loop;
   end Thread;

   protected body OBCS is
      entry ACABO when ACABAR is -- solo lo quiero para parar la parte
      begin                      -- abortable del ATC, no tiene que hacer nada
         null;
      end ACABO;
      procedure FIN is    --el objetivo es hacer ACABAR:= TRUE
      begin
         ACABAR:=True;
      end FIN;
   end OBCS;
```

```
   procedure Fin is  -- este procedimiento es visible desde fuera del paquete
                     -- es por tanto una operación del objeto
   begin
      OBCS.FIN;     -- llama a la operación interna del objeto para acabarlo
   end Fin;

end Ciclico;
```

7.7.5 Ejemplo de codificación de un objeto esporádico con una operación añadida (ASATC) y con intervalo de separación mínima entre activaciones:




```
-- especificación del paquete con los atributos del objeto esporádico
-- esto vendría a ser el OBCS del objeto (object control structure)
-- ya que los objetos esporádicos tienen un thread y también tienen OBCS
with System; use System;
package Atrib_Esp is      -- ESTE PAQUETE NO TIENE CUERPO
   Prioridad: Priority:=20;
   Separacion: Duration:=2.0; -- periodo mínimo entre dos ejecuciones
   Prioridad_OBCS: Priority:=23;
end Atrib_Esp;

-- paquete que define el objeto esporádico
-- usará el paquete que contiene sus atributos

with Atrib_Esp; use Atrib_Esp;
package Esporadico is
--presenta dos operaciones Start, obligatoria para todo esporádico
-- y el resto de las que pueda presentar deben ser del tipo  ATC
   procedure Start;
   procedure Fin; --esta deberá ser una llamada ATC
end Esporadico;


-- éste es el cuerpo del paquete que define el objeto esporádico
-- estos objetos tienen un thread y las operaciones que implementan
-- deben ser una operación START y el resto de tipo ATC,
-- esta será la operación Fin
-- recordar que solo START y FIN son visibles en la especificación
with Ada.Text_IO, Ada.Calendar;
use Ada.Text_IO, Ada.Calendar;
package body Esporadico is
   procedure OP_Esporadica is -- la acción que se ejecutará esporádicamente
   begin
      put_line("tarea esporadica");
   end OP_Esporadica;

   task Thread is   -- los objetos esporádicos tienen un  thread -> una tarea
      pragma priority(Atrib_Esp.Prioridad);
   end Thread;

   protected OBCS is            -- estructura de control del objeto
```

```
                              -- es interna al paquete, solo la maneja
                              -- el objeto esporádico
      pragma priority(Atrib_Esp.Prioridad_OBCS);
      entry Wait(Tiempo:out time);
      procedure Signal;         -- para lanzar la acción esporádica
      entry ACABO;              -- para marcar la finalización
      procedure FIN;
   private
      ACTIVACION:boolean:=False;
      ACABAR: boolean:=False;
   end OBCS;

   task body Thread is          -- hilo del objeto esporádico
      Tiempo:time;     --establecerá un intervalo mínimo entre activaciones
   begin
      loop                      -- repetir hasta acabar
        select         -- ATC
            OBCS.ACABO;
            exit;       -- se ejecutará cuando se llame a ACABO y se acepte
         then abort     -- parte abortable
            OBCS.Wait(Tiempo);  -- Estas tres sentencias se irán ejecutando
            OP_Esporadica;      -- mientras nadie llame a OBCS.ACABO, cuando
            delay until Tiempo+Separacion;    -- se produzca la llamada esta
                                               --parte se abortará inmediatamente
         end select;
      end loop;
   end Thread;

   protected body OBCS is
   -- como ya habiamos visto la acción esporádica ejecutará el wait esperando
   -- a que otra tarea ejecute Signal para desbloquearla
      entry Wait(Tiempo:out time) when ACTIVACION is
      begin
         Tiempo:=clock;         -- instante de activación
         ACTIVACION:=False;     -- poner a FALSE para el siguiente bloqueo
      end Wait;

      procedure Signal is       -- TRUE para desbloquear la tarea esporádica
      begin
         ACTIVACION:=True;
      end Signal;

      entry ACABO when ACABAR is -- parará la parte abortable del ATC
      begin                     -- no tiene que hacer nada
         null;
      end ACABO;
      procedure FIN is  --el objetivo de est procedure es hacer ACABAR:= TRUE
      begin             -- ACABAR y FIN son iguales a las del objeto CICLICO
         ACABAR:=True;
      end FIN;
   end OBCS;
   procedure Start is     -- esta parte es visible, está en la especificación
   begin
      OBCS.Signal;           -- desbloquea la tarea
   end Start;

   procedure Fin is
   begin
      OBCS.FIN;                 -- hace que se ejecute el ATC
   end Fin;
end Esporadico;
```

7.7.6 Ejemplo ilustrativo

7.7.6.1 Definición del problema

Queremos desarrollar un RTS que comprende:

a) Una entidad encargada de producir datos a una velocidad constante (p.e., cada 20 ms). PRODUCTOR.

b) Una entidad encargada de consumir esos datos, según petición del Productor. El Productor y el Consumidor tienen que almacenar un registro sobre su actividad en una estructura de datos dedicada a tal fin. CONSUMIDOR.

c) Una entidad encargada de transmitir los contenidos de dicha estructura de datos al operador del sistema, bajo petición del Productor. TRANSMISOR.

Estas entidades tendrán que realizar además otras actividades auxiliares en cada activación.

7.7.6.2 OBJETOS:

BUFFER es un objeto protegido en el que PRODUCTOR almacena datos, y del que CONSUMIDOR los retira. Proporciona 2 operaciones protegidas síncronas: Read y Write.

ESTRUCTURA es un objeto protegido en el que PRODUCTOR y CONSUMIDOR almacenan los registros de su actividad, y del que TRANSMISOR los extrae. Proporciona 2 operaciones protegidas síncronas: Read y Write.

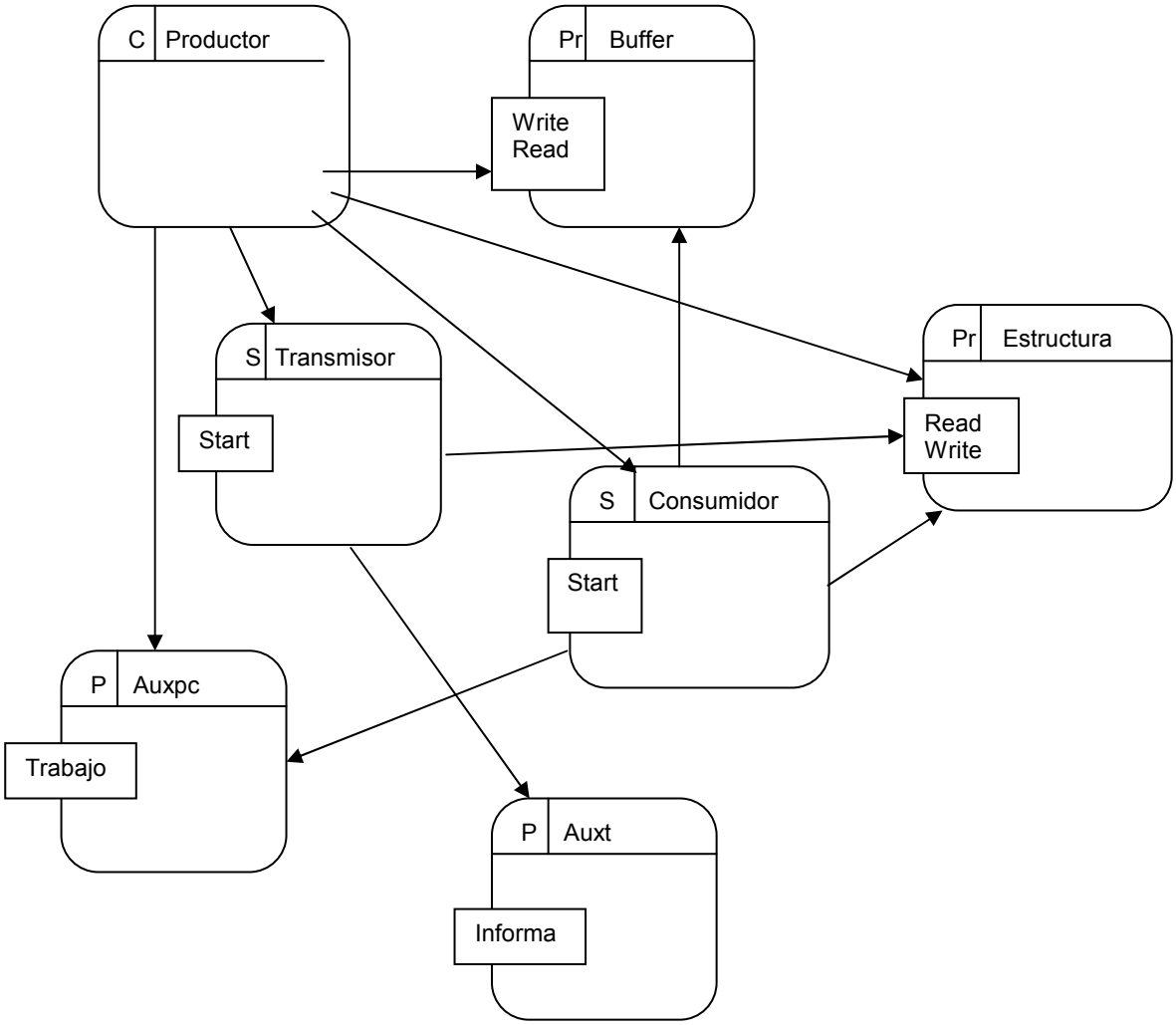
PRODUCTOR es un objeto cíclico. Dispone de 1 thread.

CONSUMIDOR es un objeto esporádico. Dispone de 1 thread y 1 OBCS. Tiene una operación asíncrona Start.

TRANSMISOR: objeto esporádico. Dispone de 1 thread y 1 OBCS.

AUXPC: objeto pasivo que contiene las operaciones auxiliares del PRODUCTOR y del CONSUMIDOR. Le asociamos una operación Trabajo (no restringida).

AUXT: objeto pasivo que contiene las operaciones auxiliares del TRANSMISOR. Le asociamos una operación Informa (no restringida).



Las flechas entre parejas de objetos representan las relaciones de uso.

En cada activación periódica, el objeto PRODUCTOR:

- Deposita un dato en el BUFFER utilizando la operación síncrona Write de éste.
- Realiza una serie de trabajo auxiliar, llamando a la operación Trabajo de AUXPC.
- Llama a la operación síncrona Write de ESTRUCTURA para almacenar información que le interese al operador.
- Activa los objetos CONSUMIDOR y TRANSMISOR, llamando a sus operaciones asíncronas Start.

En cada activación, CONSUMIDOR:

- Lee el dato almacenado en BUFFER, utilizando la operación Read de éste.
- Realiza trabajo auxiliar, llamando a la operación Trabajo de AUXPC.
- Llama a Write de ESTRUCTURA para almacenar información que le interese al operador.

En cada activación, TRANSMISOR:

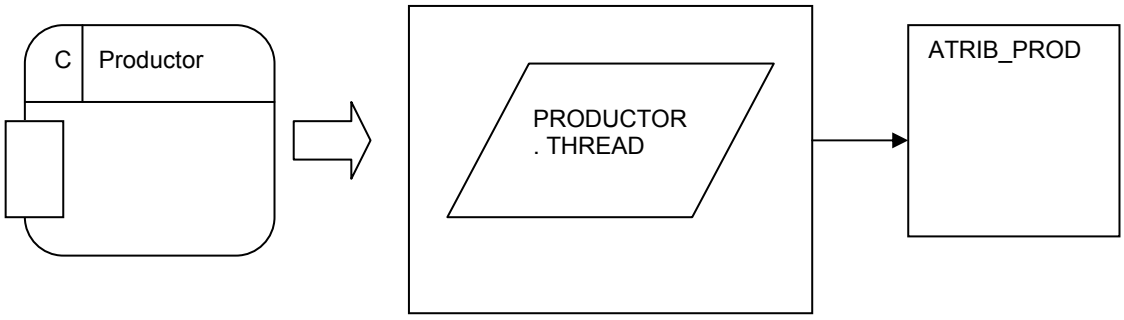
- Llama a la operación síncrona Read de ESTRUCTURA, para leer la información sobre el estado del sistema que le va a presentar al operador.
- Llama a la operación Informa de AUXT, que produce la salida de dicha información hacia el operador.

PRODUCTOR, CONSUMIDOR y TRANSMISOR son objetos activos, por lo que tiene cada uno 1 thread de control. Las operaciones de estos threads estarán sujetas a requisitos temporales específicos.

Supongamos los siguientes requisitos:

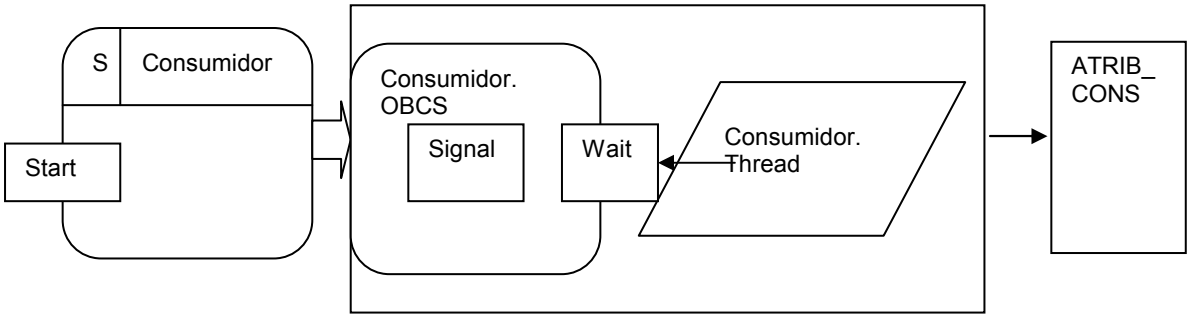
- PRODUCTOR tiene un periodo de activación (T) de 20ms, y un plazo de respuesta (deadline, D) de 9ms.
- CONSUMIDOR tiene un tiempo mínimo entre activaciones igual al periodo del productor, y un plazo que nos asegure que finalice antes del siguiente ciclo del productor. Por tanto, podemos poner su D=17ms.
- TRANSMISOR tendrá un tiempo entre activaciones igual al del consumidor, y un plazo menor que el del productor (D=8ms).

Podemos pasar ahora a realizar la primera codificación en Ada, codificación que podría generarse de manera automática con una herramienta adecuada (similar a STOOD).



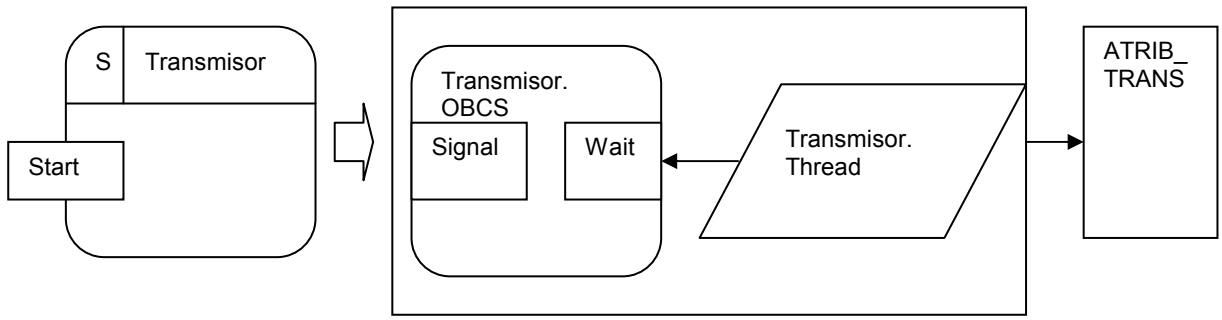

```
with System; use System;
with Ada.Calendar; use Ada.Calendar;
package ATRIB_PROD is
   Prioridad: constant priority := ...;
   Periodo: duration := 0.020;
   Deadline: duration := 0.009;
end ATRIB_PROD;
with System; use System;
with Ada.Calendar; use Ada.Calendar;
with ATRIB_PROD;
package PRODUCTOR is
   pragma elaborate_body;
end PRODUCTOR;
with BUFFER, CONSUMIDOR, TRANSMISOR, AUXPC, ESTRUCTURA, ATRIB_ESTR;
use ATRIB_ESTR;
package body PRODUCTOR is
-- supongamos que escribe un valor entero en el buffer
   Item: integer:=0;
-- y que envia otro valor entero a AUXPC
   AuxP:Integer:=500;
   procedure ACCION_CICLICA;
   task THREAD is
      pragma priority(ATRIB_PROD.Prioridad);
   end THREAD;
   task body THREAD is   -- codigo estandar de una tarea ciclica
      T: time := clock;
   begin
      loop
         delay until T;
         ACCION_CICLICA;
         T:=T+ATRIB_PROD.Periodo;
      end loop;
   end THREAD;
```

```
   procedure ACCION_CICLICA is
      Reg:Registro;
   begin
      BUFFER.write(Item);
      AUXPC.Trabajo(AuxP);
      Reg.Valor:=Item;
      Reg.C:='P';
      Reg.Hora:=clock;
      ESTRUCTURA.Write(Reg);
      CONSUMIDOR.Start;
      TRANSMISOR.Start;
   end ACCION_CICLICA;
end PRODUCTOR;
```



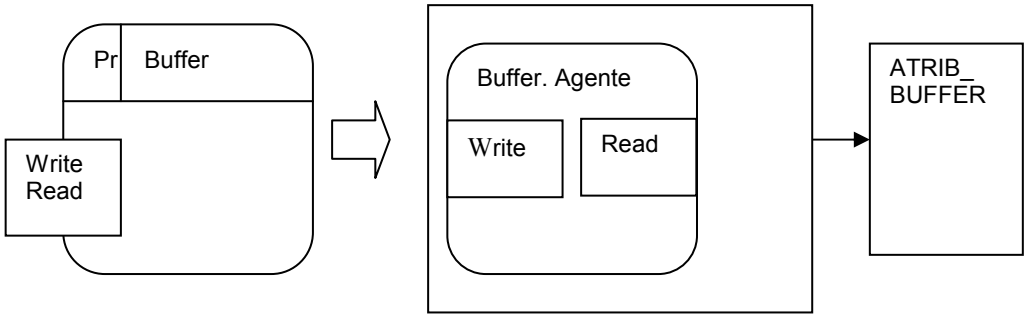

```
with System; use System;
with Ada.Calendar; use Ada.CAlendar;
package ATRIB_CONS is
   Prioridad: constant priority := ...;
   Ceiling_Priority: constant priority:= ...; --prioridad del OBCS
   Intervalo: duration := 0.020;
   Deadline: duration := 0.017;
end ATRIB_CONS;
with System; use System;
with Ada.Calendar; use Ada.Calendar;
with ATRIB_CONS;
package CONSUMIDOR is
   procedure Start;
end CONSUMIDOR;

with BUFFER, AUXPC, ESTRUCTURA, ATRIB_ESTR;
use ATRIB_ESTR;
package body CONSUMIDOR is

-- supongamos que lee un valor entero del buffer
   Item: integer;
-- y que envia otro valor entero a AUXPC
   AuxC:integer:=800;

   task THREAD is
      pragma priority(ATRIB_CONS.Prioridad);
   end THREAD;

   protected OBCS is
      pragma priority(ATRIB_CONS.Ceiling_Priority);
      procedure Signal;
      entry Wait;
   private
      SIN_BARRERA : BOOLEAN := FALSE;
   end OBCS;

   procedure ACCION_ESPORADICA;

   procedure Start is
   begin
      OBCS.Signal;
```

```
   end Start;

   protected body OBCS is
      procedure Signal is
      begin
         SIN_BARRERA := TRUE;
      end Signal;

      entry Wait when SIN_BARRERA is
      begin
         SIN_BARRERA := FALSE;
      end Wait;
   end OBCS;

   task body THREAD is – código estándar de una tarea esporádica
   begin
      loop
         OBCS.Wait;
         ACCION_ESPORADICA;
      end loop;
   end THREAD;

   procedure ACCION_ESPORADICA is
      Reg: Registro;
   begin
      BUFFER.Read(Item);
      AUXPC.Trabajo(AuxC);
      Reg.Valor:=Item;
      Reg.C:='C';
      Reg.Hora:=clock;
      ESTRUCTURA.Write(Reg);
   end ACCION_ESPORADICA;

end CONSUMIDOR;
```



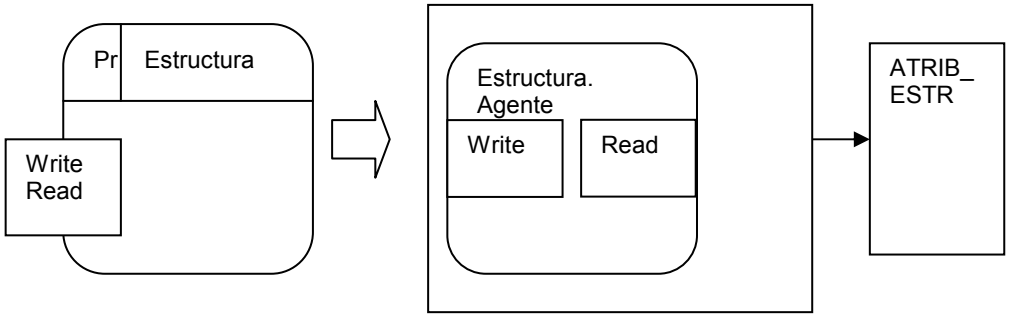

```
with System; use System;
package ATRIB_TRANS is
   Prioridad: constant priority := ...;
   Ceiling_Priority: constant priority:= ...; --prioridad del OBCS
   Intervalo: duration := 0.020;
   Deadline: duration := 0.008;
end ATRIB_TRANS;

with System; use System;
with Ada.Calendar; use Ada.Calendar;
with ATRIB_TRANS;
package TRANSMISOR is
   procedure Start;
end TRANSMISOR;

with AUXT, ESTRUCTURA, ATRIB_ESTR;
package body TRANSMISOR is
```

```
-- Supongamos que lee de la estructura un registro
   Item: ATRIB_ESTR.Pila; -- y que envía estos valores a AUXT
   task THREAD is
      pragma priority(ATRIB_TRANS.Prioridad);
   end THREAD;

   protected OBCS is
      pragma priority(ATRIB_TRANS.Ceiling_Priority);
      procedure Signal;
      entry Wait;
   private
      SIN_BARRERA : BOOLEAN := FALSE;
   end OBCS;

   procedure ACCION_ESPORADICA;
   procedure Start is
   begin
      OBCS.Signal;
   end Start;

   protected body OBCS is
      procedure Signal is
      begin
         SIN_BARRERA := TRUE;
      end Signal;
      entry Wait when SIN_BARRERA is
      begin
         SIN_BARRERA := FALSE;
      end Wait;
   end OBCS;
   task body THREAD is
   begin
      loop
         OBCS.Wait;
         ACCION_ESPORADICA;
      end loop;
   end THREAD;
   procedure ACCION_ESPORADICA is
   begin
      ESTRUCTURA.Read(Item);
      AUXT.Informa(Item);
   end ACCION_ESPORADICA;
end TRANSMISOR;
```




```
with System; use System;
package ATRIB_BUFFER is
   Ceiling_Priority: constant priority:= ...; --prioridad del Agente
end ATRIB_BUFFER;

with ATRIB_BUFFER;
package BUFFER is
   procedure Read (Item: out integer);
   procedure Write (Item : integer);
end BUFFER;

package body BUFFER is

   protected Agente is
```

```
      pragma priority(ATRIB_BUFFER.Ceiling_Priority);
      procedure Write (V: integer);
      function Read return integer;
   private
      Valor : integer;
   end Agente;

   procedure Read (Item: out Integer) is
   begin
      Item:= Agente.Read;
   end Read;
   procedure Write (Item: integer)  is
   begin
      Agente.Write(Item);
   end Write;

   protected body Agente is
      procedure Write (V: integer) is
      begin
         Valor := V;
      end Write;
      function Read return integer is
      begin
         return Valor;
      end Read;
   end Agente;
end BUFFER;
```




```
with Ada.Calendar; use Ada.Calendar;
with System; use System;
package ATRIB_ESTR is
   type Registro is
   record
      Valor: integer;
      C:character;
      Hora:time;
   end record;
   type Pila is array(1..2) of Registro; -- suponemos que guarda los
               -- dos ultimos registros
   Ceiling_Priority: constant priority:= ...; --prioridad del Agente
end ATRIB_ESTR;

with ATRIB_ESTR; use ATRIB_ESTR;
package ESTRUCTURA is
   procedure Read (P: out Pila); --  devuelve los dos ultimos valores
                                 --  almacenados en la estructura
   procedure Write (Item : Registro);
end ESTRUCTURA;

package body ESTRUCTURA is

   protected Agente is
      pragma priority(ATRIB_ESTR.ceiling_priority);
      function Read return Pila;
      procedure Write (V : Registro);
   private
      Datos : Pila;
```

```
   end Agente;
   procedure Read (P: out Pila) is
   begin
      P:= Agente.Read;
   end Read;

   procedure Write (Item: Registro)  is
   begin
      Agente.Write(Item);
   end Write;

   protected body Agente is
      procedure Write (V: Registro) is
      begin
         Datos(2) := Datos(1);
         Datos(1) := V;
      end Write;

      function Read return Pila is
      begin

         return Datos;
      end Read;
   end Agente;

end ESTRUCTURA;

-------------------------------------------------------
package AUXPC is
   procedure Trabajo(Item: integer);
end AUXPC;

with Ada.Text_IO; use Ada.Text_IO;
package body AUXPC is

   procedure Trabajo(Item: integer) is
   begin
      put_line("Valor "& integer'image(Item));
      New_Line;
   end Trabajo;
end AUXPC;

-------------------------------------------------------
with ATRIB_ESTR; use ATRIB_ESTR;
package AUXT is
   procedure Informa(Item: Pila);
end AUXT;

with Ada.Text_IO; use Ada.Text_IO;
with Ada.Calendar; use Ada.Calendar;
package body AUXT is

   procedure Informa(Item: Pila) is
      A : Year_Number;
      M : Month_Number;
      D : Day_Number;
      S : Day_Duration;
      H,Min,Seg, Aux: integer;


   begin
      for I in 1..2 loop
         put_line("Valor "& integer'image(Item(I).Valor));
         put_line("que viene de "& Item(I).C);
         split(Item(I).Hora,A,M,D,S);
         H:=integer(S)/3600;
         Aux:=integer(S-H*3600.0);
         Min:=Aux/60;
         Seg:=Aux-Min*60;
```

```
         put_line("con fecha "&integer'image(integer(D))&'-'
                 &integer'image(integer(M))&'-'
               &integer'image(integer(A))&" a las "
                 &integer'image(H)&':'&integer'image(Min)
                 &':'&integer'image(Seg));
         New_Line;
       end loop;
   end Informa;
end AUXT;
```

7.7.6.3 Análisis temporal (arquitectura física)

Un vez asociados los objetos resultantes de la arquitectura lógica a los recursos hardware disponibles, el análisis del tiempo de respuesta determina si cada uno de los distintos *threads* pueden cumplir los requisitos temporales fijados, suponiendo máximos impedimentos por parte del resto del sistema. El tiempo de respuesta R_i de un thread τ_i es función de:

- C_i: el tiempo de ejecución WCCT (*worst case computation time*) del thread. Incluirá los tiempos de los ciclos de procesamiento y de los ciclos de lectura y escritura en memoria.
- I_i: interferencia causada por la ejecución de threads más prioritarios.
- B_i: el bloqueo debido a la inversión de prioridades.

Se asume planificación por prioridades fijas con desalojo (*preemptive*).

Podríamos disponer de herramientas para realizar una medición y análisis de los tiempos de respuesta.

Supongamos que obtenemos los datos reflejados en las siguiente tablas (tiempos en μs):

Nº thread	Plazo de respuesta (D)	Prioridad (P)	Periodo (T)	Tiempo de respuesta (R)	Planificable
3	8000	6	20000	428	SI
2	9000	5	20000	7394	SI
1	17000	4	20000	14341	SI

Thread nº 1: CONSUMIDOR
Thread nº 2: PRODUCTOR
Thread nº 3: TRANSMISOR

Nº OP	Prioridad (P)	Lo usa
4	10	Threads 1,2,3
3	9	Threads 2,3
2	8	Thread 1,2
1	7	Thread 1,2

OP nº 1: CONSUMIDOR.OBCS
OP nº 2: BUFFER.AGENTE
OP nº 3: TRANSMISOR.OBCS
OP nº 4: ESTRUCTURA.AGENTE

Como práctica, se puede programar el ejemplo expuesto, cambiando las unidades de tiempo, de ms a s.

7.8 HOORA (Hierarchical Object Oriented Requirements Analysis)

El objetivo de HOORA es proporcionar un soporte para el análisis de requisitos software en el proceso de desarrollo de un proyecto, y particularmente para sistemas de tiempo real. Estos requisitos software describen la conducta del sistema, los objetos que lo componen, y las relaciones e interacciones entre los objetos. El método HOORA abarca todo el desarrollo previo al diseño arquitectónico.

7.8.1 Identificación textual de requisitos

El propósito de esta etapa del método es definir QUÉ debe hacer el software.

Todos los requisitos, independientemente del lenguaje, notación o técnica utilizada:

- Definen un objeto, función o estado.
- Limitan o controlan las acciones asociadas a un objeto, función o estado.
- Definen relaciones entre objetos, funciones o estados.

Objetos:

- Algunos requisitos definen objetos. Por ejemplo:

`El sistema debe mostrar la posición actual de un` **`satélite.`**

- Limitan objetos. En el caso anterior, se define la noción de **posición actual**.
- Especifican relaciones entre objetos:

`El sistema debe captar información de los diversos sensores a bordo del satélite`

Especifica que los satélites **contienen objetos sensores**.

Información adicional relacionada con objetos y que se recoge en esta fase:

- Atributos del objeto (por ejemplo, posición).
- Funciones realizadas por el objeto (el satélite se mueve)
- Estados del objeto (el satélite puede estar en modo de transmisión).
- Otros objetos (el satélite contiene sensores).

Funciones:

- Algunos requisitos definen funciones: `El sistema debe` **`mostrar ...`**
- Limitan funciones:

`El sistema debe mostrar los resultados` **`en un plazo de 1 segundo`**

(requisito temporal de la función).

- Especifican relaciones ente funciones:

`El sistema permitirá al usuario seleccionar los sensores a mostrar.` Establece una relación de control entre el usuario y la función de selección del sensor.

Información adicional relacionada con funciones y que se recoge en esta fase:

- El objeto que realiza la función (por ejemplo, el satélite se mueve).
- Atributos de la función (el satélite envía datos con una determinada anchura de banda).
- Estados en los que la función trabaja (el satélite no envía datos si está en modo de fallo).
- Otras funciones.

Estados:

- Algunos requisitos definen estados:

`Después de un` **`autochequeo,`** `el sistema pasa a` **`modo inicialización.`**

- Limitan estados: `El autochequeo no debe durar más de 2 minutos.`
- Especifican relaciones entre estados:

`Después del autochequeo, modo de inicialización.`

Información adicional relacionada con estados y que se recoge en esta fase:

- El objeto que está en ese estado (el satélite está en modo fallo).
- Atributos del estado (tiempo límite del autochequeo).
- Funciones dependientes del estado (durante el autochequeo, el sistema realiza check1 y check2).
- Otros estados.

Resumen de actividades y *outputs* de esta fase:

- Listar los posibles requisitos.
- Entender el contexto del sistema.
- Capturar requisitos funcionales.
- Capturar requisitos de tiempo real.
- Capturar otros requisitos no funcionales.

La salida de esta etapa es un documento de especificación de requisitos con un formato similar al siguiente:

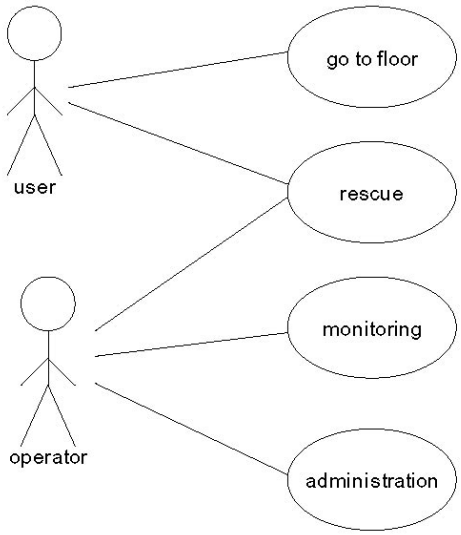
Especificación de Requisitos

```
1. INTRODUCCIÓN
1.1 Propósito
1.2 Alcance
1.3 Glosario
      1.3.1 Acrónimos
      1.3.2 Definición de Términos
1.4 Referencias
      1.4.1 Documentación

2. DESCRIPCIÓN GENERAL
2.1 Relación con otros proyectos
2.2 Función y Propósito
2.3 Consideraciones sobre el Entorno
2.4 Relación con otros Sistemas
2.5 Restricciones Generales
2.6 Descripción del Modelo

3. REQUISITOS ESPECÍFICOS
3.1 Requisitos Funcionales
3.2 Requisitos Temporales
3.3 Requisitos de Interfaz
3.4 Requisitos Operacionales
3.5 Requisitos de Recursos
      3.5.1 Hardware
      3.5.2 Software
3.6 Restricciones de Diseño e Implementación
3.7 Requisitos de Seguridad y Privacidad
3.8 Requisitos de Portabilidad
3.9 Requisitos de Calidad del Software
3.10 Requisitos de Fiabilidad del Software
3.11 Requisitos de Mantenimiento del Software
3.12 Requisitos de Robustez
3.13 Requisitos de Configuración del Software
3.14 Requisitos de Personal
3.15 Requisitos de Adaptación e Instalación
3.16 Otros Requisitos

4. REQUISITOS DE VERIFICACIÓN, VALIDACIÓN y ACEPTACIÓN

5. TRAZABILIDAD DE REQUISITOS
```

7.8.2 Casos de Uso

Un caso de uso especifica una secuencia de acciones que el sistema puede llevar a cabo y que produce un resultado de valor para un actor particular (actor principal).
Un caso de uso es una colección de posibles secuencias de interacciones entre el sistema y actores externos, con un objetivo en particular. Los actores son agentes externos al sistema, y pueden ser personas o sistemas computacionales. Un actor representa un papel que un agente puede desempeñar. El mismo agente puede desempeñar distintos papeles (ser distintos actores). El sistema es tratado como una única entidad que interacciona con los actores. Un caso de uso está relacionado con un objetivo particular que un actor (llamado actor principal) espera del sistema. Otros actores que sean parte de ese caso de uso son llamados actores secundarios.
Los casos de uso están relacionados típicamente con la funcionalidad (objetivos) requerida por los usuarios del sistema. Para hacer el método apropiado para sistemas de tiempo real, también se incluyen actividades autónomas, es decir, actividades internas no requeridas explícitamente por los usuarios finales.

Diagrama de casos de uso:



Un caso de uso tiene más información que la representada en el diagrama. Ésta puede estar recogida en un formato similar al siguiente:

CASO DE USO #	< frase corta que comience por un infinitivo verbal>	
Objetivo	<descripción>	
Precondiciones	<situación previa esperada>	
Postcondiciones exitosas	<situación final si el objetivo se cumple satisfactoriamente>	
Postcondiciones fallidas	<situación final si no se cumple el objetivo>	
Actores Principales y Secundarios	<nombre o descripción del actor principal> <otros sistemas involucrados en la consecución del objetivo>	
Trigger	<la acción que causa el comienzo del caso de uso>	
DESCRIPCIÓN	**Paso**	**Acción**
	1	<poner aquí los pasos del escenario, desde el trigger hasta finalización>
	2	<...>
	3	
EXTENSIONES	**Paso**	**Acción de ramificación**
	1a	<condición que causa la ramificación> : <acción o nombre del subcaso de uso>
SUB-VARIACIONES		**Acción de ramificación**
	1	<lista de variaciones>

INFORMACIÓN RELACIONADA	<Nombre del Caso de Uso>
Prioridad:	<Cómo es de crítico para nuestro sistema / organización>
Tiempos	<valores de tiempo que este caso de uso puede consumir>
Frecuencia	<Cómo de a menudo se espera que ocurra>
Canales para los actores	<por ejemplo, interactivo, ficheros, bases de datos>

CUESTIONES ABIERTAS	<lista de cuestiones aún sin decidir que pueden afectar a este caso de uso >
...cualquier otra información...	<...lo necesario>
Por encima	<casos de uso que puedan incluir a éste>
Subordinados	<enlaces con subcasos de uso>

Ejemplo:
Caso de Uso: Comprar Productos

INFORMACIÓN CARACTERÍSTICA
Objetivo: El comprador envía una petición a nuestra compañía, y espera recibir los productos y la factura.
Precondiciones: Conocemos los datos del comprador (dirección, etc.)
Finalización correcta: El comprador tiene los productos, nosotros tenemos el dinero de la venta.
Finalización fallida: No hemos enviado los productos, el comprador no ha pagado el importe.
Actor principal: El comprador.
Trigger: Recepción del pedido.

ESCENARIO PRINCIPAL
1. El comprador envía la petición de compra.
2. La compañía almacena el nombre del comprador, dirección, productos pedidos, etc.
3. La compañía envía al comprador información sobre productos, precios, fechas de envío, etc.
4. El comprador firma el pedido.
5. La compañía ordena el envío al comprador.
6. La compañía envía la factura al comprador.
7. El comprador paga la factura.

EXTENSIONES
3a. La compañía no dispone de uno de los artículos pedidos:
3a1. Renegociar pedido.
El comprador paga directamente con tarjeta de crédito:
a 4a1. Cobrar mediante tarjeta de crédito (otro caso de uso)

7a. El comprador devuelve el pedido:
7a1. Recibir pedido devuelto (otro caso de uso)

SUB-VARIACIONES
1. El comprador puede utilizar
llamada telefónica,
fax,
formulario web

7. El comprador puede pagar con
dinero en metálico o transferencia bancaria
cheque
tarjeta de crédito

INFORMACIÓN RELACIONADA
Prioridad: máxima
Tiempos: 5 minutos para pedido, 45 días hasta pago
Frecuencia: 200/días
Casos de Uso Subordinados:
Crear envío
Cobrar con tarjeta de crédito
Manejar productos devueltos
Canal para Actor Principal: puede ser teléfono, fichero o interactivo
Actores Secundarios: compañía de la tarjeta de crédito, banco, servicio de envíos
Canales para Actores Secundarios:

Ahora, colocar la información anterior en un formato de tabla:

CASO DE USO 1	< Comprar Productos >	
Objetivo	< El comprador envía una petición a nuestra compañía, y espera recibir los productos y la factura >	
Precondiciones	< Conocemos los datos del comprador (dirección, etc.)>	
Postcondiciones exitosas	< El comprador tiene los productos, nosotros tenemos el dinero de la venta>	
Postcondiciones fallidas	< No hemos enviado los productos, el comprador no ha pagado el importe>	
Actores Principales y Secundarios	< El comprador > < compañía de la tarjeta de crédito, banco, servicio de envíos >	
Trigger	< Recepción del pedido>	
DESCRIPCIÓN	**Paso**	**Acción**
	1. El comprador envía la petición de compra. 2. La compañía almacena el nombre del comprador, dirección, productos pedidos, etc. 3. La compañía envía al comprador información sobre productos, precios, fechas de envío, etc. 4. El comprador firma el pedido. 5. La compañía ordena el envío al comprador. 6. La compañía envía la factura al comprador. 7. El comprador paga la factura.	
EXTENSIONES	**Paso**	**Acción de ramificación**
	3a. La compañía no dispone de uno de los artículos pedidos: 3a1. Renegociar pedido. 4a El comprador paga directamente con tarjeta de crédito: 4a1. Cobrar mediante tarjeta de crédito 7a. El comprador devuelve el pedido: 7a1. Recibir pedido devuelto.	
SUB-VARIACIONES	**Paso**	**Acción de ramificación**
	1. El comprador puede utilizar: llamada telefónica, fax, formulario web 7. El comprador puede pagar con: dinero en metálico o transferencia bancaria, cheque, tarjeta de crédito	

INFORMACIÓN RELACIONADA	< Comprar Productos>
Prioridad:	< máxima >
Tiempos	<5 minutos para pedido, 45 días hasta pago >
Frecuencia	<200/días >
Canales para los actores	< puede ser teléfono, fichero o interactivo >
CUESTIONES ABIERTAS	<lista de cuestiones aún sin decidir que pueden afectar a este caso de uso >
...cualquier otra información...	<...lo necesario>
Por encima	<casos de uso que puedan incluir a éste>
Subordinados	< Crear envío, Cobrar con tarjeta de crédito, Manejar productos devueltos >

Recomendaciones para la elaboración de los casos de uso de un sistema:

- Considerar todos los agentes externos que interaccionan con el sistema. Pueden ser humanos u otros computadores. Diferenciar los diferentes papeles que cada agente puede desempeñar en su interacción con el sistema. Un agente haciendo un papel determinado se denomina actor. Analizar cada papel por separado.

- Los actores pueden ser clasificados de diversas maneras:

  - Activos/Pasivos: Actores pasivos son aquellos que participan en casos de uso, pero que nunca inician un caso de uso.

  - Principales/Secundarios, ya comentado.

  - Clientes/No clientes: Utilizan el sistema con algún propósito determinado (clientes) o simplemente le envían algún *trigger*.

- Una vez identificados y clasificados los actores, considerar cada uno por separado. En primer lugar, a los actores principales clientes o activos. Contestar a las siguientes preguntas:

  - ¿Cuál es el principal objetivo que el actor pretende del sistema?

  - ¿Cuándo son estos objetivos aplicables?

  - ¿Cuál es el efecto de alcanzar el objetivo?

  - ¿Cuáles son los requisitos temporales?

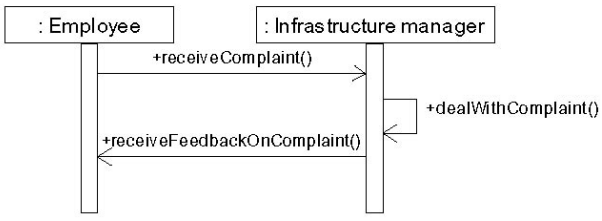
7.8.3 Elaboración de Escenarios

Un escenario es una secuencia de interacciones que ocurren bajo ciertas condiciones, cuya finalidad es conseguir el objetivo del actor principal. Estas interacciones comienzan con un trigger y continúan hasta que el objetivo es conseguido o abandonado.

Por cada caso de uso, identificamos un conjunto representativo de escenarios. Más adelante, identificaremos para cada escenario, las clases que necesitaremos para que el sistema se comporte según los objetivos del caso de uso.

Diagrama de secuencias

Un diagrama de secuencias identifica los componentes que deberían existir para cumplimentar el comportamiento descrito en un escenario de un caso de uso. Al final, tendremos identificadas las clases (nombre, propósito, responsabilidades), los servicios que deben ofrecer a las demás clases y los estados relacionados.
Si una clase A envía un mensaje X a una clase B => la clase B ofrece el servicio X.



Identificación de clases

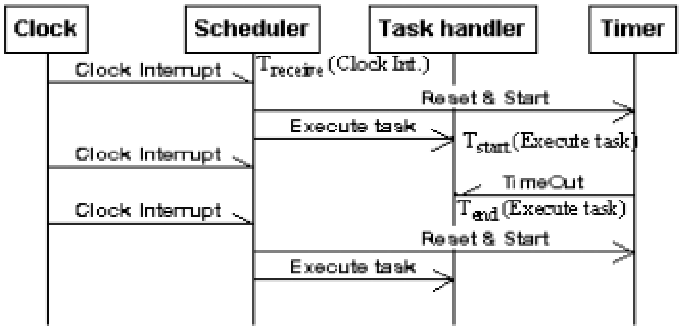
Primero identificamos responsabilidades y luego seleccionamos o identificamos la clase a la que se adjudicará la responsabilidad.

Tipos de responsabilidades:

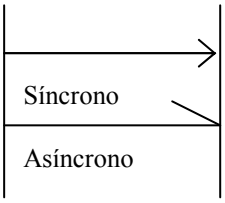
- Hacer
  - Hacer algo por sí misma.

  - Iniciar una acción en otros objetos.
  - Controlar y coordinar actividades en otros objetos.
  - Chequear.
- Conocer
  - Información sobre datos privados encapsulados.
  - Información sobre objetos relacionados.
  - Información que debe calcular.

Un diagrama de secuencias conlleva la noción de tiempo, y puede modelar los requisitos de tiempo real. Éstos se representan mediante variables de tiempo real (RT), como en el ejemplo siguiente:



Las flechas que representan los mensajes diferencian entre mensajes síncronos (el emisor espera hasta que el receptor finalice la ejecución del mensaje), y asíncronos (el emisor no espera).



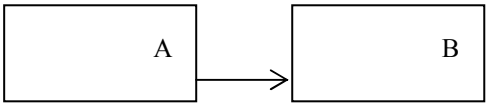
Resumen de actividades de la elaboración de escenarios

- Identificación de responsabilidades
- Identificación de clases
- Asignación de responsabilidades a clases
- Elaboración de escenarios
- Identificación de operaciones (servicios) de cada clase
- Requisitos de tiempo real

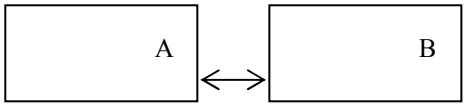
Al final, tendremos un conjunto de escenarios, representados mediante diagramas de secuencia, y una descripción de las clases (nombre, propósito, responsabilidades y servicios).

7.8.4 Modelo dinámico

Un modelo dinámico por cada clase. Es la unión de todos los diagramas de secuencia de un nivel de abstracción dado en un único diagrama para cada clase. Este nuevo diagrama muestra las clases con las que colabora:



A utiliza B



A utiliza B

B utiliza A

El principal objetivo de agrupar los diagramas de secuencias en modelos dinámicos es el de chequear la consistencia del sistema: podemos ver si las clases que forman los diagramas de secuencia pertenecen realmente al nivel en que estamos. Si los diagramas del modelo dinámico no están bien balanceados puede ser debido a que incluimos diferentes niveles de detalle, por lo que alguna clase debe ser movida a un nivel de abstracción más bajo.

Si una clase tiene muchas responsabilidades quizás deba ser descompuesta en clases más pequeñas, y a la inversa: clases con muy pocas responsabilidades quizás puedan ser agrupadas en una clase mayor.

Se puede distinguir entre tres tipos de clases:
- Objetos frontera, utilizados por los actores para comunicarse con el sistema.
- Objetos entidad, que son objetos software que modelizan objetos reales.
- Objetos de control, que sirven como enlace entre los dos anteriores.

Reglas de Jerarquía en HOORA

El nivel superior es simple: hay un sistema y varios actores.
En el siguiente nivel (nivel 1) descomponemos el sistema en clases. Todavía vemos los actores en este nivel, que se encargan de comenzar los escenarios, activando (mandando triggers) a las clases..
En el nivel 2 ya no se ven los actores. Las clases del nivel 1 activan los escenarios del nivel 2.
En cada nivel vemos por tanto las clases de ese nivel y las del nivel superior

7.8.5 Modelo estático

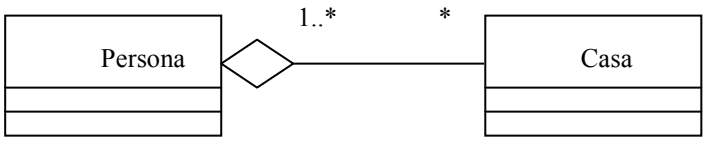
El modelo dinámico muestra las clases y sus interacciones, es decir, el diálogo entre clases.

La finalidad del modelo estático es mostrar cada clase y todas las clases estáticamente relacionadas con ella. Estas relaciones pueden ser de:

- agregación
- composición
- asociación
- herencia

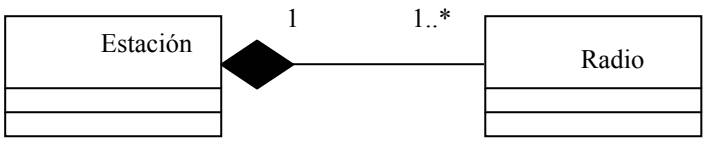
Agregación y composición denotan relaciones todo-parte. La **agregación** es menos fuerte que la composición: cuando el todo desaparece, la parte no desaparece.

Ejemplo:

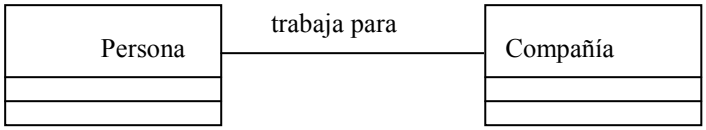


Persona posee una Casa. Una Casa puede cambiar de dueño. Cuando el dueño desaparece, la casa no desaparece, y viceversa. 1 Persona puede poseer de 0 a n Casas (denotado con). Una casa puede tener de 1 a n dueños (denotado con 1..).

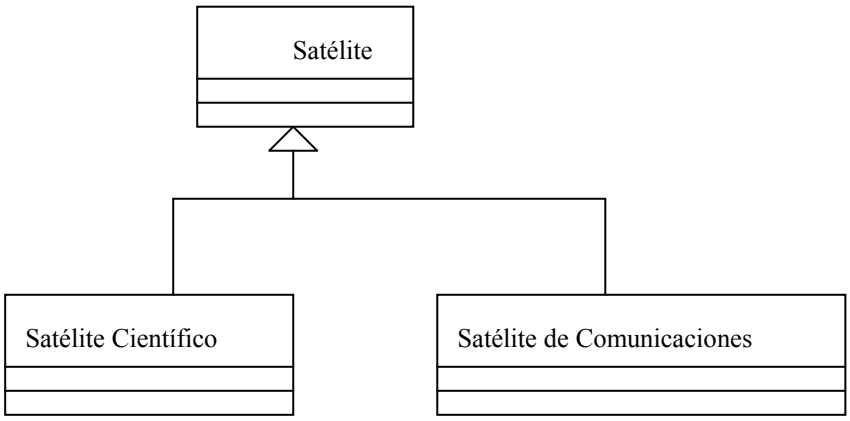
Composición. Una parte no puede existir sin un todo. Una parte sólo puede pertenecer a un todo



Asociación. A diferencia de las relaciones anteriores, las dos partes son consideradas a igual nivel, una clase nœs subordinada de la otra. La mayoría de las asociaciones son binarias (1 a 1).



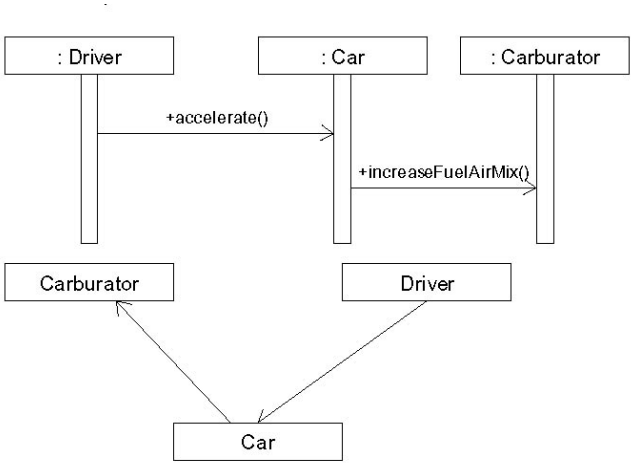
Herencia (o Generalización). Relación entre una clase más general (superclase) y otra más específica (subclase). Todas las características de la superclase son heredadas por la subclase.



Los requisitos son la motivación para la creación del modelo dinámico. El modelo dinámico prueba que se satisfacen los requisitos. La motivación para el modelo estático es doble:

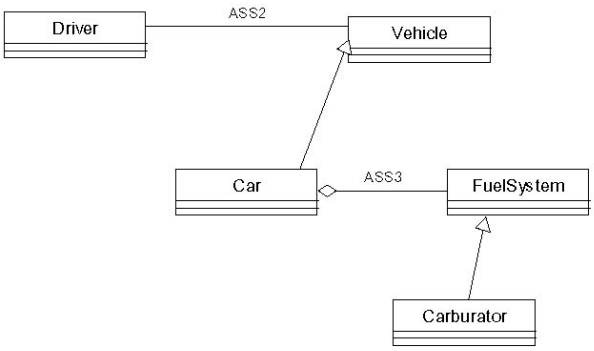
- Proporcionar una estructura que satisfaga el modelo dinámico
- Proporcionar una estructura mantenible ante la presencia de cambios.

Considerar el siguiente diagrama de secuencia y el correspondiente modelo dinámico:



Podríamos construir un modelo estático con las clases Driver, Car y Carburator, válido pero inflexible.

El siguiente modelo estático permite anticiparnos a futuros cambios: el conductor podrá conducir distintos tipos de vehículos, los coches podrán tener distiontos tipos de carburantes, ...



7.8.6 Modelo de Paquete

El modelo de Paquete compendia los modelos dinámico y estático y muestra cómo los paquetes, que contienen las clases, interaccionan y se relacionan. Muestra cada paquete y todas sus dependencias de otros paquetes. La siguiente tabla resume las convenciones para calcular estas dependencias. Se asume que la clase A pertenece al paquete A, y la clase B al paquete B.

Los paquetes representan niveles de abstracción. Un paquete puede incluir paquetes anidados. Las reglas de jerarquía de HOORA están basadas en el concepto de paquete. Aunque no impone valores, lo típico es que el número de niveles hacia abajo sea 1, y el número de niveles hacia arriba no esté limitado. Esto implica que un paquete tiene visibilidad de sus hermanos, de todos sus ancestros y de sus hijos; no ve por tanto ni a sus nietos ni a sus sobrinos.

HOORA define los **paquetes de utilidad**. Se trata de paquetes que necesitan ser accesibles a otros paquetes independientemente de su lugar en la jerarquía.

La estructura de paquete y la estructura de clase pueden ser diferentes. En un nivel alto de abstracción, dividimos al sistema en clases, desde un punto de vista de sus responsabilidades. En niveles más bajos, subclases que pertenecen a distintas clases del primer nivel pueden ser agrupadas en un paquete, debido a que tengan un comportamiento parecido, y también subclases de la misma clase principal pueden ser divididas entre distintos paquetes

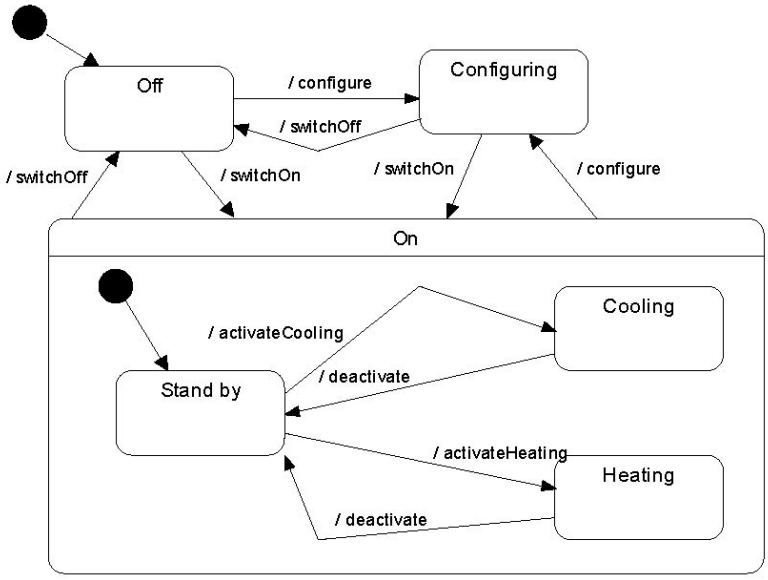
7.8.7 Modelo de Estado

Hasta ahora, hemos identificado clases y operaciones, y las hemos organizado en paquetes. Sin embargo, hay clases cuyas operaciones no pueden ser definidas sin describir las relaciones temporales entre ellas. Estas clases necesitarán un modelo de estado. En el modelo de estado, se modela el ciclo de vida de un objeto de una clase como una máquina de estados finita:

- El objeto puede asumir un número finito de condiciones de existencia, llamadas estados.
- El objeto está en un estado durante un periodo de tiempo significativo.
- El cambio entre estados se puede producir en un número finito de formas, llamadas transiciones.
- Las transiciones se activan mediante eventos, asociados a las operaciones del estado.

El modelo de estado se representa mediante diagramas de estado.

Ejemplo de un diagrama de estados de una unidad de aire acondicionado:



7.8.8 Traza de requisitos

Los requisitos identificados en la primera fase del análisis deben ser cubiertos por los elementos del modelo.

1. Asociar requisitos a los paquetes de alto nivel.
2. Propagar los requisitos hacia paquetes de más bajo nivel y hacia clases.
3. Si es necesario, propagarlos hacia elementos de más bajo nivel, como operaciones, estados y atributos.
4. Crear tablas de las trazas (requisitos frente a elementos del modelo).
5. Identificar requisitos que no estén cubiertos por ningún elemento.

7.9 EJEMPLO: Boyas Marinas

Cada boya recoge información de los siguientes sensores:

- GPS (latitud y longitud)
- Sensores de temperatura (temperatura del aire y del agua)
- Sensor de viento (velocidad del viento)

Cada boya dispone además de:

- Un radio-receptor, para recibir peticiones de información
- Un interruptor, para activar/desactivar la emisión de mensajes SOS
- Un radio-transmisor, que puede emitir
    - Última información leída de los sensores
    - Resumen de la información leída en las últimas 24 horas
    - Mensaje SOS

El sistema debe:

- Leer y almacenar velocidad del viento cada 30 segundos
- Leer y almacenar temperaturas y posición cada 10 segundos
- Cada 60 segundos, emitir la última información recogida
- Si recibe petición de información, emitir un resumen de la información de las últimas 24 horas, con mayor prioridad que la emisión periódica
- Si Interruptor pasa a ON, emitir SOS con mayor prioridad que las emisiones anteriores
- Si Interruptor pasa a OFF, dejar de emitir SOS

Formato de los datos de entrada: cadenas de caracteres

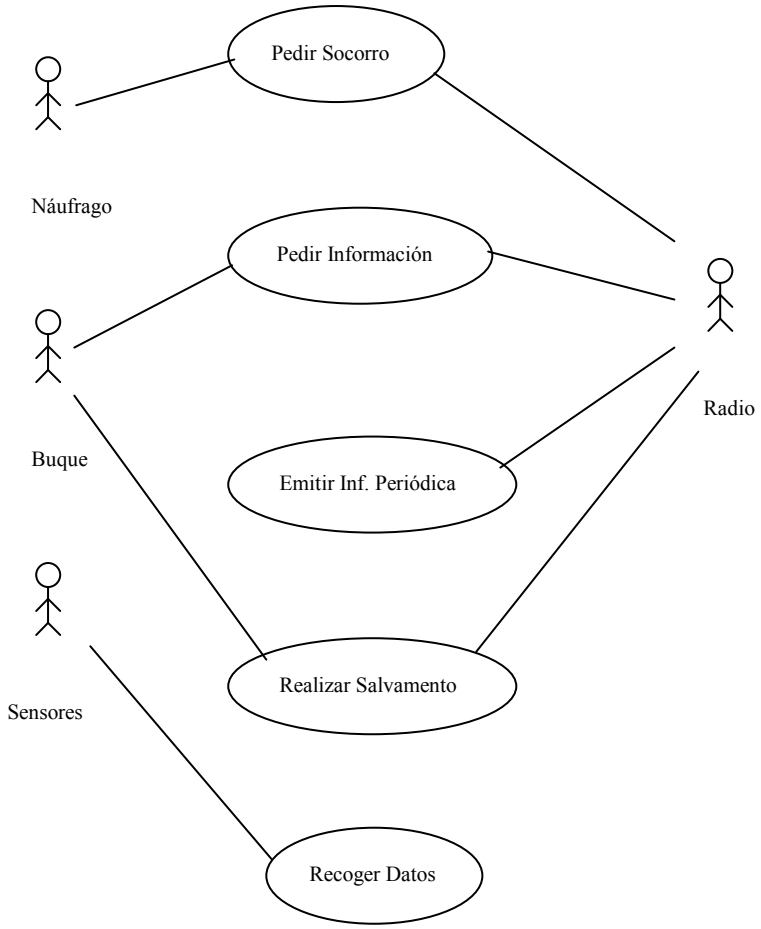
- Temperaturas en ºC, en el rango [-50.0, 50.0]
- Velocidad del viento en nudos (millas/hora, 1milla=1852m), en el rango [0.0, 100.0]
- Latitud: formato ggmm.mmmC (C=N,S), rango [0.0, 90.0]

- Longitud: formato gggmm.mmmC (C=E,W), rango [0.0, 180.0]

Formato de los datos de salida: cadenas de caracteres

- Información periódica:
  hh:mm:ss **TAgua**: xxx.x **TAire**: xxx.x **VViento**: xxx.x **Lat**: ggmm.mmmC **Lon**: gggmm.mmmC
- Información bajo petición: Toda la información recogida en las últimas 24 horas.
- Mensaje SOS: cadena de caracteres SOS SOS SOS emitida cada segundo hasta desactivación de emergencia. Esta emisión impedirá las otras emisiones.

Casos de uso



CASO DE USO 1	< Pedir Socorro >	
Objetivo	< Transmisión continuada de un mensaje SOS >	
Precondiciones	< Transmisiones normales>	
Postcondiciones exitosas	< Se bloquean las transmisiones normales, y se inicia transmisión de emergencia>	
Postcondiciones fallidas		
Actores Principales y Secundarios	< El náufrago > < La radio >	
Trigger	< Interruptor ON>	
DESCRIPCIÓN	**Paso**	**Acción**
	8. 9. 10.	El náufrago pone el interruptor en posición ON El sistema bloquea las transmisiones normales El sistema comienza a emitir SOS

CASO DE USO 2	< Pedir Información >
Objetivo	< Desde un buque se envía una petición de información >
Precondiciones	< Transmisiones normales >
Postcondiciones exitosas	< Se emite por radio un resumen de datos recogidos en las últimas 24h >
Postcondiciones fallidas	< El sistema está en modo de emergencia. No se emite información>

Actores Principales y Secundarios	< El buque > < La radio >
Trigger	< Recepción de la petición de información>
DESCRIPCIÓN	**Paso** **Acción**
	1. El buque envía la petición de información. 2. Se lee del almacén de datos el resumen de las últimas 24h 3. Se envía por radio esta información.

CASO DE USO 3	< Emitir Información Periódica >
Objetivo	< Envío periódico de información >
Precondiciones	< Transmisiones normales >
Postcondiciones exitosas	< Se emiten por radio los últimos datos recogidos >
Postcondiciones fallidas	< El sistema está en modo de emergencia. No se emite información>
Actores Principales y Secundarios	< La radio >
Trigger	< Se cumple el periodo de emisión>
DESCRIPCIÓN	**Paso** **Acción**
	1. Se leen del almacén de datos los últimos datos recogidos 2. Se envía por radio esta información.

CASO DE USO 4	< Realiza Salvamento >
Objetivo	< Señalar que el náufrago ha sido salvado >
Precondiciones	< Transmisión de emergencia>
Postcondiciones exitosas	< Se finaliza la transmisión de emergencia y se desbloquean las transmisiones normales>
Postcondiciones fallidas	
Actores Principales y Secundarios	< El buque> < La radio >
Trigger	< Interruptor OFF>
DESCRIPCIÓN	**Paso** **Acción**
	1. El buque pone el interruptor en posición OFF 2. El sistema deja de emitir SOS 3. El sistema desbloquea las transmisiones normales

CASO DE USO 5	< Recoger Datos >
Objetivo	< Almacenar datos de temperaturas y posiciones >
Precondiciones	
Postcondiciones exitosas	< Los datos recogidos quedan almacenados en el sistema >
Postcondiciones fallidas	
Actores Principales y Secundarios	< Los sensores >
Trigger	< Se cumple el periodo de captación de datos>
DESCRIPCIÓN	**Paso** **Acción**
	1. Se leen los datos de los sensores 2. Se guardan en el almacén

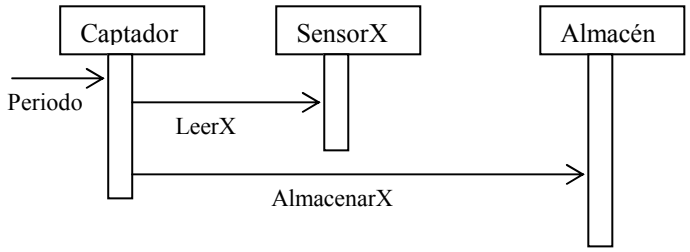
Lista de eventos

Evento externo	Trigger	Respuesta	Modo llegada	Caso de uso	Prioridad

Petición información	Mensaje desde buque	Si no hay emergencia, el sistema envía resumen de las últimas 24h	Dir	Pedir Información	Prioridad media
Petición socorro	Interruptor ON	El sistema emite SOS y bloquea el resto de emisiones	Dir	Pedir Socorro	Prioridad máxima
Salvamento	Interruptor OFF	El sistema deja de emitir SOS, y se reanudan el resto de emisiones	Dir	Realizar Salvamento	Prioridad alta
Tiempo de emisión periódica	Reloj interno	Si no hay emergencia, se emite información	Tmp	Emitir Inf. Periódica nm nman	Prioridad baja
Almacenamiento de información	Reloj interno	El sistema almacena información	Tmp	Recoger Datos	Prioridad media

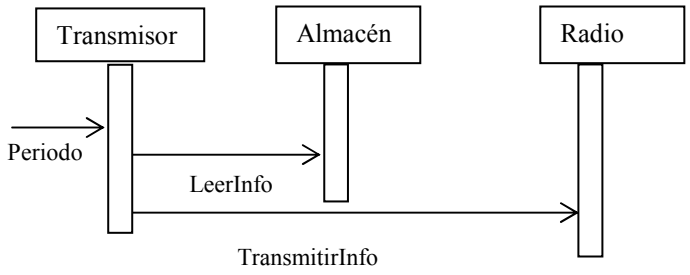
Diagramas de secuencias

– Recoger Datos

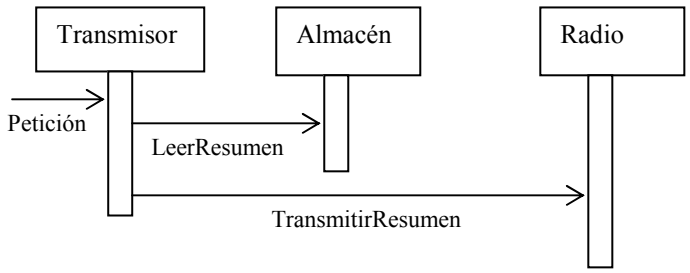


Esquema similar para todos los sensores

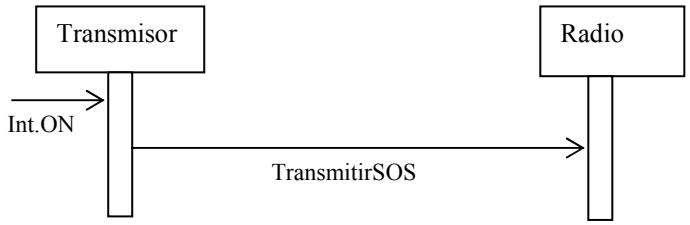
– Emitir Inf. Periódica



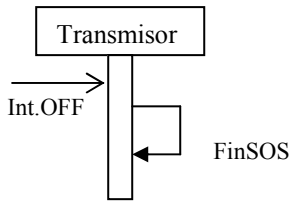
– Pedir Información



– Pedir Socorro



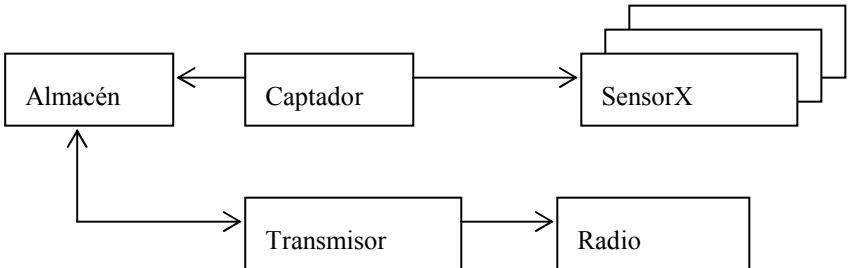
– Realizar Salvamento



Clases identificadas:

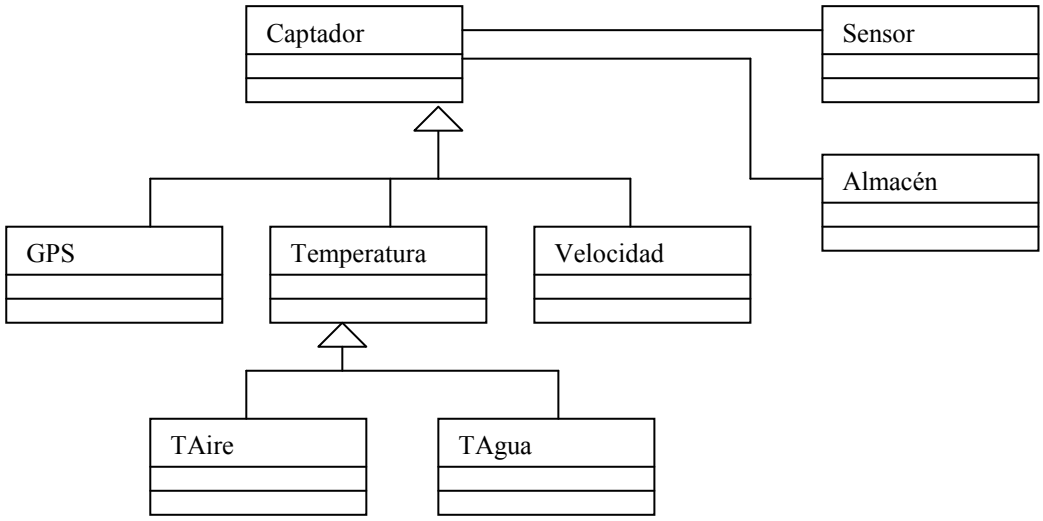
Nombre	Propósito	Operaciones
Transmisor	Recibir eventos y ordenar transmisiones	Petición IntON IntOFF
Captador	Comunicación periódica con los sensores	
Almacén	Almacenar datos de los sensores y proporcionar la información a transmitir	AlmacenarX LeerInfo LeerResumen
Radio	Transmitir información y SOS. Encapsula las características técnicas del radio-transmisor	TransmitirInfo TransmitirResumen TransmitirSOS
SensorX	Leer datos del sensor X. Encapsula las características técnicas del sensor.	LeerX

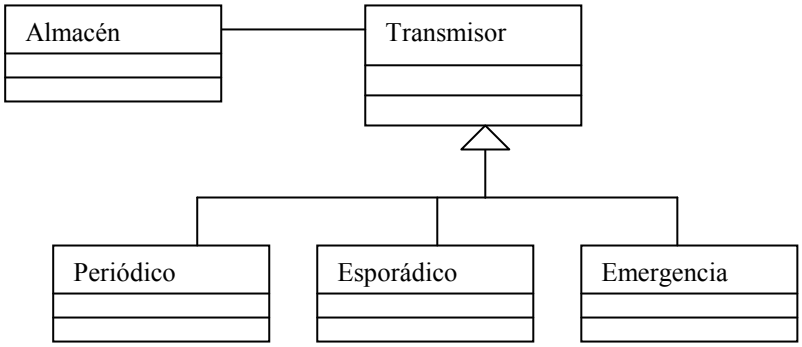
Modelo dinámico



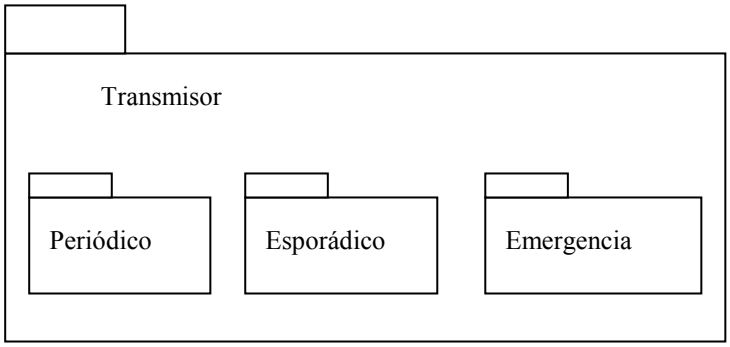
(Represento en un único modelo dinámico todas las clases, por su simplicidad)

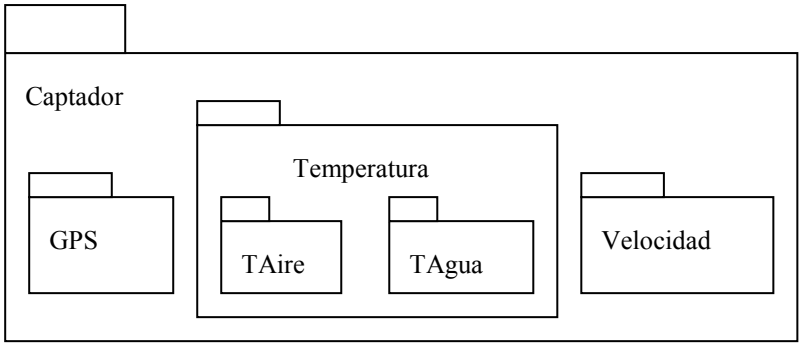
Modelo estático

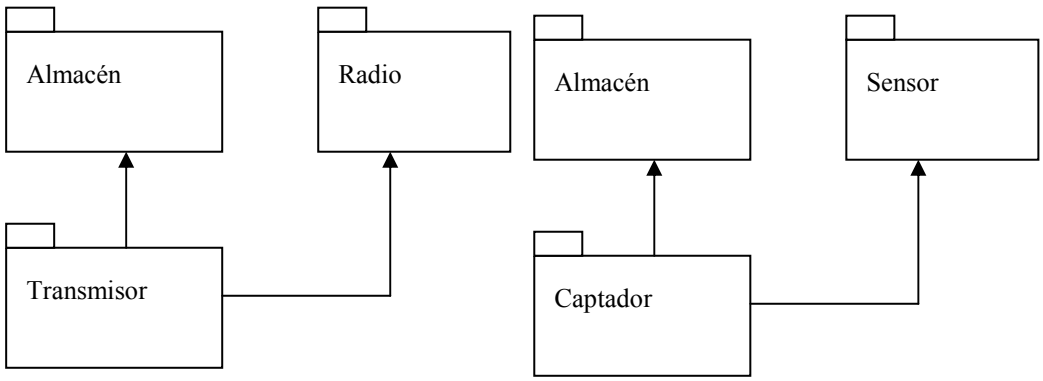




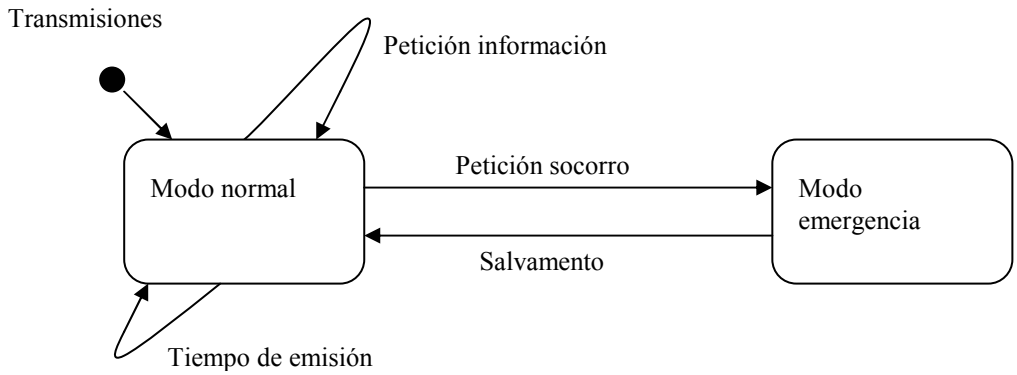
Modelo de Paquete



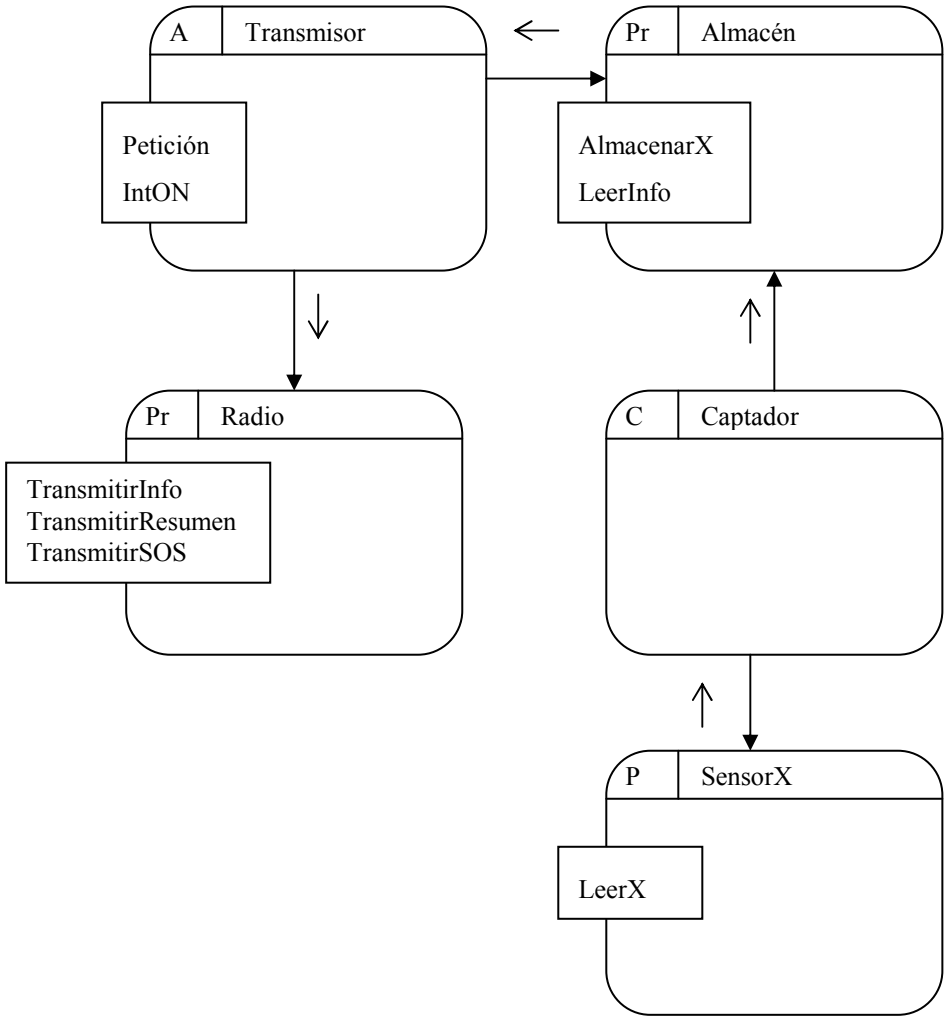


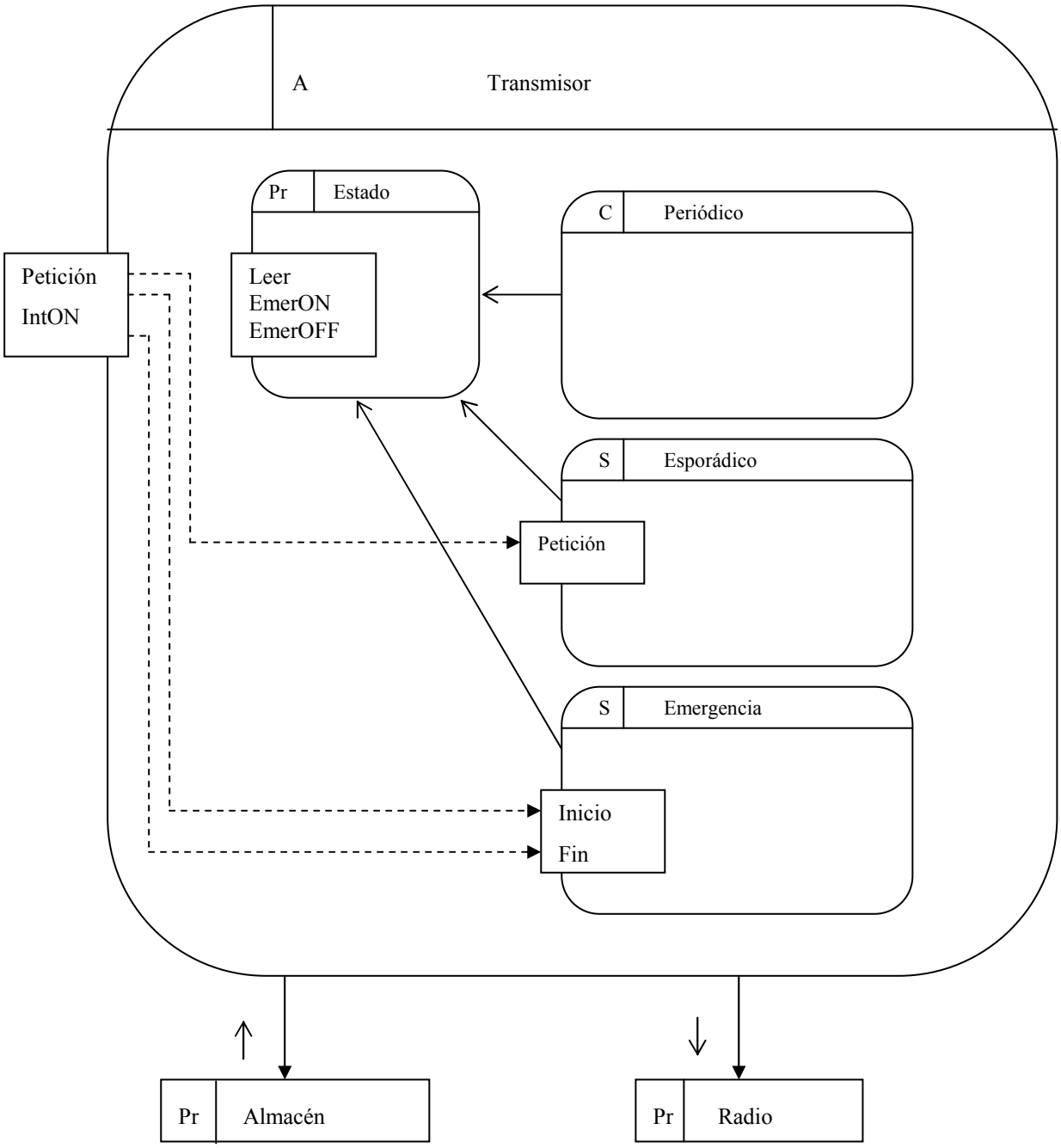


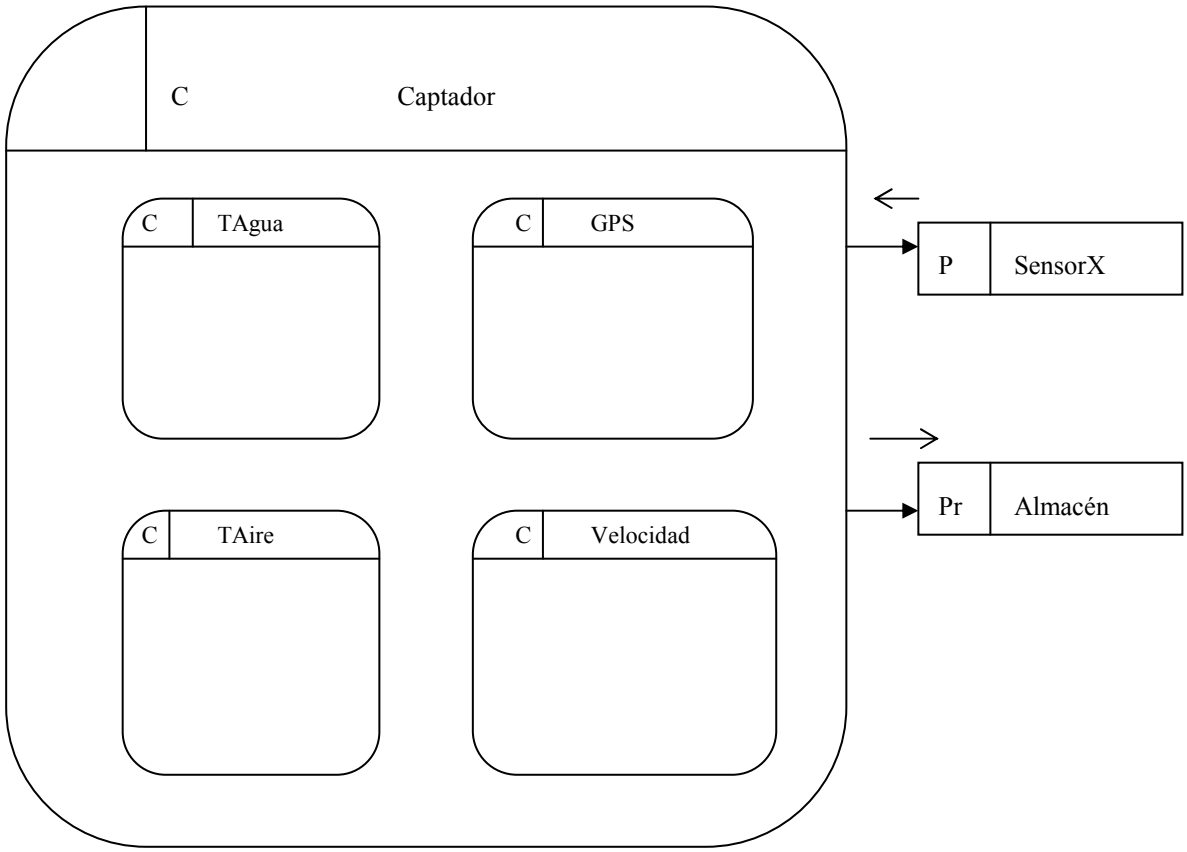
Modelo de estado

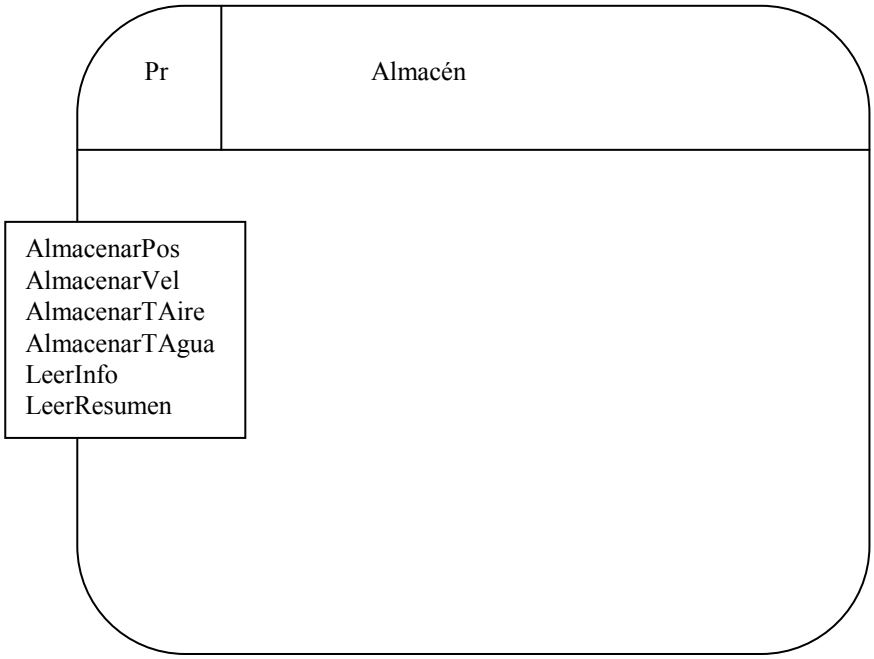


HOOD




A
Transmisor
Pr
Estado
Leer
EmerON
EmerOFF
C
Periódico
Petición
IntON
S
Esporádico
Petición
S
Emergencia
Inicio
Fin
Pr
Almacén
Pr
Radio


C
Captador
C
TAgua
C
GPS
P
SensorX
Pr
Almacén
C
TAire
C
Velocidad


Pr
Almacén
AlmacenarPos
AlmacenarVel
AlmacenarTAire
AlmacenarTAgua
LeerInfo
LeerResumen

Bibliografía:

[1] Burns, A., Wellings, A.J.; 'HRT-HOOD. A Structured Design Method for Hard Real-Time Systems'; Journal of Real-Time Systems, 6(1):73--114, (1994)

[2] Burns, A., Wellings, A.; Sistemas de Tiempo Real y Lenguajes de Programación (3ª ed); ed. Addison-Wesley (2003); http://www.cs.york.ac.uk/rts/RTSBookThirdEdition.html

[3] De la Puente, J.A.; página web de la asignatura de Sistemas de Tiempo Real. Universidad Politécnica de Madrid.; http://www.dit.upm.es/~jpuente/strl

[4] Rosen, J.P.; 'HOOD for large complex projects'; ADALOG. http://www-iasc.enst-bretagne.fr/~kermarre/workshop/Proceedings/jprosen.pdf

[5] Vardanega, T.; 'Development of On-Board Embedded Real-Time Systems: An Engineering Approach'; Tesis Doctoral; Universidad Técnica de Delft (NL), (1998)

[6] http://www.hoora.org/