Procesos de Desarrollo de Software

Copyright © 2021 Juan Marín Noguera, juan.marinn@um.es.

Esta obra está bajo la licencia Reconocimiento-CompartirIgual 4.0 Internacional de Creative Commons (CC-BY-SA 4.0). Para ver una copia de esta licencia, visite https://creativecommons.org/licenses/by-sa/4.0/.

Bibliografía:

- Universidad de Murcia. Apuntes de Procesos de Desarrollo de Software.
- Till Mossakowski (2016). *Semantics of UML class diagrams*. Otto-von-Guericke Universität Magdeburg. Recuperado de http://theo.cs.ovgu.de/lehre/lehre16s/modelling/slides6.pdf.
- Gonzalo Génova, Juan Llorens, Paloma Martínez (2001). *The meaning of multiplicity of n-ary associations in UML*. Universidad Carlos III de Madrid. DOI: 10.1007/s10270-002-0009-3. Recuperado de https://www.academia.edu/13759154/The_meaning_of_multiplicity_of_n_ary_associations_in_UML.

Capítulo 1

Introducción

El **software** esta formado por instrucciones para la máquina, estructuras de datos, bases de datos, documentos, entrenamiento, soporte al consumidor e instalación. Al ser un elemento lógico, no físico, es más difícil de medir, validar y verificar que el hardware, y aunque no se estropea, su calidad suele ir deteriorándose por los cambios.

1.1. Calidad del software

Sommerville define los siguientes factores de calidad del software:

- Facilidad de mantenimiento: Legibilidad.
- Confiabilidad: Fiabilidad (que ofrezca los mismos resultados en las mismas condiciones), seguridad y protección. Se distingue entre seguridad informática (*security*); seguridad humana o protección (*safety*) y privacidad.
- Eficiencia.
- Facilidad de uso.

También queremos que el software sea correcto (se ajuste a las especificaciones dadas por el usuario), no erróneo, robusto (tolerante a fallos), portable, adaptable o extensible y reutilizable.

Con el avance del hardware, cada vez se pueden hacer aplicaciones más complejas y con frecuencia el software es la parte más compleja de un sistema, dando lugar a problemas para estimar el tiempo, coste y esfuerzo para el desarrollo y para asegurar la calidad. Hay tanto **complejidad esencial**, la complejidad de tratar con muchos conceptos a la vez, y **complejidad accidental**, derivada de aspectos como la notación o la organización del proyecto software.

Como el software no es físico, es más difícil de probar, y como es maleable, es más fácil dejar fallos y resolverlos después que hacer un análisis exhaustivo de corrección. Además hay problemas de comunicación con los clientes y, tradicionalmente, se dedica poco tiempo al análisis y el diseño. Los métodos tradicionales de gestión de proyectos no funcionan demasiado bien en el software. No hay demasiados datos previos de guía ni estándares ampliamente usados, y la base formal no se suele usar. Todo esto hace que el tiempo y el coste del desarrollo sean

elevados, el software entregado contenga errores y constatar el progreso en el desarrollo sea difícil.

No hay una única solución para todo, pero pueden ayudar la adquisición de componentes en vez de su construcción desde cero, el refinamiento de requisitos, el prototipado rápido y el desarrollo incremental, así como tener buenos diseñadores.

La **ingeniería del software** es una parte de la ingeniería de sistemas, que Boehm define como la aplicación práctica y sistemática de conocimiento científico a la producción de programas correctos a tiempo y dentro de las estimación de presupuestos, así como de su documentación para desarrollarlos, usarlos y mantenerlos. Es una parte de la ingeniería de sistemas que se fundamente en campos como las ciencias de la computación, la programación, la ingeniería, la administración, las matemáticas o la economía.

1.2. Historia

En la década de los 50, se escribían aplicaciones sencillas en lenguajes de bajo nivel como añadido al hardware, desarrolladas a medida. En los 60 se concibió el software como producto y se crearon las primeras aplicaciones complejas, que todavía eran mayoritariamente monolíticas, con poco uso de componentes.

En 1968–69 tienen lugar las *Software Engineering Conferences* de la OTAN, donde se acuña en 1968 el término **crisis del software** para referirse a la dificultad de manejar la creciente complejidad, y se decide que la producción de programas debe abordarse como una ingeniería más.

En los 70 surgen la programación estructurada, el modelado de datos y el relacional y los primeros métodos estructurados. En los 80 coge fuerza la programación orientada a objetos y aparecen los primeros métodos orientados a objetos. Surge tecnología CASE pero fracasa.

En los 90 se populariza Internet. La programación orientada a objetos se generaliza, surge la programación virtual y se crea UML, que acaba con la guerra de los métodos y permite tecnología CASE de segunda generación. En la década de los 2000 surgen los métodos ágiles y se asume la importancia de la seguridad, y en la de los 2010 surgen la computación móvil y en la nube y se popularizan las redes sociales, los grandes datos, el aprendizaje computacional y el desarrollo de software global.

1.3. Actualidad

Actualmente existe un consenso en la importancia de la ingeniería del software y se niega la vigencia de la crisis de software, pero la disciplina no es tan madura como otras ingenierías. Los principales problemas son:

- **Heterogeneidad:** El software debe trabajar con componentes muy heterogéneos, incluyendo **sistemas heredados** antiguos que no se adhieren a las prácticas actuales.
- **Entrega:** Se prefiere usar actualizaciones incrementales a una sola entrega puntual, y el modelo de desarrollo debe ser acorde.
- **Confianza:** Muchos sistemas dependen del software, y se debe poder confiar en su seguridad, privacidad, protección, disponibilidad, etc.

Algunas aproximaciones son los métodos ágiles como XP o Scrum y métodos *lean* como Kanban, el **SWEBOK** (*guide to the SoftWare Engineering Body Of Knowledge*), títulos universitarios de grado y máster en ingeniería de software, comités para acreditar estos títulos, CMMi y las **DevOps** (automatización en el desarrollo y el despliegue del software).

1.4. UML

Un **sistema** es algo que se está desarrollando. Un **modelo** es una abstracción de un sistema para comprenderlo mejor.

El modelo permite una mayor expresividad y capacidad de comunicación que un lenguaje de programación típico, y sirve como documentación. Puede conectarse a lenguajes de programación mediante ingeniería directa e inversa y desarrollo dirigido por modelos.

Los modelos se expresan en un lenguaje, y a mediados de los 90 había muchos lenguajes de modelado orientado a objetos. Algunos métodos que usan este tipo de modelos son:

1. **OMT**, especialmente útil para datos.
2. ***Booch Method***, especialmente útil para sistemas concurrentes y de tiempo real, muy relacionado con el lenguaje Ada.
3. **OOSE** de Jacobson, guiado por los casos de uso.

Esto producía mucha confusión y una **guerra de los métodos**.

TDS

En 1994, [...] Booch, [...] Rumbaugh [...][y] Jacobson crean el **UML** [...], un lenguaje para visualizar, especificar construir y documentar modelos de un sistema de software desde una perspectiva orientada a objetos que busca eliminar confusión y reunir los puntos fuertes de cada método, y este es estandarizado por el **OMG** [(*Object Management Group*)] [...]. [...]

La notación UML también añadió mejoras, dio estabilidad al mercado y permitió mejores herramientas CASE. Permite modelar sistemas desde los requisitos hasta los ejecutables, es escalable a sistemas grandes y los pueden usar tanto personas como máquinas, debido a un equilibrio entre expresividad y simplicidad.

Una **vista** es una proyección de la estructura del sistema centrada en algún aspecto. Un **diagrama** es una representación gráfica de elementos de un modelo.

UML tiene 5 vistas, cada una con varios tipos de diagramas:

1. **Vista de casos de uso**, centrada el comportamiento, con diagramas de casos de uso.
2. **Vista de diseño**, centrada en el vocabulario del programa y su funcionalidad, con diagramas de clases, de estados y de interacción.
3. **Vista de procesos**, con los mismos diagramas que la de diseño pero centrada en el funcionamiento, la escalabilidad y el rendimiento.
4. **Vista de implementación**, con diagramas de estados, de interacción y de componentes.

5. **Vista de despliegue**, con diagramas de estado, de interacción y de despliegue.

Muchas empresas de software hacen poco o ningún modelado, pues este requiere un proceso de desarrollo, personas formadas en las técnicas, tiempo y herramientas.

1.5. Fases del desarrollo

Independientemente del área o la complejidad del proyecto, el desarrollo de un sistema estará al menos en uno de los siguientes procedimientos genéricos:

1. **Definición**, **análisis** o **qué hacer:** Se analiza cuál es la funcionalidad del sistema, qué información debe manejar y las restricciones de diseño, y se planifica el proyecto.

2. **Desarrollo** o **cómo hacerlo:** Consta del **diseño** de las estructuras de datos, los algoritmos, las interfaces y la forma de pasar del diseño al lenguaje de programación y hacer la prueba; la **implementación** o **codificación** de los programas, incluyendo documentación, y la **prueba**.

3. **Mantenimiento** o **gestión de cambios:** Una vez construido y desplegado el software, este se mejora, se adapta a nuevas necesidades de los usuarios y se corrigen los fallos que se encuentran. Es donde suele recaer la mayor parte del coste del sistema.

1.6. Responsabilidad ética

Esta cobrando más interés en los últimos años el que la responsabilidad de los ingenieros de software no es exclusivamente técnica, sino que también hacia la profesión y la sociedad. Hay áreas en las que la conducta aceptable, si los ingenieros quieren ser respetados como profesionales,[1] no está limitada solo por la ley sino también por una noción de **responsabilidad profesional**, con mandamientos como los siguientes:[2]

1. Respetarás la confidencialidad de tu jefe o clientes, aunque no hayas firmado un contrato de confidencialidad.

2. No aceptarás conscientemente trabajo fuera de tu competencia.

3. Conocerás las leyes de «propiedad intelectual» y protegerás la «propiedad intelectual» de tus jefes y clientes.

4. No usarás de forma incorrecta los ordenadores de otros. Por ejemplo, no jugarás con un ordenador de «tu» empresa ni diseminarás virus.

ACM e IEEE tienen un código de ética y conducta profesional que hay que aceptar para ser miembro de dichas organizaciones, y que ha sido adoptado por SISTEDES. El principio 8 establece como obligación el aprendizaje continuo a través de toda la vida profesional.

[1] Es decir, por sus clientes o jefes.

[2] Un código ético no debe tomarse de forma dogmática, sino que debemos usar la razón para decidir si este tiene razón o no y en qué contextos. Por ejemplo, algunos de los siguientes pueden contradecir el imperativo moral del software libre según del contexto. Por otro lado, un código ético es necesariamente incompleto, y debemos esforzarnos por determinar qué es bueno o malo y actuar en consecuencia. Una forma de hacerlo es no aceptar trabajar en proyectos o empresas cuyos fines probables sean moralmente cuestionables si hay alternativas viables.

Capítulo 2

Casos de uso

Un **sistema** es un conjunto de cosas relacionadas que contribuyen a un objetivo. Una información es un conjunto de datos (registros de hechos, acontecimientos, transacciones, etc.) procesados y puestos en contexto para que resulten útiles o significativos para el receptor. Un **sistema de información** es un conjunto de personas y equipos con el objetivo de dar la información adecuada para apoyar las operaciones, la administración y la toma de decisiones de una organización, con calidad suficiente, a la persona adecuada, en el momento y lugar oportunos y con el formato más útil para el receptor. Está formado por:

Subsistema físico Transforma un flujo de entradas físicas en uno de salidas físicas.

Subsistema de decisión Regula y controla el sistema físico, decidiendo su comportamiento según los objetivos marcados.

Subsistema de información Conecta los otros dos, almacenando y tratando la información del sistema físico para hacerla disponible al de decisión.

Toda organización tiene un sistema de información, que es **automatizado** si usa la informática para mejorar su valor, simplificando o automatizando procesos o proporcionando información que facilita la toma de decisiones, y mejorando así en eficacia, coste y calidad.

Ejemplos de sistemas de información automatizados son los sistemas **ERP** (*Enterprise Resource Planning*), los de flujo de trabajo y los de comercio electrónico.

2.1. Requisitos

Un **requisito** es una condición o capacidad que necesita un usuario para resolver un problema o conseguir cierto beneficio, o que debe cumplir un sistema para satisfacer un contrato, norma u especificación.

Según el ISO/IEC/IEEE 29148:2011, los requisitos deben ser necesarios, independientes de la implementación, no ambiguos, consistentes, completos (sin partes por determinar), singulares, viables, trazables (que se sepa por qué se ha añadido) y verificables, y los conjuntos de requisitos deben ser completos (que no falten requisitos), consistentes, abordables y limitados (no abiertos, terminados).

Un requisito puede ser:

- **Funcional**, si describe la funcionalidad del sistema y cuáles son las acciones fundamentales, con sus entradas, funciones de transformación, salidas y excepciones. Suele empezar por «El sistema deberá».
- **No funcional**, si describe cómo funciona, incluyendo aspectos de accesibilidad, compatibilidad, confidencialidad, disponibilidad, eficiencia, escalabilidad, extensibilidad, fiabilidad, forma, interfaz gráfica, mantenibilidad, privacidad, seguridad, usabilidad, etc.

  Una **restricción de diseño** es un requisito no funcional sobre un aspecto de implementación como el lenguaje de programación usado, la plataforma o las herramientas.

La **ingeniería de requisitos** es el estudio de las necesidades de los usuarios para llegar a una definición de los requisitos de un sistema hardware o software, y el refinamiento de dichos requisitos. Consta de:

1. **Extracción**, «**elicitación**» o **identificación y consenso de requisitos:** Se descubren y articulan los requisitos, usando técnicas de recogida de información como entrevistas para obtener «requisitos informales».
2. **Análisis y negociación:** Razonamiento sobre los requisitos, búsqueda de inconsistencias, etc., para obtener requisitos acordados.
3. **Especificación:** Se redactan los requisitos en documentos de requisitos, en lenguaje natural o con alguna técnica como casos de uso o historias de usuario. Se crean una **Especificación de Requisitos del Software** (**ERS** o **SRS**, *Software Requirements Specification*) y, en su caso, un **SyRS** (*System Requirements Specification*).
4. **Validación:** Los clientes o usuarios comprueban que los requisitos son correctos.

2.2. Casos de uso y actores

Un **caso de uso** (***use case***) es una especificación de una secuencia de acciones, incluyendo variantes, que el sistema puede ejecutar, y que produce un resultado observable de valor para alguien. Solo especifican acciones observables desde fuera del sistema, y representan requisitos funcionales de este.

El modelado de casos de uso permite la recolección y especificación de requisitos. Los casos de uso son fáciles de comprender y validar por los usuarios y pueden guiar el proceso de desarrollo, ayudando a la planificación y el desarrollo incrementales y el diseño de la interfaz de usuario.

Un **actor** es un conjunto coherente de roles que asumen los usuarios (personas, dispositivos u otros sistemas) al interactuar con el sistema, sin formar parte de este. Los casos de uso los inicia un actor. Un usuario puede asumir distintos roles. Puede haber actores que especialicen a otros, heredando sus roles. Un actor puede intervenir en varios casos de uso y varios actores pueden intervenir en el mismo.

Un actor que interviene en un caso de uso es **primario** en este si es el que requiere al sistema el cumplimiento del objetivo, y es **secundario** si no es primario, sino que el sistema los necesita para satisfacer el objetivo. Cuando un caso de uso se inicia automáticamente, su actor primario es «Sistema» o «Tiempo».

Normalmente los sistemas tienen casos de uso de inicialización, pero no se suelen representar estos ni los casos de uso CRUD (acceso y manipulación de datos simple).[1]

2.3. Granularidad

Un **proceso de negocio elemental** es una tarea realizada por una persona en un lugar determinado, en un tiempo no muy largo (de unos segundos a un par de horas), con un cierto objetivo, y que añade un valor para el negocio y deja los datos en un estado consistente.

La granularidad de los casos de uso puede ser:

- De **organización** en **casos de uso del negocio**, formados por procesos de negocio elementales, que definen objetivos estratégicos de la empresa de mucho valor.
- De **sistema** u **objetivos del usuario** en **casos de uso** (**del sistema software**), que representan funcionalidades, tareas de usuarios o procesos de negocio elementales. Es lo que usaremos.
- De **subfunciones**, o pasos en las descripciones de casos de uso del sistema. No representan objetivos.

2.4. Descripción

Los casos de uso tienen secuencias de interacciones, **instancias**, **alternativas** o **escenarios**, que llevan al **éxito** si se satisface el objetivo del actor primario o al **fracaso** si no.

Los casos de uso tienen un **flujo** o **escenario principal** o **básico**, que conduce al éxito, y pueden tener **flujos alternativos** o **excepcionales** o **escenarios secundarios**, y se pueden representar con:

1. **Descripción breve**: nombre del caso de uso, actores involucrados indicando cuál es el primario o **iniciador**, y un párrafo en lenguaje natural.
2. **Descripción informal**: descripción en lenguaje natural del flujo principal y de cada flujo excepcional.
3. **Descripción completa** o **expandida**, mediante una plantilla.

Un flujo de eventos se puede representar como:

1. Texto estructurado informal.
2. Texto estructurado formal, con precondiciones (que deben cumplirse antes de poder ejecutarse el flujo) y postcondiciones (que pasan a cumplirse al terminar).
3. Notaciones gráficas, como diagramas de secuencia o de estados.

[1] Como excepción, sí se representa que una junta vecinal pueda consultar los katas registrados en una aplicación de artes marciales.

La especificación de casos de uso debe ser comprensible para un usuario no experto para que este pueda validarla, y debe indicar el inicio y el final, los actores, los objetos que intervienen, el flujo principal y los flujos excepcionales.

Algunas plantillas son:

- **Plantilla de Coleman.** Las partes entre corchetes son opcionales.

  Caso de uso Identificador e historia de revisiones.

  Descripción Objetivo a conseguir.

  Actores Lista de actores, primario y secundarios.

  Asunciones Precondiciones del caso de uso.

  Pasos Lista de interacciones entre los actores y el sistema para alcanzar el objetivo.

  [Variaciones] Lista de variaciones en los pasos que llevan a escenarios alternativos, con lo que ocurre para que se ejecute la variación y la lista de pasos a partir de ahí.

  [No funcional] Requisitos no funcionales asociados.

  [Cuestiones] Lista dinámica de cuestiones por resolver.

  Las listas de pasos pueden contener pasos con subpasos, y el texto del paso puede ser algo como «REPETIR», «EN PARALELO», etc. indicando cómo deben ejecutarse los subpasos.

  Pueden representarse numeradas en una columna o dividirse en dos columnas «Acción del actor» y «Responsabilidad del sistema» para distinguir las acciones de los actores y del sistema, alternando texto en una columna y en la otra y dejando hueco en una columna cuando haya texto en la otra «al lado», salvo que las acciones se hagan en paralelo.

- **Plantilla de Cockburn.**

  Sistema El que implementará el caso de uso.

  Actor principal

  Objetivo

  Lista de pasos.

  Extensiones Flujos alternativos, como las «Variaciones» de la plantilla de Coleman.

  Variaciones Lista de conceptos que pueden concretarse de varias formas, con el nombre del concepto y una lista de las formas a soportar.

- **Plantilla de Larman.**

  Actor principal

  Personas involucradas e intereses Lista con los tipos de personas u organizaciones, actores o no, con intereses en la transacción, y lo que quieren conseguir con ella.

  Precondiciones

  Postcondiciones

  Escenario principal

Extensiones

Requisitos especiales No funcionales.

Tecnología y variaciones de datos Lista de tecnologías que se usan en el caso de uso con sus formas concretas a usar.

Frecuencia Con la que se ejecuta.

Cuestiones abiertas

2.5. Limitaciones

No toda la funcionalidad se puede asignar a un solo caso de uso, por lo que se asigna a un requisito funcional o una **característica del sistema** (***system feature***), servicio observable externamente proporcionado por el sistema que satisface una necesidad de un interesado, que se expresa como declaración breve. Es deseable no tener más de 10.

Hay sistemas, como los servidores de protocolos de Internet, que no tienen escenarios de interacción ricos, por lo que puede ser preferible usar características a casos de uso.

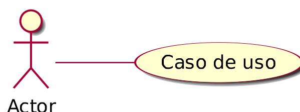
Las operaciones CRUD son numerosas, rutinarias y sin escenarios ricos asociados. Generalmente se indican como requisitos funcionales en la Especificación Complementaria, sobre todo las consultas dado que no cambian el estado del sistema, pero a veces son importantes en el dominio de la aplicación y se añaden como casos de uso. En tal caso se puede indicar un solo caso de uso «gestionar objeto», que agrupa todas las operaciones CRUD sobre un tipo de objeto de negocio, o uno por operación a representar, según el contexto.

2.6. Colaboraciones

Un caso de uso se implementa a través de una **colaboración**, una interacción entre objetos software para realizar un comportamiento deseado, con una **parte estática** formada por diagramas de clases y una **parte dinámica** formada por diagramas de secuencia o de colaboración. El objetivo es encontrar el conjunto mínimo de colaboraciones, bien estructuradas, que satisfagan el comportamiento de todos los casos de uso.

2.7. Diagramas de casos de uso

Son diagramas UML que dan una visión general de los casos de uso del sistema, aunque lo importante de los casos de uso sigue siendo su descripción textual. Los casos de uso se representan como elipses con el nombre del caso de uso, que generalmente empieza por un infinitivo, conectados por **asociaciones** a los actores involucrados, representados por monigotes.



Un cuadrado rodeando un conjunto de casos de uso, y posiblemente etiquetado, representa los límites del sistema. La herencia de actores se representa como en los diagramas de clases, normalmente con el actor «padre» arriba.

El actor primario se suele representar a la izquierda del caso de uso. Los secundarios no suelen aparecer, pero de hacerlo suelen estar a la derecha.

Una elipse punteada representa una colaboración, o el subsistema correspondiente, y se conecta con una **realización** (una flecha como la de herencia pero punteada) con un caso de uso.

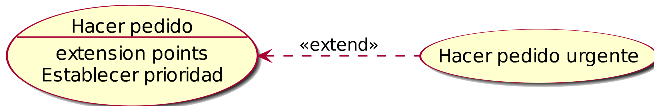
Hay 3 tipos de relación entre casos de uso:

1. **Generalización:** Un caso de uso hereda el significado de otro y lo concreta. Se representa como la herencia de clases.

2. **Inclusión:** Un caso de uso incorpora el comportamiento del otro dentro de su secuencia, para evitar repetirlo. Flecha punteada al otro con *«include»*.

3. **Extensión:** Un caso de uso tiene el comportamiento de otro, con pasos adicionales en **puntos de extensión** especificados por el otro, para modelar una parte opcional o uno o varios subflujos que se ejecutan bajo ciertas condiciones. Flecha punteada al otro con *«extend»*.

Un caso de uso puede especificar sus puntos de extensión con una línea horizontal debajo de su nombre, *extension points* y la lista de puntos.



2.8. Obtención de casos de uso

Un caso de uso es **esencial** si es neutro respecto a tecnología y diseño, y es **real** si describe el proceso según su diseño actual, teniendo en cuenta la tecnología de E/S usada. Se debe empezar especificando casos de uso esenciales y, durante el desarrollo, si es preciso, añadir detalles hasta convertirlos en casos de uso reales.

Para obtener los casos de uso, identificamos los usuarios del sistema y los roles relevantes que asumen. Entonces identificamos las formas de interactuar con el sistema de cada rol y creamos un caso de uso por forma. Finalmente estructuramos los casos de uso y los revisamos y validamos con el usuario.

Para esto debemos establecer los objetivos del proyecto, definir bien los límites del sistema y llevar cuidado con la ambigüedad y con contradicciones entre casos de uso, sin preocuparse demasiado por la notación.

Los casos de uso no deben estar ligados a elementos concretos de la interfaz de usuario o incluir detalles sobre estos. Se debe llevar mucho cuidado de no abusar de relaciones de inclusión y extensión de casos de uso, y no hacer descomposiciones funcionales de requisitos. Se puede evitar la extensión sustituyéndola por una inclusión o incluyendo toda la funcionalidad en el escenario alternativo.

Lo importante de los casos de uso no es el diagrama sino la descripción textual.

Cada empresa de desarrollo de software debería tener un manual sobre utilización de casos de uso, indicando, al menos:

- Forma de identificación.
- Granularidad, por ejemplo la de los casos CRUD. En clase diremos de juntar los casos de creación y consulta de datos en uno de gestionar, salvo si los casos separados son relevantes, pero luego crearemos un caso de uso «Consultar katas».
- Herramientas para la gestión, como procesadores de textos o **herramientas de gestión de requisitos** o **CARE** (*Computer-Aided Requirements Engineering*).
- Plantillas seleccionadas.
- Trazabilidad con otros artefactos del desarrollo: tipos de trazas y herramientas.
- Normas de estilo en los diagramas.

2.9. Especificación por casos de uso

El comportamiento requerido del sistema es conjunto completo de sus casos de uso. Una SRS puede estar formada por un diagrama de casos de uso del sistema software, un modelo conceptual del dominio y, para cada caso de uso, una descripción textual mediante una plantilla y descripciones de las interfaces de usuario, más una sección de requisitos no funcionales.

Los casos de uso ofrecen un medio sistemático, sencillo e intuitivo para capturar los requisitos funcionales de un sistema; dirigen el proceso de desarrollo, y fomentan la trazabilidad entre modelos.

Actualmente no hay consenso sobre el uso de casos de uso para capturar requisitos y guiar el modelado. Hay confusión sobre cómo usarlos y los métodos ágiles no los promueven.

Tradicionalmente se usan con programación orientada a objetos, pero esto no es necesario, y no es adecuado porque los casos de uso favorecen un enfoque funcional, se centran en las secuencias de acciones y se basan en los escenarios actuales del sistema. Así, aunque los casos de uso son útiles para validación, de usarse para desarrollo orientado a objetos debe hacerlo un equipo experto.

Capítulo 3

Modelado conceptual

El **modelado conceptual** o **del dominio** representa el vocabulario del dominio, que está formado por elementos del mundo real, no necesariamente físicos, como ideas, conceptos y objetos, y no por elementos software como clases, bases de datos, ventanas, métodos, etc.

Es un diccionario visual de las abstracciones relevantes sobre el dominio, ignorando detalles sin interés, para comunicarse con los clientes y usuarios, y se representa con diagramas de clases.

La primera iteración es muy simple, y luego se va refinando.

3.1. Clases conceptuales

Representan entidades y conceptos mediante un **símbolo** (palabra o imagen) que representa a la clase; una definición o **intensión**, y un conjunto de ejemplos a los que se aplica o **extensión**.

En UML, se representan como clases. Los nombres de clases y asociaciones usan *CamelCase* y van en singular, y los nombres de atributos y operaciones van en *lowerCamelCase*. Todos los nombres del modelo deben ser distintos.

Se pueden incluir sistemas externos, actores o partes del sistema, aunque el último caso requiere más atención en la traducción al diseño. Los conceptos «raíz», de los que solo hay una instancia en el dominio a considerar, se añaden solo si son útiles para entender el dominio y están relacionados con lo más relevante, pero nunca se añade un concepto «Sistema».

Los informes, como recibos de compra, se añaden como conceptos si tienen información que no está en otros sitios o si tienen identidad propia que permite hacer otras operaciones con ellos aparte de consultarlos. Los catálogos solo se incluyen si proporcionan un servicio no trivial accesible desde todo el sistema.

Podemos obtener clases conceptuales de los conceptos usados en los casos de uso o, si se hace modelado del negocio, de la información de entrada y salida de las actividades del diagrama de proceso. De la especificación del diccionario se pueden obtener atributos, relaciones y restricciones. No añadimos conceptos que no estén en el dominio.

Podemos usar una lista de categorías de clases para hacer una identificación inicial y luego refinarla de forma iterativa. Consideramos especialmente los conceptos de los que se guarda información, que dan servicio a otros conceptos o que tienen atributos múltiples, y las entidades

externas o dispositivos que consumen o generan información. Es preferible que sobren clases a que falten.

El **enfoque lingüístico** consiste en:

1. Identificar candidatos a conceptos mediante heurísticas.

   Por ejemplo, los nombres comunes y frases nominales son conceptos, los nombres propios son instancias de conceptos, los adjetivos que acompañan a nombre son valores de atributos, las enumeraciones de nombres pueden ser subclases de una misma clase, y las formas verbales como «es un tipo de», «forma parte de», etc. identifican relaciones.

2. Encajar los candidatos en categorías de una lista.

3. Identificar sinónimos, homónimos y polisemias, y actualizar el diccionario de datos en consecuencia.

4. Eliminar conceptos sobre la implementación o sin relación con otros.

Esto da problemas por la ambigüedad y la imprecisión del lenguaje.

A veces queremos representar especificaciones de conceptos como conceptos (modelos de coches, libros sin indicar el ejemplar, etc.). Entonces podemos modelar las instancias de esos modelos o no con otro concepto (coche, libro, etc.), y lo hacemos o no según la importancia y si las instancias tienen identidad propia.

3.2. Relaciones

Las clases conceptuales se relacionan de forma física o lógica.

Una **asociación** es una relación entre dos o más clases conceptuales. Se incluyen las asociaciones cuyos elementos el sistema deba mantener durante algún tiempo, y se excluyen las que son suficientemente momentáneas para que el sistema no tenga que guardarlas, las irrelevantes para la especificación, las orientadas a la implementación y las derivables a partir de otras.

Ejemplos son «ser parte física/lógica de», «estar físicamente/lógicamente contenida en», «ser descripción/línea de transacción/informe de», «ser registrado en», «ser miembro de», «ser parte organizacional de», «usar», «comunicar con», «estar relacionado con», «ser una transacción relacionada con otra», «estar al lado de», «ser propiedad de» o «ser evento relacionado con».

Dada una asociación R entre las clases $C_1, \dots, C_n$, la multiplicidad de una posición $i \in \{1, \dots, n\}$ de R es el conjunto de posibles valores de $|\{a_i \in C_i \mid (a_1, \dots, a_n) \in R\}|$ para cada $a_1, \dots, a_{i-1}, a_{i+1}, \dots, a_n$ fijos en cada estado válido del sistema. La multiplicidad representa el estado actual, y si hace falta un histórico, este se especifica de otra forma.

En UML, las asociaciones binarias se representan como una asociación sin navegabilidad, y el resto como un rombo conectado con una línea por posición a la clase correspondiente, posiblemente con multiplicidad y nombre de atributo en el lado de la clase. Generalmente podemos convertir las asociaciones no binarias en varias asociaciones binarias añadiendo una clase para la relación, y esto es lo más recomendable.

El nombre de atributo indica el rol de la posición en la asociación. Una asociación es **reflexiva** si involucra a una clase en más de una posición, en cuyo caso las posiciones referentes a la misma clase que otras posiciones deben indicar el rol.

Una **clase asociativa** es una que indica datos adicionales sobre una asociación, a la que se une con una línea punteada. Es fácil confundir una asociación entre dos clases A y B con clase asociativa C con una asociación entre A y C y otra entre C y B.

Encontrar clases conceptuales es más importante que encontrar asociaciones y demasiadas asociaciones dificultan la comprensión de los diagramas. Por otra parte, el exceso o falta de asociaciones puede dificultar la implementación, por lo que nos centramos en las asociaciones en las que una parte necesita conocer a la otra.

Generalmente añadimos multiplicidades siempre, así como flechas con la dirección de lectura del nombre de relación, y roles solo cuando hace falta.

Evitamos asociaciones derivadas o redundantes.

3.3. Agregaciones

TDS

- **Agregación**: Asociación en que un objeto [es miembro de un grupo, componente físico de otro objeto o está contenido en un contenedor, y hay propiedades y operaciones que se propagan del todo a las partes][...]. [...] En la referencia de la parte al todo se asume multiplicidad 1. [...]
- **Composición**: Agregación en que cada parte pertenece a un único agregado y si se elimina un agregado se eliminan todas sus partes. [...] La multiplicidad [del][...] agregado es siempre 1.

Solo usamos composición cuando hay una restricción para la existencia de las partes, tiene sentido que el creador de las partes en la implementación sea el agregado y las operaciones del agregado se propagan con frecuencia a las partes. Ante la duda es mejor usar asociaciones normales.

3.4. Herencia

Una clase conceptual A **hereda** de otra B si todo concepto que cumple la definición de A cumple la de B, formando una relación de orden parcial. La herencia se representa como la de clases (excluyendo la reflexiva y la transitiva) y permite gestionar la complejidad cuando varios conceptos tienen una definición consistente pero uno es más concreto.

Podemos usar herencia mediante:

- **Generalización**, abstrayendo conceptos parecidos en uno más general al que llevar las características comunes, cuando hay conceptos con atributos y asociaciones comunes que son variaciones de un concepto similar.

  Una clase conceptual es **abstracta** si toda instancia de esa clase lo es de alguna subclase suya, lo que indicamos poniendo el nombre de la clase en cursiva.

- **Especialización**, descomponiendo un concepto en subtipos con características propias, cuando distintas instancias tienen distintas propiedades y asociaciones, no solo en valor.

Bajo una clase con subclases podemos añadir un texto de la forma {[disjoint|overlapping], [complete|incomplete]} para restringir la jerarquía, donde *disjoint* indica que las subclases representadas son disjuntas dos a dos y *complete* indica que todo elemento de la clase lo es de alguna de las subclases representadas.

El estado de un objeto no es un subtipo de su concepto, aunque si es importante se puede modelar como subtipo de otro concepto con el que el primero se asocia.

3.5. Atributos

Son propiedades relevantes de los conceptos cuyos valores no tienen identidad. Se añaden a las clases con formato «- [/]*nombre*[: *tipo*][[*multiplicidad*]][= *valorInicial*]», donde *multiplicidad* por defecto es 1 y / indica que el atributo es derivado de otros valores (por defecto no lo es). Las partes opcionales no se añaden si no es necesario. Los atributos derivados solo se incluyen si son importantes, y se pueden almacenar o no.

Una propiedad sin identidad puede modelarse como una clase si está formada por varias partes (como nombre y apellidos o cantidad y unidad), tiene operaciones no triviales (como validación), tiene otros atributos secundarios o es una abstracción de varios tipos. Una propiedad en un «dominio», o conjunto de valores predeterminado, se representa como atributo o concepto según su importancia en el contexto del sistema.

No usamos atributos como claves ajenas. Si un atributo puede ser un concepto por tener «identidad propia», lo representamos como concepto.

3.6. Restricciones

Si son importantes, podemos añadir restricciones en el dominio mediante notas o explícitamente en los elementos. Algunas están predefinidas:

- Una etiqueta {ordered} en una posición i de una relación indica que, fijadas instancias en el resto de posiciones, las de i que se asocian con el resto están bien ordenadas por un orden que forma parte de la relación.

- Una flecha etiquetada con {xor} de una asociación a otra que comparten una única clase en una única posición cada una indica que las instancias de dicha clase se relacionan con una o con otra, y no con ambas.

- Si A es una asociación entre clases $A_1, \dots, A_n$ y B lo es entre $B_1, \dots, B_n$ y cada B_i hereda de A_i, una etiqueta {subsets A} en B indica que siempre es $B \subseteq A$. Esta estructura es preferible a tener una sola asociación con una clase asociativa.

Capítulo 4

Modelado de negocio

4.1. Diagramas de actividades

Una **actividad** es una acción realizada por un actor que produce un cambio en este o devuelve algún valor. Llamamos **estado acción** a una computación atómica y **actividad** a una actividad compuesta de otros estados acción y actividades.

Un **diagrama de procesos** o **de actividades** es un diagrama UML que muestra un flujo de actividades. Se usa para modelar flujos de trabajo, de datos, casos de uso y en programación concurrente o paralela.

Las actividades se representan con un cuadro redondeado con el nombre de la actividad. Una **transición** indica que después de una actividad se ejecuta otra y se expresa con una flecha hacia abajo de la primera a la segunda.

Toda actividad del es el inicio de una transición y el final de otra, y el flujo viene dado por una sucesión de actividades a partir de un punto de inicio, un círculo relleno único en el diagrama con una transición hacia la primera actividad, y uno de fin, un círculo relleno rodeado por una circunferencia concéntrica con una transición desde la última actividad del flujo.

Una transición puede ir hacia una **bifurcación**, un rombo con transiciones hacia dos o más actividades de las que se escoge una según una condición, indicada entre corchetes como etiqueta de la flecha. Una línea horizontal hacia la que va una transición y de la que parten varias es una **división**, que indica que el flujo se divide en varios flujos paralelos, y una a la que llegan varias y de la que parte una es una **unión**, que une los flujos paralelos en uno solo. El diagrama puede tener **calles**, divisiones verticales separadas con líneas que en la parte superior indican el nombre del actor que realiza las actividades de dicha calle.

Un objeto se representa como en un diagrama de objetos, pero cambiando la definición de atributos por el estado del objeto entre corchetes. Le llega una flecha punteada desde una actividad para indicar que la actividad establece el estado del objeto al indicado (puede que cree el objeto), y puede partir de él una flecha punteada a otra actividad indicando que esta recibe el objeto. Un mismo objeto puede aparecer varias veces si varias actividades establecen su estado. Es una parte poco importante del diagrama.

Normalmente se hace un solo diagrama de proceso para todo el sistema.

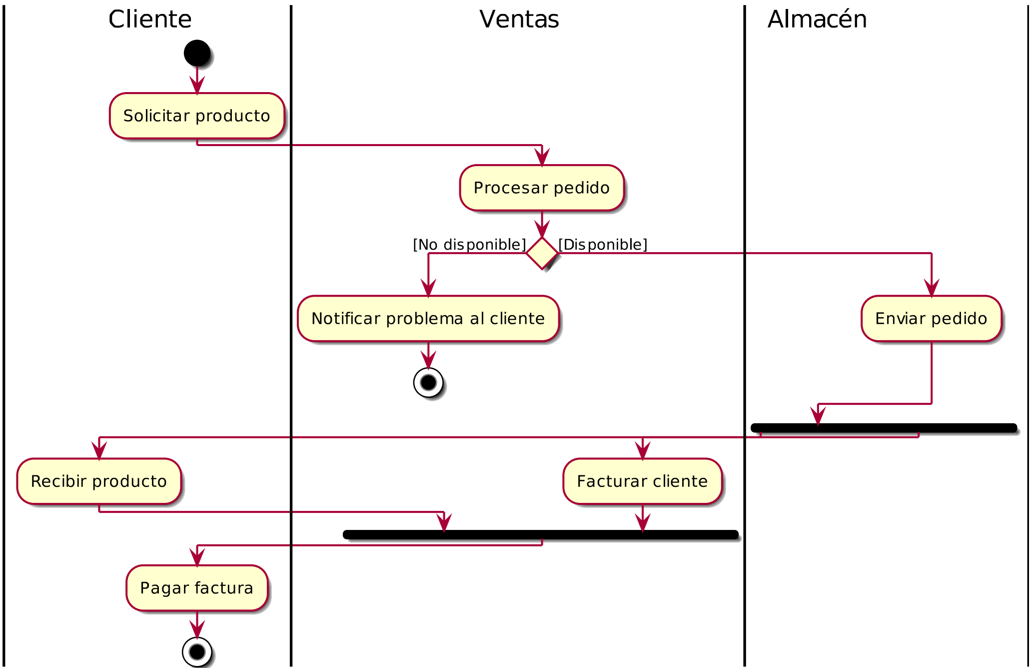


Figura 4.1: Ejemplo de diagrama de actividades.

4.2. Modelado del negocio

Una organización tiene una serie de objetivos que satisface a través de **procesos de negocio**, formados por un flujo de tareas, agentes, información y reglas, llamadas **reglas de negocio**, que describen restricciones y comportamientos en el dominio y, aunque no son requisitos, influyen en ellos.

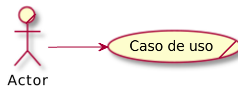
Las reglas pueden ser:

1. **De restricción:** Rigen la estructura y el comportamiento de la información.
    a) **Estímulo-respuesta:** Cuando ocurre un evento, se ejecuta una operación.
    b) **De operación:** Precondiciones y postcondiciones para ejecutar una operación.
    c) **De estructura:** Estructura de los objetos, incluyendo propiedades, asociaciones e invariantes internos.
2. **De derivación:** Permiten inferir nuevos hechos a partir de otros.

El **modelado del negocio** consiste en:

1. Identificar los objetivos estratégicos de la organización y sus subobjetivos.
2. Definir el proceso de negocio asociado a cada subobjetivo y los roles implicados.
3. Definir un caso de uso del negocio para cada proceso de negocio y crear un diagrama de casos de uso del negocio.
4. Modelar el flujo de tareas asociado a cada proceso de negocio mediante **escenarios** (diagramas de secuencia) y diagramas de procesos.
5. Especificar las informaciones y actividades incluidas en cada diagrama de actividades.

Los casos de uso del negocio se representan con una barra abajo a la derecha, y los actores de negocio con la cara tachada.



Un ***worker*** es un actor interno al sistema u organización, y se representa como un actor encerrado en un círculo con una flecha a la izquierda en el borde de arriba y una barra abajo a la derecha.

Podemos especificar un caso de uso del negocio textualmente, como con una plantilla, o con un diagrama de proceso.

4.3. Diagramas de roles

Sirven para modelar los **agentes** o roles que participan en los casos de uso del negocio. Los actores se representan como en los diagramas de casos de uso y entre ellos hay líneas con una multiplicidad a cada lado que representan canales de comunicación entre los actores.

4.4. Conversión a un modelo de requisitos

En general, podemos aproximar un diagrama de casos de uso a partir de uno de procesos convirtiendo cada actividad en un canal en un caso de uso cuyo actor primario es el indicado por el canal. Los objetos del diagrama de procesos se convierten en conceptos del dominio.

Un documento de análisis de requisitos puede estar formado por:

1. **Visión:** Perspectiva del producto, objetivos, beneficios, características, coste.

2. **Especificación de casos de uso.** La descripción de casos de uso requiere comunicación con el usuario, y si el caso de uso es complejo se pueden crear varias versiones del mismo para añadir complejidad de forma incremental.

3. **Especificación complementaria:** Objetivos y reglas de negocio; funcionalidad que abarca varios casos de uso o que no se representa como caso de uso (por ejemplo, funcionalidad CRUD); requisitos no funcionales.

4. **Glosario:** Definiciones de términos.

Capítulo 5

Modelado de diseño

El análisis estudia lo que hay que hacer, no cómo hacerlo, y enfatiza la investigación del problema para llegar a unos requisitos que deben cumplir las soluciones. El diseño traduce esto en una representación del software, una solución conceptual que satisface los requisitos. Mientras que el análisis orientado a objetos estudia un problema o un sistema según los conceptos del dominio, como clases conceptuales, asociaciones y cambios de estado, el diseño orientado a objetos especifica una solución según conceptos del software como clases, atributos, métodos y colaboraciones.

Un **diseño preliminar** es una visión ideal del sistema obtenida a partir de los modelos de caso de uso y del dominio, que define los subsistemas del software mediante paquetes sin tener en cuenta restricciones tecnológicas ni requisitos no funcionales. Esta se refina en un modelo del diseño completo añadiendo aspectos relacionados con la plataforma tecnológica concreta, patrones de diseño y requisitos no funcionales como los de rendimiento y uso de memoria.

5.1. Diagrama de secuencia del sistema (DSS)

Es un diagrama de secuencia UML que muestra los eventos generados por un actor durante un escenario de un caso de uso y las posibles comunicaciones con sistemas externos.

Los sistemas se tratan como cajas negras, teniendo claros sus límites, y se representan como objetos. El actor se nombra como objetos y se representan con un monigote. No se suele indicar el nombre del objeto.

Normalmente se hace un DSS para el escenario principal de un caso de uso y uno para cada escenario alternativo o frecuente, y se elige que el sistema sea el propio software.

Cuando el caso de uso se inicia automáticamente, lo inicia un actor de tipo Sistema, como en la figura 5.1. Los argumentos de los mensajes deben ser valores primitivos, no colecciones (salvo cadenas de caracteres, consideradas primitivas) ni objetos del dominio. Para referirse a objetos del dominio se usan identificadores.

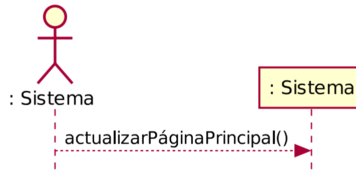


Figura 5.1: DSS de una interacción iniciada automáticamente.

5.2. Contratos

Cada evento en un DSS da lugar a una operación a implementar. Un **contrato** describe una operación mediante:

1. **Precondiciones**, condiciones que se deben cumplir para poder ejecutar la operación. Generalmente, la declaración del contrato ya restringe los valores que pueden tomar los argumentos mediante tipos.

2. **Postcondiciones**, valores devueltos si los hay y cambios en el dominio resultantes de ejecutar la operación. Los cambios se suelen indicar como creación y eliminación de objetos con ciertos atributos, creación y eliminación de asociaciones y modificación de atributos, y entendiendo que lo que no se especifica queda constante.

Los contratos sirven cuando hay mucha complejidad y se necesita precisión, o cuando la operación no están clara, pero normalmente no hacen falta ya que la mayoría de detalles se pueden inferir obviamente de los casos de uso.

Una posible plantilla es:

Nombre *nombre_operación*([*argumento*: *Tipo*, ...])

Referencias cruzadas Opcional, lista de casos de uso en los que puede darse la operación.

Precondiciones Suposiciones relevantes. Se pueden nombrar objetos con notación matemática informal y luego usarlos en las postcondiciones.

Postcondiciones Cambios en el dominio, con predicados como «se ha creado un objeto tal» o similares.

5.3. Patrones GRASP

TDS

Los objetos de una clase conocen sus datos privados y realizan acciones sobre ellos, como cálculos o creación de otros objetos, y para ello pueden iniciar acciones en otros objetos y coordinarlas. Los **patrones GRASP** ayudan a [...] distribuir responsabilidades entre las clases [...].

Bajo acoplamiento [...] Hay acoplamiento entre las clases A y B si A posee un atributo de tipo B, tiene un método con algún parámetro o valor de retorno de tipo B, es subclase

directa o indirecta de B o implementa la interfaz B. Reducir el acoplamiento favorece la reutilización, la comprensión y el mantenimiento del código.

Alta cohesión La **cohesión** es el grado de relación entre los elementos de un mismo módulo, como los métodos de una clase, las clases de un paquete, etc. Aumentarla favorece la reutilización, la comprensión y el mantenimiento del código.

Tipos de cohesión, de mayor a menor:

Funcional Se hace una sola función.

Secuencial La salida de una tarea sirve como entrada a la siguiente.

Comunicacional Las actividades comparten datos.

Procedural Actividades distintas en las que el flujo de ejecución va de una a la siguiente.

Temporal Actividades distintas relacionadas por el tiempo.

Lógica Actividades de la misma categoría lógica.

Coincidental o casual Actividades distintas sin relaciones significativas entre ellas.

TDS

Experto [...] Se asigna una responsabilidad a la clase que tiene la información necesaria para cumplirla. Las responsabilidades se distribuyen de forma homogénea, evitando crear **clases dios** [...]. [...]

El **principio «no hables con extraños»** desaconseja enviar mensajes a objetos obtenidos de forma indirecta (a través de mensajes a otros objetos) [...].

Creador Una clase A tiene la responsabilidad de crear instancias de B si A es un agregado de instancias de B, contiene o registra instancias de B, hace un uso específico de instancias de B o proporciona los datos necesarios para inicializar un objeto de B.

Controlador La **separación modelo-vista** consiste en que las clases del modelo o dominio no conozcan a las de la vista [...][y las de la vista no incluyan ninguna lógica de negocio], lo que favorece la cohesión, permite desarrollar la vista y el dominio en paralelo y permite conectar otras vistas al modelo, incluso simultáneamente, y ejecutar el modelo en un proceso independiente a la capa de presentación.

Una forma de conseguir esto es con un **controlador**, una clase [...] intermediaria entre el [...] modelo y la vista, ofreciendo a la vista una interfaz simplificada del dominio para realizar ciertas tareas relacionadas de forma independiente a posibles cambios en el dominio.

Un **controlador fachada** no tiene estado y simplemente expone operaciones. A veces esto resulta en controladores «saturados» y debemos usar un **controlador caso de uso**, que se dedica a las operaciones de un solo caso de uso y puede tener estado.

TDS

Polimorfismo Cuando se requiere una misma funcionalidad de varias clases o clases no previstas, se define una interfaz [o superclase] que provea esa funcionalidad y se usan objetos de la interfaz [o superclase], facilitando añadir nuevas alternativas.

Indirección Cuando no sea deseable un acoplamiento directo entre dos clases, crear una clase intermediaria que proporcione una interfaz más adecuada a cada parte.[1] [Muchos patrones de diseño GoF, como adaptador, fachada o mediador, usan indirección.]

Variaciones protegidas Proteger a elementos del código de las variaciones de otros [...], por ejemplo mediante interfaces. [Es un principio fundamental y se puede usar, por ejemplo, para lectura de datos de fuentes externas o búsqueda de servicios.][...]

- **Principio de ocultación de la información** [...].
- **Principio abierto-cerrado** [...].

Servicios [o fabricaciones puras] Encapsulan un proceso o transformación en el dominio que no es una responsabilidad natural de otra clase. No tienen estado.

Algunos servicios aparecen en patrones de diseño, como pueden ser:

- Fábricas, para crear objetos complejos.
- **Repositorios** o catálogos, fachadas a una colección de objetos con identidad con métodos para buscar, guardar y borrar objetos, usadas para evitar el acoplamiento de las clases de dominio con la base de datos concreta.

  Se puede definir un repositorio por concepto o uno genérico (con plantillas), y la definición de las interfaces de los repositorios, pero no sus implementaciones, forman parte de la capa de dominio.

  En esta asignatura, el acceso a los repositorios se hace solo desde el controlador y los repositorios no crean los objetos.
- **Servicios de dominio**, que encapsulan reglas de negocio que afectan a varios conceptos.
- **Servicios de aplicación**, que encapsulan operaciones como accesos al exterior.

Si se abusa de los servicios, se crea un diseño centrado en procesos o funciones y no en objetos,[2] por lo que hay que vigilar si hay fabricaciones puras con un único método.

5.4. Colaboraciones

Una **colaboración** es una secuencia de intercambios de mensajes para implementar una operación, expresada con un diagrama UML de interacción. El diseño de colaboraciones es

[1] Teorema Fundamental de la Ingeniería de Software: Todo problema se puede solucionar con un nivel más de indirección, salvo el problema de demasiados niveles de indirección.

[2] La conclusión lógica de esto es que hay que abusar de los servicios para así favorecer la programación funcional.

la parte más difícil del análisis y diseño orientado a objetos y es el punto de partida para la programación.

Hay dos tipos de diagramas de interacción: los diagramas de secuencia, centrados en la cronología de los mensajes, y los de colaboración o comunicación, centrados en la organización estructural de los objetos.

En los de colaboración, podemos indicar la cronología mediante numeración secuencial o jerárquica. También podemos mezclar ambas numeraciones, usando la secuencial en general por ser más sencilla y la jerárquica en caso de ambigüedad, como al indicar el interior de bucles y operaciones condicionales. En diagramas de secuencia se puede hacer esto, pero la ambigüedad ya la resuelven las líneas de activación.

TDS

Tipos de mensaje:

- **Simple**: *método*([*argumento*, ...]).
- **De asignación**: *variable* := *método*([*argumento*, ...]).
- **De creación**: «create»[[(*argumento*, ...)]]. [...]
- **De destrucción**: «destroy». [...][No lo usaremos ya que Java tiene recolección de basura.]
- **De condición**: [*condición*] *mensaje*.
- **De iteración**: **mensaje*, [*multiplicidad*]*mensaje*.

En esta asignatura las colaboraciones consisten en programar en UML, por lo que añadimos una sintaxis para la iteración, [* *var* IN *colección* [(WHILE|UNTIL) *condición*]] *mensaje*. Realmente te puedes inventar la sintaxis siempre que se entienda.

Además, se pueden añadir notas en UML a los objetos para aclarar lo que ocurre entre mensajes en una interacción (llamadas a métodos estáticos, asignaciones de atributos, etc.).

Los objetos se representan con su tipo, pudiendo usar añadir el nombre como aclaración o si tenemos que referirnos a estos en otra parte. Si solo hace falta que el objeto cumpla cierta interfaz, aun cuando este reciba un mensaje «create», el objeto es el del tipo interfaz, y se añade encima el estereotipo «interface».

Se puede simplificar inventando operaciones de conveniencia en las clases e interfaces de Java, pero cuando un objeto se crea, este debe inicializar sus atributos y por tanto crear un objeto por atributo no primitivo cuyo valor no venga dado al constructor.

Para diseñar una colaboración, primero escogemos un controlador de acuerdo al contexto de la operación en el sistema. Este recupera de los repositorios los objetos correspondientes a los identificadores; delega la tarea según el patrón experto o el creador, y si hace falta guarda los objetos creados.

El creador será una clase conectada en el modelo de dominio a la clase de la instancia a crear o, si ninguna de las clases conectadas es apropiada, el propio controlador.

En el inicio de la aplicación, se suele crear un objeto de dominio inicial que se encarga de crear los objetos de dominio que dependen de él.

5.5. Modelo de clases de diseño

Un diagrama de clases puede ser conceptual (de conceptos), **de especificación** (de tipos que representan la Interfaz[3]) o **de implementación** (que representa las clases tal como se implementan).

El modelo de clases del diseño es de especificación y se crea a partir del modelo conceptual y las colaboraciones. Para empezar, se toma el diagrama de clases conceptual, usando los nombres de las clases del dominio para reducir el salto entre problema y solución, y se añaden las clases y los métodos usados en las colaboraciones y los atributos usados en los contratos.

Puede haber conceptos en el modelo conceptual que no aparezcan en el de diseño. Por ejemplo, los actores de los casos de uso suelen aparecer en el modelo conceptual, pero solo hay que añadirlos al dominio si incluyen información relevante, no simplemente credenciales de inicio de sesión, que no aparecen en el diagrama de especificación. Hay que sospechar de las clases que solo contengan métodos y no atributos, o al revés, salvo que sean el resultado de aplicar algún patrón de diseño o GRASP.

Los atributos de las clases de diseño tienen tipo y el «-» inicial se cambia por su **visibilidad**, - para privada, # para protegida, + para pública y ~ para nivel de paquete. También pueden tener propiedades indicadas al final con {*propiedad*}, como {readOnly} o {addOnly}. Los atributos de clase van subrayados, y los calculados se almacenan.

Las clases tienen operaciones con formato [*visibilidad*] nombre([*parámetro*: *Tipo*, ...])[: TipoDevuelto] ({*propiedad*})*, y propiedades como {isQuery}, {sequential} o {concurrent}.

Un objeto B es **visible** por uno A si B es un atributo de A, un parámetro de un mensaje recibido por A, un valor devuelto por una llamada hecha por A o accesible globalmente. Para que A envíe un mensaje a B, B debe ser visible por A.

Si A envía un mensaje a B, crea una instancia de B o necesita una conexión con B, A tiene **navegabilidad** con B. Las asociaciones tienen navegabilidad si, a través de ellas, los objetos de una clase tienen navegabilidad con los de la otra pero no al revés, lo que se indica con una punta de flecha en el extremo de la asociación al que se navega desde el otro.

Una **dependencia** indica que los cambios en la definición de una clase B pueden afectar a la implementación de A, lo que se indica con una flecha punteada de A a B. Solo se indican las dependencias importantes, y nunca si la visibilidad es de atributo.

Una **asociación cualificada** es una en que una de las partes accede a la otra a través de un diccionario o similar. Se indica situando un rectángulo entre la parte que accede así a la otra y su extremo de la asociación, sin espacio. El nombre de la clave, como parámetro, está dentro del rectángulo, y el valor es de la otra parte.

Una **interfaz** es un conjunto de operaciones que caracteriza el comportamiento de un elemento. Se define como una clase con estereotipo «interface» o con un pequeño círculo con el nombre de la interfaz.

Se puede, pero no se suele, indicar que una clase lanza una excepción añadiendo una dependencia con estereotipo «send» de esta clase a una con estereotipo «exception».

[3]Está en mayúsculas para indicar que se trata de una interfaz en el sentido de una serie de estructuras y funciones a implementar, no necesariamente a través de una interfaz Java.

5.6. Diagramas de estados

Complementan el modelo de datos y el funcional con una máquina de estados determinista que describe el comportamiento de las instancias de una clase.

Una transición entre estados se indica con una flecha entre los estados etiquetada con *evento* [[*condición*]] [/ *acción*], donde la *acción* es una expresión de código o un envío de mensaje de la forma ^*otro_objeto.mensaje*([*argumento*, ...]). Las transiciones son atómicas y consisten en que se recibe el evento y se ejecuta la acción correspondiente si la hay, mientras se cambia de estado.

Un estado se representa con un rectángulo redondeados con el nombre del estado, seguido opcionalmente por una o más de:

- Acciones de entrada, **entry** / *acción*, y de salida, **exit** / *acción*.
- Transiciones internas, indicadas solo con la etiqueta, que no cambian el estado.
- Subestados, con un diagrama de estados dentro del estado. Si aparece, el estado es **compuesto**.
- Actividades, en segundo plano, **do** / *acción*.
- Eventos diferidos, *evento* / **defer**.

Tipos de evento:

- **Llamada:** Se recibe un mensaje, *método*([*parámetro*, ...]).
- **Señal:** Se recibe una señal asíncrona, *señal*([*parámetro*, ...]).
- **Tiempo:** Pasa un tiempo absoluto o desde que se produjo un evento, normalmente la última transición del objeto, after (*tiempo_o_duración*), after *duración* desde *algo*.
- **Cambio:** Se cumple una condición, when (*condición*).

Los diagramas de estados tienen un **estado inicial**, único si no contamos subestados, un pseudoestado que se representa como en los diagramas de actividades y que tiene una única transición, saliente, con etiqueta opcional /*acción*.

Los estados compuestos tienen dentro un **estado final**, representado como en los diagramas de actividades y único si no contamos subestados de estados compuestos del subdiagrama, aunque puede representarse varias veces. No tiene transiciones salientes, y al entrar el objeto es susceptible de las transiciones salientes del estado compuesto. El diagrama principal puede tener un estado final, y al llegar a este se destruye el objeto.

Un estado compuesto puede tener, en vez transiciones salientes normales, una única transición saliente sin evento, que se ejecuta al llegar a su estado final.

Un diagrama de estados se puede usar para un caso de uso, una clase conceptual o de diseño o incluso un sistema completo.

Capítulo 6

Diseño lógico

La transformación del diseño al código es sencilla, transformado los elementos y asociaciones de cada clase empezando por las que tienen menos dependencias en otras que no se han implementado todavía.

Para la persistencia hay varios tipos de bases de datos:

1. Ficheros planos.
2. Jerárquicas.
3. Orientadas a objetos, no extendidas ni maduras.
4. Relacionales.
5. Objeto-relacionales, con elementos de orientación a objetos que se traducen a SQL.

Lo normal es usar bases de datos relacionales SQL, ya que la tecnología es muy madura y aceptada, tiene una base formal fuerte y aparece mucho en sistemas heredados, aunque tiene menos capacidad de modelado que las bases de datos orientadas a objetos.

«El diseño orientado a objetos no debe dirigir el diseño de la base de datos ni al revés». Veamos cómo dirigir el diseño de la base de datos a partir de un modelo de clases del diseño orientado a objetos.

Representamos un modelo relacional con un diagrama de clases creando una clase con estereotipo «Table» por cada tabla con un atributo por columna, sin métodos ni asociaciones. Un atributo puede tener estereotipos que se escriben delante de la visibilidad: «PK» si la columna está en la clave primaria, «FK» si está en una clave ajena y «NULL» o «NOT NULL» para indicar opcionalmente si puede valer nulo o no. Si una columna es `UNIQUE`, se añade una nota con la palabra UNIQUE desde la columna.

Primero creamos una tabla por cada clase persistente, con una columna por atributo[1] y una columna «PK» - id, de modo que cada objeto va en una fila.

Para mapear una asociación entre dos clases A y B:

- Si es 1 a 1, juntamos A y B en la misma tabla, o si los conceptos son importantes, creamos una clave ajena de una a otra.

[1] O varias, si el atributo es compuesto.

- Si es 1 a muchos, añadimos una clave ajena de B a A. También podemos crear una tabla cuyas columnas son una clave ajena a A y otra a B que es UNIQUE, con ambas en la clave primaria.[2]

- Si es muchos a muchos se crea una tabla cuyas columnas son una clave ajena a A y otra a B, con ambas en la clave primaria.

Una multiplicidad 0..1 se puede contar como 1, y una * o 1..* cuenta como muchos. Las asociaciones n-arias con $n > 2$ se pueden mapear como de muchos a muchos.

Una generalización se puede mapear con:

1. Una única tabla con las columnas de la superclase y las que añade cada subclase. Si las subclases son disjuntas, se añade un **atributo discriminante** con el tipo, y en otro caso se añade una columna booleana por cada subclase que indica si el atributo es de la subclase.

   Con esto no hacen falta reuniones para recuperar los elementos, pero se almacenan nulos y es difícil mantener la consistencia de los atributos de distintas subclases y gestionar asociaciones en que participa una subclase específica. Esto es apropiado si las subclases se diferencian poco en sus atributos y hay consultas polimórficas, en las que se requieren los datos de las subclases.

2. Una tabla por subclase cuya clave primaria referencia a la de la superclase. El almacenamiento es eficiente[3] y es fácil crear una nueva subclase o asociaciones con subclases, pero hacen falta muchas operaciones para recuperar o almacenar objetos. Esto es útil cuando hay muchas subclases distintas y se necesita polimorfismo.

3. Una tabla por subclase con los atributos de la superclase, que no se representa aparte. La clave primaria es compartida. Esto es eficiente para recuperar datos de una única subclase o almacenar datos, pero necesita reuniones para algunas consultas y no permite representar generalizaciones incompletas.[4] Esto es apropiado si no hacen falta consultas polimórficas, sino que solo se accede a las subclases.

El acceso a los datos de la base de datos depende del tipo de almacenamiento y la fuente de datos, por lo que puede ser conveniente separar la lógica de negocio de la de acceso a datos. El **patrón DAO** encapsula los accesos a la fuente de datos y expone una interfaz simple para recuperar y almacenar objetos ocultando detalles de implementación.

Para cada clase de dominio, se crea una interfaz DAO con operaciones de búsqueda por criterios (al menos por ID) y de inserción, con una implementación por sistema de base de datos. Entonces se crea una fábrica *singleton*, con una subclase para cada tipo de bases de datos, que permite acceder al DAO de cada clase de dominio en la base de datos usada. Esto aporta bajo acoplamiento.

---

[2] Lo lógico sería que la clave primaria fuera la referencia a B que para eso es UNIQUE; con este diseño la clave primaria no es una clave porque no es minimal.

[3] Cuando no hay muchos atributos en las subclases es más eficiente una única tabla, ya que las claves primarias y los índices de las tablas de las subclases pueden superar el tamaño de los nulos si los nulos ocupan poco espacio.

[4] Además, muchas bases de datos permiten generar automáticamente el ID de los objetos al insertarlos a una tabla, pero no si este tiene que ser único entre varias tablas.

Capítulo 7

Modelado de componentes

7.1. Arquitectura en capas

TDS

Dividir una aplicación en **capas** que reúnen clases relacionadas con un mismo aspecto del sistema evita el acoplamiento[...]. [...] **Arquitectura en** [...] 5 capas:

1. **Presentación**: Muestra ventanas o similares, genera informes [...].
2. **Aplicación**: Mantiene una sesión, gestiona peticiones [...] y coordina las transiciones entre ventanas o similares.
3. **Dominio** o **negocio**: [...] Implementa [...] reglas de negocio.
4. **Servicios técnicos**: Persistencia [...] y otros servicios [...].
5. **Base**: Bibliotecas y utilidades genéricas.

La **arquitectura en 3 capas** tiene capas de presentación, dominio y persistencia.

7.2. Paquetes

Un **paquete** es un elemento organizativo con elementos de cualquier tipo. Debe tener un conjunto de elementos equilibrado y ser cohesivo y poco acoplado con otros paquetes. Los paquetes se pueden anidar.

Los paquetes se representan en UML con un rectángulo con una solapa que sobresale de la parte izquierda del borde superior, dividido horizontalmente en dos partes: el nombre del paquete y, opcionalmente, una lista de objetos, uno por línea, con un indicador de visibilidad como el de los atributos de clases.

Un paquete anidado puede ver todo lo de sus paquetes padres. Cuando un paquete **importa** otro, lo que se indica con una flecha de uno a otro con estereotipo «import», puede usar sus elementos públicos. La importación no es transitiva.

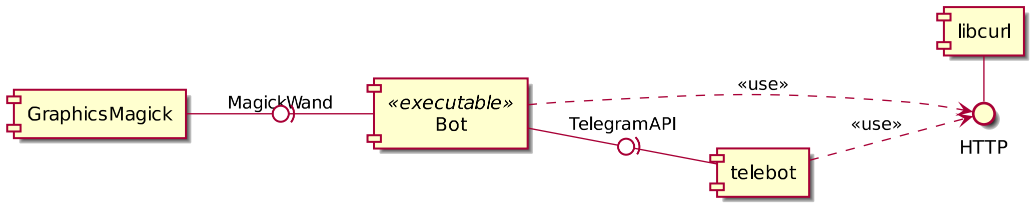


Figura 7.1: Ejemplo de diagrama de componentes.

Los paquetes permiten agrupar elementos, normalmente del mismo tipo, para manejarlos en conjunto.

7.3. Diagramas de componentes

Un **componente** es una parte lógica, modular, autocontenida y reemplazable de un sistema que conforma un conjunto de interfaces y proporciona su implementación. Modela artefactos como ejecutables, bibliotecas, archivos o documentos. Tipos:

- **Productos del trabajo:** Resultado directo del desarrollo. Código fuente, ficheros de datos, etc.
- **De despliegue:** Necesarios y suficientes para formar un sistema. Ejecutables compilados, bibliotecas dinámicas, etc.
- **De ejecución:** Se crean durante la ejecución, como los objetos COM.[1]

Un **diagrama de componentes** UML permite modelar ejecutables y bibliotecas, código fuente o una API. Los componentes se representan como en la figura 7.1.

Se puede representar una dependencia de un componente en otro con una flecha del uno al otro, o bien se pueden añadir interfaces a los componentes conectándolos a un círculo pequeño etiquetado con el nombre de la interfaz y hacer que otros componentes usen esa interfaz conectándolos a un arco de circunferencia que rodea al círculo o añadiendo una flecha punteada hacia el círculo posiblemente etiquetada con «use». Incluso se puede añadir el arco sin conectarlo con nada, etiquetado con el nombre de la interfaz, para indicar que «alguien» tiene que proporcionarle esa interfaz.

Los componentes pueden tener estereotipos como «artifact», «component», «input», «service», «user interaction», etc.

7.4. Diagramas de despliegue

Un **nodo** es un recurso que existe en tiempo de ejecución y en el que se ejecutan los componentes. Puede ser un **nodo dispositivo**, un recurso computacional físico con memoria

[1] El equivalente en software libre es GObject, usado sobre todo en GNOME. En Linux la comunicación entre componentes se hace a través de D-Bus, con bibliotecas para GObject, Qt y lenguajes de alto nivel con introspección.

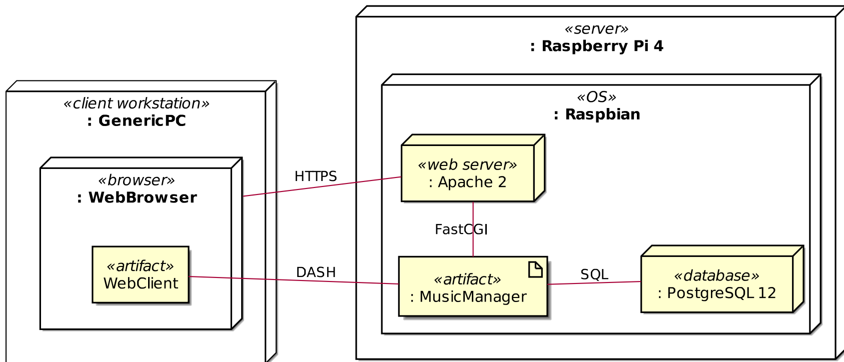


Figura 7.2: Ejemplo de diagrama de despliegue.

y capacidad de procesamiento, o un **nodo entorno de ejecución**, un recurso software que se ejecuta en otro nodo y puede albergar otros elementos software, como es el caso de un sistema operativo, un gestor de bases de datos o un navegador web.

Un **diagrama de despliegue** muestra la configuración de los nodos que participan en la ejecución y de los componentes en los nodos. Los nodos se representan como objetos pero «en relieve», con estereotipos como «web server cluster», «servlet container», etc., con unos dentro de otros y con componentes dentro de los nodos. Los componentes admiten varias notaciones.

Los nodos y componentes se pueden comunicar mediante líneas, posiblemente etiquetadas con la forma de comunicación. Esto se muestra en la figura 7.2. Los componentes se pueden represar con cualquiera de las notaciones de las figuras 7.1 y 7.2.

Capítulo 8

Modelos de ciclo de vida

El **proceso de software** (***software process***) es el ciclo de vida del software. Distintos ciclos de vida organizan las actividades de formas distintas, con distinto tiempo dado a cada actividad y distintos resultados, y distintas organizaciones usan ciclos de vida distintos para producir la mismo, pero para algunos tipos de programas hay ciclos de vida más convenientes que otros.

Un **modelo de ciclo de vida** es una descripción de un ciclo de vida desde una perspectiva particular, y se puede especificar, por ejemplo, con un modelo de flujo de trabajo. Hay muchos modelos de ciclo de vida, como el sistema de ensamblaje de componentes o el desarrollo dirigido por modelos.

8.1. Modelo en cascada

El **ciclo de vida** o **paradigma clásico**, **orientado a fases**, **lineal secuencial** o **modelo en cascada** consiste en la ejecución secuencial de una serie de fases, normalmente planificación, especificación de requisitos, diseño, implementación y mantenimiento, en la que cada fase genera documentación para la siguiente.

Aunque cada fase se valida antes de pasar a la siguiente, una validación «efectiva» requiere haber hecho las fases anteriores por lo que pasa mucho tiempo desde que se produce un error de análisis hasta que se detecta. Además, la detección de un error en una fase requiere repetir todas las fases a partir de esta.

Las fases suelen ser:

1. **Recopilación de requisitos:** Se especifican las reglas de negocio y los requisitos funcionales y no funcionales, y se revisan con el cliente.

2. **Diseño:** Se define la estructura del software que satisfaga los requisitos con la calidad necesaria, incluyendo arquitectura del software, interfaz, estructuras de datos y algoritmos. Se suele distinguir entre diseño preliminar y detallado.

3. **Codificación:** A veces se puede hacer de forma automática o semiautomática a partir de un diseño detallado.

4. **Prueba:** Pruebas unitarias, de integración, del software, del sistema y de aceptación, para asegurarse de que el sistema construido era el que se quería.

5. **Mantenimiento:** Es **correctivo** si elimina defectos; **perfectivo** si añade funciones u otras mejoras; **adaptativo** si se adapta a nuevos escenarios como plataformas y cambios legislativos, y **preventivo** si sirve para fomentar la extensibilidad posterior.

Esto es similar al enfoque de ingeniería tradicional, con el que se construyen edificios, y para software se puede usar si el proyecto es corto, de unos dos meses. Sin embargo, aunque es mejor que nada,[1] no funciona bien para proyectos más grandes.

Cuando hay mucha complejidad, congelar los requisitos durante todo el desarrollo, sin retroalimentación de implementaciones y pruebas reales, no es viable, ya que las personas involucradas pueden cambiar de idea o no imaginar lo que quieren hasta que ven un sistema concreto, y que hay cambios en el mercado, la tecnología, la ley, etc.

Se tarda mucho en pasar por todo el ciclo de desarrollo, produciendo retrasos innecesarios, y se da el **efecto bola de nieve** en que los errores de análisis y diseño se propagan a las etapas posteriores y el coste de eliminar dichos errores va aumentando. En la práctica el modelo se deforma y la validación y el mantenimiento solo se hacen al código, no al análisis ni al diseño.

8.2. Prototipos

Un **prototipo** es un modelo o maqueta del sistema con funcionalidad limitada que se hace en pocos días de desarrollo y permite comprender mejor el problema y sus posibles soluciones. Este ayuda al cliente a establecer claramente sus requisitos y a los desarrolladores a validar la corrección de la especificación, detectar problemas que se darán en el diseño y la implementación y examinar la viabilidad y utilidad del sistema, atacando la complejidad esencial del software.

Un prototipo se puede clasificar:

- Según su objetivo, como **prototipo de interfaz de usuario**, con pantallas y ventanas; **funcional**, que implementa algunas funciones, y **de rendimiento**, que evalúa el rendimiento de un programa y no sirve al análisis de requisitos.

- Según su alcance, como **vertical**, si desarrolla totalmente alguna función, u **horizontal**, si desarrolla parcialmente todas las funciones.

- Según el desarrollo:

  - **Prototipo rápido** o **desechable:** Sirve al análisis y la validación de requisitos, y una vez redactado el SyRS se desecha y se desarrolla la aplicación con un paradigma distinto. El problema es si no se desecha y se convierte en el sistema final, pues seguramente el prototipo se habrá construido rápidamente sin seguir un método de ingeniería de software y con un lenguaje poco eficiente.
  - **Prototipo evolutivo:** Comienza con un sistema relativamente simple que implementa los requisitos más importantes o mejor conocidos, y aumenta o cambia cuando se descubren nuevos requisitos hasta convertirse al sistema requerido.

[1] Según una gráfica que aparece en GPDS, «nada» es ligeramente mejor que esto.

Para el prototipado se pueden usar:

- Lenguajes dinámicos de alto nivel (**3GL**), como Java o C#, a elegir según el dominio de aplicación, la interacción de usuario y el entorno proporcionado. Algunos como Python, Ruby, o Perl aportan características de lenguajes 4GL.

- Lenguajes de cuarta generación (**4GL**), como los de acceso a bases de datos, diseño de interfaces de usuario o generación de código, como el software privativo Oracle Developer Suite o el software privativo IBM Informix.

  Son fáciles de usar y aprender y reducen claramente el coste de desarrollo, pero prestan poca atención al rendimiento, la seguridad o el mantenimiento, resultando en programas no estructurados y difíciles de entender que pueden quedar obsoletos haciendo que el usuario tenga que reescribir totalmente los programas. No están estandarizados, por lo que pueden dar lugar a incompatibilidad, y son difíciles de integrar en el sistema global.

El desarrollo de prototipos con reutilización comprende el **nivel de aplicación**, en que se integra una aplicación completa con el prototipo, y el **nivel de componente**, en que los componentes se integran en un marco de trabajo «estándar» como Visual Basic o .NET.

Para el prototipado de la interfaz de usuario podemos usar bocetos en papel, aplicaciones de dibujo como el software privativo Visio[2] o herramientas CASE específicas como el software privativo Iplotz o el software privativo Axure[3].

Tipos de prototipo de interfaz, de menor a mayor fidelidad y coste:

1. ***Wire-frame*:** Representación esquemática de baja calidad de los elementos de la interfaz, incluyendo contenido, características y navegación.

2. ***Mock-up*:** Representación estática del diseño de mayor calidad, que a menudo supone el diseño artístico o visual final.

3. **Prototipo:** Representación fiel del producto final, simulando la interacción con el usuario.

8.3. Modelo en cascada con prototipado desechable

Se modifica el modelo en cascada para incluir la creación de un prototipo rápido en el análisis de requisitos, que se modifica o recrea cuanto sea necesario hasta que el cliente lo apruebe y se desecha al terminar el análisis de requisitos.

El prototipo ayuda a determinar los requisitos y probar la viabilidad y sirve de contrato con el cliente para el desarrollo del producto, ayudando a mitigar el efecto bola de nieve, aunque no en el mantenimiento.

Sin embargo, el cliente ve una versión preliminar, sin asumir que esta no es robusta ni completa, y puede querer «parchear» el prototipo en vez de construir el sistema completo. Además, es frecuente arrastrar malas decisiones de diseño que solo eran apropiadas para el prototipo, y alguna vez puede pasar que el tiempo dedicado a construir el prototipo haga que el producto pierda oportunidad.

[2] Alternativas libres incluyen LibreOffice Draw o Impress, o Inkscape.

[3] Alternativas libres incluyen PlantUML para bocetos o GNOME Glade para aplicaciones de escritorio.

8.4. Modelos evolutivos

El software se hace en incrementos relativamente cortos en los que se hacen todas las fases, puede que con varios incrementos simultáneos, y al terminar un incremento se entrega un producto operativo.

El **modelo clúster** divide el software en **clústeres**, agrupaciones de clases con un objetivo común, y cada una se hace en un incremento cuyo subciclo de vida consta de especificación; diseño y realización, y validación.

El **modelo en espiral** tiene 4 fases; planificación, análisis de riesgo, ingeniería y evaluación del cliente, que se repiten en ese orden consecutivamente. Se llama así porque se puede dibujar una gráfica en que cada cuadrante del plano se etiqueta con el nombre de una fase y una espiral, centrada en el origen pero sin empezar desde ahí, que va hacia fuera y va pasando por las fases.

Este modelo aprovecha las ventajas del modelo en cascada y las del prototipado evolutivo, reajustando los productos con el tiempo, y es muy útil en desarrollos con requisitos inciertos o áreas importantes de riesgo.

El **método Booch 94** distingue un **macro-proceso** o marco de planificación, que interesa a la dirección, y un **micro-proceso**, guiado por el macro-proceso y que guía las acciones en el desarrollo de la arquitectura del sistema, que interesa al programador.

8.5. Verificación y validación (V&V)

La **verificación** o **prueba** consiste en comprobar si el software funciona bien. La programación **«prueba primero»** consiste en escribir código de prueba antes de escribir la implementación, para luego escribir el código que satisface las pruebas. Tiene la ventaja de que las pruebas se escriben; el programador queda satisfecho y grita «¡He superado la prueba!»[4]; se verifica la corrección, y no hay miedo a los cambios.

La **validación** consiste en comprobar con el usuario si el software que se está construyendo es el que este quiere. La hace el usuario comprobando, para cada caso de uso, que el sistema muestra el comportamiento esperado, considerando el escenario principal, los excepcionales y los requisitos no funcionales.

[4]Fuente: Apuntes de la asignatura. El grito de entusiasmo puede variar según el programador.

Capítulo 9

Métodos de desarrollo

Un **método de desarrollo de software** es un conjunto de procedimientos, técnicas, herramientas y soporte documental para ayudar a los desarrolladores a producir nuevo software. Está formado por un modelo de ciclo de vida con fases, etapas y actividades en cada etapa; procedimientos y heurísticas (recomendaciones) sobre cómo ejecutar las tareas; técnicas gráficas o textuales como casos de uso o diagramas, y herramientas CASE a utilizar.

Mientras un ciclo de vida indica qué hay que hacer, un método indica cómo hacerlo, basándose en modelos gráficos y su uso para la especificación y el diseño, y puede acomodar distintos ciclos de vida.

Podemos describir un método indicando la notación utilizada en los modelos, las reglas o restricciones aplicables a los modelos, las recomendaciones que caracterizan buenas prácticas de diseño y las guías en el proceso, descripciones de las actividades a seguir para desarrollar los modelos del sistema y la organización de estas actividades.

Un bueno método ayuda a conseguir sistemas de mayor calidad. No hay un método universal o ideal, sino que estos son aplicables a distintas áreas y están condicionados por el tamaño y la estructura de la organización y el tipo de aplicaciones, y organizaciones distintas usarán distintos métodos o, en todo caso, variaciones del mismo.

9.1. Tipos de métodos

Según el tipo de sistema, de gestión o en tiempo real. Según el nivel de formalidad, informal, semiformal o formal (con métodos formales).

Según el enfoque:

- **Estructurados:** Anteriores a la orientación a objetos.
    - **Orientados a procesos:** Centrados en los procesos ejecutados. Análisis estructurado, Análisis estructurado moderno y MÉTRICA.
    - **Orientados a datos:** se centran en las entradas y salidas.
- **Orientados a objetos.**
- **Basados en componentes**, una evolución de los métodos orientados a objetos hacia sistemas abiertos y distribuidos basados en servicios.

Según la planificación:

- **Prescriptivos:** Intentan planificar y predecir en detalle las actividades y la asignación de recursos en un intervalo largo de tiempo, como todo el proyecto. Suelen ser en cascada.
- **Adaptables:** Para dominios no predecibles. Aceptan el cambio. Suelen ser iterativos.

Según la carga de trabajo:

- **Pesados**, **burocráticos** o **monolíticos:** Lo habitual en los 80 y principios de los 90, a partir de experiencia en desarrollo de sistemas grandes y críticos. Son prescriptivos y rígidos y generan muchos artefactos.
- **Ligeros:** Evitan la sobrecarga en planificación, diseño y documentación.

9.2. Métodos ágiles

Métodos ligeros y adaptables, como XP (*eXtreme Programming*) y Scrum, surgidos a finales de los 90 para evitar la sobrecarga de trabajo de los métodos pesados al hacer software de gestión de tamaño medio o pequeño.

Son iterativos, con el cliente evaluando las iteraciones; asumen el cambio, y buscan mantener la simplicidad en el software y en el proceso, y sacar partido a las destrezas de los miembros del equipo de desarrollo.

La documentación se reduce al mínimo y en vez de casos de uso se usan **historias de usuario**, de la forma «como *rol*, quiero que el sistema *haga una funcionalidad* para obtener *un beneficio*», que se suelen complementar con pruebas de aceptación del usuario.

Son útiles para software de gestión con requisitos que cambian rápidamente. No lo son en desarrollo a gran escala con equipos de desarrollo en sitios distintos, ni en sistemas críticos en lo que hace falta un análisis detallado de los requisitos para comprender las implicaciones de seguridad informática (*security*) y humana (*safety*).

Scrum es un marco para el desarrollo ágil que supone la existencia de caos. Tiene 3 fases:

1. **Inicio.** Se planifica una versión del software a construir con una estimación inicial de tiempo y costo.
2. **Fase iterativa** en ***sprints***, iteraciones de paso fijo (unos 15 días) con objetivos establecidos al inicio y durante el cual no se introducen cambios.
3. **Cierre.** Se preparan la versión del software a instalar, la documentación final y los entornos.

El **desarrollo *lean*** es aun más flexible que el ágil, y su principal método es **Kanban**. En este la estimación es opcional; las iteraciones son de tiempo fijo pero con posibilidad de expandirlo si no se terminan las tareas y de introducir nuevas tareas una vez empezada la iteración, y no es obligatorio terminar una tarea en una iteración.

9.3. Proceso unificado

UP, *Unified Process*, es un marco de método orientado a objetos basado en UML, con 4 fases:

1. **Inicio.** Se define el alcance del proyecto. Se hace análisis de negocio, considerando objetivos y características del producto, alternativas y riesgos, para decidir la viabilidad y dar una estimación imprecisa del coste. Se crea una visión aproximada del producto.

2. **Elaboración.** Visión refinada e implementación iterativa del núcleo de la arquitectura y resolución de los riesgos más elevados.

3. **Construcción.** Implementación iterativa de los requisitos que quedan, de menor riesgo.

4. **Transición.** Pruebas beta y despliegue.

Las fases de inicio y fin son las más cortas, siendo la construcción la más larga y la que requiere de más desarrolladores. Cada fase es iterativa y cada iteración se divide en **disciplinas**: modelado del negocio, requisitos, análisis y diseño, codificación, y prueba, junto con administración del proyecto y gestión de configuración y cambio.

En las primeras iteraciones se da la mayor parte del tiempo al modelado del negocio y los requisitos, y en las últimas se dedica la mayor parte al despliegue.

Cada disciplina, del modelado de negocio al despliegue, produce modelos, como de casos de uso de negocio, de objetos de negocio, de casos de uso, de diseño, de implementación (el código fuente) y de prueba (el resultado de los *tests*).

RUP (*Rational Unified Process*) es una instancia pesada de UP.

9.4. Método de Larman

Es una instancia ágil de UP dirigida por casos de uso. Respecto a las fases:

1. El inicio dura una o dos semanas como mucho. Se hacen los primeros talleres de requisitos, se escriben casos de uso en formato breve, se crea un prototipo desechable de interfaz de usuario para validar los requisitos y se planifica la primera iteración del desarrollo.

2. Aunque no es parte del método de Larman, se puede añadir tras el inicio una fase de modelado de negocio, donde se usan diagramas de proceso y otros medios y se refinan la identificación de objetivos de usuario y casos de uso y la especificación complementaria.

3. En la elaboración y la construcción, las iteraciones duran entre 2 y 8 semanas, y en ellas se identifican, describen e implementan los casos de uso de mayor prioridad según el nivel de riesgo y la importancia para el negocio.

Las disciplinas son las siguientes y se hacen en orden en el desarrollo:

1. **Modelado de requisitos**, con modelos de caso de uso y de clases conceptuales.

2. **Diseño preliminar.** Se crea un DSS para cada caso de uso y, para cada operación de cada DSS, un contrato si se considera importante y una colaboración. Junto a las colaboraciones se crea un diagrama de clases de especificación.

3. **Diseño completo.** Se resuelven los problemas de diseño relacionados con la arquitectura del sistema, la división y distribución en paquetes, la interfaz de usuario, la plataforma, el rendimiento, la base de datos, la red, las estructuras de datos y los patrones de diseño a usar, nuevas clases y colaboraciones, etc.
4. **Implementación y pruebas**, tanto unitarias como de integración.
5. **Validación** con clientes y usuarios.

9.5. Métrica 3

MÉTRICA es una metodología de planificación, desarrollo y mantenimiento de sistemas de información promovida por el Consejo Superior de Administración Electrónica del Gobierno de España para su uso en las administraciones públicas.

Es muy prescriptiva, y establece una jerarquía formada, de arriba a abajo, por la Administración Central del Estado, la Administración Autonómica, la Administración Local y el resto de empresas e instituciones.

Tiene versiones 1, 2, 2.1 y 3, con la 1 creada por la consultora ERITEL y la 2.1 por la Universidad Carlos III.

Métrica 3 incorpora los paradigmas orientado a objetos y cliente-servidor, y se basa en UML y descomposición funcional. Está influenciado por métodos de Reino Unido y Francia, y sus componentes más usados son el Plan de General de Garantía de Calidad y la parte de seguridad, MAGERIT. Se estructura en:

- Planificación de Sistemas de Información.
- Desarrollo de Sistemas de Información.
  - Estudio de Viabilidad del Sistema.
  - Análisis del Sistema de Información.

    Actividades: Definición del sistema; Establecimiento de requisitos; Identificación de subsistemas de análisis; para orientado a objetos, Análisis de casos de uso o Análisis de clases; para estructurado, Elaboración del modelo de datos o Elaboración del modelo de procesos; etc.
  - Diseño del Sistema de Información.

    Actividades: Definición de la arquitectura del sistema; Diseño de la Arquitectura de Soporte; para orientado a objetos, Diseño de casos de uso reales o Diseño de clases; etc.
  - Construcción del Sistema de Información.
  - Implantación y Aceptación del Sistema.
- Mantenimiento de Sistemas de Información.

9.6. Herramientas CASE

Asisten y automatizan partes del proceso de desarrollo de software a lo largo de su ciclo de vida, incluyendo gestión, desarrollo y mantenimiento. Suelen estar formadas por un repositorio, un meta-modelo y un generador de informes.

Según el grado de integración, tenemos:

Tool kits Conjunto de programas que automatizan algún proceso, con poca integración.

Workbenches Conjunto de programas que comparten repositorio e interfaz de usuario y automatizan más de una fase del ciclo de vida.

Entornos IPSE *Integrated Project Support Environment*, que cubren todo el ciclo de vida y tienen integración alta.

Según los procesos que abortan, tenemos:

CASE frontales Análisis y diseño, como el software privativo Astah[1].

CASE dorsales Implementación, pruebas y mantenimiento, como Maven o Git.

ICASE ***Integrated CASE***, contemplan todo el ciclo de desarrollo, como el software privativo Enterprise Architect.

[1] El equivalente libre, mucho mejor, es PlantUML, usado en estos apuntes.