Modelado Avanzado en el Desarrollo de Software

5.1. Introducción y objetivos

En este tema se revisan conceptos y se estudian aspectos concretos más avanzados en el modelado de *software* con UML. En particular, se analizarán más en detalle ciertos elementos vinculados con la ingeniería de requisitos (la disciplina que trata de determinar y documentar los requisitos de un sistema), la semántica vinculada a las asociaciones, el modelado durante la fase de diseño usando prototipos y el modelado de componentes.

Dentro del modelado de requisitos, se repasa todo lo referente al concepto de requisito, junto con las características que debe tener todo buen requisito.

Con el estudio de este tema pretendemos alcanzar los siguientes objetivos:

* Conocer el modelado en la ingeniería de requisitos.
* Reforzar el concepto de requisito y todo lo que lo envuelve.
* Estudiar conceptos avanzados de modelado estático.
* Estudiar conceptos avanzados de modelado dinámico.
* Estudiar los prototipos para diseñar interfaces de usuario.
* Entender el modelado de componentes.

5.2. Modelado avanzado de requisitos

La extracción de requisitos de un sistema *software* es la primera, y se podría decir que la más crítica, de las tareas a realizar en todo proyecto y la que, curiosamente, demasiadas veces no se gestiona adecuadamente, implementando, en consecuencia, aplicaciones de baja calidad con las que el cliente no se siente satisfecho.

A pesar de que en la actualidad se dispone de una gran cantidad de herramientas formales que facilitan la tarea a la hora de modelar sistemas *software*, las que incluso dan soporte automatizando a algunas de las tareas de análisis, el número de proyectos que fracasan porque el sistema no hace lo que el cliente necesitaba sigue siendo demasiado elevado.

La ingeniería de requisitos es una disciplina difícil, pues no se trata únicamente de anotar lo que nuestro cliente o usuario dice necesitar. La ingeniería de requisitos es la disciplina que se encarga de obtener, analizar, documentar y validar los requisitos y, dado que estos pueden variar y evolucionar a lo largo del proyecto, las actividades de gestión serán fundamentales. Existen diferentes modelos de procesos para este conjunto de actividades, pero todos ellos ponen de manifiesto que se trata de un proceso iterativo e incremental, en el que progresivamente se va refinando y mejorando la documentación que soporta la especificación.

Un problema fundamental reside en la naturaleza dinámica de los requisitos, cambiantes a medida que las necesidades del usuario o la propia comprensión sobre el problema evolucionan a lo largo del tiempo. Por este motivo, supone un proceso que debe ser tan preciso y estricto como sea posible y, al mismo tiempo, flexible. Se trata de una fase especialmente crítica del proceso *software,* puesto que los errores en esta etapa conducen de manera inevitable a problemas posteriores en el diseño y la implementación del sistema (Sommerville, 2011, p. 37).

Muchos clientes y *project managers* tienden a pensar que el mejor indicador de avance de un proyecto es la cantidad de código implementado. Valoran, por tanto, sobre todo las actividades de diseño e implementación que, en su opinión, deben ser acometidas cuanto antes. Sin embargo, de acuerdo con la experiencia en la industria, a menudo se dedica un tiempo insuficiente a las actividades de captura, análisis, validación y gestión de requisitos.

Es muy normal que el cliente y el contratista (es decir, la organización proveedora del *software* o del servicio) sufran problemas de comunicación, tal y como se ilustra de manera muy acertada en la Figura 1. Problema que hay que saber abordar y en donde los modelos del sistema que puedan ser entendidos por los expertos del dominio del negocio (que lo normal es que no tengan un perfil técnico) juegan un papel fundamental en esta etapa.

A diagram of a tree swing

Description automatically generated

Figura 1. El típico problema de incomunicación. Fuente: Urbano *et al.*, 2015.

Descubrir los verdaderos requisitos de un nuevo producto o servicio no es siempre tarea fácil, y requiere de un esfuerzo iterativo en el que la comunicación con todas las partes implicadas juega un papel fundamental.

Concepto de requisito

Un requisito es un atributo necesario de un sistema. Indica una capacidad, característica o factor de calidad del sistema que aporta valor y utilidad para el cliente o usuario final (Young, 2004). Son importantes porque constituyen la base fundamental para el trabajo de desarrollo que comenzará a partir de su definición, y que se asienta sobre él: el diseño, la implementación, la validación e incluso la operación y el mantenimiento del sistema.

Dentro de este contexto, un requisito *software* se define como una responsabilidad del *software* que se ha de cumplir para poder alcanzar los objetivos del sistema definidos en un contrato, especificación o cualquier otro tipo de documentación proporcionada (Dorfman y Thayer, 1990).

Por tanto, un requisito se puede enfocar desde el punto de vista del usuario (su necesidad o problema de partida), desde el punto de vista del propio sistema (que debe satisfacer unas condiciones) o desde el punto de vista de la documentación.

A pesar de la aparente simplicidad del concepto, lo cierto es que, en general, el término «requisito» vendrá acompañado de una serie de adjetivos que matizarán su significado en cada caso. Se habla así de «requisitos del usuario», «requisitos del sistema», «requisitos funcionales», «requisitos no funcionales», o «requisitos de negocio», por citar algunos ejemplos.

Esta variedad se debe a que los requisitos deben ser identificados, elaborados y documentados en función de un número de factores:

* La audiencia a la que va dirigida.
* El nivel de avance en el que se encuentre el proceso de desarrollo.
* El tipo de capacidad o restricción que se desea documentar, etc.

Estas diferentes clasificaciones de los requisitos están íntimamente relacionadas con las tres dimensiones fundamentales que ayudan a comprender la complejidad del término (Durán, 2000):

* Ámbito. Esta dimensión está relacionada con la parte del sistema en desarrollo que se ve afectada por el requisito. Este puede aportar información sobre el sistema en su conjunto, acerca del *hardware* sobre el cual se ejecuta o únicamente acerca de la funcionalidad y las restricciones que quedan definidas en el *software.*
* Característica que define. En este caso podemos encontrar requisitos de tipo funcional (que indican los servicios y prestaciones concretas que el sistema debe ofrecer al usuario), requisitos no funcionales (que indican restricciones operativas o características relacionadas con la confiabilidad y seguridad del sistema, no estrictamente vinculadas a funcionalidades concretas) y requisitos de información (que indican qué tipo de información debe almacenar y manipular el sistema, y cuáles son sus componentes y características).
* Audiencia. Esta última dimensión indica a qué tipo de audiencia va dirigida la información que captura el requisito y, por lo tanto, determina el enfoque e incluso el vocabulario, o el tipo de representación que debe emplearse. En general se puede distinguir dos tipos de audiencia: los clientes y usuarios (sin formación técnica específica a priori), y los desarrolladores de *software* que participan en el proyecto.

A diagram of a diagram of software

Description automatically generated

Figura 2. Dimensiones de los requisitos. Fuente: Durán, 2000.

Desde el punto de vista de la funcionalidad ofrecida podemos identificar tres niveles diferenciados en los requisitos de *software* (Wiegers y Beatty, 2013):

* Requisitos de negocio. Son el origen de todo proyecto.
* Requisitos de usuario. Expresan las necesidades de clientes y usuarios.
* Requisitos funcionales. Detallan el comportamiento esperado del sistema.

Además, todo sistema viene restringido por un conjunto de requisitos no funcionales.

A continuación, en la lección magistral *Niveles y tipos de requisitos* se realiza una pequeña introducción al concepto de lo que es requisito de software y sus distintos niveles y tipos.

A diagram of a software system

Description automatically generated

Figura 3. Relación entre diferentes tipos de requisitos de software. Los óvalos representan tipos de requisitos, mientras que los rectángulos son los documentos que los almacenan. Las líneas sólidas indican «se almacenan en», mientras que las líneas de trazos indican «es el origen de». No se muestran los requisitos de datos Fuente: adaptada a partir de Wiegers y Beatty, 2013.

Los requisitos de negocio expresan por qué una organización desea desarrollar el sistema y el beneficio que espera alcanzar con él y, por tanto, indican sus objetivos. Vienen marcados por el patrocinador o financiador del proyecto, el cliente comprador, los usuarios, el departamento de marketing o el creador del producto.

Los requisitos de usuario describen qué desean hacer los usuarios con el sistema, los objetivos a nivel de uso cotidiano, o tareas que podrá realizar con el producto y que le aportan valor. Incluyen características y atributos, no necesariamente relacionados con la funcionalidad, pero que mejoran la satisfacción del usuario. Un mismo sistema puede tener distintos perfiles de usuarios, o *stakeholders* en un sentido amplio, cuyas necesidades hay que tener en cuenta.

Los requisitos funcionales indican qué deben implementar los desarrolladores para que los usuarios alcancen sus objetivos y, en última instancia, la organización alcance también los suyos. Son «enunciados acerca de servicios que el sistema debe proporcionar, de cómo debería reaccionar ante entradas particulares y de cómo debería comportarse en situaciones específicas» (Sommerville, 2011, p. 94). En ocasiones estos requisitos también indican lo que el sistema no debe hacer.

Este último tipo de requisitos se debe detallar con mayor precisión y empleando un lenguaje más técnico, puesto que van dirigidos al equipo de desarrollo y forman parte del vínculo contractual entre cliente y desarrolladores.

Esta información queda recopilada en el SRS *(Software Requirements Specification)* y es la base para todas las operaciones de desarrollo, pruebas, aseguramiento de la calidad y gestión del proyecto.

Por último, los requisitos no funcionales son «limitaciones sobre servicios o funciones que ofrece el sistema» (Sommerville, 2011, p. 85). Suelen tener que ver con el sistema como un todo, más que con servicios individuales.

Son a menudo más importantes y críticos que los requisitos funcionales por varios motivos:

* Influencia en el producto: afectan más a la arquitectura general del sistema que una funcionalidad concreta y aislada.
* Repercusión de los fallos: los problemas con estos requisitos pueden afectar al sistema en su conjunto y dejar inutilizadas muchas de sus funcionalidades.
* Origen de requisitos funcionales: un requisito no funcional individual puede generar una multitud de requisitos funcionales relacionados que debe contemplar el sistema.

Sommerville (2011), propone una clasificación de los requisitos no funcionales en tres grandes grupos que puede ayudar a identificarlos y conocer su razón y origen (Ver Figura 4):

A diagram of a company

Description automatically generated

Figura 4. Clasificación de los requisitos no funcionales. Fuente: Sommerville, 2011.

* Requisitos del producto: restringen el comportamiento del propio *software* en cuanto a rendimiento, fiabilidad, seguridad o usabilidad. Son características que debe presentar el producto, relacionadas con la satisfacción del usuario o impuestas por el cliente.
* Requisitos de la organización: vienen derivados de las propias políticas y procedimientos de la organización (tanto del cliente como del desarrollador) e imponen restricciones sobre la manera de utilizar el producto, sobre el propio proceso de desarrollo e incluso los lenguajes, tecnologías y estándares que serán utilizados durante el desarrollo.
* Requisitos externos: estos últimos vienen impuestos por factores externos al propio sistema y al proceso de desarrollo, y tienen que ver con aspectos regulatorios (que definen sus características para que pueda ser aprobado su uso por un ente regulador en determinado contexto), aspectos legislativos (que garantizan que el producto será compatible con las leyes vigentes) o, incluso, aspectos de tipo ético, (que tendrán en cuenta de qué manera apreciará el usuario final el *software* y si será aceptable para el público general).

A continuación, en la lección magistral *Clasificación de requisitos no funcionales,* se realiza una pequeña introducción a los requisitos de tipo no funcional y se pude ver una clasificación expuesta por Sommerville.

Características de los requisitos

A la hora de definir los requisitos y documentarlos, es importante asegurar que contienen información de calidad y claramente comprensible. De acuerdo con el estándar IEEE 830-1998 (IEEE, 1998) y con Young (2004), podemos identificar las siguientes características de un buen requisito:

* Completo. Se indican todas las condiciones en las cuales el requisito tiene sentido y expresa la necesidad de manera completa.
* Correcto. Las necesidades que describe el requisito son acertadas, y es técnica y legalmente posible su implementación.
* Comprensible. La lectura del requisito debe presentar un significado claro para el lector, en función de la audiencia.
* Necesario. Indica que refleja una necesidad real en el sistema. Si el sistema pudiera satisfacer las necesidades del usuario sin un requisito, este no sería necesario.
* Realizable. Es factible la implementación del requisito teniendo en cuenta los recursos y el tiempo disponibles.
* Conciso. Refleja que el requisito queda enunciado de manera simple.
* No ambiguo. El requisito admite una única interpretación.
* Consistente. No hay conflicto con otros requisitos.
* Verificable. Se puede comprobar de algún modo la implementación del requisito en el sistema final.
* Identificable. Cada requisito debe llevar asociado un identificador único.
* Trazable. Se puede identificar su origen y, a su vez, se puede hacer un seguimiento del requisito a lo largo del ciclo de vida del sistema (en su diseño, en el código, en las pruebas, en la documentación, etc.).
* Modificable. Debería ser fácil introducir modificaciones en el requisito manteniendo inalteradas el resto de las propiedades.
* Localizable. Se puede asignar a un componente del sistema diseñado.
* Independiente del diseño. No implica una solución de implementación concreta.
* No redundante. No indica necesidades duplicadas total o parcialmente.
* Escrito mediante una construcción estándar. Esto facilita la comprensión del requisito. Generalmente se escriben empleando frases imperativas (por ejemplo, «el sistema deberá…»).
* Exento de cláusulas de escape, excepción o salvaguardia. La expresión del requisito no debería incluir en general expresiones como «si», «cuando», «pero», «excepto», «a menos que» o «aunque». Tampoco debería emplear un lenguaje especulativo que pueda introducir ambigüedad con palabras como «generalmente», «a veces», «a menudo», «normalmente», etc.
* Priorizado. Debería ser posible asignar una importancia relativa a cada requisito dentro del conjunto.

Un aspecto fundamental en la actividad de especificación de requisitos *software* es saber distinguir los requisitos falsos de los requisitos verdaderos tal y como los definen McMenamin y Palmer (1984). Un requisito falso sería aquel que, sin ser implementado, la aplicación sigue cumpliendo sus objetivos. Cualquier característica irrelevante para el proyecto actual, así como una actividad que simplemente se ha de llevar a cabo para adaptarse a la tecnología que se va a utilizar para implementar el sistema, serían ejemplos de requisitos falsos del sistema a implementar de acuerdo con McMenamin y Palmer (1984).

Los requisitos falsos son clasificados por McMenamin y Palmer (1984) en dos categorías: requisitos tecnológicos y requisitos arbitrarios. Los requisitos falsos de tipo tecnológico pueden ser introducidos por los analistas cuando, por ejemplo, anticipan las características técnicas del nuevo sistema y las introducen en la especificación de requisitos. Por otro lado, los requisitos falsos de tipo arbitrario son los que introducen los analistas, por ejemplo, al especificar más funcionalidad de la que necesita el sistema, sin que la haya solicitado el cliente. Los requisitos falsos suponen, aunque a veces no se vea así, una gran amenaza al éxito del proyecto, ya que pueden derivar en especificaciones incorrectas y que no se entienden, proyectos que se van de tiempo y presupuesto, sistemas difíciles de mantener, etc.

Técnicas de modelado de requisitos

No existe una única manera de analizar o modelar una realidad, del mismo modo que no existe una única técnica para descubrir requisitos ni un único modo de documentarlos que sea de utilidad en cualquier situación posible.

Los modelos deben convertirse en una herramienta efectiva de comunicación entre clientes y contratistas, de tal forma que el contratista ha de ser capaz de especificar los requisitos de la manera más formalposible, procurando que resulten entendibles por los clientes, sin que existan ambigüedades ni contradicciones y, en caso de darse, que se detecten en las fases más tempranas del proyecto.

La gestión de los requisitos es por tanto una de las actividades más importantes a la que debe enfrentarse todo proyecto si quiere tener éxito. La gestión de requisitos es definida por Leffingwell y Widrig (2003) como:

«[…] el enfoque sistemático para la elicitación, organización, y documentación de los requisitos del sistema, y el proceso que establece y mantiene acuerdos entre el cliente y el equipo del proyecto para tratar los cambios en los requisitos» (p. 16).

En la fase de elicitación de requisitos, el cliente y el contratista analizan los problemas que se dan en el dominio, así como las necesidades que se identifican y sus características. En base a este análisis, se decide qué cambios se han de introducir en el dominio y la funcionalidad que se ha de llevar a cabo en el nuevo sistema para cubrir las necesidades del cliente que permitirán mejorar su negocio (Olivé, 2007, p. 27).

Durante la elicitación, es decir, durante la fase de identificar las necesidades y restricciones impuestas por los diferentes *stakeholders* del sistema (Wiegers y Beatty, 2013), se puede aplicar un gran número de técnicas para descubrir, recopilar y definir requisitos.

Para ello, hay que analizar diferentes contextos en los que podemos investigar para descubrir requisitos de manera eficiente. En ellos deberemos enfocar nuestras actividades de elicitación, y nos ayudarán a seleccionar las técnicas más adecuadas en cada caso. En una primera aproximación podemos distinguir tres grandes grupos de contextos (Alexander y Beus-Dukic, 2009):

* El contexto individual. En este caso se trata de obtener información a partir de usuarios o clientes individuales. Podemos encontrar técnicas como:
  + Las entrevistas, en las que el analista plantea preguntas al usuario.
  + La observación directa del individuo, en la que el analista observa, escucha y aprende.
* El contexto grupal. Los grupos bien dirigidos funcionan como sistemas que, ante los estímulos adecuados, proporcionan mejores resultados que individuos aislados, como resultado de su interacción. Encontramos las siguientes técnicas:
  + La etnografía o análisis etnográfico. Es una técnica que procede del ámbito de la antropología y se centra en analizar el comportamiento de los individuos y la manera en que operan en el ámbito de una organización concreta.
  + Los workshops. Son reuniones facilitadas que buscan la reflexión y la obtención de requisitos mediante dinámicas de grupo y actividades colectivas.
  + Los *focus groups* o grupos focales. Permiten obtener requisitos consultando directamente a potenciales usuarios finales, especialmente cuando estos pertenecen al público general que se encuentra fuera de la organización desarrolladora.
  + Los cuestionarios, que permiten obtener grandes cantidades de información enviando consultas a *stakeholders* distribuidos geográficamente.
  + El *brainstorming* y otras técnicas de generación de ideas y requisitos, que se apoyan en métodos que fomentan la creatividad individual y colectiva.
  + Otros medios grupales (*group media*) o medios colaborativos, formados por herramientas de diversa índole que facilitan la colaboración y el intercambio de ideas entre los participantes.
* El contexto de las cosas. Los requisitos cubren, en última instancia, necesidades de las personas. Sin embargo, en ocasiones es posible aprender y deducir requisitos mediante la observación directa de otros sistemas o de su documentación asociada. Aquí encontramos una serie de técnicas concretas:
  + El análisis de interfaces, tanto entre el usuario y el sistema, como entre el sistema y otros con los que deberá conectarse.
  + El análisis de la documentación de productos sometidos a mejora y de otros similares.
  + Los casos de uso y los escenarios, que permiten representar los requisitos del usuario, reflexionar sobre ellos y refinar la especificación o descubrir nuevas funcionalidades.
  + Los prototipos, que permiten simular el funcionamiento del sistema real para que sus futuros usuarios nos ayuden a validar la especificación, o funcionan como soporte para descubrir nuevas necesidades o sugerir nuevos requisitos.
  + La ingeniería inversa, que permite recuperar requisitos a partir de un producto o sistema existente, identificando necesidades descubiertas por otras personas.
  + La reutilización de requisitos, que permite en ocasiones tomar requisitos de proyectos similares y aplicarlos a un nuevo producto.

A continuación, en la lección magistral *Casos de uso y escenarios,*se realiza una breve descripción de los casos de uso y los escenarios como técnica de elicitación de requisitos.

Si se adopta la técnica de los casos de uso para modelar la funcionalidad del sistema, hay que ser conscientes de la ambigüedad presente en UML y definirlo de manera concisa con la documentación adicional que sea necesaria. Por ejemplo, de acuerdo a la documentación de UML (OMG, 2011) más de un actor podrá estar conectado a un caso de uso (ver Figura 5), pero no hay forma de indicar con elementos de UML si se necesita que todos los actores actúen a la vez para que se realice el caso de uso o si el caso de uso es para cada actor que lo ejecutará de manera independiente, o si se refiere a que deben seguir un orden determinado.

A black and white diagram with words

Description automatically generated

Figura 5. Actores cooperativos. Fuente: Urbano *et al.*, 2015.

Esta ambigüedad solo es posible solventarla con información adicional. Como ocurre con este ejemplo, claramente en un caso de uso se pueden dar un montón de situaciones en las que con las descripciones (especificación) textuales, precondiciones y postcondiciones con las que se suelen acompañar, no resulta suficiente. Es por ello, que se podrán complementar con diagramas de interacción, diagramas de actividad y diagramas de estados que se relacionen y favorezcan su comprensión.

Por otro lado, resulta conveniente limitar las relaciones *Include* («include») y *Extend* («extend») que se suelen utilizar en los diagramas de casos de uso (ver Figura 6), ya que, por un lado, no está clara su diferencia (reutilización vs. inserción) y, por otro lado, confunde con la definición de caso de uso que dice que un caso de uso representa una unidad coherente de funcionalidad. Además, puede llevar a la malinterpretación (observando el diagrama) de que se da un procesamiento secuencial cuando, en realidad, ese comportamiento adicional podría estar localizado en cualquier parte de la funcionalidad que representa el caso de uso.

A diagram of a diagram

Description automatically generated

Figura 6. Relaciones *Include* y *Extend.* Fuente: Urbano *et al.*, 2015.

Para modelar las situaciones que tienen en común una secuencia de acciones previa a la ejecución del resto de casos de uso como sería, por ejemplo, el acceso al sistema (identificación, inicio y cierre de sesión, etc.), la solución menos problemática sería definir dichas acciones como una subactividad dentro del diagrama de actividad correspondiente a cada caso de uso que se vea afectado por esta situación.

Por último, hay que comentar también que, a la hora de modelar casos de usos, también hay que tener en cuenta, por un lado, las relaciones de generalización que se dan entre actores cuando se identifican distintos roles que han de realizar un mismo caso de uso y que son subtipos unos de otros (ver Figura 7); y, por otro lado, las relaciones de generalización que se dan entre casos de uso, cuando se detectan especializaciones para un caso de uso concreto (ver Figura 8). Al igual que ocurría con las relaciones de Include y Extend, las generalizaciones entre casos de uso son una solución que se debería evitar en la medida de lo posible, realizando otras técnicas de modelado que favorezcan su comprensión.

A diagram of a person with arrows

Description automatically generated

Figura 7. Relaciones de generalización entre actores. Fuente: Urbano *et al.,* 2015.

A diagram of a register

Description automatically generated

Figura 8. Relaciones de generalización entre casos de uso. Fuente: Urbano *et al.,* 2015.

5.3. Modelado estático avanzado

Cuando se desarrolla una aplicación, el modelo de clases constituye una pieza muy importante de comunicación entre analistas, diseñadores y programadores para entender qué entidades se consideran, cómo están relacionadas unas con otras y de qué manera se van a comunicar.

Dentro del modelado de la vista estática (es decir, estructura) del sistema con UML, se dan una serie de aspectos que hay que tener en cuenta y se han de comprender correctamente para que los modelos realmente constituyan una fuente fiable de cómo se encuentra definido el sistema.

En este punto, también es interesante recordar la diferencia entre el modelo conceptual (análisis), que es el que identifica las clases, atributos y operaciones que corresponden a conceptos del dominio del negocio del sistema, y el modelo de *software* (diseño), que se correspondería con las clases, atributos y métodos que se corresponden con los conceptos de la plataforma de desarrollo escogida para su implementación.

Además, merece la pena comentar que se podría hacer un modelo previo al modelo conceptual, que sería lo que se puede denominar como modelo del entorno, que «define esa parte de la realidad que rodea al sistema informático y con la que este interactúa para proporcionarle determinados servicios» (Génova, Valiente y Marrero, 2006, p. 3). El modelo del entorno podría ser también de ayuda a la hora de entender y extraer los requisitos del sistema *software* a implementar.

A continuación se describen una serie de elementos que forman parte de la vista estática del sistema y que requieren especial atención.

Asociaciones reflexivas

Una asociación reflexiva o recursiva es una asociación que enlaza una clase consigo misma, tal y como se ilustra en la Figura 9. Una asociación reflexiva presenta las siguientes características:

* Los enlaces pueden conectar dos instancias diferentes de la misma clase, pero también pueden conectar una instancia consigo misma.
* De cara a poder distinguir los extremos de la asociación, los nombres de rol son obligatorios.
* No es simétrica.

A diagram of a computer code

Description automatically generated

Figura 9. Asociación reflexiva. Fuente: Urbano *et al*., 2015.

Clase asociación

Una clase asociación se corresponde con una clase que tiene las propiedades propias de una clase y una asociación, tal y como se ilustra en la Figura 10. Una clase asociación presenta las siguientes características:

* Como se trata de un elemento único, ha de tener un único nombre.
* Como cualquier otra asociación, en principio, no puede contener tuplas repetidas (es decir, no puede conectar dos mismos objetos más de una vez), aunque los valores de los atributos sean distintos. Si se deseara representar el registro histórico (es decir, que permita tuplas repetidas), se tendría que poner la restricción en cada extremo de asociación.

A close-up of a computer screen

Description automatically generated

Figura 10. Clase asociación. Fuente: Urbano *et al.,* 2015.

A la hora de programarla en una plataforma concreta (Java, .NET, etc.), en diseño, la clase asociación se tendrá que transformar en una clase intermedia haciendo un cruce de las cardinalidades, tal y como se muestra en la Figura 11.

A diagram of a person

Description automatically generated

Figura 11. Transformación de una clase asociación en clase intermedia. Fuente: Urbano *et al.,* 2015.

Asociación n-aria

Una asociación n-aria se corresponde con una asociación que enlaza n clases, tal y como se ilustra en la Figura 12. Una asociación n-aria presenta las siguientes características:

* No permite especificar la dirección del nombre, la navegabilidad ni las asociaciones de agregación.
* Admite clases asociación.
* La multiplicidad especificada en un extremo de una clase representa el número permitido de instancias para cualquier posible combinación de instancias de las otras
* La multiplicidad mínima suele ser 0.
* Efecto «rebote del uno». Este efecto indica que cuando la multiplicidad mínima especificada en un extremo de una clase sea 1 (o superior), debe existir un enlace (o más) para cualquier posible combinación de instancias de las otras .

A diagram of a computer network

Description automatically generated with medium confidence

Figura 12. Asociación n-aria. Fuente: Urbano *et al.*, 2015.

### **5.4. Modelado dinámico avanzado**

Como ya se ha comentado, con la vista estática del sistema el equipo del proyecto determina la estructura completa del sistema a implementar. Sin embargo, esta vista no es suficiente para implementar la aplicación y se hace necesario modelar el comportamiento en aquellas funcionalidades que requieran un tratamiento especial o sean difíciles de comprender si no se visualizan gráficamente.

UML ofrece varios modelos dinámicos que ayudarán a especificar el comportamiento del sistema, como son los diagramas de interacción, los diagramas de actividad y los diagramas de estado.

A continuación, se describen una serie de elementos que forman parte de la vista dinámica del sistema y que requieren especial atención.

### **La ley de Demeter**

La **ley de Demeter** sirve de referencia clave a la hora de entender cómo se realiza la comunicación vía mensajes entre los objetos que constituyen el sistema. La ley de Demeter define a qué instancias puede enviar mensajes una instancia (objeto) determinada de una clase (ver Figura 13):

* Una instancia que esté conectada mediante un enlace navegable (es decir, una instancia de asociación de la clase).
* Una instancia recibida como parámetro en la activación.
* Una instancia creada localmente en la ejecución, o bien una variable local.
* A sí misma, es decir, la instancia en sí es el emisor del mensaje (no confundir con la asociación reflexiva).

A diagram of a bank

Description automatically generated

Figura 13. Diagrama de comunicación que ilustra la ley de Demeter. Fuente: Urbano et al., 2015.

### **Creación y destrucción de objetos**

La Figura 14 ilustra cómo se ha de modelar con UML la creación y la destrucción de objetos en un diagrama de interacción. En este caso concreto, se muestra cómo se cancela una cuenta y se transfiere el saldo a una nueva de otro tipo.

A diagram of a diagram of a bank

Description automatically generated

Figura 14. Diagramas de interacción que modelan la creación y destrucción de objetos. Fuente: Urbano et al., 2015.

### **Polimorfismo de mensajes**

Como se ha comentado con anterioridad, los objetos se comunican entre sí a través del envío de mensajes. El envío de mensajes es particularmente expresivo cuando la operación invocada se ejecuta en forma **polimórfica** en los receptores, es decir que, aunque varios objetos compartan la misma interfaz por ser de un mismo tipo (de manera directa o indirecta), cada uno puede interpretar el mensaje de manera distinta.

El uso correcto de la propiedad de polimorfismo va a facilitar la evolución del sistema en el caso de que se tenga necesidad de añadir nuevas clases a su estructura, sin que haya necesidad de hacer uso de instrucciones de ramificación múltiple, ya que **la ramificación es implícita,** es decir, interpretará el mensaje según sea la clase del objeto. La Figura 15 ilustra esta situación.

A diagram of a computer

Description automatically generated

Figura 15. Polimorfismo de mensajes. Fuente: Urbano et al., 2015.

### **5.5. Diseño modelado con prototipos**

La elaboración de prototipos es fundamental durante el diseño y desarrollo de cualquier tipo de aplicación con interfaz gráfica de usuario.

Ayuda a poner en orden nuestras ideas, explorar diferentes caminos de concepto o diseño y detectar posibles problemas o carencias antes de iniciar la fase de implementación (UOC, 2020).

Aunque solemos asumir que la creación de un prototipo de interfaz de usuario es una actividad de diseño, resulta de utilidad durante el **análisis del sistema.** Si adelantamos su elaboración a las fases iniciales de especificación de requisitos obtendremos retroalimentación temprana sobre las **necesidades reales** del usuario (Pressman, 2010, p. 177).

Su utilidad se centra en la**evaluación de la experiencia de usuario** (user experience, UX), que podemos definir del siguiente modo:

* «La experiencia que una persona obtiene cuando interactúa con un producto en unas condiciones particulares» (Arhippainen y Tähti, 2003, p. 1). La complejidad de su análisis se debe a que en la práctica hay muchos tipos de personas, productos y entornos que influyen en esta experiencia.
* «El conjunto de ideas, sensaciones y valoraciones del usuario resultado de la interacción con un producto, y se produce como resultado de los objetivos del usuario, las variables culturales y el diseño de la interfaz» (Knapp, 2003, p. 15).

Esta experiencia del usuario representa:

«[…] un cambio emergente del propio concepto de usabilidad, donde el objetivo no se limita a mejorar el rendimiento del usuario en la interacción (eficacia, eficiencia y facilidad de aprendizaje), sino que se intenta resolver el problema estratégico de la utilidad del producto y el problema psicológico del placer y diversión de su uso» (Hassan y Martín, 2005).

A continuación, en la lección magistral Prototipos de interfaz de usuario, se realiza una introducción a la técnica de prototipado en general.

Como sabemos, podemos distinguir **varios tipos de prototipos** según el grado de conformidad o fidelidad con el diseño final. Todos ellos pueden emplearse como herramientas para analizar y validar requisitos, pero generalmente se utilizan en el orden que se representa en la Figura 16, que también indica el grado de sofisticación y parecido con el producto final, y la complejidad de su construcción.



Figura 16. Tipos de prototipos de menor a mayor sofisticación. Fuente: elaboración propia.

A continuación se estudian algunas de estas técnicas de prototipado que son habituales en el ámbito del diseño de interfaz de usuario (Salgado, 2015).

### **El sketch o boceto**

Es un **primer boceto,** generalmente realizado a mano alzada sobre una hoja de papel, que representa las **primeras ideas** que tenemos sobre el diseño de interfaz de un producto. Son útiles porque permiten evaluar un gran número de alternativas de manera rápida, sencilla y económica, y permite que tanto el analista como el usuario se dejen llevar por su creatividad en un proceso de interacción muy ágil. Esta técnica es útil para responder a las siguientes preguntas, muy relacionadas con la arquitectura de información:

* Dónde posicionar elementos importantes como títulos, logotipos, menús, etc.
* Dónde estará la zona de navegación y cuáles serán sus elementos.
* Dónde se mostrará la ayuda a los usuarios.
* Dónde se mostrarán los contenidos y cuál será su estructura jerárquica.

A sketch of a website

Description automatically generated

Figura 17. Dos ejemplos de bocetos. Sketch del site de la serie True Blood en HBO (izquierda) y sketch de una web personal donde se muestra la vista de detalle de un artículo tal como la imagina su autor (derecha). Fuente: Torbjornsen, 2010.

### **Wireframes**

Apoyándonos en los bocetos, podemos dar un paso más y construir wireframes, que son:

«[…] ilustraciones bidimensionales de la interfaz de una página o aplicación que se centra específicamente en la asignación de espacio y priorización del contenido, las funcionalidades disponibles y los comportamientos deseados» (Salgado, 2015).

Son representaciones gráficas de baja fidelidad que muestran los**elementos esenciales de la interfaz** de un producto (Mkrtchyan, 2018).

Por tanto, se centran en **«qué hace la pantalla, no en cómo se ve»** y, por consiguiente, carecen de detalles como estilos tipográficos, colores o imágenes, teniendo un carácter más funcional. Se utilizan también para representar relaciones entre diferentes pantallas de una aplicación, mostrando así la estructura y los flujos de navegación. Son útiles para responder a las siguientes preguntas:

* **¿Qué** contenidos? Se trata de valorar al menos cuáles son los principales grupos de contenidos que debe mostrar la interfaz.
* **¿Dónde** ubicarlos? Se analiza la estructura de la información desde el punto de vista gráfico.
* **¿Cómo** utilizarlos? Finalmente, se investiga cómo desea el usuario interactuar con la interfaz.

Nos pueden ayudar a:

* Dar importancias relativas a los contenidos, definiendo la cantidad de espacio y disposición dedicada a cada uno de ellos.
* Conectar la arquitectura de información con el diseño visual.
* Mostrar conexiones entre páginas o pantallas, reflejando la arquitectura de navegación.
* Determinar la distribución de la funcionalidad sobre la pantalla y la cantidad de funciones disponibles.

A la hora de construir un wireframe debemos pensar en un **diseño de cajas,** en el que los diferentes elementos aparecen representados en la superficie como simples rectángulos. En este punto, es útil aplicar algunos patrones de diseño de interfaz comúnmente empleados y orientados a facilitar la interacción del usuario. Por ejemplo:

* La información suele distribuirse por orden jerárquico de importancia, de arriba hacia abajo y de izquierda a derecha.
* En el caso de las páginas web es habitual emplear la estructura común de tres secciones: cabecera, cuerpo y pie.

A screenshot of a computer

Description automatically generated

Figura 18. Algunos ejemplos de wireframes. Fuente: Lucidchart, 2020.

### **Mockups o maquetas**

Los mockups son una **representación más avanzada** desde el punto de vista**gráfico y comunicativo.** Dado su carácter más elaborado, hay dos corrientes de diseño que consideran el modo de utilizar esta técnica:

* Los que creen que debe representar el producto final de manera exacta (mockups de alta fidelidad).
* Quienes creen que se trata de una herramienta transitoria a cuya construcción no se debe dedicar demasiado tiempo (mockups de baja fidelidad).

Generalmente se construyen utilizando un wireframe previo como plantilla e introduciendo progresivamente todos los elementos visuales, haciendo una transición hasta un **modelo a escala** del producto final que permite demostrar y validar un diseño. No se espera que sean funcionales, puesto que su principal cometido es validar aspectos gráficos y visuales, aunque pueden proporcionar cierta interacción dependiendo de la herramienta empleada. Dan una visión estática de la apariencia final del producto. Resultan útiles por los siguientes motivos:

* **Nivel de precisión.** Permiten determinar de manera clara y precisa la posición, relevancia, contenido y apariencia de la información.
* **Flexibilidad.** Sirven como paso previo al diseño e implementación, pero los cambios en esta etapa son más sencillos y económicos.
* **Elemento documental previo a la ejecución.** Un mockup de alta fidelidad es más preciso para los desarrolladores, que lo emplean como una hoja de especificaciones visual o guía de estilo.
* **Calidad para el usuario.** Los clientes y usuarios prefieren visualizar un mockup porque es un resultado más próximo al producto final, favoreciendo su implicación en el proceso de análisis y validación de requisitos.

Algunos elementos que se incluyen aquí y que debemos tener en cuenta son los siguientes:

* Debemos incluir contenido, aunque sea de prueba, empleando imágenes y textos genéricos.
* Se debe decidir la paleta de colores.
* Se deben determinar con precisión las dimensiones, formas y servicios ofrecidos en las diferentes secciones de la interfaz.
* Se debe determinar la tipografía e iconografía.

A screenshot of a video slider

Description automatically generated

Figura 19. Algunos ejemplos de mockups de controles gráficos de tipo slider. Fuente: CreateLy, 2017.

### **Prototipos interactivos**

Los prototipos interactivos son modelos fácilmente modificables del sistema, que no solo incluyen la interfaz, sino también funcionalidad que relaciona entradas y salidas. Son **representaciones preliminares** muy detalladas del resultado final y son muy útiles para evaluar la interacción del usuario con el producto.

Son siempre navegables, y permiten validar de manera detallada aspectos como:

* El sistema de navegación.
* La paleta de colores, la iconografía y la tipografía.
* La experiencia de usuario y la interacción al resolver operaciones comunes.
* Servicios de búsqueda, servicios de ayuda.

Lo más interesante es que permite aproximar las metas conceptuales del usuario a la realidad práctica de implementación.

Hay que tener en cuenta que es el **nivel más sofisticado** de prototipos de interacción y, por tanto, puede no ser necesario en aquellos proyectos más sencillos y repetitivos (páginas informativas, blogs personales, etc.). Cuanto más novedoso sea el producto y más incertidumbre exista en el desarrollo del proyecto, mayor será la utilidad de este tipo de herramientas.

A person standing at a podium

Description automatically generated

Figura 20. Ejemplo de prototipo interactivo de una página web. Fuente: Protshare, 2020.

A screenshot of a cell phone

Description automatically generated

Figura 21. Ejemplo de prototipo interactivo de una aplicación móvil, donde podemos observar la estructura de navegación definida. Fuente: Fluid, 2020.

Vistas estas cuatro **técnicas de prototipado** que hemos introducido en nivel creciente de complejidad, podemos resumir algunos consejos y recomendaciones para abordar la construcción de cada una, los que se recogen en la Tabla 1.

A blue and white text box

Description automatically generated with medium confidence

Tabla 1. Consejos para la construcción de diferentes prototipos de interfaz. Fuente: elaboración propia.

Como ejemplo, en los siguientes vídeos veremos una pequeña introducción de cómo trabajar con dos herramientas de prototipado, **Justinmind** (Justinmind, 2022) y **Adobe XD** (Adobe XD, 2022).

En la siguiente lección magistral, Prototipado con Justinmind, vamos a ver una pequeña introducción a la herramienta de prototipado llamada Justinmind.

En la siguiente lección magistral, Prototipado con Adobe XD, vamos a ver una pequeña introducción a la herramienta de prototipado llamada Adobe XD.

### **5.6. Modelado de componentes**

Con el objetivo de favorecer la **reutilización** de objetos y del software en general, de una manera más uniforme, el **diseño basado en componentes** ha ido adquiriendo mayor importancia dentro la ingeniería de software. Los componentes van a permitir la creación de nuevos elementos a partir de otras piezas ya creadas, con la particularidad de ofrecer, además, un alto nivel de abstracción, ya que un componente ha de ser desarrollado de manera mucho más genérica que cualquier otro artefacto software, pero primando los principios de alta cohesión interna y bajo acoplamiento externo. A pesar de que un componente va a estar formado fundamentalmente por objetos, es importante no confundir ambos conceptos. Un componente se define como una unidad de «despliegue» independiente y presenta las siguientes características (Szyperski, 1998):

* Un componente podría ser utilizado por cualquier organización sin necesidad de ningún conocimiento previo.
* Un componente no tiene estado persistente en sí mismo, lo tienen (opcionalmente) sus objetos.
* Implementa una serie de interfaces y utiliza otras.
* Encierran otros elementos de modelado, como por ejemplo clases, permitiendo a otros componentes acceder a ellos.
* Un componente, además de clases, también puede contener otros módulos de código.

Otras definiciones bastante descriptivas se encuentran en Stevens y Pooley (2000, 11, p. 213) y en Booch, Rumbaugh y Jacobson (2006, p. 208). **Stevens y Pooley** definen un componente como «un elemento reutilizable y reemplazable, lo que requiere una interfaz bien definida, abstracción cohesiva y un bajo acoplamiento con otras interfaces». Por su parte, de manera un poco más simplificada, **Booch, Rumbaugh y Jacobson** definen un componente como «una parte reemplazable de un sistema que conforma y proporciona la implementación de un conjunto de interfaces» y donde la interfaz viene definida como «una colección de operaciones que especifican un servicio proporcionado o solicitado por una clase o componente». En este punto, conviene aclarar que las interfaces implementadas o utilizadas por un componente son realmente utilizadas y/o implementadas por el correspondiente contenido.

Para alcanzar sus objetivos, los sistemas software estarán basados en componentes que hacen uso de otros componentes, que en UML se denotará a través de una relación de dependencia.

Para poder conseguir el bajo acoplamiento, no se hace que un componente dependa de otro, sino que al final dependa solamente de una o varias de sus interfaces. En UML, el diagrama de componentes da soporte tanto a **componentes lógicos** (componentes de negocio, componentes de proceso…), como a **componentes físicos** (EJB, CORBA, COM+, JavaBeans, .NET, etc.), (OMG, 2011, p. 145). Las Figuras 22 y 23 muestran ejemplos de diagramas de componentes modelado con UML.

A diagram of a warehouse

Description automatically generated

Figura 22. Diagrama de componentes con notación UML de una tienda online. Fuente: uml-diagrams.org. s. f

A diagram of email management

Description automatically generated

Figura 23. Diagrama de componentes para un software de correo electrónico. Fuente: Ionos España, 2020.