Recursos
para profesionales
de sistemas

UML

Modelado de software para profesionales

Incluye UML 2.2

Carlos Fontela

Alfaomega

UML

Modelado de software para profesionales

Carlos Fontela

UML

Modelado de software para profesionales

Carlos Fontela

Alfaomega

Buenos Aires • Bogotá • México DF • Santiago de Chile

Fontela, Carlos
UML: modelado de software para profesionales. - 1a. ed. - Buenos Aires : Alfaomega Grupo Editor Argentino, 2011.
184 p. ; 23x17 cm.

ISBN 978-987-1609-22-2

1. Informática. 2. Software. I. Título
CDD 005.3

Edición: Damián Fernandez
Corrección: Juan Manuel Arana y Silvia Mellino
Revisión de armado: Laura Lago
Diagramación de interiores: Iris Biaggini
Diseño de tapa: Iris Biaggini

Internet: http://www.alfaomega.com.mx

Empresas del grupo:

Argentina: Alfaomega Grupo Editor Argentino S.A.
Paraguay 1307 P.B. "11", Buenos Aires, Argentina, C.P. 1057
Tel.: (54-11) 4811-7183 / 8352
E-mail: ventas@alfaomegaeditor.com.ar

México: Alfaomega Grupo Editor S.A. de C.V.
Pitágoras 1139, Col. Del Valle, México, D.F., México, C.P. 03100
Tel.: (52-55) 5575-5022 – Fax: (52-55) 5575-2420 / 2490. Sin costo: 01-800-020-4396
E-mail: atencionalcliente@alfaomega.com.mx

Colombia: Alfaomega Colombiana S.A.
Carrera 15 No. 64 A 29, Bogotá, Colombia
PBX (57-1) 2100122 - Fax: (57-1) 6068648
E-mail: scliente@alfaomega.com.co

Chile: Alfaomega Grupo Editor S.A.
Dr. La Sierra 1437-Providencia, Santiago, Chile
Tel.: (56-2) 235-4248 – Fax: (56-2) 235-5786
E-mail: agechile@alfaomega.cl

A Ana, por acompañarme durante todos estos años y brindarme el apoyo necesario para enseñar y escribir.

A Clara y Joaquín, por el tiempo que estas actividades me restan del que podría tener disponible para ellos.

Carlos Fontela

Mensaje del Editor

Los conocimientos son esenciales en el desempeño profesional. Sin ellos es imposible lograr las habilidades para competir laboralmente. La Universidad o las instituciones de formación para el trabajo ofrecen la oportunidad de adquirir conocimientos que serán aprovechados más adelante en beneficio propio y de la sociedad. El avance de la ciencia y de la técnica hace necesario actualizar continuamente esos conocimientos. Cuando se toma la decisión de embarcarse en una actividad profesional, se adquiere un compromiso de por vida: mantenerse al día en los conocimientos del área u oficio que se ha decidido desempeñar.

Alfaomega tiene por misión ofrecer conocimientos actualizados a estudiantes y profesionales dentro de lineamientos pedagógicos que faciliten su utilización y permitan desarrollar las competencias requeridas por una profesión determinada. Alfaomega espera ser su compañera profesional en este viaje de por vida por el mundo del conocimiento.

Esta obra contiene numerosos gráficos, cuadros y otros recursos para despertar el interés del estudiante y facilitarle la comprensión y apropiación del conocimiento.

Cada capítulo se desarrolla con argumentos presentados en forma sencilla y estructurada claramente hacia los objetivos y metas propuestas. Asimismo, cada uno de ellos concluye con diversas actividades pedagógicas para asegurar la asimilación del conocimiento y su extensión y actualización futuras.

Los libros de Alfaomega están diseñados para ser utilizados dentro de los procesos de enseñanza-aprendizaje y pueden ser usados como textos guía en diversos cursos o como apoyo para reforzar el desarrollo profesional.

Alfaomega espera contribuir así a la formación y el desarrollo de profesionales exitosos para beneficio de la sociedad.

El autor

Carlos Fontela es un profesional informático argentino, que trabaja en el mundo académico y en el organizacional.

En el primero de los ámbitos, se ha desempeñado principalmente en la Universidad de Buenos Aires, como profesor de grado y posgrado en áreas de Programación e Ingeniería de Software. Es reconocido por su trabajo en orientación a objetos, en el cual usa UML como herramienta de modelado y en administración de proyectos de desarrollo de software.

En el ámbito corporativo, ha trabajado en empresas privadas y en el estado. Actualmente es Gerente de Ingeniería de Software en C&S informática S.A. y Director Académico del área de capacitación, ObjectLabs.

Es autor de libros y artículos científicos en diversos temas de desarrollo de software.

Antes de comenzar a leer

En este libro se utiliza la tipografía `Courier` en los casos en que se hace referencia a código o a acciones por realizar en la computadora, ya sea en un ejemplo o cuando se refiere a alguna función mencionada en el texto. También se usa para indicar menús de programas, teclas, URL, grupos de noticias o direcciones de correos electrónicos.

Contenido

C06. Modelado del diseño de alto nivel

C07. Modelado del diseño detallado y construcción

C08. Otras disciplinas

Prólogo

UML ha cumplido más de una década y es ya un lenguaje de modelado muy difundido y establecido. Sin embargo, no todos los profesionales que lo utilizan tienen la misma visión del mismo, ni tampoco la misma valoración.

A menudo me encuentro con colegas que hacen un uso muy amplio de UML para discutir y alcanzar consenso sobre diseños de software, a la vez que muchos otros descreen de las ventajas de hacerlo, o sencillamente desconocen esta posibilidad. Hay también quienes quieren ver en UML la solución al siempre esquivo problema de la modelización de requisitos, mientras que otros opinan que sus capacidades en esta área son muy pobres. Hay incluso profesionales muy sólidos que confunden a UML con un proceso de desarrollo, o con una herramienta.

Los cambios a UML han hecho su aporte para que se produjeran estos equívocos. UML comenzó siendo un lenguaje pequeño, definido en forma más o menos laxa. Con el tiempo, el deseo de convertirlo en un estándar que, a la vez, sirviera como lenguaje de generación de programas, ha llevado a una mayor complejidad y a una mayor formalización. La mejor prueba de esto es el aumento exponencial del tamaño de las especificaciones. Hemos pasado de una especificación de un centenar de páginas hace una década, a la de varios miles en el día de hoy.

Muchas personas ven esto como una ventaja. La mayor formalización permite independizar los modelos de las herramientas, admite especificar modelos con mucha mayor precisión, facilitando la generación automática de programas, y da lugar a menos ambigüedades. Por otro lado, la mayor complejidad, que se deriva del mayor número de elementos notacionales, con su semántica asociada, ha permitido modelar más situaciones de las distintas disciplinas de la Ingeniería de Software, y más construcciones de los lenguajes de programación orientados a objetos.

Pero no todos ven esta mayor complejidad y formalización como ventajas. Tal vez la crítica no siempre se exprese en forma explícita, pero la falta de ciertos diagramas, o elementos de los mismos, en tantas herramientas que se proclaman ajustadas al estándar de UML, debería hacernos sospechar. Otro indicio es la cada vez mayor informalidad en el uso de UML en analistas y desarrolladores. Incluso muy reputados especialistas en objetos hacen un uso no normativo de UML, aduciendo que el lenguaje no les basta para expresar con claridad algunas cuestiones.

Lo que ocurre es que, como toda innovación importante, UML no ha escapado a la famosa curva de adopción de Gartner. Luego del escepticismo inicial se pasó a la adopción entusiasta, irreflexiva y tal vez atropellada, para que luego mucha gente se haya desencantado de la novedad, cayendo sensiblemente su uso. Hoy, con el lenguaje ya asentado, deberíamos estar en condiciones de entrar en la meseta de la adopción fundamentada, propia de un instrumento que ha alcanzado su madurez.

En ese marco, este libro es una guía para el uso, hecho con más pragmatismo que academicismo. No pretende ser ni un manual de referencia, ni una guía para quien debe construir herramientas que soporten UML. De allí que el libro sea conciso y se mantenga un formalismo bajo. También he querido respetar el tiempo de los profesionales de nuestra época, y esa fue otra razón para la brevedad.

Para lograr todo esto, traté de reflejar en estas páginas las construcciones más usuales

y útiles de UML, ya que, como decía poco más arriba, la complejidad creciente de UML hace imposible tratar esos temas en un libro de este tamaño. Si el lector está interesado en la documentación completa y formal, puede recurrir al sitio Web oficial en `http://www.uml.org/`.

Es importante destacar que el libro se centra en las necesidades del profesional. Por lo tanto, no enseña cómo capturar requisitos, cómo diseñar o cómo hacer pruebas, habilidades que considera conocidas a priori. Solamente muestra cómo utilizar UML como soporte de esas tareas.

De todas maneras, no es necesario conocer a fondo todas las disciplinas de desarrollo ni ninguna tecnología en particular. No hace falta tener años de experiencia en análisis o diseño. Y si bien se ha usado algo de Java en algunos ejemplos, ello se debe a que su notación, compartida parcialmente por C# y otros, es la más difundida hoy, pero no es necesario conocer nada de este lenguaje para entender los ejemplos.

El libro encara UML de una manera bastante original, ya que, en vez de estudiar cada diagrama por separado, plantea los usos que tienen los mismos en el marco de las distintas actividades de desarrollo de software. De allí que cada diagrama se vea en más de un capítulo, aunque en cada caso el enfoque va a ser diferente.

En cuanto a cómo leer la obra, creo que lo mejor es encararla en forma secuencial, ya que los distintos diagramas y conceptos se van introduciendo de a poco. Al fin y al cabo, una de las ventajas de un libro corto es que puede leerse completo. De todas maneras, si el lector ya maneja parcialmente UML, puede saltear las partes que no le aporten nuevos conocimientos.

Distintos profesionales pueden obtener distintas enseñanzas del libro. Un programador puede encontrar más útil las herramientas de UML que lo ayuden a discutir diseños con otros profesionales, mientras que un analista puede aprender aquellas que le permitan construir modelos de dominio o flujos de tareas.

Carlos Fontela

Febreo de 2011

1

MODELOS Y UML

QUÉ ES UN MODELO

UML es el acrónimo en inglés para "lenguaje unificado de modelado". Pero, ¿qué significa **modelar**? La respuesta corta sería: "construir modelos". Sin embargo, esta respuesta nos lleva a otra pregunta: ¿qué es un modelo?

La palabra **modelo** tiene varias acepciones en castellano. Para nosotros, en este libro, y en el contexto de UML, un modelo "es una descripción analógica para ayudar a visualizar algo que no se puede observar directamente y que se realiza con un propósito determinado y se destina a un público específico".

A modo de ejemplos, un mapa de transportes de una ciudad, es un modelo de la ciudad en cuestión; un plano de instalaciones sanitarias de un edificio, es un modelo de ese edificio; un dibujo de un dragón, es un modelo de un dragón.

En los ejemplos anteriores, nos referimos a representaciones de cosas que no podemos observar, pero de las que nos damos una idea a partir del modelo.

No obstante, hay varios elementos en la definición anterior que necesitamos desmenuzar. Acabamos de decir que un modelo es una descripción analógica para ayudar a visualizar algo que no se puede observar directamente, y que se realiza con un propósito determinado y se destina a un público específico. veamos:

- Descripción analógica: el modelo no es aquello que se quiere observar, sino una representación simplificada. Por esta razón, el mapa no es la ciudad, el plano sanitario no es la propia instalación sanitaria y el dibujo del dragón no es el propio dragón.
- De algo que no puede ser observado directamente: el sistema de transportes de la ciudad no se puede ver, porque es un concepto abstracto que requiere de estudio y observación; las instalaciones sanitarias del edificio

no se pueden ver a menos que rompamos las paredes y pisos del mismo; el dragón no puede verse... porque no existe.

- Se realiza con un propósito determinado: el propósito o perspectiva es lo que determina para qué realizamos el modelo. Así, el mapa de transportes sirve para saber cómo llegar de un punto a otro de la ciudad usando el sistema de transportes; en el plano de instalaciones sanitarias se indica por dónde pasan las cañerías del edificio; el dibujo del dragón, para comunicar a otras personas cómo sería un ejemplar de esta especie mitológica.
- Destinado a un determinado público: es decir, existe un público potencial que va a usar el modelo en cuestión. Por ejemplo, el mapa de transportes se realiza para un ciudadano común que necesita moverse por la ciudad o para un planificador de transportes; el plano de instalaciones sanitarias le sirve a un plomero que desee hacer una refacción de un baño, entre otros; el dibujo del dragón puede tener un público infantil o adulto.

Notemos que las cosas que modelamos no tienen siquiera que existir. Tal vez lo primero que nos venga a la mente sea la imagen del dragón, que hasta donde sabemos no existen. Pero, incluso, el mapa de transportes o el plano de instalaciones sanitarias pueden referirse a situaciones hipotéticas (porque la ciudad está evaluando distintas alternativas de planeamiento del transporte) o futuras (porque el edificio todavía no se construyó).

La finalidad última de un modelo es la comunicación de algo: un proyecto, un concepto, la descripción física de algún elemento, etc.

Precisamente, por esta necesidad de comunicar y porque, además, se destina a un determinado público, con cierto propósito, un modelo es, en definitiva, una abstracción.

Mediante la técnica de la **abstracción** se simplifica una realidad compleja. Cualquier mecanismo de abstracción se centra sólo en lo que se quiere comunicar y oculta los datos innecesarios.

Los tres ejemplos anteriores explicitaron modelos gráficos. No obstante, no tiene por qué ser así: hay modelos matemáticos, analíticos y otros. Sin embargo, los ingenieros de software consideran más amigables los modelos basados en diagramas. Además, como UML es una notación basada en modelos gráficos y, más precisamente, en diagramas, utilizamos estos ejemplos.

MODELOS DE SOFTWARE

Dado que este libro se refiere a una notación de modelado de software, nos detendremos en los modelos de software. La definición es la misma. Por lo tanto, tienen los mismos elementos:

- Descripción analógica: el modelo no es el sistema de software en sí, sino su representación.

- De algo que no se puede ser observar directamente: el software nunca puede ser observado directamente porque es intangible e invisible por su propia naturaleza; de hecho, los modelos son la única manera de "observar" el software.
- Se realizó con cierto propósito: un modelo de software puede servir, entre otras cosas, para construir una aplicación todavía inexistente, para validar conceptos con otros interesados en el desarrollo o para documentar un programa existente con el fin de facilitar su mantenimiento.
- Se destina a un determinado público: puede ser un modelo para usuarios finales, analistas, programadores, testers, etc.

Por lo tanto, un mismo sistema puede tener varios modelos, que dependen del propósito y del público al que se dirige.

Pero el modelado de software tiene una complejidad adicional. Según varias definiciones, el software es en sí un modelo de la realidad. Si así fuera, y esta cuestión es un tanto filosófica, un modelo de software es el modelo de un modelo; es decir, un meta-modelo. Puede que estemos o no de acuerdo con esta apreciación, pero lo cierto es que, en muchos casos, esto se convierte en una complejidad adicional.

POR QUÉ EL SOFTWARE NECESITA MODELOS

El software necesita modelos por las mismas razones que cualquier otra construcción humana: para comunicar de manera sencilla una idea abstracta, existente o no, o para describir un producto existente.

En efecto, el modelo más detallado de un producto de software es el código fuente. Pero es como decir que el mejor modelo de un edificio es el edificio mismo; esto no nos sirve para concebirlo antes de la construcción ni para entender sus aspectos más ocultos con vistas al mantenimiento.

Sin embargo, en el software el modelado es aún más importante que en las otras ingenierías. Esto tiene varias razones de ser:

- El software es invisible e intangible: sólo se ve su comportamiento, sus efectos en el medio.
- El software es mucho más modificable que otros productos realizados por el hombre: esta modificabilidad es percibida por los ajenos a la industria, lo que provoca que haya un incentivo mucho más fuerte para pedir modificaciones.
- El software se desarrolla por proyectos, no en forma repetitiva como los productos de la industria manufacturera. Esto hace que cada vez que construyamos un producto de software estemos enfrentándonos a un problema nuevo.
- El software es substancialmente complejo,[1] con cientos o miles de partes interactuando, diferentes entre sí, y que pueden ir cambiando de estados a

lo largo de su vida: esto hace que analizar un producto de software requiera mecanismos de abstracción y de un lenguaje para representarlo.

- El desarrollo del software es inherentemente complejo: la complejidad del producto lleva a la complejidad de los proyectos, de los equipos de desarrollo y de la administración de proyectos.

Todo lo anterior no lo hemos dicho para autoflagelarnos. En primer lugar, porque varios de estos problemas, aunque tal vez en menor grado, son también los de todas las ingenierías. Y, en segundo lugar, porque la complejidad creciente no deja de ser un desafío fascinante y una medida del éxito de la disciplina para quienes son sus "clientes".

UML

Qué es UML

UML es una notación de modelado visual, que utiliza diagramas para mostrar distintos aspectos de un sistema. Si bien muchos destacan que UML es apto para modelar cualquier sistema, su mayor difusión y sus principales virtudes se advierten en el campo de los sistemas de software. Esto no obsta para que muchos profesionales intenten usar UML en situaciones diversas, haciendo uso de esa máxima que dice que "cuando la única herramienta que conocemos es el martillo, aun los tornillos nos parecen clavos".

Surgió en 1995, por iniciativa de Grady Booch, James Rumbaugh e Ivar Jacobson, tres conocidos ingenieros de software que ya habían avanzado con sus propias notaciones de modelado. Precisamente, UML se define como "unificado", porque surgió como síntesis de los mejores elementos de las notaciones previas.

Y nos ha venido muy bien, ya que, a mediados de la década de 1990, nos encontrábamos empantanados en la falta de un estándar, aunque fuese de facto, que marcase el camino para la modelización de software orientado a objetos.

Luego UML se especificó con más rigurosidad y, en 1997, se presentó la versión 1.0, que fue aprobada y establecida como estándar por el **OMG** (*Object Management Group*, un consorcio de empresas de desarrollo de estándares). De allí en más, siguió evolucionando, formalizándose y –lo que no siempre es una ventaja– creciendo y complejizándose.

Hacia 2000, UML ya se había convertido en el estándar de facto para modelización de software orientado a objetos.

En la actualidad, UML es un lenguaje de visualización, especificación y documentación de software, basado en trece tipos de diagramas, cada uno con sus objetivos, destinatarios y contexto de uso.

Se habla de **lenguaje**, en cuanto a que es una herramienta de comunicación formal, con una serie de **construcciones**, una **sintaxis** y una **semántica** definidas. Así,

los elementos constructivos son diagramas y sus partes, la sintaxis es la descripción de cómo deben realizarse esos diagramas y la semántica define el significado de cada diagrama y elemento de los mismos.

La palabra "lenguaje" puede resultar extraña en el contexto de los ingenieros de software, pero debemos acostumbrarnos a ella porque la vamos a usar a lo largo del libro.

Además, UML es extensible. Se han definido varios mecanismos de extensibilidad, que permiten aumentar usos de UML. Este es un tema que no abordaremos, porque excede los propósitos de este libro.

De hecho, UML suele ser bastante más amplio de lo que solemos necesitar en ocasiones. Los creadores de la notación llegan a afirmar el 80% de la mayoría de los sistemas se puede modelar con el 20% de las construcciones de UML[2].

Lo que sí haremos es incluir en los diagramas, cuando sea necesario, algunos dibujos, texto y notas, que los hagan más claros.

Es importante destacar que la naturaleza de modelo basado en diagramas de UML no le impide tener una definición formal. Ya, en 1997, se formó un grupo denominado pUML,[3] que reunió a desarrolladores e investigadores para convertir a UML en un lenguaje bien definido y riguroso. Luego, el OMG adoptó el lenguaje gráfico **MOF** (acrónimo de *Meta Object Facility*) y el lenguaje textual basado en lógica de primer orden **OCL** (acrónimo de *Object Constraint Language*), que se usan para definir varios lenguajes de modelado, entre ellos UML. OCL se usa también en conjunto con UML para expresar restricciones de implementación. No obstante, MOF y OCL exceden los objetivos de este libro.

Para qué usar UML

Hay varios usos que se pueden hacer de UML, pero en haras de clasificar, podemos distinguir dos:

- Como herramienta de comunicación entre humanos.
- Como herramienta de desarrollo.

En el primer caso, usamos UML para mejorar el entendimiento de alguno o varios aspectos dentro del equipo de desarrollo, entre el equipo de desarrollo y otros interesados en el proyecto, o para documentar aspectos del desarrollo para el mantenimiento posterior del sistema.

Notemos que debido a que UML se utilizará para la comunicación, el énfasis se centrará en facilitarla. Por lo tanto, no deberían sobrecargarse los diagramas con detalles innecesarios, sino colocar solamente aquellos elementos que sean centrales al objetivo de la comunicación. También es conveniente cuidar la distribución de los elementos en el diagrama, usar colores y toda otra cuestión que mejore la legibilidad y la comprensión. Además, como es muy difícil mantener actualizados cientos o miles

de diagramas, hay que guardar solamente los diagramas que estemos seguros de mantener al día; el resto, podemos desecharlos sin cargo de conciencia.

También, para este caso, es muy recomendable hacer diagramas a mano alzada, en papel o en pizarra. Si se quisiera almacenarlos como documentación, se podrían guardar fotografías de los diagramas en una *wiki* o en cualquier otro repositorio. No hay que olvidar que la documentación debe ser más útil que abundante.

Los métodos ágiles son los impulsores de este uso de UML, más alineado con transmitir aspectos del diseño de una aplicación a un equipo de trabajo para que lo materialice en el producto, o cuando dos o más personas necesiten ponerse de acuerdo sobre un diseño, o desean discutir alternativas, y esperan visualizarlo mejor en forma gráfica.

Tal vez lo más interesante de este aspecto sea observar que UML también resulta provechoso en proyectos pequeños, usándolo en su justa medida. Decimos esto, porque a menudo se afirma que el modelado sólo es útil para documentar grandes aplicaciones.

El segundo caso es menos común en general y admite varios matices, aunque suele utilizarse bastante en proyectos grandes o cuando se recurre a metodologías muy formales.

Se trata de emplear a UML como una herramienta de desarrollo en sí misma. La más extrema de estas situaciones se da cuando se hace uso de la metodología conocida como **MDD** (*Model Driven Development* o desarrollo guiado por modelos). En este caso, se parte de modelos que surgen del análisis y, mediante una serie de pasos cuidadosamente controlados, se llega al código fuente del sistema, en forma automática.

Se ha criticado mucho esta forma de trabajo, después de la expectativa que se generó en la década de 1990 con las herramientas **CASE** y su posterior fracaso, que ha convertido a la sigla CASE en poco menos que una mala palabra.[4] Se ha dicho que el desarrollo de software es una actividad muy creativa, y que el uso de herramientas automáticas limita esa creatividad. También se ha puesto el énfasis en la mala calidad del código generado, que dificulta el mantenimiento en el caso de abandonar la herramienta. Y, además, se ha destacado la dificultad de atacar la complejidad de ciertas cuestiones de los dominios específicos. No obstante, sigue siendo válido explorar, al menos desde la investigación, la posibilidad de mecanizar todo lo que se pueda del desarrollo de software, aprovechando así las ventajas de las computadoras para realizar tareas automáticamente.

Es importante destacar que en este enfoque sí necesitamos colocar el máximo detalle posible en nuestros diagramas. Al fin y al cabo, si de los diagramas surge automáticamente el código, los diagramas deben tener el mismo nivel de detalle que éste. O, como dicen algunos autores, en estos casos “los diagramas son el código”, con lo que estamos usando a UML como un lenguaje de programación. Tampoco nos queda más remedio que utilizar herramientas para hacer los diagramas. Y, al igual

que lo que ocurría antes, la claridad de los diagramas es fundamental para facilitar su mantenimiento.

Por supuesto, hay situaciones intermedias. Existen herramientas que permiten almacenar la documentación de un proyecto, que garantizan la trazabilidad entre elementos, el almacenamiento en un repositorio versionado, y generan ciertos artefactos intermedios, que incluyen algo de código. También existen herramientas que mantienen sincronizados los diagramas y el código.

Si queremos aprovechar estas herramientas, debemos hacer los diagramas con ellas y también tendremos que respetar los formalismos que nos impongan.

Ya vamos a volver varias veces sobre estos temas en distintos puntos del libro.

Qué no es UML

Entre las falacias que se repiten alrededor de UML, una de ellas tiene que ver con que UML es una metodología o proceso. Tal vez por su semejanza de nombre con el **Proceso Unificado** o **UP**, o quizá porque los creadores de UML fueron también quienes definieron UP, ha quedado en el imaginario de los ingenieros de software una asociación muy fuerte entre UML y UP. Convengamos que el uso de la sigla UML en los libros que describen UP o la presencia de un capítulo sobre UP en libros de UML no hacen más que contribuir con la confusión general.[5]

Tal vez los creadores de UML y UP creyeron que un proceso de desarrollo "unificado" era una idea tan buena como la de una notación unificada. Pero ya hace tiempo que los ingenieros de software rechazaron esta noción de un proceso que se pueda aplicar a cualquier proyecto, aun cuando ese proceso se defina como un marco genérico a instanciar en cada caso.

Lo cierto es que UP suele estar basado en UML, en lo que se refiere a la definición de artefactos del proceso de desarrollo, pero se trata de una cualidad más bien accidental. Lo que no es cierto es lo inverso: UML no necesita de UP en lo más mínimo. UML es una notación que aplica a cualquier método de desarrollo, con la única condición –que incluso puede relativizarse– de que se use para modelar una aplicación orientada a objetos.

La otra gran falacia es la asociación de UML con una herramienta específica. Por suerte, este equívoco fue desterrado hace ya varios años.

Por lo tanto, UML no es ni se encuentra asociado a ningún proceso en particular. Tampoco se vincula exclusivamente a ninguna herramienta. Es, ni más ni menos, lo que ya dijimos: una notación de modelado de software, con la cualidad de llevarse bien con la orientación a objetos.

UML, no obstante, tiene algunas limitaciones que hacen que no pueda usarse para modelar cualquier aspecto de un producto de software. Por ejemplo, no hay un diagrama para modelar interfaces de usuario y no hay ninguna construcción para especificar requisitos no funcionales, entre otras falencias.

Sin embargo, UML admite extensiones, y hay mucho trabajo realizado en ellas para atacar los déficit antes mencionados, y para muchas otras cuestiones.

UML y la orientación a objetos

La palabra "unificado" dentro del acrónimo UML ha llevado a muchos a tratar de usar UML para modelar cualquier tipo de software, independientemente del paradigma de desarrollo.

Lo cierto es que todo puede hacerse. Sin embargo, no hay que olvidar que UML surgió en el marco del paradigma orientado a objetos, por lo que se aplica más naturalmente a ellos. Por esta razón, en el libro analizaremos a UML solamente dentro del contexto de este paradigma.

La importancia relativa de las herramientas

¿Qué herramienta o paquete de software conviene usar para hacer los modelos? En este libro no recomendaremos ninguna. Existen muchas herramientas y para todas las plataformas habituales; algunas muy buenas y, en ciertos casos, gratuitas, e incluso *open-source*.

Lo cierto es que la herramienta no es lo que más importa si se utiliza UML para la comunicación entre personas. Si, en cambio, se requiere generar artefactos –o, incluso, código– en forma automática, la herramienta puede ser de gran ayuda. Esto explica que haya algunas muy sofisticadas, con precios también elevados.

Una cuestión relevante es que muchas de las herramientas no tienen todas las características de la última versión de UML, o las tienen en versiones anteriores. Es por eso que, en algunos casos, mostraremos notaciones de versiones anteriores a la última.

En cualquier caso, lo importante es contar con herramientas adecuadas al uso que se le hará y estandarizarlas en el equipo de trabajo. Cuanto más crítico sea el sistema, menos tendencia tendremos a innovar y a usar tecnologías poco probadas.

Una consideración que debemos considerar es si la herramienta permite intercambiar datos con otros programas. Por ejemplo, existe un lenguaje derivado de XML, llamado **XMI** (XML *Metadata Interchange*), que es la forma canónica para especificar un modelo UML. Esto permite el intercambio de modelos entre distintas herramientas usando el estándar del OMG. Lamentablemente, no todas las herramientas trabajan con el formato XMI.

De todas maneras, existen herramientas que no admiten cualquier construcción de UML ni cubren totalmente el estándar. Existe un metamodelo que define a UML formalmente y con precisión, pero también están especificados niveles de adecuación o ajuste, de modo que las herramientas que manifiesten soportar UML puedan declarar hasta qué nivel lo hacen. De allí que no todos los programas sean compatibles entre sí.

Perspectivas de diagramas UML

Más allá del uso que se haga de los distintos diagramas de UML, éstos tienen distintas perspectivas, que se relacionan con lo que una persona quiera ver en ellos. Varios

autores plantearon la cuestión de las perspectivas, y esta obra no es una excepción. Sin embargo, el lector encontrará algunas diferencias derivadas de la experiencia particular del autor.

En primer lugar, a veces los diagramas se hacen con una **perspectiva conceptual**, muy alejados de la implementación, y solamente para comprender el modelo de negocio, o tal vez para bosquejar un sistema antes de analizar qué partes se van a construir como software. Se los suele denominar "diagramas conceptuales" o "modelo independiente de la computación". Usualmente, son modelos de dominio que buscan establecer un vocabulario y unas relaciones entre conceptos del dominio del problema. Son muy interesantes para reducir la brecha entre los expertos de dominio y los roles más técnicos.

En segundo lugar, nos encontramos con diagramas que implican una **perspectiva de especificación**. Se trata de aquellos que se hacen para especificar el producto de software que se construirá, pero sin entrar en detalles de implementación muy concretos. A menudo, se los denomina "modelos independientes de la plataforma", pero son más que eso, ya que no incluyen los detalles innecesarios para especificar el problema que se resolverá, dependan, o no, de la plataforma de software. Sí es importante que documenten las interfaces entre sistemas o entre partes de una aplicación, sin entrar en demasiados detalles.

También se pueden realizar diagramas con una **perspectiva de implementación**, que fijan su atención en cómo se construirá la aplicación. Éstos suelen tener el mayor nivel de detalle, aun cuando sean realizados para la comunicación entre humanos. En general, estos modelos serán específicos de la plataforma, y muy relacionados con las definiciones de diseño. Incluso, podría haber varios modelos de implementación por cada modelo de especificación.

Y, finalmente, debemos mencionar la **perspectiva del producto**: el código fuente. Si bien el código es también un modelo, no es parte –obviamente– de UML.

Hay buenas razones para la utilización de una u otra perspectiva. La perspectiva conceptual se utiliza rara vez, salvo para analizar el dominio, aunque muchas veces un glosario y un diagrama a mano alzada bastan para comunicarnos. La perspectiva de especificación suele ser muy útil para discutir alternativas de diseño macro e, incluso, algunas de uso de patrones y de comunicación entre partes de la aplicación en un nivel más detallado. La de perspectiva de implementación es la más utilizada entre diseñadores y programadores.

En lo arriba expresado, privilegiamos la visión de UML como herramienta de comunicación entre personas. Si, en cambio, utilizamos UML en el marco de herramientas CASE o de MDD, debemos cuidar mejor cada perspectiva. Incluso, en MDD, se habla de cuatro tipos de modelos: independiente de la computación o CIM, independiente de la plataforma o PIM, específico de la plataforma o PSM y de implementación.

En esta obra, al plantear el estudio de UML a través de las distintas disciplinas de desarrollo, la separación entre una perspectiva y otra se hará más natural. Sin embargo, cuando haya que aclarar qué perspectiva usamos, lo haremos en forma explícita.

Modelos de UML 2.2

En UML 2.2 (la versión de este libro) existen modelos estructurales y otros de comportamiento que –como sus nombres lo sugieren– se utilizan para modelar aspectos estructurales y de comportamiento, respectivamente, de las aplicaciones de software.

Los modelos **estáticos** o **estructurales** sirven para modelar el conjunto de objetos, clases, relaciones y sus agrupaciones, presentes en un sistema. Por ejemplo, una empresa tiene clientes, proveedores, empleados; los empleados, que tienen un legajo y un sueldo, se asignan a proyectos; los proyectos pueden ser internos o externos; los segundos tienen clientes, mientras que los primeros, no; los proyectos externos tienen costo y precio de venta, mientras que los internos solamente costo, etc.

Pero, además, existen cuestiones **dinámicas** o **de comportamiento** que definen cómo evolucionan esos objetos a lo largo del tiempo, y cuáles son las causas de esa evolución. Por ejemplo, un empleado puede pasar de un proyecto a otro; el sueldo de un empleado puede variar al recibir un bono anual; un proyecto puede pasar del estado de aprobado al de comenzado, o del de terminado al de aceptado; la preventa de un proyecto puede necesitar de ciertas actividades definidas en un flujo.

UML sirve para definir ambos tipos de modelos. Esto es interesante por la interrelación substancial entre ambas cuestiones. Aquí radica también su carácter de "unificado",[6] ya que otras notaciones previas se centraban sólo en aspectos estructurales (como los diagramas de entidades y relaciones, o DER), o sólo en aspectos de comportamiento (como las redes de Petri).

Como decíamos, UML trabaja con 13 tipos de diagramas.

Los diagramas estructurales o estáticos de UML 2.2 son:

- Diagrama de casos de uso.
- Diagrama de objetos (estático).
- Diagrama de clases.
- Diagrama de paquetes.
- Diagrama de componentes.
- Diagrama de despliegue.
- Diagrama de estructuras compuestas.

Y los diagramas de comportamiento o dinámicos son:

- Diagrama de secuencia.
- Diagrama de comunicación (o de colaboración)[7].

- Diagrama de máquina de estados o de estados.
- Diagrama de actividades.
- Diagrama de visión global de la interacción.
- Diagrama de tiempos.

Muchos autores consideran al diagrama de casos de uso como un modelo de comportamiento, incluso por quienes definieron el lenguaje. Sin embargo, esto sería así si los diagramas de casos de uso fueran un modelo del comportamiento de los casos de uso. Pero los diagramas de casos de uso de UML sólo representan una vista estática de las interacciones de usuarios con el sistema. De todas maneras, dejemos esta discusión filosófica, que sólo se entenderá cuando veamos este tipo de diagramas en otro capítulo.

Más adelante estudiaremos los diagramas de las listas anteriores. No obstante, la estructura del libro no se define por los tipos de diagramas, sino por las disciplinas del desarrollo de software. De tanto en tanto, los diagramas se combinan, como en el caso los de paquetes y de clases, o en los de secuencia y de actividades, entre otros.

Además, no estudiaremos todos los diagramas con la misma profundidad. El diagrama de visión global de la interacción es un agregado de UML 2 que se utiliza muy poco, y que es reemplazable por diagramas de secuencia o de actividad. Algo parecido pasa con el diagrama de tiempos. El diagrama de casos de uso, a pesar de su popularidad, brinda una utilidad escasa visto en forma aislada. Los diagramas de objetos estáticos se utilizan, pero son parte también del diagrama de comunicación, mucho más común, y de carácter dinámico. Por otro lado, los diagramas de clases y de secuencia tienen un uso tan difundido y son de tanta utilidad, que ameritan que les dediquemos más espacio. Y hay situaciones intermedias.

Existen otros elementos que no son parte de ningún diagrama en particular. El más notable es la colaboración, que se puede usar en combinación con varios diagramas.

Extensiones a UML

Como hemos dicho antes, UML es un lenguaje que admite extensiones. Para ello define lo que se denomina perfiles. Un **perfil** de UML "es un conjunto de extensiones que especializan a UML para su uso en un dominio o contexto particular". Por ejemplo, hay perfiles para usar en un lenguaje de programación en particular, se han creado perfiles para especificar requisitos de usabilidad en interfaces de usuario, para modelar esquemas de bases de datos relacionales, para pruebas, para arquitecturas orientadas a servicio, para seguridad, etc. Algunos se definen por el OMG y otros por iniciativas académicas, industriales o particulares.

Los perfiles pueden aplicarse de a varios en forma simultánea. Por ejemplo, podemos modelar una aplicación utilizando a la vez perfiles del lenguaje Java y de especificación de desempeño en tiempo de ejecución.

En la introducción de perfiles, se utilizan los **estereotipos** de UML, que son expresiones encerradas entre paréntesis angulares dobles. Por ejemplo, para indicar

un tipo interfaz en Java, se la representa como una clase con el estereotipo <<interface>>; en el modelado de bases de datos con UML, se manejan varios estereotipos, tales como <<primaryKey>>, para indicar que un atributo representa la clave primaria en una tabla.

Como ya dijimos, no es de nuestro interés analizar los perfiles de UML, porque hay demasiados como para poder abordarlos desde un libro de este tamaño. Sin embargo, algunas veces usaremos los estereotipos más habituales.

1 Brooks [MMM] ha llegado a decir que el software es lo más complejo e intrincado que la mente humana puede crear.

2 Ver [UML Ref].

3 Por *precise UML Group*.

4 Los informáticos somos seres muy inclinados a demonizar prácticas. Entre ellas, el desarrollo en cascada y CASE tal vez sean las más comunes.

5 Por eso, en este libro hemos minimizado esta asociación.

6 Sin embargo, no ha sido esta la intención de sus creadores al denominarlo “unificado”, como ya explicamos.

7 Los diagramas de comunicación de UML 2 son los que UML 1 llamaba de colaboración, y mucha gente los sigue denominando así.

2

DISCIPLINAS Y METODOLOGÍA

ACTIVIDADES DEL DESARROLLO DE SOFTWARE Y UML

Si bien se han hecho muchas clasificaciones de las disciplinas del desarrollo de software, todas responden más o menos a la enumeración que sigue:

Disciplinas operativas:

- Captura y validación de requisitos.
- Análisis.
- Diseño.
- Construcción.
- Pruebas.
- Despliegue.

Disciplinas de soporte:

- Administración de proyectos.
- Gestión de cambios.
- Administración de la configuración.
- Gestión de los recursos humanos.
- Gestión del ambiente de trabajo.
- Gestión de la calidad.

En las actividades de soporte no hay prácticamente nada que se pueda modelar con UML, así que nos detendremos solamente en las disciplinas operativas. Son algunas de estas disciplinas operativas las que dirigirán los capítulos centrales del libro.

Denominamos **captura y validación de requisitos**[1] a la actividad mediante la cual se determina qué es lo que quiere nuestro cliente. Éste puede ser un empleado de la misma empresa en la que trabajamos o de otra compañía, y en el caso de las aplicaciones para el mercado masivo, un desconocido. Habitualmente, es una actividad de mucha participación de clientes y usuarios.

El **análisis** a menudo se define como la actividad que determina el qué del desarrollo, porque en ella definimos el sistema que vamos a construir. Difiere de la actividad anterior en que trabajamos sobre abstracciones de software y con menor contacto con los clientes y con los usuarios. A pesar de que muchos ingenieros de software no hacen una clara distinción entre esta actividad y la de captura y validación de requisitos, aquí hemos decidido separarlas porque, además de ser conceptualmente distintas, se modelan de forma diferente en UML. Eso no quita una fuerte interacción entre ambas y su desarrollo en forma iterativa.

La actividad de **diseño** es la que define *cómo* se va a realizar lo que se determinó en el análisis. Implica todas las decisiones tecnológicas, más la estructura de implementación de la aplicación.

La actividad de **construcción**[2] es la que construye el producto tal como va a entregar. Incluye tareas de programación, construcción de la base de datos si la hubiera, optimizaciones, etc.

Las **pruebas** son actividades de validación y de verificación, que se realizan para determinar que el producto construido responde a las especificaciones del análisis y –más importante aún– a los requisitos del cliente.

El **despliegue** es la tarea que consiste en poner la aplicación físicamente en la o las computadoras en las que debe correr. Cuando la aplicación que se construye es para un mercado masivo, esto no se realiza.

A menudo, se considera que existe una actividad más, el **mantenimiento**. No obstante, todo proyecto de mantenimiento suele involucrar las mismas actividades que un nuevo proyecto completo de desarrollo de software. Decimos que mantenimiento es “la tarea que consiste en reparar, extender, mejorar un producto o adaptarlo a nuevos ambientes, pero siempre después de haber sido entregado a un cliente”.

UML tiene diagramas que nos ayudan en el modelado de algunos requisitos, del análisis y del diseño, y en algunas cuestiones del despliegue.

METODOLOGÍA DE DESARROLLO DE SOFTWARE Y UML

Hemos hecho énfasis anteriormente en que UML es un lenguaje de modelado independiente del proceso de desarrollo que utilicemos. No obstante, la naturaleza secuencial de los capítulos del libro, centrados cada uno de ellos en una disciplina distinta aplicada a un desarrollo particular, puede hacer pensar que he seguido un ciclo de vida en cascada.

Nada más alejado de mi intención. Creo firmemente en el desarrollo incremental. La presentación por capítulos asociados a actividades fue la que me pareció más adecuada a los fines didácticos, pero no implica la adopción de ningún ciclo de vida ni proceso en particular.

En este ítem, sin embargo, analizaremos muy brevemente los ciclos de vida y los procesos más habituales.

El primer ciclo de vida que se definió fue el denominado posteriormente **desarrollo en cascada**, que consiste en ir cumpliendo una serie de etapas, cada una separada de las otras, de modo tal que recién se empieza una etapa cuando se terminó la anterior. Cada etapa corresponde a una actividad de desarrollo, y es efectuada por un grupo de personas especializadas en esa tarea, que generan un conjunto de documentos como cierre de la etapa. Por lo tanto, el ciclo de vida en cascada es un modelo en que las etapas están asociadas a las distintas actividades, de modo tal que se cumple una sola actividad por etapa.

La figura 2.1 muestra un diagrama de estados del ciclo en cascada (si bien veremos formalmente el diagrama de estados más adelante, es tan intuitivo que podemos usarlo aquí):

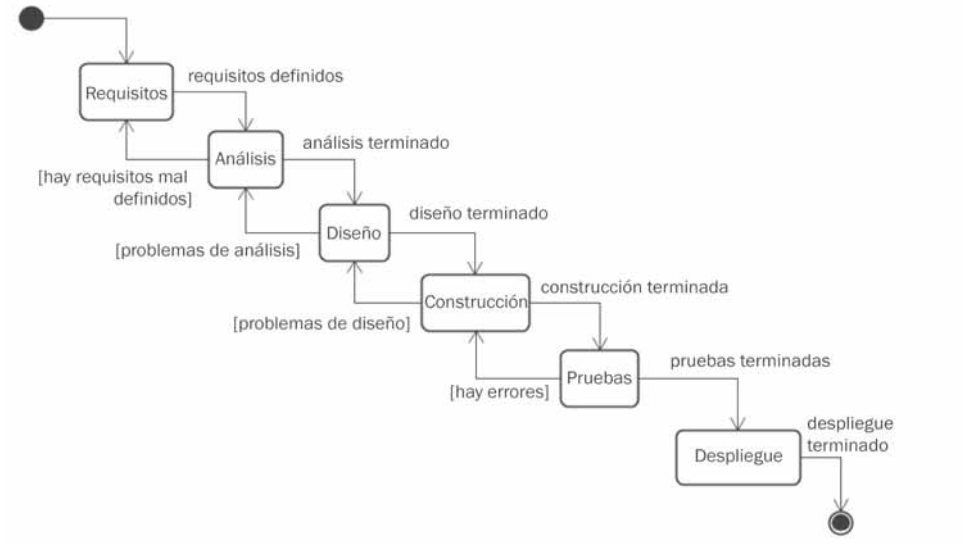

Figura 2.1 Ciclo en cascada.

A pesar de su simplicidad y aparente naturalidad, este modelo de ciclo de vida ha mostrado serias falencias, salvo en proyectos pequeños o muy especiales.

Lo que ocurre es que el ciclo de vida en cascada impone una rigidez a los cambios de requisitos, tan habituales en los proyectos de desarrollo de software. En los primeros tiempos, incluso se fomentaba esa rigidez, al pretender congelar los requisitos en etapas tempranas del proyecto, llegando al problema que se denominaba **parálisis de análisis**. Además, al ser el software un producto invisible, impide que los

interesados en su desarrollo puedan ir viendo lo que se está desarrollando, hasta el final de todo el proyecto, cuando ya no hay chances de modificar nada. Ni siquiera los *testers* del equipo de desarrollo pueden ir realizando pruebas que validen la arquitectura o cuestiones globales de la aplicación que, de requerir modificaciones, afectarían en gran medida el diseño. Finalmente –y tal vez no menos importante– provoca proyectos largos por la imposibilidad de superponer actividades sobre partes diferentes del producto.

Todas estas críticas no desconocen que hay un marco general del proyecto que hay que definir en una etapa temprana, que incluyen una visión del proyecto, algunos requisitos y un diseño macro, aun cuando admitamos refinamientos posteriores.

Como respuesta a la rigidez del modelo en cascada fueron surgiendo los ciclos evolutivos o incrementales. Lo que buscan estos ciclos de vida es permitir todo aquello que el ciclo por etapas no permite, principalmente la evolución de los requisitos y la entrega del producto en forma incremental. Por eso las etapas no se definen por las disciplinas, sino por las funcionalidades que se entregarán al final de cada una. También mejoran la visibilidad de los interesados y las posibilidades de probar tempranamente el producto. Y coadyuvan a disminuir los riesgos de los proyectos. Si las décadas de 1970 y 1980 fueron el reino del modelo en cascada, las de 1990 y 2000 marcan el auge de los métodos iterativos.

La diferencia fundamental entre el ciclo en cascada y los ciclos incrementales está en cómo subdividen el cronograma de tareas: en el primero, son las actividades las que se programan en el tiempo; en los segundos, las funcionalidades.

La primera respuesta al ciclo en cascada fue plantear cascadas parciales, que se realizaban en forma iterativa. Esta modalidad se denomina **desarrollo incremental** o **en espiral**. Otro planteo, bastante similar en sus resultados, fue el de desarrollar a **través de prototipos**, de modo tal que el refinamiento sucesivo de los mismos fuera llevando gradualmente al sistema final. Un **prototipo** es toda versión preliminar, intencionalmente incompleta y en menor escala de un sistema, aunque puede (y según muchos, debe) ser un producto que se pueda entregar. La figura 2.2 muestra el diagrama de estados de un ciclo de vida en espiral.

El planteo que siguió en el tiempo fue el del **proceso unificado de desarrollo de software** (**UP**), que mantiene la noción de disciplina de desarrollo de software, pero no impone una correlación entre etapa y disciplina, sino que define cuatro fases (inicio, elaboración, construcción y transición) que involucran todas ellas, superponiéndolas en el tiempo, cada una con hitos de terminación bien definidos. Cada fase puede dividirse en iteraciones, y algunas de ellas admiten entregas parciales del producto. Además de su carácter iterativo, entre sus prácticas destacadas están el modelado visual, con un abundante uso de UML, el énfasis puesto en el control de cambios y en la administración de requisitos, el uso de una arquitectura basada en componentes, con foco en su robustez desde el comienzo y la verificación constante de la calidad. La figura 2.3 muestra un diagrama de estados del ciclo de vida de UP.

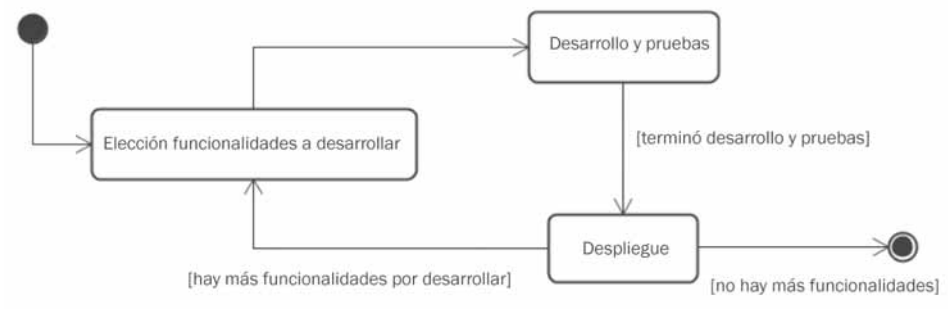

Figura 2.2 Ciclo en espiral.

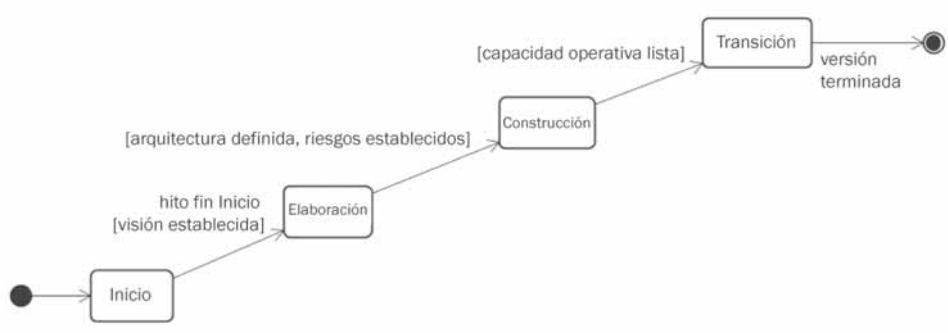

Figura 2.3 Ciclo de vida de UP.

UP parte de un modelo de casos de uso, y en todo momento está dirigido por los casos de uso que se van agregando y refinando. Las iteraciones se hacen implementando casos de uso en forma completa, y empiezan por los que tienen mayor importancia para el usuario o por los que tendrán un mayor impacto en la arquitectura de la aplicación si no contamos con ellos al inicio. Y las pruebas se hacen también verificando las especificaciones de los casos de uso. Por eso, los casos de uso integran el trabajo a través de las distintas disciplinas: se capturan y se definen en requisitos, se realizan en el análisis, en el diseño y en la implementación y se verifica que se satisfagan en las pruebas.

Se lo suele considerar un método formal por la gran cantidad de artefactos que recomienda como "entregables" y la gran cantidad de roles que define, además de su escasa consideración hacia las personas concretas.

Como respuesta a los procesos muy formales, a mediados de la década de 1990, se comenzaron a definir metodologías más livianas, más adelante bautizadas como ágiles. Incluso se escribió, en 2001, un *Manifiesto ágil*, firmado por varios reconocidos especialistas.

Si bien esta clasificación tiende a ser demasiado rígida, lo cierto es que hay métodos de desarrollo más ceremoniosos y otros más espontáneos.

Usando el marco del manifiesto, surgieron varios métodos, entre los que destacan especialmente ***Extreme Programming*** (o programación extrema, a menudo abreviada **XP**), un conjunto de prácticas ágiles de desarrollo centradas en la programación y en el diseño, y ***Scrum***, un marco de desarrollo que no define el proceso ni los "entregables" a un nivel de detalle, pero que a la vez da una referencia general para la construcción de software. Incluso, a varios métodos más formales se les definieron sus variantes ágiles, como ocurre con AUP (acrónimo de *Agile Unified Process*), que es la variante ágil de UP.

Respecto de los métodos formales y los métodos ágiles, reconozcamos que no hay un enfoque que sea mejor que otro: hay ocasiones para ser más ceremonioso y otras para la espontaneidad. Lo cierto es que el método que se empleará depende mucho del tamaño de los proyectos, del cliente, de la tecnología, de la criticidad del sistema, el grado de confianza entre cliente y desarrolladores y la necesidad, o no, de generar una mayor cantidad de entregas incrementales, con mayor valor para el cliente, en el menor tiempo posible. Por ejemplo, en un equipo de desarrollo muy grande, es necesario mucha más comunicación formal, lo que implica más documentación y burocracia. Además, los sistemas comerciales habitualmente sufren muchos cambios de requisitos, y por eso se adaptan mejor a los métodos ágiles de desarrollo.

Lo que ha favorecido a los métodos ágiles es su capacidad para entregar valor en el menor tiempo posible, puesto que pueden liberar un producto en forma parcial sin tanto protocolo. En la actualidad, además, ante la urgencia por salir al mercado, estos métodos suelen ser más dúctiles, ya que permiten la interrupción de un desarrollo, sin que pierda su utilidad. Esto es fundamental para el que paga, puesto que puede interrumpir un proyecto cualquier momento, y para el que cobra, ya que puede ir financiando el desarrollo mediante el cobro por entregas parciales.

Es importante observar que una de las características de los métodos ágiles es que necesitan una documentación menos detallada. XP llega a descreer de la documentación de desarrollo una vez que éste se terminó. Esto es así porque el énfasis está puesto en la comunicación cara a cara y por la dificultad de mantener en sincronismo los diagramas y el código. Sin embargo, no quiere decir que no se necesite nada de documentación, ni que no se usen diagramas para comunicar ideas entre personas. De allí que UML sea válido en el marco de los proyectos ágiles. Tal vez lo que haya que destacar es que los diagramas que se usan en los métodos ágiles tendrán un grado de detalle menor y se utilizarán en menor cantidad. Incluso, existe una metodología denominada **AMDD** (*Agile Model Driven Development* o *desarrollo ágil guiado por modelos*). Hasta los impulsores de XP afirman que se pueden seguir sus buenas prácticas aunque se recurra a herramientas que generan código a partir de los diagramas, pues, en ese caso, el sincronismo se garantiza por la preeminencia de los diagramas.

Como este es un libro de UML, no vamos a incursionar más en temas de metodología. Con lo que acabamos de ver ya hemos demostrado que los procesos de desarrollo son independientes del uso, o no, de UML.

EL LENGUAJE UNIFICADO DE MODELADO

El lenguaje unificado de modelado —que tratamos en esta obra— es lenguaje, es unificado y es de modelado.

Es un **lenguaje** porque define la sintaxis y la semántica de los distintos elementos que se utilizan para la comunicación entre personas, entre éstas y las máquinas e, incluso, de las máquinas entre sí.

Es **unificado** porque se puede utilizar –o, al menos, eso pretende– para todas las actividades del desarrollo de software en forma uniforme. Esto es: hay conceptos, como en los casos de uso, de clase, de objeto o de mensaje, que se usan en más de una actividad. Otra acepción de unificado que también se utiliza es la que enfatiza la independencia del proceso de desarrollo, lo que permite que se recurra a él con cualquier metodología. Y, finalmente, nos encontramos con la razón inicial de este calificativo, que se debe a que sus creadores unificaron criterios y notaciones anteriores al crear UML.

Y es **de modelado** porque su finalidad es servir para modelar sistemas de software.

Detengámonos un poco en lo expresado hasta aquí. Es importante que analicemos la pretensión de unificación de todas las disciplinas bajo un mismo lenguaje. Lo cierto es que UML ha resultado excelente para modelar análisis y diseño. En el modelado de requisitos, ha demostrado algunas falencias, mientras que en el resto de las actividades sólo presta una ayuda limitada.

La idea de que se puede usar en el marco de cualquier proceso de desarrollo es indiscutible. Quedaría por analizar más detenidamente por qué algunos metodólogos se niegan a usarlo. A juicio del autor de estas líneas, es más una discusión religiosa que un análisis basado en el raciocinio. Al fin y al cabo, sólo tiene sentido discutir qué construcciones de UML son necesarias en el marco de cada proceso, pero sin descartar el lenguaje como un todo.

En cuanto a que permite la comunicación entre humanos y computadoras indistintamente, no todos están de acuerdo. El desacuerdo no pasa por la imposibilidad, sino por la conveniencia, o no, de estos enfoques. Muchas personas sostienen que UML es bueno para la comunicación entre humanos y, por lo tanto, descreen de las formalidades del lenguaje. Otros, tal vez más cerca de MDD, enfatizan en las definiciones formales de UML –que, por supuesto, existen– que les permitan la generación de código a partir de modelos. En este libro, no entraremos en estas cuestiones, aunque sí seremos pragmáticos y mostraremos lo que más uso tiene dentro de la notación.

Finalmente, UML no es un lenguaje que se utilice siempre de la misma manera. Lo dicho en las páginas anteriores es bastante elocuente. Sin embargo, a riesgo de ser reiterativos, recordemos que un mismo diagrama se puede usar en distintas disciplinas y, en consecuencia, variará la manera de realizarlo. El cómo de los diagramas UML también depende de las distintas fases del proyecto: no es lo mismo usar UML para el análisis o el diseño preliminar, que recurrir a él para comprender un diseño de un sistema ya construido y que debemos mantener. Además, como decíamos poco más arriba, si UML se realiza para comunicar una idea entre

humanos en un pizarrón que luego se va a borrar, el grado de detalle y de formalidad será necesariamente menor que si se lo utiliza para la documentación formal o para derivar el código.

Todas estas cuestiones se tratarán en este breve libro.

1 He cuidado mucho decir “requisitos” y no “requerimientos”, que es el término más habitual, porque esta última no es una palabra castellana, sino un barbarismo proveniente del inglés *requirements*, mal traducido con mucha frecuencia. La palabra “requerimiento” se suele usar en castellano en el sentido de solicitud o pedido.

2 A veces se la llama “implementación”, aunque lo equívoco del término hizo que no lo utilizara en este libro.

3

RESOLUCIÓN DE UN PROBLEMA DE DESARROLLO DE SOFTWARE

EL PROBLEMA

Vamos a trabajar, a lo largo de esta obra, sobre una aplicación de seguimiento de proyectos de una organización que desarrolla software utilizando *Scrum* como *framework* metodológico.

¿Por qué elegimos este ejemplo? ¿No hay nada parecido en el mercado? Muy lejos de ello, aplicaciones de gestión como la que estamos nombrando hay centenares, y cada año se agregan muchas más.

Sin embargo, al elegir este problema, primó la finalidad didáctica. Esto es, como es un problema conocido, se nos hará más fácil ir desarrollando UML sin necesidad de explicar demasiado los elementos de dominio, los requisitos detallados, etc.

Para facilitar la referencia al sistema, le daremos un nombre de fantasía, *FollowScrum*.

Para organizar mejor el problema, partamos de unos requisitos de muy alto nivel, tal como suelen ser enunciados por un cliente en una charla informal.

En la primera versión, *FollowScrum* debe:

- Brindar la aplicación vía Web a quien quiera usarla, a modo de software como servicio. Es decir, los clientes no tendrán el software instalado en sus propias máquinas, sino que usarán el servicio, pagando un abono mensual.
- Poder trabajar con varias organizaciones, definiendo usuarios por organización y manteniendo separadas las distintas organizaciones, de modo que nadie pueda ver la información de una organización que no es la propia.

- Mantener el legajo del personal de cada organización, permitiendo agregar, modificar y eliminar sus datos. También se le cargará datos desde otros sistemas externos.
- Permitir definir costos por empleado.
- Mantener la lista de proyectos de cada organización, permitiendo dar de alta, modificarlos y dejarlos como inactivos una vez terminados. Los proyectos se consideran tales aun cuando no hayan comenzado y estén en una etapa de pre-venta.
- Definir tres fases estándar al modo de *Scrum*, para todos los proyectos, y luego iteraciones dentro de la fase central.
- Guardar documentos que cada organización haya definido como estándares o plantillas que usa en sus proyectos. Una plantilla puede ser un documento en blanco.
- Generar instancias de las plantillas, haciendo documentos concretos a partir de ellas, guardarlos y consultarlos; también imprimirlos.
- Permitir la definición, estimación, priorización y asignación de funcionalidades, con sus ulteriores modificaciones y su posible eliminación.
- Emitir reportes de los proyectos e iteraciones cerrados de cada organización, que incluyan reportes de rentabilidad, de fidelidad de la estimación en tiempos y en costos, de retrabajo, de velocidad, etc.
- Para proyectos y fases en ejecución de cada organización, emitir reportes de avance, *burndown charts*, velocidad, proyecciones de costos y tiempos, etc.
- Manejar roles de usuarios que estén autorizados a realizar diferentes tareas dentro del sistema. Cada empresa definirá sus roles y sus permisos.
- Mantener un *log* o bitácora de todas las acciones que los usuarios hagan sobre el sistema.
- Habrá un administrador del sistema como un todo, que accederá con una interfaz de escritorio gráfico a la aplicación, y permitirá dar de alta e inhabilitar empresas por falta de pago.
- Para evitar el ataque a datos sensibles, el sistema estará ubicado en un servidor de aplicaciones seguro, y los datos persistentes se almacenarán en otro servidor más seguro aún, al que sólo podrá acceder la aplicación que estamos construyendo.
- Otros sistemas podrán acceder a algunas funcionalidades de *FollowScrum*, que proveerá al efecto interfaces de integración tipo servicios web.

Aunque sea una decisión de diseño adelantada, trabajaremos con Java, porque es la plataforma que mejor maneja nuestro personal.

Como se ve, los requisitos que tenemos son de granularidad muy alta y muy vagos, y los funcionales están mezclados con los demás, pero así es como comienza en general un proyecto de desarrollo.

BREVE DESCRIPCIÓN DE SCRUM

Dado que la aplicación sobre la que vamos a trabajar, *FollowScrum*, es un sistema de gestión de proyectos siguiendo *Scrum*, es necesario explicar algunos conceptos básicos del método. Si no lo hiciésemos así, aunque sea en forma muy resumida, no podremos abordar los capítulos que siguen. Si el lector ya conoce *Scrum*, puede saltearse este ítem.

Desde ya, esto no implica ninguna vinculación necesaria entre *Scrum* y UML, que muchos fundamentalistas considerarían herética.

Como decíamos, *Scrum* es un proceso marco para desarrollo ágil, que da una referencia general para la construcción de software.

Scrum estructura el desarrollo del producto en ciclos o iteraciones que denomina *Sprints*. La idea central es que un *Sprint* se fije objetivos (funcionalidades que desarrollará) al comienzo del mismo, y que el trabajo acordado esté finalizado al terminar.

El equipo de *Scrum* está conformado por tres roles:

- Un único *Product Owner*, que hace las veces de cliente en el lugar.
- Un *Scrum Master*, que debe velar porque se den las condiciones para trabajar, removiendo obstáculos, y por el seguimiento de las prácticas de *Scrum*.
- Varios *Team Members*, que diseñarán, codificarán y probarán las funcionalidades definidas para el *Sprint* en curso.

El *Product Owner* es quien, sobre la base de los requisitos planteados por clientes y usuarios, elabora la lista de requisitos, denominada *Product Backlog*, y les asigna prioridades.

Antes de comenzar un *Sprint*, el *Product Owner* discute y define con el resto del equipo qué requisitos o ítems del *Product Backlog* habrá que desarrollar en el *Sprint*, construyendo con ellos el *Sprint Backlog*. Se parte de requisitos para generar tareas. El *Sprint Backlog*, por lo tanto, no es un subconjunto del *Product Backlog*, ya que sus ítems son tareas, mientras que los de éste son requisitos.

En cuanto a la organización de los *Sprints*, también existen algunas reglas.

Durante un *Sprint*, no se pueden cambiar los integrantes de un grupo de trabajo ni el *Sprint Backlog*. Lo único que se puede hacer es cancelar un *Sprint* por razones de fuerza mayor.

Durante el *Sprint,* se realizan diariamente las *Scrum Daily Meetings*, en las que participa todo el equipo, y en las que se analiza el avance y el trabajo del día. Estas reuniones son las que dieron nombre al método. El *Scrum Master* hace de moderador de estas reuniones.

Al finalizar el *Sprint,* se realiza una reunión denominada *Sprint Review Meeting*, de la que se obtienen las lecciones aprendidas, que se dejan registradas en un artefacto denominado *Sprint Retrospective*.

En cuanto a las métricas, se pueden usar las que desee el equipo. No obstante, hay un reporte típico de *Scrum*, denominado *Burndown Chart*. El objetivo de este gráfico es hacer un seguimiento sobre la base del trabajo que falta por hacer. Se puede realizar durante el *Sprint,* en cuyo caso se llama *Sprint Burndown Chart*, y muestra día a día cuánto trabajo falta realizar dentro del *Sprint,* y su relación con el que se esperaba realizar. También se puede proceder *Sprint* a *Sprint* el *Product Burndown Chart*, en el nivel del proyecto.

Visto en forma más global, un proyecto *Scrum* tiene tres fases:

- Inicio: incluye la planificación de una versión, con una primera estimación en tiempo y costo, y un diseño de alto nivel.
- Fase iterativa, con varios *Sprints*.
- Cierre: preparación para la versión desplegable, documentación final y entornos necesarios.

DISCIPLINAS Y CAPÍTULOS

En los próximos capítulos, trabajaremos sobre el enunciado de *FollowScrum* planteado un poco más arriba, a razón de un capítulo por disciplina operativa, centrándonos en las que tienen mayor uso de UML.

El hecho de encarar una disciplina por capítulo no implica, de ninguna manera, el modelo de ciclo de vida en cascada ni ningún modelo que asocie etapas o hitos de calendarios con disciplinas particulares. Separamos el desarrollo en actividades solamente para poder analizar mejor los usos de UML en las distintas disciplinas.

Incluso puede parecer irónico que usemos UML para modelar cuestiones de una aplicación que gestiona proyectos con *Scrum,* cuando los impulsores de *Scrum* no son precisamente muy afectos al modelado, o al menos a un modelado muy estricto. No obstante, *Scrum* está mucho más allá de una cuestión de modelado o no-modelado.

Por lo tanto, en los capítulos que siguen veremos el modelado de software en:

- Requisitos del cliente.
- Análisis o definición del sistema.
- Diseño de alto nivel.
- Diseño detallado y construcción.

En cada uno de los capítulos, veremos los elementos de UML que más contribuyen a cada disciplina. Prestaremos especial atención a la manera y el grado de detalle en que se los suele usar en el contexto de cada actividad particular.

Esta elección del modo de presentar UML puede resultar impráctica en algunos casos. Por eso, al final del libro, hay un capítulo de cierre que muestra todos los usos posibles de cada diagrama. De esta manera, tenemos una especie de cuadro de doble entrada: una entrada por capítulo, de modo tal de ver los diagramas usados en cada

disciplina, y una entrada por tipo de diagrama en el último capítulo, con las disciplinas que se pueden abordar con cada diagrama.

De todas maneras, no seamos excesivamente optimistas. No vamos a resolver el problema completo, ni mucho menos. Hemos elegido esta aplicación para que nos sirva de guía y poder ir analizando distintas actividades con un marco lo más real posible. Un libro de este tamaño no alcanzaría ni remotamente para hacer un desarrollo total de *FollowScrum*. Por esta razón, en ocasiones simplificaremos notoriamente el problema para hacer más accesible lo que queremos mostrar, a la vez que, en algunos casos, abordaremos un tema con un nivel de detalle mucho mayor para poder mostrar alguna característica particular de UML.

4

MODELADO DE REQUISITOS DEL CLIENTE

INGENIERÍA DE REQUISITOS Y TIPOS DE REQUISITOS

Dentro de la Ingeniería de Software, hay una disciplina que se denomina Ingeniería de Requisitos. Está orientada a desarrollar requisitos a través de un proceso cooperativo e iterativo de analizar el problema, documentar las observaciones resultantes y chequear lo obtenido.

Tradicionalmente, se distinguen tres actividades principales de la Ingeniería de Requisitos: captura o elicitación, modelado y validación de requisitos.

Los requisitos de un sistema de software se suelen clasificar en funcionales y no funcionales. Los **funcionales** son descripciones de lo que el sistema hace o debe hacer. Los **no funcionales** son restricciones globales sobre cómo debe construirse y funcionar el sistema.

A menudo, también se mencionan otros requisitos, denominados **organizacionales**, que engloban las razones de la creación del sistema, las restricciones del ambiente en el que el sistema funciona y el significado de los requisitos del sistema.

Como veremos, UML presta ayuda –no demasiada– en la modelización de requisitos funcionales. No hay ningún artefacto del lenguaje que ayude en la modelización de requisitos no funcionales. Respecto de los requisitos organizacionales, sólo contamos con la posibilidad de hacer algún modelo de negocio usando algún diagrama de UML fuera de su uso habitual.

CASOS DE USO

Casos de uso

Los **casos de uso** son herramientas de modelización de requisitos funcionales, que preceden (en el tiempo) y exceden (en alcance) a UML. Pero fueron UML y UP los que les dieron mayor difusión. Tal vez por esa asociación con UP, los métodos ágiles evitan hablar de casos de uso.

Un caso de uso especifica una interacción entre un actor y el sistema, de modo tal que pueda ser entendida por una persona sin conocimientos técnicos. Es importante también que capte una función visible para un actor. Es posible que sirva de contrato entre el equipo de desarrollo y los interesados en el mismo.

Los **actores** son los roles de los agentes externos que necesitan algo del sistema. Pueden ser personas o no: por ejemplo, un actor puede ser otra aplicación que se comunica con la nuestra para solicitar algún servicio. Pusimos también el énfasis en destacar que son roles y no personas con nombre y apellido. Por ejemplo, el empleado de una empresa cliente de *FollowScrum* puede ser, a la vez, administrador y usuario de reportes, pero como actores se trata de dos roles diferentes.

En el cuadro 4.1, se muestra un ejemplo de caso de uso para dar de alta un nuevo cliente en *FollowScrum*:

NOMBRE	AGREGAR EMPRESA
Actores	Administrador
Descripción	Permite agregar nuevas empresas clientes del sistema.
Disparador	Llega un mail de administración con los datos de un nuevo cliente.
Precondiciones	1. El Administrador debe estar logueado en el sistema.
Postcondiciones	1. Queda una nueva empresa activa en el sistema. 2. Se envía un mail al usuario administrador del cliente notificándole que la empresa ha sido agregada al sistema, junto con el nombre de usuario y clave de acceso. 3. Se deja un rastro de auditoría en el log del sistema.
Flujo Normal	1. El usuario solicita dar de alta una nueva empresa (**S1**). 2. El sistema muestra los datos a ser ingresados (**S1**): a. Nombre (*). b. Domicilio. c. Nombre del administrador del cliente (*). d. Mail del administrador del cliente (*). e. Teléfono de contacto (*). 3. El usuario completa los campos (**S1**). 4. El sistema valida los datos. (**E1 a E4**).

NOMBRE	AGREGAR EMPRESA
Flujo Normal	5. El sistema guarda los datos en la base de datos. 6. El sistema genera un usuario y una clave para el administrador del cliente. 7. El sistema envía un mail al administrador del cliente notificándole que la empresa ha sido agregada al sistema, junto con el nombre de usuario y clave de acceso. 8. El sistema guarda en el log el nombre de usuario (Administrador) y la acción realizada (“Alta de nueva empresa: <nombre de empresa>”). 9. Finaliza el caso de uso.
Flujos alternativos	S1. El usuario abandona la carga sin terminar antes del paso 4 del flujo normal. S1.1. El sistema pregunta al usuario si desea abandonar. S1.2. Si la respuesta del usuario es positiva, el sistema guarda en el log el nombre de usuario (Administrador) y la acción abandonada (“abandonó el alta de una empresa”) S1.3. Finaliza el caso de uso.
Excepciones	E1. No se cargaron todos los datos requeridos. E1.1 El sistema indica que existen datos requeridos no cargados. E1.2 Vuelve al flujo principal, paso **3**. E2. Ya hay cargada una empresa con el mismo nombre. E2.1 El sistema informa que ya hay una empresa con el mismo nombre. E2.2 Vuelve al flujo principal, paso **3** (el usuario podrá abandonar o cambiar el nombre de la empresa). E3. El mail o el teléfono ingresados no son válidos. E3.1 El sistema indica que los datos ingresados no son válidos. E3.2 Vuelve al flujo principal, paso **3**.
Prioridad	Alta
Frecuencia de uso	Media
Reglas de negocio	Al crear un cliente, éste siempre queda en estado activo.
Requerimientos especiales	-
Suposiciones	-
Notas y preguntas	*: el dato es obligatorio

Cuadro 4.1 Caso de uso “Agregar empresa”

La plantilla puede ser la que acabamos de usar o alguna parecida. Las secciones de la plantilla significan:

- Nombre: nombre corto, que identifique al caso de uso (en lo posible debiera ser verbal). En general, el nombre debiera consistir en el objetivo del actor principal (o **iniciador**) en el caso de uso.
- Actores: tipos de usuarios que actúan en el caso de uso. En ocasiones, es útil separar el actor iniciador de otros actores secundarios.
- Descripción: descripción más detallada de la interacción del caso de uso, en una oración simple.
- Disparador: evento que provoca que el actor iniciador deba abordar las actividades del caso de uso.
- Precondiciones: situación en la que deben estar los actores y el sistema antes de comenzar el caso de uso.
- Postcondiciones: cambios en el sistema y en el medio producidos por la ejecución normal y exitosa del caso de uso.
- Flujo Normal: descripción de los pasos del caso de uso, tal como se espera que se realicen en una situación normal.
- Flujos alternativos: descripción de los pasos del caso de uso, a partir de un cierto paso del flujo principal, cuando éste se aparte de la situación habitual.
- Excepciones: descripción de los pasos del caso de uso, a partir de un cierto paso del flujo principal, cuando éste produzca un error o situación excepcional.
- Prioridad: grado de prioridad que tiene la implementación de este caso de uso en el sistema para el cliente.
- Frecuencia de uso: para el cliente, subjetivo.
- Reglas de negocio: aclaraciones que hagan a las reglas de dominio y que no hayan sido especificadas antes.

Si bien es una herramienta para requisitos funcionales, los no funcionales que estén asociados a un solo requisito funcional se pueden incluir en el caso de uso en forma textual.

Es importante destacar que no todo caso de uso requiere el nivel de detalle del de más arriba. Muchas veces, conviene centrarse más en el qué que en el cómo, y en esas situaciones escribimos casos de uso de más alto nivel, con menor grado de detalle. Por ejemplo, cuando escribimos: “El sistema guarda los datos en la base de datos”, estamos presuponiendo que habrá una base de datos, que es una decisión de diseño, del cómo. No obstante, dejando fundamentalismos de lado, lo cierto es que, muchas veces, existen restricciones de diseño que surgen de los requisitos, con lo cual, expresiones como la citada no son tan inusuales en los casos de uso.

También ocurre que no siempre los casos de uso se utilizan para especificar requisitos de un futuro sistema. A veces, se los utiliza también para describir procesos de negocio.

Hay varios procesos de desarrollo que utilizan los casos de uso para dirigir el análisis, el diseño y las pruebas, ya que todos los elementos se pueden estructurar a partir de los casos de uso. UP fue el primer proceso que hizo ese planteo, pero buena parte de los demás métodos estructura el desarrollo basándose en requisitos funcionales, independientemente del nombre que les dé a éstos.

Además, los casos de uso son, en cuanto a los requisitos funcionales, la base de la trazabilidad de los demás modelos del sistema, que pueden llegar hasta el código.

Una alternativa: User Stories

Hay métodos de desarrollo que no utilizan casos de uso, destacándose los *user stories* de XP y la mayor parte de los métodos ágiles. En realidad, una ***user story*** es un requisito expresado de manera simple y en términos del usuario, por esta razón podría ser muy similar a un caso de uso de poco detalle. Lo único que las diferencia de éstos es que no hay tanta formalidad en su descripción. Habitualmente, una user story se expresa con una oración en los siguientes términos:

```
Como <rol>
Quiero que el sistema haga <funcionalidad>
Para obtener <beneficio esperado>
```

Por ejemplo, aquí hay una user story para nuestra aplicación *FollowScrum*:

```
Como administrador
Quiero que el sistema permita dar de alta una nueva
empresa cliente, incluyendo al administrador del cliente,
más su usuario y clave
Para permitirle el uso del sistema
```

La idea de las user stories es que, al definir solamente requisitos de alto nivel, sirven para tener una visión global del alcance y de los beneficios esperados. Además, sirven para hacer estimaciones gruesas, planificaciones y seguimiento de los proyectos de desarrollo.

Como una *user story* no alcanza para precisar en detalle un requisito, a menudo se la acompaña con **pruebas de aceptación** del usuario (*user acceptance test* o **UAT**).

A los efectos del resto de las cuestiones, cuando hablemos de casos de uso, se puede reemplazar este término por el de *user story*, tal vez acompañada de sus UAT.

Escenarios

Las instancias de casos de uso o de *user stories* se denominan **escenarios**. Un escenario típico es:

```
    El usuario solicita dar de alta una nueva empresa

    El sistema muestra los datos a ser ingresados:

           Nombre (*)

           Domicilio

           Nombre del administrador del cliente (*)

           Mail del administrador del cliente (*)

           Teléfono de contacto (*)

    El usuario completa los campos:

           Nombre: "Desarrolladores del Sur SA"

           Domicilio: "Av. Boyacá 22345 - Esquel - Chubut"

           Nombre del administrador del cliente: "Juan Pérez"

           Mail del administrador del cliente: "jperez@dscom"

           Teléfono de contacto: "02945-112564"

    El sistema valida los datos y muestra un error en el
formato del mail

    El usuario abandona la operación

    El sistema no cambia la base de datos ni genera un
usuario y una clave para el administrador del cliente.

    El sistema guarda en el log "El usuario Administrador
abandonó el alta de una empresa".
```

La verdadera utilidad de los escenarios es que sirven para escribir pruebas concretas de aceptación, positivas o negativas, sea en el marco del testing tradicional o en métodos de desarrollo como TDD, ATDD y BDD[1].

En muchas ocasiones, los escenarios son fuente de casos de uso, y eso ocurre porque las personas se expresan más fácilmente con ejemplos concretos que con

casos generales. En estas situaciones, el analista puede sintetizar distintos escenarios surgidos de conversaciones con los usuarios y formalizar un caso de uso.

DIAGRAMAS DE CASOS DE USO

Cuestiones esenciales

El modelo de casos de uso suele servir, entre otras cosas, para delimitar el alcance del sistema, esbozar quiénes interactuarán con el sistema, a modo de actores, cuáles son las funcionalidades esperadas y capturar un primer glosario de términos del dominio. Y, por sobre todas las cosas, para validar los requisitos con el cliente.

UML, como notación de modelado visual, define un tipo de diagrama denominado **de casos de uso**. Estos diagramas no especifican el comportamiento de los casos de uso, sino solamente relaciones entre distintos casos de uso, y de casos de uso con actores. Por eso en nuestra clasificación los hemos incluido entre los diagramas estructurales y no de comportamiento, aun cuando no ignoremos que un caso de uso, especificado con todos sus detalles, es un modelo de comportamiento.

En la figura 4.1, se muestra un pequeño diagrama de un caso de uso y su actor en *FollowScrum*:

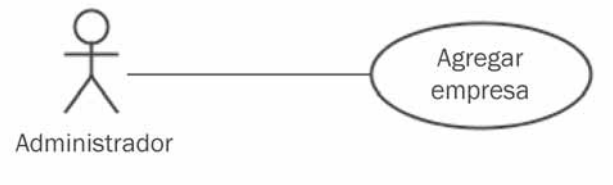

Figura 4.1 Un diagrama de caso de uso simple.

Como vemos, el actor se representa como una persona en forma esquemática, el caso de uso con una elipse con su nombre adentro, y la relación entre ambos con una línea, que en UML se llama asociación.

A veces se incluye una flecha en el extremo de la relación que corresponde al caso de uso, para mostrar que la visibilidad va del actor al caso de uso (es el actor quien se vale del caso de uso, y no al revés). Así se muestra en la figura 4.2.

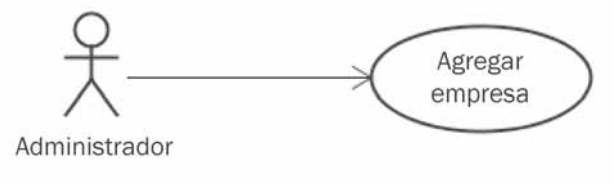

Figura 4.2 Navegabilidad del actor al caso de uso.

Como ya dijimos, un actor puede representar una persona o un sistema externo. Si bien es válido representar los sistemas externos con el esquema de persona, hay muchos profesionales que estiman que es mejor diferenciarlos, y usan un rectángulo (una clase de UML) con el estereotipo `<<actor>>` para representar sistemas externos en relación con los casos de uso. Eso se muestra en la figura 4.3.

Figura 4.3 Actor no humano.

También es posible utilizar alguna figura representativa del actor, además del esquema de persona y la clase estereotipada como actor. Sin embargo, esto no es muy usual y puede llevar a confusiones sobre lo que es un actor y lo que no lo es.

Un **diagrama de casos de uso**, no obstante, suele ser más complejo, puesto que muestra todas las interacciones entre casos de uso de un sistema o subsistema. En la figura 4.4, vemos el diagrama de casos de uso del subsistema de administración de *FollowScrum*.

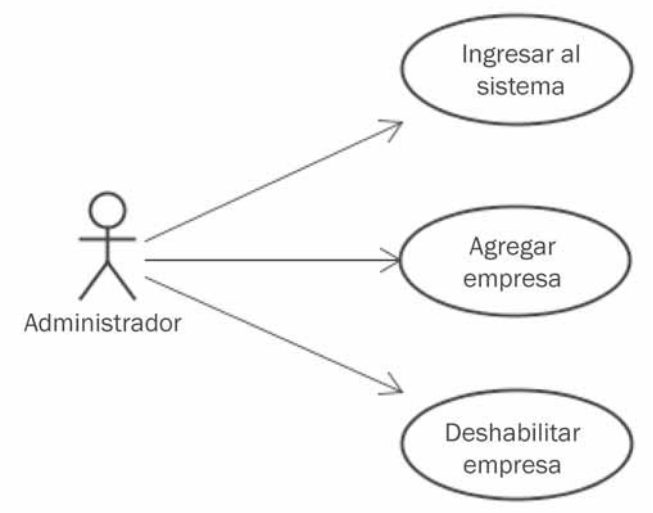

Figura 4.4 Diagrama de casos de uso de un subsistema.

Diagramas de casos de uso y contexto

Se afirma habitualmente que los diagramas de casos de uso son útiles para definir el contexto de un sistema antes de construirlo. Y tal vez sea una de sus grandes fuerzas. Sin embargo, si el sistema se construye en forma realmente incremental, y no se pretende conocer todo el alcance al comenzar, esto deja de ser cierto. Y aun cuando trabajemos con un proceso más estático, que parta de requisitos enumerados (aunque no necesariamente definidos en detalle) al inicio del proyecto, si hay cambios de

alcance habrá cambios en los casos de uso. Como esto ocurre casi siempre, consideremos al diagrama de casos de uso como un artefacto que deberá actualizarse en forma repetida, si es que queremos que sirva para ver el contexto.

Cuando al diagrama de casos de uso se lo va a utilizar para describir el contexto de un sistema o subsistema, se suele rodear los casos de uso por un rectángulo que denote la frontera del sistema o subsistema. Esto se muestra en la figura 4.5.

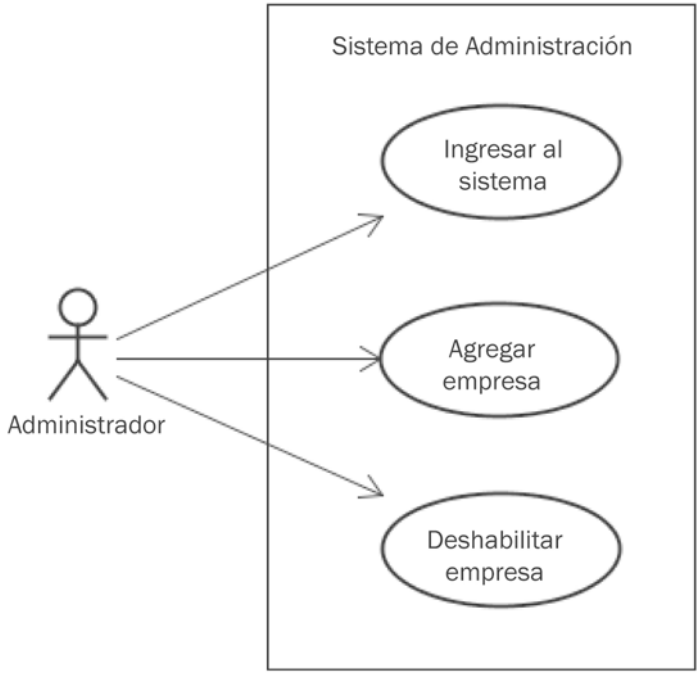

Figura 4.5 Contexto de un subsistema.

Los actores son una buena fuente para encontrar los límites del sistema, ya que, por su propia definición, son externos al mismo. Esto es especialmente valioso cuando los actores son sistemas externos, puesto que no siempre es sencillo encontrar qué funcionalidades deben quedar dentro del sistema cuyos requisitos estamos modelando y cuáles en los sistemas externos.

Utilidad de los diagramas de casos de uso

De los modelos de UML, el diagrama de casos de uso es, tal vez, el más decepcionante. No es que no sirva para nada, sino que se generan expectativas superiores a sus posibilidades.

Y lo más curioso es que hay muchos profesionales que lo mencionan como una de las grandes cualidades de UML. Quizá la confusión venga de la real utilidad de los casos de uso textuales, que no son parte de UML, y no de los propios diagramas. Pero no hay nada en UML que defina cómo se debe describir el comportamiento de un caso de uso.

Su principal inconveniente es su escaso nivel de detalle, que ni siquiera los hace útiles para modelar requisitos a muy alto nivel. Tal vez su única utilidad sea la de modelar el contexto de un sistema o subsistema.

Los defensores de la utilización de los diagramas de casos de uso ponen el acento, precisamente, en el hecho de que estos diagramas sirven para modelar lo que se hace y quién lo hace, sin entrar en detalles de cómo se hace. También en el hecho de que una rápida mirada al diagrama establece el comportamiento esperado del sistema, factorizado en las interacciones que éste tiene con el exterior y relacionándolas con sus actores. Notemos que eso mismo se puede lograr con *user stories.*

En definitiva, el uso o no de estos diagramas, como tantas otras cosas, es una cuestión de preferencias: habrá quienes prefieran trabajar con casos de uso en forma gráfica y quienes se inclinen por *user stories* textuales. De todas maneras, no hay que exagerar su utilidad.

MODELADO DEL COMPORTAMIENTO EN REQUISITOS

Diagrama de actividades

Hay ocasiones en las que hay que definir de una manera clara un flujo de proceso o de un requisito. Para ello, son de gran utilidad los **diagramas de actividades.**

En el contexto del modelado de requisitos, un diagrama de actividades puede ayudar a comprender el flujo de actividades de un caso de uso.

En la figura 4.6, se muestra un diagrama de actividades que modela el flujo del caso de uso “Agregar empresa”, ya analizado.

Notemos los elementos típicos de un **diagrama de actividades:**

- Los rectángulos de bordes redondeados son actividades o acciones en el flujo. Dentro de los mismos se coloca una descripción breve de la actividad. A veces, se los asocia a estados de algún objeto, pero desde UML 2 esto ya no es necesario.
- Las flechas indican el sentido del flujo.
- El comienzo y fin del flujo se indican con un círculo negro y un círculo blanco con uno negro concéntrico, respectivamente.
- Las bifurcaciones condicionales se especifican con un rombo, colocando la condición de las ramas –denominada **condición de guarda**– entre corchetes. Notemos que no siempre colocamos la condición de guarda, sino solamente cuando agrega claridad al diagrama (es similar a colocar la expresión `[else]`, como recomiendan los creadores de UML).
- Las acciones concurrentes se dibujan naturalmente, con dos barras gruesas (denominadas **barras de sincronización**), una para indicar el comienzo de la concurrencia y otra para el fin (se los suele denominar conectores *fork* y *join*).

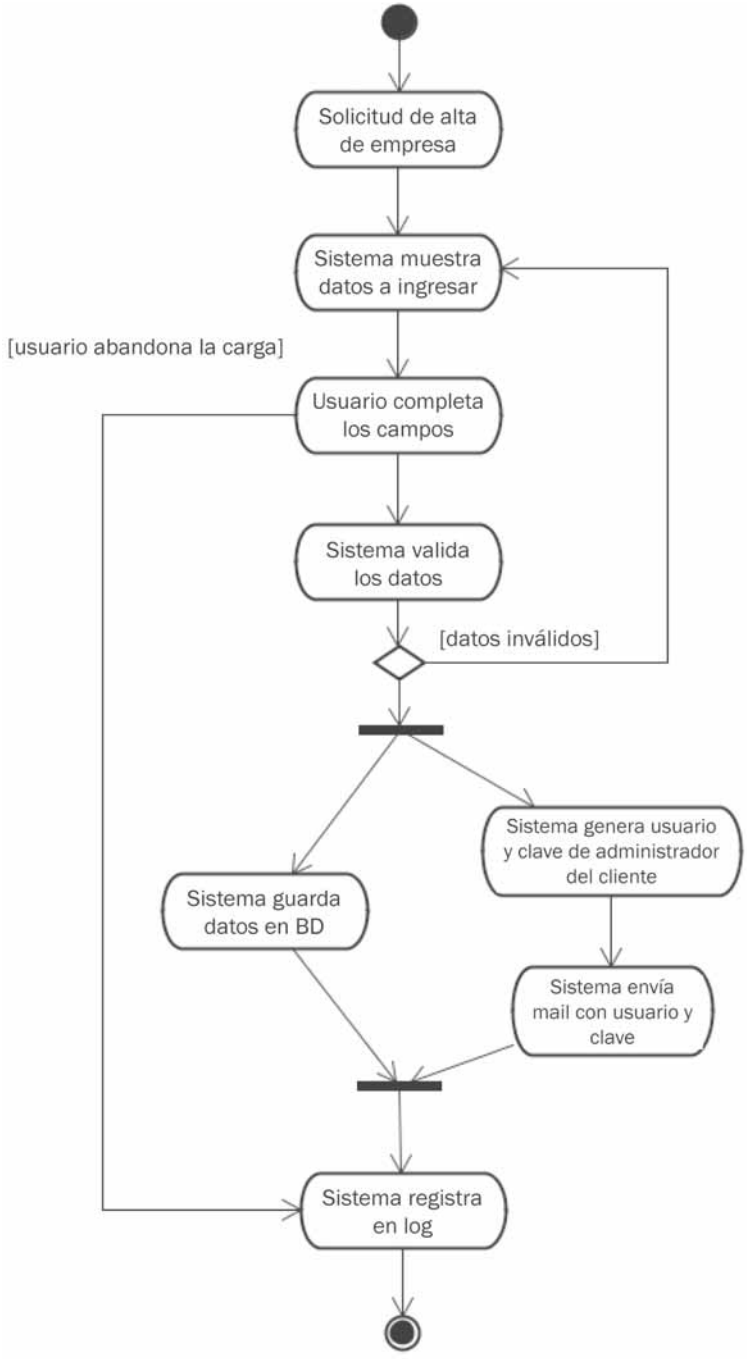

Figura 4.6 Diagrama de actividades de un caso de uso.

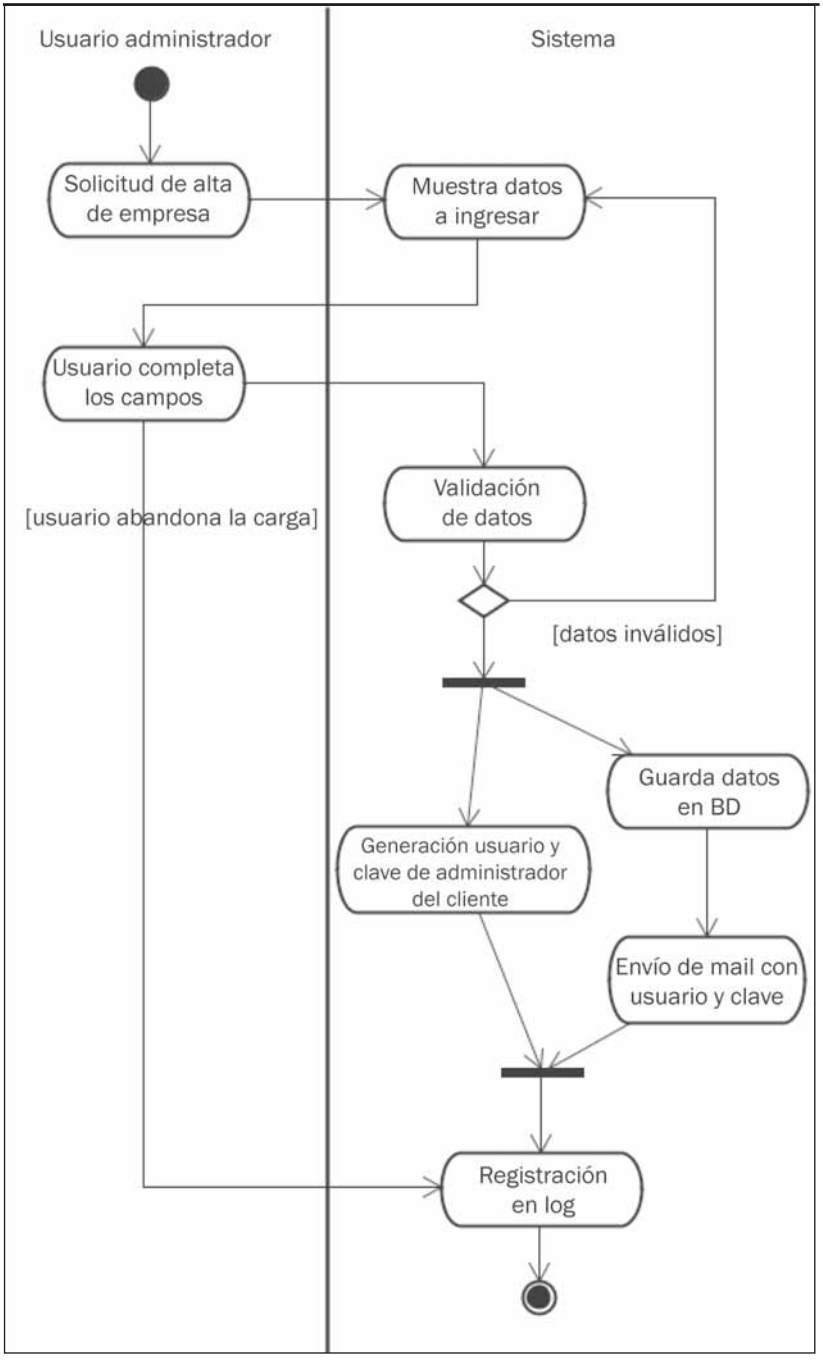

Figura 4.7 Diagrama de actividades con calles.

UML distingue las acciones, que son instantáneas, de las actividades, que pueden requerir cierto tiempo para ejecutarse. Sin embargo, las personas que usan los diagramas de actividades no siempre hacen estas distinciones, que son más teóricas que útiles.

Respecto de la concurrencia, notemos que, cuando estamos especificando un caso de uso, la concurrencia que modelamos es sólo teórica: implica que las tareas puestas en paralelo se pueden hacer a la vez, no que tienen que hacerse necesariamente al mismo tiempo en el sistema que construyamos (esto sería una decisión de diseño).

Un diagrama de actividades es una herramienta interesante para especificar requisitos, sumamente simple a la vista de un usuario inexperto y, en general, suficientemente comunicativa.

Calles y particiones

En ocasiones, se le agregan **calles**[2] a los diagramas de actividades para especificar qué o quién realiza las acciones. En la figura 4.7, se muestra un diagrama de actividades con calles.

Al usar calles, como vemos en la figura, pudimos eliminar el nombre del sujeto en cada acción, y también quedó más claro el flujo de las actividades.

UML 2 usa más el nombre de **particiones** que el de calles, además de permitir particiones tanto horizontales como verticales y admitir particiones internas a otras.

Objetos, señales y eventos

A veces, puede indicarse el flujo de objetos físicos en un diagrama de actividades, para modelar objetos que estén involucrados en el escenario en cuestión. Por ejemplo, en la figura 4.8 mostramos una parte del mismo diagrama anterior, en el que indicamos la creación automática del mail que se enviará al cliente.

Se puede mostrar en el diagrama el paso del objeto `m` por diversos estados. En este caso, sólo le pusimos el rótulo `[generado]`.

Tal vez el anterior no sea un buen ejemplo. De hecho, es difícil encontrar un buen uso del modelado de objetos físicos en diagramas de actividades que especifiquen casos de uso. Quizá en un diagrama de actividades que muestre un flujo de documentos (que es un uso posible, aunque no lo estudiamos en este libro), la idea de modelar objetos-documentos y su ciclo de vida sea una buena idea.

Si el objeto en cuestión fuera un repositorio de datos persistentes (una base de datos, por ejemplo), se puede indicar con el estereotipo `<<datastore>>`.

Otro aspecto que se suele representar en algunos diagramas de actividades son las **señales**. Las señales se pueden usar para indicar un evento temporal o de otro tipo, tanto como precondición de alguna actividad o generado por una actividad.

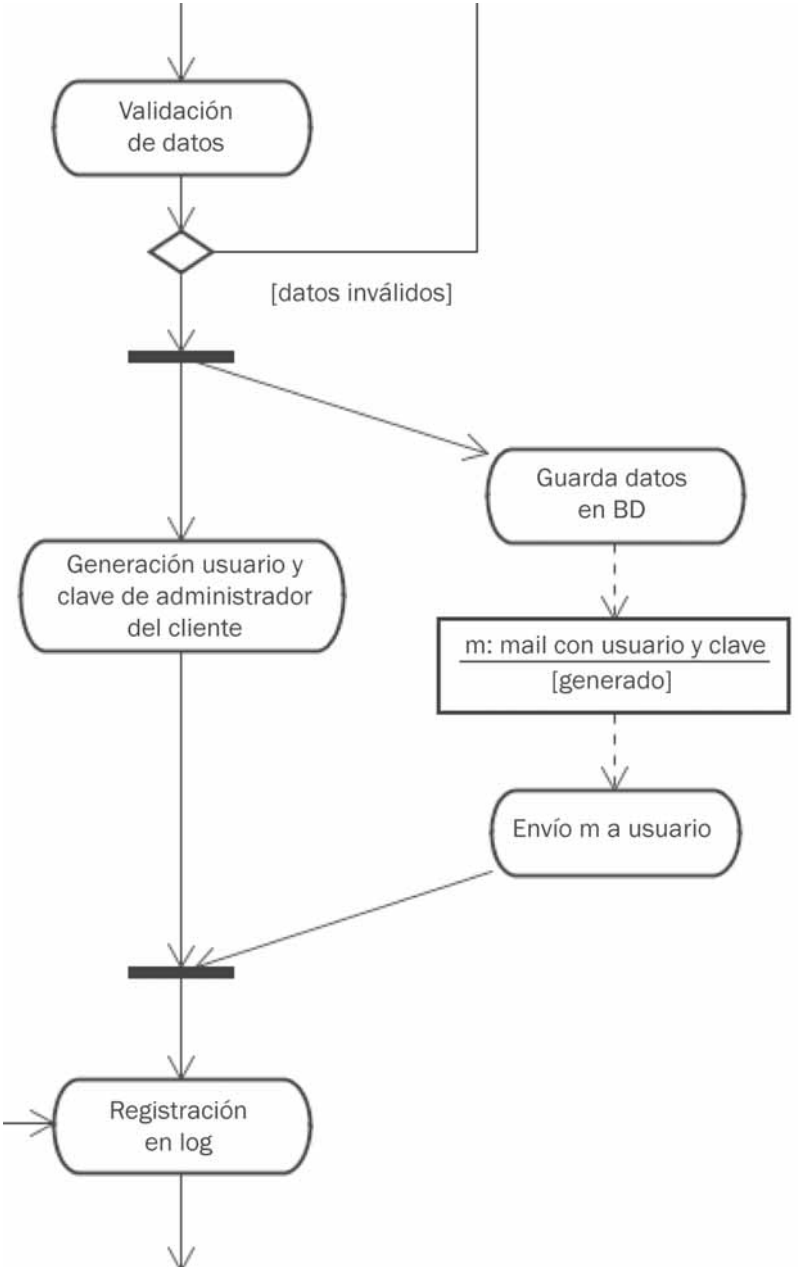

Figura 4.8 Objeto en diagrama de actividades.

Por ejemplo, en la aplicación *FollowScrum*, podemos definir que un proyecto que estuvo en etapa de pre-venta por más de dos meses, debe ser dado de baja. Otra

condición de baja antes del comienzo de la planificación podría ser el aviso del cliente de que el proyecto fue asignado a otro proveedor. En la figura 4.9, se muestran estas dos situaciones.

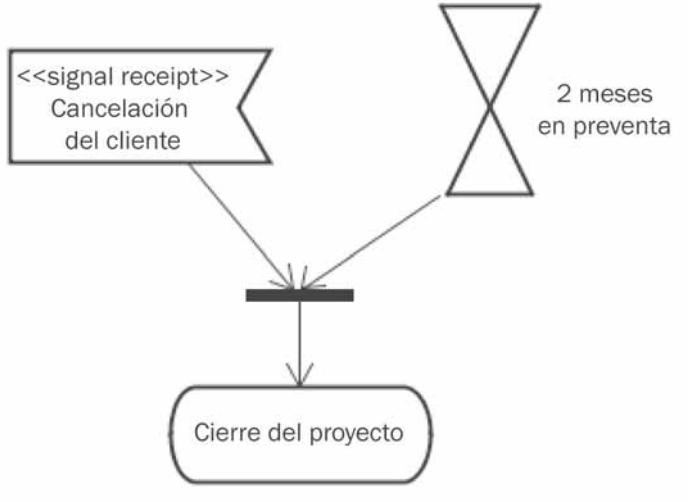

Figura 4.9 Señal temporal y evento proveniente del exterior.

Como vemos, la señal temporal se representa con un reloj de arena, mientras que un evento proveniente del exterior se indica con un rectángulo con el lado entrante de ángulo cóncavo.

Si un evento es generado por alguna actividad, esto se muestra con un rectángulo con el lado saliente de ángulo convexo. Esto es lo que hicimos en la figura 4.10, al representar de otra manera el envío de mail de la figura 4.8.

Los diagramas de actividades son útiles en el modelado de requisitos cuando nuestro interlocutor se siente cómodo con una notación gráfica para ver el flujo de acciones o procesos. Destaca en ellos la simplicidad de modelado de concurrencia conceptual.

El modelado de los pasos de un caso de uso es uno de los pocos usos prácticos de los diagramas de actividades. Esto no es una crítica, sin embargo, como herramienta resulta excelente por su potencia expresiva y su simplicidad.

No obstante, recordemos que los casos de uso deben ser bien comprendidos por todos los interesados, incluyendo especialistas de negocio y clientes. Por lo tanto, hay que tener presente que una condición fundamental es su facilidad de comprensión. De hecho, hay algunas cuestiones adicionales que se pueden modelar con diagramas de actividades, que veremos en el ítem siguiente, pero hay que hacerlo solamente en la medida en que sirvan como herramienta de comunicación.

A los diagramas de actividades también se los suele usar para modelar procesos de negocio, independientemente del software, aunque esto excede lo que pretende este libro.

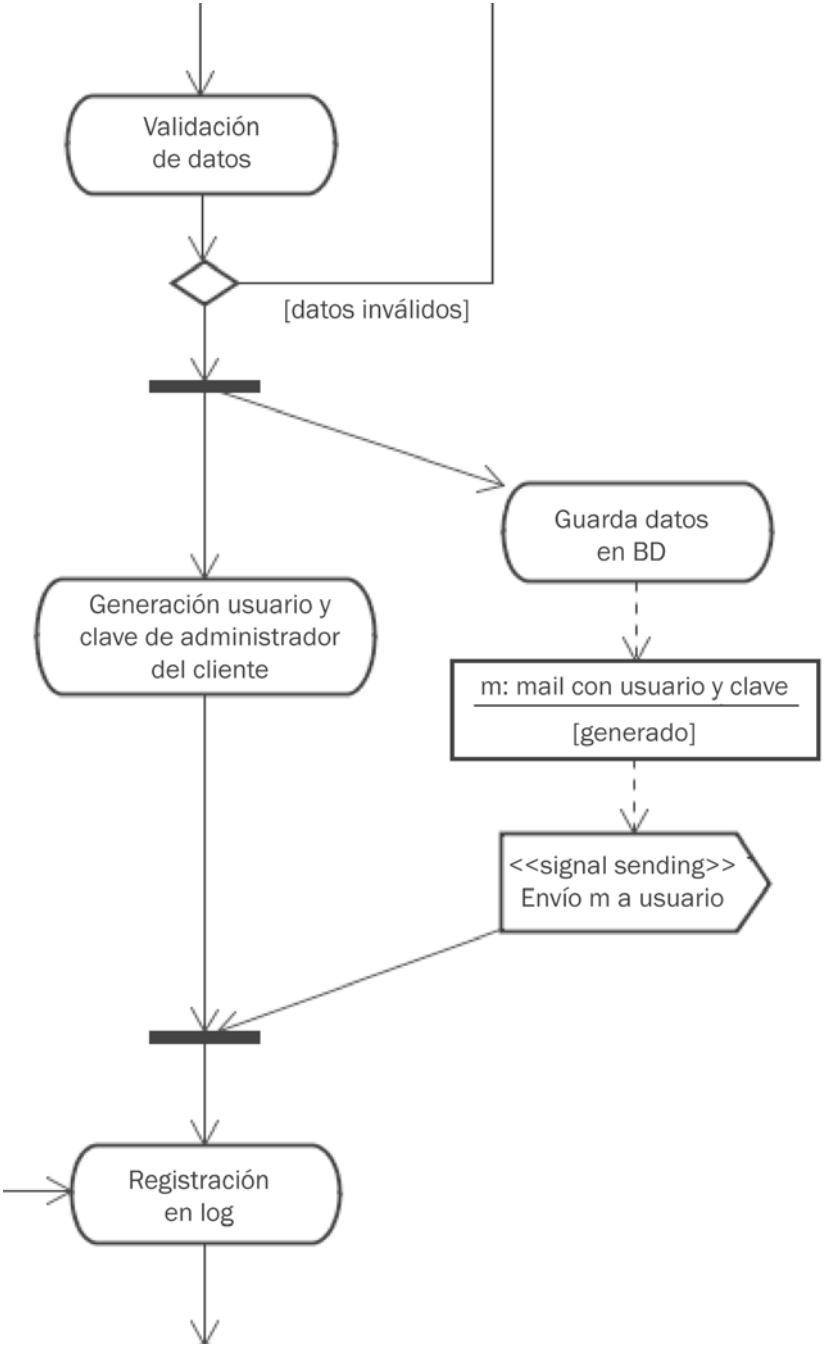

Figura 4.10 Evento emitido.

Aspectos avanzados de los diagramas de actividades

El diagrama de actividades es uno de los que más cambios ha sufrido a lo largo de las versiones de UML. Casi todos los cambios han sido mejoras reales, aunque el cambio permanente ha hecho que muy pocos profesionales utilicen todos los aspectos de este diagrama.

Algunas cuestiones avanzadas las veremos en este ítem. Aunque en aras de lograr una mejor comprensión, tal vez convendría usar estas características con medida.

Hay ocasiones en las que puede ser conveniente abrir una actividad en varias sub-actividades, o reunir algunas actividades en una actividad compuesta. En ese caso, se puede modelar como se muestra la **actividad compuesta** `Cierre del alta` de la figura 4.11. De esa manera, la figura queda más sencilla.

Notemos que la figura 4.11 ha usado un esquema más cerrado para el diagrama de actividades, que también es válido.

La actividad compuesta se puede mostrar en detalle como en la figura 4.12.

Suele ser útil representar actividades que se realizan varias veces sobre una lista de objetos. La mejor forma de representar esto es mediante una **región de expansión**.

Por ejemplo, *FollowScrum* podría tener una funcionalidad que emitiese reportes de varios proyectos activos. Esto es lo que se muestra en la figura 4.13 para el caso de uso `Emisión de reportes`.

La lista de entrada, en la figura 4.13, es la misma que la lista de salida. A veces no es así, como cuando la lista de salida se genera dentro de la región de expansión.

El estereotipo `<<iterative>>` de la figura indica que los elementos de la lista se recorren uno a continuación del otro, hasta terminar. La otra posibilidad es colocar la leyenda `<<concurrent>>`, que indicaría que los distintos elementos se podrían tratar en forma paralela.

También hemos indicado la posibilidad de que el usuario que pidió los reportes pueda salir luego de terminados los de un proyecto en particular. Si bien esto puede parecer un poco extraño en este caso, nos sirvió para mostrar lo que se conoce como el **final del flujo**, que es la X dentro del círculo que usamos en el diagrama.

Se pueden colocar conectores entre actividades, cuando las flechas del diagrama se deban ver interrumpidas por algún motivo. En ese caso, se hace como en la figura 4.14.

El uso de conectores no es muy recomendable, porque hace perder la claridad conceptual que brindan las flechas. De hecho, conviene buscar alternativas, como evitar cruces de líneas y organizar mejor el diagrama. Tal vez su única justificación sea en el caso de un diagrama en varias páginas, aunque en este caso tendríamos que preguntarnos si no podemos achicarlo usando actividades compuestas y sub-actividades.

A veces, necesitamos poner precondiciones o postcondiciones locales de una acción en particular. Esto se hace con una nota especial con el estereotipo `<<local Precondition>>` o `<<localPostcondition>>`.

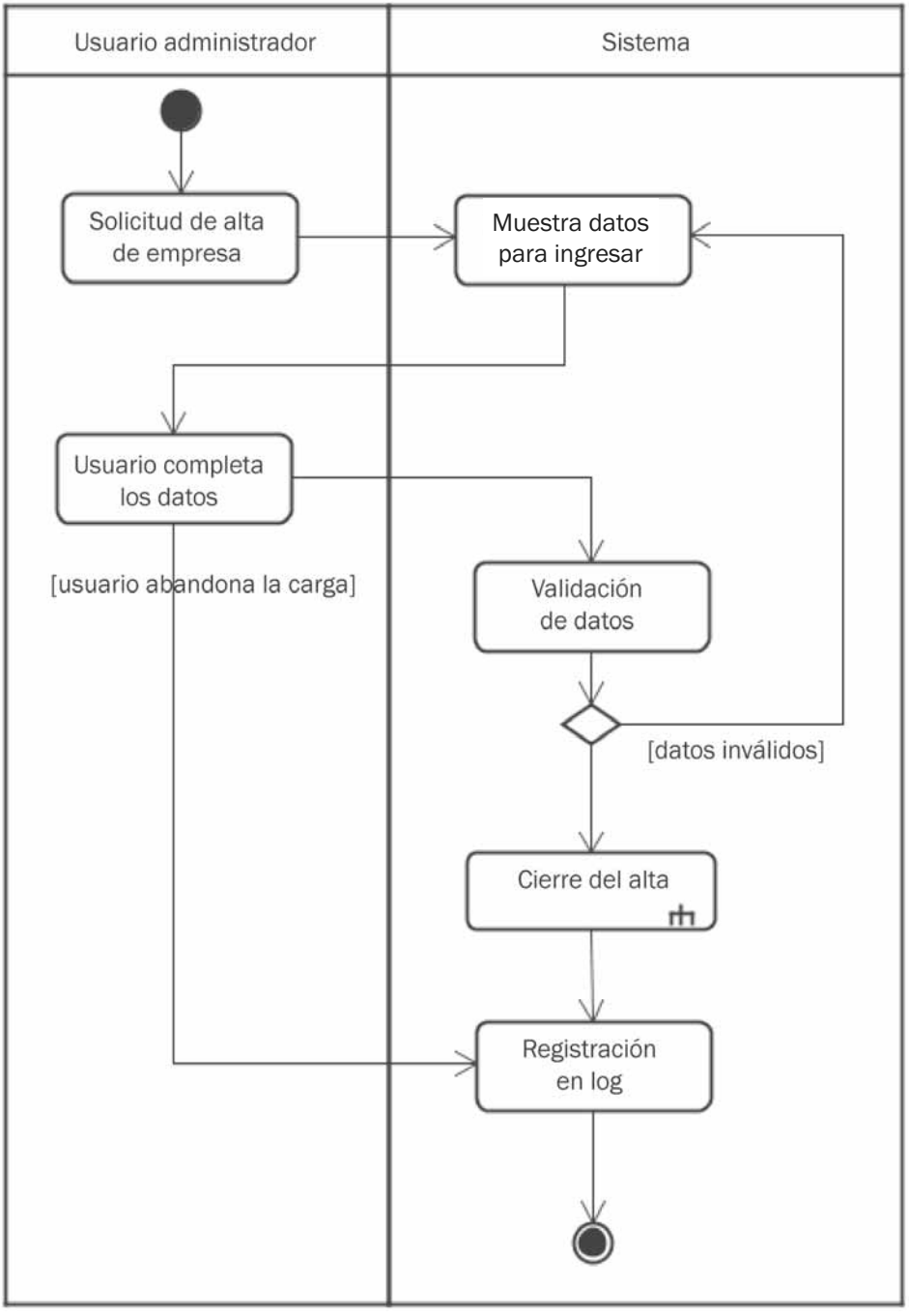

Figura 4.11 Actividad compuesta de cierre del alta.

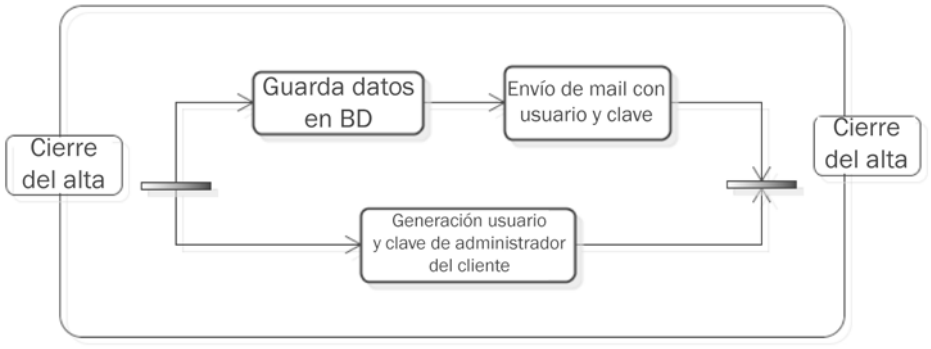

Figura 4.12 Sub-actividades.

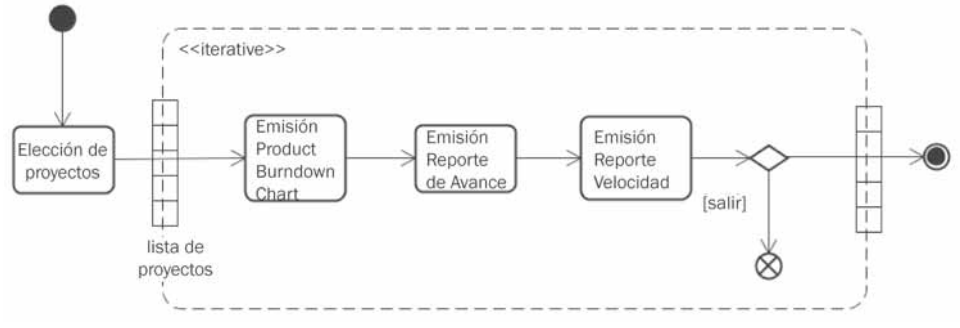

Figura 4.13 Región de expansión.

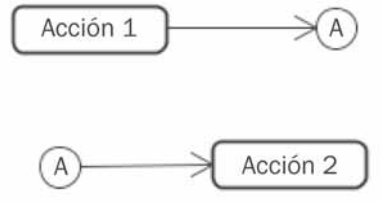

Figura 4.14 Actividades y conectores.

Diagrama de secuencia del sistema

Hay quienes no usan diagramas de actividades para modelar el comportamiento de los casos de uso, y prefieren, en cambio, los diagramas de secuencia de sistema.

Si bien explicaremos los diagramas de secuencia en el próximo capítulo, los **diagramas de secuencia de sistema**, como muestra el de la figura 4.15, son diagramas en los cuales se colocan dos entidades, el `Actor` y el `Sistema`, y se modelan

las actividades con el paso de mensajes entre ambos. En este caso, hemos representado el mismo caso de uso de la figura 4.11.

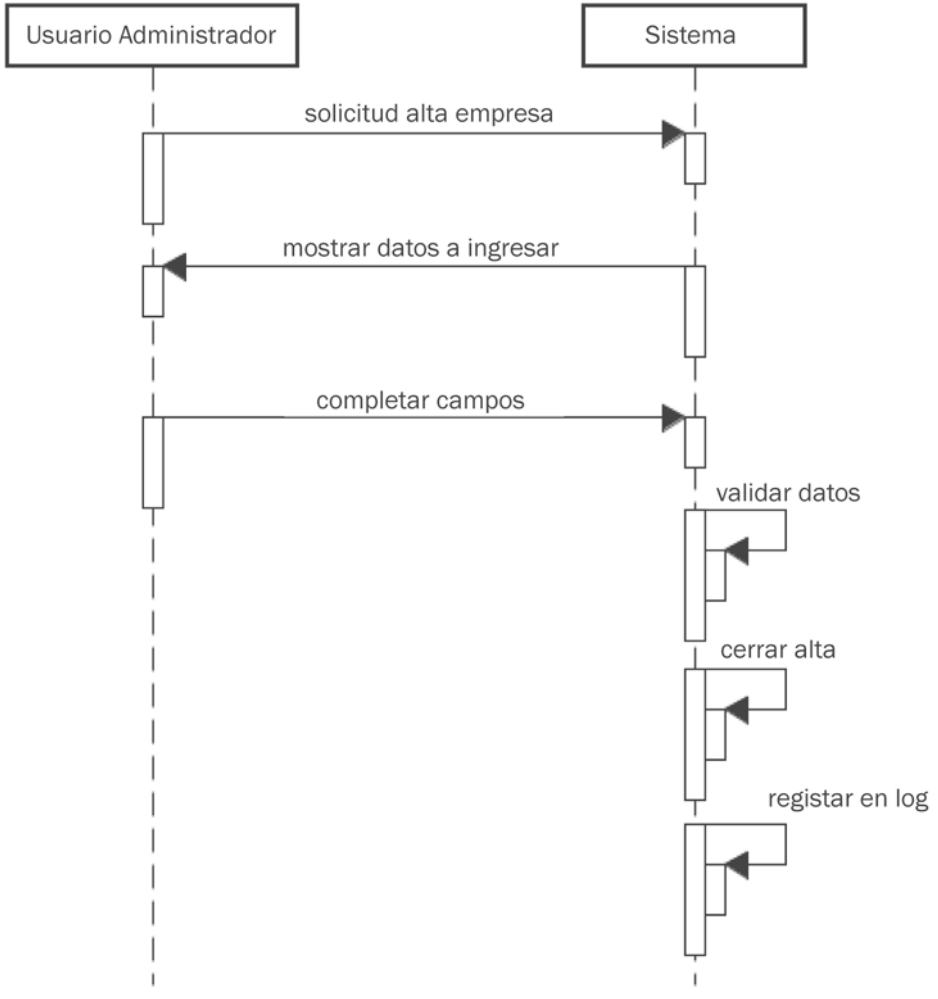

Figura 4.15 Diagrama de secuencia de sistema.

Si hubiera más actores en un mismo caso de uso, figurarían también a la izquierda, como otras entidades, a las que se les podría agregar el estereotipo `<<actor>>`. Lo que sí debe limitarse es a un único `Sistema`, ya que las entidades internas al mismo no nos interesan hasta el análisis. De modo que el diagrama de secuencia del sistema también muestra claramente las funciones de la aplicación, sin indicar nada de su implementación.

Los diagramas de secuencia de sistema suelen ser considerados más orientados a objetos por sus defensores, debido a que se basan en el paso de mensajes. De

hecho, lo cierto es que puede ser más sencillo derivar el comportamiento del sistema de estos diagramas que de los diagramas de actividades.

Lo único que hay que objetar es que en el diagrama de secuencia de sistema no es tan sencillo mostrar bifurcaciones condicionales. Si bien se puede hacer, complica el modelo, y éste pierde simplicidad. Por eso, en la figura 4.15, hemos evitado modelar la situación de falla en la validación de datos.

Si bien veo que hay mucha razonabilidad en los argumentos a favor del diagrama de secuencia de sistema, a lo que tal vez convendría agregarle su menor complejidad, lo cierto es que no es un camino muy habitual entre los profesionales, quizá porque quienes están acostumbrados a trabajar en requisitos están más habituados a analizar procesos de negocio y flujos de procesos y documentos. Pero esto no es descalificatorio para con los diagramas de secuencia de sistema, que tal vez deberían considerarse más frecuentemente una buena opción.

DIAGRAMAS DE CLASES PARA MODELADO CONCEPTUAL DE DOMINIO

Mecanismos de abstracción

Es importante, antes de abordar el modelado conceptual, entender los mecanismos de abstracción.

Ante todo, qué queremos decir con **abstracción**. Denominamos así "al proceso de enfatizar algunas cuestiones para comprenderlas mejor, dejando otras de lado o manteniéndolas con un nivel de detalle menor". Ésto es algo que hacemos continuamente en nuestros razonamientos humanos para analizar y representar realidades complejas, pero que se debe hacer con cuidado para no simplificar excesivamente.

Los mecanismos de abstracción más característicos son:

- Clasificación.
- Asociación.
- Agregación o agrupación.
- Composición.
- Generalización.

La **clasificación** es un mecanismo que relaciona individuos con especies. Así, decimos que Lassie es un perro, que Dumbo es un elefante (un poco especial, por cierto), que quien escribe estas líneas es un humano, que el aparato en el que estoy escribiendo es una computadora, etc.

Este mecanismo es extremadamente útil para deducir aspectos generales de un conjunto de individuos, y tiene especial uso en el desarrollo de software. Por ejemplo, a la aplicación *FollowScrum* podrán acceder usuarios, tales como Pedro, Agustina, Javier y Mercedes. Sin embargo, cuando abstraemos no nos suele interesar –al menos no siempre– quiénes son exactamente, sino el hecho de que son individuos

particulares de la categoría de los usuarios. O, como diremos más a menudo, **instancias** de la **clase** de los usuarios.

La razón por la que nos interesamos más en las clases que en sus instancias tiene que ver con que nos suelen interesar más las propiedades comunes y las acciones esperables de las instancias que, salvo detalles accidentales, son las mismas para todas las instancias de una misma clase. Por ejemplo, todos los usuarios tendrán un identificador, una clave de acceso, trabajarán en una empresa, etc.

Como la clase es también el conjunto de las instancias, la clasificación nos sirve para obtener entidades o individuos concretos como casos particulares de las clases, mediante el mecanismo de **instanciación**, que por definición es el inverso de la clasificación.

Cada individuo puede estar relacionado de una manera u otra con otros individuos, de su misma clase o de otra. Esto es lo que denominamos **asociación**. Por ejemplo, si decimos que Juan trabaja para la empresa Aikén Software, mientras que Catalina lo hace para Software del Sur, estamos estableciendo una relación entre instancias de personas (o usuarios, si seguimos hablando de usuarios de *FollowScrum*) e instancias de empresas (o clientes de *FollowScrum*).

Si analizamos un poco podríamos concluir que todos los usuarios trabajan en alguna empresa, con lo cual la relación entre usuario y empresa se da a nivel de clases (como conjunto de objetos de un determinado tipo) y no sólo de objetos vistos individualmente. Por eso decimos que la asociación es una relación entre las clases que implica una relación entre sus instancias.

La relación de asociación –como decíamos– puede darse entre individuos de una misma especie o clase. Por ejemplo, si decimos que Ángeles es la jefa de Alejandro, estamos estableciendo una asociación entre dos usuarios, que muy probablemente se pueda establecer para todas las instancias de la clase, a lo sumo con unas pocas excepciones.

Una asociación es inherentemente bidireccional. Así como dijimos que Juan trabajaba para Aikén Software, podríamos haber dicho que Aikén Software emplea a Juan. De la misma manera, si Ángeles es jefa de Alejandro, este último le reporta a aquélla.

Hay ocasiones en las que las asociaciones tienen una **cardinalidad** esperada, fija o definida mediante un rango. Por ejemplo, una empresa cliente debe tener al menos un usuario. Y cada usuario debe estar asociado a una y solo una empresa.

La asociación sirve como mecanismo de abstracción porque nos permite separar conceptos, manteniendo sus relaciones vinculares.

Hay un tipo de asociación especial que denominamos **agregación**. Se da cuando uno de los extremos de la asociación puede considerarse parte del otro extremo. Por ejemplo, cuando dijimos que Software de Sur emplea a Catalina, suponemos que también lo hace con muchas otras personas.

Si bien la agregación es bidireccional como en cualquier asociación, es claramente asimétrica: el todo y las partes no son intercambiables.

La agregación es una propiedad transitiva. Si C es parte de B y B es parte de A, entonces C es parte de A.

Como toda asociación, puede haber agregación entre individuos de una misma clase. Por ejemplo, una organización compuesta podrá contener otras organizaciones como partes.

Ahora bien, hay ocasiones en las que en una agregación las partes no tienen sentido fuera del agregado, o no pueden formar parte de más de un agregado. Esto se llama **composición**.

En el caso de los usuarios y de las empresas, todos los usuarios son parte de una organización para *FollowScrum*, y dejan de tener sentido como usuarios del sistema si dejan la organización que las emplea.

Notemos que la composición debe ser analizada en un contexto, y por eso mismo es un poderoso mecanismo de abstracción. No es que Catalina deje de existir si deja Software del Sur: lo que ocurre es que, desde el punto de vista del sistema analizado, no tiene sentido seguir considerando un usuario si deja una de las empresas clientes del sistema. Si justo se diera el caso de que el usuario en cuestión pasara a trabajar en otra empresa cliente, bien podríamos considerarlo otro usuario.

La **generalización** es "la operación por la cual establecemos que una o más clases tienen elementos en común que deseamos agrupar en una clase más genérica". La operación inversa, que denominamos **especialización**, permite encontrar clases más específicas mediante la búsqueda de diferencias entre individuos de las clases genéricas.

Por ejemplo, en *FollowScrum* tenemos usuarios administradores y usuarios de clientes. Ambos son usuarios, pero a la vez que hay elementos en común entre ellos, existen otros elementos que los diferencian. Estos elementos pueden ser tanto características como comportamientos. Por ejemplo, un usuario administrador no trabaja para una empresa cliente, mientras que los otros sí: ésta es una diferencia por sus características. Pero, además, un usuario administrador puede dar de alta nuevos clientes, cosa que no puede hacer un usuario de un cliente: ésta es una diferencia por su comportamiento.

Si decíamos que una clase es el conjunto de sus instancias, la clase genérica es un conjunto de conjuntos: contiene las instancias de todas las clases que la especializan.

La generalización es unidireccional, transitiva, ACÍCLICA y asimétrica. Unidireccional, porque si una clase es un caso particular de otra, ésta no lo puede ser de aquélla. Transitiva, porque si A es una generalización de B y B una generalización de C, entonces A es una generalización de C. Acíclica, porque una sucesión de generalizaciones no puede llevar a que una clase sea una generalización ni una especialización de sí misma. Asimétrica, porque si A es una generalización de B, B no puede ser una generalización de A.

Se trata de un mecanismo de abstracción ideal para separar detalles y concentrarse en aspectos comunes entre conceptos.

Cada uno de estos mecanismos de abstracción nos permite analizar y modelar la realidad de maneras distintas, incluso ortogonales.

Modelado de dominio

El **modelo de dominio**, también denominado modelo de negocio o modelo conceptual, describe el negocio u organización para la cual se desarrolla el producto de software. Contiene una serie de conceptos que requieren ser entendidos para avanzar en el desarrollo.

Como todo modelo, tiene elementos estructurales y otros de comportamiento. Para los aspectos de comportamiento, puede usarse el diagrama de actividades, que hemos visto anteriormente. En cuanto a los elementos estructurales, es de gran ayuda el diagrama de clases.

Una cuestión central del modelo estructural de dominio es el manejo de los conceptos (vocabulario) y de las relaciones entre los mismos.

Modelado de dominio con clases de UML

El **diagrama de clases** de UML nos puede servir para analizar los conceptos de un dominio y sus relaciones, con una notación gráfica. En ese sentido, se convierte en una especie de diccionario o glosario gráfico.

Sin embargo, el modelo de dominio con clases no nos va a servir para ver todas las reglas de negocio que se aplican en el dominio, los valores aceptables para las propiedades o un nivel de detalle muy grande.

Por ejemplo, un modelo de dominio simplificado de la aplicación *FollowScrum* podría ser el de la figura 4.16. Allí se ven los conceptos principales del negocio, y las relaciones de asociación, generalización y dependencia entre ellos.

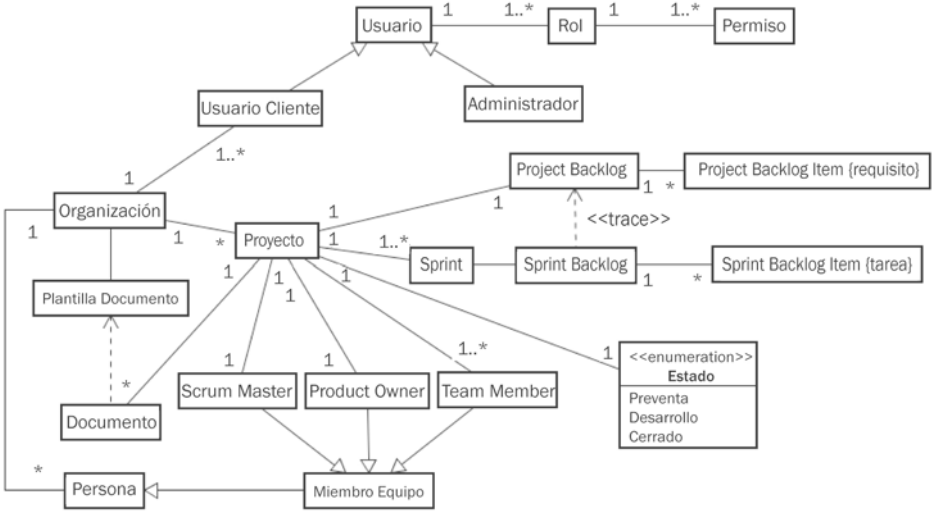

Figura 4.16 Modelo de dominio de *FollowScrum.*

El diagrama que acabamos de realizar es bastante rico, aunque sus elementos no son tantos. Veamos.

- Cada concepto del dominio del problema se ha representado como una clase de UML, mediante un rectángulo con su nombre dentro.
- Las clases pueden tener estereotipos, como hicimos con el concepto de `Estado` que, al ser declarado como una enumeración, implica que sólo puede tomar los valores que se indican más abajo.
- También podemos agregar comentarios en los nombres de las clases, como hicimos con `Product Backlog Item` y `Sprint Backlog Item`. De todas maneras, esto no es lo más ortodoxo, y tal vez sea mejor una nota, como veremos más adelante.
- Las relaciones entre clases son de tres tipos en UML: asociación, generalización-especialización y dependencia.
- La asociación, representada por una simple línea, indica que un concepto está relacionado con otro. Eso se muestra entre las clases `Organización` y `Proyecto` o `Proyecto` y `Sprint`, entre muchas otras.
- La asociación es una relación transitiva. Esto es: si el concepto de `Organización` está relacionado con `Proyecto`, y éste, a su vez, con el concepto `Sprint`, hay una asociación entre `Organización` y `Sprint`. Estas asociaciones implícitas no se representan en los diagramas de clases, por lo que hay que buscar activamente conceptos intermedios al construirlos.
- Las asociaciones tienen una cardinalidad indicada por números o rangos al extremo de cada asociación. La cardinalidad de la relación nos informa con cuántos elementos de un concepto se relaciona cada elemento del otro. Por ejemplo, en el diagrama anterior, hemos indicado que cada `Proyecto` tiene uno y sólo un `Scrum Master`, colocando un número 1 en cada extremo de la relación. También hemos indicado que cada `Sprint Backlog` está relacionado con varios `Sprint Backlog Item`, al colocar `1` en un extremo y `1..*` en el otro.
- La cardinalidad de una relación puede expresarse de tres maneras: con un número, con un rango de números mediante el formato `N..M`, o mediante el uso del asterisco, que implica varios. Por ejemplo, si una cardinalidad se expresa como `1..3`, implica un rango de 1 a 3; si se expresa como `2..*`, implica que como mínimo es de 2, sin un máximo establecido; si se expresa sólo con un asterisco, tiene el significado de `0..*`, que significa que puede haber o no elementos, e incluso varios de un concepto, asociados con el otro.
- Habitualmente, uno de los extremos de asociación tiene cardinalidad 1. Esto surge directamente de la definición de cardinalidad, que indica cuántas instancias de una clase se relacionan con cada instancia de la otra. Por eso se suele asumir 1 cuando la cardinalidad no se indica, aunque esto no es normativo.

- La relación de generalización-especialización se usa para indicar que un concepto es un caso particular de otro. Por ejemplo, en nuestro diagrama, estamos indicando que un `Usuario` puede ser un `Administrador` o un `Usuario Cliente`. Dicho en términos de UML; la clase `Usuario` está relacionada con dos clases más específicas: `Administrador` y `Usuario Cliente`. La relación inversa a la especialización es la de generalización: la clase `Usuario` generaliza a las clases `Administrador` y `Usuario Cliente`.
- La generalización se indica con una línea terminada en flecha triangular vacía hacia la clase más general.
- La dependencia es la más débil de las relaciones. Se utiliza para indicar algún tipo de relación más débil que una asociación o una generalización-especialización. Es lo que hicimos en nuestro ejemplo con los conceptos `Documento` y `Plantilla Documento`, así como entre `Product Backlog` y `Sprint Backlog`.
- La dependencia se representa con una línea punteada terminada en flecha, que indica el sentido de la dependencia, partiendo del dependiente.
- En las relaciones de dependencia se pueden usar estereotipos. Por ejemplo, en nuestro caso, usamos el estereotipo `<<trace>>` para indicar que el `Sprint Backlog` tiene elementos que se pueden obtener a partir del `Product Backlog`. Si bien hay muchos estereotipos estándares, hemos resuelto mantener la simplicidad, usando solamente aquellos que nos sirvan en cada caso. Por ejemplo, existe un estereotipo `<<use>>`, que indica que el elemento dependiente requiere del otro para algo de su implementación; sin embargo, esto es lo más general en cuanto a dependencias, y por eso no lo hemos usado nunca.

El modelo de dominio debería servir para comprender el contexto del problema, tal como lo ven los clientes y usuarios. Este diagrama que hemos realizado tiene exactamente esa finalidad: establecer los conceptos y sus relaciones.

Podríamos haberlo refinado más. Por ejemplo, no hay duda de que entre las clases `Proyecto` y los distintos miembros del equipo podríamos –y tal vez deberíamos– haber colocado una clase `Equipo`. También podríamos haber modelado el concepto de `Duración`, asociado a un `Sprint`, pero en este caso lo evitamos a conciencia. De hecho, no conviene representar como clases de primer nivel conceptos cuyo tipo sea simple, como veremos en el próximo capítulo.

Un límite que hay que tratar de no traspasar es el de indicar solamente las asociaciones que surjan de los requisitos. Un analista con visión de futuro tal vez se sienta inclinado a crear una asociación entre `Miembro Equipo` y `Usuario Cliente`, por ejemplo, por parecerle muy razonable. No obstante, si esta relación no surge de los requisitos, no debería modelarse, ya que –entre otras cosas– va a complejizar innecesariamente un diagrama que se debe poder leer rápidamente.

Más sobre asociaciones

Hay muchas cuestiones más que se pueden agregar a los diagramas de clases, y varias de ellas pueden ser usadas también para el modelado de dominio.

Por ejemplo, en el caso anterior hemos modelado asociaciones entre las clases, de modo genérico. Pero UML permite también definir dos tipos de relaciones interesantes desde el punto de vista del modelado conceptual: la agregación y la composición.

La agregación es una asociación que representa la relación todo-partes. Por ejemplo, los ítems de `Product Backlog` son cada una de las partes del `Product Backlog`. Por eso, decimos que la clase `Product Backlog` tiene una relación de agregación con la clase `Product Backlog Item`. Eso mismo pasa con varios de los conceptos representados en el diagrama anterior.

En un diagrama de clases, la relación de agregación se representa mediante un rombo en el extremo de la asociación que corresponde al agregado, como se ve en la figura 4.17.

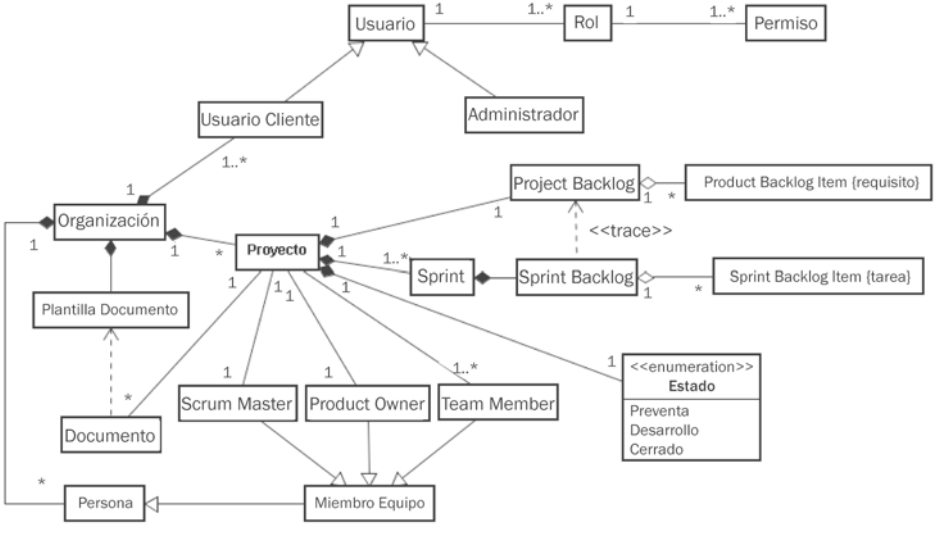

Figura 4.17 Agregación y composición.

La figura 4.17 muestra dos tipos de rombos, unos vacíos y otros rellenos. Esto no es un error ni un capricho. Ocurre que hay un tipo particular de agregación, que UML llama composición, que se representa con un rombo relleno. El significado de la composición es que, cuando hay este tipo de agregación, las partes no pueden ser independientes del todo. Esto es: si deja de existir el todo, dejan de existir las partes, pues no tienen existencia independiente.

Por ejemplo, hemos representado una relación de composición entre `Organización` y `Proyecto`, ya que no tiene sentido la existencia de un proyecto en ausencia

de una Organización que lo lleve adelante. Lo mismo ocurre con la relación entre Organización y Persona o entre Organización y Usuario Cliente.

Notemos que todo el modelado que estamos haciendo se refiere al punto de vista del sistema que estamos estudiando. No es que la persona física deje de existir, si deja de existir la organización en la que trabaja. Pero desde el punto de vista del sistema bajo estudio, el concepto de Persona está asociado al concepto de Organización en cuanto la persona es empleada de la organización.

Observemos –como corolario– que la composición, al atar el objeto contenido a su contenedor, impide que un objeto esté contenido en más de un contenedor a la vez. De todas maneras, ésta es una propiedad entre objetos, y no necesariamente entre clases.

Ahora bien, hay un elemento de los diagramas de clases que podría ayudarnos a mostrar en qué consiste la relación entre dos conceptos, y es el rol de la relación. Éste se representa poniendo el nombre del rol en el extremo que corresponde. Por ejemplo, la figura 4.18 muestra que el rol de la Persona en relación con la Organización es el rol de empleado, mientras que el rol de la Organización en relación con la Persona es el de empleador.

Figura 4.18 Roles de una asociación.

A veces hay varias asociaciones entre dos clases. La figura 4.19 nos muestra uno de esos casos. Notemos que los nombres de roles se tornan fundamentales para poder leer el diagrama.

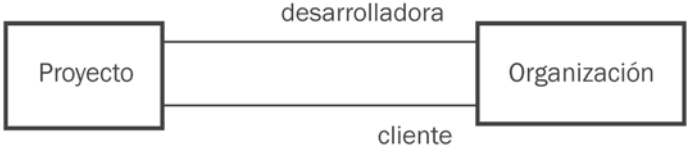

Figura 4.19 Más de una asociación entre dos clases.

De todas maneras, hay que tener cierto cuidado en estos casos. Tal vez la organización cliente no sea lo mismo, en cuanto concepto del dominio del problema, que la organización proveedora. En ese caso, quizá deberíamos tener dos clases separadas.

Existen ocasiones en las que no es necesario colocar en el diagrama el rol de ambos extremos de una asociación. Es lo que ocurrió en la figura 4.18, en la que los roles "empleado" y "empleador" son opuestos y, por lo tanto, redundantes. Quizá bastaría con indicar un nombre para la propia asociación. La figura 4.20 muestra la misma asociación a la que se le puso un nombre: Emplea a.

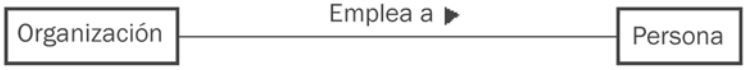

Figura 4.20 Asociación con nombre.

La flecha de una asociación con nombre no implica nada de la navegabilidad ni la visibilidad entre dos elementos, temas que abordaremos en capítulos posteriores. Sólo es una ayuda visual que facilita la lectura, y no tiene ninguna implicancia sobre la implementación.

También existen situaciones en las cuales la propia asociación tiene sentido como clase. Por ejemplo, la figura 4.21 muestra lo que se denomina una **clase de asociación**, que ilustra la relación de empleo entre una `Organización` y una `Persona` como una clase en sí, denominada `Empleo`.

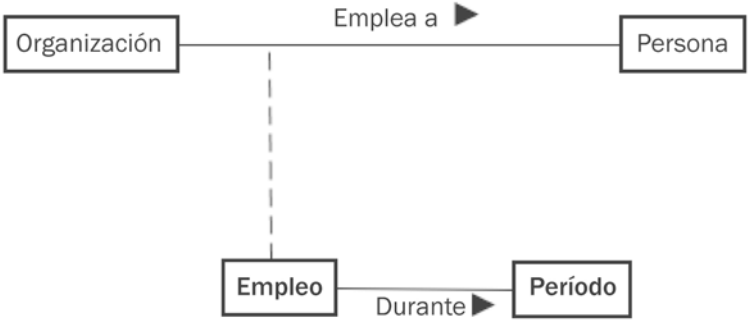

Figura 4.21 Clase de asociación.

Más sobre generalizaciones y especializaciones

Las especializaciones que mostramos en un diagrama no tienen por qué ser permanentes. Por ejemplo, una persona en particular puede ser `Scrum Master` en un proyecto y `Team Member` en otro. En esos casos, se puede utilizar el estereotipo `<<dynamic>>`, como se observa en la figura 4.22.

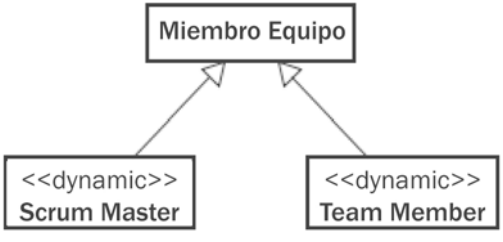

Figura 4.23 Especializaciones transitorias.

También puede ocurrir que haya especializaciones varias para un mismo concepto. Por ejemplo, podríamos querer representar que la clase `Miembro Equipo` puede ser especializada como `Product Owner`, `Scrum Master` y `Team Member`, en cuanto a su rol en el equipo, y a la vez indicar con dos subclases si se trata de personas asignadas al equipo a tiempo completo o a tiempo parcial. Para ello, debemos usar una especialización discriminada, como se muestra en la figura 4.23.

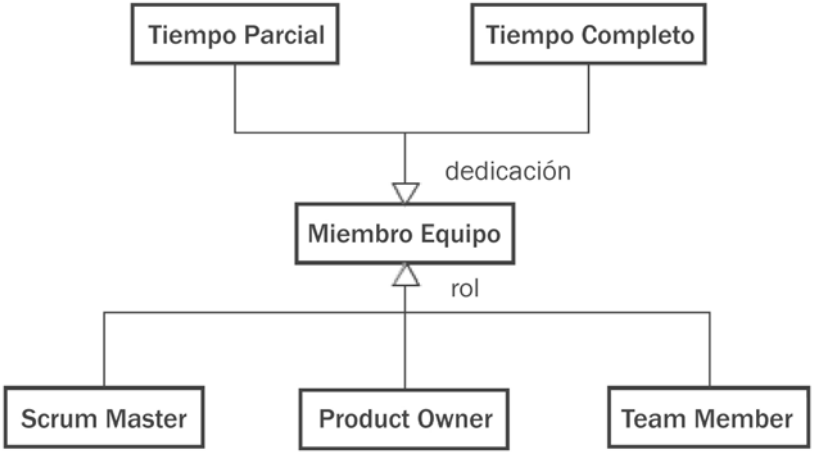

Figura 4.23 Especialización con discriminadores.

Notas en diagramas UML

Lo más probable es que usar todas las características de los diagramas de clases no ayude demasiado a la comunicación con los usuarios y otros interesados, sino que la compliquen. Por eso, en muchos casos, no hay mejor herramienta que un comentario en forma de texto agregado al diagrama. UML nos provee comentarios mediante un elemento que llama **nota**.

La figura 4.24 muestra una nota aclaratoria sobre la característica transitoria de una especialización, que bien puede reemplazar a la de la figura 4.22.

También sería mejor incluir una nota aclaratoria para explicar mejor la naturaleza de un `Product Backlog Item` y un `Sprint Backlog Item`. Eso lo hemos hecho en la figura 4.25, que resulta en un diagrama más ortodoxo que el comentario introducido en el diagrama original.

Si bien las notas recién aparecieron a esta altura del libro, se pueden usar en cualquier diagrama de UML, y suelen ser muy útiles para agregar información no estructurada a los mismos.

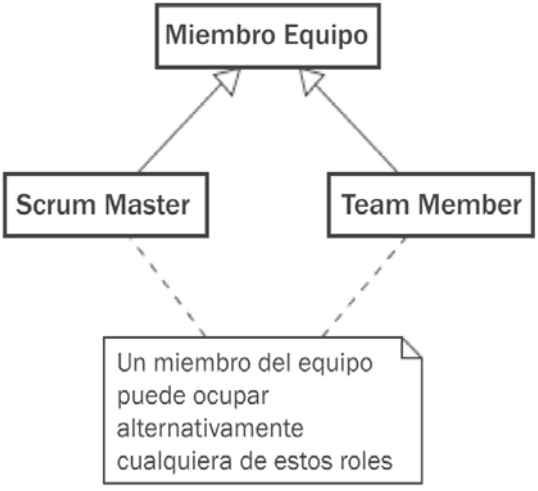

Figura 4.24 Nota aclaratoria.

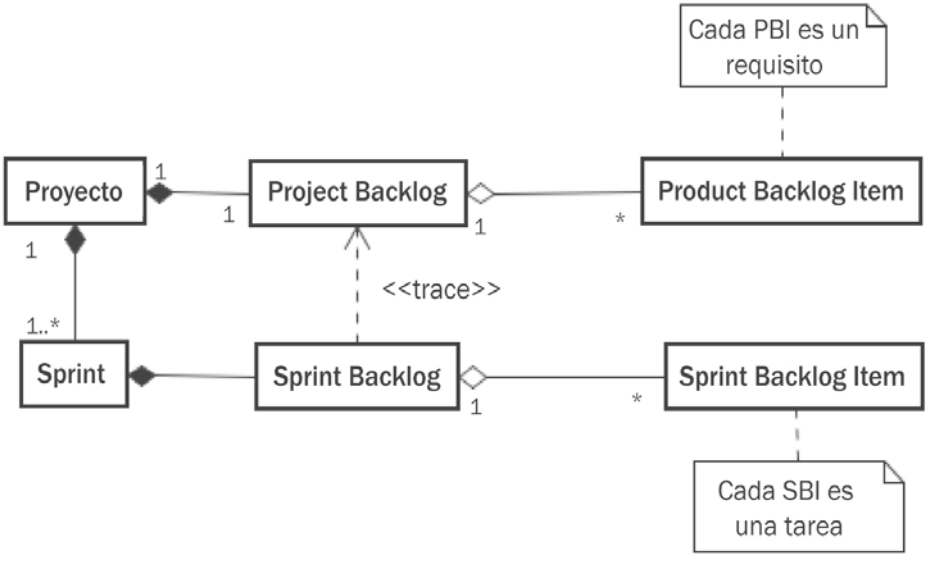

Figura 4.25 Nota aclaratoria.

Mecanismos de abstracción y relaciones entre clases

A modo de resumen, en la el cuadro 4.2 mostramos las descripciones de distintos escenarios de los mecanismos de abstracción. Esta lista no pretende ser exhaustiva, pero muestra algunas de las situaciones más comunes y cómo se deberían modelar.

SITUACIÓN	EJEMPLO	RELACIÓN
Una instancia de una clase es una parte física de una instancia de otra	`Documento – Encabezado de Documento`	Agregación
Una instancia de una clase es una parte lógica de una instancia de otra	`Product Backlog – Product Backlog Item`	Agregación
Una instancia de una clase está contenida físicamente en una instancia de otra	`Organización – Usuario Cliente`	Composición
Una instancia de una clase está contenida lógicamente en una instancia de otra	`Organización – Proyecto`	Composición
Una clase es parte de la descripción de otra	`Sprint Backlog – Duración`	Asociación
Una clase obtiene información de otra	`Sprint Backlog – Product Backlog`	Dependencia
Una clase es una descripción de una propiedad de otra	`Equipo – Scrum Master`	Asociación
Una clase es un caso más general de otra clase (las instancias de una son un superconjunto de las instancias de la otra)	`Miembro Equipo – Persona`	Especialización (inversa de Generalización)
Una clase es un caso particular de otra clase (las instancias de una son siempre instancias de la otra)	`Persona – Miembro Equipo`	Generalización (inversa de Especialización)
Las instancias de una clase se registran, temporal o permanentemente, en otra clase	`Product Owner – Proyecto`	Asociación

Cuadro 4.2 Relaciones entre clases.

Diagramas de clases conceptuales

El diagrama de clases es el más rico de UML, y se aplica –de diferentes maneras– en la práctica totalidad de las actividades de desarrollo. Aquí sólo hemos visto su uso en el modelado conceptual. Sin embargo, aun así, tal vez hayamos ido demasiado lejos: no debemos olvidar que el modelado de requisitos debe ser validado con el cliente, y no tiene sentido usar una notación muy compleja con un cliente que no la comprende del todo. Por eso, recomiendo usar solamente los elementos de los diagramas de clases que se adapten al cliente en cuestión. Cuando algo parezca muy complejo a nuestro interlocutor, lo mejor es simplificar el diagrama y agregarle notas a modo de comentarios.

Tengamos en cuenta que, en rigor de verdad, un diagrama de clases hecho a nivel conceptual, como lo hicimos aquí, no está representando necesariamente la noción de **clase** de la programación orientada a objetos. Si bien se usa el mismo diagrama y los mismos conceptos, no hay una relación unívoca. Además, los diagramas de clase que se hacen con fines de implementación tienen un nivel de detalle diferente, como veremos en futuros capítulos del libro. Lo que ocurre es que la noción de clase en UML es muy amplia, e incluye cualquier clasificador, como las clases software.

No obstante, es muy probable que algunos de los conceptos que aparecieron aquí se conviertan en clases del futuro programa, siguiendo la forma de modularización más común de la orientación a objetos, que analizaremos en el capítulo próximo. Por eso mismo es que no hemos demostrado mucha resistencia a denominarlas simplemente clases. Pero es importante destacar que son clases conceptuales, no clases software.

DIAGRAMAS DE CASOS DE USO: CUESTIONES AVANZADAS

Hay cuestiones adicionales que se pueden modelar en los diagramas de casos de uso. Entre ellas, la generalización y las dependencias de inclusión o extensión. La utilidad de estas relaciones dentro del diagrama es la de reutilizar casos de uso, de modo de no repetir un comportamiento común en distintos casos de uso.

Los conceptos de generalización-especialización y de dependencia tienen el mismo significado que en los diagramas de clases conceptuales que ya vimos.

En la figura 4.26 se muestra que el actor `Administrador` es un caso particular del actor `Usuario del sistema` (dicho de otro modo, el actor `Usuario del sistema` generaliza al actor `Administrador`).

Figura 4.26 Generalización de actor.

El actor especializado hereda el comportamiento del actor más general y lo refina de alguna forma.

También se puede utilizar la generalización entre casos de uso. Por ejemplo, el caso de uso `Dar de alta usuario` puede ser una generalización de los casos de uso `Dar de alta usuario de reportes`, `Dar de alta Product Owner` y `Dar de alta Scrum Master`. El problema con este uso es su escasa utilidad y la confusión que surge con las relaciones de dependencia que veremos a continuación. El caso recién descripto se puede ver en la figura 4.27.

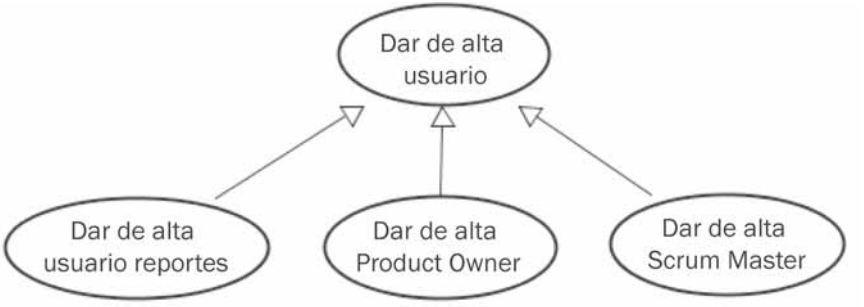

Figura 4.27 Generalización de casos de uso.

Las relaciones de dependencia entre casos de uso son de dos tipos: de inclusión y de extensión. ambas se especifican con una flecha de línea de puntos y el estereotipo `<<include>>` o `<<extends>>`, respectivamente.

La **extensión** implica que un caso de uso se define como una extensión incremental de otro **caso de uso base** (el extendido) en un **punto de extensión**. El punto de extensión es lo que distingue la relación de extensión de la relación de generalización entre casos de uso.

Es una relación que enriquece un caso de uso sin modificarlo. A veces es un flujo alternativo de un caso de uso, que por su importancia adquiere la entidad de un caso de uso en sí mismo.

La figura 4.28 muestra que el caso de uso `Emitir reporte en formato pdf` extiende al caso de uso `Emitir reporte en papel`.

Figura 4.28 Extensión entre casos de uso.

Notemos nuevamente que la extensión no es una generalización. `Emitir reporte en formato pdf` no es un caso particular de `Emitir reporte`, sino que es una extensión a un caso de uso que, en principio, emite un reporte en papel de todas maneras.

La indicación de extensión se puede acompañar de una condición, lo que se hace mediante una nota vinculada a la dependencia, con el rótulo `Condition`, como se ilustra en la figura 4.29.

También se puede indicar sólo el punto de extensión, como se indica en la figura 4.30.

La **inclusión** implica que un caso de uso (el incluyente), incorpora el comportamiento de otro caso de uso (el incluido) como parte de su propio comportamiento en

un determinado momento de su curso de acción. Modelar la inclusión sirve para evitar la repetición de pasos en distintos casos de uso, cuando son los mismos.

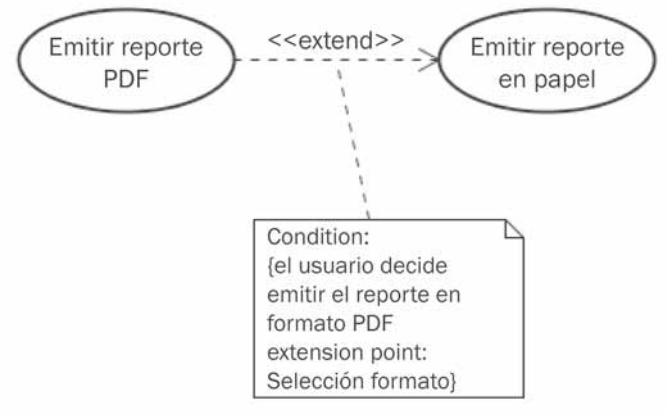

Figura 4.29 Condición de extensión entre casos de uso.

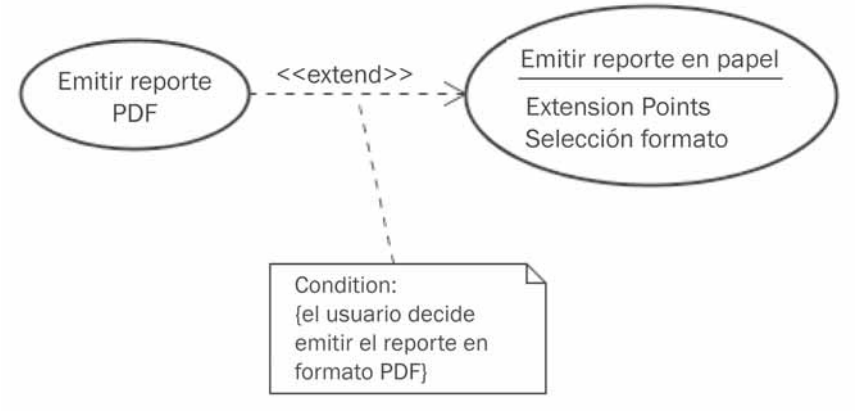

Figura 4.30 Punto de extensión de un caso de uso.

La figura 4.31 muestra que el caso de uso `Modificar datos de un usuario` incluye al caso de uso `Modificar domicilio de un usuario`.

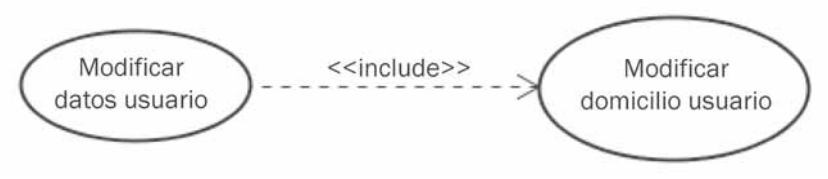

Figura 4.31 Inclusión entre casos de uso.

En ocasiones más sencillas, una inclusión puede expresarse simplemente como un paso más del caso de uso más general.

Contrariamente a lo que pueda pensarse, estos elementos avanzados de los casos de uso no ayudan a convertirlos en una herramienta más útil. De hecho, tal vez la inclusión y la extensión puedan parecer útiles, pero lo son más en los casos de uso textuales que en los diagramáticos. Incluso hay muchas personas experimentadas que confunden la dependencia de extensión con la generalización y usan mal o no usan la relación de inclusión. Y, en ningún caso, estos agregados en los diagramas sirven para modelar comportamiento.

Por todo esto, si se desea usar diagramas de casos de uso, hay que tener especial cuidado en no utilizar relaciones complejas si no van a ser entendidas por los interlocutores. Por ejemplo, la relación de generalización entre casos de uso habría que evitarla. Las relaciones de inclusión entre casos de uso o de generalización entre actores suelen ser mejor comprendidas. La relación de extensión está en un punto intermedio, que habría que chequear que sea bien comprendida antes de usarla. Y por sobre todas las cosas, recordemos siempre el valor de la simplicidad, evitando los diagramas excesivamente complejos.

¿Y LOS REQUISITOS NO FUNCIONALES?

Como dijimos con anterioridad, UML no tiene ningún artefacto para la modelización de requisitos no funcionales. Este déficit debe ser cubierto con otras notaciones o artefactos.

En realidad –siendo sinceros– UML no define demasiadas herramientas para el modelado de requisitos en general. Al fin y al cabo, los diagramas de casos de uso sólo modelan unos pocos aspectos de los requisitos funcionales. Los diagramas de actividades, más allá de su utilidad, fueron concebidos para describir flujos de procesos, mientras que los de clases son herramientas de análisis y diseño que se han podido utilizar también para algunos requisitos.

Por suerte, como veremos en los capítulos sucesivos, sí hay herramientas de UML para cubrir completamente el análisis y el diseño de una aplicación orientada a objetos.

ARTEFACTOS PARA EL MODELADO DE REQUISITOS QUE NO SON PARTE DE UML

La insuficiencia de UML para modelar requisitos de productos de software hace que se hayan buscado opciones por fuera del propio lenguaje de modelado.

Por ejemplo, los **prototipos** han sido, desde hace ya mucho tiempo, una excelente herramienta de especificación y de validación de los requisitos de clientes. Entre ellos, se destacan los de interfaz de usuario, los de comportamiento parcial, los de interacción con otras aplicaciones, etc. Los prototipos terminan siendo fundamentales para todos aquellos usuarios que no saben lo que desean en detalle hasta que no lo ven funcionando. Incluso, los prototipos en papel son una herramienta de captura y validación también.

Los **glosarios de términos** ayudan mucho al modelado conceptual. Si bien los diagramas de clases son útiles para mostrar relaciones entre conceptos, no nos proveen definiciones de los mismos, que sólo pueden expresarse en modo textual mediante glosarios.

Finalmente, los casos de uso de forma textual, como los que vimos al principio del capítulo, son, tal vez y mal que les pese a los defensores de la notación gráfica, la mejor herramienta de modelado y validación de los requisitos.

DE LOS REQUISITOS DEL CLIENTE AL ANÁLISIS DEL SISTEMA

Lo que acabamos de ver en este capítulo ha sido una serie de herramientas que nos ayudan a modelar los requisitos del cliente.

La tarea de determinar los requisitos no es trivial, porque:

- Muchos clientes no saben lo que quieren hasta que no ven algo del sistema funcionando.
- Muchos clientes que sí saben lo que quieren, les cuesta expresarlo en forma de requisitos.
- Muchos clientes y usuarios tienden a intentar mantener, en los nuevos sistemas, características que sólo tenían sentido en sistemas anteriores.
- Otros usuarios, cuando ven el sistema funcionando, tienden a darse cuenta de que hay más posibilidades de las que imaginaban, y empiezan a solicitarlas.
- Algunos de los anteriores caen en la práctica humorísticamente denominada *featuritis*, que es la búsqueda de más y más características en el sistema en cuestión y, a veces, los informáticos los impulsamos más en ese sentido.

Por eso, orientar a todos los interesados es parte de nuestro trabajo. Para ello, es fundamental definir quién es el usuario, qué necesita, qué cree necesitar y qué quiere. Luego, no olvidar al cliente, en el sentido de que es quien paga, y también espera beneficios del sistema. Y tener en cuenta a todos los interesados que no sean el cliente o el usuario, porque a ellos también afectará el sistema. Por todo esto, identificar a los interesados también es parte de la captura de requisitos.

Tal vez, por esta suma de cosas, muchos profesionales sostienen que la parte más difícil del desarrollo es la captura de requisitos y su validación. Y quizá por eso, irónicamente, UML no ha colaborado tanto en el modelado de los requisitos.

A continuación veremos que UML sí nos brinda especial apoyo en el modelado del sistema si los requisitos del cliente están claros.

1 TDD es el acrónimo inglés para *Test Driven Development*, una técnica de diseño de software guiada por pruebas unitarias automatizadas. ATDD significa *Acceptance Test Driven Development,* y parte de pruebas de aceptación. BDD es el acrónimo de Behaviour Driven Development, y se basa en derivar el comportamiento del sistema de las pruebas de aceptación. Son tres técnicas complementarias y con muchas prácticas en común.

2 El nombre en inglés es *swimlanes*.

5

MODELADO DEL ANÁLISIS O DE LA DEFINICIÓN DEL PRODUCTO

ANÁLISIS ORIENTADO A OBJETOS

Hasta este momento, no apareció en el libro ningún concepto de orientación a objetos, a pesar de que UML es un lenguaje de modelado para sistemas orientados a objetos.

Desde el análisis ya es necesario encarar el paradigma de objetos y sus conceptos. De allí que debamos detenernos un poco a precisar conceptos que, siendo del paradigma, también existen en el lenguaje UML.

La orientación a objetos es un paradigma que, entre otras cosas, busca reducir la brecha entre el dominio del problema y el dominio de la solución, como camino para construir software más complejo con mayor simplicidad. Por eso nos hemos centrado en actividades destinadas a definir el sistema que se construirá: el modelado del dominio del problema lo vimos en el capítulo de requisitos, mientras que en éste nos vamos a concentrar en definir la solución.

Para lograr este objetivo, la orientación a objetos plantea el viejo principio de modularización, o "divide y vencerás", que consiste en dividir el problema en partes.

Sin embargo, las metodologías estructuradas ya hacían esto. De hecho, la idea de la división en módulos surge en el marco del paradigma estructurado. La diferencia fundamental entre ambos es que, mientras en el paradigma estructurado la división en partes se hace sobre la solución procedural de la solución, el paradigma de objetos empieza por dividir en partes al propio problema, para que la división de la solución sea una consecuencia de la división del problema. En este sentido, en el paradigma estructurado la división en módulos es una decisión de diseño, mientras que en la orientación a objetos es una decisión de análisis.

Lo que hace la orientación a objetos es fraccionar el problema sobre la base de las entidades del dominio, que deberían venir del análisis del dominio realizado con los requisitos, y que nosotros hemos modelado con el diagrama de clases de dominio. A partir de estas clases, un refinamiento posterior, que nos acerque al sistema que construiremos, nos va a llevar a las clases de análisis y, recién a partir de allí, llegaremos a las clases de diseño. Si bien todo esto no siempre es demasiado consciente, es lo que subyace a todo el paradigma.

MODELADO DE OBJETOS Y CLASES

Objetos y clases

El concepto central de la orientación a objetos, como no podía ser de otra manera, es el de objeto.

En el paradigma de objetos, un sistema de software es siempre un conjunto de objetos que se envían mensajes y los responden. Por lo tanto, un **objeto** es toda entidad capaz de entender un mensaje y responder a él con algún comportamiento establecido. Además, los objetos tienen datos internos, que conforman su estado, y que les permiten responder los mensajes de maneras distintas según ese estado.

Todo objeto pertenece a una **clase** o, como decimos habitualmente, es instancia de una clase. En ese sentido, una clase es un conjunto de objetos. Pero, además, es la clase la que define qué mensajes puede entender un objeto y el comportamiento esperado como reacción a cada mensaje recibido. Asimismo, la clase es un molde del estado de los objetos, ya que en ella está definida la estructura interna del objeto.

Por lo tanto, los conceptos de objeto y clase están muy asociados. Si bien los objetos son las únicas entidades que tienen existencia en un programa orientado a objetos, las clases son las que definen cuáles son los estados y comportamientos posibles de los objetos que son sus instancias.

Una clase, en definitiva, es un conjunto de objetos con la misma estructura y con el mismo comportamiento. La relación entre objeto y clase se denomina **instanciación**. Cada objeto particular es una **instancia** de la clase.

UML, que es una notación orientada a objetos, permite modelar los dos conceptos.

Modelado simple de objetos

Un objeto se representa con un rectángulo en el que figura el nombre del objeto y la clase de la que éste es instancia, subrayados, como se muestra en la figura 5.1.

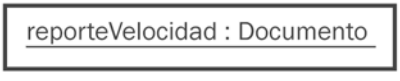

Figura 5.1 Un objeto.

Hay ocasiones en las que necesitamos modelar un objeto sin que nos interese conocer exactamente su clase. Otras en las que nos interesa representar una instancia

de una clase sin darle un nombre. Estas dos circunstancias se ejemplifican en las figuras 5.2 y 5.3, respectivamente.

Figura 5.2 Un objeto del que no nos interesa su clase.

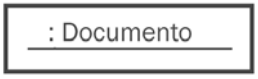

Figura 5.3 Una instancia de la que sólo conocemos su clase.

Los objetos, en un sistema, pueden relacionarse con otros objetos. Esto se puede representar con un **diagrama de objetos** que muestre varios de ellos en un momento dado, con sus relaciones o enlaces, como se muestra en la figura 5.4.

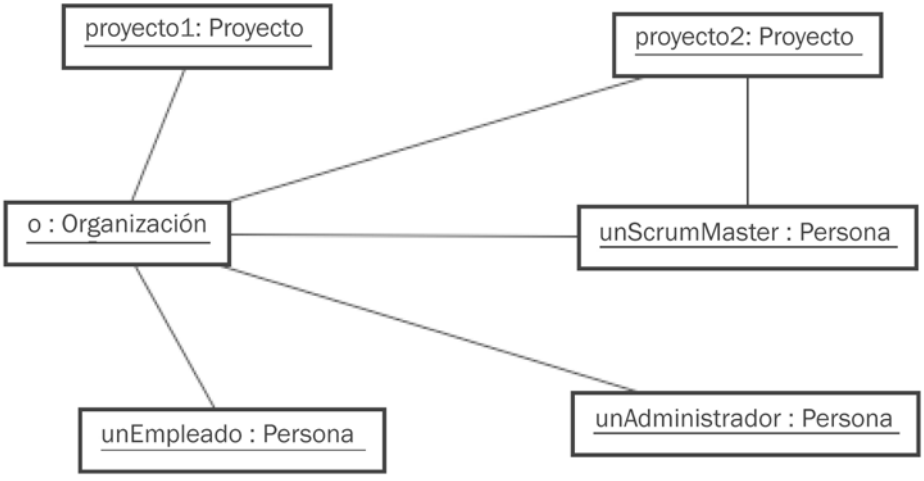

Figura 5.4 Diagrama de objetos.

Modelado de clases con responsabilidades

Una clase se representa como un rectángulo que, como mínimo, debe tener su nombre. Pero hay muchas variantes adicionales, que veremos a lo largo del libro. En el capítulo anterior, desarrollamos una forma simple, que puede ser suficiente en algunos casos. A continuación, estudiaremos una de las maneras más sencillas de analizar clases, mediante el agregado de responsabilidades.

Las **responsabilidades** que tiene un objeto son aquellas cosas que esperamos de ellos. En principio, al menos, esperamos de ellos que almacenen ciertos datos y exhiban cierto comportamiento. Como son las clases las que definen las

responsabilidades de los objetos que son sus instancias, se suelen representar las responsabilidades en el nivel de clases.

Un buen diagrama de clases de análisis podría realizarse indicando las clases del sistema, sus relaciones, y agregando, en cada una, las responsabilidades que le competen. Esto se hace mediante el agregado de un compartimiento debajo del nombre de la clase, con el título `Responsabilidades`, y cada una de las responsabilidades se escribe, en este compartimiento, luego de un guión doble. La figura 5.5 muestra un caso en el que las únicas relaciones que elegimos modelar son las dependencias entre clases.

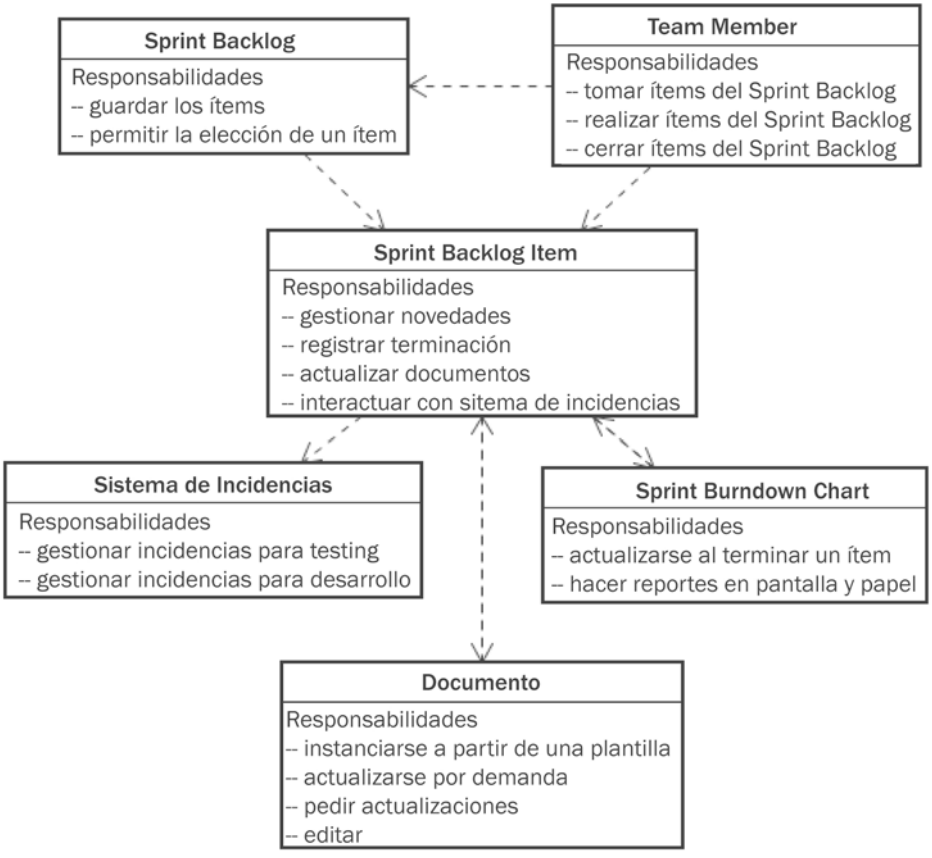

Figura 5.5 Diagrama de clases con responsabilidades.

Ésta es una forma abstracta e independiente de la implementación de modelar clases de análisis. Tal vez sea la mejor. No obstante, hay quienes pretenden que quede más información como resultado del análisis. Una posibilidad es basarnos en el comportamiento de los objetos, usando diagramas de interacción y diagramas de clases más ricos, como veremos poco más adelante.

Otros autores prefieren clasificar las clases en el análisis con estereotipos que indiquen si se trata de entidades, de clases de interfaz de usuario (que sirven para interactuar con usuarios externos) o de clases de control o coordinación. Para ello, usan los estereotipos `<<entity>>`, `<<boundary>>` y `<<control>>`.

En principio yo no recomiendo este uso, y menos en el análisis, ya que me parecen decisiones de diseño y un tanto limitadas. Aún en el diseño, prefiero usar paquetes para separar incumbencias. Sin embargo, a quienes trabajen con UP, les puede ser útil.

ANÁLISIS BASADO EN COMPORTAMIENTO

Comportamiento y métodos

El **comportamiento** describe los estímulos y reacciones de un objeto, o dicho de otro modo, los servicios que brinda ante solicitudes de otros objetos. Es la manera en que éste reacciona ante mensajes o estímulos recibidos, básicamente enviando mensajes a otros objetos, respondiendo al que le envía el mensaje o cambiando de estado. Las distintas formas de respuesta se denominan **métodos**. Los métodos –u **operaciones**, como también se los llama– tienen un nombre, y pueden tener uno o más parámetros y devolver, o no, un valor. Cuando se invoca un método sobre un objeto se dice que le estamos pasando un **mensaje**.

Existen métodos que no representan el comportamiento de los objetos, sino de la clase como un todo. Los denominamos **métodos de clase**.

El comportamiento es un ingrediente fundamental de los objetos y las clases. Una clase sin comportamiento es –salvo unas pocas excepciones– indicativa de un mal diseño. Si bien no estamos todavía realizando clases de diseño, es bueno que las clases de análisis sean un buen punto de partida. Por ello, es habitual refinar el análisis de clases mediante técnicas basadas en el comportamiento.

Diagramas de comunicación o de colaboración

Tal vez el mejor diagrama para encontrar métodos en objetos sea el diagrama de comunicación o colaboración.

Antes de seguir, debo hacer una digresión. Se llama diagrama de comunicación en UML2 lo que en versiones anteriores se conocía como diagrama de colaboración. Si bien el nombre fue modificado porque en UML ya existía otro concepto denominado colaboración, casi todos los profesionales siguen refiriéndose al diagrama con el nombre antiguo. A pesar de ello, en aras de mantener el léxico oficial, y no obstante la escasa utilidad de las colaboraciones de UML 2, en este libro vamos a denominarlo diagrama de comunicación.

El **diagrama de comunicación** es un diagrama de objetos en un escenario, al que se le agregan los mensajes que los objetos se envían entre sí.

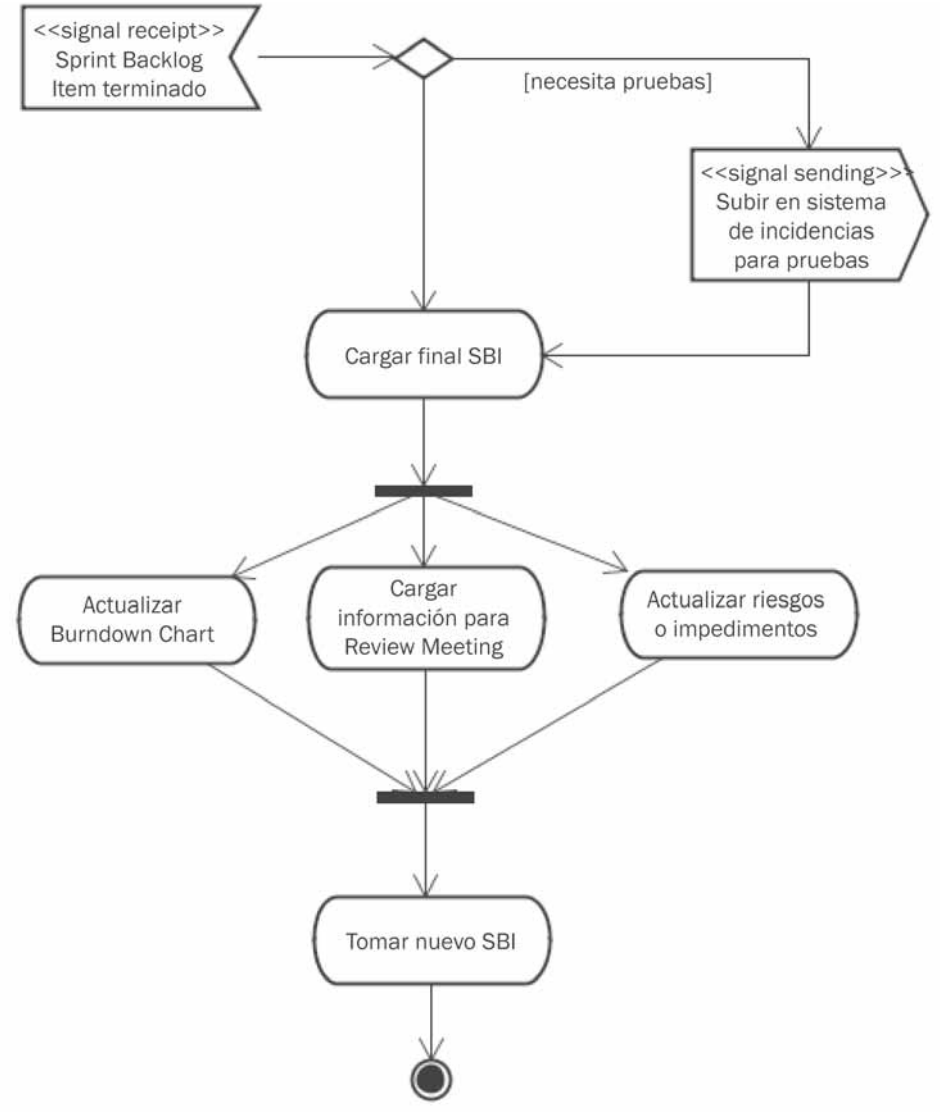

Figura 5.6 Diagrama de actividades del cierre de una tarea.

Por ejemplo, en Scrum, cada vez que un miembro del equipo termina una tarea (denominada *Sprint Backlog Item*), debería cerrarla e iniciar otra. Supongamos, además, que en nuestro sistema *FollowScrum* manejamos por separado el desarrollo y las pruebas de cada *Sprint Backlog Item*, de modo tal que cada vez que se termina una tarea, si corresponde probarla, deba subirse al sistema de incidencias para que otra persona la pruebe. Vamos a suponer también que se deben dejar novedades para la

reunión de revisión del *Sprint* y actualizar el documento de riesgos. El diagrama de actividades podría ser el que se muestra en la figura 5.6.

Basándonos en el diagrama de la figura 5.6, haremos un diagrama de comunicación para ese escenario, que se muestra en la figura 5.7.

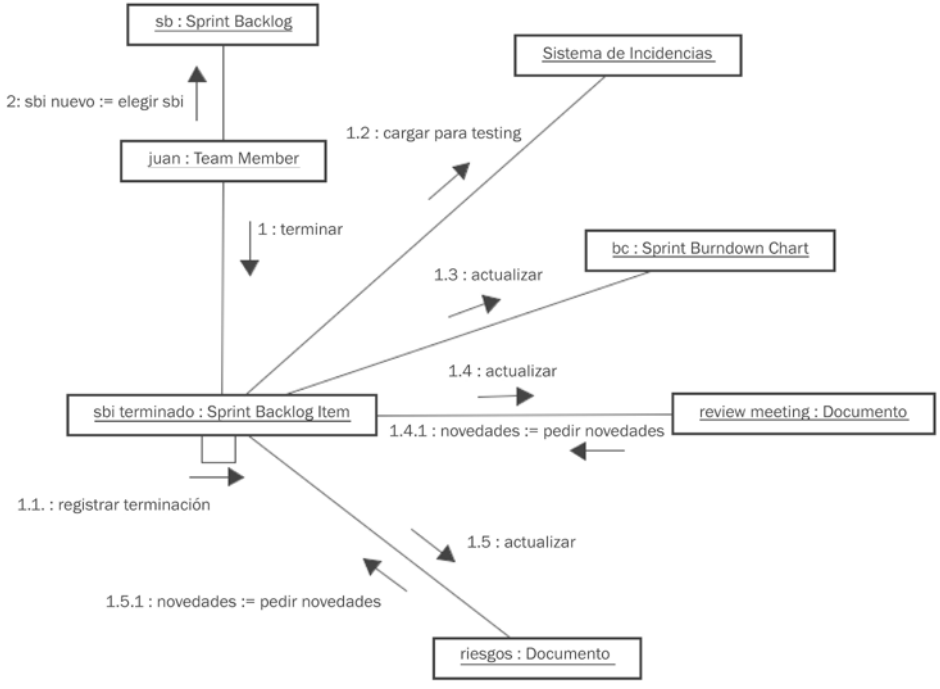

Figura 5.7 Diagrama de comunicación o colaboración.

Como decíamos, es un diagrama de objetos. Pero además de representar los vínculos entre objetos, hemos agregado los mensajes que esos objetos se envían en el escenario en cuestión.

Los mensajes están representados con flechas sobre las líneas que denotan vínculos. Además, el orden de los mensajes se representa mediante el número que los precede que, como muestra el ejemplo, pueden ser anidados, para mostrar que un mensaje provoca el envío de otro. Asimismo, vemos que es posible modelar el envío de un mensaje de un objeto a sí mismo, como el mensaje `registrar terminación`. Si el mensaje provoca la devolución de otro objeto, éste se puede nombrar delante de un signo ":=".

Como también se puede ver, no es sencillo representar concurrencia en un diagrama de comunicación. Sí se puede representar el hecho de que un mensaje sea asíncrono, esto es, que el objeto que lo envió no se queda esperando la respuesta. Un mensaje asíncrono se representa con una flecha de punta abierta. Por el momento, nos pareció una distinción demasiado fina para un diagrama de análisis.

Decíamos que el diagrama de comunicación es ideal para encontrar métodos de las clases. Esto es así porque, al modelar los mensajes y las relaciones entre objetos en un escenario, nos indica qué métodos deben tener las clases de las cuales esos objetos son instancias.

Los diagramas de comunicación no permiten mostrar comportamiento condicional, concurrencia ni iteraciones, aunque hay muchos profesionales que han inventado sus propias notaciones para esto, puesto que se suelen usar para modelar interacciones simples y secuenciales.

Sin embargo, el diagrama de secuencia, que veremos poco más adelante, suele ser un buen sucedáneo para indicar estas cuestiones.

El diagrama de comunicación puede servir también para modelar actividades de un caso de uso de una manera más orientada a objetos que con los diagramas de actividades. En ese caso, ni siquiera es necesario que los participantes sean objetos. Si no lo fueran, lo que se suele hacer es no subrayar los nombres.

Diagramas de clases orientados al análisis basado en el comportamiento

Un diagrama de clases puede mostrar los métodos de cada clase. En estos casos, cada clase se representa como un rectángulo de tres compartimientos. El primero es para el nombre de la clase, el segundo –que por ahora no usaremos– para los atributos, como veremos luego, y, el tercero, para los métodos.

La figura 5.8 muestra un diagrama de clases con comportamiento. Un diagrama como éste sigue siendo conceptual, como el último que vimos, pero incorpora métodos, con lo cual se acerca más al diseño. Es opinable si este nivel de detalle debe ser parte o no del análisis, pero vamos a ser amplios de criterio y los incluiremos aquí.

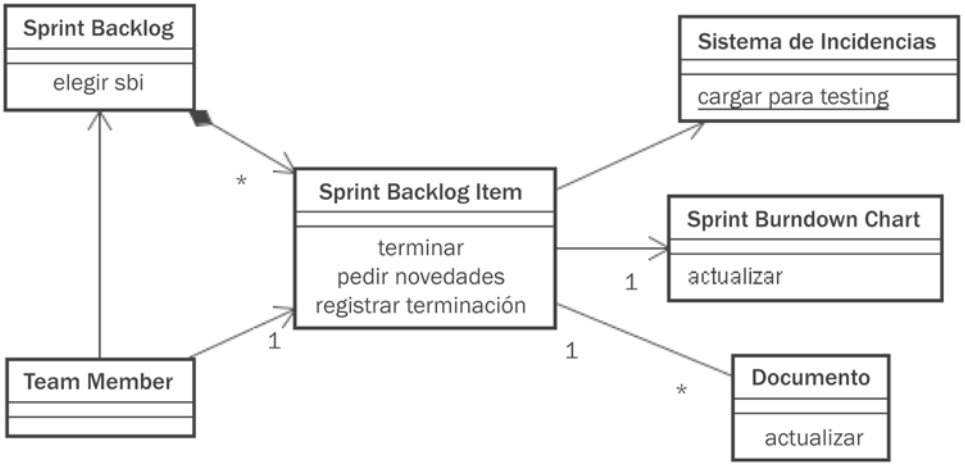

Figura 5.8 Diagrama de clases con comportamiento.

El diagrama anterior es intencionalmente incompleto, pues lo hicimos basándonos en un único diagrama de comunicación. Sería más rico si contásemos con mayor cantidad de escenarios. Sin embargo, agregamos a lo que teníamos en el diagrama de comunicación de la figura 5.7 el vínculo entre las clases `Sprint Backlog` y `Sprint Backlog Item`, que conocemos del análisis de dominio del capítulo anterior.

Lo único novedoso que aparece en este diagrama son los tres compartimientos, los métodos u operaciones y la **navegabilidad**.

En los diagramas de clases puede indicarse la navegabilidad con una flecha en las asociaciones entre clases. Esto se hace para indicar cuál es la clase que necesita los servicios de la otra. Si hemos trabajado con diagramas de comunicación es fácil determinar la navegabilidad entre clases, ya que los envíos de mensajes entre objetos lo indican. Cuando no se indica la navegabilidad, UML explicita que ésta es bidireccional. Sin embargo, dado que tampoco indicamos navegabilidad cuando no queremos modelarla, pienso que es una buena práctica indicar la navegabilidad bidireccional con dos flechas en la relación.

En nuestro diagrama, por ejemplo, la clase `Team Member` debe conocer a `Sprint Backlog Item`, pues ésta le brinda servicios, tal como se observa en el diagrama de comunicación mediante el mensaje `terminar`. De allí que la navegabilidad sea de `Team Member` hacia `Sprint Backlog Item`.

También observamos la aparición de un método de clase, que es el método `cargar para testing` en la clase `Sistema de Incidencias`, circunstancia que hemos indicado subrayándolo. En realidad, como suponemos que sistema de incidencias hay uno solo, no tiene sentido tener un método por instancia, sino uno para toda la clase.

Si bien se pueden colocar los tipos de los parámetros de los métodos en el diagrama y el tipo del valor devuelto, creemos que no es conveniente hacerlo en un modelo de análisis.

Una opción que algunos profesionales consideran más ortodoxa es no hablar de asociaciones y navegabilidad en el análisis, sino sólo de dependencias. En realidad, si se realizaron diagramas de comunicación y se hizo un análisis de dominio, no hay razones para no indicar asociaciones. No obstante, el enfoque que muestra sólo dependencias se ilustra en la figura 5.9.

Una aclaración. Notemos que entre las clases puede haber asociaciones recursivas, esto es, una clase puede tener una asociación consigo misma, como muestra la figura 5.10. El significado en esta figura es que una `Organización` puede estar relacionada, a su vez, con otras `Organizaciones`. Como además hay un símbolo de agregación, se está indicando que hay `Organizaciones` que contienen otras `Organizaciones`.

Incluso, la recursividad puede ser indirecta, como la de la figura 5.11. En este otro caso, estamos diciendo que un `Proyecto Compuesto` es un agregado de subproyectos, que son, a su vez, `Proyectos` que, a su vez, podrían ser `Proyectos Compuestos`.

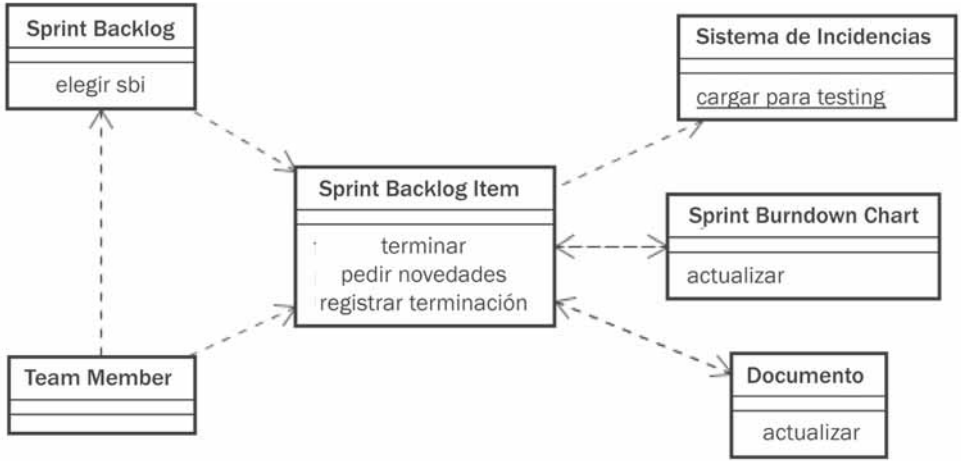

Figura 5.9 Diagrama de clases con dependencias.

Figura 5.10 Asociación recursiva.

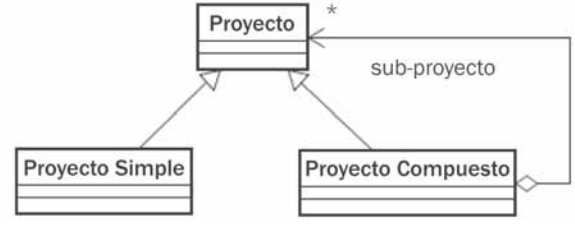

Figura 5.11 Asociación recursiva indirecta.

Ahora bien, esto que decimos de las asociaciones no es válido en las generalizaciones. Una generalización no puede ser ni recursiva ni bidireccional.

Generalización en el modelo conceptual de análisis

Ya que hablamos de generalización, vayan algunas reglas que en los requisitos tal vez no eran importantes, pero que en el nivel de análisis sí debemos considerar.

Ello ocurre porque es muy común que los desarrolladores poco experimentados en el análisis orientado a objetos caigan en generalizaciones y especializaciones especulativas o, por decirlo de manera más cruda, sin sentido.

En el análisis orientado a objetos, una clase A **generaliza** a B (o, lo que es lo mismo, B **especializa** a A), si y sólo si:

- B es un subconjunto de A, esto es, todas las instancias de B son también instancias de A.
- Toda asociación de A tiene sentido como asociación de B.
- Todo método de A tiene sentido como método de B.
- Todo atributo de A tiene sentido como atributo de B (esto último es redundante, y además no hemos visto atributos todavía, así que el lector que no entienda puede saltearlo sin problema).

Si alguna de las condiciones anteriores no se diera, estaríamos ante un mal uso de la generalización.

Además, hay otras pruebas que conviene hacer para que realmente convenga definir, en nuestro sistema, la clase B como una subclase de A. Para asegurarnos de la conveniencia, al menos una de las siguientes condiciones debe ser válida:

- B tiene responsabilidades o métodos adicionales a los de A.
- B no tiene métodos adicionales a los de A, pero los métodos de B deben operar en alguna forma diferente que los definidos en A (esto es, tienen un significado similar pero no idéntico).
- Existen otras clases que deben asociarse a B y no a A.
- B tiene atributos adicionales a los definidos en A (veremos atributos al estudiar el análisis basado en cuestiones estructurales).

Si ninguna de las condiciones anteriores se satisficiera, la generalización podría ser válida desde el punto de vista estrictamente de dominio, pero no tendría sentido en nuestro sistema de software.

Diagramas de estados

El **diagrama de estados** de UML es una herramienta que sirve para modelar cómo afecta un escenario a los estados que un objeto toma, en conjunto con los eventos que provocan las transiciones de estado. También es habitual denominarlo **diagrama de máquina de estados**, o **máquina de estados** a secas. En este libro, usaremos siempre el nombre más habitual de diagrama de estados.

Se pueden usar para modelar objetos reactivos, es decir, aquellos objetos para los cuales la mejor forma de caracterizar su comportamiento sea señalar cuál es su comportamiento frente a estímulos provenientes desde fuera de su contexto, o aquellos que están ociosos hasta el momento en que reciben un evento.

Incluso, podríamos haberlos usado, en una forma muy simplificada, en el modelado de requisitos, para indicar, en nuestro ejemplo, las transiciones entre estados de un proyecto y los eventos que las causan. Sin embargo, su mayor utilidad se ve en el análisis, cuando considerar estados del sistema o de un objeto particular tiene

especial interés para los desarrolladores. Por esta razón, lo hemos examinado en este capítulo.

Los cambios de estado que sufren los objetos se deben a estímulos o mensajes recibidos de otros objetos, cuyos efectos UML llama **eventos**.

Un **evento** indica la aparición de un estímulo que puede disparar una transición de estados. Es la especificación de un acontecimiento significativo, como una señal recibida, un cambio de estado o el paso de un intervalo de tiempo.

Una **transición** entre estados es el paso de un estado a otro: un objeto que esté en un primer estado realizará ciertas acciones y entrará en un segundo estado cuando ocurra algún evento especificado y se satisfagan ciertas condiciones.

El diagrama de estados es un modelo dinámico que muestra los cambios de estado que sufre un objeto a través del tiempo. Se puede modelar toda la vida del objeto o su existencia dentro de un escenario particular.

El diagrama de estados, por lo tanto, es una abstracción que representa los estados, eventos y transiciones de estados para un objeto o un sistema. Es lo que se denomina una **máquina de estados**, que muestra la secuencia de estados en la vida de un objeto, los eventos que modifican estados y las acciones y respuestas del objeto.

La representación gráfica consiste en un grafo o red con nodos para los estados y arcos para las transiciones, además de un texto en correspondencia con los arcos que describen eventos, condiciones y acciones de las transiciones.

Por ejemplo, la figura 5.12 muestra un diagrama de los estados de un proyecto en la aplicación *FollowScrum*.

Como vemos, se parece al diagrama de actividades. De hecho, en UML 1 se consideraba al diagrama de actividades como un caso particular de diagrama de estados. Sin embargo, esto ha cambiado en UML 2.

Los elementos distintivos del diagrama de estados son:

- Los estados por los que pasa el objeto (el proyecto, en este caso) se representan con rectángulos de puntas redondeadas.
- Los nodos inicial y final se representan igual que las actividades de inicio y fin del diagrama de actividades. Como en realidad no son estados, estrictamente hablando, en UML 1.x se los solía llamar pseudo-estado inicial y pseudo-estado final. Hoy se usa simplemente el término **nodo**.
- Las transiciones entre estados se representan con flechas.

Las transiciones de estados pueden tener una leyenda del tipo:

```
evento [condición] / acción
```

El significado de esta leyenda es que la transición se realiza cuando sucede el evento y se cumple la condición, realizando la acción durante la transición.

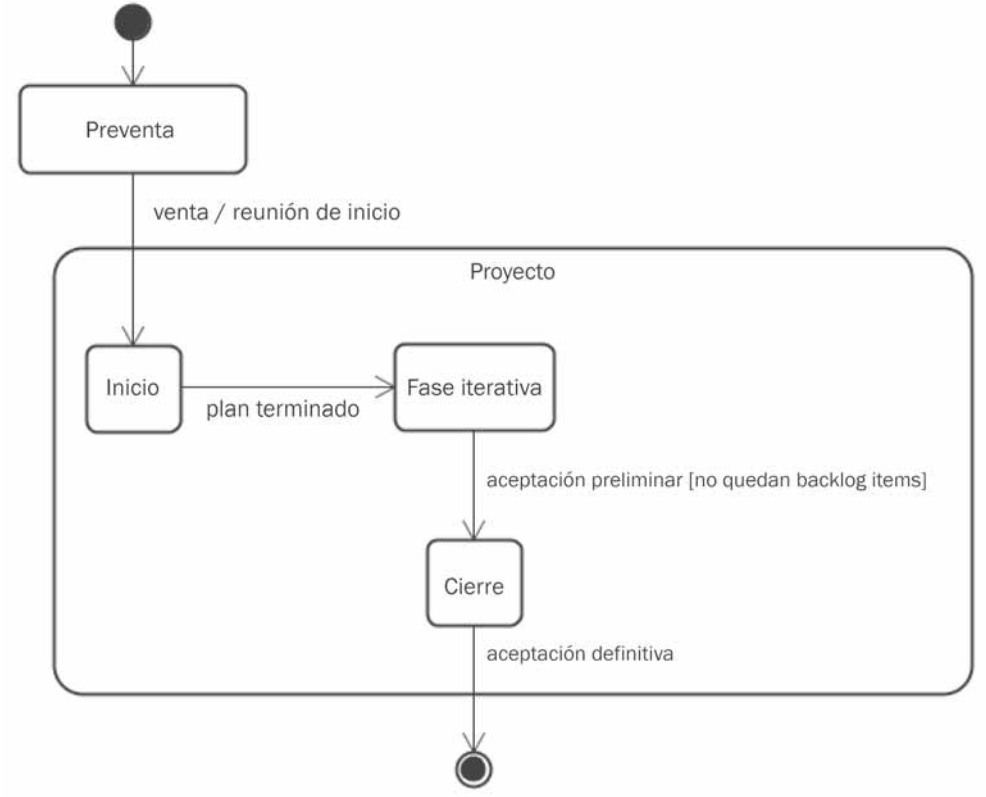

Figura 5.12 Diagrama de estados simple.

De todas maneras, las tres partes de la leyenda son opcionales, y puede no colocarse ninguna.

Como vemos en el diagrama, un estado puede agrupar varios estados internos, como ocurre entre `Proyecto` y los estados `Inicio`, `Fase iterativa` y `Cierre`. Esto mismo puede ser usado para no sobrecargar de flechas un diagrama. Por ejemplo, la figura 5.13 muestra que, estando en cualquier estado dentro de un Sprint, se lo puede cancelar y, en ese caso, se pasa al estado de `Armado del Sprint Backlog`.

Asimismo, existe el concepto de **concurrencia de estados**, que explicaremos sobre la base de la figura 5.14. En ella, estamos mostrando que, durante el *Sprint,* además de realizarse las tareas que allí figuran, en forma paralela y con otra frecuencia, se hacen las reuniones diarias denominadas `Scrum Daily Meetings`.

También se pueden definir acciones internas a un estado, que permiten simplificar un diagrama eliminando las transiciones y estados menos relevantes. La figura 5.15 muestra un diagrama de estados más sencillo que los anteriores, aunque no se ven tan claras las transiciones dentro del *Sprint*.

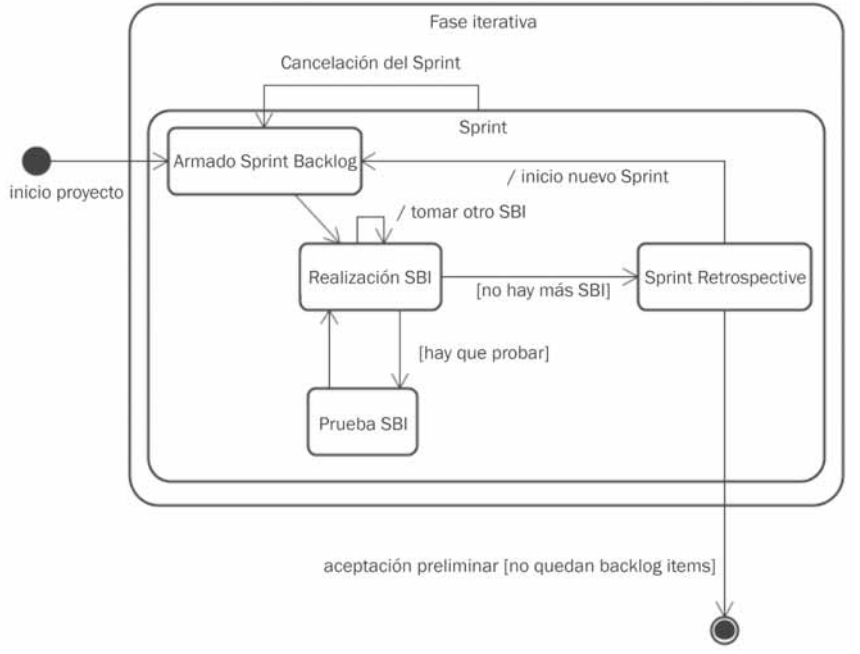

Figura 5.13 Agrupación de estados.

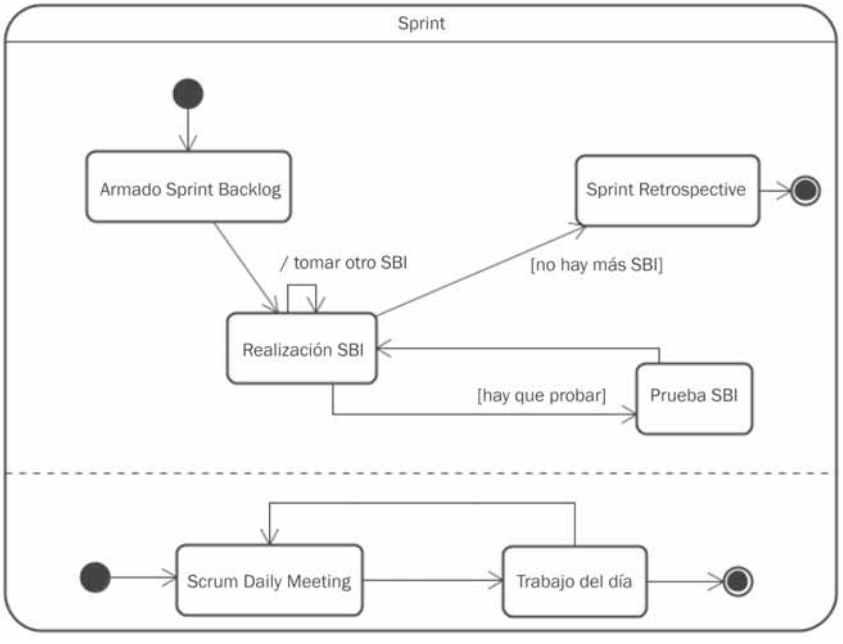

Figura 5.14 Concurrencia de estados.

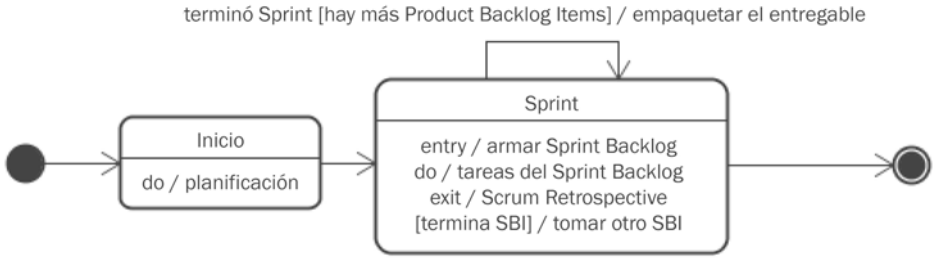

Figura 5.15 Actividades dentro de un estado.

Algunas de las acciones internas más usuales que se pueden definir son las que se ilustran en el cuadro 5.1

RÓTULO	SIGNIFICADO
`entry`	Actividad que se realiza inmediatamente cuando se ingresa al estado.
`exit`	Actividad que se realiza inmediatamente antes de salir del estado.
`evento [condición] / acción`	Transición interna
`do`	Actividad que se realiza dentro del estado

Cuadro 5.1 Tipos de actividades internas a un estado.

De todas estas actividades de estado, las transiciones internas suelen ser más confusas que el resto, y se recomienda evitarlas, modelándolas más bien con transiciones entre estados, salvo que esto último resulte en un diagrama más engorroso.

Como vemos, el diagrama de estados es un modelo sencillo. Por eso mismo, hay que tratar de que permanezca simple, cuidando el nivel de detalle, sin hacer diagramas muy complejos.

Los diagramas de estados representan lo que se conoce como **máquinas de estado**, en las cuales:

- Cada objeto está en un estado en cierto instante.
- Las transiciones de estado son instantáneas.
- Puede haber transiciones temporizadas, que esperan un intervalo de tiempo antes de ocurrir: esto se indica poniendo una condición temporal al evento.
- Lo que provoca una transición es un evento, y éste se asocia a un mensaje recibido.
- El estado en que está el objeto determina su comportamiento.

Son ideales en el caso en el que una especificación mediante estados es más ilustrativa que una que describa actividades, o cuando los objetos tienen un comportamiento complejo.

Diagramas de secuencia

El diagrama de comunicación es un caso particular de una familia de diagramas que se denominan **diagramas de interacción**. Otro miembro de la familia es el **diagrama de secuencia**.

Semánticamente hablando, los diagramas de comunicación y de secuencia son equivalentes. Sin embargo, hay algunas posibilidades de modelado del diagrama de secuencia que no están presentes en el de comunicación.

Por ejemplo, en la figura 5.16, hicimos un diagrama de secuencia equivalente al de comunicación de la figura 5.7.

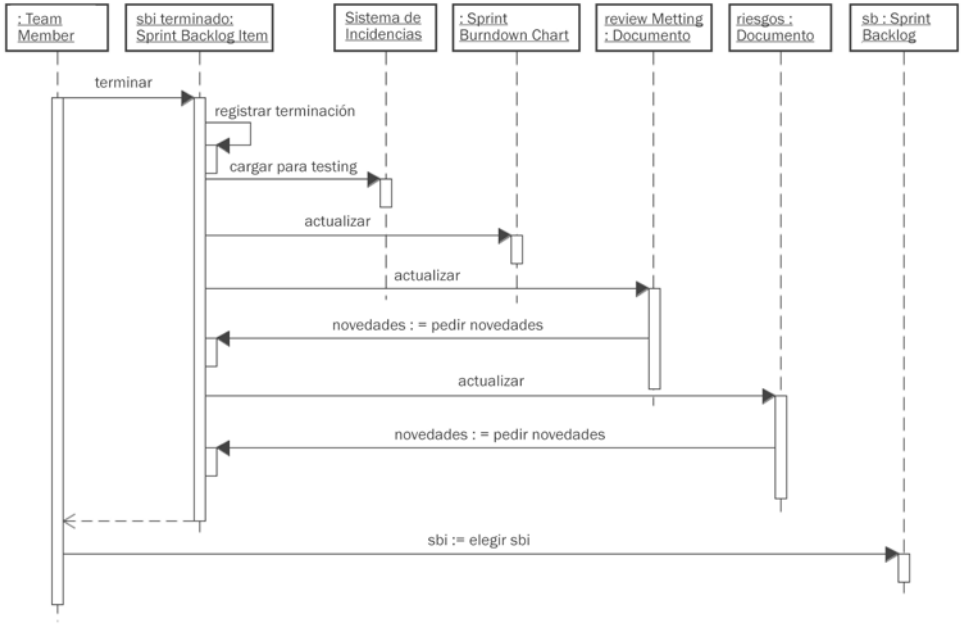

Figura 5.16 Diagrama de secuencia.

Algunas de las cuestiones que se pueden observar en el diagrama son:

- Cada objeto se coloca arriba de una línea, denominada **línea de vida**. La misma puede tener un rectángulo cubriéndola parcialmente, para indicar que en ese momento el objeto está activo.
- Cada mensaje se escribe en forma horizontal, entre la línea de vida del objeto que envía el mensaje y la de aquél que la recibe.
- Si un objeto se envía un mensaje a sí mismo, se dibuja como una flecha que vuelve, como se hizo con el mensaje `registrar terminación`.
- El ordenamiento vertical representa el paso del tiempo. Esto es, un envío de mensaje que está más arriba en el diagrama quiere decir que ocurre

antes que el que figura más abajo. Esta simbología hace que no sea necesario colocar a los mensajes números de orden, como sí era necesario en el diagrama de comunicación.

- La terminación de un mensaje con varios envíos de mensajes anidados se puede indicar con una flecha de línea punteada. Si bien esto no es obligatorio, a veces agrega claridad al diagrama. Nosotros usamos este recurso con el retorno del mensaje `terminar`.

Si bien no es necesario poner números que indiquen el orden y anidamiento de mensajes, hay personas que los colocan.

El diagrama de secuencia permite indicar también condicionales y ciclos iterativos. Si bien recomendamos tratar de evitar estas prácticas, para facilitar la lectura del diagrama, la figura 5.17 muestra varias de estas posibilidades, incluyendo números de orden y la creación y eliminación de objetos:

En la figura 5.17, vemos que, cuando un mensaje crea un objeto, se le coloca el estereotipo `<<create>>` y, cuando provoca la destrucción, se usa `<<destroy>>`. También vemos que la muerte de un objeto se representa con una cruz en su línea de vida. Los estereotipos son opcionales, debido a que el significado de los mismos –creación o destrucción– puede ser deducido del diagrama sin ellos; sin embargo, es habitual colocarlos.

Notemos especialmente que, en la última figura, no hemos subrayado los nombres de los participantes. Esto es válido en UML 2, ya que en esta versión no es necesario que los participantes sean objetos, permitiendo de este modo modelar comportamiento de entidades más abstractas. En UML 1 el subrayado era obligatorio.

Un ciclo iterativo se representa con un marco alrededor de la parte del diagrama que modela las acciones que se van a repetir, con el rótulo `loop` y alguna descripción de las condiciones del ciclo. De la misma manera, se podría indicar un comportamiento condicional, pero en ese caso se utilizaría el rótulo `opt`. Por ejemplo, la condición de guarda `[hay que probar]` se podría haber reemplazado por un marco `opt`. También se puede indicar la ejecución de varias alternativas, mediante el rótulo `alt`, como se muestra en la figura 5.18 para un diagrama de secuencia de sistema. Todo esto vale a partir de UML 2.

En UML 1, y mucha gente sigue haciéndolo así, se podía colocar una guarda sobre la flecha del mensaje condicional o repetitivo. Cuando era una condición, en vez del marco `opt`, simplemente se ponía la guarda entre corchetes, como hemos hecho al escribir `[hay que probar]` en la figura 5.17. Cuando se trataba de una repetición, en vez del marco `loop`, se precedía la guarda con un asterisco. Esto provoca una notación más clara en los casos en que las secciones condicionales o iterativas son de un solo mensaje.

No obstante, tal vez no sea la mejor idea mostrar condiciones o repeticiones en los diagramas de interacción. Lo mejor es representar escenarios simples que sean fácilmente comprendidos de un vistazo. Puede que no siempre convenga eliminar las iteraciones de los diagramas, pero las condiciones suele ser conveniente separarlas en dos diagramas, tal vez usando la técnica de visión global de interacciones que veremos en el próximo ítem.

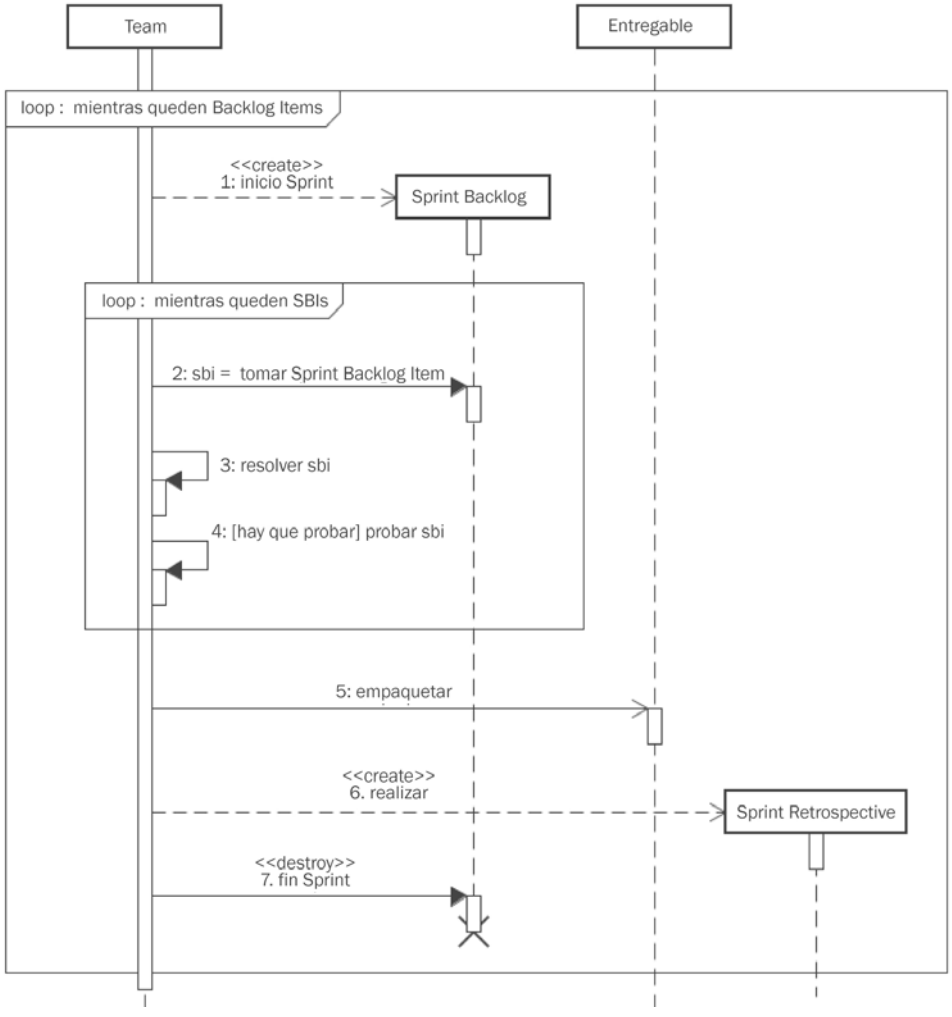

Figura 5.17 Diagrama de secuencia con condicionales y ciclos.

Los rótulos `loop`, `opt` y `alt` que hemos usado en nuestras figuras 5.17 y 5.18 se denominan, en la jerga de UML, **operadores de interacción**. El cuadro que rodea las acciones respectivas es un **marco**.

Los diagramas de secuencia admiten varios tipos de operadores de interacción, además de estos tres, pero los que hemos mostrado son los más comunes.

Vale la pena mencionar también aquéllos de mayor utilidad en contextos concurrentes, tales como `par`, que define acciones en paralelo, y `critical`, que define secciones críticas de una interacción.

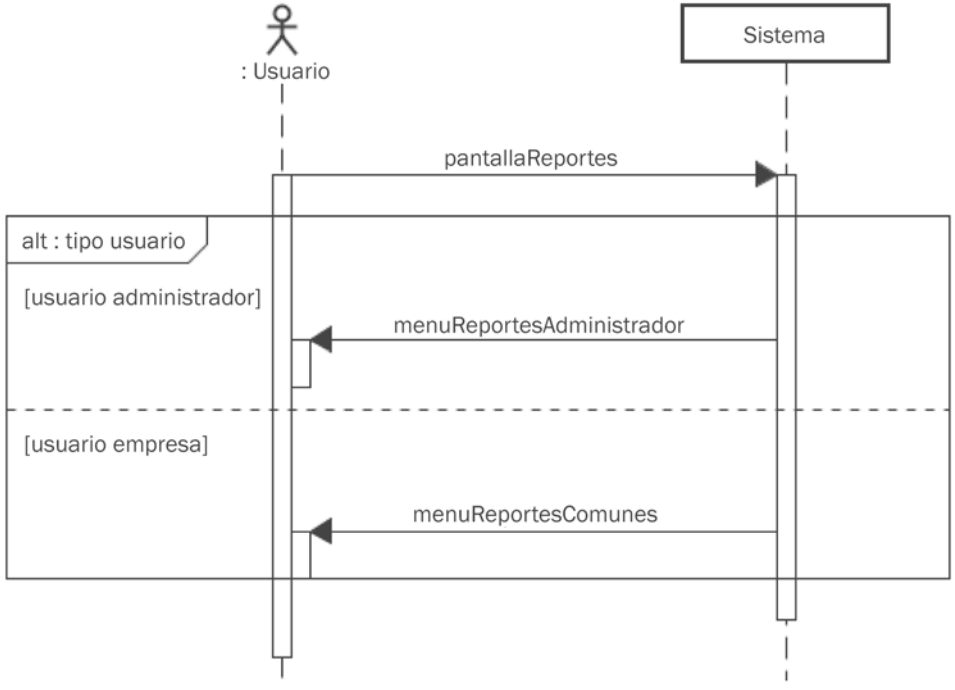

Figura 5.18 Indicación de varias alternativas en diagramas de secuencia.

UML permite que cualquier diagrama se enmarque con un marco con nombre. En este libro, no lo hemos hecho, porque casi siempre empeora la legibilidad. Sin embargo, puede hacerse. Por ejemplo, nuestro diagrama de la figura 5.16 podría haber quedado enmarcado por un marco denominado `FinBacklogItem`, como se ve en la figura 5.19, donde el rótulo `sd` indica que es un diagrama de interacción.

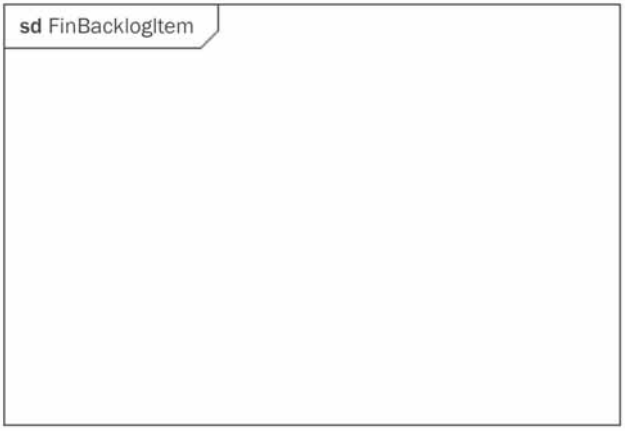

Figura 5.19 Marco de diagrama de secuencia.

Otra característica posible de los diagramas de interacción (comunicación o secuencia) es mostrar cuándo un mensaje es asíncrono, es decir, el remitente no queda esperando una respuesta del receptor. Esta circunstancia se representa con una flecha de punta abierta, como hicimos con el mensaje `empaquetar`.

Hasta la versión 1.3 de UML, los mensajes asíncronos, tanto en diagramas de comunicación como de secuencia, se representaban con una flecha de media punta, como la de la figura 5.20. Hay muchas personas que siguen representando de esta manera los mensajes asíncronos.

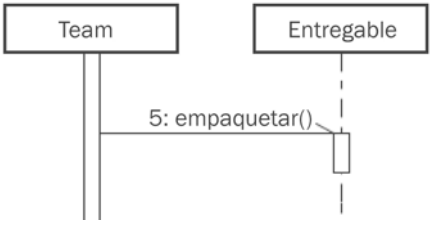

Figura 5.20 Mensaje asíncrono en UML 1.3.

Los diagramas de secuencia son muy ricos y admiten muchas variantes más. Hay profesionales que, para indicar que la llamada a un mensaje no es instantánea, sino que lleva un tiempo, realizan las flechas que los representan levemente oblicuas en vez de horizontales. Usando esta misma posibilidad, podríamos representar mensajes cuyas flechas se cruzan, cuando el comienzo de la llamada de uno es previo a otro, pero tarda más en llegar al receptor y puede adelantarse. No obstante, a pesar de que algunas personas usan estas facilidades, no es lo más habitual.

Dado que los diagramas de secuencia son equivalentes en semántica a los de comunicación, una pregunta interesante para plantearse es cuándo conviene usar uno u otro. Lo cierto es que, por la forma en que se dibujan, los diagramas de secuencia son más aptos para representar el paso del tiempo y analizar temporalmente un escenario, mientras que los diagramas de comunicación resultan interesantes para hacer un primer esbozo de las relaciones entre objetos, brindando mayor libertad al permitir desplegar los objetos en dos dimensiones.

Por eso mismo creo que el diagrama de comunicación se presta más para tareas de análisis, mientras que el de secuencia es mejor para evaluar alternativas de diseño. Pero todo es cuestión de gustos.

Visión global de interacciones

Un **diagrama de visión global de interacciones** sirve para combinar diagramas de actividades con diagramas de interacción. UML lo considera otro diagrama de la familia de los de interacción. Tal vez su uso más interesante sea evitar las alternativas en diagramas de secuencia, usando las construcciones de bifurcación de los diagramas de actividades.

Muchos profesionales lo encuentran una mezcla indeseable de tipos de diagramas diferentes, pero viéndolos desde un aspecto práctico, pueden ser un buen vehículo de comunicación visual.

A primera vista, se parece a un diagrama de actividades con acciones cuya representación se hace con diagramas de interacción. Y ello lleva a preguntarnos si es realmente un diagrama de interacción o debiera ser considerado una variante más detallada del diagrama de actividades.

Tal es la confusión que la especificación oficial de UML hace énfasis en que no se lo confunda con diagramas de actividades, recalcando que se trata de un diagrama de interacción. Pero, a la vez, dice que es una variante de diagrama de actividades.

La figura 5.21 muestra un posible diagrama de estos, que modela la misma situación de la figura 5.18. Si bien resulta un diagrama simple, se nota lo que decíamos acerca de la mezcla de notaciones.

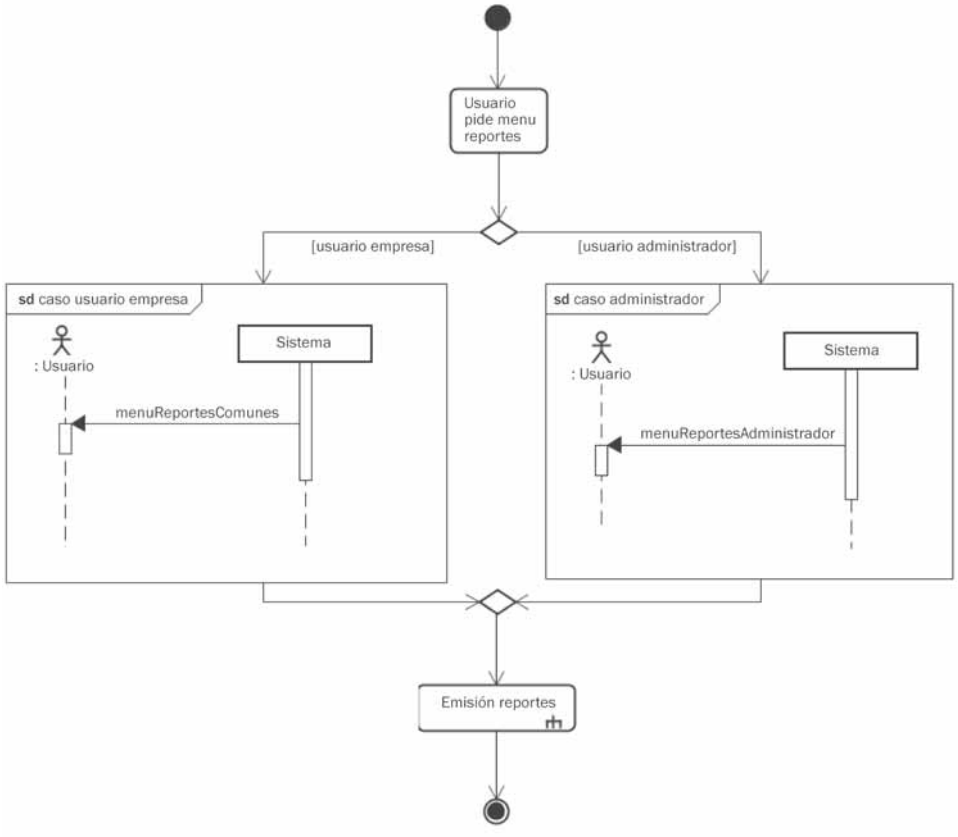

Figura 5.21 Visión global de interacción.

Si miramos la figura 5.21, lo que destaca en lo macro es su parecido con un diagrama de actividades, pero en el cual los nodos no son actividades simples, sino interacciones.

La idea de los creadores de UML, y de la inclusión de este diagrama, es que muestre interacciones, pero en las que destaque la visión global del flujo de control.

ANÁLISIS BASADO EN ASPECTOS ESTRUCTURALES

Si bien el análisis a partir de responsabilidades de clases y basado en comportamiento es lo más ortodoxo desde el punto de vista de la orientación a objetos, hay profesionales que –por provenir de otros paradigmas o por estar modelando un sistema de comportamiento simple– prefieren focalizarse en la estructura de los objetos más que en el comportamiento.

Para ser justos, debemos reconocer que hay muchos profesionales muy reconocidos –tanto en la industria como en la academia– que abogan por el análisis estructural, considerando el análisis del comportamiento como una tarea de diseño.

Como no somos fundamentalistas, a continuación abordaremos el análisis basado en aspectos estructurales.

El **estado** de un objeto está dado por el conjunto de valores internos y vínculos que un objeto tiene, que representa la situación en que está. Los valores internos se guardan en variables que habitualmente se denominan **atributos** o –menos habitual– **campos**.

Hay atributos que no representan el estado de los objetos, sino de la clase como un todo. Los llamamos **atributos de clase**. Un atributo de clase tiene sentido cuando parte del estado de un objeto es siempre el mismo que el de los demás objetos de la clase, o cuando hay estado que es atribuible a la clase.

Los atributos se indican en el segundo compartimiento del rectángulo que representa la clase, y se subrayan en el caso de ser atributos de clase. La figura 5.22 muestra un pequeño diagrama de clases con atributos.

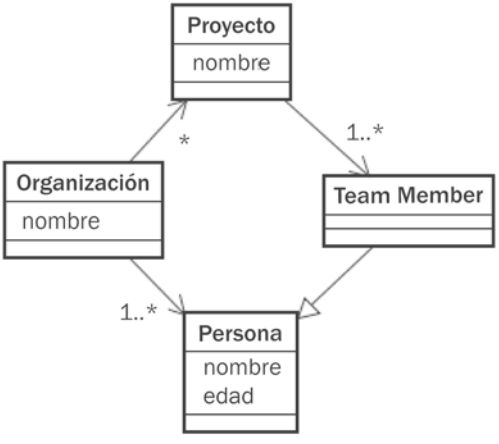

Figura 5.22 Clases con atributos.

Este diagrama no difiere mucho de los anteriores, pero hemos elegido representar atributos que puedan contener valores internos de los objetos. Así, estamos

diciendo que todo objeto de la clase `Proyecto` va a tener un `nombre` y todo objeto `Persona` un `nombre` y `edad`.

A los atributos puede agregárseles un tipo, pero nos parece que eso no corresponde en el nivel de análisis, sino que es más una incumbencia de diseño. De la misma manera, mientras nos mantengamos en el nivel del análisis, no usaremos en el mismo diagrama simultáneamente los compartimientos de los atributos y de los métodos.

También se pueden representar atributos y estado en los diagramas de objetos. La figura 5.23 muestra esta posibilidad.

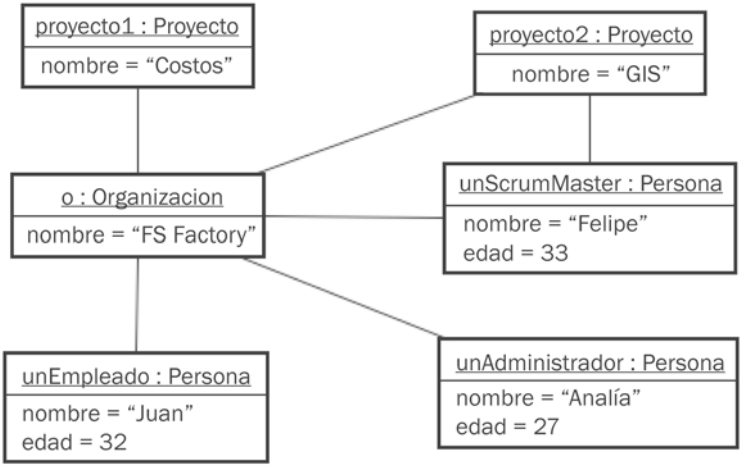

Figura 5.23 Objetos con atributos.

Lo que vemos en la figura 5.23 es una posible representación de objetos instancias de las clases de la figura 5.22. Allí observamos distintos objetos en un determinado momento del tiempo, con sus vinculaciones y los valores que están tomando sus atributos.

Una discusión que puede surgir en este contexto es cuáles conceptos deberían modelarse como atributos y cuáles como clases. Por ejemplo, si un *Sprint* tiene una duración, ¿es ésta una clase asociada a *Sprint* o más bien un atributo de las instancias de *Sprint*? Si bien no hay una regla fija, y en muchos casos depende del buen criterio del analista, una buena regla es preguntarse por el tipo de los datos en cuestión. Si el tipo del dato es un tipo simple o no propio del dominio del problema, como un número, una fecha, un texto, un período o un domicilio, podemos modelarlos como atributos. En ese sentido, la duración es más un atributo que una clase en sí misma. No obstante, como hemos dicho, depende mucho del criterio del analista, y el caso de la duración del *Sprint* está un poco en el límite de ese criterio.

DEL ANÁLISIS AL DISEÑO

Este capítulo trató de las herramientas de modelado de UML para realizar la actividad de análisis.

Sin embargo, puede que muchos estén en desacuerdo en cuanto al nivel de detalle que le hemos dado a los diagramas en este capítulo, diciendo que parecen diagramas de diseño. También puede que haya personas que no vean con claridad por qué el diagrama de estados apareció aquí y no en la actividad de requisitos. Todo puede ser.

Lo que ocurre es que la actividad de análisis es cada vez menos autónoma en el moderno desarrollo de software. Hay quienes realizan el análisis junto con los requisitos y hay quienes lo hacen junto con el diseño. E, incluso, quienes pasan directamente de los requisitos al diseño, sin nada de análisis.

Eso es lo que pasa, en definitiva. Muchos de los modelos que vimos pueden usarse en la actividad de diseño, y varios en la de requisitos. Por ejemplo, pueden usarse para requisitos los diagramas de:

- Estados, para modelado de comportamiento de dominio.
- Comunicación, para modelado de comportamiento de objetos u otros participantes en un caso de uso.
- Secuencia, para modelado de comportamiento de objetos u otros participantes en un caso de uso.
- Clases con atributos, para modelado estructural de dominio.

Y pueden usarse en el diseño los diagramas de:

- Clases con métodos, para modelado estructural de clases de diseño.
- Comunicación, con mayor detalle, para modelado de comportamiento en diseño.
- Secuencia, con mayor detalle, para modelado de comportamiento en diseño.

De allí que todos los diagramas que vimos en este capítulo van a volver a aparecer en el de diseño detallado.

Sin embargo, la separación del análisis en este capítulo nos permitió ver mejor, aunque con algunas sutilezas y cuestionamientos, la diferencia con el diseño, más cercano a la implementación, y que abordaremos en los próximos dos capítulos.

¿Y por qué hemos separado análisis y requisitos? Lo cierto es que ambas actividades están muy relacionadas e, incluso, se encuentran iterativamente acopladas. Ahora bien, en cuanto al modelado, es importante separarlas, ya que los requisitos se modelan con el lenguaje del cliente, a modo de contrato entre cliente y desarrollador,

mientras que el análisis es una vista del desarrollador, con su propio lenguaje, y a medio camino hacia el diseño. Mientras los requisitos brindan una vista externa del sistema, el análisis habla de cuestiones internas.

Por esta razón, mientras los requisitos están estructurados sobre la base de casos de uso u otras herramientas de modelización de requisitos, el análisis se estructura mediante clases. En cuanto al comportamiento también hay diferencias: los requisitos pueden modelarse con actividades y, a lo sumo, con estados; el análisis, en cambio, lo modelamos con diagramas de interacción, y también de estados.

6

MODELADO DEL DISEÑO DE ALTO NIVEL

MODELADO DE LAS PARTES LÓGICAS DE UN SISTEMA

Diseño lógico de alto nivel

En las aplicaciones medianas y grandes, el código fuente se suele agrupar por cuestiones de organización y flexibilidad. Esto facilita el mantenimiento y la reutilización, entre otras cuestiones no menos importantes. Así, por ejemplo, en Java existe la construcción *package*, que agrupa clases, y lo mismo ocurre con la construcción *namespace* de C# y C++. En otros lenguajes, no hay un mecanismo tan directo, pero siempre hay forma de agrupar de manera lógica el código fuente.

La agrupación lógica no tiene una semántica muy relevante en la orientación a objetos. Sin embargo, nos sirve para organizar el código cuando una aplicación deja de ser trivial. Por ejemplo, muchos especialistas han puesto énfasis en utilizar el criterio de separación de incumbencias para factorizar el código fuente en partes. Así se llega a los clásicos patrones de diseño macro o de alto nivel, tales como los patrones en capas, *Model-View-Controller*, etc.

UML nos da algunas facilidades para modelar la separación lógica del código fuente, y la más relevante es el diagrama de paquetes. También el diagrama de componentes nos otorga una facilidad en este sentido.

Diagramas de paquetes

El **diagrama de paquetes** de UML es una herramienta que sirve para agrupar elementos estáticos y es, por definición, un elemento estructural.

Cuando decimos que sirve para agrupar elementos estáticos estamos englobando varios tipos de diagramas. Por ejemplo, un paquete podría agrupar clases, pero también objetos o casos de uso e, incluso, otros paquetes.

En este capítulo, analizaremos los paquetes desde una vista externa, sin preocuparnos demasiado por sus cuestiones internas.

Veamos. Si la aplicación *FollowScrum* se desarrollará siguiendo el patrón de arquitectura de tres capas, un diagrama de paquetes podría ser el de la figura 6.1.

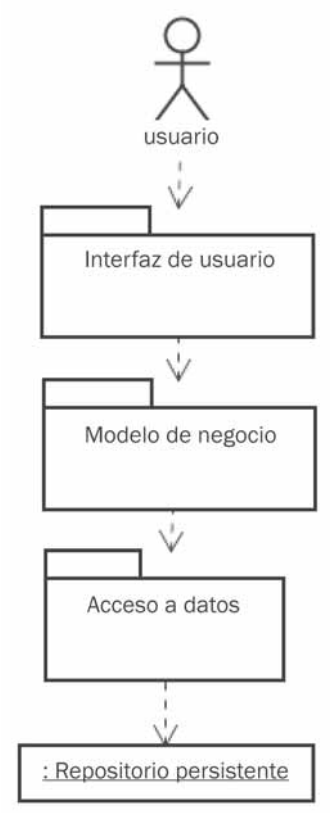

Figura 6.1 Diagrama de paquetes para una arquitectura de tres capas.

Notemos los siguientes elementos básicos del diagrama de paquetes:

- Un paquete se representa como una carpeta con solapa. La solapa se puede utilizar para poner el nombre del paquete, aunque esto sólo se suele hacer en casos en los que se necesite escribir otras cosas dentro de la carpeta, como hicimos en la figura 6.2.
- Las dependencias entre paquetes se muestran como las dependencias simples del diagrama de clases.

Se dice que hay una dependencia entre paquetes cuando una modificación en un paquete puede provocar una modificación en el dependiente. El ejemplo del patrón de tres capas es bastante ilustrativo: la capa de interfaz de usuario depende de la capa lógica o de modelo, ésta depende de la de acceso a datos, y esta última del repositorio de datos persistentes.

Hay algunos elementos en el diagrama anterior que no son del todo parte de UML estándar. El símbolo que usamos para representar el repositorio persistente es el de un objeto, que no suelen representarse en diagramas de paquetes pero son bastante adecuados en diagramas de arquitectura como éste. Tal vez sería más ortodoxo usar un artefacto, elemento que veremos luego. Respecto del usuario, lo hemos representado con el mismo esquema de persona que se usa para los actores en los diagramas de casos de uso. Esto es bastante usual, pero no del todo estándar.

Por supuesto, como decíamos más arriba, un paquete puede contener paquetes internos. En este caso, simplemente se dibuja el paquete interno dentro del externo y se lleva el nombre del más externo a la solapa. Es lo hemos hecho en la figura 6.2, que muestra, con un poco más de detalle, la estructura lógica del sistema *FollowScrum*.

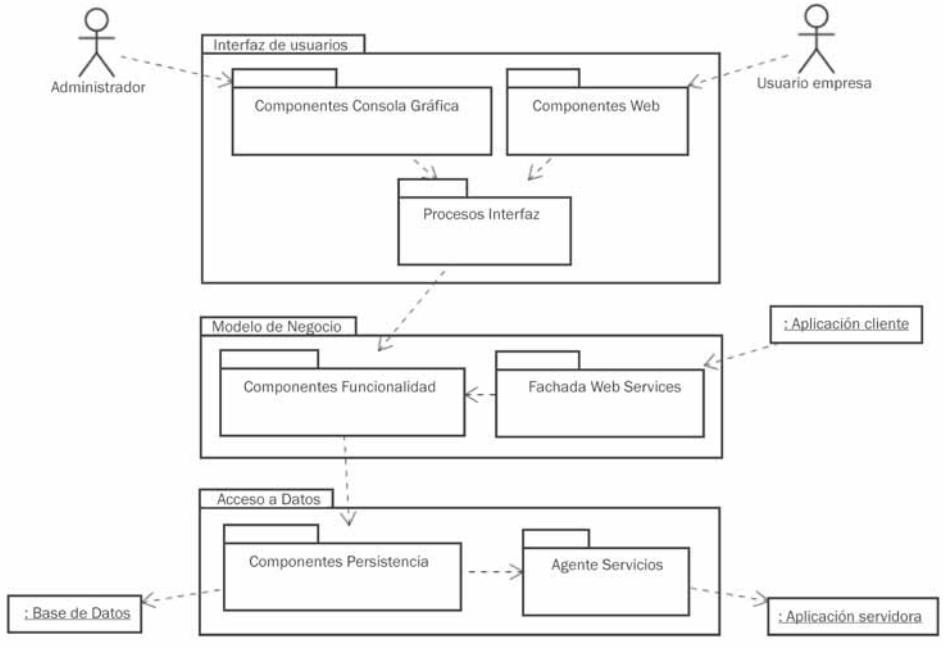

Figura 6.2 Diagrama con paquetes internos.

También se puede indicar, en un diagrama de paquetes, el uso de bibliotecas externas y *frameworks*, con los estereotipos respectivos. La figura 6.3 ilustra un caso más detallado de la figura 6.2, donde mostramos que para la persistencia estaremos usando el *framework Hibernate*, para servicios Web el *framework Axis*, para compo-

nentes Web el *framework Struts* y para componentes de consola gráfica, la biblioteca *Swing*.

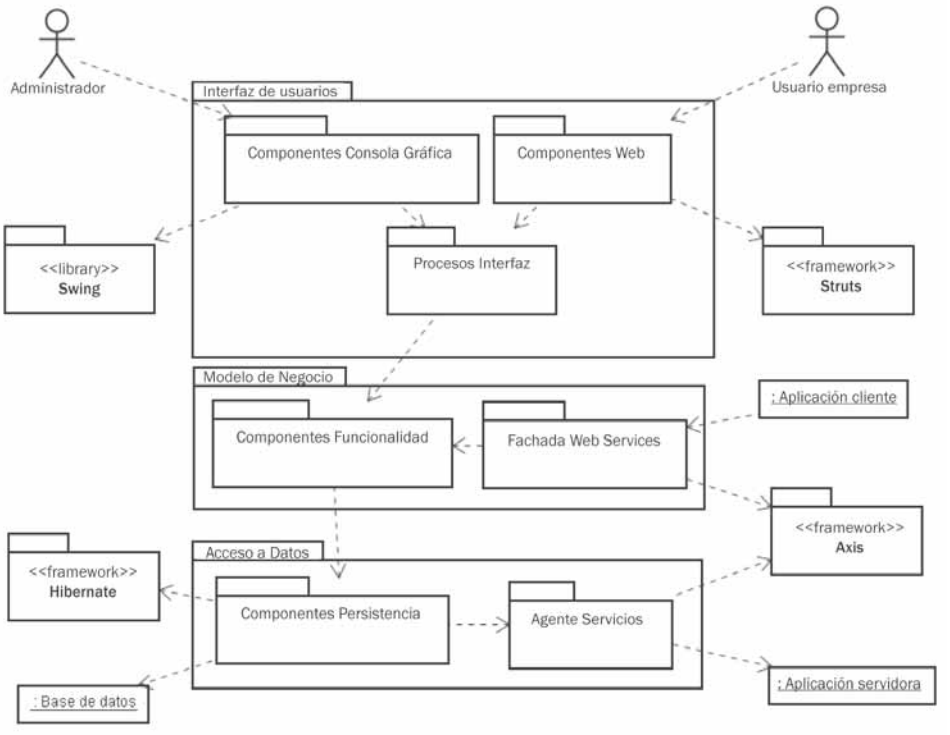

Figura 6.3 Diagrama con *frameworks* y bibliotecas.

La diferencia entre biblioteca y *framework* es muy sutil para muchos profesionales. Sin embargo, una biblioteca brinda servicios a través de clases y funciones listas para usar, mientras que un *framework* brinda servicios a través de comportamientos por defecto que se pueden cambiar o ampliar mediante clases provistas por el programador.

Dicho de otra manera, una biblioteca está lista para ser usada mediante la invocación de sus servicios, mientras que un *framework* es un programa que invoca a los comportamientos definidos por el programador.

Hay mucho más que se puede avanzar con diagramas de paquetes. Entre ellos, hay distintos estereotipos estándar para las dependencias, aunque rara vez se usan. Y también tenemos la posibilidad de agrupar otros elementos, de los cuales se utilizan sobre todo las clases, como veremos en el próximo capítulo.

Por el momento, vamos a quedarnos con la belleza de lo simple.

Diagramas de componentes

La separación en partes del software que hicimos con diagramas de paquetes es más bien una separación lógica, que hace a la organización de nuestro código fuente.

Si lo que necesitamos representar es cómo está organizada la aplicación, pero a nivel de componentes que se pueden adquirir y enchufar, reemplazables y reutilizables, podemos usar diagramas de componentes. Los **componentes** se suponen partes reemplazables del sistema, que se pueden adquirir y versionar.

También puede modelarse una biblioteca para uso externo, mediante un componente.

La representación de un componente ha variado levemente de UML 1 a UML 2. En la versión antigua, un componente se representaba como en la figura 6.4, mientras que la figura 6.5 muestra el mismo componente en la versión actual, con dos posibilidades. Poca gente hace esta distinción entre versiones de UML, y usa más bien la notación que le provee la herramienta de modelado que utiliza.

Figura 6.4 Componente en UML 1.x.

Figura 6.5 Componente en UML 2.x.

Los **diagramas de componentes** de UML muestran componentes y sus comunicaciones mediante **interfaces**.

La representación de interfaces puede ser la de una clase con el estereotipo `<<interface>>`, en la que se muestren las propiedades, observables desde afuera, que exponga o requiera un componente. No obstante, si bien en los diagramas de clases esto es lo más común, lo más habitual en los diagramas de componentes es usar la notación de un círculo unido por una línea al componente.

La figura 6.6 muestra tres componentes de *FollowScrum*, uno que autentica usuarios y sistemas externos, y los otros dos que lo utilizan. El vínculo entre un componente y una interfaz es una dependencia, ya que un componente depende de una interfaz.

Obsérvese en la figura 6.6 la notación para modelar interfaces. En realidad, hay dos elementos en una misma interfaz: la noción de **interfaces requeridas**, que en este caso son, por ejemplo, las que necesitan los dos componentes que dependen de cada una de las interfaces, y la de la **interfaces proporcionadas**, que son las que expone el componente `Autenticador`. La figura 6.7 resalta esta doble visión sobre el mismo diagrama anterior, con la notación de chupetín o pirulí para la interfaz proporcionada y de enchufe para la interfaz requerida.

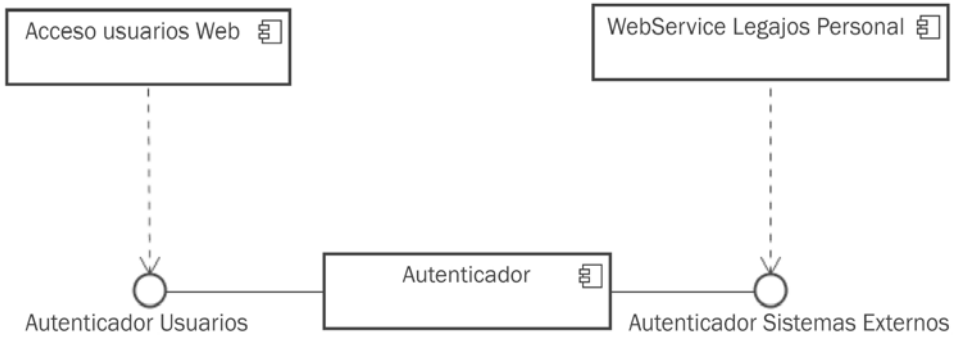

Figura 6.6 Componentes e interfaces.

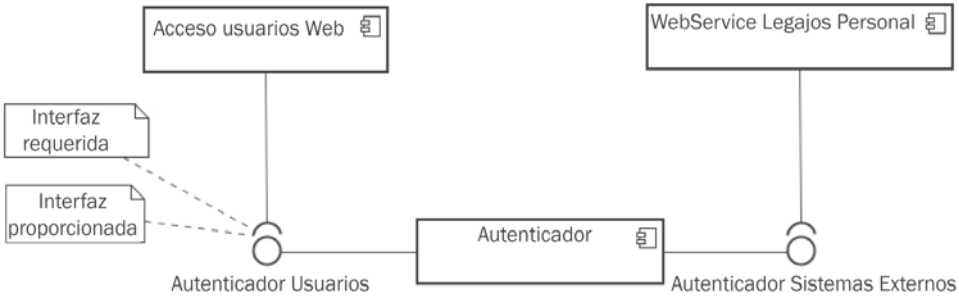

Figura 6.7 Interfaz requerida e interfaz proporcionada.

Hay ocasiones en las que se desea mostrar que un componente está compuesto por otros y, para ello, usamos la notación habitual de composición, como muestra la figura 6.8. También podemos –y es más habitual– mostrar un componente con componentes internos, como en la figura 6.9.

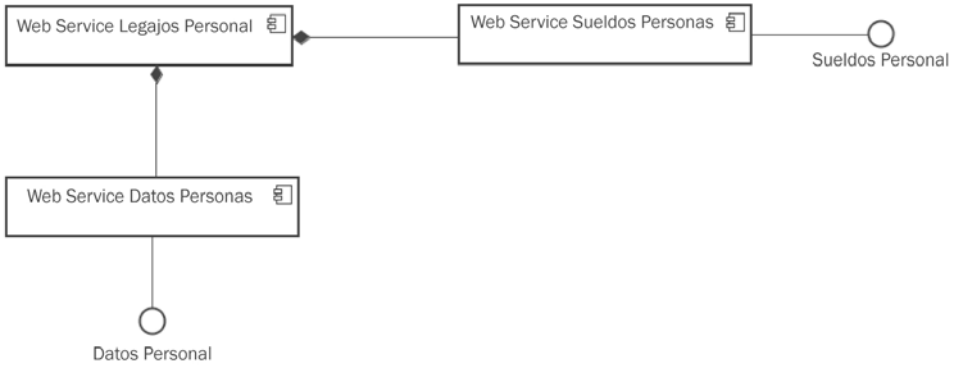

Figura 6.8 Componente compuesto.

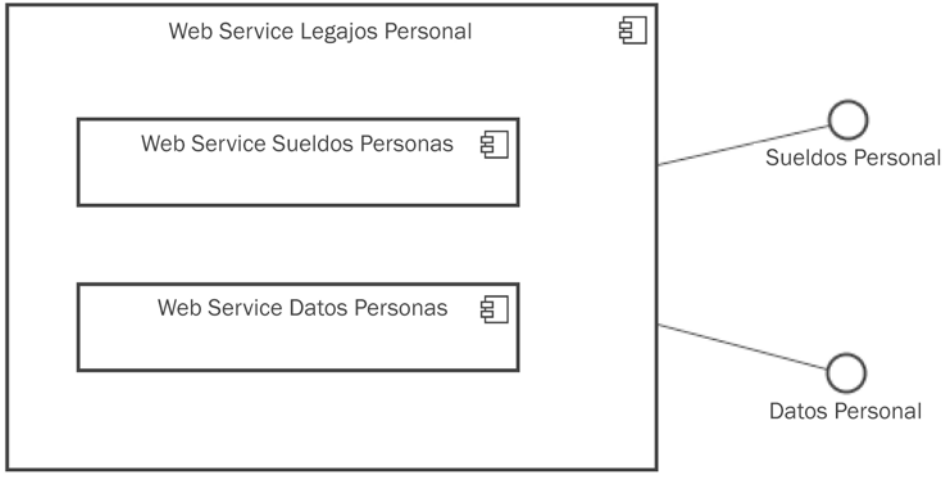

Figura 6.9 Componente con otros componentes internos.

Los diagramas de componentes pueden contener más elementos, como muestra el más complejo de la figura 6.10, que también se suele denominar **diagrama de estructura compuesta**, y que, en realidad, es otro tipo de diagrama UML, que veremos en el capítulo 7.

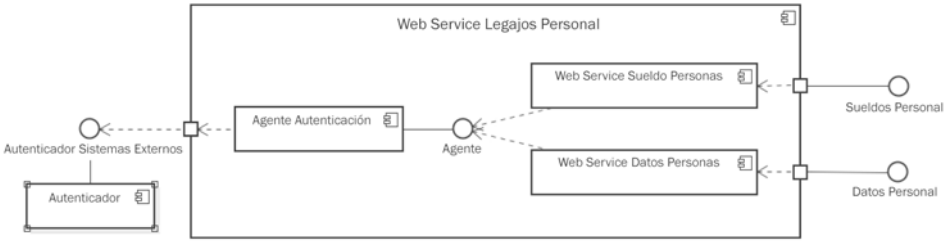

Figura 6.10 Diagrama de componentes complejo.

En la figura, vemos la aparición de los **puertos**, esos pequeños cuadrados en el borde del componente que sirven para comunicarlo con el exterior.

Un puerto en el diagrama de componentes sirve cuando se le quiere dar un nombre específico a la manera en que se comunica con una o con un conjunto de interfaces. También sirve para indicar el sentido de la dependencia, como hicimos, en nuestro caso, con las flechas que salen o entran en los puertos, y que muestran cuando un componente depende de un puerto, o viceversa. De todas maneras, es dudosa la utilidad real de representar puertos, sobre todo si no se necesita darles un nombre, como en nuestro caso.

El hecho de que en nuestro diagrama hubiese una sola interfaz asociada a cada puerto no implica que siempre deba ser así. Hay ocasiones en las que un puerto puede implementar varias interfaces, requerir otras e, incluso, proveer otras más.

En realidad, un componente es un objeto, que a su vez podría ser instancia de alguna clase o tipo. Cuando quiera aclararse esto en un diagrama de componentes, puede indicarse el nombre del componente mediante la notación `NombreComponente : NombreTipo`.

MODELADO FÍSICO DEL SISTEMA

Artefactos

Los **artefactos**, en UML, son partes físicas del sistema de software. Pueden ser archivos fuente, archivos *jar* o *war* que usamos en una aplicación Java, bibliotecas de enlace dinámico (DLL) de Windows, *assemblies* de .NET, programas ejecutables, etc.

Son "físicos", no en el sentido de algo tangible, como el hardware, sino en el sentido de que se pueden desplegar en hardware.

Un artefacto, como muestra la figura 6.11, se representa con un rectángulo que tiene el estereotipo `<<artifact>>`. También se pueden representar dependencias entre artefactos.

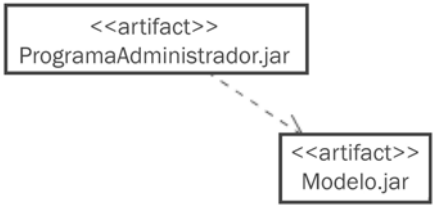

Figura 6.11 Artefactos y dependencias.

Asimismo se le pueden agregar a los artefactos algunas propiedades de despliegue, aunque no es demasiado usual, salvo en el contexto de los diagramas de despliegue que veremos a continuación.

Hay distintos tipos de artefactos definidos en UML, que pueden indicarse usando estereotipos especializados. En el cuadro 6.1, se muestran los tipos de archivos físicos, todos subtipos de `<<file>>`.

ESTEREOTIPO	SIGNIFICADO
<<file>>	Cualquier tipo de archivo físico.
<<document>>	Cualquier archivo no ejecutable. Es un caso particular de de <<file>>.
<<executable>>	Cualquier archivo ejecutable. Es un caso particular de de <<file>>.
<<source>>	Archivo de código fuente que puede ser traducido a un ejecutable. Es un caso particular de de <<document>>.

ESTEREOTIPO	SIGNIFICADO
<<script>>	Archivo de guiones que puede ser ejecutado en un ambiente determinado. Es un caso particular de de <<executable>>.
<<library>>	Archivo que contiene una biblioteca ejecutable de enlace dinámico o estático. Es un caso particular de de <<executable>>.

Cuadro 6.1 Tipos de artefactos.

Diagramas de despliegue

No existen sistemas de software que no necesiten hardware para ejecutarse. Como la decisión de cómo realizar la distribución de una aplicación en hardware es una decisión de diseño, la hemos abordado en este capítulo. UML nos permite representar **nodos de hardware** en los cuales podemos desplegar artefactos en los **diagramas de despliegue**.

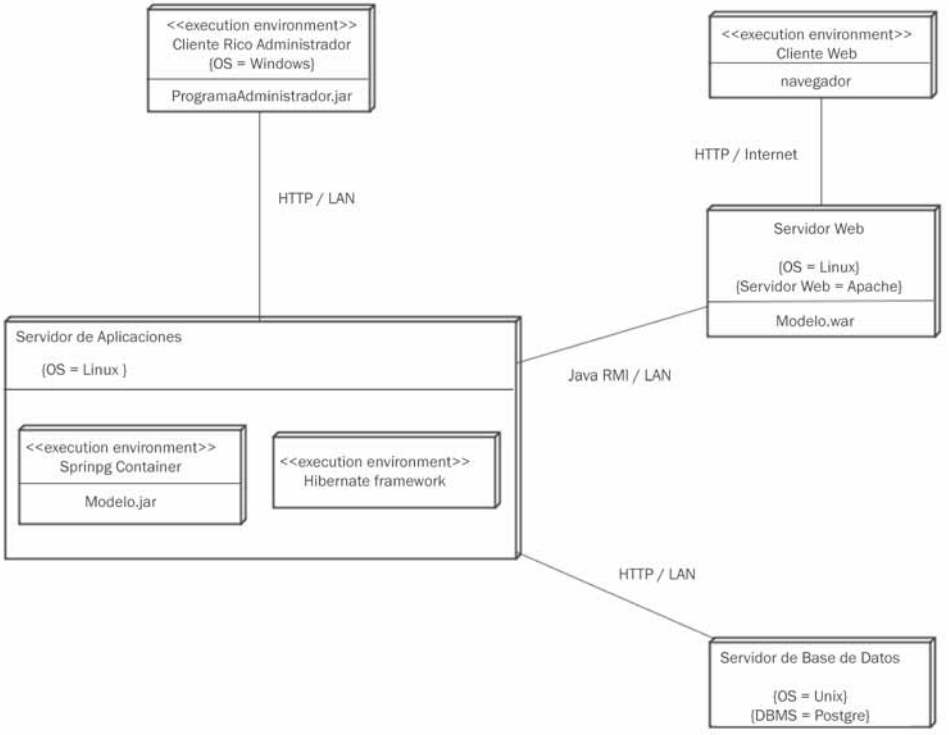

Figura 6.12 Diagrama de despliegue.

De la misma manera, como a veces debemos desplegar artefactos en servidores de aplicaciones u otros contenedores de software, también se los modela mediante nodos, en este caso, de software, que denominamos **entornos de ejecución**.

Los distintos tipos de nodos pueden denotarse con estereotipos.

En la figura 6.12, se muestra el diagrama de despliegue de la aplicación *FollowScrum*. La misma es bastante explicativa: hay nodos, representados como cubos, y vínculos de comunicación entre ellos. A los vínculos se les puede colocar una leyenda que indique el tipo de vínculo o el protocolo de comunicación. Cada nodo puede tener uno o más artefactos. Los artefactos pueden dibujarse con su propia notación, dentro del nodo, o quedar simplemente enumerados, como hicimos en este caso (por ejemplo, con `ProgramaAdministrador.jar` en `Cliente Rico Administrador`).

Hay algunas confusiones bastante generalizadas en el uso de diagramas de despliegue. Muchas herramientas –y, por lo tanto, los profesionales que las usan– permiten usar artefactos en el marco de diagramas de componentes y no en los de despliegue. Al mismo tiempo, mucha gente marca los artefactos de los diagramas de despliegue como componentes. Esto es una derivación del significado de componente y de diagramas de despliegue de UML 1.

El diagrama de despliegue es una herramienta interesante y relativamente simple. Como casi todas las construcciones de UML, se puede hacer más preciso y, por consiguiente, más complejo. A modo de ejemplos, se pueden especificar mejor las relaciones entre nodos, como asociaciones con navegabilidad o sin ella, dependencias o generalizaciones. En este libro, hemos evitado estas cuestiones, para cumplir con nuestras premisas básicas de simplicidad y practicidad.

DISEÑO MACRO Y UML

Como hemos visto, UML nos da distintas herramientas para encarar el diseño de alto nivel. Tal vez parezcan herramientas un tanto disímiles, así que vamos a detenernos un poco en ello.

El diagrama de despliegue no debiera ofrecer mayores dificultades: muestra cómo distintos artefactos físicos de software resultan desplegados o instalados en distintos nodos, sean estos elementos de hardware o contenedores de software. Es un diagrama interesante cuando el despliegue no es trivial.

El diagrama de componentes tal vez sea el que menor uso tiene, en general, porque mucha gente lo reemplaza por diagramas de paquetes más simples. Sin embargo, es de utilidad si se quieren mostrar componentes intercambiables y sus interfaces. Tal vez su mejor uso se dé en el marco de aplicaciones basadas en arquitecturas orientadas a servicios.

El diagrama de paquetes, como lo vimos hasta ahora, es una buena herramienta para la comunicación de la arquitectura del sistema de una manera simple, mostrando asimismo las dependencias entre las partes.

El capítulo que sigue nos ilustrará los aspectos más detallados del diseño, más cercanos a la programación. No veremos ningún tipo de diagrama nuevo: nos encontraremos con los de clases, de paquetes, de estados y de interacción. Pero, dado que el modelo de diseño debe proveer siempre orientación para la construcción, los veremos en un grado mucho más detallado.

7

MODELADO DEL DISEÑO DETALLADO Y CONSTRUCCIÓN

MODELADO DE COMPORTAMIENTO DETALLADO

Diagramas de estados

Los diagramas de estado, así como se pueden usar en el análisis, e incluso en los requisitos, pueden emplearse también para el diseño. Sin embargo, no hay construcciones nuevas que nos interesen sobre las que ya vimos, salvo el concepto de **pseudo-estado de historia**. Este tipo particular de estado muestra que un estado compuesto puede "recordar" el estado activo antes de salir por última vez de la máquina de estados.

La figura 7.1 muestra el diagrama de estados correspondiente a la cancelación de un Sprint sin terminarlo, que luego de una reunión debe retomarse. Nótese que la representación de este pseudo-estado se hace con un círculo que contiene una `H`.

Los pseudo-estados de historia no tienen por qué estar a la entrada de un estado compuesto, como en la figura 7.1. Puede haber pseudo-estados históricos internos, en cuyo caso en vez de colocarse una `H` se emplea un símbolo `H*`.

Hay muchos otros pseudo-estados que se pueden representar. Por ejemplo, en UML 2, se pueden modelar **pseudo-estados de entrada** para indicar inicios repetitivos del ciclo de estados. También existen los **pseudo-estados de salida** para indicar interrupciones del ciclo de estados. Ambos son estados internos. Sin embargo, todos estos pseudo-estados son de escasa utilización en general, salvo que se haga un uso muy exhaustivo de los diagramas de estados.

En los diagramas de estados se pueden también representar acciones con rectángulos de bordes rectos. Incluso, se pueden modelar acciones de envío y recepción de señales, con las mismas notaciones que en los diagramas de actividades. En este libro, siguiendo con la premisa de mantener la simplicidad y los usos comunes, no hemos incluido ninguna de estas características. Asimismo, pueden dividirse en regiones, que es lo que hicimos para representar en estados concurrentes en la figura 5.14.

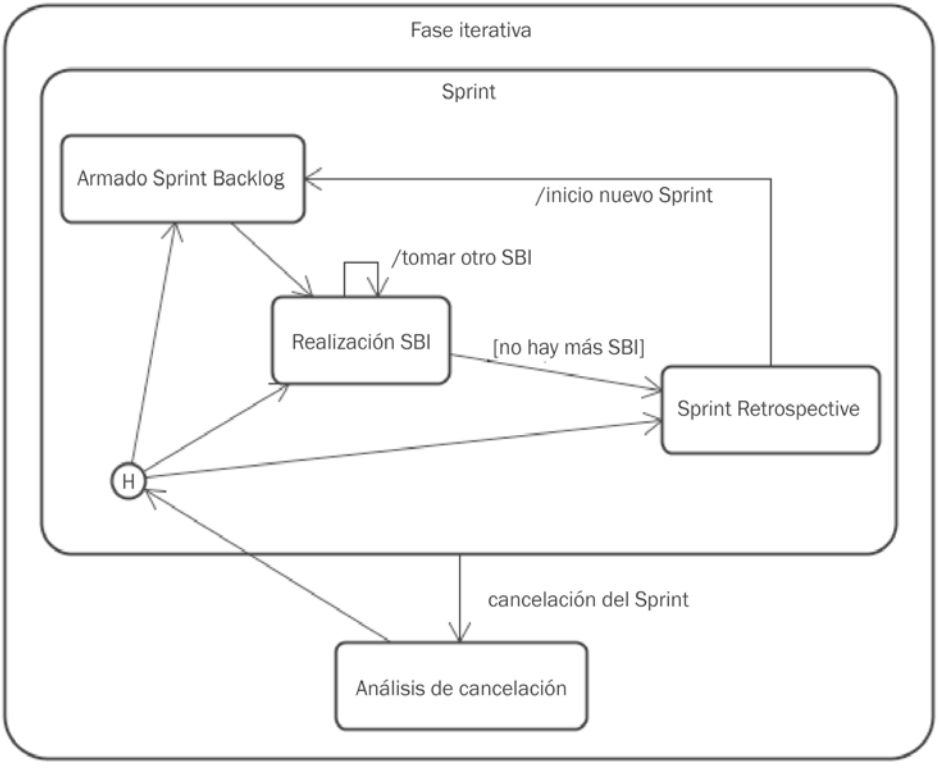

Figura 7.1 Diagrama de estados con historia.

Diagramas de secuencia

Así como –entre los dos diagramas de interacción más comunes– el de comunicación nos pareció más interesante para modelar el análisis, resulta ser más cómodo y habitual modelar el diseño con el diagrama de secuencia. Esto ocurre porque, al estar hablando de diseño, el paso del tiempo adquiere mayor relevancia que en el análisis.

Ahora bien, en esta actividad, el diagrama de secuencia debería ser usado para mostrar interacciones entre objetos de diseño. La figura 7.2 muestra un escenario de diseño (simplificado) en el cual un sistema externo va a consultar el *backlog* de un proyecto de *FollowScrum* a partir del uso de un servicio Web.

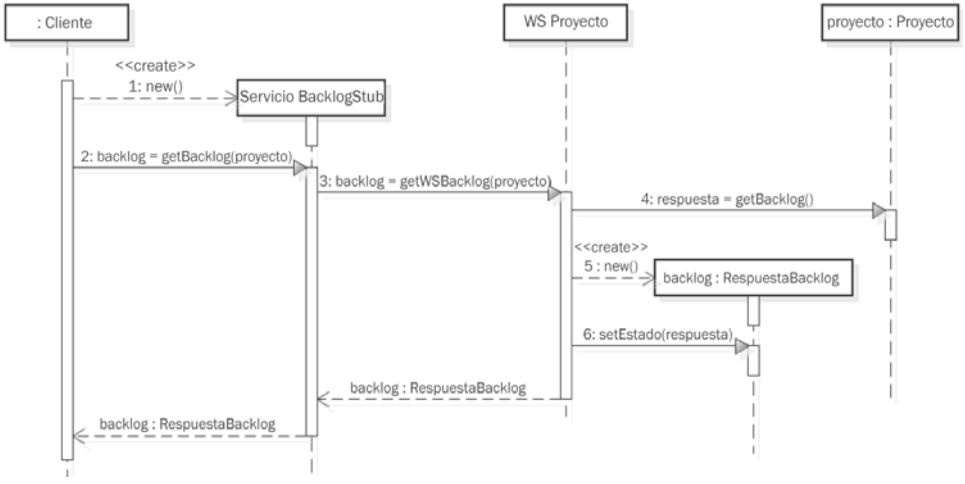

Figura 7.2 Diagrama de secuencia de diseño.

Como se ve, no hay muchas novedades respecto de los diagramas de secuencia que ya conocemos. Lo único que hicimos fue agregar los parámetros de los métodos, cosa que hasta ahora habíamos evitado, y usar una notación más similar a la del lenguaje de programación, Java en nuestro caso.

Tal vez sea interesante revisar el significado del rectángulo a lo largo de la línea de vida, que dijimos que indicaba que el objeto estaba activo. En términos más cercanos a la implementación, podemos agregar que es un indicador de la duración de la ejecución de un método. Así, por ejemplo, en la figura 7.2, el método `getWSBacklog` tiene su rectángulo de activación, en la línea de vida de `WSProyecto`, que termina recién cuando se finaliza la ejecución del método `setEstado` llamado desde aquél.

Notemos que, en el diagrama de la figura 7.2, nos hemos detenido en unas cuantas cuestiones que apuntan a la construcción. En un diagrama de análisis no nos preocuparía –y, de hecho, no figuraría– un objeto como el `ServicioBacklogStub`.

Hay ocasiones en las que no nos interesa modelar el participante que envía el mensaje disparador de un diagrama de secuencia, sea porque es irrelevante para el modelo o porque no conocemos exactamente de qué objeto se trata. Esto es habitual cuando una aplicación brinda servicios que pueden invocarse desde contextos muy variados. UML llama a estos mensajes, **mensajes encontrados**, y los representa como provenientes de un punto sin nombre. En la figura 7.3 mostramos esta posibilidad.

También hay ocasiones en las que no nos interesa modelar quién recibe cierto mensaje. Esto último, un poco menos habitual, se podría usar para modelar un mensaje enviado a cualquier objeto que esté disponible para recibirlo. En términos de UML, decimos que estamos en presencia de un **mensaje perdido**, que se modela como en la figura 7.4.

En teoría, un diagrama de secuencia permite diagramar cualquier algoritmo. Por eso, muchos profesionales caen en la tentación de usarlos para modelar código fuente. La figura 7.5 nos muestra el diagrama de secuencia de una búsqueda en un arreglo de `tareas` contenidos en la clase `SprintBacklog`.

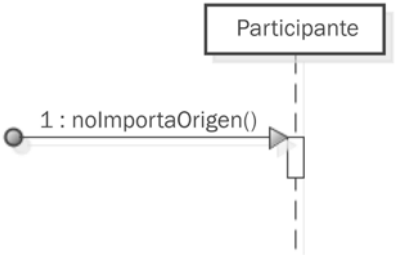

Figura 7.3 Mensaje encontrado.

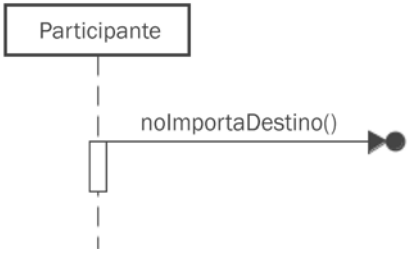

Figura 7.4 Mensaje perdido.

Ahora bien, que sea posible modelar código fuente no quiere decir que sea conveniente. Analicémoslo.

En la figura 7.6 mostramos el código fuente en Java que está modelando este diagrama. Para ver la pertinencia de hacer el modelo, hagámonos un par de preguntas:

- ¿Es conveniente hacer el diagrama de secuencia de la figura 7.5 antes del código de la figura 7.6, como modelo de diseño a probar o discutir? La respuesta sería positiva solamente si hacer el diagrama nos permitiese ver la interacción entre objetos con mayor claridad, y si el costo de hacer el diagrama fuese más bajo que el de escribir el propio código. En cualquier otro caso, diríamos que no.

- ¿Es conveniente hacer el diagrama de secuencia de la figura 7.5 después del código, como documentación? De nuevo, en este caso, la respuesta sería que la conveniencia o no depende de la legibilidad de uno y de otro.

Cada uno de los lectores es libre de sacar sus propias conclusiones.

Nuestra conclusión personal es que, en general, no se justifica, ya que el código fuente es también un modelo, y no tiene sentido repetirlo tanto en forma textual como gráfica. Hay quienes modelan con estos diagramas solamente el código fuente más intrincado o incomprensible. No obstante, preferimos mejorar la calidad del código antes de modelarlo con diagramas de secuencia detallados.

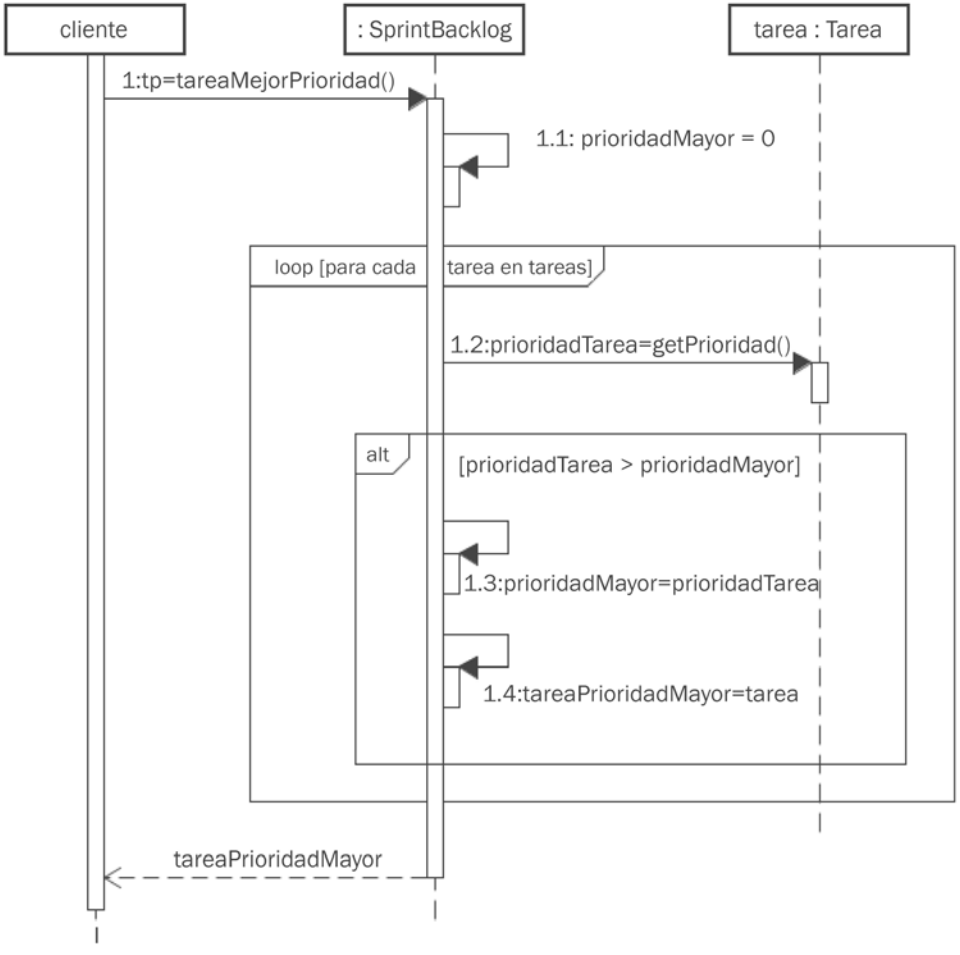

Figura 7.5 Diagrama de secuencia de un algoritmo de búsqueda.

```
package followscrum.modelo;

public class SprintBacklog {

  private Iterable<Tarea> tareas;
  ...

  private Tarea tareaMejorPrioridad () {
    int prioridadMayor = 0;
    Tarea tareaPrioridadMayor = null;
    for (Tarea tarea : tareas) {
      int prioridadTarea = tarea.getPrioridad();
      if (prioridadTarea > prioridadMayor) {
        prioridadMayor = prioridadTarea;
        tareaPrioridadMayor = tarea;
      }
    }
    return tareaPrioridadMayor;
  }
}
```

Diagramas de secuencia y tiempos

El diagrama de secuencia, tal como lo hemos venido estudiando, muestra tiempos solamente en el sentido de indicarnos el orden en que ocurren los envíos de mensajes.

Sin embargo, UML admite mostrar restricciones temporales específicas, que indican cuánto tiempo debe pasar entre un mensaje y otro, o el momento exacto en que debe ocurrir un evento.

Por ejemplo, el diagrama de la figura 7.6 ilustra algunas de estas posibilidades.

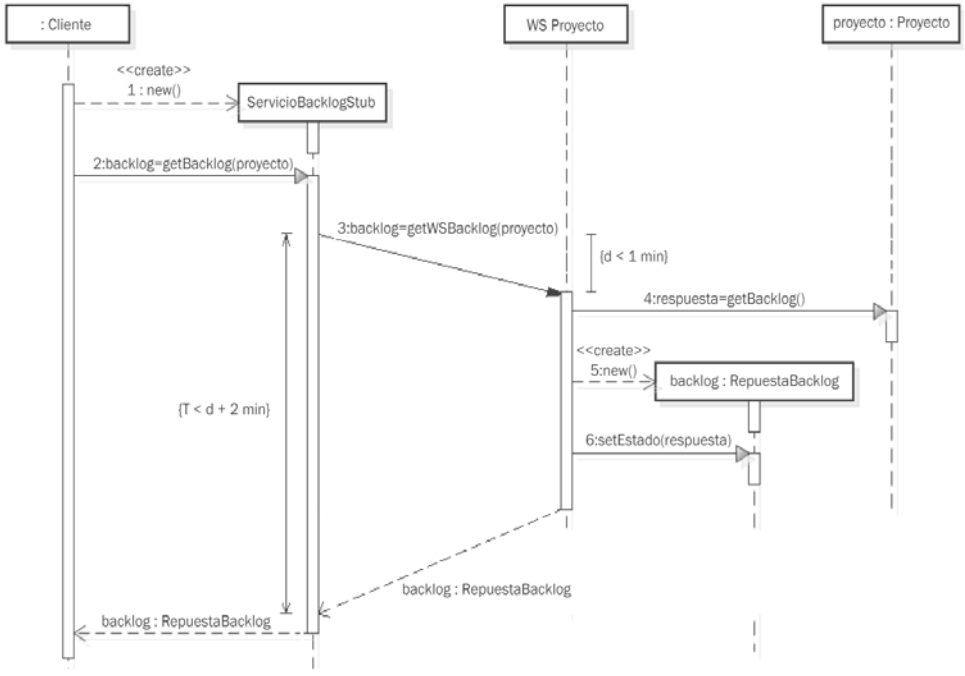

Figura 7.6 Diagrama de secuencia con restricciones temporales.

Notemos que, en la figura 7.6, figuran envíos de mensajes que no están dibujados con una flecha horizontal, como veníamos haciendo hasta ahora. Estos mensajes de flecha oblicua indican que el envío de los mismos lleva un tiempo significativo, que incluso puede tener una restricción o duración especificada.

Las duraciones que aquí se muestran son dos. Por un lado, se nos dice que la llamada del método `getWSBacklog(proyecto)` no puede demorar una duración `d` mayor a un minuto: esto se indicó con el texto `{ d < 1 min }`. Por otro, muestra también que, entre la llamada al mismo método y su retorno, no pueden pasar más de dos minutos adicionales a la duración `d`, como se lee en el texto `{ T < d + 2 min }`.

UML, desde su versión 2, nos provee otro diagrama de interacción que hace foco en el paso del tiempo y en las restricciones. Se llama diagrama de tiempos y lo veremos más adelante en este capítulo.

No obstante, la inexistencia del diagrama de tiempos en versiones previas de UML, más el hecho de que el diagrama de secuencia es considerado por muchos profesionales como el modelo de interacción por excelencia, hace que se utilice el diagrama de secuencia aun cuando se quiera modelar cuestiones temporales.

Diagramas de comunicación

Como con el diagrama de secuencia, el diagrama de comunicación tampoco ofrece muchas novedades al usarlo para el diseño detallado, más allá de estar orientado a la construcción y, por lo mismo, contener más detalles.

En la figura 7.7 se muestra el diagrama de comunicación semánticamente equivalente al de secuencia de la figura 7.2.

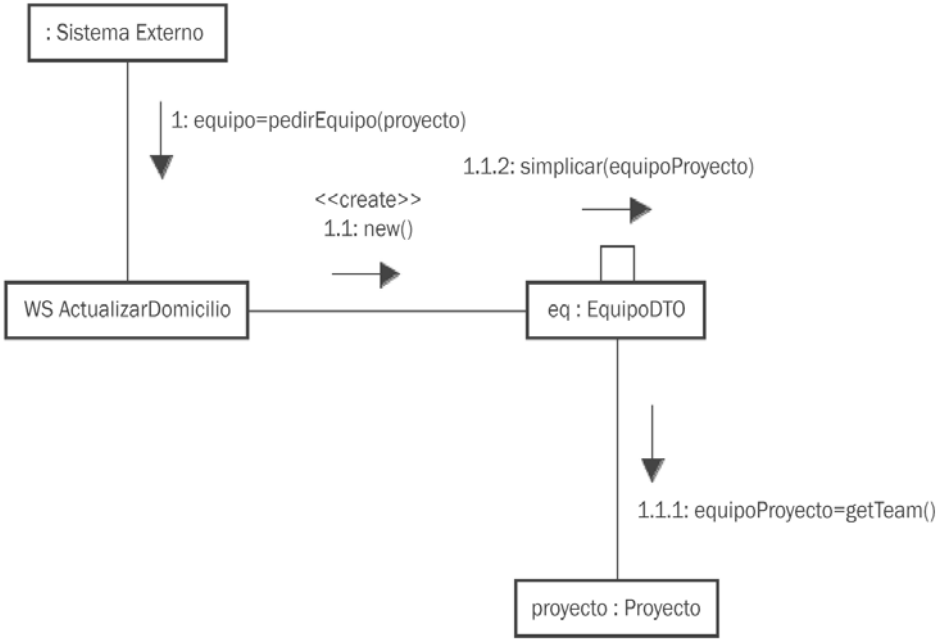

Figura 7.7 Diagrama de comunicación de diseño.

MODELADO ESTRUCTURAL DETALLADO

Diagramas de clases

La clase es una construcción de todos los lenguajes orientados a objetos. Esto hace que el diagrama de clases sea el diagrama estructural más importante a la hora de modelar diseño detallado y programación.

Ya hemos visto bastante de diagramas de clases. Hemos visto cómo representar asociaciones, agregación, composición, generalización, dependencia, atributos y

métodos. Algunas de las cuestiones ya vistas se usan también en el diseño detallado. Otras no tanto, como los nombres de asociaciones. Pero también aparecen nuevos significados y elementos que no hemos visto.

Como ya dijimos en el capítulo 1, se suele hablar de distintas perspectivas en diagramas UML: conceptual, de especificación, de implementación y de producto. Esta idea es la que hemos seguido a lo largo del libro, aunque supeditada a las disciplinas del desarrollo de software. La perspectiva conceptual la analizamos en el capítulo de requisitos, con el modelado conceptual con clases de dominio. La de especificación, la abordamos con los diagramas de clases de análisis y la de implementación, la enfrentaremos en este capítulo junto con el diseño detallado y la construcción.

Al ir avanzando de una perspectiva a otra, lo que vamos haciendo es agregar mayor información de implementación, y acercarnos al lenguaje de programación elegido para la construcción. Esto es así porque lo que se dibuje en un modelo de clases orientado al diseño detallado va a depender en gran medida del lenguaje de programación utilizado.

Esto es lo que iremos viendo en los ítems subsiguientes.

Elementos adicionales básicos en diagramas de clases

Hay algunas cuestiones de los diagramas de clases que, por haber estado trabajando en un nivel mayor de abstracción, no habíamos analizado todavía. Entre ellos, los tipos de métodos, parámetros y atributos y la visibilidad de atributos y métodos.

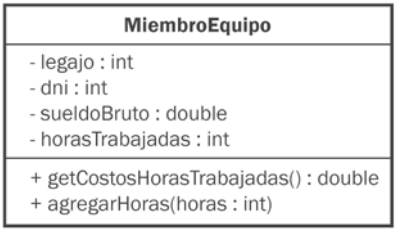

Figura 7.8 Tipos y visibilidad en un diagrama de clases.

Por ejemplo, observemos la figura 7.8. En ella, podemos destacar:

- Se puede indicar el tipo de un atributo. Por ejemplo, el atributo `legajo` es de tipo `int`, mientras que `sueldoBruto` es de tipo `double`. Esto sólo tiene sentido en lenguajes de programación en los que las variables se definan con tipos que luego no se pueden cambiar.
- Se puede especificar el tipo del valor que devuelve un método. Así lo hicimos con el método `getCostoHorasTrabajadas`, al decir que devuelve un valor de tipo `double`. El método `agregarHoras`, en cambio, como no le ponemos el tipo del valor devuelto, asumimos que no devuelve ningún valor; como estamos trabajando en Java, podríamos haber escrito `void`.

- También podemos indicar una lista de parámetros para los métodos, y colocarles los tipos. Así, el método `agregarHoras` tiene el parámetro `horas` de tipo `int`.
- Es posible indicar la visibilidad de un atributo o método con los calificadores de visibilidad +, -, # y ~ .

Los calificadores de visibilidad tienen los siguientes significados:

CALIFICADOR	SIGNIFICADO
+	Visibilidad pública. Habitualmente significa la misma visibilidad que la clase que contiene el elemento. Esto es: el elemento es visible para cualquier otro elemento para el que la clase sea visible.
-	Visibilidad privada. Habitualmente significa que el elemento sólo es visible dentro de la clase que lo contiene. En algunos lenguajes, como ocurre en Smalltalk, sólo es visible para el objeto en sí mismo, no para la clase.
#	Visibilidad protegida. Habitualmente significa que el elemento sólo es visible dentro de la clase que lo contiene y las clases que heredan de ella.
~	Visibilidad de paquete. Especialmente introducido en UML para representar la visibilidad de paquete de Java. Habitualmente significa que el elemento sólo es visible para las clases del mismo paquete de la clase que lo contiene.

No obstante, los distintos lenguajes pueden definir significados distintos para las visibilidades, y lo usual es que, en un diagrama de diseño, la visibilidad indique el mismo significado que en el lenguaje de programación en el que estemos trabajando. Efectivamente, el lector atento habrá observado que, en el cuadro anterior, usamos varias veces la expresión “habitualmente”, para no hacer foco en ninguna particularidad de los lenguajes de programación.

A modo de ejemplo, en Java, la visibilidad protegida permite también el acceso a los elementos del mismo paquete. En .NET no existe la visibilidad de paquete, pero sí la visibilidad a nivel de unidad de despliegue o *assembly*, de modo que muchos profesionales usan el símbolo de visibilidad de paquete para la visibilidad de *assembly* en .NET. Asimismo, hay lenguajes que no tienen ciertos tipos de visibilidades, o que la definen en función del tipo de elemento en cuestión. Por ejemplo, Smalltalk considera a todos los atributos como protegidos, mientras que, en Python, son siempre públicos.

Hay algunos lenguajes de programación en los que se puede indicar que ciertos parámetros son sólo de entrada, únicamente de salida o de entrada y salida. Así ocurre con los lenguajes que admiten pasaje por valor o por referencia, como ocurre en la plataforma .NET y en C++. Desde ya, esto no tiene sentido en Java o Smalltalk. Si se desea indicar estas circunstancias en diagramas de clases, se puede colocar alguno de los siguientes calificadores delante de los nombres de los parámetros:

CALIFICADOR	SIGNIFICADO
`in`	El método no puede modificar el parámetro, sólo lo recibe como dato.
`out`	El método no recibe ningún valor en el parámetro, sino que el mismo se carga dentro del método.
`in out`	El método recibe el parámetro como dato y puede cambiarlo para devolverlo modificado.

Hay lenguajes que necesitan que las instancias de las clases se creen con un método especial, habitualmente denominado **constructor**. Aun cuando así no fuera, hay métodos que tienen como misión principal la creación de una instancia. A estos métodos se los puede modelar con el agregado del estereotipo `<<create>>`.

La especialización-generalización suele estar implementada en los lenguajes orientados a objetos, mediante un mecanismo denominado **herencia**. Por ejemplo, tomemos el diagrama de la figura 7.9. En él, la clase `Usuario` **generaliza** a la clase `Administrador`, o lo que es lo mismo, la clase `Administrador` **especializa** a la clase `Usuario`. Entonces, decimos que la clase `Administrador` **hereda** de la clase `Usuario`. Hay otros términos también habituales: decimos que la clase `Administrador` **deriva** de la clase `Usuario`, o que **desciende** de ella. A la inversa, decimos que la clase `Usuario` es **ancestro** o **base** de la clase `Administrador`.

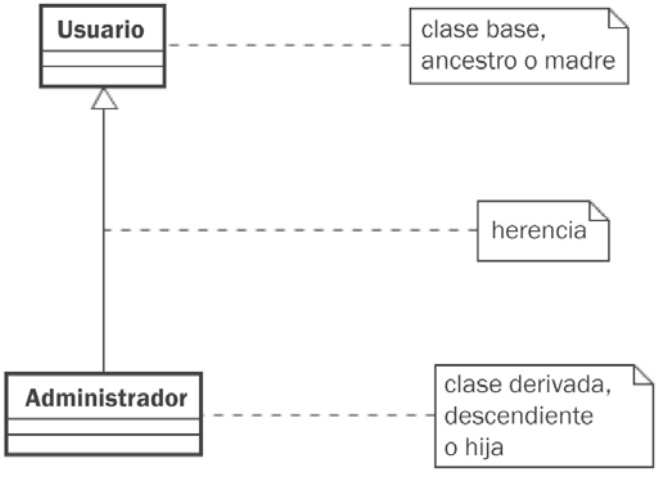

Figura 7.9 Herencia.

Muchos lenguajes orientados a objetos permiten definir clases que no deben tener instancias. UML, como muchos otros lenguajes, las llama **clases abstractas**, y se indican en letra cursiva en el diagrama.

De la misma manera, los **métodos abstractos**, que existen en algunos lenguajes para indicar que los mismos no deben ser invocados, en UML también se escriben en cursiva.

La figura 7.10 muestra la clase abstracta `Usuario`, que tiene el método abstracto `asignarPermisos`.

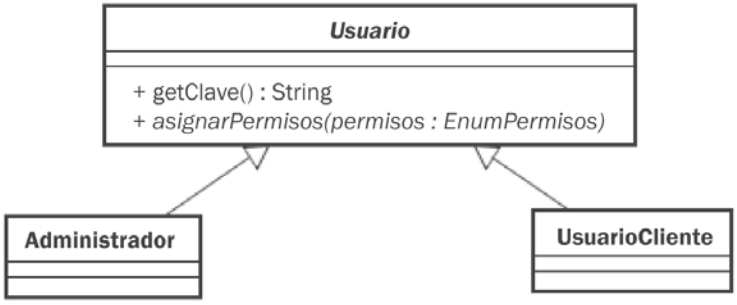

Figura 7.10 Clase y método abstracto.

Hay personas que en vez de escribir las clases abstractas en cursiva, le agregan la propiedad `{abstract}` al lado del nombre. Esto es especialmente importante cuando, en vez de usar una herramienta de software, modelamos a mano alzada para comunicarnos informalmente entre humanos, ya que no es sencillo escribir en cursiva en una pizarra o papel de forma que resulte clara esta distinción.

Hay lenguajes que admiten el concepto de genericidad o plantillas de tipos. Ambos conceptos no son exactamente lo mismo, pero tienen un uso parecido. El primero fue introducido por Eiffel, mientras que el segundo es típico de C++. Con el tiempo, hubo muchos lenguajes que admitieron que los tipos de datos fuesen parámetros de clases y métodos.

UML permite representar una clase con un tipo parámetro, no así en el caso de métodos. La figura 7.11 muestra que la clase `Documento` tiene como parámetro al tipo `TipoDocumento`, y la representación de una clase particular con el tipo parámetro instanciado, para lo que usamos una dependencia con el estereotipo `<<bind>>`.

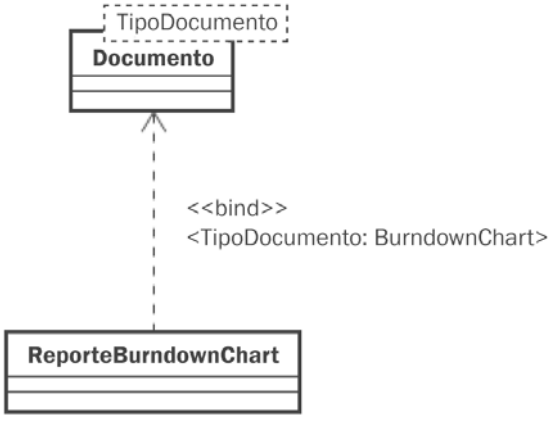

Figura 7.11 Clase parametrizada.

También existe la posibilidad de representar clases internas, como muestra la figura 7.12, en la que la clase `Estado` es interna a `Proyecto`. Esto, por supuesto, tiene diferentes significados según el lenguaje de programación.

Incluso la particularidad de las clases internas anónimas, como existen en Java, se puede representar con el estereotipo `<<anonymus>>`. Esto puede verse en la figura 7.13, que indica que hay una clase sin nombre (anónima), interna a la clase `Sprint`, y que deriva de la clase `SprintBacklog`. Insistamos en que ésta es una particularidad de unos pocos lenguajes, y que no estamos diciendo que `SprintBacklog` sea la clase interna anónima, sino que la clase interna anónima es una descendiente de `SprintBacklog`.

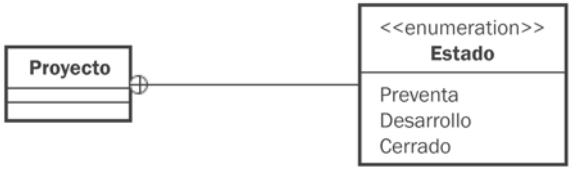

Figura 7.12 Clase interna.

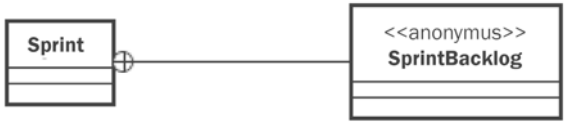

Figura 7.13 Clase interna anónima.

Desde sus inicios, UML permitió representar **clases activas**, esto es, clases cuyas instancias podrían ejecutar cada una un hilo de ejecución distinto. La figura 7.14 muestra la notación de clase activa en UML 1.x, con un contorno más grueso, mientras la 7.15 muestra la nueva notación en UML 2, con bordes laterales dobles.

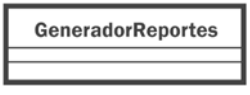

Figura 7.14 Clase activa en UML 1.

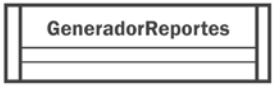

Figura 7.15 Clase activa en UML 2.

No todos los lenguajes de programación orientados a objetos respetan a fondo el paradigma. Algunos de ellos, por cuestiones diversas, contienen funciones de la biblioteca estándar que, al no estar encapsuladas como métodos de clases, se las

ha agrupado en una clase utilitaria especial, cuyos métodos son todos de clase. Esto pasa, por ejemplo, con la clase `Math` de las plataformas Java y .NET.

Y así como los lenguajes contienen estas irregularidades, hay muchos diseñadores que caen en ellas también. Sin ponernos a juzgar aquí la pertinencia o no de dichos diseños, y admitiendo pragmáticamente esta realidad, digamos que en UML hay una manera de representar estas "clases" tan particulares. Esto se hace mediante el uso del estereotipo `<<utility>>`, que indica que se trata de una pseudo-clase, que es abstracta por definición y sin atributos ni métodos de instancia. La figura 7.16 muestra una de estas clases.

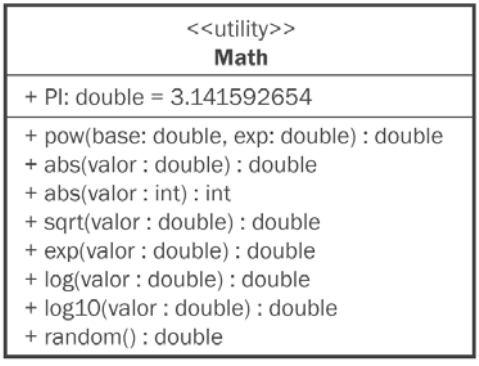

Figura 7.16 Clase utilitaria.

En la figura no hemos puesto a los métodos y a los atributos como de clase, pues resulta obvio. De hecho, tampoco tiene mucho sentido colocar la visibilidad de los mismos, aunque lo hicimos para mayor claridad.

Para los lenguajes que, como Smalltalk, manejan **metaclases** –esto es: clases cuyas instancias son, a su vez, otras clases– existe el estereotipo `<<metaclass>>`.

Asociaciones en lenguajes de programación

Venimos viendo desde el capítulo de requisitos las distintas asociaciones en diagramas de clases.

Ahora bien, ¿qué significa una asociación y sus distintos elementos en el contexto del diseño detallado? Veámoslo con el ejemplo del diagrama de la figura 7.17.

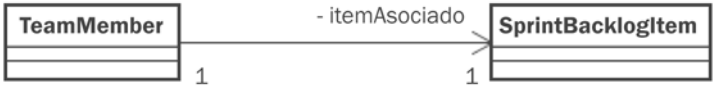

Figura 7.17 Clases con asociaciones.

Una asociación entre dos clases, en el nivel de implementación, significa que una clase tiene un atributo cuyo tipo es otra clase. La navegabilidad nos indica cuál es la clase del atributo, y el nombre del atributo es el rol de la asociación en el sentido de la navegabilidad.

Así, en la figura 7.18, la clase `TeamMember` tiene un atributo `itemAsociado` cuya clase es `SprintBacklogItem`.

Como vemos, el rol de la asociación tiene una visibilidad, que indica la visibilidad del atributo en cuestión. En nuestro caso, estamos diciendo que el atributo `itemAsociado` es privado, esto es, que sólo puede conocerlo la clase `TeamMember`, que es la que lo contiene.

También hemos visto los conceptos de agregación y composición. El concepto de agregación, pobremente definido en UML, no tiene demasiado significado en el nivel de diseño detallado. Sí es una buena idea representarla en diagramas de análisis, como lo hicimos oportunamente.

Dijimos que había composición cuando el contenedor es responsable del ciclo de vida del objeto contenido. En el nivel de implementación, esto es fuertemente dependiente del lenguaje. Por ejemplo, en los lenguajes que vinculan los objetos entre sí con un modelo de referencias, algo como la composición es una decisión de diseño que el programador deberá implementar en código, y que muy probablemente afecte otras cuestiones de implementación, tales como la manera de determinar igualdad, cómo hacer copias o cómo manejar la persistencia. En lenguajes como C++, en los cuales un objeto puede estar contenido físicamente dentro de otro, la composición se da más naturalmente.

No obstante, hay profesionales que prefieren representar la composición en sus diagramas de diseño detallado, aunque –como acabamos de decir– en algunos lenguajes sea una cuestión solamente conceptual. En estos casos, el modelado de la composición les sirve a los programadores para saber cómo crear los objetos contenidos, así como la manera en que se espera que implementen la igualdad, la copia de objetos y la persistencia.

En cuanto a la multiplicidad de las asociaciones, el valor 1 es el único que no ofrece matices. Cualquier valor mayor o rango de valores, implica alguna forma de colección de objetos como atributo de la clase. Si el rango de valores tuviera una cota superior o inferior, esto es, una indicación para que el programador la tenga en cuenta en el momento de la codificación.

Por ejemplo, en la figura 7.19, el atributo `items` de la clase `SprintBacklog` deberá estar implementado como una colección. Hemos agregado la palabra clave `{ordered}`, que implica que el orden en que se guarden los elementos de la colección es importante. Sin embargo, ese diagrama no nos dice nada del tipo de colección en cuestión. Si se desease indicar el tipo de la colección, debe usarse una **asociación calificada**, como hacemos en la figura 7.20, en la que decimos que el atributo ítems es un `ArrayList`. En la misma, eliminamos la palabra clave `{ordered}`, porque un `ArrayList` es de por sí una colección ordenada.

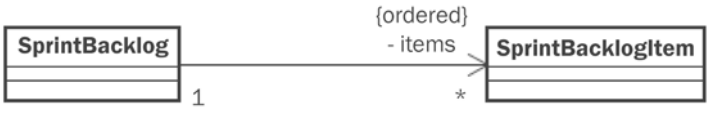

Figura 7.18 Asociación de cardinalidad múltiple.

Figura 7.19 Asociación calificada.

Hay quienes en vez de una asociación calificada usan una clase de asociación, como la que vimos, con otro fin, en el capítulo 4. La figura 7.20 es un ejemplo de esto. Sin embargo, lo más usual es usar asociaciones calificadas en estos casos, dejando las clases de asociación para los diagramas más conceptuales.

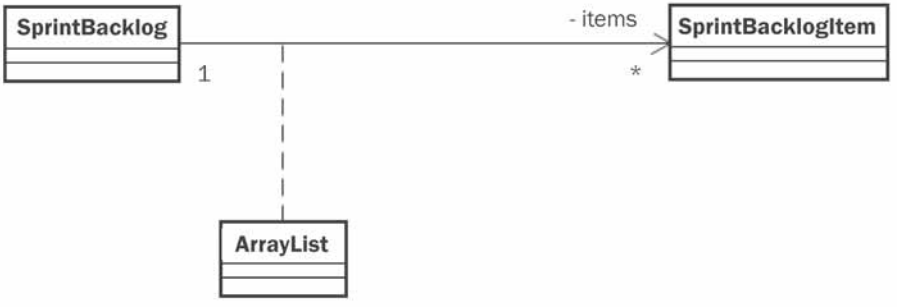

Figura 7.20 Clase de asociación como alternativa a una asociación calificada.

A los atributos y los roles de las asociaciones se les puede colocar también la palabra clave `{readonly}`, para indicar que los clientes de esa clase sólo pueden consultar su valor, pero no alterarlo.

Tipos de dependencias en diagramas de clases

Hay muchos tipos de dependencias entre las clases de un diagrama. La asociación y la herencia son formas de dependencia fuertes, en el sentido de que la clase dependiente es muy probable que cambie cuando cambia aquella de la cual depende. La herencia, en los hechos, denota un grado de dependencia mucho mayor que la asociación.

Sin embargo, UML permite representar dependencias más débiles, con la línea punteada entre dos clases. En realidad, no hay una única forma de dependencia débil y, por ello, en UML se han previsto algunos estereotipos para denotar distintas formas de dependencia.

Por ejemplo, en el caso de tener que usar clases parametrizadas e indicar una clase concreta resultante, vimos el estereotipo `<<bind>>` (ver figura 7.11).

Pero hay muchos estereotipos más, algunos de los cuales rara vez se usan.

Un ejemplo de estereotipo que se suele usar es el `<<create>>`, que indica que las instancias de una clase tienen entre su comportamiento la posibilidad de crear instancias de la otra. La figura 7.21 muestra un ejemplo en el que el método `ordenar` de la clase `SprintBacklog` crea instancias de una clase de `ComparadorSBI`. Es habitual poner una nota explicativa para que quede claro cuál es el método creador de instancias, ya que la notación gráfica no lo indica. Notemos que, al ser `ComparadorSBI` una interfaz, lo que estamos diciendo es que se crean instancias de alguna clase que implemente esa interfaz.

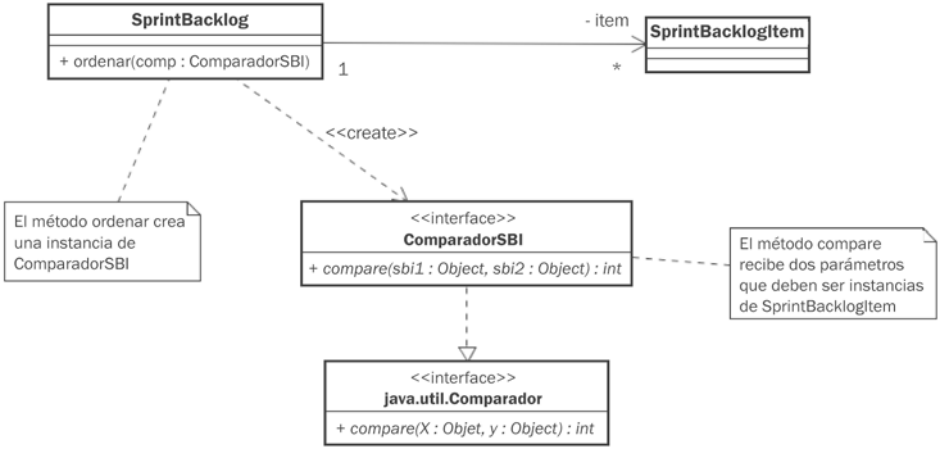

Figura 7.21 Uso del estereotipo create.

A veces se usa `<<instantiate>>` en vez de `<<create>>`. La diferencia en el significado de ambos estereotipos es bastante sutil, y prácticamente no hay profesionales que hagan el distingo.

Otro estereotipo, en este caso poco usual, es `<<local>>`, que tiene el mismo significado del anterior, pero indica que la instancia creada se mantiene en una variable local de un método, perdiéndose cuando el método termina. De hecho, en la figura 7.21, pudo haberse colocado el estereotipo `<<local>>`, pero su uso es muy raro, y por eso no lo hemos incluido.

El estereotipo `<<parameter>>` es otro de los que rara vez se representan, e indica si una clase se usa como tipo del parámetro de un método de otra clase. La figura 7.22 muestra el uso de este estereotipo. Notemos que el mismo estereotipo pudo haberse colocado en la dependencia entre `SprintBacklog` y `ComparadorSBI`, en la figura 7.21, ya que el método `ordenar` no sólo crea instancias de una clase de `ComparadorSBI`, sino que también tiene ese tipo como parámetro. Pero no lo hemos puesto porque nos parece más relevante el primero.

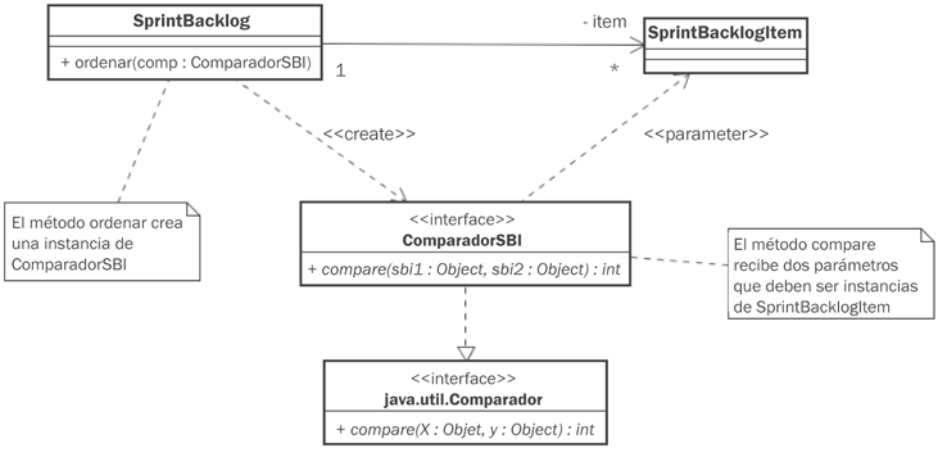

Figura 7.22 Uso del estereotipo `parameter`.

Finalmente, existe el estereotipo `<<delegate>>`, que indica que una clase delega comportamiento en otra. Si bien es bastante habitual que una clase delegue en otra, lo más usual es que esta situación se dé cuando hay asociaciones entre ambas clases. Si así no fuera, puede indicarse con este estereotipo en una relación de dependencia para hacer hincapié en esta situación poco común. La figura 7.23 muestra el mismo ejemplo, pero indicando que la clase `SprintBacklog` delega comportamiento en un `ComparadorSBI`.

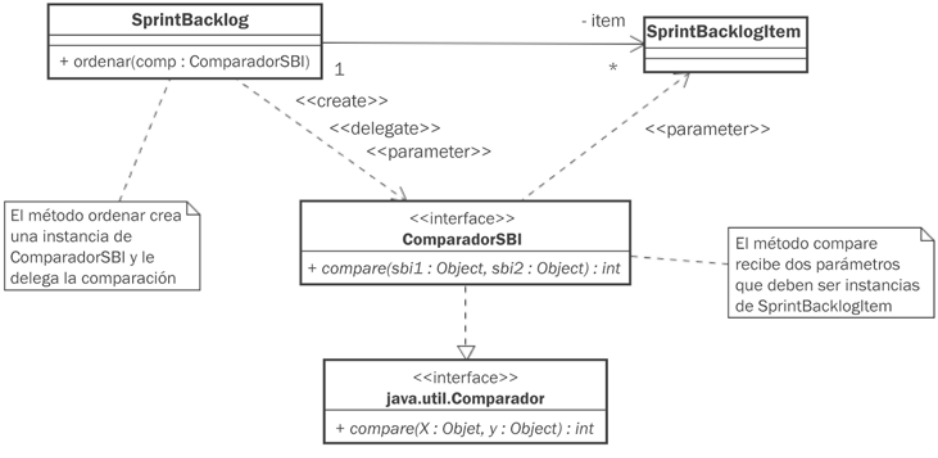

Figura 7.23 Uso del estereotipo `delegate`.

La figura 7.23, con su sobrecarga de estereotipos en las dependencias, es un buen ejemplo de que no hay que hacer un uso exhaustivo de los mismos.

Interfaces y realización en diagramas de clases

Hay lenguajes que tienen una construcción denominada **interfaz**, como ocurre con Java y la plataforma .NET. Una interfaz es, a la vez, una clase sin estado y abstracta, y un contrato que especifica los métodos que una clase que realice esa interfaz debe implementar.

En UML, una interfaz se representa como una clase, con el estereotipo `<<interface>>`, y la **realización** de la interfaz (a veces denominada implementación) con la misma flecha de la herencia, pero de línea punteada.

La figura 7.24 muestra que la clase `LegajoRemoto` realiza (o implementa) la interfaz `LegajoEmpleado`. Por supuesto, los métodos definidos en `LegajoEmpleado` deben estar implementados en `LegajoRemoto`, pues ésta es una clase concreta. Sin embargo, por resultar algo obvio, no lo hemos representado. Otra cuestión optativa es el uso de la cursiva en el nombre `LegajoEmpleado`, ya que su condición de abstracta resulta obvia.

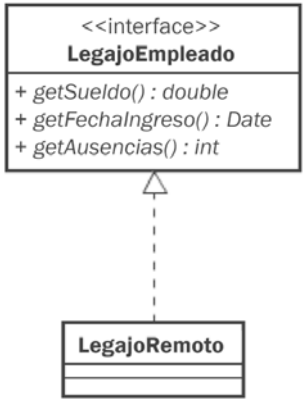

Figura 7.24 Interfaz y realización por una clase.

UML especifica que la realización de una interfaz puede acompañarse por el estereotipo `<<realization>>`. En este libro, lo hemos omitido.

Por supuesto, como el tipo de un atributo puede ser una interfaz, es posible encontrar asociaciones entre clases e interfaces, como muestra la figura 7.25.

En este caso, en realidad, estamos diciendo que el atributo `legajo` de la clase `Empleado` es de tipo `LegajoEmpleado`. Pero por ser `LegajoEmpleado` una interfaz, que es por definición una clase abstracta, el objeto que se referencie a través del atributo `legajo` podrá ser una instancia de la clase `LegajoRemoto` o de la clase `LegajoLocal`.

En los lenguajes en los que no existe una construcción del tipo de la interfaz, no tiene mucho sentido representarlas en diagramas de clases.

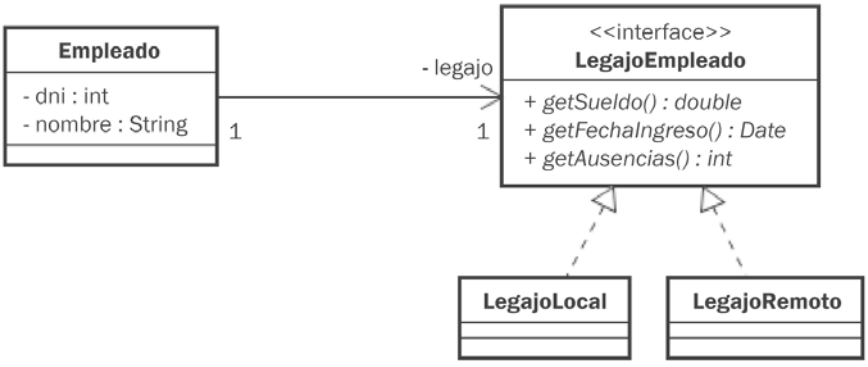

Figura 7.25 Asociación de clase con interfaz.

Además, esta notación de interfaces, denominada **canónica**, puede usarse en los diagramas de componentes para especificar mejor los métodos que expone un componente a través de su interfaz.

También la notación de interfaces que vimos con los diagramas de componentes puede usarse en diagramas de clases, aunque rara vez se utiliza.

Diagramas de paquetes

Puede parecer una contradicción hablar del diagrama de paquetes junto con el diseño detallado, cuando el diagrama de paquetes es un modelo macro y de alto nivel.

Sin embargo, el paquete no es solamente un modelo estructural de agrupamiento de UML. Hay varios lenguajes de programación que permiten agrupar las clases. Así ocurre con la construcción *package* de Java y con los *namespaces* de C# y C++.

Si bien esta noción de paquete no es equivalente en los distintos lenguajes, es útil para organizar el código. Así, en Java y C#, los paquetes son parte de la definición de la clase y sirven como directivas de resolución de nombres.

En Java incluso indican cómo debe ser la organización física del código en el sistema de archivos. En Smalltalk son más bien agrupaciones que hace el programador en el entorno de desarrollo, sin ningún significado adicional.

La manera de representar las agrupaciones de clases en UML es con el ya conocido diagrama de paquetes. La figura 7.26 muestra un diagrama de paquetes y clases combinados.

También se pueden mostrar las clases internas de un paquete sin decir nada de sus interrelaciones ni de cómo se relacionan las clases entre paquetes, solamente colocando sus nombres. En este caso se puede también especificar si las clases

internas son o no visibles para otros paquetes, con los atributos de visibilidad que vimos en los diagramas de clases. La figura 7.27 muestra el mismo diagrama anterior con este otro formato.

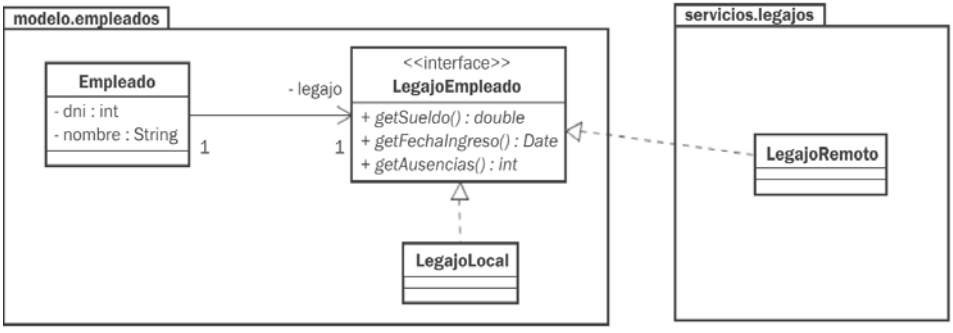

Figura 7.26 Paquetes y clases.

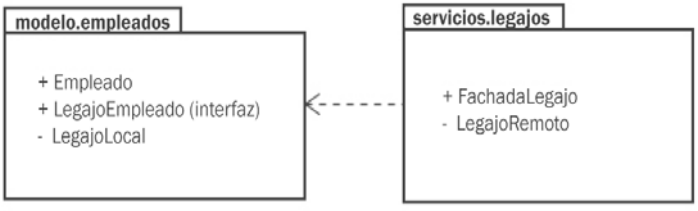

Figura 7.27 Visibilidad de elementos internos.

La visibilidad de los elementos internos puede tener diferentes significados según el lenguaje. Por ejemplo, en Java, existe la posibilidad de que una clase sea visible solamente para las demás clases de su mismo paquete, y en ese sentido sería privada afuera del mismo. Muchos otros lenguajes no tienen esta posibilidad.

Diagramas de objetos

Los diagramas de objetos, indicando vínculos entre instancias y valores de atributos, pueden ser utilizados para razonar sobre el diseño.

Un aspecto interesante de los mismos es que nos muestran algunos objetos y sus enlaces en un momento particular de la ejecución de la aplicación que nos interesa analizar. Como ya los hemos visto suficientemente en el capítulo de análisis, no abundaremos más aquí, repitiendo la figura 5.24, con el número 7.28.

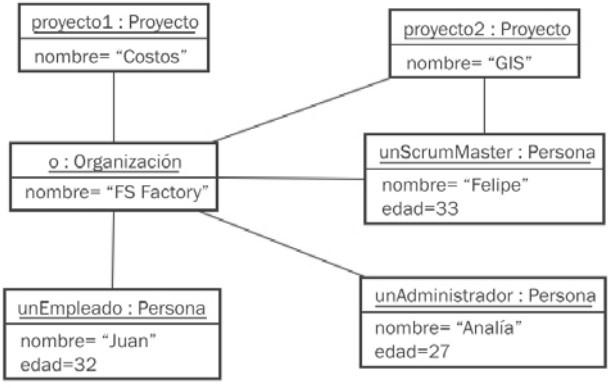

Figura 7.28 Objetos con atributos.

Colaboraciones

En UML, una **colaboración** es un modelo conceptual que incluye modelos estructurales y de comportamiento para representar cómo se materializa un determinado escenario funcional o de diseño. Puede servir para describir el diseño de un caso de uso o de un mecanismo de implementación.

Si bien existen diagramas para representar estos escenarios, como los de clases y los de interacción, en ambos se está ante un único tipo de vista: en un caso es una vista estructural y, en el otro, de comportamiento.

Lo que busca la colaboración es dar una representación y un nombre únicos para ambas cosas.

En UML, una colaboración se representa como una elipse de bordes discontinuos, con el nombre de la colaboración adentro, como se muestra en la figura 7.29.

Figura 7.29 Una colaboración.

Sin embargo, mostrar una colaboración aislada no tiene demasiado sentido. Hay quienes han pretendido mostrar relaciones entre colaboraciones, incluyendo dependencias, asociaciones y generalizaciones. Pero esto no tiene demasiada utilidad.

Un poco más interesante es analizar cómo se relaciona una colaboración con sus aspectos estructurales y de comportamiento. La figura 7.30 muestra un escenario más interesante. En ella estamos especificando en qué consiste la colaboración que, en este caso, representa los diagramas de análisis del cierre de un *Sprint Backlog Item*, que ya representamos en las figuras 5.7 y 5.8.

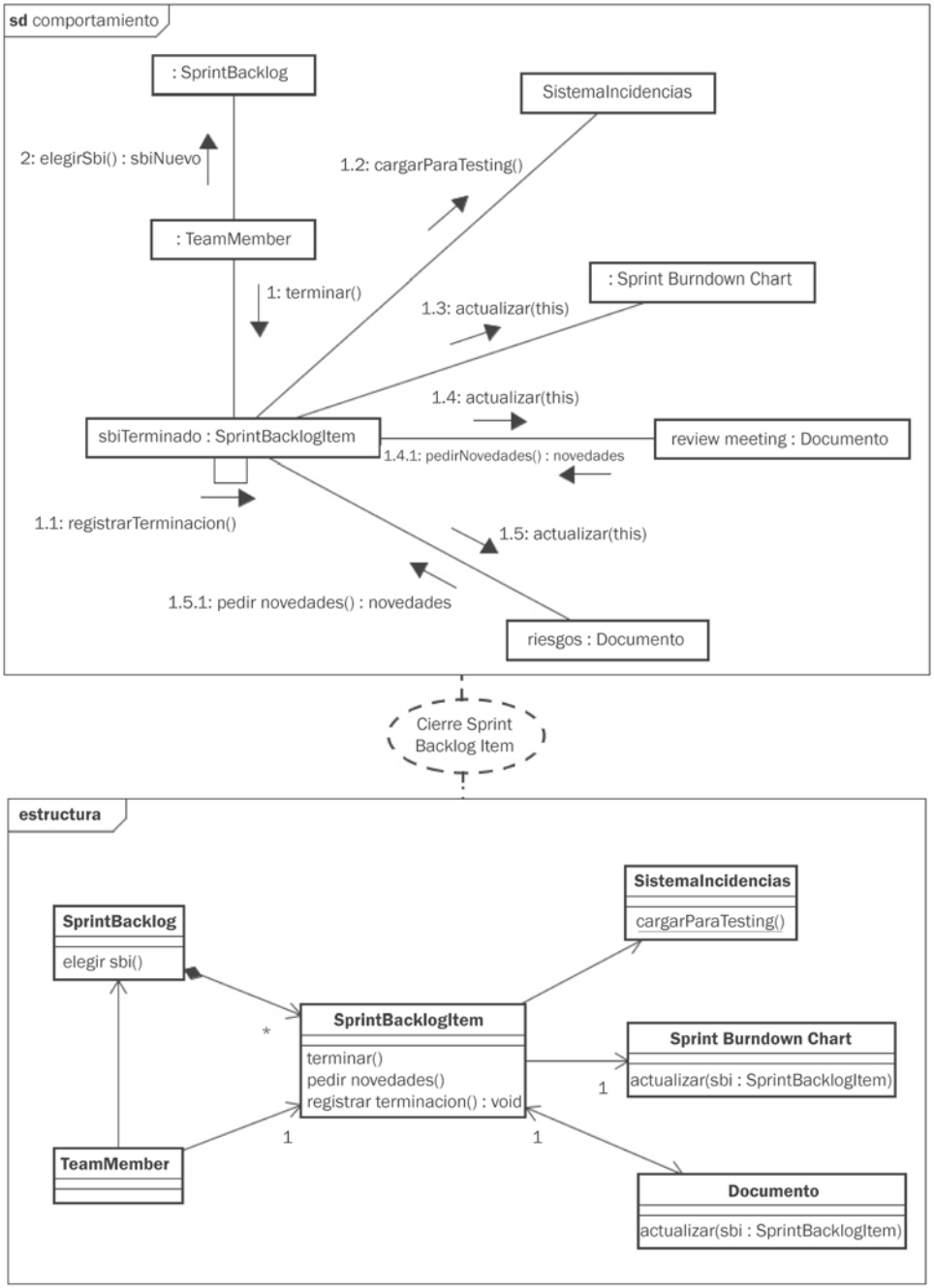

Figura 7.30 Una colaboración con aspectos estructurales y de comportamiento.

Como las colaboraciones se usan para mostrar un aspecto puntual del diseño, es importante mantener solamente el nivel de detalle necesario para eso, sin incluir detalles que no hagan al escenario que se quiere mostrar. Por lo tanto, habría que suprimir todo aspecto que no sea relevante para el propósito del modelo.

Lo más común es que se asocie una colaboración a diagramas de clases y/o de interacción. Sin embargo, hay ocasiones en los que conviene dibujar diagramas de objetos, de paquetes, de estados u otros, según lo que se quiera mostrar. Por ejemplo, en la figura 7.30, podríamos haber representado los aspectos estructurales en un diagrama de objetos que muestre una situación particular en el tiempo. Eso es lo que hicimos en la figura 7.31.

Si desea asociar una colaboración con un único diagrama, se puede hacer como ya vimos, o representarlo todo dentro de la elipse de borde discontinuo. En este último caso, se hace una separación horizontal entre el nombre de la colaboración y el diagrama, con una línea discontinua, como se muestra en la figura 7.31.

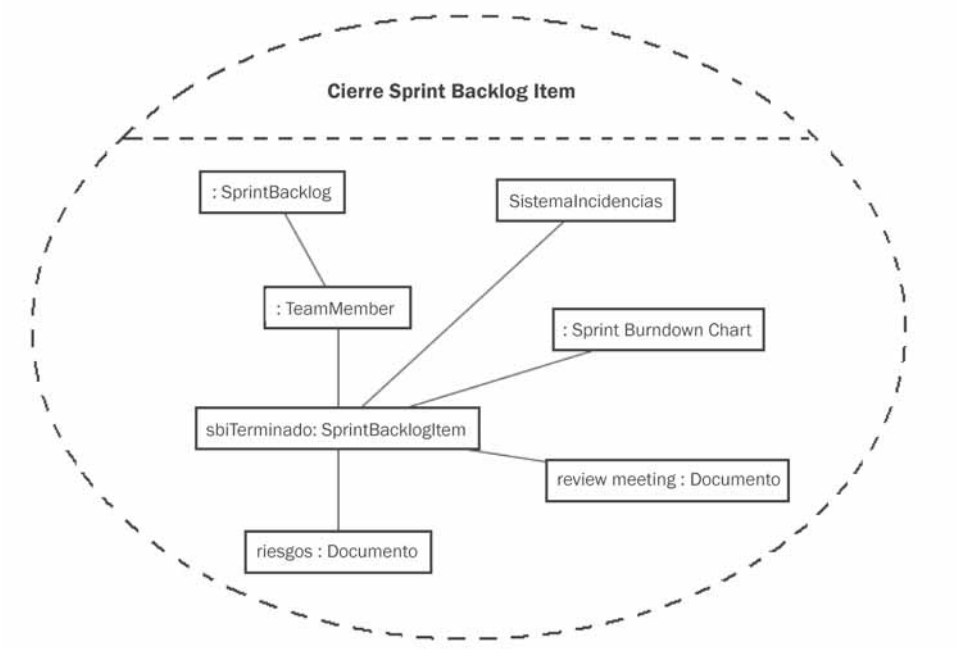

Figura 7.31 Una colaboración representada junto con su diagrama.

Es sintomático lo poco que se usan las colaboraciones en la práctica profesional. Un indicio de ello es la cantidad de herramientas de UML que no permiten diagramar colaboraciones. Otra es la confusión reinante sobre su uso. También hay que destacar el enredo que provocó la propia definición de UML, que en sus versiones 1.x tenía diagramas de colaboración y colaboraciones como conceptos separados. La versión 2 denominó diagramas de comunicación a los que antes se llamaban de colaboración,

pero probablemente sea uno de los cambios a los que mayor cantidad de gente se ha resistido.Ya volveremos sobre las colaboraciones al tratar el modelado de patrones de diseño.

Diagramas de estructura compuesta

La figura 6.10, en el capítulo anterior, introdujo un diagrama de componentes compuesto que, desde el punto de vista de UML, es un tipo de diagrama diferente. Se trata del **diagrama de estructura compuesta**, de los cuales el de la figura 6.10 es sólo una variante simple; incluso podría caracterizarse de poco ortodoxa, pues muestra componentes, que no es lo que pretende el estándar.

El diagrama de estructura compuesta muestra la estructura interna de una clase y sus puertos. El término **clase**, en este contexto, es más amplio que el de una clase software: podría ser un componente, una clase software, una clase de análisis o una entidad del modelo de dominio, entre otros.

La figura 7.32 muestra un diagrama de la estructura compuesta `Organizacion`, del que pasaremos a analizar sus elementos.

Un **puerto**, que se representa como un pequeño cuadrado, generalmente en el borde de una estructura compuesta o de un elemento interno de la misma, especifica un punto de interacción de las partes internas de la clase con su entorno. Las líneas que unen los puertos con las partes internas de la clase se llaman **conectores**. En el diagrama de la figura 7.32, se representaron todos los puertos, aunque esto no es obligatorio. De la misma manera, a los puertos no se les colocó ningún nombre, cosa que podría haberse hecho. Incluso, los conectores podrían tener nombres.

Cada puerto puede tener una o más interfaces, tanto requeridas como proporcionadas. Las interfaces requeridas indican las solicitudes que la clase puede hacer a su entorno. Las interfaces proporcionadas, en cambio, indican cuáles son las solicitudes que el entorno le puede hacer a la clase. La notación para indicar interfaces requeridas y proporcionadas es la misma de los diagramas de componentes que ya vimos.

Cada uno de los rectángulos que representan partes de la estructura compuesta se denominan –como bien podría deducirse– **partes**. Cada parte puede tener un nombre y un tipo, que se representan con la notación habitual de UML. En la figura, se muestran tres tipos de partes: unas anónimas de tipo `Empleado`, otras anónimas de tipo `Proyecto`, y una con nombre de tipo `DatosGeneralesEmpresa`. Notemos que las partes pueden tener una cardinalidad, que se suele indicar cuando es distinta de 1. Por ejemplo, las partes de tipo `Empleado` pueden ser varias, pero deberán ser como mínimo 1. Las de tipo `Proyecto` pueden ser cualquier cantidad.

Según la definición de UML, las estructuras compuestas podrían contener colaboraciones.

Si bien venimos hablando de las estructuras compuestas como clases, lo cierto es que eso obedece a cómo prevé UML que se usen. Sin embargo, si nos rigiésemos por el uso más común, deberíamos hablar de componentes. Que, por otro lado, es lo que hicimos en la figura 6.10.

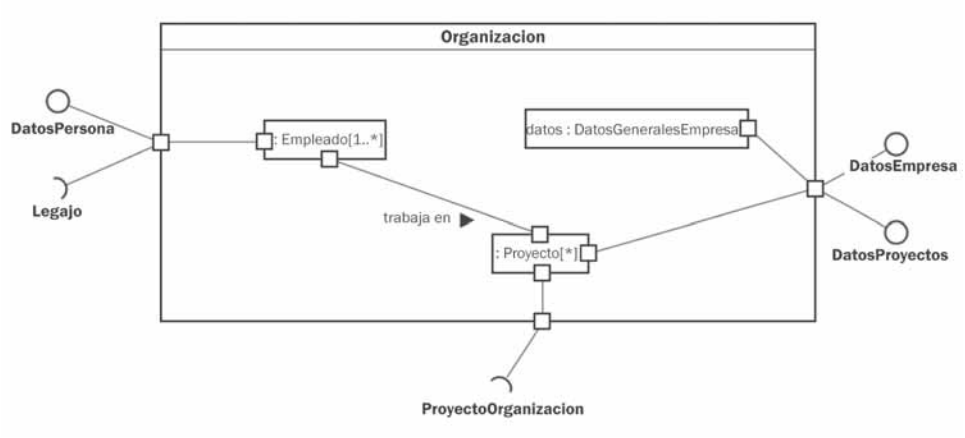

Figura 7.32 Diagrama de estructura compuesta.

La razón por la que hemos dejado la explicación de este diagrama hasta este momento es su escaso uso. De hecho, es un añadido de UML 2 que pocos profesionales usan, aunque a veces se lo dibuja informalmente para analizar algún aspecto de diseño.

La otra contra que tiene es que tiende a ser más útil para realizar análisis o diseño basado en aspectos estructurales, no en comportamiento, y por esta razón no es demasiado feliz como diagrama orientado a objetos, salvo en el nivel de implementación.

INGENIERÍA INVERSA DE UML DESDE LA PROGRAMACIÓN

Noción de ingeniería inversa

Denominamos **ingeniería inversa** a la generación de modelos de mayor abstracción desde otros modelos más concretos. El caso más típico es la generación de diagramas a partir de código fuente, que es, por definición, el modelo de menor abstracción posible.

El porqué de la denominación de "inversa" es bastante obvio: se está procediendo al revés que lo que esperaría quien aborda los modelos como herramienta de desarrollo. Sin embargo, como veremos enseguida, no tiene nada de descabellado o infrecuente. Además, hace mucho tiempo que existen herramientas para hacer ingeniería inversa desde el código fuente, así que tampoco estamos hablando de una tarea difícil de realizar.

Usos y limitaciones de la ingeniería inversa

El principal uso de la ingeniería inversa es el análisis del código, y, en consecuencia, de la calidad del diseño, en forma gráfica. Puede argüirse que cada vez es más

sencillo leer código fuente, e incluso que hay herramientas que, sin necesidad de generar diagramas, analizan y generan excelentes reportes de calidad de diseño a partir del código fuente. Sin embargo, hay mucha gente que se siente más cómoda analizando diagramas que código fuente, y para ellas la ingeniería inversa es una solución.

Otro uso posible es poder brindar documentación gráfica de una biblioteca de clases a partir del propio código de esas clases, para ser usada por un cliente de esta biblioteca. De hecho, esto es algo que el autor echa de menos en la documentación de los *frameworks* y bibliotecas más populares, aunque cuenten con excelentes referencias hipertextuales. Como siempre, habría que tener en cuenta que existen muchos profesionales que leen mejor las notaciones gráficas que los textos.

No obstante, hay que tener algo de cuidado con la herramienta de ingeniería inversa que se elija. Algunas de las disponibles en el mercado tratan de llevar todos los detalles del código a los diagramas, creando así modelos de un nivel de detalle tal que resultan difíciles de utilizar. Por ejemplo, si lo que se desea es documentar una biblioteca para facilitar su uso, no deberían figurar elementos privados o no visibles para los clientes. Y, como regla general, no mostrar detalles que confunden al lector.

Tal vez la mejor herramienta es aquélla que permita al usuario definir el nivel de detalle deseado.

Otra limitación que se suele señalar es que, al pasar del código al diagrama, perdemos la abstracción propia del modelo. Al fin y al cabo, y como vimos en el capítulo 1, un modelo se realiza para abstraer, mientras que el código es el más concreto de los artefactos. Esta desventaja –a pesar de ser muy mencionada– tal vez no sea tan importante. Por otra parte, en el momento en que decidimos hacer ingeniería inversa deberíamos ser conscientes de que venimos de un modelo de menor abstracción, y de que nuestro objetivo no es encontrar un modelo muy abstracto, sino solamente generar documentación o una visión gráfica del propio código.

Clases desde código

La operación más común de ingeniería inversa es generar diagramas de clases a partir del código.

Esto tiene varias ventajas. Entre ellas:

- Permite documentar las clases para su uso.
- Permite ver acoplamiento y dependencias entre clases.
- Permite analizar el uso de clases desde otras mediante herencia, asociación o dependencia.
- Permite ver los servicios que ofrece cada clase.
- Permite ver qué métodos y atributos están ubicados en cada clase, previamente a una refactorización.
- Permite descubrir patrones en el diseño.
- Nos da una idea de las responsabilidades de cada clase.

Si bien un diagrama de clases generado a partir del código suele ser muy detallado, es probable que, si deseamos utilizarlo para analizar alguno que otro aspecto estructural, tanto detalle nos abrume y nos impida ver lo realmente importante. En consecuencia, habitualmente, un diagrama de clases obtenido por ingeniería inversa requiere algunos retoques por parte de la persona que va a trabajar con él.

Por ejemplo, un diagrama de clases para analizar acoplamiento podría contener solamente los nombres de las clases y las relaciones de herencia y asociación, sin detalles internos de las clases. Otra posibilidad es usar directamente diagramas de paquetes, como veremos a continuación.

Si, en cambio, queremos analizar patrones, tal vez necesitamos algunos métodos y atributos, aunque no forzosamente todos los detalles.

De esta manera, en cada caso, el diagrama que se usará podría ser diferente para un mismo código fuente.

Paquetes desde código

Los desarrolladores novatos no ven mucha ventaja a generar un diagrama de paquetes a partir del código. Sin embargo, una característica esencial del buen diseño de paquetes es que el acoplamiento entre éstos sea mínimo. Por esta razón, un buen diseñador suele analizar muy especialmente el acoplamiento entre paquetes una vez que la aplicación ya está construida, y los diagramas obtenidos mediante ingeniería inversa pueden ser una excelente herramienta.

Muchos analizadores automáticos de código trabajan analizando el acoplamiento de paquetes sin generar diagrama alguno, y son muy útiles. Sin embargo, realizar un diagrama de paquetes UML a partir del código sirve para quienes se llevan mejor con las notaciones gráficas.

Una vez generado el diagrama de paquetes, se puede analizar el acoplamiento entre los mismos y también buscar dependencias cíclicas indirectas. Hay quienes afirman que ésta es la mayor utilidad del diagrama de paquetes y tienen una especial afición por ellos.

Las dependencias entre paquetes son también útiles para estudiar secuencias de compilación.

Interacciones desde código

Los diagramas de interacción más comunes (secuencia y comunicación) también son susceptibles de ser derivados desde el código. Empero, no es muy común hacer este uso.

Lo cierto es que no brindan tanta utilidad. Una posibilidad interesante, sin embargo, es usarlos para analizar casos de prueba. Otra es analizar aquellos que resultan muy intrincados, para realizar refactorizaciones que mejoren el diseño y simplifiquen las interacciones típicas.

De los distintos diagramas de interacción, el de secuencia suele ser mejor para ver relaciones temporales y, por lo tanto, es más recomendable para las refactorizaciones. Aunque, como siempre, es una cuestión de gustos.

TEMAS ADICIONALES DE DISEÑO Y CONSTRUCCIÓN

Más allá de UML en la documentación de código

Cuando hablamos de UML como herramienta de documentación, no debemos olvidar que el código se puede documentar de muchas maneras, y no siempre los modelos en forma de diagramas son la mejor de ellas. En general, se prefiere documentar el código fuente en el propio código, de manera tal de facilitar el mantenimiento, tanto del código como de la documentación.

La manera más elemental de documentar código es hacerlo legible, con buenos nombres y estructuras claras y conocidas. Es casi obvio que la mejor documentación del código es el propio código, pero a menudo se olvida.

Una manera de mejorar la calidad del código fuente es utilizar comentarios. Ésta es una facilidad que existe desde los primeros lenguajes de programación, y que permite insertar aclaraciones al código.

Algunas situaciones en las que conviene usar comentarios son:

- Cuando haya que aclarar código que no puede escribirse más claro.
- Como resumen de una secuencia de acciones.
- Como resumen de las responsabilidades de una clase, a manera de prólogo.
- Como resumen del comportamiento de un método, a manera de prólogo.
- Para explicitar restricciones de un método o clase, a manera de prólogo.

También hay situaciones en las que los comentarios deberían evitarse. Entre ellas, se destacan:

- Cuando el código puede aclararse sin necesidad de comentarios.
- Cuando una secuencia de acciones que se va a comentar se puede separar en un módulo aparte.
- Cuando vamos a repetir lo que dice el código en el comentario.

Hay algunos lenguajes que permiten generar documentación de referencia a partir del código fuente, con ciertas herramientas especializadas. La más conocida tal vez sea el estándar de Java, *javadoc*, que, a partir de ciertos comentarios en el código y de ciertos formatos y palabras clave, generan documentación de referencia en formato HTML. Hay otras herramientas en otros lenguajes y plataformas que generan XML, archivos de texto y demás.

Lo interesante de esta documentación es que, si está bien diseñada, sirve tanto de comentario como de documentación externa de referencia. La única limitación es que no suelen proveer, aunque en algunos casos sí, diagramas UML que den una imagen más gráfica de la documentación. Por ejemplo, en ocasiones, nos encontramos con documentación de referencia que nos vendría bien que viniese acompañada de diagramas de clases para entenderla mejor.

Modelado de patrones

Los patrones de diseño no están previstos en UML.

Sin embargo, las colaboraciones pueden ser una herramienta interesante. Por ejemplo, es habitual representar la implementación del patrón Proxy en un sistema con un esquema parecido al de las figuras 7.33, 7.34 y 7.35. La primera muestra un Proxy típico, mientras que las siguientes ilustran su uso en un diagrama de secuencia y en uno de clases, respectivamente.

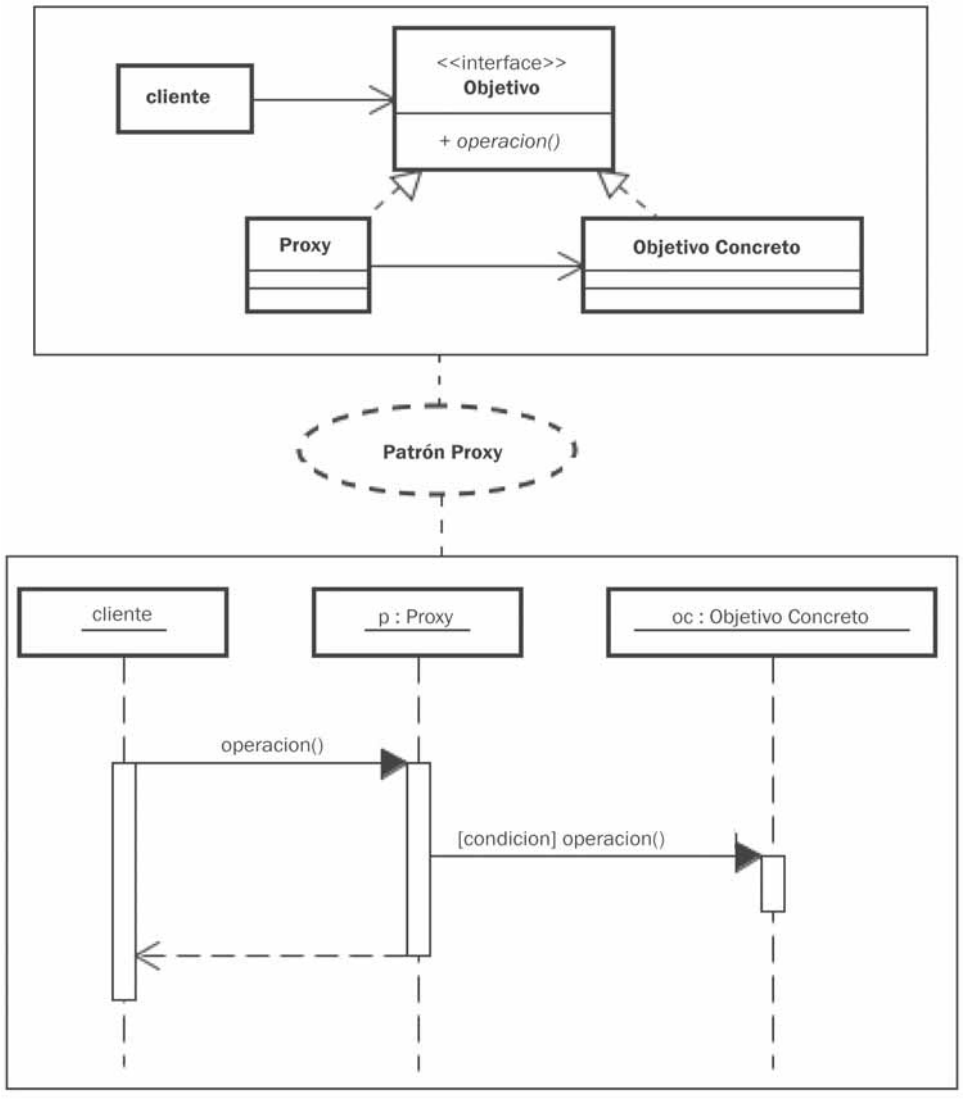

Figura 7.33 Colaboración de un `Proxy`.

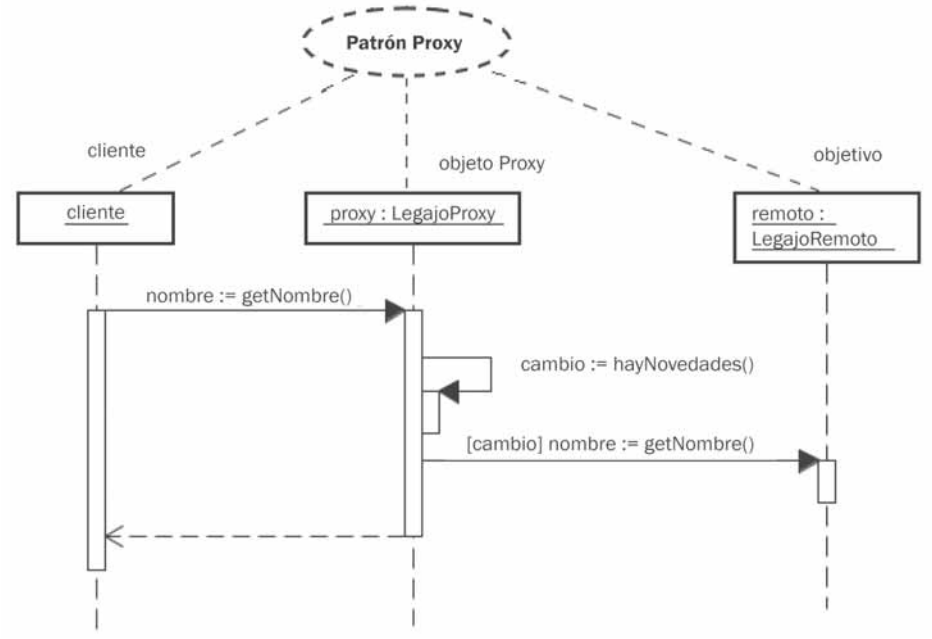

Figura 7.34 Diagrama de secuencia de un `Proxy`.

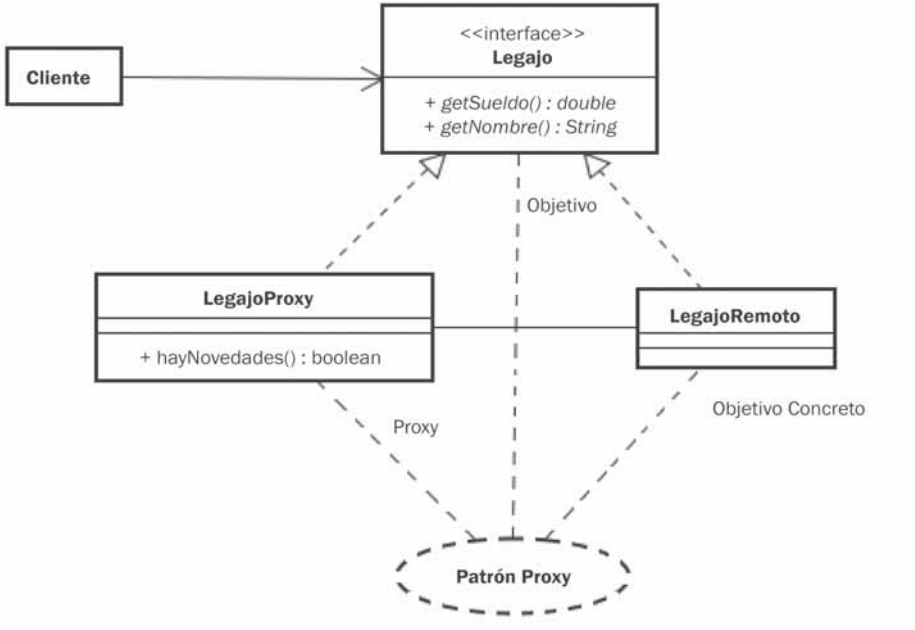

Figura 7.35 Diagrama de clases de un `Proxy`.

Si bien las colaboraciones son una buena herramienta para modelar patrones, son contados los casos de profesionales que las usan para ello.

Es más común que se hagan diagramas de interacción y de clases por separado, con una descripción textual.

Por ejemplo, las figuras 7.36 y 7.37 muestran un diagrama de clases y uno de secuencia, respectivamente, para modelar el patrón `Composite` de la manera más habitual, separando ambos diagramas. El déficit de este modelo es que no estamos aclarando que se trata de una implementación de `Composite`, aunque esto podríamos hacerlo agregando notas.

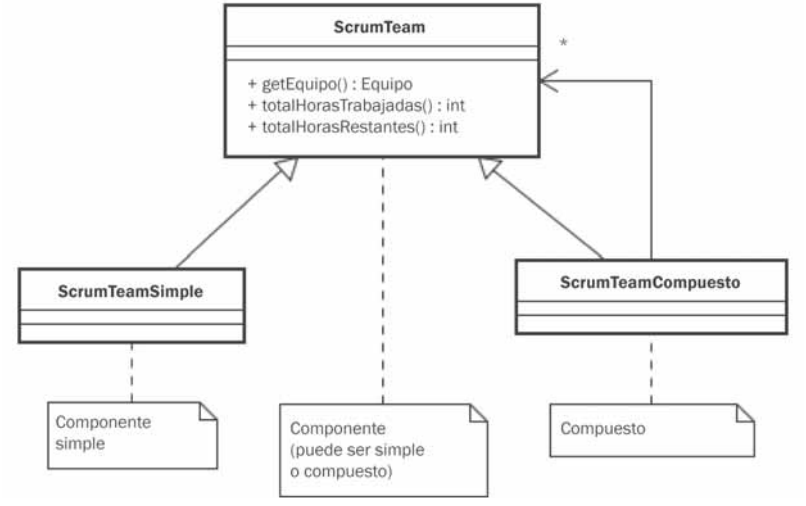

Figura 7.36 Diagrama de clases de un *Composite.*

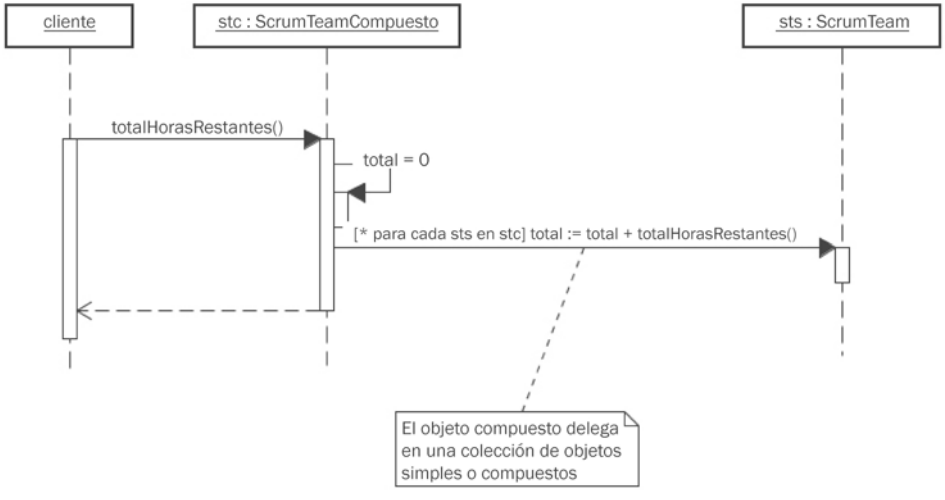

Figura 7.37 Diagrama de secuencia de un `Composite`.

Diagramas de tiempos

UML 2 introdujo un diagrama que permite modelar situaciones en las cuales es importante razonar sobre el paso del tiempo y restricciones de tiempo. Como era esperable, este diagrama se denomina **diagrama de tiempos**.

Se trata de un diagrama de interacción, tanto como los de secuencia, de comunicación y de visión global de interacción.

En la figura 7.38, se muestra un diagrama de tiempos simple, que muestra el mismo escenario de la figura 7.6.

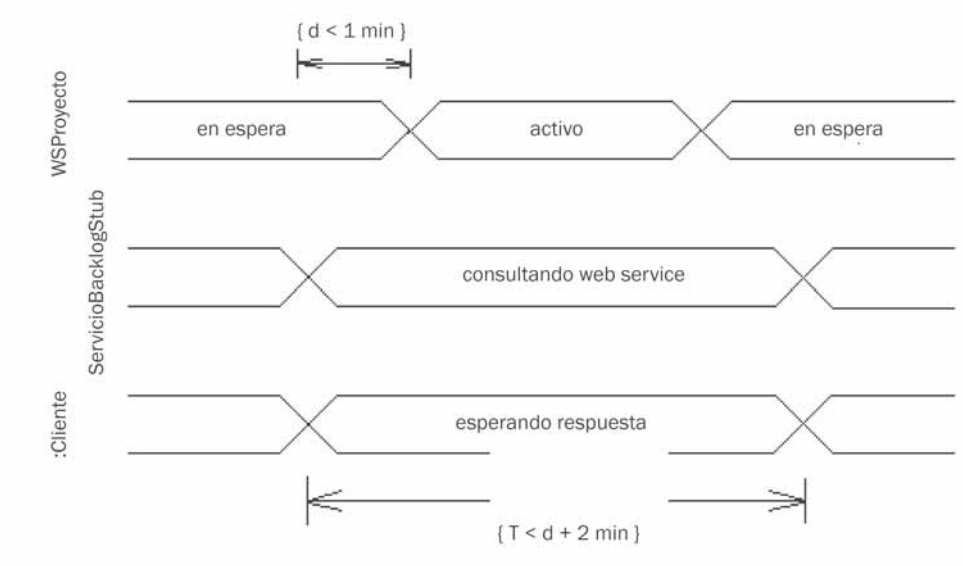

Figura 7.38 Diagrama de tiempos simple.

Como vemos, se están mostrando tres participantes: `WSProyecto`, `ServicioBacklogStub` y una instancia de `Cliente`. Los textos que hemos introducido para cada participante corresponden a estados de los participantes. Y lo que se coloca entre llaves son restricciones temporales. En nuestro diagrama estamos indicando que el tiempo que transcurre entre el cambio de estado de `ServicioBacklogStub` y el paso al estado `activo` de `WSProyecto` es de, al menos un minuto, tiempo éste que denotamos `d`. También se indica que el objeto `Cliente` estará en el estado `esperando respuesta` como máximo durante dos minutos adicionales al tiempo `d`.

Ésta es uno de los formatos más habituales para el diagrama de tiempos. Como ya dijimos, se trata de otro diagrama de interacción, como el de secuencia, aunque el paso del tiempo está indicado en forma horizontal y no vertical.

De todas maneras, notemos que no es un diagrama de interacción típico: la propia mención de estados es un tanto irregular para un diagrama de interacción. Por ello, hay muchas personas que lo consideran un diagrama mixto, entre las máquinas de estados y los diagramas de interacción.

Otra cosa que se puede notar es que en el diagrama de tiempos de la figura 7.38 no se ha mostrado el paso de mensajes entre los participantes, cosa que sí hicimos en el diagrama de secuencia de la figura 7.6. Si quisiésemos mostrar mensajes entre los objetos, podemos hacerlo, como se muestra en el diagrama de tiempos de la figura 7.39.

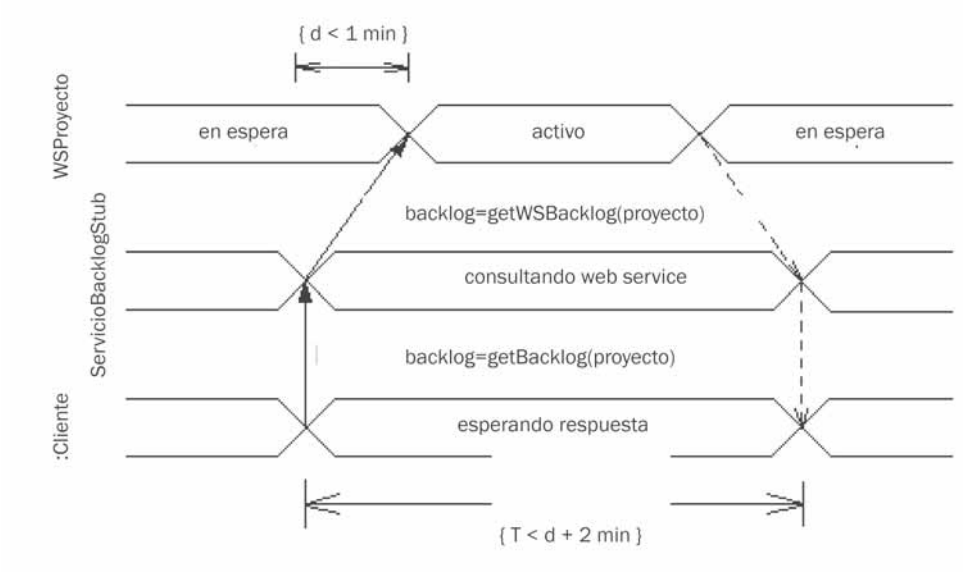

Figura 7.39 Diagrama de tiempos con mensajes.

El diagrama de tiempos se puede dibujar con dos notaciones distintas: una como la que vimos y, la otra, como la que se muestra en la figura 7.40. Esta última, no obstante la notación diferente, está modelando exactamente lo mismo que la figura 7.39.

La notación de la figura 7.40, cuyo significado es bastante obvio después de lo dicho, es más habitual cuando los estados que se enumeran de los participantes son discretos, más aún si son valores numéricos. La notación previa, en cambio, se puede usar con valores continuos.

Notemos que esta segunda notación que estamos viendo enfatiza aún más esta mezcla de modelos, que permite modelar tanto estados como interacciones.

Algunas cuestiones adicionales, propias de los diagramas de secuencia, también pueden llevarse al diagrama de tiempos, como la X para indicar que un objeto es destruido, los estereotipos de creación y destrucción, los mensajes asincrónicos, las guardas, etc.

En principio, todo lo que se modela con el diagrama de tiempos puede modelarse con el de secuencia. Por ello y porque la mayor parte de los profesionales se llevan mejor con el diagrama de secuencia, que además existe desde la primera versión de UML, es que el diagrama de tiempos no ha tenido mucho uso.

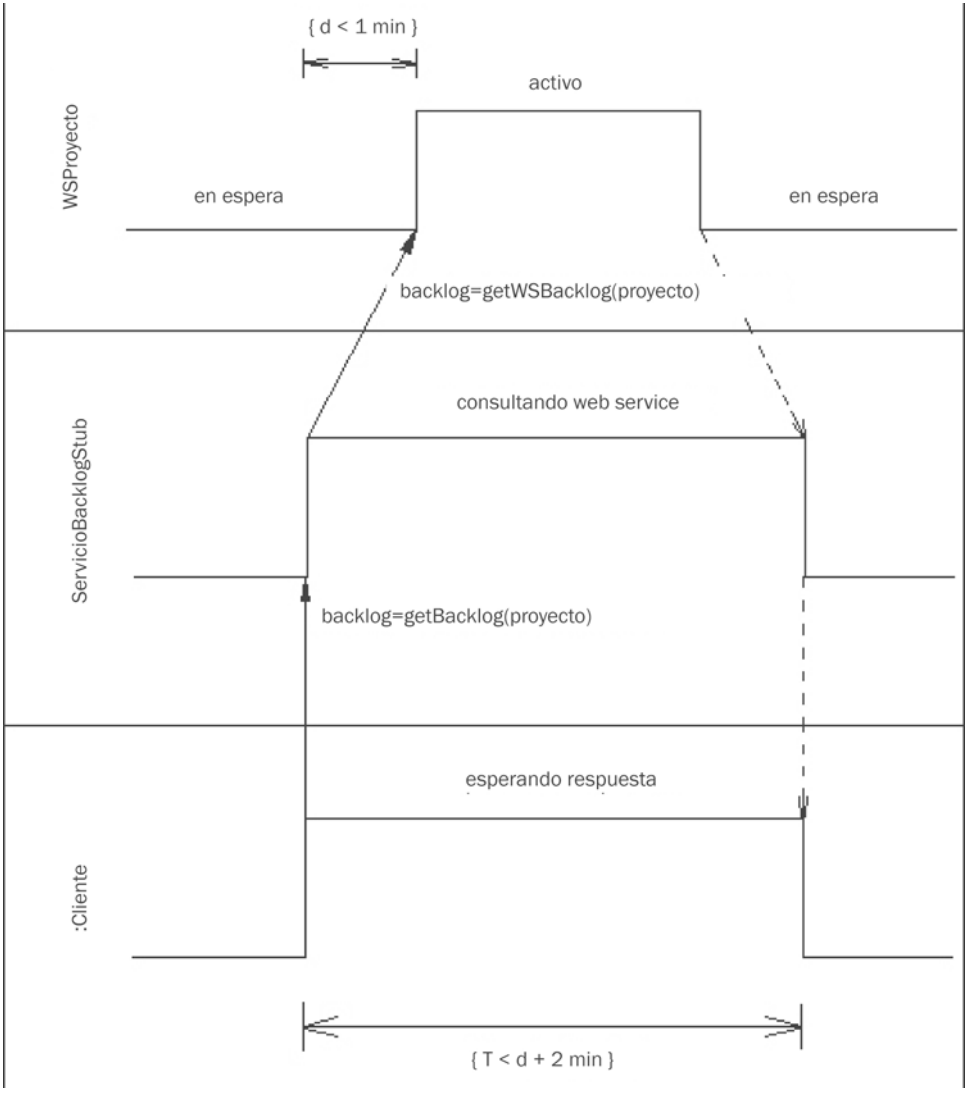

Figura 7.40 Diagrama de tiempos para estados discretos.

Por esta causa no lo hemos tratado con anterioridad, aunque la facilidad que brinda para modelar los estados de los participantes podría ser de gran utilidad.

DISEÑO Y CONSTRUCCIÓN CON UML

Las actividades de diseño detallado y construcción se realizan en forma conjunta en los métodos de desarrollo modernos, y es por esta razón que las hemos reunido.

Como hemos visto, existen muchas situaciones de uso de UML relacionadas con estas actividades. Es usual que se trabaje con modelos para discutir diseños entre desarrolladores y comunicar decisiones de diseño. Es también muy común ver mucho uso de diagramas como bosquejos hechos en pizarras para estas cuestiones. Más allá de que se hagan sobre una pizarra o sobre un producto de software, es importante que los desarrolladores los tengan siempre a mano, de modo tal que sirvan como referencia constante del diseño. También es habitual usar UML para documentar lo desarrollado, con vistas al mantenimiento posterior.

Es bastante claro que al usar UML en estas actividades, los diagramas adquieren una perspectiva más de implementación. Por eso mismo, salvo que UML se use para generar código, no hay que dejarse llevar por la tentación de pasar todo a UML, ya que –de hacerse– vamos a encontrarnos con una cantidad enorme de modelos imposible de manejar y mantener actualizados. En consecuencia hay una recomendación de varios expertos que dicen que solamente hay que guardar aquellos diagramas que luego sepamos que sea posible y tenga sentido mantener.

Adicionalmente, hay cosas que no se deben hacer. Por ejemplo, existen personas que usan diagramas de actividades para modelar código, tal vez porque los vean parecidos a los antiguos diagramas de flujo. Sin embargo, esto no conviene, por las mismas razones que no convienen los diagramas de flujo: el código de por sí, si es de buena calidad, es más explicativo que un diagrama.

8

OTRAS DISCIPLINAS

PRUEBAS

UML no provee modelos para pruebas de producto. Sin embargo, hay algunos modelos de UML que pueden servir como auxiliares de las pruebas y, otros, para derivar pruebas.

De todas maneras, convengamos en que no es lo más usual usar UML en las actividades de pruebas.

Casos de prueba

Los casos de prueba se suelen derivar de los escenarios de los casos de uso. Si bien los diagramas de casos de uso –ya de por sí limitados para la modelización de requisitos– no sirven para especificar pruebas, los casos de uso textuales pueden ser de ayuda.

Tomemos, por ejemplo, el escenario definido en el capítulo 4, que reproducimos para comodidad del lector:

```
El usuario solicita dar de alta una nueva empresa
El sistema muestra los datos a ser ingresados:
       Nombre (*)
       Domicilio
       Nombre del administrador del cliente (*)
       Mail del administrador del cliente (*)
       Teléfono de contacto (*)
```

```
    El usuario completa los campos:

           Nombre: "Desarrolladores del Norte SA"

           Domicilio: "Av. Boyacá 22345 – Esquel – Chubut"

           Nombre del administrador del cliente: "Juan Pérez"

           Mail del administrador del cliente: "jperez@dncom"

           Teléfono de contacto: "02945-112564"

    El sistema valida los datos y muestra un error en el
formato del mail

    El usuario abandona la operación

    El sistema no cambia la base de datos.

    El sistema no genera un usuario y una clave para el
administrador del cliente.

    El sistema no envía un mail al administrador del
cliente.

    El sistema guarda en el log "El usuario Administrador
abandonó el alta de una empresa".
```

El mismo escenario es un caso de prueba negativo, y los testers podrían basarse en él para realizar sus pruebas. Esto está en línea con algunas metodologías de especificación de requisitos, sobre todo aquellas que hablan de especificar con ejemplos ya que, en las mismas, son los propios ejemplos que conforman la especificación los que luego se convertirán en casos de prueba.

Recordemos, incluso, que al hablar de *user stories* dijimos que se pueden complementar con pruebas de aceptación.

También el propio caso de uso podría servir para escribir escenarios de pruebas positivas y negativas adicionales. Si el caso de uso incluyese prototipos de pantallas, también podrían usarse éstas en los casos de pruebas.

Diagramas y pruebas

Los diagramas de UML también pueden servir para algunas pruebas.

Por ejemplo, los diagramas de actividades pueden sernos útiles para alguna prueba de caja blanca. La figura 8.1 muestra un grafo de flujo de caja blanca realizado con un diagrama de actividades. Para su mejor seguimiento, hicimos el diagrama que corresponde al diagrama de secuencia de la figura 7.5, cuyo código es el de la figura 7.6.

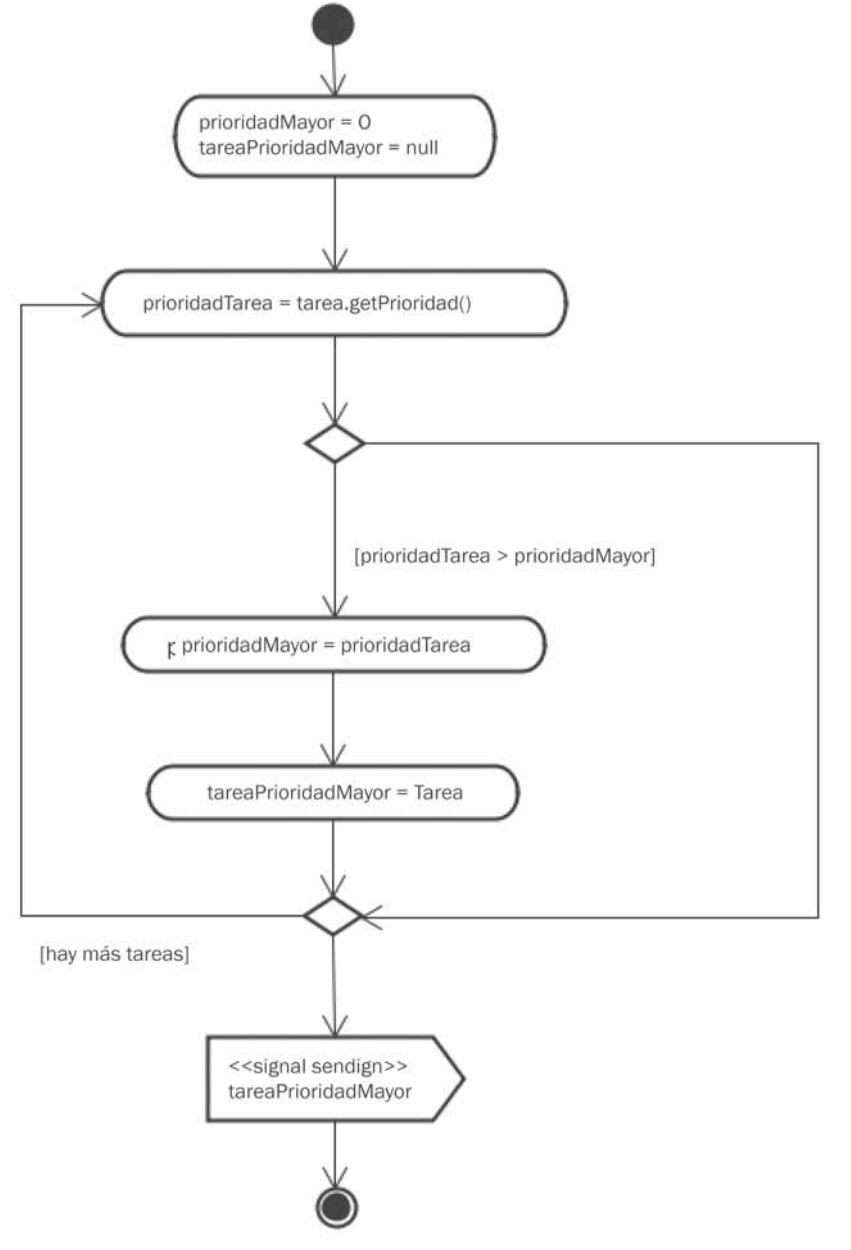

Figura 8.1 Grafo de flujo de actividades.

Sin embargo, muy poca gente hace los grafos de caja blanca con pretensiones de formalidad. En general, si se hacen, se los hace solamente como un bosquejo que se tira a la basura una vez utilizado. Lo más común es que no se hagan nunca.

Otros diagramas también podrían servir, aunque rara vez se utilizan.

Por ejemplo, si hubiésemos hecho diagramas de secuencia o de comunicación con anterioridad, mirarlos nos puede ayudar a entender problemas durante las pruebas de caja blanca, analizar defectos y realizar depuraciones. No obstante, en opinión de la mayor parte de los profesionales, es mejor manejarse con el código fuente directamente.

Además, los diagramas de clases también pueden ayudarnos a ver cómo es la relación entre clases, aunque de verdad son útiles si representan la realidad del sistema que se está probando. Tal vez una buena idea sería hacerlos por ingeniería inversa, para asegurarnos que son exactamente los diagramas de clases del sistema real.

DESPLIEGUE

El despliegue en hardware de una aplicación construida es una actividad más del desarrollo de software, al menos del software a medida. Y es una buena idea modelar el despliegue de la aplicación antes de realizarlo.

En principio, es una decisión de diseño y, por lo tanto, su modelado corresponde a aquélla actividad, como ya lo hemos discutido en el capítulo 6. Sin embargo, suele ser necesario explicitar mejor alguna de las cuestiones de despliegue en el momento concreto de tener que hacerlo.

Para ello, podemos usar los diagramas de despliegue que vimos en el capítulo de diseño de alto nivel, con mayor grado de detalle. Por ejemplo, la figura 8.2 muestra una variante un poco más detallada del diagrama realizado en la figura 6.12.

Como vemos, no hay mucha diferencia con la figura 6.12, salvo por el hecho de que, al estar planificando una instalación en un entorno concreto y ya totalmente conocido, podemos ser más específicos. Además, algunos nodos de despliegue los representamos como máquinas concretas –instancias– subrayando el nombre.

EVOLUCIÓN

Es habitual oír hablar del software como un producto maleable, modificable. Esta maleabilidad nos permite hacer varias cosas:

- Cambiar el producto debido a cambios de requisitos durante la propia ejecución del proyecto.
- Adaptar el producto conforme pasa el tiempo después de haber terminado el proyecto.
- Reutilizar parte o todo un producto de software para construir otros productos.

El término **evolución** se refiere a los dos últimos casos.

Notemos que el hecho de que el software sea maleable no quiere decir que sea sencillo modificarlo bien. Es decir, podemos cambiar el comportamiento de un producto de software, pero no necesariamente vamos a obtener un producto de la misma calidad de una manera fácil.

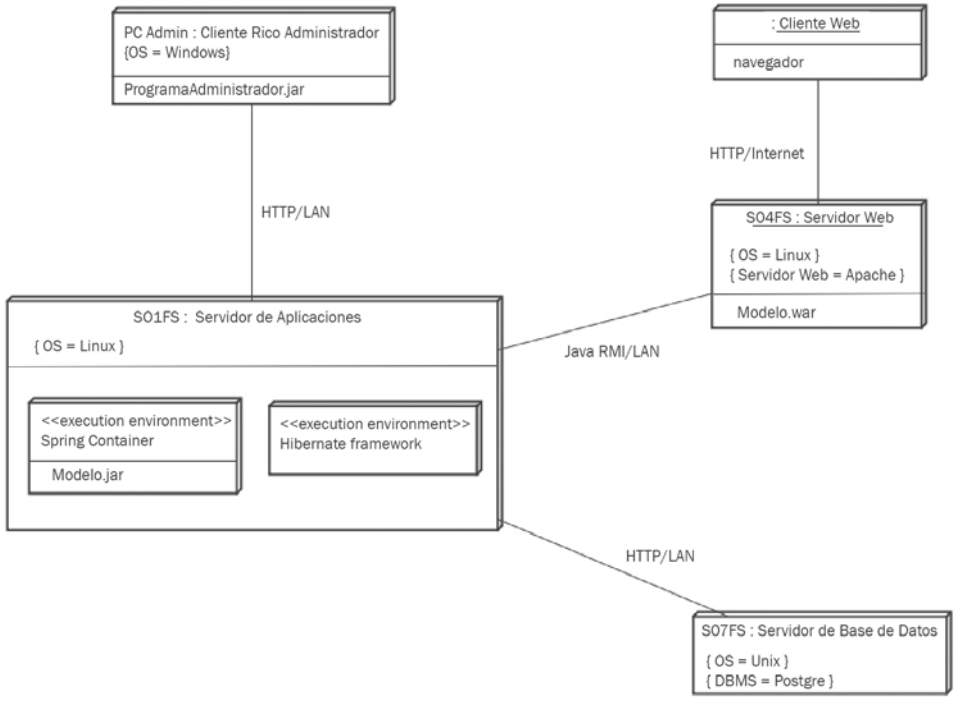

Figura 8.2 Diagrama de despliegue para despliegue.

Por lo tanto, la posibilidad no implica sencillez. En general, se da lo contrario. A medida que cambiamos un producto de software, el mismo se va degradando, los errores van aumentando y se van potenciando entre ellos. Las razones para que esto ocurra son varias, y no es éste el lugar para analizarlas, pero lo cierto es que el mantenimiento de software produce un aumento de la entropía, del desorden, que es necesario controlar.

No sólo el software que estamos modificando se degrada. Es habitual que durante las migraciones se pierdan datos o relaciones entre ellos y que otros sistemas que usaban las funciones del que se está cambiando empiecen a mostrar resultados inesperados.

¿Y en qué nos puede ayudar UML?

En realidad, un sistema de software bien documentado es más fácil de mantener. Los modelos de UML nos pueden servir tanto para darnos visión de las partes de un sistema, como puede ocurrir con los diagramas de paquetes o clases, como para permitirnos analizar comportamientos puntuales, como haríamos con diagramas de estados o interacciones. Los diagramas de componentes y de despliegue nos muestran conexiones entre partes físicas y sus interfaces. Pero los que hemos mencionado sólo son ejemplos. En la práctica, todos los modelos de UML son muy útiles para analizar un sistema antes y durante su mantenimiento.

Lo que hay que tener en cuenta es que, si queremos que los diagramas sigan sirviendo para futuras modificaciones, la documentación debe mantenerse actualizada.

Aquí volvemos a algo que ya dijimos. Como el costo de mantener todos los modelos actualizados puede llegar a ser muy alto, muchos profesionales desdeñan esta tarea y no la realizan. Por lo tanto, una buena práctica es mantener solamente aquellos diagramas que sepamos que se van a mantener actualizados, y eliminar los restantes. Al fin y al cabo, peor que no tener documentación es tenerla de mala calidad, con la falsa sensación de seguridad que solemos sentir en estos casos.

PLANIFICACIÓN, SEGUIMIENTO Y CONTROL

Si bien las tareas de planificación, seguimiento y control son inherentes a cualquier proyecto, y los de software no son la excepción, UML no ha previsto ninguna notación para llevar adelante estas disciplinas.

Hay muchos tipos de métricas, métodos y notaciones para soportar estas tareas en proyectos, sean estos o no proyectos de software. Por nombrar sólo algunos, recordemos los diagramas de Gantt, las redes PERT[1] y CPM[2]; los indicadores de Valor Ganado y su metodología subyacente; las WBS[3], etc. También existen muchos métodos y métricas asociadas a ciertos procesos de desarrollo de software, como los indicadores de velocidad y los Burndown Chart de Scrum. Tal vez por esta proliferación de métodos, herramientas y notaciones exitosas, los creadores de UML tuvieron el buen juicio de no agregar otras de su propia factura.

Dicho esto, reconozcamos también que los diagramas de UML son herramientas fundamentales para la comunicación, y la comunicación es esencial en proyectos de software, en especial en los proyectos grandes.

Pero no es ni necesario ni suficiente el lenguaje UML para realizar estas actividades de soporte al desarrollo de software.

UML MÁS ALLÁ DEL ANÁLISIS Y EL DISEÑO

UML se creó especialmente para modelar cuestiones de ingeniería de software y especialmente aspectos de análisis y diseño. Por esta razón, no es una notación especialmente dúctil para modelar la planificación y el control de proyectos, ni tampoco las actividades de desarrollo de software que exceden al análisis y al diseño.

1 Acrónimo en inglés de *Program Evaluation and Review Technique*, que se traduce como *técnica de revisión y evaluación de programas*.

2 Acrónimo en inglés de *Critical Path Method,* que se traduce como *método del camino crítico*.

3 Acrónimo en inglés de *Work Breakdown Structure*, que se traduce como *estructura de descomposición del trabajo*.

9

USOS DE LOS DIAGRAMAS DE UML

ELEMENTOS DE UML

Diagramas

La figura 9.1 muestra una taxonomía, realizada con diagramas de clases, de los distintos tipos de diagramas de UML.

Sin embargo, hay que destacar que esta figura es una simplificación, ya que varios tipos de diagramas pueden superponerse.

Además, hemos dicho que, a nuestro juicio, el diagrama de casos de uso es un diagrama estructural. Sin embargo, la especificación de UML, tal vez pensando en el hecho de que un caso de uso —no un diagrama de casos de uso— implica comportamiento, lo considera entre los diagramas de comportamiento. Por eso en la figura lo hemos colocado como un modelo mixto.

Algo similar ocurre con las colaboraciones que UML 2 considera un tipo especial de clasificador, pero que hemos separado por considerar que las clases son elementos estructurales, mientras que las colaboraciones son modelos que también pueden especificar aspectos de comportamiento.

Respecto de los diagramas de interacción, en líneas generales son intercambiables, y depende de las preferencias del profesional el uso de uno u otro (esto no es así en un ciento por ciento, pero lo es en lo fundamental). La especificación de UML dice que el de secuencia es el diagrama de interacción más común, y es así si se tiene en cuenta el uso que se hace del mismo.

Asimismo, lo que resulta relativamente extraño es la presencia de los diagramas de visión global de interacción entre los diagramas de interacción. A juicio del autor, se trata de un diagrama mixto, a medio camino entre el de actividades y los de interacción. Sin embargo, la declaración de la especificación de UML, diciendo que los modeladores no debieran interpretar los diagramas de visión global como diagramas

de actividades resulta tan intimidante que preferimos colocarlos en la categoría que dicha especificación les asigna. Tal vez sea el uso futuro el que defina su clasificación práctica.

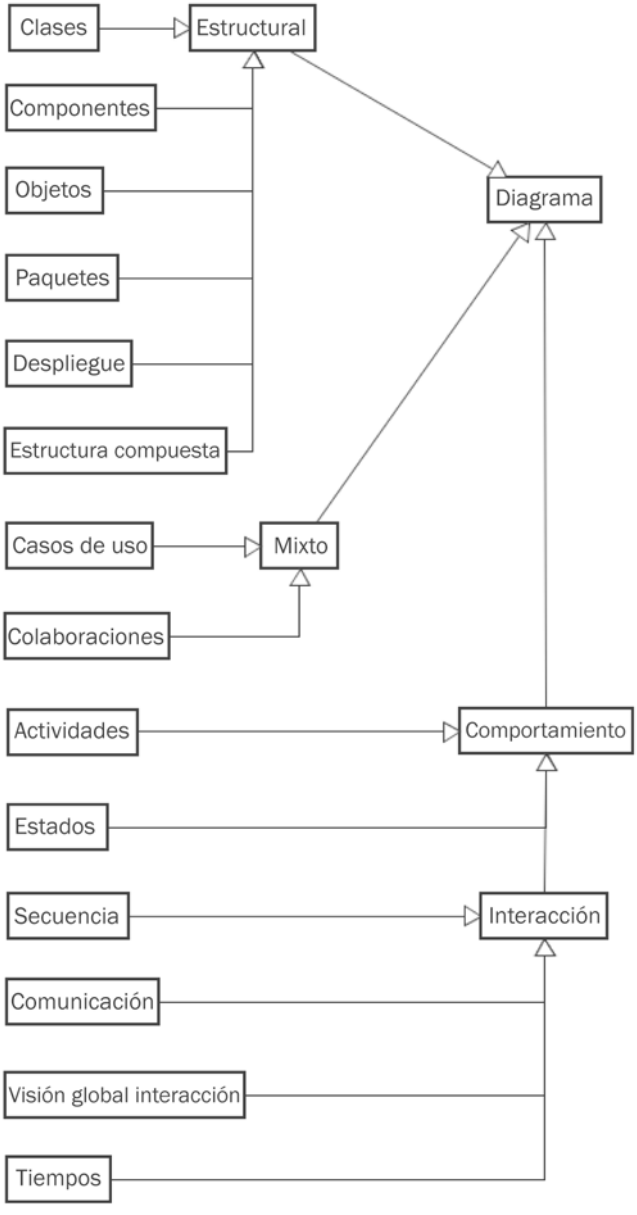

Figura 9.1 Taxonomía de diagramas.

Otros elementos

Pero UML no sólo define diagramas, sino algunas construcciones que se utilizan con los mismos. Si bien no podemos hacer aquí una enumeración exhaustiva, mencionaremos algunos elementos importantes.

Por ejemplo, los diagramas estructurales contienen una serie de elementos, que dependen del tipo de diagrama: clases en los diagramas de clases, paquetes y clases en los de paquetes, componentes e interfaces en los de componentes, nodos y artefactos en los de despliegue, varios otros en los de estructura compuesta.

Pero los distintos elementos de los diagramas estructurales se encuentran conectados por vínculos que indican dependencias de distintos tipos. La figura 9.2 muestra los conectores habituales en los diagramas estructurales.

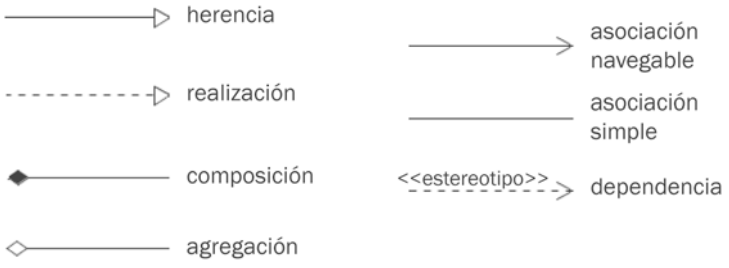

Figura 9.2 Conectores en diagramas estructurales.

En realidad, hay varios conectores que son casos particulares de otros. La figura 9.3 ilustra una taxonomía de conectores.

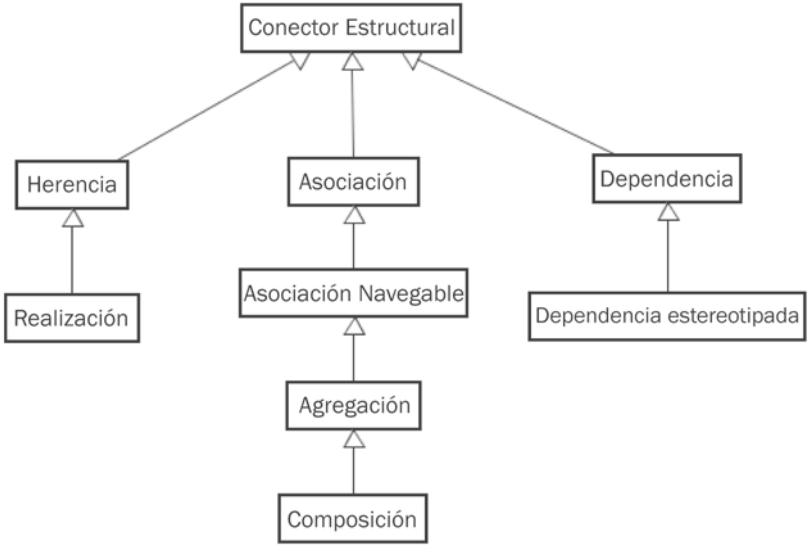

Figura 9.3 Taxonomía de conectores en diagramas estructurales.

No es necesariamente conveniente o posible usar todos los conectores en cualquier diagrama. El cuadro 9.1 muestra qué usos dar a los mismos.

CONECTOR	DIAGRAMAS EN LOS QUE SE USA	USO HABITUAL	USOS MENOS HABITUALES
Herencia.	Clases.	Relaciones de generalización-especialización entre clases.	Diagramas de paquetes, de componentes y de despliegue.
Realización.	Clases.	Relaciones de realización de una interfaz por una clase.	Diagramas de componentes.
Composición.	Clases.	Relaciones de composición entre instancias de dos clases.	Diagramas de paquetes, de componentes y de despliegue.
Agregación.	Clases.	Relaciones de agregación entre instancias de dos clases.	Diagramas de paquetes, de componentes y de despliegue.
Asociación navegable.	Clases, componentes.	Relaciones de asociación entre instancias de dos clases o componentes, con indicación de la dependencia.	Diagramas de estructura compuesta y de paquetes.
Asociación simple.	Clases, componentes, objetos, despliegue, estructura compuesta.	Relaciones de asociación entre instancias de dos clasificadores, sin indicación de la dependencia.	Diagrama de paquetes.
Dependencia.	Todos los estructurales.	Dependencias más débiles que una asociación. Un estereotipo puede especificar el tipo de dependencia.	

Cuadro 9.1 Usos de los conectores en diagramas estructurales.

De la misma manera que en los diagramas estructurales, los diagramas de comportamiento utilizan elementos propios de cada uno: estados, actividades, señales,

participantes, etc. Y, al igual que en los diagramas estructurales, en los diagramas de comportamiento suele haber vínculos entre los elementos que se modelan con conectores, que son los que se observan en la figura 9.4.

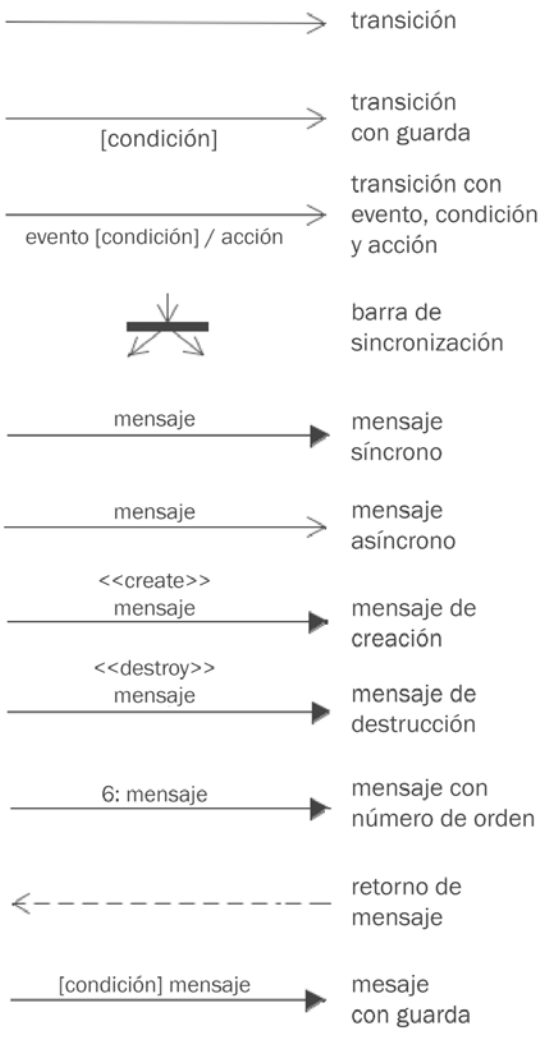

Figura 9.4 Conectores en diagramas de comportamiento.

Como pasaba con los diagramas estructurales, hay conectores que son casos particulares de otros. La figura 9.5 ejemplifica una taxonomía de conectores.

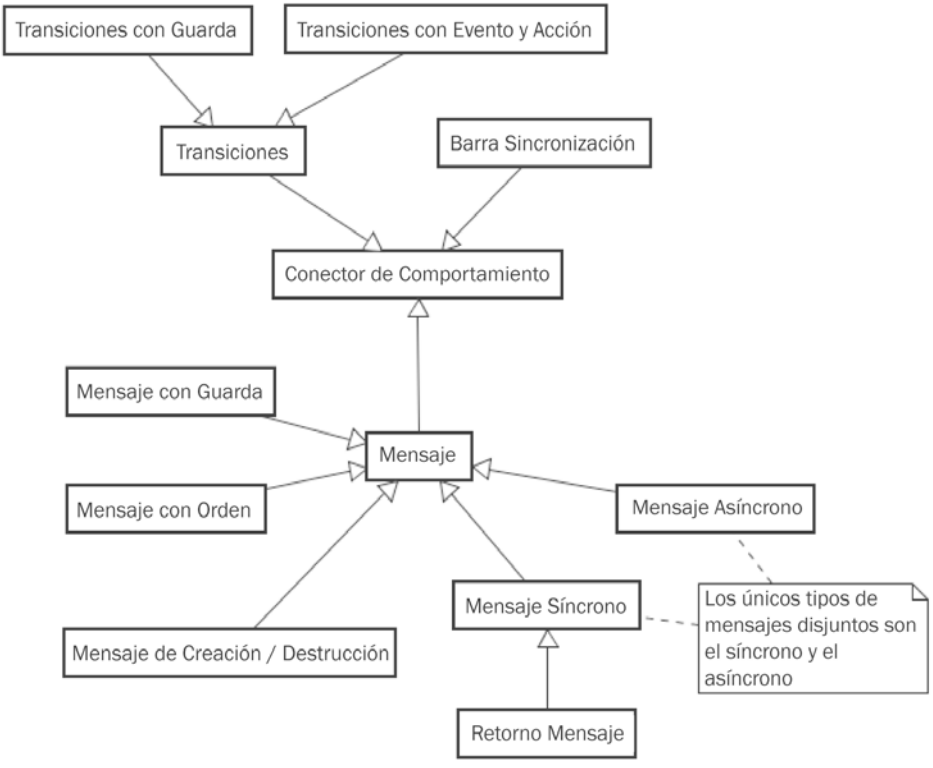

Figura 9.5 Taxonomía de conectores en diagramas de comportamiento.

El cuadro 9.2 muestra qué usos dar a los conectores.

CONECTOR	DIAGRAMAS EN LOS QUE SE USA	USO HABITUAL	USOS MENOS HABITUALES
Transiciones.	Estados, actividades.	Transiciones incondicionales entre acciones, actividades, estados, pseudo-estados y nodos.	Diagramas de visión global de interacción.
Transiciones con guarda.	Estados, actividades.	Transiciones condicionales entre acciones, actividades, estados, pseudo-estados y nodos.	Diagramas de visión global de interacción.

CONECTOR	DIAGRAMAS EN LOS QUE SE USA	USO HABITUAL	USOS MENOS HABITUALES
Transiciones con indicación de evento, condición y/o acción.	Estados.	Transiciones condicionales entre estados, pseudo-estados y nodos, con indicación de evento, condición y/o acción asociados a la transición.	
Barras de sincronización	Actividades, visión global de interacción	Comienzo o fin de concurrencia.	Diagramas de estados.
Mensaje síncrono.	Todos los de interacción.	Mensaje enviado a un participante que debe esperar respuesta del receptor.	
Mensaje asíncrono.	Todos los de interacción.	Mensaje enviado a un participante que no está forzado a esperar respuesta del recepto.	
Mensaje de creación o destrucción.	Todos los de interacción.	Mensaje enviado a un participante para crearlo o destruirlo. El estereotipo que se utilice indica si se trata de una creación o una destrucción.	
Mensaje con indicación de orden.	Comunicación.	Mensaje enviado, con un número que indica ordenamiento temporal y/o anidamiento.	Diagramas de secuencia y de tiempos.
Retorno de mensaje.	Secuencia.	Recepción del final del mensaje por parte del emisor.	Diagramas de tiempos.
Mensaje con guarda.	Todos los de interacción.	Mensaje cuya emisión está sujeta al cumplimiento de una condición.	

Cuadro 9.2. Usos de los conectores en diagramas de comportamiento.

Los diagramas mixtos implican otras construcciones: actores y casos de uso en diagramas de casos de uso; colaboraciones y diagramas varios en los de colaboración. Los conectores que se usan suelen ser los mismos de los diagramas estructurales.

Los usos de los conectores en diagramas mixtos se ilustran en la cuadro 9.3.

CONECTOR	DIAGRAMAS EN LOS QUE SE USA	USO HABITUAL	USOS MENOS HABITUALES
Herencia.	Casos de uso.	Relaciones de generalización-especialización entre actores.	Entre casos de uso y entre colaboraciones.
Asociación navegable.	Casos de uso.	Relaciones de asociación entre un actor y un caso de uso.	Asociaciones entre colaboraciones.
Asociación simple.	Casos de uso.	Relaciones de asociación entre un actor y un caso de uso.	Asociaciones entre colaboraciones.
Dependencia.	Casos de uso, Colaboraciones.	Dependencias más débiles que una asociación. Entre casos de uso, un estereotipo debe especificar el tipo de dependencia.	

Cuadro 9.3 Usos de los conectores en diagramas mixtos.

DISCIPLINAS Y DIAGRAMAS

A continuación, se enumeran los diagramas que recomendamos utilizar para cada una de las disciplinas operativas de desarrollo de software. De alguna manera, esta lista se podría ver como un resumen de los capítulos anteriores.

Para la actividad de requisitos:

- Diagramas de casos de uso, para contexto del sistema, relaciones entre requisitos, relaciones de requisitos y actores.
- Diagramas de actividades, para especificación de pasos de casos de uso y escenarios, con una notación que enfatice el flujo de acciones.
- Diagramas de secuencia de sistema, para modelar interacciones del sistema con el medio, cuando se quiera enfatizar el paso de mensajes entre uno y otro.
- Diagramas de clases, para modelar el dominio del sistema a nivel conceptual.

Para la actividad de análisis:

- Diagramas de clases con responsabilidades, para analizar las responsabilidades de los participantes y las dependencias entre los mismos.

- Diagramas de objetos, para analizar relaciones entre participantes concretos de un caso de uso o del sistema.
- Diagramas de comunicación, para modelar el comportamiento de los participantes en determinados escenarios.
- Diagramas de secuencia, como un sucedáneo de los diagramas de comunicación, en los casos en que sea interesante enfatizar el paso del tiempo.
- Diagramas de visión global de interacción, como un compromiso entre los diagramas de secuencia y los de actividades, para modelar comportamiento de participantes en un escenario, con bifurcaciones condicionales.
- Diagramas de clases con comportamiento, para modelar relaciones entre clases de participantes basadas en el comportamiento de los mismos.
- Diagramas de estados, para analizar cómo afecta un escenario a los cambios de estado de un participante particular, y los eventos que provocan las transiciones.
- Diagramas de clases estructurales, para modelar atributos de los participantes y relaciones entre participantes y clases basadas en cuestiones estructurales.
- Diagramas de objetos con atributos, cuando se quiera representar el estado y los vínculos entre participantes en un momento dado.

Para la actividad de diseño:

- Diagramas de paquetes, para modelar relaciones y dependencias entre subsistemas, *frameworks* y/o bibliotecas.
- Diagramas de paquetes, para modelar agrupaciones de clases.
- Diagramas de componentes, para modelar relaciones entre componentes de software y sus interfaces.
- Diagramas de despliegue, para modelar el despliegue de artefactos de software en diferentes nodos de hardware o entornos de ejecución.
- Diagramas de estados, para modelar cómo se producen los cambios de un estado de objeto software en un escenario de ejecución.
- Diagramas de secuencia, para modelar el paso de mensajes entre varios objetos software, durante la ejecución de un escenario.
- Diagramas de comunicación, como sucedáneo de los diagramas de secuencia, cuando el paso del tiempo no sea un aspecto a destacar.
- Diagramas de clases, para representar clases software[1] en forma estática, con la posibilidad de incluir aspectos internos y cuestiones de implementación, y sus interrelaciones.

- Diagramas de objetos, cuando se quiera representar el estado y los vínculos entre objetos en un momento dado de la ejecución de la aplicación.
- Colaboraciones, cuando se desee representar en conjunto aspectos estructurales y de comportamiento, o cuando se quieran vincular diagramas estáticos con otros dinámicos.
- Colaboraciones, para modelar patrones de diseño.
- Diagramas de tiempos, para modelar situaciones de comunicaciones entre objetos en los cuales el tiempo o las restricciones de tiempos cumplan un papel central.
- Diagramas de tiempos, para modelar situaciones de comunicaciones entre objetos en los cuales se deseen mostrar cambios de estados.
- Diagramas de estructura compuesta, cuando se desee mostrar aspectos estáticos internos a una clase o un componente, más las interfaces que permiten la comunicación con otros componentes.

Para la actividad de construcción:

- Diagramas de clases, generados mediante ingeniería inversa, para tener una visión de las clases y las relaciones entre las mismas.
- Diagramas de paquetes, generados mediante ingeniería inversa, para analizar acoplamiento entre paquetes.
- Diagramas de secuencia, generados mediante ingeniería inversa, para analizar código.
- Cualquiera de los anteriores, generados mediante ingeniería inversa, para utilizar como documentación *post-mortem*[2].

Para la actividad de pruebas:

- Diagramas de actividades, para modelar flujos de pruebas de caja blanca.
- Diagramas de clases, generados mediante ingeniería inversa, para tener una visión de las clases y relaciones.
- Diagramas de estados, para comprender los estados de un objeto sometido a pruebas, más los eventos que provocan los cambios de estados.

Para la actividad de despliegue:

- Diagrama de despliegue detallado, con indicación de nodos concretos, y en general como documentación a guardar del despliegue de la aplicación.

TRAZABILIDAD ENTRE MODELOS

Trazabilidad, en la acepción que vamos a usar, se refiere a conocer cómo podemos obtener un modelo a partir de otros. Si vamos en el sentido de mayor a menor abstracción, la denominamos **trazabilidad directa**. Si, en cambio, queremos conocer la traza en el sentido de mayor abstracción, la llamamos **trazabilidad indirecta**.

Si bien UML no hace ninguna recomendación de proceso, y el tema de cómo obtener los distintos modelos es fuertemente dependiente del proceso de desarrollo, hay algunos modelos que podemos obtener a partir de otros.

Por ejemplo, el diagrama de la figura 9.6 es habitual en los libros y documentación del UP.

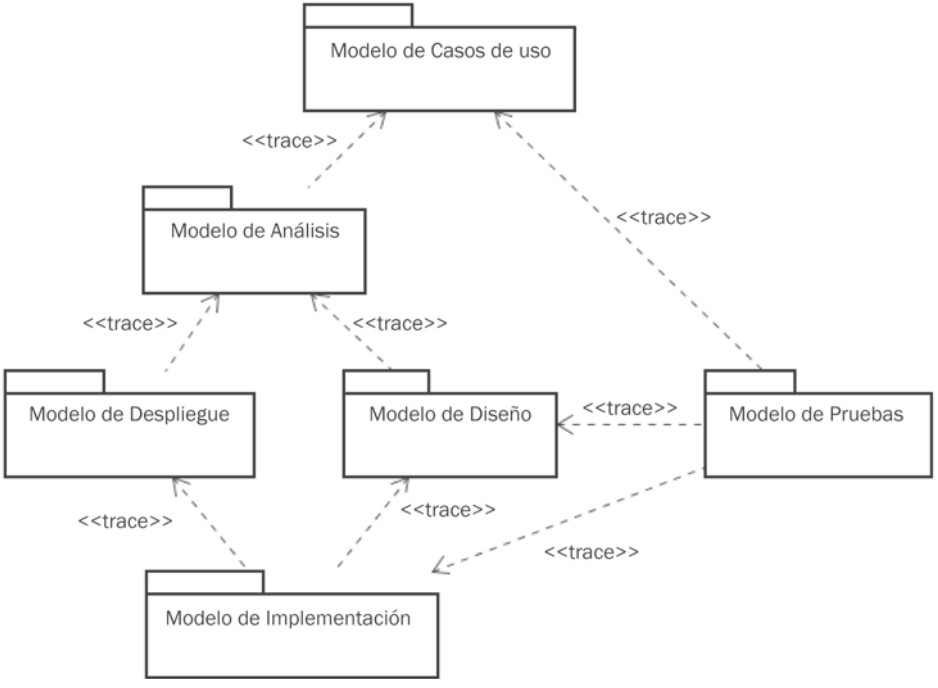

Figura 9.6 Trazabilidad de modelos en UP.

El diagrama de la figura 9.7 muestra la trazabilidad de modelos en el esquema de MDD.

Los métodos ágiles son mucho más reacios a especificar cómo debe ser la trazabilidad, sobre todo porque les parece que implica un desarrollo por etapas y con un análisis y diseño definidos de antemano.

Pero no tiene porqué ser así. Por ejemplo, para *Scrum*, la trazabilidad entre el *Sprint Backlog* y el *Project Backlog* se puede expresar como en la figura 9.8.

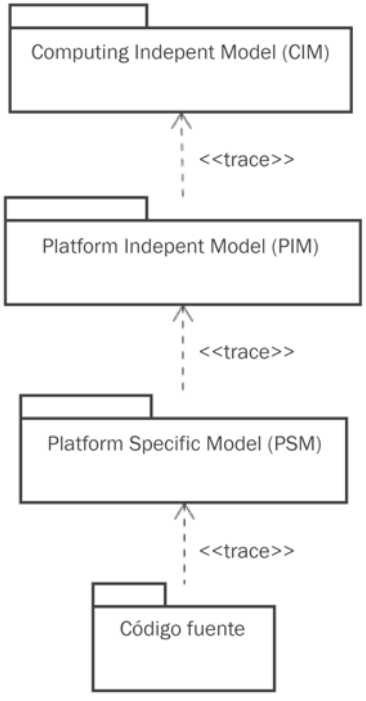

Figura 9.7 Trazabilidad de modelos en MDD.

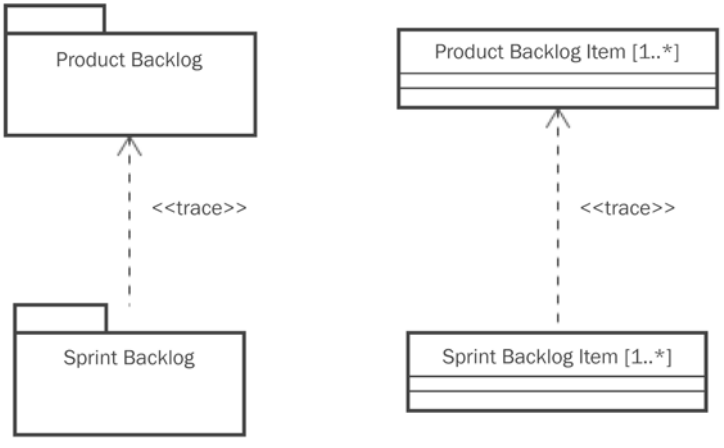

Figura 9.8 Trazabilidad en *Scrum.*

Asimismo, hay muchos modelos que admiten que se les defina la trazabilidad. Por ejemplo, la trazabilidad para llegar a clases software podemos indicarla como en la figura 9.9, en la cual hemos eliminado el estereotipo `<<trace>>` para simplificar el diagrama, y porque lo único que se está modelando es la trazabilidad.

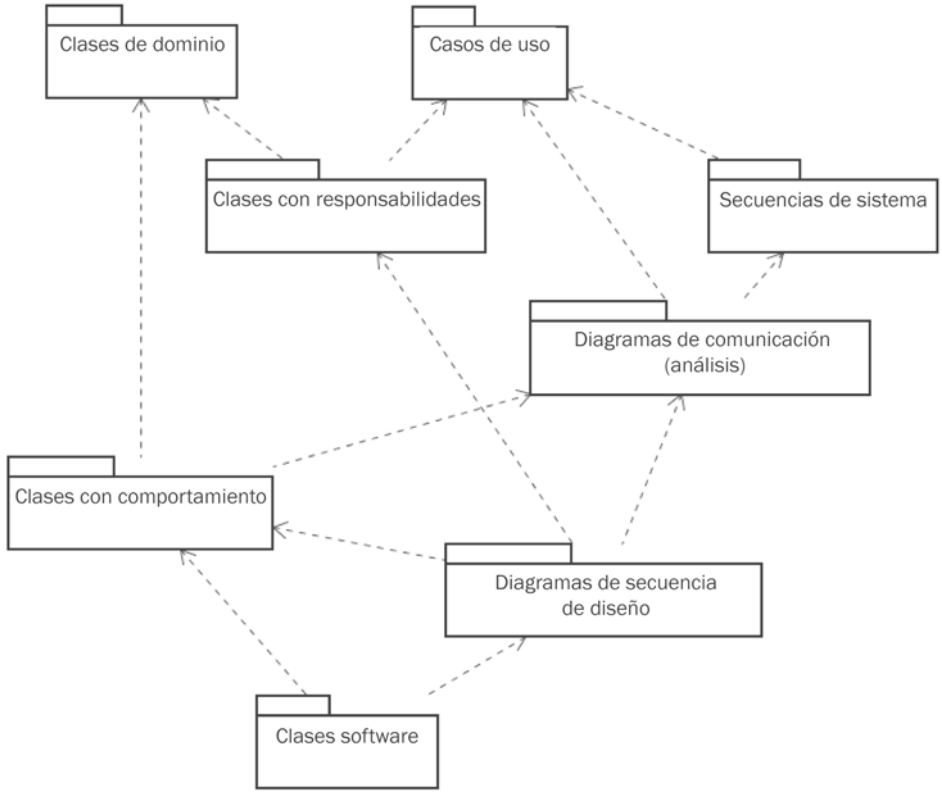

Figura 9.9 Trazabilidad hasta las clases software.

A pesar de todo lo dicho, lo cierto es que definir la trazabilidad con diagramas UML resulta demasiado imperativo para mi gusto, y es propio de cada proceso de desarrollo definirla o no. Cualquier persona sensata estaría de acuerdo conmigo en que los diagramas de las figuras anteriores son demasiado prescriptivos para resultar de uso general, y habrá quienes estén de acuerdo y quienes no con cada uno de ellos.

Además, es dudoso que sirva de mucho hacer estos modelos. Por ejemplo, la figura 9.8 no dice nada que no quede mejor expresado en palabras. La figura 9.9 dice bastante más, e incluso muestra varios caminos posibles, no excluyentes, para encontrar las clases software, pero resulta poco menos que incomprensible si no se la acompaña de mucho texto explicativo. Efectivamente, el diagrama de la figura 9.9 está mostrando algunas posibilidades en línea de lo ya explicado a lo largo del libro de cómo se pueden obtener los distintos diagramas. Y, en todos los casos vistos, son modelos de procesos más que de proyectos de software.

Lo realmente relevante de este apartado es entender que UML puede servir también para mostrar trazabilidad de artefactos. Y que cada profesional le podrá encontrar usos a esta posibilidad, o no.

Si bien hemos marcado trazabilidad usando solamente diagramas de paquetes, y eventualmente de clases, UML no impide que indiquemos trazabilidad en modelos heterogéneos.

Aunque no lo hemos hecho, podríamos indicar **trazabilidad bidireccional**, esto es, trazabilidad directa e inversa en el mismo diagrama. Sin embargo, resulta difícil encontrar ventajas de hacerlo en forma gráfica, salvo que nos sirva para especificar cómo o con qué herramientas se manejará esa trazabilidad.

DIAGRAMAS Y USOS

Así como más arriba analizamos los diagramas que se suelen utilizar para cada actividad de desarrollo de software, en el cuadro 9.4, mostramos una tabla que es la inversa de aquella lista: son los usos que se pueden dar a cada uno de los diagramas de UML.

Hemos ordenado los diagramas desde los de mayor a los de menor difusión. Es bastante sintomático cómo hay tantas herramientas —que dicen soportar UML 2— que ignoran determinados tipos de modelos. De hecho, en los diagramas que figuran en este libro, algunos de ellos debieron ser retocados a mano porque no hay herramientas suficientemente buenas para realizarlos siguiendo el estándar. Con tantos profesionales ignorando activamente algunas construcciones del lenguaje, más el desdén de tantas herramientas, queda en evidencia que no todo UML tiene la misma utilidad, o al menos la misma habitualidad.

DIAGRAMA	ACTIVIDAD	PARA MODELAR	OBSERVACIONES
Clases	Requisitos.	dominio del negocio.	Sólo representar nombres de clases y relaciones, sin navegabilidad.
	Análisis.	Clases conceptuales y relaciones.	Representar clases, relaciones y navegabilidad. Puede ser orientado a responsabilidades, a comportamiento (métodos), o a estructura (atributos).
	Diseño y construcción.	Clases software.	En el nivel de detalle que nos interese.
	Construcción.	Clases software para documentación *post-mortem*.	Obtenido por ingeniería inversa, en el nivel de detalle que nos interese.
	Pruebas.	Clases software, como documentación de referencia.	Al nivel de detalle que interese modelar.

DIAGRAMA	ACTIVIDAD	PARA MODELAR	OBSERVACIONES
Secuencia	Requisitos.	Detalles de casos de uso o *user-stories*.	Diagrama de secuencia de sistema.
	Análisis.	Solamente como sucedáneo de los diagramas de comunicación, si se desea enfatizar el paso del tiempo.	Orientado a mostrar mensajes entre participantes.
	Diseño y construcción.	Paso de mensajes entre varios objetos software, durante la ejecución de un escenario.	En el nivel de detalle que nos interese.
	Construcción.	Escenarios en el software concreto, para documentación *post-mortem*	Obtenido por ingeniería inversa, en el nivel de detalle que nos interese.
Estados	Requisitos.	Estados del sistema o de partes del mismo (no en este libro).	En el nivel de detalle que nos interese.
	Análisis.	Cambios de estado de un participante particular, y los eventos que provocan las transiciones.	En el nivel de detalle que nos interese.
	Diseño y construcción.	Cambios de estado de un objeto software, y los eventos software que provocan las transiciones.	En el nivel de detalle que nos interese.
	Pruebas.	Cambios de estados y eventos de objetos sometidos a pruebas.	Si se hizo antes. Si no, no tiene sentido.
Actividades	Requisitos.	Especificación de pasos de casos de uso y escenarios, con una notación que enfatice el flujo de acciones.	Cuidar el nivel de detalle para que sirva como herramienta de comunicación.
	Requisitos y análisis.	Flujos de negocio (no en este libro).	Al nivel de detalle que nos interese.
	Pruebas.	Flujos de prueba de caja blanca.	Uso poco habitual.

DIAGRAMA	ACTIVIDAD	PARA MODELAR	OBSERVACIONES
Comunicación	Análisis.	Comportamiento de los participantes en un escenario.	Orientado a mostrar mensajes entre participantes.
	Diseño y construcción.	Solamente como sucedáneo de los diagramas de secuencia, si se desean enfatizar relaciones entre objetos.	En el nivel de detalle que nos interese.
	Pruebas.	Objetos software que están siendo probados, y los mensajes entre ellos.	Si se hizo antes. Si no, no tiene sentido.
Paquetes	Requisitos.	Agrupaciones de casos de uso o *user-stories*.	Uso poco habitual.
	Análisis.	Agrupaciones de clases conceptuales.	Uso poco habitual.
	Diseño y construcción.	Relaciones y dependencias entre subsistemas, frameworks y bibliotecas.	Solamente dibujar paquetes con sus dependencias.
	Diseño y construcción.	Agrupaciones de clases software.	Excelente para analizar acoplamiento
	Construcción.	Agrupaciones de clases software y acoplamiento para documentación *post-mortem*.	Obtenido por ingeniería inversa, al nivel de detalle que nos interese.
Objetos (estático)	Análisis.	Relaciones entre participantes concretos de un escenario.	Uso poco habitual.
	Diseño y construcción.	Estado y vínculos entre objetos en un momento dado de la ejecución de la aplicación.	En el nivel de detalle que nos interese.
Despliegue	Diseño y construcción.	Nodos de hardware o entornos de ejecución, sus relaciones, y artefactos, como se espera desplegarlos.	En el nivel de tipos de nodos y artefactos.
	Despliegue.	Nodos de hardware o entornos de ejecución, sus relaciones, y artefactos, como están siendo desplegados.	En el nivel de nodos y artefactos concretos

DIAGRAMA	ACTIVIDAD	PARA MODELAR	OBSERVACIONES
Componentes	Diseño y construcción.	Relaciones entre componentes de software y sus interfaces.	En el nivel de detalle que nos interese.
Casos de uso	Requisitos.	Contexto del sistema, relaciones entre requisitos, relaciones de requisitos y actores.	Cuidar el nivel de detalle para que sirva como herramienta de comunicación.
Colaboraciones	Diseño y construcción.	Aspectos estructurales y de comportamiento en forma conjunta.	En el nivel de detalle que nos interese.
	Diseño y construcción.	Patrones de diseño.	Cuidar el nivel de detalle para que sirva como herramienta de comunicación.
Visión global de la interacción	Requisitos.	Detalles de casos de uso o *user-stories*, combinando actividades e interacciones.	Uso poco habitual.
	Diseño y construcción.	Solamente como sucedáneo de los diagramas de secuencia, si se desea enfatizar actividades globales.	Uso poco habitual.
	Pruebas.	Solamente como sucedáneo de los diagramas de secuencia y de actividades para flujos de prueba de caja blanca.	Uso poco habitual.
Tiempos	Diseño y construcción.	Solamente como sucedáneo de los diagramas de secuencia, si interesa modelar restricciones temporales.	Uso poco habitual, pero interesante.
	Pruebas	Solamente como sucedáneo de los diagramas de secuencia, si interesa modelar restricciones temporales.	Uso poco habitual.
Estructura compuesta	Diseño y construcción	Estructura interna de una clase o componente.	Uso poco habitual.

Cuadro 9.4 Diagramas y sus usos en cada actividad.

USO Y ABUSO

Como hemos visto hasta ahora en este capítulo, UML se puede usar en casi todas las disciplinas del desarrollo de software, y todos sus diagramas son de utilidad. También hemos visto que hay diagramas que se utilizan mucho más que otros, como el de clases y el de secuencia.

Lo importante de todo esto es tomar conciencia de que estamos en presencia de una herramienta. Y que, como toda herramienta, hay que saber cuándo usarla y cuándo evitarla. No porque existan los diagramas de despliegue vamos a hacer diagramas de despliegue de aplicaciones *stand-alone,*[3] por ejemplo. Tampoco tiene sentido diagramar casos de uso de todo el sistema si esto no mejora la comunicación con los usuarios.

En las organizaciones en las que los roles de las personas están definidos de manera muy excluyente, se suele ver a UML de maneras distintas según el rol. Probablemente, un analista se focalice en diagramas de clases conceptuales y de actividades, presentados de manera muy formal, mientras que un programador tal vez se interese más en los diagramas de clases software realizados sobre una pizarra.

En ocasiones, el uso de UML viene impuesto por un proceso de desarrollo. Si bien esto a veces ocurre, en general no es bueno hacer diagramas solamente porque lo pide un proceso. Los modelos deben surgir de las necesidades concretas, sean éstas comunicar humanos entre sí o, en el otro extremo, generar aplicaciones a partir de los mismos.

MODELOS GUIANDO EL DESARROLLO

Un modelo, como dijimos en el primer capítulo, sirve —entre otras cosas— para representar algo que no existe y que se quiere probar o analizar para saber si funcionará. Ahora bien, ¿UML es útil para esto?

La pregunta tiene dos aristas. La primera es saber si UML es comprobable sin comprobar el código que modela. Este es un tema que aún no está cerrado, y en el que se ha trabajado y se sigue bregando mucho, sobre todo desde el punto de vista de la formalización del lenguaje.

También dijimos que tiene sentido modelar para probar previamente si es más económico hacer y probar el modelo que hacer y probar el propio artefacto modelado. En efecto, si el modelo es igual de caro de construir o de probar que el producto final, ¿por qué no haríamos directamente el producto, sin modelar? Entonces la pregunta se reduce a: ¿es más barato hacer diagramas que probar el código directamente, escrito a mano? La respuesta es: a veces sí, a veces no, dependiendo del grado de detalle de los diagramas frente a lo que se espera obtener.

Lo que ocurre, en este sentido, es que el software, como creación intelectual que es, requiere de “materiales” parecidos y de “mano de obra” semejante para el modelo que para el producto. Por lo tanto, requiriéndose elementos y habilidades equivalentes para hacer modelos que para construir el producto, puede que, en muchos casos, resulte muy costoso hacer modelos como paso previo a la construcción de software.

Esta es una de las principales razones de fracaso de las herramientas CASE[4] en los años 1990. Sin embargo, hay un nuevo intento de insistir con esta idea, para aprovechar las ventajas de la automatización. Así es como surgió MDD, que puede funcionar si logramos que sea fácil manipular diagramas, y a la vez conseguimos que el lenguaje sea idóneo para la comunicación con perfiles no técnicos.

USOS HETERODOXOS DE UML

Si bien UML tiene una definición precisa sobre lo que deben significar cada uno de los modelos, ya hemos visto algunas situaciones en las cuales hay gente que hace usos diferentes del lenguaje.

No podemos aquí detenernos en todos los usos heterodoxos, pero sí me gustaría citar un par de casos. Uno de ellos es usar el diagrama de clases de UML para representar el esquema de una base de datos relacional, al modo de un diagrama de entidad-relación (DER). La figura 9.10 muestra uno de estos casos.

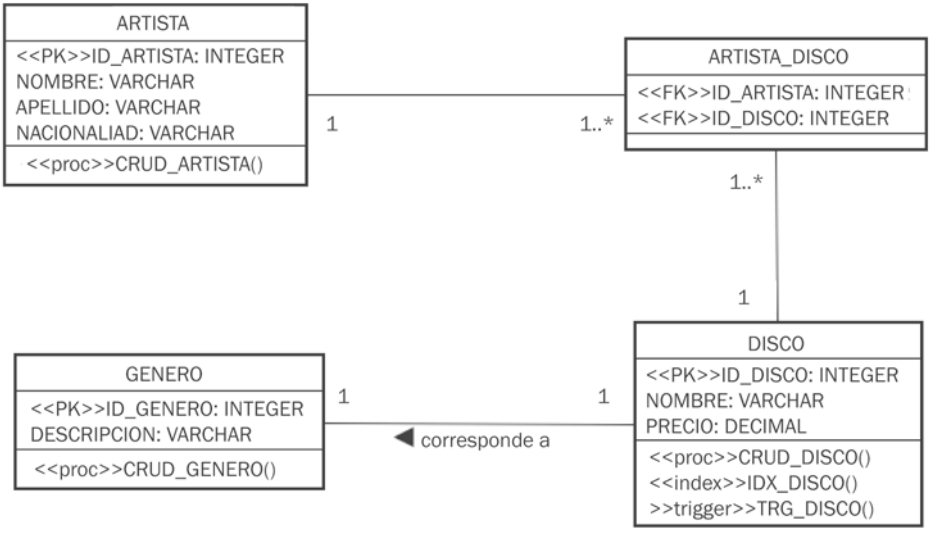

Figura 9.10 Esquema de base de datos con UML.

Ahora bien, el caso anterior no es tan anómalo. Si bien no es del todo respetuoso con los perfiles tradicionales de UML para bases de datos relacionales, es bastante aceptable. Notemos que se han usado los estereotipos `<<PK>>`, `<<FK>>`, `<<proc>>`, `<<trigger>>` e `<<index>>`, para representar claves primarias, claves foráneas, procedimientos almacenados, disparadores e índices, respectivamente.

En cambio, es menos ortodoxo —aunque no incorrecto— el diagrama de la figura 9.11, que pretende modelar la navegabilidad entre pantallas de una aplicación usando un diagrama de actividades.

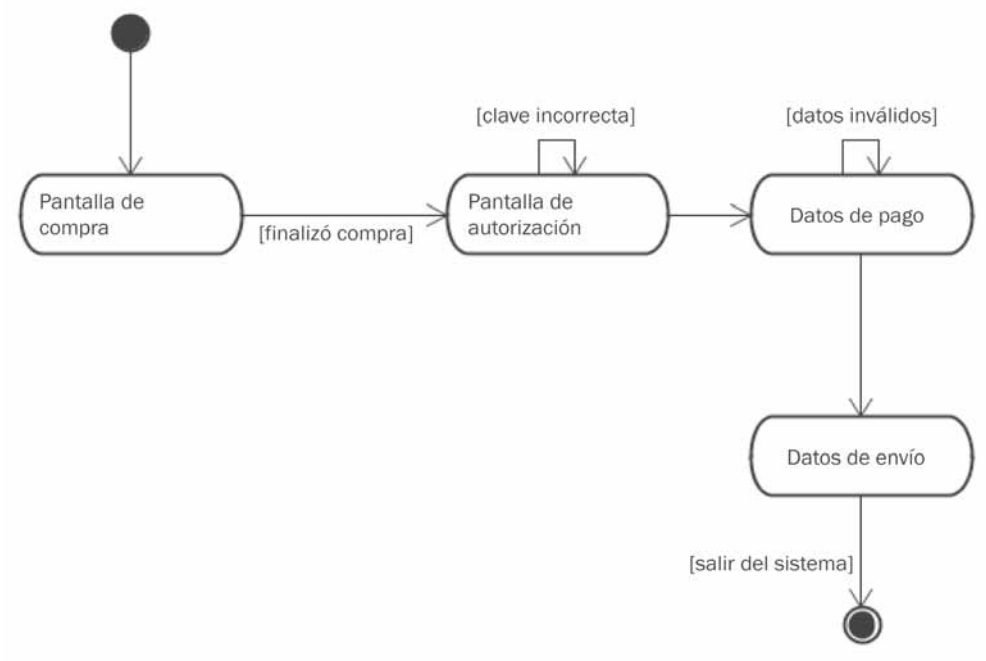

Figura 9.11 Diagrama de navegación con UML.

Lo que ocurre es que hay muchos profesionales que se las ingenian para apartarse de los usos establecidos de UML y aprovechan el lenguaje para modelar situaciones no previstas o de una manera que tal vez resulte más clara en cierto contexto.

Por ejemplo, si estamos usando UML para comunicar un diseño dentro de un equipo de trabajo, ¿por qué no usar la palabra castellana "interfaz" en el estereotipo `<<interface>>`, escrito así, con la palabra en inglés? Y si estuviésemos modelando una clase de una sola instancia, ¿por qué no usar un estereotipo como `<<singleton>>`, que resultaría claro para todos?

1 El término "clases software", en este contexto, incluye las interfaces, para los lenguajes de programación que tienen una construcción de ese nombre.

2 Estamos denominando así a la documentación que queda una vez finalizado un proyecto, para su mantenimiento posterior.

3 Estamos llamando así a las aplicaciones que corren en una sola computadora aislada.

4 Acrónimo inglés para *Computer-Aided Software Engineering* (ingeniería de software asistida por computadora). En los años 1990 se presentaba como la panacea y el futuro del desarrollo de software, pero nunca adquirió demasiado uso.

APÉNDICE

VERSIONES DE UML Y CAMBIOS MÁS IMPORTANTES INTRODUCIDOS

A medida que han ido sucediéndose las distintas versiones de UML, se han producido cambios en el lenguaje. Algunos de ellos fueron agregados, generalmente manteniendo la compatibilidad hacia atrás. En otras ocasiones, los cambios han impactado en la compatibilidad.

Manteniendo la premisa de la simplicidad y practicidad que nos hemos planteado en este libro, vamos a mostrar los principales cambios que introdujo cada versión de UML. Como era de esperar, el salto del número de versión al llegar UML 2.0, implicó el cambio mayor.

VERSIÓN	DIAGRAMA	CAMBIOS	OBSERVACIONES
1.1	Clases.	La composición deja de significar que la relación entre las clases sea inmutable.	En la versión 1.0 la composición implicaba inmutabilidad de la relación.
	Secuencia.	Los retornos de mensajes pasan a representarse con línea de puntos.	En la versión 1.0, los retornos se representaban con línea llena.
	Colaboraciones.	Aparición como concepto.	Comienzan las confusiones con los “diagramas de colaboración”.

VERSIÓN	DIAGRAMA	CAMBIOS	OBSERVACIONES
1.2 y 1.3	Casos de uso.	Se especifica mejor la generalización de casos de uso.	En la versión 1.1 la inclusión era un estereotipo de generalización
	Casos de uso.	Se distinguen las relaciones de inclusión y extensión de las generalizaciones, definiendo que son dependencias y no generalizaciones.	En la versión 1.1 el estereotipo de extensión no existía como tal.
	Casos de uso.	Se desaconseja el estereotipo <<uses>>.	En la versión 1.1 era confusa la distinción entre estereotipos
	Actividades.	Se especifica la semántica.	En la versión 1.1 había una semántica laxa que tendía a confundir bifurcaciones condicionales con barras de sincronización.
	Actividades.	Surgen las bifurcaciones concurrentes anidadas	
1.4	Global.	Aparecen los perfiles de UML para hacer extensiones al lenguaje con propósitos particulares.	En versiones anteriores su uso era caótico.
	Todos.	Se permiten varios estereotipos por elemento.	En versiones anteriores sólo podía haber uno.
	Artefactos.	Aparecen como manifestación física de componentes.	En versiones anteriores, se usaban componentes en diagramas de despliegue.
	Clases.	Aparece el símbolo ~ para la visibilidad de paquete de Java.	
	Interacción (secuencia y "colaboración").	Cambia la forma de representar los mensajes asíncronos.	Ver figuras 5.18 y 5.21.
1.5	Global.	Se define la semántica precisa de UML.	
2.0 y posteriores	Global.	Cambios y definiciones importantes al metamodelo.	No se trata en este libro.
	Objetos.	Aparece como un tipo de diagrama.	Antes eran parte de los diagramas de clases.
	Paquetes.	Aparece como un tipo de diagrama.	Antes eran parte de los diagramas de clases.

VERSIÓN	DIAGRAMA	CAMBIOS	OBSERVACIONES
2.0 y posteriores	Visión global de interacción.	Aparece el diagrama, totalmente nuevo.	
	Tiempos	Aparece el diagrama, totalmente nuevo.	
	Estructura compuesta.	Aparece el diagrama, totalmente nuevo.	
	Interacción (todos).	Aparición de marcos para manejar opciones, condiciones, iteraciones y concurrencia.	Desaparecen las guardas y guardas con asterisco, aunque se siguen usando.
	Interacción (todos).	Los participantes no tienen por qué ser objetos o instancias.	No se deberían subrayar en ese caso.
	Comunicación.	Sin cambios importantes. Sólo cambia de nombre.	Nuevo nombre para el ex ”diagrama de colaboración”.
	Estados y Actividades.	Se independizan entre sí.	Antes ambos eran tipos de diagramas de estados.
	Componentes.	Cambio de notación.	Ver figuras 6.4 y 6.5
	Clases y Objetos.	Cambio de notación para entidades activas.	Ver figuras 7.15 y 7.16.
	Componentes y Clases.	Se separan las nociones y notaciones de interfaces requeridas y proporcionadas.	Antes sólo se usaba la notación chupetín. Ver figuras 6.6 y 6.7.
	Clases.	Cambios en estereotipos.	

Versiones de UML con los cambios más importantes introducidos”

BIBLIOGRAFÍA CITADA

[MMM] Fred Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition (2nd Edition)*, Addison-Wesley Professional, 1995.

[UML Ref] Grady Booch, James Rumbaugh, Ivar Jacobson, *The Unified Modeling Language User Guide, (2nd Edition)*, Addison-Wesley Professional, 2005.