

1

MODELOS Y UML

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

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

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

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 UML².

Lo que sí faremos 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 pUML³ 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 aras 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 quizás 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.

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

- 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

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

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.

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

- 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):

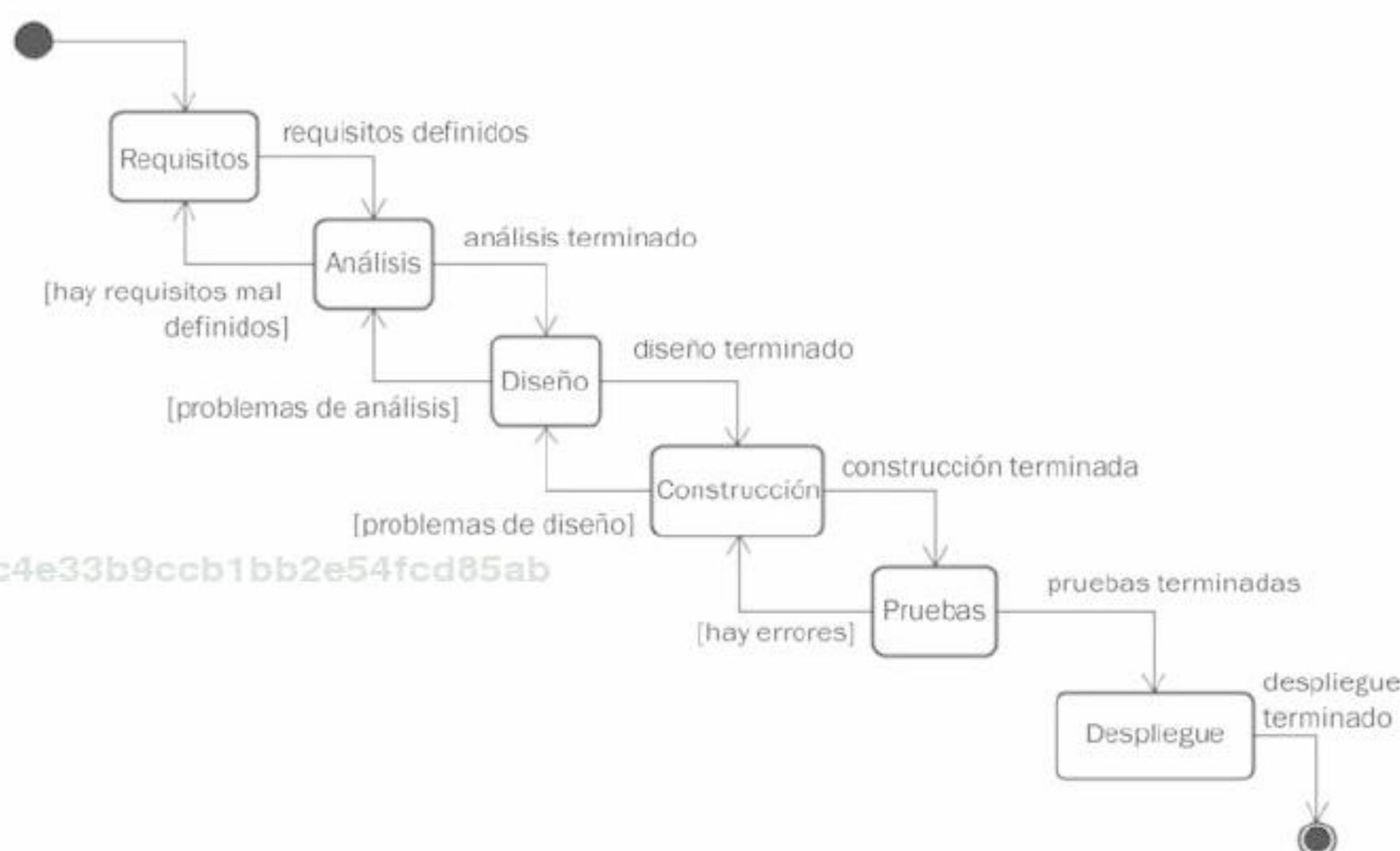


Figura 2.1 Ciclo en cascada.

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.

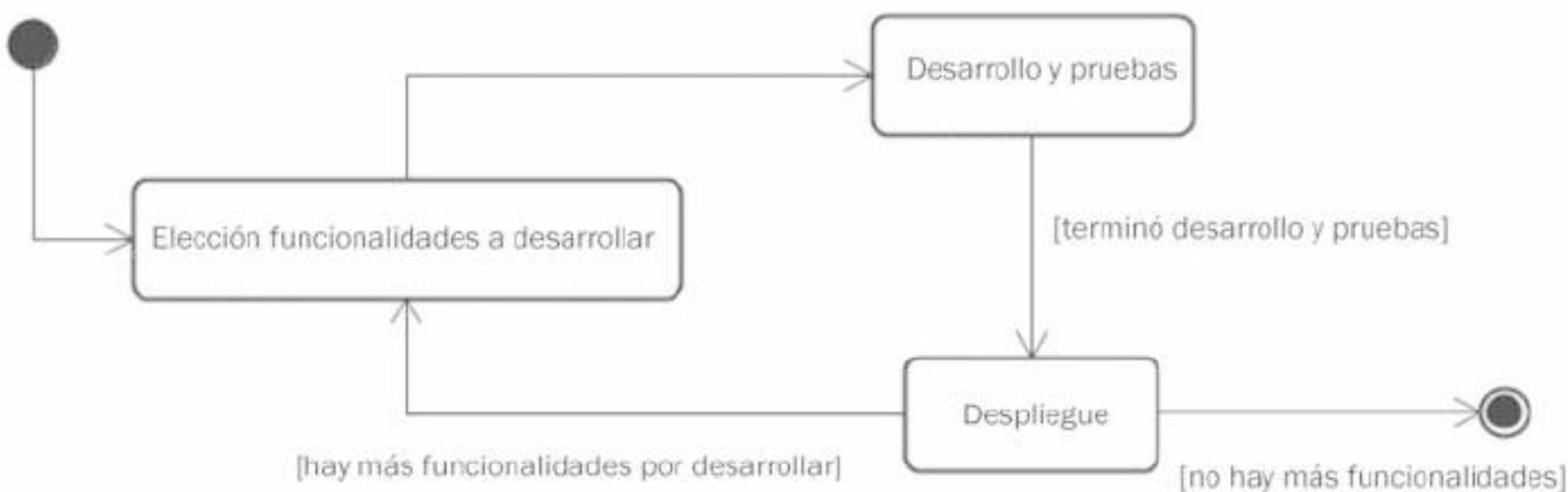
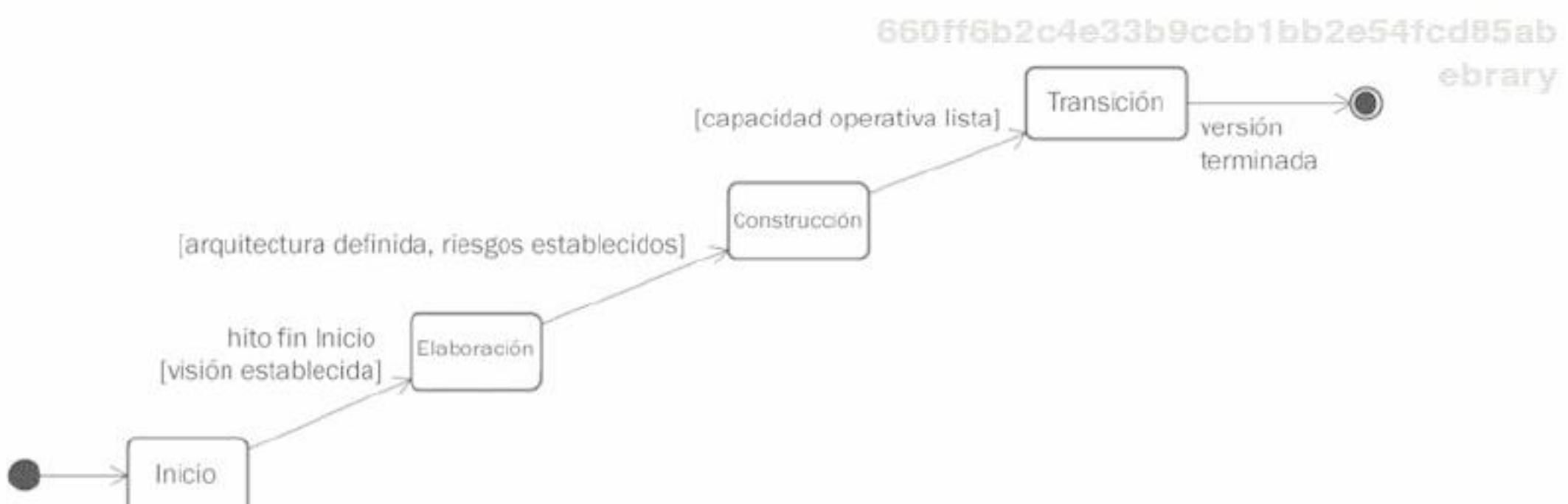
Todas estas críticas no desconocen que hay un marco general del proyecto que hay que definir en una etapa temprana, que incluyen una visión del proyecto, algunos requisitos y un diseño macro, aun cuando admitamos refinamientos posteriores.

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, reconoczamos 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.

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

-
- 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

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

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 cargarán 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 imprimirlas.
 - 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.
- 660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary
- 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 saltarse 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 deseé 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* planteando 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.

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

4

MODELADO DE REQUISITOS DEL CLIENTE

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

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 elicitation, 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	<p>1. El Administrador debe estar logueado en el sistema.</p> <p>2. Queda una nueva empresa activa en el sistema.</p> <p>3. 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.</p> <p>4. Se deja un rastro de auditoría en el log del sistema.</p>
Postcondiciones	<p>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).</p>
Flujo Normal	

NOMBRE	AGREGAR EMPRESA
Flujo Normal	<p>5. El sistema guarda los datos en la base de datos.</p> <p>6. El sistema genera un usuario y una clave para el administrador del cliente.</p> <p>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.</p> <p>8. El sistema guarda en el log el nombre de usuario (Administrador) y la acción realizada ("Alta de nueva empresa: <nombre de empresa>").</p> <p>9. Finaliza el caso de uso.</p>
Flujos alternativos	<p>S1. El usuario abandona la carga sin terminar antes del paso 4 del flujo normal.</p> <p>S1.1. El sistema pregunta al usuario si desea abandonar.</p> <p>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")</p> <p>S1.3. Finaliza el caso de uso.</p>
Excepciones	<p>E1. No se cargaron todos los datos requeridos.</p> <p>E1.1 El sistema indica que existen datos requeridos no cargados.</p> <p>E1.2 Vuelve al flujo principal, paso 3.</p> <p>E2. Ya hay cargada una empresa con el mismo nombre.</p> <p>E2.1 El sistema informa que ya hay una empresa con el mismo nombre.</p> <p>E2.2 Vuelve al flujo principal, paso 3 (el usuario podrá abandonar o cambiar el nombre de la empresa).</p> <p>E3. El mail o el teléfono ingresados no son válidos.</p> <p>E3.1 El sistema indica que los datos ingresados no son válidos.</p> <p>E3.2 Vuelve al flujo principal, paso 3.</p>
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 hablamos 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

ebrary

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 BDD¹.

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.

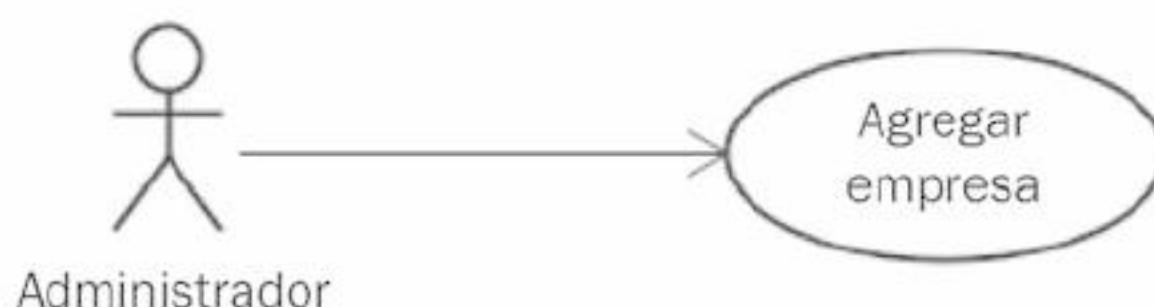


Figura 4.2 Navegabilidad del actor al caso de uso.

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*.

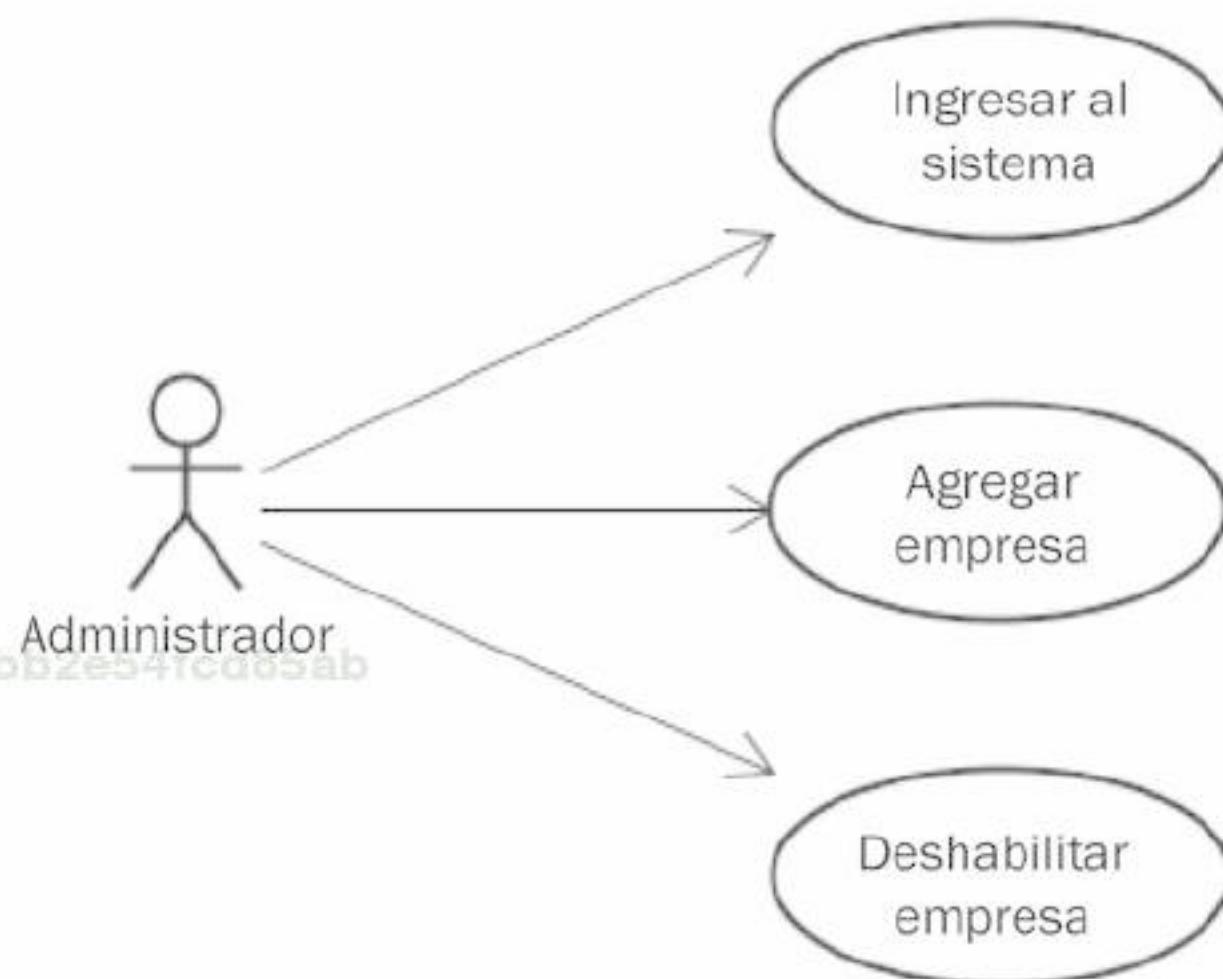


Figura 4.4 Diagrama de casos de uso de un subsistema.

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 parte 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.

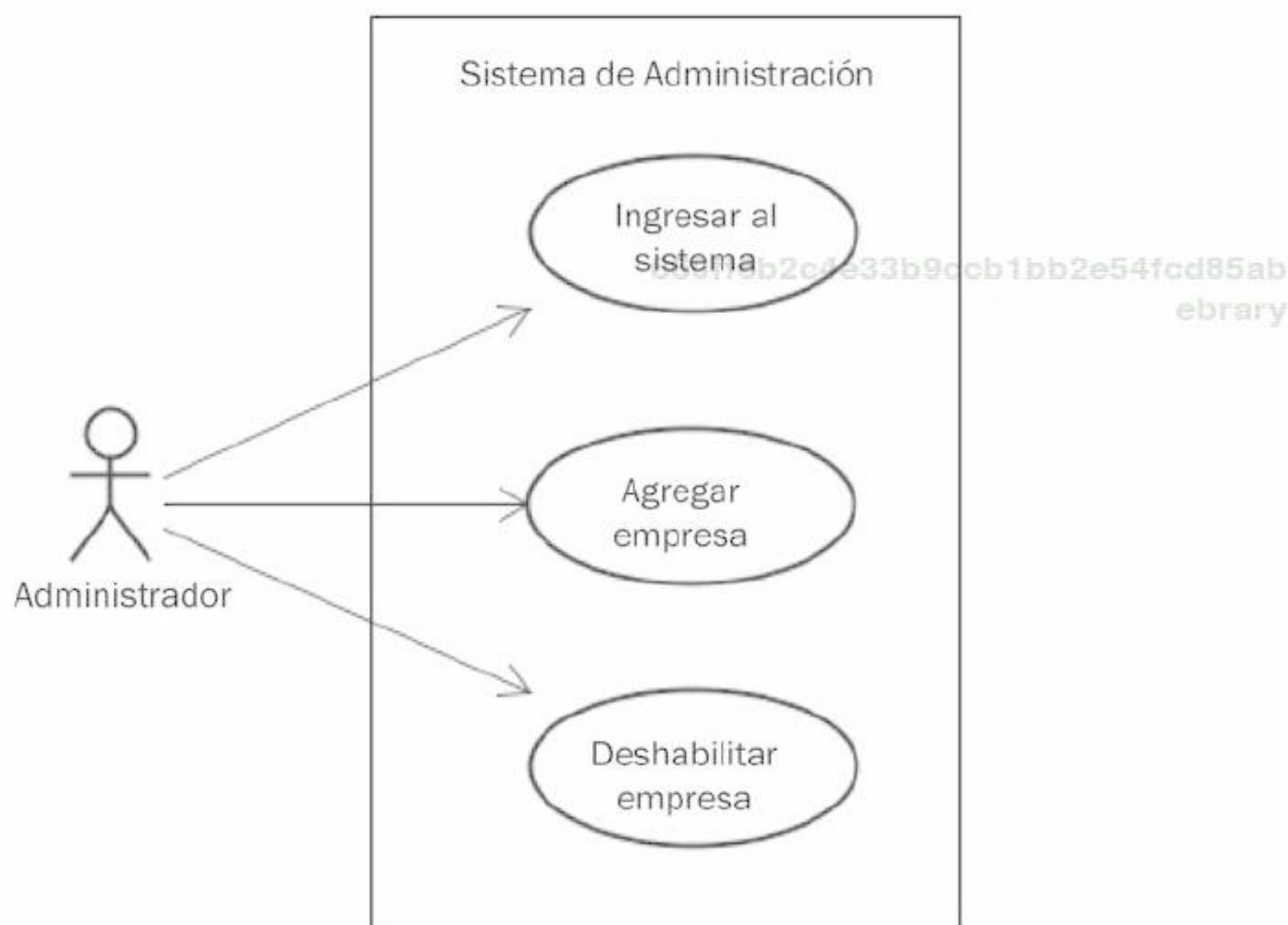


Figura 4.5 Contexto de un subsistema.

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.

660ff6b2c4e33b9ccb1bb2e54fcd85ab
ebrary

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 “Aregar 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*).

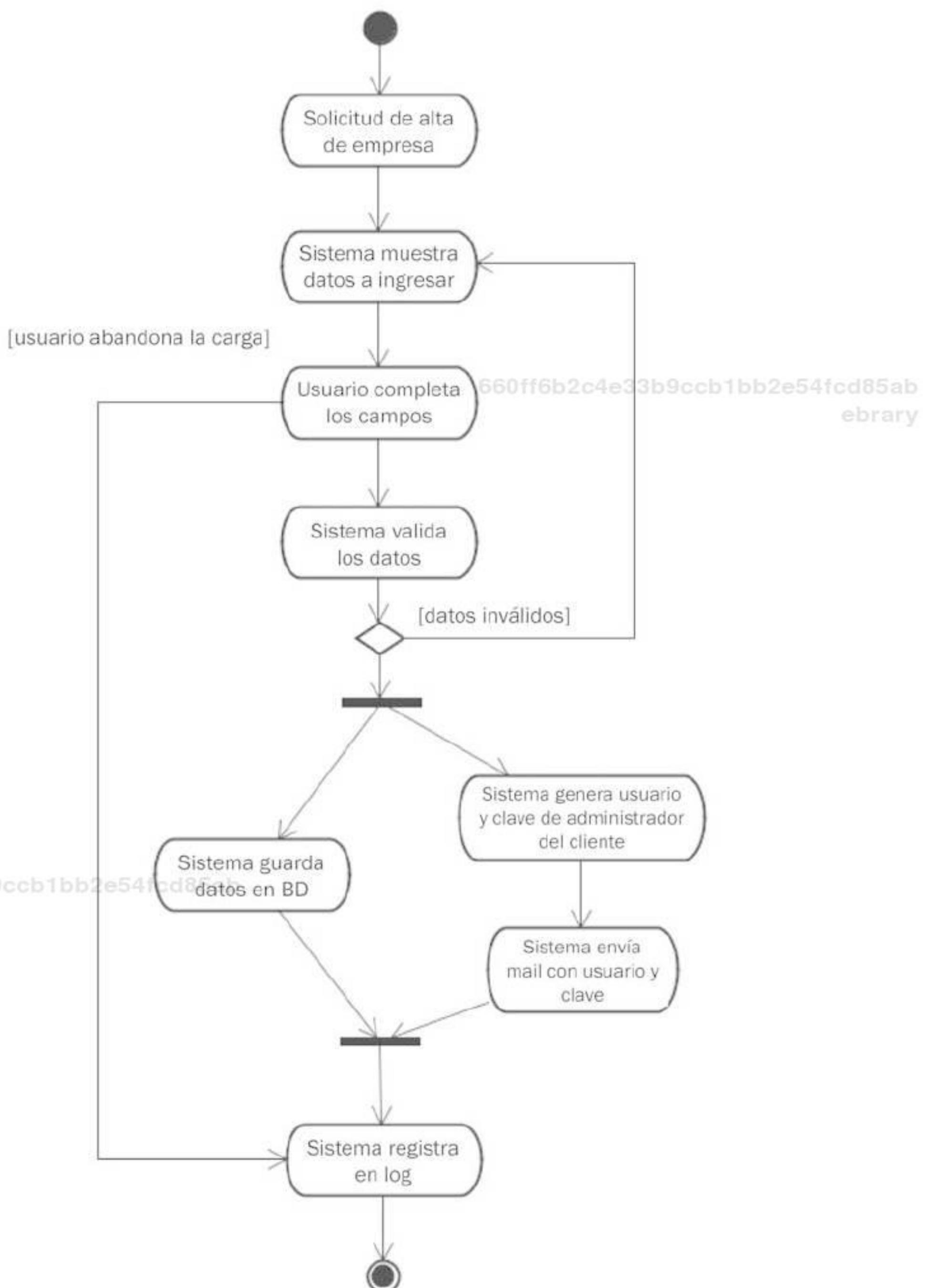


Figura 4.6 Diagrama de actividades de un caso de uso.

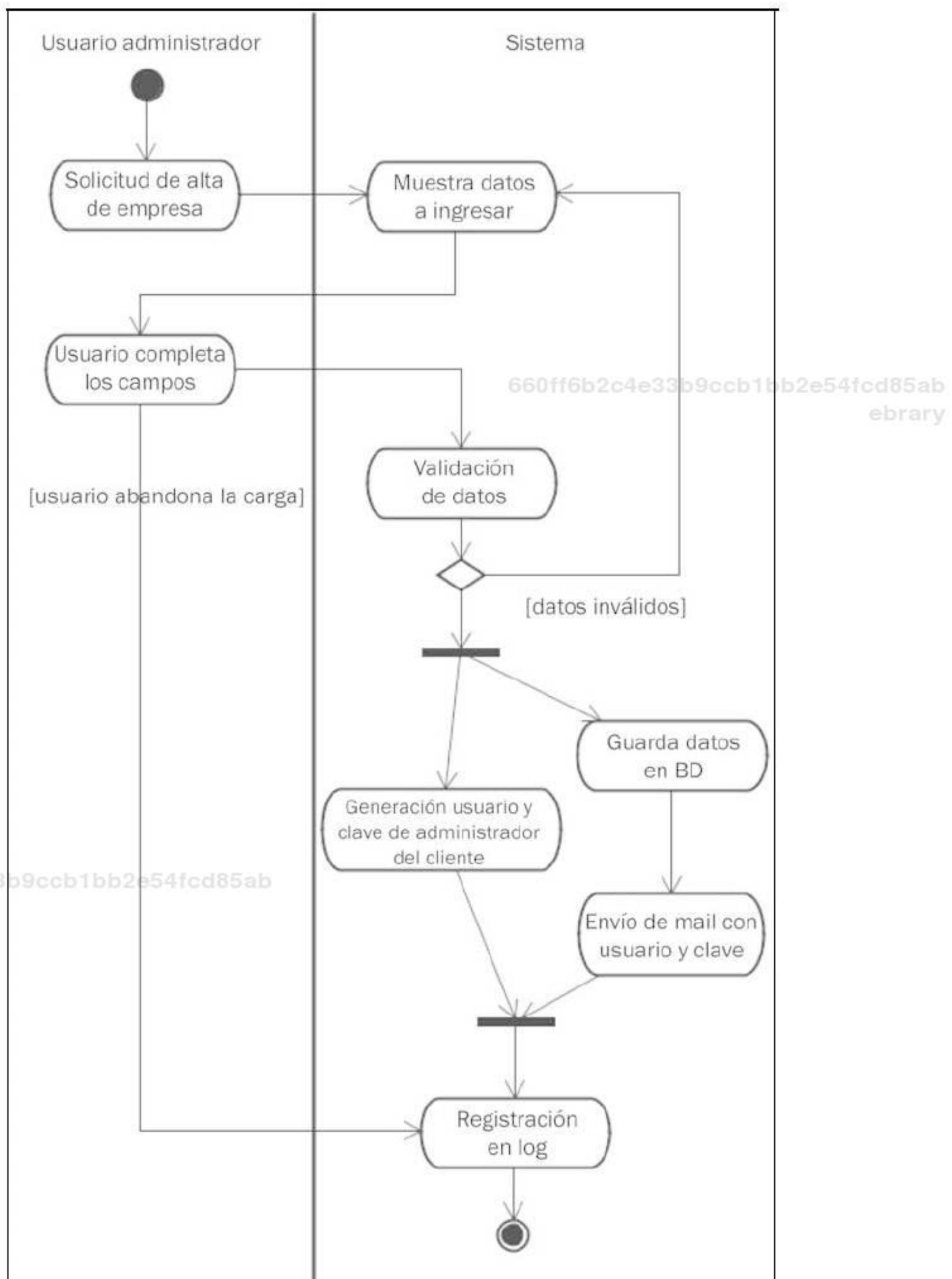


Figura 4.7 Diagrama de actividades con calles.

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.

660ff6b2c4e33b9ccb1bb2e54fcda85ab
ebrary

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.

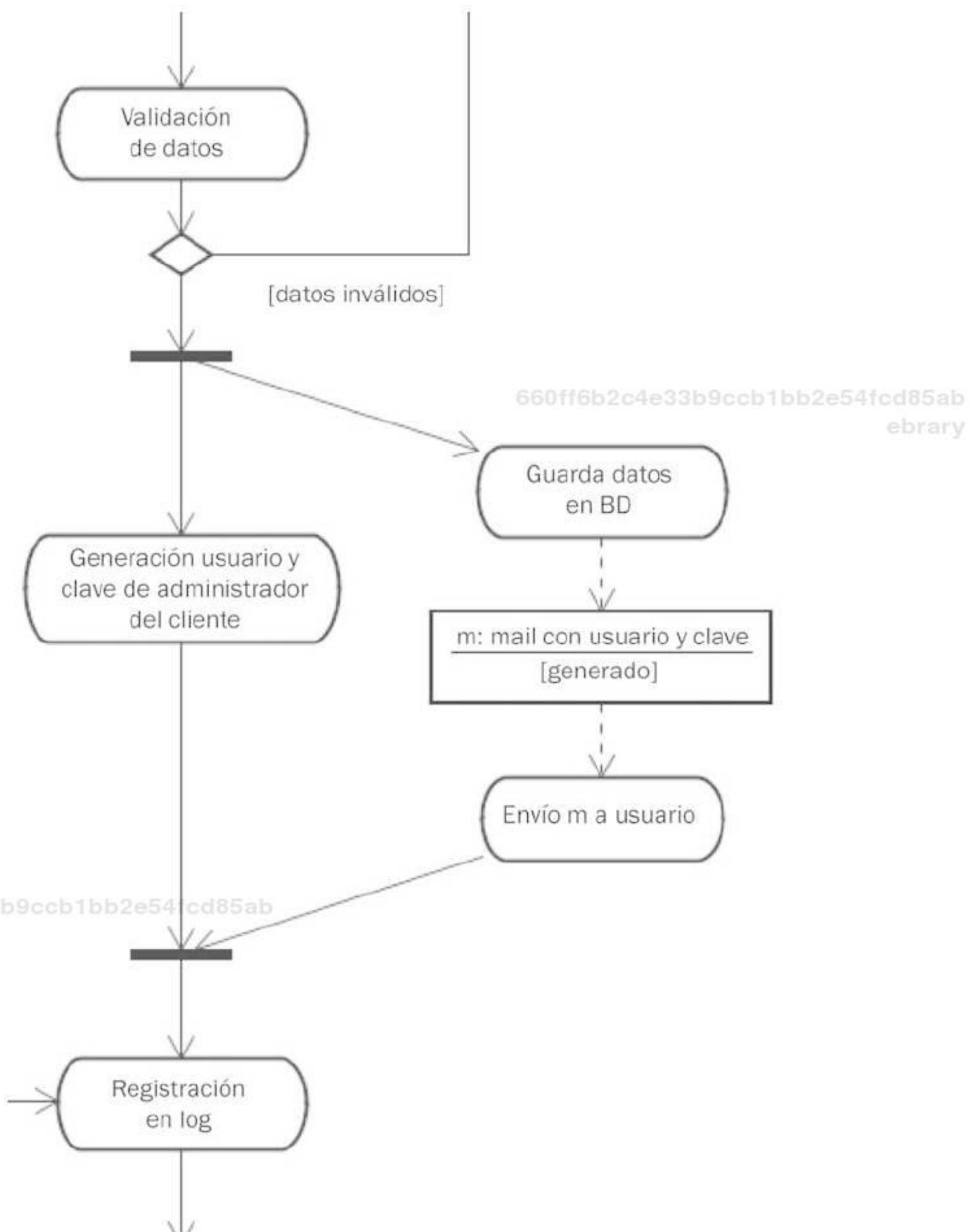


Figura 4.8 Objeto en diagrama de actividades.

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.

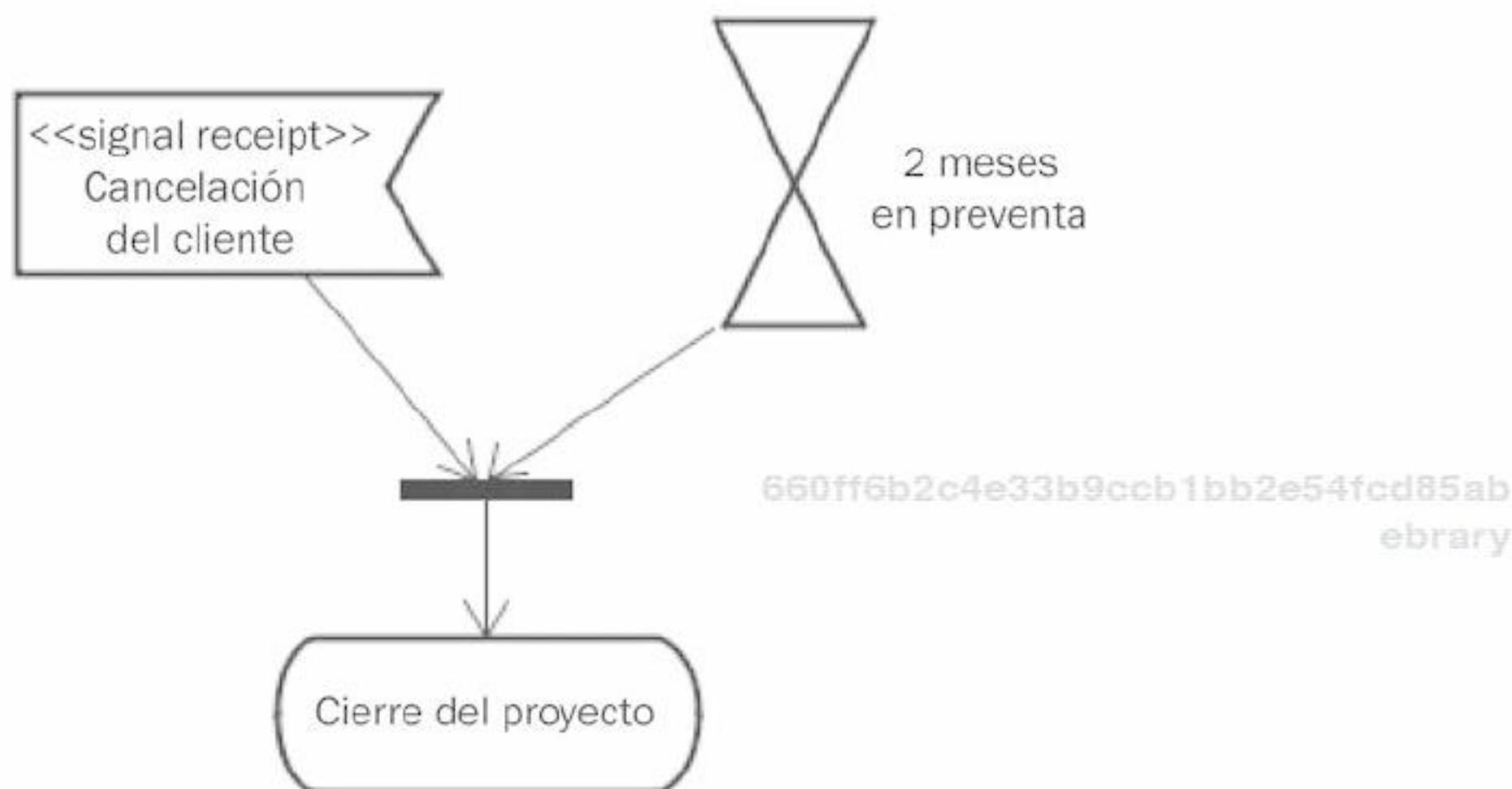


Figura 4.9 Señal temporal y evento proveniente del exterior.

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.

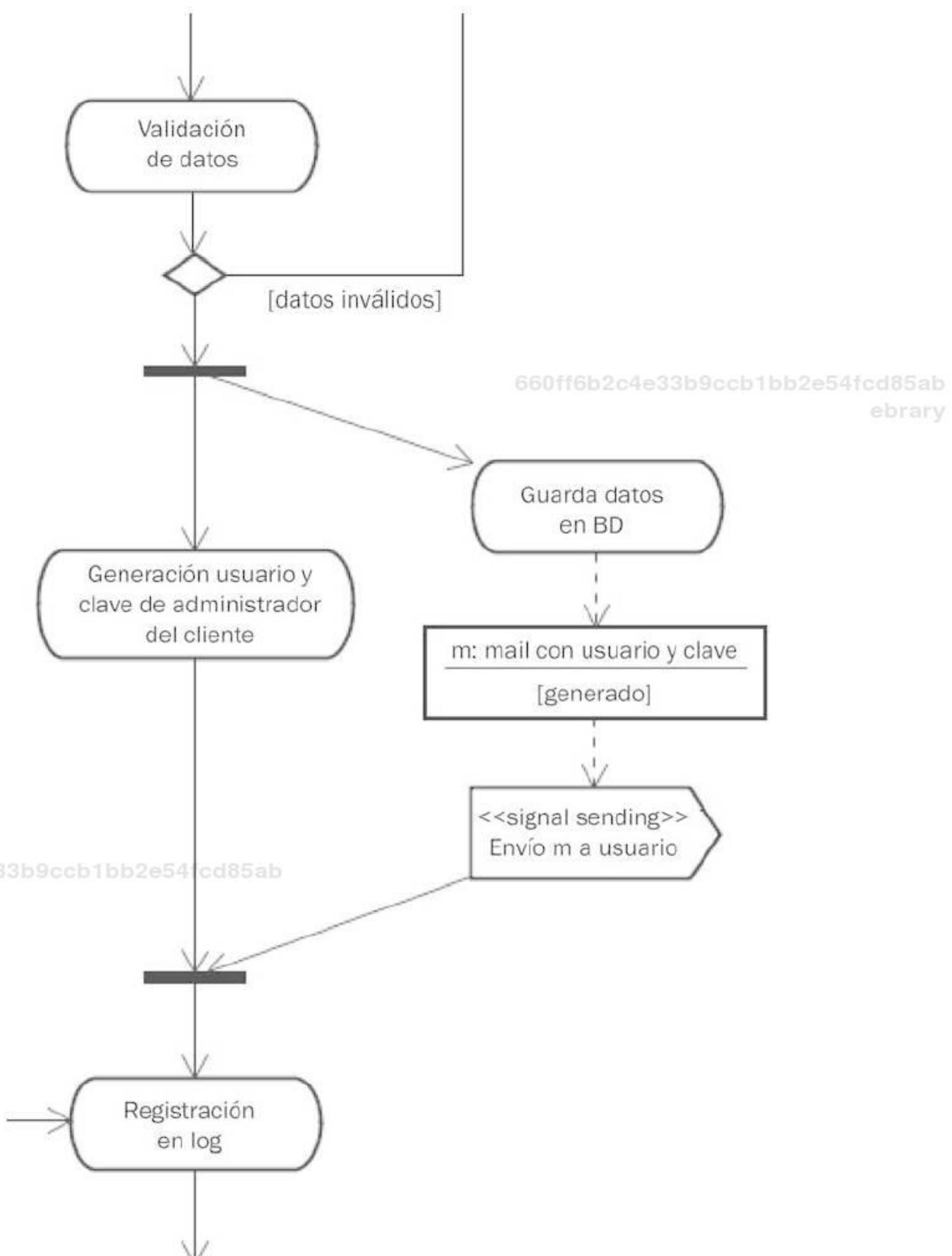


Figura 4.10 Evento emitido.

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>>.

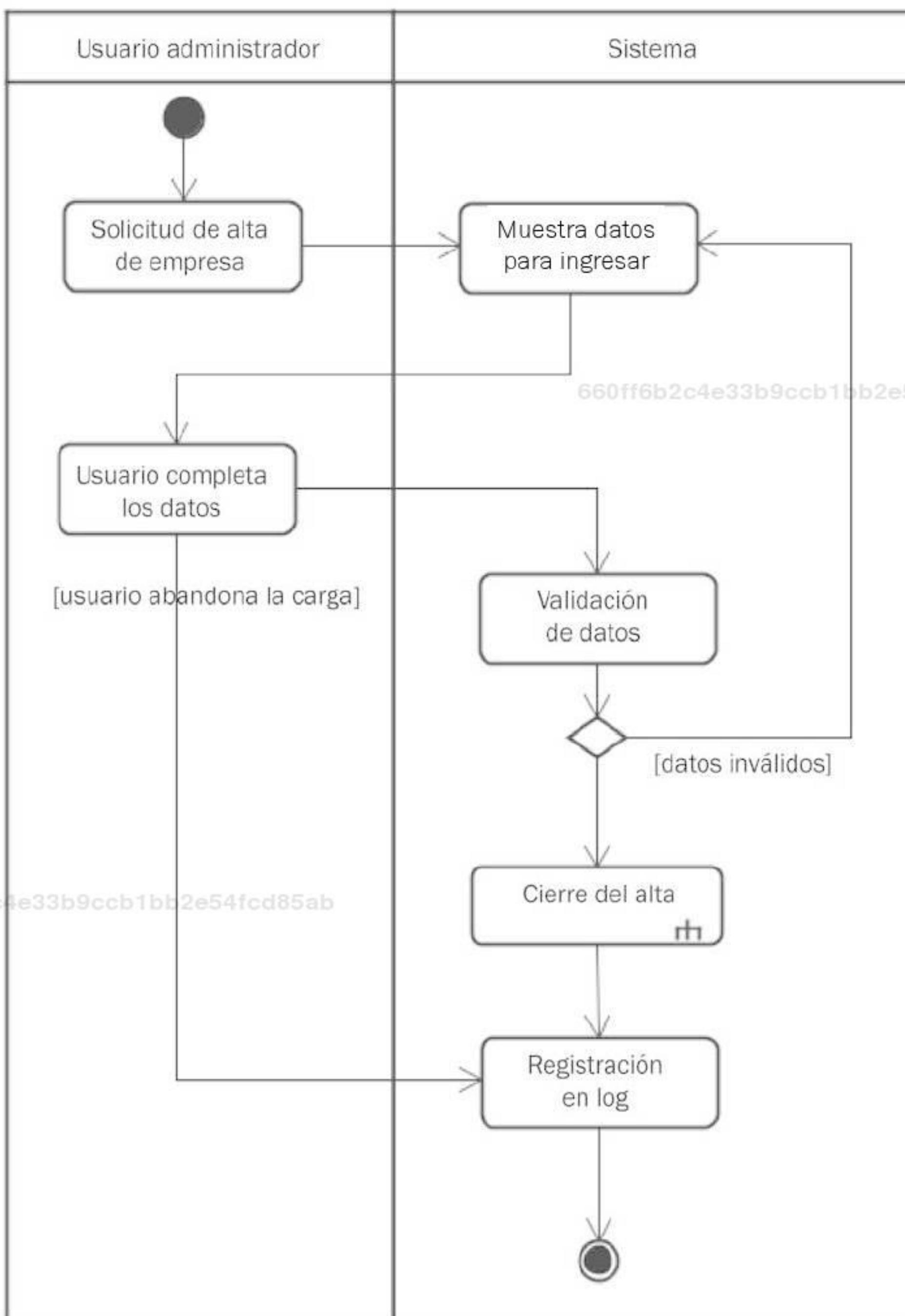


Figura 4.11 Actividad compuesta de cierre del alta.

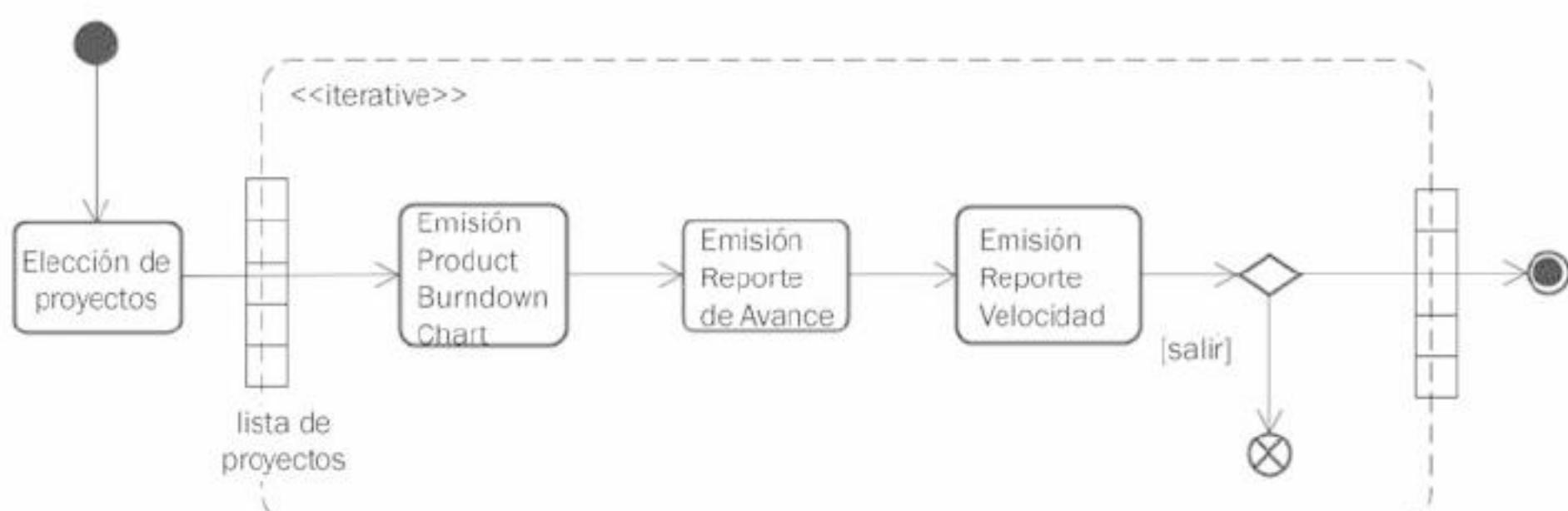
**Figura 4.12** Sub-actividades.**Figura 4.13** Región de expansión.**Figura 4.14** Actividades y conectores.

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.

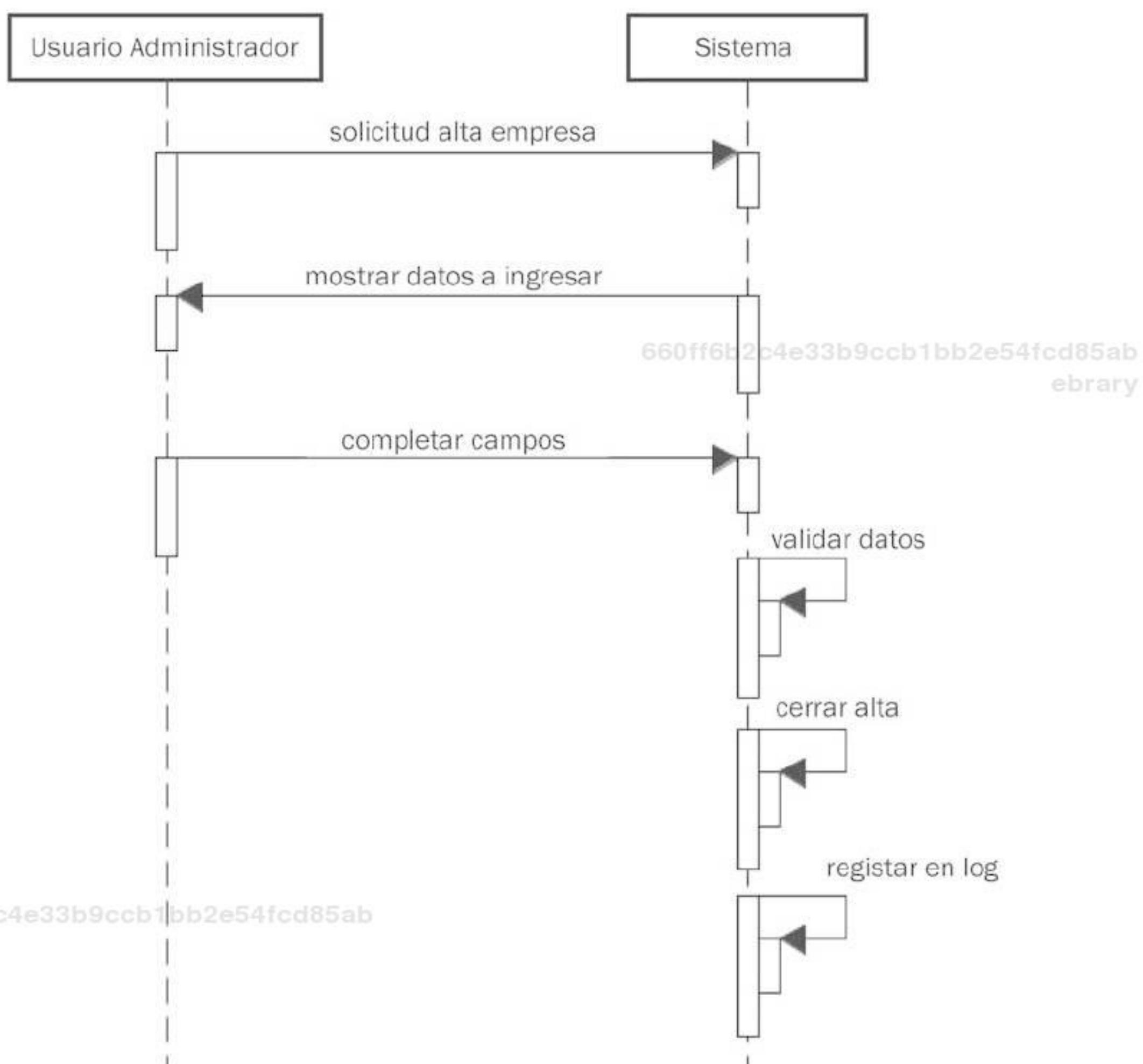


Figura 4.15 Diagrama de secuencia de sistema.

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, quizás 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 descalificadorio 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.

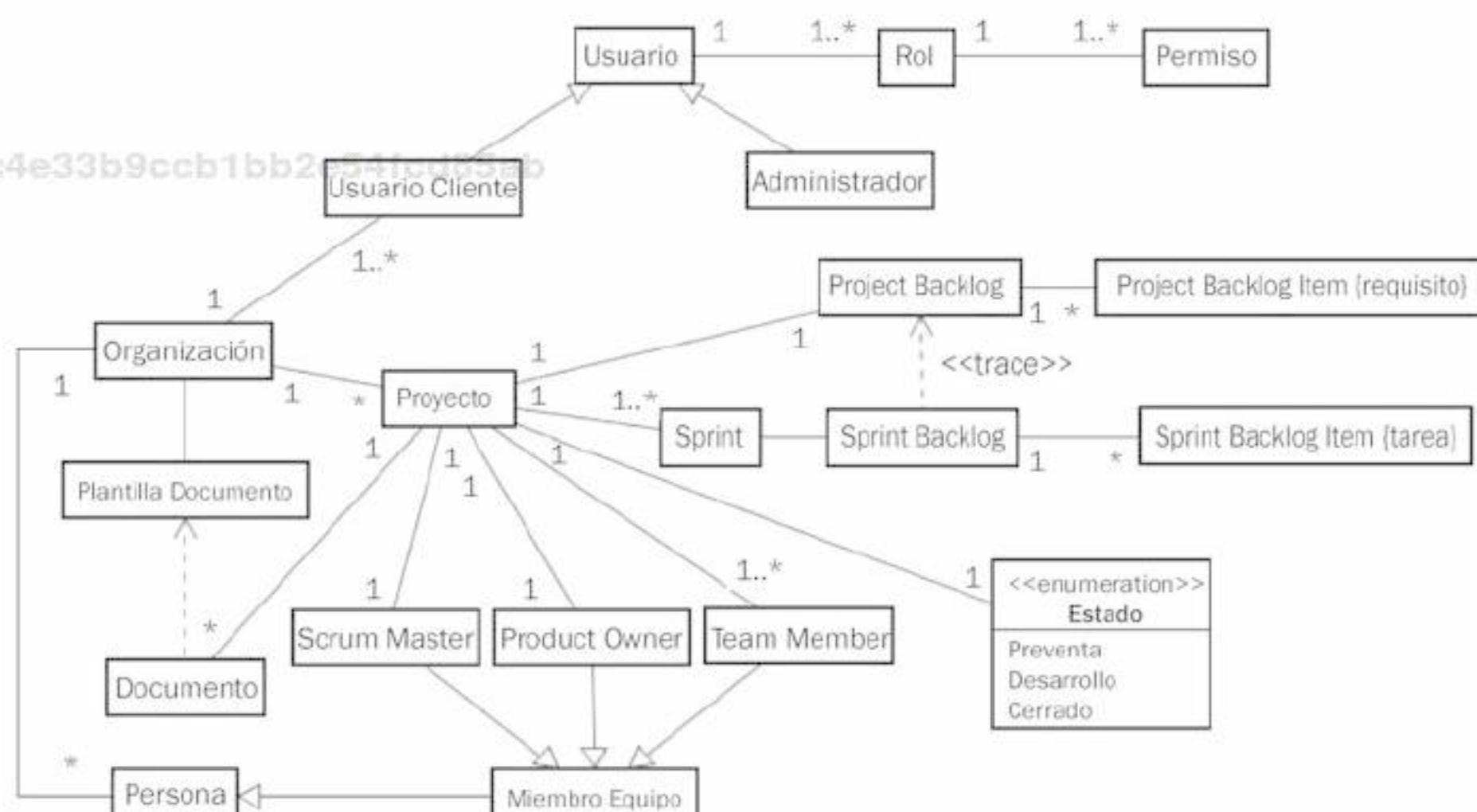


Figura 4.16 Modelo de dominio de *FollowScrum*

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-parte. 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.

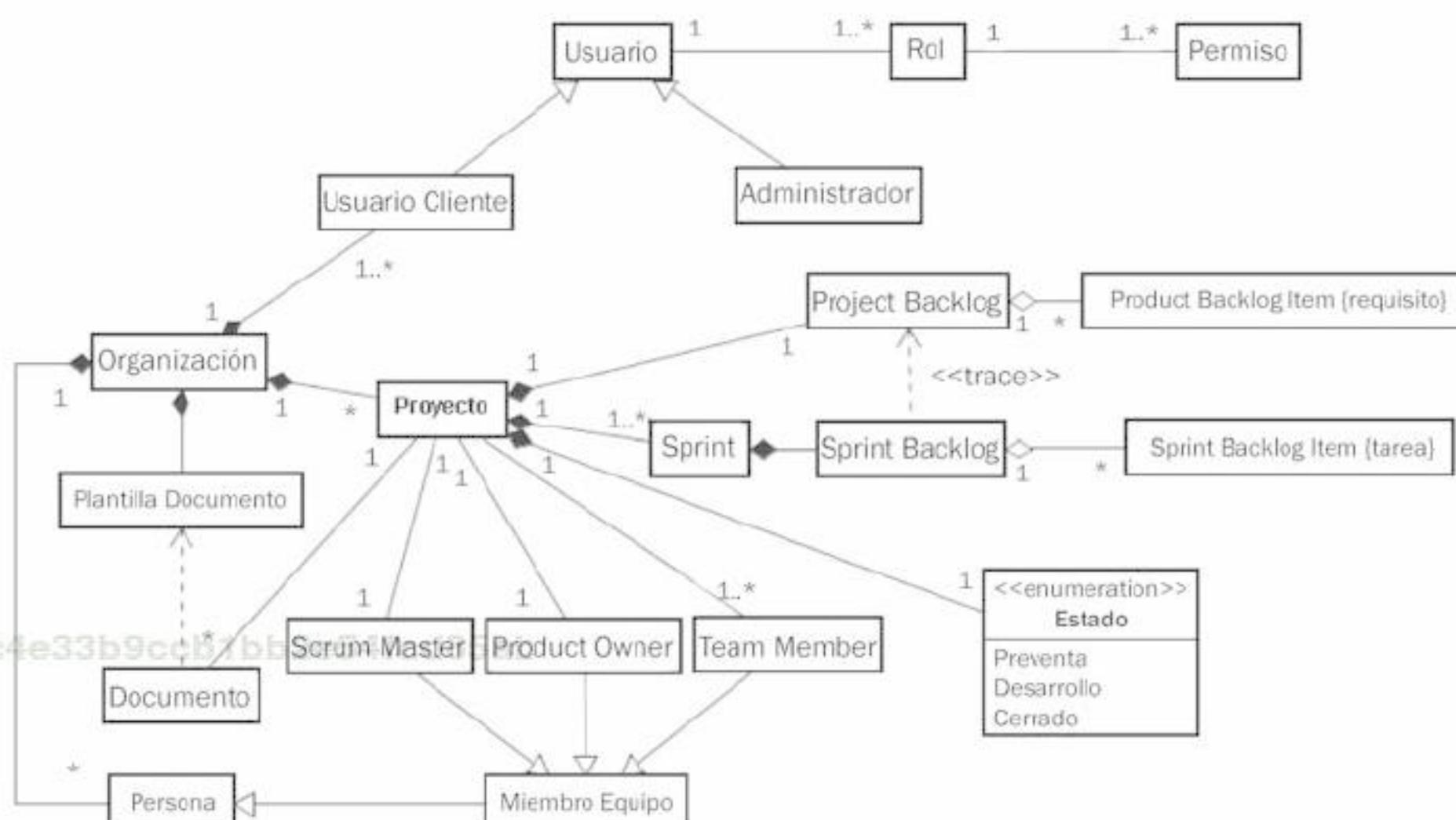


Figura 4.17 Agregación y composición.

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, quizás 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.

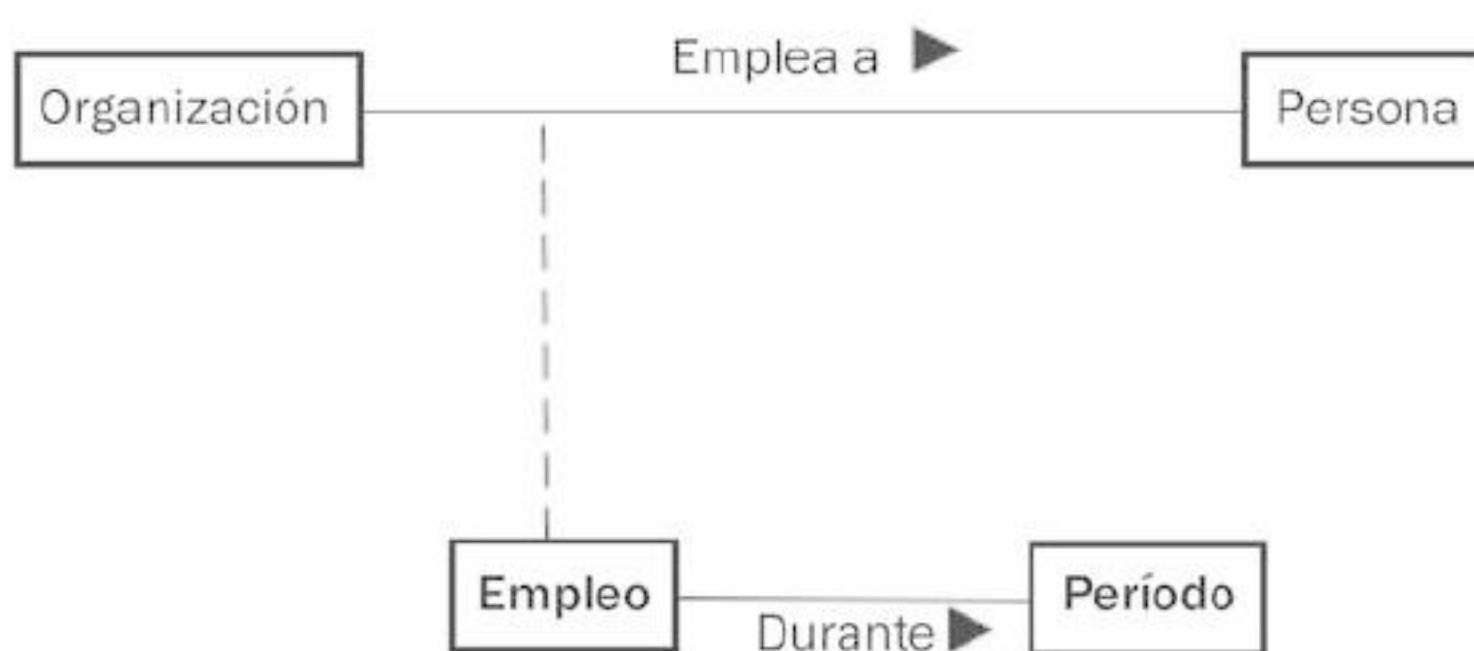


Figura 4.21 Clase de asociación.

660ff6b2c4e33b9ccb1bb2e54fcdb85ab Más sobre generalizaciones y especializaciones ebrary

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.

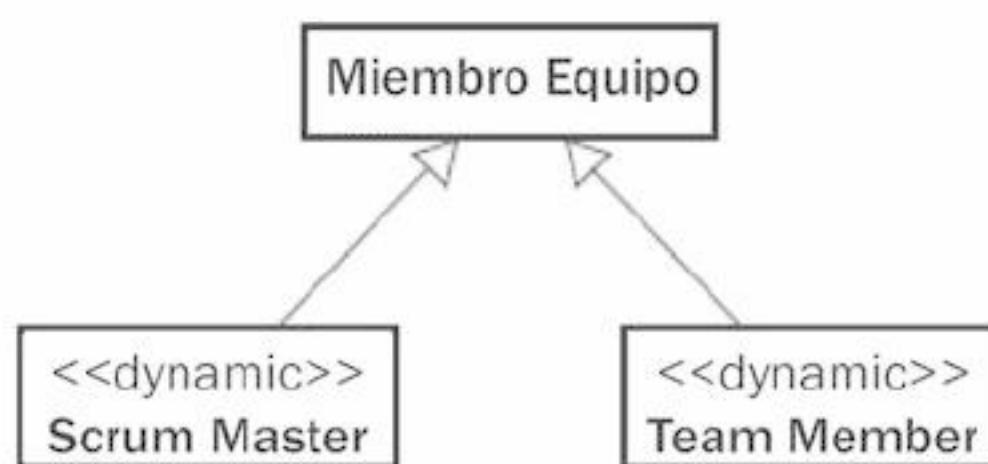


Figura 4.22 Especializaciones transitorias.