CHAPTER 8

UML El Lenguaje de Modelación Unificado

8.1 Diseño en el contexto de procesos ágiles

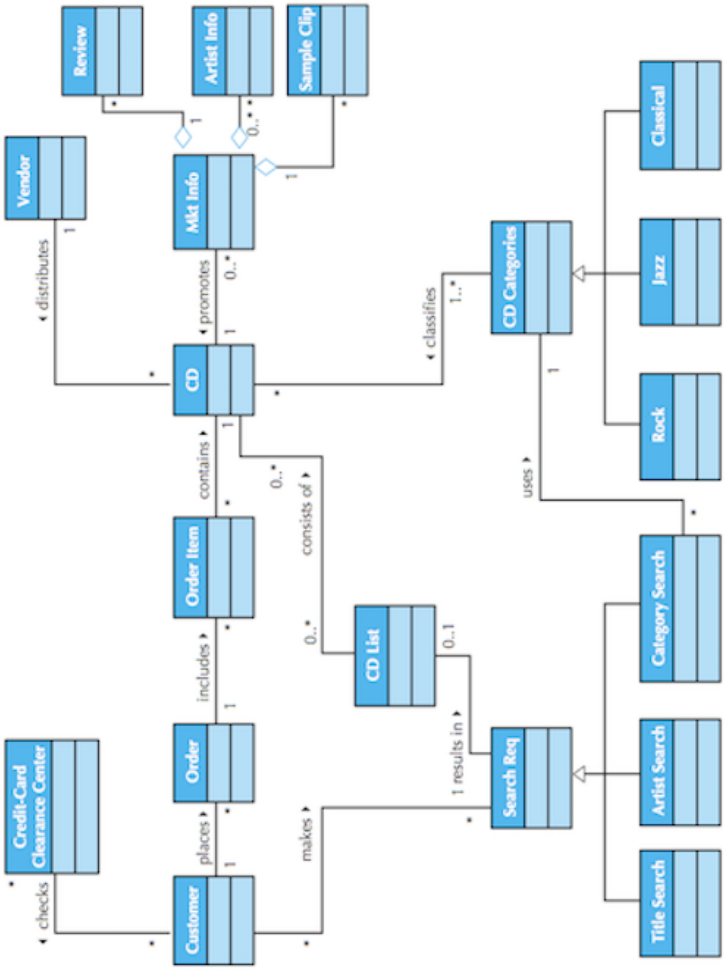
Tradicionalmente la etapa de diseño sigue a una etapa de análisis que a su vez cierra el levantamiento de requisitos. La diferencia esencial que se suele hacer entre análisis y diseño es que en el análisis nos enfocamos en "qué" mientras que en diseño el foco está en el "cómo".

En el desarrollo ágil la separación entre análisis y diseño es muy sutil, casi imperceptible. Tanto durante el análisis como durante el diseño se busca construir un modelo: el modelo de análisis o el modelo de diseño. El modelo de análisis se parece mucho a un modelo de datos, mientras que el modelo de diseño incluye aspectos mucho más cercanos a la implementación.

8.2 Modelos de Dominio

El modelo de análisis, también llamado modelo de dominio, requiere identificar las clases de dominio y sus principales atributos y en este sentido se parece a un modelo de datos (modelo entidad relación), ya que las clases de dominio incluyen las entidades persistentes que deberán ser almacenadas (aunque puede incluir algunas otras adicionales también).

La figura muestra un ejemplo de modelo de dominio para una tienda que vende música en forma de CDs (¡todavía existen!). Puede observarse que se ha utilizado un diagrama de clases UML, pero no se han indicado ni atributos ni métodos de las clases. Lo esencial aquí es la identificación de las entidades y de las asociaciones entre ellas que son marcadas con líneas.



8.3 Cómo encontrar las clases (objetos) de dominio

Los candidatos deben buscarse en el trabajo realizado de levantamiento de requisitos funcionales, ya sea en los relatos de usuario, en los casos de uso o documentos clásicos de requisitos. Adicionalmente, podemos pensar en candidatos en el espacio general del problema (aquí sirve mucho la experiencia del analista) , roles, eventos, ubicaciones geográficas o unidades organizacionales.

Pon atención en que estamos hablando de candidatos, ya que luego de esta etapa muchos de ellos finalmente no serán considerados como clases de dominio por distintas razones. Lo más común es que se trate mas bien de un atributo de una clase o que es una clase que ya fue considerada con otro nombre, etc.

Algunos autores sugieren revisar cuidadosamente el texto que describe los requisitos y asociar estructuras gramaticales como elementos del dominio: un sustantivo común sería una clase, un adjetivo un atributo, un verbo un

método, un verbo tener como agregación, etc. En mi opinión esto no es una muy buena idea, especialmente si las descripciones están en Castellano porque la flexibilidad del lenguaje admite muchísimas formas de decir lo mismo.

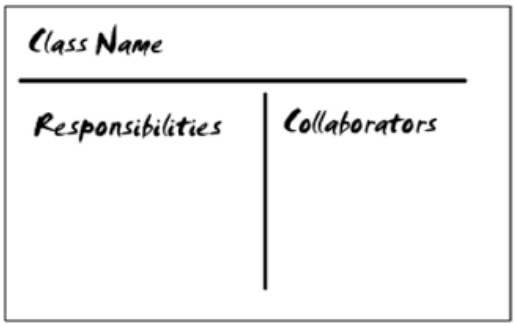
Algunos criterios para evaluar los candidatos encontrados son:

- se cumple que los objetos de esa clase guardan alguna información
- se puede pensar en más de un atributo
- se puede pensar en métodos
- todas las instancias de esa clase compartirán los mismos atributos y métodos

8.4 Clase, responsabilidad, colaborador (CRC)

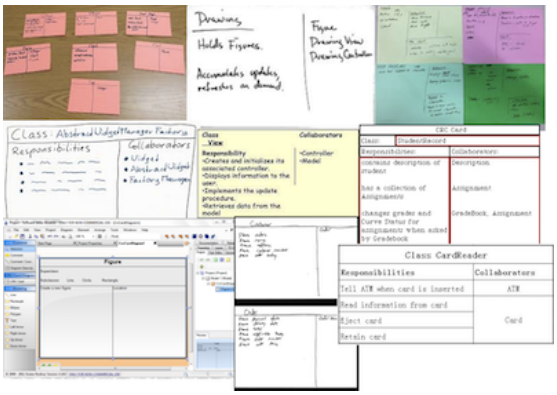
Las responsabilidades de un objeto tienen que ver con qué es lo que el objeto sabe (atributos) y qué es lo que el objeto es capaz de hacer (métodos). Los colaboradores son objetos de otras clases que son necesarios para que nuestro objeto pueda realizar lo que es capaz de hacer.

Una metodología muy popular que permite construir y mejorar el modelo de dominio es el de las tarjetas CRC (por *class - responsibility - collaborators*). Se construye una tarjeta por cada clase en que se pone el nombre de la clase, sus responsabilidades y los colaboradores (ver figura).



La idea es distribuir una tarjeta a cada persona e intentar llevar a cabo alguna funcionalidad. En el proceso puede encontrarse que faltaban o sobraban responsabilidades o colaboradores.

Es común el uso de tarjetas de 3x5 pulgadas para este fin, pero puede usarse cualquier otro medio. La siguiente figura muestra algunas tarjetas reales:



8.5 Del análisis al diseño

Ya hemos dicho que en la etapa de diseño pasamos del qué al cómo. Es particularmente importante tener criterios para decidir si un diseño es superior a otro (ambos responden el mismo qué pero con cómos diferentes)

La complejidad del problema necesariamente hará necesario descomponer la solución completa en elementos menores. Por ejemplo, 5 subsistemas cada uno con 4 componentes, cada una de las cuales tiene su propio diagrama de clases. La forma de descomponer el problema da origen a muchas posibles soluciones.

Los atributos que caracterizan a un buen diseño pueden ser:

- fácil de entender
- fácil de modificar y extender
- fácil de reutilizar en otro problema
- fácil de testear la implementación
- fácil integrar las distintas unidades
- fácil de implementar (programar)

8.6 Acoplamiento y Cohesión

Hay dos atributos que permiten sintetizar buena parte de lo que caracteriza un buen diseño: el acoplamiento y la cohesión. El acoplamiento representa el grado de interacción entre las distintas unidades y un bajo nivel de acoplamiento va a incidir en que el sistema sea más fácil de entender, modificar, extender y testear. La cohesión tiene que ver con que tanto tienen que ver entre sí las cosas que decidimos dejar juntas en una misma unidad. Una clase enfocada en una sola cosa es más fácil de entender, mantener y extender.

Lo que buscamos entonces es un diseño que tenga una alta cohesión y un bajo acoplamiento. Cabe hacer notar que estas dos cosas deben ser vistas

simultáneamente, ya que es fácil pensar en diseños de bajísimo o nulo acoplamiento pero con muy mala cohesión (por ejemplo todo en una sola unidad) o al revés, diseños de altísima cohesión pero enorme acoplamiento (clases extremadamente pequeñas).

Piensa en un tren. Los carros solo están conectados en un punto entre ellos (bajo acoplamiento) y cada carro lleva un contenido que es de un mismo tipo o para un mismo cliente (cohesión). El hecho de tener carros pequeños hace más fácil extender el tren, cambiar un carro por otro, etc.

Carros acoplados mediante interfaz simple y pequeña
Separación en carros permite separar contenidos cohesionados

Es imposible no tener acoplamiento, pero hay mejores y peores categorías de acoplamiento. Ordenados de peor a mejor los tipos de acoplamiento:

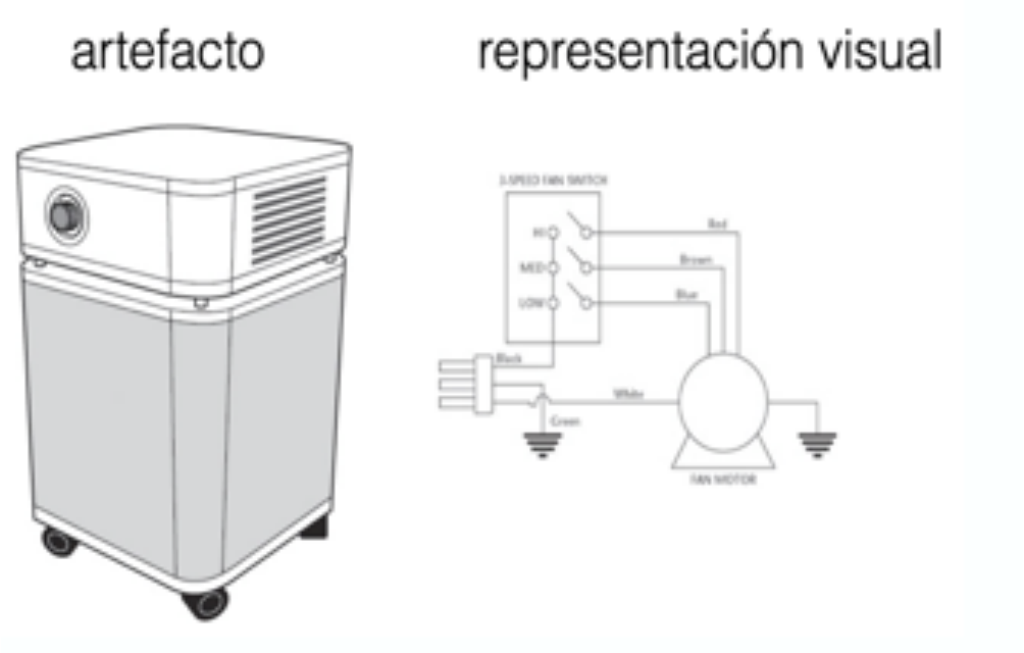
- acoplamiento por contenido (acceso a datos internos)
- acoplamiento por variables globales
- acoplamiento de control (una unidad influye en el flujo de control de otra)
- acoplamiento de datos (paso de parámetros necesarios)

En cuanto a cohesión, las categorías desde menor (más malo) a mayor (deseable) serían:

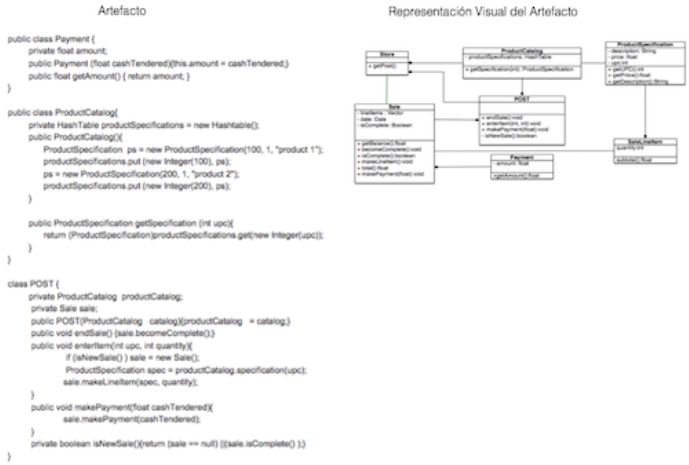
- coincidencial
- lógica
- temporal
- procedural
- comunicacional
- secuencial
- funcional

8.7 Un lenguaje visual para el software

La actividad de diseño se favorece enormemente al trabajar con un lenguaje gráfico de alto nivel. En otras áreas de la ingeniería se habla de artefactos y representación de ese artefacto. Por ejemplo, una máquina y su diagrama de conexión eléctrico (ver figura)



Si el código mismo representa a un artefacto de *software*, entonces lo queremos es una representación visual del artefacto.



Al igual que como en el caso de describir una casa no basta un solo tipo de diagrama (planta, elevación, instalaciones, etc), en el caso del *software* también se requieren diversos tipos de diagramas. Para ello UML proporciona más de diez tipos distintos de diagramas pero nosotros solo veremos los mas utilizados: diagrama de clases, diagrama de secuencia, diagrama de estados, etc.

8.8 El diagrama de clases

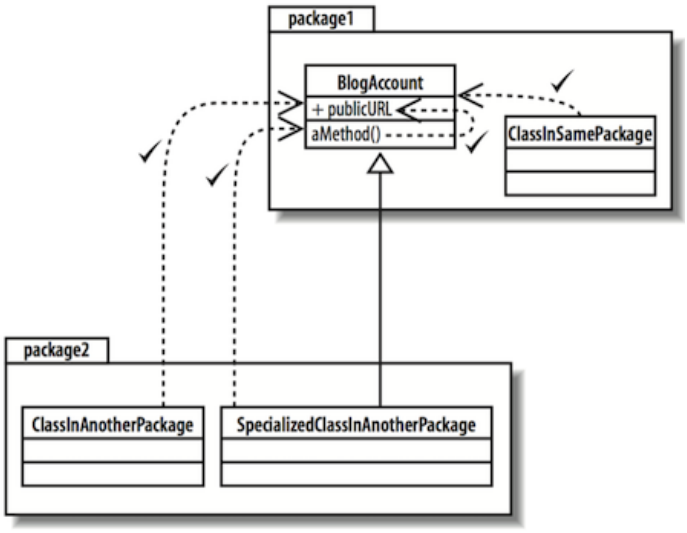
Hay rectángulos que denotan clases y líneas que representan asociaciones o dependencias entre las clases. Cada rectángulo puede tener hasta 3 compartimentos para albergar el nombre de la clase, sus atributos y sus métodos respectivamente (ver figura) clases. Las líneas representan asociaciones y pueden terminar con un adorno especial en un extremo (rombo, triángulo) que indica un tipo de asociación especial (ver figura) lineas.

8.9 Visibilidad

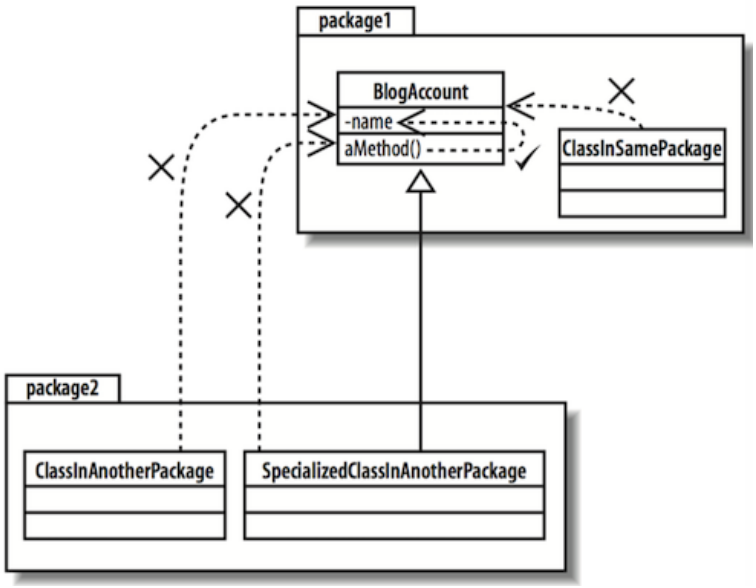
Los nombres de los atributos y los métodos de una clase pueden aparecer precedidos por un caracter especial: +, -, # y ~. Estos decoradores están asociados a lo que se conoce como la visibilidad del atributo o del método. La visibilidad indica desde donde está permitido acceder al elemento y desde donde no está permitido y ello es importante para asegurar que modificaciones posteriores no afecten a todo el código.

Un atributo siempre es accesible a los métodos de la misma clase al que pertenece, pero por lo general no es accesible a métodos de otras clases. Esta situación se indica como que el atributo es privado y se marca con un caracter "-". Si por alguna razón queremos que el atributo sea accesible desde todos lados (público) se indica con un caracter "+". Por lo general los métodos de una clase se hacen públicos, pero no necesariamente todos los métodos lo son. A veces queremos que el atributo sea privado con la excepción de las subclases (hijos) de la clase donde está definido. En ese caso se habla de una visibilidad protegida y se indica con un "#". Finalmente, es posible indicar un acceso libre a todas las clases que están encerradas en un mismo paquete y eso se indica con un caracter "~".

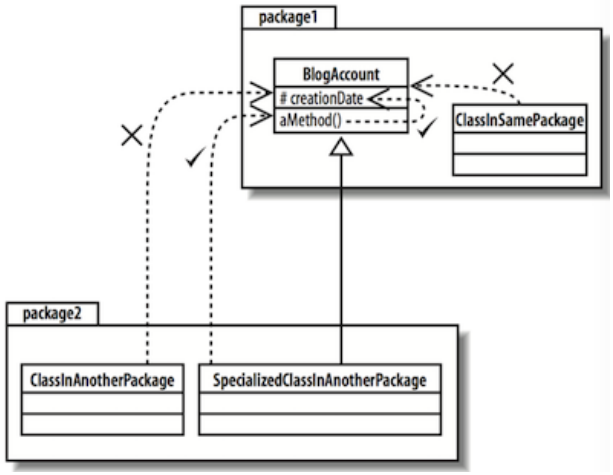
La figura siguiente ilustra un atributo público (`+ publicURL`) que puede ser accesado desde los métodos de la misma clase (`aMethod`), desde otra clase en el mismo paquete o desde cualquier clase en otro paquete.



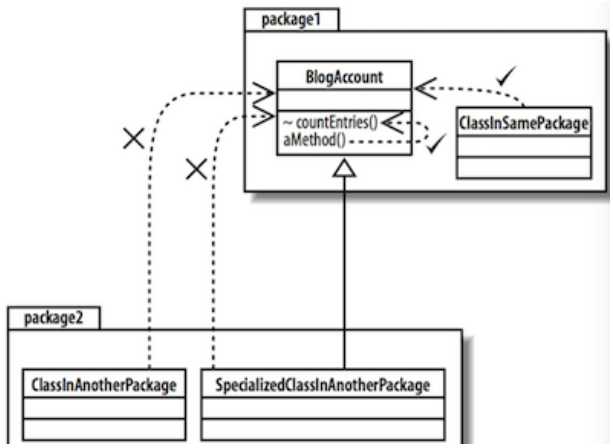
La siguiente figura muestra que sucede cuando un atributo (`name`) es declarado privado. Solo puede ser accedido por métodos de la misma clase.



En el caso de la visibilidad protegida (atributo `#creationDate`), además de los métodos de la misma clase, puede ser accedido desde métodos de subclases de `BlogAccount` como en este caso `SpecializedClassInAnotherPackage`, pero no desde otras clases, ni siquiera estando en el mismo paquete.



Finalmente, la figura siguiente muestra lo que sucede cuando a un elemento (en este caso el método `countEntries`) se le asigna visibilidad de paquete. En este caso, aparte de los métodos en la misma clase, sólo los métodos de clases en el mismo paquete pueden acceder a él.

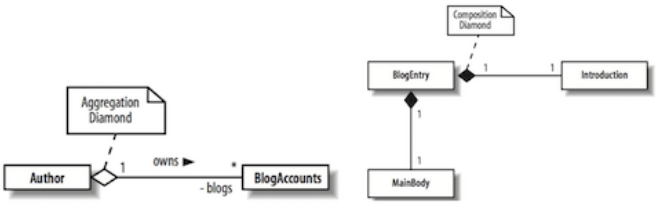


8.10 Asociaciones

Las clases de un modelo no son independientes. Para llevar a cabo una operación puede ser necesario que cooperen dos o más clases. La dependencia entre clases se indica mediante líneas que conectan las clases asociadas. Hay algunos tipos de asociación especiales que se indican con un símbolo en uno de los extremos de la línea que conecta las clases. Un rombo relleno al comienzo de la línea que conecta la clase A con la clase B denota que un objeto de la clase A contiene uno o más objetos de la clase B (composición). Si el rombo está vacío, la clase A contiene referencias a objetos de la clase B (agregación) . Un triángulo se usa para modelar una relación de especialización (clase - subclase).

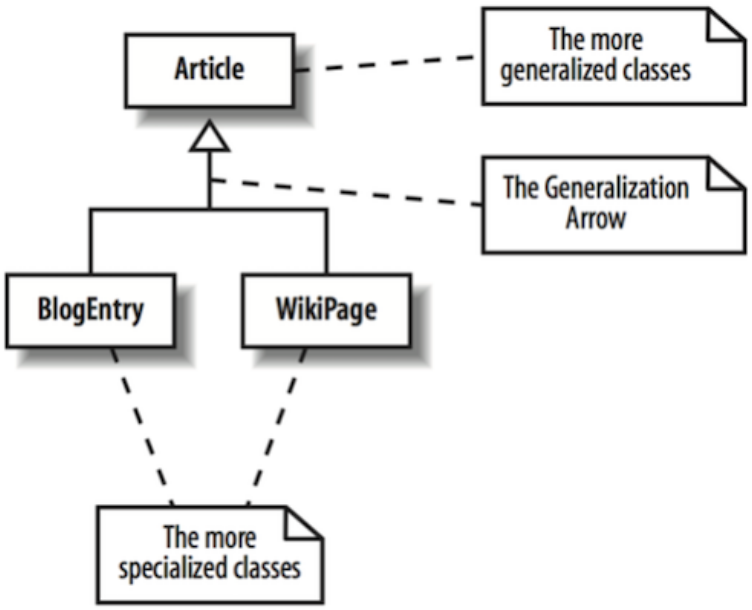
Las figuras muestran ejemplos de asociaciones de asociaciones de agregación

y composición.

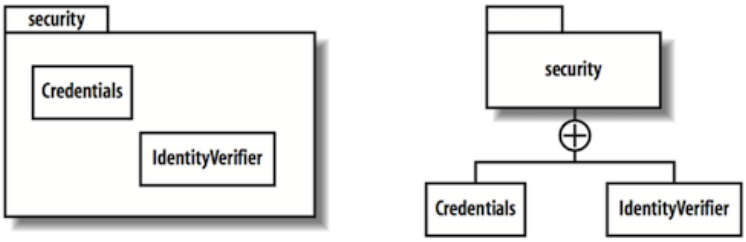


En el diagrama de la izquierda un autor tiene referencias a una o más cuentas. En el diagrama de la izquierda una entrada en el blog está compuesta de una introducción y un cuerpo. Tiene propiedad exclusiva de estos objetos y ellos no tienen sentido de existir fuera del objeto que los contiene.

La siguiente figura muestra un ejemplo de una asociación de generalización. Una entrada en un blog o una página wiki son casos especiales de un artículo. Esta es una relación de tipo-subtipo que a veces se suele llamar (equivocadamente en mi opinión) relación de herencia.

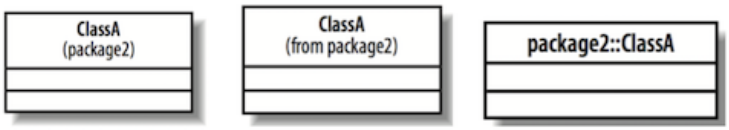


Para mostrar agrupaciones de clases en paquetes se usan carpetas. Debido a que los paquetes pueden contener otros paquetes, hay dos formas alternativas de mostrar en los diagramas (ver figura).



En el ejemplo, las clases `Credentials` y `IdentifyVerifier` están dentro del paquete `security`.

Si uno quiere que en una clase (A) se haga explícito que pertenece a un paquete determinado (package 2), puede usarse una de las siguientes 3 formas:

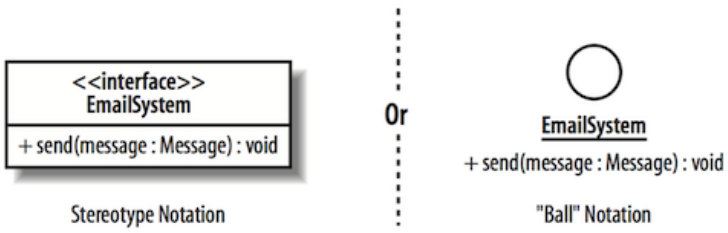


8.11 Interfaces

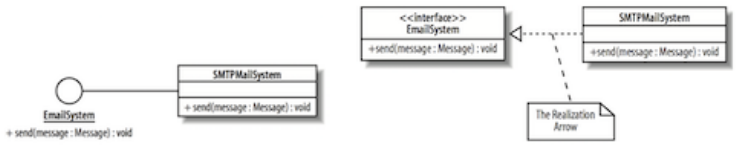
Una interfaz puede ser pensada como un contrato de disponibilidad de ciertas operaciones. Una interfaz define cierta funcionalidad en forma de métodos, pero no los detalles de la implementación. Una clase puede implementar una determinada interfaz, y en este caso está obligada a suministrar la implementación de cada uno de los métodos definidos en ella. Puede haber más de una clase que implementa la misma interfaz y también puede haber una clase que implementa más de una interfaz.

En algunos lenguajes de programación no hay soporte de interfaces (por ejemplo en Ruby) y se usa una clase abstracta para simular una interfaz.

En UML hay dos formas de mostrar una interfaz: igual que una clase, pero con estereotipo *interface* («*interface*») o simplemente como un círculo (ver figura).



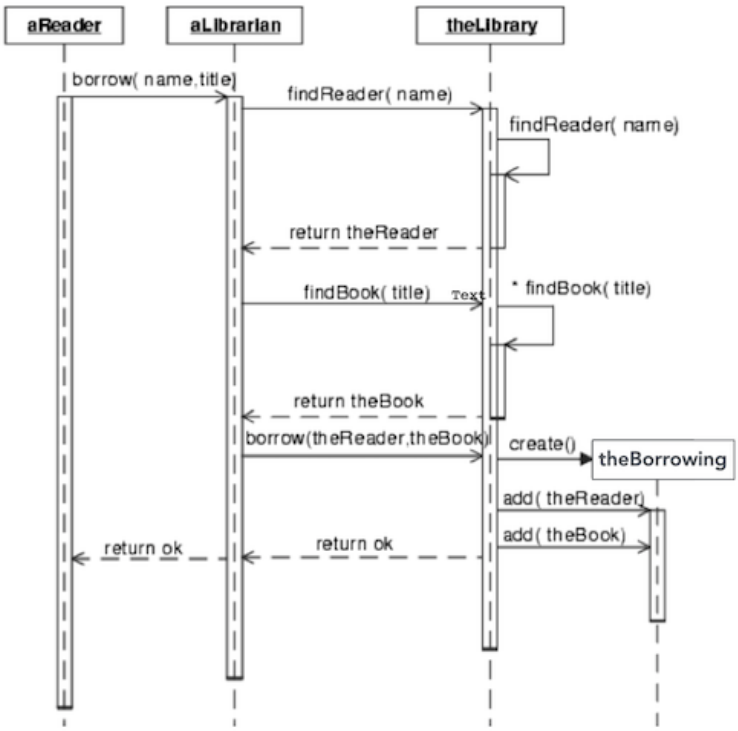
La manera como se muestra que una clase implementa una interfaz depende de cuál de las dos notaciones de interfaz se escogió



8.12 El diagrama de secuencia

Este tipo de diagrama UML es tal vez el segundo más utilizado después del diagrama de clases. Su utilidad radica en que nos permite entender la dinámica de la interacción de los objetos en el tiempo.

En este diagrama se muestra el intercambio de mensajes (flechas horizontales) entre un grupo seleccionado de objetos. El tiempo transcurre hacia abajo, por lo que una flecha más alta que otra indica que ello sucede con anterioridad. Veamos un ejemplo:



La figura muestra como se lleva a cabo la acción de crear un nuevo préstamo. Inicialmente, participan 3 objetos: uno de clase `Reader` (`aReader`), uno de clase `Librarian` (`aLibrarian`) y uno de clase `Library` (`theLibrary`). Inicialmente `aReader` envía el mensaje borrow a objeto aLibrarian pasando como parámetros el nombre del lector y el nombre del libro que le interesa. Un breve tiempo después (ver que la flecha parte más abajo) el objeto `aLibrarian` invoca el método `findReader` sobre el objeto `theLibrary`, pasando el

nombre del lector. El objeto `theLibrary`, después de buscar al lector, devuelve a `aLibrarian` el objeto que corresponde a ese lector (puede incluir muchos atributos). El objeto `Librarian` ahora invoca el método `findBook` sobre `theLibrary` y esta vez lo que retorna es el objeto que corresponde al libro pedido. Finalmente, el `Librarian` procede a enviar el mensaje para crear un nuevo préstamo sobre (`theBorrowing`) pasando como parámetros los objetos que corresponden al lector y al libro. El objeto `theLibrary` crea el objeto y agrega a él los datos del lector y del libro. Una vez que `aLibrarian` ha concluido su trabajo, retorna a `aReader` y la operación concluye.

Algunas observaciones importantes:

- normalmente, no se incluyen flechas para los retornos, salvo que se retornen objetos
- los rectángulos angostos sobre las líneas de tiempo denotan el período de tiempo en que el objeto está activo
- pueden haber invocaciones de métodos de un objeto sobre sí mismo