UNIDAD DIDÁCTICA II
TEMA 3 – Elaboración en la Iteración 1

1. MODELO DE CASOS DE USO: REPRESENTACIÓN DE LOS DIAGRAMAS DE SECUENCIA DEL SISTEMA

En la fase de inicio se ha realizado un trabajo ligero de requisitos para ayudar a decidir si merecía la pena más investigación seria en el proyecto y completada la planificación de la primera iteración, se decide abordar un escenario de éxito del caso de uso.

Antes de empezar el trabajo de diseño de la iteración 1, resulta útil realizar un estudio adicional del dominio del problema: aclaración de los eventos del sistema de entrada y salida relacionados con el sistema, que puede representarse en diagramas de secuencia UML.

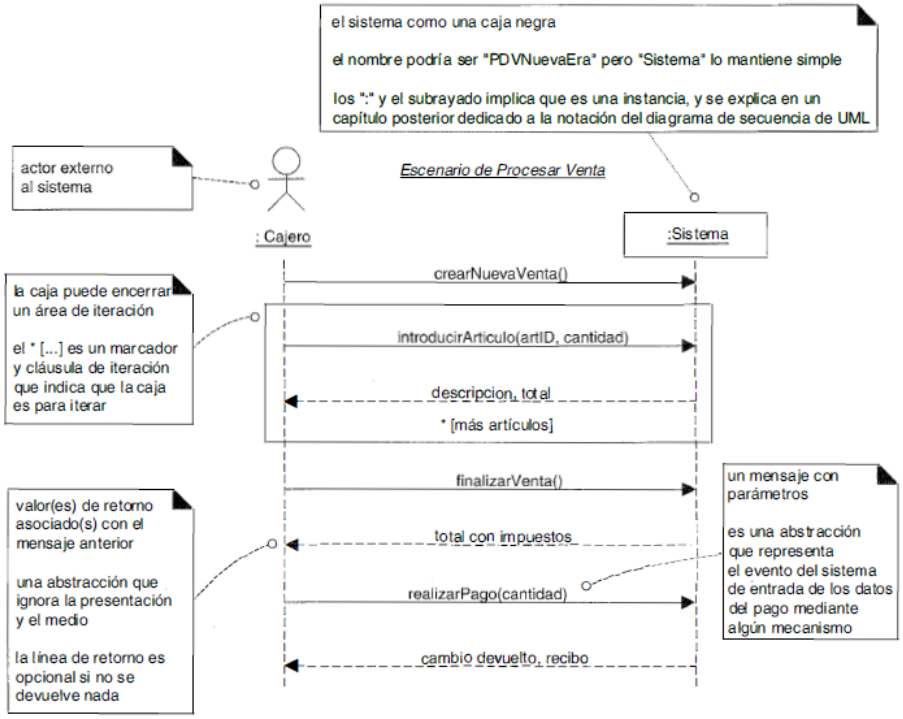
Un **diagrama de secuencia del sistema** es un artefacto creado de manera rápida y fácil, que muestra los eventos de entrada y salida relacionados con el sistema que se está estudiando. UML incluye la notación de los diagramas de secuencia para representar los eventos que parten de los actores externos hacia el sistema.

1.1. Comportamiento del sistema

Antes de continuar con el diseño lógico, es conveniente estudiar y definir su comportamiento como una "caja negra". El **comportamiento del sistema** es una descripción de qué hace el sistema, sin explicar cómo lo hace. Una parte de esa descripción es un diagrama de secuencia del sistema, los casos de uso y los contratos del sistema

1.2. Diagramas de secuencia del sistema

Los casos de uso describen cómo interactúan los actores externos con el sistema. Un actor genera eventos sobre un sistema, normalmente solicitando alguna operación como respuesta. Es deseable aislar e ilustrar las operaciones que un actor externo solicita a un sistema, porque constituyen una parte importante de la comprensión del comportamiento del sistema. UML incluye los diagramas de secuencia como notación que representa las interacciones de los actores y las operaciones que inician.



Un **diagrama de secuencia del sistema (DSS)** es un dibujo que muestra, para un escenario específico de un caso de uso, los eventos que generan los actores externos, el orden y los eventos entre los sistemas. Todos los sistemas se tratan como cajas negras. Destacan los eventos que cruzan los límites del sistema desde los actores a los sistemas. Debería hacerse un DSS para el escenario principal de éxito del caso de uso, los escenarios alternativos complejos o frecuentes.

En un DSS, el tiempo avanza hacia abajo, y la ordenación de los eventos debería seguir su orden en el caso de uso. Los eventos del sistema podrían contener parámetros.

Los DSS también pueden utilizarse para ilustrar las colaboraciones entre sistemas, aunque se pospone hasta una iteración posterior, puesto que esta iteración no incluye las colaboraciones con sistemas remotos.

1.3. Eventos del sistema y los límites del sistema

Para identificar los eventos del sistema, es necesario tener claros los **límites del sistema**. El límite del sistema normalmente se elige para que sea el propio sistema software (y posiblemente hardware) y un evento del sistema es un evento externo que lanza un estímulo directamente al software. Primero, se deben determinar los actores que interactúan directamente con el sistema software.

Los **eventos del sistema** (y sus operaciones del sistema asociadas) deberían expresarse al nivel de intenciones en lugar de en términos del medio de entrada físico o a nivel de elementos de la interfaz de usuario. También se mejora la claridad, el comenzar el nombre de un evento del sistema con un verbo (añadir..., insertar..., finalizar..., crear...) capturando la intención de la operación, al mismo tiempo que permanece abstracta y sin compromiso respecto a las elecciones de diseño sobre qué interfaz utilizar para capturar el evento del sistema.

A veces es deseable mostrar fragmentos del texto del caso de uso del escenario, con el fin de aclarar o enriquecer las dos vistas. El texto proporciona los detalles y el contexto; el diagrama resume visualmente la interacción. Los términos representados en los DSS son concisos y podrían necesitar una explicación más adecuada. Si no se explicó en los casos de uso, podría utilizarse el Glosario.

1.4. DSS en el UP

Los DSSs forman parte del Modelo de Casos de Uso. Son un ejemplo de los muchos posibles artefactos o actividades de análisis y diseño de utilidad que los documentos UP o el RUP no mencionan.

Fases

- **Inicio**: Los DSS no se incentivan normalmente en la fase de inicio.
- **Elaboración**: La mayoría de los DSS se crean durante la elaboración, cuando es útil identificar los detalles de los eventos del sistema.

No es necesario crear DSS para todos los escenarios de todos los casos de uso. Se crearán sólo para algunos escenarios seleccionados de la iteración actual. Sólo debería llevar unos pocos minutos o una media hora la creación de los DSSs.

Tabla 9.1. Muestra de los artefactos UP y evolución temporal. c – comenzar; r – refinar.

Disciplina	*Artefacto* *Iteración →*	*Inicio* *I1*	*Elab.* *E1...En*	*Const.* *C1...Cn*	*Trans.* *T1...T2*
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	***Modelo de Casos de Uso (DSS)*** Visión Especificación Complementaria Glosario	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

1.5. Artefactos del UP

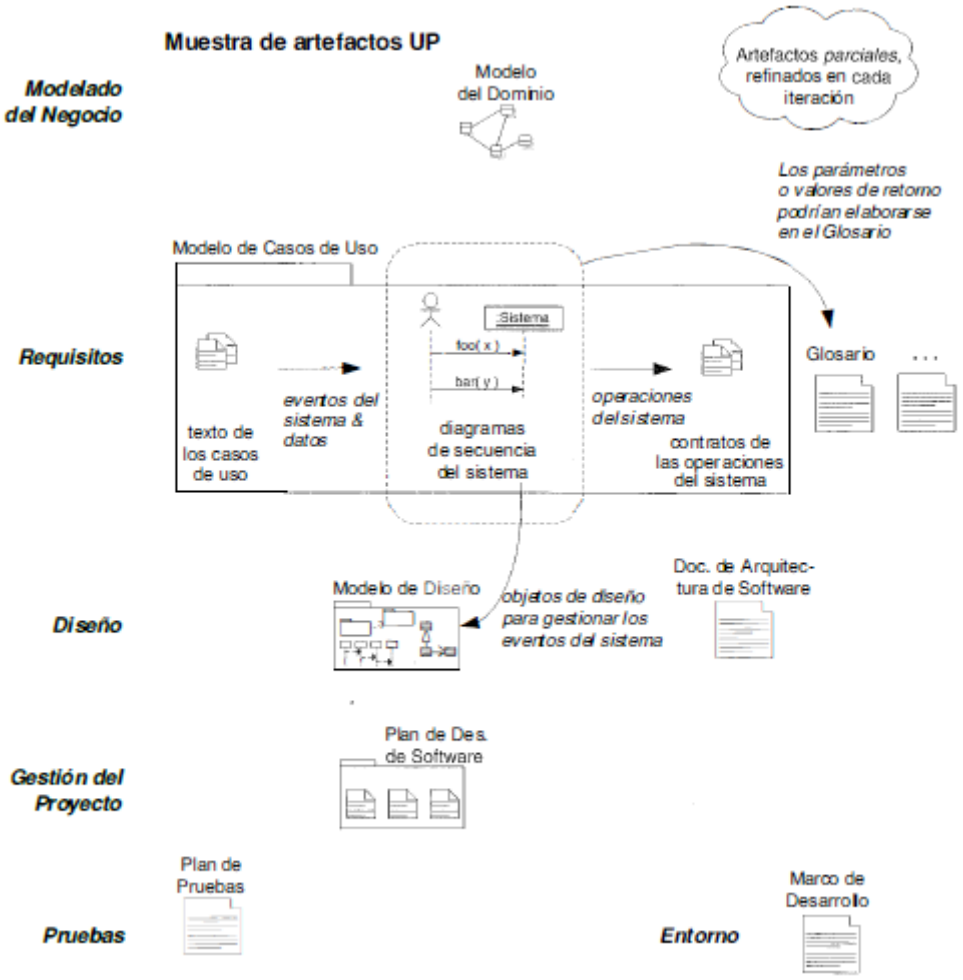


Figura 9.6. Muestra de la influencia de los artefactos UP.

2. MODELO DEL DOMINIO: VISUALIZACIÓN DE CONCEPTOS

Un modelo del dominio se utiliza como fuente de inspiración para el diseño de los objetos software. Muestra las clases conceptuales significativas en un dominio del problema y es el artefacto más importante que se crea durante el análisis orientado a objetos.

La identificación de un conjunto rico de objetos o clases conceptuales es una parte esencial del análisis orientado a objetos. La identificación de las clases conceptuales forma parte del estudio del dominio del problema. UML contiene notación, en forma de diagramas de clases, para representar los modelos del dominio. Es una representación de las clases conceptuales del mundo real, no de componentes software.

2.1. Modelos del dominio

Un **modelo del dominio** es una representación *visual* de las clases conceptuales u objetos del mundo real en un dominio de interés. También se denomina **modelos conceptuales**, **modelo de objetos del dominio** y **modelos de objetos de análisis**. El UP define un Modelo de Dominio como uno de los artefactos que podrían crearse en la disciplina del Modelado del Negocio. En la notación UML, un modelo del dominio se representa con un conjunto de **diagramas de clases** en los que no se define ninguna operación:

- Objetos del dominio o clases conceptuales.
- Asociaciones entre las clases conceptuales.
- Atributos de las clases conceptuales.

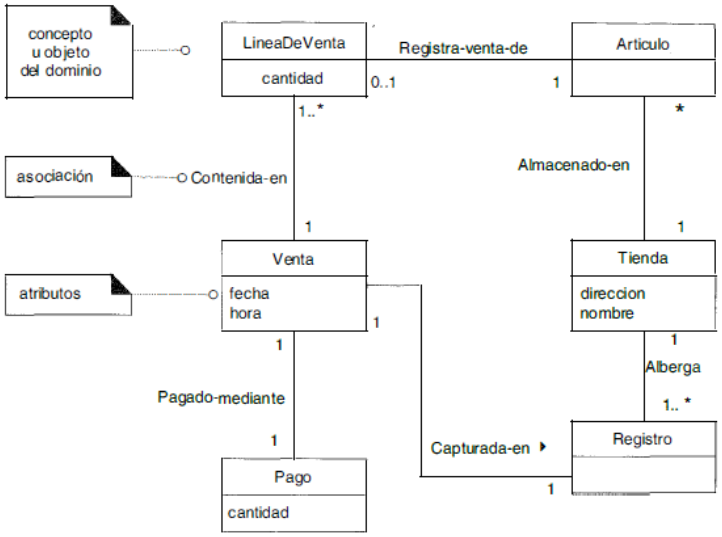


Figura 10.1. Modelo del dominio parcial —un diccionario visual—. El número de cada extremo de la línea indica la multiplicidad, que se describirá en un capítulo posterior.

El modelo de dominio visualiza y relaciona algunas palabras o clases conceptuales del dominio. Describe una *abstracción* de las clases conceptuales. La información que presenta podría haberse expresado en prosa, mediante sentencias en el Glosario o en algún otro sitio. El modelo del dominio podría considerarse como un *diccionario visual* de las abstracciones relevantes, vocabulario del dominio e información del dominio

Un modelo del dominio es una representación de las cosas del mundo real del dominio de interés, *no* de componentes software u objetos software con responsabilidades. No son adecuados en un modelo del dominio:

- Artefactos software.
- Responsabilidades o métodos.

Clases conceptuales

El modelo del dominio muestra las clases conceptuales o vocabulario del dominio. Una clase conceptual es una idea, cosa u objeto y podría considerarse en términos de su símbolo, intensión, y extensión.

- **Símbolo**: palabras o imágenes que representan una clase conceptual.
- **Intensión**: la definición de una clase conceptual.
- **Extensión**: el conjunto de ejemplos a los que se aplica la clase conceptual.

Cuando se crea un modelo del dominio, el símbolo y la vista intensional de la clase conceptual son los que tienen mayor interés práctico.

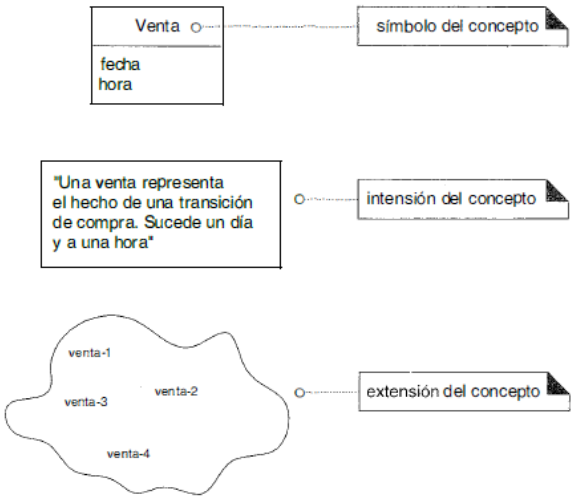


Figura 10.4. Una clase conceptual tiene un símbolo, una intensión y una extensión.

Modelos y descomposición del dominio

Una estrategia común para tratar la complejidad de los problemas software es mediante la división del espacio del problema en unidades fáciles de comprender. En el **análisis estructurado**, la dimensión de la descomposición es por procesos o por *funciones*. En el análisis orientado a objetos, la dimensión de la descomposición es por cosas o entidades del dominio. La principal tarea del análisis es identificar diferentes conceptos en el dominio del problema y documentar el resultado en un modelo del dominio.

2.2. Identificación de las clases conceptuales

En el desarrollo iterativo, incrementalmente se construye un modelo del dominio a lo largo de varias iteraciones en la fase de elaboración. En cada una, el modelo de dominio se limita a los escenarios anteriores y actual en estudio, en lugar

de un modelo de "gran explosión". La tarea central es identificar las clases conceptuales relacionadas con el escenario que se está diseñando.

- Es mejor especificar en exceso un modelo del dominio con muchas clases conceptuales de grano fino que especificar por defecto.
- No es mejor si contiene pocas clases conceptuales.
- Es normal obviar clases conceptuales durante la etapa de identificación inicial, y descubrirlas más tarde al considerar los atributos y asociaciones.
- No se excluye una clase conceptual porque los requisitos no indican ninguna necesidad obvia para registrar información sobre ella o porque la clase conceptual no tiene atributos.
- Es válido tener clases conceptuales sin atributos, o clases conceptuales con un rol puramente de comportamiento en el dominio.

Estrategias para identificar clases conceptuales

Dos técnicas:

1. Utilización de una lista de categorías de clases conceptuales.
2. Identificación de frases nominales.

Otra es el uso de **patrones de análisis**, modelos de dominios parciales existentes creados por expertos.

Utilización de una lista de categorías de clases conceptuales

Tabla 10.1. Lista de categorías de clases conceptuales.

Categoría de clase conceptual	*Ejemplos*
objetos tangibles o físicos	*Registro* *Avion*
especificaciones, diseños o descripciones de la cosas	*EspecificacionDelProducto* *DescripcionDelVuelo*
lugares	*Tienda*
transacciones	*Venta, Pago* *Reserva*
líneas de la transacción	*LineaDeVenta*
roles de la gente	*Cajero* *Piloto*
contenedores de otras cosas	*Tienda, Lata* *Avion*
cosas en un contenedor	*Articulo* *Pasajero*
otros sistemas informáticos o electromecánicos externos al sistema	*SistemaAutorizacionPagoCredito* *ControlDeTraficoAereo*
conceptos abstractos	*Ansia* *Acrofobia*
organizaciones	*DepartamentoDeVentas* *CompañiaAerea*
hechos	*Venta, Pago, Reunion* *Vuelo, Colision, Aterrizaje*

Tabla 10.1. Lista de categorías de clases conceptuales. *(Continuación)*

Categoría de clase conceptual	*Ejemplos*
procesos (normalmente *no* se representan como conceptos, pero podría ocurrir)	*VentaDeUnProducto* *ReservaUnAsiento*
reglas y políticas	*PoliticaDeReintegro* *PoliticaDeCancelación*
catálogos	*CatalogoDeProductos* *CatalogoDePiezas*
registros de finanzas, trabajo, contratos, cuestiones legales	*Recibo, LibroMayor, ContratoEmpleo* *RegistroMantenimiento*
instrumentos y servicios financieros	*LineaDeCredito* *Stock*
manuales, documentos, artículos de referencia, libros	*ListaDeCambiosDePreciosDiarios* *ManualReparaciones*

Descubrimiento de clases conceptuales mediante la identificación de frases nominales

Mediante análisis lingüístico se pueden identificar los nombres y frases nominales en las descripciones textuales de un dominio, y considerarlos como clases conceptuales o atributos candidatos (aunque no es posible realizar una correspondencia mecánica de nombres a clases). Los casos de uso en formato completo constituyen una descripción excelente a partir de la cual extraer este análisis.

Algunas de las frases nominales son clases conceptuales candidatas, que pueden hacer referencia a clases conceptuales que se ignoran en esta iteración. Un punto débil de este enfoque es la imprecisión del lenguaje natural. Se recomienda que se combine con la técnica la *Lista de Categorías de Clases Conceptuales*.

A partir del análisis de la Lista de Categorías de Clases Conceptuales y las frases nominales, se genera una lista de clases conceptuales candidatas del dominio. La lista está restringida a los requisitos y simplificaciones que se están estudiando. No existe una lista "correcta". Es una colección arbitraria de abstracciones y vocabulario del dominio que el modelador considera relevantes.

Un factor a tener en cuenta es que no es útil mostrar un informe de otra información en un modelo del dominio puesto que toda esta información se deriva de otras fuentes; duplica información.

2.3. Guías para el modelado de negocio

Cómo hacer un modelo del dominio

1. Liste las clases conceptuales candidatas, utilizando las técnicas de la Lista de Categorías de Clases Conceptuales y la identificación de frases nominales, relacionadas con los requisitos actuales en estudio.
2. Represéntelos en un modelo del dominio.
3. Añada las asociaciones necesarias para registrar las relaciones que hay que mantener en memoria (se discutirá en un capítulo siguiente).
4. Añada los atributos necesarios para satisfacer los requisitos de información (se discutirá en un capítulo siguiente).

Un método útil auxiliar es aprender y copiar patrones de análisis.

Nombrar y modelar cosas: el cartógrafo

Haga un modelo del dominio con el espíritu del modo de trabajo de los cartógrafos:
- Utilice los nombres existentes en el territorio.
- Excluya las características irrelevantes.
- No añada cosas que no están ahí.

Un modelo del dominio es un tipo de mapa de conceptos o cosas de un dominio. Destacando el rol analítico de un modelo del dominio:

- Utilizar el vocabulario del dominio al nombrar los nombres de las clases conceptuales y los atributos.
- Un modelo de dominio puede excluir clases conceptuales del dominio del problema que no son pertinentes para los requisitos.
- El modelo del dominio debe excluir cosas que *no* se encuentran en el dominio del problema que se está estudiando.

Esta estrategia se resume como *utilice el vocabulario del dominio*.

El error más típico al crear un modelo del dominio es representar algo como un atributo cuando debería haber sido un concepto. Para ayudar a prevenir este error, si no se considera alguna clase conceptual X que sea un número o texto en el mundo real, X es probablemente una clase conceptual, no un atributo. En caso de duda, es mejor considerarlo como un concepto separado. Los atributos deberían ser bastante raros en un modelo del dominio.

Un modelo del dominio no es absolutamente correcto o equivocado, sino más o menos útil; es una herramienta de comunicación. Algunos sistemas software son para dominios que encuentran muy poca analogía con dominios naturales o de negocios. Es posible crear un modelo del dominio en estos dominios, pero requiere un alto grado de abstracción.

2.4. Clases conceptuales de especificación o descripción

La necesidad de las clases conceptuales de especificación es habitual en muchos modelos del dominio. Los objetos de descripción o especificación están fuertemente relacionados con las cosas que describen. En un modelo del dominio, es típico establecer que una *Especificación De X Describe un X*.

La necesidad de clases conceptuales de especificación es habitual en los dominios de ventas y productos o la fabricación. Son muy frecuentes, no es un concepto de modelado raro. Se añade una clase conceptual de especificación o descripción cuando:

- Se necesita la descripción de un artículo o servicio, independiente de la existencia actual de algún ejemplo de esos artículos o servicios.
- La eliminación de instancias de las cosas que describen da como resultado una pérdida de información que necesita mantenerse, debido a la asociación incorrecta de información con la cosa eliminada.
- Reduce información redundante o duplicada.

2.5. Notación UML, modelos y métodos: perspectivas múltiples

El UP define el Modelo del Dominio, que se representa con la notación UML. Sin embargo, no existe un término "Modelo del Dominio" en la documentación oficial de UML. UML simplemente describe tipos de diagramas, como los diagramas de clases y los diagramas de secuencia. No superpone un método o perspectiva de modelado sobre ellos.

No hay que confundir la notación básica de los diagramas UML, con su aplicación para visualizar distintos tipos de modelos definidos por los metodologistas. Otro ejemplo es que los diagramas de secuencia UML se pueden utilizar para representar el paso de mensajes entre los objetos software (Modelo de Diseño del UP), o la interacción entre personas y grupos en el mundo real (Modelo de Objetos del Negocio del UP).

La misma notación basada en diagramas se puede utilizar en tres perspectivas y tipos de modelos:

1. **Perspectiva esencial o conceptual**: se interpreta que los diagramas describen cosas del mundo real o de un dominio de interés.
2. **Perspectiva de especificación**: se interpreta que los diagramas describen abstracciones software o componentes con especificaciones e interfaces, pero no comprometidas a ninguna implementación en particular.
3. **Perspectiva de implementación**: se interpreta que los describen implementaciones software con una tecnología y lenguaje particular.

Superposición de terminología: UML vs. métodos

Es útil distinguir entre la perspectiva de un analista que mira conceptos del mundo real y los diseñadores de software que especifican componentes software. UML se puede utilizar para ilustrar ambas perspectivas con una notación y perspectiva muy similar:

- **Clase conceptual**: concepto o cosa del mundo real. Una perspectiva conceptual o esencial. El Modelo del Dominio del UP contiene clases conceptuales.
- **Clase software**: una clase que representa una perspectiva de especificación o implementación de un componente software, independientemente del proceso o método.
- **Clase de diseño**: un miembro del Modelo de Diseño del UP. Es un sinónimo de clase software, pero por alguna razón deseo resaltar que es una clase del Modelo de Diseño. El UP permite que una clase de diseño tenga perspectiva de especificación o implementación, según desee el modelador.
- **Clase de implementación**: una clase implementada en un lenguaje orientado a objetos como Java.
- **Clase**: como en UML, el término general que representa o una cosa del mundo real (una clase conceptual) o del software (una clase software).

2.6. Reducción del salto en la representación

Eligiendo nombres que reflejan el vocabulario del dominio se favorece la rápida comprensión y proporciona una pista acerca de lo que se espera del trozo de código de la clase software. El Modelo del Dominio proporciona un diccionario visual del vocabulario y conceptos del dominio a partir de los cuales nos inspiramos para nombrar algunas cosas del diseño software.

El **salto de la representación** o salto semántico define el salto entre el modelo mental del dominio y su representación en el software. Las tecnologías de objetos permiten partir el código en clases cuyos nombres reflejan el tipo de partición que se percibe en el dominio. Esta estrecha correspondencia reduce el salto de la representación, acelera la comprensión del código existente y sugiere formas "naturales" para extender el código que se corresponden análogamente con el dominio.

Una reducción del salto en la representación es útil, pero se puede sostener que es secundario a la ventaja que ofrecen los objetos de facilitar los cambios y extensiones, y el soporte que ofrecen para el manejo y ocultación de la complejidad.

2.7. Modelos del Dominio en el UP

Un Modelo del Dominio, normalmente, se inicia y completa en la elaboración.

Inicio

Los modelos del dominio no se incentivan fuertemente en la fase de inicio, puesto que el propósito no es llevar a cabo un estudio serio.

Elaboración

El Modelo del Dominio se crea sobre todo durante las iteraciones de la elaboración, cuando la necesidad más importante es entender los conceptos relevantes y trasladar algunos a clases software durante el trabajo de diseño.

El desarrollo de un modelo del dominio (parcial, desarrollado incrementalmente) en cada iteración debería durar unas pocas horas. Esto se acorta mediante el uso de patrones de análisis predefinidos.

Tabla 10.2. Muestra de los artefactos UP y evolución temporal. c – comenzar; r - refinar.

Disciplina	*Artefacto* *Iteración →*	*Inicio* *I1*	*Elab.* *E1...En*	*Const.* *C1...Cn*	*Trans.* *T1...T2*
Modelado del Negocio	***Modelo del Dominio***		c		
Requisitos	Modelo de Casos de Uso Visión Especificación Complementaria Glosario	c c c c	r r r r		
Diseño	Modelo de Diseño Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

El Modelo de Objetos del Negocio del UP vs. El Modelo del Dominio

El Modelo del Dominio del UP es una variación oficial del menos común Modelo de Objetos del Negocio del UP (BOM). Es un tipo de modelo de empresa utilizado para describir el negocio completo. El BOM del UP sirve como abstracción del modo en el que los trabajadores y las entidades del negocio necesitan relacionarse y cómo necesitan colaborar para llevar a cabo el negocio. Se representa con varios diagramas diferentes que muestran cómo funciona toda la empresa. El UP define el Modelo del Dominio como un artefacto subconjunto o una especialización del BOM.

2.8. Artefactos del UP

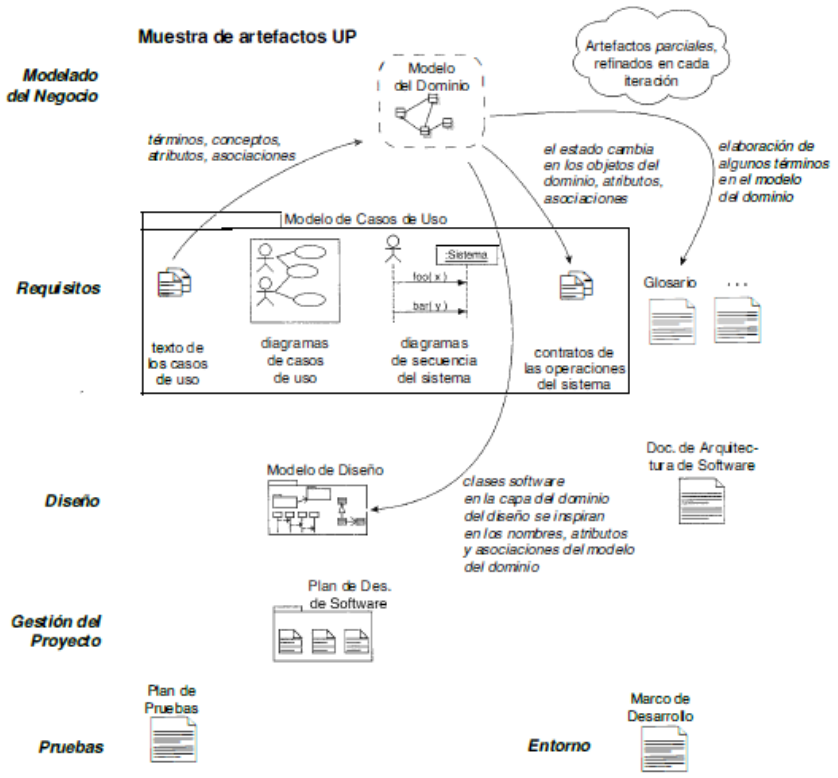


Figura 10.12. Muestra de la influencia entre los artefactos UP.

3. MODELO DEL DOMINIO: AÑADIR ASOCIACIONES

Identificar las asociaciones entre clases conceptuales que son necesarias para satisfacer los requisitos de información de los escenarios actuales que se están desarrollando, ayudan a entender el modelo del dominio

3.1. Asociaciones

Una **asociación** es una relación entre tipos (instancias de estos tipos) que indica alguna conexión significativa e interesante. En UML se definen como "la relación semántica entre dos o más clasificadores que implica conexiones entre sus instancias".

Las asociaciones que merece la pena registrar implican conocimiento de una relación que es necesario conservar durante algún tiempo. En un modelo del dominio con *n* clases del dominio diferentes, pueden existir *n·(n - 1)* asociaciones entre diferentes clases conceptuales. Muchas líneas en un diagrama añadirán "ruido visual" y lo hará menos comprensible.

> Considere la inclusión de las siguientes asociaciones en un modelo del dominio:
> - Asociaciones de las que es necesario conservar el conocimiento de la relación durante algún tiempo (asociaciones "necesito-conocer").
> - Asociaciones derivadas de la Lista de Asociaciones Comunes.

3.2. Notación de las asociaciones en UML

Una asociación se representa como una línea entre clases con un nombre de asociación. Es inherentemente bidireccional, lo que significa que, desde las instancias de cualquiera de las dos clases, es posible el recorrido lógico hacia la otra. Este recorrido es puramente abstracto. Los extremos de la asociación podrían contener una expresión de multiplicidad que indica la relación numérica entre las instancias de las clases.

Una "flecha de dirección de lectura" opcional indica la dirección de la lectura del nombre de la asociación; no indica la dirección de la visibilidad o navegación. Si no está presente, la convención es leer la asociación de izquierda a derecha o de arriba hacia abajo.

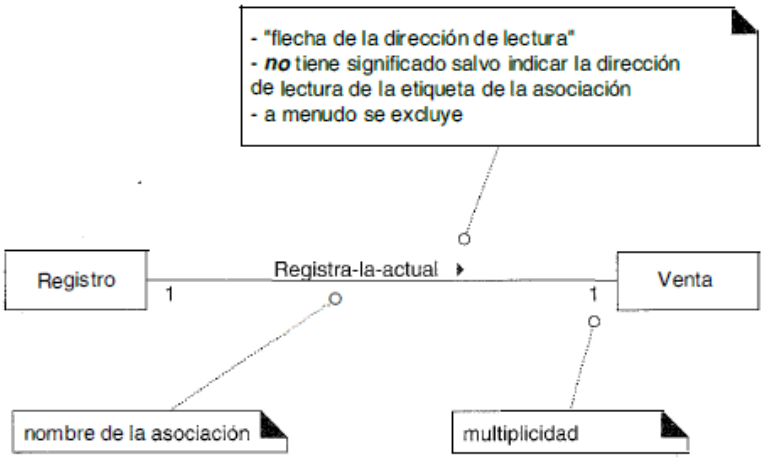


Figura 11.2. Notación UML para las asociaciones.

3.3. Localización de las asociaciones – lista de asociaciones comunes

Tabla 11.1. Lista de asociaciones comunes.

Categoría	*Ejemplos*
A es una parte física de B	*Cajon-Registro (o más concretamente, TPDV) Ala-Avion*
A es una parte lógica de B	*LineaDeVenta-Venta EtapaVuelo-RutaVuelo*
A está contenido físicamente en B	*Registo-Tienda, Articulo-Estanteria Pasajero-Avion*
A está contenido lógicamente en B	*DescripcionDelArticulo-Catalogo Vuelo-PlanificacionVuelo*
A es una descripción de B	*DescripcionDelArticulo-Articulo DescripcionDelVuelo-Vuelo*
A es un línea de una transacción o informe de B	*LineaDeVenta-Venta TrabajoMantenimiento-RegistroDe-Mantenimiento*
A se conoce/registra/recoge/informa/captura en B	*Venta-Registro Reserva-ListaPasajeros*
A es miembro de B	*Cajero-Tienda Piloto-CompañiaAerea*
A es una subunidad organizativa de B	*Departamento-Tienda Mantenimiento-CompañiaAerea*
A utiliza o gestiona B	*Cajero-Registro Piloto-Avion*
A se comunica con B	*Cliente-Cajero AgenteDeReservas-Pasajero*
A está relacionado con una transacción B	*Cliente-Pago Pasajero-Billete*
A es una transacción relacionada con otra transacción B	*Pago-Venta Reserva-Cancelacion*
A está al lado de B	*LineaDeVenta-LineaDeVenta Ciudad-Ciudad*
A es propiedad de B	*Registro-Tienda Avion-CompañiaAerea*
A es un evento relacionado con B	*Venta-Cliente,Venta-Tienda Salida-Vuelo*

Asociaciones de prioridad alta

Son útiles incluirlas en un modelo del dominio:

- A es una *parte lógica o física* de B.

- A está *contenida física o lógicamente* en B.
- A *se registra* en B.

Guías para las asociaciones

- Céntrese en aquellas asociaciones para las que se necesita conservar el conocimiento de la relación durante algún tiempo (asociaciones "necesito-conocer").
- Es más importante identificar *clases conceptuales* que identificar asociaciones.
- Demasiadas asociaciones tienden a confundir un modelo del dominio en lugar de aclararlo. Su descubrimiento puede llevar tiempo, con beneficio marginal.
- Evite mostrar asociaciones redundantes o derivadas.

Es más importante encontrar las *clases conceptuales* que las asociaciones. La mayoría del tiempo dedicado a la creación del modelo del dominio debería emplearse en la identificación de las clases conceptuales, no de las asociaciones.

3.4. Roles

Cada extremo de una asociación se denomina **rol**. Los roles pueden tener opcionalmente:

- Nombre.
- Expresión de multiplicidad.
- Navegabilidad.

Multiplicidad

La **multiplicidad** define cuántas instancias de una clase A pueden asociarse con una instancia de una clase B. El valor de la multiplicidad indica cuántas instancias se puede asociar legalmente con otra, en un momento concreto, en lugar de a lo largo de un periodo de tiempo. El valor de la multiplicidad depende de nuestros intereses como modeladores y pone de manifiesto una restricción de diseño que será reflejada en el software.

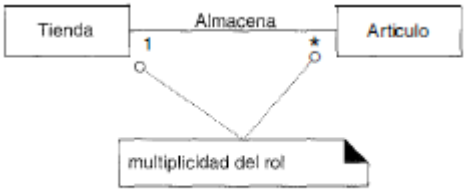


Figura 11.3. Multiplicidad de una asociación.

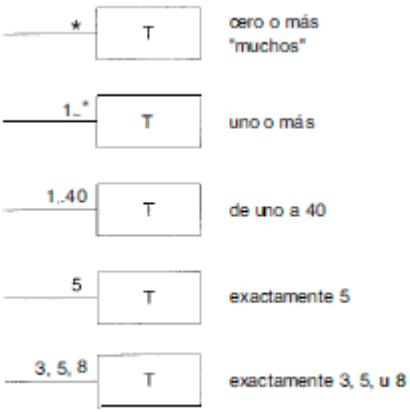


Figura 11.4. Valores de la multiplicidad.

3.5. Asignación de nombres a las asociaciones

Es más importante encontrar las *clases conceptuales* que las asociaciones. La mayoría del tiempo dedicado a la creación del modelo del dominio debería emplearse en la identificación de las clases conceptuales, no de las asociaciones.

Deben comenzar con una letra mayúscula.

3.6. Múltiples asociaciones entre dos tipos

Dos tipos podrían tener múltiples asociaciones entre ellos. Si son relaciones diferentes, se deberían mostrar de manera separada.

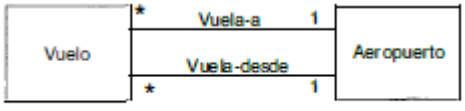


Figura 11.7. Múltiples asociaciones.

3.7. Asociaciones e implementación

Durante el modelado del dominio, una asociación no es una declaración sobre el flujo de datos, variables de instancia o conexiones entre objetos en una solución software; es una manifestación de que una relación es significativa. Desde un punto de vista práctico, muchas de estas relaciones se implementarán generalmente en software como caminos de navegación y visibilidad, pero su presencia en una vista conceptual de un modelo de dominio no requiere su implementación.

Al crear un modelo de dominio, podríamos definir asociaciones que no son necesarias durante la implementación. A la inversa, descubrir asociaciones que necesitan implementarse pero que se obviaron durante el modelado del dominio. El modelo del dominio puede actualizarse para reflejar estos descubrimientos.

Posponer las consideraciones de diseño libera de informaciones y decisiones extrañas mientras hacemos un estudio de "análisis" puro y maximiza las opciones de diseño más adelante.

Asociaciones necesito-conocer vs. comprensión

Un criterio necesito-conocer estricto para el mantenimiento de las asociaciones generará un "modelo de información" mínimo de lo que se necesita para modelar el dominio del problema. Sin embargo, este enfoque podría crear un modelo que no transmite una comprensión completa del dominio.

Además de ser un modelo necesito-conocer, el modelo del dominio es una herramienta de comunicación con la que entender y comunicar a otros los conceptos importantes y sus relaciones. Un buen modelo se sitúa en alguna parte entre un modelo necesito-conocer mínimo y uno que ilustra cada relación concebible

> Céntrese en las asociaciones necesito-conocer, pero contemple las asociaciones de sólo-comprensión para enriquecer el conocimiento básico del dominio.

4. MODELO DEL DOMINIO: AÑADIR ATRIBUTOS

Identificar aquellos atributos de las clases conceptuales que se necesitan para satisfacer los requisitos de información de los actuales escenarios en estudio.

4.1. Atributos

Un **atributo** es un valor de datos lógico de un objeto. Son aquellos para los que los requisitos sugieren o implican una necesidad de registrar la información.

Los atributos se muestran en el segundo compartimento del rectángulo de la clase. Sus tipos podrían mostrarse opcionalmente.

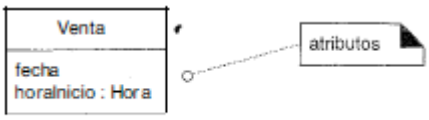


Figura 12.1. Clases y atributos.

4.2. Tipos de atributos válidos

Algunas cosas que no deberían representarse como atributos, sino como asociaciones.

Mantener atributos simples

La mayoría de los atributos simples son los que se conocen como tipos de datos primitivos. El tipo de un atributo, normalmente, no debería ser un concepto de dominio complejo. Los atributos en un modelo del dominio deberían ser, preferiblemente, **atributos simples** o **tipos de datos**. Los tipos de datos de los atributos muy comunes incluyen: *Boolean*, *Fecha*, *Número*, *String* (*Texto*), *Hora*.

Un error típico es modelar un concepto del dominio complejo como un atributo. Debería relacionarse mejor mediante una asociación, no con un atributo.

Perspectiva conceptual vs. Implementación: atributos en el código

El modelo del dominio se centra en declaraciones conceptuales puras sobre un dominio del problema, no en componentes software. Posteriormente, durante el trabajo de diseño e implementación, las asociaciones entre objetos representadas en el modelo del dominio, a menudo, se implementarán como atributos que referencian a otros objetos software complejos. La decisión debe posponerse durante el modelado del dominio.

Tipos de datos

Los atributos deben ser, generalmente, **tipos de datos**. En UML implica un conjunto de valores para los cuales no es significativa una identidad única. Todos los tipos primitivos (número, string) son tipos de datos UML, pero no todos los tipos de datos son primitivos. Estos valores de tipos de datos también se conocen como **objetos valor**.

Los tipos de datos se hacen atributo si se considera de manera natural como un número, string, booleano, fecha u hora (etcétera); en otro caso, como una clase conceptual aparte.

4.3. Clases de tipos de datos no primitivos

El tipo de un atributo podría representarse como una clase no primitiva por derecho propio en un modelo del dominio. Se representa lo que podría considerarse, inicialmente, como un tipo de dato primitivo como una clase no primitiva si:

- Está compuesto de secciones separadas.
- Hay operaciones asociadas con él, como análisis sintáctico o validación.
- Tiene otros atributos.
- Es una cantidad con una unidad.
- Es una abstracción de uno o más tipos con alguna cualidad.

Un tipo de datos que es una clase no primitiva, con sus propios atributos y asociaciones, podría ser interesante mostrarla como una clase conceptual en su propio rectángulo. No existe una respuesta correcta; depende de cómo se esté utilizando el modelo del dominio como herramienta de comunicación, y la importancia de los conceptos en el dominio.

4.4. Ningún atributo como clave ajena

No se deberían utilizar los atributos para relacionar las clases conceptuales en el modelo del dominio. La violación más típica de este principio es añadir un tipo de **atributo de clave ajena** para asociar dos tipos. La mejor manera es con una asociación, no con un atributo de clave ajena.

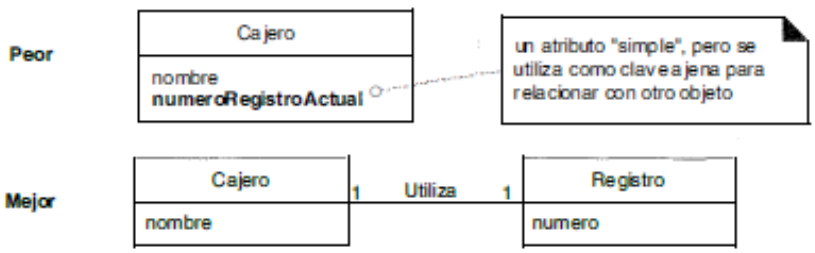


Figura 12.5. No utilice atributos como claves ajenas.

4.5. Modelado de cantidades y unidades de los atributos

La mayoría de las cantidades numéricas no deberían representarse simplemente como números. Es aceptable recoger su representación en la sección de atributos del rectángulo de clase.

La cantidad se puede calcular a partir del valor de la multiplicidad actual de la relación, de ahí que pudiera caracterizarse como un **atributo derivado**, puede derivarse a partir de otra información. En UML, un atributo derivado se indica con el símbolo "/".

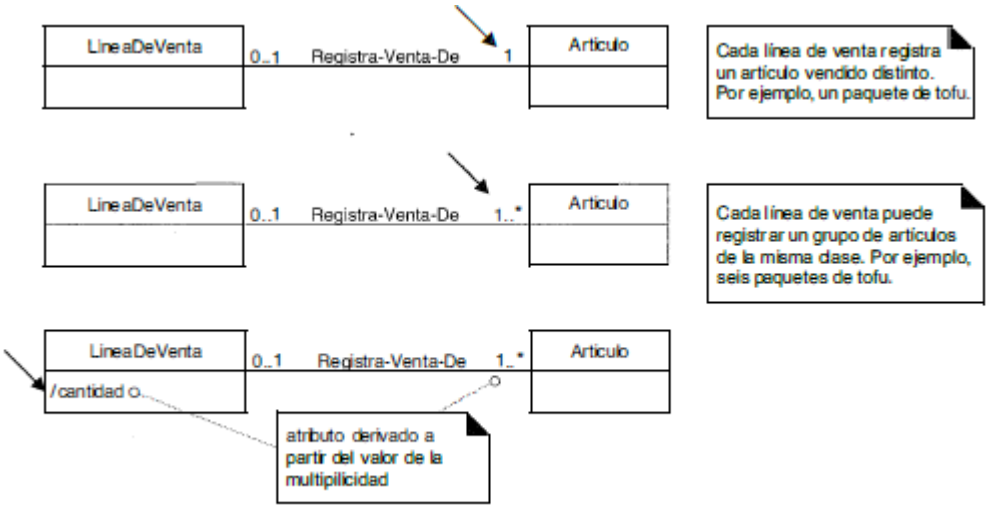


Figura 12.8. Registro de la cantidad de artículos vendidos en una línea de venta.

4.6. Conclusión del Modelo del Dominio

No existe un único modelo correcto. Todos los modelos son aproximaciones del dominio que estamos intentando entender. Un buen modelo del dominio captura las abstracciones y la información esenciales necesarias para entender el dominio en el contexto de los requisitos actuales, y ayuda a la gente a entender el dominio (sus conceptos, terminología y relaciones).

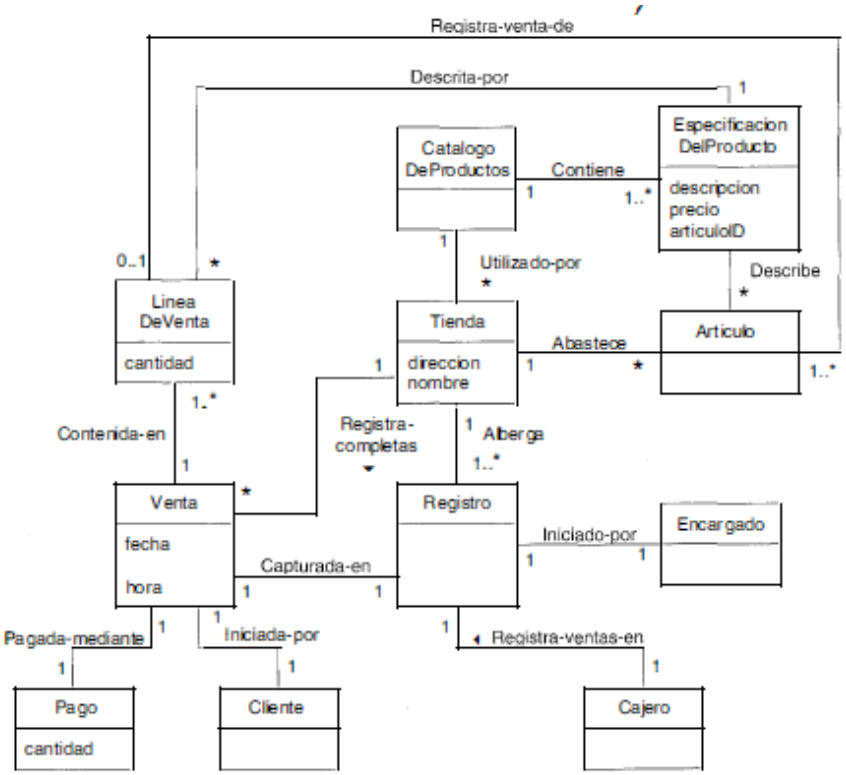


Figura 12.9. Un modelo del dominio parcial.

5. MODELO DE CASOS DE USO: AÑADIR DETALLES CON LOS CONTRATOS DE LAS OPERACIONES

Los contratos de las operaciones pueden ayudar a definir el comportamiento del sistema; describen el resultado de la ejecución de las operaciones en función de los cambios de estado de los objetos del dominio. Por ello, necesitan tener definido antes el Modelo del Dominio.

5.1. Contratos

Los casos de uso son el principal mecanismo del UP para describir el comportamiento del sistema y, normalmente es suficiente. Algunas veces se necesita una descripción más detallada. Los **contratos** describen el comportamiento detallado del sistema en función de los cambios de estado de los objetos del Modelo del Dominio, después de la ejecución de una operación del sistema.

Operaciones del sistema y la interfaz del sistema

Se pueden definir contratos para las **operaciones del sistema**, operaciones que el sistema, como una caja negra, ofrece en su interfaz pública para manejar los eventos del sistema entrantes. Las operaciones del sistema se pueden identificar descubriendo estos eventos del sistema.

El conjunto completo de operaciones del sistema, de todos los casos de uso, define la interfaz pública del sistema, viendo al sistema como un componente o clase individual. En UML, el sistema como un todo se puede representar mediante una clase.

Secciones del contrato

Operación:	Nombre de la operación y parámetros.
Referencias cruzadas:	(opcional) Casos de uso en los que pueden tener lugar esta operación.
Precondiciones:	*Suposiciones* relevantes sobre el estado del sistema o de los objetos del Modelo del Dominio, antes de la ejecución de la operación. No se comprobará en la lógica de esta operación, se asume que son verdad, y son suposiciones no triviales que el lector debe saber que se hicieron.
Postcondiciones:	- El estado de los objetos del Modelo del Dominio después de que se complete la operación. Se discute con detalle en la siguiente sección.

5.2. Postcondiciones

La postcondición describe cambios en el estado de los objetos del Modelo del Dominio. Los cambios comprenden la creación de instancias, formación o rotura de asociaciones y cambio en los atributos. No son acciones que se ejecutarán durante la operación; más bien, son declaraciones sobre los objetos del Modelo del Dominio que son verdad cuando la operación ha terminado.

Las postcondiciones se dividen en estas categorías:

- Creación y eliminación de instancias (más rara esta última).
- Modificación de atributos.
- Formación y rotura de asociaciones (enlaces UML).

La cualidad importante es ser declarativo y enunciar con un estilo orientado al cambio en lugar de orientado a la acción, puesto que las postcondiciones son declaraciones sobre los estados o resultados, en lugar de una descripción de las acciones a ejecutar, o un diseño de una solución. Exprese las postcondiciones en pasado, para resaltar que son declaraciones sobre un cambio de estado en el pasado.

Las postcondiciones se relacionan con el Modelo del Dominio

Estas postcondiciones se expresan en el contexto de los objetos del Modelo del Dominio.

- ¿Qué instancias se pueden crear?: aquellas del Modelo del Dominio.
- ¿Qué asociaciones se pueden formar?: las que se encuentran en el Modelo del Dominio; y así sucesivamente.

Es normal así, durante la creación de los contratos, descubrir la necesidad de registrar nuevas clases conceptuales, atributos o asociaciones en el Modelo del Dominio

Una ventaja de las postcondiciones: detalle analítico

Expresados en un estilo declarativo de cambio de estado, los contratos son una herramienta excelente para el análisis de requisitos que describen los cambios de estado que requiere una operación del sistema sin tener que describir *cómo* se van a llevar a cabo. El diseño del software y la solución se puede diferir, y uno puede centrarse, analíticamente en qué debe suceder. Además, las postcondiciones soportan detalles de grano fino y una declaración más específica de cuál debe ser el resultado de la operación.

Si se utilizan contratos, ¿cómo de completas deben ser las postcondiciones?

Los contratos podrían no ser necesarios. Asumiendo que se desean algunos contratos, no es probable que se genere un conjunto completo y detallado de postcondiciones para una operación del sistema, durante el trabajo de requisitos. Su temprana creación (incluso si es incompleta) es, ciertamente, mejor que diferir su estudio hasta el trabajo de diseño.

Algunos de los detalles finos se descubrirán durante el trabajo de diseño. Esta es una de las ventajas del desarrollo iterativo: los descubrimientos que se generan en una iteración anterior pueden impulsar el estudio y el trabajo de análisis de la siguiente.

5.3. Utilidad de los contratos: contratos vs. casos de uso

Los casos de uso son el principal repositorio de requisitos del proyecto. En este caso, los contratos no son útiles. Sin embargo, hay situaciones en las que los detalles y la complejidad de los cambios de estado requeridos, son difíciles de capturar en los casos de uso.

Estos detalles de grano fino se *pueden* escribir en detalle en el caso de uso asociado a esta operación, pero dará lugar a un caso de uso extremadamente detallado. El formato de la postcondición del contrato ofrece y promueve un lenguaje muy preciso, analítico y exigente que soporta una detallada minuciosidad.

Si, únicamente basándose en los casos de uso y mediante continuas colaboraciones con un experto en la materia de estudio, los desarrolladores pueden entender cómodamente qué hacer, entonces evite la escritura de los contratos. En aquellas situaciones donde la complejidad es alta y añade valor la precisión detallada, los contratos son otra herramienta de requisitos. Creando contratos para todas las operaciones del sistema de cada caso de uso, es una advertencia de que, o bien los casos de uso son algo deficientes, o no hay suficiente y continua colaboración o acceso a los expertos en la materia de estudio.

5.4. Guías: contratos

Para hacer contratos:

1. Identifique las operaciones del sistema a partir de los DSSs.
2. Construya un contrato para las operaciones del sistema complejas y quizás sutiles en sus resultados, o que no están claras en el caso de uso.
3. Para describir la postcondiciones utilice las siguientes categorías:
   - creación y eliminación de instancias
   - modificación de atributos
   - formación y rotura de asociaciones

Consejos acerca de la escritura de contratos

- Establezca las postcondiciones de forma declarativa, con una sentencia pasiva expresada en pasado para destacar que se trata de una declaración de un cambio de estado en lugar del diseño de la manera en la que se va a realizar.
- Recuerde establecer una relación entre los objetos existentes o aquellos creados recientemente mediante la definición de la formación de asociaciones.

El error más habitual en la creación de contratos

El problema más común es olvidarse de incluir la *formación de asociaciones*. En particular, cuando se crean nuevas instancias, es muy probable que se necesiten establecer asociaciones con varios objetos.

5.5. Contratos, operaciones y UML

UML define las **operaciones** formalmente. Una operación es una especificación de una transformación o consulta que se puede invocar para que la ejecute un objeto. Es una abstracción, no una implementación (un **método** es una implementación de una operación).

Una operación UML tiene una **signatura** (nombre y parámetros), y también una **especificación de operación**, que describe los efectos producidos por la ejecución de la operación; esto es, la postcondición. El formato de la especificación es flexible. Los documentos de UML proporcionan como ejemplos el estilo de contratos con pre- y postcondiciones.

Los contratos se pueden aplicar a las operaciones de cualquier nivel de granularidad: las operaciones públicas (o interfaz) de un subsistema, una clase abstracta, etcétera.

Contratos de las operaciones expresados con OCL

Existe un lenguaje formal asociado con UML denominado **Lenguaje de Restricciones de Objetos (OCL)** que se puede utilizar para expresar las restricciones en los modelos. OCL define un formato oficial para la especificación de las pre- y postcondiciones de las operaciones

Contratos en el Diseño por Contrato

La forma de los contratos con pre- y postcondiciones se lleva impulsando en una técnica de diseño denominada **Diseño por Contrato**. En el Diseño por Contrato, también se escriben los contratos para las operaciones de las clases de grano fino, no sólo para las operaciones públicas de los sistemas y subsistemas.

El Diseño por Contrato fomenta la inclusión de una sección *invariante* que define las cosas que no deben cambiar de estado antes y después de que se ejecute una operación.

Soporte de los lenguajes de programación para los contratos

Algunos lenguajes incluyen soporte a nivel de expresiones del lenguaje para los invariantes, las pre- y postcondiciones. Existen pre-procesadores en Jama que proporcionan un soporte parecido.

5.6. Contratos de las operaciones en el UP

Los contratos de especificación de operaciones para el nivel del *Sistema* forman parte del Modelo de Casos de Uso.

Fases

- **Inicio**: Los contratos no se justifican durante la fase de inicio, son demasiado detallados.
- **Elaboración**: Si es que se utilizan, los contratos se escribirán durante la elaboración, cuando se escriben la mayoría de los casos de uso. Solamente para las operaciones del sistema más complejas y sutiles.

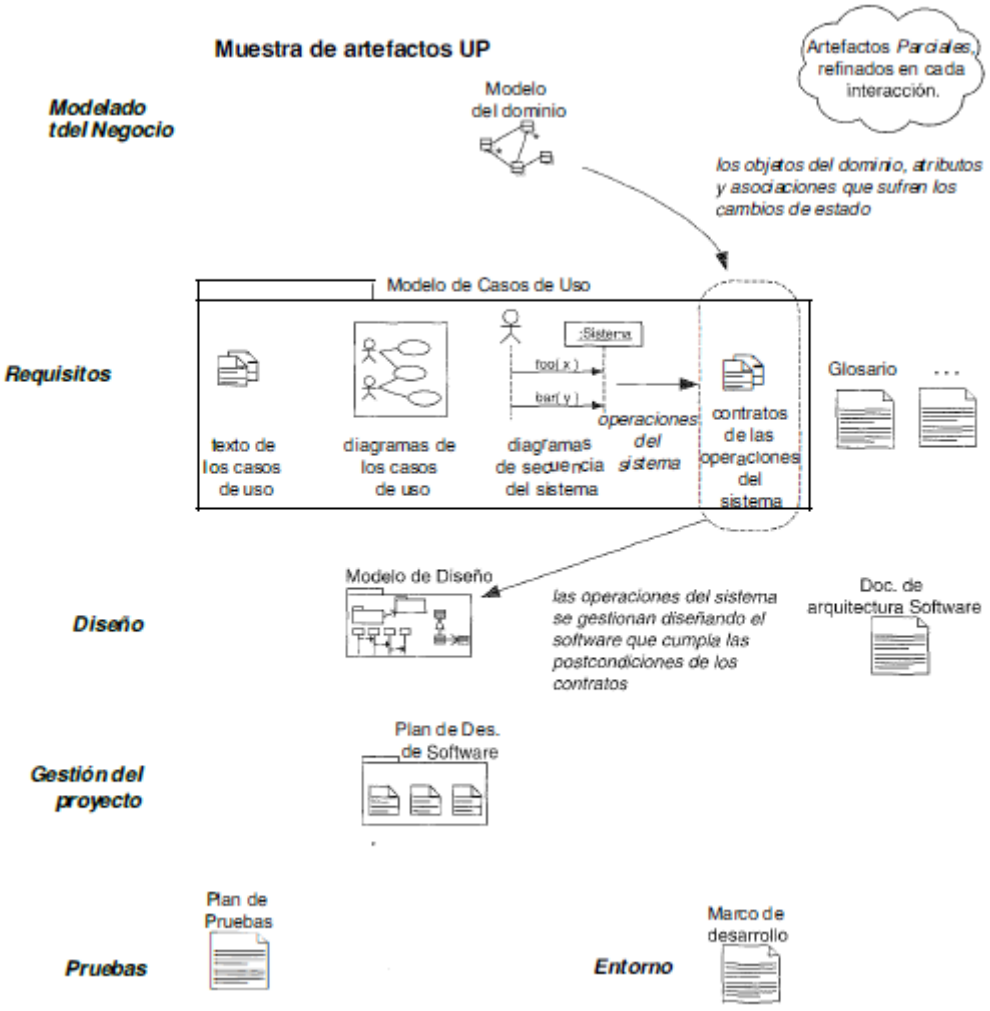


Figura 13.2. Muestra de la influencia entre los artefactos UP.

6. DE LOS REQUISITOS AL DISEÑO EN ESTA ITERACIÓN

6.1. Iterativamente hacer lo correcto y hacerlo correcto

Los requisitos y el análisis orientado a objetos se han centrado en aprender a *hacer lo correcto*, entender algunos de los objetivos importantes y las reglas y restricciones relacionadas. El siguiente trabajo de diseño pondrá de relieve *hacerlo correcto*, diseñar con destreza una solución que satisfaga los requisitos de esta iteración.

En el desarrollo iterativo, en cada iteración tendrá lugar una transición desde un enfoque centrado en los requisitos a un enfoque centrado en el diseño y la implementación. Es normal y saludable descubrir y cambiar algunos requisitos de las *primeras* iteraciones durante el trabajo de diseño e implementación que clarificarán los objetivos del trabajo de diseño de esta iteración, y refinarán la comprensión de los requisitos para las iteraciones futuras. Al final de la elaboración quizás se definan con detalle y de manera fiable el 80% de los requisitos.

El modelado real hasta el momento, de modo realista, será sólo de unos *pocos* días. Eso no significa que sólo hayan pasado unos días desde el comienzo del proyecto. Muchas otras actividades, como programación de pruebas de conceptos,

búsqueda de recursos, planificación, acondicionamiento del entorno, etcétera, podrían consumir unas pocas semanas de preparación.

6.2. Pasar al diseño de objetos

Durante el diseño de objetos, se desarrolla una solución lógica basada en el paradigma orientado a objetos. Lo esencial es la creación de los **diagramas de interacción**, que representan el modo en el que los objetos colaboran para satisfacer los requisitos. Después de la elaboración de los diagramas de interacción, se pueden representar los **diagramas de clases** (del diseño). Éstos resumen la definición de las clases software (e interfaces) que se van a implementar en el software. Estos artefactos forman parte del **Modelo de Diseño**.

Las técnicas del diseño de objetos vs. las técnicas de la notación UML

Los diagramas de interacción son los más importantes, desde el punto de vista del desarrollo de un buen diseño, y requiere el mayor grado de esfuerzo creativo. La creación de los diagramas de interacción requiere la aplicación de los principios para la asignación de **responsabilidades** y el uso de los **principios y patrones de diseño**.

> *Las técnicas del diseño de objetos vs. las técnicas de la notación UML*
>
> La representación de los diagramas de interacción UML es un reflejo de la toma de decisiones sobre el diseño de objetos.
>
> Las técnicas del diseño de objetos son lo que realmente importan, en lugar de saber cómo dibujar los diagramas UML.
>
> El diseño de objetos básico requiere que se tenga conocimiento sobre:
>
> • Los principios de asignación de responsabilidades.
>
> • Patrones de diseño.

7. NOTACIÓN DE LOS DIAGRAMAS DE INTERACCIÓN

El lenguaje utilizado para ilustrar los diseños es los diagramas de interacción. UML incluye los **diagramas de interacción** para ilustrar el modo en el que los objetos interaccionan por medio de mensajes.

7.1. Diagramas de secuencia y colaboración

El *diagrama de interacción* es una generalización de dos tipos de diagramas UML más especializados para representar de forma similar interacciones de mensajes:

- Diagramas de colaboración.
- Diagramas de secuencia.

Los **diagramas de colaboración** ilustran las interacciones entre objetos en un formato de grafo o red, en el cual los objetos se pueden colocar en cualquier lugar del diagrama.

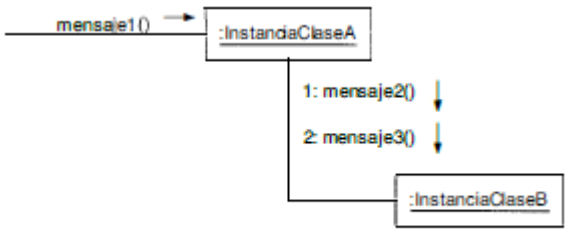


Figura 15.1. Diagrama de colaboración.

Los **diagramas de secuencia** ilustran las interacciones en un tipo de formato con el aspecto de una valla, en el que cada objeto nuevo se añade a la derecha.

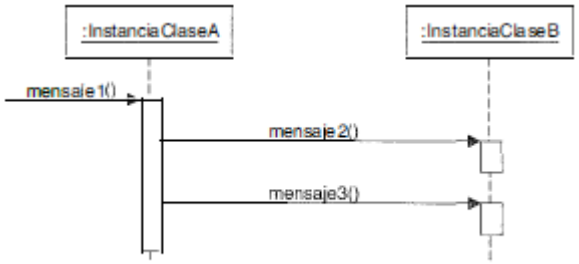


Figura 15.2. Diagrama de secuencia.

Cada tipo tiene puntos fuertes y débiles.

Tipo	*Puntos fuertes*	*Puntos débiles*
secuencia	muestra claramente la secuencia u ordenación en el tiempo de los mensajes notación simple	fuerza a extender por la derecha cuando se añaden nuevos objetos; consume espacio horizontal
colaboración	economiza espacio, flexibilidad al añadir nuevos objetos en dos dimensiones es mejor para ilustrar bifurcaciones complejas, iteraciones y comportamiento concurrente	difícil ver la secuencia de mensajes notación más compleja

7.2. Los diagramas de interacción son importantes

Un problema típico en los proyectos de tecnología de objetos es que no aprecian el valor de llevar a cabo el diseño de objetos mediante el uso de los diagramas de interacción.

Se debería dedicar un tiempo y esfuerzo no trivial a la creación de diagramas de interacción, como reflejo de que se ha estudiado cuidadosamente los detalles del diseño de objetos. El diseño que se representa en los diagramas será imperfecto y especulativo, y se modificará durante la programación, pero proporcionará un punto de partida serio, consistente y común.

Es principalmente durante esta etapa donde se requiere la aplicación de las técnicas de diseño, en términos de patrones, estilos y principios. La creación de los casos de uso, modelos del dominio, y otros artefactos, es más sencilla que la asignación de responsabilidades y la creación de diagramas de interacción bien diseñados. Existen un gran número de principios de diseño sutiles y "grados de libertad" que subyacen a un diagrama de interacción bien diseñado, que a la mayoría de los otros artefactos de A/DOO.

Los principios de diseño para la construcción con éxito de los diagramas de interacción *pueden* codificarse, explicarse y aplicarse de forma sistemática. Se basa en los **patrones** (guías y principios estructurados).

7.3. Notación general de los diagramas de interacción

Representación de clases e interfaces

Para mostrar una instancia de una clase en un diagrama de interacción, se utiliza el símbolo gráfico para una clase (el rectángulo), pero con el nombre subrayado. Los ":" preceden al nombre de la clase.

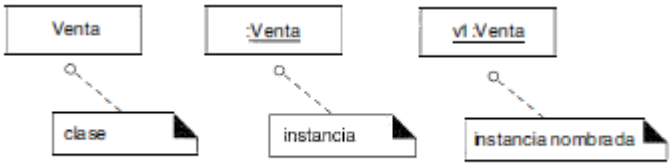


Figura 15.5. Clases e instancias.

Sintaxis de la expresión de mensaje básica

UML cuenta con una sintaxis básica para las expresiones de los mensajes:

```
return := mensaje(parametro: tipoParametro): tipoRetorno
```

Podría excluirse la información del tipo si es obvia o no es importante:

```
espec := getEspecProducto(id)
espec := getEspecProducto(id:ArticulolD)
espec getEspecProducto(id:ArticuloID):EspecificacionDelProducto
```

7.4. Notación básica de los diagramas de colaboración

Enlaces

Un **enlace** es un camino de conexión entre dos objetos; indica que es posible alguna forma de navegación y visibilidad entre los objetos. Es una instancia de una asociación. A lo largo del mismo enlace, pueden fluir múltiples mensajes, y mensajes en ambas direcciones.

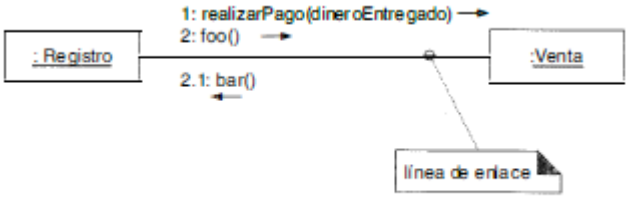


Figura 15.6. Líneas de enlaces.

Mensajes

Cada mensaje entre objetos se representa con una expresión de mensaje y una pequeña flecha que indica la dirección del mensaje. Podrían fluir muchos mensajes a lo largo de este enlace. Se añade un número de secuencia para mostrar el orden secuencial de los mensajes en el hilo de control actual.

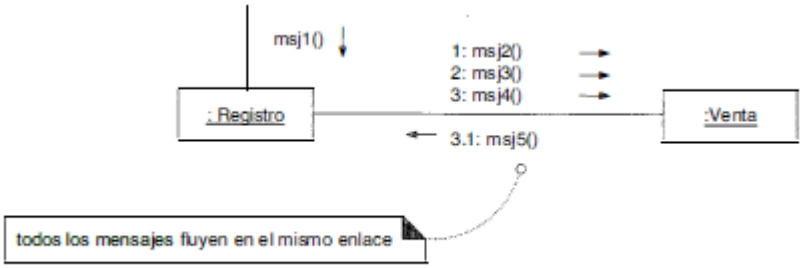


Figura 15.7. Mensajes.

Mensajes a "self" o "this"

Se puede enviar un mensaje desde un objeto a él mismo. Se representa mediante un enlace a él mismo, con mensajes que fluyen a lo largo del enlace.

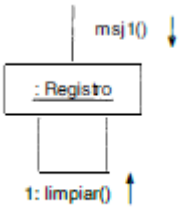


Figura 15.8. Mensajes a "this".

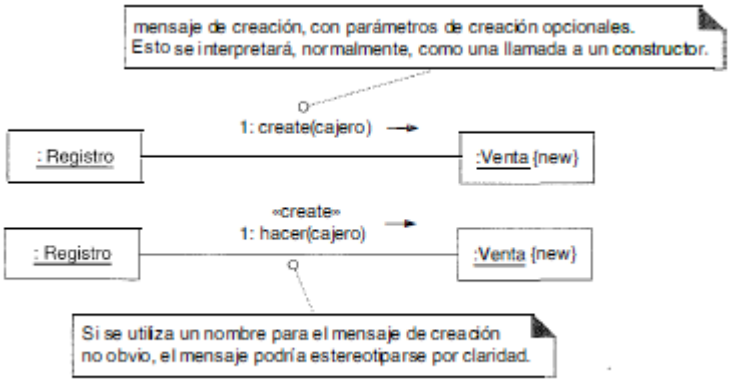


Figura 15.9. Creación de instancias.

Creación de instancias

Cualquier mensaje se puede utilizar para crear una instancia, pero en UML existe el mensaje *créate*, que podría incluir parámetros, que indican el paso de valores iniciales. Esto indica, por ejemplo, la llamada a un constructor con parámetros en Java. Podría añadirse opcionalmente la propiedad UML (*new*) a la caja de la instancia para resaltar la creación.

Secuencia de números de mensaje

El orden de los mensajes se representa mediante **números de secuencia**. El esquema de numeración es:

1. No se numera el primer mensaje.
2. El orden y anidamiento de los siguientes mensajes se muestran con el esquema de numeración válido en el que los mensajes anidados tienen un número adjunto. El anidamiento se denota anteponiendo el número del mensaje entrante al número del mensaje saliente.

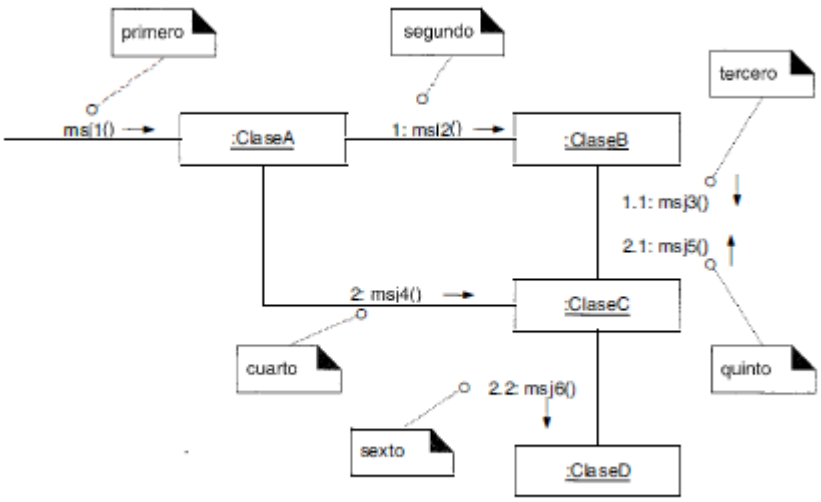


Figura 15.11. Secuencia de numeración compleja.

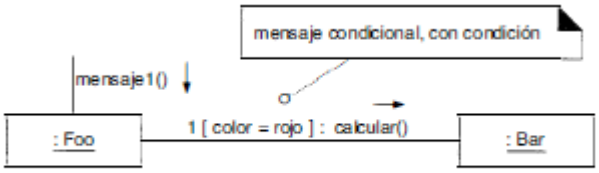


Figura 15.12. Mensaje condicional.

Mensajes condicionales

Un mensaje condicional se muestra con una cláusula condicional, similar a una cláusula de iteración, entre corchetes, a continuación del número de secuencia. El mensaje sólo se envía si la evaluación de la cláusula es *verdad*.

Caminos condicionales mutuamente excluidos

Es necesario modificar las expresiones de la secuencia con una letra de camino condicional (a por convenio). Ambos tienen el número de secuencia 1 puesto que cualquiera de ellos podría ser el primer mensaje interno.

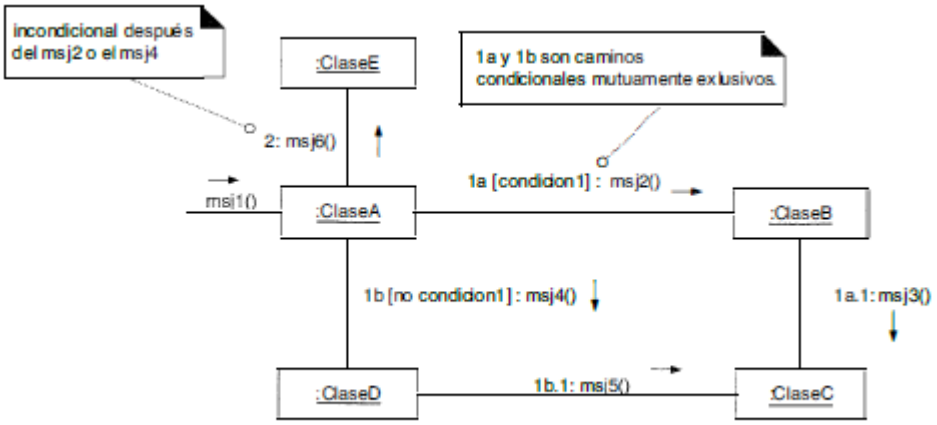


Figura 15.13. Mensajes mutuamente exclusivos.

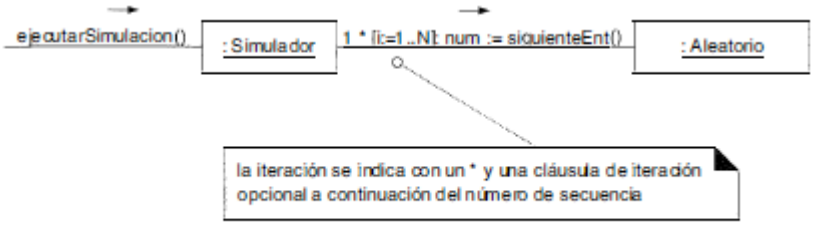


Figura 15.14. Iteración.

Iteración o bucle

Si los detalles de la cláusula de iteración no son importantes para el modelador, se puede utilizar simplemente un "*".

Iteración sobre una colección (multiobjeto)

Un algoritmo típico es iterar sobre todos los miembros de una colección, enviando un mensaje a cada uno de ellos. En UML, el término **multiobjeto** se utiliza para denotar un conjunto de instancias (una colección).

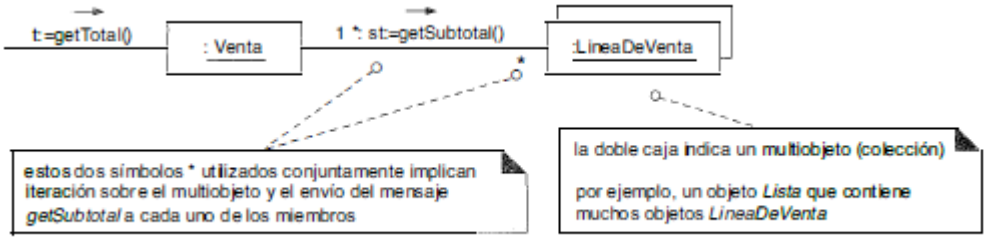


Figura 15.15. Iteración sobre un multiobjeto.

Mensaje a un objeto clase

Los mensajes se podrían enviar a las propias clases, en lugar de una instancia, para invocar a los **métodos estáticos** o de clase. Se muestra un mensaje hacia el rectángulo de una clase cuyo no está subrayado, lo que indica que el mensaje se está enviando a una clase en lugar de a una instancia.

Es importante ser consistente y subrayar los nombres de las instancias cuando lo que se desea es una instancia, de otra manera, se podrían interpretar incorrectamente los mensajes a clases o instancias.

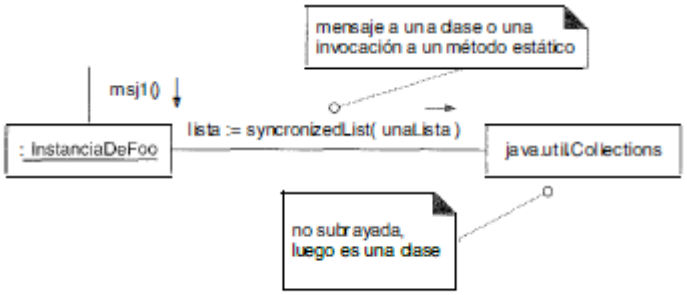


Figura 15.16. Mensaje a un objeto clase (invocación de un método estático).

7.5. Notación básica de los diagramas de secuencia

Enlaces

Los diagramas de secuencia no muestran enlaces.

Mensajes

Cada mensaje entre objetos se representa con una expresión de mensaje sobre una línea con punta de flecha entre los objetos. El orden en el tiempo se organiza de arriba a abajo.

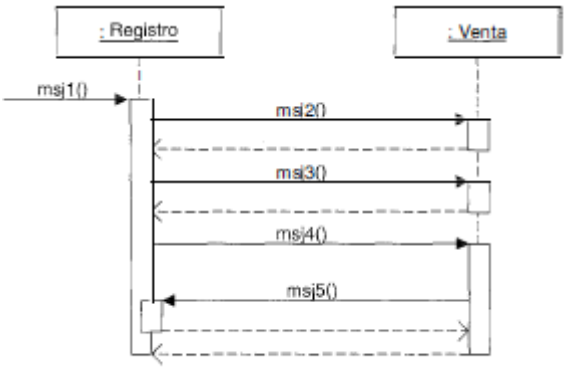


Figura 15.18. Representación de retornos.

Focos de control y cajas de activación

Los diagramas de secuencia podrían también mostrar los focos de control (en una llamada de rutina ordinaria, la operación se encuentra en la pila de llamadas) utilizando una **caja de activación**.

Representación de retornos

Un diagrama de secuencia podría opcionalmente mostrar el retorno de un mensaje mediante una línea punteada con la punta de flecha abierta, al final de una caja de activación (lo normal es que se excluya).

Mensajes a "self" o "this"

Se puede representar un mensaje que se envía de un objeto a él mismo utilizando una caja de activación anidada.

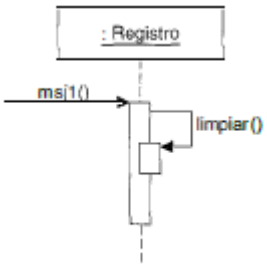


Figura 15.19. Mensajes a "this".

Línea de vida del objeto y creación y destrucción de objetos

Las **líneas de vida de los objetos** (líneas punteadas verticales bajo los objetos) indican la duración de la vida de los objetos en el diagrama. En algunas circunstancias es deseable mostrar la destrucción explícita de un objeto. La notación UML para las líneas de vida proporcionan una forma para expresar esta destrucción.

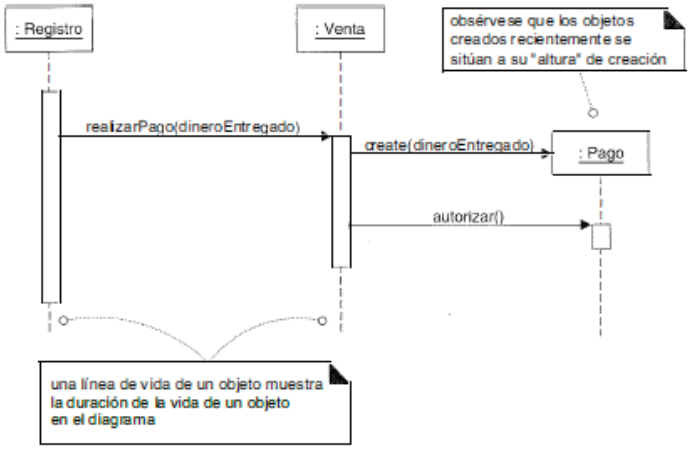


Figura 15.20. Creación de instancias y línea de vida de los objetos.

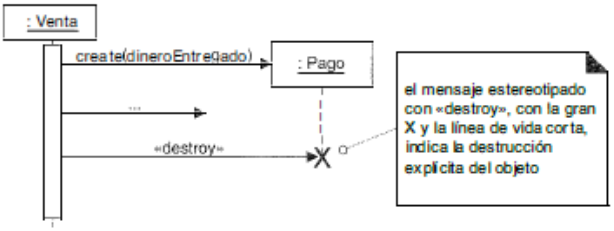


Figura 15.21. Destrucción de objetos.

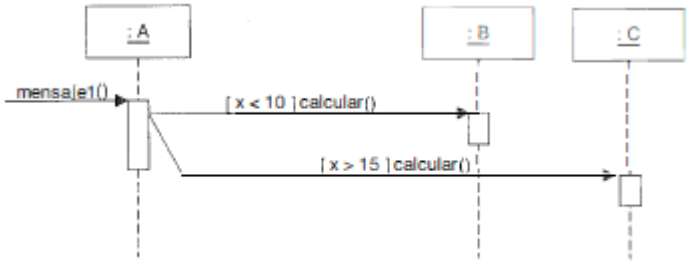


Figura 15.23. Mensajes condicionales mutuamente exclusivos.

Mensajes condicionales y mutuamente exclusivos

Para el segundo caso es un tipo de línea de mensaje con forma de ángulo que nace desde un mismo punto.

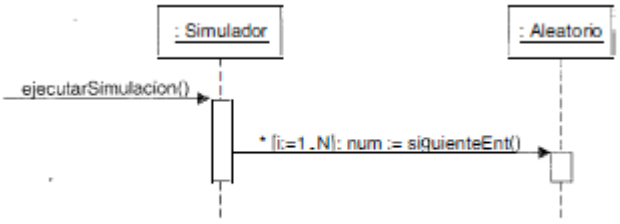


Figura 15.24. Iteración para un mensaje.

Iteración para un único mensaje

Iteración de una serie de mensajes y sobre una colección (multiobjeto)

Con los diagramas de colaboración de UML, se especifica un marcador de multiplicidad "*" al final del rol (al lado del multiobjeto) para indicar el envío de un mensaje a cada elemento, en lugar de repetidamente a la propia colección. UML no especifica cómo hacer esto con los diagramas de secuencia.

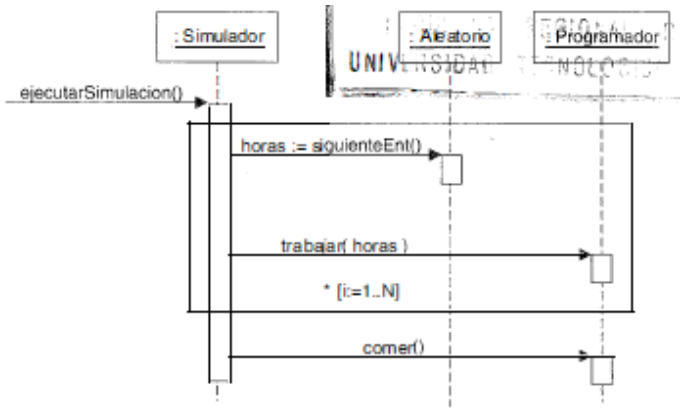


Figura 15.25. Iteración sobre una secuencia de mensajes.

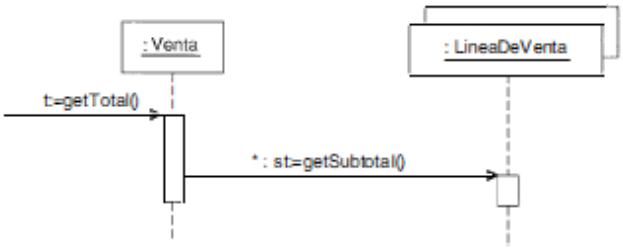


Figura 15.26. Iteración sobre un multiobjeto.

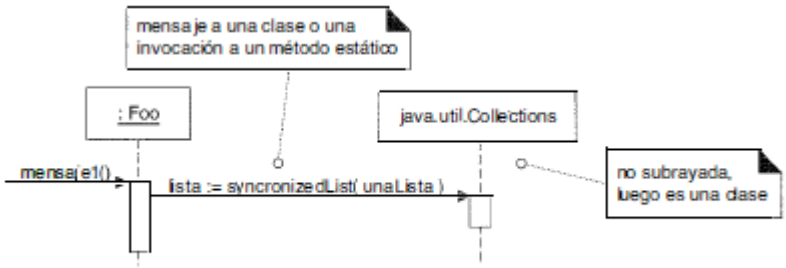


Figura 15.27. Invocación a un método de clase o estático.

Mensajes a objetos de clase

Las llamadas a los métodos de clase o estáticos se representan no subrayando el nombre del clasificador, lo que significa que se trata de una clase en lugar de una instancia.

8. GRASP: DISEÑO DE OBJETOS CON RESPONSABILIDADES

El diseño de objetos se describe como, después de la identificación de sus requisitos y la creación de un modelo del dominio, añadir métodos a las clases del software, y definir el paso de mensajes entre los objetos para satisfacer los requisitos.

Esto no es especialmente útil, porque existen profundos principios y cuestiones involucrados en estas etapas. Esta es una etapa crítica; es la parte esencial del desarrollo de un sistema orientado a objetos.

GRASP como un enfoque sistemático para aprender el diseño de objetos básico

Los patrones GRASP constituyen un apoyo para entender el diseño de objetos esencial, y aplica el razonamiento para el diseño de una forma sistemática, racional y explicable. Se basa en los patrones de *asignación de responsabilidades*.

8.1. Responsabilidades y métodos

UML define una **responsabilidad** como "un contrato u obligación de un clasificador". Las responsabilidades están relacionadas con las obligaciones de un objeto en cuanto a su comportamiento. Son de dos tipos:

- Conocer.
- Hacer.

Entre las responsabilidades de **hacer** de un objeto se encuentran:

- Hacer algo él mismo, como crear un objeto o hacer un cálculo.
- Iniciar una acción en otros objetos.
- Controlar y coordinar actividades en otros objetos.

Entre las responsabilidades de **conocer** de un objeto se encuentran:

- Conocer los datos privados encapsulados.
- Conocer los objetos relacionados.

- Conocer las cosas que puede derivar o calcular.

Las responsabilidades se asignan a las clases de los objetos durante el diseño de objetos. Las responsabilidades relevantes relacionadas con "conocer" a menudo se pueden inferir a partir del modelo del dominio, debido a los atributos y asociaciones que describe.

La granularidad de las responsabilidades influye en la conversión de las responsabilidades a clases y métodos. Una responsabilidad no es lo mismo que un método, pero los métodos se implementan para llevar a cabo responsabilidades. Las responsabilidades se implementan utilizando métodos que actúan solos o colaboran con otros métodos u objetos.

Responsabilidades y los diagramas de interacción

En los artefactos UML, un contexto habitual donde se tiene en cuenta estas responsabilidades (implementadas como métodos) es durante la creación de los diagramas de interacción (que forman parte del Modelo de Diseño del UP).

Los diagramas de interacción muestran elecciones en la asignación de responsabilidades a los objetos. Cuando se crean, se han tomado las decisiones acerca de la asignación de responsabilidades, lo que se refleja en los mensajes que se envían a diferentes clases de objetos.

8.2. Patrones

Un **patrón** es una descripción de un problema y la solución, a la que se da un nombre, y que se puede aplicar a nuevos contextos. Proporciona consejos sobre el modo de aplicarlo en varias circunstancias, y considera los puntos fuertes y compromisos. Muchos patrones proporcionan guías sobre el modo en el que deberían asignarse las responsabilidades a los objetos.

Patrones repetitivos

Lo importante de los patrones no es expresar nuevas ideas de diseño. Es lo contrario, los patrones pretenden codificar conocimiento, estilos y principios *existentes* y que se han probado que son válidos. los patrones GRASP no establecen nuevas ideas, son una codificación de principios básicos ampliamente utilizados.

Los patrones tienen nombres

Todos los patrones, idealmente, tienen nombres sugerentes. Tiene las siguientes ventajas:

- Apoya la identificación e incorporación de ese concepto en nuestro conocimiento y memoria.
- Facilita la comunicación.

Asignar un nombre a una idea compleja como un patrón es un ejemplo del poder de la abstracción, mejora la comunicación y eleva el nivel de la investigación a un grado de abstracción más alto.

8.3. GRASP: Patrones de Principios Generales para Asignar Responsabilidades

- La asignación habilidosa de responsabilidades es extremadamente importante en el diseño de objetos.
- La decisión acerca de la asignación de responsabilidades tiene lugar durante la creación de los diagramas de interacción y con seguridad durante la programación.
- Los patrones son pares problema/solución con un nombre que codifican buenos consejos y principios relacionados con frecuencia con la asignación de responsabilidades.

Cinco patrones GRASP:

- Experto en Información.
- Creador.
- Alta Cohesión.
- Bajo Acoplamiento.
- Controlador.

Se refieren a cuestiones muy básicas, comunes y a aspectos fundamentales del diseño.

8.4. Experto en Información (o Experto)

Solución	Asignar una responsabilidad al experto en información. La clase que tiene la *información* necesaria para realizar la responsabilidad.

Problema

Durante el diseño de objetos, cuando se definen las interacciones entre los objetos, tomamos decisiones sobre la asignación de responsabilidades a las clases software. Si se hace bien, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, y existen más oportunidades para reutilizar componentes en futuras aplicaciones.

Discusión

Este enfoque mantiene un salto en la representación bajo en el que el diseño de los objetos software se corresponde con nuestra concepción sobre cómo se organiza el mundo real.
El contexto en el que se consideraron y optaron por estas responsabilidades fue durante la elaboración de un diagrama de interacción. La sección de métodos de un diagrama de clases puede entonces incluir los métodos.
El Experto en Información se utiliza con frecuencia en la asignación de responsabilidades; es un principio de guía básico que se utiliza continuamente en el diseño de objetos. Expresa la "intuición" común de que los objetos hacen las cosas relacionadas con la información que tienen.
El cumplimiento de la responsabilidad a menudo requiere información que se encuentra dispersa por diferentes clases de objetos. Esto implica que hay muchos expertos en información "parcial" que colaborarán en la tarea. Cada vez que la información se encuentre esparcida por objetos diferentes, necesitarán interactuar mediante el paso de mensajes para compartir el trabajo.
El Experto conduce a diseños donde los objetos del software realizan aquellas operaciones que normalmente se hacen a los objetos inanimados del mundo real que representan.
El patrón Experto en Información tiene una analogía en el mundo real. Otorgamos responsabilidades a los individuos con la información necesaria para llevar a cabo una tarea. Del mismo modo en que los objetos colaboran porque la información se encuentra dispersa, así pasa con las personas.

Contraindicaciones

En algunas ocasiones la solución que sugiere el Experto no es deseable, normalmente debido a problemas de acoplamiento y cohesión. Estos problemas indican la violación de un principio arquitectural básico: diseñe separando los principales aspectos del sistema. Mantenga la lógica de la aplicación en un sitio (objetos software del dominio), mantenga la lógica de la base de datos en otro sitio (subsistema de servicios de persistencia separado), y así sucesivamente, en lugar de entremezclar aspectos del sistema diferentes en el mismo componente.
Separando los aspectos importantes se mejora el acoplamiento y la cohesión del diseño.

Beneficios

- Se mantiene el encapsulamiento de la información, los objetos utilizan su propia información para llevar a cabo las tareas. Conlleva un bajo acoplamiento (sistemas más robustos).
- Se distribuye el comportamiento entre las clases que contienen la información requerida, se estimula las definiciones de clases más cohesivas y "ligeras" que son más fáciles de entender y mantener. Se soporta normalmente una alta cohesión.

Patrones o Principios relacionados

- Bajo Acoplamiento.
- Alta Cohesión

8.5. Creador

Solución

Asignar a la clase B la responsabilidad de crear una instancia de clase A si se cumple uno o más de los casos siguientes:

- B *agrega* objetos de A.
- B *contiene* objetos de A.
- B *registra* instancias de objetos de A.
- B *utiliza más estrechamente* objetos de A.
- B *tiene los datos de inicialización* que se pasarán a un objeto A cuando sea creado.

Si se puede aplicar más de una opción, inclínese por una clase B que *agregue* o *contenga* la clase A.

Problema

La creación de instancias es una de las actividades más comunes en un sistema orientado a objetos. Es útil contar con un principio general para la asignación de las responsabilidades de creación. Si se asignan bien, el diseño puede soportar un bajo acoplamiento, mayor claridad, encapsulación y reutilización.

Discusión

El patrón Creador guía la asignación de responsabilidades relacionadas con la creación de objetos. La intención básica es encontrar un creador que necesite conectarse al objeto creado en alguna situación. Eligiéndolo como el creador se favorece el bajo acoplamiento.
El Agregado *agrega* Partes, el Contenedor *contiene* Contenido, y el Registro *registra* Registros, son todas relaciones comunes entre las clases en un diagrama de clases. El Creador sugiere que la clase contenedor o registro es una buena candidata para asignarle la responsabilidad de crear lo que contiene o registra.
La **agregación** involucra cosas que se encuentran en una relación Todo-Parte o Ensamblaje-Parte.
Algunas veces se encuentra un creador buscando las clases que tienen los datos de inicialización que se pasarán durante la creación. Se trata de un ejemplo del patrón Experto. Los datos de inicialización se pasan durante la creación por medio de algún tipo de método de inicialización, como un constructor Java que tiene parámetros.

Contraindicaciones

La creación requiere una complejidad significativa, como utilizar instancias recicladas por motivos de rendimiento, crear condicionalmente una instancia a partir de una familia de clases similares basado en el valor de alguna propiedad externa, etcétera. Es aconsejable delegar la creación a una clase auxiliar denominada *Factoría* en lugar de utilizar la clase que sugiere el *Creador*.

Beneficios

- Se soporta bajo acoplamiento que implica menos dependencias de mantenimiento y mayores oportunidades para reutilizar. No se incrementa el acoplamiento porque la clase *creada* es presumible que ya sea visible a la clase *creadora* debido a las asociaciones existentes que motivaron su elección como creador.

Patrones o Principios relacionados

- Bajo Acoplamiento.
- Factoría.

8.6. Bajo Acoplamiento

Solución

Asignar una responsabilidad de manera que el acoplamiento permanezca bajo.

Problema

El **acoplamiento** es una medida de la fuerza con que un elemento está conectado a, tiene conocimiento de, confía en, otros elementos. Un elemento con bajo (o débil) acoplamiento no depende de demasiados otros elementos (pueden ser clases, subsistemas, sistemas, etcétera).
Una clase con alto (o fuerte) acoplamiento confía en muchas otras clases. Tales clases podrían no ser deseables:

- Los cambios en las clases relacionadas fuerzan cambios locales.
- Son difíciles de entender de manera aislada.
- Son difíciles de reutilizar puesto que su uso requiere la presencia adicional de las clases de las que depende.

El nivel de acoplamiento no se puede considerar de manera aislada a otros principios como el Experto o Alta Cohesión. Sin embargo, es un factor a tener en cuenta para mejorar un diseño.

Discusión

El patrón de Bajo Acoplamiento es un principio a tener en mente en todas las decisiones de diseño; es un objetivo subyacente a tener en cuenta continuamente. Es un **principio evaluativo** que aplica un diseñador mientras evalúa todas las decisiones de diseño.
En los lenguajes orientados a objetos como C++, Java y C#, algunas de las formas comunes de acoplamiento entre el *TipoX* y el *TipoY* son:

- El *TipoX* tiene un atributo que hace referencia a una instancia de *TipoY*, o al propio *TipoY*.
- Un objeto de *TipoX* invoca los servicios de un objeto de *TipoY*.
- El *TipoX* tiene un método que referencia a una instancia de *TipoY*, o al propio *TipoY*, de algún modo.
- El *TipoX* es una subclase, directa o indirecta, del *TipoY*.
- El *TipoY* es una interfaz y el *TipoX* implementa esa interfaz.

El patrón Bajo Acoplamiento impulsa la asignación de responsabilidades de manera que su localización no incremente el acoplamiento hasta un nivel que lleve a los resultados negativos que puede producir un acoplamiento alto.
Soporta el diseño de clases que son más independientes, lo que reduce el impacto del cambio. No se puede considerar de manera aislada a otros patrones como el Experto o el de Alta Cohesión, sino que necesita incluirse como uno de los diferentes principios de diseño que influyen en una elección al asignar una responsabilidad.

No existe una medida absoluta de cuando el acoplamiento es demasiado alto. Lo que es importante es que un desarrollador pueda medir el gradó de acoplamiento actual, y evaluar si aumentarlo le causará problemas. En general, las clases que son inherentemente muy genéricas por naturaleza, y con una probabilidad de reutilización alta, debería tener un acoplamiento especialmente bajo.
Si el Bajo Acoplamiento se lleva al extremo, producirá un diseño pobre porque dará lugar a objetos inconexos, saturados, y con actividad compleja que hacen todo el trabajo, con muchos objetos muy pasivos.

Beneficios

- No afectan los cambios en otros componentes.
- Fácil de entender de manera aislada.
- Conveniente para reutilizar.

Patrones Relacionados

- Variaciones Protegidas.

8.7. Alta Cohesión

Solución

Asignar una responsabilidad de manera que la cohesión permanezca alta.

Problema

La **cohesión** es una medida de la fuerza con la que se relacionan y del grado de focalización de las responsabilidades de un elemento. Un elemento con responsabilidades altamente relacionadas, y que no hace una gran cantidad de trabajo, tiene alta cohesión. Estos elementos pueden ser clases, subsistemas, etcétera.
Una clase con baja cohesión hace muchas cosas no relacionadas, o demasiado trabajo. Tales clases no son convenientes:

- Difíciles de entender.
- Difíciles de reutilizar.
- Difíciles de mantener.
- Delicadas, constantemente afectadas por los cambios.

Las clases con baja cohesión representan un “grano grande” o se les ha asignado responsabilidades que deberían haberse delegado en otros objetos.
El nivel de cohesión no se puede considerar de manera aislada a otras responsabilidades y otros principios como los patrones Experto y Bajo Acoplamiento.

Discusión

El patrón de Alta Cohesión es un principio a tener en mente durante todas las decisiones de diseño; es un objetivo subyacente a tener en cuenta continuamente. Es un principio evaluativo que aplica un diseñador mientras evalúa todas las decisiones de diseño.
Existe alta cohesión funcional cuando los elementos de un componente (como una clase) “trabajan todos juntos para proporcionar algún comportamiento bien delimitado”. Algunos escenarios que ilustran diferentes grados de cohesión funcional son:

1. *Muy baja cohesión*. Una única clase es responsable de muchas cosas en áreas funcionales muy diferentes.
2. *Baja cohesión*. Una única clase tiene la responsabilidad de una tarea compleja en un área funcional.
3. *Alta cohesión*. Una clase tiene una responsabilidad moderada en un área funcional y colabora con otras clases para llevar a cabo las tareas.
4. *Moderada cohesión*. Una clase tiene responsabilidades ligeras y únicas en unas pocas áreas diferentes que están lógicamente relacionadas con el concepto de la clase, pero no entre ellas.

Una clase con alta cohesión tiene un número relativamente pequeño de métodos, con funcionalidad altamente relacionada, y no realiza mucho trabajo. Colabora con otros objetos para compartir el esfuerzo si la tarea es extensa. Una clase con alta cohesión es ventajosa porque es relativamente fácil de mantener.
El patrón de Alta Cohesión tiene una analogía en el mundo real. Si una persona tiene demasiadas responsabilidades no relacionadas, no es efectiva.
Otro principio clásico: diseño modular
Otro principio fuertemente relacionado con el acoplamiento y la cohesión es promover el **diseño modular**, la propiedad de un sistema que se ha descompuesto en un conjunto de módulos cohesivos y débilmente acoplados.

La modularidad se alcanza diseñando cada método con un único y claro objetivo, y agrupando un conjunto de aspectos relacionados en una clase.

Cohesión y acoplamiento; el yin y el yang

Una mala cohesión causa, normalmente, un mal acoplamiento, y viceversa.

Contraindicaciones

Existen pocos casos en los que esté justificada la aceptación de baja cohesión.

Un caso es la agrupación de responsabilidades o código en una clase o componente para simplificar el mantenimiento por una persona.

Otro caso lo constituyen los objetos servidores distribuidos. Debido a implicaciones de costes fijos y rendimientos asociados, a veces, es deseable crear menos objetos servidores, de mayor tamaño y menos cohesivos que proporcionen una interfaz para muchas operaciones. Esto lleva al patrón **Interfaz Remota de Grano Grueso** donde las operaciones remotas se hacen de grano más grueso con objeto de realizar o solicitar más trabajo en las llamadas a operaciones remotas, debido a las penalizaciones de rendimiento de las llamadas remotas en la red.

Beneficios

- Se incrementa la claridad y facilita la comprensión del diseño.
- Se simplifican el mantenimiento y las mejoras.
- Se soporta a menudo bajo acoplamiento.
- El grano fino de funcionalidad altamente relacionada incrementa la reutilización porque una clase cohesiva se puede utilizar para un propósito muy específico.

8.8. Controlador

Solución

Asignar la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las siguientes opciones:

- Representa el sistema global, dispositivo o subsistema (*controlador de fachada*).
- Representa un escenario de caso de uso en el que tiene lugar el evento del sistema, <NombreDelCasoDeUso>Manejador, <NombreDelCasoDeUso>Coordinador o <NombreDelCasoDeUso>Sesion (*controlador de sesión o de caso de uso*).
    - Utilice la misma clase controlador para todos los eventos del sistema en el mismo escenario de caso de uso.
    - Una sesión es una instancia de una conversación con un actor. Las sesiones pueden tener cualquier duración, pero se organizan a menudo en función de los casos de uso (sesiones de casos de uso).

Problema

Un **evento del sistema** de entrada es un evento generado por un actor externo. Se asocian con **operaciones del sistema**, tal como se relacionan los mensajes y los métodos.

Un **Controlador** es un objeto que no pertenece a la interfaz de usuario, responsable de recibir o manejar un evento del sistema. Un Controlador define el método para la operación del sistema.

Discusión

Los sistemas reciben eventos de entrada externos, normalmente a través de una GUI manejada por una persona. Otros medios pueden ser mensajes externos o señales desde sensores.

Si se utiliza un diseño de objetos, se debe escoger algún manejador para estos eventos. El patrón Controlador proporciona guías acerca de las opciones aceptadas y adecuadas. El controlador es una especie de fachada en la capa del dominio para la capa de la interfaz.

A menudo, es conveniente utilizar la misma clase controlador para todos los eventos del sistema de un caso de uso de manera que es posible mantener la información acerca del estado del caso de uso en el controlador. Podrían utilizarse diferentes controladores para casos de usos distintos.

Un error típico del diseño de los controladores es otorgarles demasiada responsabilidad.

Un controlador debería *delegar* en otros objetos el trabajo que se necesita hacer; coordina o controla la actividad. No realiza mucho trabajo por sí mismo.

El **controlador de fachada** representa al sistema global, dispositivo o subsistema. La idea es elegir algún nombre de clase que sugiera una cubierta, o fachada, sobre las otras capas de la aplicación, y que proporciona las llamadas a los servicios más importantes desde la capa de UI hacia las otras capas.

Son adecuados cuando no existen “demasiados” eventos del sistema, o no es posible que la interfaz de usuario (UI) redireccione mensajes de los eventos del sistema a controladores alternativos.

En un **controlador de casos de uso** hay un controlador diferente para cada caso de uso. No es un objeto del dominio; es una construcción artificial para dar soporte al sistema. Es una alternativa cuando la asignación de las responsabilidades a un controlador de fachada conduce a diseños con baja cohesión o alto acoplamiento. Es una buena elección cuando hay muchos eventos del sistema repartidos en diferentes procesos; el controlador factoriza la gestión en clases separadas manejables, y proporciona una base para conocer y razonar sobre el estado de los escenarios actuales en marcha.

En el UP existen los conceptos de clases frontera, control y entidad. Los **objetos frontera** son abstracciones de las interfaces, los **objetos entidad** son los objetos software del dominio independiente de la aplicación y los **objetos control** son los manejadores de los casos de uso tal. Los objetos interfaz y la capa de presentación no deberían ser responsables de llevar a cabo los eventos del sistema. Las operaciones del sistema se deberían manejar en la lógica de la aplicación o capas del dominio en lugar de en la capa de interfaz del sistema.

El objeto Controlador es normalmente un objeto del lado del cliente en el mismo proceso que la UI, entonces no es exactamente aplicable cuando el UI es un cliente Web en un navegador, y hay software del lado del servidor involucrado. Existen patrones comunes para manejar los eventos del sistema que están fuertemente influenciados por el marco tecnológico escogido en el lado del servidor.
Si la UI no es un cliente web, pero la aplicación invoca servicios remotos, es todavía común el uso del patrón Controlador. La UI reenvía la solicitud al Controlador local del lado del cliente, y el Controlador podría reenviar toda o parte de la gestión de la petición a los servicios remotos.

El Controlador recibe la solicitud del servicio desde la capa de UI y coordina su realización, delegando a otros objetos.

Beneficios

- *Aumenta el potencial para reutilizar y las interfaces conectables (pluggable)*. La lógica de la aplicación *no* se maneja en la capa de interfaz.
- *Razonamiento sobre el estado de los casos de uso*. Asegurar que las operaciones del sistema tienen lugar en una secuencia válida, o sobre el estado actual de la actividad y operaciones del caso de uso que está en marcha.

Cuestiones y Soluciones

Controladores saturados

Un controlador saturado es una clase controlador pobremente diseñada que tendrá baja cohesión.

- Existe una *única* clase controlador que recibe *todos* los eventos del sistema en el sistema, y hay muchos (controlador de fachada).
- El propio controlador realiza muchas de las tareas para llevar a cabo los eventos del sistema, sin delegar trabajo.
- Un controlador tiene muchos atributos y mantiene información significativa sobre el sistema o el dominio

Remedios para un controlador saturado:

1. Añadir más controladores: un sistema no tiene que tener sólo uno. En lugar de un controlador de fachada, utilice controladores de caso de uso.
2. Diseñe el controlador que delegue el cumplimiento de cada responsabilidad de una operación del sistema de otros objetos.

La capa de interfaz no maneja eventos del sistema

Los objetos interfaz y la capa de interfaz, no deberían ser responsables de manejar los eventos del sistema.
Asignando la responsabilidad de las operaciones del sistema a objetos en la capa de aplicación o del dominio en lugar de en la capa de interfaz, incrementa el potencial para reutilizar. Si un objeto de la capa de interfaz maneja una operación del sistema, entonces la lógica del proceso del negocio estaría contenida en un objeto interfaz, por lo que la oportunidad para reutilizar es baja debido a su acoplamiento con una interfaz particular y con la aplicación.
También facilita la desconexión de la capa de interfaz y la utilización de un framework o tecnología de interfaz diferente o ejecutar el sistema en un modo "por lotes" sin conexión.

Patrones Relacionados

- ***Command***: En un sistema de manejo de mensajes, cada mensaje podría representarse y manejarse mediante un objeto *Command* separado.
- **Fachada**: Un controlador de fachada es un tipo de Fachada (*Facade*).
- **Capas**: La ubicación de la lógica del dominio en la capa del dominio en lugar de en la capa de presentación forma parte del patrón de Capas (*Layers*).

- **Fabricación Pura**: Es una creación arbitraria del diseñador, no una clase software cuyo nombre se inspira en el Modelo del Dominio. Un controlador de caso de uso es un tipo de Fabricación Pura.

8.9. Diseño de objetos y tarjetas CRC

Otro mecanismo que se utiliza para ayudar a asignar responsabilidades e indicar las colaboraciones con otros objetos son las **tarjetas CRC** (tarjetas Clase-Responsabilidad-Colaborador). Son fichas, una por cada clase, en las que se escriben brevemente las responsabilidades de la clase, y una lista de los objetos con los que colabora para llevar a cabo esas responsabilidades.

Los patrones GRASP se podrían aplicar cuando se tiene en cuenta el diseño mientras se utilizan las tarjetas CRC. Las tarjetas CRC son una técnica para registrar los resultados de la asignación de responsabilidades y asignaciones. La información recopilada se puede enriquecer utilizando diagramas de clases y de interacción.

9. MODELO DE DISEÑO: REALIZACIÓN DE CASOS DE USO CON LOS PATRONES GRASP

La asignación de responsabilidades y el diseño de colaboraciones son etapas muy importantes y creativas durante el diseño, mientras se elaboran los diagramas o mientras se programa.

9.1. Realizaciones de casos de uso

Una realización de caso de uso describe cómo se realiza un caso de uso particular en el modelo de diseño, en función de los objetos que colaboran. Un diseñador puede describir el diseño de uno o más *escenarios* de un caso de uso; cada uno de estos se denomina una **realización del caso de uso**, que recuerdan la conexión entre los requisitos expresados como casos de uso, y el diseño de objetos que satisface los requisitos.

Los diagramas de interacción UML son un lenguaje común para ilustrar las realizaciones de los casos de uso. Las relaciones entre algunos artefactos del UP quedan:

- El caso de uso sugiere los eventos del sistema que se muestran explícitamente en los diagramas de secuencia del sistema.
- Describen los detalles de los efectos de los eventos del sistema (cambios de los objetos del dominio) en los contratos de las operaciones del sistema.
- Los eventos del sistema representan los mensajes que inician los diagramas de interacción (representan el modo en el que los objetos interactúan para llevar a cabo las tareas requeridas, la realización del caso de uso).
- Los diagramas de interacción comprenden la interacción de mensajes entre objetos software.

9.2. Comentarios sobre los artefactos

Diagramas de interacción y realizaciones de casos de uso

Si se utilizan los diagramas de interacción para representar las realizaciones de los casos de uso, se necesitará un diagrama de colaboración diferente para mostrar el manejo de cada mensaje de evento del sistema. Si se utilizan los diagramas de secuencia, podría ser posible encajar todos los mensajes de eventos del sistema en el mismo diagrama.

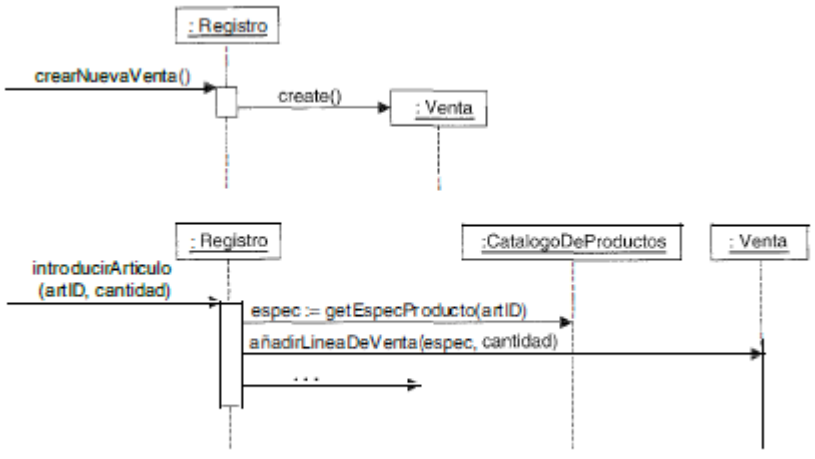


Figura 17.3. Múltiples diagramas de secuencia y manejo de los mensajes de eventos del sistema.

Ocurre a menudo que el diagrama de secuencia es entonces demasiado complejo o largo. Es común, como con los diagramas de interacción, utilizar un diagrama de secuencia para cada mensaje de evento del sistema.

Así también, para cada contrato, se expresan en el cambio de estado en las postcondiciones y se diseñan las interacciones para satisfacer los requisitos.

Los requisitos no son perfectos

Los casos de uso que se han escrito previamente y los contratos dan sólo una idea aproximada de lo que se tiene que conseguir y puede que los requisitos no sean perfectos, o han cambiado. Se necesita involucrar continuamente a los clientes y a los expertos en la materia en estudio para revisar y proporcionar retroalimentación sobre el comportamiento del sistema que se está desarrollando.

Una ventaja del desarrollo iterativo es que favorece de manera natural el descubrimiento de nuevos resultados del análisis y diseño durante el trabajo de diseño e implementación.

El Modelo del Dominio y las realizaciones de los casos de uso

Algunos de los objetos software que interactúan mediante el paso de mensajes en los diagramas de interacción se inspiran en el Modelo del Dominio. La elección de la asignación adecuada de la responsabilidad utilizando los patrones GRASP depende de la información del Modelo del Dominio.

Clases conceptuales vs. clases del diseño

El Modelo del Dominio del UP no representa clases software, pero podría utilizarse para inspirar la presencia y los nombres de algunas clases software en el Modelo de Diseño. Durante la elaboración de los diagramas de interacción, los desarrolladores podrían mirar en el Modelo del Dominio para asignar los nombres a algunas clases del diseño y se crea un diseño con un salto en la representación más bajo entre el diseño del software y nuestra percepción del dominio del mundo real con el que el software está relacionado.

Durante este trabajo de diseño es conveniente descubrir nuevas clases conceptuales que se obviaron durante el análisis de dominio inicial, y también crear clases software cuyos nombres y objetivos no estén relacionados en absoluto con el Modelo del Dominio.

9.3. Diseño de objetos

Mostrar por pantalla

Debido a un principio de diseño denominado **Separación Modelo-Vista**, los objetos que no pertenecen a la GUI no son responsables de involucrarse en las tareas de salida. Por tanto, aunque el caso de uso establezca que se muestran por pantalla, el diseño lo ignorará en este momento.

Visibilidad

La **visibilidad** es la capacidad de un objeto de "ver" o tener una referencia a otro objeto. Para que un objeto envíe un mensaje a otro objeto éste tiene que ser visible a aquél.

Mensaje a los multiobjetos

La interpretación en UML del envío de un mensaje a un multiobjeto es que se trata de un mensaje a la propia colección de objetos, en vez de una transmisión implícita a los miembros de la misma. Esto es especialmente obvio para las operaciones de colección genéricas como *buscar* y *añadir*.

- El mensaje *buscar* enviado es un mensaje que se envía una vez a la estructura de datos de la colección representada por el multiobjeto.
    - El mensaje genérico e independiente del lenguaje *buscar* se traducirá, durante la programación, para un lenguaje específico y librería.
- El mensaje *añadir* enviado al multiobjeto es para añadir un elemento a la estructura de datos de la colección representada por un multiobjeto.

Notación UML para mostrar las restricciones, notas y algoritmos

Algunas veces en UML deseamos utilizar texto para describir el algoritmo de un método, o especificar alguna restricción. UML proporciona tanto restricciones como notas. Una **restricción UML** es una información semánticamente significativa que se anexa a un elemento del modelo. Son texto encerrado entre llaves {}. Se puede utilizar cualquier lenguaje formal o informal para las restricciones, y UML incluye especialmente OCL si uno desea utilizarlo.

Una **nota de UML** es un comentario que no tiene impacto semántico, como la fecha o el autor. Siempre se muestra en un (un cuadro de texto con la esquina doblada). Las restricciones largas también podrían colocarse en un "cuadro de nota", en cuyo caso el presunto cuadro de nota realmente contiene una restricción en lugar de una nota. El texto del cuadro va entre llaves para indicar que es una restricción.

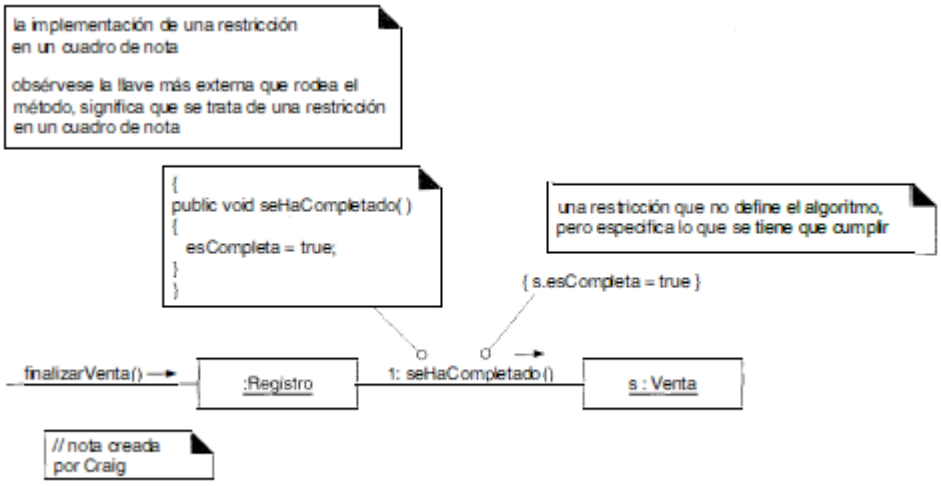


Figura 17.10. Restricciones y notas.

Diseño

No todos los diagramas de interacción comienzan con un mensaje de evento del sistema; pueden comenzar con cualquier mensaje para el que el diseñador desee mostrar las interacciones.

Cuando hay elecciones de diseño alternativas, las implicaciones en cuanto a la cohesión y el acoplamiento de las alternativas, y posiblemente las futuras presiones de evolución de las alternativas. Elija una alternativa con buena cohesión, acoplamiento y estabilidad ante posibles cambios futuros.

Considere algunas de las implicaciones de estas elecciones en función de los patrones GRASP Alta Cohesión y Bajo Acoplamiento.

9.4. Diseño de objetos: ponerEnMarcha

¿Cuándo crear el diseño de ponerEnMarcha?

La mayoría de los sistemas tienen el caso de uso *ponerEnMarcha*, y alguna operación del sistema inicial relacionada con el comienzo de la aplicación. Aunque *ponerEnMarcha* es la primera que se va a ejecutar, posponga el desarrollo de un diagrama de interacción para ella hasta que se hayan tenido en cuenta todas las otras operaciones del sistema.

Cómo comienzan las aplicaciones

La operación *ponerEnMarcha* representa de manera abstracta la fase de inicialización de la ejecución cuando se lanza una aplicación. Un estilo de diseño común es crear en último término un **objeto del dominio inicial**, que es el primer objeto software del "dominio" que se crea. Las aplicaciones se organizan en capas lógicas que separan los aspectos más importantes de la aplicación. Esto comprende la capa de UI y una capa del "dominio". La capa del dominio del Modelo de Diseño está formada por las clases software y que contienen la lógica de la aplicación.

El objeto de dominio inicial, una vez creado, es el responsable de la creación de los objetos del dominio. El lugar donde se crea este objeto del dominio inicial depende de la tecnología de objetos escogida. En una aplicación Java, podría crearlo el método *main*, o delegar el trabajo al objeto *factoría* que lo crea.

```
public class Main
{
public static void main ( String [ ] args )
{
    // La Tienda es el o b j et o d e l dominio inicial.
    // La Tienda crea algún otro objeto del dominio.
    Tienda tienda = new Tienda();
    Registro registro = tienda.getRegistro();
    JFrameProcesarVenta frame = new JFrameProcesarVenta(registro);
...
}
```

}

Interpretación de la operación del sistema ponerEnMarcha

La operación del sistema *ponerEnMarcha* es una abstracción independiente del lenguaje. Durante el diseño, existen variaciones en cuanto al lugar de creación del objeto inicial, y si controla o no el proceso. El objeto del dominio inicial no suele tomar el control si se trata de una GUI; en otro caso, lo hace con frecuencia.

Los diagramas de operación para la operación *ponerEnMarcha* representan lo que ocurre cuando se crea el objeto inicial del dominio del problema, y opcionalmente lo que sucede si toma el control. No incluyen ninguna actividad anterior o siguiente de los objetos en la capa de GUI, si existe alguno.

El objeto del dominio inicial *no* es responsable de controlar el proceso; el control permanecerá en la capa de UI después de que se cree el objeto del dominio inicial. El diagrama de interacción para la operación *ponerEnMarcha* podría reinterpretarse únicamente como el envío del mensaje *create()* para crear el objeto inicial.

Elección del objeto del dominio inicial

El objeto del dominio inicial es la clase de la raíz de la jerarquía de agregación o contención, o cercana a ella. Esto podría ser un controlador de fachada o algún otro objeto que se considera que contiene todos o la mayoría de los objetos. Las consideraciones de Alta Cohesión y Bajo Acoplamiento podrían influir en la elección entre las alternativas.

Objetos persistentes

Un medio de almacenamiento persistente, como una base de datos relacional u objetual, durante la operación *ponerEnMarcha*, si sólo hay unos pocos de estos objetos, se podrían cargar todos en la memoria principal del ordenador. Si hay muchos, cargarlos todos consumiría demasiada memoria o tiempo. Alternativamente se cargarán en memoria bajo demanda las instancias individuales cuando se requieran.

El diseño de la manera de cargar dinámicamente bajo demanda es sencillo si se utiliza una base de datos objetual, pero difícil para una base de datos relacional.

Diseño de *create()*

Las tareas de creación e inicialización se derivan a partir de las necesidades del trabajo de diseño anterior. Este enfoque de creación de las especificaciones es temporal. En el diseño final, se materializará desde una base de datos, cuando sea necesario.

La multiplicidad entre las clases de objetos del Modelo del Dominio y el Modelo del Diseño podría no ser la misma.

9.5. Conexión de la capa de UI con la capa del dominio

Las aplicaciones se organizan en capas lógicas que separan los aspectos más importantes de la aplicación, como la capa de UI (para las cuestiones de la UI) y una capa de "dominio" (para las cuestiones de la lógica del dominio). Entre los diseños típicos según los cuales los objetos de la capa del dominio son visibles a los objetos de la capa de la UI encontramos:

- Una rutina de inicialización (un método *main* en Java) crea tanto un objeto de la UI como un objeto del dominio, y pasa el objeto del dominio a la UI.
- Un objeto de la UI recupera el objeto del dominio de una fuente bien conocida, como un objeto factoría que es responsable de la creación de los objetos del dominio.

Una vez que el objeto de la UI está conectado, el controlador puede reenviarle mensajes de eventos del sistema.

Responsabilidades de la capa del dominio y de interfaz

La capa de UI no debería tener ninguna responsabilidad de la lógica del dominio. Sólo debería ser responsable de las tareas de la interfaz de usuario, como actualizar los elementos gráficos.
La capa de UI debería remitir las solicitudes de las tareas orientadas al dominio a la capa del dominio, que es la responsable de manejarlas.

9.6. Realizaciones de casos de uso en el UP

Las realizaciones de los casos de uso forman parte del Modelo de Diseño del UP.

Tabla 17.1. Muestra de los artefactos UP y evolución temporal. c – comenzar; r – refinar.

Disciplina	*Artefacto* *Iteración →*	*Inicio* *I1*	*Elab.* *E1…En*	*Const.* *C1…Cn*	*Trans.* *T1…T2*
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	Modelo de Casos de Uso (DSS) Visión Especificación complementaria Glosario	c c c c	r r r r		
Diseño	***Modelo de Diseño*** Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

Fases

- **Inicio**: El Modelo de Diseño y las realizaciones de los casos de uso normalmente no comenzarán hasta la elaboración (comprende decisiones de diseño detalladas).
- **Elaboración**: Podrían crearse las realizaciones de los casos de uso para los escenarios más significativos desde el punto de vista de la arquitectura o de más riesgo del diseño. No se harán los diagramas de UML para cada escenario. La idea es realizar los diagramas de interacción para las realizaciones de los casos de uso claves que se benefician de algún estudio anticipado.
- **Construcción**: Se crean las realizaciones de los casos de uso para el resto de problemas de diseño.

En el UP, el trabajo de la realización de los casos de uso es una actividad de diseño

10. MODELO DE DISEÑO: DETERMINACIÓN DE LA VISIBILIDAD

La **visibilidad** es la capacidad de un objeto de ver o tener una referencia a otro.

10.1. Visibilidad entre objetos

Para que un objeto emisor envíe un mensaje a un objeto receptor, el receptor debe ser visible al emisor, éste debe tener algún tipo de referencia o apuntador al objeto receptor. En un diseño de objetos que interaccionan es necesario asegurar que se presenta la visibilidad adecuada para soportar la interacción de mensajes.

UML tiene una notación especial para representar la visibilidad.

10.2. Visibilidad

La **visibilidad** es la capacidad de un objeto de "ver" o tener una referencia a otro objeto. Está relacionada con el tema del alcance: ¿se encuentra un recurso (tal como una instancia) al alcance de otro? Hay cuatro formas de alcanzar la visibilidad desde un objeto A a un objeto B:

- **Visibilidad de atributo**: B es un atributo de A.
- **Visibilidad de parámetro**: B es un parámetro de un método de A.
- **Visibilidad local**: B es un objeto local (no un parámetro) en un método de A.
- **Visibilidad global**: B es de algún modo visible globalmente.

El motivo para tener en cuenta la visibilidad es que para que un objeto A envíe un mensaje a un objeto B, B debe ser visible a A.

Visibilidad de atributo

La **visibilidad de atributo** desde A a B existe cuando B es un atributo de A. Es relativamente permanente porque persiste mientras existan A y B. Es una forma de visibilidad muy común en los sistemas orientados a objetos.

Visibilidad de parámetro

La **visibilidad de parámetro** desde A a B existe cuando B se pasa como parámetro a un método de A. Es relativamente temporal porque persiste sólo en el alcance del método. Es la segunda forma más común de visibilidad en los sistemas orientados a objetos. Es habitual transformar la visibilidad de parámetro en visibilidad de atributo.

Visibilidad local

La **visibilidad local** desde A a B existe cuando B se declara como un objeto local en un método de A. Es una visibilidad relativamente temporal porque sólo persiste en el alcance del método. Es la tercera forma de visibilidad más común en los sistemas orientados a objetos.

Dos medios comunes de alcanzar la visibilidad local son:

- Crear una nueva instancia local y asignarla a una variable local.
- Asignar a una variable local el objeto de retorno de la invocación a un método.

Es habitual transformar la visibilidad declarada localmente en visibilidad de atributo.

Visibilidad global

La **visibilidad global** de A a B existe cuando B es global a A. Es relativamente permanente porque persiste mientras existan A y B. Es la forma menos común de visibilidad. Un medio de conseguir visibilidad global es asignar una instancia a una variable global, posible en algunos lenguajes como C++; pero no en otros como Java. El método que se prefiere para conseguir la visibilidad global es utilizar el patrón **Singleton**.

10.3. Representación de la visibilidad en UML

UML incluye notación para representar el tipo de visibilidad en un diagrama de colaboración. Estos adornos son opcionales y normalmente no se exigen (son útiles cuando se necesita alguna aclaración).

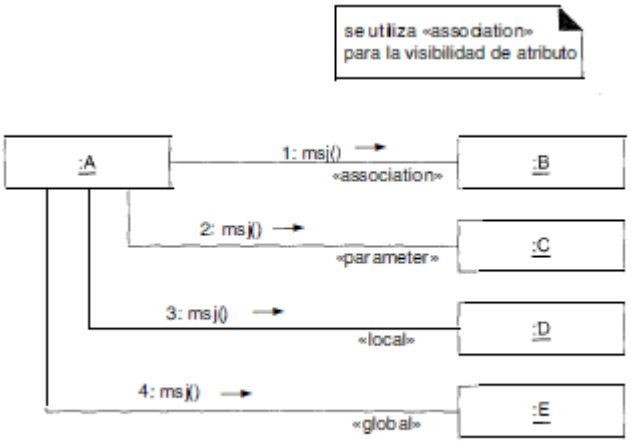


Figura 18.6. Implementación de los estereotipos para la visibilidad.

11. MODELO DE DISEÑO: CREACIÓN DE LOS DIAGRAMAS DE CLASES DE DISEÑO

Terminados los diagramas de interacción para las realizaciones de los casos de uso de la iteración actual, es posible identificar la especificación de las clases software (e interfaces) que participan en la solución software, y añadirles detalles de diseño, como los métodos. UML proporciona la notación para representar los detalles de diseño en los diagramas de clase.

11.1. Diagramas de Clases de Diseño (DCD)

Los DCD vienen después de la creación de los diagramas de interacción y se crean en paralelo. Al comienzo del diseño se podrían esbozar muchas clases, nombres de métodos y relaciones aplicando los patrones para asignar responsabilidades, antes de la elaboración de los diagramas de interacción.

Es posible y deseable elaborar algo de los diagramas de interacción, actualizar entonces los DCD, después extender los diagramas de interacción algo más, y así sucesivamente. Estos diagramas de clases podrían utilizarse como una notación más gráfica alternativa a las tarjetas CRC para recoger las responsabilidades y colaboraciones.

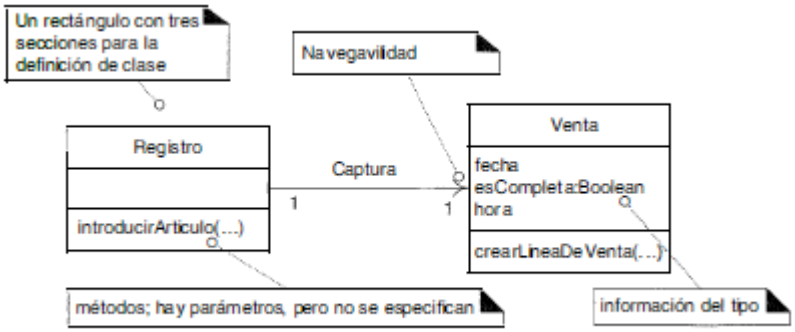


Figura 19.1. Ejemplo de diagrama de clases de diseño.

Además de las asociaciones y atributos básicos, el diagrama se amplía para representar los métodos de cada clase, información del tipo de los atributos, visibilidad de los atributos y navegación entre los objetos.

11.2. Terminología del DCD y el UP

Un diagrama de clases de diseño (DCD) representa las especificaciones de las clases e interfaces software en una aplicación:

- Clases, asociaciones y atributos.
- Interfaces, con sus operaciones y constantes.
- Métodos.
- Información acerca del tipo de los atributos.
- Navegabilidad.
- Dependencias.

Las clases de diseño de los DCD muestran las definiciones de las clases software en lugar de los conceptos del mundo real. El UP no define de manera específica ningún artefacto de "diagrama de clases de diseño". El UP define el Modelo de Diseño, que contiene varios tipos de diagramas, que incluye los diagramas de interacción, de paquetes, y los de clases. "Diagramas de clases de diseño" implica "diagramas de clases en el Modelo de Diseño".

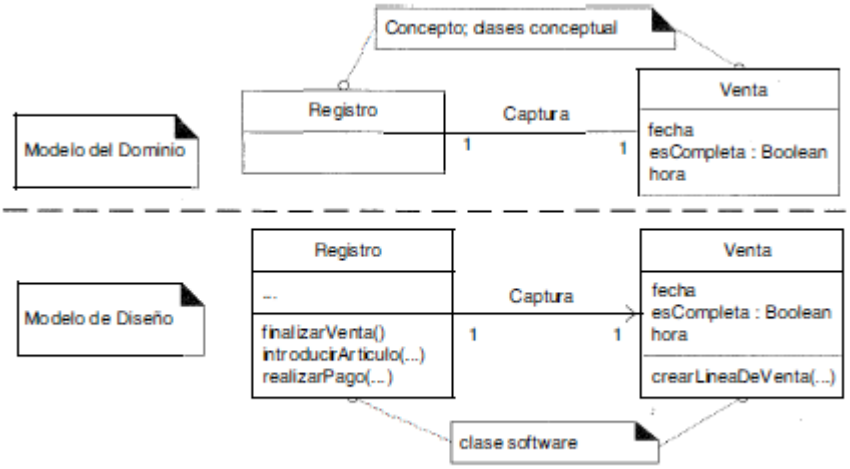


Figura 19.2. Clases del Modelo del Dominio vs. clases del Modelo de Diseño.

Clases del Modelo de Dominio vs. clases del Modelo de Diseño

El Modelo de Dominio del UP trata de una abstracción de un concepto del mundo real. Los DCD expresan, para la aplicación software, la definición de las clases como componentes software.

11.3. Creación de un DCD

Identificación y representación de las clases software

El primer paso es identificar aquellas clases que participan en la solución software. Se pueden encontrar examinando todos los diagramas de interacción y listando las clases que se mencionan. El siguiente paso es dibujar un diagrama de clases para estas clases e incluir los atributos que se identificaron previamente en el Modelo del Dominio que también se utilizan en el diseño.

Algunos de los conceptos del Modelo del Dominio no se incluyen en el diseño. No es necesario representarlos en el software (en iteraciones posteriores podrían formar parte del diseño).

Añadir los nombres de los métodos

Se pueden identificar los nombres de los métodos analizando los diagramas de interacción. El conjunto de todos los mensajes enviados a una clase X a lo largo de todos los diagramas de interacción indican la mayoría de los métodos que debe definir la clase X.

Cuestiones acerca de los nombres de los métodos

Cuestiones especiales en relación con los nombres de los métodos:

- Interpretación del mensaje créate.
- Descripción de los métodos de acceso.
- Interpretación de los mensajes a los multiobjetos.
- Sintaxis dependiente del lenguaje.

Nombres de los métodos - *create*

create es una forma independiente del lenguaje posible en UML para indicar instanciación e inicialización. No existe un método *create* ni en C++, ni en Java. En C++ implica la asignación automática, o asignación de almacenamiento libre con el operador *new* seguido de una llamada al constructor. En Java, implica la invocación del operador *new* seguido de una llamada al constructor.

Debido a que la inicialización es una actividad muy común, es habitual omitir en el DCD los métodos relacionados con la creación y los constructores.

Nombres de los métodos - métodos de acceso

Los métodos de acceso recuperan o establecen el valor de los atributos. En algunos lenguajes es común tener un *accessor* y un *mutator* para cada atributo, y declarar todos los atributos como privados (encapsulación de datos). Normalmente, se excluye la descripción de estos métodos en el diagrama de clases debido a que generan mucho ruido.

Nombres de los métodos - multiobjetos

Un mensaje a un multiobjeto se interpreta como un mensaje al propio objeto contenedor/colección. El método no forma parte de la clase sino de la interfaz del multiobjeto. Estas interfaces o clases de contenedores/colecciones son elementos de las librerías predefinidas, y no es útil mostrar explícitamente estas clases en el DCD, porque añaden ruido, pero poca información nueva.

Nombres de los métodos - sintaxis dependiente del lenguaje

Algunos lenguajes, como Smalltalk, tienen una sintaxis que es muy diferente del formato UML básico de *nombreMetodo(listaParametros)*. Se recomienda que se utilice el formato UML básico, incluso si el lenguaje de implementación que se planea utilizar tiene una sintaxis diferente (la traducción debería tener lugar en el momento de la generación del código). Sin embargo, UML permite otra sintaxis para la especificación de los métodos.

Añadir más información sobre los tipos

Todos los tipos de los atributos, parámetros de los métodos y los valores de retorno se podrían mostrar. La cuestión sobre si se muestra o no se debe considerar en el siguiente contexto:

Se debería crear un DCD teniendo en cuenta los destinatarios.

- Si se está creando en una herramienta CASE con generación automática de código, son necesarios todos los detalles y de modo exhaustivo.
- Si se está creando para que lo lean los desarrolladores de software, los detalles exhaustivos de bajo nivel podrían afectar negativamente por el nivel de ruido.

Añadir asociaciones y navegabilidad

Cada extremo de asociación se denomina rol, y en los DCD el rol podría decorarse con una flecha de navegabilidad. La **navegabilidad** es una propiedad del rol que indica que es posible navegar unidireccionalmente a través de la asociación desde los objetos de la clase origen a la clase destino. La navegabilidad implica visibilidad, normalmente de atributo.

La interpretación habitual de una asociación con una flecha de visibilidad es la visibilidad de atributo desde la clase origen hasta la clase destino. Durante la implementación en un lenguaje orientado a objetos por lo general se transforma en un atributo en la clase origen que hace referencia a una instancia de la clase destino. La mayoría de las asociaciones en los DCD deberían adornarse con las flechas de navegabilidad necesarias.

De acuerdo a un criterio necesito-conocer orientado al software estricto, esto contrasta con las asociaciones en el Modelo del Dominio, que se podrían justificar por la intención de mejorar la comprensión del dominio del problema (existe

una diferencia entre los objetivos del Modelo del Diseño y del Modelo del Dominio: uno es analítico, el otro una descripción de los componentes software).

La visibilidad y las asociaciones requeridas entre las clases se dan a conocer mediante los diagramas de interacción. Situaciones comunes que sugieren la necesidad de definir una asociación con un adorno de visibilidad de A a B:

- A envía un mensaje a B.
- A crea una instancia de B.
- A necesita mantener una conexión a B.

Las conexiones implícitas se presentarán como asociaciones en el diagrama de clases.

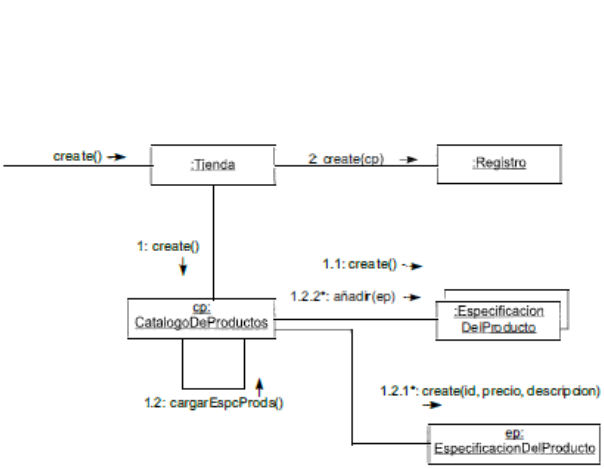


Figura 19.9. La navegabilidad se identifica a partir de los diagramas de interacción.

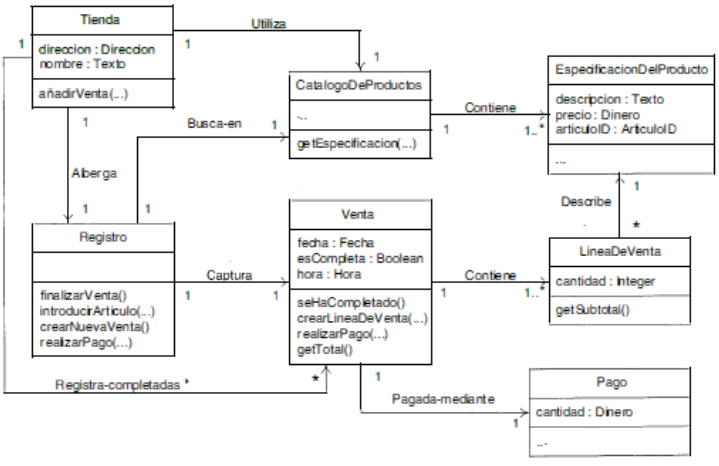


Figura 19.10. Asociaciones con adornos de navegabilidad.

Añadir las relaciones de dependencia

UML incluye una **relación de dependencia** general, que indica que un elemento tiene conocimiento de otro elemento. Se representa mediante una línea de flecha punteada. En los diagramas de clases la relación de dependencia es útil para describir la visibilidad entre clases que no es de atributo (de parámetro, local o global). La visibilidad de atributo simple se muestra mediante una línea de asociación ordinaria y una flecha de navegabilidad.

UML proporciona una notación variada para describir las características de los miembros de las clases e interfaces, como la visibilidad, valores iniciales, etcétera. Si no se muestra explícitamente ningún marcador de visibilidad para un atributo o método, no hay un valor por defecto. En UML significa “sin especificar”. Sin embargo, la convención común es asumir que los atributos son privados y los métodos públicos, a menos que se indique otra cosa.

DCD, dibujo y herramientas CASE

Las herramientas CASE pueden hacer ingeniería inversa (generar) de los DCD a partir del código fuente.

11.4. DCD en el UP

Los DCD forman parte de la realización de los casos de uso y, por tanto, miembros del Modelo de Diseño del UP.

Fases

- **Inicio**: El Modelo de Diseño y los DCD no comenzarán hasta la elaboración porque comprenden decisiones de diseño (prematuras durante la fase de inicio).
- **Elaboración**: Los DCD acompañarán a los diagramas de interacción de las realizaciones de los casos de uso; podrían crearse para las clases del diseño.
- **Construcción**: Los DCD se continuarán generando a partir del código fuente como apoyo a la visualización de la estructura estática del sistema.

Tabla 19.1. Muestra de los artefactos UP y evolución temporal. c-comenzar; r-refinar

Disciplina	*Artefacto* / *Iteración →*	*Inicio* *I1*	*Elab.* *E1…En*	*Const.* *C1…Cn*	*Trans.* *T1…T2*
Modelado del Negocio	Modelo del Dominio		c		
Requisitos	Modelo de Casos de Uso (DSS) Visión Especificación Complementaria Glosario	c c c c	r r r r		
Diseño	***Modelo de Diseño*** Documento de Arquitectura SW Modelo de Datos		c c c	r r	
Implementación	Modelo de Implementación		c	r	r
Gestión del Proyecto	Plan de Desarrollo SW	c	r	r	r
Pruebas	Modelo de Pruebas		c	r	
Entorno	Marco de Desarrollo	c	r		

11.5. Artefactos del UP

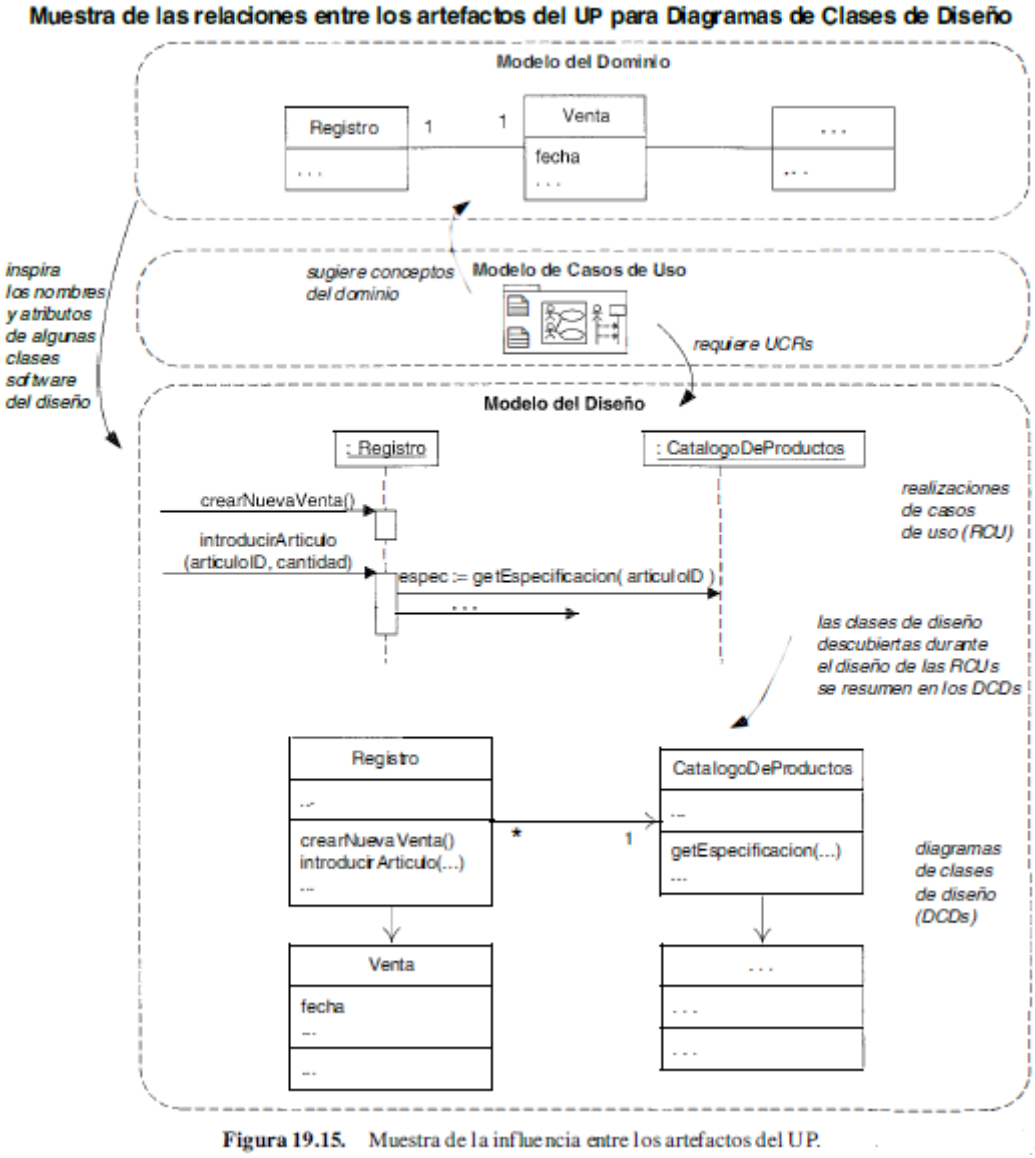


Figura 19.15. Muestra de la influencia entre los artefactos del UP.

12. MODELO DE IMPLEMENTACIÓN: TRANSFORMACIÓN DE LOS DISEÑOS EN CÓDIGO

Después de completar los diagramas de interacción y los DCDs para la iteración actual, disponemos de suficientes detalles para generar el código de la capa del dominio de los objetos. Los artefactos UML creados durante el trabajo de

diseño (los diagramas de interacción y los DCDs) se utilizarán como entradas en el proceso de generación de código. El UP define el Modelo de Implementación. contiene los artefactos de implementación como el código fuente, las definiciones de bases de datos, las páginas JSP/XML/HTML, etcétera.

12.1. Programación y el proceso de desarrollo

El trabajo de diseño anterior no debería implicar que no exista prototipado o diseño durante la programación (normalmente merece la pena elaborar algo, o incluso mucho, de diseño mientras se programa). Algunos desarrolladores encuentran útil anticipar un poco mediante el modelado visual antes de programar.

La creación de código en un lenguaje de programación orientado a objetos no forma parte del A/DOO; es un objetivo final. Los artefactos creados en el Modelo de Diseño del UP proporcionan parte de la información necesaria para generar el código.

Una ventaja del A/DOO y de la programación OO, cuando se utiliza con el UP, es que proporcionan una guía de principio a fin desde los requisitos hasta el código. Los distintos artefactos suministran información a los artefactos posteriores de manera útil y siguiendo una traza, culminando finalmente en una aplicación en ejecución.

Creatividad y cambio durante la implementación

Durante el trabajo de diseño se tomaron algunas decisiones y se llevó a cabo un trabajo creativo. La generación de código es un proceso de traducción relativamente mecánico. El trabajo de programación no es una etapa de generación de código trivial, más bien lo contrario. Los resultados generados durante el diseño son un primer paso incompleto; durante la programación y pruebas, se realizarán innumerables cambios y se descubrirán y resolverán problemas complicados.

Los artefactos del diseño proporcionarán un núcleo elástico que podrá extenderse con elegancia y robustez para satisfacer los nuevos problemas.

Cambios del código y el proceso iterativo

Una ventaja del proceso de desarrollo iterativo e incremental es que los resultados de la iteración anterior pueden proporcionar los datos para el comienzo de la siguiente iteración. Los resultados del análisis y diseño actual se están refinando y enriqueciendo continuamente a partir del trabajo de implementación anterior.

Una de las primeras actividades en una iteración es sincronizar los diagramas de diseño; los primeros diagramas de la iteración N no corresponderán con el código final de esa misma iteración, necesitan sincronizarse antes de que se extiendan con nuevos resultados del diseño.

Cambios en el código, herramientas CASE, e ingeniería inversa

Es deseable que los diagramas generados se actualicen de manera semi-automática para reflejar los cambios en el trabajo de codificación siguiente. Esto debería hacerse con una herramienta CASE que puede leer el código fuente y generar automáticamente diagramas de paquetes, clases y secuencia → **ingeniería inversa**: la actividad de generar diagramas a partir del código fuente.

12.2. Transformación de los diseños en código

La implementación en un lenguaje orientado a objetos requiere la escritura de código fuente para:

- Las definiciones de las clases e interfaces.
- Las definiciones de los métodos.

12.3. Creación de las definiciones de las clases a partir de los DCDs

Los DCDs describen los nombres de las clases o interfaces, las superclases, signatura de los métodos y los atributos simples de una clase. Esto es suficiente para crear una definición de clase básica en un lenguaje de programación orientado a objetos.

Definición de una clase con métodos y atributos simples

A partir del DCD, la transformación de las definiciones de los atributos básicas y las signaturas de los métodos a la definición Java es directa. En Java se requiere un constructor que soporte parámetros. A menudo se excluye de los diagramas de clases del método *create* puesto que es común que aparezca y tiene múltiples interpretaciones, dependiendo del lenguaje que se vaya a utilizar.

Añadir atributos de referencia

Un **atributo de referencia** es un atributo que referencia a otro objeto complejo, no a un tipo primitivo como una cadena de texto, un número, etcétera. Se deducen de las asociaciones y la navegabilidad de un diagrama de clases. Los atributos de referencia de una clase a menudo están implícitos, en lugar de explícitos, en un DCD. Existe una visibilidad de atributo *recomendada*, indicada por la asociación y navegabilidad, que se define explícitamente mediante un atributo durante la fase de generación de código.

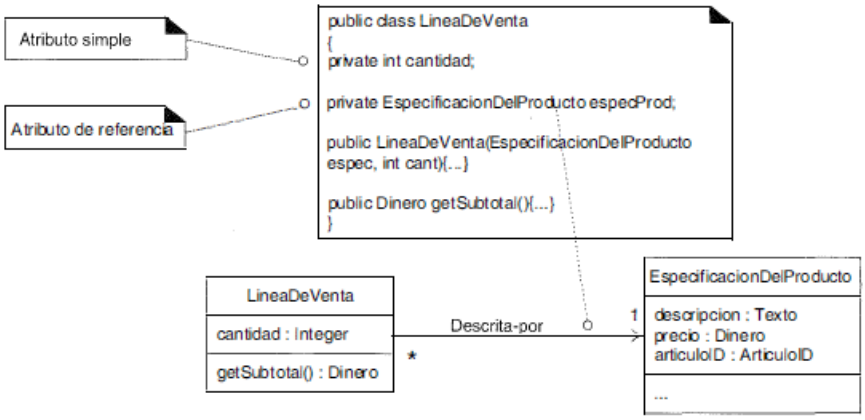


Figura 20.3. Inclusión de atributos de referencia.

Atributos de referencia y los nombres de los roles

La siguiente iteración estudiará el concepto de los nombres de los roles en los diagramas de estructura estáticos. Un **nombre de rol** es un nombre que identifica al rol y, a menudo, proporciona algo del contexto semántico acerca de la naturaleza del rol.

Si el nombre de un rol está presente en un diagrama de clases, utilícelo como base para el nombre del atributo de referencia durante la generación de código.

Transformación de los atributos

En algunos casos se debe considerar la transformación de los atributos desde el diseño al código en lenguajes diferentes.

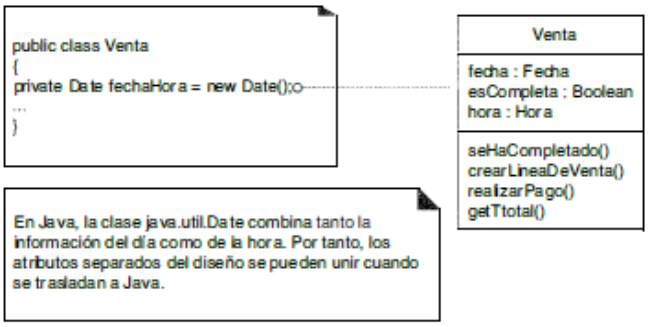


Figura 20.5. Transformación de la fecha y la hora en Java.

12.4. Creación de métodos a partir de los diagramas de interacción

Un diagrama de interacción muestra los mensajes que se envían como respuesta a la invocación de un método. La secuencia de estos mensajes se traduce en una serie de sentencias en la definición del método.

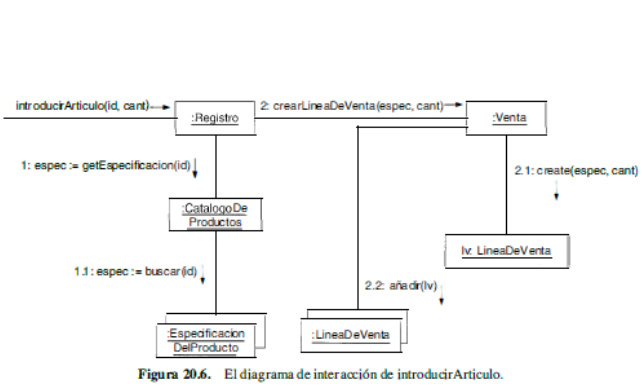


Figura 20.6. El diagrama de interacción de introducirArticulo.

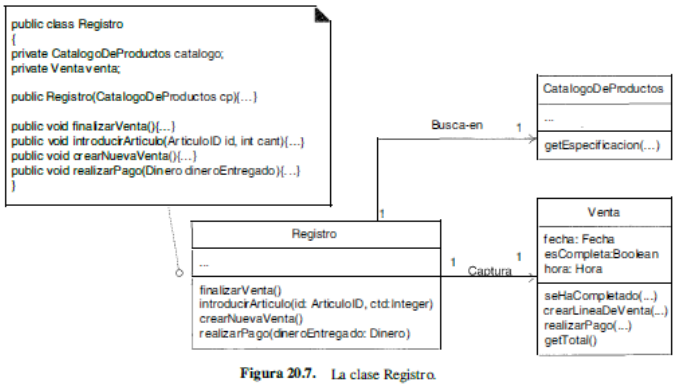


Figura 20.7. La clase Registro.

12.5. Clases contenedoras/colecciones en el código

A menudo es necesario que un objeto mantenga la visibilidad a un grupo de otros objetos; normalmente esta necesidad es evidente a partir del valor de la multiplicidad en el diagrama de clases. En los lenguajes de programación OO se implementan estas relaciones introduciendo un contenedor o colección intermedia. La clase del lado del uno define un atributo de referencia que apunta a la instancia del contenedor/colección, que contiene instancias de la clase del lado de muchos.

Los requisitos influyen en la elección de la clase colección; las búsquedas por claves requieren el uso de un *Map*, una lista ordenada creciente requiere una *List*, etcétera.

12.6. Manejo de excepciones y de errores

En el desarrollo de la aplicación, es aconsejable tener en cuenta el manejo de las excepciones durante el trabajo de diseño y, por supuesto, durante la implementación. En UML, las excepciones se representan como mensajes asíncronos en los diagramas de interacción.

12.7. Orden de implementación

Es necesario implementar las clases (e idealmente, hacer las pruebas de unidad completamente) desde la menos a la más acoplada.

Una práctica excelente promovida por el método de Programación Extrema, y aplicable al UP, es **programar probando primero**. Se escribe el código de las pruebas de unidad *antes* del código que se va a probar, y los desarrolladores escriben el código de las pruebas de unidad para *todo* el código de producción. La secuencia básica es escribir un poco del código de prueba, escribir un poco del código de producción, hacer que pase las pruebas, y entonces escribir más código de prueba, y así sucesivamente.

Ventajas:

- Se escriben realmente las pruebas de unidad.
- Satisfacción de los programadores.
- Aclaración de las interfaces y el comportamiento.
- Verificación demostrable.
- La confianza en cambiar cosas.

No es necesario escribir todos los métodos de prueba de antemano. Un desarrollador escribe un método de prueba, después el código de producción que la satisface, a continuación, otro método de prueba, y así sucesivamente.

12.8. Resumen de la transformación de los diseños en código

El proceso de traducción de los DCD a las definiciones de las clases, y de los diagramas de interacción en los métodos, es relativamente directo. Durante el trabajo de programación todavía hay cabida para tomar decisiones, realizar cambios en el diseño y explorar, pero algunas de las grandes ideas del diseño se tuvieron en cuenta antes de la programación.