Historial de cambios

Versión	Fecha	Autor	Descripción de cambios
01.00	12/09/2017	Rosa Sanabria	Introducción a la Ingeniería de Software

Índice.

Contenido.

1 Conceptos Iniciales

1.1 Concepto de Ingeniería de Software

Desde sus inicios en la década de 1940, la forma de escribir *software* ha evolucionado hasta convertirse hoy en día en una profesión, que se ocupa de **crear** software y maximizar su calidad.

¿Porque crear y no construir software como lo hacen las demás disciplinas ingenieriles? Los sistemas de software son abstractos e intangibles. No están restringidos por las propiedades de los materiales, regidos por leyes físicas ni por procesos de fabricación. Esto simplifica la ingeniería de software, pues no existen límites naturales a su potencial. Sin embargo, debido a la falta de restricciones físicas, los sistemas de software pueden volverse rápidamente muy complejos, difíciles de entender y costosos de cambiar.

Los ingenieros de software son los responsables de crear software, y como ingenieros tienen el compromiso de realizarlo con calidad. La calidad puede referirse a cuán mantenible es el software, su estabilidad, velocidad, usabilidad, comprobabilidad, legibilidad, tamaño, costo, seguridad y número de fallas o "bugs", así como, entre muchos otros atributos, a cualidades menos medibles como elegancia, concisión y satisfacción del cliente. La mejor manera de crear *software* de alta calidad es un problema separado y controvertido cubriendo el **diseño de *software***, principios para escribir código, llamados **"mejores prácticas"**, así como cuestiones más amplias de gestión como tamaño óptimo del equipo de trabajo, el proceso, la mejor manera de entregar el *software* a tiempo y tan rápidamente como sea posible, la "cultura" del lugar de trabajo, prácticas de contratación y así sucesivamente. Todo esto cae bajo la rúbrica general de ingeniería de software.

La ingeniería de software es esencial para el funcionamiento de las sociedades, ya que todo hoy en día tiene software. Hay muchos tipos diferentes de sistemas de software, desde los simples sistemas embebidos, hasta los complejos sistemas de información mundial. Por los que no tiene sentido buscar notaciones, métodos o técnicas universales para la ingeniería de software, ya que diferentes tipos de software requieren distintos enfoques. Desarrollar un sistema organizacional de información es completamente diferente de un controlador para un instrumento científico. Ninguno de estos sistemas tiene mucho en común con un juego por computadora de gráficos intensivos. Aunque todas estas aplicaciones necesitan *ingeniería de software, no todas requieren las mismas técnicas de ingeniería de software*. Aún existen muchos reportes tanto de proyectos de software que salen mal como de “fallas de software”. Por ello, a la ingeniería de software se le considera inadecuada para el desarrollo del software moderno. Sin embargo, existen factores que determinan que la construcción de software, a pesar de utilizar un proceso ingenieril tenga fallas:

- Demandas crecientes Conforme las nuevas técnicas de ingeniería de software ayudan a construir sistemas más grandes y complejos, las demandas cambian. Los sistemas tienen que construirse y distribuirse más rápidamente; se requieren sistemas más grandes e incluso más complejos; los

sistemas deben tener nuevas capacidades que anteriormente se consideraban imposibles. Los métodos existentes de ingeniería de software no pueden enfrentar la situación, y tienen que desarrollarse nuevas técnicas de ingeniería de software para satisfacer nuevas demandas.

- Expectativas bajas Es relativamente sencillo escribir programas de cómputo sin usar métodos y técnicas de ingeniería de software. Muchas compañías se deslizan hacia la ingeniería de software conforme evolucionan sus productos y servicios. No usan métodos de ingeniería de software en su trabajo diario. Por lo tanto, su software con frecuencia es más costoso y menos confiable de lo que debiera. Es necesaria una mejor educación y capacitación en ingeniería de software para solucionar este problema. Los ingenieros de software pueden estar orgullosos de sus logros. Desde luego, todavía se presentan problemas al desarrollar software complejo, pero, sin ingeniería de software, no se habría explorado el espacio ni se tendría Internet o las telecomunicaciones modernas. Todas las formas de viaje serían más peligrosas y caras. La ingeniería de software ha contribuido en gran medida, y sus aportaciones en el siglo XXI serán aún mayores.

1.1.1 Un poco de Historia.

El concepto “ingeniería de software” se propuso originalmente en 1968, en una conferencia realizada para discutir lo que entonces se llamaba la “crisis del software” (Naur y Randell, 1969). Aquí nace formalmente la rama de la ingeniería de software. El término se adjudica a F. L. Bauer, aunque previamente había sido utilizado por Edsger Dijkstra en su obra The Humble Programmer.

Básicamente, la crisis del software se refiere a la dificultad en escribir programas libres de defectos, fácilmente comprensibles, y que sean verificables. Las causas son, entre otras, la complejidad que supone la tarea de programar, y los cambios a los que se tiene que ver sometido un programa para ser continuamente adaptado a las necesidades de los usuarios.

El desarrollo del software estaba en "crisis". Se necesitaban nuevas técnicas y métodos para controlar la complejidad inherente a los sistemas grandes. Estas técnicas han llegado a ser parte de la ingeniería de software y son ampliamente utilizadas.

Sin embargo, cuanto más crezca nuestra capacidad para producir software, también lo hará la complejidad de los sistemas solicitados. De este modo, los problemas asociados con el desarrollo del software se han caracterizado como una "crisis".

A lo largo de las décadas de 1970 y 1980 se desarrolló una variedad de nuevas técnicas y métodos de ingeniería de software, tales como la programación estructurada, el encubrimiento de información y el desarrollo orientado a objetos. Se perfeccionaron herramientas y notaciones estándar y ahora se usan de manera extensa.

Sin embargo, había indicadores de que dejaban claro que algo estaba fallando y que era necesario un cambio en la forma de desarrollar productos software.

2 Estadísticas % de proyectos

1.1.1.1 ¿Cuáles son las razones para la crisis del software?

La respuesta está en el análisis de los siguientes aspectos:

Base inestable: con respecto a las necesidades del negocio

La complejidad del software: La demanda del software de negocios se está incrementando. Los sistemas cada vez son más complejos y resulta dificultoso comprenderlos.

Formación del recurso humano: Es frecuente que los gestores y administradores de las empresas de software, sean gestores reconvertidos de otras áreas, que no poseen una formación informática específica, por lo que no siempre comprenden bien los problemas y las necesidades que se van a presentar durante el desarrollo de una aplicación software.

Fallas en el manejo de riesgos: la utilización de un modelo de proceso que no se adecua a las necesidades de las organizaciones es a veces una de las razones por las que la implementación de un producto falle ya que sólo se tiene la certeza de que el sistema funciona o no cerca de ser terminado.

1.2 Conceptos teóricos

La ingeniería de software busca apoyar el desarrollo de software profesional, en lugar de la programación individual. Incluye técnicas que apoyan la especificación, el diseño y la evolución del programa. Es una disciplina de ingeniería que se interesa por todos los aspectos de la producción de software, desde las primeras etapas de la especificación del sistema hasta el mantenimiento del sistema después de que se pone en operación. En esta definición se presentan dos frases clave:

1. Disciplina de ingeniería Los ingenieros hacen que las cosas funcionen. Aplican teorías, métodos y herramientas donde es adecuado. Sin embargo, los usan de manera selectiva y siempre tratan de encontrar soluciones a problemas, incluso cuando no hay teorías ni métodos aplicables. Los ingenieros también reconocen que deben trabajar ante restricciones organizacionales y financieras, de modo que buscan soluciones dentro de tales limitaciones.

2. Todos los aspectos de la producción del software La ingeniería de software no sólo se interesa por los procesos técnicos del desarrollo de software, sino también incluye actividades como la administración del proyecto de software y el desarrollo de herramientas, así como métodos y teorías para apoyar la producción de software.

La ingeniería busca obtener resultados de la **calidad** requerida dentro de la fecha y del presupuesto. A menudo esto requiere contraer compromisos: los ingenieros no deben ser perfeccionistas. En general, los ingenieros de software adoptan en su trabajo un enfoque sistemático y organizado, pues usualmente ésta es la forma más efectiva de producir software de alta calidad. No obstante, la ingeniería busca seleccionar el método más adecuado para un conjunto de circunstancias y, de esta manera, un acercamiento al desarrollo más creativo y menos formal sería efectivo en ciertas situaciones. El desarrollo menos formal es particularmente adecuado para la creación de sistemas basados en la Web, que requieren una mezcla de habilidades de software y diseño gráfico.

1.2.1 La ingeniería de software es importante por dos razones:

1. Cada vez con mayor frecuencia, los individuos y la sociedad se apoyan en los avanzados sistemas de software. Por ende, se requiere producir económica y rápidamente sistemas confiables.
2. A menudo resulta más barato a largo plazo usar métodos y técnicas de ingeniería de software para los sistemas de software, que sólo diseñar los programas como si fuera un proyecto de programación personal. Para muchos tipos de sistemas, la mayoría de los costos consisten en cambiar el software después de ponerlo en operación.

El enfoque sistemático que se usa en la ingeniería de software se conoce en ocasiones como proceso de software. Un proceso de software es una secuencia de actividades que conducen a la elaboración de un producto de software. Existen cuatro actividades fundamentales que son comunes a todos los procesos de software, y éstas son:

1. Especificación del software, donde clientes e ingenieros definen el software que se producirá y las restricciones en su operación.

2. Desarrollo del software, donde se diseña y programa el software.

3. Validación del software, donde se verifica el software para asegurar que sea lo que el cliente requiere.

4. Evolución del software, donde se modifica el software para reflejar los requerimientos cambiantes del cliente y del mercado.

Diferentes tipos de sistemas necesitan distintos procesos de desarrollo. Por ejemplo, el software en tiempo real en una aeronave debe especificarse por completo antes de comenzar el desarrollo. En los sistemas de comercio electrónico, la especificación y el programa por lo general se desarrollan en

conjunto. En consecuencia, tales actividades genéricas pueden organizarse en diferentes formas y describirse en distintos niveles de detalle, dependiendo del tipo de software que se vaya a desarrollar.

1.2.2 Diseño en el contexto de la ingeniería de software

El diseño de software se ubica en el área técnica de la ingeniería de software y se aplica sin importar el modelo del proceso que se utilice. Comienza una vez que se han analizado y modelado los requerimientos, es la última acción de la ingeniería de software dentro de la actividad de modelado y prepara la etapa de construcción (generación y prueba de código).

El diseño de software agrupa el conjunto de principios, conceptos y prácticas que llevan al desarrollo de un sistema o producto de alta calidad. Los principios de diseño establecen una filosofía general que guía el trabajo de diseño que debe ejecutarse.

El diseño es lo que casi todo ingeniero quiere hacer. Es el lugar en el que las reglas de la creatividad, los requerimientos de los participantes, las necesidades del negocio y las consideraciones técnicas se unen para formular un producto o sistema. El diseño crea una representación o modelo del software, pero, a diferencia del modelo de los requerimientos (que se centra en describir los datos que se necesitan, la función y el comportamiento), el modelo de diseño proporciona detalles sobre arquitectura del software, estructuras de datos, interfaces y componentes que se necesitan para implementar el sistema.

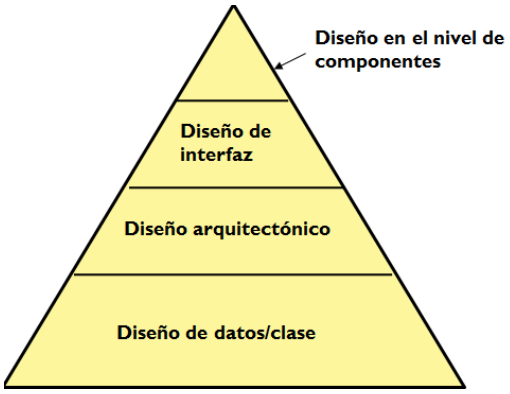


Ilustración 3 Modelo de Diseño

¿Porque es importante? Porque permite modelar el sistema o producto que se va a construir. Este modelo se evalúa respecto de la calidad y su mejora antes de generar código; después, se efectúan pruebas y se involucra a muchos usuarios finales. **El diseño es el lugar en el que se establece la calidad del software.**

1.2.2.1 El diseño representa al software de varias maneras.

En primer lugar, debe representarse la arquitectura del sistema o producto. Después se modelan las interfaces que conectan al software con los usuarios finales, con otros sistemas y dispositivos, y con sus propios componentes constitutivos. Por último, se diseñan los componentes del software que se utilizan para construir el sistema. Cada una de estas perspectivas representa una acción de diseño distinta, pero

todas deben apegarse a un conjunto básico de conceptos de diseño que guíe el trabajo de producción de software.

1.2.2.2 ¿Cuál es el producto final?

El trabajo principal que se produce durante el diseño del software es un modelo de diseño que agrupa las representaciones arquitectónicas, interfaces en el nivel de componente y despliegue.

1.2.2.3 ¿Cómo me aseguro de que lo hice bien?

El modelo de diseño es evaluado por el equipo de software en un esfuerzo por determinar si contiene errores, inconsistencias u omisiones, si existen mejores alternativas y si es posible implementar el modelo dentro de las restricciones, plazo y costo que se hayan establecido.

El diseño del software cambia continuamente conforme evolucionan los nuevos métodos, surgen mejores análisis y se obtiene una comprensión más amplia.

La entrada al modelo de diseño es el modelo de análisis que proporciona información necesaria para crear los cuatro modelos de diseño necesarios para la especificación completa del diseño.

1.2.2.4 El proceso de diseño

El diseño de software es un proceso iterativo por medio del cual se traducen los requerimientos en un "plano" para construir el software. Al principio, el diseño se representa en un nivel alto de abstracción, en el que se rastrea directamente el objetivo específico del sistema y los requerimientos más detallados de datos, funcionamiento y comportamiento. A

medida que tienen lugar las iteraciones del diseño, las mejoras posteriores conducen a niveles menores de abstracción. Éstos también pueden rastrearse hasta los requerimientos, pero la conexión es más sutil.

1.2.2.5 Lineamientos y atributos de la calidad del software

1. Debe implementar todos los requerimientos explícitos contenidos en el modelo de requerimientos y dar cabida a todos los requerimientos implícitos que desean los participantes.
2. Debe ser una guía legible y comprensible para quienes generan el código y para los que lo prueban y dan el apoyo posterior.
3. Debe proporcionar el panorama completo del software, y abordar los dominios de los datos, las funciones y el comportamiento desde el punto de vista de la implementación.

1.2.2.6 Conceptos del buen diseño

1. Debe tener una arquitectura que
   (a) se haya creado con el empleo de estilos o patrones arquitectónicos reconocibles,
   (b) esté compuesta de componentes con buenas características de diseño (éstas se analizan más adelante, en este capítulo), y

(c) se implementen en forma evolutiva, de modo que faciliten la implementación y las pruebas.

2. Debe ser modular, es decir, el software debe estar dividido de manera lógica en elementos o subsistemas.
3. Debe contener distintas representaciones de datos, arquitectura, interfaces y componentes.
4. Debe conducir a estructuras de datos apropiadas para las clases que se van a implementar y que surjan de patrones reconocibles de datos.
5. Debe llevar a componentes que tengan características funcionales independientes.
6. Debe conducir a interfaces que reduzcan la complejidad de las conexiones entre los componentes y el ambiente externo.
7. Debe obtenerse con el empleo de un método repetible motivado por la información obtenida durante el análisis de los requerimientos del software.
8. Debe representarse con una notación que comunique con eficacia su significado.

1.2.2.7 Atributos de la calidad.

Hewlett-Packard [Gra 87] desarrolló un conjunto de atributos de la calidad del software a los que se dio el acrónimo FURPS: funcionalidad, usabilidad, confiabilidad, rendimiento y mantenibilidad. Los atributos de calidad FURPS representan el objetivo de todo diseño de software:

• La **funcionalidad** se califica de acuerdo con el conjunto de características y capacidades del programa, la generalidad de las funciones que se entregan y la seguridad general del sistema.

• La **usabilidad** se evalúa tomando en cuenta factores humanos, la estética, general, la consistencia y la documentación.

• La **confiabilidad** se evalúa con la medición de la frecuencia y gravedad de las fallas, la exactitud de los resultados que salen, el tiempo medio para que ocurra una falla (TMPF), la capacidad de recuperación ante ésta y lo predecible del programa.

• El **rendimiento** se mide con base en la velocidad de procesamiento, el tiempo de respuesta, el uso de recursos, el conjunto y la eficiencia.

• La **mantenibilidad** combina la capacidad del programa para ser ampliable (extensibilidad), adaptable y servicial (estos tres atributos se denotan con un término más común: **mantenibilidad**), y además que pueda probarse, ser compatible y configurable (capacidad de organizar y controlar los elementos de la configuración del software) y que cuente con la facilidad para instalarse en el sistema y para que se detecten los problemas.

No todo atributo de la calidad del software se pondera por igual al diseñarlo. Una aplicación tal vez se aboque a lo funcional con énfasis en la seguridad. Otra quizá busque rendimiento con la mira puesta en la velocidad de procesamiento. En una tercera se persigue la confiabilidad. Sin importar la ponderación, es importante observar que estos atributos de la calidad deben tomarse en cuenta cuando comienza el diseño, no cuando haya terminado éste y la construcción se encuentre en marcha.

1.2.2.8 Conjunto de tareas generales para el diseño

1. Estudiar el modelo del dominio de la información y diseñar las estructuras de datos apropiadas para los objetos de datos y sus atributos.

2. Seleccionar un estilo de arquitectura que sea adecuado para el software con el uso del modelo de análisis.

3. Hacer la partición del modelo de análisis en subsistemas de diseño y asignar éstos dentro de la arquitectura:

Asegúrese de que cada subsistema sea cohesivo en sus funciones.
Diseñe interfaces del subsistema.
Asigne clases de análisis o funciones a cada subsistema.

4. Crear un conjunto de clases de diseño o componentes:

Traduzca la descripción de clases de análisis a una clase de diseño.
Compare cada clase de diseño con los criterios de diseño; considere los aspectos hereditarios.
Defina métodos y mensajes asociados con cada clase de diseño.
Evalúe y seleccione patrones de diseño para una clase de diseño o subsistema.
Revise las clases de diseño y, si se requiere, modifíquelas.

5. Diseñar cualesquiera interfaces requeridas con sistemas o dispositivos externos.

6. Diseñar la interfaz de usuario.

Revise los resultados del análisis de tareas.
Especifique la secuencia de acciones con base en los escenarios de usuario.
Cree un modelo de comportamiento de la interfaz.
Defina los objetos de la interfaz y los mecanismos de control.
Revise el diseño de la interfaz y, si se requiere, modifíquelo.

7. Efectuar el diseño en el nivel de componente.

Especifique todos los algoritmos en un nivel de abstracción relativamente bajo.
Mejore la interfaz de cada componente.
Defina estructuras de datos en el nivel de componente.
Revise cada componente y corrija todos los errores que se detecten.

8. Desarrollar un modelo de despliegue.

2 Metodologías

2.1 Modelos de Proceso de desarrollo de Software

Por su naturaleza, los modelos son simplificaciones; por lo tanto, un modelo de procesos de software es una simplificación o abstracción de un proceso real. Podemos definir un modelo de proceso del software como una representación abstracta de alto nivel de un proceso software. Cada modelo es una descripción de un proceso software que se presenta desde una perspectiva particular. Alternativamente, a veces se usan los términos ciclo de vida y Modelo de ciclo de vida. Cada modelo describe una sucesión de fases y un encadenamiento entre ellas. Según las fases y el modo en que se produzca este encadenamiento, tenemos diferentes modelos de proceso. Un modelo es más adecuado que otro para desarrollar un proyecto dependiendo de un conjunto de características de éste.

Los modelos de proceso de software se clasifican en descriptivos y prescriptivo. Los primeros suelen ser informativos o analíticos y en algunos casos se realizan una vez que el proceso fue realizado. Los modelos de proceso prescriptivos son aquellos que definen las actividades, acciones, tareas, hitos y artefactos para

desarrollar un producto software de calidad y estas actividades pueden ser lineales, incrementales, evolutivas.

Un proceso del software es un conjunto de actividades que conducen a la creación de un producto software. Estas actividades pueden consistir en el desarrollo de software desde cero en un lenguaje de programación estándar como Java o C. Sin embargo, cada vez más, se desarrolla nuevo software ampliando y modificando los sistemas existentes y configurando e integrando software comercial o componentes del sistema.

Los procesos del software son complejos y, como todos los procesos intelectuales y creativos, dependen de las personas que toman decisiones y juicios. Debido a la necesidad de juzgar y crear, los intentos para automatizar estos procesos han tenido un éxito limitado. Las herramientas de ingeniería del software asistida por computadora (CASE) pueden ayudar a algunas actividades del proceso. Sin embargo, no existe posibilidad alguna, al menos en los próximos años, de una automatización mayor en el diseño creativo del software realizado por los ingenieros relacionados con el proceso del software.

Una razón por la cual la eficacia de las herramientas CASE está limitada se halla en la inmensa diversidad de procesos del software. No existe un proceso ideal, y muchas organizaciones han desarrollado su propio enfoque para el desarrollo del software. Los procesos han evolucionado para explotar las capacidades de las personas de una organización, así como las características específicas de los sistemas que se están desarrollando. Para algunos sistemas, como los sistemas críticos, se requiere un proceso de desarrollo muy estructurado. Para sistemas de negocio, con requisitos rápidamente cambiantes, un proceso flexible y ágil probablemente sea más efectivo.

Aunque no existe un proceso del software «ideal», en las organizaciones existen enfoques para mejorarlos. Los procesos pueden incluir técnicas anticuadas o no aprovecharse de las mejores prácticas en la ingeniería del software industrial. De hecho, muchas organizaciones aún no aprovechan los métodos de la ingeniería del software en el desarrollo de su software.

Los procesos del software se pueden mejorar por la estandarización del proceso donde la diversidad de los procesos del software en una organización sea reducida. Esto conduce a mejorar la comunicación y a una reducción del tiempo de formación, y hace la ayuda al proceso automatizado más económica. La estandarización también es un primer paso importante para introducir nuevos métodos, técnicas y buenas prácticas de ingeniería del software.

El presente apunte describe los principales modelos de proceso. Para una mejor comprensión se presenta una cronología de estos modelos y luego se describen cada uno.

BREVE CRONOLOGÏA DE LOS MODELOS DE PROCESO:
1970 Royce – El modelo Cascada
1980 Mills – Estrategia Incremental
1982 McCraken – Estrategia Iterativa
1983 Blazer – Modelo de Transformación
1984 Floyd – Estrategia con Prototipos
1988 Boehm – Modelo en Espiral
1988 Jones – Modelo basado en la Reutilización
1991 Madhavji – Modelo de cambios Prism
1996 Gregor Kiczales - Orientación a Aspectos
1999 Ivar Jacobson, Grady Booch y James Rumbaugh - RUP
2001 – Desarrollo Ágil

2.1.1 Modelo en Cascada

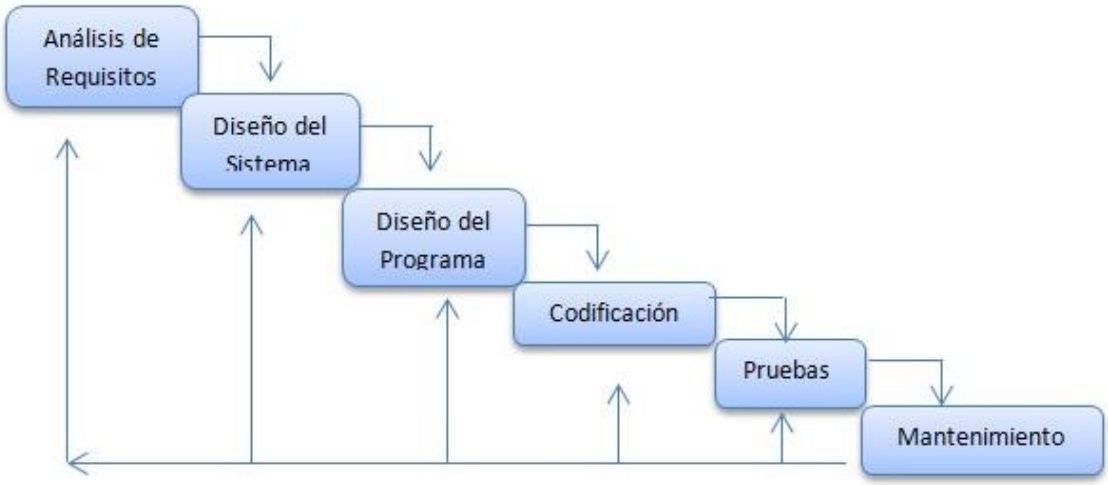
Modelo Prescriptivo

El conocido Modelo en Cascada nunca fue así denominado por su creador. Royce, en 1970, escribe un artículo [Royce 70] donde describe el modelo "non-working model" (modelo que no funciona) del cual se desprende luego el Modelo en Cascada debido a su proyección "secuencial lineal". La propuesta ideal de Royce consiste en una secuencia de etapas (de aquí se desprende lo secuencial) las cuales hacen imposible el desarrollo si se está en constante cambio, por tal motivo propone no retornar a etapas anteriores (de aquí el se desprende el concepto de lineal). Pero la realidad es otra y concibe que el mismo modelo pueda tener retroalimentación.

Este modelo considera a cada actividad del proceso como una fase autónoma que produce una salida o documento aprobado. La siguiente fase comienza cuando se ha finalizado con la anterior. Las fases se suceden en orden estrictamente secuencial.

Interesante pasaje del artículo original de Royce:

> "Winston is aware that software designs are subject to wide interpretation even after previous agreement. So he suggested to involve customers regularly. For that, a dedicated person (we call her the product owner (PO)) is representing a wider audience of stakeholders. But other users are also invited to give feedback on a regular basis during so called sprint reviews or show & tells."



4 Modelo de Proceso en Cascada de Royce

Ventajas:

- Fácil de administrar. El modelo da una buena visibilidad[1] del proceso, es decir, permite detectar fácilmente el avance en el desarrollo, debido a que cada actividad produce un artefacto, ya sea un documento, modelo o software.
- En cada fase están bien definidas las salidas a producir para avanzar a la siguiente etapa.
- El proceso de desarrollo es claro de entender por los clientes.
- La separación de análisis, diseño e implementación conduce a sistemas robustos[2], lo que facilita el cambio posterior.

Desventajas:

- Modelo inflexible: presenta dificultades para hacer cambios entre etapas.
- Este modelo tiene una visión estática de la ingeniería de requisitos, ignora la volatilidad natural de los requisitos y cómo repercute ésta en las etapas del desarrollo. No está preparado para responder a cambios en los requisitos pues se deberían rehacer las fases. Los cambios deben ser ignorados o deben realizarse "por fuera del proceso", lo que provoca que los documentos se vayan tornando obsoletos con el paso del tiempo.
- El congelamiento prematuro de los requisitos puede implicar que el sistema no haga lo que los usuarios desean.
- Los clientes y usuarios no participan en etapas posteriores a la especificación de requisitos sino hasta la prueba de aceptación. Por lo tanto, no hay un compromiso de los clientes y usuarios a lo largo de todo el ciclo de vida del software.
- Este modelo no trata al proceso de software como un proceso de resolución de problemas, pues presenta una visión de manufactura sobre el desarrollo de software, es decir, producir un artículo en particular y reproducirlo muchas veces. Este modelo no trata los avances y retrocesos normales hasta crear el producto final (el sistema de software).
- Errores y omisiones en los requisitos originales se descubren recién en las fases finales del ciclo.

2.1.2 Modelo Incremental

Modelo Prescriptivo

La propuesta del enfoque es diseñar sistemas que puedan entregarse por piezas.

- A partir de la evaluación se planea el siguiente incremento y así sucesivamente
- Es interactivo por naturaleza
- Es útil cuando el personal no es suficiente para la implementación completa
- En lugar de entrega del sistema en una sola entrega, el desarrollo y la entrega están fracturados bajo incrementos, con cada incremento que entrega parte de la funcionalidad requerida.
- Los requerimientos del usuario se priorizan y los requerimientos de prioridad más altos son incluidos en los incrementos tempranos.
- Hechos de incrementos tempranos como un prototipo, ayudan a obtener requisitos para los incrementos más tardíos.
- El usuario no tiene que esperar.
- Pueden aumentar el coste debido a las pruebas.

[1] *Visibilidad del proceso*: capacidad de poder detectar los avances en el desarrollo.

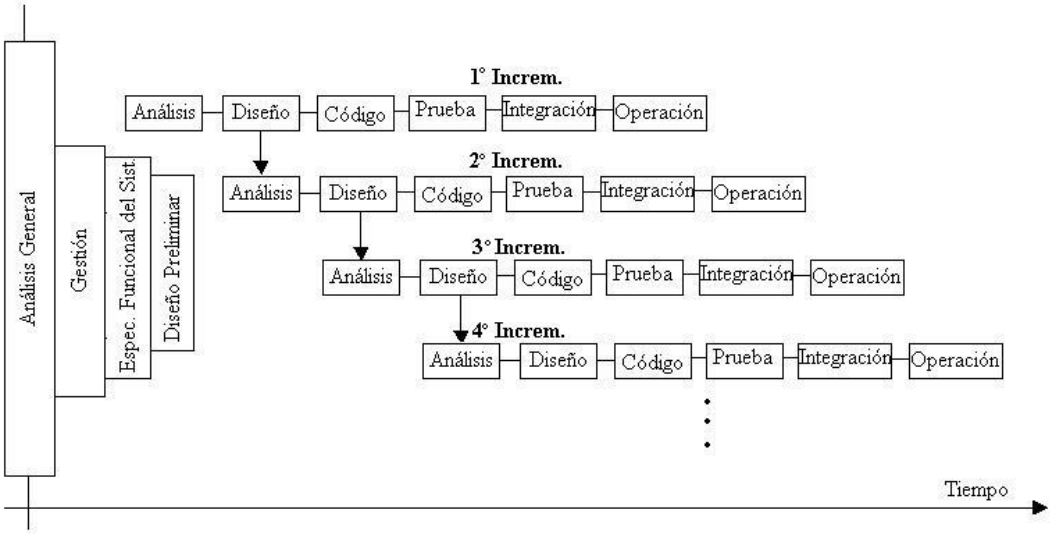
[2] *Robustez*: capacidad de un sistema para adaptarse al cambio.

- El desarrollo incremental es el proceso de construcción siempre incrementando subconjuntos de requerimientos del sistema.
- El modelo incremental presupone que el conjunto completo de requerimientos es conocido al comenzar
- Se evitan proyectos largos y se entrega "Algo de valor" a los usuarios con cierta frecuencia
- El usuario se involucra más
- Riesgos largos y complejos.
- Difícil de aplicar a sistemas transaccionales que tienden a ser integrados y a operar como un todo
- Requiere gestores experimentados
- Los errores en los requisitos se detectan tarde.

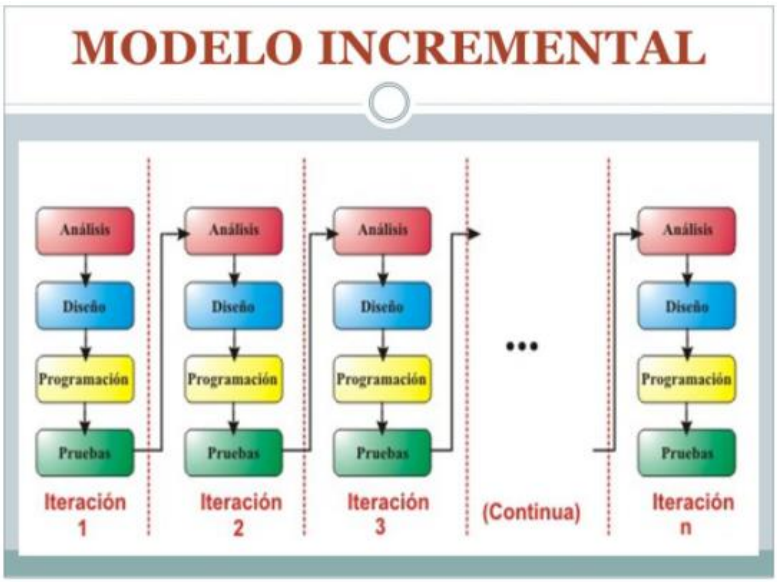
Bajo este modelo se entrega software "por partes funcionales más pequeñas", pero reutilizables, llamadas incrementos. En general cada incremento se construye sobre aquel que ya fue entregado.

Beneficios:

- Construir un sistema pequeño es siempre menos riesgoso que construir un sistema grande.
- Al ir desarrollando parte de las funcionalidades, es más fácil determinar si los requerimientos planeados para los niveles subsiguientes son correctos.
- Si un error importante es realizado, sólo la última iteración necesita ser descartada.
- Reduciendo el tiempo de desarrollo de un sistema (en este caso en incremento del sistema) decrecen las probabilidades que esos requerimientos de usuarios puedan cambiar durante el desarrollo.
- Si un error importante es realizado, el incremento previo puede ser usado. Los errores de desarrollo realizados en un incremento, pueden ser arreglados antes del comienzo del próximo incremento
- El resultado puede ser muy positivo



5 Modelo Incremental - 1



6 Modelo Incremental - 2

2.1.3 Modelo Iterativo / Espiral

Modelo Prescriptivo

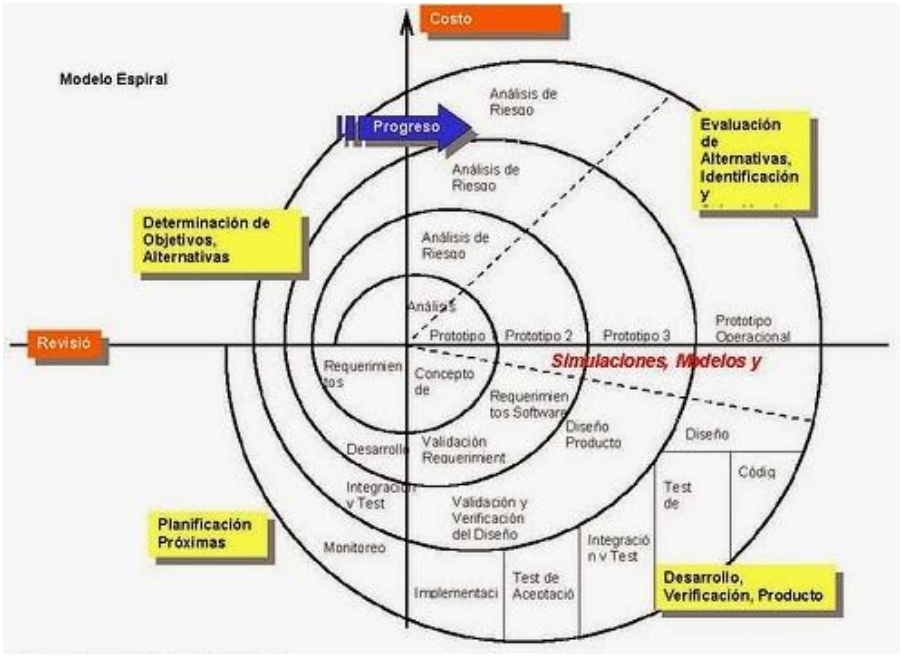
El modelo evolutivo en espiral es la experiencia realizada por Boehm [Boehm 88] durante varios años de utilizar el modelo en cascada para proyectos de gobierno. O sea, que este modelo resulta de un refinamiento exhaustivo del modelo en cascada al que se le agregó el desarrollo incremental.

Cada ciclo de la espiral comienza con la identificación de objetivos de la porción de producto que se va a elaborar, la implementación de la porción del producto (diseño, reuso, etc.) y sus restricciones (costo, tiempos, etc.).

Cada espira produce un producto software que puede ser una versión del software o un componente del mismo. Por lo tanto, el concepto de espiral se basa en la idea de volver a repetir todos los pasos por cada producto de software realizado. También puede darse que se realicen diferentes espirales en paralelo para desarrollar diferentes componentes o incrementos.

Es un modelo dirigido a los riesgos, y estos son los referidos a costos, tiempos, no comprensión de los requisitos para cada porción del software que se esté realizando. Este análisis de riesgos implica detectarlos, evaluarlos y prever caminos alternativos si es necesario.

La aplicación del modelo en cascada en la concepción del modelo en espiral:



7 Modelo Espiral

Ventajas:

- Incorpora muchas de las ventajas de los otros ciclos de vida
- Conjuga la naturaleza iterativa de los prototipos con los aspectos controlados y sistemáticos del modelo clásico
- El modelo en espiral puede adaptarse y aplicarse a lo largo de la vida del software de computadora.
- Como el software evoluciona a medida que progresa el proceso, el desarrollador y el cliente comprenden y reaccionan mejor ante riesgos en cada uno de los nivele evolutivos.
- El modelo en espiral permite a quien lo desarrolla aplicar el enfoque de construcción de prototipos en cualquier etapa de evolución del producto.
- El modelo en espiral demanda una consideración directa de los riesgos técnicos en todas las etapas del proyecto y si se aplica adecuadamente debe reducir los riesgos antes de que se conviertan en problemas.
- Proporciona el potencial para el desarrollo rápido de versiones incrementales
- Permite aplicar el enfoque de construcción de prototipos en cualquier momento para reducir riesgos.

Desventajas:

- Solo resulta aplicable para proyectos de gran tamaño
- Supone una carga de trabajo adicional, no presente en otros ciclos de vida
- Requiere una considerable habilidad para la evaluación y resolución del riesgo, y se basa en esta habilidad para el éxito
- Si un riesgo importante no es descubierto y gestionado, indudablemente surgirán problemas

- Es bastante complicado de realizar y su complejidad puede incrementarse hasta hacerlo impracticable
- El modelo no se ha utilizado tanto como otros, por lo que tendrán que pasar años antes de que determine con certeza la eficacia de este modelo

2.1.4 Modelo de Prototipado

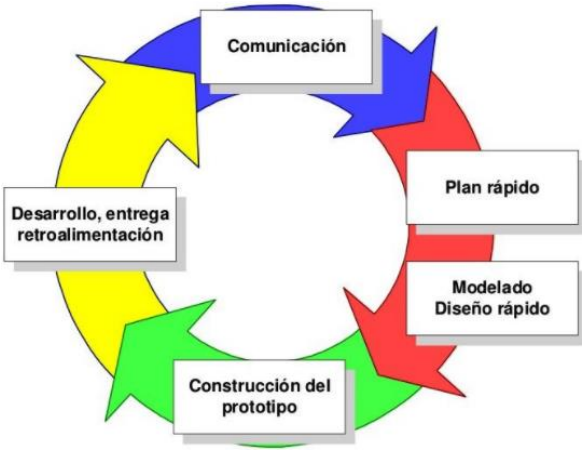
Modelo Prescriptivo

La estrategia de Prototipado [Floyd 84] consiste en construir un mecanismo para generar software que sea evaluado por el cliente en conjunto con el programador. Es importante recalcar que para Floyd esta estrategia no es en sí misma un método para desarrollar sistemas de software, más bien debe ser considerado como un procedimiento dentro de la construcción del software. La construcción de un prototipo suele ser muy adecuado al comienzo de la etapa de análisis, ya que el prototipo es el único medio a través del cual se pueden obtener de una manera más eficaz los requisitos.

Según Floyd, existen tres enfoques para utilizar prototipos: el prototipo exploratorio, el prototipo experimental y el prototipo evolutivo.

El prototipo exploratorio se utiliza en etapas muy tempranas del desarrollo con el objetivo de clarificar o elicitar requisitos del software. El prototipo experimental sirve para simular aspectos o partes de un sistema de software para evaluar aspectos técnicos desconocidos. Existen dos tipos de Prototipo Evolutivo:

1) Desarrollo incremental ("slowly growing systems"). El sistema evoluciona gradualmente en incrementos parciales. No hay cambios en el diseño. Las necesidades del usuario son conocidas al principio y se realiza un diseño completo que se realiza en incrementos con un prototipo.
2) Desarrollo evolutivo. Mira al desarrollo como una secuencia de ciclos: re-diseño, re-implementación, re-evaluación. Producción del software en un ambiente dinámico y de cambios. Los requisitos no se terminan de conocer nunca y se van descubriendo constantemente. El sistema evoluciona de manera continua, por lo tanto, nunca se distingue una etapa de mantenimiento.



8 Modelo Prototipado

2.2 Metodología RUP – Proceso Unificado

Ivar Jacobson, Grady Booch y James Rumbaugh (1999)

El Proceso Unificado propuesto por IBM, incluye prácticas claves y aspectos relacionados a la planeación estratégica y administración de riesgos; y actualmente guían de forma natural el proceso de desarrollo de software complejo por lo que ha sido considerado como un estándar el desarrollo de software en las empresas.

El proceso unificado conocido como RUP, es un modelo de software que permite el desarrollo de software a gran escala, mediante un proceso continuo de pruebas y retroalimentación, garantizando el cumplimiento de ciertos estándares de calidad. Aunque con el inconveniente de generar mayor complejidad en los controles de administración del mismo. Sin embargo, los beneficios obtenidos recompensan el esfuerzo invertido en este aspecto.

El Proceso Unificado Racional (RUP, por las siglas de Rational Unified Process) (Krutchen, 2003) es un ejemplo de un modelo de proceso moderno que se derivó del trabajo sobre el UML y el proceso asociado de desarrollo de software unificado (Rumbaugh et al., 1999; Arlow y Neustadt, 2005). Aquí se incluye una descripción, pues es un buen ejemplo de un modelo de **proceso híbrido**. Conjunta elementos de todos los modelos de proceso genéricos, ilustra la buena práctica en especificación y diseño, y apoya la creación de prototipos y entrega incremental. RUP reconoce que los modelos de proceso convencionales presentan una sola visión del proceso. En contraste, RUP por lo general se describe desde tres perspectivas:

1. Una perspectiva dinámica que muestra las fases del modelo a través del tiempo.
2. Una perspectiva estática que presenta las actividades del proceso que se establecen.
3. Una perspectiva práctica que sugiere buenas prácticas a usar durante el proceso.

La mayoría de las descripciones de RUP buscan combinar las perspectivas estática y dinámica en un solo diagrama (Krutchen, 2003).

RUP es un modelo en fases que identifica cuatro fases discretas en el proceso de software. Sin embargo, a diferencia del modelo en cascada, donde las fases se igualan con actividades del proceso, las fases en el RUP están más estrechamente vinculadas con la empresa que con las preocupaciones técnicas.

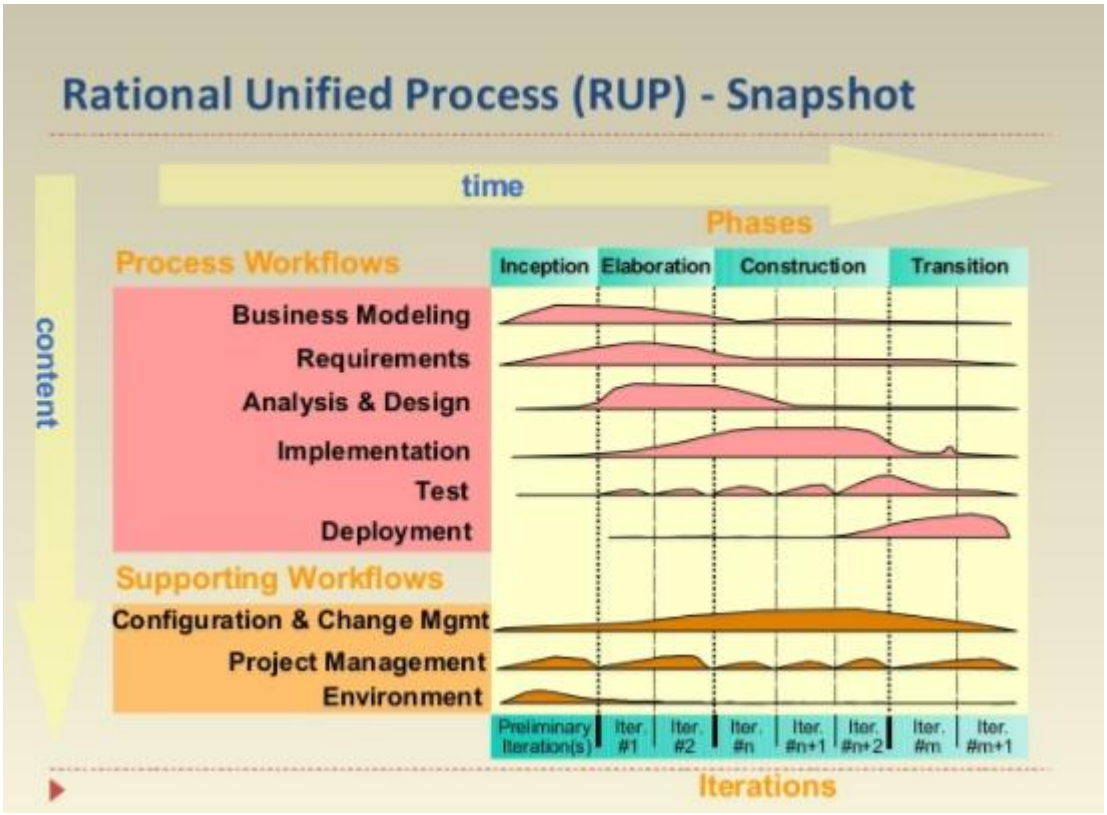


Ilustración 9 RUP

2.3 Las fases del RUP

1. **Concepción** La meta de la fase de concepción es establecer un caso empresarial para el sistema. Deben identificarse todas las entidades externas (personas y sistemas) que interactuarán con el sistema y definirán dichas interacciones. Luego se usa esta información para valorar la aportación del sistema hacia la empresa. Si esta aportación es menor, entonces el proyecto puede cancelarse después de esta fase.

2. **Elaboración** Las metas de la fase de elaboración consisten en desarrollar la comprensión del problema de dominio, establecer un marco conceptual arquitectónico para el sistema, diseñar el plan del proyecto e identificar los riesgos clave del proyecto. Al completar esta fase, debe tenerse un modelo de requerimientos para el sistema, que podría ser una serie de casos de uso del UML, una descripción arquitectónica y un plan de desarrollo para el software.

3. **Construcción** La fase de construcción incluye diseño, programación y pruebas del sistema. Partes del sistema se desarrollan en paralelo y se integran durante esta fase. Al completar ésta, debe tenerse un sistema de software funcionando y la documentación relacionada y lista para entregarse al usuario.

4. Transición La fase final del RUP se interesa por el cambio del sistema desde la comunidad de desarrollo hacia la comunidad de usuarios, y por ponerlo a funcionar en un ambiente real. Esto es algo ignorado en la mayoría de los modelos de proceso de software, aunque, en efecto, es una actividad costosa y en ocasiones problemática. En el complemento de esta fase se debe tener un sistema de software documentado que funcione correctamente en su entorno operacional. La iteración con RUP se apoya en dos formas. Cada fase puede presentarse en una forma iterativa, con los resultados desarrollados incrementalmente. Además, todo el conjunto de fases puede expresarse de manera incremental. La visión

estática del RUP se enfoca en las actividades que tienen lugar durante el proceso de desarrollo. Se les llama flujos de trabajo en la descripción RUP. En el proceso se identifican seis flujos de trabajo de proceso centrales y tres flujos de trabajo de apoyo centrales. El RUP se diseñó en conjunto con el UML, de manera que la descripción del flujo de trabajo se orienta sobre modelos UML asociados, como modelos de secuencia, modelos de objeto, etcétera. La ventaja en la presentación de las visiones dinámica y estática radica en que las fases del proceso de desarrollo no están asociadas con flujos de trabajo específicos. En principio, al menos, todos los flujos de trabajo RUP pueden estar activos en la totalidad de las etapas del proceso. En las fases iniciales del proceso, es probable que se use mayor esfuerzo en los flujos de trabajo como modelado del negocio y requerimientos y, en fases posteriores, en las pruebas y el despliegue.

El enfoque práctico de RUP describe las buenas prácticas de ingeniería de software que se recomiendan para su uso en el desarrollo de sistemas.

2.3.1 Las seis mejores prácticas fundamentales recomendadas

i) **Desarrollo de software de manera iterativa** Incrementar el plan del sistema con base en las prioridades del cliente, y desarrollar oportunamente las características del sistema de mayor prioridad en el proceso de desarrollo.
ii) **Gestión de requerimientos** Documentar de manera explícita los requerimientos del cliente y seguir la huella de los cambios a dichos requerimientos. Analizar el efecto de los cambios sobre el sistema antes de aceptarlos.
iii) **Usar arquitecturas basadas en componentes** Estructurar la arquitectura del sistema en componentes, como se estudió anteriormente en este capítulo.
iv) **Software modelado visualmente** Usar modelos UML gráficos para elaborar representaciones de software estáticas y dinámicas.
v) **Verificar la calidad del software** Garantizar que el software cumpla con los estándares de calidad de la organización.
vi) **Controlar los cambios al software** Gestionar los cambios al software con un sistema de administración del cambio, así como con procedimientos y herramientas de administración de la configuración.

RUP no es un proceso adecuado para todos los tipos de desarrollo, por ejemplo, para desarrollo de software embebido. Sin embargo, sí representa un enfoque que potencialmente combina los tres modelos de proceso genéricos, cascada, iterativo, incremental. Las innovaciones más importantes en el RUP son la separación de fases y flujos de trabajo, y el reconocimiento de que el despliegue del software en un entorno del usuario forma parte del proceso. Las fases son dinámicas y tienen metas. Los flujos de trabajo son estáticos y son actividades técnicas que no se asocian con una sola fase, sino que pueden usarse a lo largo del desarrollo para lograr las metas de cada fase.

Flujo de trabajo	Descripción
Modelado del negocio	Se modelan los procesos de negocios utilizando casos de uso de la empresa.
Requerimientos	Se identifican los actores que interactúan con el sistema y se desarrollan casos de uso para modelar los requerimientos del sistema.
Análisis y diseño	Se crea y documenta un modelo de diseño utilizando modelos arquitectónicos, de componentes, de objetos y de secuencias.
Implementación	Se implementan y estructuran los componentes del sistema en subsistemas de implementación. La generación automática de código a partir de modelos de diseño ayuda a acelerar este proceso.
Pruebas	Las pruebas son un proceso iterativo que se realiza en conjunto con la implementación. Las pruebas del sistema siguen al completar la implementación.
Despliegue	Se crea la liberación de un producto, se distribuye a los usuarios y se instala en su lugar de trabajo.
Administración de la configuración y del cambio	Este flujo de trabajo de apoyo gestiona los cambios al sistema (véase el capítulo 25).
Administración del proyecto	Este flujo de trabajo de apoyo gestiona el desarrollo del sistema (véase los capítulos 22 y 23).
Entorno	Este flujo de trabajo pone a disposición del equipo de desarrollo de software, las herramientas adecuadas de software.

2.4 Conclusión

El problema crucial que atraviesan los métodos de la Ingeniería de Software es tratar con requisitos siempre cambiantes. La evolución de los errores y la evolución de los requisitos son dos características del desarrollo de software estrechamente vinculadas. Pues las causas de los cambios en el software no son necesariamente debido a cambios en los requisitos por cambios en el universo de discurso, sino muchas veces por reparación de errores o por una mejora en la comprensión de los requisitos.

Belady y Lehman construyeron un modelo que representa la evolución de los errores en un sistema de software a lo largo del tiempo, en función de las versiones del software que se generan para corregir errores en los programas y actualizar / mejorar la funcionalidad del mismo. El punto mínimo de la curva representa la versión de software con la menor cantidad de errores, pasado el cual la curva asciende rápidamente, pues cualquier cambio a realizar en el software se torna cada vez más dificultoso debido a un paulatino y constante desmejoramiento en su arquitectura, lo cual a su vez facilita la gestación de nuevos errores. Los métodos de la Ingeniería de Software deberían tender a que la parte ascendente de la curva de Belady- Lehman posterior al punto mínimo sea lo más suave posible postergando el aumento de la entropía del sistema.

El resto de los modelos están representados por funciones que se acercan más a la función de necesidades que el enfoque convencional. Es decir, estos modelos disminuyen:

i) el tiempo para satisfacer una necesidad desde su ocurrencia, y
ii) la distancia entre las funcionalidades incluidas en el sistema en un momento dado y el total de necesidades requeridas hasta ese momento.

Los modelos desarrollados con posterioridad al modelo convencional han mejorado de distintas formas los inconvenientes presentados por el modelo de cascada, cuyas consecuencias se manifestaron en los fracasos paradigmáticos de la crisis del software y posteriores. Pero, además, se desprende que la evolución de los requisitos es prácticamente ignorada durante el desarrollo del software por cualquiera de los modelos. Es por ello que nuevos enfoques en el desarrollo de software deben apuntar a obtener resultados parciales del sistema de software que sean más adaptables a la continua evolución de los requisitos. Es decir, deben producir versiones de sistemas de software que atiendan la totalidad de las necesidades a un momento dado y que sean altamente flexibles para satisfacer rápidamente los próximos nuevos requisitos.

Los modelos de prototipado, operacional y transformación formal surgieron como solución al problema planteado por el modelo de cascada referido a la poca participación del usuario en el desarrollo. Por lo tanto, los tres modelos proponen la creación de un artefacto (prototipo, especificación operacional, especificación formal) tempranamente en el ciclo de vida, que pueda ser usado para comprender y validar los requisitos.

Cabe destacar por otro lado que los modelos que más soportan la evolución de los requisitos son aquellos que presentan procesos iterativos, como el modelo evolutivo, el incremental y el espiral, donde la especificación de los requisitos acompaña la implementación del software. Debido a lo cual, no se cuenta con un documento SRS completo final que sirva como contrato para el desarrollo del software. Lamentablemente, esto es una condición mandataria en muchas organizaciones, principalmente gubernamentales, cuando el proveedor de software pertenece a una organización diferente de la del cliente y existe una relación contractual, formal o informal entre ellas.

Comparativamente, los modelos de prototipado, operacional y espiral, apuntan más a la validación de los requisitos, mediante la producción de un artefacto que les permita a los usuarios experimentar tempranamente.

En resumen, no existe un modelo de proceso ideal de desarrollo de software aplicable a cualquier tipo de sistema, en cualquier tipo de organización, y que garantice el mejor producto a un costo acorde. De ahí que surja una diversidad de métodos, que se basan en algunos de estos modelos o combinaciones de ellos, y que atienden algunos de los subprocesos involucrados en el desarrollo de software. Muchas organizaciones usan una combinación del modelo incremental con el modelo iterativo (denominada prácticas IID13), donde en cada versión se agregan funcionalidades y se mejoran funcionalidades existentes en la versión actual. Un ejemplo de método basado en desarrollo iterativo e incremental ampliamente difundido es el Rational Unified Process. Recientemente se establecieron los “métodos ágiles” cuando en febrero del 2001 se reunieron expertos en DSDM, XP, ADS, FDD y otros, y formaron la “Agile Alliance”.

Estos métodos ágiles, basados en prácticas IID, sí tienen en cuenta la evolución de los requisitos, ya que en ellos todo evoluciona, pero no presentan un proceso dedicado a la definición de requisitos, sino que éste es más bien informal.

Se debe tener presente que los desarrollos iterativos e incrementales, a pesar de su popularidad actual como piedra basal de los métodos ágiles, y de su puesta en conocimiento a fines de los 80 a través de reportes de resultados de proyectos y artículos científicos, realmente ya habían sido puestos en práctica y con éxito desde la década del 70 en pleno auge del desarrollo en cascada e inclusive se remontan a proyectos aislados desde fines de los 50.

3 Conceptos de UML

El lenguaje UML comenzó a gestarse en octubre de 1994, cuando Rumbaugh se unió a la compañía Rational Software fundada por Booch (dos reputados investigadores en el área de metodología del software). El objetivo de ambos era unificar dos métodos que habían desarrollado: el método Booch y el OMT (Object Modelling Tool). El primer borrador apareció en octubre de 1995. En esa misma época otro reputado investigador, Jacobson, se unió a Rational y se incluyeron ideas suyas sobre casos de uso. Estas tres personas son conocidas como los "tres amigos". Además, este lenguaje se abrió a la colaboración de otras empresas para que aportaran sus ideas. Todas estas colaboraciones condujeron a la definición de la primera versión de UML. El lenguaje ha ganado un significante soporte de la industria de varias organizaciones vía el consorcio de socios de UML y ha sido presentado al Object Management Group (OMG) y aprobado por éste como un estándar (noviembre 17 de 1997)

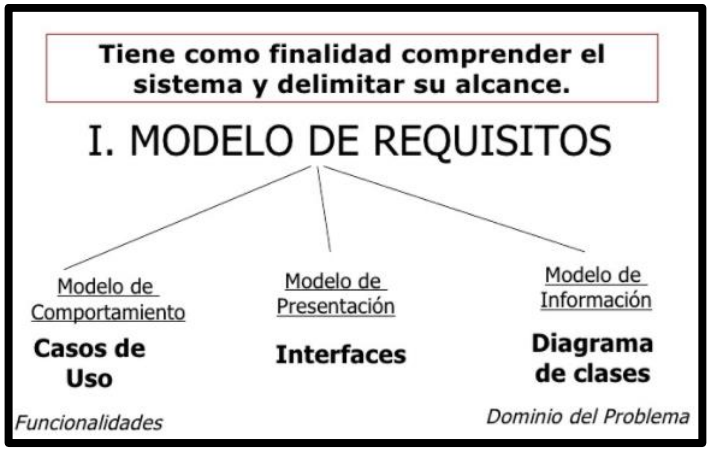
El UML puede usarse para visualizar, especificar, construir y documentar los artefactos de un sistema de software intensivo" [Boo 05]. En otras palabras, tal como los arquitectos de edificios crean planos para que los use una compañía constructora, los arquitectos de software crean diagramas de UML para ayudar a los desarrolladores de software a construir el software. Si usted entiende el vocabulario del UML (los elementos pictóricos de los diagramas y su significado) puede comprender y especificar con mucha más facilidad un sistema, y explicar su diseño a otros.

El UML fusionó algunas notaciones de modelado que competían entre sí y que se usaban en la industria del software en la época. En 1997, UML 1.0 se envió al Object Management Group, un consorcio sin fines de lucro involucrado en especificaciones de mantenimiento para su empleo en la industria de la computación. El UML 1.0 se revisó y dio como resultado la adopción del UML 1.1 ese mismo año. El estándar actual es UML 2.0 y ahora es un estándar ISO.

3.1 Análisis

Conocer los requisitos de un producto de software es la primera etapa para crearlo. Mientras que los clientes piensan que ellos saben lo que el software tiene que hacer, se requiere de habilidad y experiencia en la ingeniería de software para reconocer requisitos incompletos, ambiguos o contradictorios. El resultado del análisis de requisitos con el cliente se plasma en el documento ERS, *Especificación de Requerimientos del Sistema*. Asimismo, se define un diagrama de Entidad/Relación, en el que se plasman las principales entidades que participarán en el desarrollo del software. La captura, análisis y especificación de requisitos (incluso pruebas de ellos), es una parte crucial; de esta etapa depende en gran medida el logro de los objetivos finales. Se han ideado modelos y diversos procesos de trabajo para estos fines. Aunque aún no está formalizada, ya se habla de la Ingeniería de Requisitos. La IEEE Std. 830-1998 normaliza la creación de las Especificaciones de Requisitos Software (Software Requirements Specification).

3.1.1 Modelo de Requerimientos



10 Modelo de Requisitos

Durante esta etapa se obtienen diferentes modelos que ayudan a determinar los requisitos del software. Utilizando diferentes diagramas de UML y de una manera sencilla es posible entender, analizar y comunicar las necesidades de los usuarios, sus actividades y objetivos al realizar una tarea determinada.

Existen diferentes técnicas o estrategias que se pueden utilizar para tal fin, la más utilizada es la de casos de uso, otra no tan utilizada es la basada en escenarios.

Un modelo de requisitos termina siendo un conjunto de estos diagramas, cada uno de los cuales se centra en un aspecto diferente de las necesidades de los usuarios.

Un modelo de requisitos le ayuda a:

- Centrarse en el comportamiento externo del sistema, independientemente de su diseño interno.
- Describir las necesidades de los usuarios y otras partes interesadas con mucha menos ambigüedad que con el lenguaje natural.
- Definir un glosario de términos coherente que puedan usar los usuarios, los desarrolladores y los evaluadores.
- Reducir los vacíos y las incoherencias de los requisitos.
- Reducir el trabajo necesario para responder a los cambios de los requisitos.
- Planear el orden en el que se van a desarrollar las características.
- Usar los modelos como base para las pruebas del sistema, estableciendo una relación inequívoca entre las pruebas y los requisitos. Si cambian los requisitos, esta relación le ayudará a actualizar las pruebas correctamente. Esto garantiza que el sistema cumple los requisitos nuevos.

Un modelo de requisitos proporciona el máximo beneficio si se lo utiliza como negociación con los usuarios o Stakeholders. No es necesario completarlo detalladamente antes de escribir el código. El modelo es un método eficaz para resumir los resultados de estas conversaciones. Para obtener más información, consulta Usar modelos en el proceso de desarrollo.

Nota:

En estos temas, "sistema" hace referencia al sistema o la aplicación que está desarrollando. Podría ser una colección grande de muchos componentes de hardware y software, una sola aplicación o un componente de software incluido en un sistema de mayor tamaño. En cada caso, el modelo de requisitos describe el comportamiento que es visible desde fuera del sistema, ya sea a través de una API o interfaz de usuario.

Tareas comunes

Puede crear varias vistas diferentes de los requisitos de los usuarios. Cada vista proporciona un tipo determinado de información. Al crear estas vistas, es mejor pasar con frecuencia de una a otra. Puede empezar desde cualquier vista.

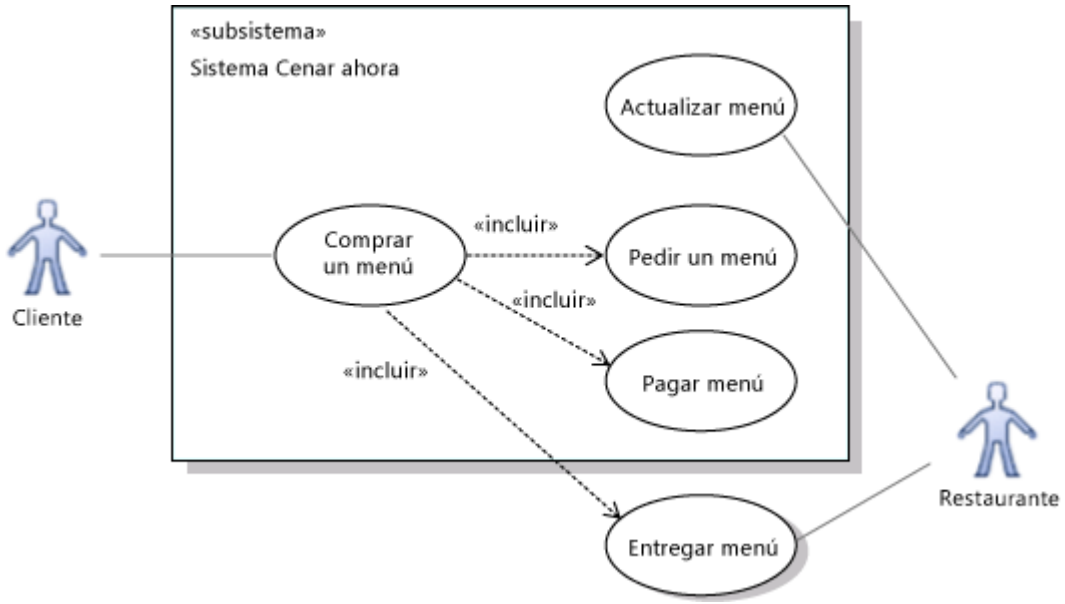
Diagrama o Documento	Qué se describe en un modelo de requisitos	Sección
Diagrama de casos de uso	Quién usa el sistema y para qué lo usa.	Describir cómo se usa el sistema
Diagrama de clases conceptuales	Glosario de los tipos que se usan para describir los requisitos; los tipos visibles en la interfaz del sistema.	Definición de términos que se usan para describir los requisitos
Diagrama de actividades	Flujo de trabajo e información que existe entre las actividades realizadas por los usuarios y el sistema o partes del sistema.	Mostrar el flujo de trabajo que existe entre los usuarios y el sistema
Diagrama de secuencia	Secuencia de interacciones que existen entre los usuarios y el sistema o partes del sistema. Vista alternativa al diagrama de actividades.	Mostrar las interacciones que existen entre los usuarios y el sistema
Otros documentos o elementos de trabajo	Criterios de rendimiento, seguridad, facilidad de uso y confiabilidad.	Describir los requisitos de calidad de servicio
Otros documentos o elementos de trabajo	Restricciones y reglas no específicas para un determinado caso de uso	Muestran las reglas de negocio

3.1.2 ¿Cómo se empieza?

Por ejemplo, un sistema de venta de comida en línea debe permitir a los clientes elegir platos de un menú, y a los restaurantes correspondientes actualizar dicho menú. Esto se puede resumir en un diagrama de casos de uso:

11 Venta de Comida en Línea

También puede mostrar cómo un caso de uso se compone de casos más pequeños. Por ejemplo, pedir un menú forma parte de la compra de comida, que también incluye el pago y el envío:



12 Venta de Comida en Línea 2

También puede mostrar los casos de uso que se incluyen en el ámbito del sistema que está desarrollando. Por ejemplo, el sistema de la ilustración no interviene en el caso de uso de reparto de comida. Esto ayuda a establecer el contexto para el trabajo de desarrollo.

(En un diagrama de casos de uso, se pueden usar los contenedores del subsistema para representar el sistema o sus componentes).

También ayuda al equipo a analizar lo que se incluirá en las versiones posteriores. Por ejemplo, puede discutir si, en la versión inicial del sistema, el pago de la comida se situará directamente entre el restaurante y el cliente, en lugar de pasar por el sistema. En ese caso, podría sacar el pago de la comida fuera del rectángulo del sistema de Dinner Now.

Un diagrama de casos de uso solo proporciona un resumen de los casos de uso. Para proporcionar descripciones más detalladas, puede vincular los casos de uso del diagrama a documentos independientes y a otros diagramas.

Dibujar un diagrama de casos de uso ayuda a su equipo a:

- Centrarse en lo que los usuarios esperan hacer con el sistema, sin necesidad de detenerse en los detalles de la implementación.
- Analizar el ámbito del sistema o de versiones específicas del sistema.

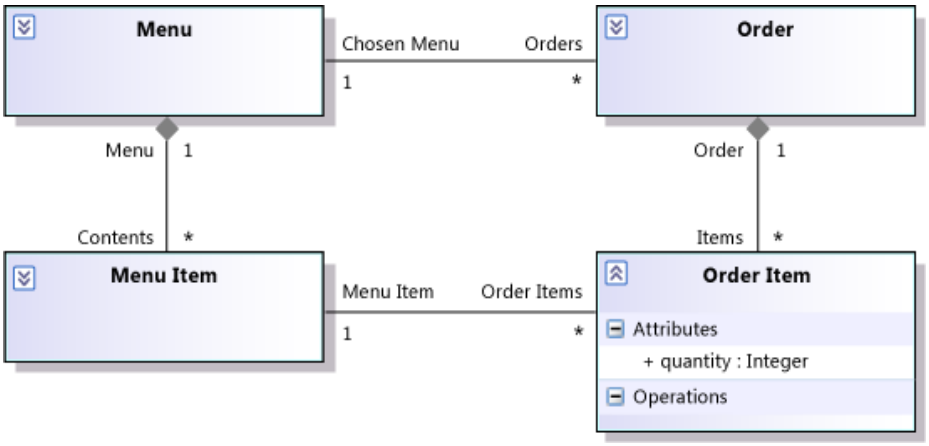
3.1.2.1 Definición de términos que se usan para describir los requisitos

Puede usar diagramas de clases de UML para desarrollar un vocabulario coherente de los **conceptos de negocio** usados para los siguientes fines:

- Analizar el negocio en el que funciona el sistema con los propios usuarios.
- Describir las necesidades de los usuarios como, por ejemplo, en las descripciones de los casos de uso, las reglas de negocios y los casos de usuario.
- Los tipos de información que se intercambian en la API del sistema o a través de la interfaz de usuario.
- Descripciones de las pruebas del sistema o de aceptación.

Cuando se usan para este fin, el contenido de un diagrama de clases de UML se denomina "**diagrama de clases conceptuales**". (También conocido como *modelo de dominio o modelo de clases de análisis).*

En un diagrama de **clases conceptuales**, solo se muestran las clases necesarias en las descripciones de los requisitos; no se muestra ningún detalle del diseño interno del sistema. Normalmente no se muestran operaciones ni interfaces en las clases conceptuales.
Por ejemplo, puede dibujar las siguientes clases conceptuales para el sistema de Dinner Now:



13 Diagrama de Clases de Negocio

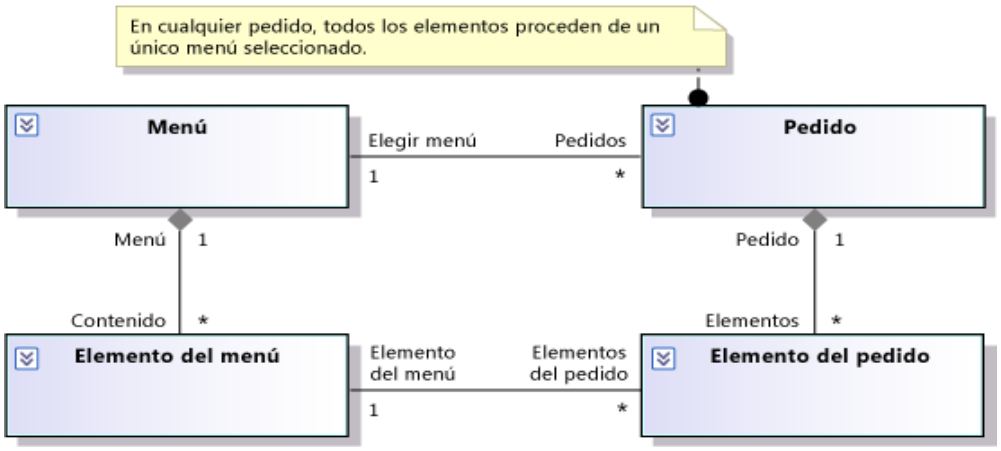
Un diagrama de clases conceptuales proporciona el vocabulario que se usa en el modelo de requisitos. Por ejemplo, en la descripción detallada del caso de uso para pedir un menú, puede escribir:
El cliente elige un ***menú*** para iniciar un ***pedido*** y, después, crea ***elementos*** *del pedido* en el ***pedido*** mediante la selección de *elementos de menú* en el *menú*.
Observe cómo los términos usados en esta descripción son los nombres de las clases del modelo. El diagrama elimina las ambigüedades de las relaciones entre esas clases. Por ejemplo, *muestra claramente que cada pedido está asociado a un solo menú*.
Los malentendidos sobre los requisitos de los usuarios suelen deberse a malentendidos relacionados con el significado exacto de las palabras. Por ejemplo, la mayoría de los restaurantes compartirán la visión de los términos "menú" y "pedido", pero la diferencia entre un elemento de un pedido y un elemento de un menú es menos clara. Cuando se tratan los requisitos con las partes interesadas del negocio, es importante exponer estas diferencias. **El diagrama de clases es una herramienta útil para ayudar a aclarar los términos y sus relaciones**.
El modelo de clases conceptuales puede formar el vocabulario básico con el que describir la lógica de negocios de su sistema. Pero las clases del software suelen ser mucho más complejas que el modelo conceptual, ya que la implementación debe tener en cuenta aspectos como el rendimiento, la distribución, la flexibilidad y otros factores. Con frecuencia, en un sistema se encuentran varias implementaciones diferentes de una clase conceptual.
Por ejemplo, los pedidos pueden representarse en XML, SQL, HTML y C# en diferentes partes del sistema y en distintas interfaces entre las partes. La asociación entre un pedido y un menú se puede representar de muchas formas diferentes, por ejemplo, como referencias en código de C#, relaciones en una base de datos o identificadores con referencias cruzadas en XML. Pero, a pesar de estas variaciones, el modelo conceptual

proporciona información importante que es verdadera en todas las partes del software. El diagrama de clases del ejemplo nos indica que, en cada implementación, habrá un único menú asociado a cada pedido.
Un diagrama de clases de dominio ayuda al equipo de desarrollo a:

- Definir y estandarizar los términos básicos que se usan en el análisis de las necesidades de los usuarios.
- Aclarar las relaciones entre dichos términos.

Mostrar reglas de negocio

Una regla de negocio es un requisito que no está asociado a ningún caso de uso determinado y que se debe observar en todo el sistema.
Muchas reglas de negocio son restricciones en las relaciones entre las clases conceptuales. Puede escribir estas *reglas de negocio estáticas* como comentarios asociados a las clases pertinentes en un diagrama de clases conceptuales. Por ejemplo:



14 Diagrama de Clases de Negocio 2

Las *reglas de negocio dinámicas* restringen las secuencias de eventos permitidas. Por ejemplo, puede usar un diagrama de secuencia o actividades para mostrar que un usuario debe iniciar sesión antes de realizar otras operaciones en el sistema.
Pero muchas reglas dinámicas se pueden aplicar de una forma más eficaz y genérica al reemplazarlas por reglas estáticas. Por ejemplo, puede agregar un atributo booleano "Logged In" a una clase en el modelo de clases conceptuales. Agregaría "Logged In" como condición posterior del caso de uso de inicio de sesión y lo agregaría como condición previa de la mayoría de los demás casos de uso. Este enfoque le permite evitar definir todas las combinaciones de secuencias de eventos posibles. También es más flexible cuando necesita agregar nuevos casos de uso al modelo.
Observe que la opción es sobre la definición de los requisitos, y que esto es independiente de la implementación de los requisitos en el código del programa.

3.1.2.2 Descripción de los requisitos de calidad del servicio

Existen varias categorías de requisito de calidad de servicio. Entre esos tipos se incluyen los siguientes:

- Rendimiento
- Seguridad

- Facilidad de uso
- Confiabilidad
- Solidez

Puede incluir algunos de estos requisitos en las descripciones de casos de uso concretos. Otros requisitos no son específicos de los casos de uso y resulta más eficaz incluirlos en un documento independiente. Cuando sea posible, resulta útil ajustarse al vocabulario definido en el modelo de requisitos. En el ejemplo siguiente, observe que las principales palabras que se usan en el requisito son los títulos de los actores, los casos de uso y las clases de las ilustraciones anteriores:
Si un restaurante elimina un elemento del menú mientras un cliente pide un menú, cualquier elemento del pedido que haga referencia a ese elemento del menú se mostrará de color rojo.

3.1.2.3 Diagrama de Actividad para mostrar el flujo de trabajo entre los distintos casos de uso.

Puede usar un diagrama de actividades para mostrar el flujo de trabajo existente entre los distintos casos de uso. A menudo, resulta útil empezar un modelo de requisitos con el dibujo de un diagrama de actividades que muestre las principales tareas que realizan los usuarios, tanto en el sistema como fuera de él.
Por ejemplo:

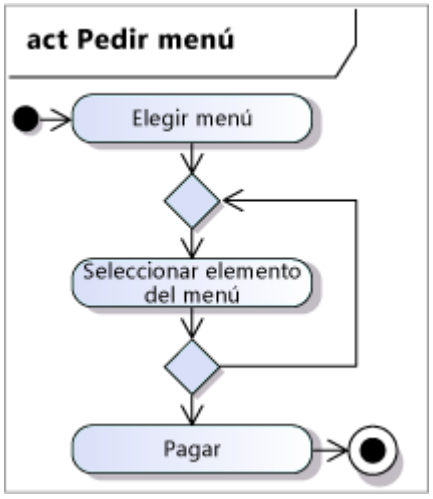
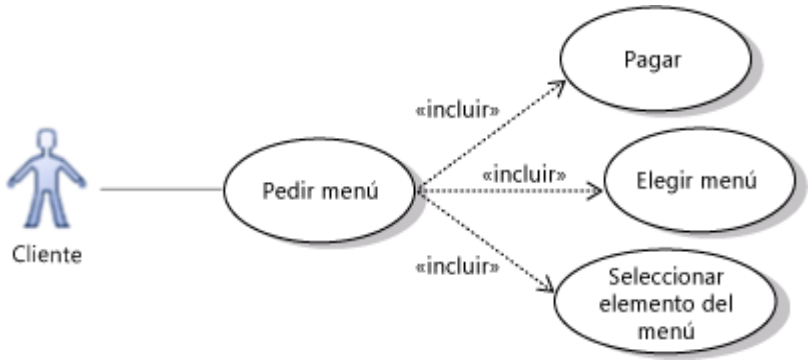


Ilustración 15 Diagrama de Actividad - Pedir Menú

Puede dibujar diagramas de casos de uso y diagramas de actividades para mostrar distintas vistas de la misma información. El diagrama de casos de uso es más eficaz a la hora de mostrar el anidamiento de las acciones más pequeñas dentro de una actividad mayor, pero no muestra el flujo de trabajo. Por ejemplo:

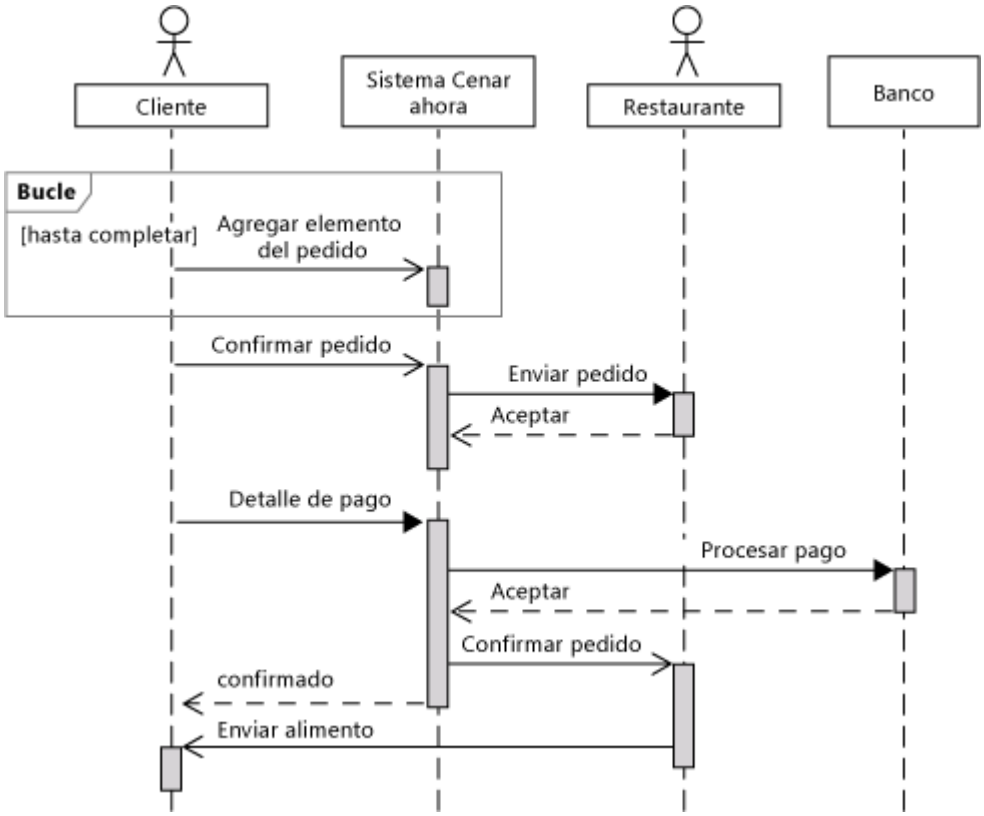


16 Casos de Uso

Observe que también puede usar diagramas de actividades para describir los algoritmos en el software, pero cuando use los diagramas para el proceso de negocio, debe centrarse en las acciones que son visibles fuera del sistema.

3.1.2.4 Interacciones entre los Usuarios y el Sistema

Puede usar un diagrama de secuencia para mostrar el intercambio de mensajes entre el sistema y los actores externos, o bien entre las partes del sistema. Esto proporciona una vista de los pasos de un caso de uso que muestra claramente la secuencia de interacciones. Los diagramas de secuencia resultan especialmente útiles cuando hay varias partes que interactúan en un caso de uso y también cuando el sistema tiene una API.
Por ejemplo:



17 Diagrama de Secuencia del Sistema

Una ventaja de los diagramas de secuencia es que resulta fácil ver qué mensajes entran en el sistema que se está desarrollando. Para diseñar el sistema, puede reemplazar la única línea de vida del sistema por una línea de vida diferente para cada uno de sus componentes y, después, mostrar las interacciones entre ellos en respuesta a cada mensaje entrante.

3.1.2.5 El Modelo de requisitos para reducir ambigüedades

La creación de un modelo suele reducir significativamente las incoherencias y las ambigüedades de los requisitos de los usuarios. Las diferentes partes interesadas suelen tener concepciones distintas del mundo empresarial en el que funciona el sistema. Por lo tanto, la primera tarea consiste en resolver estas diferencias entre los clientes.

Comprobará que muchas preguntas sobre el ámbito empresarial surgen de forma natural durante la creación de un modelo. Al formular estas preguntas a los usuarios, reducirá la necesidad de realizar cambios en una fase posterior del proyecto. Algunas preguntas específicas que puede hacerse al principio y, después, preguntar a las partes interesadas del negocio si no tiene clara la respuesta:

- Para cada clase del modelo de requisitos, pregunte: ¿Qué caso de uso crea instancias de esta clase? Por ejemplo, en un servicio de pedidos de comida en línea, puede preguntar: ¿Qué caso de uso crea instancias de la clase de menú del restaurante? Esto daría lugar a un análisis sobre cómo un restaurante *nuevo se adhiere al servicio y contribuye con su menú*. Puede plantearse preguntas similares sobre qué elementos crean o modifican atributos y asociaciones.

- Para cada caso de uso del modelo de requisitos, intente describir el resultado o la condición posterior de cada caso de uso con las palabras proporcionadas por los diagramas de clases. A menudo, resulta útil mostrar el efecto de un caso de uso con la realización de bocetos de instancias de las clases antes y después de una ocurrencia del caso de uso. Por ejemplo, si la condición posterior del caso de uso dice "se agrega un elemento del menú al pedido del cliente", cree bocetos de las instancias de las clases de pedido y elemento del menú. Muestre los efectos del caso de uso como, por ejemplo, un nuevo vínculo o un nuevo objeto, con un color diferente o en un dibujo nuevo. Esto suele dar lugar a análisis sobre qué información es necesaria en el modelo. Aunque las clases de requisitos no están relacionadas directamente con la implementación, describen la información que el sistema necesitará almacenar y transmitir.
- Formule preguntas sobre las restricciones en los atributos y las asociaciones, especialmente sobre las restricciones que implican más de un atributo o una asociación.
- Pregunte sobre secuencias válidas y no válidas de casos de uso. Para ello, dibuje diagramas de secuencia o de actividades para representarlas.

Al examinar las relaciones entre las vistas que proporcionan los diferentes diagramas, podrá reconocer fácilmente los principales conceptos con los que trabajan los usuarios y ayudarles a entender lo que necesitan del sistema. También podrá conocer mejor los requisitos que las partes interesadas tienen menos claros. Puede planear el desarrollo de esas características, al menos de forma simplificada, en una fase temprana del proyecto para que los usuarios puedan experimentar con ellas.

El propósito del modelo de requisitos es comprender completamente el problema y sus implicaciones.

3.1.3 Modelo de Casos de Uso

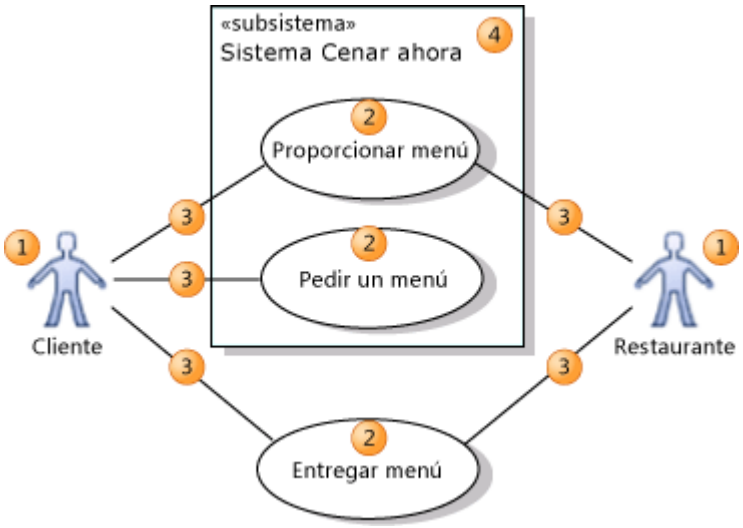
El modelo de casos de uso describe la funcionalidad propuesta del nuevo sistema. En un diagrama de casos de usos se describen las relaciones entre los requisitos, los usuarios y los componentes principales. Los requisitos no se describen en detalle, ya que esto puede hacerse en otros diagramas o en documentos que pueden vincularse a cada caso de uso. Un caso de uso representa una unidad discreta de interacción entre un usuario (humano o máquina) y el sistema.

Con la ayuda de un diagrama de casos de uso, es posible analizar y comunicar:

- Los escenarios en los que el sistema o aplicación interactúa con personas, organizaciones o sistemas externos.
- Los objetivos que el sistema o aplicación contribuye a lograr.
- El ámbito del sistema.

En un diagrama de casos de uso no se muestran los casos de uso en detalle, no se muestra el orden en que se llevan a cabo los pasos para lograr los objetivos de cada caso de uso. Esos detalles pueden describirse en otros diagramas y documentos, que pueden vincularse a cada caso de uso.

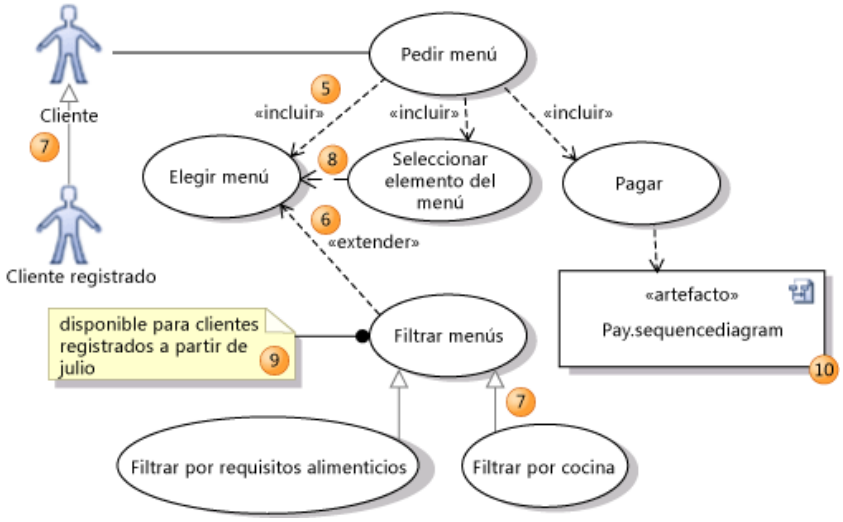
Los casos de uso solamente se usan para los requisitos funcionales de un sistema. Otros requisitos, como las reglas de negocio, los requisitos de calidad del servicio y las restricciones de implementación, deben representarse por separado. La arquitectura y los detalles internos también deben describirse por separado.



18 Casos de Uso

- **Un *actor* (1)** es una clase de persona, organización, dispositivo o componente de software externo que interactúa con el sistema. Los actores del ejemplo son cliente, restaurante, sensor de temperatura y titular de tarjeta de crédito.
- **Un *caso de uso* (2)** representa las acciones que uno o varios de los actores realizan a fin de conseguir un objetivo determinado. Los casos de uso del ejemplo son "Pedir menú", "Actualizar menú" y "Procesar pago".
  En un diagrama de casos de uso, casos de uso están asociados (3) a los actores que los realizan.
- **Asociación (3)** Indica que un actor forma parte de un caso de uso.
- **El *sistema (4)*** es aquello que se está desarrollando. Puede ser un pequeño componente de software cuyos actores simplemente son otros componentes de software; puede ser una aplicación completa; o puede ser un gran conjunto de aplicaciones distribuidas que se implementan en muchos equipos y dispositivos. Los subsistemas del ejemplo son "Sitio web de pedidos de menú", "Empresa de entrega de menús" y "Versión 2 del sitio web".
  En un diagrama de casos de uso pueden mostrarse los casos de uso que el sistema o sus subsistemas admiten.

3.1.3.1 Estructuración de Casos de Uso



19 Casos de Uso

- **Includ (5)** Un caso de uso de inclusión llama o invoca al caso de uso incluido. La inclusión se usa para mostrar cómo se divide un caso de uso en pasos más pequeños. El caso de uso incluido se encuentra en el extremo con la punta de flecha. . Generalmente se asume que los casos de uso incluidos se llamarán cada vez que se ejecute el camino base. Un ejemplo puede ser listar un conjunto de órdenes de clientes de las cuáles poder elegir antes de modificar una orden seleccionada; en este caso, el Caso de Uso <listar órdenes> se puede incluir en el Caso de Uso <modificar orden> cada vez que éste se ejecute.

Un Caso de Uso puede ser incluido por uno o más casos de uso, ayudando así a reducir la duplicación de funcionalidad al factorizar el comportamiento común en los casos de uso que se reutilizan muchas veces.

- **Extend (6)** Un caso de uso de extensión agrega objetivos y pasos al caso de uso extendido. Las extensiones solamente funcionan en ciertas condiciones. El caso de uso extendido se encuentra en el extremo con la punta de flecha.

Un Caso de Uso puede extender el comportamiento de otro Caso de Uso; típicamente cuando ocurren situaciones excepcionales. Por ejemplo, si antes de modificar un tipo particular de orden de cliente, un usuario debe obtener la aprobación de alguna autoridad superior, entonces el Caso de Uso <obtener aprobación> puede extender opcionalmente el Caso de Uso normal <modificar orden>.

- **Herencia (7)** Relaciona un elemento especializado y un elemento generalizado. El elemento generalizado se encuentra en el extremo con la punta de flecha.
  Un caso de uso especializado hereda los objetivos y actores de su generalización y puede agregar objetivos más específicos y los pasos para llevarlos a cabo.
  Un actor especializado hereda los casos de uso, los atributos y las asociaciones de su generalización y puede agregar más elementos.
- **Dependencia (8)** Indica que el diseño del origen depende del diseño del destino.
- **Comentario o Nota (9)** Se usa para agregar notas generales al diagrama.
- **Artefacto (10)** Un artefacto proporciona un vínculo a otro diagrama o documento y se crea arrastrando un archivo desde el Explorador de soluciones. Se puede vincular mediante una relación de dependencia a otro elemento del diagrama. Un artefacto se usa normalmente para vincular un

caso de uso a un diagrama de secuencia, un documento de Word o una presentación de PowerPoint que describe el caso de uso en detalle.

- Hipervínculo: dirección URL o ruta de acceso del diagrama o documento.

3.1.3.2 Una descripción de caso de uso generalmente incluirá

Comentarios generales y notas describiendo el caso de uso

Requisitos -cosas que el caso de uso debe permitir hacer al usuario, tales como <capacidad para actualizar pedido>, <capacidad para modificar pedido>, etc.

Restricciones -reglas acerca de qué se puede y qué no se puede hacer-. Incluye:

Pre-condiciones que deben ser verdaderas antes de que el caso de uso se ejecute, por ejemplo <crear pedido> debe preceder a <modificar pedido>

Post-condiciones que deben ser verdaderas una vez que el caso de uso se ejecutó, por ejemplo <el pedido está modificado y es consistente>

invariantes: éstas son siempre verdaderas -por ejemplo, un pedido debe tener siempre un número de cliente.

Escenarios -descripciones secuenciales de los pasos que se toman para llevar a cabo el caso de uso. Pueden incluir escenarios múltiples, para satisfacer circunstancias excepcionales y caminos de proceso alternativos

Diagramas de escenarios -diagramas de secuencia para describir el flujo de trabajo- similar al punto 4 pero descrito gráficamente.

Atributos adicionales como fase de implementación, número de versión, rango de complejidad, estereotipo y estado

3.1.4 Diagramas de Actividades

Un diagrama de actividades muestra un proceso de negocio o un proceso de software como un flujo de trabajo a través de una serie de acciones. Las personas, los componentes de software o los equipos pueden realizar estas acciones.

Puede usar un diagrama de actividades para describir procesos de varios tipos, como los ejemplos siguientes:

Un proceso de negocio o un flujo de trabajo entre los usuarios y el sistema. Para más información, vea Requisitos del usuario de modelos.

Los pasos que se realizan en un caso de uso. Para más información, vea Diagramas de casos de uso de UML: Instrucciones.

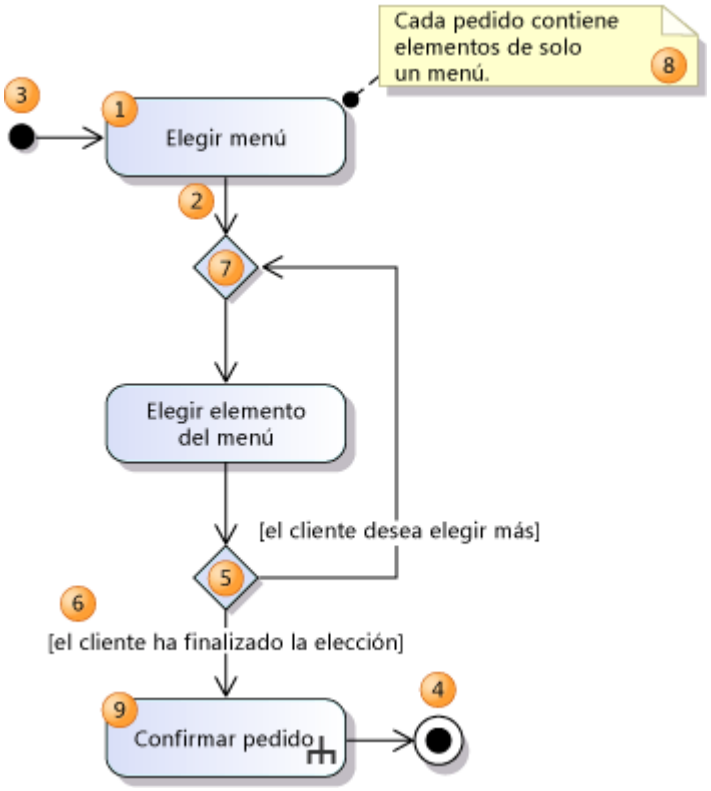
Un protocolo de software, es decir, las secuencias de interacciones entre componentes permitidas.

Un algoritmo de software.

En este tema se describen los elementos que puede usar en los diagramas de actividades. Para obtener información detallada sobre el dibujo de diagramas de actividades, vea Diagramas de actividades UML: Instrucciones. Para crear un diagrama de actividades UML, en el menú Arquitectura, haga clic en Nuevo diagrama de UML o de capas. Para más información sobre cómo dibujar diagramas de modelado en general, vea Editar modelos y diagramas UML.

3.1.4.1 Flujos de control simple

Puede mostrar una secuencia de acciones con bifurcaciones y bucles. Para más información sobre cómo usar los elementos que se describen aquí, vea la sección Describir el flujo de control del tema Diagramas de actividades UML: Instrucciones.



20 Diagrama de Actividad – Flujo de Control Simple

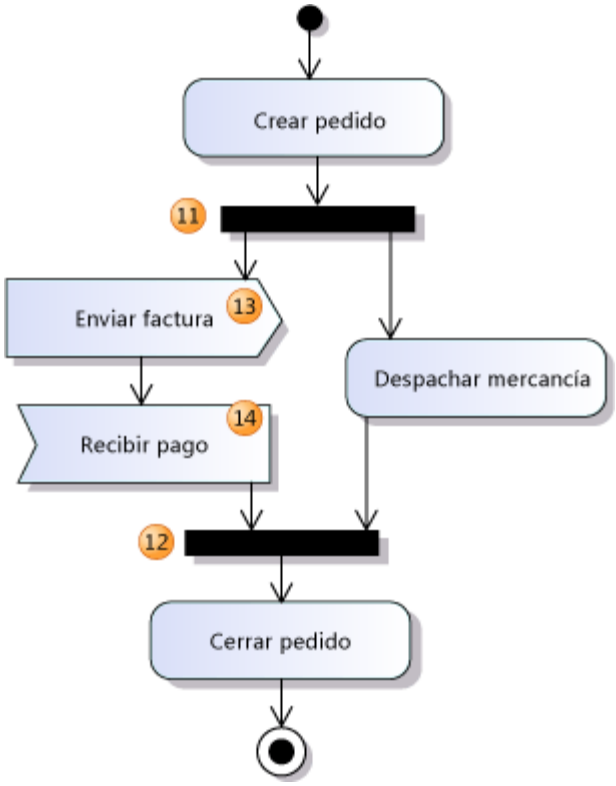
- Accion (1) Paso de la actividad en el que los usuarios o el software realizan alguna tarea. La acción se puede iniciar cuando un token llega a todos sus flujos entrantes. Cuando termina, los tokens se envían en todos los flujos salientes.
- Flujo de Control (2) Conector que muestra el flujo de control entre las acciones. Para interpretar el diagrama, imagine que un token fluye de una acción a la siguiente
  Para crear un flujo de control, use la herramienta **Conector**.
- Nodo Inicial (3) Indica la primera acción o las primeras acciones de la actividad. Cuando se inicia la actividad, un token fluye desde el nodo inicial.
- Actividad Nodo Final (4) Fin de la actividad. Cuando llega un token, la actividad finaliza.
- Decisión Condición (5) Bifurcación condicional de un flujo. Tiene una entrada y dos o más salidas. Un token entrante solo emerge en una de las salidas.
- Restricción (6) Condición que especifica si un token puede fluir por un conector. Se usa con más frecuencia en los flujos salientes de un nodo de decisión.

Para establecer una restricción, haga clic con el botón derecho en un flujo, haga clic en **Propiedades** y, después, establezca la propiedad **Restricción**.

- Merge (7) Necesario para combinar los flujos que se dividieron mediante un nodo de decisión. Tiene dos o más entradas y una salida. Un token en cualquier entrada emerge en la salida.
- Comentario o Note (8) Proporciona información adicional sobre los elementos a los que está vinculado.
- Action (9) Acción que se define con más detalle en otro diagrama de actividades.

3.1.4.2 Flujos simultáneos

Puede describir secuencias de acciones que se ejecutan al mismo tiempo.



21 Diagrama de Actividad - Flujos Simultáneos

- **Fork Node** (11) Divide un único flujo en flujos simultáneos. Cada token entrante genera un token en cada conector saliente.
- **Join Node** (12) Combina flujos simultáneos en un único flujo. Cuando cada flujo entrante tiene un token en espera, se genera un token en la salida.
- **Flujo de Salida** (13) Acción que envía un mensaje o una señal a otra actividad o a un subproceso simultáneo de la misma actividad. El tipo y el contenido del mensaje están implícitos en el título de la acción o se especifican en los comentarios adicionales.

La acción puede enviar datos de la señal, que se pueden pasar a la acción de un flujo de objeto o terminal de entrada (16).

- **Action Entrante** (14) Acción que espera un mensaje o una señal antes de continuar con la acción. El tipo de mensaje que la acción puede recibir está implícito en el título o se especifica en los comentarios adicionales.

Si la acción no tiene ningún flujo de control entrante, genera un token cada vez que recibe un mensaje.

La acción puede recibir datos de la señal, que se pueden pasar a un flujo de objeto o terminal de salida (17).

- **IsUnmarshall**: si es true, puede haber varios terminales de salida con tipo y los datos se deserializan en ellos. Si es false, todos los datos aparecen en un terminal.

3.1.5 Diagramas de Transición de Estados

Un diagrama de transición de estados muestra el comportamiento dependiente del tiempo de un sistema de información. Representa los estados que puede tomar un componente o un sistema o un objeto y muestra los eventos que implican el cambio de un estado a otro.

3.1.5.1 Elementos del Diagrama

Los dos elementos principales en estos diagramas son los estados y las posibles transiciones entre ellos.

- El ***estado*** de un componente o sistema representa algún comportamiento que es observable externamente y que perdura durante un periodo de tiempo finito. Viene dado por el valor de uno o varios atributos que lo caracterizan en un momento dado.
- Una ***transición*** es un cambio de estado producido por un evento y refleja los posibles caminos para llegar a un estado final desde un estado inicial.

Desde un estado pueden surgir varias transiciones en función del evento que desencadena el cambio de estado, teniendo en cuenta que, las transiciones que provienen del mismo estado no pueden tener el mismo evento, salvo que exista alguna condición que se aplique al evento.

Un diagrama solo puede tener un estado inicial, que se representa mediante una transición sin etiquetar al primer estado normal del diagrama. Pueden existir varias transiciones desde el estado inicial, pero deben tener asociadas condiciones, de manera que sólo una de ellas sea la responsable de iniciar el flujo. En ningún caso puede haber una transición dirigida al estado inicial.

El estado final representa que un componente ha dejado de tener cualquier interacción o actividad. No se permiten transiciones que partan del estado final. Puede haber varios estados finales en un diagrama, ya que es posible concluir el ciclo de vida de un componente desde distintos estados y mediante diferentes eventos, pero dichos estados son mutuamente excluyentes, es decir, sólo uno de ellos puede ocurrir durante una ejecución del sistema.

Los diagramas de transición de estados comprenden además otros dos elementos que ayudan a clarificar el significado de los distintos estados por los que pasa un componente o sistema. Estos elementos se conocen como acciones y actividades. Una acción es una operación instantánea asociada a un evento, cuya duración se considera no significativa y que se puede ejecutar: dentro de un estado, al entrar en un estado o al salir

del mismo. Una actividad es una operación asociada a un estado que se ejecuta durante un intervalo de tiempo hasta que se produce el cambio a otro estado.

Para aquellos estados que tengan un comportamiento complejo, se puede utilizar un diagrama de transición de estados de más bajo nivel. Estos diagramas se pueden mostrar por separado o bien incluirse en el diagrama de más alto nivel, dentro del contorno del estado que representa. En cualquier caso, su contenido formará un contexto independiente del resto, con sus propios estados inicial y final.

3.1.5.2 Notación

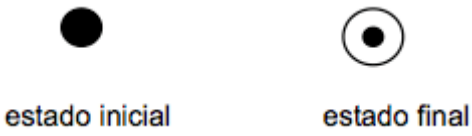
Estado

Un estado se representa como un rectángulo con las esquinas redondeadas. El nombre del estado se coloca dentro del rectángulo y debe ser único en el diagrama. Si se repite algún nombre, se asume que simboliza el mismo estado.

Las acciones y actividades descritas como respuesta a eventos que no producen un cambio de estado, se representan dentro del rectángulo con el formato:

nombre-evento (parámetros) [condición] /acción

El estado inicial se representa con un pequeño circulo relleno, y el estado final como un pequeño circulo relleno con una circunferencia que lo rodea.

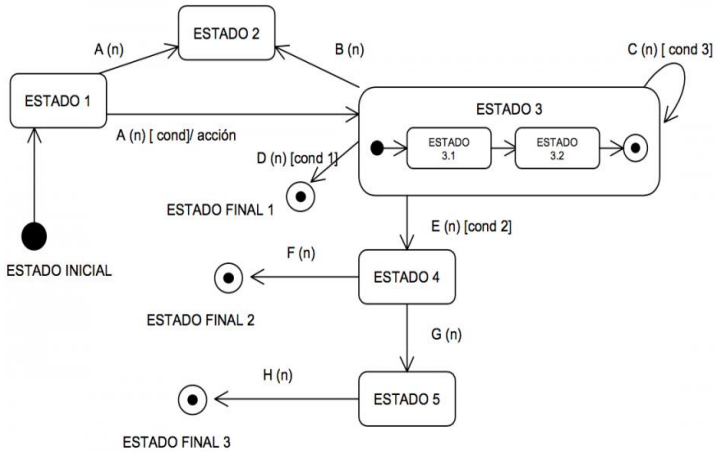


Transición

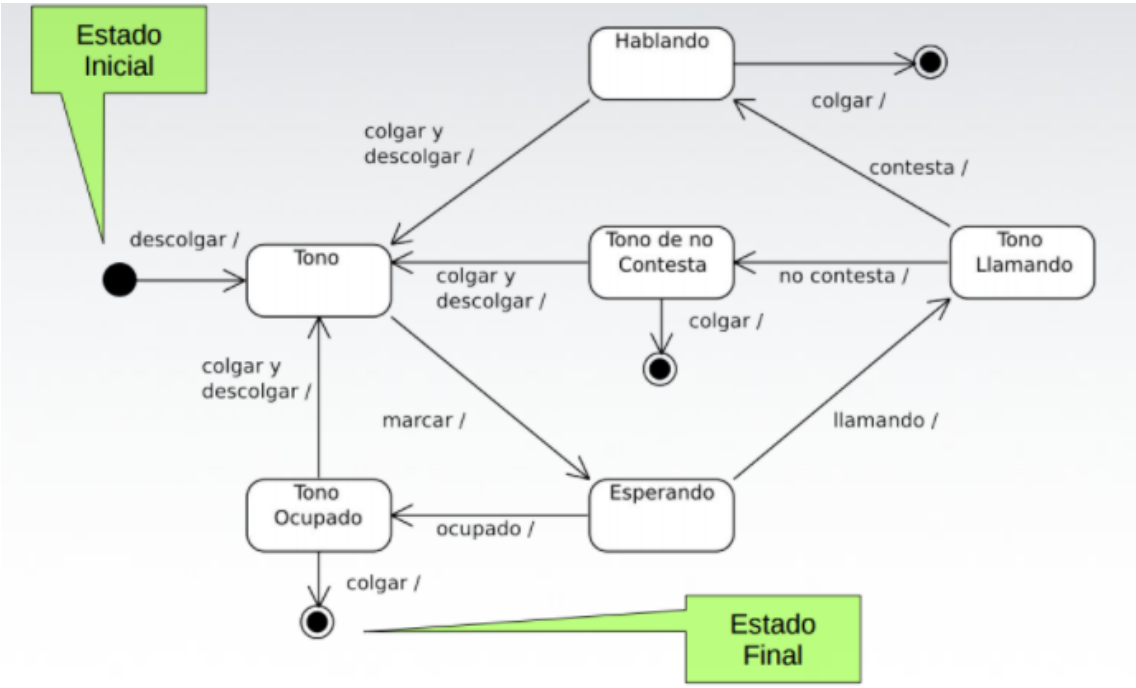
Una transición se representa con una flecha continua que une dos estados y que se dirige al estado al que cambia el componente. Junto a ella se coloca una etiqueta que debe contener al menos el nombre del evento que provoca la transición. Según el nivel de detalle, puede presentar otros elementos con el formato siguiente:

nombre-evento (parámetros) [condición] /acción

Ejemplo



22 Diagrama de Estados – Notación



23 Diagrama de Estados – Ejemplo de llamada telefónica

Ejemplo Horno Microondas

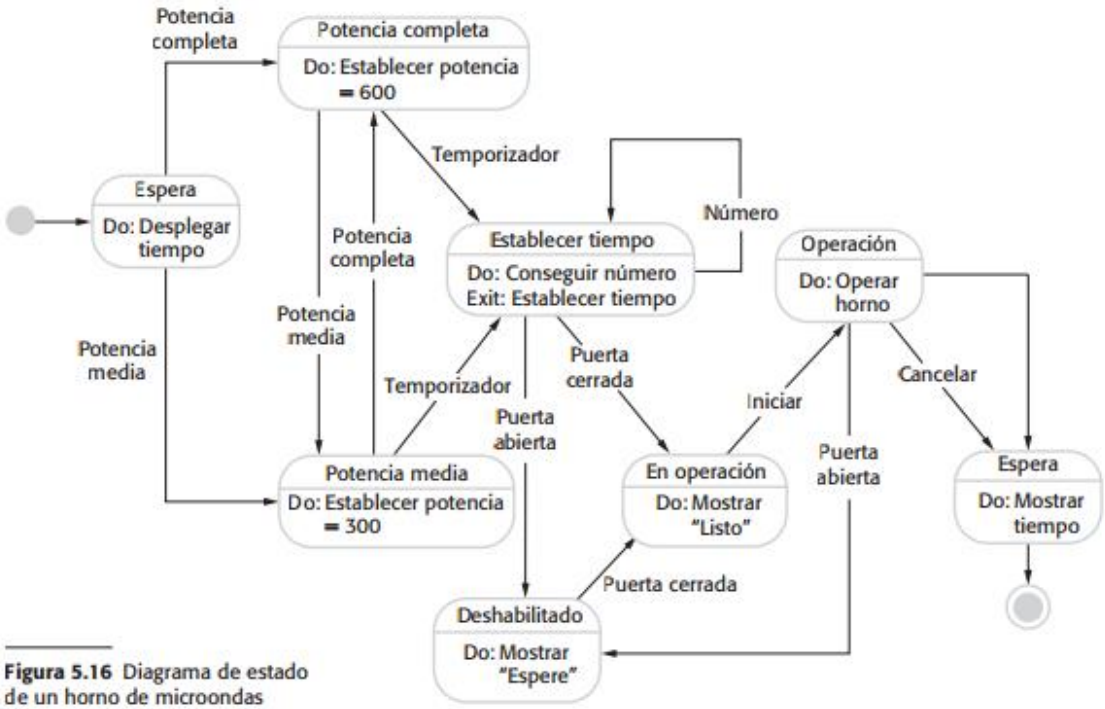


Figura 5.16 Diagrama de estado de un horno de microondas

24 Diagrama de Estados - Ejemplo microondas

Los hornos de microondas reales son mucho más complejos que este sistema, pero el sistema simplificado es más fácil de entender. Este microondas sencillo tiene un interruptor para seleccionar potencia completa o media, un teclado numérico para ingresar el tiempo de cocción, un botón de iniciar/detener y una pantalla alfanumérica. Se supone que la secuencia de acciones al usar el horno de microondas es: 1. Seleccionar el nivel de potencia (ya sea media o completa) 2. Ingresar el tiempo de cocción con el teclado numérico. 3. Presionar “Iniciar”, y la comida se cocina durante el tiempo dado. Por razones de seguridad, el horno no opera cuando la puerta esté abierta y, al completar la cocción, se escuchará un timbre. El horno tiene una pantalla alfanumérica muy sencilla que se usa para mostrar varios avisos de alerta y mensajes de advertencia.

el sistema empieza en un estado de espera e, inicialmente, responde al botón de potencia completa o al botón de potencia media. Los usuarios pueden cambiar su opinión después de seleccionar uno de ellos y oprimir el otro botón. Se establece el tiempo y, si la puerta está cerrada, se habilita el botón Iniciar. Al presionar este botón comienza la operación del horno y tiene lugar la cocción durante el tiempo especificado. Éste es el final del ciclo de cocción y el sistema regresa al estado de espera

Estado	Descripción
Esperar	El horno espera la entrada. La pantalla indica el tiempo actual.
Potencia media	La potencia del horno se establece en 300 watts. La pantalla muestra "Potencia media".
Potencia completa	La potencia del horno se establece en 600 watts. La pantalla muestra "Potencia completa".
Establecer tiempo	El tiempo de cocción se establece al valor de entrada del usuario. La pantalla indica el tiempo de cocción seleccionado y se actualiza conforme se establece el tiempo.
Deshabilitado	La operación del horno se deshabilita por cuestiones de seguridad. La luz interior del horno está encendida. La pantalla indica "No está listo".
Habilitado	Se habilita la operación del horno. La luz interior del horno está apagada. La pantalla muestra "Listo para cocinar".
Operación	Horno en operación. La luz interior del horno está encendida. La pantalla muestra la cuenta descendente del temporizador. Al completar la cocción, suena el timbre durante cinco segundos. La luz del horno está encendida. La pantalla muestra "Cocción completa" mientras suena el timbre.
Estímulo	**Descripción**
Potencia media	El usuario oprime el botón de potencia media.
Potencia completa	El usuario oprime el botón de potencia completa.
Temporizador	El usuario oprime uno de los botones del temporizador.
Número	El usuario oprime una tecla numérica.
Puerta abierta	El interruptor de la puerta del horno no está cerrado.
Puerta cerrada	El interruptor de la puerta del horno está cerrado.
Iniciar	El usuario oprime el botón Iniciar.
Cancelar	El usuario oprime el botón Cancelar.

La notación UML permite indicar la actividad que ocurre en un estado. En una especificación detallada del sistema, hay que proporcionar más detalle tanto de los estímulos como de los estados del sistema. Esto se ilustra en la figura 5.17, la cual señala una descripción tabular de cada estado y cómo se generan los estímulos que fuerzan transiciones de estado. El problema con el modelado basado en el estado es que el número de posibles estados se incrementa rápidamente. Por lo tanto, para modelos de sistemas grandes, necesita ocultar detalles en los modelos. Una forma de hacer esto es mediante la noción de un superestado que encapsule algunos estados separados. Este superestado se parece a un solo estado en un modelo de nivel superior, pero entonces se expande para mostrar más detalles en un diagrama separado. Para ilustrar este concepto, considere el estado Operación en la figura 5.15. Éste es un superestado que puede expandirse, como se ilustra en la figura 5.18. El estado Operación incluye algunos subestados. Muestra que la operación comienza con una comprobación de estatus y que, si se descubren problemas, se indica una alarma y la operación se deshabilita. La cocción implica operar el generador de microondas durante el

tiempo especificado; al terminar, suena un timbre. Si la puerta está abierta durante la operación, el sistema se mueve hacia el estado deshabilitado, como se muestra en la figura 5.15.

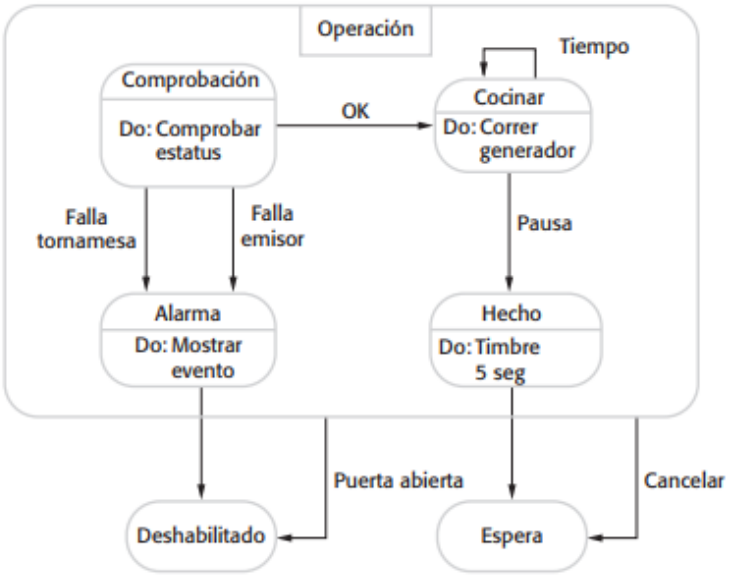


Figura 5.18 Operación del horno de microondas

Ilustración 25 Diagrama de Estados - Operación del horno microondas

3.2 Diseño

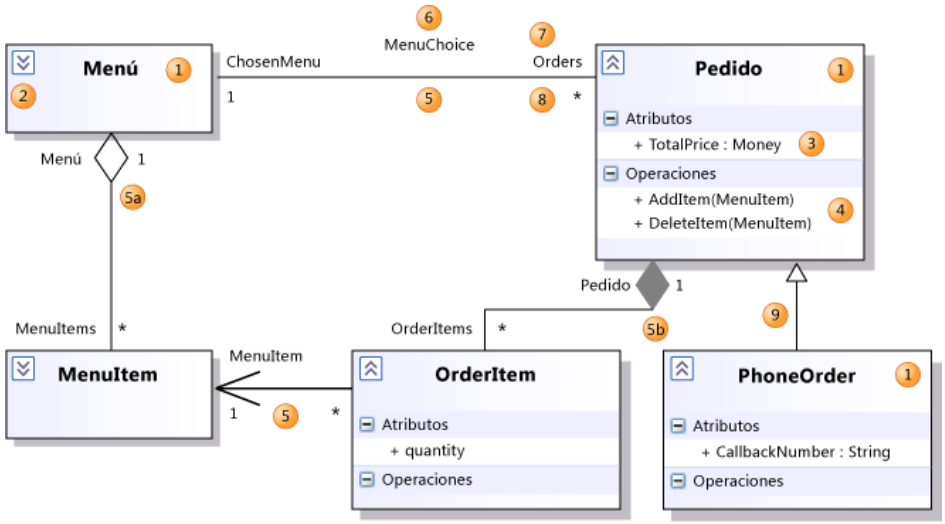
Artefacto de análisis	Preguntan que se contestan
Casos de Uso	¿Cuáles son los procesos del dominio?
Modelo Conceptual	¿Cuáles son los conceptos, los términos?
Diagramas de la secuencia de un sistema	¿Cuáles son los eventos y las operaciones del sistema?
Contratos	¿Qué hacen las operaciones del sistema?

3.2.1 Modelo de Clases

Un diagrama de clase, aporta una visión estática o de estructura de un sistema, sin mostrar la naturaleza dinámica de las comunicaciones entre los objetos de las clases. En los diagramas de clases de UML se describen el objeto y las estructuras de información que se usan en la aplicación, tanto de forma interna como en la comunicación con los usuarios. Esta información se describe sin hacer referencia a ninguna implementación concreta. Las clases y relaciones se pueden implementar de muchas maneras, por ejemplo, en tablas de bases de datos, en nodos XML o en composiciones de objetos de software.

Los diagramas de clases de UML pueden utilizarse para una gran variedad de propósitos:

- Para proporcionar una descripción de los tipos que se utilizan en un sistema y se pasan entre sus componentes que no tenga nada que ver con su implementación.
- Para clarificar el glosario de términos que se utiliza en la comunicación entre la aplicación y los usuarios y en las descripciones de las necesidades de los usuarios. Por ejemplo, piense en los casos de usuario, los casos de uso y otras descripciones de los requisitos de la aplicación de un restaurante. En este tipo de descripción, encontrará términos como Menú, Pedido, Comida, Precio, Pago, etc. Puede dibujar un diagrama de clases de UML en el que se definan las relaciones entre estos términos. De este modo, se reducirá el riesgo de inconsistencias en las descripciones de los requisitos, así como en la interfaz de usuario y los documentos de ayuda.



26 Diagrama de clases

Los elementos principales de un diagrama de clase son cajas, que son los íconos utilizados para representar clases e interfaces. Cada caja se divide en partes horizontales. La parte superior contiene el nombre de lalase. La sección media menciona sus atributos.

La tercera sección del diagrama de clase contiene las

- **Clase (1)** Definición de objetos que comparten determinadas características estructurales y de comportamiento. Muestran las clases de objeto en el sistema y las asociaciones entre estas clases. Una clase abstracta o un método abstracto se indica con el uso de cursivas en el nombre del diagrama de clase. “<<interface>>” (llamada estereotipo) arriba del nombre.

- **Clasificador (2)** (Nombre de la clase) Nombre general de una clase, interfaz o enumeración. Los componentes, casos de uso y actores también son clasificadores.

- **Atributo (3)** Un atributo es algo que un objeto de dicha clase conoce o puede proporcionar todo el tiempo. Por lo general, los atributos se implementan como campos de la clase, pero no necesitan serlo. Podrían ser valores que la clase puede calcular a partir de sus variables o valores instancia y que puede obtener de otros objetos de los cuales está compuesto. Por ejemplo, un objeto puede conocer siempre la hora actual y regresarla siempre que se le solicite. Por tanto, sería adecuado mencionar la hora actual como un atributo de dicha clase de objetos. Sin embargo, el objeto muy probablemente no tendría dicha hora almacenada en una de sus variables instancia, porque necesitaría actualizar de manera continua ese campo. En vez de ello, el objeto probablemente calcularía la hora actual (por ejemplo, a través de consulta con objetos de otras clases) en el momento en el que se le solicite la hora.

  Cada atributo puede tener un nombre, un tipo y un nivel de visibilidad. El tipo y la visibilidad son opcionales. El tipo sigue al nombre y se separa de él mediante dos puntos. La visibilidad se indica mediante un –, #, ~ o + precedente, que indica, respectivamente, visibilidad privada, protegida, paquete o pública. También es posible especificar que un atributo es estático o de clase, subrayándolo. Cada

operación puede desplegarse con un nivel de visibilidad, parámetros con nombres y tipos, y un tipo de retorno.

- **Método (4)** operaciones o comportamientos de la clase. Una operación es lo que pueden hacer los objetos de la clase. Por lo general, se implementa como un método de la clase.
- **Agregación (5a)** Asociación que representa una relación de propiedad compartida.
- Composición (5b) Asociación que representa una relación parte/todo.

Nombre de la Relación (6) Nombre de una asociación. Puede dejarse vacío.

- **Multiplicidad (8)** Indica cuántos de los objetos de este extremo se pueden vincular a cada objeto del otro. En el ejemplo, cada pedido debe vincularse exactamente a un solo menú.

* significa que no hay límite superior en el número de vínculos que se pueden establecer.

- **Generalización (9)** Una Clase específico hereda parte de su definición del clasificador general. El clasificador general se encuentra en el extremo del conector de la flecha. El clasificador específico hereda los atributos, las asociaciones y las operaciones.

Use la herramienta **Herencia** para crear una generalización entre dos clasificadores.

Las flechas en cualquiera o en ambos lados de una línea de asociación indican navegabilidad. Además, cada extremo de la línea de asociación puede tener un valor de multiplicidad desplegado. Navegabilidad y multiplicidad se explican con más detalle más adelante, en esta sección. Una asociación también puede conectar una clase consigo misma, mediante un bucle. Tal asociación indica la conexión de un objeto de la clase con otros objetos de la misma clase. Una asociación con una flecha en un extremo indica navegabilidad en un sentido. La flecha significa que, desde una clase, es posible acceder con facilidad a la segunda clase asociada hacia la que apunta la asociación; sin embargo, desde la segunda clase, no necesariamente puede accederse con facilidad a la primera clase. Otra forma de pensar en esto es que la primera clase está al tanto de la segunda, pero el segundo objeto de clase no necesariamente está directamente al tanto de la primera clase. Una asociación sin flechas por lo general indica una asociación de dos vías, pero también simplemente podría significar que la navegabilidad no es importante y, por tanto, que queda fuera.

Debe observarse que un atributo de una clase es muy parecido a una asociación de la clase con el tipo de clase del atributo. Es decir, para indicar que una clase tiene una propiedad llamada “name” (nombre) de tipo String, podría desplegarse dicha propiedad como atributo.

Una relación de dependencia representa otra conexión entre clases y se indica mediante una línea punteada (con flechas opcionales en los extremos y con etiquetas opcionales). Una clase depende de otra si los cambios en la segunda clase pueden requerir cambios en la primera. Una asociación de una clase con otra

automáticamente indica una dependencia. No se necesitan líneas punteadas entre clases si ya existe una asociación entre ellas. Sin embargo, para una relación transitoria (es decir, una clase que no mantiene alguna conexión de largo plazo con otra, sino que usa dicha clase de manera ocasional), debe dibujarse una línea punteada desde la primera clase hasta la segunda.

La multiplicidad de un extremo de una asociación significa el número de objetos de dicha clase que se asocia con la otra clase. Una multiplicidad se especifica mediante un entero no negativo o mediante un rango de enteros. Una multiplicidad especificada por “0..1” significa que existen 0 o 1 objetos en dicho extremo de la asociación. Por ejemplo, cada persona en el mundo tiene un número de seguridad social o no lo tiene (especialmente si no son ciudadanos estadounidenses); por tanto, una multiplicidad de 0..1 podría usarse en una asociación entre una clase Person y una clase SocialSecurityNumber (número de seguridad social) en un diagrama de clase. Una multiplicidad que se especifica como “1...*” significa uno o más, y una multiplicidad especificada como “0...*” o sólo “*” significa cero o más. Un * se usa como la multiplicidad en

el extremo OwnedObject de la asociación con la clase Person en la figura A 1 . 2, porque una Person puede poseer cero o más objetos.

Si un extremo de una asociación tiene multiplicidad mayor que 1, entonces los objetos de la clase a la que se refiere en dicho extremo de la asociación probablemente se almacenan en una colección, como un conjunto o lista ordenada. También podría incluirse dicha clase colección en sí misma en el diagrama UML, pero tal clase por lo general queda fuera y se supone, de manera implícita, que está ahí debido a la multiplicidad de la asociación.

Una agregación es un tipo especial de asociación que se indica mediante un diamante hueco en un extremo del ícono. Ello indica una relación “entero/parte”, en la que la clase a la que apunta la flecha se considera como una “parte” de la clase en el extremo diamante de la asociación. Una composición es una agregación que indica fuerte propiedad de las partes. En una composición, las partes viven y mueren con el propietario porque no tienen papel en el sistema de software independiente del propietario.

Otro elemento común de un diagrama de clase es una nota, que se representa mediante una caja con una esquina doblada y se conecta a otros íconos mediante una línea punteada. Puede tener contenido arbitrario (texto y gráficos) y es similar a comentarios en lenguajes de programación. Puede contener comentarios acerca del papel de una clase o restricciones que todos los objetos de dicha clase deban satisfacer. Si los contenidos son una restricción, se encierran entre llaves.

3.2.2 Modelo de Secuencias

Los diagramas de secuencia en el UML se usan principalmente para modelar las interacciones entre los actores y los objetos en un sistema, así como las interacciones entre los objetos en sí. El UML tiene una amplia sintaxis para diagramas de secuencia, lo cual permite muchos tipos diferentes de interacción a modelar.

Es posible modelar un escenario especifico de un caso de uso, los eventos que generan los factores externos, el orden y los eventos del sistema. En este caso los sistemas se tratan como cajas negras; los diagramas destacan los eventos que cruzan los límites del sistema desde los actores a los sistemas.

Sería conveniente hacer un Diagrama de secuencia del sistema para el escenario principal de éxito del caso de uso, y los escenarios alternativos complejos o frecuentes.

UML no define nada como DSS (diagrama de secuencia del sistema), pero se utiliza para representar sistemas como cajas negras. Los diagramas de secuencia de diseño se utilizan para representar la interacción entre objetos de una clase para completar una funcionalidad dentro del sistema software.

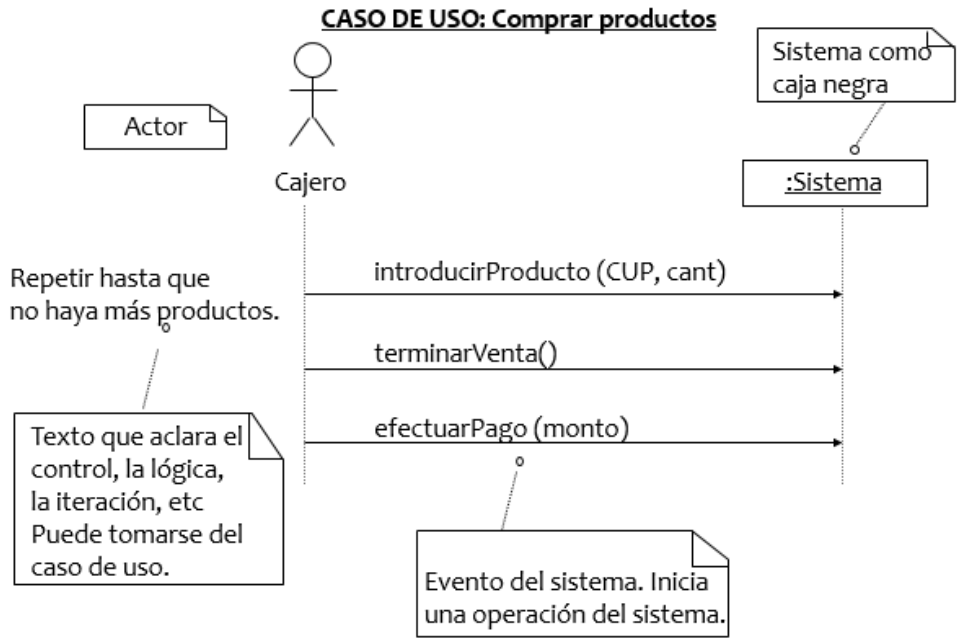


Ilustración 27 Diagrama de Secuencia para CU Comprar Productos

En este diagrama se puede visualizar al sistema como una caja negra, cuyo nombre podría ser el nombre del sistema.

Otro ejemplo para el diagrama de secuencia en el que incluyen las interacciones en el caso de uso "ver información de paciente", donde un recepcionista médico puede conocer la información de algún paciente, se puede utilizar para ver la interacción de mensajes para comprender mejor el escenario del caso de uso.

Nota: las clases se representan sin el subrayado, lo que denota que son clases y no instancias de clases.

Ejemplo de Sistema de recepción de pacientes con problemas de salud mental MHC-PMS. En ocasiones, los pacientes que sufren de problemas de salud mental son un riesgo para otros o para sí mismos. Por ello, es posible que en un hospital deban mantenerse contra su voluntad para que se les suministre el tratamiento. Tal detención está sujeta a estrictas protecciones legales, por ejemplo, la decisión de detener a un paciente tiene que revisarse con regularidad, para que no se detenga a la persona indefinidamente sin una buena razón. Una de las funciones del MHC-PMS es garantizar que se implementen dichas protecciones.

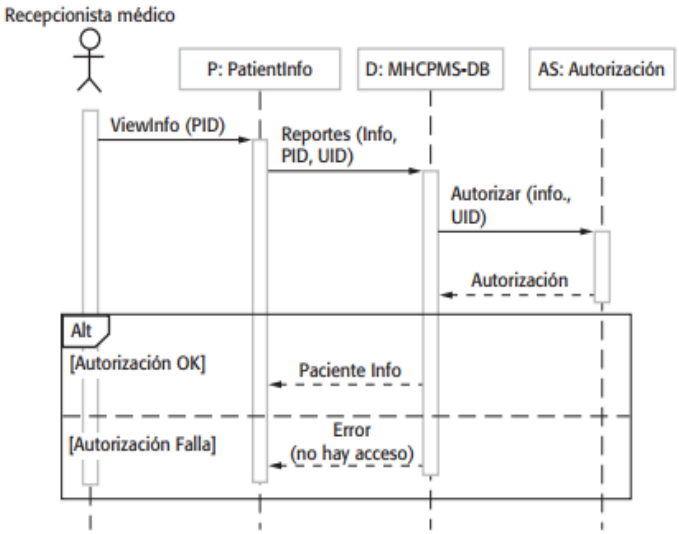


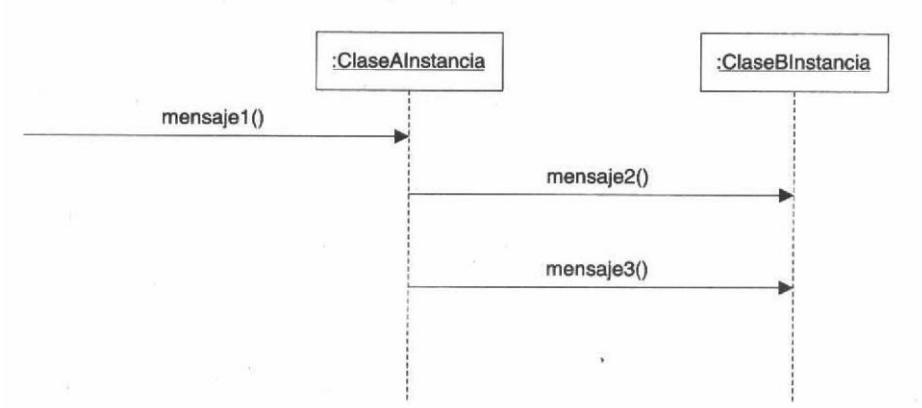
Figura 5.6 Diagrama de secuencia para "ver información del paciente"

28 Diagrama de Secuencia del Sistema

En este ejemplo, también se muestra la notación empleada para exponer alternativas. Un recuadro marcado con "alt" se usa con las condiciones indicadas entre corchetes.

se lee del siguiente modo: 1. El recepcionista médico activa el método ViewInfo (ver información) en una instancia P de la clase de objeto PatientInfo, y suministra el identificador del paciente, PID. P es un objeto de interfaz de usuario, que se despliega como un formato que muestra la información del paciente. 2. La instancia P llama a la base de datos para regresar la información requerida, y suministra el identificador del recepcionista para permitir la verificación de seguridad (en esta etapa no se preocupe de dónde proviene este UID). La base de datos comprueba, mediante un sistema de autorización, que el usuario esté autorizado para tal acción. 4. Si está autorizado, se regresa la información del paciente y se llena un formato en la pantalla del usuario. Si la autorización falla, entonces se regresa un mensaje de error.

3.2.2.1 Notación básica de los diagramas de secuencia en el diseño



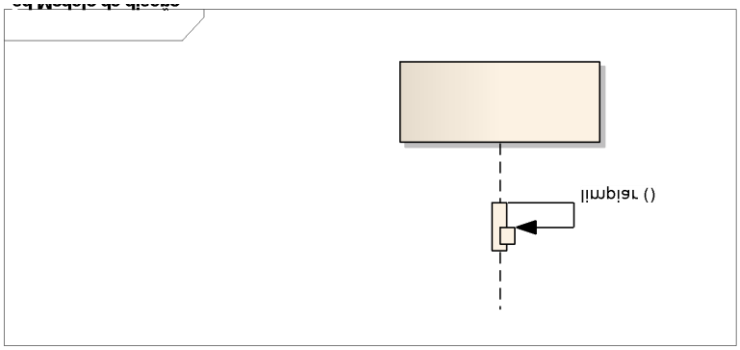
29 Diagrama de Secuencia - Notación

Los objetos que intervienen se muestran a lo largo de la parte superior del diagrama, con una línea punteada que se dibuja verticalmente a partir de éstos. Cada mensaje entre objetos se representa con una

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

Para representar la creación de una instancia de una clase se utiliza las cajas de activación, (lo que sería una llamada de rutina ordinaria). En un diagrama de secuencia se puede no representar el retoro de un mensaje mediante una línea punteada con la punta de flecha abierta, al final de la caja de activación. Lo normal es que se excluya, pero en ocasiones es utilizada para describir lo que se está devolviendo (si es el caso) a partir del mensaje.

También es posible representar un mensaje que se envía de un objeto a él mismo utilizando una caja de activación anidada.



Ejemplo para transferir datos:

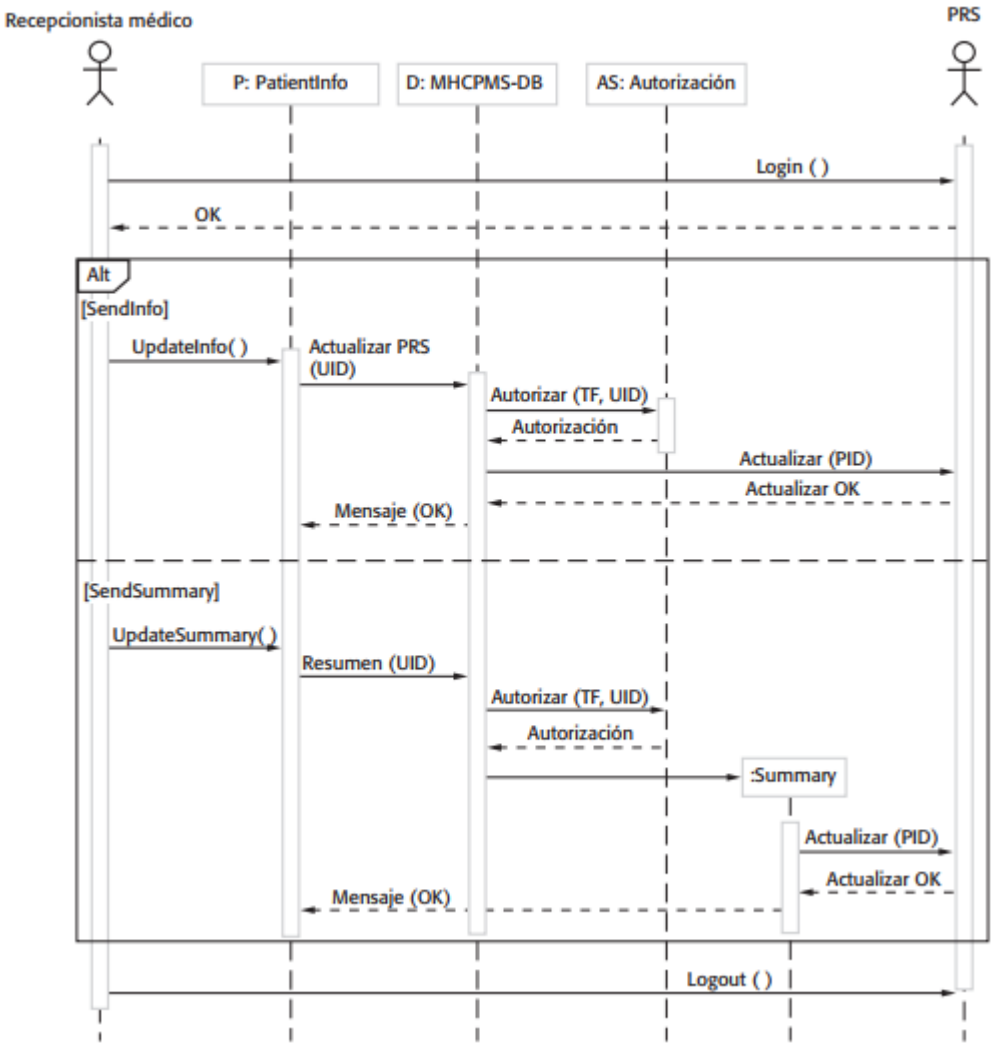


Figura 5.7 Diagrama de secuencia para transferir datos

30 Diagrama de Secuencia para transferir datos

La figura 5.7 es otro ejemplo de un diagrama de secuencia que representa la comunicación directa entre los actores en el sistema y la creación de objetos como parte de una secuencia de operaciones. En este ejemplo, un objeto del tipo Summary (resumen) se crea para contener los datos del resumen que deben subirse al PRS (patient record system, es decir, el sistema de registro de paciente). Este diagrama se lee de la siguiente manera: 1. El recepcionista inicia sesión (log) en el PRS. 2. Hay dos opciones disponibles. Las opciones permiten la transferencia directa de información actualizada del paciente al PRS, y la transferencia de datos del resumen de salud del MHC-PMS al PRS. 3. En cada caso, se verifican los permisos del recepcionista usando el sistema de autorización. 4. La información personal se transfiere directamente del objeto de interfaz del usuario al PRS. De manera alternativa, es posible crear un registro del resumen de la base de datos y, luego, transferir dicho registro. 5. Al completar la transferencia, el PRS emite un mensaje de estatus y el usuario termina la sesión (log off). A menos que use diagramas de secuencia para generación de código o documentación detallada, en dichos diagramas no tiene que incluir todas las interacciones. Si desarrolla modelos iniciales de sistema en el proceso de desarrollo para apoyar la ingeniería de

requerimientos y el diseño de alto nivel, habrá muchas interacciones que dependan de decisiones de implementación. Por ejemplo, en la figura 5.7, la decisión sobre cómo conseguir el identificador del usuario para comprobar la autorización podría demorarse. En una implementación, esto implicaría la interacción con un objeto User (usuario), pero esto no es importante en esta etapa y, por lo tanto, no necesita incluirse en el diagrama de secuencia.

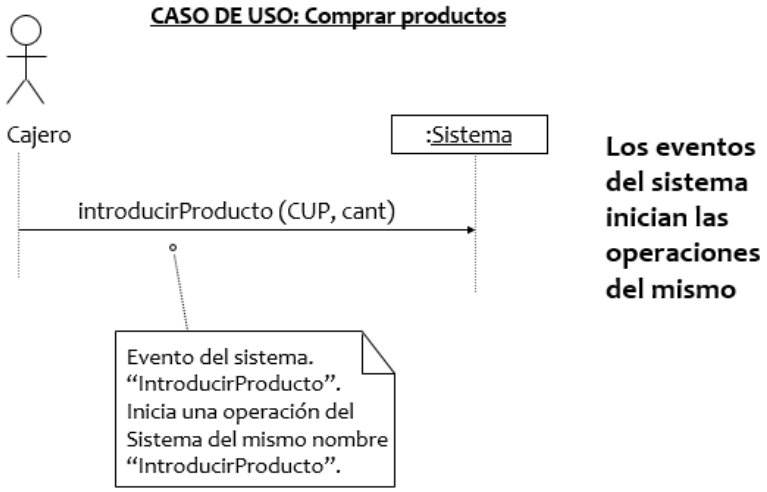
3.2.3 Modelo de Objetos / Comunicación

Diagramas de Interacción => explican gráficamente cómo los objetos interactúan a través de mensajes para realizar las tareas.
Los **diagramas de interacción** se realizan en la fase de diseño. Antes deben generarse los siguientes artefactos:

- **Modelo Conceptual**: con ellos se podrán definir las clases de software correspondientes a los conceptos.
- **Contratos de la operación del sistema**: => identifican las responsabilidades y las poscondiciones que deben prever los diagramas de interacción.
- **Casos de uso reales** (o esenciales): => brindan información sobre las tareas que realizan los diagramas de interacción y lo estipulado en los contratos.

Un Diagrama de Interacción explica gráficamente las interacciones existentes entre las instancias. Su punto de partida es el cumplimiento de las postcondiciones de los contratos de una operación.

- Hay dos tipos de estos diagramas: ambos sirven para expresar interacciones semejantes:
- 1. - Diagramas de Colaboración.
- 2. - Diagramas de Secuencia.



31 Diagrama de Colaboración

Diagramas de Colaboración => describen las interacciones entre los objetos en un formato de grafo o red:

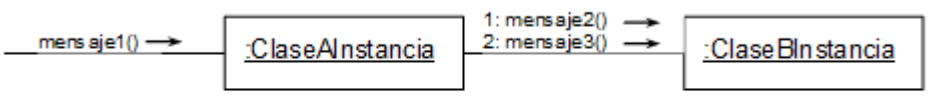
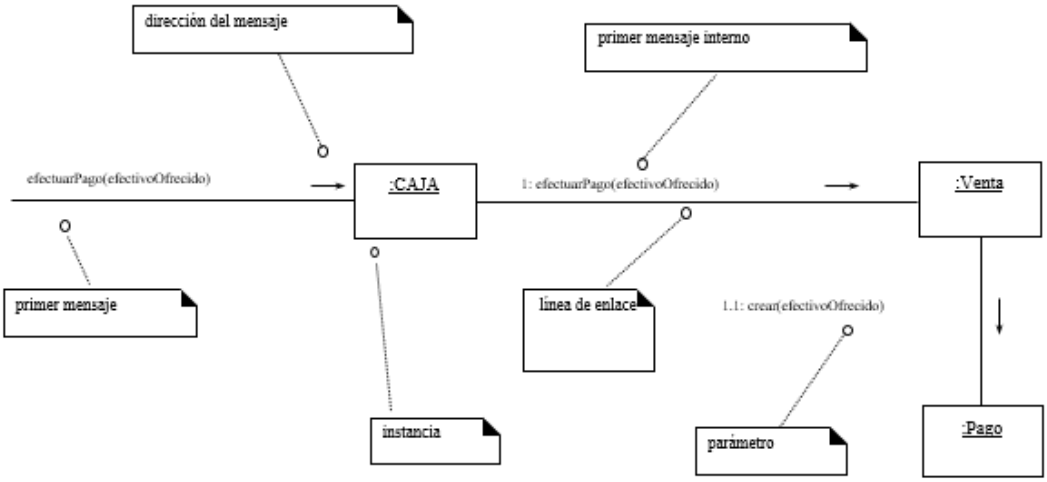


Diagrama de Colaboración

Ejemplo de diagrama de comunicación: efectuarPago



Este Diagrama de Colaboración se lee así:

1) El mensaje *efectuarPago* se envía a una instancia de *CAJA*. La instancia corresponde al mensaje *efectuarPago* de la operación del sistema.

2) El objeto *CAJA* envía el mensaje *efectuarPago* a la instancia *Venta.*

3) El objeto *Venta* crea una instancia de un *Pago.*

Los diagramas de interacción son un artefacto de gran utilidad

- Constituyen un artefacto muy importante para la estructuración de las etapas posteriores.
- El tiempo y el esfuerzo dedicados a su preparación deberían absorber un porcentaje considerable del proyecto.
- Para mejorar la calidad de su diseño, es posible aplicar patrones, principios y expresiones codificados.

Los principios de diseño necesarios para construir eficazmente los diagramas de interacción *pueden* codificarse, explicarse y aplicarse metódicamente.

Nomenclatura de los diagramas de colaboración

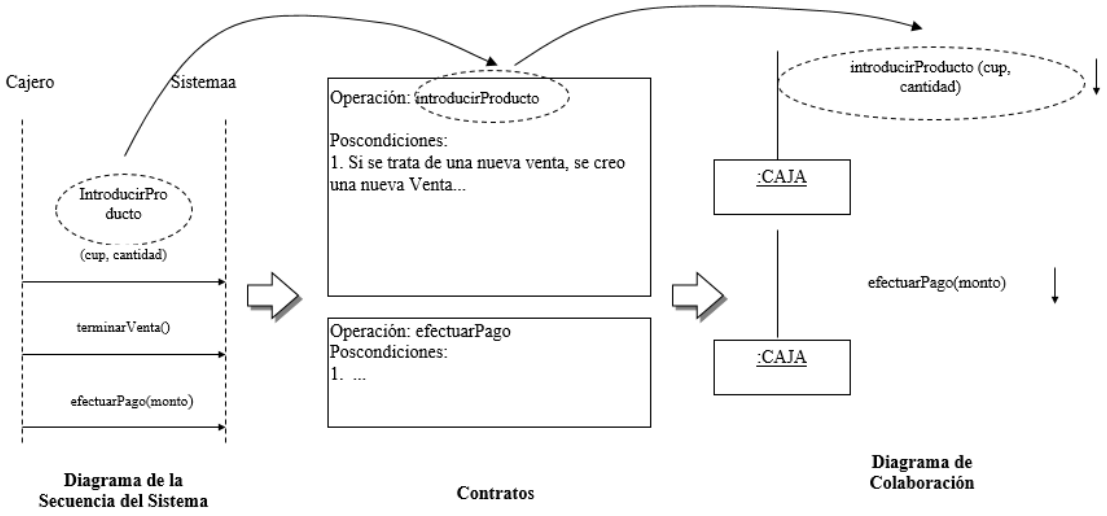
1) Elabore el diagrama por cada operación del sistema.

2) Para cada mensaje del sistema, dibuje un diagrama incluyéndolo como mensaje inicial.

3) Si el diagrama se torna complejo (por ejemplo, si no cabe en una hoja), divídalo en diagramas más pequeños.

4) Diseñe un sistema de objetos interactivos que realicen las tareas, usando como punto de partida las responsabilidades obtenidas de los contratos.

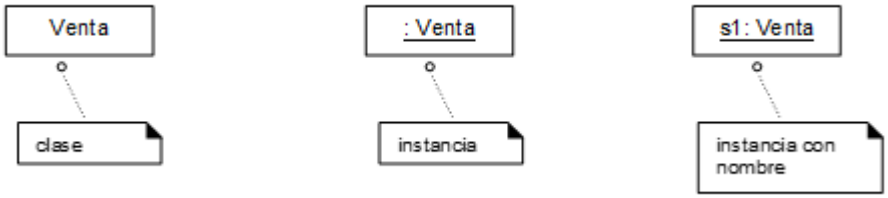
Los diagramas de colaboración y otros artefactos



Como se observa en la figura, la relación entre los artefactos incluye lo siguiente:

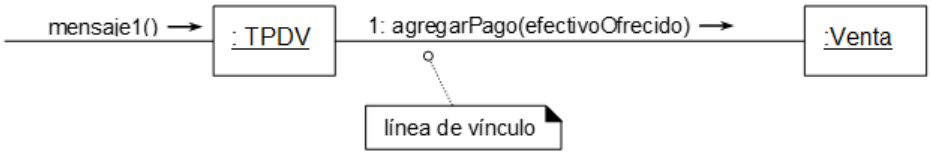
- Los casos de uso indican los eventos del sistema que se muestran explícitamente en los diagramas de secuencia.
- En los contratos se describen las operaciones del sistema.
- Las operaciones del sistema representan mensajes y éstos originan diagramas que explican gráficamente cómo los objetos interactúan para llevar a cabo las funciones requeridas (Diagramas de Colaboración).

1- Representación gráfica de las clases y de las instancias



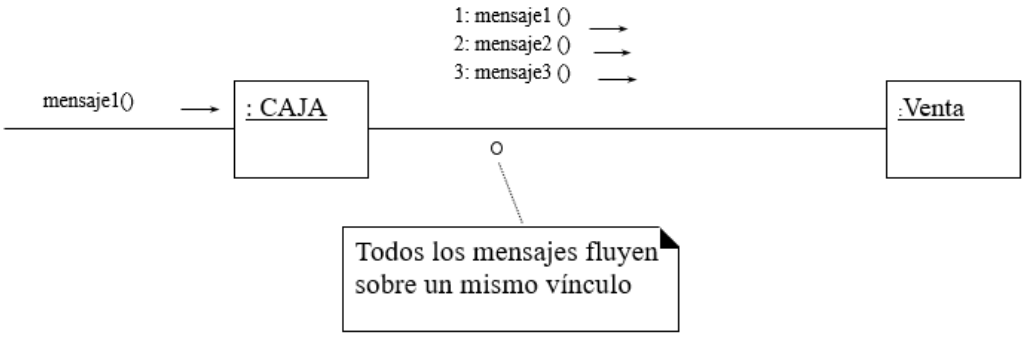
2- Representación gráfica de los vínculos

El **vínculo** (o enlace) es una trayectoria de conexión entre dos instancias; indica alguna forma de navegación y visibilidad que es posible entre las instancias. Un **vínculo** es una **instancia de una asociación**.



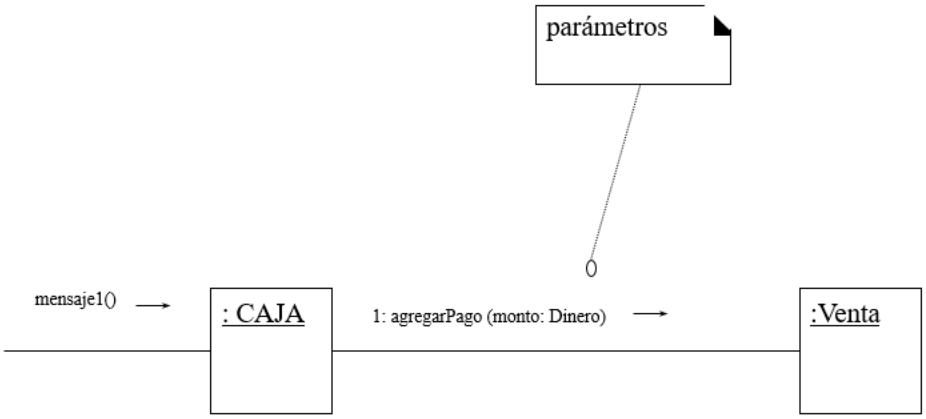
3- Representación gráfica de los mensajes

Los mensajes entre objetos pueden representarse por medio de una flecha con un nombre y situada sobre una línea de vínculo. Por un vínculo puede fluir un número indefinido de mensajes. Se agrega un número de secuencia que indique el orden consecutivo de los mensajes



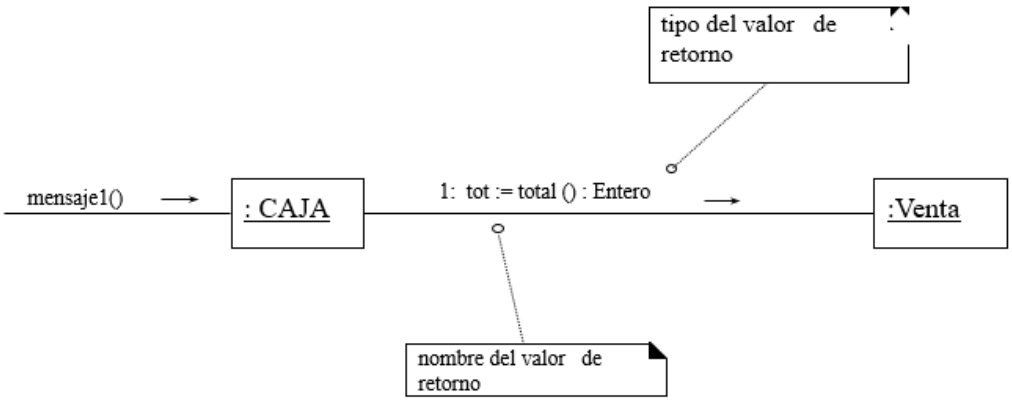
4 - Representación gráfica de los parámetros

Los parámetros de un mensaje van dentro de un paréntesis después del nombre del mensaje. Es opcional incluir el tipo de parámetro.



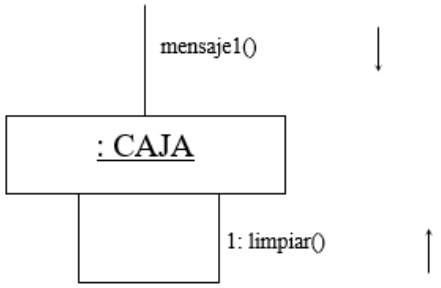
5 - Representación gráfica del mensaje de devolver valor.

Puede incluirse un valor de retorno anteponiendo al mensaje un nombre de variable de esa instrucción y un operador de asignación (":="). Es opcional mostrar el tipo del valor de retorno.

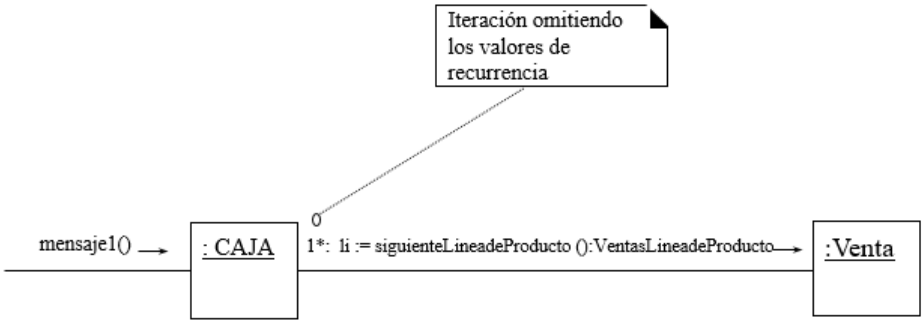


6 Sintaxis de los mensajes:
El lenguaje UML cuenta con una sintaxis estándar para los mensajes:
Retorno : mensaje (parametro : tipoparametro) : tiporetorno
Pueden usarse sintaxis de Java o Smalltalk.

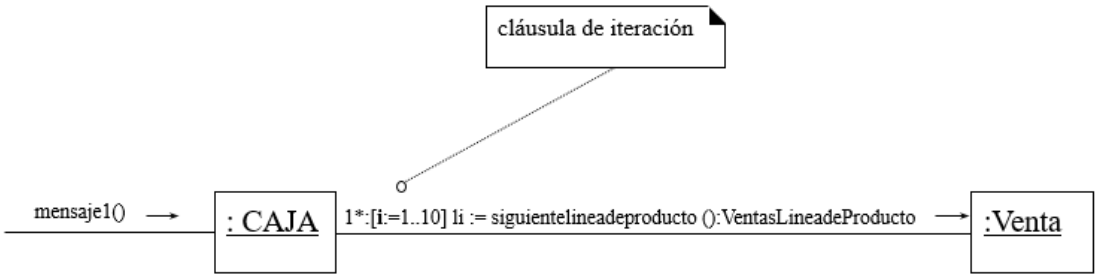
7- Representación gráfica de los mensajes al "emisor":
Puede enviarse un mensaje de un objeto a sí mismo.



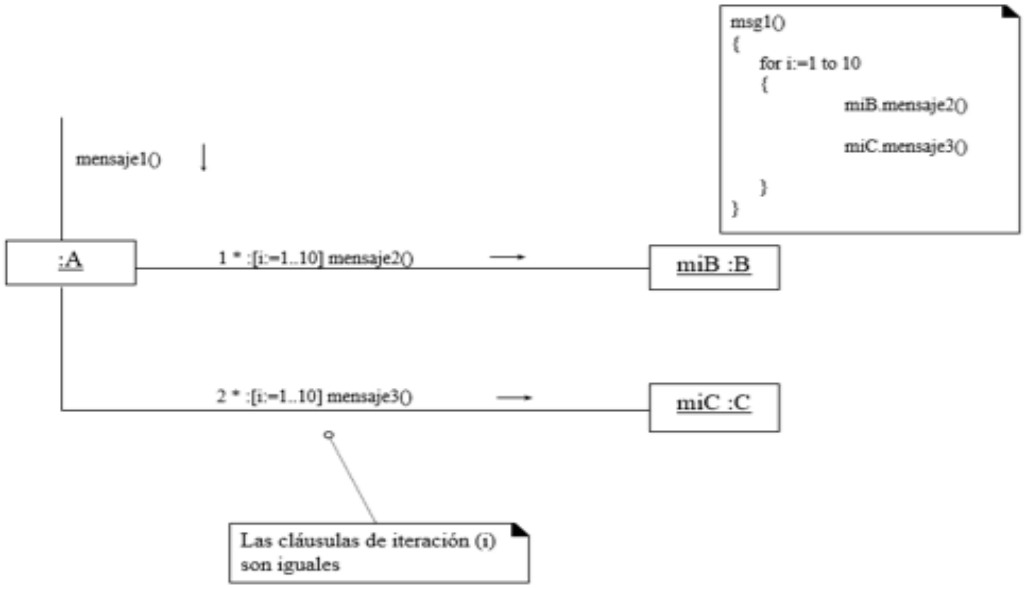
8- Representación gráfica de la iteración:
La iteración se indica posponiendo un asterisco (*) al número de secuencia.
Ese símbolo significa que, dentro de un ciclo, el mensaje va a ser enviado repetidamente al receptor



También se puede incluir una cláusula de iteración con los valores de recurrencia:



Si más de un mensaje ocurre dentro de la misma cláusula de iteración (por ejemplo, varios mensajes en un ciclo *for*), se repetirá la cláusula con cada mensaje:

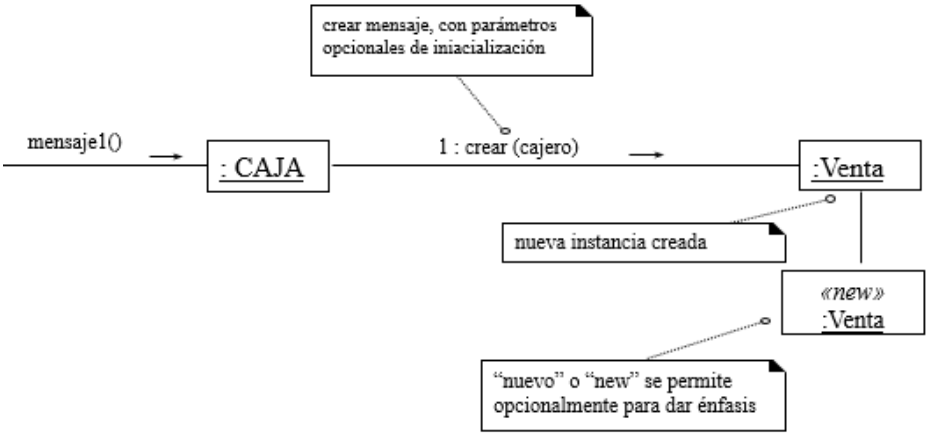


9 - Representación gráfica de la creación de instancias:

El mensaje de creación es "*crear*", que se muestra en el momento de ser enviado a la instancia que vamos a generar.

Es opcional que la nueva instancia contenga un símbolo «*nuevo*».

El mensaje *crear* puede contener parámetros, que indica la transferencia de valores iniciales.

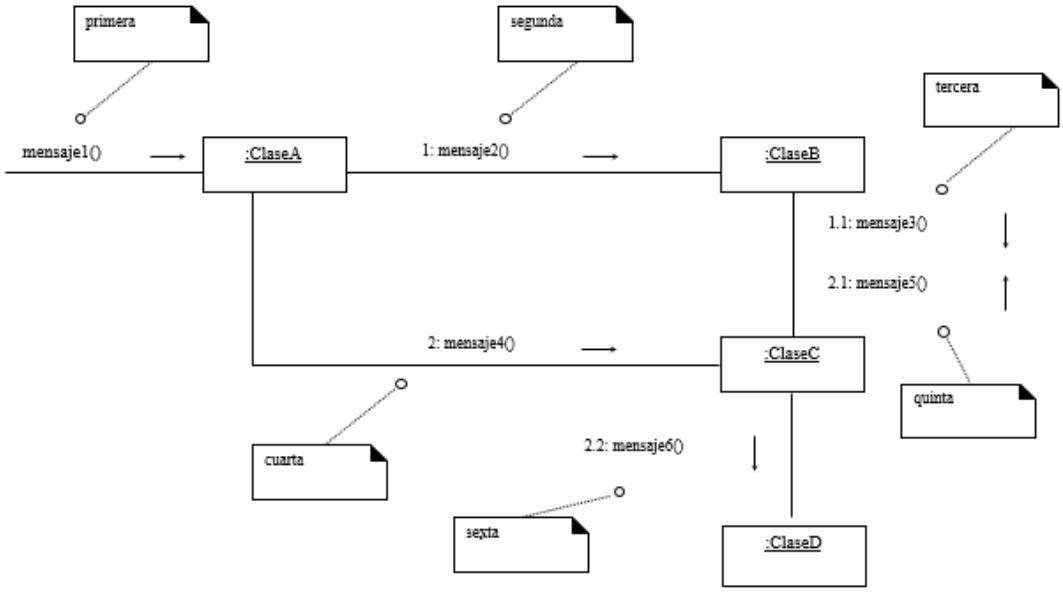


10 - Representación gráfica de la secuencia de número de los mensajes:

El orden de los mensajes se indica con un **número de secuencia**. Reglas:

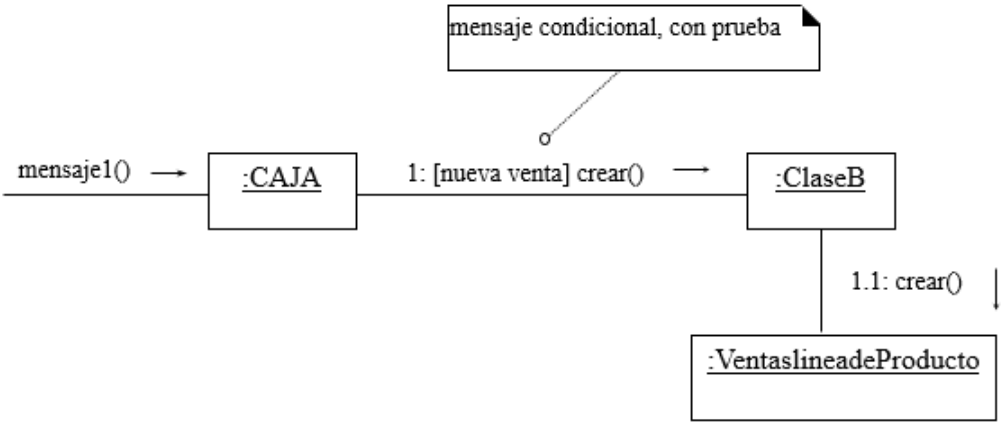
El primer mensaje no se numera.

El orden y el anidamiento de los mensajes siguientes se indican con un **esquema legal de numeración**.

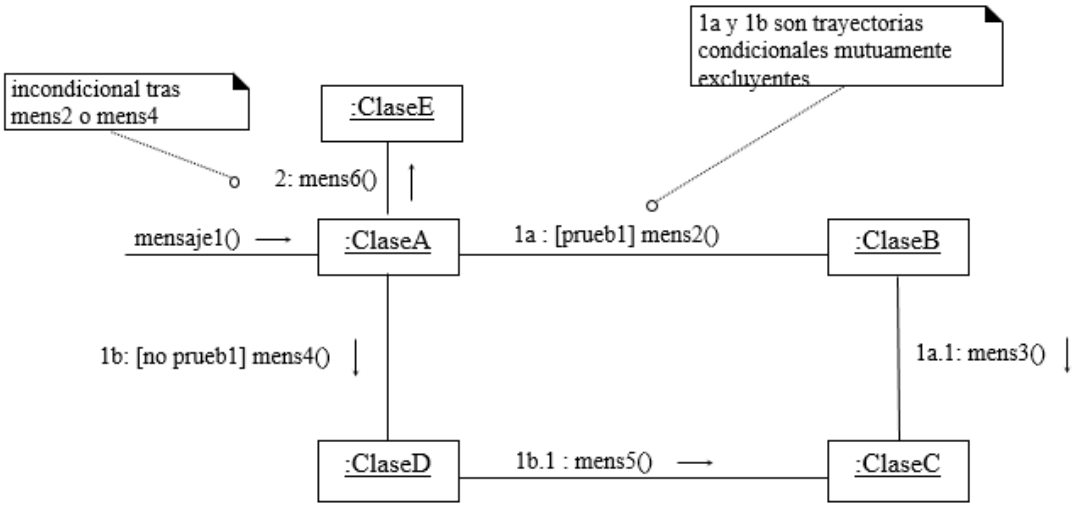


11- Representación gráfica de los mensajes condicionales:

Un mensaje condicional se indica posponiendo al número de la secuencia una cláusula condicional entre corchetes (parecido a como se hace con una cláusula de iteración). El mensaje se envía sólo si la cláusula se evalúa como *verdadera*.



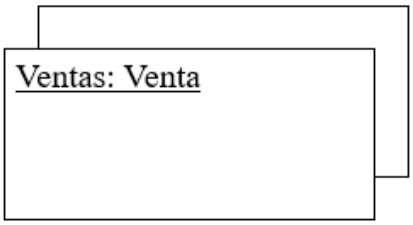
12- Representación gráfica de trayectorias condicionales mutuamente excluyentes:
Trayectorias que se excluyen mutuamente



En este caso es necesario agregar una letra en la trayectoria condicional. Por convención, la primera letra en usarse es a. Tanto 1a como 1b podrían ejecutarse después de *mens1()*. Ambas tienen el número de secuencia 1 porque pueden ser el primer mensaje interno.

Observe los subsecuentes mensajes anidados. *1b.1* es un mensaje anidado dentro de *1b*.

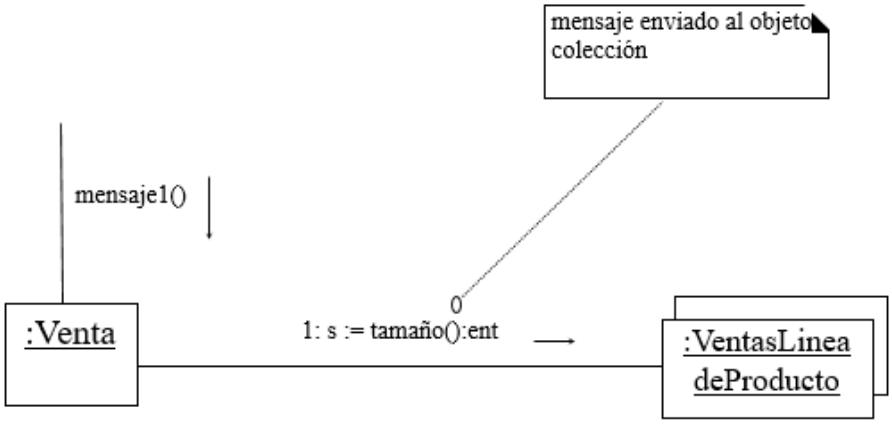
13- Representación gráfica de las colecciones:
Un **multiobjeto**, o conjunto de instancias, se dibuja como un icono de pila.



Un multiobjeto se implementa como un grupo de instancias guardadas en un contenedor u objeto colección. Representa tan sólo un conjunto lógico de instancias.

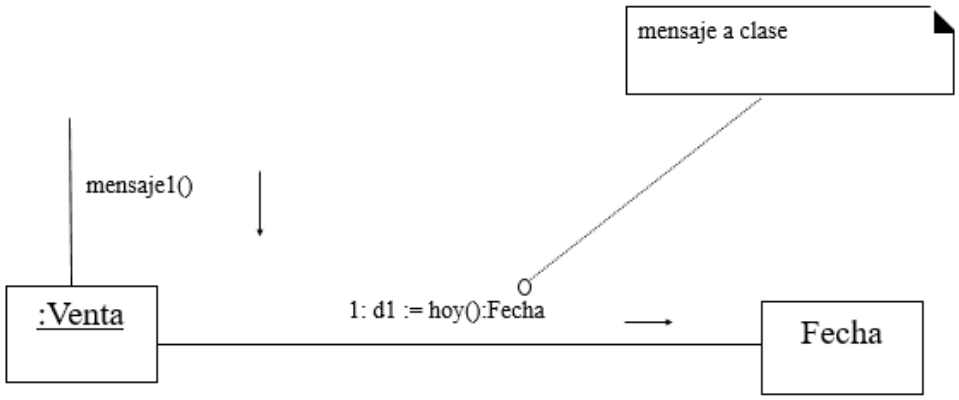
14 - Representación gráfica de mensajes dirigidos a multiobjetos:

Un mensaje dirigido a un icono de multiobjetos indica que se envía al objeto colección. No a todos sus elementos.



15 -Representación gráfica de los mensajes dirigidos a un objeto «Clase»:

Los mensajes pueden también ser dirigidos a una clase y no únicamente a una instancia:



3.2.4 Modelo de Componentes

un *diagrama de componentes* muestra los elementos de un diseño de un sistema de software. Un diagrama de componentes permite visualizar la estructura de alto nivel del sistema y el comportamiento del

servicio que estos componentes proporcionan y usan a través de interfaces. Para crear un diagrama de componentes UML, en el menú **Arquitectura**, haga clic en **Nuevo diagrama UML o de capas**.

Para ver qué versiones de Visual Studio admiten esta característica, vea Compatibilidad de versiones con las herramientas de arquitectura y modelado.

Puede usar un diagrama de componentes para describir un diseño que está implementado en cualquier idioma o estilo. Solo es necesario identificar los elementos del diseño que interactúan con los otros elementos del diseño a través de un conjunto restringido de entradas y salidas. Los componentes pueden ser de cualquier escala y pueden estar interconectados de cualquier manera.

Para obtener más información acerca de cómo se usan los diagramas de componentes en el proceso de diseño, vea Modelar la arquitectura de la aplicación.

Un componente es una unidad modular que puede reemplazarse en su propio entorno. Sus elementos internos quedan ocultos, pero tiene una o varias *interfaces proporcionadas* bien definidas a través de las cuales se puede obtener acceso a sus funciones. Un componente también puede tener *interfaces necesarias*. En una interfaz necesaria, se definen las funciones o servicios de otros componentes que son necesarios. Mediante la conexión de las interfaces proporcionadas y las interfaces necesarias de distintos componentes, puede construirse un componente mayor. Un sistema de software completo se puede concebir como un componente.

El uso de diagramas de componentes tiene algunas ventajas:

Concebir el diseño atendiendo a los bloques principales ayuda al equipo de desarrollo a entender un diseño existente y a crear uno nuevo.

Al pensar en el sistema como una colección de componentes con interfaces proporcionadas y necesarias bien definidas, se mejora la separación entre los componentes. Esto, a su vez, facilita la comprensión y los cambios cuando se modifican los requisitos.

Puede utilizar un diagrama de componentes para representar el diseño con independencia del lenguaje o plataforma que el diseño utiliza o va a utilizar.

3.2.5 Modelo de Entidades y Relaciones

El modelo entidad-relación ER es un modelo de datos que permite representar cualquier abstracción, percepción y conocimiento en un sistema de información formado por un conjunto de objetos denominados entidades y relaciones, incorporando una representación visual conocida como diagrama entidad-relación.

El diagrama de entidad-relación es la mejor forma de representar la estructura de las bases de datos relacionales (o de representar sus esquemas), ayuda a entender los datos y como se relacionan entre ellos, debe de ser completado con un pequeño resumen con la lista de los atributos y las relaciones de cada elemento.

3.2.5.1 Elementos del modelo entidad-relación

3.2.5.1.1 Entidad

Las entidades representan cosas u objetos (ya sean reales o abstractos), que se diferencian claramente entre sí.

Por ejemplo, para un taller mecánico, se pueden crear las siguientes entidades:

- Coches (objeto físico): contiene la información de cada coche.
- Empleado (objeto físico): información de los trabajadores.

- Cargo del empleado (cosa abstracta): información de la función del empleado.

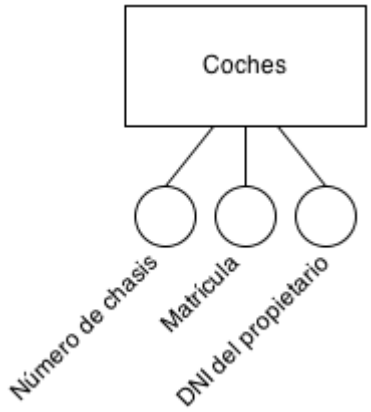
Estas entidades se representan en un diagrama con un rectángulo, como los siguientes.

3.2.5.1.2 Atributos

Los atributos definen o identifican las características de entidad (**es el contenido de esta entidad**). Cada entidad contiene distintos atributos, que dan información sobre esta entidad. Estos atributos pueden ser de distintos tipos (numéricos, texto, fecha...).

Siguiendo el ejemplo de antes los atributos de la entidad "**Coches**", dan la información sobre los coches del taller mecánico, por ejemplo; ***número de chasis***, ***matrícula***, ***DNI del propietario***, ***marca***, ***modelo*** y otros que complementen la información de cada coche.

Los atributos se representan como círculos que descienden de una entidad, y no es necesario representarlos todos, sino los más significativos, como a continuación.



En un modelo relacional (ya implementado en una base de datos) un ejemplo de tabla dentro de una ***BBDD*** podría ser el siguiente.

Número de chasis	Matrícula	DNI del propietario
5tfem5f10ax007210	4817 BFK	45338600L
6hsen2j98as001982	8810 CLM	02405068K
5rgsb7a19js001982	0019 GGL	40588860J

Este ejemplo es con tres atributos, pero un coche podría tener cientos (si fuese necesario) y seguirían la misma estructura de columnas, tras implementarlo en una ***BBDD***.

3.2.5.1.3 Relación

Es un vínculo que permite definir una dependencia entre varias entidades, es decir, permite exigir que varias entidades compartan ciertos atributos de forma indispensable.

Por ejemplo, los empleados del taller (de la entidad "**Empleados**") tienen un cargo (según la entidad "**Cargo del empleado**"). Es decir, un atributo de la entidad "***Empleados***" especificará que cargo tiene en el taller, y tiene que ser idéntico al que ya existe en la entidad "***Cargo del empleado***".

Las relaciones se muestran en los diagramas como rombos, que se unen a las entidades mediante líneas.

Como se expresaría en una tabla de la base de datos.

Empleados

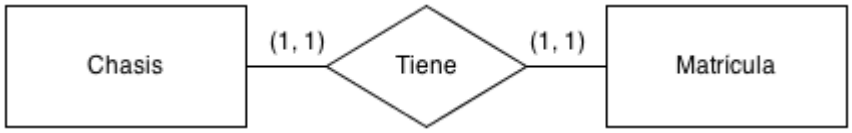
Nombre	DNI	Cargo
Carlos Sánchez	45338600L	001
Pepe Sánchez	02405068K	002
Juan Sánchez	40588860J	002

Cargo del empleado

ID del cargo	Descripción
001	Jefe de taller
002	Mecánico

3.2.5.1.4 Relaciones de cardinalidad

Es posible encontrar distintos tipos de relaciones según como participen en ellas las entidades. Es decir, en el caso anterior cada empleado puede tener un cargo, pero un mismo cargo lo pueden compartir varios empleados.
Esto complementa a las representaciones de las relaciones, mediante un intervalo en cada extremo de la relación que especifica cuantos ***objetos*** o ***cosas*** (de cada entidad) pueden intervenir en esa relación.
Uno a uno: Una entidad se relaciona únicamente con otra y viceversa. Por ejemplo, para una entidad con distintos chasis y otra con matrículas se determina que cada chasis solo puede tener una matrícula (y cada matrícula un chasis).



Uno a varios o **varios a uno**: determina que un registro de una entidad puede estar relacionado con varios de otra entidad, pero en esta entidad existir solo una vez. Como ha sido en el caso anterior del trabajador del taller.

Varios a varios: determina que una entidad puede relacionarse con otra con ninguno o varios registros y viceversa. Por ejemplo, en el taller un coche puede ser reparado por varios mecánicos distintos y esos mecánicos pueden reparar varios coches distintos.

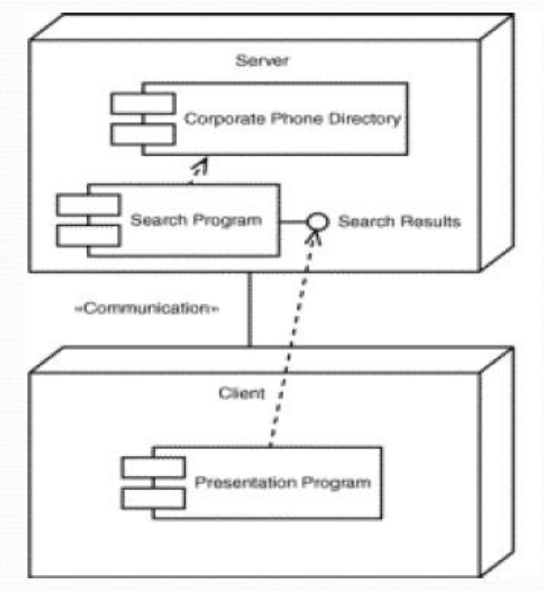
Los indicadores numéricos indican el primero el número mínimo de registros en una relación y posteriormente el máximo (si no hay límite se representa con una "n").

3.2.5.1.5 Claves

Es el atributo de una entidad, que posee una distinción de los demás registros (no permitiendo que el atributo específico se repita en la entidad) o le aplica un vínculo . Estos son los distintos tipos:
Superclave: aplica una clave o restricción a varios atributos de la entidad, para así asegurarse que en su conjunto no se repitan varias veces y así no poder entrar en dudas al querer identificar un registro.
Clave primaria: identifica inequívocamente un solo atributo no permitiendo que se repita en la misma entidad. Como sería la matrícula o el número de chasis de un coche (no puede existir dos veces el mismo).
Clave externa o clave foránea: este campo tiene que estar estrictamente relacionado con la clave primaria de otra entidad, para así exigir que exista previamente ese clave.

3.2.6 Modelo de Despliegue

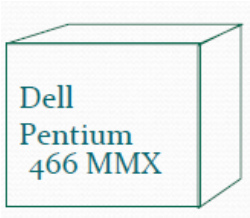
El Diagrama de Despliegue es un tipo de diagrama del Lenguaje Unificado de Modelado que se utiliza para modelar el hardware utilizado en las implementaciones de sistemas y las relaciones entre sus componentes.



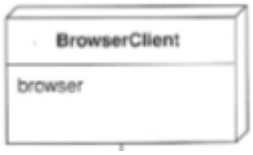
Un diagrama de despliegue muestra las relaciones físicas entre los componentes hardware y software en el sistema final, es decir, la configuración de los elementos de procesamiento en tiempo de ejecución y los componentes software (procesos y objetos que se ejecutan en ellos).

Características

- Describen la arquitectura física del sistema durante la ejecución, en términos de:
  –procesadores
  –dispositivos
  –componentes de software
- Describen la topología del sistema: la estructura de los elementos de hardware y el software que ejecuta cada uno de ellos.
- **Los nodos** son objetos físicos que existen en tiempo de ejecución, y que representan algún tipo de recurso computacional (capacidad de memoria y procesamiento):
  - –Computadores con procesadores
  - –Otros dispositivos
  - •impresoras
  - •lectoras de códigos de barras
  - •dispositivos de comunicación

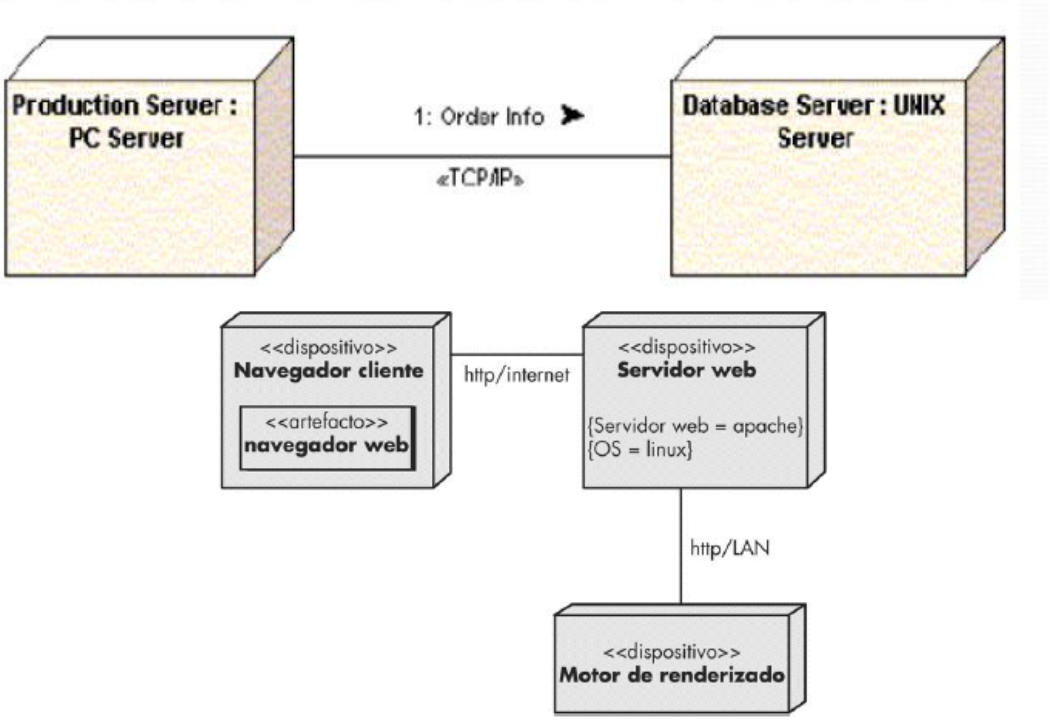


- Los dispositivos del sistema también se representan como nodos.
  Generalmente se usan estereotipos para identificar el tipo de dispositivo



Los nodos se conectan mediante asociaciones de comunicación.

•Estas asociaciones indican:
–Algún tipo de ruta de comunicación entre los nodos
–Los nodos intercambian objetos o envían mensajes a través de esta ruta.
•El tipo de comunicación se identifica con un estereotipo que indica el protocolo de comunicación o la red.



4 Patrones de Diseño

Los patrones de diseño son una técnica para resolver problemas comunes en el desarrollo de software y otros ámbitos referentes al diseño de interacción o interfaces.

Un patrón de diseño resulta ser una solución a un problema de diseño. Para que una solución sea considerada un patrón debe poseer ciertas características. Una de ellas es que debe haber comprobado su efectividad resolviendo problemas similares en ocasiones anteriores. Otra es que debe ser reutilizable, lo que significa que es aplicable a diferentes problemas de diseño en distintas circunstancias.

Tienen su origen en un campo distinto al de la informática: La Arquitectura.
En 1979 el arquitecto Christopher Alexander aportó al mundo de la arquitectura el libro The Timeless Way of Building; en él proponía el aprendizaje y uso de una serie de patrones para la construcción de edificios de una mayor calidad. Dentro de las soluciones de Christopher Alexander se encuentran cómo se deben diseñar ciudades y dónde deben ir las perillas de las puertas.
En 1987, Ward Cunningham y Kent Beck, para resolver problemas que causaban los nuevos programadores de objetos, usaron varias ideas de Alexander para desarrollar cinco patrones de interacción hombre-ordenador (HCI) y publicaron un artículo en OOPSLA-87 titulado Using Pattern Languages for OO Programs.
Pero recién en la década 1990 los patrones de diseño tuvieron un gran éxito en el mundo de la informática a partir de la publicación del libro Design Patterns escrito por el grupo Gang of Four (GoF) compuesto por Erich Gamma, Richard Helm, Ralph Johnson y John Vlisides, en el que se recogían 23 patrones de diseño comunes.

"Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, para describir después el núcleo de la solución a ese problema, de tal manera que esa solución pueda ser usada más de un millón de veces sin hacerlo ni siquiera dos veces de la misma forma."

Christopher Alexander

Patrón: <**problema, contexto, solución**>
+<recurrencia, enseña, nombre>

4.1 ¿Qué es un Design Pattern?

Un Design Pattern:

- "es una regla que expresa la relación entre un contexto, un problema y una solución" (Christopher Alexander, creador de Patterns para la Ingeniería Civil)
- "Una solución (probada) a un problema en un determinado contexto" (Erich Gamma)
- "A Design Pattern names, asbtracts and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design." (nuevamente, Erich Gamma dixit)

Partiendo de estas definiciones, definimos qué debe contener un pattern:

Un nombre que describe el problema.
Esto nos permite:

- Tener un vocabulario de diseño común con otras personas. Un pattern se transforma en una herramienta de comunicación con gran poder de simplificación
- Por otra parte, se logra un nivel de abstracción mucho mayor. De la misma manera que una lista doblemente enlazada define cómo se estructura el tipo de dato y qué operaciones podemos pedirle, el pattern trabaja una capa más arriba: define un conjunto de objetos/clases y cómo se relacionarán entre sí (resumiéndolo, en una palabra).
- Proporcionar catálogos de elementos reusables en el diseño de sistemas software.
- Evitar la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente.
- Estandarizar el modo en que se realiza el diseño.
- Facilitar el aprendizaje de las nuevas generaciones de diseñadores condensando conocimiento ya existente

El problema define cuándo aplicar el pattern, siempre que el contexto lo haga relevante. La solución contiene un template genérico de los elementos que componen el diseño, sus responsabilidades, relaciones y colaboraciones. Un pattern no es instanciable per se, la abstracción representada en el problema debe aplicarse a nuestro dominio.

En las consecuencias se analiza el impacto de aplicar un pattern en la solución, tanto a favor como en contra

4.2 ¿Qué no es un Design Pattern?

- No es garantía de un sistema bien diseñado. Tengo las respuestas, pero me falta saber si hice la pregunta correcta.

- Es un buen punto de partida para pensar una solución, no la solución. Al ser una herramienta, no puede reemplazar al diseñador, que es quien maneja la herramienta.

- De la misma manera que los estudiantes de psicología encuentran rasgos de neurosis obsesiva entre sus parientes, amigos e incluso en sí mismos, el estudiante de patterns suele querer “descubrir” patrones en su solución aun cuando no siempre se justifique. Entonces aparecen signos de sobre diseño: abuso por prever todo tipo de escenarios.

- Un pattern no es instanciable, y no es dependiente de un dominio. Es el bosquejo de una idea que ha servido en otras ocasiones, para una problemática similar. Representa una unidad de abstracción mayor a simples líneas de código.

Cómo no usarlos (Consejo de Gamma)
Cada vez que decidimos flexibilizar una parte de un sistema, se agrega complejidad al diseño y esto redunda en un mayor costo de implementación.
Aparecen fuerzas contrapuestas:

Los Design Patterns no deben ser usados indiscriminadamente. Si bien se logra mayor flexibilidad, también se agregan niveles de indirección que pueden complicar el diseño y/o bajar la performance. Un patrón de diseño sólo debería usarse cuando la flexibilidad pague su costo.

4.3 Características

Solucionar un problema: los patrones capturan soluciones, no sólo principios o estrategias abstractas
Ser un concepto probado: capturan soluciones demostradas, no teorías o especulaciones
La solución no es obvia: los mejores patrones generan una solución a un problema de forma indirecta
Describe participantes y relaciones entre ellos: describen módulos, estructuras del sistema y mecanismos complejos
El patrón tiene un componente humano significativo: todo software proporciona a los seres humanos confort y calidad de vida (estética y utilidad)

4.4 Clases de Patrones

Patrones de arquitectura: se expresa una organización o esquema estructural fundamental para sistemas software. Proporciona un conjunto de subsistemas predefinidos y sus responsabilidades.

Patrones de diseño: proporciona esquemas para refinar subsistemas o componentes de un sistema

Patrones de programación: Describe la implementación de aspectos de componentes

Patrones de análisis: prácticas que aseguran la consecución de un buen modelo de un problema y su solución

Patrones organizacionales: describen la estructura y prácticas de las organizaciones humanas

4.5 Elementos Característicos

Un patrón de diseño tiene cuatro elementos característicos:

- El **nombre** del **patrón**, describe el problema de diseño, su solución, y consecuencias en una o dos palabras. Tener un vocabulario de patrones nos permite hablar sobre ellos.
- El **problema** describe cuando aplicar el patrón. Se explica el problema y su **contexto**. Puede describir estructuras de clases u objetos que son sintomáticas de un diseño inflexible. Se incluye una lista de condiciones.
- La **solución** describe los elementos que forma el diseño, sus relaciones, responsabilidades y colaboraciones. No se describe un diseño particular. Un patrón es una plantilla.
- Las **consecuencias** son los resultados de aplicar el patrón.

4.6 Estructura de un Patrón

Para describir un patrón se usan plantillas más o menos estandarizadas, de forma que se expresen uniformemente y puedan constituir efectivamente un medio de comunicación uniforme entre diseñadores. Varios autores eminentes en esta área han propuesto plantillas ligeramente distintas, si bien la mayoría definen los mismos conceptos básicos.

La plantilla más común es la utilizada precisamente por el GoF y consta de los siguientes apartados:

- **Nombre del patrón**: nombre estándar del patrón por el cual será reconocido en la comunidad (normalmente se expresan en inglés).
- **Clasificación del patrón**: creacional, estructural o de comportamiento.
- **Intención**: ¿Qué problema pretende resolver el patrón?
- **También conocido como**: Otros nombres de uso común para el patrón.
- **Motivación**: Escenario de ejemplo para la aplicación del patrón.
- **Aplicabilidad**: Usos comunes y criterios de aplicabilidad del patrón.
- **Estructura**: Diagramas de clases oportunos para describir las clases que intervienen en el patrón.
- **Participantes**: Enumeración y descripción de las entidades abstractas (y sus roles) que participan en el patrón.
- **Colaboraciones**: Explicación de las interrelaciones que se dan entre los participantes.
- **Consecuencias**: Consecuencias positivas y negativas en el diseño derivadas de la aplicación del patrón.
- **Implementación**: Técnicas o comentarios oportunos de cara a la implementación del patrón.
- **Código de ejemplo**: Código fuente ejemplo de implementación del patrón.
- **Usos conocidos**: Ejemplos de sistemas reales que usan el patrón.
- **Patrones relacionados**: Referencias cruzadas con otros patrones

4.7 Tipos de Patrones

Los patrones se agrupan en tres grandes categorías:

- **Creacionales: abstraen el proceso de instanciación, Muestran la guía de cómo crear objetos cuando sus creaciones requieren tomar decisiones. Estas decisiones normalmente serán resueltas dinámicamente decidiendo que clases instanciar o sobre que objetos un objeto delegará responsabilidades**.

Abstract Factory: Proporciona una interfaz para **crear familias** de objetos relacionados o dependientes sin especificar su clase concreta
Builder: Permite a un objeto **construir** un objeto complejo especificando sólo su tipo y contenido
Factory Method: Define una interfaz para **crear** un objeto, dejando a las subclases decidir el tipo específico
Prototype: Permite a un objeto **crear** objetos personalizados sin conocer su clase exacta o los detalles de cómo crearlos
Singleton: Garantiza que solamente se **crea** una instancia de la clase y provee un punto de acceso global a él

- **Estructurales: se ocupan de generar estructuras entre clases y objetos, se estudian con los diagramas de clases/objetos y Describen la forma en que diferentes tipos de objetos pueden ser organizados para trabajar unos con otros**

Adapter: **convierte la interfaz** que ofrece una clase en otra esperada por los clientes
Bridge: **desacopla** una abstracción de su implementación y les permite variar independientemente
Composite: permite **construir** objetos complejos mediante **composición** recursiva de objetos similares
Decorator: **extiende la funcionalidad** de un objeto dinámicamente de tal modo que es transparente a sus clientes
Facade: **simplifica los accesos** a un conjunto de objetos relacionados proporcionando un objeto de comunicación
Flyweight: usa la **compartición** para dar soporte a un gran número de objetos de grano fino de forma eficiente
Proxy: **proporciona un objeto** con el que controlamos el acceso a otro objeto

- **Comportamiento: se encargan de la asignación de responsabilidades entre objetos y cómo se comunican entre sí, se estudian con diagramas de secuencia/colaboración. Se utilizan para organizar, manejar y combinar comportamientos.**

Chain of Responsability: evita el acoplamiento entre quien envía una petición y el receptor de la misma
Command: **encapsula una petición** de un comando como un objeto
Interpreter: dado un lenguaje define una representación para su gramática y permite **interpretar sus sentencias**
Iterator: acceso secuencial a los elementos de una colección
Mediator: define una **comunicación simplificada** entre clases
Memento: **captura y restaura** un estado interno de un objeto.
Observer: una forma de **notificar cambios** a diferentes clases dependientes
State: **modifica el comportamiento** de un objeto cuando su estado interno cambia
Strategy: define una **familia de algoritmos**, encapsula cada uno y los hace intercambiables
Template Method: define un esqueleto de algoritmo y **delega partes** concretas de un algoritmo a las subclases
Visitor: representa una operación que será realizada sobre los elementos de una estructura de objetos, permitiendo **definir nuevas operaciones** sin cambiar las clases de los elementos
sobre los que opera.

4.7.1 Creacionales

4.7.1.1 Abstract Fatory

AbstractFactory: Declara una interfaz de operaciones que crean productos abstractos
ConcreteFactory: Implementa las operaciones que crean los objetos producto
AbstractProduct: Declara una interfaz para un tipo de objeto producto
Product: Define un objeto producto que será creado por el correspondiente ConcreteFactory
Client: Usa las interfaces

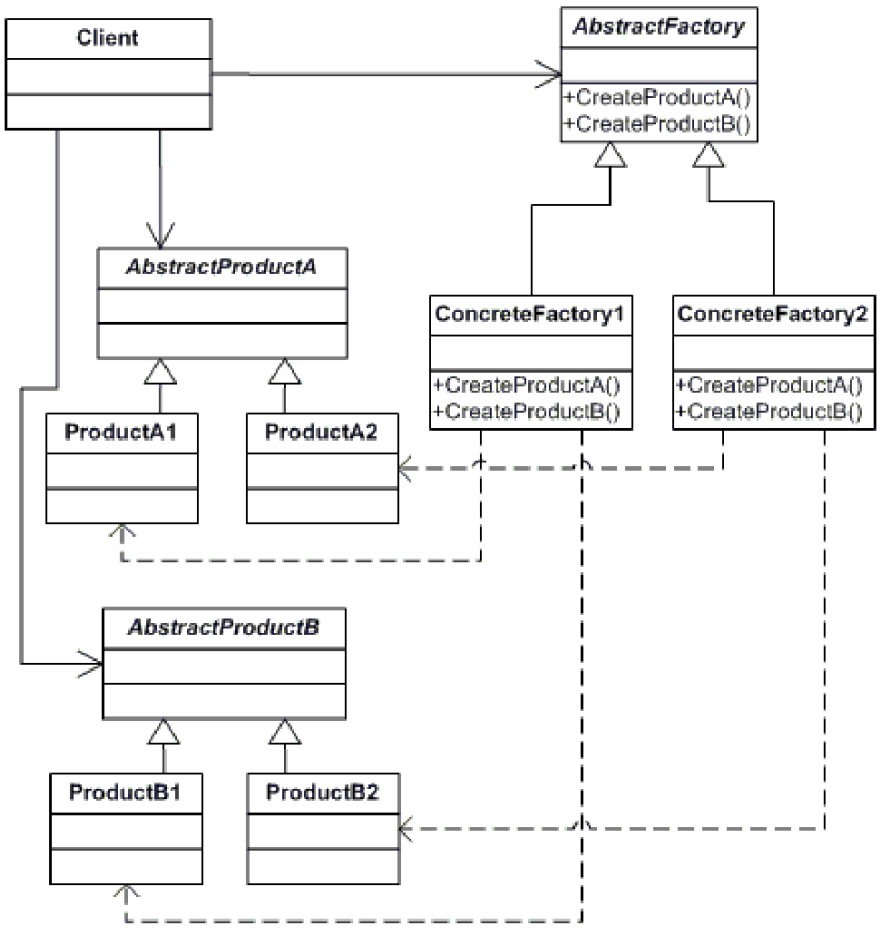


Ilustración 32 Patrón: Abstract Factory

4.7.1.2 Builder

Builder: Define una interfaz para la creación de partes de un objeto complejo
ConcreteBuilder: Implementa la interfaz de Builder, mantiene referencia del producto creado y proporciona una interfaz que retorna el producto. Ensambla las piezas constituyentes
Director: Construye un objeto usando la interfaz proporcionada por Builder

Product: Define las partes constituyentes del producto y representa un objeto complejo bajo construcción

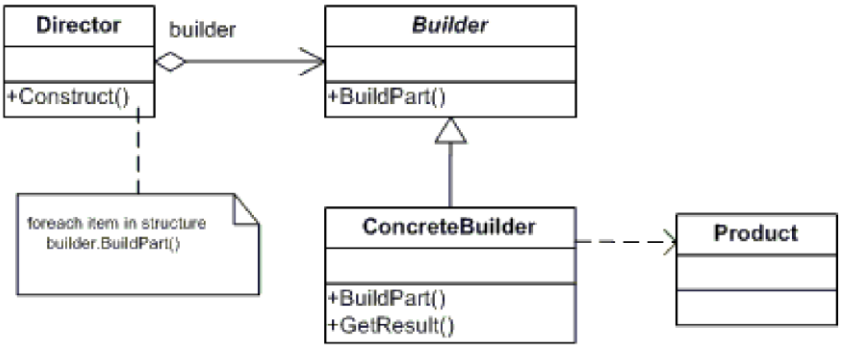


Ilustración 33 Patrón: Builder

4.7.1.3 FactoryMethod

Creator: Declara el método FactoryMethod(). Se puede definir una implementación por defecto para él
ConcreteCreator: Sobrescribe el método FactoryMethod(),devolviendo una instancia de ConcreteProduct
Product: Define la interfaz de objetos que crea el método FactoryMethod
ConcreteProduct: Implementa la interfaz del producto

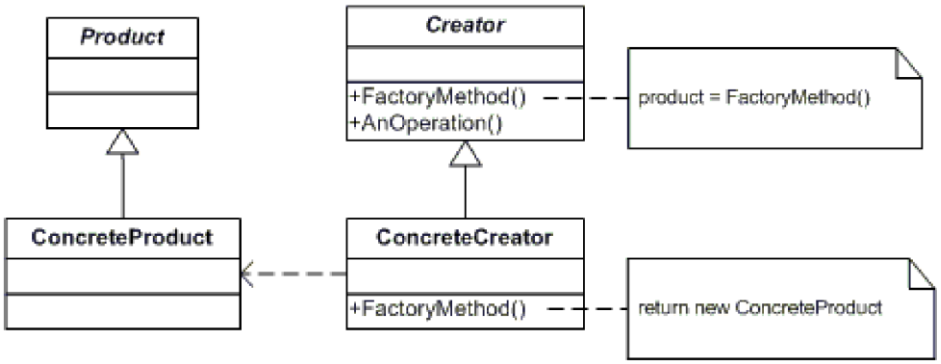


Ilustración 34 Patrón: FactoryMethod

4.7.1.4 Prototype

Prototype: Declara una interfaz definiendo el método clone()
ConcretePrototype: implementa el método clone()
Return: (Prototype) this
Client: Crea un nuevo objeto pidiendo a un prototipo que se clone

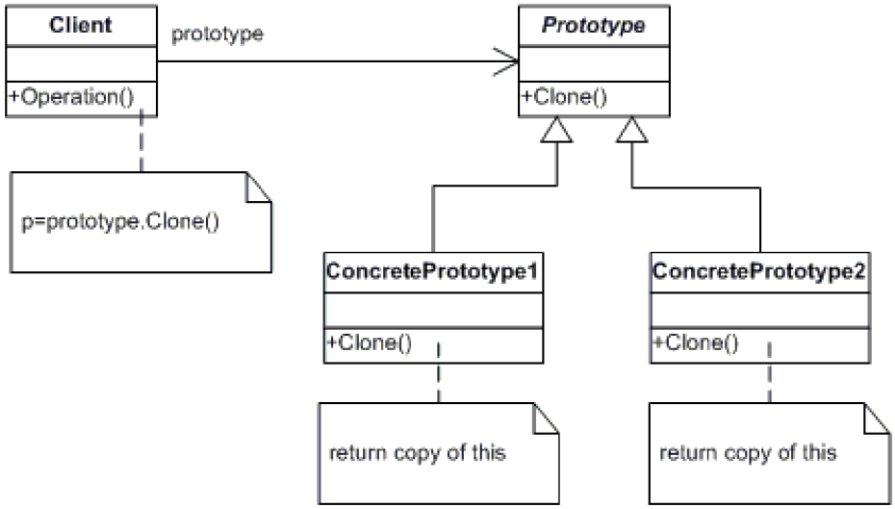


Ilustración 35 Patrón: Prototype

4.7.1.5 Singleton

Singleton: Define un método Instance() que permite al cliente el acceso a su única instancia. El método Instance() es estático. Esta misma clase es la responsable de la creación y mantenimiento de su propia instancia

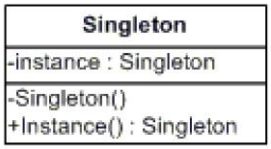


Ilustración 36 Patrón: Singleton

4.7.2 Estructurales

4.7.2.1 Adapter

Target: Define una interfaz de dominio específico que el cliente utiliza
Adapter: Adapta interfaces de Adaptee y Target
Adaptee: Define una interfaz que existiendo necesita adaptación
Client: utiliza objetos ajustándose a la interfaz ofrecida por Target

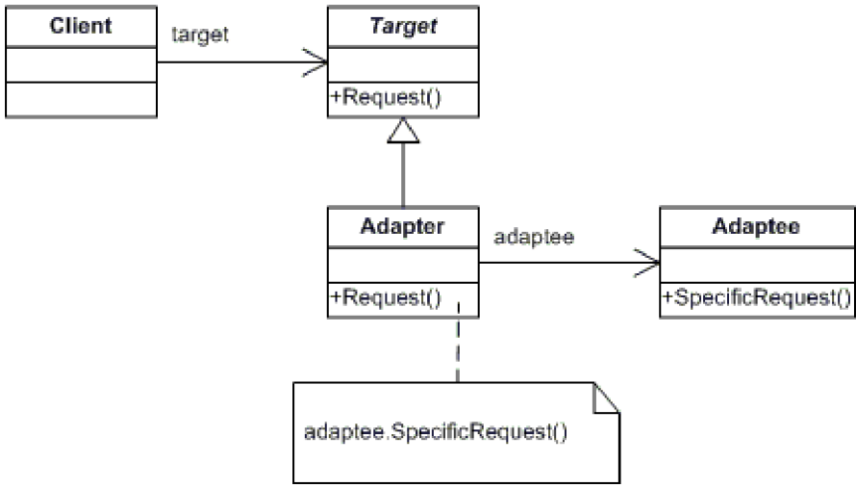


Ilustración 37 Patrón: Adapter

4.7.2.2 Bridge

Abstraction: define la interfaz de la abstracción
RefinedAbstraction: Extiende el interfaz definido por la abstracción
Implementor: define la interfaz de las clases de implementación, no tiene por qué corresponderse con el interfaz de Abstraction. Aquí se suelen proporcionar primitivas, en Abstraction operaciones de más alto nivel
ConcreteImplementor: Implementa la interfaz de Implementor

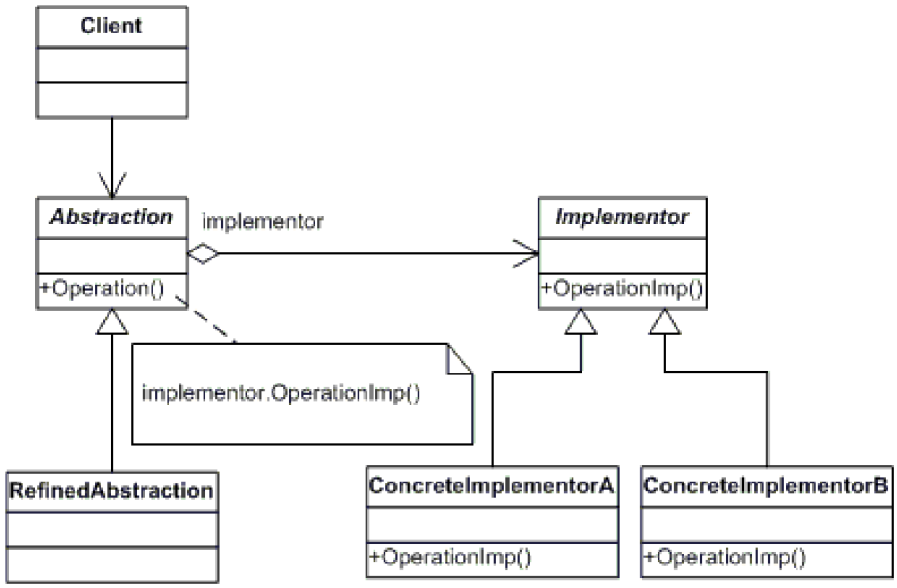


Ilustración 38 Patrón: Bridge

4.7.2.3 Composite

Component: declara la interfaz de los objetos involucrados en la composición, declara un interfaz para acceder y manejar sus hijos
Leaf: representa objetos hoja en la composición.
Composite: define el comportamiento de los componentes que tienen hijos, almacena los componentes hijo, implementa operaciones relacionadas con los hijos
Client: manipula los objetos de la composición a través de Component

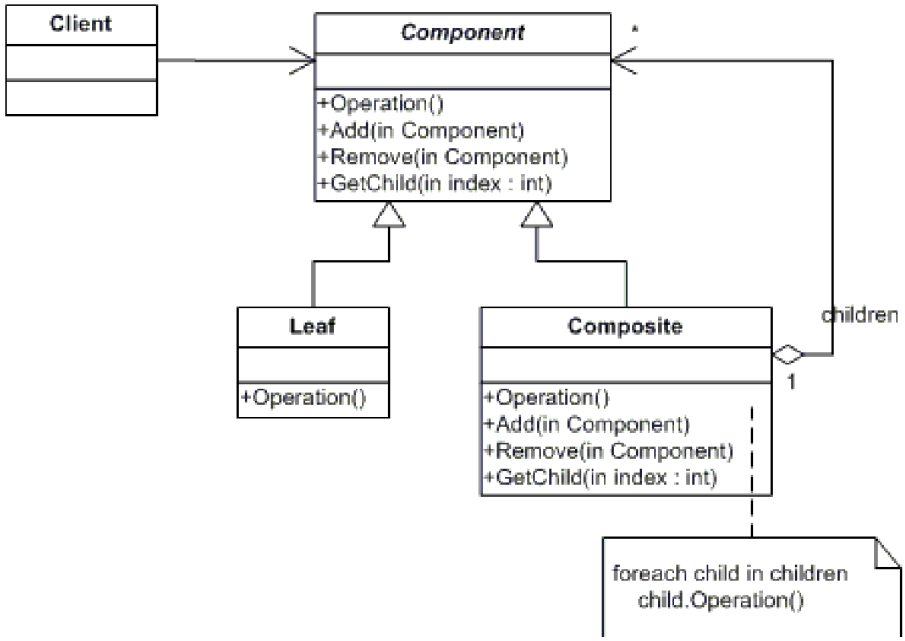


Ilustración 39 Patrón: Composite

4.7.2.4 Decorator

Component: Define la interfaz que ofrecen los objetos que poseen responsabilidades añadidas dinámicamente
ConcreteComponent: Define un objeto al que responsabilidades adicionales pueden serle añadidas
Decorator: Mantiene referencia a un objeto Component y define una interfaz que conforma el interfaz del Component
ConcreteDecorator: Añade las responsabilidades al Component

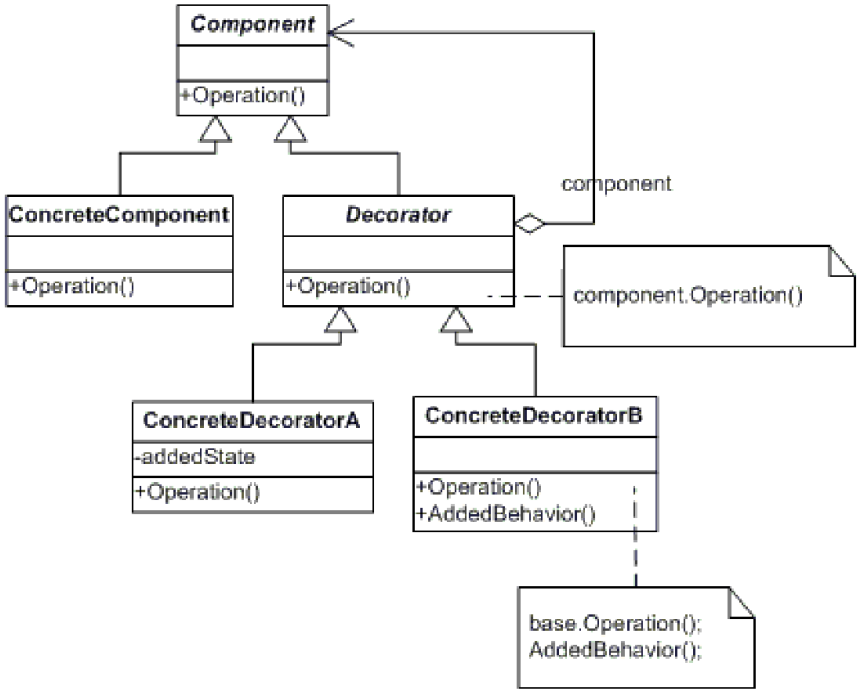


Ilustración 40 Patrón: Decorator

4.7.2.5 Facade

Facade: Conoce que clases subsistema son responsables de que peticiones y así, delega las peticiones a los objetos apropiados

Subsystem: Implementa la funcionalidad del subsistema, realiza el trabajo solicitado por el objeto Facade y no conoce, ni mantiene referencia alguna del objeto Facade

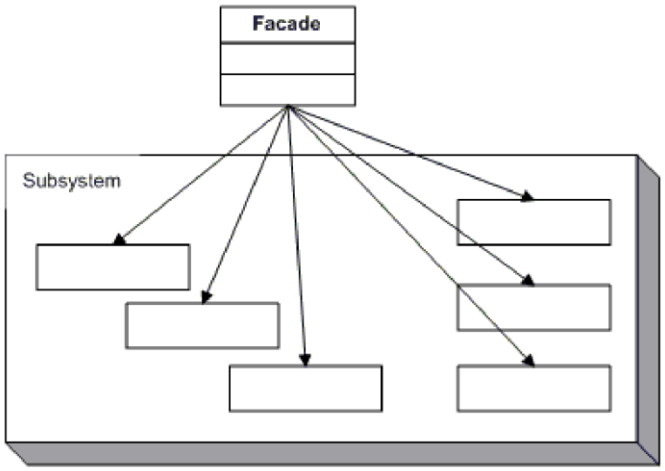


Ilustración 41 Patrón: Facade

4.7.2.6 Flyweight

Flyweight: declaran una interface a través de la que flyweights pueden recibir y actuar sobre estados externos
ConcreteFlyweight: implementa la interfaz del flyweight y añade almacenamiento para el estado interno
UnsharedConcreteFlyweight: no todas las subclases de Flyweight necesitan ser compartidas
FlyweightFactory: crea y gestiona los objetos Flyweight.
Client: mantiene una referencia a objetos flyweights.

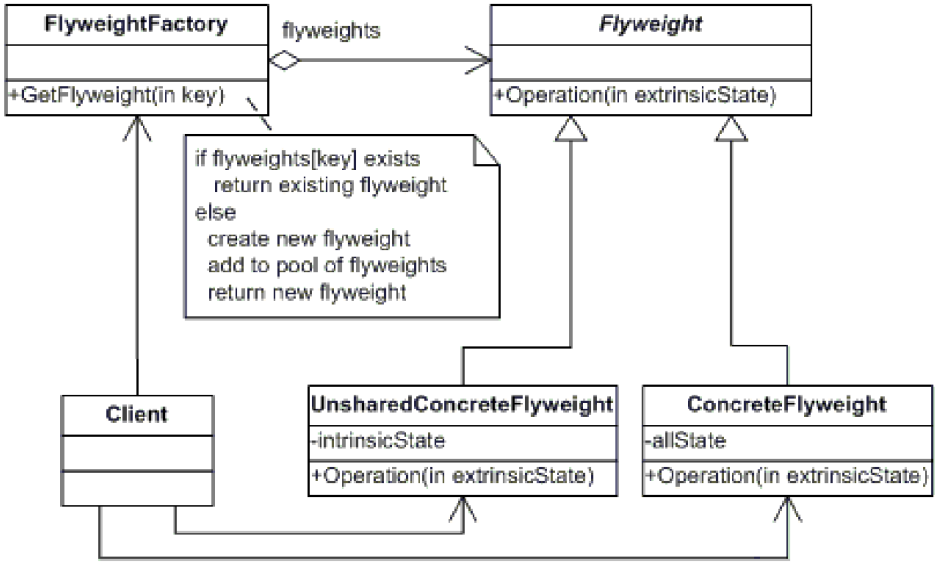


Ilustración 42 Patrón: Flyweight

4.7.2.7 Proxy

Proxy: tiene referencia al objeto RealSubject, tiene una interfaz idéntica a la de Subject así un proxy puede sustituirse por un RealSubject, y controla el acceso al RealSubject y puede ser el responsable de su creación y borrado
Subject: define una interfaz común para RealSubject y Proxy
RealSubject: define el objeto real que Proxy representa

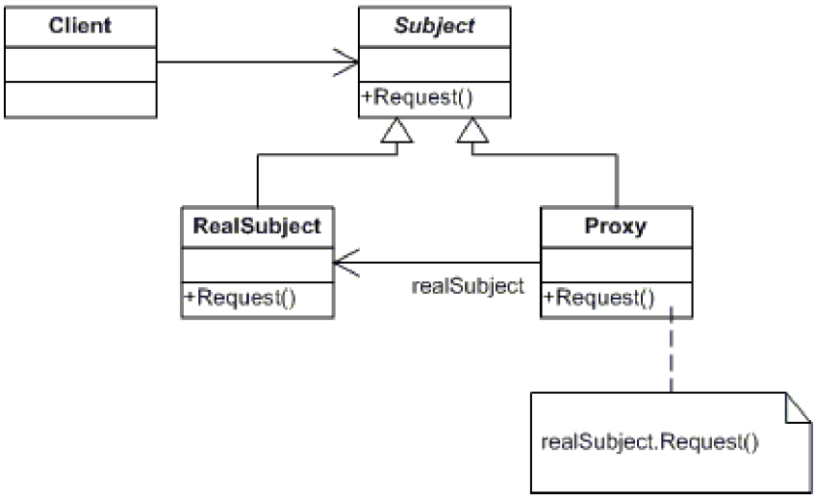


Ilustración 43 Patrón: Proxy

4.7.3 Comportamiento

4.7.3.1 Chain of Responsability

Handler: define una interfaz para manejar las peticiones, opcionalmente implementa enlaces de sucesión de atención de tales peticiones
ConcreteHandler: recibe las peticiones y las atiende si es posible, de otra forma pasa la petición a su sucesor
Client: hace las peticiones a los objetos ConcreteHandler pertenecientes a la cadena

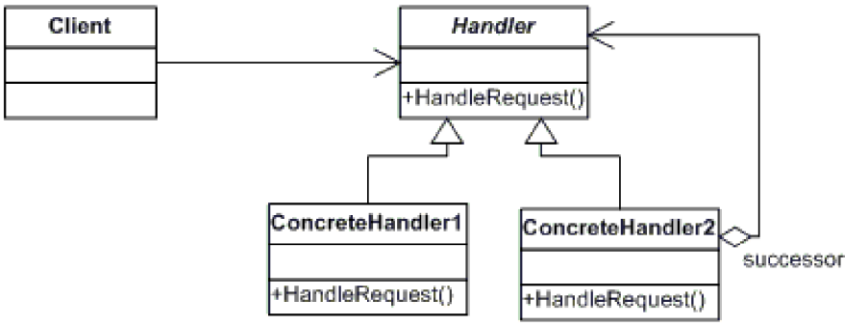


Ilustración 44 Patrón: Chain of Responsability

4.7.3.2 Command

Command: declara una interface para la ejecución de una operación
ConcreteCommand: define un vínculo entre un objeto Receiver y una acción, implementa Execute() invocando al método correspondiente de Receiver
Cliente: Crea un objeto ConcreteCommand y establece su receptor
Invoker: pide a Command que lleve a cabo su petición
Receiver: sabe cómo realizar las operaciones asociadas con la puesta en marcha de la petición

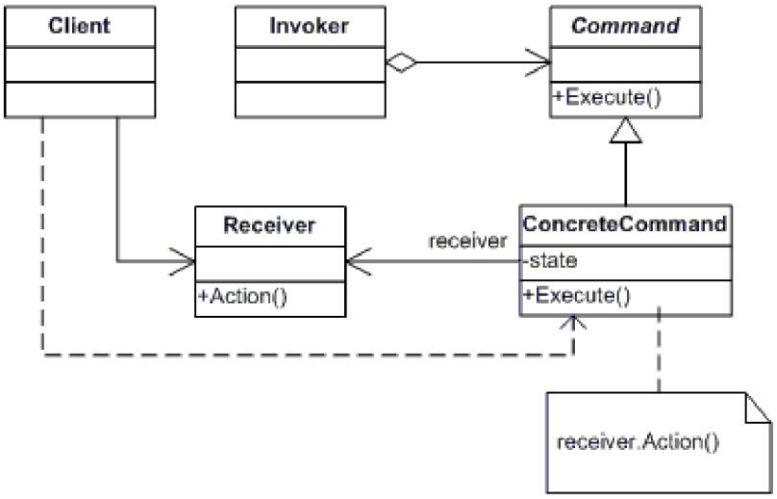


Ilustración 45 Patrón: Command

4.7.3.3 *Interpreter*

AbstractExpression: declara un interfaz para la ejecución de una operación
TerminalExpression: Implementa una operación de interpretación asociada con símbolos terminales asociados con la gramática
NonterminalExpression: implementa una operación de interpretación asociada con los símbolos no terminales asociados con la gramática
Context: contiene información global para el interprete
Client: construye un árbol sintáctico abstracto que representa una sentencia particular en el lenguaje que la gramática define

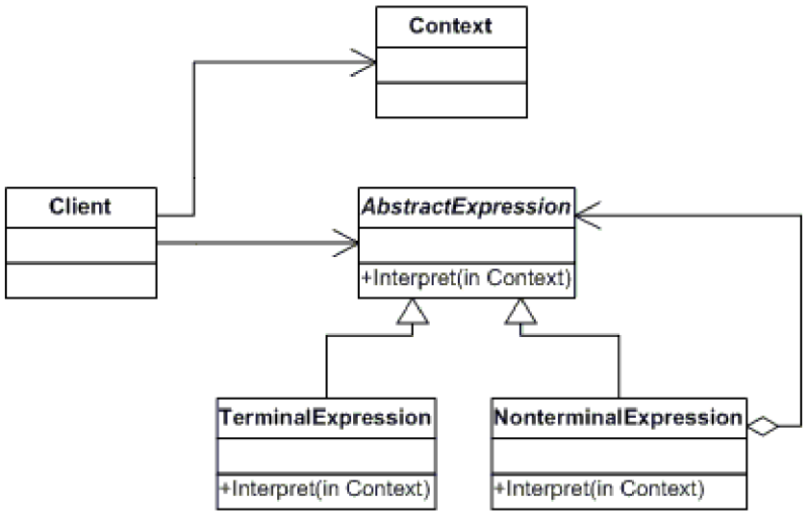


Ilustración 46 Patrón: Interpreter

4.7.3.4 *Iterator*

Iterator: define una interfaz para atravesar y acceder a los elementos
ConcreteIterator: implementa la interfaz del iterator, mantiene un puntero al conjunto de elementos

Aggregate:define un interfaz para crear un objeto iterator
ConcreteAggregator: Implementa la interfaz de creación del Iterator devolviendo un objeto de ConcreteIterator apropiado.

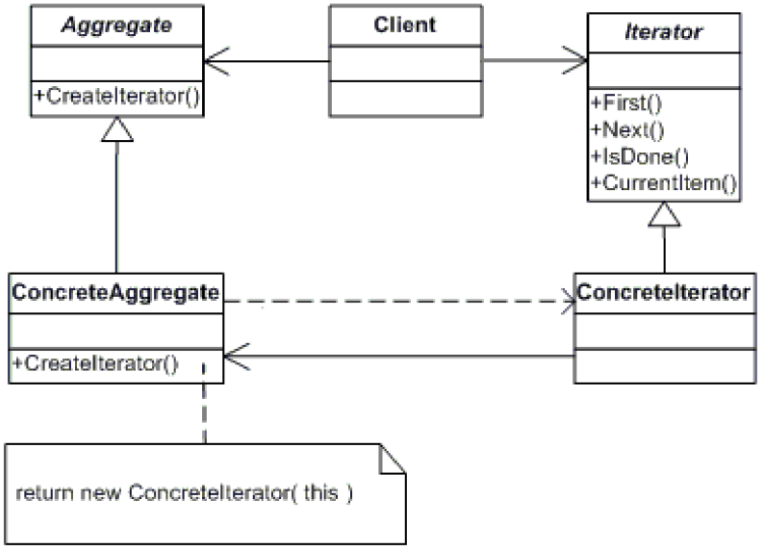


Ilustración 47 Patrón: Iterator

4.7.3.5 Mediator

Mediator: define una interfaz para la comunicación entre objetos Colleagues
ConcreteMediator: Implementa un comportamiento cooperativo coordinando objetos Colleagues
Colleague classes: cada objeto Colleague conoce su objeto Mediator, cada colleague comunica con su mediador cuando tiene que comunicar con otro colleague.

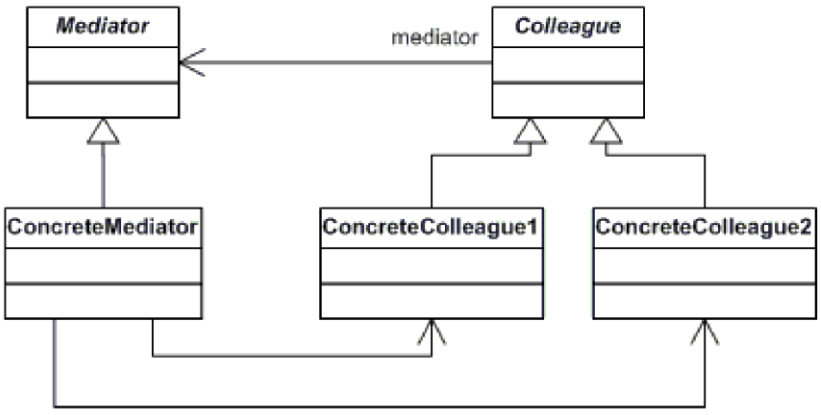


Ilustración 48 Patrón: Mediator

4.7.3.6 Memento

Memento: Almacena el estado interno del objeto Originator, protege del acceso por parte de objetos distintos al Originator
Originator: crea un objeto memento, que contiene una foto fija de su estado interno

Caretaker: es el responsable de mantener la seguridad del objeto memento, no opera o examina el contenido de memento

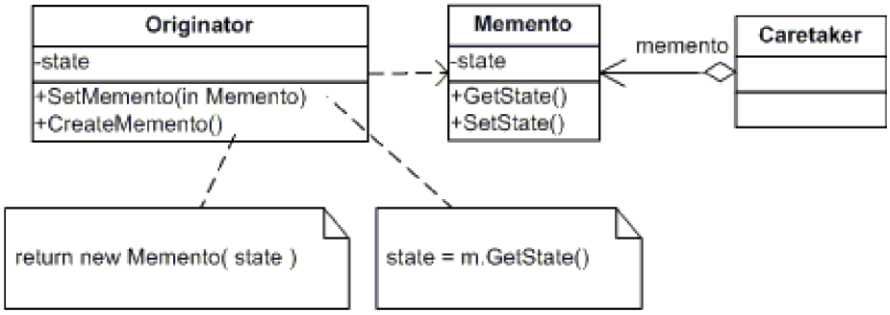


Ilustración 49 Patrón: Memento

4.7.3.7 Observer

Subject: conoce sus observadores y proporciona una interfaz para gestionarlos
ConcreteSubject: almacena el estado de interés y envía una notificación a sus observadores cuando su estado cambia
Observer: define una interfaz para los objetos a los que debe notificarse el cambio de estado en ConcreteSubject
ConcretObserver: mantiene una referencia a un objeto ConcreteSubject, mantiene información consistente relacionada con el estado implementando el método Update()

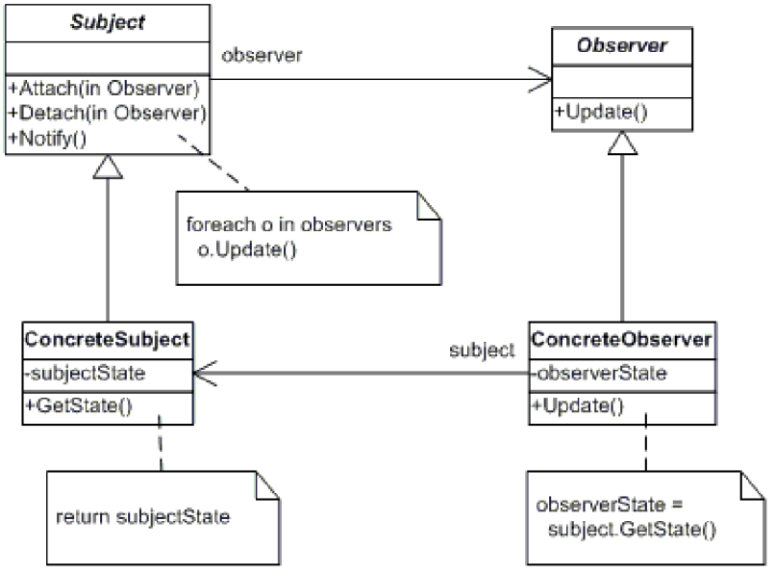


Ilustración 50 Patrón: Observer

4.7.3.8 State

Context: define la interfaz de interés a clientes
State: define una interfaz para encapsular el comportamiento asociado con un estado particular del contexto
ConcreteState: cada subclase implementa un comportamiento asociado con un estado del Context

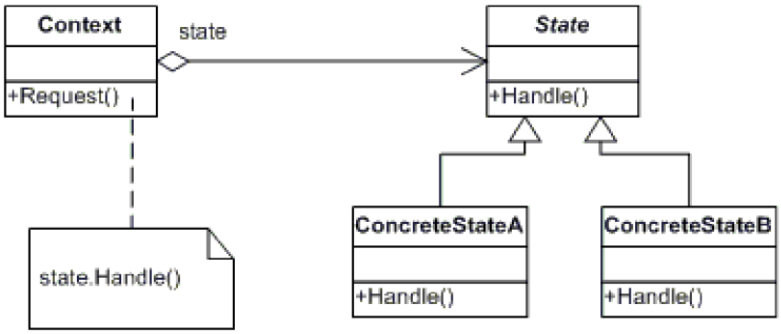


Ilustración 51 Patrón: State

4.7.3.9 Strategy

Strategy: declara un interfaz común para dar soporte a todos los algoritmos. Context utiliza esta interfaz para llamar al algoritmo definido por un ConcreteStrategy
ConcreteStrategy: Implementa el algoritmo usando la interfaz Strategy
Context: se configura con un objeto ConcreteStrategy, sobre el que mantiene una referencia, puede definir una interfaz que permita a Strategy acceder a sus datos.

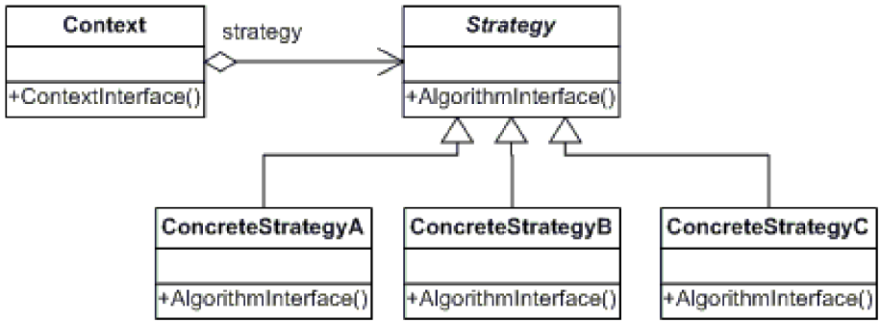


Ilustración 52 Patrón: Strategy

4.7.3.10 Template Method

AbstractClass: define operaciones primitivas abstractas cuya concreción se delega a las subclases, estas primitivas se utilizan en el cuerpo de un algoritmo esqueleto
ConcreteClass: implementa las operaciones primitivas abstractas anteriores

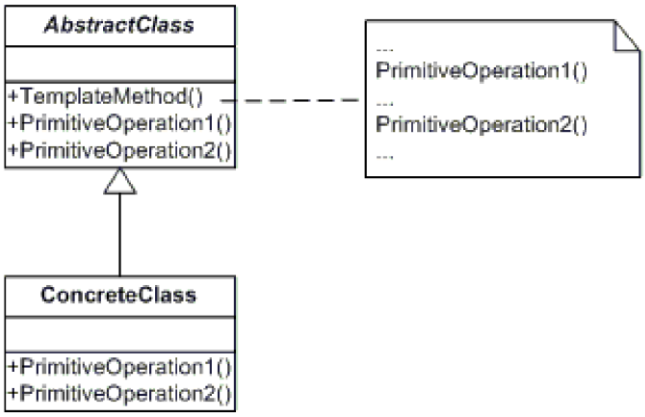


Ilustración 53 Patrón: Template Method

4.7.3.11 Visitor

Visitor: Establece las operaciones a realizar
ConcreteVisitor: implementa dichas operaciones
Element: define un método Accept() que toma un Visitor como argumento
ConcreteElement: implementa el método Accept()**ObjectStructure**:

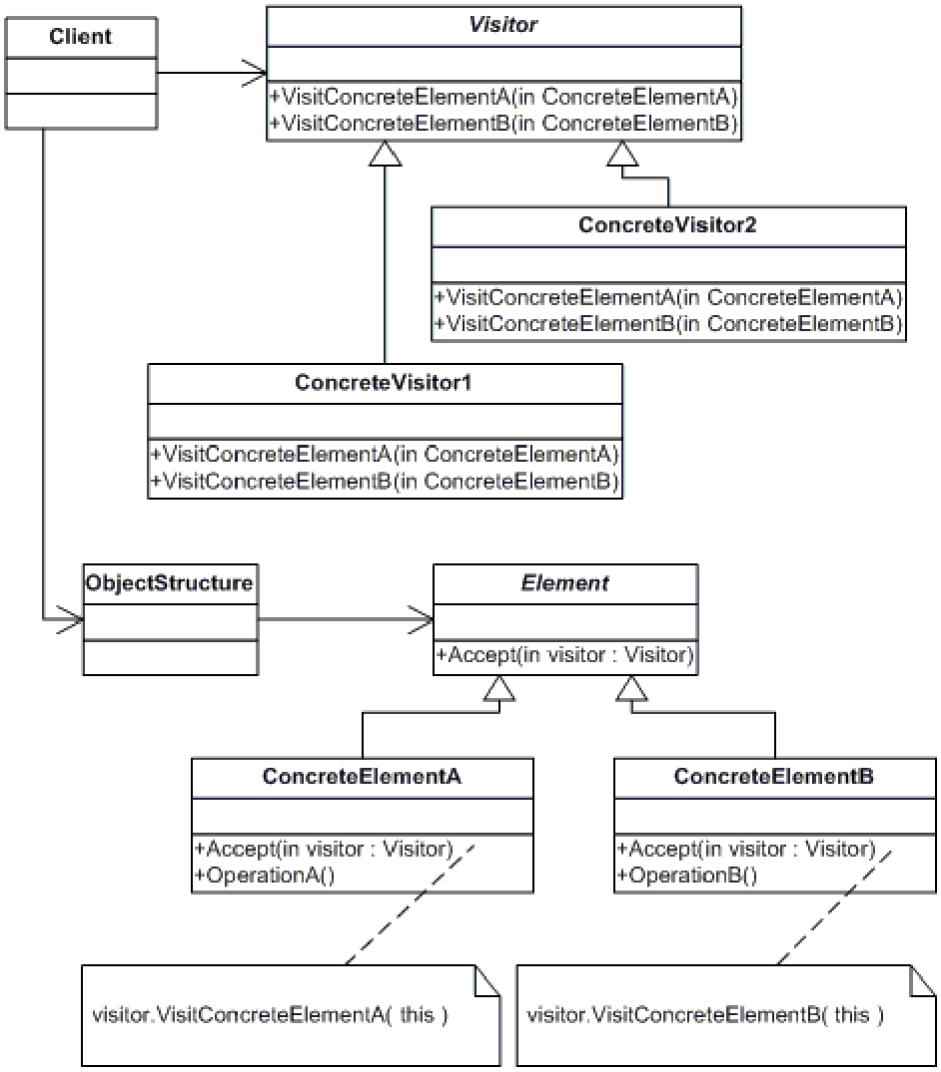


Ilustración 54 Patrón: Visitor

Bibliografía.

Balzer, R., Cheatham, T.E., Green, C., "Software technology in the 1990s: using a new paradigm", IEEE Computer, Noviembre 1983.
Belady, L.A., Lehman, M.M., "A Model of Large Program Development", IBM Systems Journal, Vol.15, Nº3, 1976, pp. 225-252.
Berry, D., "The Inevitable Pain of Software Development, Including of Extreme Programming, Caused by Requirements Volatility", International Workshop on Time-Constrained Requirements Engineering (TCRE'02), Essen, Alemania, 2002, http://www-di.inf.puc-rio.br/~julio/tcre-site/p2.pdf accedido el 28-03-2005.
Boehm, B.W., "A Spiral Model of Software Development and Enhancement", IEEE Computer, Vol.21, Nº 5, Mayo 1988, pp.61-72.
Coad P., Lefebvre E., De Luca J. Java Modeling In Color With UML: Enterprise Components and Process. Prentice Hall. 1999.
Davis, A.M., Bersoff, E.H., Comer, E.R., "A Strategy for Comparing Alternative Software Development Life Cycle Models", IEEE TSE, Vol.14, Nº10, Octubre 1988, pp.1453-1461. Reimpreso en "Software Requirements Engineering", editores Richard H. Thayer y Merlin Dorfman, IEEE Computer Society Press, 2º edición, Los Alamitos, CA, 1997, pp.408-415.
Floyd, C., "A systematic look at prototyping", en el libro Approaches to Prototyping, editor R. Buddle, Springer-Verlag, Berlín, 1984.
Jacobson I. and Ng P. W. Aspect-oriented Software Development with Use Cases. Addison Wesley
Kruchten, P., "The Rational Unified Process: An Introduction", Addison-Wesley, 3º edición, 2004.
Larman, C. 2003. Agile and Iterative Development: A Manager's Guide. AW.
Larman, C. and Basili, V. Iterative and Incremental Development. IEEE Computer. Junio, 2003.
Larman,C., Basili, V.R., "Iterative and Incremental Development: A Brief History", IEEE Computer, ISSN: 0018-9162, Junio 2003, pp.47-56.
Mills, H.D., O'Neill, D., et al., "The management of software engineering", IBM Systems Journal, Vol.24, Nº2, 1980, pp.414-477.
Royce, W.W., "Managing the Development of Large Software Systems: concepts and techniques", IEEE WESCON, Los Angeles, California, Agosto 1970.
Vienneau, R., "A Review of Formal Methods", Kaman Science Corporation, 1993, pp.3-15 y 27-33. Reimpreso en "Software Requirements Engineering", editores R.H. Thayer y M. Dorfman, IEEE Computer Society Press, 2º edición, Los Alamitos, CA, 1997, pp.324- 335.
Gamma, E., Helm, R., Johnson, R., Vlissides, "Design Patterns. Elements of Reusable Object-Oriented Software", J. Addison Wesley. 1994.
Graig Larman, "UML y Patrones. Introducción al análisis y diseño orientado a objetos.", Prentice Hall. 1999.
Roger S. Pressman, "Ingeniería del Software, Un Enfoque Práctico", Mcgraw-Hill., 2da Edición, 2010
Ian Sommervielle, "Ingeniería de Software", J. Addison Wesley. 2011
Modelo UML, Microsoft, https://msdn.microsoft.com/es-es/library/ee329484.aspx
Colección de Alexander: http://www.jacana.demon.co.uk/pattern/P0.htm
Guía de patrones de diseño: http://webs.teleprogramadores.com/patrones/

Data & object factory. Developer Training. http://www.dofactory.com/patterns/Patterns.aspx