Resumen teórico de Diseño de Sistemas

Pablo G. Gallardo

27 de noviembre de 2014

Índice

1. Unidad 1: Análisis de Información Orientado a Objetos con UML

1.1. Revisión de UML

1.2. Revisión de Proceso Unificado de Desarrollo

1.3. Análisis en el Proceso Unificado de Desarrollo

1.3.1. Objetivo

El modelo de análisis nos ayuda a refinar y estructurar los requisitos, nos permite razonar sobre los aspectos internos del sistema; incluidos sus recursos compartidos internos, nos ofrece un mayor poder expresivo y una mayor formalización debido a que se describe utilizando el lenguaje de los desarrolladores.

El modelo de análisis hace abstracciones y evita resolver algunos problemas y tratar algunos requisitos que pensamos que es mejor posponer al diseño y a la implementación.

1.3.2. Artefactos

Modelo de análisis Se presenta mediante un sistema de análisis que denota el paquete de más alto nivel del modelo.

Clase del análisis Representa una abstracción de una o varias clases y/o subsistemas del diseño del sistema.

- Se centra en el tratamiento de los requisitos funcionales y pospone los no funcionales.
- Es más conceptual, a menudo de mayor granularidad que sus contrapartidas de diseño e implementación.
- Raramente define u ofrece una interfaz en términos de operaciones.
- Atributos de alto nivel.
- Las relaciones que tiene son más que todo conceptuales.
- Encajan fácilmente en uno de tres estereotipos básicos: de interfaz, de control o de entidad.

Clases de interfaz Se utilizan para modelar la interacción entre el sistema y sus actores. Representan a menudo abstracciones de ventanas, formularios, paneles, interfaces de comunicaciones, sensores, terminales y API.

Clases de entidad Se utilizan para modelar información que posee una vida larga y que es a menudo persistente. En la mayoría de los casos, las clases de entidad se derivan directamente de una clase de entidad del negocio.

Clases de control Representan coordinación, secuencia, transacciones y control de otros objetos y se usan con frecuencia para encapsular el control de un caso de uso en concreto.

Realización de caso de uso de análisis Es una colaboración dentro del modelo de análisis que describe cómo se lleva a cabo y se ejecuta un caso de uso determinado en términos de las clases del análisis y de sus objetos del análisis en interacción.

Diagrama de clases Es un diagrama en donde se ven representadas las clases participantes en el caso de uso y sus relaciones.

Diagrama de interacción Muestran las interacciones entre objetos creando enlaces entre ellos y añadiendo mensajes a estos enlaces.

Flujo de sucesos de análisis Es un texto adicional que explique los diagramas que son difíciles de entender por sí mismos.

Requisitos especiales Son descripciones textuales que recogen todos los requisitos no funcionales sobre una realización de caso de uso. Estos pueden ser requisitos nuevos (que no se habían capturado en el flujo de requerimientos) o derivados que se encuentran a medida que avanza el trabajo de análisis.

Paquete del análisis Proporcionan un medio para organizar los artefactos del modelo de análisis en piezas manejables. Deben ser cohesivos y débilmente acoplados.
Tienen las siguientes características:

- Pueden representar una separación de intereses de análisis.
- Deberían crearse basándose en los requisitos funcionales y en el dominio del problema.
- Hay muchas probabilidades de que los paquetes se conviertan en subsistemas en las dos capas de aplicación superiores del modelo de diseño.

Paquetes de servicio Los paquetes de servicio se utilizan en un nivel más bajo de la jerarquía de paquetes del análisis para estructurar el sistema de acuerdo a los servicios que proporciona.

Descripción de la arquitectura (vista del modelo de análisis) Contiene una vista de la arquitectura del modelo de análisis que muestra sus artefactos más significativos para la arquitectura, estos son:

- Descomposición del modelo de análisis en paquetes de análisis y sus dependencias.
- Las clases fundamentales del análisis como las clases de entidad que encapsulan un fenómeno importante del dominio del problema.
- Realizaciones de casos de uso que describen cierta funcionalidad importante y crítica.

1.3.3. Trabajadores

Arquitecto Es responsable de la integridad del modelo de análisis, garantizando que este sea correcto, consistente y legible como un todo.

Ingeniero de casos de uso Es responsable de la integridad de una o más realizaciones de caso de uso; garantizando que cumplen los requisitos que recaen sobre ellos.

Ingeniero de componentes Define y mantiene las responsabilidades, atributos, realiciones y requisitos especiales de una o varias clases del análisis, asegurándose de que cada clase del análisis cumple los requisitos que se esperan de ella de acuerdo a las realizaciones de caso de uso en las que participa.

1.3.4. Actividades

Análisis de la arquitectura Su propósito es esbozar el modelo de análisis y la arquitectura mediante la identificación de paquetes del análisis evidentes y requisitos especiales comunes.

Analizar un caso de uso Lo hacemos para:

- Identificar las clases del análisis cuyos objetos son necesarios para llevar a cabo el flujo de sucesos del caso de uso.
- Distribuir el comportamiento del caso de uso entre los objetos del análisis que interactúan.
- Capturar requisitos especiales sobre la realización del caso de uso.

Analizar una clase Estos son los objetivos de analizar una clase:

- Identificar y mantener las responsabilidades de una clase del análisis, basadas en su papel en las realizaciones de caso de uso.
- Identificar y mantener los atributos y relaciones de la clase del análisis.
- Capturar requisitos especiales sobre la realización de la clase del análisis.

Analizar un paquete Objetivos:

- Garantizar que el paquete del análisis es tan independiente de otros paquetes como sea posible.
- Garantizar que el paquete del análisis cumple su objetivo de realizar algunas clases del dominio o casos de uso.
- Describir las dependencias de forma que pueda estimarse el efecto de los cambios futuros.

Normas:

- Definir y mantener las dependencias del paquete con otros paquetes cuyas clases contenidas estén asociadas con él.
- Asegurarnos de que el paquete contiene las clases correctas. Intentar hacer cohesivo el paquete incluyendo solo objetos relacionados funcionalmente.

1.4. Análisis Orientado a Objetos

1.4.1. Modelado de comportamiento en el análisis

Se emplean para visualizar, especificar, construir y documentar los aspectos dinámicos de un sistema. Estos involucran cosas tales como el flujo de mensajes a lo largo del tiempo y el movimiento físico de componentes en una red.
Así están organizados los diagramas de comportamiento de UML:

Diagrama de caso de uso Representa un conjunto de casos de uso y actores y sus relaciones. Se utilizan para describir la vista de casos de uso estática de un sistema. Son especialmente importantes para organizar y modelar el comportamiento de un sistema.

Diagrama de comunicación Es un diagrama de interacción que resalta la organización estructural de los objetos que envían y reciben mensajes. Muestra un conjunto de roles, enlaces entre ellos y los mensajes enviados y recibidos por las instancias que interpretan esos roles. Se utilizan para describir la vista dinámica de un sistema.

Diagrama de estados Representa una máquina de estados, transiciones, eventos y actividades. Se utilizan para describir la vista dinámica de un sistema. Son importantes para modelar el comportamiento de una interfaz, una clase o una colaboración.

Diagrama de actividades Muestra el flujo paso a paso en una computación. Una actividad muestra un conjunto de acciones, el flujo secuencial o ramificado de acción en acción y los valores que son producidos o consumidos por las acciones. Se utilizan para ilustrar la vista dinámica de un sistema. Son especialmente importantes para modelar la función de un sistema y resaltar el flujo de control en la ejecución de un comportamiento.

1.4.2. Modelado de estructura en el análisis

Se emplean para visualizar, especificar, construir y documentar los aspectos estáticos de un sistema. Estos incluyen la existencia y ubicación de clases, interfaces, colaboraciones, componentes y nodos.
Así están organizados los diagramas de estructura de UML:

Diagrama de clases Presenta un conjunto de clases, interfaces y colaboraciones y las relaciones entre ellas. Se utilizan para describir la vista de diseño estática de un sistema.

Diagrama de componentes Muestra las partes internas, los conectores y los puertos que implementan un componente.

Diagrama de estructura compuesta Muestra la estructura interna de una clase o una colaboración. *La diferencia entre componentes y estructura compuesta es mínima.*

Diagrama de objetos Representa un conjunto de objetos y sus relaciones. Se utilizan para describir estructuras de datos, instantáneas estáticas de las instancias de los elementos existentes en los diagramas de clases.

Diagrama de artefactos Muestra un conjunto de artefactos y sus relaciones con otros artefactos y con las clases a las que implementan. Se utilizan para mostrar las unidades físicas de implementación del sistema.

Diagrama de despliegue Muestra un conjunto de nodos y sus relaciones. Se utilizan para describir la vista de despliegue estática de una arquitectura.

1.5. Patrones Generales de Asignación de Responsabilidades (GRASP)

Los patrones GRASP constituyen un apoyo para la enseñanza que ayuda a uno a entender el diseño de objetos esencial, y aplica el razonamiento para el diseño de una forma sistemática, racional y explicable.

1.5.1. Responsabilidades y métodos

UML define una responsabilidad como "un contrato u obligación de un clasificador". Hay dos tipos de responsabilidades:

- Hacer.
  - Hacer algo por él mismo, como crear un objeto o hacer un cálculo.
  - Iniciar una acción en otros objetos.
  - Controlar y coordinar actividades en otros objetos.
- Conocer.
  - Conocer los datos privador encapsulados.
  - Conocer los objetos relacionados.
  - Conocer las cosas que puede derivar o calcular.

1.5.2. Patrones

Es un repertorio tanto de principios generales como de soluciones basadas en aplicar ciertos estilos que les guían en la creación de software.
Un patrón es una descripción de un problema y la solución, a la que se da un nombre y que se puede aplicar a nuevos contextos. Proporciona consejos sobre el modo de aplicarlo en varias circunstancias y considera los puntos fuertes y compromisos. El hecho de que tengan nombres apoya la identificación y facilita la comunicación.

1.5.3. Patrones GRASP

Son cinco:

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

Experto en Información

Problema Asignar responsabilidades a los objetos de una forma que el sistema sea fácil de entender, mantener y ampliar; y existan más oportunidades para reutilizar componentes en futuras aplicaciones.

Solución Asignar una responsabilidad al experto en información; es decir, la clase que tiene la información necesaria para realizar la responsabilidad.

Contraindicaciones En algunas ocaciones la solución que sugiere el Experto no es deseable, normalmente debido a problemas de acoplamiento y cohesión. Por ejemplo, asignar responsabilidades de almacenamiento en bases de datos.

Beneficios Se mantiene el encapsulamiento de la información y esto conlleva un bajo acoplamiento y también se estimula las definiciones de clases más cohesivas y "lijeras" que son más fáciles de entender y mantener.

Creador

Problema Asignar el responsable de la creación de una nueva instancia de alguna clase de manera que el diseño pueda soportar un bajo acoplamiento, mayor claridad, encapsulación y reutilización.

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

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

Contraindicaciones Cuando la creación requiere una complejidad significativa, como utilizar instancias recicladas por motivos de rendimiento, crear condicionalmente una instancia a partir de una familia de clases similares basado en el valor de alguna propiedad externa, etc.

Beneficios Se soporta bajo acoplamiento, lo que implica menos dependencias de mantenimiento y mayores oportunidades para reutilizar.

Bajo Acoplamiento

Problema Soportar bajas dependencias, bajo impacto del cambio e incremento de la reutilización. *Evitar* los siguientes problemas:

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

Solución Asignar una responsabilidad de manera que el acoplamiento permanezca bajo.

Contraindicaciones No suele ser un problema el acoplamiento alto entre objetos estables y elementos generalizados. Ejemplo, acoplamiento con las librerías *java.util* de Java.

Beneficios

- No afectan en las clases los cambios en otros componentes.
- Clases fáciles de entender de manera aislada.
- Clases convenientes para reutilizar.

Alta Cohesión

Problema Mantener la complejidad manejable. *Evitar* los siguientes problemas:

- Clases difíciles de entender.
- Clases difíciles de reutilizar.
- Clases difíciles de mantener.
- Clases delicadas, constantemente afectadas por los cambios.

Solución Asignar una responsabilidad de manera que la cohesión permanezca alta.

Contraindicaciones Puede ser la agrupación de responsabilidades o código en una clase o componente para simplificar el mantenimiento por una persona. Ejemplo, una clase que tenga sentencias de SQL embebidas que siguiendo otros buenos principios de diseño deberían distribuirse por diez clases.

Beneficios

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

Controlador

Problema Definir el responsable de gestionar un evento de entrada al sistema.

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

- Representa el sistema global, dispositivo o subsistema.
- Representa un escenario de caso de uso en el que tiene lugar el evento del sistema.

Beneficios

- Aumenta el potencial para la reutilización e tener interfaces conectables.
- Asegura que las operaciones del sistema tengan lugar en una secuencia válida.

2. Unidad 2: Diseño de Sistemas de Información Orientado a Objetos con UML

2.1. Definición de Diseño, principios de diseño de software orientado a objetos

2.1.1. Definición

Proceso mediante el cual se aplican varias técnicas y principios con el objetivo de definir un dispositivo, un proceso o un sistema con suficiente nivel de detalle como para permitir su realización física.
Proceso iterativo de transformar un modelo lógico en un modelo físico, teniendo en cuenta las restricciones del negocio.

2.1.2. Concepto

- Diseño es el proceso de decidir cómo se va a construir el producto final.
  - Produce especificaciones completas y detalladas.
- El seño define las partes principales del producto.
  - Cuáles son esas partes.
  - Cómo interactúan.
  - Cómo se integran.
- Los diseños imprecisos hacen perder tiempo de ingeniería.
  - Se requerirá tiempo para completar los espacios vacíos en la especificación.
  - Las decisiones pueden no ser consistentes con la vista general del sistema.
  - Las incosistencias existentes puede que se detecten recién en la integración o prueba del sistema.

2.1.3. Diferencias entre modelo de análisis y de diseño

Modelo de análisis	*Modelo de diseño*
Modelo conceptual, porque es una obstracción del sistema y permite aspectos de la implementación.	Modelo físico, porque es un plano de la implementación.
Genérico en cuanto al diseño.	No genérico, específico para una implementación.
Tres estereotipos conceptiales sobre las clases: Control, Entidad, Interfaz.	Cualquier número de estereotipos (físicos) sobre las clases, dependiendo del lenguaje de implementación.
Menos formal.	Más formal.
Menos claro de desarrollar.	Más claro de desarrollar.
Menos capas.	Más capas.
Dinámico y no muy centrado en la secuencia.	Dinámico y muy centrado en la secuencia.
Bosquejo del diseño del sistema, incluyendo su arquitectura.	Manifiesto del diseño del sistema, incluyendo su arquitectura.
Puede no estar mantenido durante todo el ciclo de vida del software.	Debe ser mantenido durante todo el ciclo de vida del software.
Define una estructura.	Da forma al sistema preservando la estructura definida.

2.1.4. Principios de diseño

Características de un buen diseño

Identificación y tratamiento de excepciones

Independencia de componentes

Prevención y tolerancia de defectos

Principios de diseño

Identificar los aspectos de la aplicación que varían y separarlas de las que son estables Hay partes del código que son más probables de que cambien con cada nuevo requerimiento del cliente, por lo tanto, es un comportamiento que debe ser separado de lo que no cambia.

Programar hacia una interfaz, no una implementación Para hacer un programa flexible es recomendable usar supertipos (Interfaces o Clases Abstractas). El punto es explotar el polimorfismo. De esta manera, el objeto en tiempo de ejecución no está ligado al código.

Favorecer la composición sobre la herencia Crear sistemas usando la composición nos da mucha flexibilidad. No solo nos deja encapsular una familia de algoritmos en distintas clases, sino también *cambiar el comportamiento en tiempo de ejecución* siempre y cuando el objeto con el que estamos "componiento" implemente la interfaz de comportamiento apropiada.

***Luchar* por obtener diseños de bajo acoplamiento entre objetos que interactúan** Diseños de bajo acoplamiento nos permite construir sistemas orientados a objetos flexibles que pueden lidiar con cambios de manera eficiente porque minimizan la interdependencia entre objetos.

Las clases deben estar abiertas para la extensión y cerradas para la modificación El propósito es permitir la expansión de las clases para que puedan incorporar nuevos comportamientos sin modificar el código existente. De esta manera, se evitan errores que se puedan introducir al código que ya ha sido probado y optimizado.

Depender de abstracciones, no de clases concretas Esto sugiere que nuestros componentes de alto nivel no deberían depender de los de bajo nivel. Ambos deberían depender de abstracciones.

Principio del menor conocimiento - Habla solo con tus amigos inmediatos Este principio nos ayuda a prevenir diseños que tienen muchas clases acopladas que, si cambian en una parte del sistema, tiene que haber cambios en cascada a las partes subsiguientes. El principio dice que solo deberíamos invocar un método en un objeto si pertenece a:

- El objeto mismo.
- Objetos pasados por parámetros al método.
- Cualquier objeto en donde el método crea instancias.
- Cualquier componente del objeto.

El principio de Hollywood - No nos llames, nostros te llamaremos Nos ayuda a prevenir la degradación de dependencias. Esto pasa cuando tenemos componentes de alto nivel dependiendo de componentes de bajo nivel y estos, a su vez, dependen de componentes de alto nivel y estos dependen de otros componentes del mismo nivel y estos dependen de otros componentes de bajo nivel y así sucesivamente. Con este principio permitimos que los componentes de bajo nivel se relaciones entre sí formando un sistema y que los componentes de alto nivel determinen cuándo son necesarios y cómo.

Principio de Responsabilidad Única - Una clase debería tener solamente una razón para cambiar Cuando se modifica el código de una clase se incrementan las oportunidades de introducir problemas. Teniendo dos razones para cambiar una clase hay más probabilidades de que este cambio introduzca problemas a los dos aspectos del diseño de la misma. Este principio nos guía para asignar cada responsabilidad a una clase y solo una clase.

No te repitas Evitar la duplicación del código abstrayendo lo común y poniéndolo en un solo lugar

Principio de Liskov Los subtipos deben ser sustituibles por sus tipos base. Este principio nos ayuda a plantear buenas herencias.

2.2. Aspectos que se diseñan en un sistema de información

Diseño arquitectónico Define la relación entre los principales elementos estructurales del programa.

Diseño de datos Transforma los requerimientos en las estructuras de datos necesarias para implementar el software.

Diseño de los procesos Transforma elementos estructurales de la arquitectura del programa en una descripción procedimental de los componentes del software.

Diseño de la interfaz Describe cómo se comunica el software consigo mismo, con los sistemas que operan en él, con los dispositivos externos y con los usuarios.

Diseño de formas de entrada/salida Describe cómo se ingresa información al software y cómo se presentarán las salidas del mismo.

Diseño de procediminetos manuales Describe cómo integra el software al Sistema de Negocio.

2.3. Estrategias de Prototipado y de Ensamblaje de Componentes

2.3.1. Estrategia de prototipado

Definición Es una elección del modelo de proceso que se recomienda elegir a la hora de implementar un proyecto complejo, con dominio no familiar, que utilizará una tecnología desconocida; de ahí que surge la necesidad de requirirse el uso de prototipos en el diseño y la implementación, además de utilizarlos durante la validación de requerimientos.

- Es un modo de desarrollo de software.
- El prototipo es una aplicación que funciona.

- La finalidad del prototipo es probar varias suposiciones formuladas por analistas y usuarios con respecto a las características requeridas del sistema.
- Los prototipos se crean con rapidez, evolucionan a través de un proceso interactivo y tienen un bajo costo de desarrollo.

Concepto

- Primera versión de un nuevo tipo de producto, en el que se han incorporado solo algunas características del sistema final, o no se han realizado completamente.
- Modelo o maqueta del sistema que se construye para complender mejor el problema y sus posibles soluciones:
  - Evaluar mejor los requisitos.
  - Probar opciones de diseño.

Características de los prototipos

- Funcionalidad limitada.
- Poca fiabilidad.
- Características de operación pobres.
- Prototipo aproximado de 10 % del presupuesto del proyecto.

Objetivo

- Aclarar los requerimientos de los usuarios.
- Verificar la factibilidad del diseño del sistema.

Cuándo es conveniente usarlos Siempre es conveniente pero especialmente cuando:

- El Área de aplicación no está bien definida.
- El costo del rechazo por parte de los usuarios, por no cumplir sus expectativas, es muy alto.
- Es necesario evaluar previamente el impacto del sistema en los usuarios y en la organización.
- Se usan nuevos métodos, técnicas, tecnología.
- No se conocen los requerimientos o es necesario realizar una evaluación de los requerimientos.
- Cuando los costos de inversión son altos.
- Cuando hay factores de riesgo alto asociados al proyecto.

Uso de los prototipos
Para el cliente:

- Ayuda a establecer claramente los requisitos.

Para los desarrolladores:

- Validar corrección de la especificación.
- Aprender sobre problemas que se presentarán durante el diseño e implementación del sistema.
- Mejorar el producto.
- Examinar viabilidad y utilidad de la aplicación.

Beneficios

- Aumentar la productividad.
- Desarrollo planificado.
- Entusiasmo de los usuarios respecto a los prototipos.

Tipos de prototipos

Prototipado de interfaz de usuario: Modelos de pantallas.

Prototipado funcional (Operacional): Implementa algunas funciones y a medida que se comprueba que son las apropiadas, se ocrrigen, refinan y se añaden otras.

Modelos de rendimiento: Evalúan el rendimiento de una aplicación crítica (no sirven al análisis de requisitos).

Desde el punto de vista de la utilidad, los prototipos pueden construirse:

- Rápido o desechable:
  - Sirve al análisis y validación de los requisitos.
  - Después se redacta la especificación del sistema y se desecha el prototipo.
  - La aplicación se desarrolla siguiendo un paradigma diferente.
  - Puede ser un problema cuando el prototipo no se desecha y termina convirtiéndose en el sistema final.
- Evolutivos
  - Comienza con un sistema relativamente simple que implementa los requisitos más importantes o mejor conocidos.
  - El prototipo se aumenta o cambia en cuanto se descubren nuevos requisitos.
  - Finalmente, se convierte en el sistema requerido.
  - Actualmente se usa en el desarrollo de sitios Web y en aplicaciones de comercio electrónico.

Con respecto al alcance, los prototipos pueden clasificarse en:

Vertical Desarrolla completamente alguna de las funciones.

Horizontal Desarrolla parcialmente todas las funciones.

Etapas del método con prototipos

1. Identificación de requerimientos conocidos.
2. Desarrollo de un modelo de trabajo.
3. Participación del usuario.
4. Revisión del prototipo.
5. Iteración del proceso de refinamiento.

Estrategias para el desarrollo de prototipos

Prototipos para pantallas El elemento clave es el intercambio de información con el usuario.

Prototipos para procedimientos de procesamiento El prototipo incluye solo procesos sin considerar errores.

Prototipos para funciones básicas Solo se desarrolla el núcleo de la aplicación, es decir solo los procesos básicos.

Tareas para los usuarios

- Identificar la finalidad del sistema.
- Describir la salida del sistema.
- Describir los requerimientos de datos.
- Utilizar y evaluar el prototipo.
- Identificar las mejoras necesarias.
- Documentar las características no deseables.

Críticas

- El cliente cree que es el sistema funcional.
- Peligro de familiarización con malas elecciones iniciales.
- Difícil de administrar: se necesita mucha experiencia.
- Alto costo

2.3.2. Estrategia de Ensamblaje de Componentes

- Es un modelo evolutivo de desarrollo de software.
- El modelo de ensamblaje de componentes incorpora muchas de las características del modelo en espiral.
- Es evolutivo por naturaleza y exige un enfoque iterativo para la creación del software.
- Configura aplicaciones desde componentes preparados de software (COTS).

En qué consiste

- Identificar las clases candidatas. Se lleva a cabo examinando los datos que se van a manejar por parte de la aplicación y el algoritmo que se va a aplicar para conseguir el tratamiento.
- Las clases creadas en los proyectos de ingeniería del software anteriores se almacenan en una biblioteca de clases.
- La biblioteca de clases se examina para determinar si estas clases ya existe.

Niveles

- De aplicación, en el que una aplicación completa se integra con el prototipo.
- De componente, en el que los componentes se integran en un marco de trabajo estándar.

Beneficios Según estudios de reutilización, el ensamblaje de componentes lleva a una reducción del 70 % de tiempo de ciclo de desarrollo, un 84 % del coste del proyecto y un índice de productividad del 26,2 %.

Ventajas de usar el Desarrollo de software basado en componentes

Reutilización del software Nos lleva a alcanzar un mayor nivel de reutilización de software.

Simplifica las pruebas Permite que las pruebas sean ejecutadas probando cada uno de los componentes antes de probar el conjunto completo de componentes ensamblados.

Simplifica el mantenimiento del sistema Cuando existe un débil acoplamiento entre componentes, el desarrollador es libre de actualizar y/o agregar componentes segñun sea necesario sin afectar otras partes del sistema.

Mayor calidad Dado que un componente puede ser construido y luego mejorado continuamente por un experto u organización, la calidad de una aplicación basada en componentes mejorará con el paso del tiempo.

Ventajas de usar componentes de terceros:

Ciclos de desarrollo más cortos La adicción de una pieza dada de funcionalidad tomará días en lugar de meses o años.

Mejor ROI Usando correctamente esta estrategia, el retorno sobre la inversión puede ser más favorable que desarrollando los componentes uno mismo.

Funcionalidad mejorada Para usar un componente que contenga una pieza de funcionalidad, solo se necesita entender su naturaleza, más no sus detalles internos.

2.4. Diseño en el Proceso Unificado de Desarrollo

2.4.1. Objetivo

- Aquirir una comprensión en profundidad de los aspectos relacionados con los requisitos no funcionales y restricciones relacionadas con los lenguajes de programación, componentes reutilizables, sistemas operativos, tecnologías, etc.
- Crear una entrada apropiada y un punto de partida para actividades de implementación subsiguientes capturando los requisitos o subsistemas individuales, interfaces y clases.
- Ser capaces de descomponer los trabajos de implementación en partes más manejables que puedan ser llevadas a cabo por diferentes equipos de desarrollo teniendo en cuenta la posible concurrencia.

2.4.2. El papel del diseño en el ciclo de vida del software

El diseño es el centro de atención al final de la fase de elaboración y el comienzo de las iteraciones de construcción. El diseño está muy cercando al de implementación, lo que es natural para guardar y mantener el modelo de diseño a través del ciclo de vida completo del software.

2.4.3. Artefactos

Modelo de diseño Es un modelo de objetos que describe la realización física de los casos de uso centrándose en cómo los requisitos funcionales y no funcionales, junto con otras restricciones relacionadas con el entorno de implementación, tienen impacto en el sistema a considerar.

Clase de diseño Es una abstracción sin costuras de una clase o construcción similar en la implementación del sistema.

Realización de caso de uso de diseño Es una colaboración en el modelo de diseño que describe cómo se realiza un caso de uso específico y como se ejecuta en términos de clase de diseño y sus objetivos. Proporciona una traza directa a una realización de caso de uso de análisis en el modelo de análisis.

Diagramas de clases Es una clase de diseño y sus objetivos.

Diagramas de interacción La secuencia de acciones en un caso de uso comienza cuando un actor invoca el caso de uso mediante el envío de algún tipo de mensaje al sistema.

Flujo de sucesos de diseño Es una descripción textual que explica y complementa a los diagramas y a sus etiquetas.

Requisitos de la implementación Son una descripción textual que recoge requisitos tales como los requisitos no funcionales sobre una realización de caso de uso.

Subsistema de diseño Son una forma de organizar los artefactos del modelo de diseño en piezas más manejables.

Subsistemas de servicio Se utilizan en un nivel inferior de la jerarquía de subsistemas de diseño para aislar los cambios en los correspondientes subsistemas de servicio.

Interfaz Las interfaces se utilizan para especificar las operaciones que proporcionan las clases y los subsistemas del disñeo.

Descripción de la arquitectura (Vista del modelo de diseño) Muestra los artefactos del modelo de diseño relevantes para la arquitectura.

- La descomposición del modelo de diseño en subsistemas, sus interfaces y las dependencias entre ellos.
- Clases del diseño fundamentales.
- Realizaciones de caso de uso de diseño que describen alguna funcionalidad importante y crítica que debe desarrollarse pronto dentro del ciclo de vida del software.

Modelo de despliegue Es un modelo de objetos que describe la distribución física del sistema en términos de cómo se distribuye la funcionalidad entre los nodos del cómputo.

Descripción de la arquitectura (Vista del modelo de despliegue Contiene una vista de la arquitectura del modelo de despliegue que muestra sus artefactos relevantes para la arquitectura.

2.4.4. Trabajadores

Ingeniero de casos de uso Es responsable de la integridad de una o más realizaciones de casos de uso de diseño y debe garantizar que cumplen los requisitos que se esperan de ellos.

Ingeniero de componentes Define y mantiene las operaciones, métodos, atributos, relaciones y requisitos de implementación de una o más clases del diseño garantizando que cada clase del diseño cumple los requisitos que se esperan de ella según las realizaciones de caso de uso en las que participa.

2.4.5. Actividades

Diseño de la arquitectura Su objetivo es esbozar los modelos de diseño y despliegue y su arquitectura mediante la identificación de los siguientes elementos:

- Nodos y sus configuraciones.
- Subsistemas y sus interfaces.
- Clases del diseño significativas para la arquitectura.
- Mecanismos de diseño genéricos que tratan requisitos comunes.

Diseño de un caso de uso Los objetivos de esta actividad son:

- Identificar las clases del diseño y/o los subsistemas cuyas instancias son necesarias para llevar a cabo el flujo de sucesos del caso de uso.
- Distribuir el comportamiento del caso de uso entre los objetos del diseño que interactúan y/o entre los subsistemas participantes.
- Definir los requisitos sobre las operaciones de las clases del diseño y/o sobre los subsistemas y sus interfaces.
- Capturar los requisitos de implementación del caso de uso.

Diseño de una clase El objetivo es crear una clase del diseño que cumpla su papel en las realizaciones de los casos de uso y los requisitos no funcionales que se aplican a estos.

Diseño de un subsistema Los objetivos de esta actividad son:

- Garantizar que el subsistema es tan independiente como sea posible de otros subsistemas y/o de sus intereses.
- Garantizar que el subsistema proporciona las interfaces correctas.
- Garantizar que el subsistema cumple su propósito de ofrecer una realización correcta de las operaciones tal y como se definen en las interfaces que proporciona.

2.5. Diseño del Software Orientado a Objetos

2.5.1. Diseño del Comportamiento del Software

Se refieren a algoritmos y asignación de responsabilidades a objetos.

2.5.2. Diseño de la Estructura del Software

Se refiere a cómo se combinan las clases y los objetos para formar estructuras más grandes.

2.5.3. Patrones de diseño

Introducción Los patrones de diseño ayudan a diseñar software orientado a objetos reutilizable. Estos patrones resuelven problemas concretos de diseño y hacen que los diseños orientados a objetos sean más flexibles, elegantes y reutilizables.
Cada patrón nomina, explica y evalúa un diseño importante y recurrente en los sistemas orientados a objetos. Pueden incluso mejorar la documentación y el mantenimiento de los sistemas existentes al proporcionar una especificación explícita de las interacciones entre clases y objetos.

¿Qué son los Patrones de Diseño? Son descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto.

Tipos de Patrones de Diseño

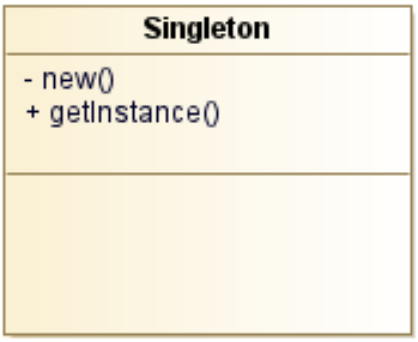
Patrones de Creación Los patrones de creación tienen que ver con el proceso de creación de objetos. Los patrones de creación de clases delegan alguna parte del proceso de creación de objetos en las subclases, mientras que los patrones de creación de objetos lo hacen en otro objeto.

Patrones Estructurales Los patrones estructurales tratan con la composición de clases u objetos. Los patrones estructurales de clases usan la herencia para componer clases, mientras que los de objetos describen formas de ensamblar objetos.

Patrones de Comportamiento Caracterizan el modo en que las clases y objetos interactúan y se reparten la responsabilidad. Los patrones de comportamiento de clases usan la herencia para describir algoritmos y flujos de control, mientras que los de objetos describen cómo cooperan un grupo de objetos para realizar una tarea que ningún objeto puede llevar a cabo por sí solo.

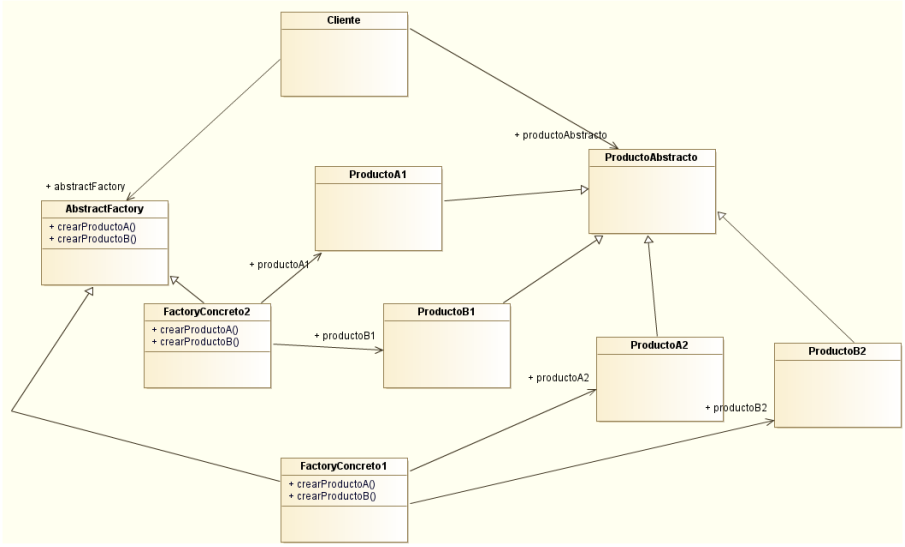
Patrones de Diseño de Creación

Singleton El patrón de diseño singleton (instancia única) está diseñado para restringir la creación de objetos pertenecientes a una clase o el valor de un tipo a un único objeto.
Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.
El patrón singleton se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor (con atributos como protegido o privado).

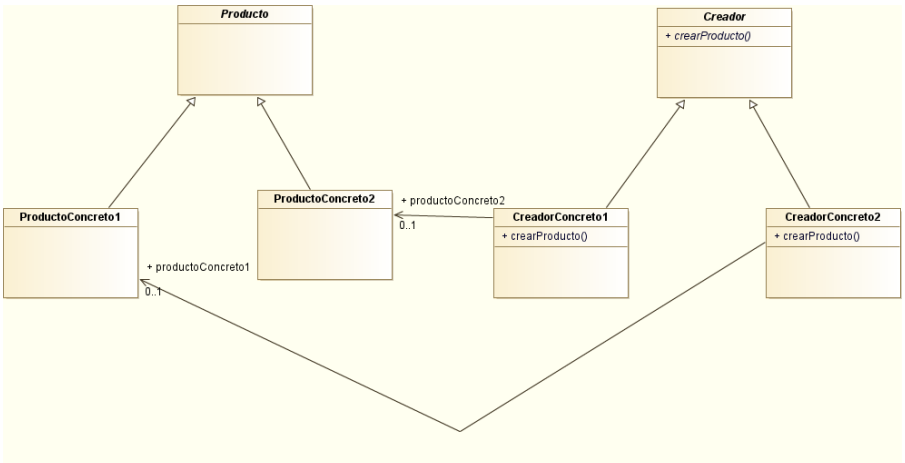


Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase controla el acceso a un recurso físico único (como puede ser el ratón o un archivo abierto en modo exclusivo) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

Abstract Factory Contexto: Debemos crear diferentes objetos, todos pertenecientes a la misma familia. Por ejemplo: las bibliotecas para crear interfaces gráficas suelen utilizar este patrón y cada familia sería un sistema operativo distinto. Así pues, el usuario declara un Botón, pero de forma más interna lo que está creando es un BotónWindows o un BotónLinux, por ejemplo.
El problema que intenta solucionar este patrón es el de crear diferentes familias de objetos.
El patrón Abstract Factory está aconsejado cuando se prevé la inclusión de nuevas familias de productos, pero puede resultar contraproducente cuando se añaden nuevos productos o cambian los existentes, puesto que afectaría a todas las familias creadas

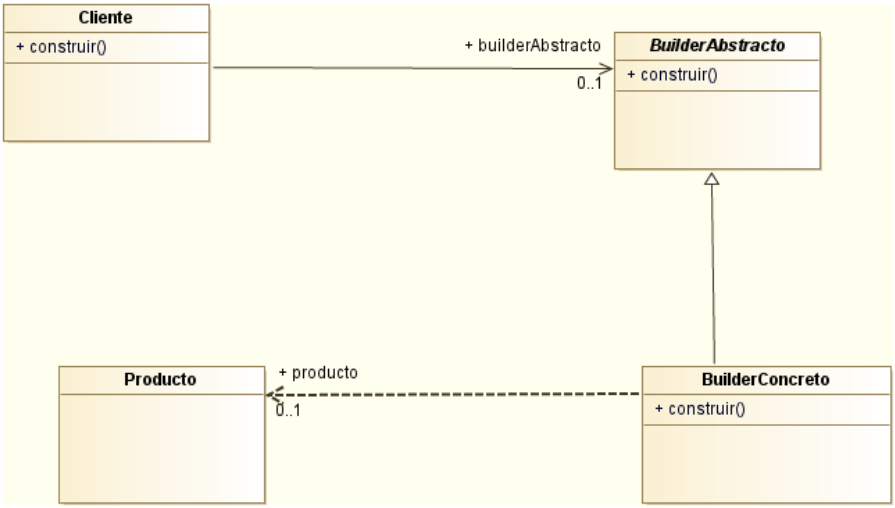


Factory Method El patrón de diseño Factory Method consiste en utilizar una clase constructora (al estilo del Abstract Factory) abstracta con unos cuantos métodos definidos y otro(s) abstracto(s): el dedicado a la construcción de objetos de un subtipo de un tipo determinado. Es una simplificación del Abstract Factory, en la que la clase abstracta tiene métodos concretos que usan algunos de los abstractos; según usemos una u otra hija de esta clase abstracta, tendremos uno u otro comportamiento.

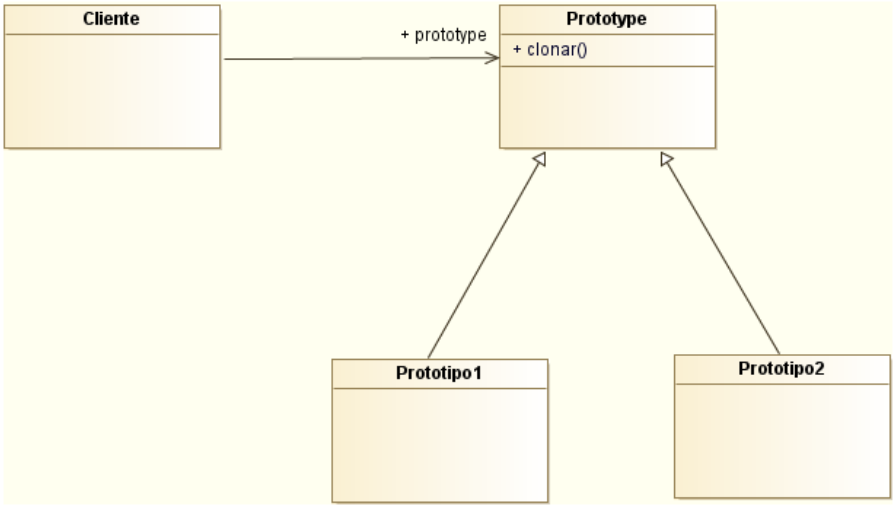


Builder El patrón builder (Constructor) es usado para permitir la creación de una variedad de objetos complejos desde un objeto fuente (Producto), el objeto fuente se compone de una variedad de partes que contribuyen individualmente a la creación de cada objeto complejo a través de un conjunto de llamadas a interfaces comunes de la clase Abstract Builder.
Intención: Abstrae el proceso de creación de un objeto complejo, centralizando

dicho proceso en un único punto, de tal forma que el mismo proceso de construcción pueda crear representaciones diferentes.

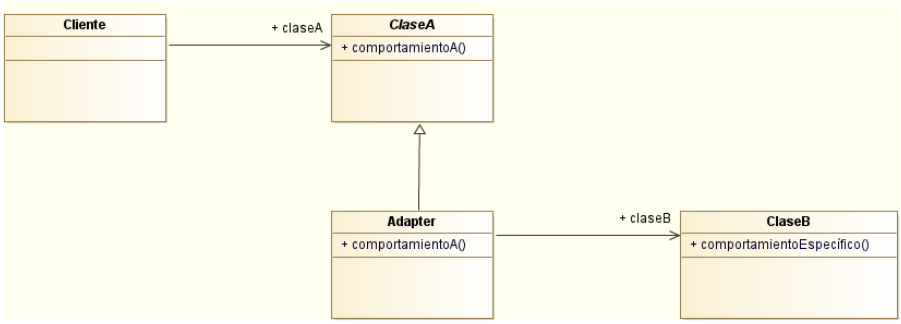


Prototype El patrón de diseño Prototype (Prototipo), tiene como finalidad crear nuevos objetos duplicándolos, clonando una instancia creada previamente.
Este patrón especifica la clase de objetos a crear mediante la clonación de un prototipo que es una instancia ya creada. La clase de los objetos que servirán de prototipo deberá incluir en su interfaz la manera de solicitar una copia, que será desarrollada luego por las clases concretas de prototipos.

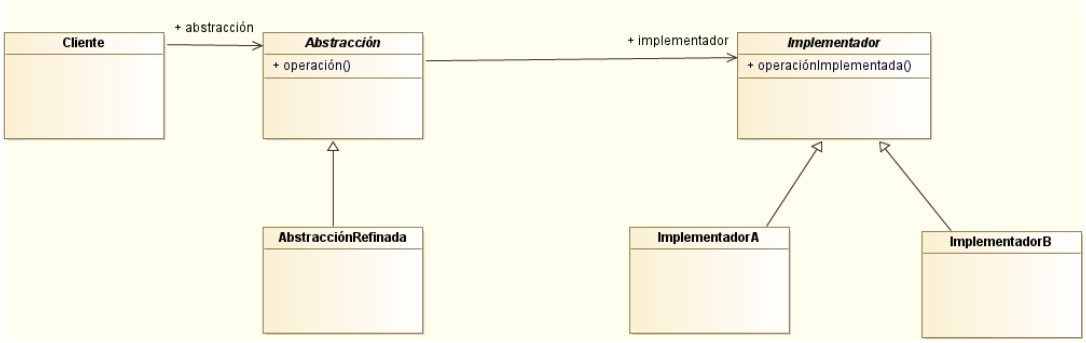


Patrones de Estructura

Adapter El patrón Adapter (Adaptador) se utiliza para transformar una interfaz en otra, de tal modo que una clase que no pudiera utilizar la primera, haga uso de ella a través de la segunda.



Bridge l patrón Bridge, también conocido como Handle/Body, es una técnica usada en programación para desacoplar una abstracción de su implementación, de manera que ambas puedan ser modificadas independientemente sin necesidad de alterar por ello la otra.
Esto es, se desacopla una abstracción de su implementación para que puedan variar independientemente.



Composite El patrón Composite sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol.
Esto simplifica el tratamiento de los objetos creados, ya que al poseer todos ellos una interfaz común, se tratan todos de la misma manera.