Año 2013

Resumen de DSS

Diseño de Sistemas Software

Modelos de proceso

Definición

Representación simplificada de un proceso software, representada desde una perspectiva específica.

Genéricos

Los modelos genéricos son abstracciones útiles que pueden ser utilizadas para explicar diferentes enfoques del desarrollo de software, no son descripciones definitivas del mismo.

Principales modelos (según Sommerville)

No modelar

Codificar y corregir (años 60): Codificar y corregir problemas sobre el código.

Problemas:

- Mala estructura después de un par de correcciones. Lo que significa: arreglos costosos.
- Aun siendo el software bien diseñado, no se ajusta a las necesidades del usuario.
    - Rechazado
    - Cara reconstrucción
- Código difícil de reparar por su pobre preparación para probar y modificar.

Desarrollo secuencial

Modelo en cascada: Conjunto de etapas que se realizan secuencialmente. Creado por Winston Royce, quien no lo recomienda para grandes proyectos.

Secuencia de etapas:

- Definición de requisitos
- Diseño de sistema y software
- Implementación y pruebas de unidad
- Integración y pruebas de sistema
- Operación y mantenimiento

Ventajas: Si se conocen todos los requisitos permite una planificación muy precisa del desarrollo.

Problemas:

- Iteraciones costosas por la producción y aprobación de **documentos**.
- Los problemas se resuelven al final:
    - Pueden ser ignorados
    - Pueden ser corregidos de mala forma
- Largo tiempo de entrega, implica alta probabilidad de que no cumpla los requisitos del usuario.
- Inflexible a la hora de incorporar cambios.

Desarrollo formal de sistemas: Refleja el paradigma de programación automática y se caracteriza por un conjunto de transformaciones formales de los requisitos hasta llegar a un programa ejecutable. Algunos autores (Sommerville) lo ven como una variación del modelo cascada.

Ventajas:

- Permite la corrección del sistema durante el proceso de transformación.
- Es atractivo para sistemas donde hay requisitos de seguridad y confiabilidad importantes.

Problemas:

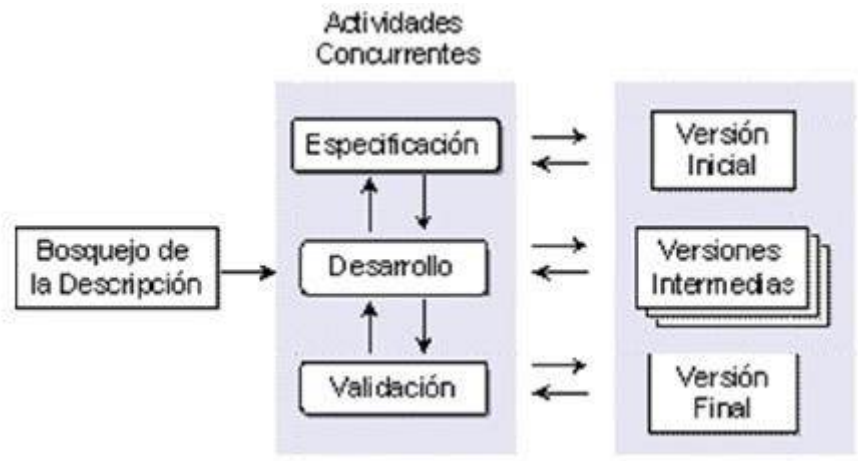
- Requiere desarrolladores especializados y experimentados en este proceso.
- Requiere herramientas específicas.

Desarrollo evolutivo:

Dos tipos:

- Desarrollo explorativo: comienza por lo que está más claro.
- Enfoque basado en el prototipado: comienza por lo más confuso creando prototipos que ayudan a aclarar requisitos.

Secuencia de etapas:



Ventajas:

- La especificación puede desarrollarse de forma creciente.
- Los usuarios y desarrolladores logran un mejor entendimiento del sistema. Mejora de la calidad del software.
- Es más efectivo que el modelo de cascada, ya que cumple con las necesidades inmediatas del cliente.

Problemas:

- Proceso no visible: las actividades de especificación, desarrollo y validación están mezcladas por lo que no hay una idea clara del punto del desarrollo en el que se está.
- Sistemas pobremente estructurados: los cambios continuos no son buenos para la estructura del software. Implica costoso mantenimiento.
- Se requieren técnicas y herramientas específicas que pueden ser incompatibles con el estilo y/o tipo de formación de la organización.

Desarrollo basado en reutilización:

Secuencia de etapas:

- Definición de requisitos
- Análisis de componentes
- Modificación de requisitos
- Diseño del sistema con reutilización
- Desarrollo e integración
- Validación del sistema

Ventajas:

- Disminuye el costo y esfuerzo de desarrollo
- Reduce el tiempo de entrega
- Disminuye los riesgos durante el desarrollo

Problemas:

- Compromisos en los requisitos son inevitables, puede no cumplir con las expectativas del cliente.
- Las actualizaciones de los componentes adquiridos no están en manos de los desarrolladores del sistema.

Desarrollo iterativo

Desarrollo incremental: Se trata de un híbrido entre cascada y evolutivo en el cual se congelan los requisitos en cada intervalo.

Secuencia de etapas:

- Definición de requisitos
- Asignar requisitos a los incrementos
- Diseñar la arquitectura del sistema
    - Desarrollar incrementos del sistema
    - Validar incrementos
    - Integrar incrementos
    - Validar sistema

Ventajas:

- Los clientes no esperan hasta el final del proyecto para utilizar el sistema.
- Se adapta fácilmente a los cambios.
- Los clientes pueden aclarar los requisitos que no tengan claros a lo largo del proyecto.
- Se disminuye el riesgo de fracaso de todo el proyecto.
- Las partes importantes son entregadas al principio, por lo que sobre ellas se realizan más pruebas que disminuyen el riesgo de fallos.

Problemas:

- Cada incremento ha de ser pequeño para limitar el riesgo.
- Cada incremento debe aumentar la funcionalidad.
- Es difícil establecer las correspondencias de los requisitos contra los incrementos.
- Es difícil detectar las unidades o servicios genéricos para todo el sistema.

Desarrollo en espiral: Es completamente híbrido, en cada iteración se planifica cómo va a ser la siguiente fase, las actividades entre fases pueden ser distintas entre sí. Tiene en cuenta explícitamente el riesgo desde el principio (cuando es mayor) hasta el final (que disminuye).

Ventajas:

- Gran énfasis en la gestión de riesgos.
- Se adapta fácilmente a los cambios (planificando cada iteración).
- Buena estimación de costes en cada iteración.

Problemas:

- La evaluación de riesgos puede ser muy costosa y necesita de expertos.
- El éxito del proyecto depende en gran medida del análisis de riesgos.
- Sólo es apto para grandes proyectos.

Procesos ágiles

Modelo Scrum: Se ve más adelante.

Programación extrema: o extrem programming, se ve más adelante.

Proceso unificado: Se ve más adelante.

Comparativa

Modelo de proceso	Funciones con requisitos y arquitecturas no predefinidas	Produce software altamente fiable	Gestión de riesgos	Permite correcciones sobre la marcha	Visión del progreso por el Cliente y el Jefe de proyecto
Codificar y corregir	Bajo	Bajo	Bajo	Alto	Medio
Cascada	Bajo	Alto	Bajo	Bajo	Bajo
Evolutivo exploratorio	Medio o Alto	Medio o Alto	Medio	Medio o Alto	Medio o Alto
Evolutivo prototipado	Alto	Medio	Medio	Alto	Alto
Desarrollo formal de sistemas	Bajo	Alto	Bajo o Medio	Bajo	Bajo
Desarrollo orientado a reutilización	Medio	Bajo a Alto	Bajo a Medio	Alto	Alto
Incremental	Bajo	Alto	Medio	Bajo	Bajo
Espiral	Alto	Alto	Alto	Medio	Medio

Conclusiones

- Proceso en cascada: sólo si se conocen completamente y bien los requisitos de antemano.
- Desarrollo evolutivo: efectivo para pequeños o medianos proyectos (<100.000 líneas de código o < 500.000), con poco tiempo para su desarrollo y sin documentación en cada versión.
- Combinación del modelo en cascada y evolutivo para proyectos largos. Prototipo global del sistema y posteriormente reimplementarlo con un acercamiento más estructurado.
    - Los subsistemas con requisitos bien definidos y estables se pueden realizar en cascada.
    - La interfaz de usuario utilizando un enfoque explorativo.

Scrum

Definición

- Proceso ágil que nos permite centrarnos en ofrecer el más alto valor de negocio en el menor tiempo posible.
- Nos permite rápidamente y en repetidas ocasiones inspeccionar software real de trabajo (cada dos semanas o un mes).
- El negocio fija las prioridades. Los equipos se auto-organizan a fin de determinar la mejor manera de entregar las funcionalidades de más alta prioridad.
- Cada dos semanas o un mes, cualquiera puede ver el software real funcionando y decidir si liberarlo o seguir mejorándolo en otro sprint.

"Agile Software Development with SCRUM" es un libro de referencia sobre la metodología.

Ha sido utilizado por muchas grandes compañías para todo tipo de proyectos software con éxito.

Características

- Equipos auto-organizados.
- El producto avanza en sprints de dos semanas a un mes de duración.
- Los requisitos son capturados como elementos de una lista (Product Backlog).
- No hay prácticas de ingeniería prescritas.
- Utiliza normas generativas para crear un entorno ágil para la entrega de proyectos.

Manifiesto ágil

- **Individuos e interacciones** sobre procesos y herramientas: para que no se quede estancado el proyecto, los individuos han de estar involucrados de forma que no se vean forzados en in proceso rígido.
- **Software que funciona** sobre documentación exhaustiva: Se prioriza la creación de software a crear mucha documentación.
- **Colaboración con el cliente** sobre negociación de contratos: La negociación no se puede evitar, pero se han de dejar abiertos para poder tratar con modificaciones a lo largo del proyecto.
- **Responder ante el cambio** sobre el seguimiento de un plan.

Secuencia de etapas

- Product backlog
- Sprint
    - Sprint backlog
    - Daily scrum meeting
    - Potentially shippable product increment

Sprint

- Análogo a las iteraciones en XP.
- Duración 2-4 semanas.
- Duración constante, no se puede cambiar la duración de los sprints dentro del mismo proyecto.
- El producto es diseñado, codificado y testeado durante el sprint. Se hace de todo un poco a lo largo de todo el sprint.
- No se pueden producir cambios en los requisitos durante el sprint.

Roles

Product owner

- Define las funcionalidades, si al equipo se le ocurren cosas nuevas se lo han de comunicar a él.
- Decide sobre las fechas y contenidos de los realeases.
- Es responsable por la rentabilidad del producto.
- Prioriza funcionalidades de acuerdo al valor del mercado /negocio (lo que en menos tiempo da más dinero) porque es quien sabe qué necesita (los stakeholders tienen intereses sobre el proyecto pero no participan en las prioridades).
- Ajusta funcionalidades y prioridades en cada iteración si es necesario.

Scrum master

- Representa a la gestión del proyecto.
- Responsable de promover los valores y prácticas scrum.
- Elimina impedimentos e interferencias externas.
- Se asegura de que el equipo es completamente funcional y productivo.
- Permite la cooperación entre los roles y funciones.

Team

- Tamaño: 5 a 9 personas
    - Menos de 5 es complicado conseguir funcionalidad en tan poco tiempo.
    - Más de 9 es complicado la auto-organización.
- Multifuncional: programadores, testers, analistas, diseñadores…
- A tiempo completo (puede haber excepciones como infraestructura, SCM…).
- Equipos auto-organizativos: no existen títulos pero a veces se utilizan de acuerdo a la organización.
- Sólo puede haber cambio de personal entre sprints.

Reuniones

Sprint planning

- Se ha de conocer:
    - La capacidad del equipo (velocidad de trabajo).
    - El product backlog.
    - Condiciones del negocio.
    - Producto actual (si existe).
    - Tecnología.
- Priorización:
    - Analizar y evaluar el product backlog.
    - Seleccionar el **objetivo del sprint**.
- Planificación
    - Decidir cómo alcanzar el objetivo del sprint (diseño).
    - Crear el **Sprint backlog** (tareas) en base a los temas del product backlog (user stories/ features).
    - Estimar Sprint backlog en horas.
    - Tareas:
        - Entre 1 y 16 horas de trabajo para evitar errores y dinamizar el trabajo.
        - Se realizan en equipo, no sólo el scrum master.
        - El product owner ha de estar presente para solucionar posibles dudas sobre las historias.

Sprint review

- El equipo presenta lo realizado durante el sprint.
    - Si algo no funciona o se ha quedado a medias, queda como prioritario para el siguiente sprint.
- Demo con las nuevas características o la arquitectura subyacente.
- Informal.
    - 2 horas de preparación.
    - Sin diapositivas.
- Todo el equipo participa.

Sprint retrospective

- Se mira lo que ha funcionado o no durante el sprint.
    - Qué se puede empezar a hacer.
    - Qué se ha de dejar de hacer.
    - Qué se debe seguir haciendo.
- No se habla del producto, sino de la forma de trabajo.
- Entre 15 y 30 minutos.
- Participa todo el equipo.

Daily scrum meeting

- Diaria y de unos 15 minutos de duración (de pie para que no se alargue).

- Todo el mundo puede escuchar pero sólo el equipo, el scrum master y el product owner pueden hablar.
- No se resuelven problemas, sólo se informa.
- Contenido:
    - ¿Qué hiciste ayer?
    - ¿Qué vas a hacer hoy?
    - ¿Hay obstáculos en tu camino? (El scrum master se encargará de solucionar los problemas).

Herramientas/Artefactos

Product backlog

- Se trata de una lista de requisitos o elementos deseados en el proyecto.
- Cada tena tiene valor para el usuario o cliente.
- Se reprioriza al comienzo de cada sprint, por parte del product owner.

Sprint backlog

- Los individuos eligen tareas, no se asigna trabajo. Cualquier miembro del equipo puede añadir, borrar o cambiar el sprint backlog.
- La estimación del trabajo restante es actualizada diariamente.
- Trabajo no claro: definir un tema del sprint backlog con mayor cantidad de tiempo y subdividirla más tarde.
- Tema del sprint: declaración de cuál será el foco del trabajo durante el sprint.

Burndown charts

- Indica la cantidad de trabajo que se va avanzando.
- Lo ideal es que la gráfica salga lineal.

Extreme Programming

Definición

Aproximación extrema al desarrollo iterativo

- Se compilan nuevas versiones del software diariamente, varias veces al día.
- Los incrementos se entregan al cliente en periodos de dos semanas.
- Las pruebas se ejecutan en cada compilación, y la nueva versión sólo se acepta si supera todas las pruebas correctamente.

5 valores fundamentales

- **Simplicidad**: no hacer más de lo estrictamente necesario.
- **Comunicación**: interacción diaria cara a cara.
- **Realimentación**: demostraciones de software funcional en cada iteración para detectar los cambios necesarios.
- **Respeto**:
    - Los desarrolladores respetan la experiencia de los clientes y viceversa.
    - Los desarrolladores tienen derecho a estableces sus propias responsabilidades.
- **Coraje**: la verdad por delante, las estimaciones y los informes de progreso del proyecto se debe hacer de forma veraz.

Características

Desarrollo incremental

- Favorece el **desarrollo incremental** mediante releases del sistema frecuentes.
    - SECUENCIAS E ITERACIONES

Cliente

- El **cliente debe implicarse** a tiempo completo con el equipo de trabajo.
- Cliente = Pieza clave y parte del equipo.
- Roles:
    - Ayudar a desarrollar historias de usuario que definan los requisitos.
    - Ayudar a priorizar las características que se implementarán en cada reléase.
    - Ayudar a desarrollar pruebas de aceptación que determinarán si el sistema cumple con los requisitos.
- Los requisitos
    - Se expresan en forma de historias de usuario.
    - Creación:
        - Se escriben en tarjetas
            - En lenguaje informal explicando la funcionalidad.
        - El equipo las divide en tareas de implementación
            - En lenguaje informal también pero explicando la parte técnica para conseguir la historia.
            - Base de la planificación y la estimación de costes.
    - El cliente selecciona las historias de la siguiente release de acuerdo a sus prioridades y las estimaciones de tiempo.

Gente sobre proceso

- Se da **prioridad a la gente** frente al proceso
- Programación en parejas
    - Mientras uno escribe el otro detrás comprueba el código, indica la forma de hacerlo…
    - Ayuda a mantener la propiedad colectiva del código y fomenta la compartición de conocimientos.
    - Indicadores: la productividad con esta técnica = dos personas trabajando por separado.
- Propiedad colectiva del código
    - Todo el mundo toca todo el código y van rotando para que todos sepan todo de todo el código.
- Class-Responsibility-Collaboration Cards
    - El diseño en XP se limita a modelar las clases mediante las tarjetas CRC
    - Todos participan en la elaboración de las mismas con el juego de roles.
- Pruebas de código
    - Se diseñan antes de escribir el código.
    - Se realizan de forma incremental conforme se añaden nuevas funcionalidades.
    - El cliente se involucra activamente en el desarrollo y validación de las mismas.
    - Se usan sistemas de automatización para ejecutarlas cada vez que se compila una nueva release.
- Evitar trabajar muchas horas seguidas.

Adaptación

- **Fácil adaptación** a los cambios gracias a las frecuentes releases del sistema.
- El tiempo y esfuerzo dedicado a anticipar cambios reduce costes más adelante.
- Pero como no se pueden anticipar la totalidad de los cambios, en XP se promueve la constante mejora del código para que los posibles cambios sean más fáciles de realizar: refractorización.

Refractorización

- Se mantiene la **simplicidad** del sistema mediante la constante refractorización del código.
- Proceso de reescribir y reorganizar el código para mejorarlo y hacerlo más eficiente.
- El elevado número de releases y el desarrollo incremental implican un código complicado. Refractorizar es necesario.
- No debe cambiar la funcionalidad del sistema.
- La automatización de las pruebas ayuda a comprobar si el código cambiado mantiene la funcionalidad.

Problemas

Implicación de los clientes

Complicado encontrar a un cliente representante de los stakeholders que pueda dejar su trabajo para implicarse en el equipo de desarrollo.

No existe un cliente para productos genéricos.

Diseño de la arquitectura

Las decisiones sobre la arquitectura pueden ser malas pero no se descubre hasta que sea demasiado tarde.

Pérdida del objetivo en las pruebas

Tendencia a desarrollar pruebas fáciles de automatizar.

Se han de realizar baterías de pruebas completas y correctas.

Proceso Unificado

Definición

- Marco de proceso, que puede instanciarse a otros marcos (RUP) o procesos concretos para proyectos concretos.
- Soporte opcional para grados mayores de formalidad y documentación
  - Ofrece alrededor de 50 productos de trabajo opcionales no software.
  - Se recomienda el uso de la menor cantidad posible.
    - Al menos Visión y lista de riesgos.
  - El orden de las tareas es opcional.
  - Development case:
    - Configuración personalizada
    - Se han de especificar los artefactos a usar dependiendo de la disciplina, práctica y la iteración.
- Proporciona un vocabulario común para grandes proyectos.

Ciclo de desarrollo

- Contiene cuatro fases con distintos objetivos y sus propias iteraciones.
  - Milestone: punto final en una iteración, se Han de tomar decisiones significativas o la evaluación del progreso.
  - Release: ejecutable estable parte del proyecto final.
    - Al final de cada iteración ha de haber un pequeño ejecutable.
  - Increment: diferencia entre releases consecutivas.
- Distribución de actividades
  - En cada iteración se realiza de todo, ya que ha de haber un software funcional al final de cada iteración.
  - Dependiendo de la fase de proyecto, el esfuerzo dedicado a cada actividad (análisis, diseño, implementación, pruebas…) varía.

Fases

Comienzo (inception)

- Propósito
  - Visión general, descripción de los objetivos por encima…
  - Definición del 10% de los requerimientos más importantes en detalle.
  - Lista de riesgos.
  - Estimación de costes.
- Actividades posibles
  - Requerimientos.
  - Visión y lista de riesgos.
  - Prototipado.
- No es la fase de requerimientos, pero sí tienen gran importancia en esta fase.
- Consta de una sola iteración.

Elaboración

- Propósito
  - Las partes significantes de la arquitectura codificadas y testadas.
  - Los riesgos más importantes identificados y mitigados.
  - El 80% de los requerimientos definidos y detallados.
  - Suficiente información y estabilidad para una correcta estimación de costes y duración.
- Actividades posibles
  - Testeado, programación y diseño en cortas iteraciones.
  - Refinamiento de la visión y la creación requerimientos.
  - Refinamiento del entorno de trabajo y del proyecto.
- No es la fase de diseño, aunque las actividades relacionadas con él tienen mucha importancia.

Construcción

Transición (transition)

Introducción a los patrones

Definición

Mecanismo para capturar conocimiento sobre problemas y soluciones exitosas en el desarrollo software.

Permiten la reutilización de la experiencia de otros diseñadores, reducir el esfuerzo asociado a la producción de sistemas más efectivos y flexibles.

Tipos de patrones

- Patrones de análisis (Fowler)
- Patrones de diseño
    - Patrones GRASP (Larman)
    - Microarquitectura (diseño detallado)
    - Arquitecturales y microarquitecturales
- Patrones de implementación
    - Idioms (específicos de paradigma y lenguaje programación)
- Patrones de integración de aplicaciones
- Patrones de proceso de desarrollo software (diseño de procesos)
- Patrones de organización (estructura de organizaciones/proyectos)
- Patrones de UI: Interfaces de usuario
    - Existen distintas categorías bien diferenciadas: algunas se encargan de detalles relacionados con la cognición, memoria a corto plazo y mejoras en la experiencia del usuario, mientras que otros describen técnicas de ingeniería para crear interfaces de usuario.
- Patrones de prueba
- Antipatrones (de cualquier categoría)

Principios subyacentes

- Según Larman:
    - Prever la variación y evolución del software.
    - Favorecer el bajo acoplamiento, la alta cohesión y el reuso del software.
- Según GOF:
    - Programar para interfaces y no para una implementación
    - Favorecer la composición de objetos frente a la herencia.
- Según Buchmann
    - Funcionales:
        - Abstracción
        - Encapsulación
        - Ocultación de información
        - Modularización
        - Separación de preocupaciones
        - Acoplamiento y cohesión
        - Suficiencia, completitud y primitividad
        - Separación entre política e implementación

        - Separación entre interfaz e implementación
        - Único punto de referencia
        - Divide y vencerás
    - No funcionales:
        - Facilidad para realizar cambios (mantenibilidad, extensibilidad, restructuración y portabilidad)
        - Interoperabilidad
        - Eficiencia
        - Confiabilidad
        - Testabilidad
        - Reusabilidad
- Principio común: **Encapsulación** de cualquier variación: datos, comportamiento, responsabilidades…

Análisis de variabilidad

Coplien: "Para diseñar OO hay que detectar dónde varían las cosas y dónde no (análisis de lo común) y **cómo** varían (análisis de variabilidad).

- Conceptos comunes: clases abstractas, perspectiva de análisis y diseño.
- Variaciones: clases concretas, perspectiva de implementación.

Proceso

- Identifica los conceptos de tu sistema y organízalos de la forma más cohesiva que sea posible.
- Crea abstracciones para esos elementos comunes.
- Identifica variaciones de esos conceptos comunes.
- Identifica cómo los elementos comunes se relacionan.

Documentación

- La documentación de los patrones se realiza mediante el uso de plantillas.
- Forma canónica de las plantillas:
    - Nombre: significativo y corto.
    - Problema: intención del patrón.
    - Contexto: precondiciones de aplicabilidad.
    - Fuerzas: aspectos que deben ser considerados en la solución.
    - Solución: descripción de relaciones estáticas y dinámicas entro los componentes del patrón.
- Información optativa:
    - Ejemplos de aplicaciones.
    - Contexto resultante: estado del sistema después de aplicar el patrón.
    - Fundamentos: explicación de los pasos o reglas en el patrón.
    - Patrones relacionados: relaciones estáticas y dinámicas.
    - Usos reales del patrón y su aplicación en sistemas existentes.

Lenguaje de patrones

- Definiciones de Coplien

    - “Colección estructurada de patrones que se aplican unos sobre otros para transformar necesidades y restricciones en una arquitectura.”
    - “Define una colección de patrones y reglas para combinarlos en un estilo arquitectural” “Describe frameworks de software o familias de sistemas relacionados.”
    - Las reglas y guías sugieren el orden y la granularidad con la que aplicar cada patrón del lenguaje.”
- Lenguaje de patrones frente a sistema de patrones
    - El lenguaje agrupa patrones más estrechamente relacionados que trabajan juntos para solucionar problemas en un dominio determinado.
    - El lenguaje es computacionalmente completo.
    - En la práctica la diferencia es muy difícil de dilucidar.

Catálogo

Grupo de patrones relacionados, normalmente divididos en subcategorías, que pueden ser utilizados en conjunto o por separado. [Buchmann]

Patrones vs Frameworks

- Los patrones son más abstractos y generales.
- El patrón es una descripción de cómo resolver un problema pero no la solución en sí.
- Un patrón no puede ser directamente implementado en un entorno software particular. Un implementación exitosa = ejemplo de patrón de diseño.
- Los patrones son más sencillos que los frameworks.
    - Un patrón no puede incorporar un framework.
    - Un framework puede hacer uso de varios patrones.

Beneficios

- Reuso de soluciones genéricas evita errores a los diseñadores noveles.
- Provee un vocabulario para discutir el dominio del problema a un nivel de abstracción más elevado.

Peligros

- Puede limitar la creatividad.
- Puede llevar al sobre-diseño.
    - Un patrón ha de ser utilizado siempre en el contexto adecuado y tras evaluarse sus ventajas e inconvenientes.
- Son aplicables sólo dentro de una cultura de reuso.
- Afrontan únicamente algunos de los problemas que aparecen durante el desarrollo de sistemas.

Guías

- ¿Hay algún patrón que resuelva un problema similar?
- ¿Ese patrón proporciona una solución alternativa que puede ser más aceptable?
- ¿Hay una solución más sencilla? Si la hay, olvidarse del patrón.
- ¿Es el contexto del patrón consistente con respecto al del problema?

- ¿Son las consecuencias de utilizar el patrón aceptables?
- ¿Existen restricciones impuestas por el entorno de software que entrarían en conflicto con el uso del patrón?

Patrones GRASP

Introducción

- Diagrama de clases
    - Ilustran clases, interfaces y sus asociaciones.
    - **Diagrama de clases de análisis**: representa el modelo de dominio.
    - **Diagrama de clases de diseño**: representa a las clases software, las que finalmente se implementarán. (Los diagramas que usaremos en clase)

Diseño de objetos

- Definición: tras haber identificado tus requisitos, y haber creado un modelo de dominio, añade las operaciones a las clases y define las secuencias de mensajes entre los objetos para cubrir los requisitos.
- Principios fundamentales de diseño:
    - Decidir qué operaciones hay que asignar a qué clases
    - Cómo los objetos deberían interactuar para dar respuesta a los casos de uso.
- Artefacto más importante del flujo de trabajo de diseño: **Modelo de diseño** (**diagrama de clases software** y **diagramas de interacción**)
- Del DC de análisis al DC de diseño
    - Generación vs Herencia
        - El modelo de dominio implica que la superclase es un superconjunto y la subclase un subconjunto.
        - En el DCD implica la relación de herencia de los lenguajes de programación OO de una superclase a una subclase.
        - Las relaciones de generalización no tienen por qué traducirse en herencia en el DCD.
    - Aparición de nuevas clases:
        - **Clases de utilidad**: encapsulan algoritmos genéricos que pueden ser accedidos por más de una clase.
            - Añadirlos a clases existentes disminuiría la cohesión.
            - También pueden añadir clases de librería o aplicaciones/funciones que no son OO.
        - **Librerías**: referencias a librerías proporcionadas por el entorno.
        - **Interfaces**: abstracción de comportamiento.
        - **Clases de ayuda**: Asisten a una clase ya existente para la realización de una tarea.
            - Aumentan la cohesión de la clase origen.

Responsabilidades

- Definición: son un contrato u obligación de un clasificador. Están relacionados con las obligaciones o comportamiento de un objeto en términos de su rol.
- Tipos:
    - **Doing**
        - Hacer algo él mismo.
        - Iniciar la acción en otros objetos.

        - Controlar y coordinar actividades en otros objetos.
    - **Knowing**
        - Conocer los datos privados (encapsulados).
        - Conocer los objetos con los que se relaciona.
        - Conocer las cosas que puede derivar o calcular.
        - Estas responsabilidades suelen estar relacionadas con el modelo de dominio (debido a los atributos y asociaciones que ilustra).
- Responsabilidades vs métodos
    - La complejidad del proceso de traducción de responsabilidades a clases y métodos, está influenciada por la granularidad de la responsabilidad.
        - Cuantas más clases estén implicadas, mayor complejidad tendrá.
    - Una responsabilidad no es un método, pero un método puede satisfacer una responsabilidad.
    - Las responsabilidades tienen asociada la idea de colaboración con otros métodos y/u objetos
- Diagramas de interacción
    - La descripción de las responsabilidades se pueden definir a la hora de codificar y a la hora de modelar.
    - Durante el modelado, los diagramas de iteración, nos dan una descripción de las responsabilidades.

GRASP

- General Responsibility Assignment Software Patterns.
- Definen nueve principios básicos del diseño OO (objetos y asignación de responsabilidades).

Patrones GRASP básicos

Creador

Nombre:	Creator
Problema:	¿Quién crea un A?
Solución:	Asignar a la clase la responsabilidad de crear una instancia de la clase A si se cumple alguna de las siguientes condiciones (cuántas más mejor) • B contiene o tiene agregado A • B graba A • B usa de forma cercana A • B tiene la información para inicializar A
Discusión:	• Creator busca el objeto creador que requiere estar conectado al objeto creado en cualquier momento. • Soporta bajo acoplamiento. • Es común que el creator haya de buscar la clase que tiene los datos de inicialización que serán pasados por parámetro.
Contraindicaciones:	La creación podría ser compleja (creación condicional, reciclamiento de instancias…). En este caso se ha de hacer uso del patrón Factory (GoF)
Beneficios:	Favorecer el bajo acoplamiento lo que implica: • Bajo nivel de dependencias de mantenimiento. • Alta oportunidad de reuso.

Patrones y principios relacionados:	• Bajo acoplamiento. • Patrón factoría (concreta o abstracta). • Whole-Part: describe un patrón para definir objetos agregados que soportan la encapsulación de componentes.

Experto (en información)

Nombre:	Information expert
Problema:	¿Cuál es el principio general de asignación de responsabilidades a objetos?
Solución:	Asignar responsabilidades a la clase que tiene la información necesaria para satisfacerla.
Discusión:	• Los objetos hacen tareas relacionadas con la información que poseen. La realización de la tarea muchas veces requerirá de información de distintas clases (aplicado en cascada). • Principio de animación (caricatura animada): los objetos en la programación OO son objetos vivos o animados que tienen responsabilidades y realizan cosas. • Tal como en el mundo real, las tareas las hacen quienes tienen la información para realizarlas.
Contradicciones:	Problemas de cohesión y acoplamiento.
Beneficios:	• Mantiene encapsulamiento de información (bajo acoplamiento) • Motiva clases ligeras distribuyendo responsabilidades (alta cohesión).
Principios relacionados:	• Bajo acoplamiento. • Alta cohesión.
Conocido como:	• “Place responsibilities with data” • “That which knows, does” • “Do it myself” • “Put services with the attributes they work on”

Bajo acoplamiento

Nombre:	Low coupling
Problema:	¿Cómo reducir el impacto del cambio?
Solución:	Asigna responsabilidades de forma que el acoplamiento sea mínimo.
Discusión:	• No hay una medida específica para el acoplamiento, pero en general, clases genéricas y candidatas al reuso deberían asegurar bajo acoplamiento. • Siempre habrá algo de acoplamiento entre objetos (ya que existe colaboración entre ellos).
Contradicciones:	• Rara vez es problema tener alto acoplamiento en elementos estables y de uso amplio. • No vale la pena desgastarse en disminuir acoplamiento cuando no hay una motivación real.
Beneficios:	• No existen efectos colaterales debido a cambios en otros componentes. • Simple de entender aisladamente. • Muy conveniente para el reuso. • La mayoría de patrones favorecen el bajo acoplamiento.
Background:	El acoplamiento y la cohesión son principios fundamentales en el diseño, y la mayoría de patrones favorecen el bajo acoplamiento.

Patrones relacionados	Protected Variation

- **Acoplamiento**: medida que indica cómo de fuertemente un elemento está conectado, tiene conocimiento o depende de otros elementos.
- Alto acoplamiento:
  - La clase depende de muchas clases.
  - Cambios en las clases relacionadas fuerzan cambios en la clase afectada.
  - La clase afectada es más difícil de entender por sí sola.
  - La clase afectada es más difícil de reutilizar, porque requiere la presencia adicional de las demás clases de las que depende.
- Tipos de acoplamiento:
  - Definición de atributos: x tiene un atributo que refiere a una instancia de y.
  - Definición de interfaces de métodos: un parámetro o una variable local de tipo y se encuentra en un método de x.
  - Definición de subclases: x es una subclase de y.
  - Definición de tipos: x implementa la interfaz y.

Controlador

Nombre:	Controller
Problema:	¿Cuál es el primer objeto por debajo de la interfaz de usuario que debe recibir y coordinar el mensaje del usuario?
Solución:	Asignar responsabilidades al objeto que representa una de estas opciones: • Representa el sistema completo (control fachada). • Representa un escenario de un caso de uso donde la operación se encuentra (control de sesión).
Discusión:	• La capa interfaz no debe contener lógica de aplicación. • El controlador en sí no realiza muchas tareas, sólo delega en otros. • El controlador fachada será el que se encargue de todos los eventos, por lo que es un único controlador. • El controlador sesión sólo de alguno, por lo que habrá varios controladores pequeños.
Beneficios:	• Incrementa la posibilidad del reuso y la conexión de interfaces. • Permiten implementar una secuencia de operaciones o mantener el estado del caso de uso.
Patrones relacionados:	• Command • Facade • Layers: a POSA pattern. Placing domain logic in the domain layer rather than the presentation layer is part of the Layers. • Pure fabrication: a GRASP pattern. Arbitrary creation of designer, not a software class whose name is inspired by de domain model. A use case controller is a kind of Pure Fabrication.

Alta cohesión

Nombre:	High cohesion
Problema:	¿Cómo mantener los objetos centrado, entendible y manejable, y que

	además soporte el bajo acoplamiento?
Solución:	Asignar responsabilidades de forma que la cohesión se mantenga alta. Lo mejor es evaluar alternativas.
Discusión	Regla general: Una clase con alta cohesión tiene un número relativamente bajo de operaciones, con funcionalidad altamente relacionada y no hace demasiado trabajo, colabora y delega.
Contraindicaciones:	Existen pocos casos que contraindiquen la alta cohesión: • Agrupar responsabilidades o código en una sola clase o componente para simplificar el mantenimiento por una sola persona. • Servidor de objetos distribuidos, debido a las implicaciones de overhead o performance asociadas a objetos remotos y a la comunicación remota. Relativo al patrón interfaz remota de granularidad gruesa.
Beneficios	• Aumento de la claridad y fácil comprensión del diseño. • Mantenimiento y mejoras simplificadas. • Bajo acoplamiento. • Fina granularidad en el reuso, la funcionalidad aumenta porque una clase cohesiva puede ser usada para propósitos muy específicos.

- **Cohesión**: medida de cómo de fuertemente se relacionan y enfocan las responsabilidades de un elemento.
- Baja cohesión:
    - La clase realiza muchas actividades poco relacionadas o realiza demasiado trabajo.
    - Problemas:
        - Difíciles de entender.
        - Difíciles de usar.
        - Difíciles de mantener.
        - Delicados: fácilmente afectables por cambios.
- Grados de cohesión:
    - Muy baja: clase responsable de muchas cosas en muchas áreas distintas.
    - Baja cohesión: clase responsable de una tarea compleja en un área funcional.
    - Alta cohesión: clase que tiene responsabilidades moderadas en un área funcional y colabora con otras clases para realizar una tarea.
    - Moderada cohesión: clase que tiene pocas responsabilidades en varias áreas funcionales, estando éstas relacionadas lógicamente con la clase pero no entre ellas.

Patrones GRASP avanzados

Polimorfismo / Polymorphism

Problema. ¿Cómo manejar alternativas basadas en el tipo? ¿Cómo crear componentes software para conectar?

- Alternativas basadas en el tipo: usando sentencias if-else o case, cuando lleguen nuevas variaciones serán mucho más complicadas de controlar.

- Componentes conectados: cómo reemplazar un componente con otro sin que afecte al cliente.

Solución. Cuando alternativas o comportamientos relacionados varían por el tipo, asigna la responsabilidad del comportamiento usando "operaciones polimórficas" (usa el mismo nombre a servicios en diferentes objetos) a los tipos para los cuales el comportamiento varía.

Discusión. El uso de polimorfismo para la asignación de responsabilidades hace que el sistema sea más abierto a manejar nuevos cambios similares. En la mayoría de lenguajes, aplicar polimorfismo implica el uso de clases abstractas. ¿Cuándo debe considerarse el uso de interfaces? La respuesta es introducirlas cuando quieres soportar el polimorfismo sin comprometerse a una jerarquía de clases. Una regla general usando jerarquía de clases, es que la clase base implemente una interfaz con las signaturas de los métodos públicos de la misma.

Contraindicaciones. No utilizar el polimorfismo si no se esperan posibles cambios, ya que el polimorfismo conlleva tiempo y esfuerzo.

Beneficios

- Extensiones para el manejo de variaciones son más fáciles de añadir.
- Se pueden introducir nuevas implementaciones sin necesidad de afectar al cliente.

Patrones relacionados

- Protected Variation.
- Muchos patrones GoF: adapter, command, composite, proxy, state and strategy.

También conocido como: Choosing message, Don't ask "what kind?"

Fabricación pura / Pure Fabrication

Problema. ¿Qué objeto debería tener la responsabilidad cuando no se desean violar los principios de alta cohesión y bajo acoplamiento, pero las soluciones sugeridas como Expert no son apropiadas?

Solución. Asignar un conjunto de responsabilidades altamente cohesivas a una clase artificial que represente el un concepto del dominio del problema. Pure fabrication implica que nos inventamos totalmente algo que soporte la alta cohesión, el bajo acoplamiento y el reuso.

Este principio es el que se utiliza en la aplicación de Helper y clases Utiliity.

Discusión. El diseño de objetos se divide en dos grandes grupos:

- Los diseñados por **descomposición representacional**.
- Los diseñados por **descomposición conductual**.

Es el segundo caso el más común para objetos de fabricación pura. Las clases de este estilo son inventadas (ya que no representan a ningún objeto del dominio) para concentrar comportamiento común.

Beneficios.

- Alta cohesión, porque las responsabilidades han sido refractorizadas en una clase de fina granularidad enfocada a un grupo de tareas muy específicas.
- Aumenta el potencial del reuso por la fina granularidad de estas clases.

Contraindicaciones. En ocasiones se sobreutiliza por novatos a la hora de dividir el software que lo dividen en términos de funciones, creado clases algorítmicas.

Patrones y principios relacionados.

- Bajo acoplamiento.
- Alta cohesión.
- Expert pattern, se basa en él a la hora de asignar responsabilidades.
- Todos los GoF patterns, como adapter, command, strategy son pure fabrication.
- Virtually.

Indirección / Indirection

Problema. ¿Dónde asignar responsabilidades para evitar acoplamiento directo entre dos o más cosas? ¿Cómo desacoplar objetos de forma que se mantenga un bajo acoplamiento y se soporte el reuso?

Solución. Asignar la responsabilidad a una clase intermedia que medie entre los otros componentes o servicios de forma que se acoplen directamente. De esa forma se crea una indirección entre los componentes.

Discusión. Un dicho en programación dice que "la mayoría de problemas en la computación se resuelven añadiendo otra capa de indirección". Este tiene una particular relevancia a la hora del diseño OO. Muchos patrones son especializaciones de la fabricación pura, otros también lo son de la indirección (adapter, facade, observer). Muchas fabricaciones puras son generadas a causa de la indirección.

Beneficios. El bajo acoplamiento. La principal motivación de uso de este patrón es mantener el bajo acoplamiento o desacoplar componentes o servicios.

Patrones y principios relacionados.

- Protected Variations.
- Low coupling.
- Muchos del os patrones GoF como adapter, bridge, facade, observer y mediator.
- Muchos indirecciones son fabricaciones puras.

Protección de variaciones / Protected variations

Problema. ¿Cómo diseñar objetos, subsistemas y sistemas de forma que las variaciones o inestabilidad de los elementos no tengan un impacto negativo sobre otros?

Solución. Identificar los puntos de variación o inestabilidad (hot spots) y asignar responsabilidades para crear una interfaz estable a su alrededor. Interfaz se usa en el más extenso sentido de la palabra, no significa literalmente algo como una interfaz Java.

Discusión. Este es un principio de diseño software fundamental y muy importante. Casi todos los trucos o consejos de diseño software o arquitectural son especializaciones de este patrón.

Contraindicaciones. Hay que tener cuidado a la hora de predecir los puntos de variaciones o evolución. No se ha de trabajar más haciendo un diseño perfecto para cambios si estos no se van a producir. Se ha de encontrar un punto medio en el proyecto.

Beneficios.

- Extensiones necesarias para nuevas variaciones son fáciles de añadir.
- Se pueden introducir nuevas implementaciones sin afectar a los clientes.
- Disminuye el acoplamiento.
- El impacto y coste de los cambios disminuye.

Patrones y principios relacionados. Muchos de los patrones y principios de diseño son mecanismos de protección de variaciones.

Conocido como. Antes de ser establecido como un patrón, este concepto se conocía como ocultación de la información (hidding information) o principio de abierto/cerrado (open-closed principles).

- Ocultación de la información: no se debe confundir con el concepto de encapsulación. Lo que se ha de ocultar es la información acerca del diseño de otros módulos, en los puntos difíciles o de posible cambio.
- Principio de abierto/cerrado: los módulos han de ser abiertos a extensiones (adaptables), pero cerrados a modificaciones que afecten a los clientes. No siempre es posible conseguir este principio de manera completa, pero the most the better.

Principios de diseño motivados por PV

- Encapsulación.
- Diseñar operaciones para que consulten o modifiquen pero no ambas cosas a la vez.
- Separación del Modelo-Vista: los objetos del modelo no deberían conocer lo objetos de presentación, para proporcionar bajo acoplamiento de capas inferiores a la superior, interfaz, que es la que más cambia.
- Principio de sustitución de Liskov: Toda instancia de una clase derivada puede tomar el lugar una instancia de su clase base.
- El principio open-closed.
- Principio de inversión de dependencia:
    - Los módulos de alto nivel no deberían depender de módulos de bajo nivel. Todos deberían depender de abstracciones.
    - Las abstracciones no deberían depender de detalles, sino al contrario: los detalles deberían depender de las abstracciones.
    - Implicación: el acoplamiento entre objetos usados y que usan debe ser a nivel conceptual, sin tener en cuenta los detalles concretos de la implementación.
    - Llevarlo al extremo:
        - No deberían existir variables que referencien a clases concretas.
        - Ninguna clase debería derivar de una clase concreta.
        - Ningún método debería sobreescribir un método implementado de una de sus clases base.

- El principio de segregación de interfaz (interface segregation principle): Los clientes no deben ser forzados a depender de interfaces que no usan. Es mejor tener muchas interfaces específicas que una muy general.
- Principio de equivalencia de liberación y reuso (reuse/release equivalency principle): La granularidad del reuso ha de ser la misma que la de la liberación. Solo los componentes que son liberados pueden ser reusados de forma efectiva.
- Principio del reuso común (common reuse principle): Las clases que no son reusadas juntas no han de estar agrupadas juntas.
- Principio de cierre común (common closure principle): Las clases que cambian juntas, permanecen juntas.
- Principio de abstracciones estables (stable abstraction principle): Cuanto más estable sea una categoría de clases, más razón para que éstas sean abstractas.
- Principio de la mínima sorpresa (least astonishment principle): Cuando existen dos elementos de una interfaz conflictiva o ambigua, su comportamiento debe ser lo menos inesperado posible ya que ese es normalmente el correcto.
- Principio de jerarquías profundas y abstractas (deep abstract hierarchies principle): Las jerarquías de clases deben profundas y abstractas.
- Principio de dependencias acíclicas (acyclic dependendies principle): No deben existir ciclos en los grafos de dependencias.
- Principio de dependencias estables (stable dependencies principle): La dependencia debe ir en dirección a la estabilidad, es decir, un componente debe depender sólo de componentes más estables que él.
- No hablar con extraños (don't talk to strangers), ley de Demeter: Cada componente debe relacionarse con otros con los que esté relacionado de alguna forma (no son extraños). Así evitar acoplamientos con objetos indirectos y protegerse de cambios estructurales.
    - Al objeto this/self.
    - Un atributo de this.
    - Un elemento de una colección que es atributo de this.
    - Un parámetro del método.
    - Un objeto creado dentro del método.
- ID to Objects: Convertir las claves de identificación en objetos lo antes posible. El objeto real flexibiliza la aplicación según el diseño crece.
- Pasar objeto agregado como parámetro: cuando una operación requiere como parámetros objetos que están agregados dentro de otros, se ha de pasar el objeto agregado. Aumenta la flexibilidad del sistema.

Patrones GoF

Patrones creacionales

Abstract Factory

Contexto/Problema. ¿Cómo crear familias de clases relacionadas que implementen una interfaz común?

Solución. Definir una interfaz factoría (factoría abstracta) y factorías concretas para cada una de las familias. Opcionalmente puedes crear una verdadera clase abstracta que implemente la interfaz factoría y que provea de los servicios comunes al resto de factorías concretas que la extiendan.

Provee una interfaz para crear familias de objetos-producto relacionados o que dependen entre sí, sin especificar sus clases concretas.

Una variación bastante común es la de crear una clase abstracta factoría que utiliza el patrón singleton. De esa forma los objetos puede colaborar con la superclase abstracta y obtener los objetos concretos que necesiten.

Factory Method

Modificación del abstract factory que define una interfaz para crear un objeto delegando la decisión de qué clase crear en las subclases. Este onfoque también puede ser llamado constructor “virtual”.

~~Singleton~~

Builder

Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones. Simplifica la construcción de objetos con estructura interna compleja y permite la construcción de objetos paso a paso.

~~Prototype~~

Patrones estructurales

~~Adapter~~

~~Bridge~~

Composite

Compone objetos en estructuras de árboles para representar jerarquías parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los complejos.

Façade

Proporciona una interfaz simplificada para un conjunto de interfaces de subsistemas. Define una interfaz de alto nivel que hace que un subsistema sea más fácil de usar.

~~Decorator~~

~~Flyweight~~

Proxy
Provee un sustituto o representante de un objeto para controlar el acceso a éste. Este patrón posee las siguientes variantes:

- Proxy remoto: se encarga de representar un objeto remoto como si estuviese localmente.
- Proxy virtual: se encarga de crear objetos de gran tamaño bajo demanda.
- Proxy de protección: se encarga de controlar el acceso al objeto representado.

Patrones de comportamiento

~~Chain of responsibility~~

Command
Representa una solicitud con un objeto, de manera tal de poder parametrizar a los clientes con distintas solicitudes, encolarlas o llevar un registro de las mismas, y poder deshacer las operaciones. Estas solicitudes, al ser representadas como un objeto también pueden pasarse como parámetro o devolverse como resultados.

~~Interpreter~~

~~Mediator~~

~~Iterator~~

~~Memento~~

Strategy
Define una jerarquía de clases que representan algoritmos, los cuales son intercambiables. Éstos pueden ser intercambiados por la aplicación en tiempo de ejecución.

Observer
Brinda un mecanismo que permite a un componente trasmitir de forma flexible mensajes a aquellos objetos que hayan expresado interés en él. Estos mensajes se disparan cuando el objeto ha sido actualizado, y la idea es que quienes hayan expresado interés reaccionen ante este evento.

~~State~~

~~Template Method~~

~~Visitor~~