Introducción a la Ingeniería de Software

Juan Carlos Vargas

Que es?

Tres grandes áreas: - modelación: conceptos → requerimientos → diseño → código → ejecutable - diseño: encontrar solución que responda a requisitos - optimización: encontrar transformaciones de mejor calidad y económicas

Procesos de Desarrollo de Software

Idea de proceso

Para construir software se requiere un proceso bien definido. Dentro de las principales actividad se encuentra: - Estudio de factibilidad - Ingeniería de requisitos - Diseño de UI - Diseño de arquitectura - Diseño detallado - Programación - etc...

Desarrollar software sin proceso alguno se suele denominar 'hacking'.

Modelos de Proceso

Cascada

Es un procedimiento lineal que se caracteriza por dividir los procesos de desarrollo en sucesivas fases de proyecto. Al contrario que en los modelos iterativos, cada una de estas fases se ejecuta tan solo una vez.

NO FUNCIONA SALVO EN PROYECTOS MUY CORTOS

Iterativos

Es un procedimiento repetitivo en el cual a través de una secuencia de iteraciones se acerca cada vez mas a la meta.

Ejemplos: - Modelo espiral: análisis de requisitos, construcción, evaluación y análisis de errores ← fue el primero en considerar el riesgo de forma explicita - Modelo de prototipos: se crean dos prototipos, desechables y evolucionarios - Modelo unificado: cuatro fases en las que se llevan a cabo muchas actividades, pero en cada una se enfoca en algo especifico: incepción, elaboración, construcción y transición (RUP)

MODELOS ITERATIVOS PROVOCAN LA DISMINUCIÓN GRADUAL DEL RIESGO

Ágiles / Iterativos Incrementales

En cada iteración de produce un incremente de valor para el cliente (un relato de usuario).

En los procesos ágiles se destacan cuatro valores: - individuos e interacciones por sobre procesos y herramientas - software que corre por sobre documentación - colaboración con el cliente por sobre negociación de contratos - responder a los cambios en lugar de apegarse a un plan

Los procesos ágiles mas populares son Scrum y Kaban

Scrum

Es el modelo de proceso ágil más popular.

Funciona así: - Se divide el trabajo en pedazos llamados sprints - Cada sprint agrega valor para el cliente al producto - Lo que se hace en cada sprint se decide junto al "product owner" - Al final de cada sprint hay dos reuniones - review - que logramos y que falta - retrospectiva - como funcionó el proceso

Existen 3 roles en Scrum: - Product Owner - decide que features se incorporan. Es la voz del cliente - Scrum Master - Líder servicial para el equipo Scrum. Debe asegurar un ambiente de trabajo productivo y garantizar que Scrum se cumpla - Scrum Team - Equipo de desarrollo, trabajan en las historias de usuario pendientes para el sprint

Existe un "backlog" que lleva cuenta de cuales son los items o tareas que deben ser desarrolladas. Esta lista se revisada y prioriza constantemente

Cada día durante el sprint se hace una reunion de pie de 15 - 20 min para hablar sobre las cosas que se hicieron el día anterior, las que harán hoy y los problemas que tienen. NO SE RESUELVEN LOS PROBLEMAS EN ESTA INSTANCIA

También se hacen Sprint Reviews, donde se inspecciona y adapta el producto a medida que transcurren las iteraciones.

Sprint Retrospective es la segunda reunion del final de sprint, se ven formas de mejorar el proceso.

Requisitos

Existen 2 tipos:

- Funcionales - Añaden funcionalidades
- No Funcionales - Agregan restricciones

Los requisitos se deben levantar centrados en el usuarios. Los requisitos deben capturar como el sistema va a ayudar al usuario.

User Stories

Es una descripción informal y en lenguaje natural de las features que requiere un programa. Generalmente se escriben en un post-it o index card

Estructura

```
 Yo, como _____ necesito ______ para _____________
          (ROL)          (META)      (JUSTIFICACION)
```

La justificación es para evitar añadir funcionalidades que no son necesarias y para entender mejor la funcionalidad

En el reverso se ponen las condiciones de satisfacción, las cuales son "tests de validancias" para el avance de la funcionalidad.

```
Condiciones de satisfaccion

-
-
-
```

El nivel de detalle de un relato de usuario debe ser tal como para que la funcionalidad se pueda implementar en unos pocos días y por lo tanto desarrollado durante un sprint

Los relatos no deben tener interdependencias, gold platting, demasiado detalle o tener un rol muy genérico.

Épicas y Temas

La idea es poner los relatos de usuario en un contexto mayor. Esto se puede hacer a través de épicas y temas.

Épica: Objetivo mayor que requiere ser separado en varios relatos

Tema: Agrupación de relatos relacionados de algún modo

Casos de Uso

A veces una persona puede ejercer mas de un rol, por lo que es útil capturar como un actor usa un sistema.

Cada Caso de Uso tiene 3 tipos de actores:

- Principales: Inician el CU
- Secundarios: Participan pero no lo inician
- Fuera de Escena: No participan pero impactan en la secuencia

Para hacer un caso de uso se grafican los distintos flujos posibles que podrían haber, con sus flujos alternativos

Planeación

En software es muy difícil planificar un proyecto por completo desde un principio. La parte mas difícil es estimar el tiempo y esfuerzo de desarrollo.

Niveles de Planeación

En los procesos ágiles se realizan varias iteraciones con sus niveles de iteraciones. - Primer Nivel (Iteración): Estimar y seleccionar los relatos de usuarios a ser implementados

- Segundo Nivel: (Release): Estimar la cantidad de iteraciones a realizar y las funcionalidades a incorporar. (backlog inicial)
- Tercer Nivel (Producto): Funcionalidades incorporadas en cada release, en el próximo año y así
- Portafolio: Que productos se van a hacer

Planeación a Nivel de Release

Existen distintos tipos de ajuste que se pueden hacer si es que no se puede cumplir la meta de tiempo:

- Ajuste de Tiempo: Re negociar la fecha de entrega
- Ajuste por Alcance: Re negociar las funcionalidad que serán incluidas
- Ajuste por Calidad (NO DEBE HACERSE): Producto con todas las funcionalidades pero mas baja calidad

Estimaciones

Una planificación adecuada requiere conocer

- Tiempo necesario para realizarlo (semanas)
- Esfuerzo requerido (semanas-hombre)

No hay que dar una estimación improvisada nunca

Es mucho mejor sobrestimar que subestimar en cuanto a penalizaciones.

Fuentes de error en las estimaciones

- Información imprecisa sobre le proyecto
- Imprecisiones sobre capacidad del equipo
- Imprecisiones al estimar
- El proyecto es “blanco móvil”

Tamaño

En el software ocurre que mientras más grande el proyecto, menos productividad existe.

Puntos de Función

Esta métrica se caracteriza por 5 factores numero y complejidad de: - inputs - outputs - consultas - archivos internos - archivos externos

Relatos y Puntos de Relato

Se califican los relatos del usuario con algún multiplicador que diga la complejidad

Para estimar que tanto tan difícil va a ser un relato, se suelen usar cartas con alguna escala como Fibonacci o exponencial, y cada persona del grupo califica en forma secreta que tan complejo cree que va a ser.

Kaban

Se basa en tres principios fundamentales: - Visualizar lo que se hace hoy - Limitar el trabajo en progreso - Optimizar el flujo de tareas

En un tablero Kanban, el trabajo se muestra en un proyecto en forma de tablero organizado por columnas.

Diseño

Tradicionalmente la etapa de diseño sigue a una etapa de análisis que cierra el levantamiento te requisitos.

- Análisis: Que
- Diseño: Como

Pero en el desarrollo ágil la separación entre diseño y análisis es casi imperceptible.

Durante el análisis y diseño se buscan construir dos modelos distintos

Modelos de Dominio (Análisis)

Modela clases de dominio, principales atributos y relaciones entre ellas.

Clases de dominio

Candidatos Se buscan una vez ya se levantaron los requisitos funcionales. (EJ: Roles, eventos, ubicaciones, unidades organizacionales, etc)

Se suelen encontrar muchos candidatos pero solo algunos de ellos terminarán siendo Clases. Algunos criterios para evaluar candidatos son: - Cumple que

guarda alguna información - Se puede pensar en mas de un atributo - Se puede pensar en métodos - Todas las instancias competirán esos mismos atributos y métodos

Clase, responsabilidad, colaborador (CRC)

- Responsabilidades: Lo que el objeto sabe (atributos) y hace (métodos)
- Colaboradores: Objetos de otras clases necesarios para que el objeto haga lo que hace

Del análisis al diseño

Es importante tener criterios para decidir si un diseño es superior a otro (responden de al mismo cómo)

La complejidad del problema hará necesario descomponer la solución en elementos menores.

Los atributos que caracterizan un buen diseño pueden ser: - fácil de entender - fácil de modificar y extender - fácil de reutilizar - fácil de testar la implementación - fácil de integrar las disientas unidades - fácil de programar

Acoplamiento y cohesión

El acoplamiento debe ser bajo y la cohesión alta. Esto caracteriza un buen diseño

Acoplamiento de peor a mejor: - Por contenidos (acceso a datos internos) - Por variables globales - De control (una unidad influye en el flujo de la otra) - De datos (paso de parametros)

Cohesión de peor a mejor: - Coincidental (arbitraria) - Lógica (hacen lo mismo pero son diferentes) - Temporal (por cuando son procesadas) - Procedural (agrupadas por el orden de ejecución) - Comunicacional (operan en el mismo tipo de datos) - Secuencial (output de una es el input de la otra) - Funcional (contribuyen a una tarea bien definida)

UML: lenguaje de modelación unificado

Diagrama de clases

Rectángulos denotan las clases y lineas representan sus asociaciones

```
-----------
Class Name |
-----------
Attribute  |
Attribute  |
```

```
----------
Operation  |
Operation  |
----------
```

A los atributos y método se le pueden añadir los caracteres "+, -, #, ~" - -: atributo privado - +: atributo publico - #: visibilidad protegida (solo subclases) - ~: mismo paquete

Asociaciones

Las clases de un modelo te relacionan entre si

- Rombo relleno: composición
- Rombo vacío: agregación
- Triangulo: herencia

Interfaces

"Contrato" de disponibilidad de ciertas operaciones

Se pueden mostrar igual que una clase pero con un "«interface»" añadido o con un circulo

Diagrama de secuencia

Se utiliza para visualizar como interactúan objetos en el tiempo Cada clase corresponde a un columnas, las interacciones entre ellas se ordenan de manera temporal de arriba hacia abajo y se representan con una flecha

Solo se utilizada una linea punteada de retorno si es que se devuelve un objeto

Diagrama de estados

Se representan los estados de de un determinado comportamiento con rectángulos redondeados. Se conectan estos estados con lineas que tienen escritas arriba su trigger y condición

Patrones de diseño

Un patron es la solución de un problema en un contexto dado. Los patronees son útiles debido a que se evita "reinventar la rueda" y se dan soluciones conocidas a problemas comunes

Existen 3 grandes grupos:

- Estructurales: Relaciones entre objetos
- Creacionales: Creación de objetos
- Comportamiento: Comunicación entre objetos

Adaptador

Permite que una clase existente pueda ser utilizada con una interfaz diferente.

Fachada (Facade)

Se utiliza un objeto como una interfaz simple para enmascarar un comportamiento mas complejo.

Singleton

Se restringe las instanciación de una clase de tal forma que solo se pueda acceder a una instancia

Observer

Permite que muchos objetos se suscriban a eventos que son hechos por un objeto en especifico. (se llama una funciona cuando los datos cambian)

Template

Hay una secuencia de pasos que se repite, así que creo un método con todos estos pasos en orden y luego sobreescribo los métodos los cuales estoy corriendo dependiendo de el tipo de clase

Strategy

El patrón Strategy sugiere que tomes esa clase que hace algo específico de muchas formas diferentes y extraigas todos esos algoritmos para colocarlos en clases separadas llamadas estrategias

Decorador

Permite cambiar dinámicamente el comportamiento de un objeto al envolverlo e una clase decoradora.

Composite

Se organizan las clases en forma de árbol, cada entidad esta formada por otra entidad formada del mismo tipo. La clase composite ejecuta cada uno de los petalos

Command

Se hace una clase con cada uno de los comandos que se quieren realizar. Cada una de estas clases tiene el método execute

Método fábrica

Se crean clases que instancian con un método que crea objetos

Fábrica abstracta

Se crea una fabrica abstracta con cada uno de los métodos que nos interesan y luego se hace una clase para cada fabrica especifica

Builder

Nos permite construir objetos complejos paso a paso. Se crea una clase Builder que contenga una instancia de la clase que queremos crear, después por cada atributo que queramos añadir hacemos un método que devuelva la instancia de si mismo (self) y luego el método build que devuelva la instancia de la clase que queremos crear

Equipos de Desarrollo de Software

Fases de maduración de un equipo

- Forming: Periodo exploratorio
- Storming: Se define poder control y liderazgo
- Norming: Se establece cohesion y apreciar diferencias
- Performing: Sentido de identidad, alto nivel de trabajo

Modelos de equipo según tarea predominante

Creatividad, resolución de problemas y ejecución táctica son los 3 tipos de tarea

Modelo de programador jefe (Chief programming team)

Un programador experto y rápido hace todo y el resto del equipo se encarga de hacer lo que el requiera

Es mejor cuando los equipos son pequeños (5-7)

Topología de equipos de desarrollo

En spotify tienen equipos pequeños de 5 a 7 personas denominados squads, varias squads relacionadas forman una tribu, especialistas de diversos squads de la misma tribu forman un chapter y de diversas tributos son una guild

Tipos de equipos

- Stream-aligned: scrum
- Enabling-team: tecnologías nuevas y updates

- Complicated-subsystem team: subsistemas complejos (motores de física o algoritmos)
- Platform team (API)

Tipos de colaboración

- Collaboration: Trabajo mano a mano
- X-as-a-service: Un equipo consume lo que el otro hace
- Facilitating: Un equipo ayuda al segundo a superar algún problema

Arquitectura de Software

La arquitectura influye en los requisitos no funcionales como por ejemplo el desempeño y escalabilidad

Para tomar decisiones sobre la arquitectura se deben tomar en cuenta 3 factores: - Atributos de calidad - Restricciones - Principios

Principios

Atributos de Calidad

Ejemplos: - Performance - Escalabilidad - Disponibilidad

Restricciones

Ejemplos: - Tiempo y presupuesto - Tecnológicas - Relativas a personas: Tamaño de equipo, habilidades

Principios

- Desarrollo: estándares de codificación, full unit testing, revision de código
- Arquitectónicos

Arquitectura vs Ágilidad

Es erróneo que la arquitectura debe pasar a un segundo plano en los desarrollos ágiles.

La arquitectura correcta puede habilitar la agilidad

Descripción de arquitectura

Una forma de mostrar la arquitectura es a través de un lenguaje gráfico llamado C4: - Contexto - Contenedores - Componentes - Código

Diagrama de Contexto

El foco es en las personas y los sistemas de software, no en las tecnologías y protocolos

Diagrama de Contenedores

Muestra la arquitectura de los sistemas en alto nivel

Diagrama de Componentes

Muestra los componentes de cada contenedor, muestra controladores e interacciones entre ellos

Código

UML

Arquitecturas comunes y patrones arquitectónicos

Cliente-Servidor

El cliente le pide una operación al servidor, el servidor le pide datos a la base de datos

Layered Architecture

Se van creando abstracciones de cada vez más alto nivel

Servicios

Los componentes están débilmente acoplados, corren en un proceso totalmente distinto

Microservicios

Servicios completamente independientes se comunican a través de protocolos ligeros

Documentación

Existen dos tipos: - Dirigida al usuario - Técnica

Costo vs Beneficio

Producir documentación tiene un costo, pero no hacerla suele tener costos mayores a futuro

La documentación del usuario evita que tengan que llamar al soporte La técnica que la mantención sea mas barata

Documentación técnica

Permite tener distintas visiones o perspectivas del producto

Buena documentación

- Esfuerzo equivalente al 10% del tiempo de desarrollo
- Control de versiones
- Código como componente de documentación técnica
- Documentos de diseño cortos y relevantes
- Racionalidad del diseño
- Lecciones aprendidas
- Formato estándar

Documentación de usuario

Es parte de la experiencia del usuario - know your customer - Lenguaje simple - Siempre lo mas simple - Apoyo visual - Screenshots con anotaciones - Foco en los problemas - Indice - Jerarquía - Buen diseño - Feedback de usuarios para hacerla - Links

Aseguramiento de Calidad

Identificar software de calidad

- Pocos defecto
- Alta fiabilidad y estabilidad
- Alta satisfacción
- Cumple los que los consumidores quieren
- Estructura de código y densidad de comentarios que minimice errores
- Servicio al consumidor efectivo
- Arreglos rápidos en caso de defecto

Verificación y Validación

- Verificación: Are we building the product right?
- Validación: Are we building the right product?

Hacia un software sin defectos

1. No incorporar errores al escribir código
2. Análisis estático del código
3. Inspección formal del código (equipos de pares)
4. Testing

Testing

Se debe revisar si el output es correcto para un cierto input

Tests Unitarios

Cada unidad se debe testear, existen dos metodologías - black box - white box

White Box Cada flujo posible es testeado y un porcentaje es asignado dependiendo de cuantas de estas se cubran #### Black Box Solo se revisan inputs y outputs, procurando que sean correctos. Se separan las posibilidades en clases de equivalencia y se testean

Mas allá de los tests unitarios

El enfoque clásico consiste en un uniendo las unidades y probando subconjuntos cada vez mas grandes. (tests de integración)

El enfoque moderno se llama integración continua, se testean lineas de inmediato y se hace un build para ser testeado

Los tests de regresión tienen por objetivo asegurar que los cambios nuevos no afecten a lo que ya existe

Testing

Tests de Integración

Se prueban si disantos módulos funcionan bien entre sí

Tests de Regresión

Después de añadir una funcionalidad nueva, corremos todos los otros tests

Smoke Test y Sanity test

Smoke test, si algo no se ve bien paramos de inmediato a revisarlo Sanity test, revisa si lo que se testeo tiene sentido y funciona bien con el resto de los módulos

Test de Sistema

Se prueba si el sistema funciona bien como un todo. Lo hacen profesionales o gente especializada

Test de Carga y Esfuerzo

Sirven para saber que el programa funcionará bien sobre cierto estrés

Tests de Usabilidad

5 a 8 usuarios utilizan la aplicacion y se monitorea su comportamiento

Test de Aceptacion

Se ve si satisface al usuario, se utilizan usuarios reales del producto