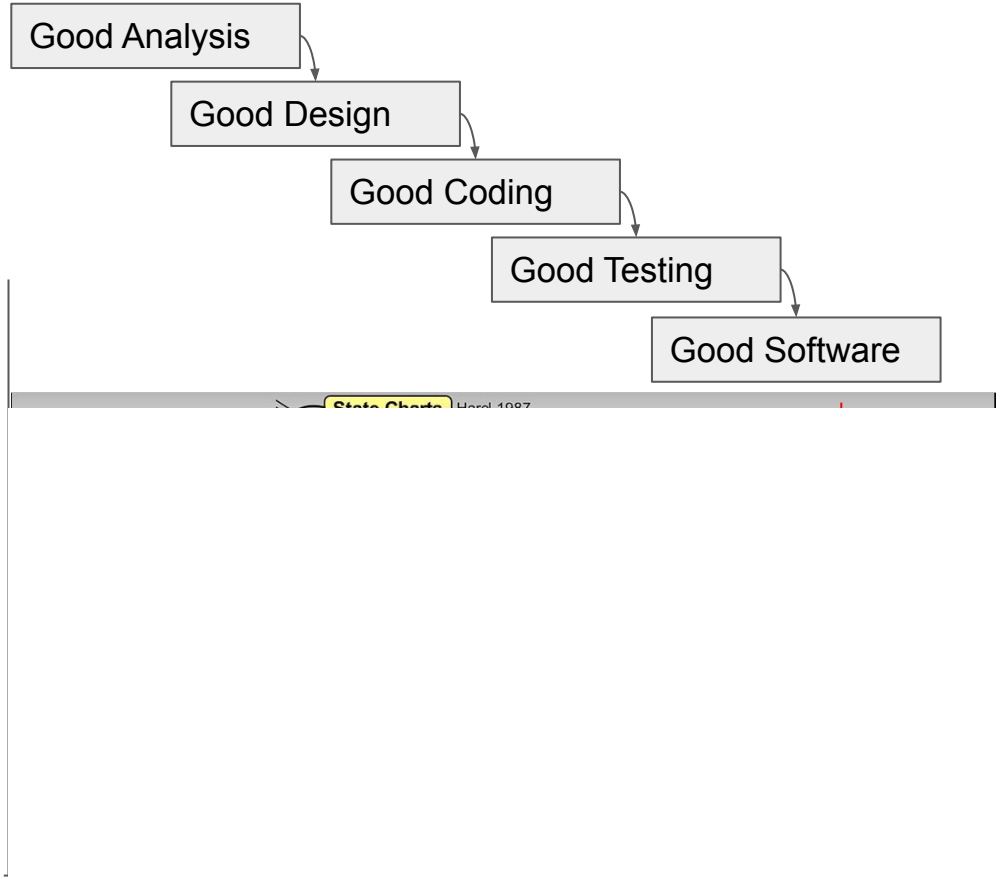
Desarrollo Dirigido por Pruebas

Test Driven Development


Good Analysis
Good Design
Good Coding
Good Testing
Good Software

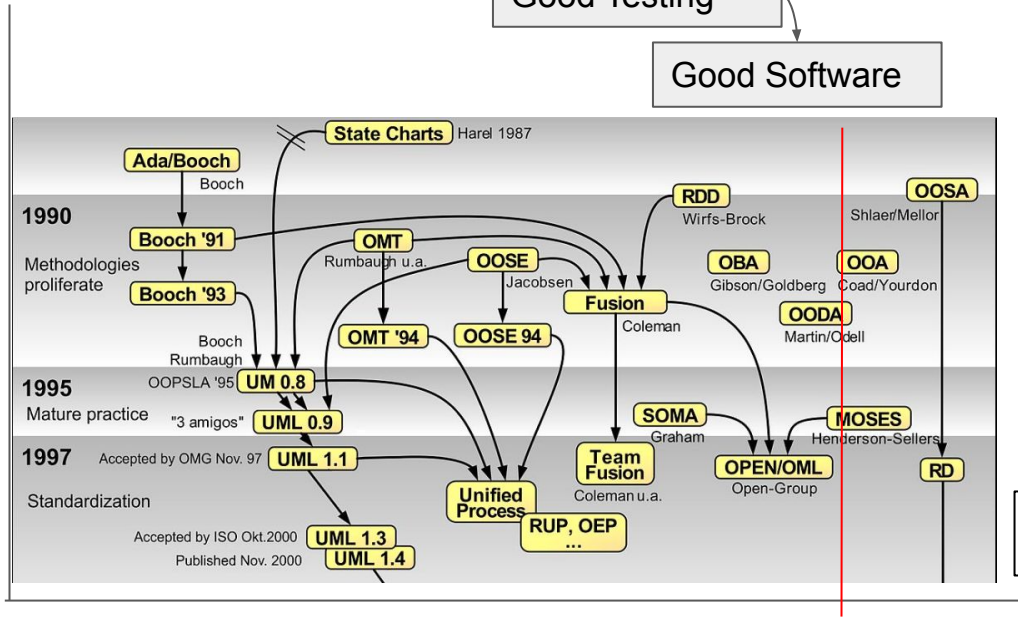
Good Analysis

Good Design

Good Coding

Good Testing

Good Software



1990

Change == Caos

2000==WTF

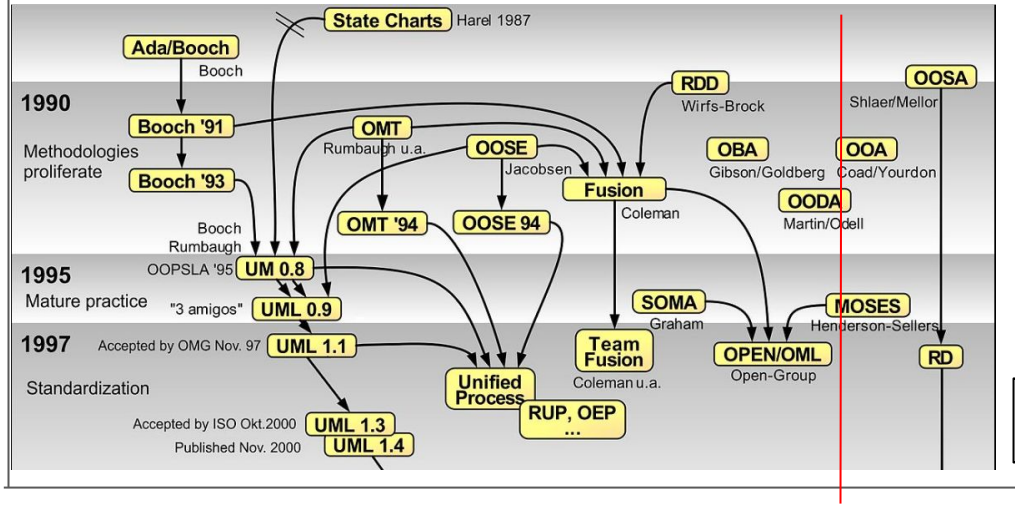
Good Analysis

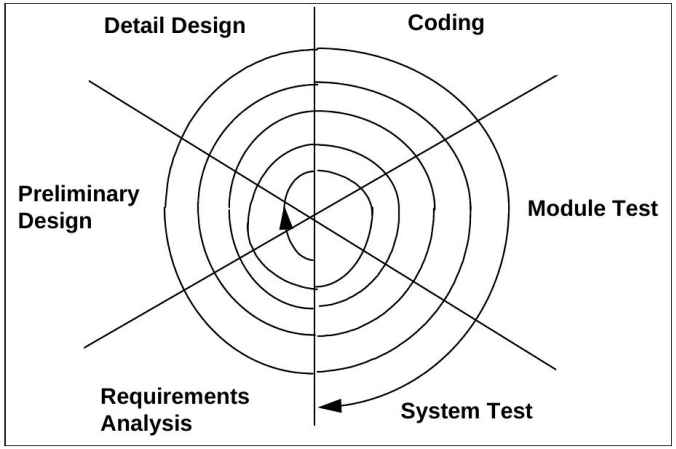
Good Design

Good Coding

Good Testing

Good Software





1990

Change == Caos

2000==WTF

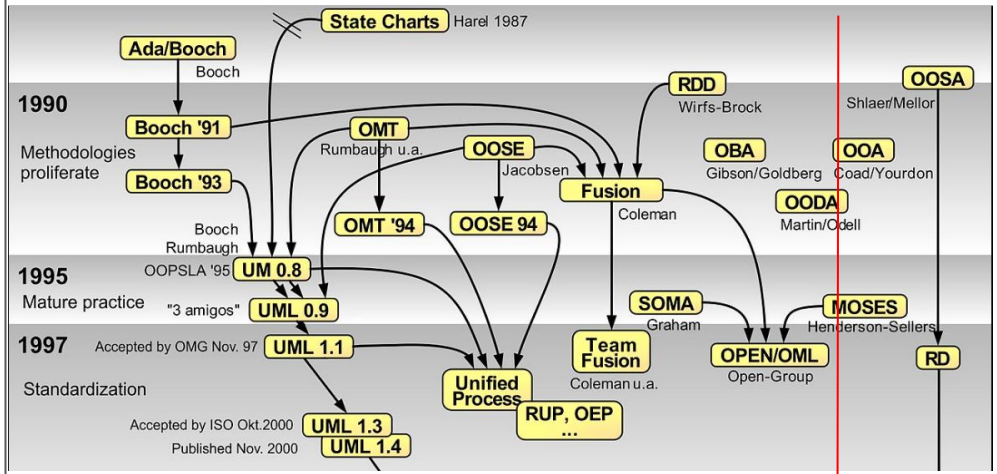
Good Analysis

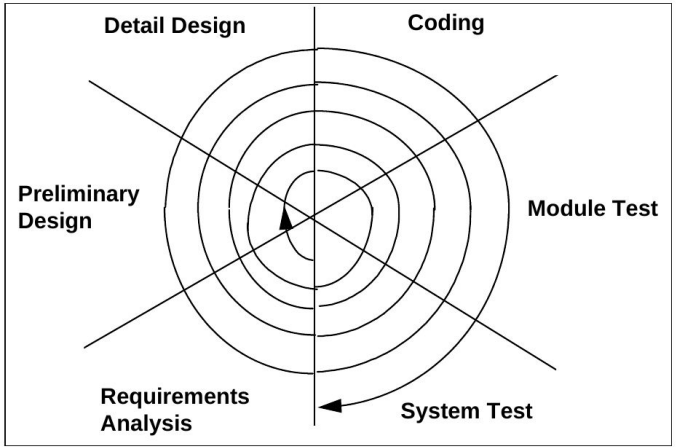
Good Design

Good Coding

Good Testing

Good Software





Kanban

TDD

XP

SCRUM

Agile

1990

Change == Caos

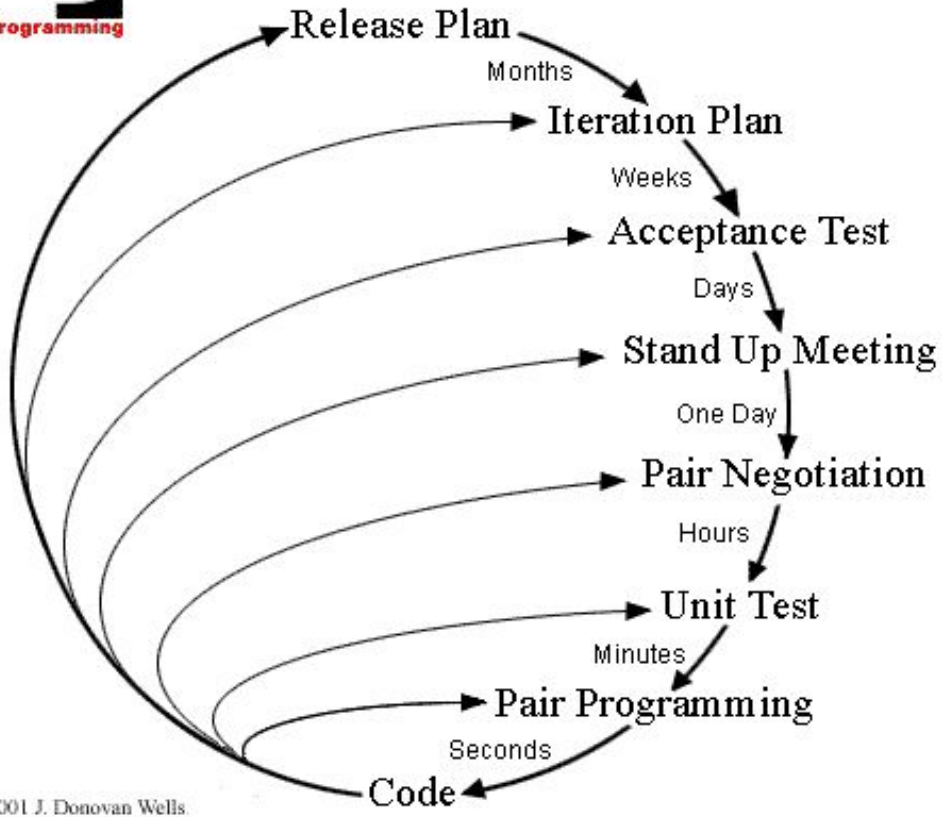
2000==WTF

Embrace change

Extreme Programming

Planning/Feedback Loops

Zoom Out



Características principales

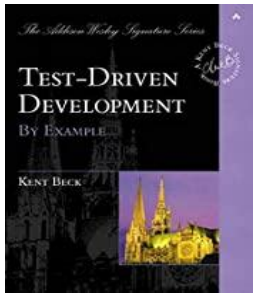
Agile Approach
Mismo grupo de personas para todo el desarrollo que trabajan en un mismo espacio
Comunicación de calidad
Desarrollo iterativo e incremental
Producto funcionando en cada “build”
Se valora el feedback
Cambios bienvenidos “changes embraced”

Por dónde empezar

- Si se debe ir tomando de a un requerimiento o pocos por vez para tener un producto funcionando al final de cada iteración, no cuento con un diseño completo para empezar a desarrollar

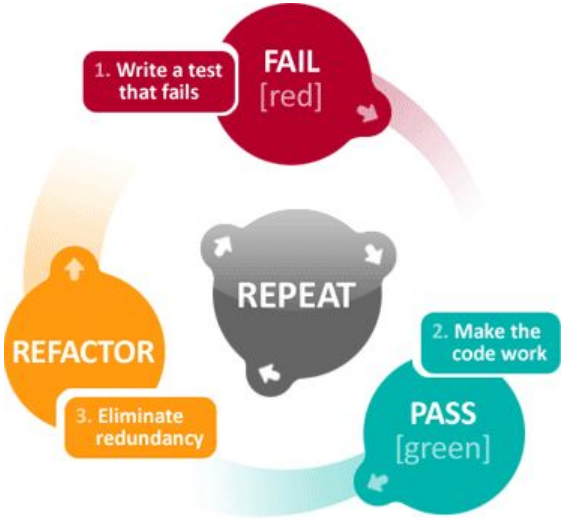
- Qué cosa guía el desarrollo? Por dónde empezar si no es de un diseño completo?

Test Driven Development



Test Driven Development

- Empezar por el test!



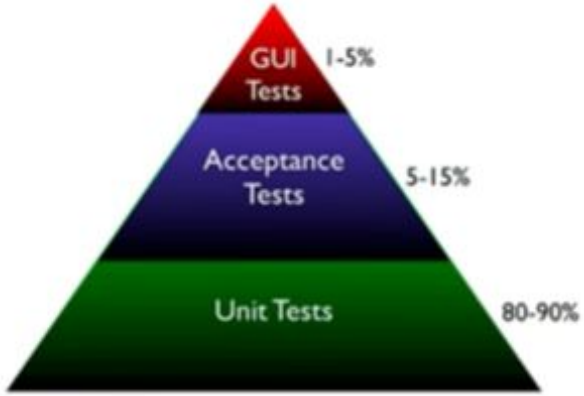
Test Driven Development (TDD)

- Combina:
  - *Test First Development*: escribir el test antes del código que haga pasar el test
  - ***Refactoring***
- Objetivo:
  - pensar en el diseño y qué se espera de cada requerimiento antes de escribir código
  - escribir código limpio que funcione (como técnica de programación)

¿Por qué no dejar testing para el final?

- Para conocer cuál es el final
- Para mantener bajo control un proyecto con restricciones de tiempo ajustadas (permite estimar)
- Para poder refactorizar rápido y seguro
- Para darle confianza al desarrollador de que va por buen camino
- Como una medida de progreso

Granularidad



- Test de aceptación
  - Por cada funcionalidad esperada.
  - Escritos desde la perspectiva del cliente

- Test de unidad
  - aislar cada unidad de un programa y mostrar que funciona correctamente.
  - Escritos desde la perspectiva del programador

Filosofía de TDD

- Vuelco completo al desarrollo de software tradicional. En vez de escribir el código primero y luego los tests, se escriben los tests primero antes que el código.
- Se escriben tests funcionales para capturar use cases que se validan automáticamente
- Se escriben tests de unidad para enfocarse en pequeñas partes a la vez y aislar los errores

Filosofía de TDD (cont.)

- No agregar funcionalidad hasta que no haya un test que no pasa porque esa funcionalidad no existe.
- Una vez escrito el test, se codifica lo necesario para que todo el test pase.
- Pequeños pasos: un test, un poco de código
- Una vez que los tests pasan, se **refactoriza** para asegurar que se mantenga una buena calidad en el código.

Trabajo en grupo (2 o 3)

- La app es entrenada con el perfil del usuario
- La app puede
  - Responder mensajes texto
  - Responder con mensajes de voz
  - Responder con imágenes
  - Tomar el lugar del usuario en llamadas entrantes
  - Organizar reuniones en función del calendario
  - Sugerir actividades como: “llamar a sus padres en el del dia del aniversario”
- La app puede ser des-entrenada por el usuario
- La app puede adaptarse a cambios del perfil del usuario

Escriba 5 test:

- Unidad
- Aceptación

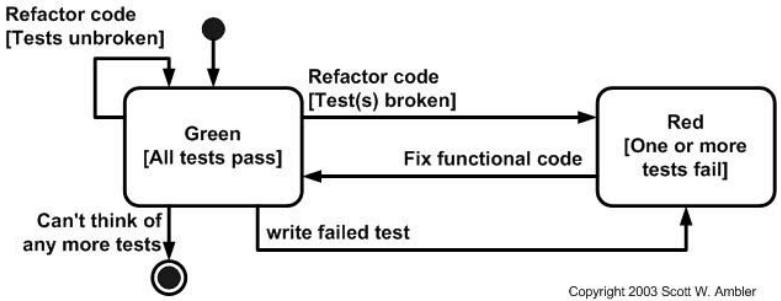
Enviar solución a: federico.balaguer@lifia.info.unlp.edu.ar

Algunas reglas de TDD

- Diseñar incrementalmente:
  - teniendo código que funciona como feedback para ayudar en las decisiones entre iteraciones.
- Los programadores escriben sus propios tests:
  - no es efectivo tener que esperar a otro que los escriba por ellos.
- El diseño debe consistir de componentes altamente cohesivos y desacoplados entre si:
  - mejora evolución y mantenimiento del sistema.

Automatización de TDD

- TDD asume la presencia de herramientas de testing (como las de la familia xUnit).
- Sin herramientas que automaticen el testing, TDD es prácticamente imposible.
- El ambiente de desarrollo debe proveer respuesta rápida ante cada cambio (build en 10 minutos).



Problemas y respuestas

- Unit testing infinito: por cada método público
- Test coupling: al estar los tests atados a la implementación

- “Test with a purpose” (Kent Beck)
- Saber por qué se testea algo y a qué nivel debe testearse.
- El objetivo de testear es encontrar bugs
- Testear tanto como sea el **riesgo** del artefacto

TDD

- ✔ Menor chance de Sobrediseño
- ✔ Proceso de elicitación del domino
- ✔ Arquitectura surge incrementalmente
- ✔ Refactoring mantiene código mínimo y limpio
- ✔ Requerimientos se convierten en test cases
- ✔ Último “Release” como entregable
- ✔ Adaptabilidad al cambio

- ⚠️ Refactoring mantiene código mínimo y limpio
- ⚠️ Mantener objetivo a largo plazo
- ⚠️ Cambios a DB puede ser costosos
- ⚠️ Cambios a API's son dificiles de realizar sin romper el código de los clientes
- ⚠️⚠️ Interacción expertos del dominio

Bibliografia

- “Test Driven Development: by Example”. Kent Beck. Addison Wesley. 2002
- Introduction to Test Driven Development. Scott Ambler. http://agiledata.org/essays/tdd.html
- FitNesse: http://fitnesse.org/
  framework para acceptance testing a través de una wiki

Preguntas frecuentes

TDD propone:

- ❏ Desarrollar funcionalidad teniendo en cuenta casos excepcionales
- ❏ Codificar los casos de prueba que cubren la funcionalidad a desarrollar
- ❏ Especificar posibles casos de pruebas
- ❏ Ejecutar los "refactorings" necesarios para implementar nueva funcionalidad

¿Cuál de estas opciones son ventajas de escribir casos de prueba como lo propone TDD?

- ❏ Definir qué se espera de cada requerimiento antes de escribir código
- ❏ Aumenta él numero de Patrones de Diseño aplicados
- ❏ Esfuerzo en modificar a la base de datos puede ser grande
- ❏ Por cada requerimiento se conoce cuando se termina de programar

¿Cuál de estas opciones son desventajas o riesgos de TDD?

- ❏ Se depende de expertos del dominio o buenas especificaciones de los requerimientos/historias
- ❏ Escribir casos de prueba confunde a los desarrolladores
- ❏ Base de Datos y API's pueden romper el código de otros programas o clientes