

# Aseguramiento de Calidad del Software (SQA)

¿Qué entendemos por calidad del software?

- ▶ Conformancia total a requerimientos
  - ▶ errores en requerimientos
  - ▶ requerimientos poco claros
- ▶ Observar "factores de calidad" (augmentability, compatibility, expandability, flexibility, interoperability, maintainability, operability, portability, reliability, scalability, usability, etc)
  - ▶ poco que ver con calidad percibida por el usuario

# Calidad (Capers Jones)

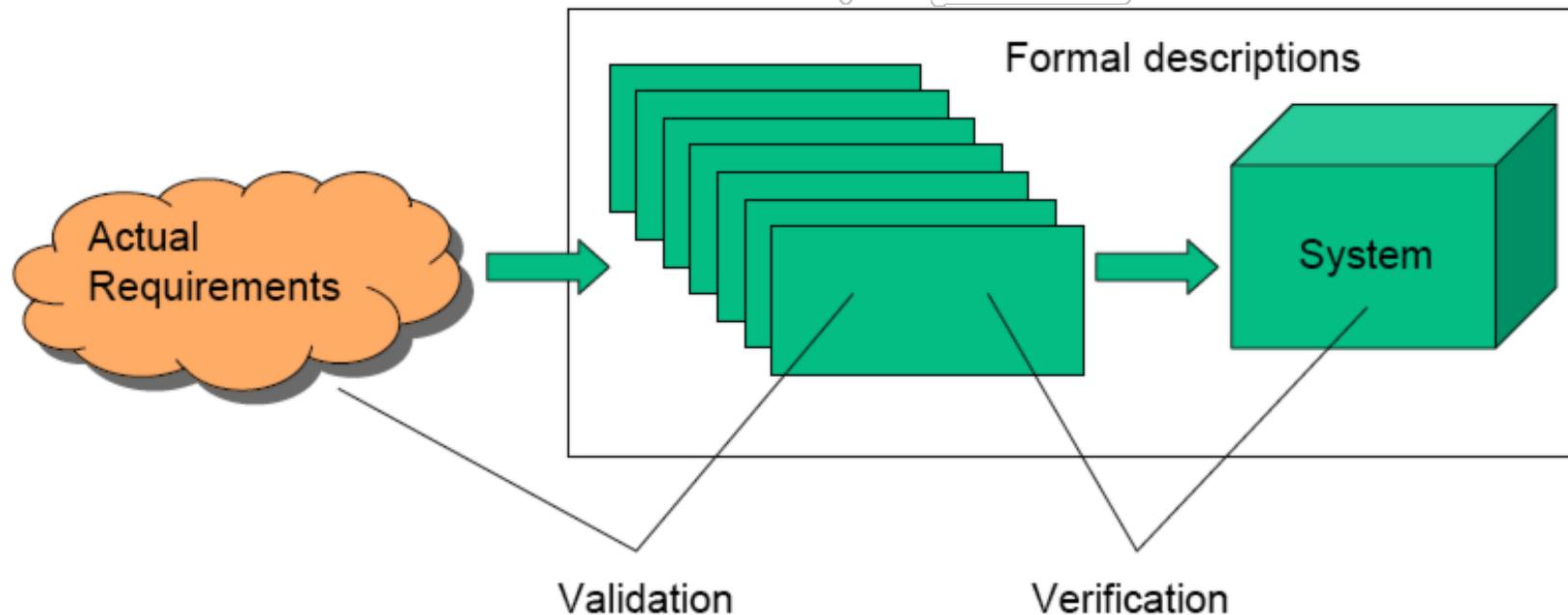
- ▶ Ausencia de defectos que causarían que la aplicación deje de funcionar o entregue resultados erróneos
- ▶ Nivel de severidad del defecto
  - ▶ severidad 1 - la aplicación no funciona
  - ▶ severidad 2 - mayoría de funciones no habilitadas o resultados erróneos
  - ▶ severidad 3 - problemas menores
  - ▶ severidad 4 - problemas cosméticos que no afectan funcionalidad

# Como identificar calidad

## (Capers Jones)

- ▶ Quality implies **low levels of defects** when software is deployed, ideally approaching zero defects.
- ▶ Quality implies **high reliability**, or being able to run without stop or strange and unexpected results or sluggish performance.
- ▶ Quality implies **high levels of user satisfaction** when users are surveyed about software applications and its features.
- ▶ Quality implies a feature set that meets **the normal operational needs of a majority of customers or users**.
- ▶ Quality implies **a code structure and comment density that minimize bad fixes or accidentally inserting new bugs when attempting to repair old bugs**. This same structure will facilitate adding new features.
- ▶ Quality implies **effective customer support when problems do occur**, with minimal difficulty for customers in contacting the support team and getting assistance.
- ▶ Quality implies **rapid repairs of known defects**, and especially so for high-severity defects.
- ▶ Quality should be supported by **meaningful guarantees and warranties** offered by software developers to software users.
- ▶ Effective definitions of quality should lead to quality improvements. This means that **quality needs to be defined rigorously enough so that both improvements and degradations can be identified**, and also averages. If a definition for quality cannot show changes or improvements, then it is of very limited value.

# Verificación vs Validación



- ▶ Verificación - el sistema hace correctamente lo que se especificó
- ▶ Validación - el sistema hace lo que realmente se necesitaba que hiciera

## | **Verification**

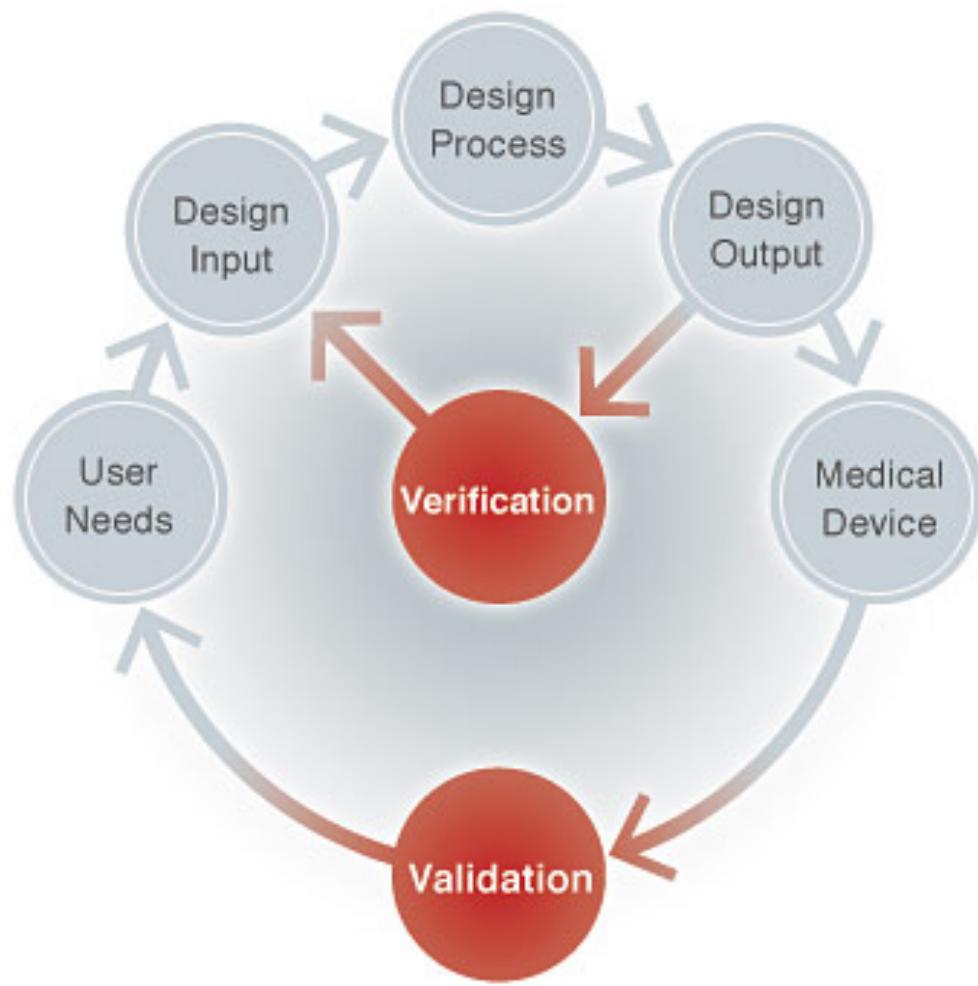
"Are we building the product right"

- | The software should conform to its specification

## | **Validation**

"Are we building the right product"

- | The software should do what the user really requires



# Hacia Ausencia de Defectos

- ▶ no incorporarlos al construir el software (muy difícil en la práctica)
- ▶ análisis estático
  - ▶ se examina el código con herramientas de software en busca de problemas o cosas sospechosas (código duplicado, peligros de seguridad, manejo de errores, etc)
- ▶ inspección formal de código
  - ▶ código es examinado por una o mas personas que no son quienes lo escribieron
- ▶ testing
  - ▶ se diseñan casos de prueba (input y output conocidos) y se somete el software a ellas

# Análisis Estático

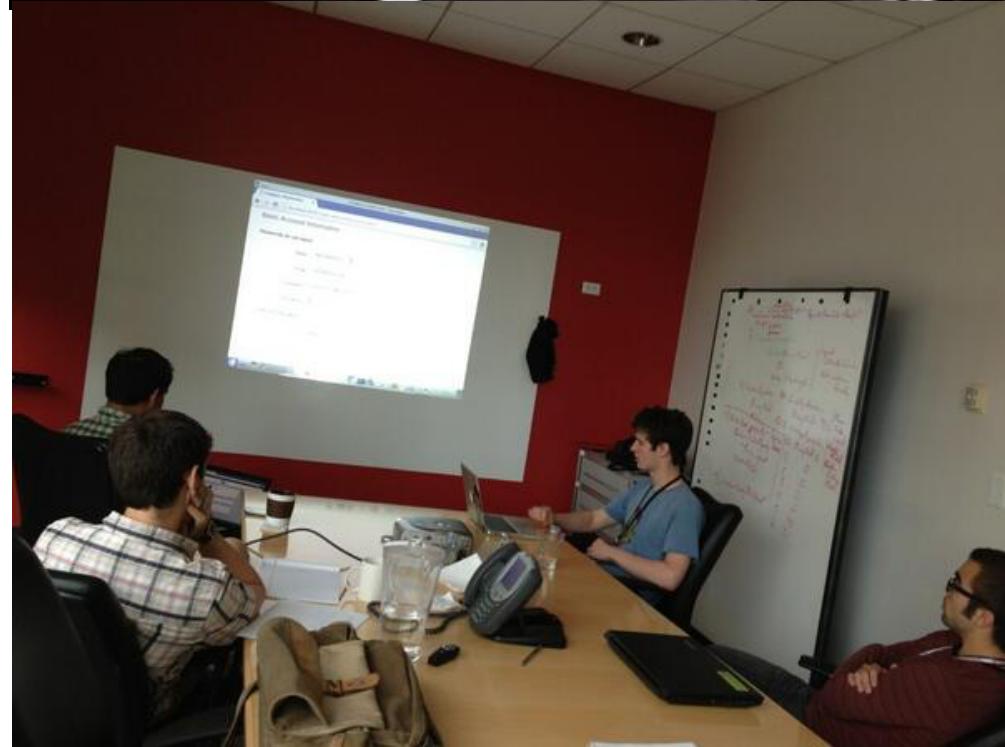
- ▶ Análisis del código (sin correrlo) mediante herramientas que permiten detectar elementos sospechosos
  - ▶ más usado con lenguajes como C/C++, Java que con lenguajes dinámicos (Ruby, Python)
  - ▶ encuentra algunos problemas fácilmente: buffer overflow, SQL injection, etc
  - ▶ muchos falsos positivos
  - ▶ muchos problemas importantes no son detectados

# Ejemplo: cppcheck (C/C++)

- ▶ Out of bounds checking
- ▶ Memory leaks checking
- ▶ Detect possible null pointer dereferences
- ▶ Check for uninitialized variables
- ▶ Check for invalid usage of STL
- ▶ Checking exception safety
- ▶ Warn if obsolete or unsafe functions are used
- ▶ Warn about unused or redundant code
- ▶ Detect various suspicious code indicating bugs

# Inspección Formal de Código

- ▶ Forma demostradamente efectiva para producir código de calidad
- ▶ Código es revisado por equipo de pares con el objetivo de detectar la mayor cantidad de defectos
- ▶ Equipo revisor se prepara antes de sesión de revisión
- ▶ Pueden usarse checklists para búsqueda mas dirigida



# Ojo: evitar ésto ...

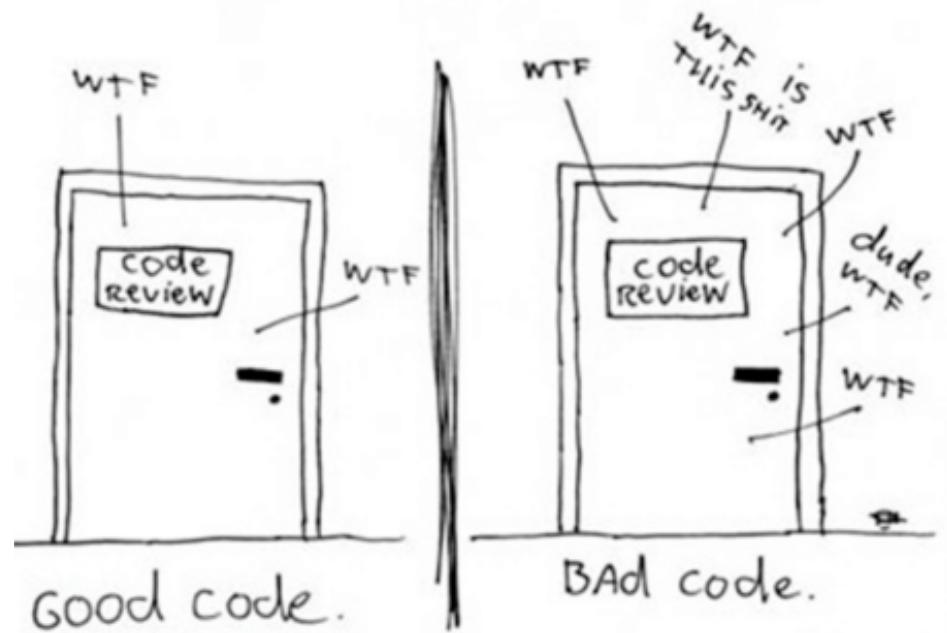
## The Peer Code Review



# Sesión de Inspección

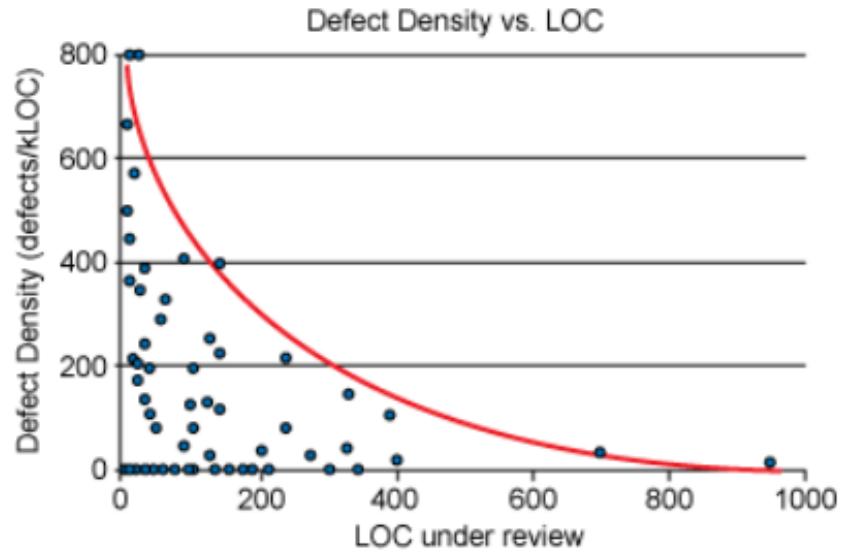
- ▶ un facilitador dirige al grupo sobre el código
- ▶ al llegar a un bloque en que hay observaciones los revisores plantean sus dudas para ver si se trata en verdad de un problema
- ▶ Si es un problema se registra (no se corrige)
- ▶ Aprovechar de recolectar datos (defectos por grado de severidad, por página, por número de líneas, etc)

The ONLY VALID MEASUREMENT  
OF CODE QUALITY: WTFs/MINUTE



# Recomendaciones

- ▶ métricas
  - ▶ densidad de defectos - número promedio de errores por línea de código
  - ▶ tasa de defectos - errores por hora de revisión
  - ▶ tasa de inspección - líneas revisadas en una hora
- ▶ Revisar máximo 400 líneas de código por sesión
- ▶ No mas de 1 hora por sesión
- ▶ Lento - 400 líneas deberían tomar 1 hora o más
- ▶ Anotar código antes de la revisión
- ▶ Cultivar cultura positiva de revisión



# Testing

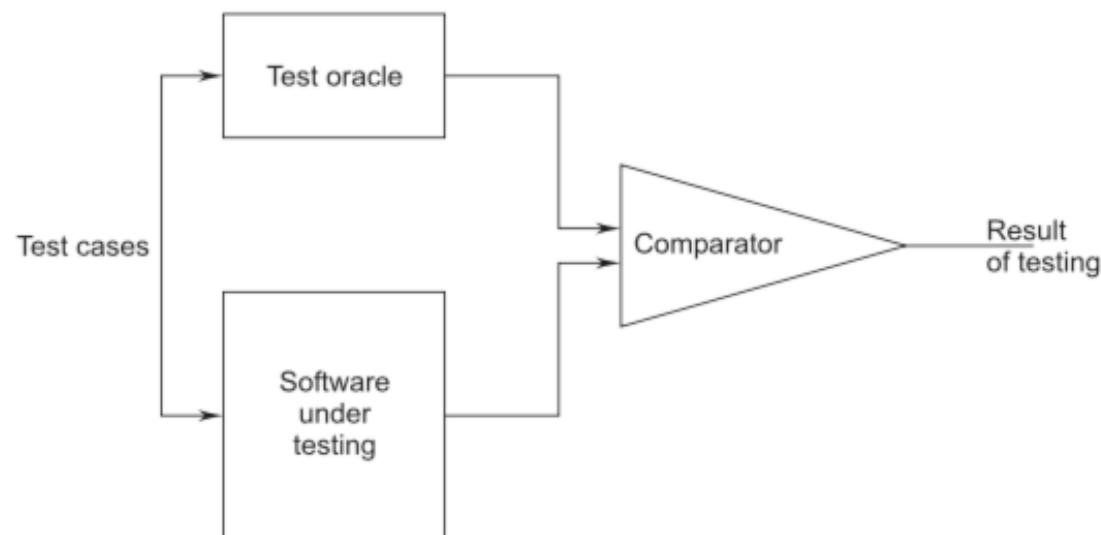
- ▶ Proceso de ejecutar un programa con el objetivo de encontrar un error
- ▶ un buen caso de prueba es uno con una alta probabilidad de encontrar un error oculto
- ▶ un test exitoso es aquel que descubre un error que no se conocía

# Dificultad en Testing

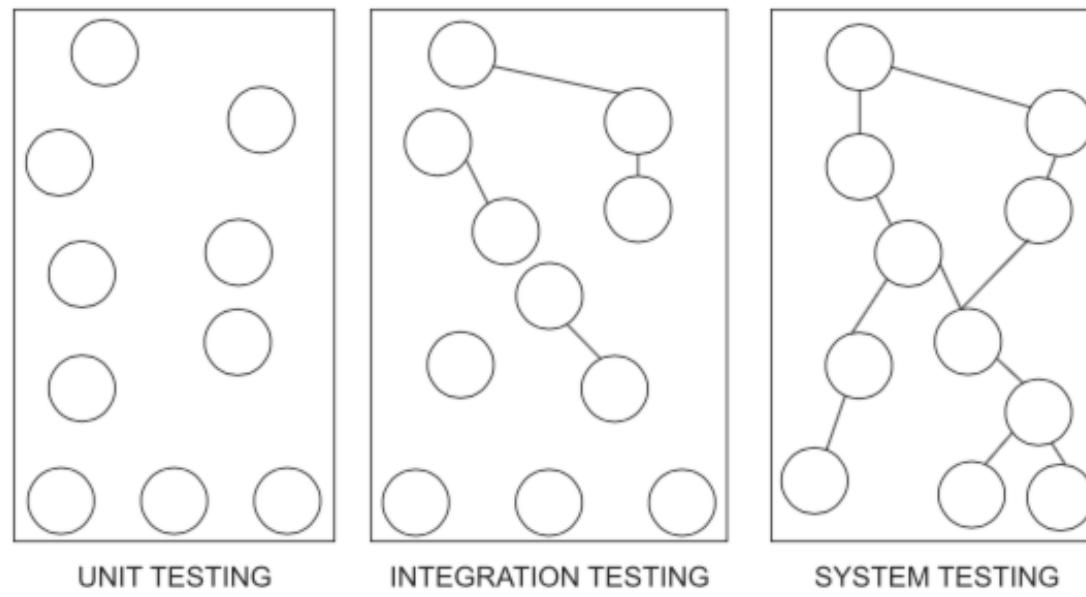


# Oráculos

- ▶ Mecanismo diferente al programa mismo que permite chequear la correctitud del caso de test
- ▶ Por lo general basado en personas

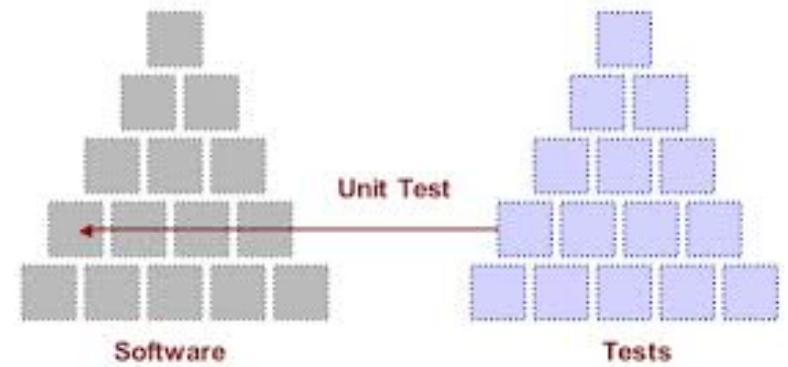


# Niveles de Tests



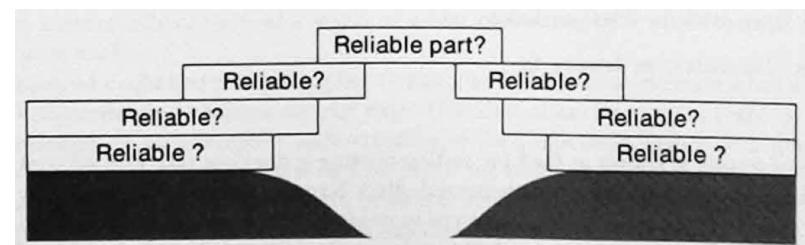
# Tests Unitarios

- ▶ Realizado por los desarrolladores en paralelo con la implementación o inmediatamente después de terminar la parte a probar
- ▶ Pueden usarse tests de black box o white box
- ▶ A veces hay que crear inputs y outputs (unidades fuera de análisis)
- ▶ La confiabilidad de todo el software recae finalmente en los tests unitarios



I'm a programmer and #iwritebugs. All the time. Embarrassingly obvious bugs. Brain dead stupid. Fortunately, I also write tests.

Howard Lewis Ship



# Tests de Blackbox y Whitebox



- ▶ blackbox test - no se conocen detalles del software, solo se considera input y output
- ▶ whitebox test - se conocen los detalles del software y se puede utilizar en diseño del test

```
using namespace std;
int main()
{
    const double THIRDPi = 3.14
    float radius;
    float height;
    double volume;

    cout << "Enter Radius" << e
    cin >> radius;
    cout << "Enter height" << e
    cin >> height;
```

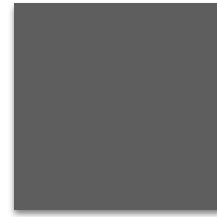
# Estrategias según tipo de test

```
using namespace std;
int main()
{
    const double THIRDPI = 3.14;
    float radius;
    float height;
    double volume;

    cout << "Enter Radius" << endl;
    cin >> radius;
    cout << "Enter height" << endl;
    cin >> height;
```

## ▶ White Box

- ▶ cobertura de instrucciones
- ▶ cobertura de caminos
- ▶ cobertura de decisiones

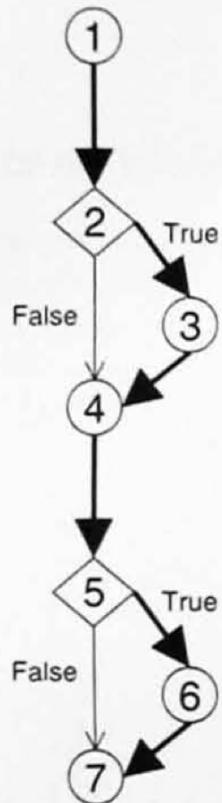


## ▶ Black Box

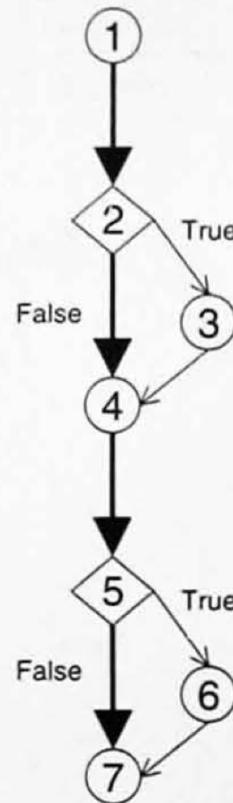
- ▶ clases de equivalencia
- ▶ análisis de valores frontera

# Cobertura de Caminos (white box)

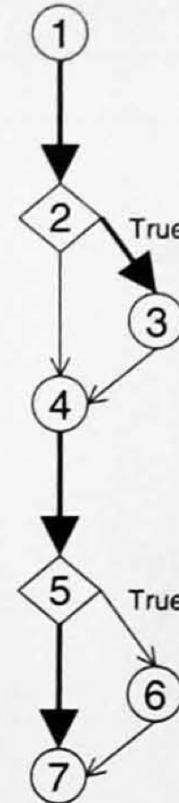
Path #1



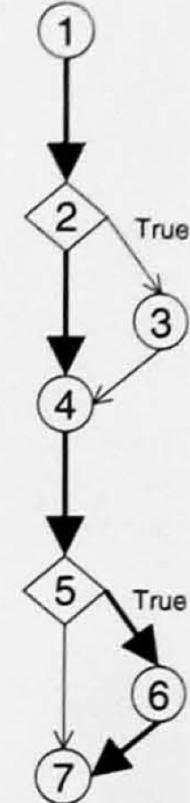
Path #2



Path #3

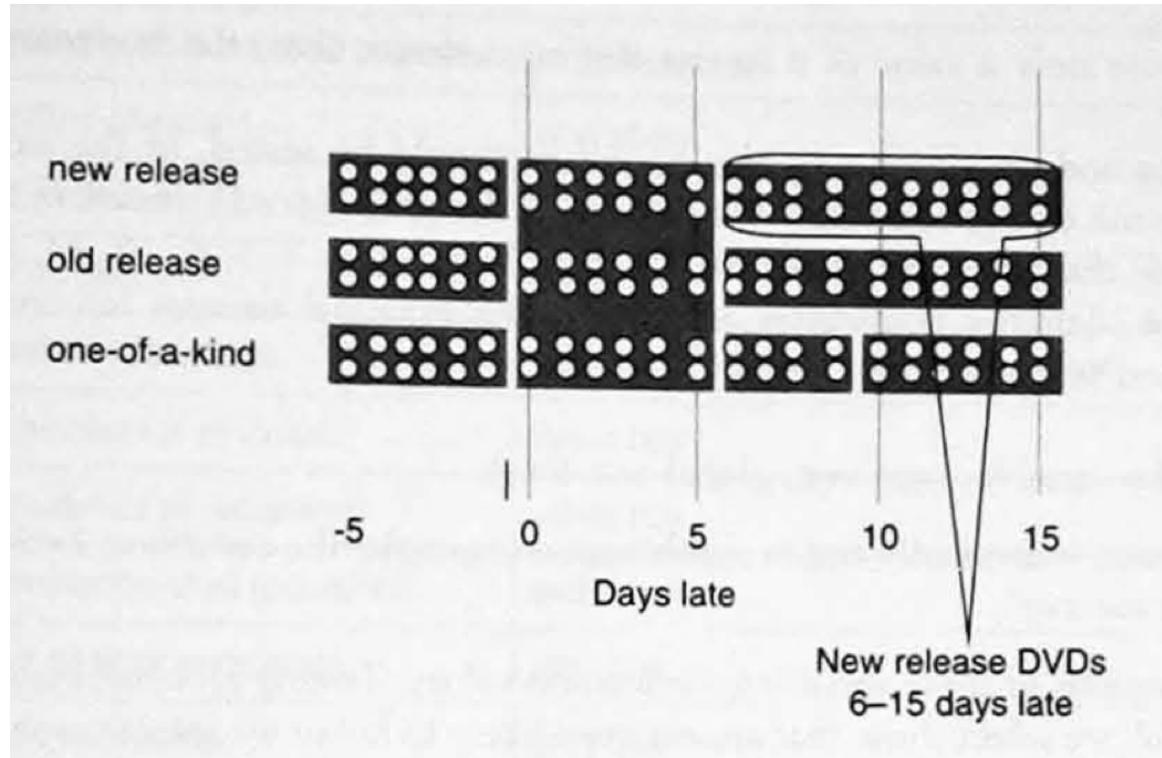


Path #4



# Clases de Equivalencia (black box)

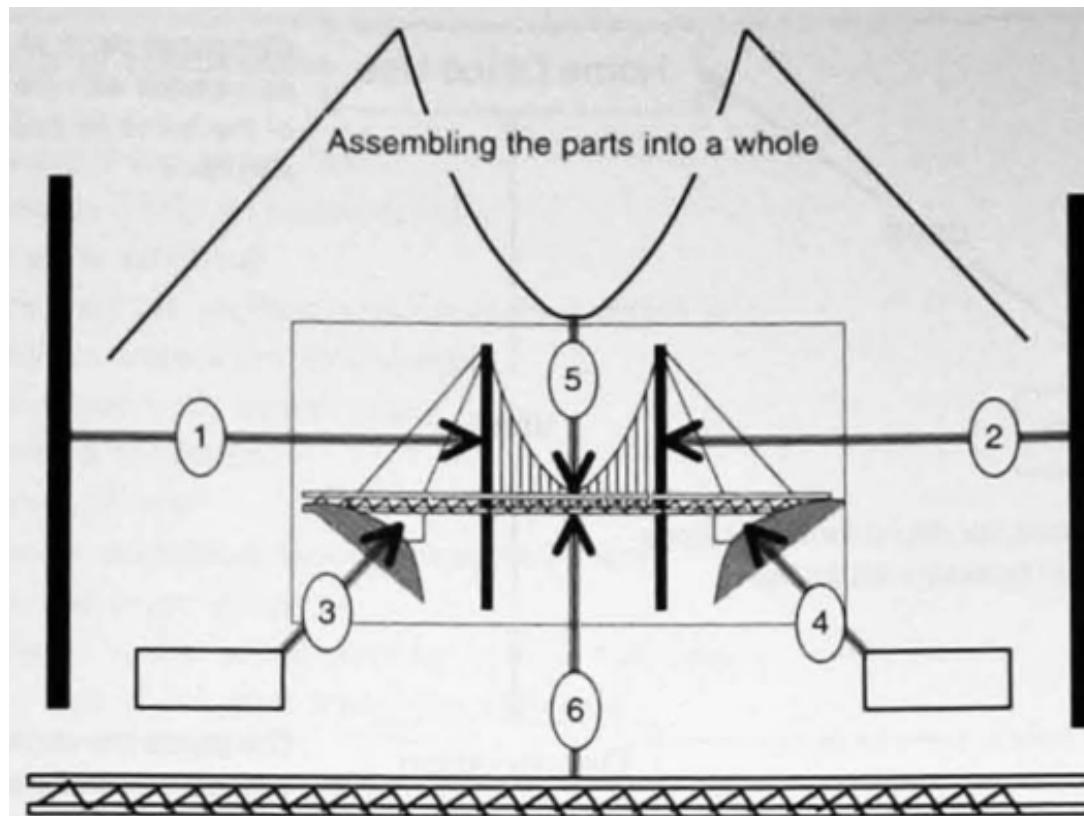
Ejemplo de Clases para probar clase que calcula las multas en arriendo de películas



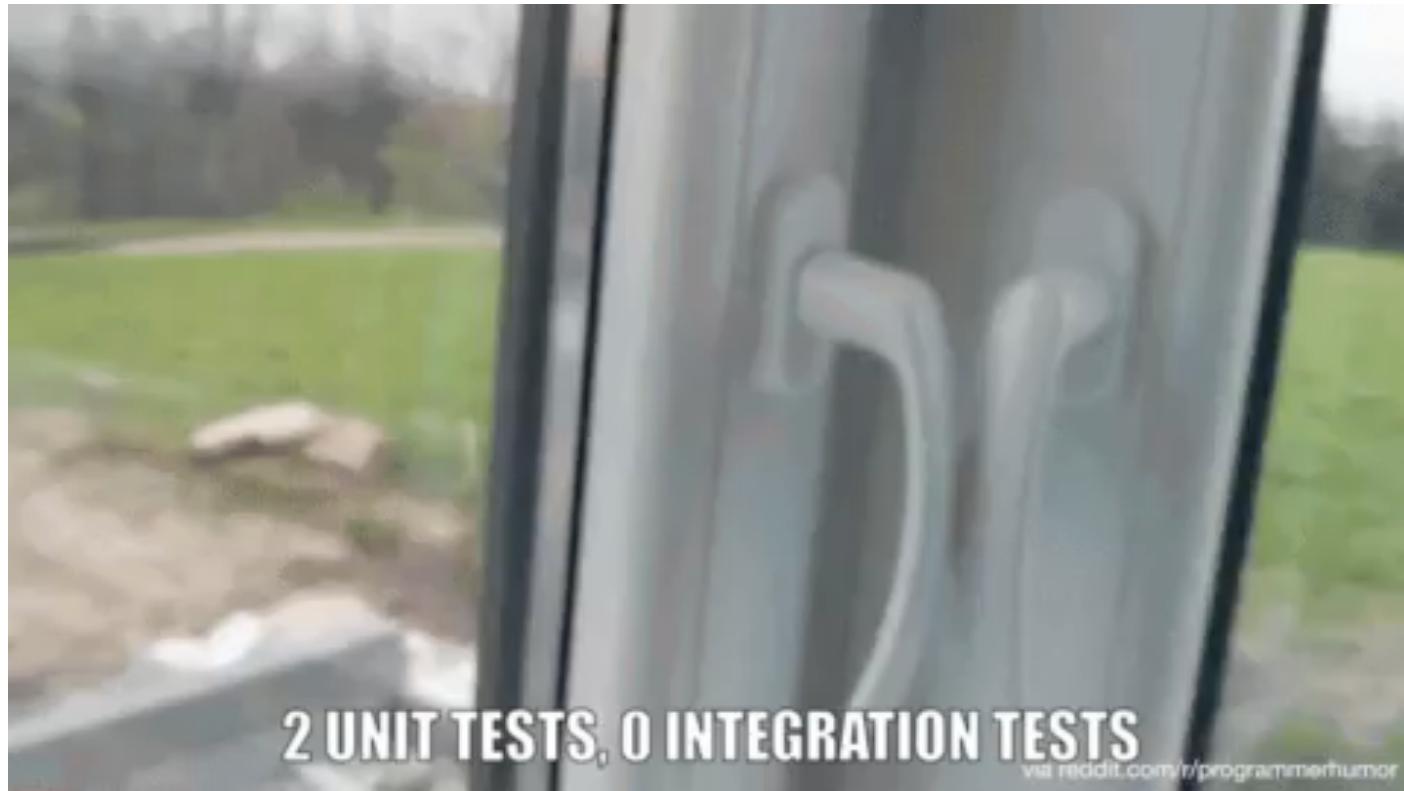


# Tests de Integración

- ▶ Cada una de las partes puede haber pasado las pruebas pero aún así el puente puede fallar



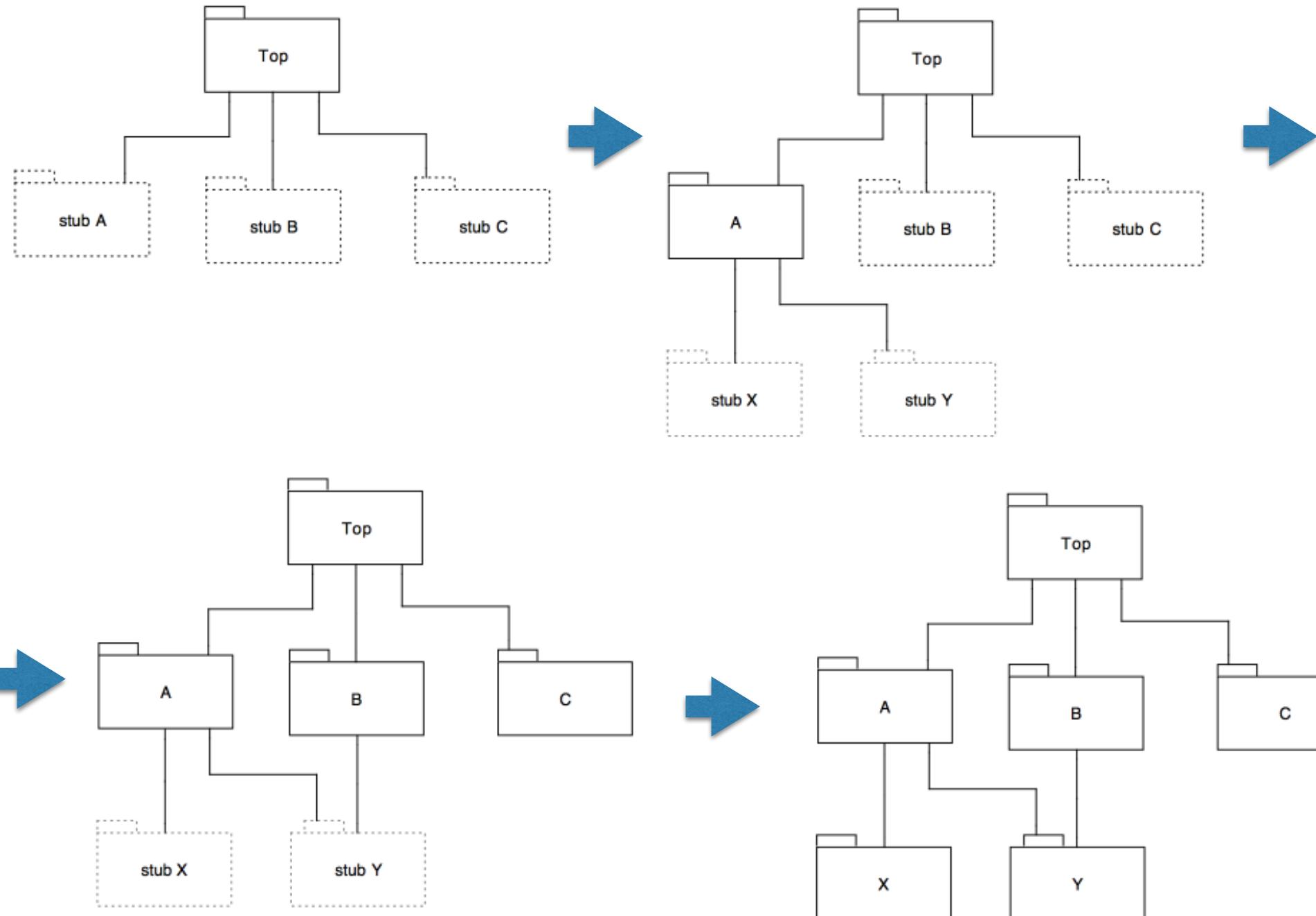
# La necesidad



# Unit test vs. Integration test

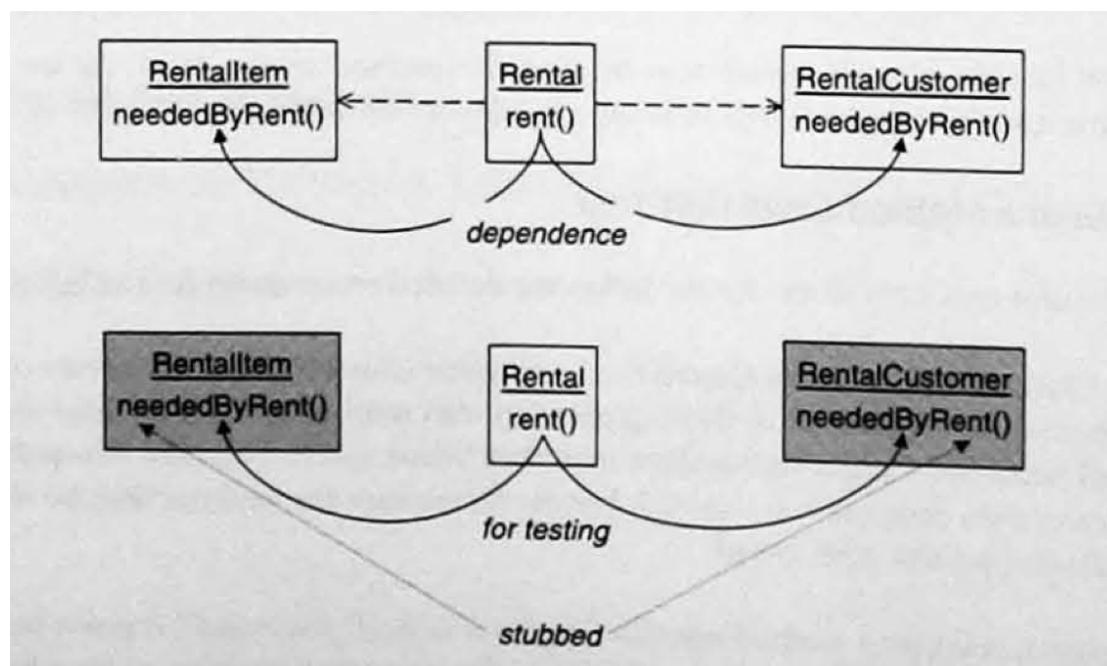


# Estrategia Top Down

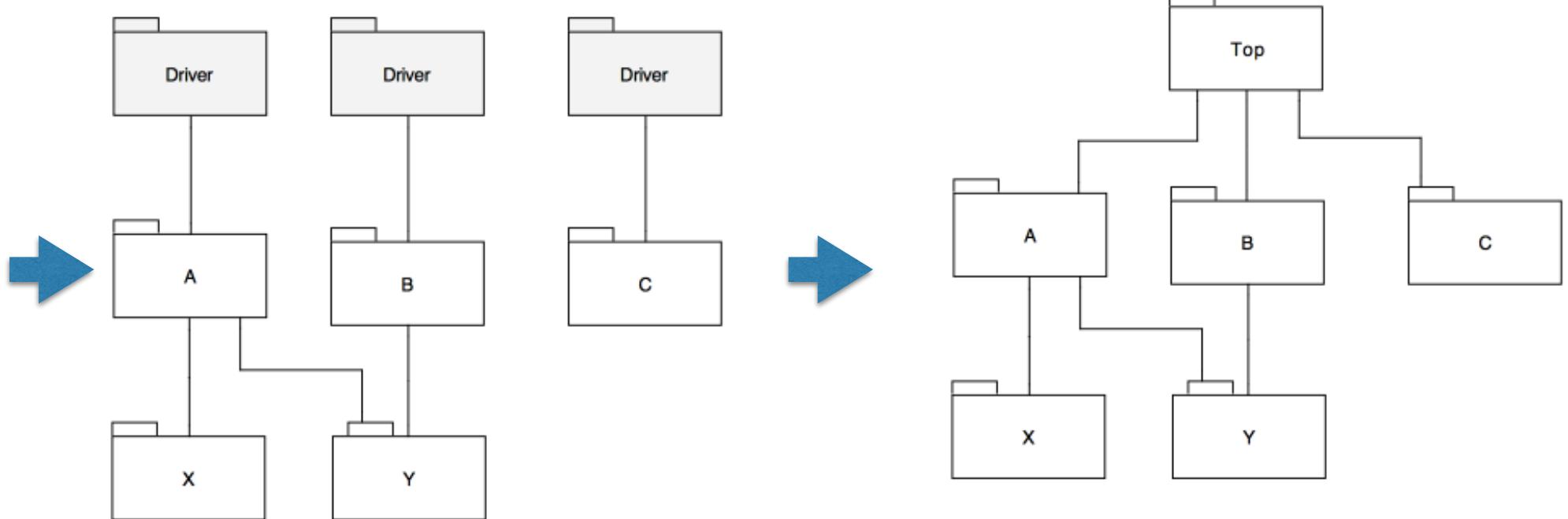
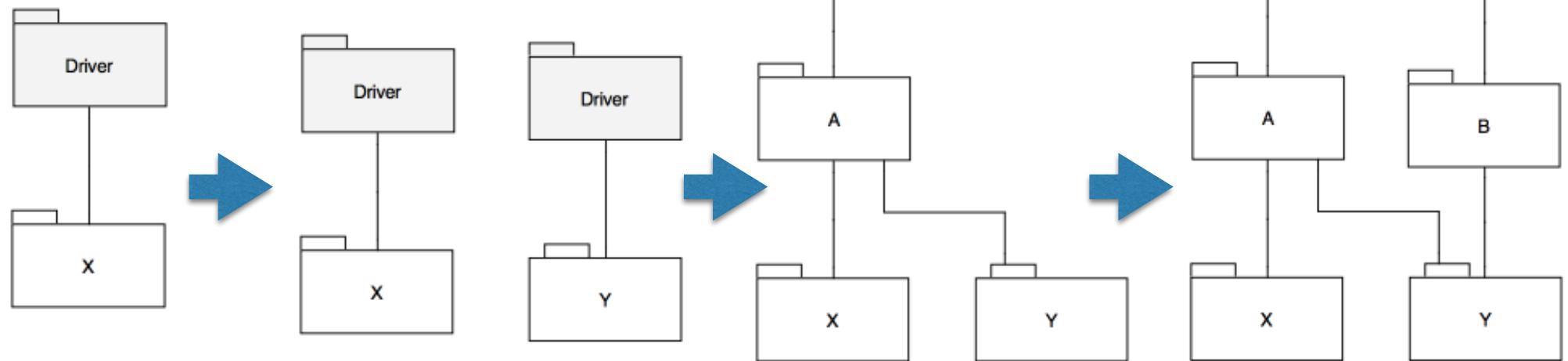


# Uso de Stubs

- ▶ Puede ser necesario escribir clases/métodos que no existen para probar el que nos interesa
- ▶ En ese caso se escribe lo mínimo necesario (stub)

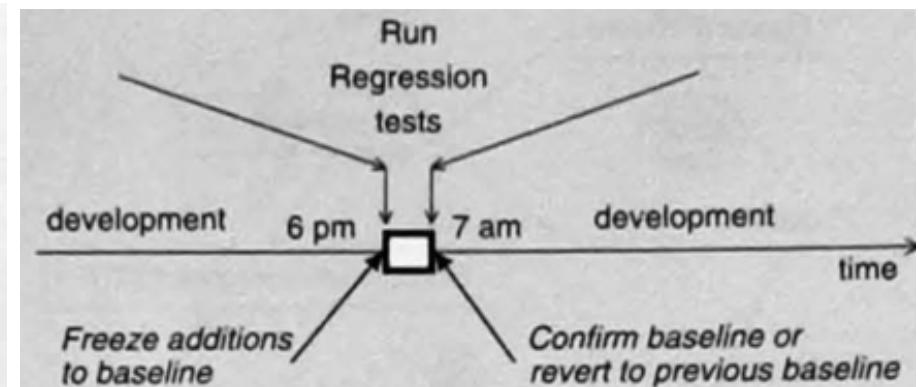
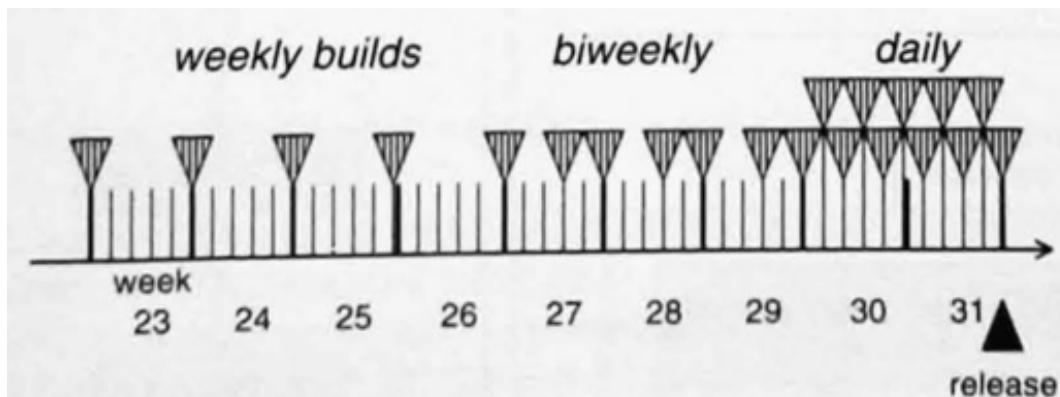


# Bottom up



# Integración Contínua

- ▶ Tipo de integración incremental en que se agregan piezas en forma muy frecuente (diario, semanal)
- ▶ El incremento ha sido probado en forma individual, se integra y si la línea de base no se daña queda



# Test de Regresión

- ▶ combinación de test unitarios y de integración asociado a la entrega continua
- ▶ busca asegurar que los cambios no introduzcan nuevos errores en tests unitarios
- ▶ una suite de tests que puede incluir
  - ▶ tests de nuevas funcionalidades
  - ▶ una muestra representativa de tests que ejerciten todas las funciones
  - ▶ tests que se centren en las componentes modificadas

# Test de Sistema y Test de Aceptación

- ▶ Sistema
  - ▶ Verificación (no validación) del sistema completo
  - ▶ Contra requisitos funcionales y no funcionales
  - ▶ Dos tipos
    - ▶ Alpha test - lo hace cliente en el sitio de desarrollo bajo dirección del jefe del proyecto
    - ▶ Beta test - lo hace un número de usuarios en su propio ambiente (registran cualquier problema)
- ▶ Aceptación - validación, efectuada por el cliente con el objeto de aceptar o rechazar el producto (puede durar semanas o meses)

# Tests de Sistema, de aceptación y de regresión

	System	Acceptance	Regression
Test for ...	Correctness, completion	Usefulness, satisfaction	Accidental changes
Test by ...	Development test group	Test group with users	Development test group
	Verification	<i>Validation</i>	Verification

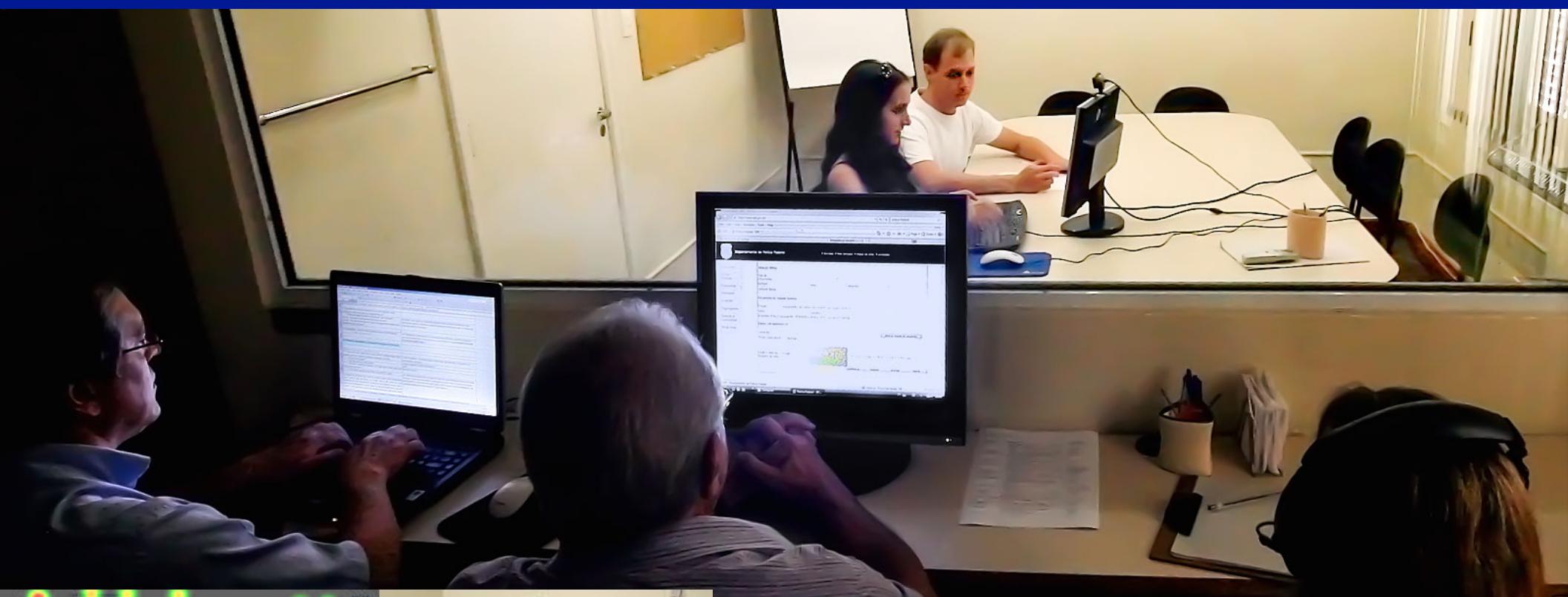
# Tests No Funcionales

- ▶ Performance - se prueba que el sistema satisfaga los requerimientos de desempeño
- ▶ stress (carga) - se prueba que tan bien el sistema responde a una carga mayor de la planeada
- ▶ confiabilidad - se mide el tiempo medio entre fallas (MTBF)
- ▶ usabilidad - se mide tiempo promedio en completar tareas, tasa de errores del usuario, etc
- ▶ seguridad - hackers

# Tests de Usabilidad



- ▶ observación de personas reales usando el software (5 a 8)
  - ▶ pueden usarse cámaras, trackers de mirada, etc
  - ▶ se registra todo
- ▶ se hacen mediciones sobre
  - ▶ tiempo necesario para llevar a cabo ciertas tareas específicas
  - ▶ errores en proceso (repeticiones, correcciones)
- ▶ opinión de los testers es menos importante (subjetivo)



# A/B Testing (no funcional)

- ▶ Ampliamente usado con sitios Web
- ▶ Se ponen a prueba dos versiones de la página y se ve cual funciona mejor



# A (image)



# B (video)



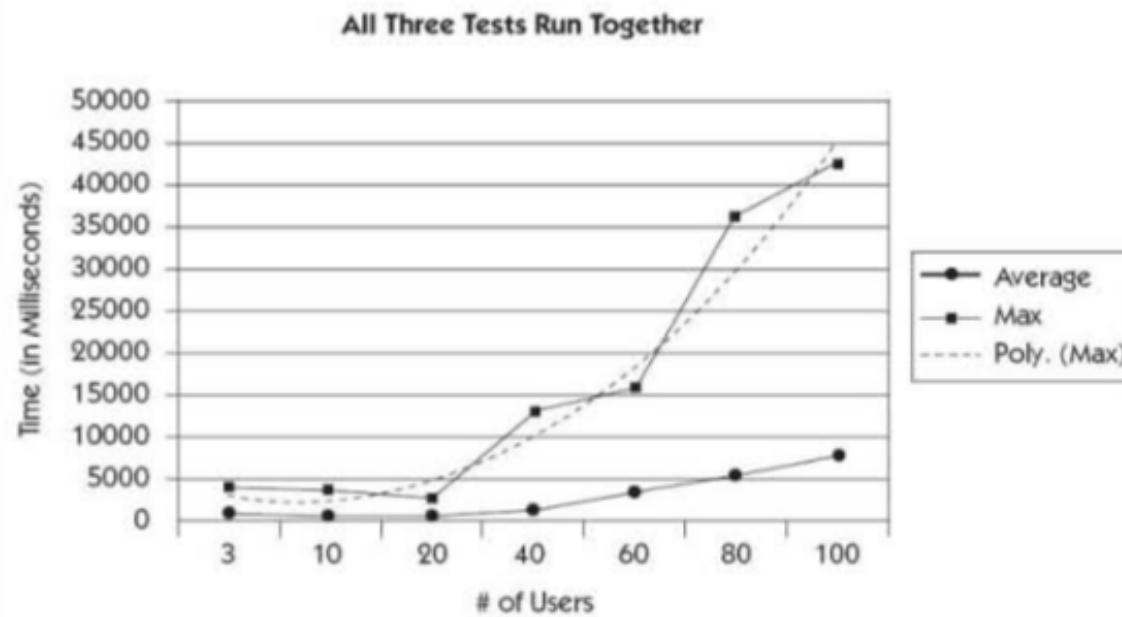
**A Wins!**

40.6% increase

= additional *2.8 million* e-mail addresses  
& *\$60million* in estimated donations

# Desempeño, carga, stress, escalabilidad

- tests de desempeño ayudan a detectar cuellos de botella
- hay muchas herramientas de software que ayudan a esta tarea (open source y comerciales)

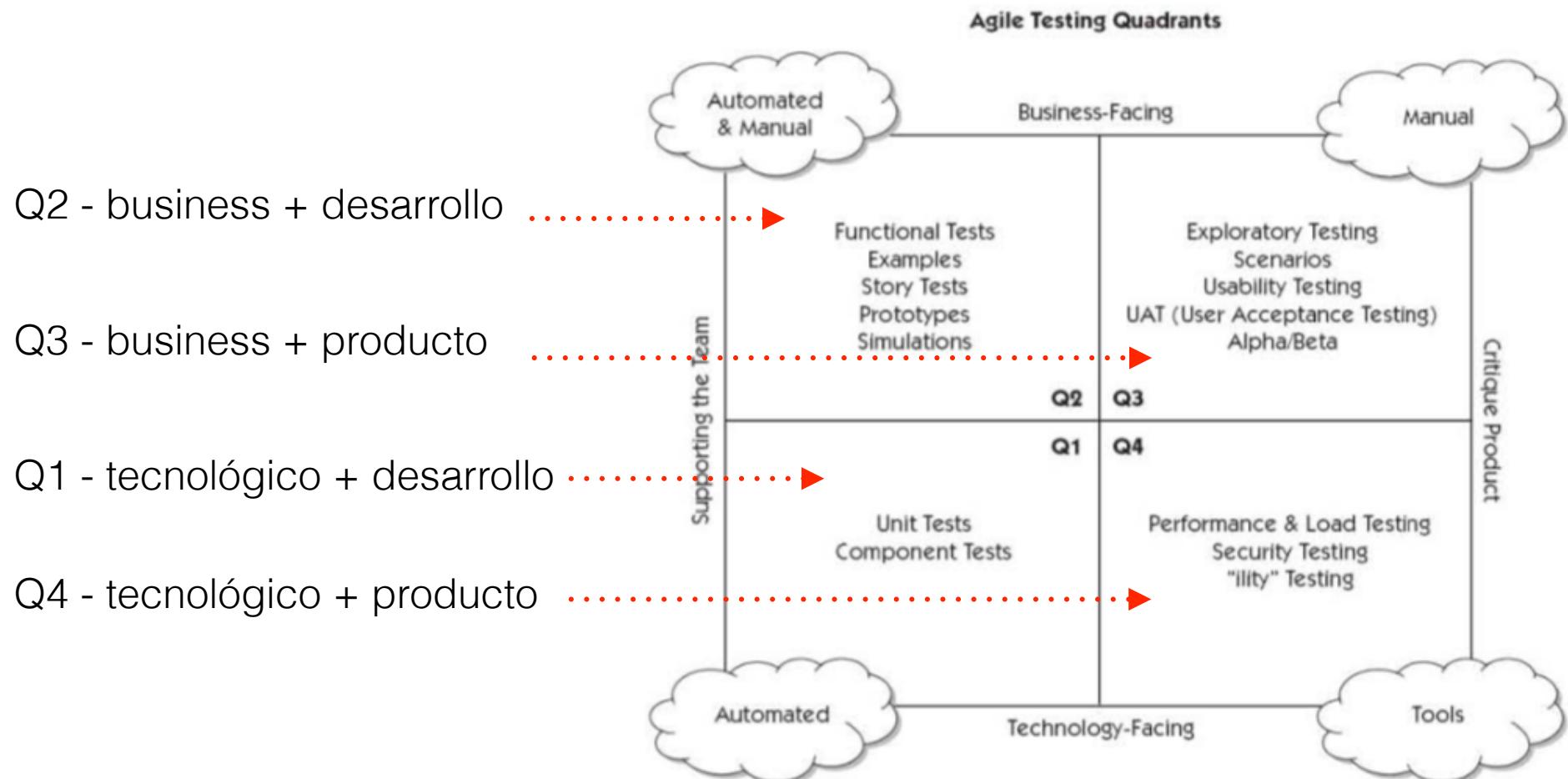


**Figure 11-2** Max and average transaction times at different user loads.

# Testing: the big picture

Eje X - apoyo al desarrollo vs crítica al producto

Eje Y - tests tecnológicos vs tests orientados al negocio



Q2 - business + desarrollo

Q3 - business + producto

Q1 - tecnológico + desarrollo

Q4 - tecnológico + producto

# Testing como apoyo al desarrollo

- ▶ Facilita la comprensión a fondo de la funcionalidad deseada
- ▶ Facilita el construir rápidamente una solución e ir mejorándola en forma posterior (refactoring)
- ▶ Ciclo red/green/refactor
- ▶ La idea llevada al extremo: TDD (test driven development)
  - ▶ escribir los tests asociados a un trozo de código ANTES de escribir el código
  - ▶ probar el código (que no existe) - fails
  - ▶ escribir código para que pueda superar el test
  - ▶ refactor
  - ▶ escribir el siguiente test

# Tests Automatizados

- ▶ dependiendo de la plataforma hay herramientas que facilitan el desarrollo de los tests
- ▶ muchas reproducen el modelo de JUnit (Java)
- ▶ La plataforma Ruby desde sus inicios ha incluido herramientas para hacer test unitarios
  - ▶ test::unit
  - ▶ minitest (reemplaza al anterior a partir de 1.9.3, default)
  - ▶ gema rspec (recomendable)