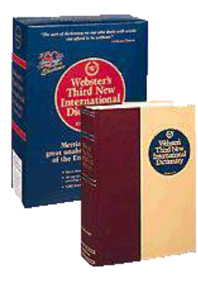
escuela técnica superior
de ingeniería informática
Pruebas de software
Departamento de
Lenguajes y Sistemas Informáticos
UNIVERSIDAD DE SEVILLA

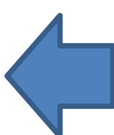
Ejemplo de otro dominio

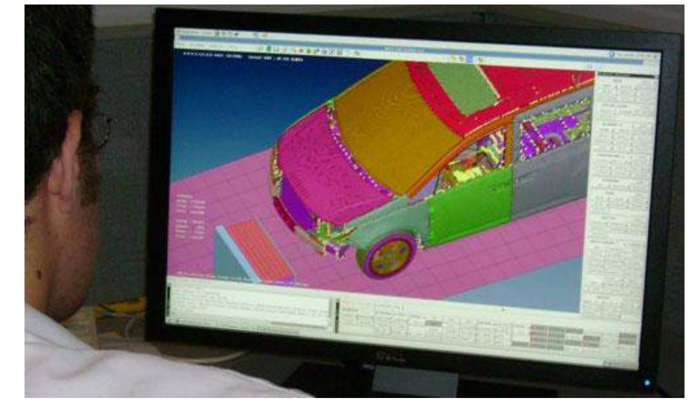
Probar: 1. tr. Hacer examen y experimento de las cualidades de alguien o algo.

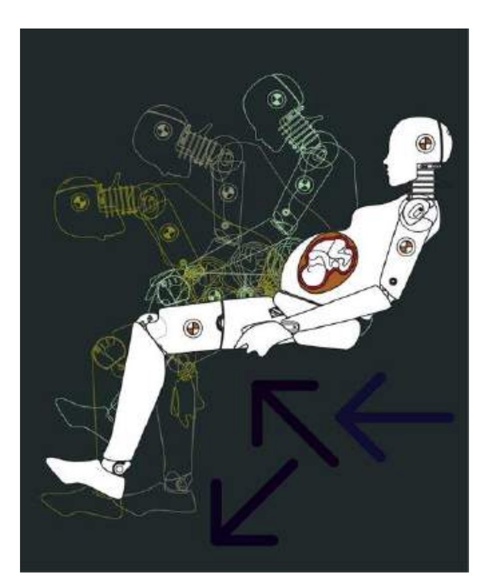


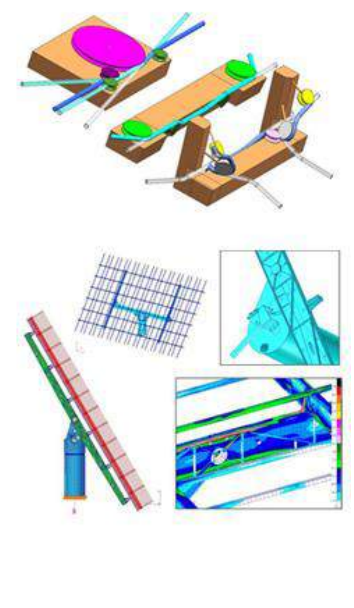
Test: a critical examination, observation, or evaluation

Ejemplo de otro dominio









Ejecución de la prueba

Diseño de la prueba

Índice

- Introducción
- ¿Cuándo hacer pruebas?
- Definiciones
- Proceso general
- Actividades
- Diseño de casos de prueba
- Técnicas de diseño de casos de prueba
- Resumen
- Bibliografía

Aplicación de ejemplo

Acciones que debemos permitir:

- Identificarse
- Consultar votaciones
- Añadir elementos a la papeleta
- Confirmar la votación

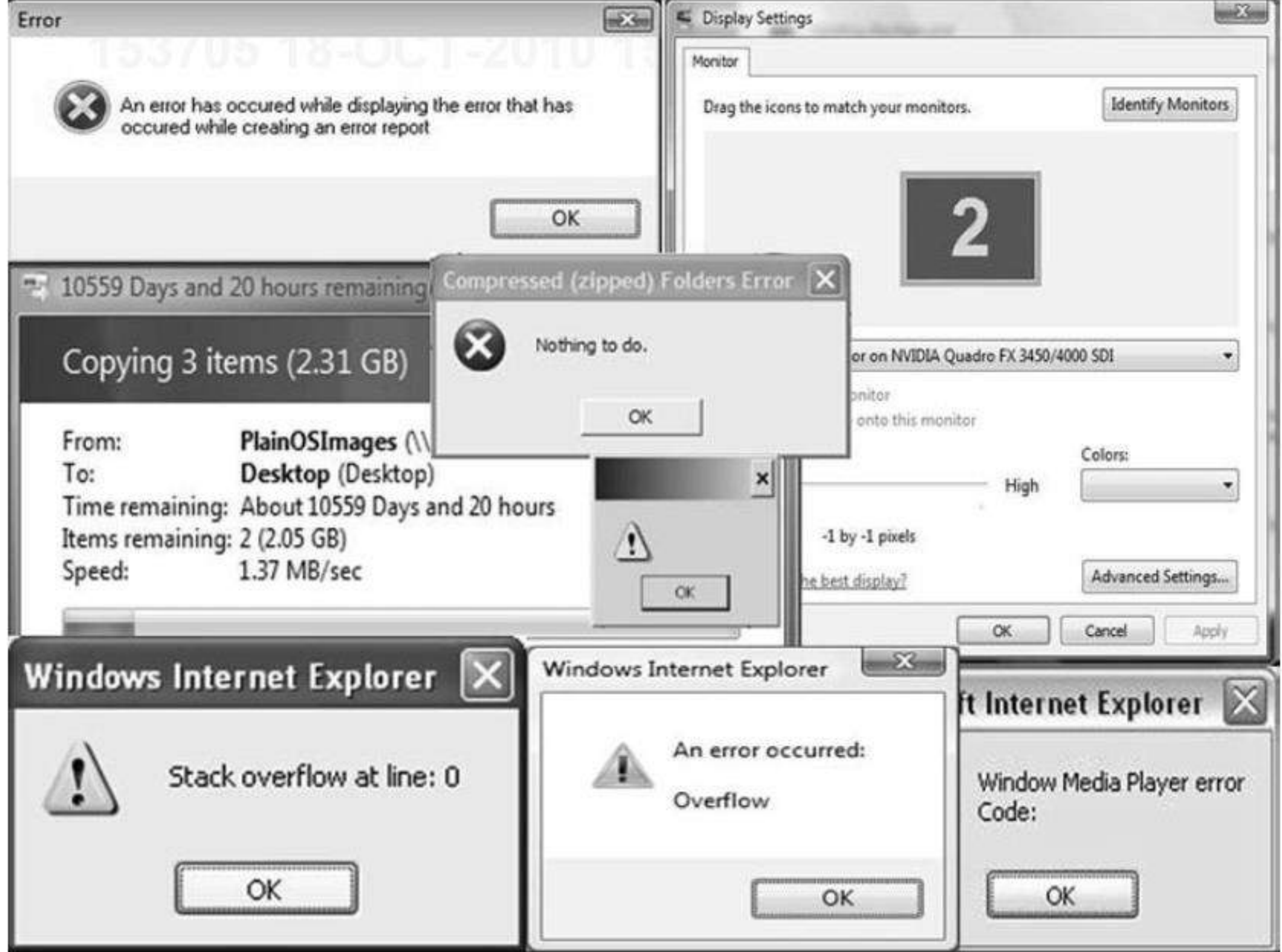
Ejemplos de posibles pruebas

- Probar que con la papeleta vacía no se puede hacer un voto
- Probar que si se introducen candidatos que no están en las papeletas, no se permite añadir la papeleta a la urna.
- Probar que la interfaz gráfica es conforme a una norma de accesibilidad
- Probar que no se puede acceder a partes de la aplicación que son sólo para usuarios autorizados
- Probar que los votos se almacenan, se actualizan, se borran y se consultan adecuadamente
- Probar que se pueden conectar 100 usuarios concurrentemente

¿Cuál es el problema?

¿Cómo probar que un sistema software está haciendo algo bien/mal? ¿Cómo diseñar las pruebas?

¿Por qué es un problema importante?



Ejemplos de fallos

"There are two ways to write error-free programs; only the third one works."

—Alan J. Perlis

Índice

Introducción

¿Cuándo hacer pruebas?

Definiciones

Proceso general

Actividades

Diseño de casos de prueba

Técnicas de diseño de casos de prueba

Resumen

Bibliografía

Más tarde, más caro

Large software projects

IBM-SSD

GTE

80%
Median (TRW survey)
20%

SAFEGUARD

Smaller software projects

– [Boehm, 1980]

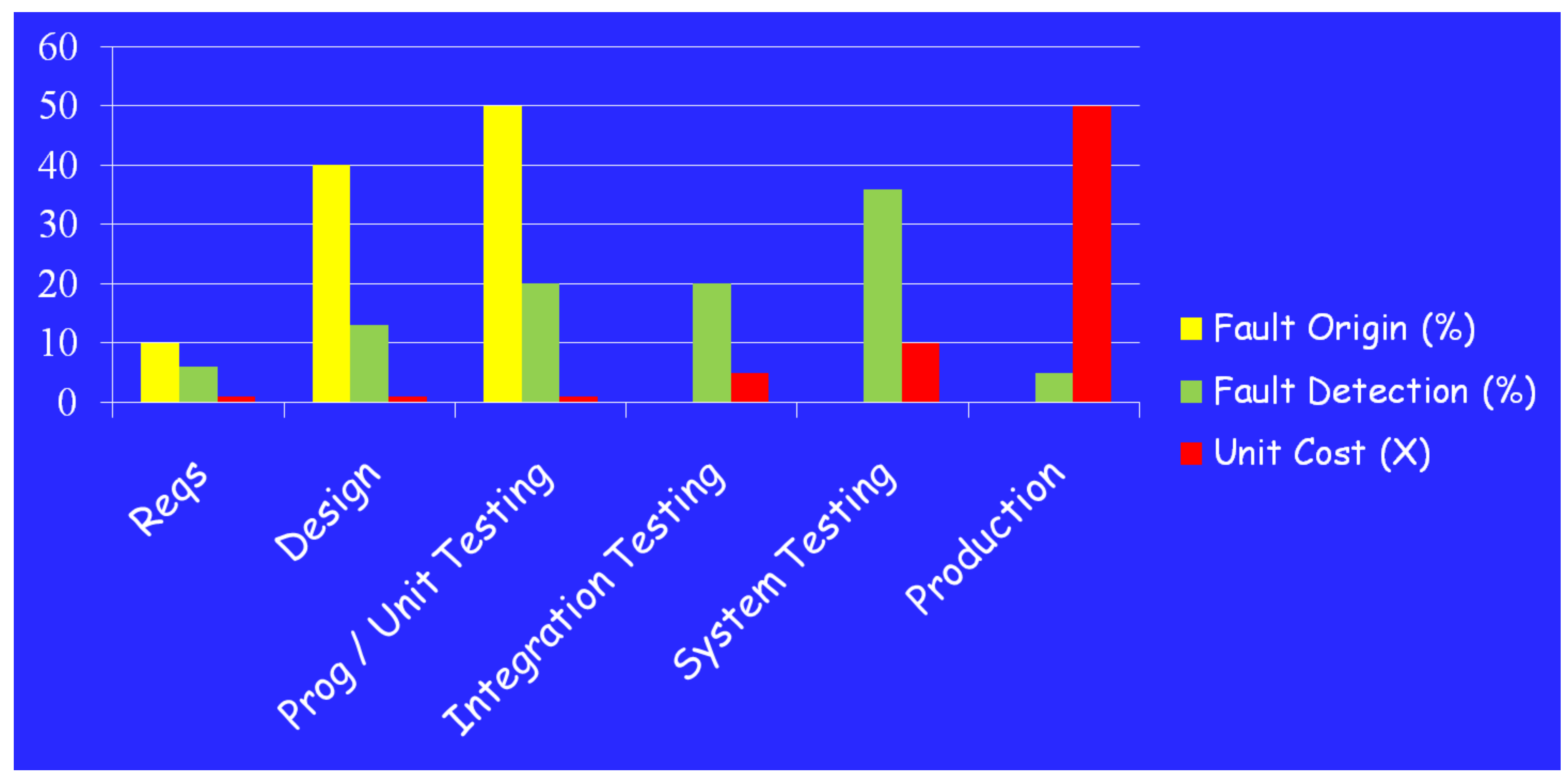
Relative cost to fix error

1000
500
200
100
50
20
10
5
2
1

Requirements
Design
Code
Development test
Acceptance test
Operation

Phase in which error was detected and corrected

Etapas dónde se producen errores



Software Engineering Institute; Carnegie Mellon University; Handbook CMU/SEI-96-HB-002

Momento de pruebas

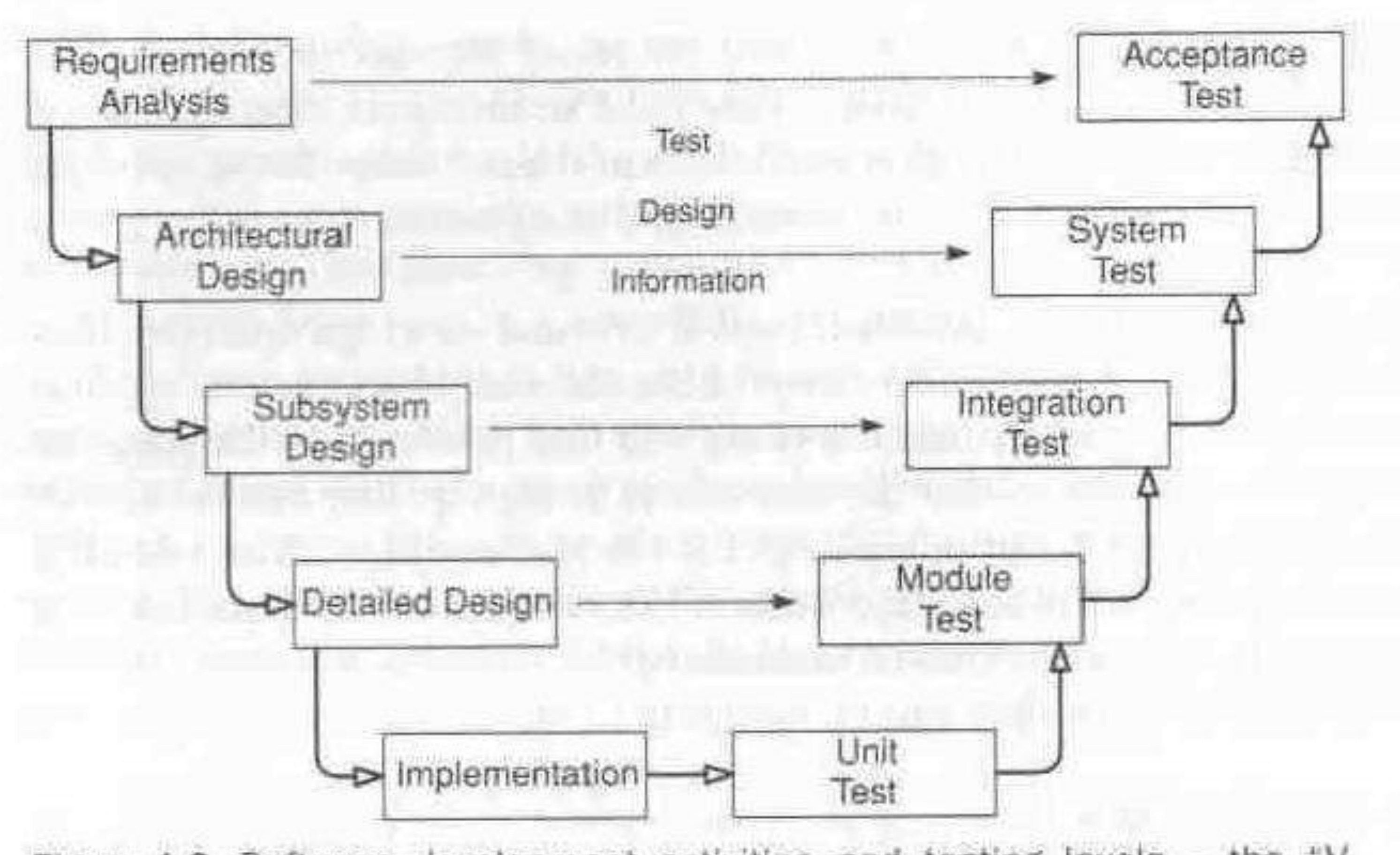
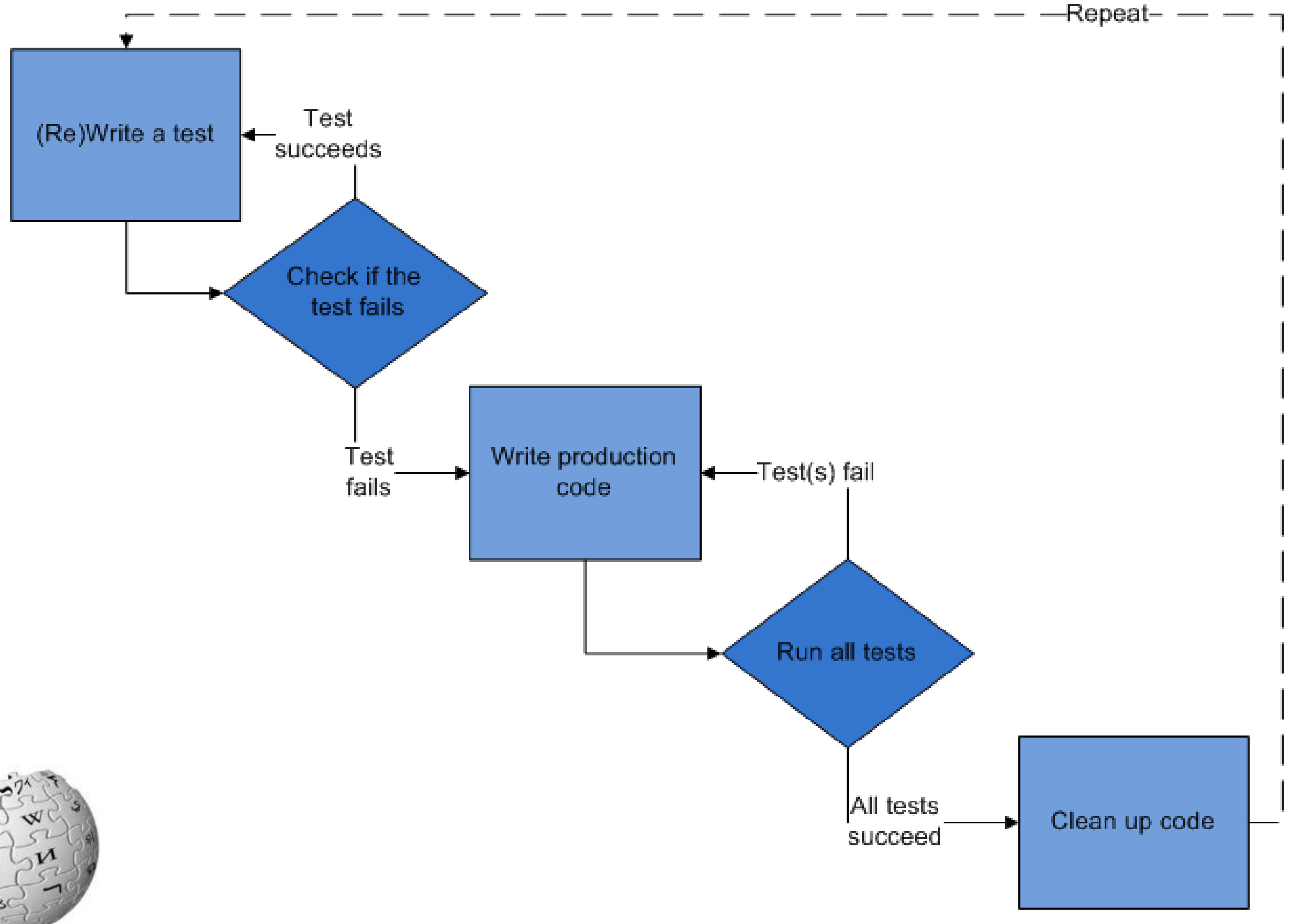


Figure 1.2. Software development activities and testing levels – the "V Model".

[Ammann y Offutt]

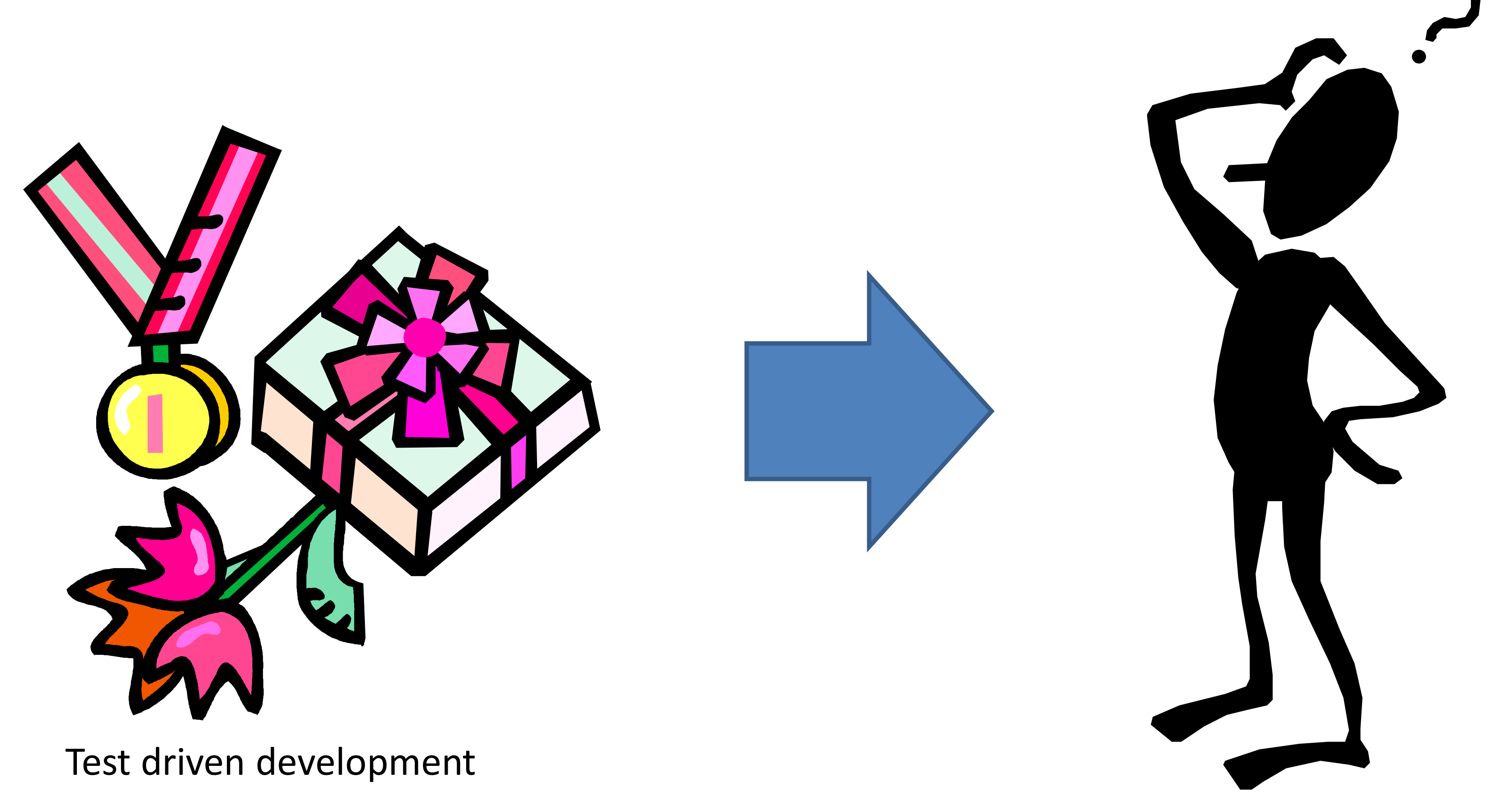
Desarrollo guiado por pruebas (TDD: Test Driven Development)



Desarrollo guiado por pruebas

- Pros:
  - Mientras antes pruebe antes detecto errores
  - Desarrollo incremental
  - Aumenta la productividad
  - Reduce tareas de depuración
- Cons:
  - Se pierde el diseño
  - No siempre aplicable en la práctica (GUIs)
  - Muy dependiente de las habilidades de pruebas de los desarrolladores

Desarrollo guiado por pruebas



Editor: Forrest Shull ■ Fraunhofer Center for Experimental Software Engineering, Maryland ■ fshull@fc-md.umd.edu

What Do We Know about Test-Driven Development?

Forrest Shull, Grigori Melnik, Burak Turhan, Lucas Layman, Madeline Diep, and Hakan Erdogmus

What if someone argued that one of your basic conceptions about how to develop software was misguided? What would it take to change your mind?

That's essentially the dilemma faced by advocates of test-driven development (TDD). The TDD paradigm argues that the basic cycle of developing code and then testing it to make sure it does what it's supposed to do—something drilled into most of us from the time we began learning software development—isn't the most effective approach. TDD replaces the traditional "code then test" cycle. First, you develop test cases for a small increment of functionality; then you write code that makes those tests run correctly. After each increment, you refactor the code to maintain code quality.[1]

TDD proponents assert that frequent, incremental testing not only improves the delivered code's quality but also generates a cleaner design. If you haven't already tried TDD, what data might convince you to try radically changing your software development approach to get those benefits? Would the experience of a recognized expert help?

In this column, we offer both data regarding TDD's effectiveness and the critique of an expert based on applying it in the field.

Compiling the Evidence

Our data comes from a study conducted by five of us—namely, Burak Turhan, Lucas Layman, Madeline Diep, Forrest Shull, and Hakan Erdogmus.[2] The study was based on a systematic literature review to aggregate demonstrated evidence about TDD's effectiveness. The review searched the literature from 1999, looking for any study that provided some quantitative assessment of TDD's effectiveness compared to traditional software development. The search results were filtered for quality, which left 22 published articles that described 33 unique studies.

The review distinguished three types of studies:

- *Controlled experiments* compared TDD to traditional development under controlled conditions to minimize the effects of confounding factors, such as developer experience or the type of software being developed.
- *Pilot studies* reported comparisons under somewhat realistic conditions but tended to be of short duration or on small problems.
- *Industry studies* reported comparisons regarding TDD's effectiveness on real projects being developed for a customer under real commercial pressures.

Reasoning that more rigorous studies might be fewer in number but should be more trustworthy, the reviewers defined a category of "high rigor" studies that met the following conditions:

- The subjects included only graduate students or professionals—that is, people who are more experienced than the general population and who should behave the most like developers in industry or government organizations.
- The study used a TDD process description that matched the textbook definition and

0740-7459/10/$26.00 © 2010 IEEE

Shull, F. et al, "What Do We Know about Test-Driven Development?," ***Software, IEEE*** , vol.27, no.6, pp.16-19, **Nov.-Dec. 2010**
doi: 10.1109/MS.2010.152

Índice

- Introducción
- ¿Cuándo hacer pruebas?

- Definiciones
- Proceso general
- Actividades
- Diseño de casos de prueba
- Técnicas de diseño de casos de prueba
- Resumen
- Bibliografía

Pruebas funcionales vs pruebas no funcionales

La Universidad de Sevilla colapsada el primer día de automatrícula

LUNES, 07 DE SEPTIEMBRE DE 2009 10:33 SEVILLA - CULTURA Y EDUCACIÓN

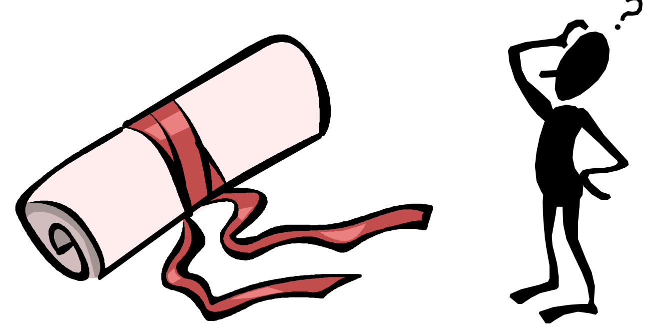
Los servidores de la Universidad de Sevilla quedaron completamente bloqueados con la apertura del plazo de automatrícula desde las 9 de esta mañana. Los universitarios que consiguieron acceder a la aplicación, tras esperar 10-15 minutos para que cargase la ventana de automatrícula, recibían una indicación de error de acceso obligándoles a repetir el proceso una y otra vez.

Universidad de Sevilla
Acceso a la automatricula

Ha habido un error en el acceso
BDO-0018: La sesión ya no está disponible.

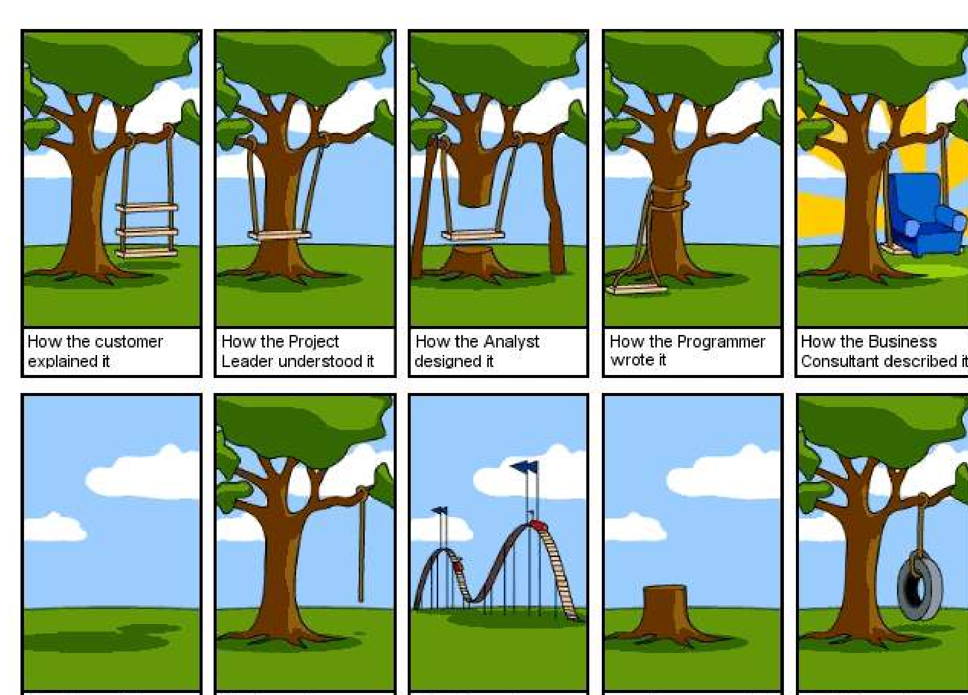
Volver a intentarlo

Puede activar o cambiar su clave de acceso en cualquier aula de informática de la Universidad, en el SOS de alumnos, sito en Avda. Reina Mercedes (Edificio Rojo, segunda planta), o en los Puntos de Información Universitaria accediendo con el carné universitario.



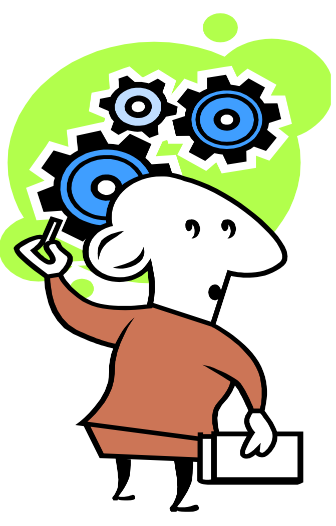
Definiciones

- **Verificación**: El proceso para determinar si los productos de una determinada fase del desarrollo software cumplen con los requisitos (contrato) de la fase previa. *¿El sistemas software está haciendo algo correctamente?*
- **Validación**: El proceso de evaluar el software al final del desarrollo para asegurar su conformidad con el uso que se le quiere dar. *¿El sistema software es el que necesito? ¿nos falta algo? ¿nos sobra algo?*



Definiciones

- Demostración formal: Análisis que permite *afirmar* que la implementación cumple sus requisitos
- Prueba: Análisis que permite *aumentar nuestra confianza* en que la implementación cumple sus requisitos



"Testing shows the presence, not the absence of bugs"
Dijkstra

Definiciones

- **Failure** (fallo): La incapacidad de un sistema para realizar su función, es decir, hay una diferencia entre lo que se espera que haga el sistema y lo que está haciendo el sistema. Se podría definir como "los síntomas".
- **Fault** (bug): La causa que provoca el fallo. Se podría definir como "la causa"
- **Error**: El estado interno de un programa que contiene un *bug.* Algunos estados no revelan un fallo.
- **Testing.** Evaluar el software observando su ejecución.
- **Debugging**: El proceso de encontrar un bug dado un fallo.

Definiciones

Se necesitan tres condiciones para poder observar un fallo (RIP)

1. Accesibilidad (R:Reachability): El lugar/lugares dónde se encuentra el fallo deben ser accedidos
2. Infección (I): Una vez accedido el estado del programa debe ser incorrecto
3. Propagación (P) : El estado infectado debe propagarse para poder observar su estado incorrecto

Índice

- Introducción
- ¿Cuándo hacer pruebas?
- Definiciones

- Proceso general
- Actividades
- Diseño de casos de prueba
- Técnicas de diseño de casos de prueba
- Resumen
- Bibliografía

Proceso general de prueba

- Un caso de prueba se suele documentar identificando los siguientes puntos (IEEE Standard for Software Testing Documentation):
  - Identificador del caso de prueba.
  - Entradas
  - Salidas esperadas
  - Dependencias (si es necesario ejecutar antes otros casos de prueba)

Índice

- Introducción
- ¿Cuándo hacer pruebas?
- Definiciones
- Proceso general
- Actividades
- Diseño de casos de prueba
- Técnicas de diseño de casos de prueba
- Resumen
- Bibliografía

Actividades en pruebas

- Se pueden distinguir 4 tipos de actividades
  1. **Diseño**
  2. **Automatización**
  3. **Ejecución**
  4. **Evaluación**

  1.a) Basado en criterios

  1.b) Basado en conocimiento
- Cada actividad requiere distintas competencias pero es frecuente que se use a la misma persona para las 4 actividades (no recomendado, especialmente para sistemas complejos y críticos)

Índice

- Introducción
- ¿Cuándo hacer pruebas?
- Definiciones
- Proceso general
- Actividades

- Diseño de casos de prueba
- Técnicas de diseño de casos de prueba
- Resumen
- Bibliografía

Diseño de casos de prueba

Un plan de pruebas exhaustivo es impracticable

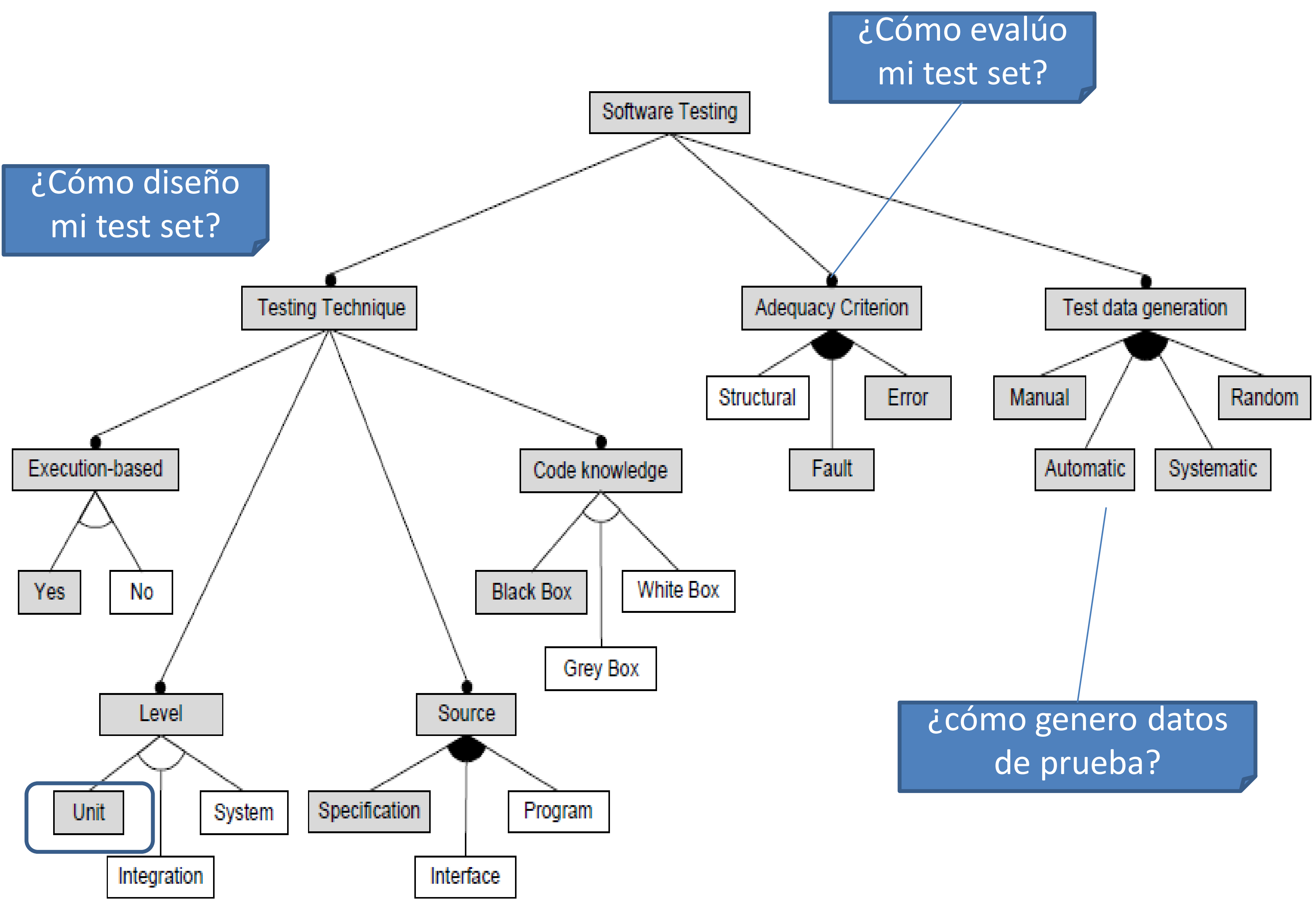
Las posibles entradas de un programa pueden (y suelen) ser infinitas, e.g. compilador

El objetivo de un buen banco de pruebas (*test set, test suite*): pocas entradas, muchos fallos.

¿Cómo seleccionar las entradas? Usando un criterio de cobertura (*coverage criteria*)

Criterio de cobertura: conjunto de reglas que imponen una serie de requisitos a un banco de pruebas (*test suite/ test set*).

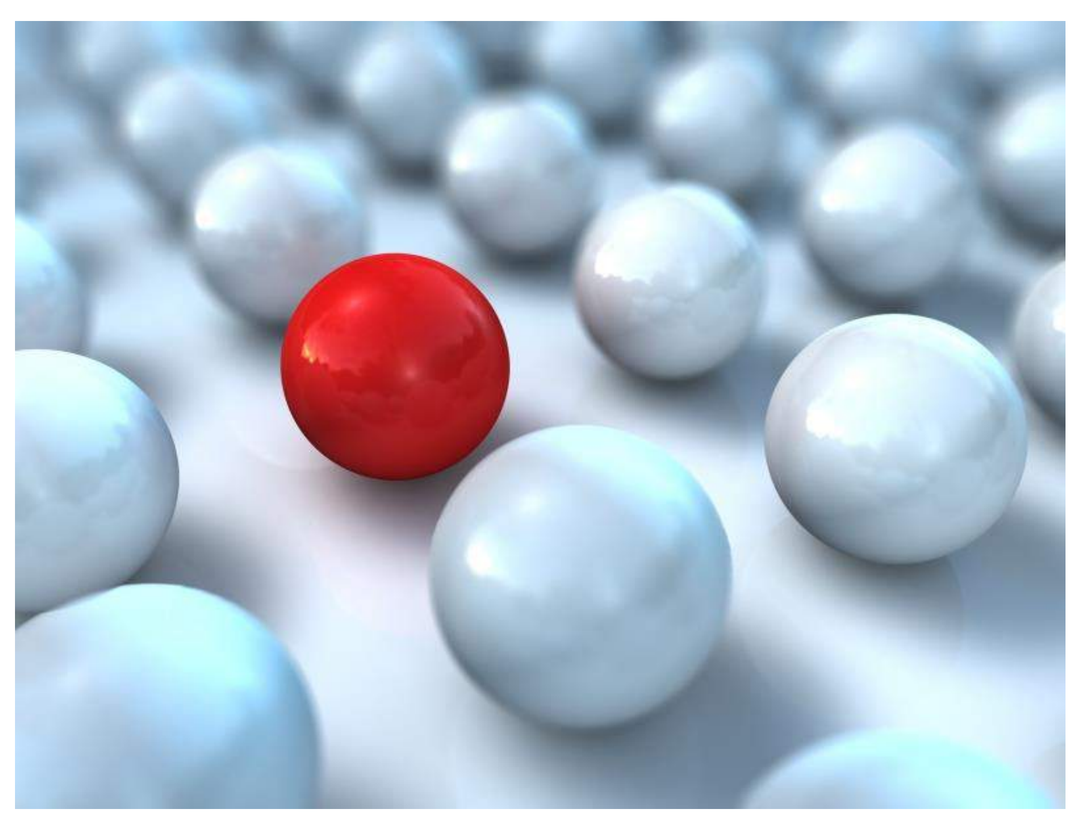
Clasificación



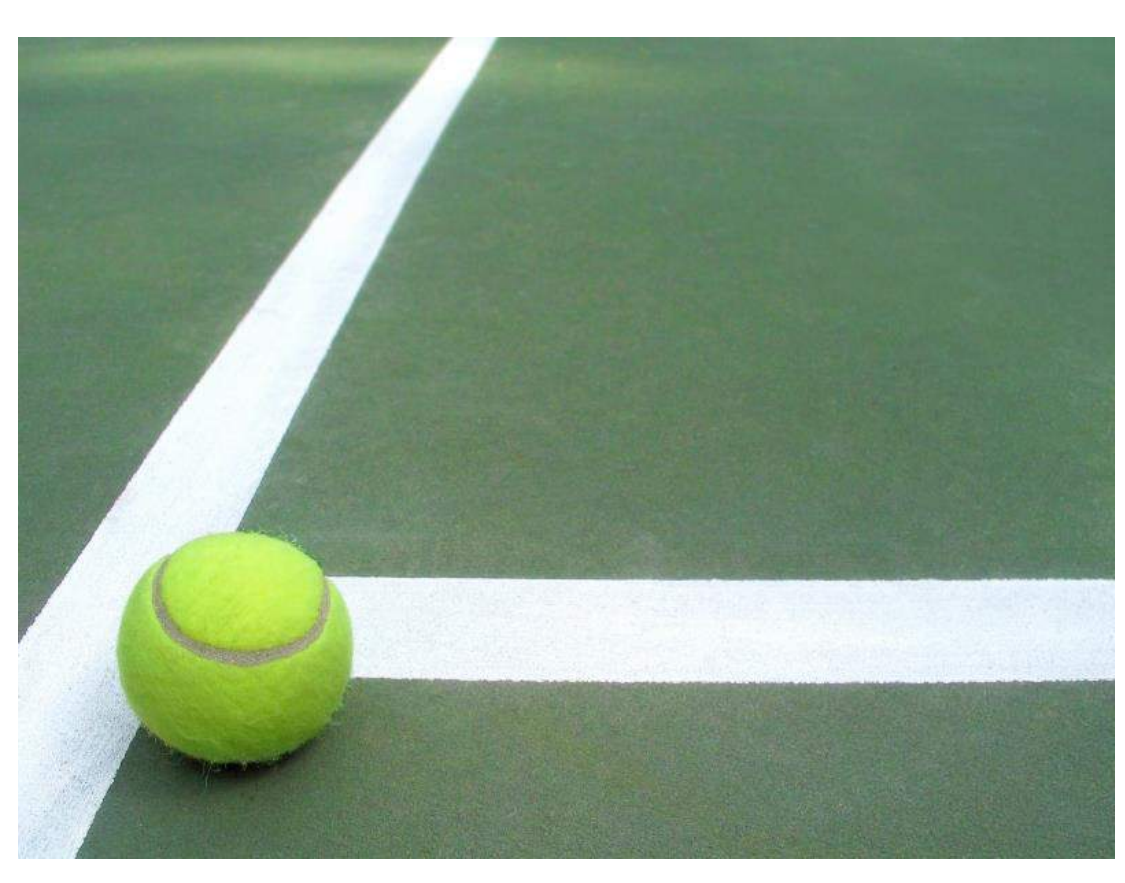
Las pruebas unitarias están diseñadas para ejercitar una parte pequeña y específica de funcionalidad

Con pruebas unitarias pequeñas acotamos el ámbito de un fallo

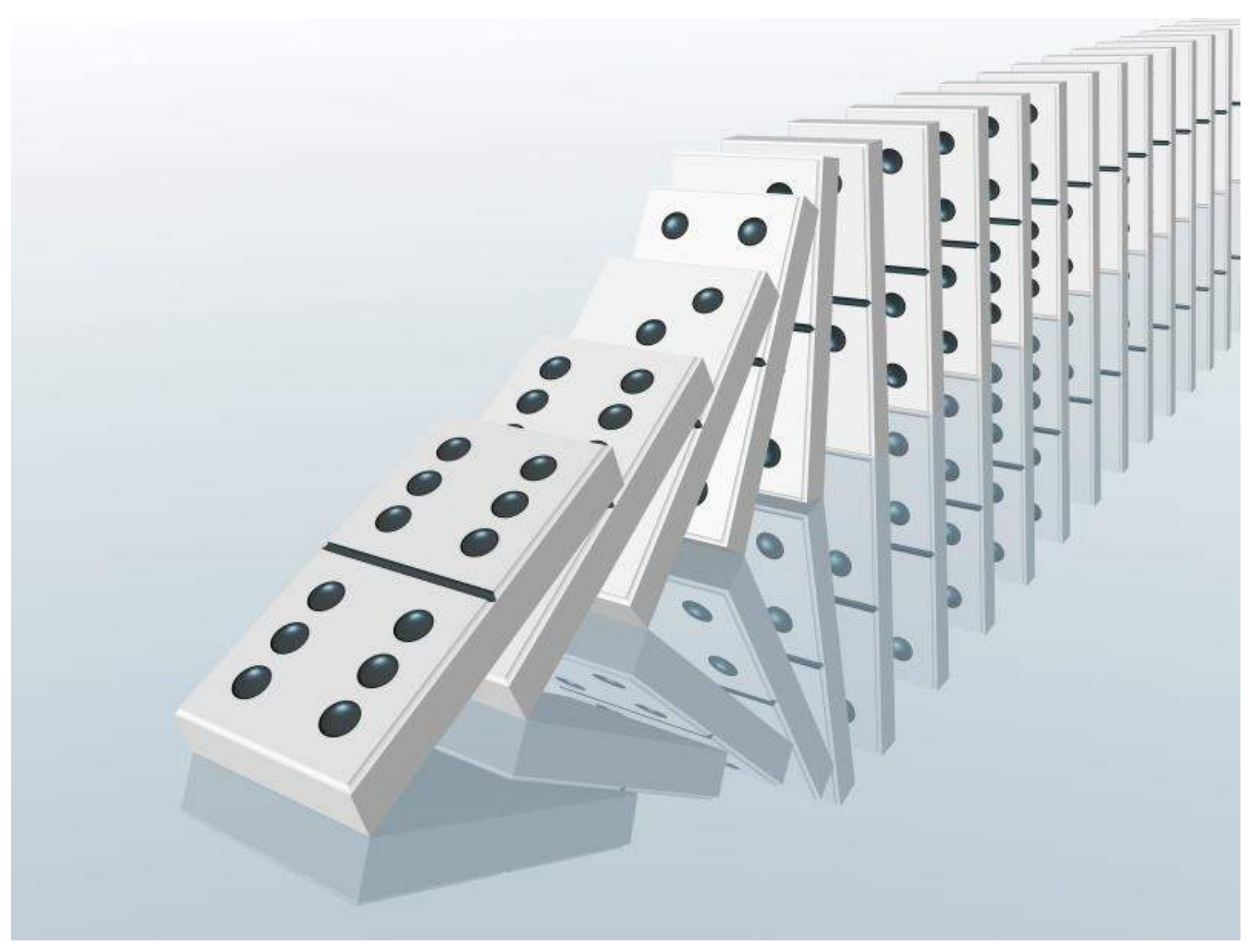
Las pruebas unitarias prueban que una pequeña parte aislada del software es correcta



Una prueba unitaria no debe ir más allá de sus límites

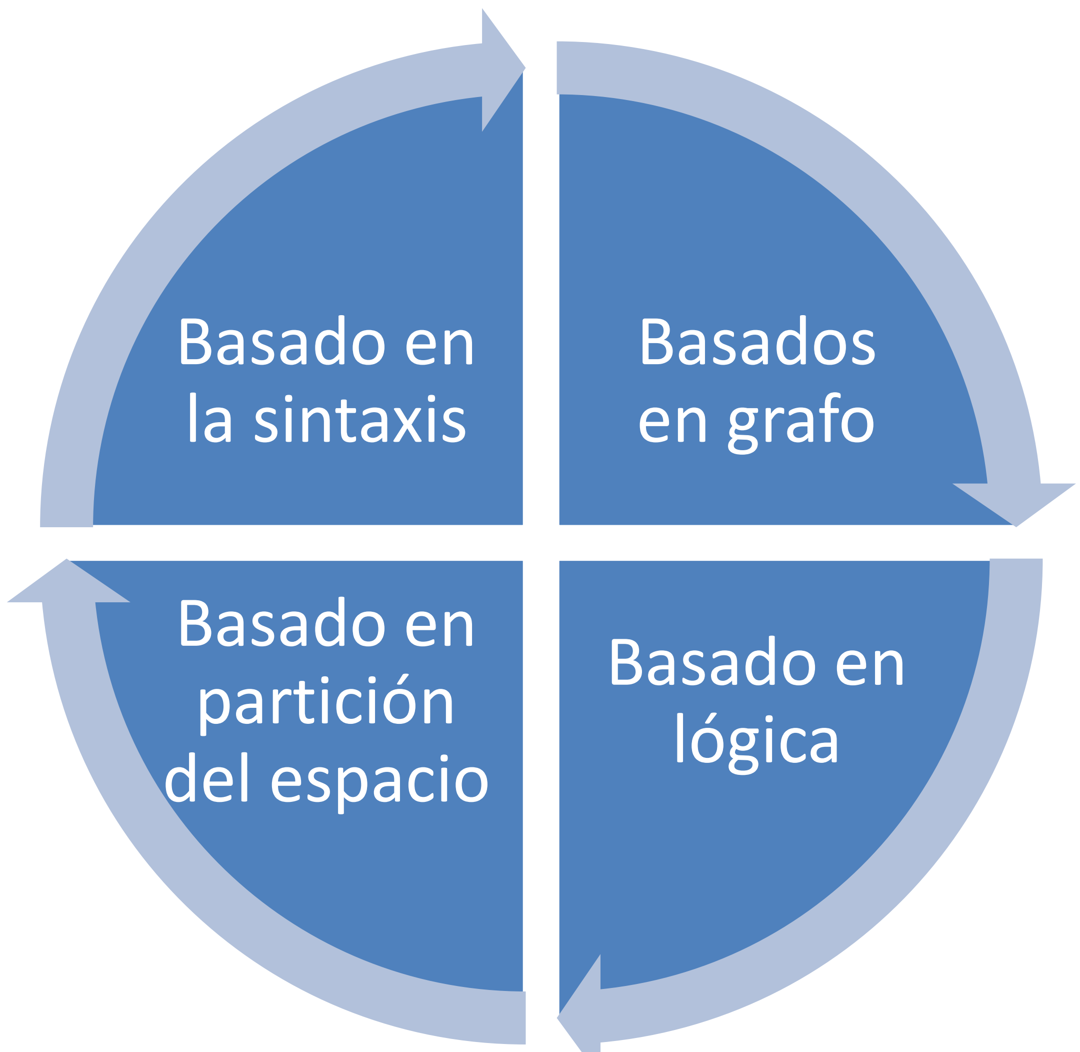


A veces no se pueden evitar las dependecias



El diseño de pruebas unitarias es una habilidad que se aprende con la práctica

Grupos de criterios



Paul AMMANN Jeff OFFUTT
INTRODUCTION TO SOFTWARE TESTING

Índice

Introducción

¿Cuándo hacer pruebas?

Definiciones

Proceso general

Actividades

Diseño de casos de prueba

Técnicas de diseño de casos de prueba

Resumen

Bibliografía

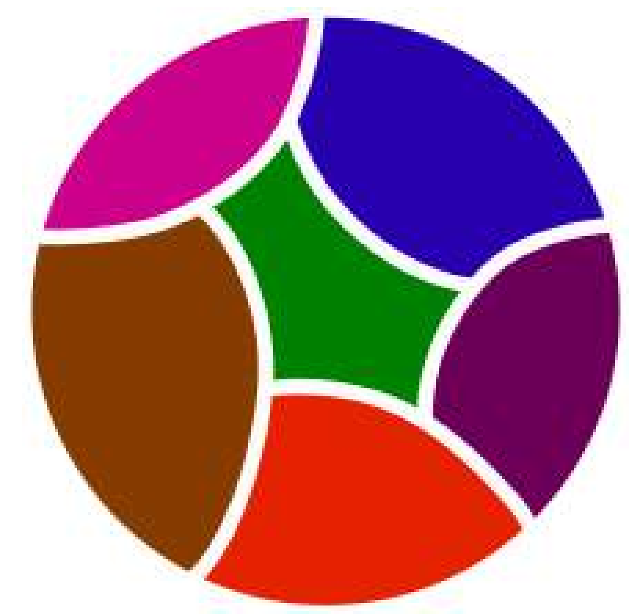
Índice

Técnicas para el diseño de casos de prueba

- **Particiones equivalentes (equivalent partitioning)**
- Valores límite (boundary analysis)
- Combinaciones: par-wise, n-wise
- Errores conocidos (error guessing)
- Cobertura CRUD

Particiones equivalentes

- Se divide el espacio de prueba en *clases equivalentes*
- Las particiones deben ser "disjuntas"



- Inconveniente: rápida explosión

Índice

Técnicas para el diseño de casos de prueba

- Particiones equivalentes (equivalent partitioning)
- **Valores límite (boundary analysis)**
- Combinaciones: par-wise, n-wise
- Errores conocidos (error guessing)
- Cobertura CRUD

Valores límite

- Según esta técnica los errores en los programas suelen estar en los valores límite de las entradas de un programa.
- Se prueban los valores límite de las particiones equivalentes en caso de haberlas

Particiones equivalentes y valores límite

- Ejemplo: El precio de la matrícula depende de la edad del cliente y hay tres rangos: menores de 18 años, entre 18 y 65 y mayores de 65.
  - 3 clases equivalente:

0-17	18-64	65-*

  - Usamos los valore límite: 17,18,19,64,65,66
  - O simplificado (el límite y los adyacentes): 17,18,64,65

Índice

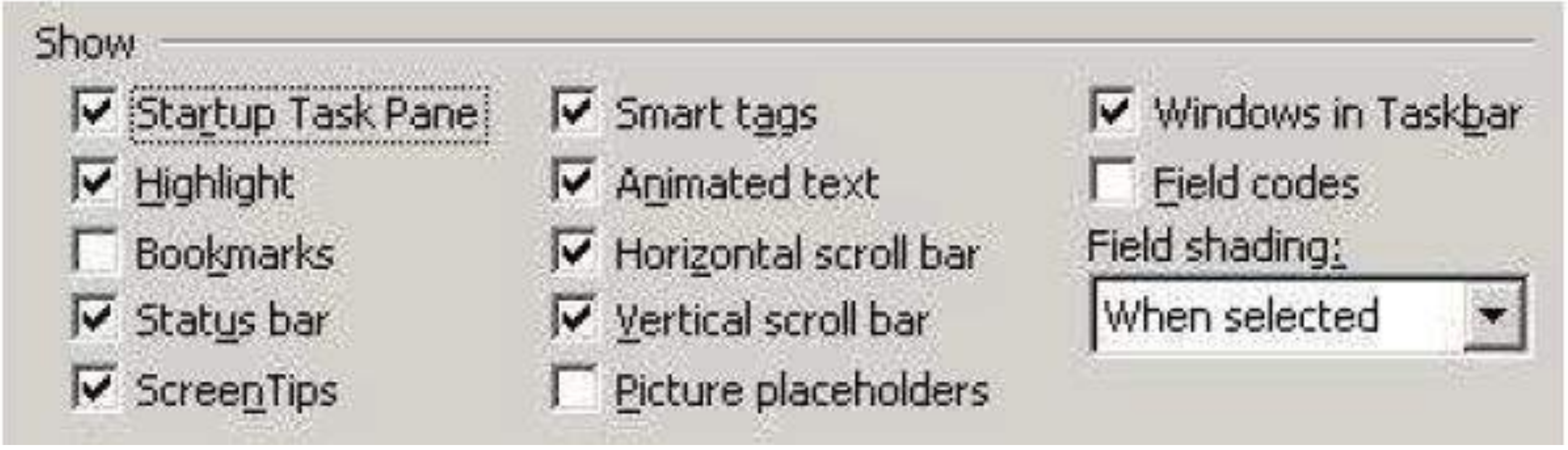
Técnicas para el diseño de casos de prueba

- Particiones equivalentes (equivalent partitioning)
- Valores límite (boundary analysis)
- **Combinaciones: par-wise, n-wise**
- Errores conocidos (error guessing)
- Cobertura CRUD

Pair-wise

- *Pairwise testing* (también *2-wise testing*). Es un método dentro de los llamados "combinatorios" que propone hacer pruebas sobre todas las posibles combinaciones de 2 parámetros de entrada
- La hipótesis que maneja es que la mayoría de los errores son debidos a errores en un parámetro de entrada, la siguiente es la combinación de pares de parámetros de entrada, etc.

Pair-wise



- 12.288 combinaciones = 2^{12} x 3
- Con pair-wise se reduciría ese número

Pair-wise

- Imagina X,Y,Z booleanos

Todas las posibles pruebas

	X	Y	Z
T1	0	0	0
T2	0	0	1
T3	0	1	0
T4	0	1	1
T5	1	0	0
T6	1	0	1
T7	1	1	0
T8	1	1	1

Aplicando 2-wise

	X	Y	Z
T1	0	0	0
T4	0	1	1
T6	1	0	1
T7	1	1	0

Índice

Técnicas para el diseño de casos de prueba

- Particiones equivalentes (equivalent partitioning)
- Valores límite (boundary analysis)
- Combinaciones: par-wise, n-wise
- **Errores conocidos (error guessing)**
- Cobertura CRUD

Errores conocidos, *error guessing*

- Errores comunes según nuestra experiencia
- Ejemplo: Cuándo se almacena un voto se puede almacenar con el mismo ID

Índice

Técnicas para el diseño de casos de prueba

- Particiones equivalentes (equivalent partitioning)
- Valores límite (boundary analysis)
- Combinaciones: par-wise, n-wise
- Errores conocidos (error guessing)
- **Cobertura CRUD**

Cobertura CRUD

- Para cada elemento persistente, se prueban todas sus operaciones *CRUD*
- Cada caso de prueba comienza por una *C,* seguida por todas las *U y se* termina por una D
- Tras cada *C, U o D,* se ejecuta una R que nos sirve de oráculo

- Ejemplo:
- Crear *Papeleta*
- Verificar que se ha creado
- Actualizar sus valores
- Verificar que se han actualizado
- Borrar *Papeleta*
- Verificar que se ha borrado

Índice

Introducción

¿Cuándo hacer pruebas?

Definiciones

Proceso general

Actividades

Diseño de casos de prueba

Técnicas de diseño de casos de prueba

Resumen

Bibliografía

Resumen

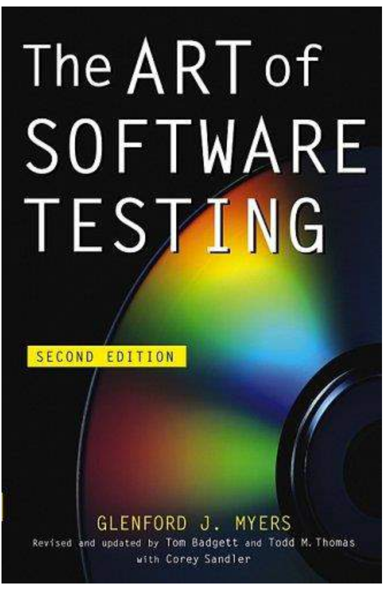
- ¿Qué hemos aprendido?
  - Hemos afianzado nuestra noción de que las pruebas de software son importantes
  - Diferencia entre ejecución y diseño de pruebas
  - Alcance de las pruebas unitarias
  - Técnicas para el diseño de casos de prueba: particiones equivalentes, valores límite, n-wise testing, errores conocidos, cobertura CRUD
- ¿Qué podemos hacer en el proyecto?
  - Identificar cómo nuestros proyectos hacen pruebas o proponer cómo se podrían hacer
  - Observar qué técnicas de casos de prueba se usan o proponer alguna
  - Observar qué herramientas se usan para pruebas tanto funcionales como no funcionales

Índice

- Introducción
- ¿Cuándo hacer pruebas?
- Definiciones
- Proceso general
- Actividades
- Diseño de casos de prueba
- Técnicas de diseño de casos de prueba
- Resumen

- Bibliografía

Bibliografía



IEEE Std 829-1998
(Revision of
IEEE Std 829-1983)

IEEE Standard for Software Test Documentation