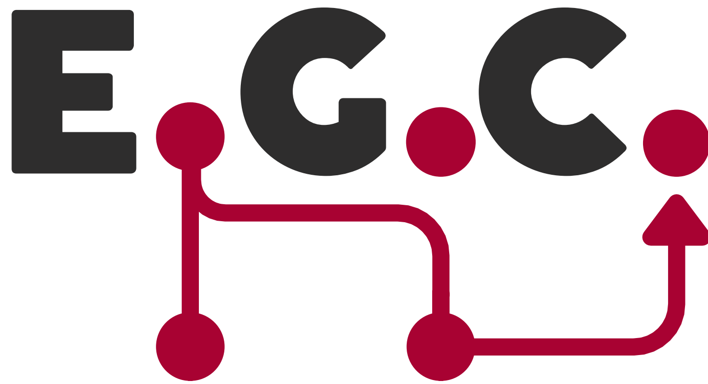
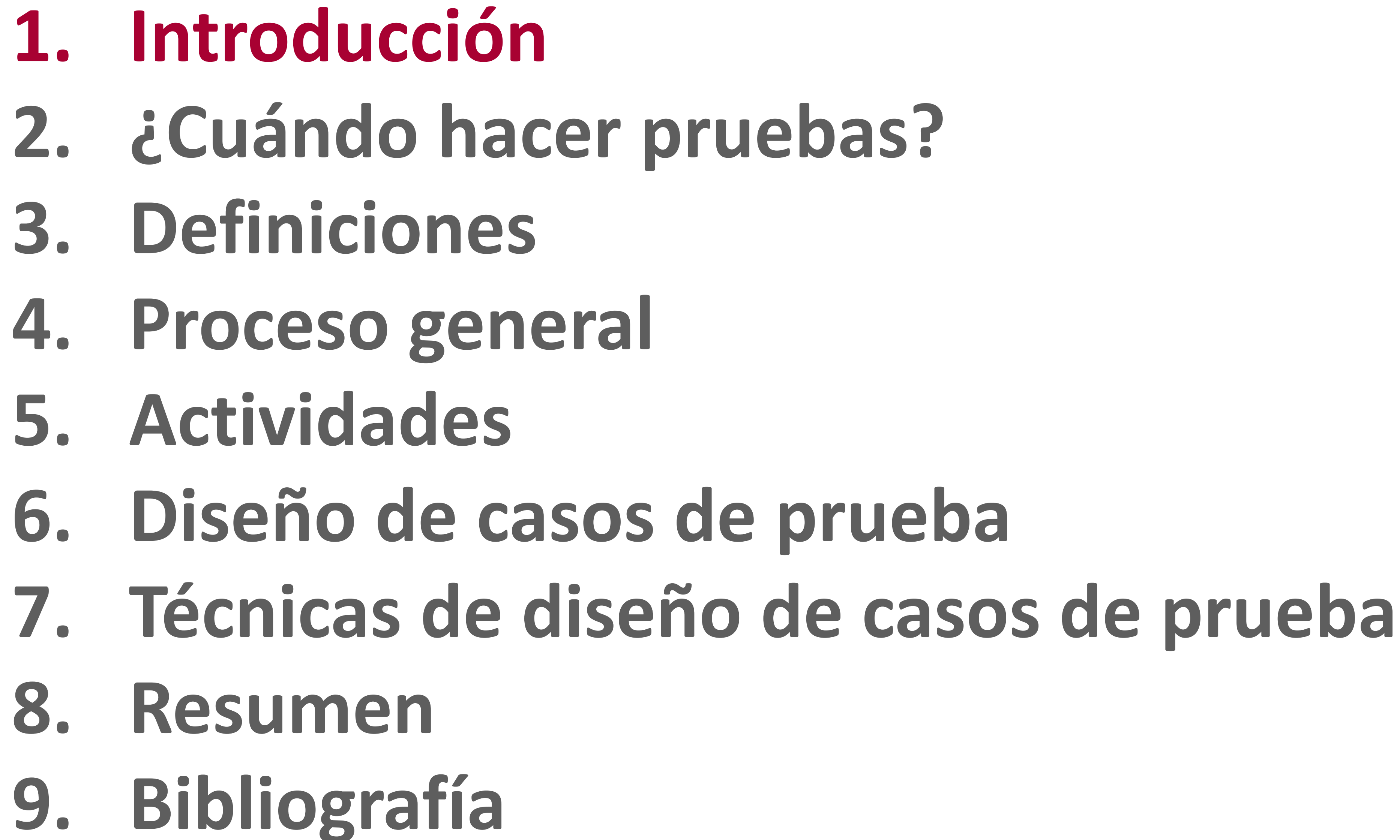


Grado en Ingeniería Informática – Ingeniería del Software

Evolución y Gestión de la Configuración

Escuela Técnica Superior de
Ingeniería Informática

# Pruebas de software

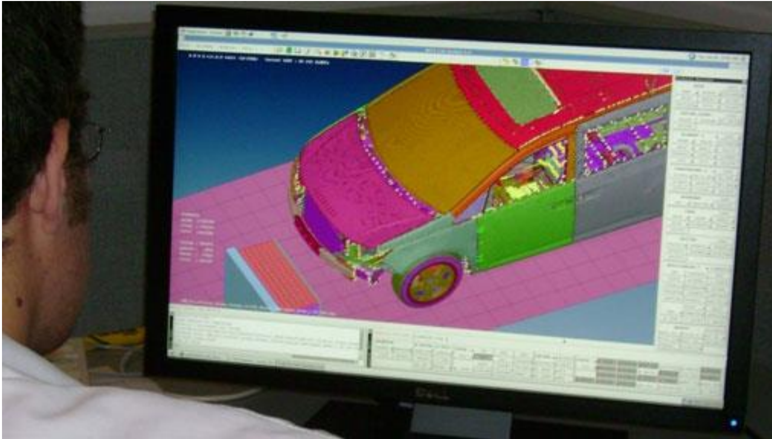

1. Introducción
2. ¿Cuándo hacer pruebas?
3. Definiciones
4. Proceso general
5. Actividades
6. Diseño de casos de prueba
7. Técnicas de diseño de casos de prueba
8. Resumen
9. Bibliografía

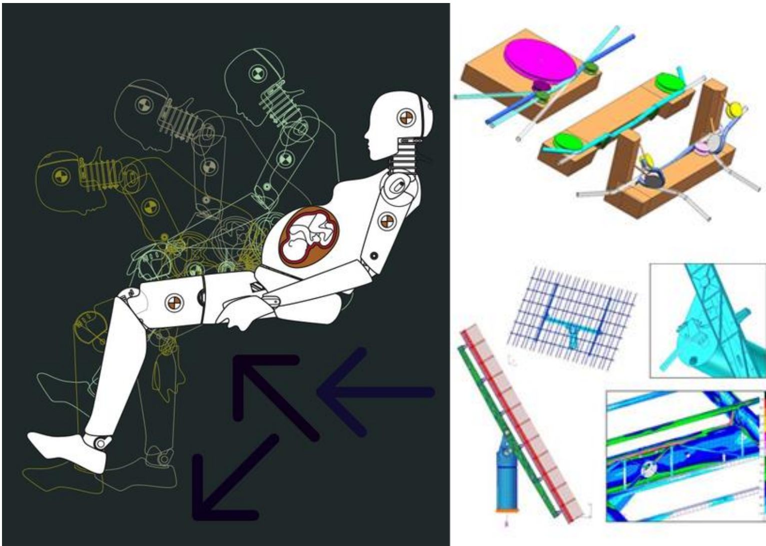
# 1. Introducción | ejemplo de otro dominio

**Probar:** hacer examen y experimento de las cualidades de alguien o algo.

***Test:*** *a critical examination, observation or evaluation.*

# 1. Introducción | ejemplo de otro dominio

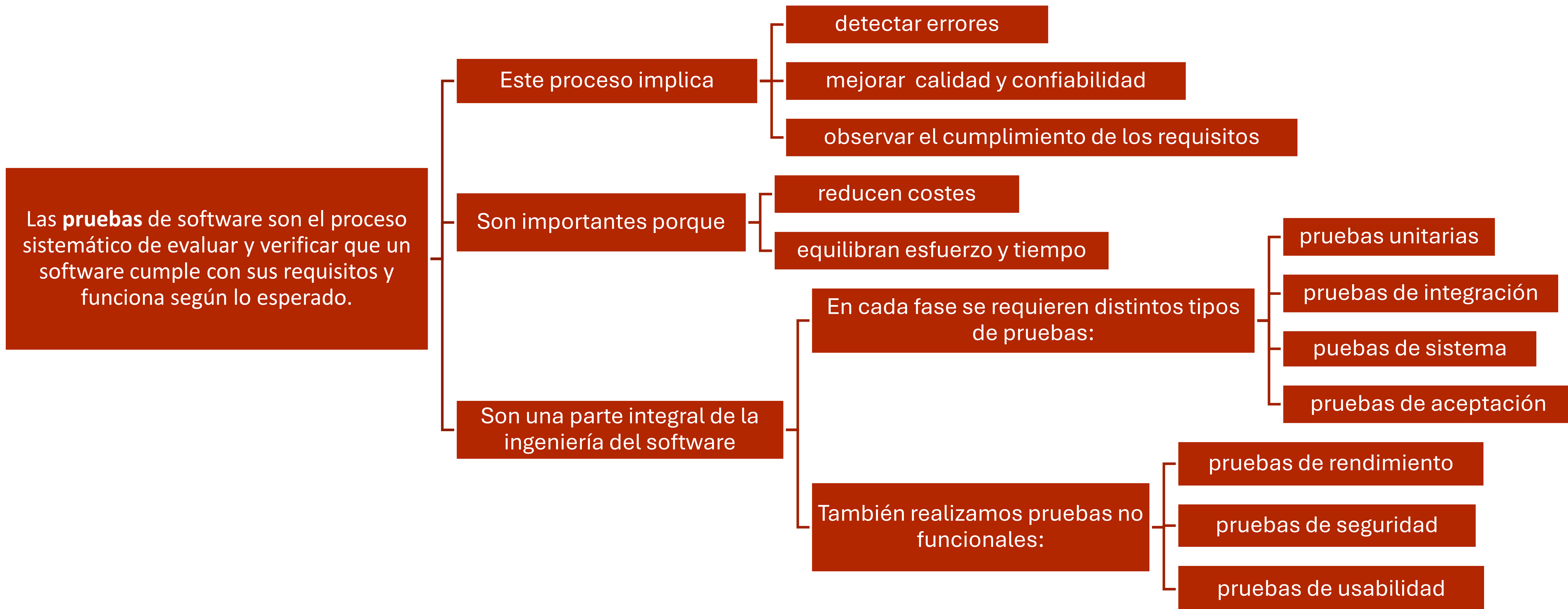




**Diseño** de la prueba

**Ejecución** de la prueba

# 1. Introducción | esquema



# 1. Introducción | ejemplos

uvlhub.io

## Autenticación:

- **Crear usuario** → probar si un usuario puede ser creado exitosamente en el sistema mediante el formulario de registro
- **Inicio de sesión** → probar que el usuario puede iniciar sesión con credenciales válidas
- **Autenticación fallida** → probar que los intentos de iniciar sesión con credenciales incorrectas son rechazados

## Hubfile:

- **Transformación UVL exitosa** → probar que un archivo UVL es procesado y transformado correctamente a otro formato. Por ejemplo, JSON
- **UVL mal formado** → probar si el sistema responde como se espera (con un error controlado) cuando el archivo UVL no es válido

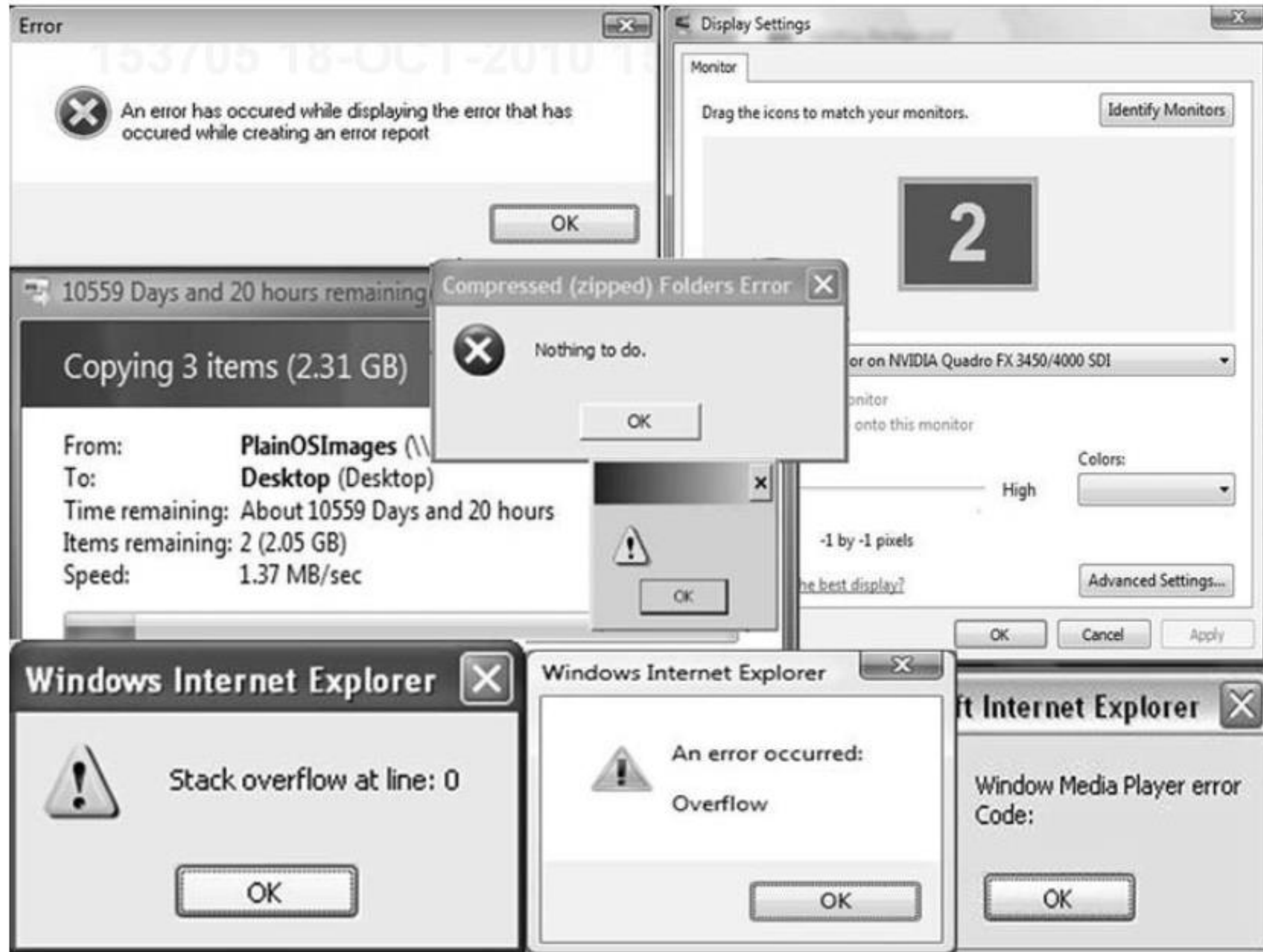
# 1. Introducción |

## ¿cuál es el problema?

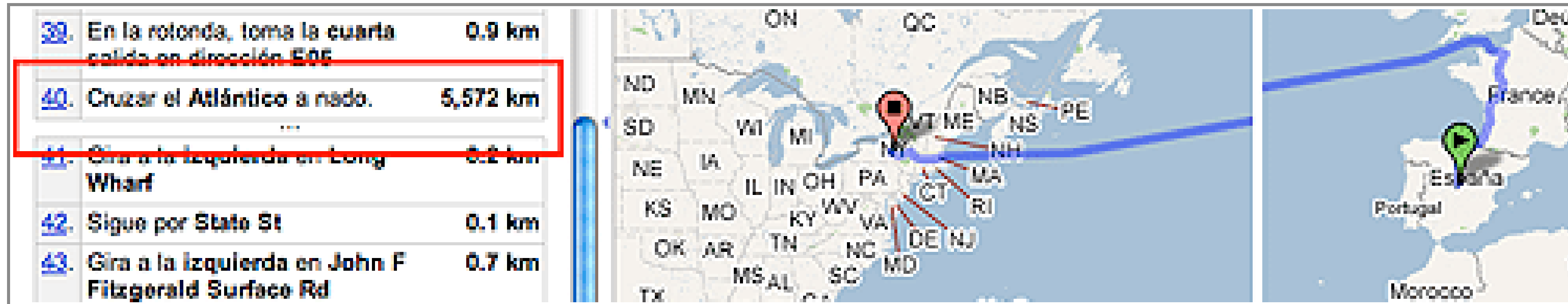
¿Cómo probar que un sistema software está haciendo algo bien/mal?

¿Cómo diseñar las pruebas?

# 1. Introducción | ¿por qué es importante?



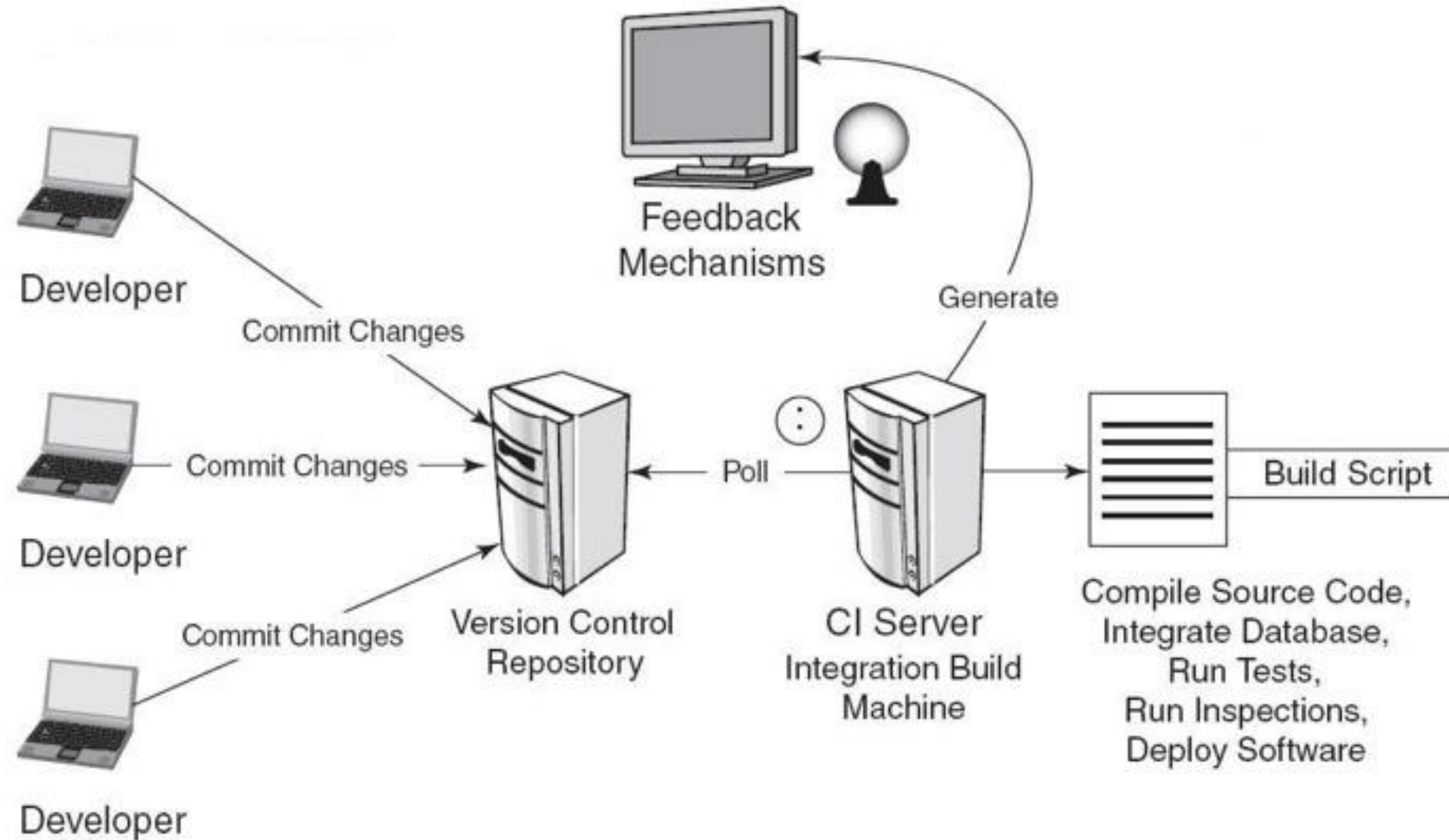
# 1. Introducción | ejemplos de fallos



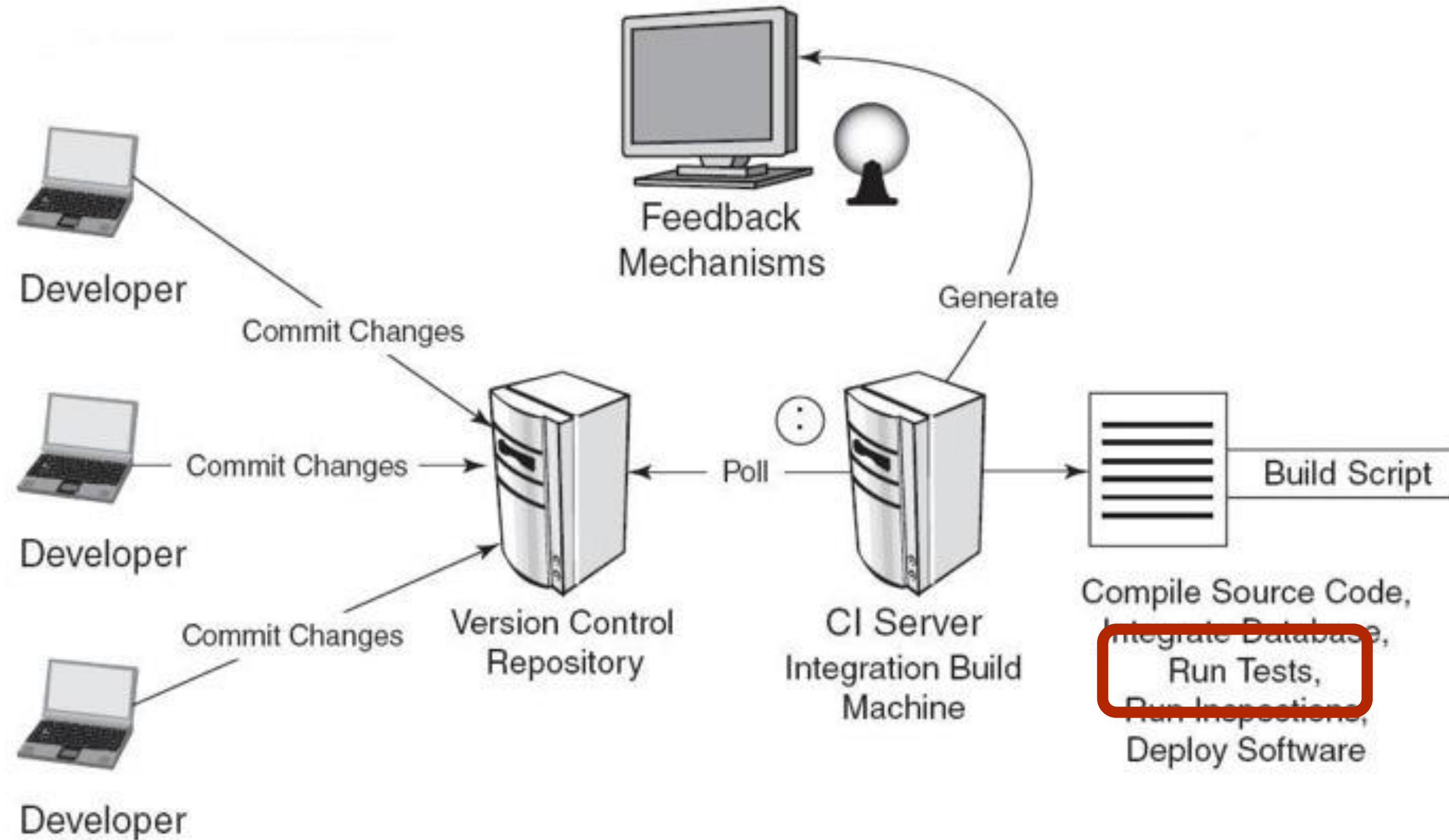

**WhatsApp, Facebook e Instagram sufren una caída mundial de más de seis horas**

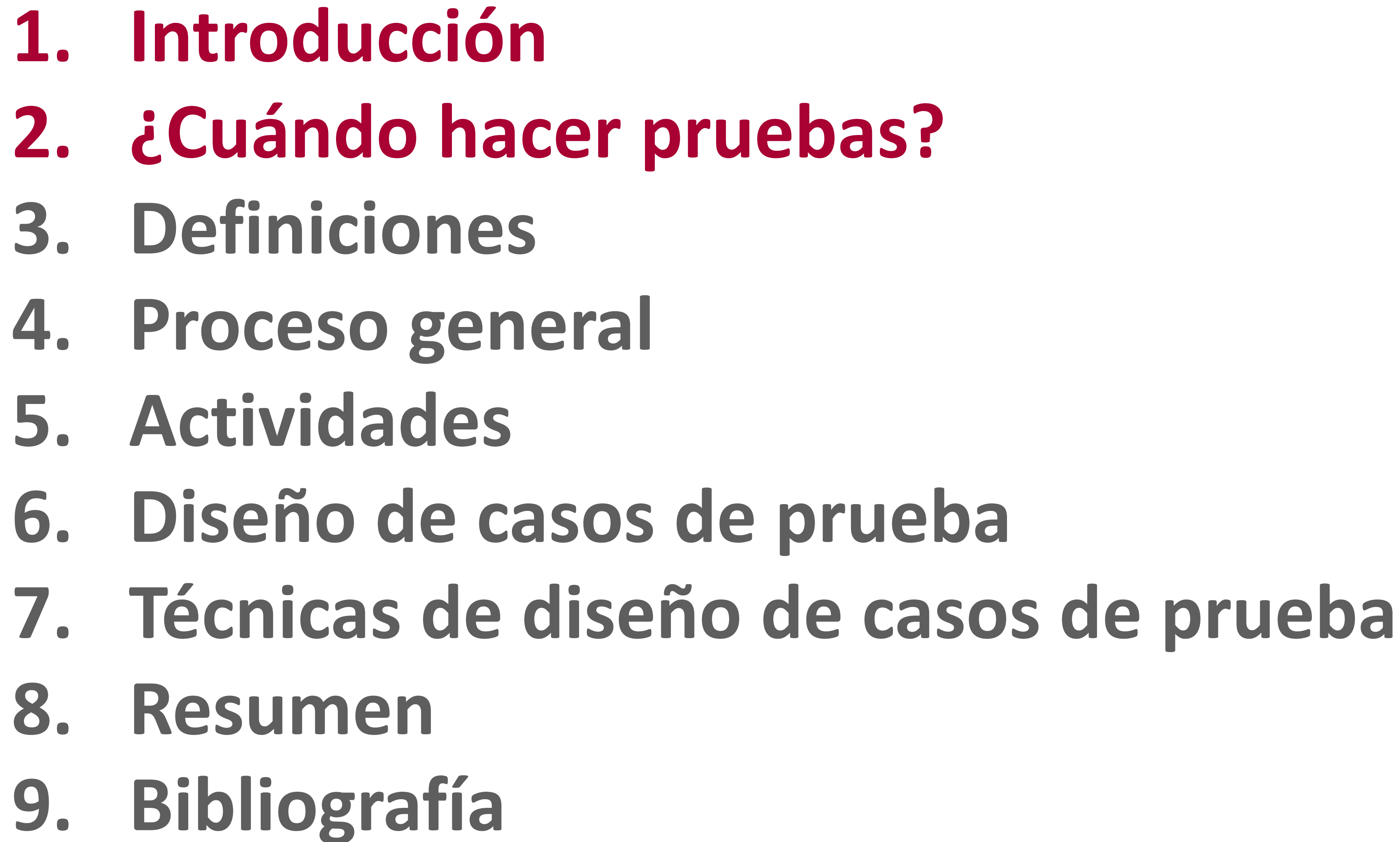
> *"There are two ways to write error-free proframs; only the third one Works"*
>
> Alan J. Perlis

# 1. Introducción | ¿por qué es importante?

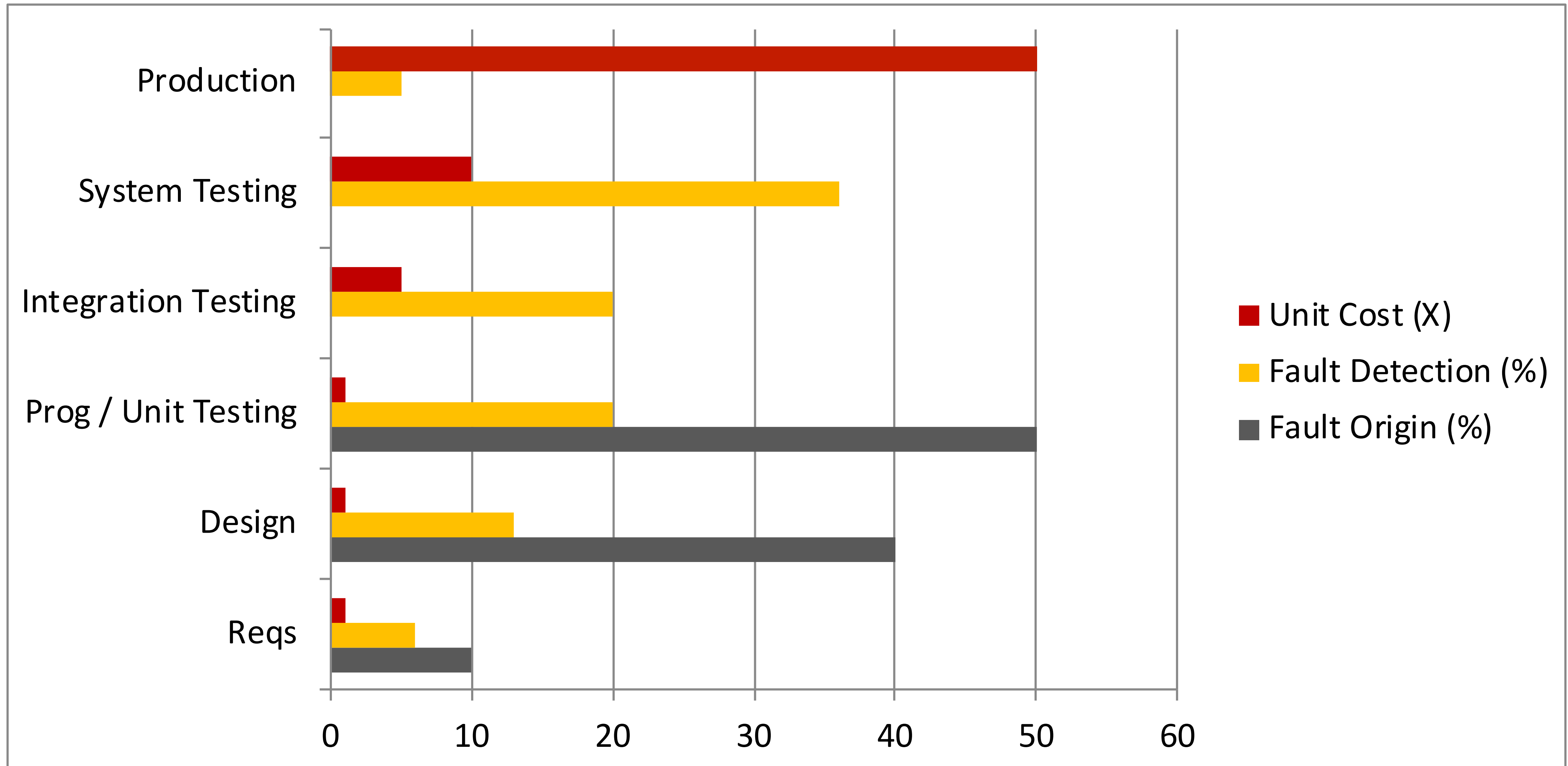


# 1. Introducción | ¿por qué es importante?



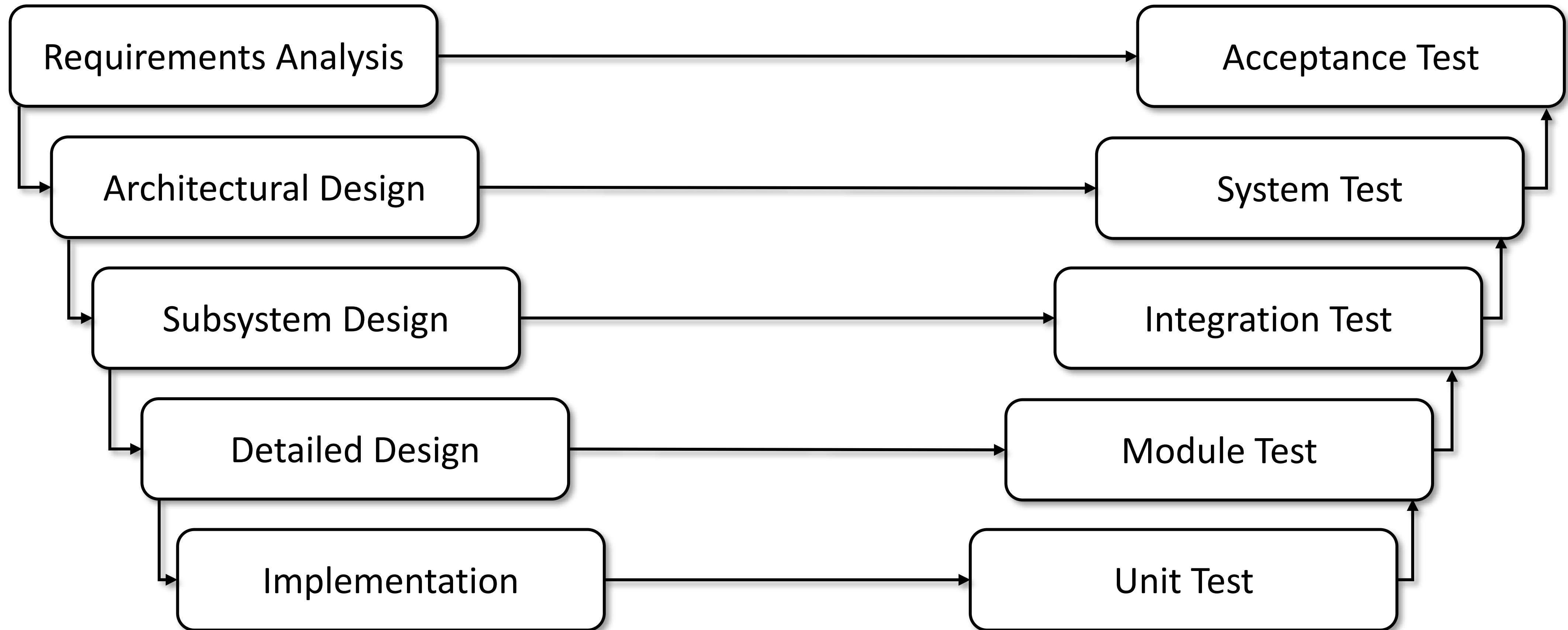

1. Introducción
2. ¿Cuándo hacer pruebas?
3. Definiciones
4. Proceso general
5. Actividades
6. Diseño de casos de prueba
7. Técnicas de diseño de casos de prueba
8. Resumen
9. Bibliografía

# 2. ¿Cuándo hacer pruebas? |




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

# 2. ¿Cuándo hacer pruebas? |momento de pruebas




[Ammann y Offutt]

# 2. ¿Cuándo hacer pruebas? | desarrollo guiado por pruebas

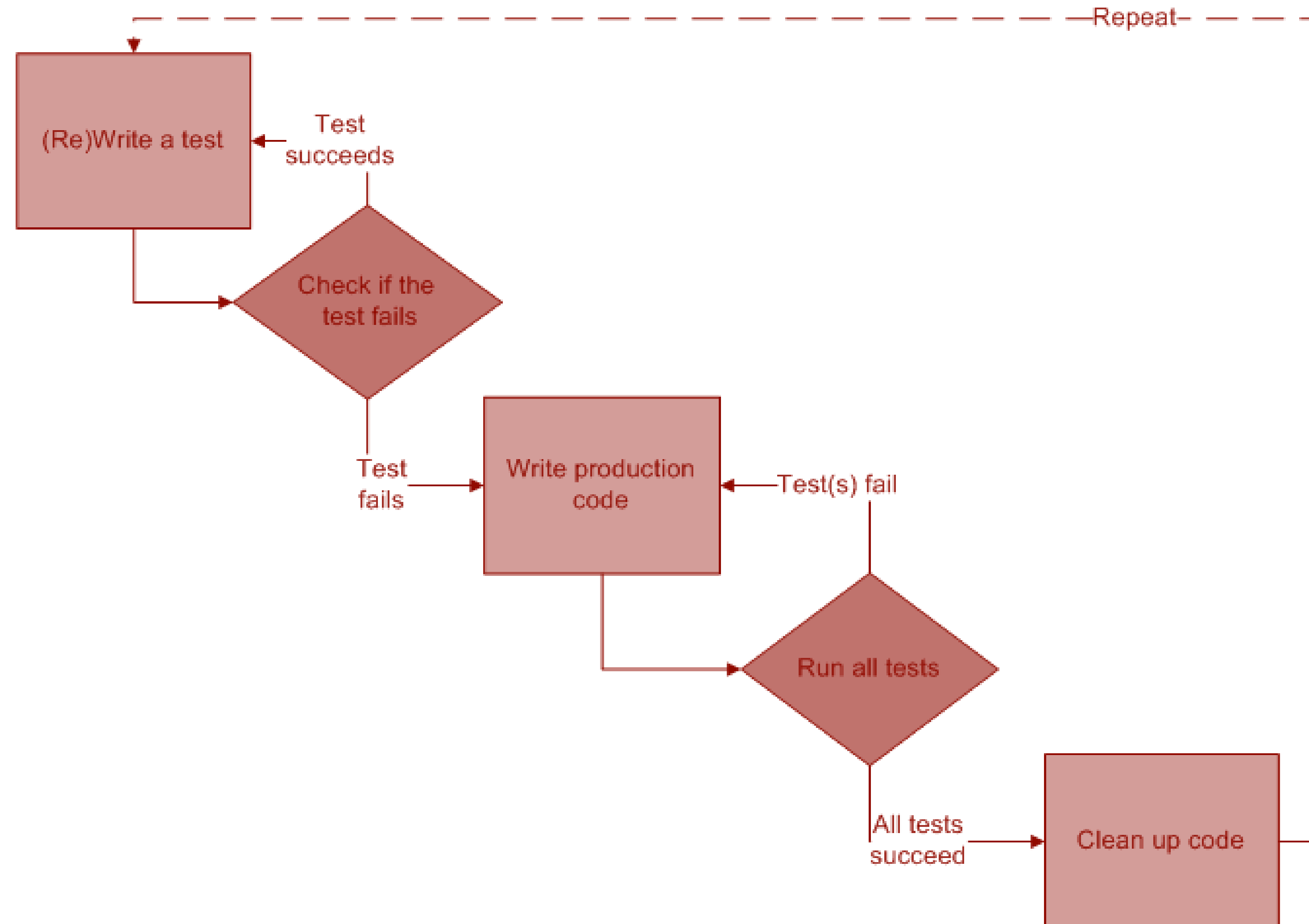
## TDD: Test Driven Development

Metodología de desarrollo de software en la que las pruebas unitarias se escriben antes del código de la funcionalidad que se quiere implementar. El ciclo básico de TDD sigue tres pasos:

**1. Escribir una prueba fallida**: escribir una prueba para una funcionalidad que aún no está implementada.

**2. Escribir el código mínimo**: escribir el código necesario para pasar la prueba.

**3. Refactorizar:** mejorar el código manteniendo las pruebas exitosas.



# 2. ¿Cuándo hacer pruebas? | desarrollo guiado por pruebas

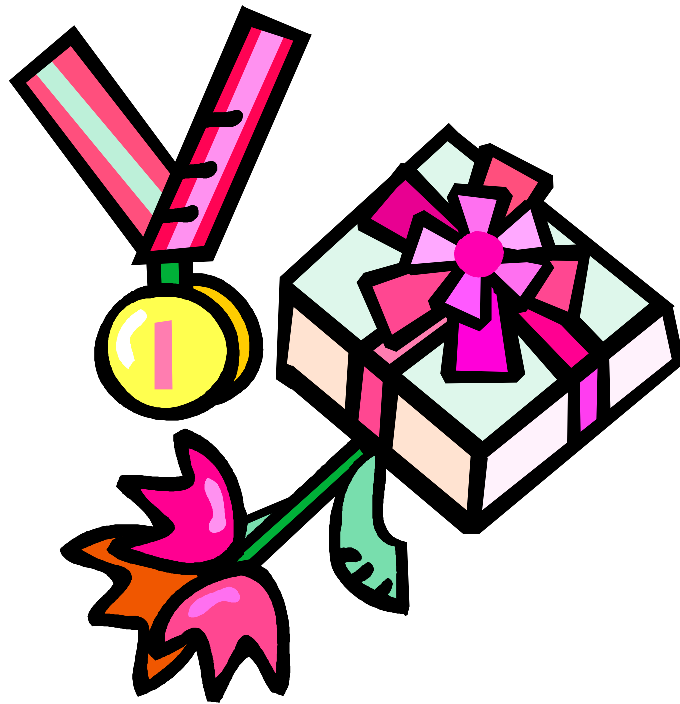
**VENTAJAS**

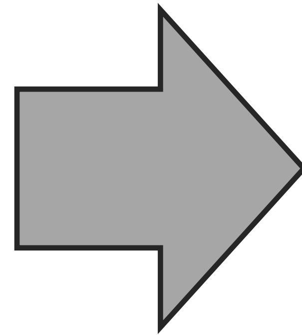
- Mientras antes pruebe, antes detecto errores
- Desarrollo incremental
- Aumenta la productividad
- Reduce tareas de depuración

**DESVENTAJAS**

- Se pierde el diseño
- No siempre es aplicable en la práctica (GUIs)
- Muy dependiente de las habilidades de pruebas de los desarrolladores

# 2. ¿Cuándo hacer pruebas? | desarrollo guiado por pruebas





# 2. ¿Cuándo hacer pruebas? | desarrollo guiado por pruebas

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

voice of evidence

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:

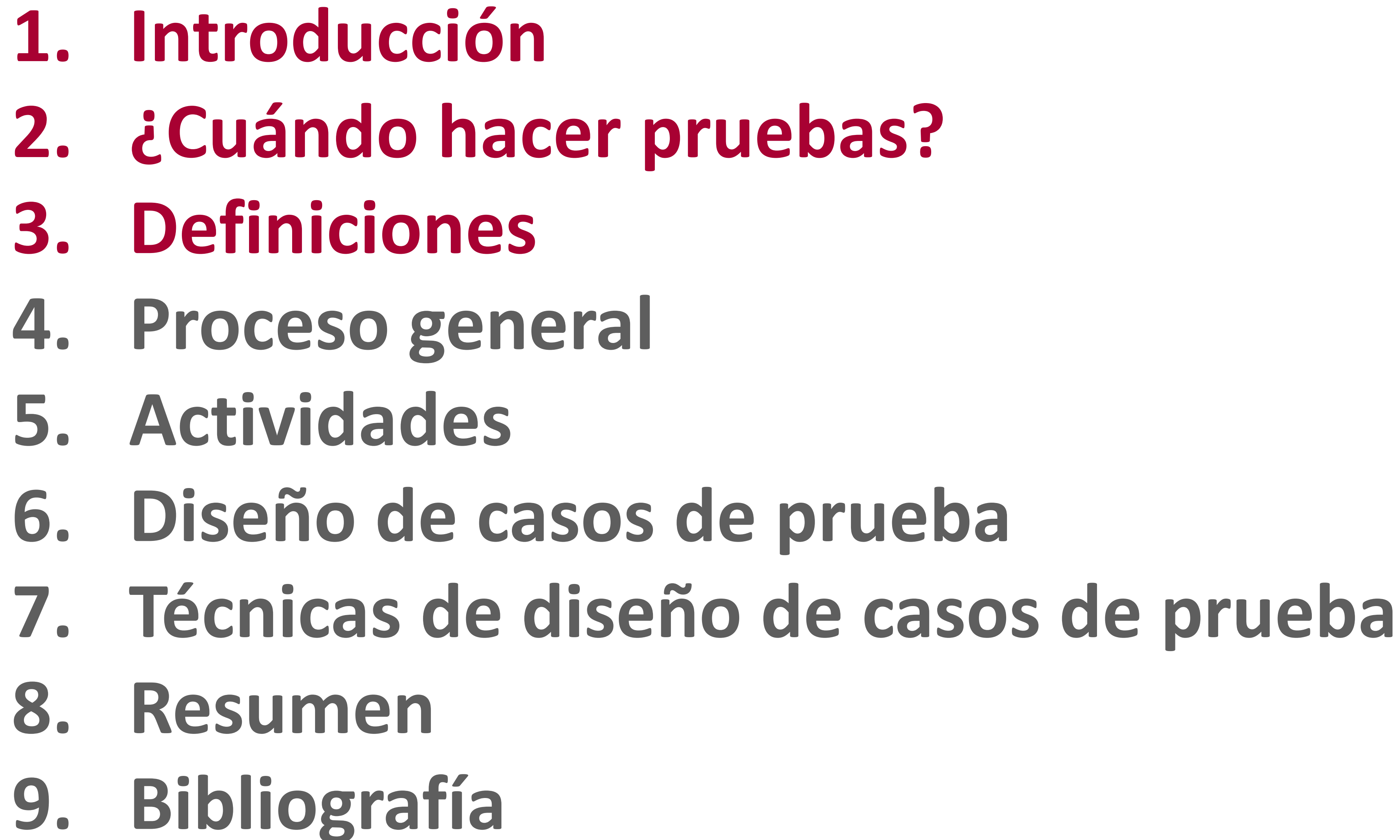
- *Controlled experiments* compared TDD to traditional development under controlled conditions to minimize the effects of confounding factors, such as developer experience or the type of software being developed.
- *Pilot studies* reported comparisons under somewhat realistic conditions but tended to 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


1. Introducción
2. ¿Cuándo hacer pruebas?
3. Definiciones
4. Proceso general
5. Actividades
6. Diseño de casos de prueba
7. Técnicas de diseño de casos de prueba
8. Resumen
9. Bibliografía

# 3. Definiciones|pruebas funcionales vs 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.

Automatrícula de la Universidad de Sevi... | Automatrícula de la Universidad ...

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

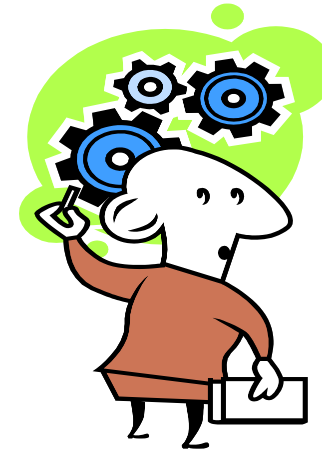
Puede activar o cambiar su clave de acceso en cualquier aula de informática de la Universidad, en el SOS de alumnos, sito en Avda. Reina Mercedes (Edificio Rojo, segunda planta), o en los Puntos de Información Universitaria accediendo con el carné universitario.

# 3. Definiciones| funcionales vs no funcionales

| | Pruebas funcionales | Pruebas no funcionales |
|---|---|---|
| Definición | Evaluar que el sistema haga lo que se supone que debe hacer | Evaluar cómo funciona el sistema |
| Objetivo | Optimizar la correcta implementación de los requisitos del sistema | Optimizar y observar aspectos de calidad del sistema |
| Qué evalúan | Funciones específicas | Características de calidad |
| Beneficios | Identificación temprana de fallos y verificación de requisitos | Mejorar de la experiencia del usuario, garantía de escalabilidad y seguridad del sistema |
| Cuándo se realizan | Fases iniciales y durante el desarrollo | Hacia el final del ciclo |

# 3. Definiciones | ¿para qué sirven las pruebas?

**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

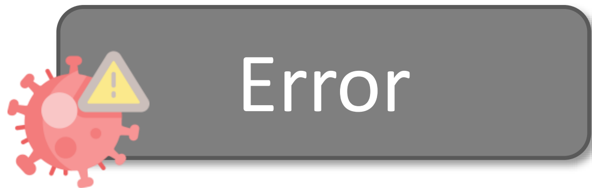
> *"Testing shows the presence, not the absence of bugs"*
>
> Dijkstra

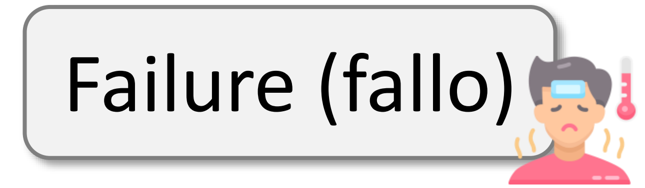
# 3. Definiciones |

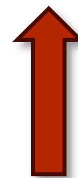
El estado interno de un programa que contiene un *bug*. Algunos estados no revelan un fallo.

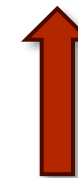
La causa que provoca el fallo. Se podría definir como "la causa".

La incapacidad de un sistema para realizar su función. Se podría definir como "los síntomas".







Evaluar el software observando su ejecución.

El proceso de encontrar un bug dado un fallo.

# 3. Definiciones | ¿cómo se observa un fallo?

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

**Accesibilidad**
(Reachability)

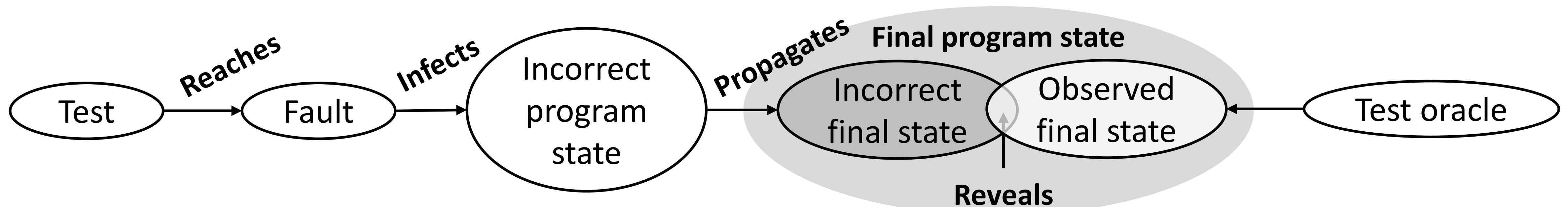
El lugar/lugares donde se encuentra el fallo deben ser accedidos.

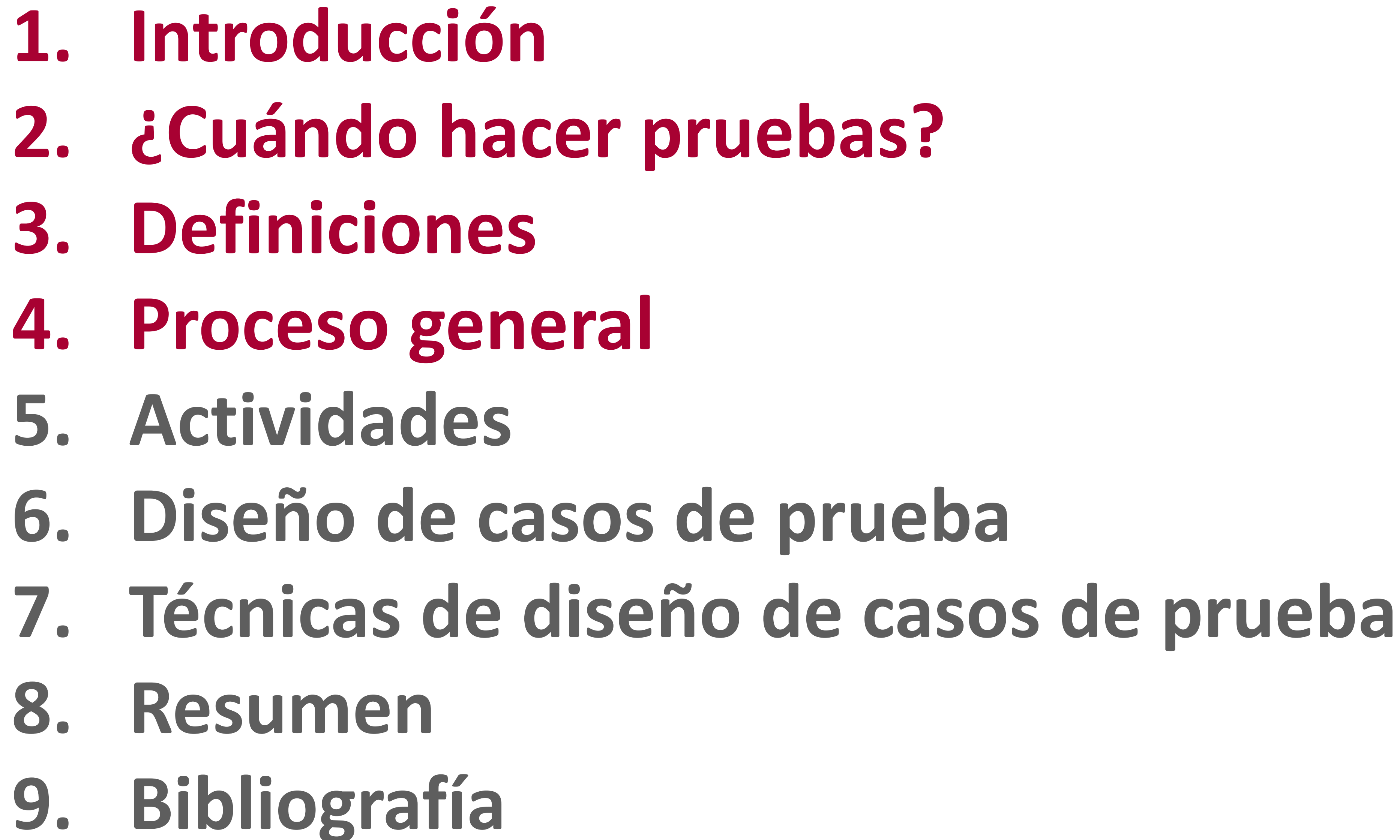
**Infección**
(Infection)

Una vez accedido, el estado del programa debe ser incorrecto.

**Propagación**
(Propagation)

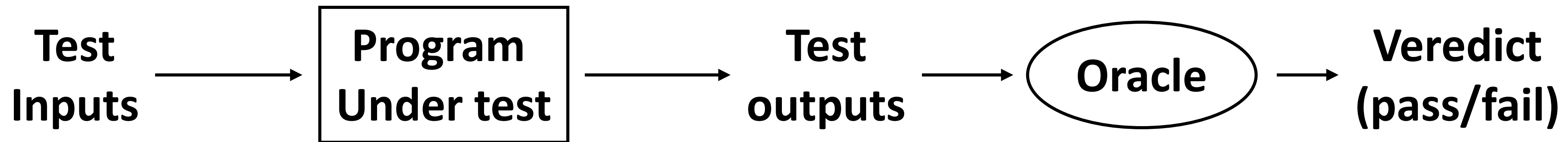
El estado infectado debe propagarse para poder observar su estado incorrecto



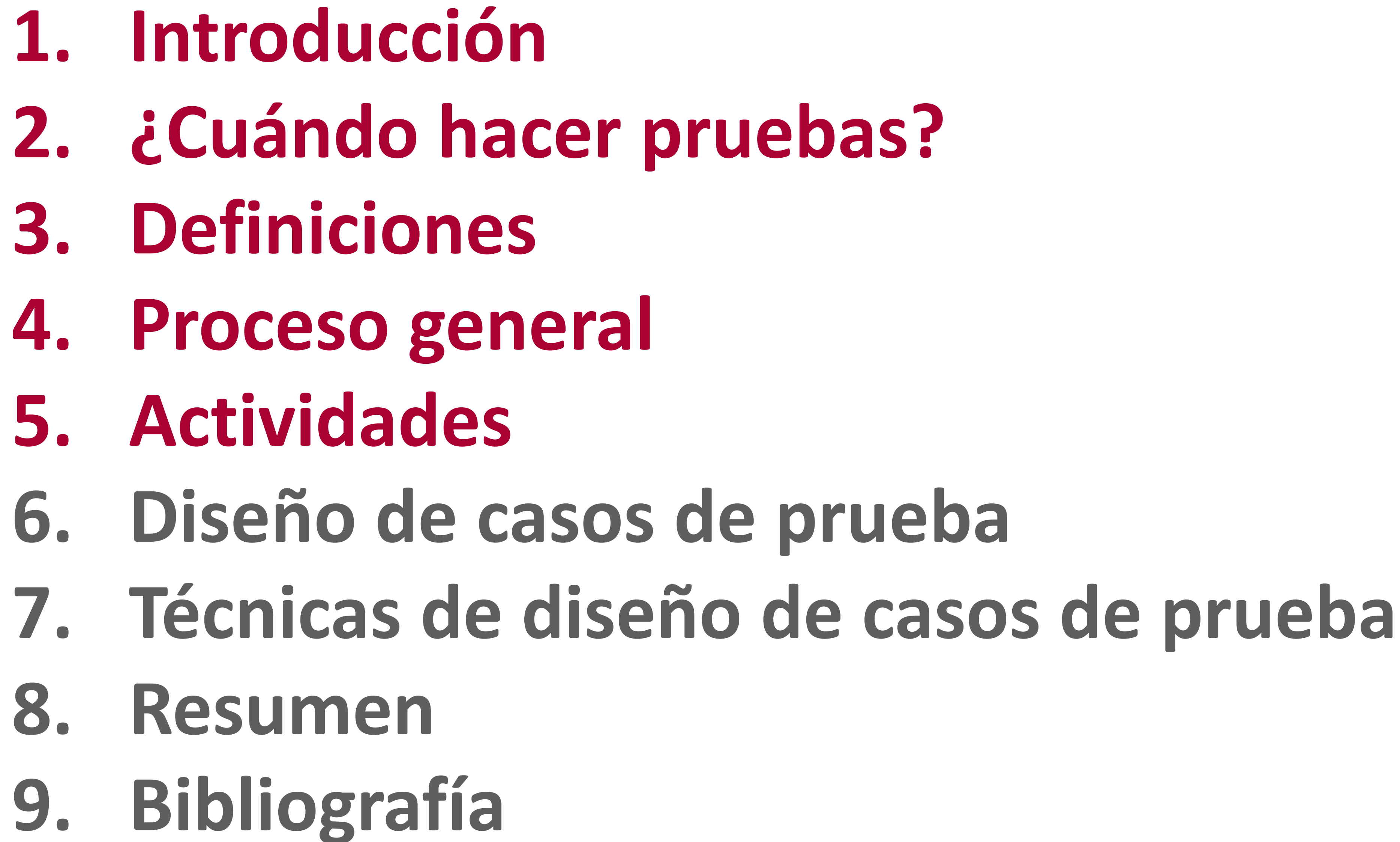


1. Introducción
2. ¿Cuándo hacer pruebas?
3. Definiciones
4. Proceso general
5. Actividades
6. Diseño de casos de prueba
7. Técnicas de diseño de casos de prueba
8. Resumen
9. Bibliografía

# 4. Proceso general de prueba |

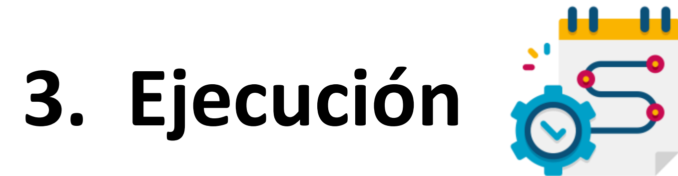


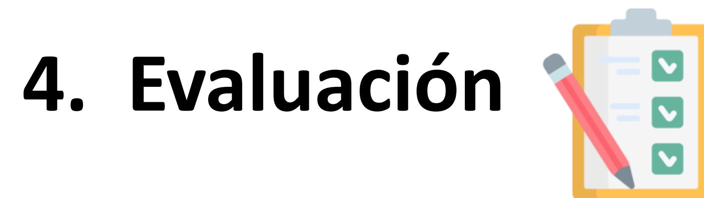

- Un caso de prueba se suele documentar identificando los siguientes puntos (*IEEE Standard for Software Testing Documentation*):
  - **Identificador** del caso de prueba
  - **Entradas**
  - **Salidas** esperadas
  - **Dependencias** (si es necesario ejecutar antes otros casos de prueba)


1. Introducción
2. ¿Cuándo hacer pruebas?
3. Definiciones
4. Proceso general
5. Actividades
6. Diseño de casos de prueba
7. Técnicas de diseño de casos de prueba
8. Resumen
9. Bibliografía

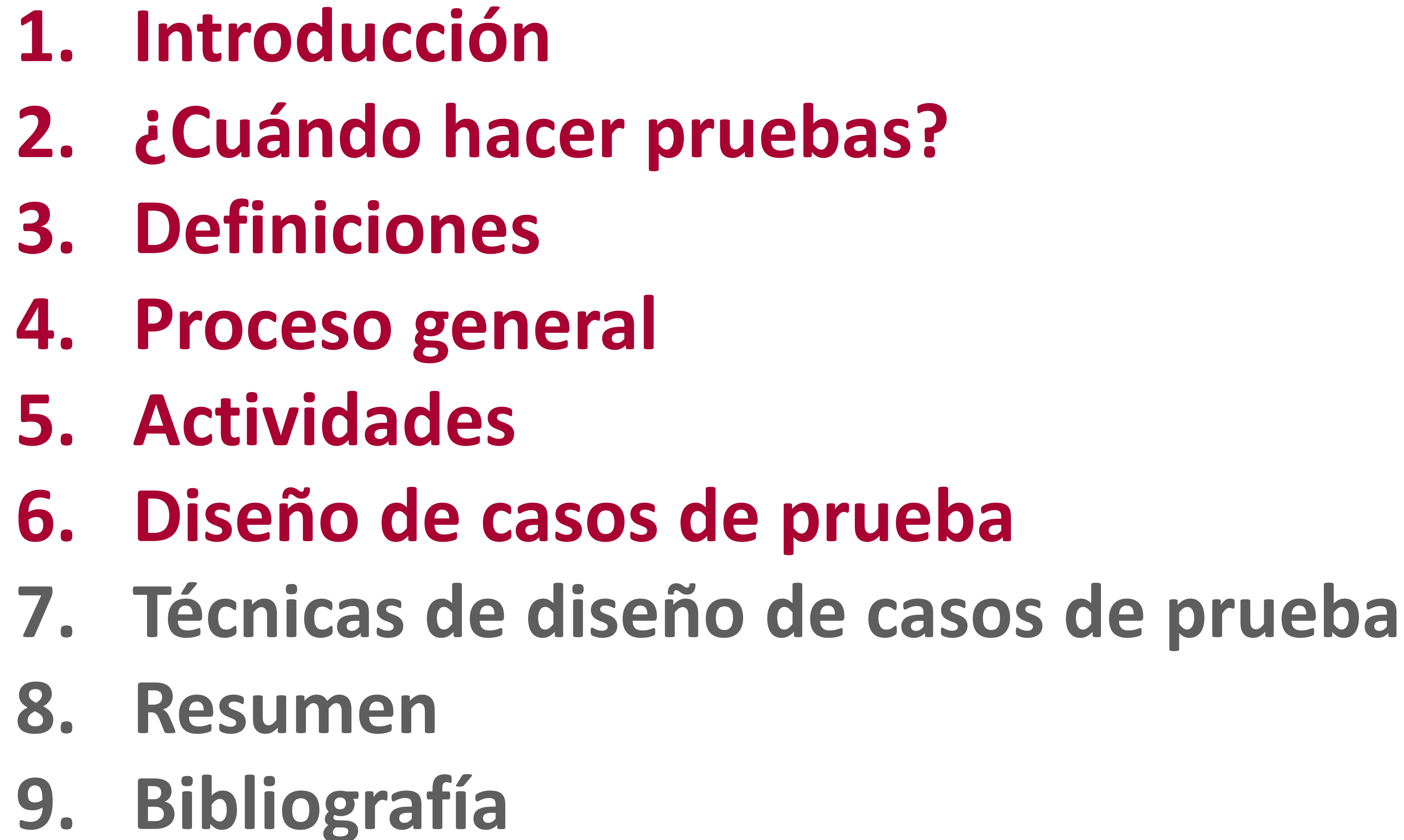
# 5. Actividades|

- Se pueden distinguir 4 tipos de actividades




- 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).


1. Introducción
2. ¿Cuándo hacer pruebas?
3. Definiciones
4. Proceso general
5. Actividades
6. Diseño de casos de prueba
7. Técnicas de diseño de casos de prueba
8. Resumen
9. Bibliografía

# 6. 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*).

# 6. Diseño de casos de prueba | pruebas unitarias

- Diseñadas para ejercitar una parte **pequeña y específica** de funcionalidad.
- Ayudan a **acotar** el ámbito de un fallo.
- Prueban que una pequeña parte **aislada** del software es correcta.
- No debe ir más allá de sus **límites**, aunque a veces no se pueden evitar las dependencias

# 6. Diseño de casos de prueba | pruebas de módulo

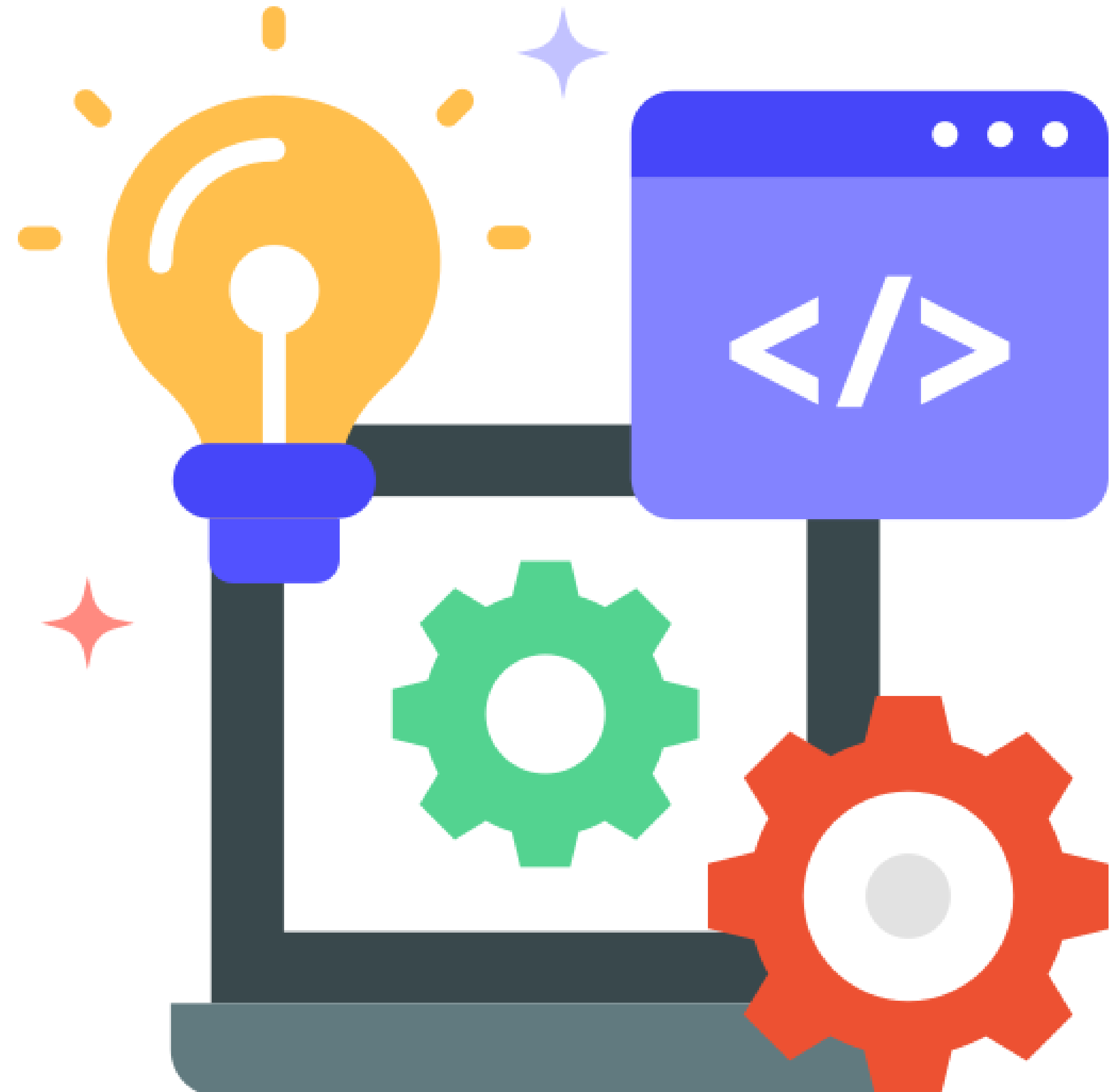
- Evaluar **unidades mayores** que en pruebas unitarias, como funciones o clases relacionadas.
- Validan cómo los **componentes** interactúan dentro de un módulo.
- Aíslan y prueban **funcionalidades específicas**, permitiendo comprobar que la integración de varios componentes sigue siendo correcta.

# 6. Diseño de casos de prueba | pruebas de integración

- Observar si los módulos o componentes funcionan bien cuando se **combinan**.
- Interacciones **internas** (entre módulos) o **externas** (con servicios).
- Detectan fallos en la **interfaz de comunicación** entre componentes.

# 6. Diseño de casos de prueba | pruebas de sistema

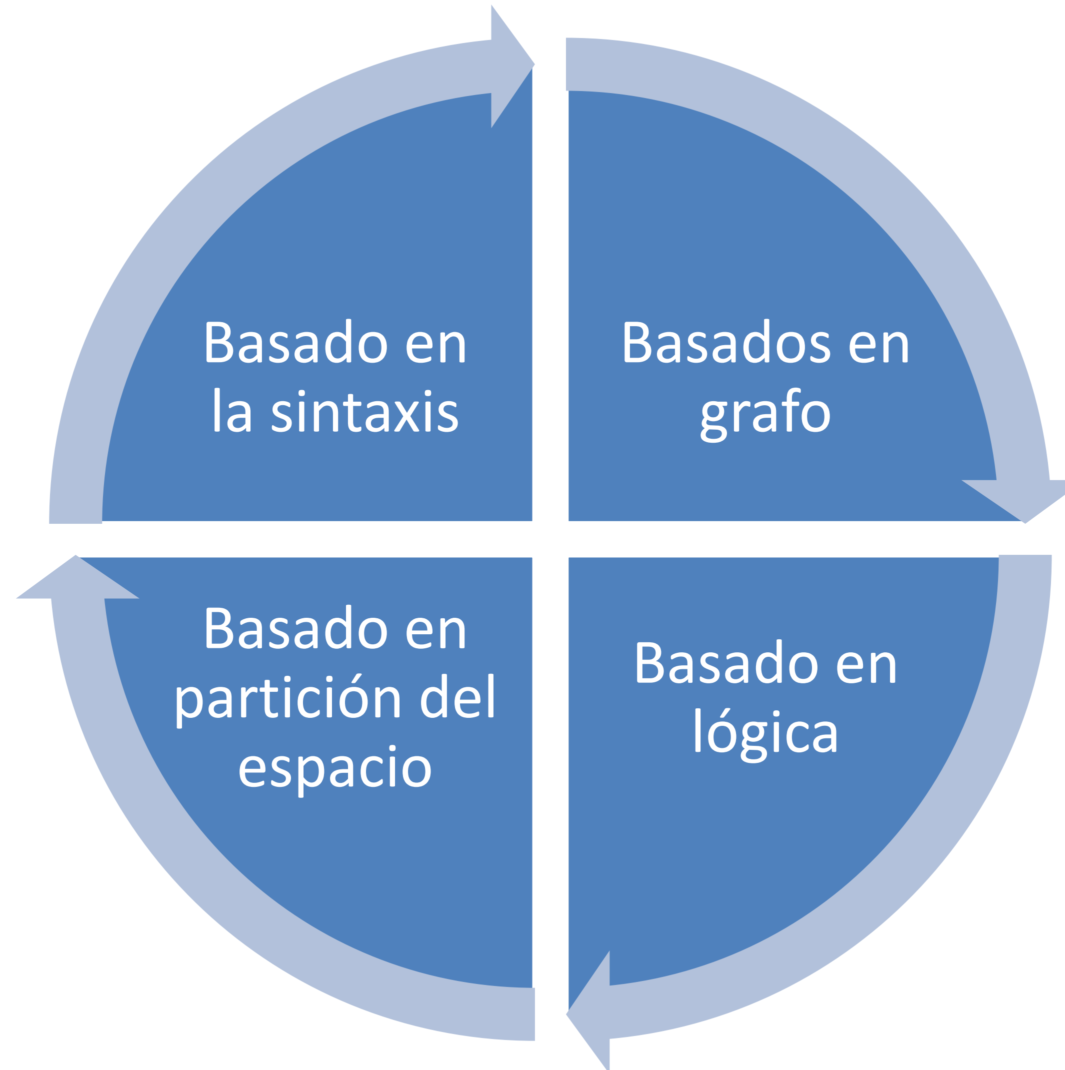
- Validar que el sistema completo cumple con los **requisitos** funcionales y no funcionales.
- Abarcan **todos** los componentes y su **interacción** en el entorno real.
- Detectan errores de **arquitectura** y observan si el sistema completo se comporta como se espera.



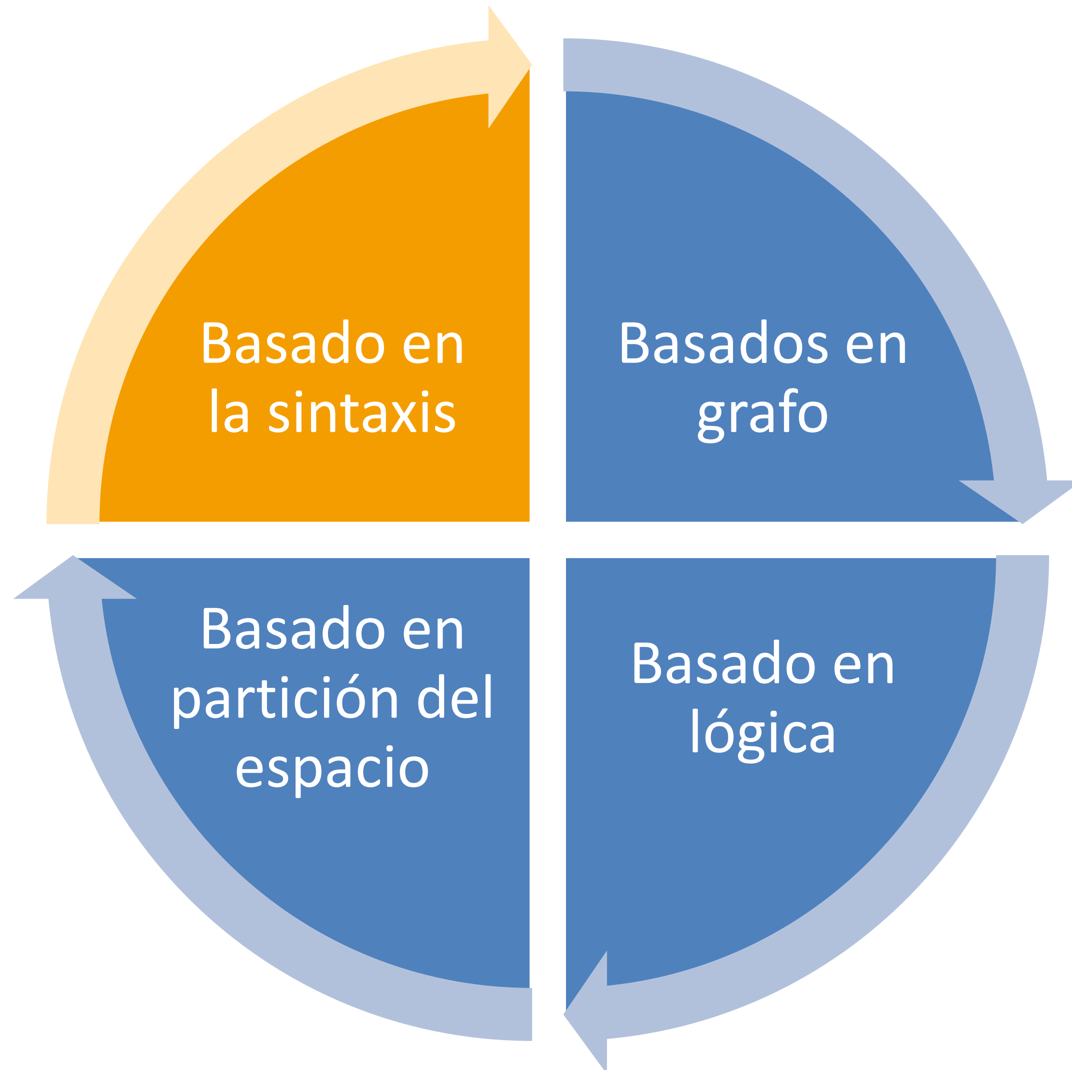
# 6. Diseño de casos de prueba | pruebas de aceptación

- Confirmar que el software cumple con los **criterios de aceptación** definidos por el cliente.
- Abarcan todo el sistema desde el punto de vista del **usuario final**.
- Son el acuerdo por el que que el sistema es **apto para su entrega y uso**, centrándose en la satisfacción del usuario.

# 6. Diseño de casos de prueba | criterios de cobertura



# 6. Diseño de casos de prueba | criterios de cobertura

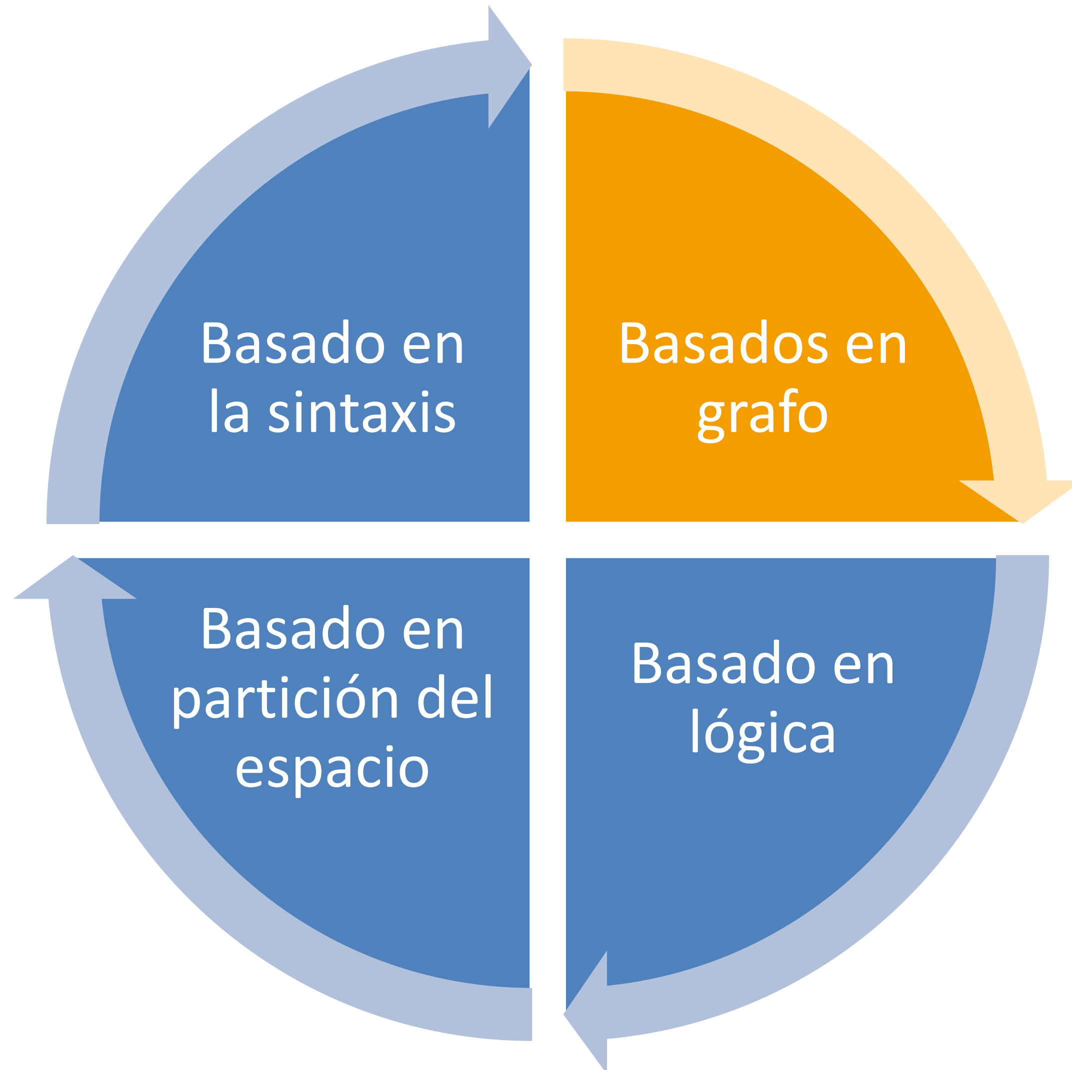



**Descripción**: se centra en validar la correcta ejecución del código en base a su estructura sintáctica, intentando maximizar el número de líneas, condiciones y decisiones del código sean evaluadas.

**Ventajas**: maximiza las decisiones y condiciones se evalúen completamente.

**Desventajas**: puede ser insuficiente si no se incluyen suficientes datos que prueben la funcionalidad lógica.

# 6. Diseño de casos de prueba | criterios de cobertura

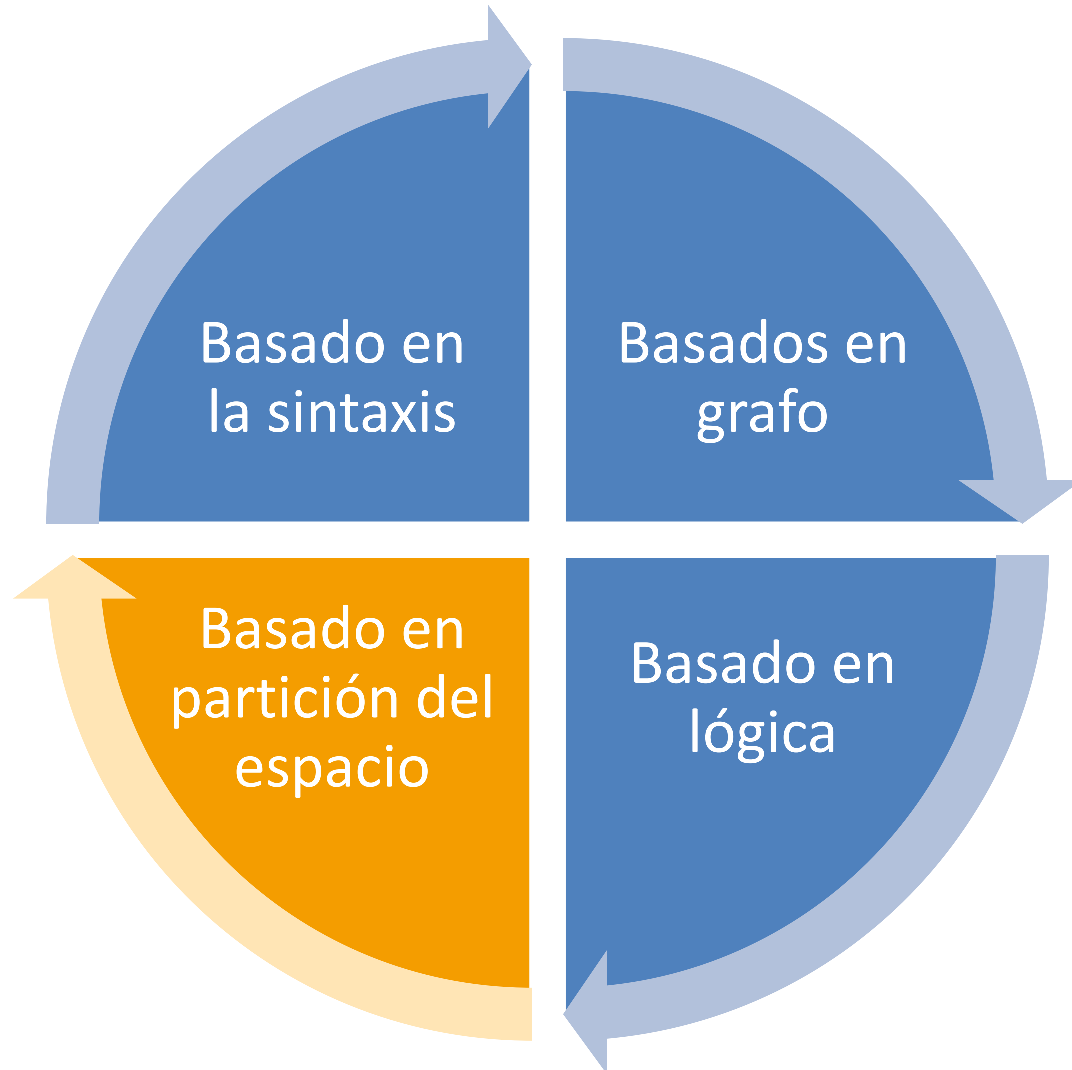



**Descripción**: representa el programa como un grafo, donde los nodos son estados y las aristas son transiciones, y se busca recorrer todas las aristas o caminos.

**Ventajas**: ayuda a detectar errores en la lógica del flujo de control y en la interacción de diferentes componentes.

**Desventajas**: puede ser complejo debido al gran número de combinaciones posibles en sistemas grandes.

# 6. Diseño de casos de prueba | criterios de cobertura

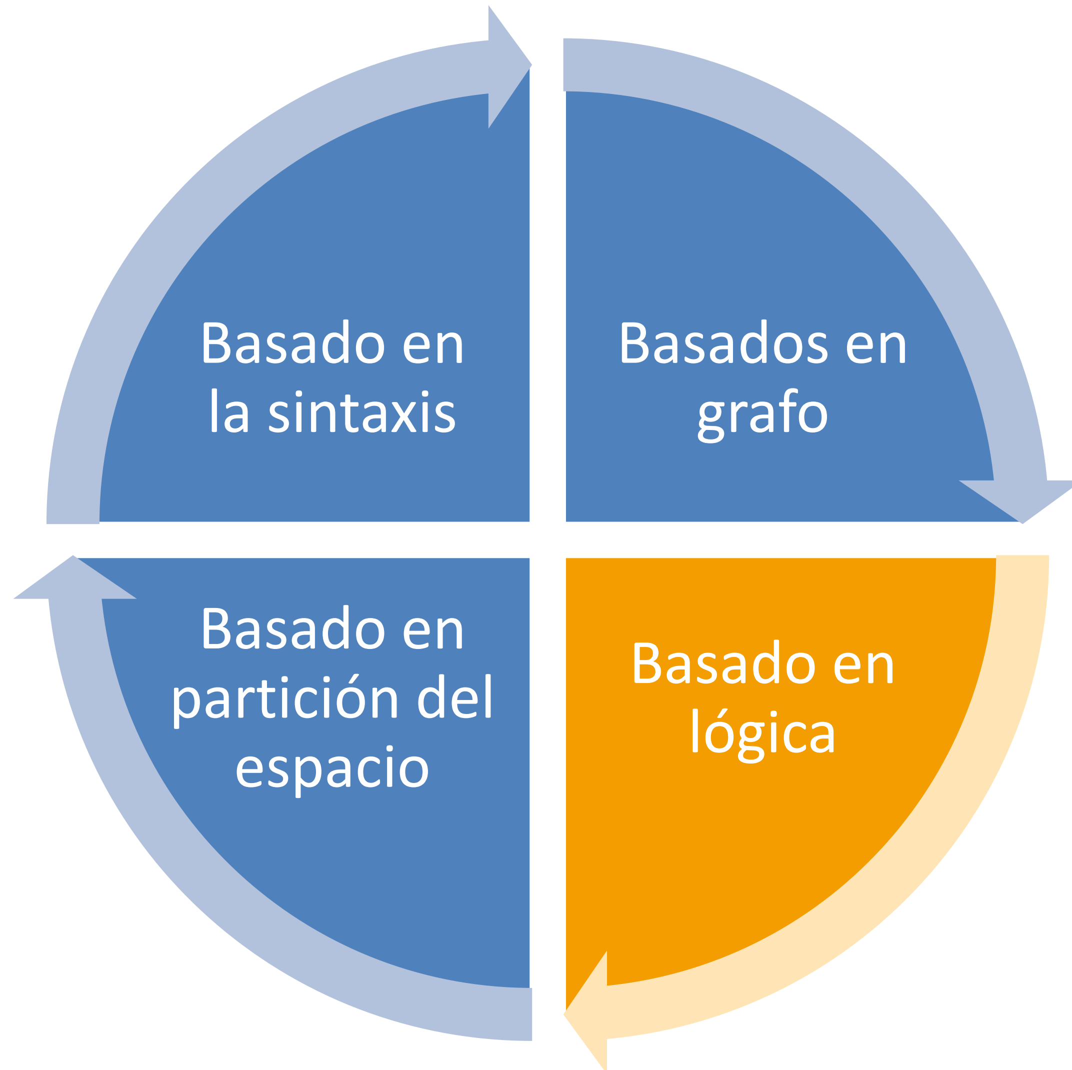



**Descripción**: se basa en dividir el espacio de entrada en clases de equivalencia y casos límite, para representar condiciones similares y casos extremos.

**Ventajas**: minimiza la cantidad de pruebas sin sacrificar la cobertura, al concentrarse en clases representativas.

**Desventajas**: podría no detectar errores que ocurren entre los límites de clases.

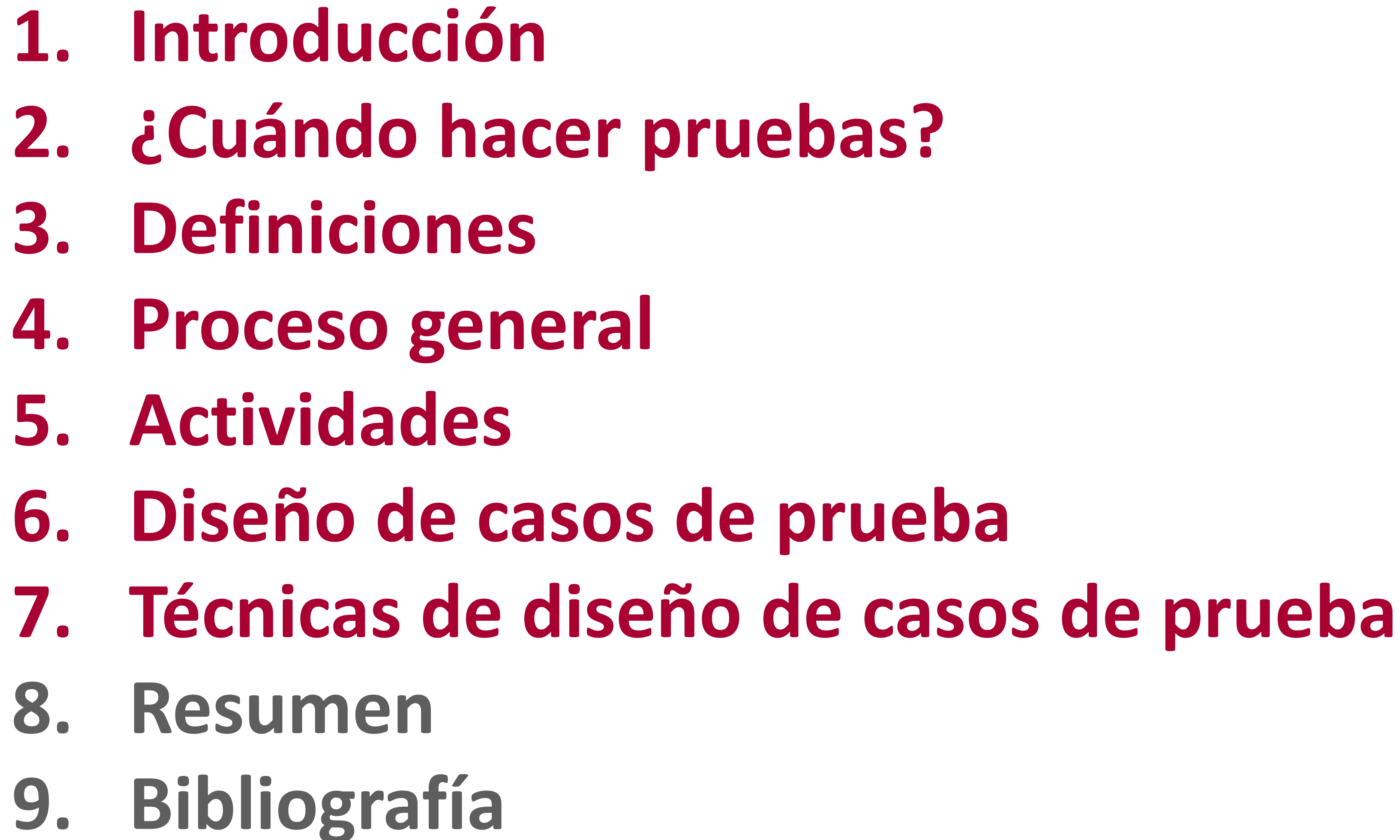
# 6. Diseño de casos de prueba | criterios de cobertura




**Descripción**: evalúa las decisiones y las combinaciones lógicas en el software, cubriendo todas las posibles combinaciones de condiciones lógicas y sus resultados.

**Ventajas**: muy efectivo para encontrar errores en la lógica de control y en casos complejos con múltiples condiciones.

**Desventajas**: puede resultar en un gran número de casos de prueba, especialmente con muchas condiciones, aumentando el esfuerzo de pruebas.



1. Introducción
2. ¿Cuándo hacer pruebas?
3. Definiciones
4. Proceso general
5. Actividades
6. Diseño de casos de prueba
7. Técnicas de diseño de casos de prueba
8. Resumen
9. Bibliografía

# 7. Técnicas de diseño de casos de prueba |particiones equivalentes

- Se divide el espacio de prueba en clases equivalentes
- Las particiones deben ser disjuntas
- Inconveniente: rápida explosión

# 7. Técnicas de diseño de casos de prueba | valores límite

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

# 7. Técnicas de diseño de casos de prueba | particiones equivalentes y valores límite

## Ejemplo

El precio de la matrícula depende de la edad del cliente. Hay tres rangos: menores de 18 años, entre 18 y 65 años y mayores de 65 años.

Entonces, hay 3 **clases equivalentes**: **0-17, 18-64, 65-***

Usamos los **valores límite**: **16,17,18,19,64,65,66**

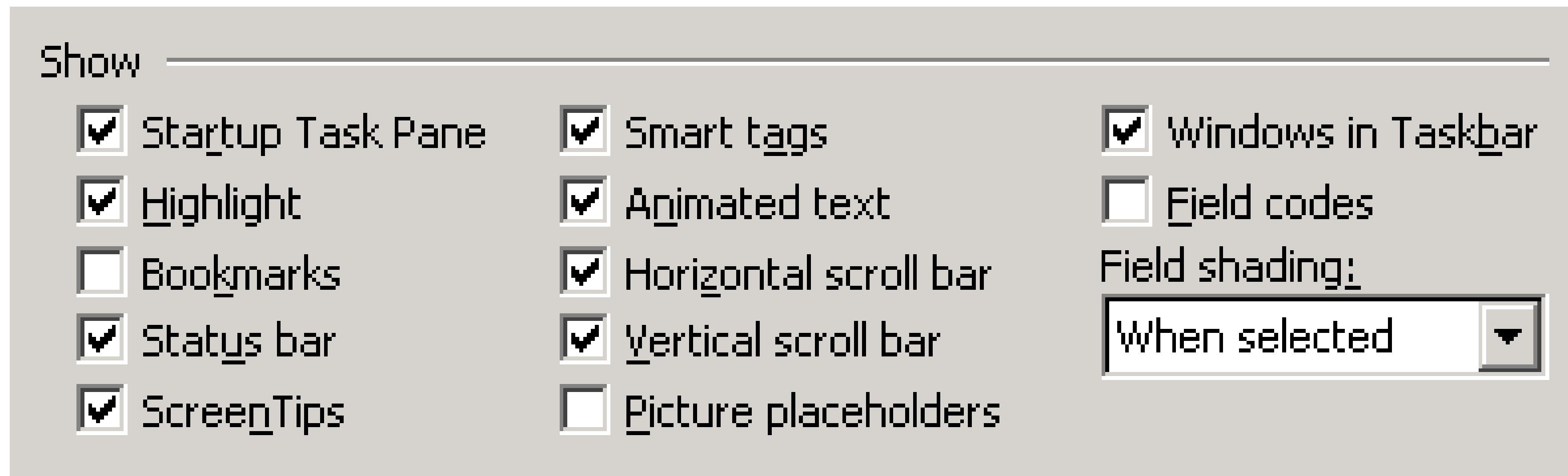
O simplificado (el límite y los adyacentes): 17,18,64,65

# 7. Técnicas de diseño de casos de prueba | combinaciones pair-wise, n-wise

Método **combinatorio** que propone hacer pruebas sobre todas las posibles combinaciones de dos parámetros de entrada.

**Para un sistema que tiene n entradas binarias, el número total de combinaciones posibles de esas entradas es $2^n$**

# 7. Técnicas de diseño de casos de prueba | combinaciones pair-wise, n-wise




$2^{12}$ x 3 = ¡¡ 12.288 combinaciones !!

# 7. Técnicas de diseño de casos de prueba | combinaciones pair-wise, n-wise

Imagina X, Y, Z booleanos:

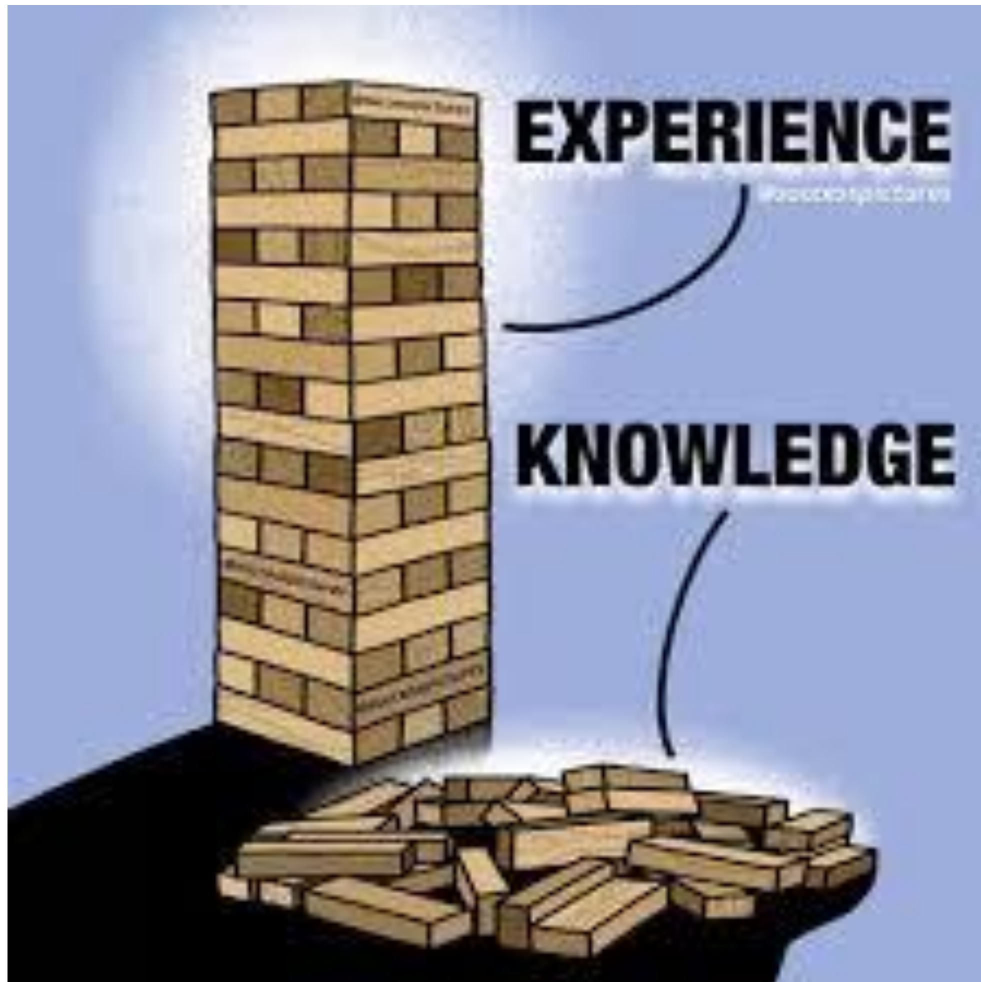
**Todas las posibles pruebas**

| | 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 pair wise**

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

# 7. Técnicas de diseño de casos de prueba | errores conocidos

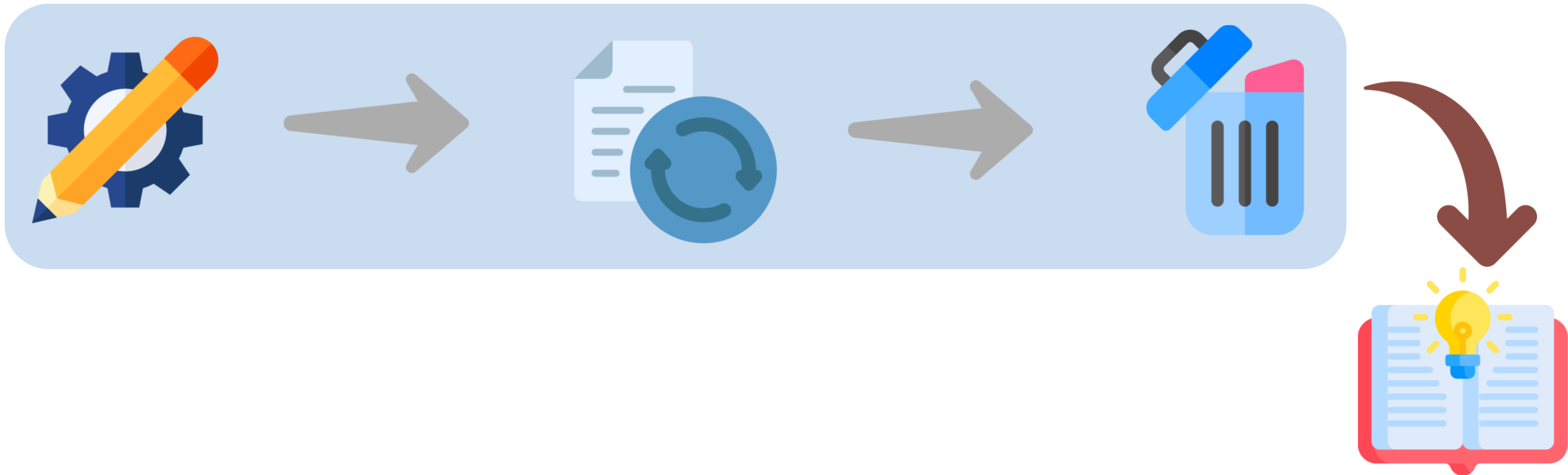



Se basa en la **experiencia e intuición** del *tester* para identificar errores en el sistema.

En lugar de seguir un conjunto estructurado de casos de prueba, el *tester* se anticipa en detectar **áreas donde es probable que existan defectos**.

# 7. Técnicas de diseño de casos de prueba | cobertura CRUD

Para cada elemento persistente, se prueban todas sus **operaciones CRUD**.

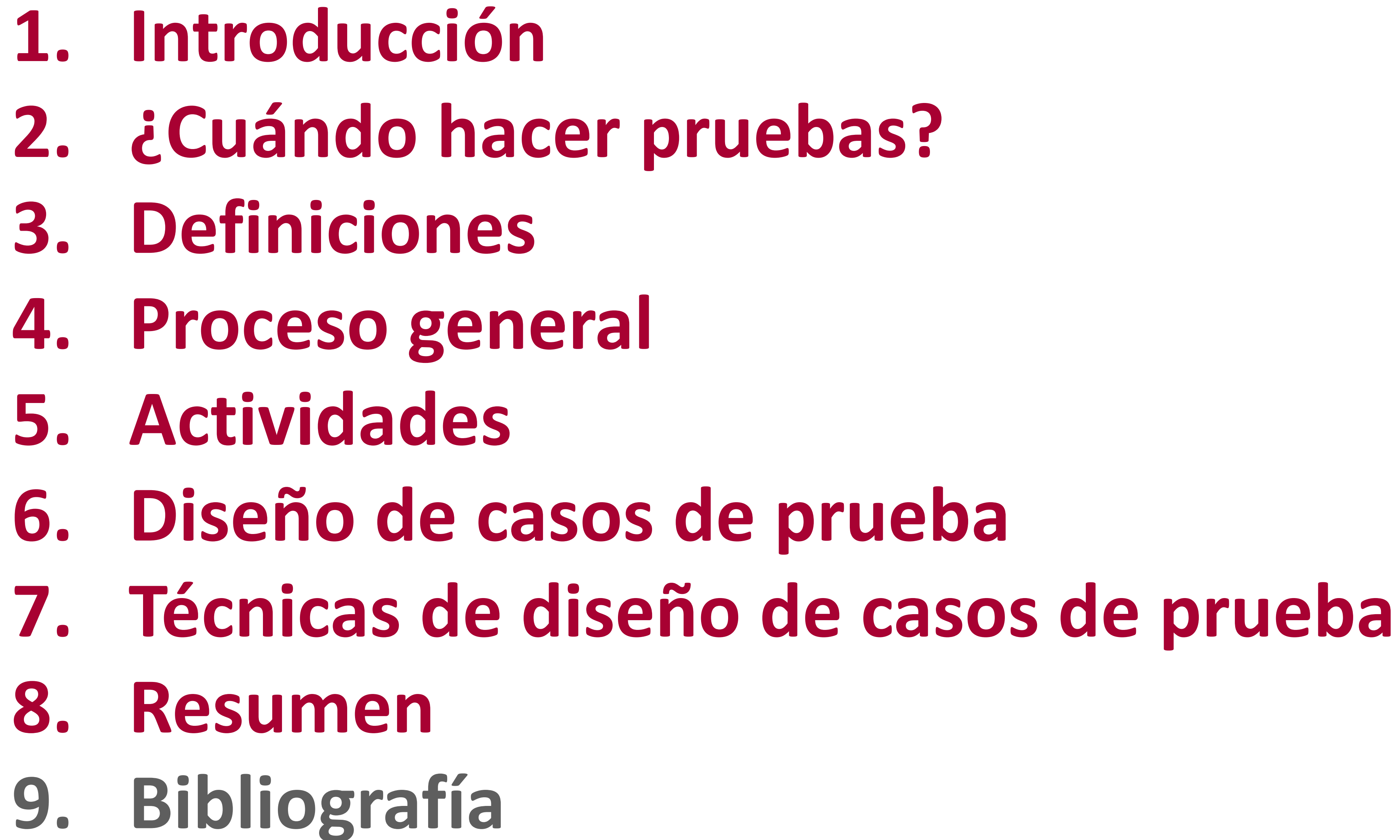


# 7. Técnicas de diseño de casos de prueba | cobertura CRUD

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

**Ejemplo:**

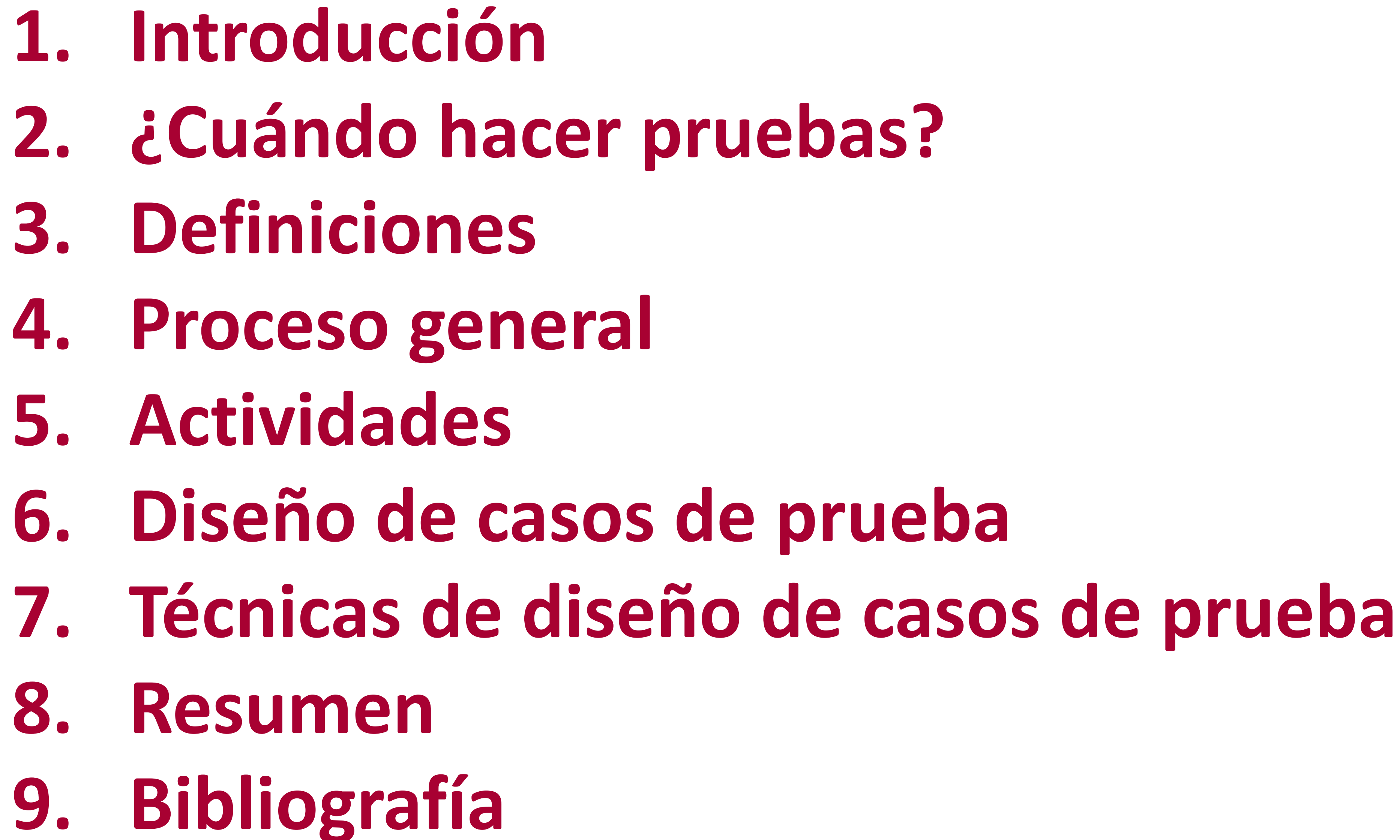
- Crear *DataSet (C)*
- Verificar que se ha creado(R)
- Actualizar sus valores (U)
- Verificar que se han actualizado (R)
- Borrar *DataSet* (D)
- Verificar que se ha borrado (R)


1. Introducción
2. ¿Cuándo hacer pruebas?
3. Definiciones
4. Proceso general
5. Actividades
6. Diseño de casos de prueba
7. Técnicas de diseño de casos de prueba
8. Resumen
9. Bibliografía

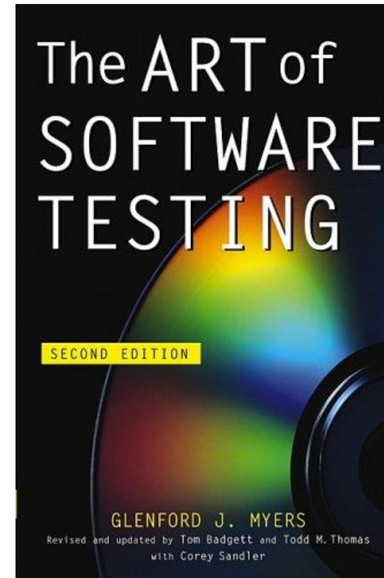
# 8. Resumen|

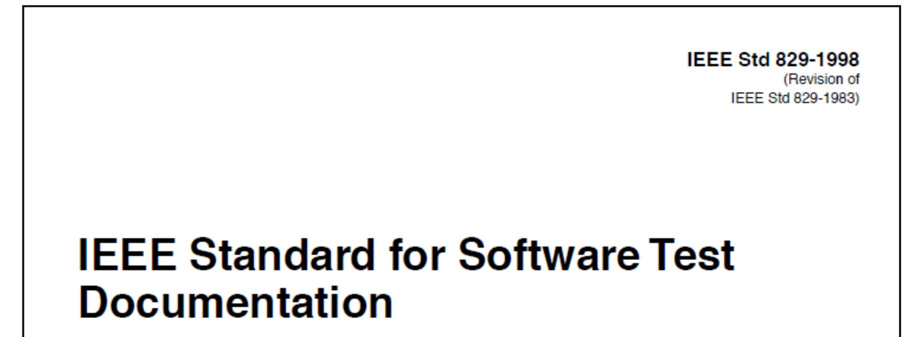
- **¿Qué hemos aprendido?**
  - Hemos afianzado nuestra noción de que las pruebas de software son importantes
  - Diferencia entre ejecución y diseño de pruebas
  - Alcance de las pruebas unitarias
  - Técnicas para el diseño de casos de prueba: particiones equivalentes, valores límite, n-wise testing, 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

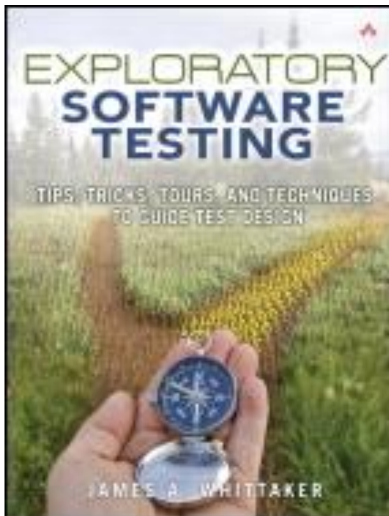

1. Introducción
2. ¿Cuándo hacer pruebas?
3. Definiciones
4. Proceso general
5. Actividades
6. Diseño de casos de prueba
7. Técnicas de diseño de casos de prueba
8. Resumen
9. Bibliografía

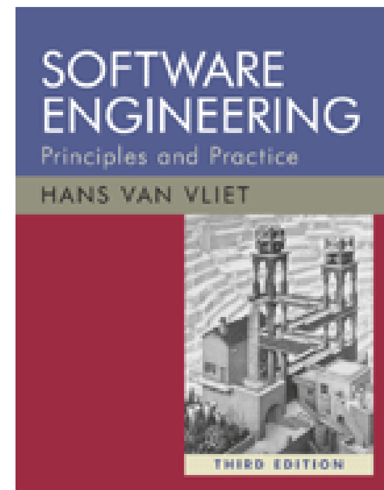
# 9. Bibliografía |

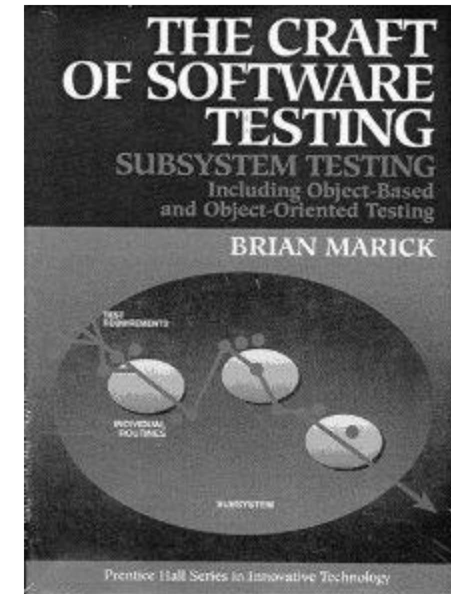


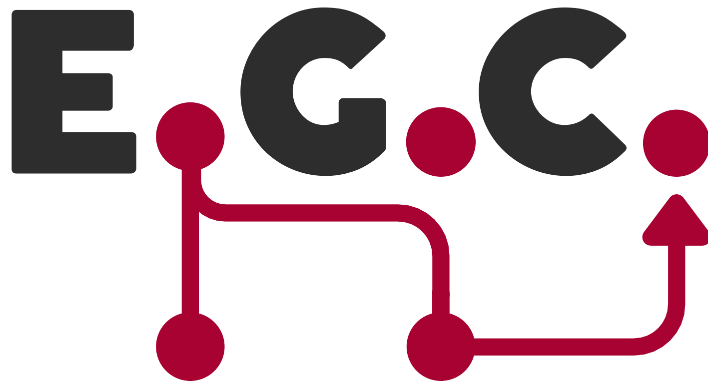


IEEE Std 829-1998
(Revision of
IEEE Std 829-1983)

IEEE Standard for Software Test Documentation






Grado en Ingeniería Informática - Ingeniería del Software

# Evolución y Gestión de la Configuración

Escuela Técnica Superior de
Ingeniería Informática

## ¡Gracias!