

Verificación del Software

## 3. Revisión y pruebas

# Verificación del Software

Tema 1. Fundamentos de Calidad del Software

Tema 2. QA en el SDLC

**Tema 3. Revisión y pruebas**

Tema 4. Técnicas y herramientas

Tema 5. Gestión de las pruebas

# Revisión y pruebas

- 3.1. Tipos de pruebas
- 3.2. Pruebas de caja blanca
- 3.3. Análisis estático de código
- 3.4. Revisión
- 3.5. Pruebas de caja negra
- 3.6. Pruebas basadas en experiencia

# Introducción

En este módulo vamos a revisar una serie de técnicas de prueba que permiten analizar el software desde varios puntos de vista.

Estas pruebas son complementarias entre sí, y una buena estrategia de prueba consistirá en seleccionar el mix más adecuado.

Todas ellas pueden aplicarse en distintos modelos de SDLC y a la hora de implementarlas cubren toda la gama: desde técnicas que sólo pueden realizarse manualmente, a otras que sirven de base para la automatización del testing.

**Verificación del Software**

## **3.1 Tipos de pruebas**

# Tipos de pruebas

## Se fijan en el propósito del test

Los niveles definen cuando se realiza cada tipo de test, los tipos determinan qué se quiere probar:

- Test de caja blanca: se fijan en cómo está hecho el código.
    - ¿Hay ramas innecesarias o redundantes? ¿Se prueban todas?
  - Test de caja negra: se fijan en cómo funciona el código
    - A esta categoría pertenecen los funcionales y no funcionales
  - Test funcionales: se fijan en el comportamiento del software.
    - En un buscador ¿Encuentra los elementos de acuerdo con la cadena búsqueda?
  - Test no funcionales: se fijan en requisitos no funcionales (a veces llamado implícitos).
    - En un buscador ¿Los encuentra en un tiempo aceptable? ¿Soporta múltiples búsquedas simultáneamente?
  - Tests de cambios (*change-related testing*): se fijan en circunstancias en las que haya cambiado el código. Por ejemplo, asegurar que arreglar un defecto no introduce otros.
- Un nivel o actividad de testing puede contener varios de estos tipos en función de lo que se quiera probar.

# Tipos de pruebas: funcionales

## Cumplimiento de los requisitos

Se trata de comprobar la fidelidad entre las especificaciones y el producto (requisitos, historias de usuario, ...). Obviamente, esas especificaciones tienen que estar documentadas debidamente: no es posible evaluar en base a requisitos “orales” o informales.

Es conveniente una visión suficientemente independiente (por ejemplo, con test de caja negra) y al mismo tiempo conocimiento del dominio de negocio del software y del contexto en el que se entrega.

Deben valorar si los requisitos su cumplen fielmente, completamente (toda la funcionalidad definida) y de una forma apropiada.

Una métrica importante es la cobertura, como porcentaje de funcionalidad que ha sido probada.

# Tipos de pruebas: no-funcionales

## Dimensiones implícitas del producto o servicio

Con ellos se busca determinar el cumplimiento de características transversales (según el modelo de calidad ISO/IEC 25010):

- Usabilidad: facilidad de uso, efectividad, eficiencia y satisfacción.
- Rendimiento (performance): como el uso adecuado y eficiente de los recursos de computación, almacenaje, conectividad.
- Fiabilidad: hasta qué punto los resultados entregados se acercan a los esperados.
- Seguridad: grado de protección frente accesos o usos no autorizados.
- Mantenibilidad: facilidad para modificar o corregir el sistema.
- Portabilidad: hasta que punto puede transferirse y usarse en otro entorno o plataforma.
- Compatibilidad: capacidad para intercambiar información con otros sistemas o componentes.
- Idoneidad funcional (*Functional Suitability*): grado en el que se cumplen necesidades funcionales implícitas.

Pueden formar parte de los requisitos y evaluarse en cualquier momento. Pueden necesitar técnicas y herramientas muy especializadas.



# Tipos de pruebas: caja blanca

## Corrección de la construcción del código

Se fijan en si el código, su estructura o arquitectura son correctos.

Eso supone revisar la implementación examinando el código.

También hay métricas de cobertura, pero aquí se fijan si se prueban los distintos elementos.

Requiere conocimientos especializados sobre cómo se construye el software y otros elementos auxiliares. Normalmente cuenta con el apoyo de herramientas de análisis.

# Tipos de pruebas: caja negra

## Cumplimiento de requisitos sin conocer cómo está hecho el software

Son un tipo especial de pruebas funcionales que se fijan en si el comportamiento del código en escenarios simulados o reales es correcto sin conocer cómo está hecho.

Para ello, el punto de partida son los requisitos que definen el software y su comportamiento.

Las métricas de cobertura se refieren al porcentaje de tests ejecutados frente a los totales, y a la validación de los distintos requisitos, por eso es tan importante la trazabilidad.

Como curiosidad, hay unas pruebas “de caja gris” (*gray box testing*), que combinan las dos perspectivas: caja negra y caja blanca. Es decir, pruebas funcionales a partir de un conocimiento limitado de cómo está hecho el software.

# Tipos de pruebas: cambios

## Impacto de los cambios en el software

Son pruebas que se lanzan para analizar el efecto en el código de correcciones o cambios. Esas dos situaciones dan lugar a dos tipos de tests:

- De confirmación:
  - Para comprobar que se ha arreglado un defecto. Normalmente ejecutando los tests que fallaban debidos al defecto.
  - Por ejemplo un sanity test.
- De regresión
  - Para asegurarse de que un cambio o corrección no ha afectado a otras partes del software o su comportamiento.
  - Deberían lanzarse ante cada cambio, por eso merece la pena automatizarlos, al menos en gran parte ya que se ejecutan con frecuencia y tienden a permanecer estables.
  - Tienen especial aplicación en el paradigma ágil y contribuyen decisivamente a incrementar la confianza ante cambios, lo que incluye la refactorización.
  - Puede haber un smoke test previo a una regresión general.

# Ejercicio: tipos de pruebas

## Encaja los siguientes tests en el nivel y tipo que corresponda

Tipos:

- Funcional
- No-funcional
- Caja blanca
- Caja negra

Niveles:

- De componente
- De integración
- De sistema
- De aceptación

Test:

1. Probar el tratamiento de IBAN en la función de entrada de cuentas.
2. Validar que el sistema es sencillo de usar.
3. Comprobar la cobertura del código de todas las opciones del menú y de la navegación del sistema.
4. Comprobar como de rápida es la transferencia de datos entre componentes.

# Ejercicio: tipos de pruebas *SOLUCIÓN*

## Encaja los siguientes tests en el nivel y tipo que corresponda

Test:

1. Probar el tratamiento de IBAN en la función de entrada de cuentas.
  - Componente / funcional / caja negra/blanca
2. Validar que el sistema es sencillo de usar.
  - Aceptación / no funcional / normalmente caja negra
3. Comprobar la cobertura del código de todas las opciones del menú y de la navegación del sistema.
  - Sistema / normalmente caja blanca
4. Comprobar como de rápida es la transferencia de datos entre componentes.
  - Integración / no funcional / caja negra/blanca

**Verificación del Software**

## **3.2 Pruebas de caja blanca**

# Pruebas de caja blanca

También llamados “estructurales” o de “caja de cristal”.

En este caso miramos “dentro” del objeto de prueba (SUT), lo que quiere decir que nos interesamos por cómo está hecho. Las pruebas buscan cubrir todos los elementos y ramas del código.

Los tests de caja blanca necesitan contar de antemano con la arquitectura del software, el diseño detallado y el propio código.

Manualmente o con la ayuda de herramientas se analizan los elementos estructurales para identificar módulos, ramas, acciones, o interfaces, y asegurar que se validan todos ellos.

La cobertura en los tests de caja blanca se mide como el porcentaje de elementos que han sido activados durante la prueba.

Los resultados esperados se extraen de los requisitos del producto.

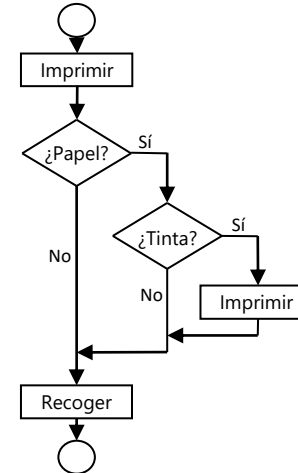
Aplican a todos los niveles, en componente validan instrucciones; en integración las llamadas e interfaces; en sistemas cada uno de elementos de menú, páginas, opciones, funciones, ...

# Pruebas de caja blanca (2)

## De instrucciones

Este tipo se fija en los elementos ejecutables. La cobertura es el porcentaje de instrucciones ejecutadas, lo que significa no incluir elementos como declaraciones, nombres de funciones, finales de bucles, ... que no “hacen nada”.

```
Mandar fichero a impresora
If hay papel
    If tiene tinta
        Imprimir
    End If
End If
Recoger papel
```



Hay que determinar cuántos casos de prueba son necesarios para probar todas las instrucciones



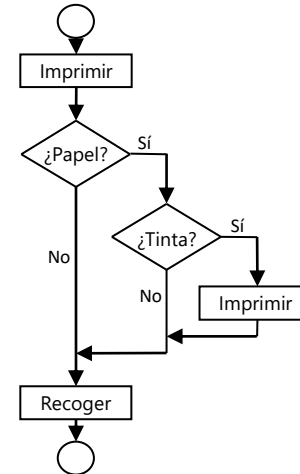
# Pruebas de caja blanca (2)

## De instrucciones

Este tipo se fija en los elementos ejecutables. La cobertura es el porcentaje de instrucciones ejecutadas, lo que significa no incluir elementos como declaraciones, nombres de funciones, finales de bucles, ... que no “hacen nada”.

```
Mandar fichero a impresora
If hay papel
    If tiene tinta
        Imprimir
    End If
End If
Recoger papel
```

Hay que determinar cuántos casos de prueba son necesarios para probar todas las instrucciones



En realidad basta con una única prueba (si hay papel y tinta se ejecutan todas las instrucciones).

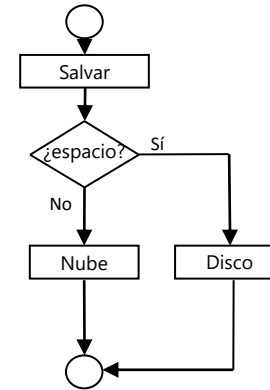
# Pruebas de caja blanca (3)

## De decisiones

Cuando nos fijamos en las decisiones, la cobertura se expresa como el porcentaje de ramas de decisión que han sido validadas por los test cases.

```
Guardar fichero
If hay espacio en disco
    Guardar en disco
Else
    Guardar en nube
End If
```

Hay que determinar cuántos casos de prueba son necesarios para probar todas las decisiones



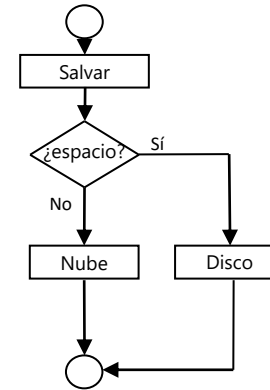
# Pruebas de caja blanca (3)

## De decisiones

Cuando nos fijamos en las **decisiones**, la cobertura se expresa como el porcentaje de ramas de decisión que han sido validadas por los test cases.

```
Guardar fichero
If hay espacio en disco
    Guardar en disco
Else
    Guardar en nube
End If
```

Hay que determinar cuántos casos de prueba son necesarios para probar todas las **decisiones**



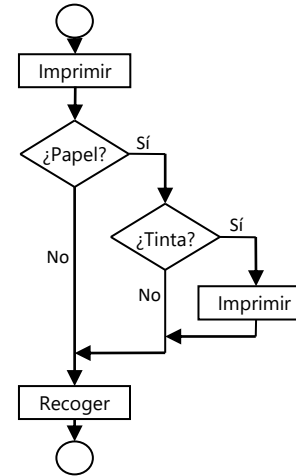
En este caso hacen falta dos pruebas para cubrir todos los casos

# Pruebas de caja blanca (4)

## De decisiones

```
Mandar fichero a impresora
If hay papel
    If tiene tinta
        Imprimir
    End If
End If
Recoger papel
```

Es el primer ejemplo. Ahora hay que determinar cuántos casos de prueba son necesarios para probar todas las **decisiones**

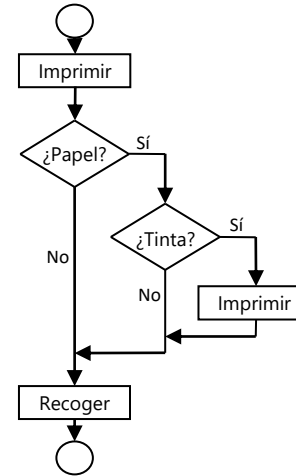


# Pruebas de caja blanca (4)

## De decisiones

```
Mandar fichero a impresora
If hay papel
    If tiene tinta
        Imprimir
    End If
End If
Recoger papel
```

Es el primer ejemplo. Ahora hay que determinar cuántos casos de prueba son necesarios para probar todas las **decisiones**



En este caso hacen falta **tres** pruebas para cubrir todos los casos

# Pruebas de caja blanca

## **Beneficios y uso**

El hecho de conocer el interior del código, permite tener también mayor precisión en la cobertura y no dejar código sin probar.

La prueba de decisión facilita la **cobertura**: un 100% de cobertura de decisiones supone un 100% de cobertura de instrucciones, cosa que al revés no sucede.

Las pruebas de caja blanca complementan a otras técnicas, no las sustituyen. De hecho, no garantizan que el funcionamiento sea correcto, sólo que el código se prueba íntegramente.

Son especialmente útiles cuando es muy importante la calidad, por ejemplo en aplicaciones críticas.

Conseguir la máxima cobertura en este tipo de pruebas tiene sentido especialmente en las pruebas de componente (unitarias). Al mismo tiempo ayuda a detectar código más diseñado o estructurado (ramas o instrucciones que no se ejecutan, por ejemplo).

## Verificación del Software

# 3.3 Análisis estático de código

# Análisis estático (1)

## Examen de los productos del proceso de construcción

Este tipo de prueba se basa en la revisión de los productos del proceso de construcción, lo que incluye documentos, diseño, arquitectura, especificaciones, manuales, casos de prueba y el propio código.

Este análisis puede ser manual o automático. Si es manual, entonces se realiza a través del proceso de revisiones formales que veremos en el siguiente apartado. Si es automático, normalmente se va a aplicar únicamente al código debido a las dificultades de tratar con lenguaje natural.

Así, el análisis estático de código, es la revisión **sin ejecutar** el código para detectar defectos en él. Este proceso puede ser manual, por ejemplo con una *peer review* o una revisión, o automatizado con el uso de herramientas específicas.

El análisis estático es un complemento del testing y además una prueba adicional de gran valor en software crítico, por ejemplo. También ayuda a encontrar potenciales problemas de seguridad, y con la ayuda de herramientas, tiene un papel muy relevante a la hora de aplicar mejora continua (*refactoring*).



# Análisis estático (2)

## ¿Qué puede analizar?

- Requisitos, especificaciones funcionales y no funcionales. Épicas, *features*, historias de usuario y sus criterios de aceptación. Aquí se puede aplicar el concepto de DoR o *Definition of Ready*.
- Diseños de arquitectura y del software.
- Documentación de usuario, de operación, manuales, herramientas de ayuda.
- Diseños de UX (wireframes, visuales, información de interacción).
- Planes, estrategias, directrices.
- Todos lo relacionado con el testing: planes de pruebas, paquetes de regresión, datasets, definiciones de casos de uso, procedimientos de prueba, tests automáticos, resultados.
- **Código**

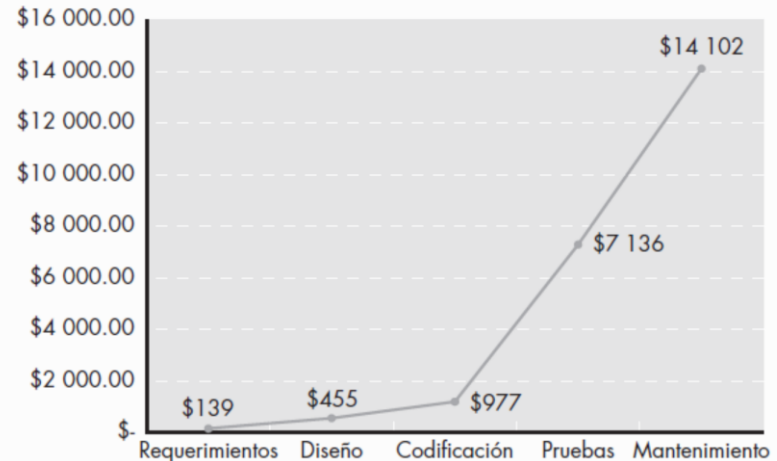
Todo aquello que pueda procesarse automática por tener una estructura formal y una gramática, puede analizarse con una herramienta, lo que incrementa la capacidad de análisis y reduce el esfuerzo necesario para llevarlo a cabo.

# Beneficios del análisis estático

## ¿Qué ganamos con el análisis estático?

Sobre todo se busca anticipación y reducción de costes.

Ya sabemos que cuanto más avanzamos en el ciclo de vida más costosa es la resolución de defectos. Resolver un problema generado en el diseño o en la especificación del producto o servicio es mucho más barato que hacerlo cuando esté desplegado. En análisis estático nos permite **intervenir antes**.



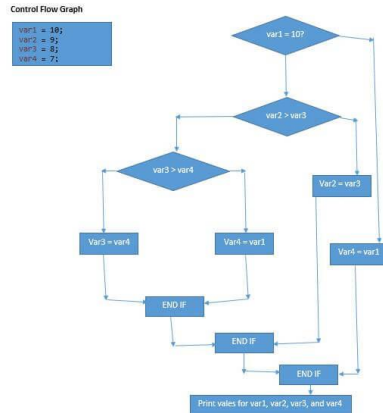
# Análisis estático del código (1)

- Cuando se apoya en herramientas permite analizar aspectos que sería muy complicado revisar manualmente: ramas que no se ejecutan, complejidad (ciclomática), ...
- El análisis manual también permite detectar desviaciones de los estándares de codificación y revisar la legibilidad y capacidad de evolución del código.
- Permite descubrir riesgos de seguridad.
- Permite detectar incoherencias (unidades, llamadas, interfaces).
- Ayuda a perfeccionar el testing descubriendo el comportamiento interno del código, por ejemplo encontrando partes que rara vez se activan en condiciones normales.

# Análisis estático del código (2)

## La complejidad ciclomática: un indicador de calidad y mantenibilidad

- Se basa en el análisis de los caminos posibles en un programa como si fuera un grafo siguiendo la fórmula:  
$$M (\text{Complejidad}) = \text{Aristas} - \text{Nodos} + \text{Salidas} * 2$$
- Aunque no despierta unanimidad entre los autores, generalmente se acepta que a mayor complejidad, mayor probabilidad de errores y peor mantenibilidad del código.



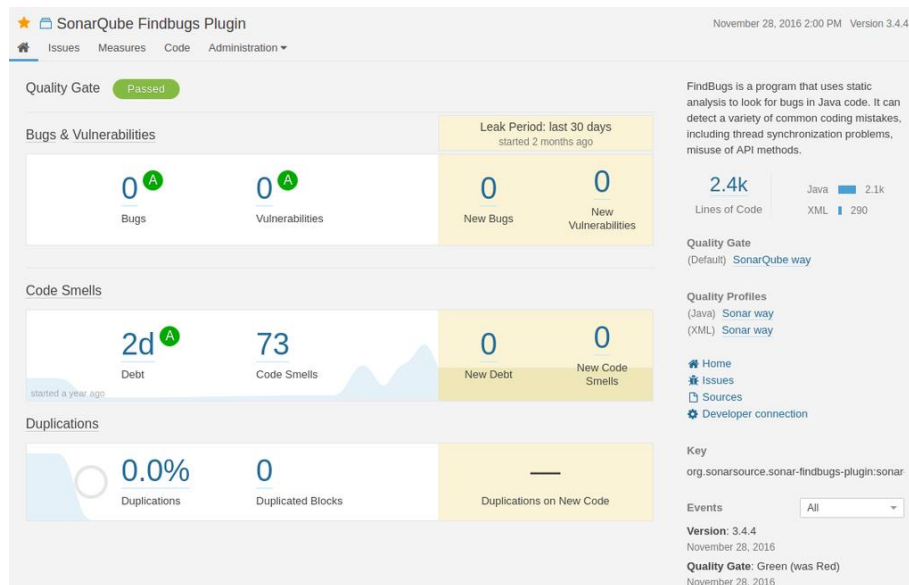
$$M = 11 - 11 + 2 * 1 = 2$$

Complejidad Ciclomática	Evaluación del Riesgo
1-10	Programa Simple, sin mucho riesgo
11-20	Más complejo, riesgo moderado
21-50	Complejo, Programa de alto riesgo
50	Programa no testeable, Muy alto riesgo

# Análisis estático del código (3)

## Sonaqube: una herramienta open source para el análisis del código

Analiza código en varios lenguajes para identificar posibles problemas, complejidad, vulnerabilidades de seguridad, “code smells”, ...



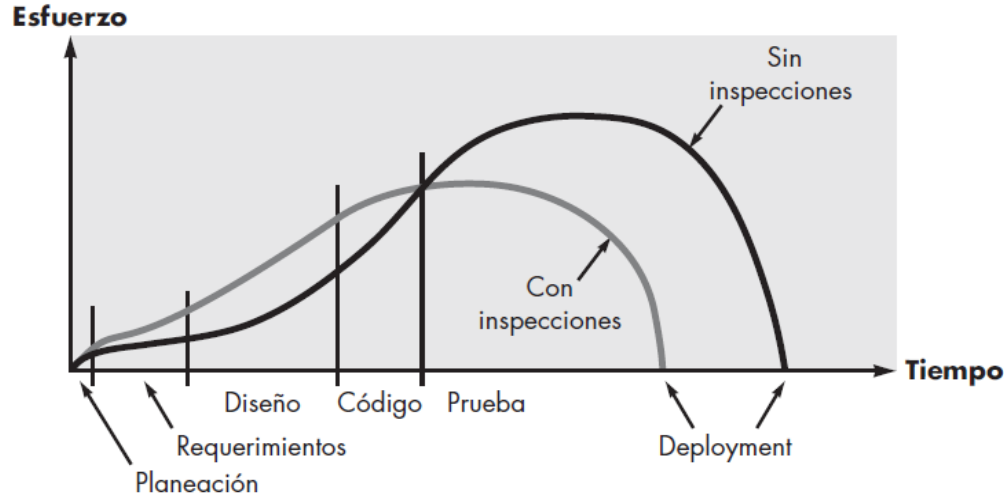
[https://en.wikipedia.org/wiki/SonarQube#/media/File:SonarQube\\_Project\\_page.png](https://en.wikipedia.org/wiki/SonarQube#/media/File:SonarQube_Project_page.png)

**Verificación del Software**

## **3.4. Revisión**

# ¿Por qué hacemos revisiones?

## Reducción de esfuerzo y tiempo, riesgo y mejora del producto final



A cambio de un incremento de esfuerzo en las primeras etapas, las revisiones (también llamadas RTF o Revisiones Técnicas Formales) generan un ahorro de tiempo y esfuerzo en las etapas finales. Laitenberger, A., *"A Survey of Software Inspection Technologies"*, 2002.

# Proceso de revisión (1)

**La revisión es un conjunto de técnicas que dan soporte al análisis estático manual.**

Las revisiones pueden ser de informales a absolutamente formales con muchos grados intermedios.

Informal:

- Sin un proceso definido
- Sin documentación de resultados

Formal:

- Roles
- Proceso establecido y documentado
- Registro de entrada y salida, documentación de resultados
- Soportado por herramientas

El grado de formalidad depende de aspectos como el modelo de ciclo de vida aplicado y su madurez, la naturaleza del producto y, especialmente, requisitos regulatorios, de auditoría o legales.



# Proceso de revisión (2)

Las revisiones pueden tener varios objetivos:

- **Descubrir defectos**, que es su propósito principal
- Generar **conocimiento** y **entendimiento** compartidos sobre los objetos revisados
- **Formación**, especialmente para personas que se incorporen al equipo, o para compartir información sobre el producto o servicio y cómo está hecho con personas de testing, de operaciones, seguridad, ...
- **Analizar problemas técnicos** (no necesariamente defectos), y buscar soluciones alternativas

Se trata de un proceso iterativo que una vez iniciado se aplica continuamente en el producto o servicio.

Hay bastante literatura sobre las revisiones (bajo el nombre de “revisiones técnicas formales”) ya que es una técnica con muchos años de recorrido. También hay estándares, como el ISO/IEC 20246 que detalla el proceso, técnicas, roles y resultados.

# Proceso de revisión (3)

## Actividades

- **Planificación**
  - Definición de alcance, objetivos, participantes, formato, técnicas, ...
  - Definición de requisitos para iniciar y finalizar la revisión.
- **Inicio de la revisión**
  - Alinear a los participantes
  - Distribuir los documentos, elementos de la revisión y contenido adicional
- **Preparación** (revisión) individual
  - De forma separada e individual, cada persona hace su propia revisión y documenta lo que encuentra, sus preguntas, observaciones, ...
- **Revisión**
  - Una reunión para poner en común lo encontrado entre los participantes y debatir sobre ello
- **Información y resolución**
  - Documentación de resultados, solucionar defectos y validar esa resolución, generar métricas y verificar que se cumplen los criterios para finalizar la revisión.

# Roles y responsabilidades

## Dependen del tipo de revisión, y algunos se puede combinar

- Liderazgo
  - Planifica la revisión, y es responsable de los resultados
- Autoría
  - Quien creó el producto y solucionará los defectos.
- Facilitación o moderación
  - Se encarga de que las reuniones sean fluidas y que no haya impedimentos que dificulten llegar a los resultados esperados.
- Revisión
  - Pueden ser personas expertas en la temática, stakeholders, miembros del equipo y, en general, personas que puedan contribuir.
  - Es una buena práctica incluir varios puntos de vista.
- Registro
  - Tomar nota de los defectos encontrados, de las decisiones tomadas y de cualquier aspecto relevante que ocurra durante la revisión.
- [Gestión general: ]
  - [Estrategia general, planificación, asignación de personas, monitorización]

# Tipos de revisiones

Los más habituales son:

- Revisión informal
- Walkthrough
- Revisión técnica
- Inspección

Cada una ofrece resultados distintos en función de los recursos implicados, los tipos de proyectos, el dominio técnico, la cultura de la organización, las necesidades, la regulación, etc.

Un mismo producto puede pasar por varias revisiones que pueden ser de menos a más formales.

# Tipos de revisiones: informal

Puede ser tan simple como una *peer review* o una revisión rápida por parte de un miembro del equipo. Muy habitual en el desarrollo sw en marcos ágiles.

Propósito:

- **Encontrar defectos**
- Facilitar el conocimiento compartido
- Generar nuevas ideas
- Solucionar problemas

No es un proceso estricto ni formal, ni siquiera necesita de una reunión. La documentación final puede ser muy básica (como un correo electrónico de resumen). Puede sacar partido del uso de checklists o patrones.

La utilidad y resultados depende mucho de las personas implicadas.

# Tipos de revisiones: walkthrough

Se trata de revisar el producto bajo la guía del autor/a.

Propósito:

- **Encontrar defectos**
- Encontrar mejoras y alternativas
- Compartir información, formación de nuevas incorporaciones, ...

Puede llevarse a cabo con una reunión dirigida por el autor/a, más o menos formal. Si es formal necesitará alguien que haga la facilitación y el registro (escriba).

Puede requerir de preparación individual previa y puede beneficiarse del uso de checklists o patrones.

# Tipos de revisiones: revisión técnica

Tiene un grado de formalidad mayor que las dos modalidades anteriores.

Propósito:

- **Encontrar defectos**
- Obtener consenso sobre la solución adoptada
- Evaluar la calidad
- Reducción de riesgos
- Generar nuevas ideas y alternativas
- Motivación para el autor/a
- Encontrar mejoras y alternativas
- Compartir información, formación de nuevas incorporaciones, ...

Requiere contar con personas que aporten conocimiento técnico experto, y personas del equipo (*peers*).

Es necesario que haya una persona para el registro (escriba). Conveniente contar con persona facilitadora, preparación individual y hacer uso de checklists.

# Tipos de revisiones: inspección

Es la revisión con mayor grado de formalidad.

Propósito:

- **Encontrar defectos**
- Evaluar la calidad
- Generar confianza en el producto
- Prevención de problemas similares por medio de la búsqueda de causa raíz y la generación de aprendizajes
- Motivación para el autor/a

Las personas en la revisión deben ser expertas, o miembros del equipo (*peers*).

Es obligatorio hacer una preparación individual, hacer uso de checklists, tener facilitador/a (no vale el autor/a) y registro con escriba, contar con criterios de entrada y salida, y recopilar métricas.



**Verificación del Software**

## **3.5. Pruebas de caja negra**

# Análisis dinámico

Se trata del conjunto de pruebas más extenso y habitual.

Implica estudiar el comportamiento de los productos del proceso de construcción del software (código, configuraciones, integraciones) cuando está en funcionamiento.

Para ello se trabajan en modalidades más o menos simuladas de las condiciones reales (desde un contexto en el que sólo el software que se está probando es real, hasta probarlo en el entorno de Producción).

Las pruebas pueden ser de **caja blanca** y de **caja negra**, y **basadas en experiencia**.

Además, cada prueba puede atender a aspectos funcionales y no funcionales.

En el siguiente apartado veremos qué tipos de técnicas se aplican en el análisis dinámico.

# Elección de técnicas

Tiene impacto sobre la definición de casos y condiciones de prueba.

Requieren apoyarse en las habilidades y conocimientos adecuados y contar con las herramientas necesarias.

Hay que conocer muy bien las fortalezas y debilidades de cada técnica.

Lo que buscamos es:

- Métodos repetibles y consistentes. La prueba debe poder replicarse.
- Cobertura medible objetivamente, lo que fundamenta la toma de decisiones.
- Generar aprendizajes que puedan usarse para procesos posteriores como parte de la mejora continua.
- En el siguiente apartado veremos qué tipos de técnicas se aplican en el análisis dinámico.
- Usar la técnica más apropiada para cada caso.

Este proceso de elección se materializa en las etapas de Análisis, Diseño e Implementación.

# Técnicas de caja negra (1)

Se basan en la idea de “no mirar” (porque no es posible o no nos interesa) en el interior de la “caja” que es el producto, el software:

- Todo lo que sabemos procede de su comportamiento a través de sus interfaces, entradas y salidas.
- No nos interesa saber cómo está hecho.

Por ese motivo, las condiciones, casos de prueba y datasets se derivan de la información disponible: requisitos, especificaciones, definiciones, historias de usuario, casos de uso, ...

Los casos de prueba nos ayudarán a detectar diferencias entre los requisitos y como están implementados (verificación).

La cobertura se basará en los elementos de prueba, no en cómo esté construido el producto (podemos dejar fuera ramas de código porque los requisitos tal y como están definidos no los disparan).

# Técnicas de caja negra (2)

Para evitar que el número de tests sea inmanejable y el proceso sea muy costoso, hay técnicas que permiten definir un número mínimo de casos de prueba, que al mismo tiempo aseguren la completitud de la prueba (no se dejan casos sin probar).

## EJEMPLO DE PRUEBA EXHAUSTIVA:

Tenemos que probar un sistema de programación agua caliente que entrega agua entre 10 y 90°, en incrementos de 0,5°; que se enciende en cualquier momento del día en saltos de 5'; y que puede estar funcionando hasta entre 5' y 24 horas en saltos de 5':

Casos posibles:  $80 \times 2 \times 24 \times 20 \times 24 \times 20 = 36.864.000$

Si cada caso se prueba durante 5" : 8 años

Si cada caso se prueba durante 1" : 19 meses

Son técnicas dinámicas que se deben lanzar sobre entornos que reproduzcan las condiciones reales con la fidelidad necesaria en cada caso (depende del nivel de prueba).

# Partición equivalente

# Partición equivalente (1)

Es una técnica muy habitual que permite construir rápidamente conjuntos de prueba con objeto de validar, al menos una vez, algunos de los elementos que se consideren básicos en la prueba.

Normalmente se hacen porque requeriría demasiado tiempo ejecutar el conjunto de todos los datos posibles de prueba. O como una forma rápida de validar antes de hacer una prueba exhaustiva.

Tiene sentido si todos los elementos dentro de un grupo se van a comportar de la misma forma, por lo que probar uno debería ser suficiente.

Se basa en construir **particiones equivalentes** o de equivalencia: grupos de elementos que se comportan de la misma forma:

- Si pasa el test en uno, podemos asumir que pasará en todos
- Si falla el test en uno, podemos asumir que fallará en todos

# Partición equivalente (2)

## Ejemplo

Un criterio de aceptación de la historia de usuario dice:

“Sólo se aceptarán números enteros mayores de 50”



# Partición equivalente (2)

## Ejemplo

Un criterio de aceptación de la historia de usuario dice:  
“Sólo se aceptarán números enteros mayores de 50”

Clases:

- Números negativos
- Números positivos menores o iguales que 50
- Números positivos mayores que 50
- Caracteres no numéricos
- Vacío

Test Case	Entrada	Resultado esperado
1	10	NOK
2	100	OK
3	-10	NOK
4	X	NOK
5	null	NOK

# Ejercicio: partición equivalente

## **Busca las particiones para este supuesto**

Precio de la entrada según la edad:

- Hasta 18 años: gratis
- 18 a 55: 20€
- 55 a 65: 18€
- 65 a 80: 10€
- Más de 80 años: gratis

¿Cuántos casos de prueba hay?

# Ejercicio: partición equivalente

SOLUCIÓN

## Busca las particiones para este supuesto

Precio de la entrada según la edad:

- Hasta 18 años: gratis
- 18 a 55: 20€
- 55 a 65: 18€
- 65 a 80: 10€
- Más de 80 años: gratis

¿Cuántos casos de prueba hay?

- Números negativos: -15
- Positivos menores de 18: 10
- Entre 18 y 55: 40
- Entre 55 y 65: 60
- Entre 65 y 80: 75
- Más de 80: 90
- Caracteres no numéricos: “!po
- Null o entrada vacía

# Partición equivalente (3)

## ¿Qué se puede meter en particiones?

Cualquier dato que esté relacionado con el objeto que se esté probando:

- Entrada y salida de usuarios o de otros sistemas o interfaces:
  - Rangos numéricos
  - Con contenido o vacío
  - Conjuntos de datos
  - Relaciones
- Valores relacionados con fecha y tiempo
- Resultados intermedios de cálculos
- Precondiciones (existencia o ausencia)
- Configuraciones

Tan importante como la entrada es analizar la salida con particiones

# Partición equivalente (4)

## ¡Cuidado con las cosas que asumimos!

Las particiones equivalentes se basan en la idea de que todo lo que está dentro de una partición se procesa de la misma forma ...

... pero no hay garantía de que eso suceda así (y por eso hacemos pruebas)

Es importante validar las asunciones con otras personas (desarrollo, analistas, los mismos usuarios) para asegurar su validez y, sobre todo, el **riesgo** asociado a esas asunciones.

Cuidado con ciertos casos:

- ¿Se tratan igual los valores positivos y negativos?
- ¿Cómo se trata el cero? ¿Cómo entero positivo?
- No es lo mismo espacios que “nada” (null)
- Cuidado con los distintos juegos de caracteres y codificaciones ¿Cómo está llegando la entrada del usuario? ¿Filtros por idioma?

# Ejercicio: partición equivalente

## Busca las particiones para este supuesto

Una ferretería hace descuentos según lo que gaste cada cliente:

- 25€ o más: 2%
- 50€ o más: 5%
- A partir de 100€: 10%

Para preparar los casos de prueba de la aplicación de caja hay que identificar las particiones equivalentes de los datos anteriores. Te puedes ayudar con esta tabla:

Caso de prueba	Entrada	Salida esperada		Partición
	Gasto	Descuento	Total	

# Ejercicio: partición equivalente

SOLUCIÓN

Busca las particiones para este supuesto

0%		No válido		2%		5%		10%	
$-\infty$	-0,01	0,00	24,99	25,00	49,99	50,00	99,99	100,00	$\infty$

Caso de prueba	Entrada	Salida esperada		Partición
	Gasto	Descuento	Total	
1	20,35	0	20,35	0%
2	44,44	0,89	43,55	2%
3	75,75	3,79	71,96	5%
4	1234,56	123,46	1111,10	10%
5	-123,45	Mensaje de error	Mensaje de error	No válido

# **BVA. Análisis de valor límite**

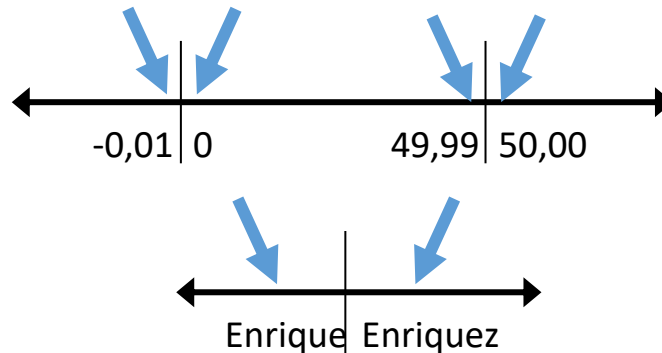


# BVA: análisis de valor límite (1)

El BVA o *Boundary Value Analysis* es una extensión de la Partición equivalente para particiones ordenadas.

Se basa en la idea (y la experiencia real) de que los defectos suelen aparecer en los límites. Además de que a la hora de construir el software hay operadores que son fáciles de confundir, incluso por errores al teclear ( $= < > = > < >$ ) y olvidos (“NOT”, signo “-”).

Los límites son los de las particiones: valor mínimo, máximo, primero o último de una partición. Por eso es también **BVA de 2 puntos**.



# BVA: análisis de valor límite (2)

En el BVA con valores numéricos, hay que tener la seguridad de la unidad más pequeña (decimales). Debería formar parte de los requisitos o los criterios de aceptación explícitos, o de unos criterios implícitos aceptados.

En el ejercicio anterior:

No válido		0%		2%		5%		10%	
$-\infty$	-0,01	0,00	24,99	25,00	49,99	50,00	99,99	100,00	$\infty$

Esto se traduce en 5 particiones y 8 valores límite, de los cuales, 4 particiones y 7 valores límite dan resultados válidos.

# Ejercicio: BVA

## Busca valores límite en este supuesto

Un monitor de concentración de medicamento en sangre recibe medidas cada minuto. Muestra un indicador en función del nivel detectado de acuerdo con esta escala:

- “Bajo”  $\leq 0,2\%$
- “Normal”  $> 0,2\%$  y  $\leq 1\%$
- “Vigilar”  $> 1\%$  y  $\leq 3\%$
- “Reducir”  $> 3\%$  y  $\leq 5\%$
- “ALERTA”  $> 5\%$

El sensor no devuelve valores negativos. Usando la técnica BVA de 2 puntos, identifica los tests cases que verifican el sistema. Puedes usar esta tabla:

Caso de prueba	Entrada	Salida esperada		Partición

# Ejercicio: BVA

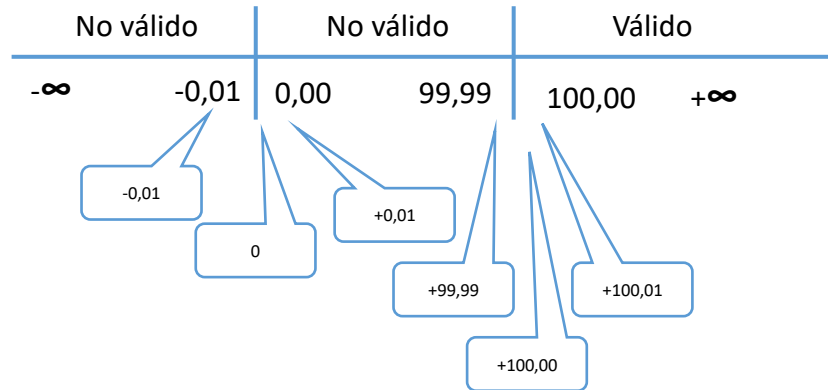
SOLUCIÓN

Busca valores límite en este supuesto

Caso de prueba	Entrada	Salida esperada	Partición
1	0	Bajo	1
2	0,2	Bajo	1
3	0,201	Normal	2
4	1,0	Normal	2
5	1,001	Vigilar	3
6	3,0	Vigilar	3
7	3,001	Reducir	4
8	5,0	Reducir	4
9	5,001	ALERTA	5

# BVA3: BVA de tres puntos (1)

La técnica BVA es válida para la mayoría de los errores relacionados con valores límites. De todos modos, pueden escaparse algunas situaciones con valores numéricos, así que tendremos que aplicar una técnica más refinada que se fije en el punto exacto del límite. Por ejemplo, el requisito dice: el “programa aceptará valores positivos mayores de 100”



Tenemos 3 valores por límite.

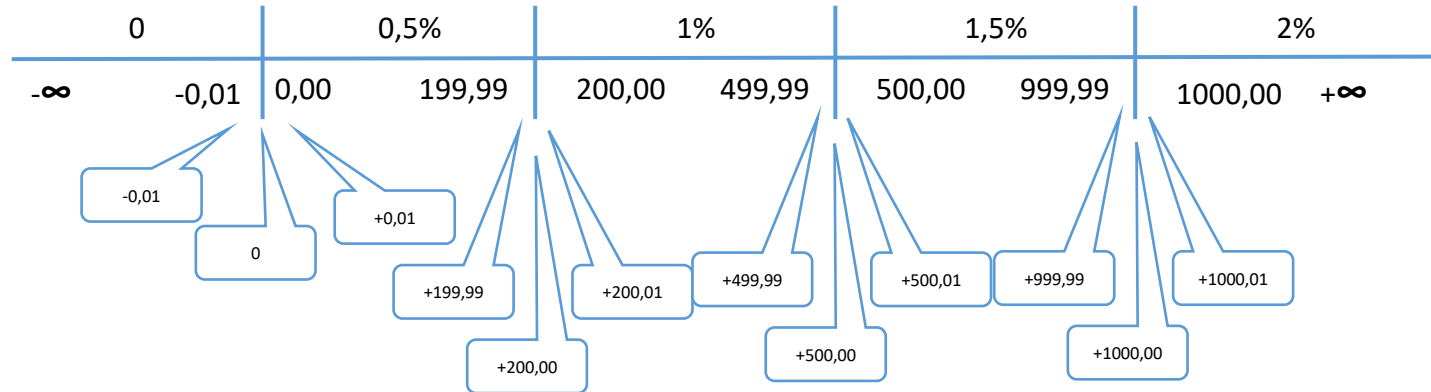
En este caso estamos contando con decimales. Hay que conocer bien de qué tipo de límite estamos hablando en cada caso. ¿Qué valores usaríamos si fueran números enteros?

# BVA3: BVA de tres puntos (2)

## Ejemplo:

El requisito dice “el cálculo de descuentos en la venta de gasoil de calefacción es:

- Hasta 199,99€: 0,5%
- Hasta 499,99€: 1%
- Hasta 999,99€: 1,5%
- Desde 1.000€: 2%



# BVA3: BVA de tres puntos (3)

## Ejemplo: casos de prueba resultantes

Caso de prueba	Entrada	Salida esperada
1	-0,01	Sin interés
2	0	0
3	0,01	0
4	199,99	0
5	200	0,5%
6	200,01	0,5%
7	499,99	0,5%
8	500	1,5%
9	500,01	1,5%
10	999,99	1,5%
11	1000	2%
12	1000,01	2%

# Ejercicio: BVA3

## Busca valores límite en este supuesto

El programa que calcula las retenciones en la nómina debe calcular el porcentaje en función del sueldo bruto de acuerdo con los siguientes tramos:

- Hasta 12.500€: 19%
- Hasta 20.200: 24%
- Hasta 35.200: 30%
- Hasta 60.000: 37%
- Por encima: 45%

El cálculo se hace hasta céntimos.

¿Qué tests cases deben usarse para verificar que el programa calcula correctamente?

Caso de prueba	Entrada	Salida esperada	Valor por debajo, exacto o por encima



# Ejercicio: BVA3

SOLUCIÓN

Busca valores límite en este supuesto

Caso de prueba	Entrada	Salida esperada	Valor por debajo, exacto o por encima
1	-0,01€	Error	Por debajo
2	0	19%	Exacto
3	0,01	19%	Por encima
4	12.499,99	19%	Por debajo
5	12.500,00	19%	Exacto
6	12.500,01	24%	Por encima
7	35.199,99	24%	Por debajo
8	35.200,00	24%	Exacto
9	35.200,01	30%	Por encima
10	59.999,99	30%	Por debajo
11	60.000,00	30%	Exacto
12	60.000,01	45%	Por encima

# BVA: cuándo usarlo

Hay cuatro tipos de situaciones donde el Boundary Value Analysis es adecuado:

- Rangos
- Subparticiones
- Cuentas
- Relaciones

# BVA sobre rangos

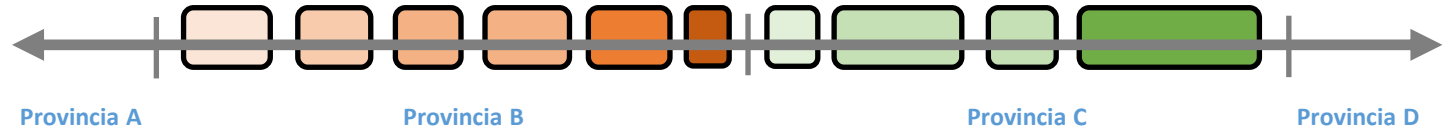
Se trata de las situaciones más habituales, y las que hemos visto en los ejemplos. Pueden ser números enteros, reales o secuencias de caracteres no numéricos (ordenación de palabras, por ejemplo).

Posibles casos:

- Cálculo de impuestos y retenciones
- Cálculo de descuentos
- Códigos de identificación
- Número de cuentas
- Números de identificación
- Edad
- Salarios
- Préstamos
- Pólizas e indemnizaciones
- Orden alfabético
- ...

# BVA sobre subparticiones

Rangos anidados dentro de otros subrangos: por ejemplo códigos postales dentro de provincias, o los códigos las ciudades principales.



# BVA sobre cuentas

Números elementos, que serán siempre enteros y positivos.

Por ejemplo:

- Número de caracteres
- Paginación
- Resultados de búsqueda
- Registros
- Entradas en un menú
- Número de días
- Número de empleados
- Número de hijos
- ...

# BVA en relaciones

Normalmente comparaciones entre un valor con otro de referencia.

Por ejemplo:

- Límite para alquiler con respecto al salario
- Nuevo % de intereses con respecto al anterior
- Fecha actual con respecto a la fecha límite y otras fechas de referencia
- Salario con respecto al límite para recibir una ayuda o una beca

# Ejercicio: aplicación de BVA

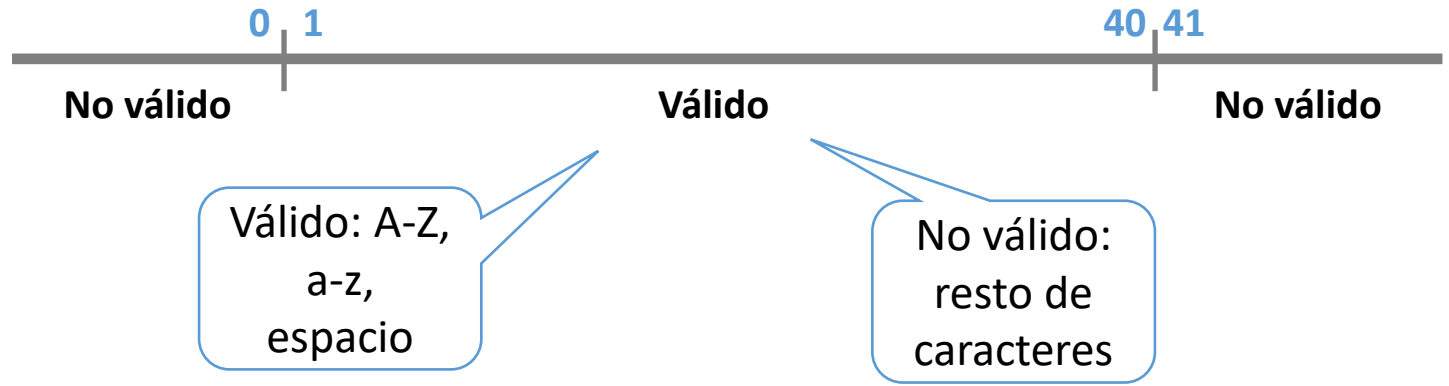
## **Buscar particiones y valores límite**

El nombre del titular de una tarjeta de crédito puede tener hasta un máximo de 40 caracteres, y no puede estar vacío. Los caracteres válidos son caracteres latinos mayúsculos y minúsculos, y espacios.

¿Qué particiones y valores límite son necesarios para la creación de tests cases?

# Ejercicio: aplicación de BVA *SOLUCIÓN*

Buscar particiones y valores límite





# Tablas de decisión

# Tablas de decisión (1)

Muchas especificaciones se basan en aplicar reglas de negocio para definir el comportamiento del sistema o una de sus partes. Suelen ser reglas simples, aunque es habitual combinarlas.

Para anticipar los valores de salida hay que contar con todas las reglas. El número de combinaciones puede hacer que haya muchas posibilidades y que éstas sean complejas, aumentando la aparición de defectos.

Las tablas de decisión son una herramienta de gestión de la complejidad, haciendo **visible** cada combinación, para tomar decisiones sobre probarla y cómo.

# Tablas de decisión (2)

Por ejemplo, en un proceso de contratación, sólo se va a entrevistar a las personas que son mayores de edad y tienen la ESO. Con esas dos reglas, tenemos estos casos:

1. Menor de edad, sin ESO
2. Mayor de edad, sin ESO
3. Menor de edad, con ESO
4. Mayor de edad, con ESO

Trasladado a una tabla

En las condiciones, cada fila contiene las reglas de negocio del caso

Las filas de acciones definen las posibles consecuencias

Combinaciones	1	2	3	4
Condiciones				
¿Mayor de edad?	S	S	N	N
¿ESO?	S	N	S	N
Acciones				
¿Entrevistar?	X	-	-	-

Cada columna define una combinación única de los resultados esperados

# Tablas de decisión (3)

Las condiciones binarias pueden representarse con letras (S/N, Y/N, T/F) o números (1/0). Si la condición tiene múltiples valores es mejor usar el valor real o algún tipo de código (P, Primaria; S, Secundaria; B, Bachillerato; G, Grado; M, Máster; D, Doctorado; O, Otros)

Desencadenar una acción puede ser “X”, “1”, “S”, “Y”, “T”, ... y no hacerlo “-”, “0”, “N”, ... Si las acciones no son binarias (tasa de descuento, de retención) es mejor usar los valores reales.

Esta técnica se presta bastante a usar hojas de cálculo, sobre todo cuando hay que tener en cuenta muchas reglas de negocio de las que se derivan muchos casos de prueba. El número de combinaciones será el producto del número de posibles valores de todas las condiciones consideradas. Calcular este número de antemano ayuda a entender las dimensiones del problema, y sirve de comprobación para analizar la corrección del planteamiento de casos.

# Tablas de decisión (4)

Para pasar de la tabla a test cases, creamos un caso para cada una de las posibles combinaciones:

<i>Test Case</i>	<i>Combinación</i>	<i>Resultado</i>
1	Edad 16, con ESO	No se entrevista
2	Edad 22, sin ESO	No se entrevista
3	Edad 17, sin ESO	No se entrevista
4	Edad 18, con ESO	Se entrevista

Además, pueden tenerse en cuenta valores límite (17, 18, 19 en este ejemplo).

# Tablas de decisión (5)

El coste de envío para peso “Normal” es:

- 1€ en paquete “Pequeño, S”
- 1,50€ en paquete “Mediano, M”
- 2€ en paquete “Grande, L”

Si el peso es “Pesado”, el coste se incrementa en 1€; si es “Muy pesado” en 2€

Si el envío se paga con tarjeta, se incrementa un 5%:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
<i>Condiciones</i>																		
Tamaño: S / M / L	S	S	S	S	S	S	M	M	M	M	M	M	L	L	L	L	L	L
Peso, N / P / MP	N	N	P	P	MP	MP	N	N	P	P	MP	MP	N	N	P	P	MP	MP
Tarjeta S / N	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N	S	N
<i>Acciones</i>																		
Coste básico, €	1	1	1	1	1	1	1,5	1,5	1,5	1,5	1,5	1,5	2	2	2	2	2	2
Extra por peso, €	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2	2
Recargo, %	0	5%	0	5%	0	5%	0	5%	0	5%	0	5%	0	5%	0	5%	0	5%

18 combinaciones ( $3 \times 3 \times 2$ ). Para un tamaño determinado de paquete 6 ( $3 \times 2$ )

# Ejercicio: Tablas de decisión

## Tests cases para combinaciones de reglas

Una compañía telefónica completa sólo las llamadas de los números que son válidos. Algunas llamadas son gratuitas (112), pero otras tienen cargo. Si tienen cargo, hay que asegurarse de si la persona que hace la llamada puede pagarla (tiene saldo o es de contrato). Sólo si puede pagarla se completa la llamada y se descuenta el importe. Crea la tabla de decisión necesaria. Empezar por calcular el número de combinaciones ayudará a saber si la tabla es correcta o no.

# Ejercicio: Tablas de decisión **SOLUCIÓN**

## Tests cases para combinaciones de reglas

Número de combinaciones:  $2*2*2 = 8$

Casos: Número válido (S/N); Llamada gratuita (S/N); Puede pagarla (S/N)

Resultado: Llamada conectada (S/N); Llamada cobrada (S/N)

Combinaciones	1	2	3	4	5	6	7	8
Condiciones								
¿Número válido?	N	N	N	N	S	S	S	S
¿Llamada gratuita?	S	S	N	N	S	S	N	N
¿Puede pagarla?	N	S	N	S	N	S	N	S
Acciones								
Llamada conectada	N	N	N	N	S	S	N	S
Descuenta importe	N	N	N	N	N	N	N	S



# Colapsar tablas de decisión (1)

Es una técnica para simplificar tablas de decisión, reduciendo el número de test cases al eliminar combinaciones innecesarias. También se denomina “condensar”, “racionalizar” y “simplificar”.

Esta técnica tiene lugar debido a que:

- Puede haber combinaciones que no ocurren, o no tienen sentido.
- Hay combinaciones que dan lugar a las **mismas** acciones por las **mismas** razones (ojo, esto no aplica cuando lugar a las mismas acciones por distintas razones).

Hay que tener **mucho cuidado con las asunciones** que se hagan:

- Lo que ocurra en realidad dentro del software depende de cómo esté escrito el código. Estamos en pruebas de caja negra, no sabemos cómo está hecho.
- Normalmente las condiciones se programan en el orden en el que se especifican, pero no tiene que ser necesariamente así.
- Una simplificación siguiendo un orden que no sea el mismo del código puede dar lugar a **falsos positivos** o **falsos negativos**


# Colapsar tablas de decisión (2)

## Mismas acciones por las mismas razones

En el ejemplo de las entrevistas de trabajo, sólo las hacemos a personas mayores de edad con el certificado de ESO.

Si la persona candidata es menor de edad nos da igual el resto de datos.

Combinaciones	1	2	3	4
Condiciones				
¿Mayor de edad?	S	S	N	N
¿ESO?	S	N	S	N
Acciones				
¿Entrevistar?	X	-	-	-



Combinaciones	1	2	3
Condiciones			
¿Mayor de edad?	S	S	N
¿ESO?	S	N	~
Acciones			
¿Entrevistar?	X	-	-

# Colapsar tablas de decisión (2)

## Combinaciones sin sentido (1)

Se trata de identificar situaciones que no puedan darse en la vida real (por ejemplo, estudios a nivel de máster, 10 años de experiencia y menos de 25 años).

En el caso del envío por correo, supongamos que las reglas de negocio cambian para proteger la salud de los empleados, y los paquetes pequeños sólo puedan tener peso normal, y los medianos, sólo normal y pesado:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
<i>Condiciones</i>																		
Tamaño: S / M / L	S	S	S	S	S	S	M	M	M	M	M	M	L	L	L	L	L	L
Peso, N / P / MP	N	N	P	P	MP	MP	N	N	P	P	MP	MP	N	N	P	P	MP	MP
Tarjeta S / N	S	N	~	~	~	~	S	N	S	N	~	~	S	N	S	N	S	N
<i>Acciones</i>																		
Coste básico, €	1	1	1	1	1	1	1,5	1,5	1,5	1,5	1,5	1,5	2	2	2	2	2	2
Extra por peso, €	0	0	1	1	2	2	0	0	1	1	2	2	0	0	1	1	2	2
Recargo, %	0	5%	0	5%	0	5%	0	5%	0	5%	0	5%	0	5%	0	5%	0	5%

# Colapsar tablas de decisión (3)

## Combinaciones sin sentido (2)

En este caso hemos podido eliminar 3 casos de los 18 definidos:

	1	2	3	5	7	8	9	10	11	13	14	15	16	17	18
<i>Condiciones</i>															
Tamaño: S / M / L	S	S	S	S	M	M	M	M	M	L	L	L	L	L	L
Peso, N / P / MP	N	N	P	MP	N	N	P	P	MP	N	N	P	P	MP	MP
Tarjeta S / N	S	N	~	~	S	N	S	N	~	S	N	S	N	S	N
<i>Acciones</i>															
Coste básico, €	1	1	1	1	1,5	1,5	1,5	1,5	1,5	2	2	2	2	2	2
Extra por peso, €	0	0	1	2	0	0	1	1	2	0	0	1	1	2	2
Recargo, %	0	5%	0	0	0	5%	0	5%	0	0	5%	0	5%	0	5%

Si aún siguen siendo muchas, puede elegirse un subconjunto para prueba en función de la frecuencia de aparición, la importancia, la propensión a error, opinión de expertos, ... y también se puede rotar para no probar todos los casos siempre (happy path y casos de error, por bloques, ...)

# Ejercicio: colapsar tablas de decisión

## Simplificar tabla del ejemplo de la compañía telefónica

Una compañía telefónica completa sólo las llamadas de los números que son válidos. Algunas llamadas son gratuitas (112), pero otras tienen cargo. Si tienen cargo, hay que asegurarse de si la persona que hace la llamada puede pagarla (tiene saldo o es de contrato). Sólo si puede pagarla se completa la llamada y se descuenta el importe.

Combinaciones	1	2	3	4	5	6	7	8
Condiciones								
¿Número válido?	N	N	N	N	S	S	S	S
¿Llamada gratuita?	S	S	N	N	S	S	N	N
¿Puede pagarla?	N	S	N	S	N	S	N	S
Acciones								
Llamada conectada	N	N	N	N	S	S	N	S
Account debited	N	N	N	N	N	N	N	S

# Ejercicio: colapsar tablas de decisión

## Simplificar tabla del ejemplo de la compañía telefónica

Dejamos los casos en la mitad:

- Si el número no es válido, no miramos nada más
- Si la llamada es gratuita, no nos importa la última condición

Combinaciones	1	2	3	4	5	6	7	8
Condiciones								
¿Número válido?	N	N	N	N	S	S	S	S
¿Llamada gratuita?	~	~	~	~	S	S	N	N
¿Puede pagarla?	~	~	~	~	~	~	N	S
Acciones								
Llamada conectada	N	N	N	N	S	S	N	S
Descuenta importe	N	N	N	N	N	N	N	S



Combinaciones	1	2	3	4
Condiciones				
¿Número válido?	N	S	S	S
¿Llamada gratuita?	~	S	N	N
¿Puede pagarla?	~	~	N	S
Acciones				
Llamada conectada	N	S	N	S
Descuenta importe	N	N	N	S

# Transición entre estados

# Transición entre estados (1)

Se usa cuando tenemos una serie de situaciones estables gobernadas por reglas que determinan los cambios entre situaciones.

Este tipo de situaciones son muy comunes, por ejemplo para:

- Workflows e historiales (por ejemplo, presentar una solicitud, revisarla, aprobarla, comunicar decisión, comprobar ejecución, archivarla)
- Secuencias de flujo en procesos multipágina en Webs (por ejemplo las acciones para validar carrito, dirección, medio de pago, ...)
- Proceso de fabricación, de creación y despliegue de software (Jenkins), secuencia de pruebas automatizadas o manuales, ...

Los test cases de transición de estados pueden servir para validar:

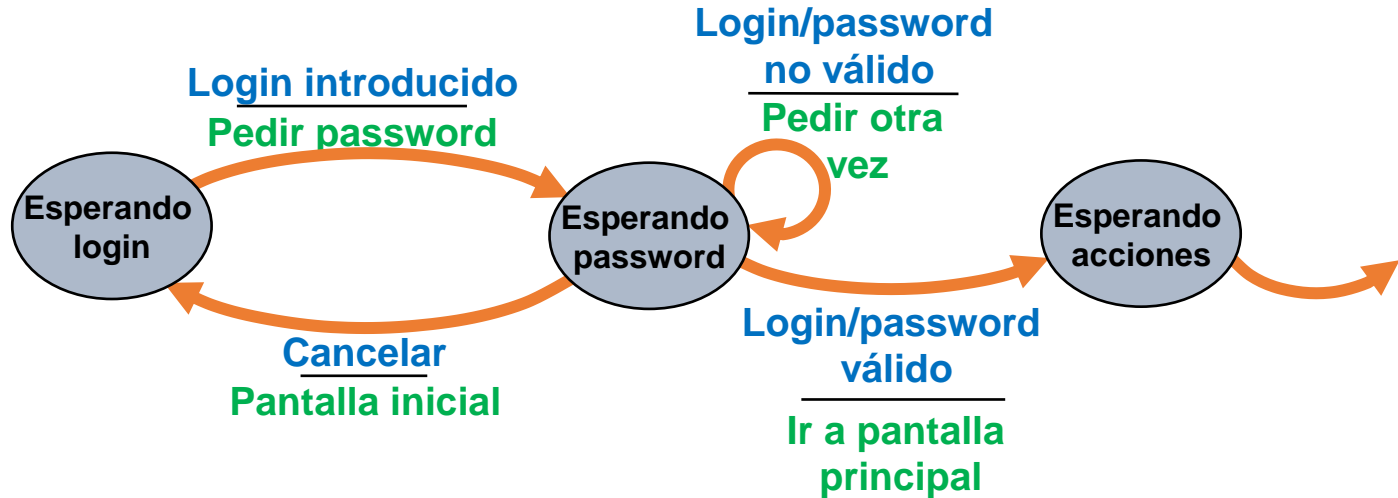
- Una secuencia típica, básica u óptima
- Cada uno de los estados
- Cada una de las transiciones individuales y las secuencias de transiciones
- También transiciones incorrectas, no especificadas o efectos de otras acciones del sistema



# Transición entre estados (2)

Clarifica mucho el uso de un diagrama de transición o estados, que contiene:

- **Estados** de la aplicación, proceso, ...
- **Transiciones** entre estados (no todas son válidas)
- **Eventos** que disparan las transiciones
- **Acciones** como resultado de cada transición



# Transición entre estados (3)

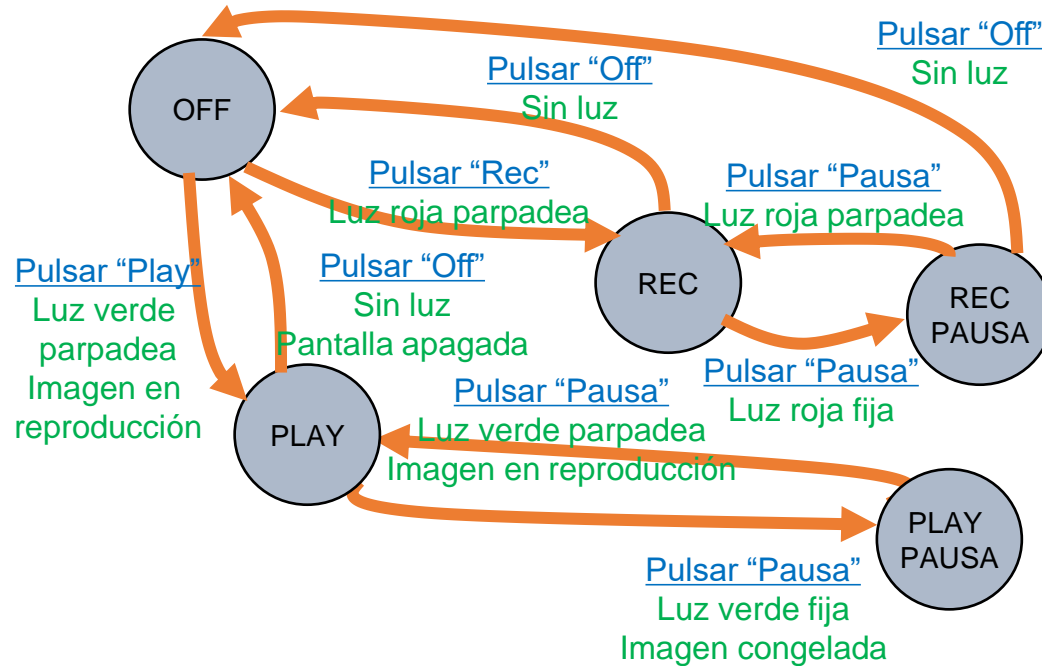
Esta técnica solo tiene sentido si es posible identificar un número **finito** de estados:

- Cada estado debe ser **identificable**
- Los estados **no deben simultanearse** y deben cubrir **todas las posibilidades** (no hay situaciones “sin estado”)
- Tiene que haber formas de moverse entre estados: las **transiciones**.
- Todas las transiciones tienen que tener un **evento** que las active, y ese evento tiene que ser identificable, ya que es necesario para la prueba. Los eventos pueden ser acciones, datos, un periodo de tiempo, ...
- Las transiciones pueden dar lugar a **acciones**, que también deberán probarse
- Una transición puede llevar a estados anteriores o mantener el sistema en el mismo estado (caso anterior cuando se reintroduce la password y es incorrecta)
- Puede haber **condiciones** que bloqueen transiciones. Así, en el ejemplo anterior, una condición para pedir la password puede ser que el login exista:



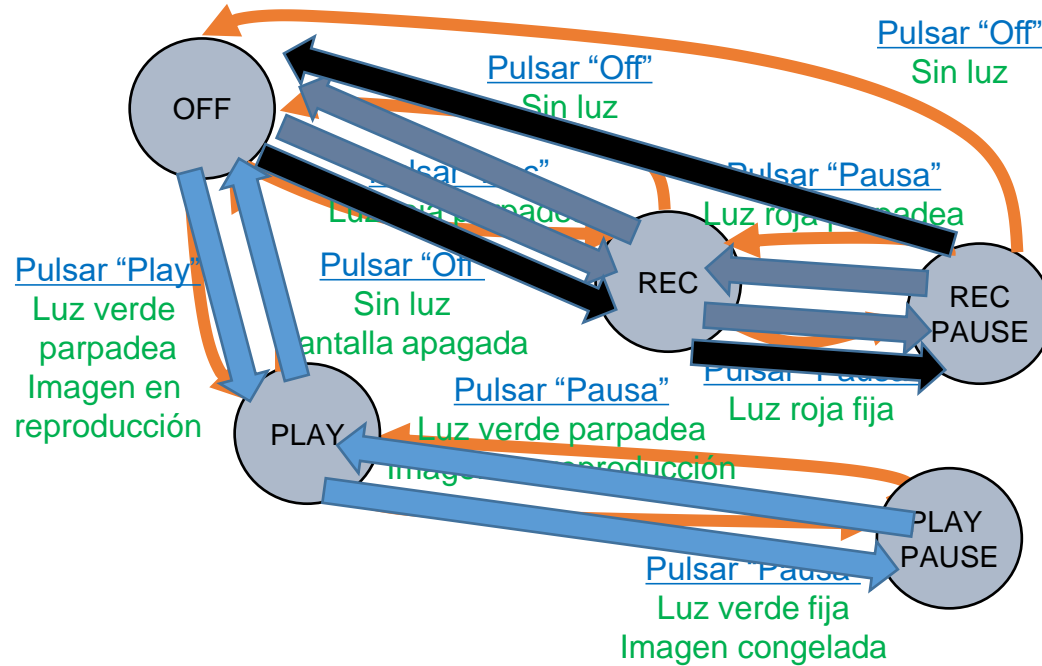
# Transición entre estados (4)

Ejemplo: cámara de video.



# Transición entre estados (5)

Casos de prueba:



# Transición entre estados (6)

Tabla de tests positivos: transiciones correctas

Estado inicial	Evento	Acciones	Estado final
OFF	Pulsar "Play"	Luz verde parpadea Imagen en reproducción	PLAY
OFF	Pulsar "Rec"	Luz roja parpadea	REC
PLAY	Pulsar "Off"	Sin luz Pantalla apagada	OFF
PLAY	Pulsar "Pausa"	Luz verde fija Imagen congelada	PLAY PAUSA
REC	Pulsar "Off"	Sin luz	OFF
REC	Pulsar "Pausa"	Luz roja fija	REC PAUSA
PLAY PAUSA	Pulsar "Play"	Luz verde parpadea Imagen en reproducción	PLAY
REC PAUSA	Pulsar "Pausa"	Luz roja parpadea	REC
REC PAUSA	Pulsar "Off"	Sin luz	OFF

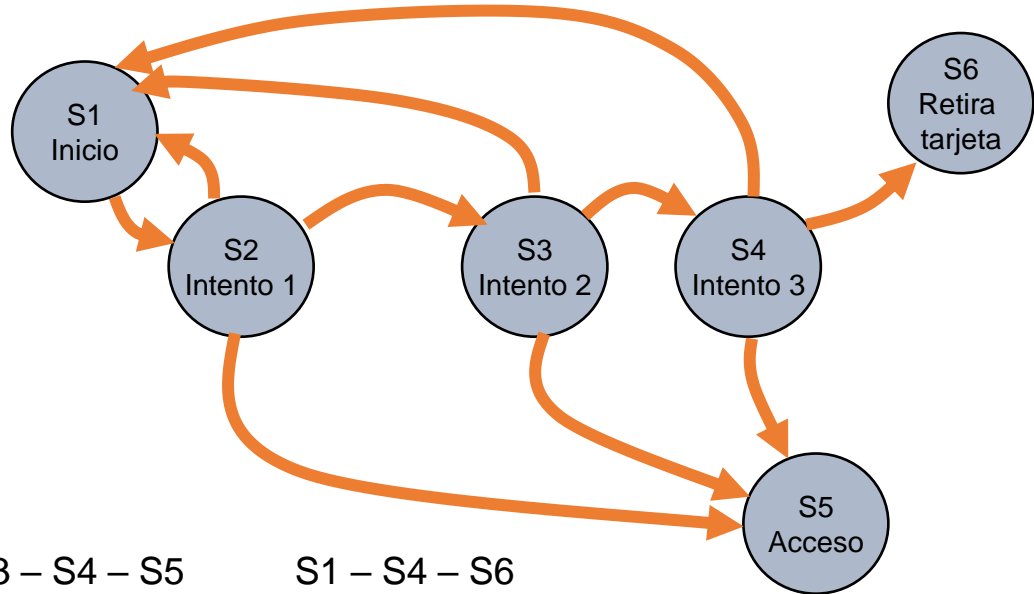
# Transición entre estados (7)

Tabla con todas las transiciones: muestra las transiciones correctas y las erróneas

	Pulsar “Off”	Pulsar “Rec”	Pulsar “Play”	Pulsar “Pausa”
OFF	-	REC	PLAY	-
PLAY	OFF	-	-	PLAY PAUSA
REC	OFF	-	-	REC PAUSA
PLAY PAUSA	-	-	-	PLAY
REC PAUSA	OFF	-	-	REC

# Transición entre estados (8)

¿Qué par de test cubren todos los estados?

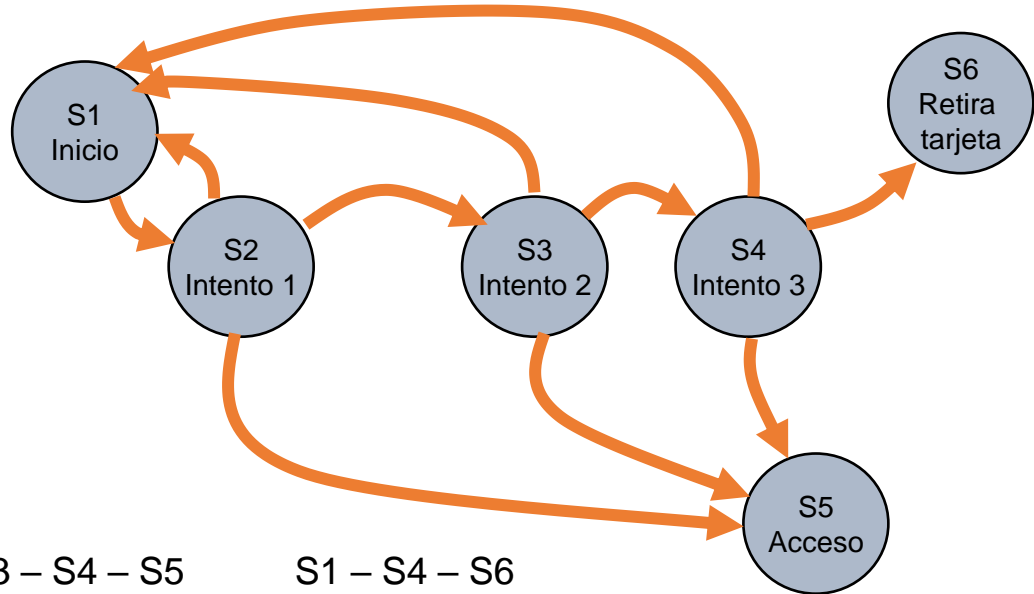


1. S1 – S2 – S3 – S4 – S5
2. S1 – S2 – S4 – S5 – S6
3. S1 – S2 – S5
4. S1 – S2 – S3 – S4 – S5

1. S1 – S4 – S6
1. S1 – S2 – S3 – S4 – S5
1. S1 – S2 – S3 – S4 – S6
1. S4 – S6

# Transición entre estados (9)

¿Qué par de test cubren todos los estados?



1. S1 – S2 – S3 – S4 – S5
2. S1 – S2 – S4 – S5 – S6
3. **S1 – S2 – S5**
4. S1 – S2 – S3 – S4 – S5

- S1 – S4 – S6
- S1 – S2 – S3 – S4 – S5
- S1 – S2 – S3 – S4 – S6**
- S4 – S6



# Transición entre estados (10)

¿Qué transición produce un resultado erróneo?

	A	B	C	D
S1	S2	-	-	-
S2	S3	-	S4	-
S3	-	S1	-	S4
S4	-	S2	S1	-

1. S2 y A
2. S4 y B
3. S2 y C
4. S4 y D

# Transición entre estados (10)

¿Qué transición produce un resultado erróneo?

	A	B	C	D
S1	S2	-	-	-
S2	S3	-	S4	-
S3	-	S1	-	S4
S4	-	S2	S1	-

1. S2 y A
2. S4 y B
3. S2 y C
4. **S4 y D**

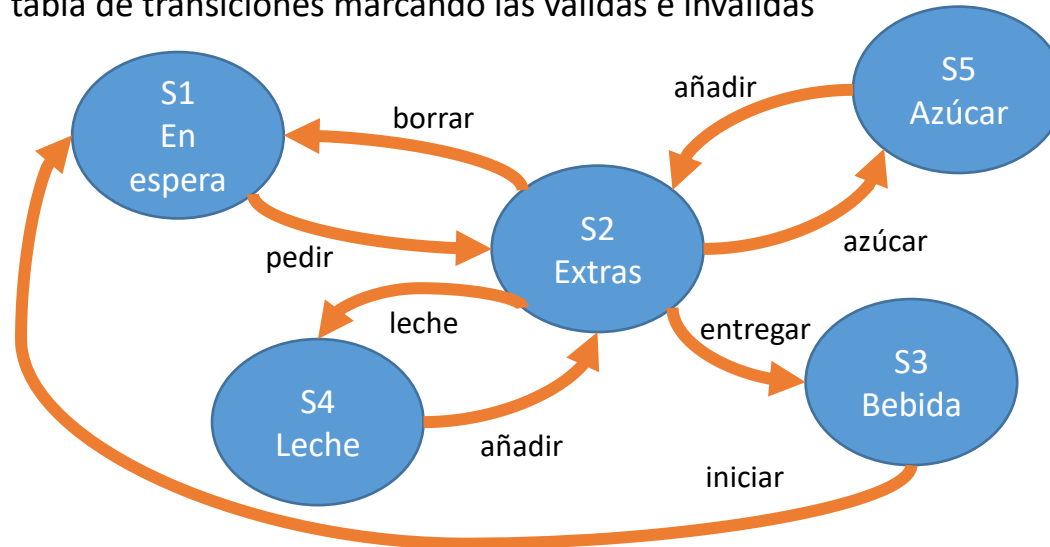
# Ejercicio: Transición entre estados

## Estados que representan una máquina de vending

Crea los test cases para:

1. Una transacción típica
2. Pasar por todos los estados
3. Lanzar todas las transiciones

Crear la tabla de transiciones marcando las válidas e inválidas



# Ejercicio: Transición entre estados

SOLUCIÓN

## Estados que representan una máquina de vending

1. Una transacción típica:
  - S1 – pedir – S2 – azúcar – S5 – añadir – S2 – entregar – S3 – iniciar – S1
2. Pasar por todos los estados:
  - S1 – pedir – S2 – azúcar – S5 – añadir – S2 – leche – S4 – añadir – S2 – entregar – S3 – iniciar – S1
3. Lanzar todas las transiciones
  - S1 – pedir – S2 – azúcar – S5 – añadir – S2 – leche – S4 – añadir – S2 – entregar – S3 – iniciar – S1 – pedir – S2 – borrar

	pedir	azúcar	leche	añadir	entregar	borrar	iniciar
S1	S2	-	-	-	-	-	-
S2	-	S5	S4	-	S3	S1	-
S3	-	-	-	-	-	-	S1
S4	-	S2	S1	S2	-	-	-
S5	-	-	-	S2	-	-	-

# Casos de uso

# Casos de uso (1)

Son una forma de modelar la interacciones que tienen lugar con el software, y ayudan tanto al diseño y desarrollo como a las pruebas.

Los casos de uso tienen actores y SUT:

- Los actores son los usuarios (personas), otros sistemas o equipos que interactúan con el sistema
- SUT (*System Under Test*) es el objeto de la prueba, tanto en su conjunto como alguno de sus componentes o partes

Los casos de uso tratan de describir usos reales del sistema, desde el punto de vista del end-to-end de un actor, o el actor principal (si hay varios).

Deben seguir este esquema:

- Se lanzan con un objetivo relevante (desde el punto de vista de negocio o dominio de aplicación, como por ejemplo “ver una película”)
- Culminan con un resultado o *outcome* (“película vista y valorada”)
- Debe ser independiente de los detalles técnicos de implementación y no verse condicionado por ellos
- Se describe desde el punto de vista de las interacciones y las actividades necesarias para ir desde el comienzo hasta el final

# Casos de uso (2)

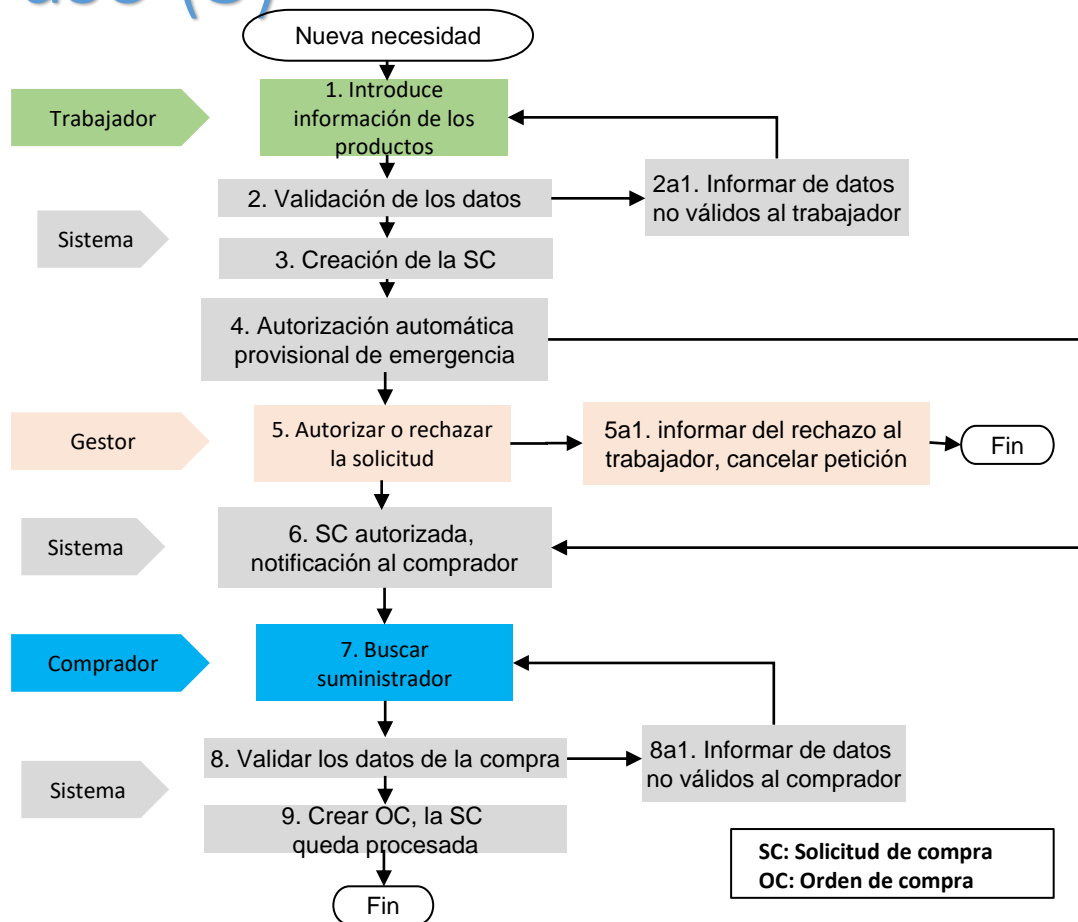
## Elementos de un caso de uso

- Precondiciones: estado inicial, punto de partida
- Poscondiciones: estado final, resultados esperados
- Pasos descritos con diagramas y el uso de lenguaje natural cuando sea necesario
- Interacciones con los actores, que pueden cambiar el estado del sistema bajo prueba, por lo que puede ser de ayuda contar con un diagrama de estados
- Los comportamientos (interacciones y actividades) se modelan con flujos o caminos
  - El flujo principal (o normal, o básico, o “happy path”) que además es único
  - Flujos o caminos alternativos: gestión de errores, salidas alternativas, paso por otros estados, ... Son de tres tipos:
    - Pasos iniciales para luego continuar por el camino principal (para corregir errores, ...)
    - Saltándose algunos pasos, cuando hay acciones en el principal que no son necesarias puntualmente
    - Dejar el camino principal para ir a un estado final diferente, sin completar una actividad normal (no sacar dinero del cajero por falta de fondos, por ejemplo)

# Casos de uso (3)

## Ejemplo

Proceso de compras en una empresa.  
El camino principal es la línea recta vertical.





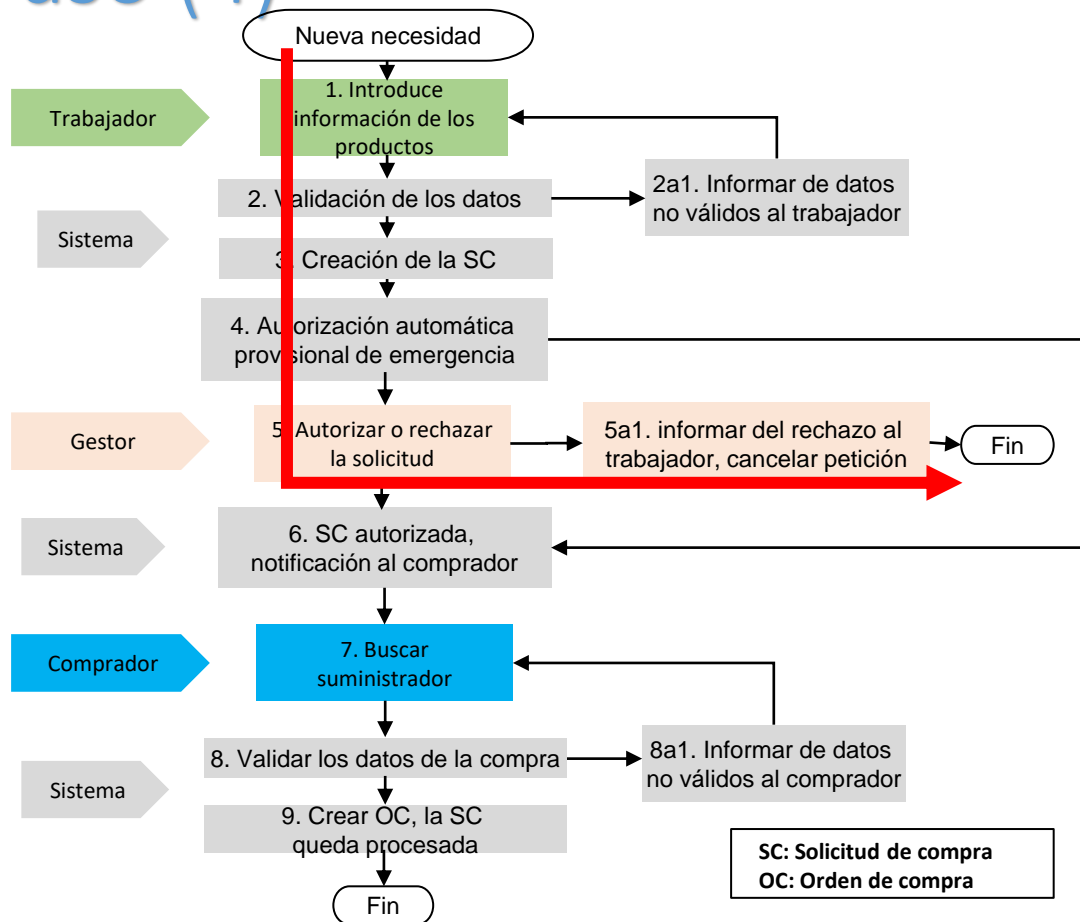
# Casos de uso (4)

## Ejemplo

El camino alternativo se marca con la línea de color



Compra no autorizada



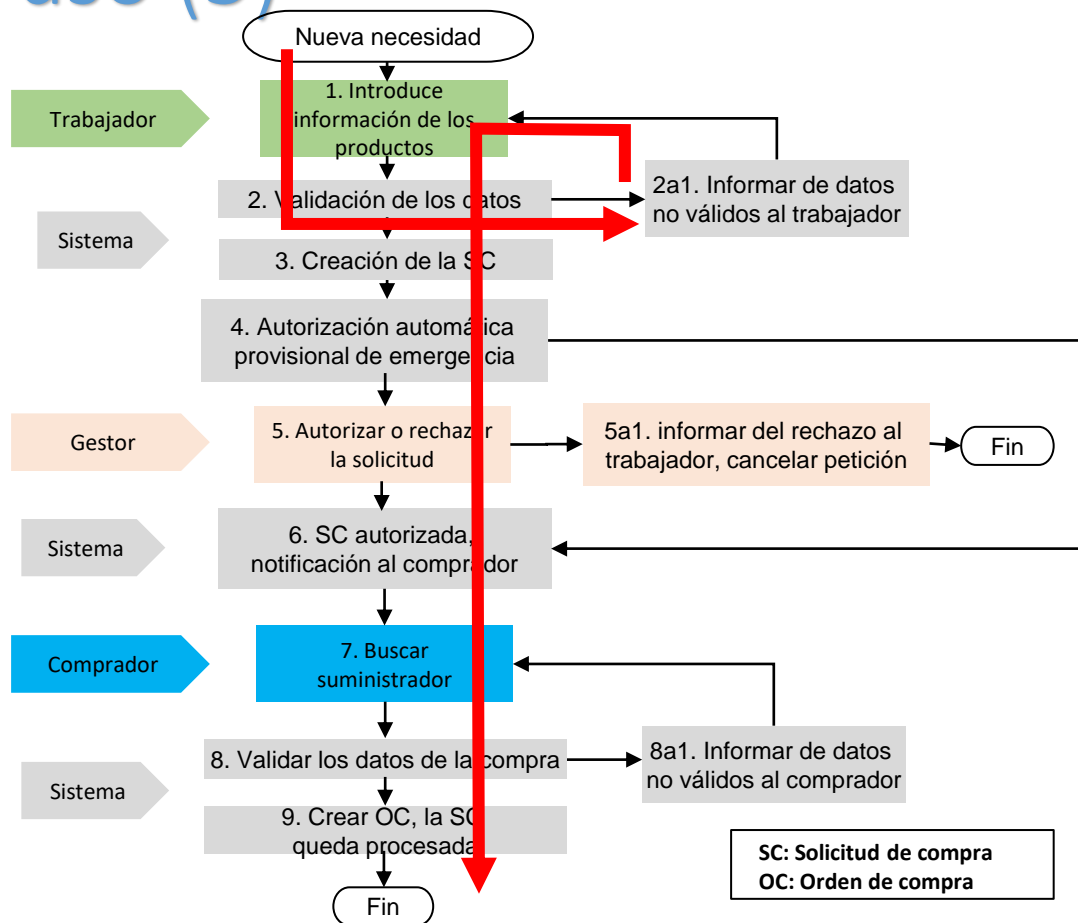
# Casos de uso (5)

## Ejemplo

El camino alternativo se marca con la línea de color



Datos no válidos

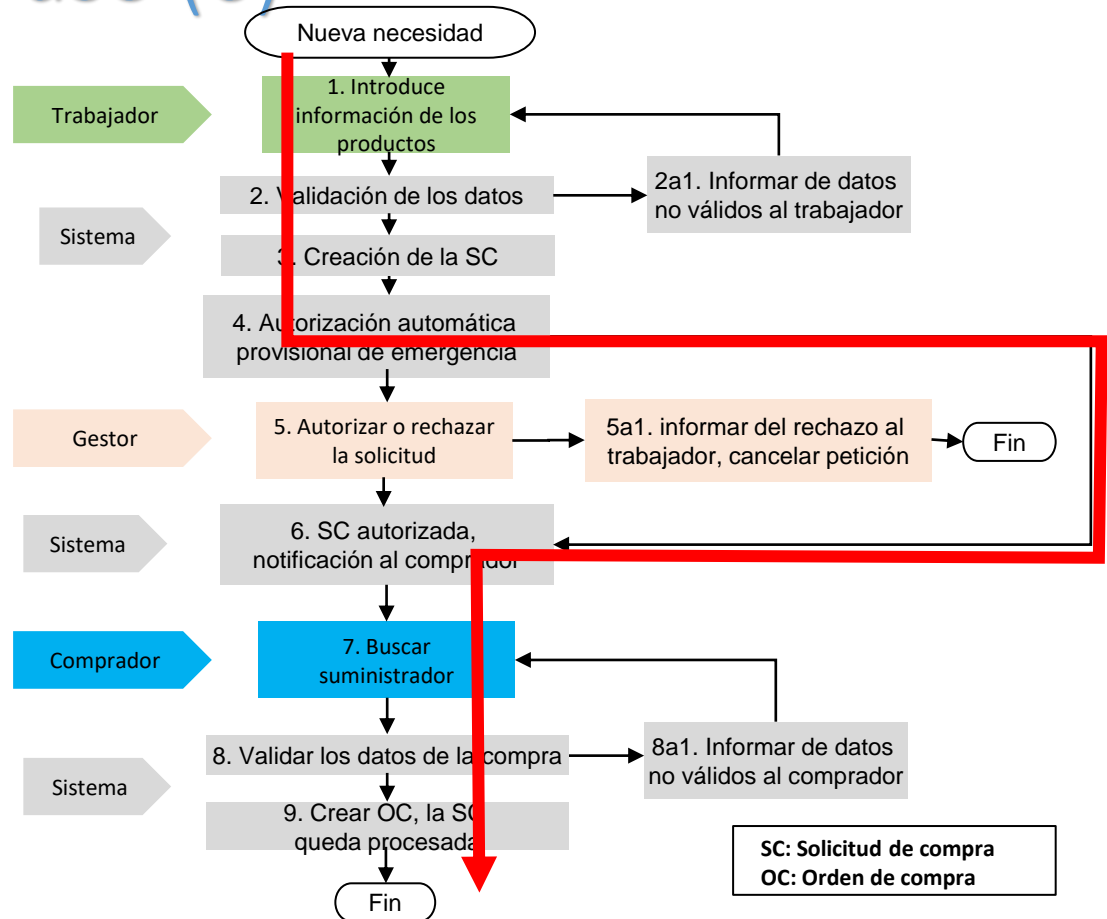


# Casos de uso (6)

## Ejemplo

El camino alternativo se marca con la línea de color

→ Aprobación de emergencia

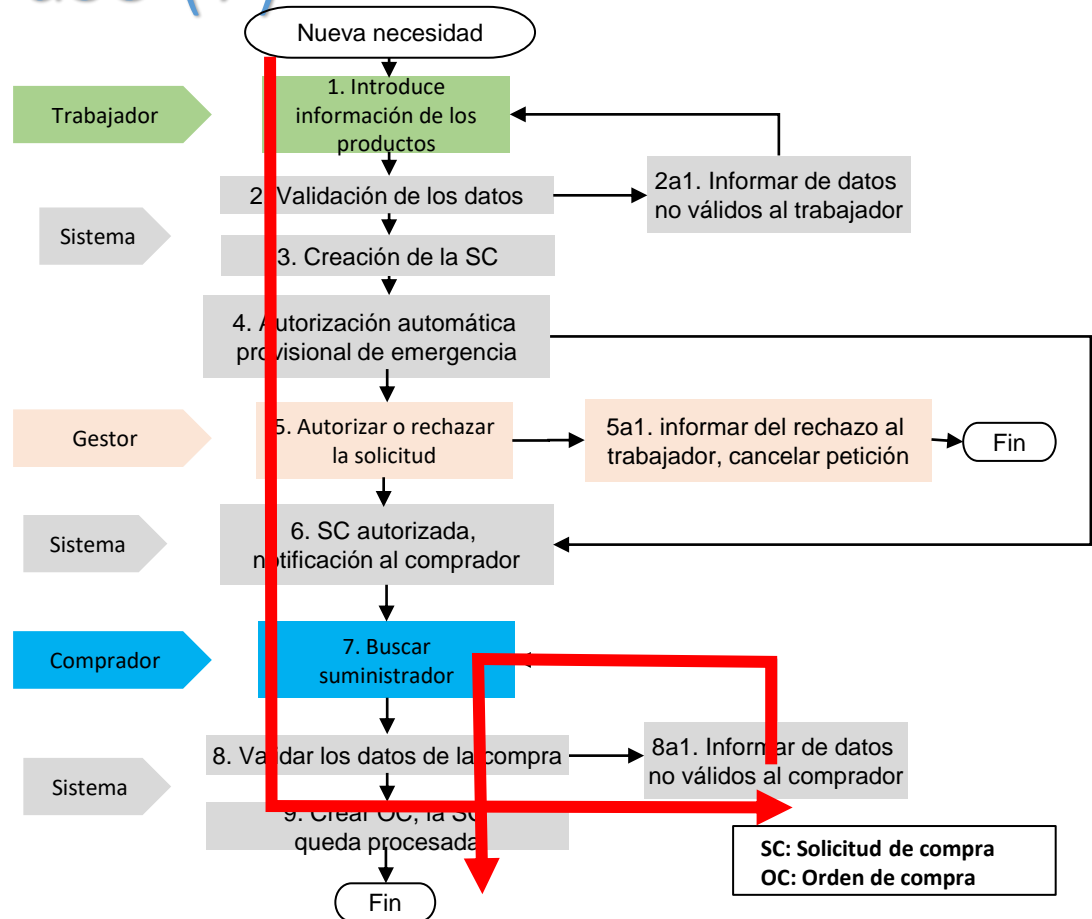


# Casos de uso (7)

## Ejemplo

El camino alternativo se marca con la línea de color

→ Datos de compra no válidos



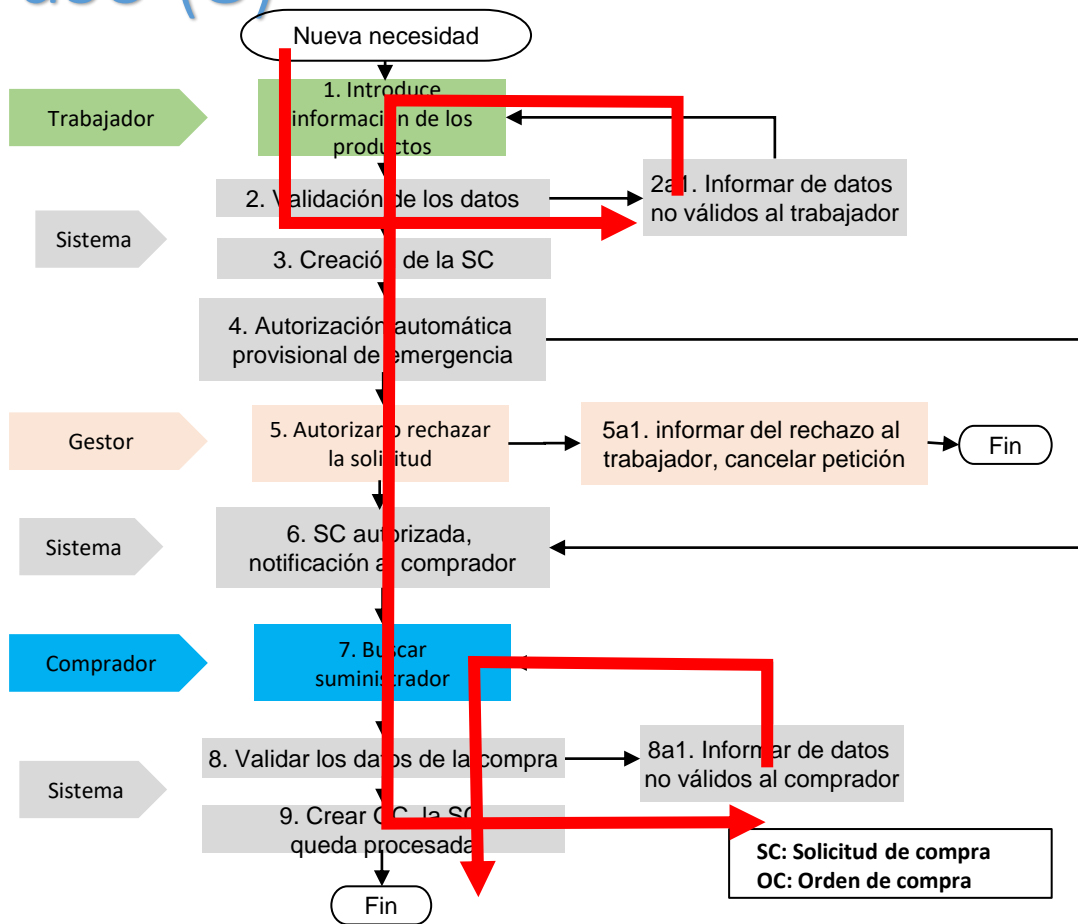
# Casos de uso (8)

## Ejemplo

El camino alternativo se marca con la línea de color



Combinación de los dos escenarios con datos no válidos



# Casos de uso (9)

## Ejemplo de caso uso en formato texto

- Identificador: CU001
- Objetivo: proceso de solicitud y orden de compra
- Actores: Trabajador, Gestor, Sistema, Comprador
- Precondiciones: todos los actores están registrados con permisos suficientes para realizar las operaciones
- Poscondiciones: Orden de Compra creada y Solicitud de compra procesada
- Escenario principal:
  - **Empleado** introduce información de los productos que quiere comprar
  - **Sistema** valida los datos y crea la solicitud
  - El **Gestor** del empleado valida la solicitud
  - El **Sistema** notifica al Comprador

- El **Comprador** busca suministrador
- El **Sistema** valida los datos de la compra
- El **Sistema** crea la Orden de Compra, la Solicitud queda procesada

## Escenario alternativo:

- **Empleado** introduce información de los productos que quiere comprar
- El **sistema** informa de datos no válidos y vuelve al trabajador
- **Empleado** introduce información de los productos que quiere comprar
- **Sistema** valida los datos y crea la solicitud
- El **Gestor** del empleado valida la solicitud
- El **Sistema** notifica al Comprador
- El **Comprador** busca suministrador
- El **Sistema** valida los datos de la compra
- El **Sistema** crea la Orden de Compra, la Solicitud queda procesada

# Casos de uso (10)

## **Cobertura**

La cobertura es porcentaje de escenarios o “comportamiento” probados con respecto al total.

Cada “comportamiento” es uno de casos de uso, de principio a fin.

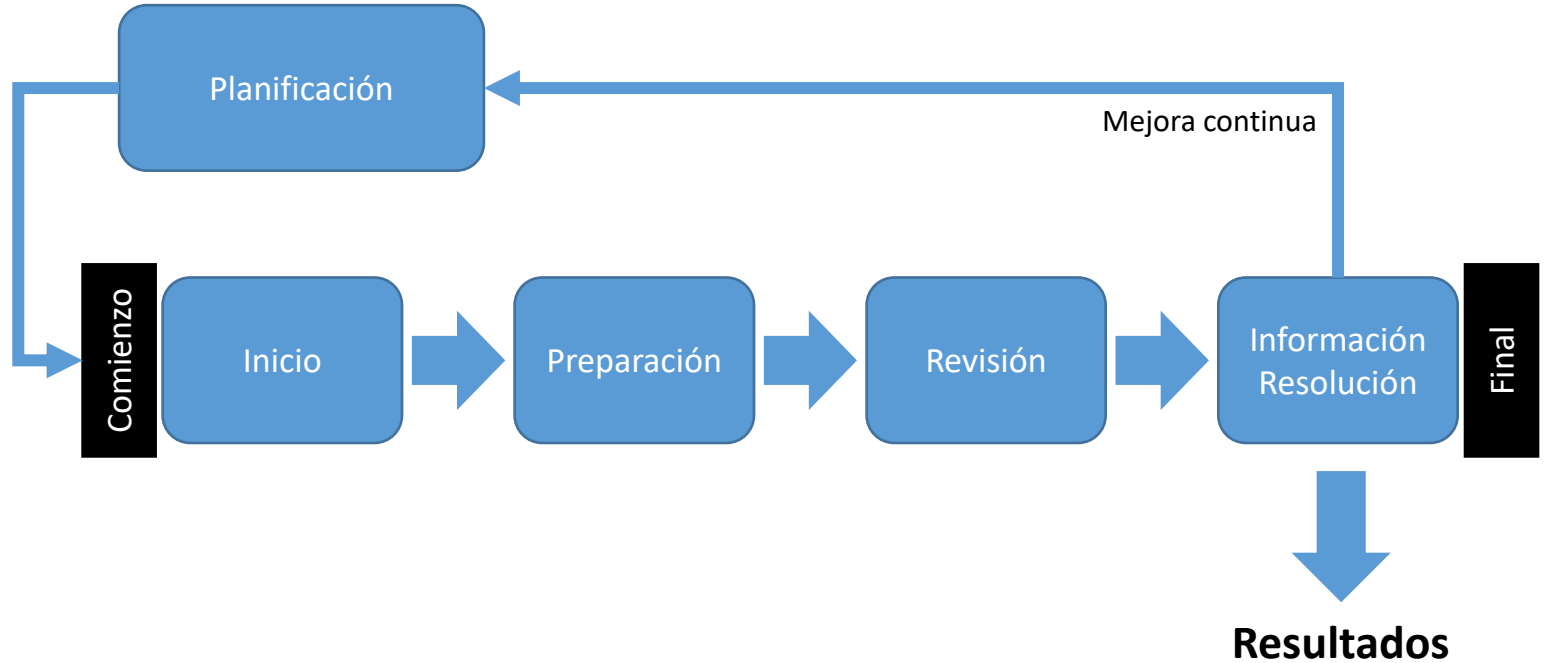
Cada escenario o comportamiento puede estar formado por una combinación de alternativas.

Normalmente se validan antes las más frecuentes o que aporten más valor, y se va ampliando progresivamente el número. Este es un caso en el que hay que estar atentos a la “paradoja del pesticida”, es decir, que no esperemos encontrar nuevos defectos si hacemos siempre las mismas pruebas.

# **Anexo. Información adicional**



# Proceso de revisión (4)



# Factores que impactan en la revisión

## **Determinan la efectividad del proceso y sus resultados**

- Tener o no objetivos claros
- Técnica de revisión seleccionada
- División adecuada del trabajo (mejor en bloques pequeños)
- Frecuencia del feedback
- Información: cantidad, formato, frecuencia, ...
- Apoyo de la dirección, rol de la revisión en los procesos de la organización
- Contar con las personas apropiadas (por ejemplo, incluir testers)
- Entorno seguro y de confianza, cultura de aprendizaje: encontrar defectos no es un problema, no se trata de criticar a las personas, ...
- Facilitación adecuada
- Información adecuada y con antelación

# Técnicas en revisión (1)

Hay varias técnicas que se pueden aplicar en las distintas partes del proceso de revisión en función del tipo de revisión que se esté usando:

## **Ad hoc**

Habitual en revisiones informales, se basa en revisar el contenido, documento, código ... secuencialmente sin apenas guía, documentando lo que se encuentre a medida que se avanza.

Requiere poca preparación, pero a cambio depende mucho de la habilidad de la persona revisora. Si la usan varias personas para la preparación habrá posiblemente duplicidades.

## **Checklist**

Es una forma sistemática de detectar problemas. Las checklists contienen preguntas o listas de puntos donde hay que poner el foco de atención.

Generalmente se recurre a la experiencia previa para elaborar esas checklists, pero es necesario mantenerlas actualizadas, por ejemplo a partir del resultado de revisiones previas.

Las checklists son una ayuda pero también pueden ser un riesgo si no se atiende a elementos que pueden no estar recogidos en ellas: hay que “salir del guión” a veces.

# Técnicas en revisión (2)

## **Escenarios y “dry runs”**

Se parte de casos de uso o situaciones reales. Se trata de reproducir el comportamiento del producto en la situación prevista para así determinar si se ajusta o no a lo esperado. Es mejor que no sea la única técnica aplicada.

## **Lectura basada en perspectiva (perspectiva-based reading)**

Técnica muy efectiva que resulta apropiada para la preparación individual previa. Consiste en adoptar el punto de vista de los distintos stakeholders en relación al producto. Ayuda mucho también en la generación de productos o elementos adicionales (como criterios de aceptación).

## **Basada en roles**

Similar a la anterior pero pensada para reuniones de revisión en la que cada asistente adopta los puntos de vista de stakeholders diferentes, que además se pueden matizar con otros rasgos (administrador novato, manager experimentada, ...).

# Elección de técnicas (2)

## Factores

Cómo de exhaustivo debe ser el proceso:

- Complejidad del sistema y sus componentes
- Estándares y regulaciones
- Requisitos contractuales o del cliente
- Riesgos

Características del proceso y el producto:

- Tipo de componente o sistema
- Tipos de defectos esperados
- Tipos de riesgos
- Uso esperado del producto

Limitan la variedad en la elección:

- Documentación disponible
- Conocimiento y habilidades del equipo
- Experiencia previa usando distintas técnicas
- Herramientas disponibles

Y además:

- Modelo de ciclo de vida
- Objetivos
- Tiempo y presupuesto

# Elección de técnicas (3)

Algunas pueden usarse en todos los niveles, mientras que otras sólo tienen sentido en determinados niveles o situaciones.

Se suelen alcanzar los mejores resultados usando combinaciones de técnicas.

El uso que se haga de ellas puede ir desde muy formal (muy documentado y procedimentado) a muy informal, con múltiples estados intermedios.

Hay que recordar de nuevo el principio “El testing depende del contexto”.

Factores de contexto ya mencionados son:

- La madurez del proceso y el equipo.
- Las restricciones de tiempo o presupuesto.
- Los requisitos legales, contractuales o regulatorios.
- Conocimientos y habilidades de las personas implicadas.
- El modelo de ciclo de vida del software.

**Verificación del Software**

## **3.6 Pruebas basadas en experiencia**

# Pruebas basadas en experiencia

Son un tipo especial de prueba menos sistemáticas pero más afinadas a partir del conocimiento técnico y del dominio de negocio. Pueden usar checklists y baterías de pruebas, pero tiene mucho peso también la intuición y el sentido del momento.

Las pruebas se basan en el conocimiento del sistema que se está probando, de sistemas similares, y del contexto y entorno de uso. Por eso mismo no tienen porqué estar restringidas únicamente a testers. Por ejemplo, las UAT hechas por cliente entrarían en esta categoría.

Aunque permiten probar condiciones, especialmente en el end-to-end, que a veces se escapan a técnicas más sistemáticas, la cobertura tiende a ser menos completa que con otras técnicas.

Son de tres tipos:

- Error guessing
- Tests exploratorios
- Y basadas en checklist



# Error guessing

Simples y productivas en el sentido de que pueden detectar defectos que pueden escapar a otras técnicas más sistemáticas.

Se puede decir que se basan en la idea de preguntarse “¿Qué podría salir mal?”.

Se trata de buscar defectos a partir de:

- Experiencia previa defectos encontrados en sistemas similares
- En el conocimiento de cómo funciona y las potenciales debilidades del software
- La propia intuición

Se pueden construir test cases basados en esta técnica e incorporarlos a planes de prueba convencionales.

# Test exploratorios (1)

Más que una técnica en sí, puede definirse como una aproximación a las pruebas, y puede incluir otras técnicas, tanto basadas en experiencia, como de caja negra y caja blanca-

Los test de diseñan, ejecutan y registran durante la propia ejecución de la prueba.

El resultado de las pruebas es también una forma de conocer más sobre el objeto de prueba, y ser fuente para crear otras pruebas.

De hecho, el propio resultado de una prueba, nos puede dar una pista de qué es lo que hay que probar a continuación, a medida que vamos construyendo un modelo mental del objeto que se prueba. Hay que ser conscientes de que muchas no hay un conocimiento completo del sistema que se prueba: suele haber mucho foco en sus partes, y el end-to-end (que no se puede realizar hasta que hay un nivel de completitud suficiente) ayuda a descubrir cómo es el sistema completo.

Es mejor hacer una pequeña planificación y realizarla en sesiones:

- Con objetivos definidos
- Con una duración limitada y predefinida
- Que se documentan

# Test exploratorios (2)

Si los comparamos con los sistemáticos, los exploratorios son más adecuados cuando:

- No hay especificaciones o la documentación, o estas son escasas, o si hay poco conocimiento sobre el sistema
- Si no hay mucho tiempo ni recursos para las pruebas
- Si es necesaria una revisión preliminar de calidad, y detectar riesgos y defectos, y antes de proceder a unas pruebas más sistemáticas
- Si las pruebas sistemáticas no son suficientes y dejan pasar defectos, o incluso para validar la efectividad de las pruebas sistemáticas.

En cambio, es mejor decantarse por unas pruebas sistemáticas si:

- Se dispone de información completa y detallada, especialmente de las especificaciones
- Si hay recursos disponibles
- Si es posible automatizar tests. Es imposible hacerlo con unos exploratorios
- Si la responsabilidad de la ejecución de las pruebas puede recaer en otras personas sin el conocimiento previo necesario (baterías sistemáticas de tests les ayudarán a conseguirlo)
- Y si hay requisitos que obliguen a hacerlo (por ejemplo regularorios)

# Pruebas basadas en checklists

Las checklists contiene las condiciones de prueba, y los tests se diseñan, implementan y ejecutan para cubrirlas con respecto al objeto que se analizando (SUT).

Estas checklists pueden existir de antemano, modificarse durante el análisis y ejecución, o crearse enteramente nuevas.

Estas checklists puede basarse en el conocimiento y dominio del producto software en concreto, o a partir del dominio técnico o de negocio.

Normalmente las checklists reflejan pruebas de alto nivel, no muy detalladas, y su efectividad depende mucho de las personas que hagan cada prueba.

Es mejor que estas listas reflejen un aspecto determinado del problema.

Por ejemplo, una checklist para probar un software reproductor de video incluiría:

- Play
- Pausa
- Stop
- Desplazar a un punto determinado
- Siguiente/anterior contenido
- Mute, bajar/subir volumen
- ...