

# Introducción a la Ingeniería del Software



## TEMA 1: INTRODUCCIÓN A LA INGENIERÍA DEL SOFTWARE

Grado en Ingeniería Informática



# Índice de contenidos

2

- Justificación de la Ingeniería del Software
- Definiciones
- Características del Software
- Mitos del Software
- Aspectos éticos del Software

# Índice de contenidos

3

- **Justificación de la Ingeniería del Software**
- Definiciones
- Características del Software
- Mitos del Software
- Aspectos éticos del Software

# Algunas preguntas:

4

- ¿Qué pasaría si el ingeniero civil o el arquitecto construye una casa o un edificio sin hacer sus planos, proyectos o maquetas? ¿Crees que la obra pueda concluirse cubriendo las necesidades, con la calidad necesaria y a tiempo?
- ¿Permitirías que tu propio cirujano te interviniera sin hacer los estudios respectivos para obtener las evidencias del problema de salud que te aqueja?
- ¿Permitirías a tu abogado que te defendiera sin conocer las pruebas y sin un plan para tu defensa?

# En cambio:

5

- ¿Por qué los ingenieros en Informática cedemos al "chantaje de la falta de tiempo" y construimos software sin el análisis y diseño expresado en un proyecto, más allá de las ideas existentes "en nuestra cabeza"?
- ¿Por qué ante un problema nos ponemos enseguida a codificar, sin plantearnos que hacer software es en realidad un proceso de *ingeniería*?: **Ingeniería en Informática: Software, Sistemas, Inteligencia Artificial.**

# Ingeniería del Software

6

- Definición (IEEE Standard 610.12):
  - *The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.*
- Definición:
  - *Software Engineering is an engineering discipline that is concerned with all aspects of software production from the early stages of system specification to maintaining the system after it has gone into use.*

# El software es complejo

7

- **Ejemplos:**
  - Sistema software para el control de un avión de pasajeros
  - Comercio Web (ej.: Amazon)
  - Juegos de ordenador de última generación
- **Intuitivamente, hacen falta:**
  - Equipos de personal cualificado
  - Una disciplina de ingeniería
- **Ingeniería del software: análisis, especificación, planificación, diseño, codificación, pruebas, mantenimiento**

# Costes del Software

8

- **Algunas evidencias:**
  - El coste del software frecuentemente predomina en coste global de un sistema informático
  - El coste de mantenimiento del software es mayor que el coste de desarrollarlo
    - ✖ En sistemas con una vida larga los costes de mantenimiento pueden ser varias veces el coste de su desarrollo
- **Ingeniería del software: producir software con un coste adecuado**

# Comparación

9

- Ingeniería del Software vs Ingeniería de caminos
  - Construir un producto software vs construir un puente
- Coincidencias
  - El tamaño importa
  - Trabajo en equipo de gente con distinto perfil
  - Parecidas dificultades antes los cambios de diseño
  - Parecidas consecuencias ante fallos
  - Términos comunes: diseño, arquitectura, componentes, planificación, ...

# Índice de contenidos

10

- Justificación de la Ingeniería del Software
- **Definiciones**
- Características del Software
- Mitos del Software
- Aspectos éticos del Software

# Definiciones

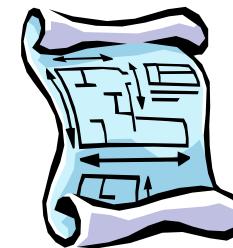
11

- ¿Qué es el software?
- Características del software
- ¿Qué es la Ingeniería del Software?
- ¿Qué es un proceso software?
- ¿Qué es un modelo de proceso software?
- ¿Cuáles son los costes de la Ingeniería del Software?
- ¿Qué es CASE (Computer-Aided Software Engineering)?

# ¿Qué es el software?

12

- Software es un conjunto de:
  - Programas de ordenador
  - Documentación asociada
  - Datos



# ¿Qué es el software?

13

- El software puede ser desarrollado
  - Para un mercado general
    - Desarrollados para ser vendidos a un amplio rango de clientes (ejemplos: Microsoft Office, Adobe Photoshop, etc.)
  - Para un cliente particular (software a medida)
    - Desarrollados para un cliente concreto de acuerdo a sus especificaciones
- Un software nuevo se puede crear
  - Desarrollando nuevos programas
  - Reconfigurando sistemas software genéricos
  - Reutilizando software existente

# Características del software

14

- El software se desarrolla
  - No se fabrica en un sentido clásico
  - Ejemplo: fabricar un automóvil

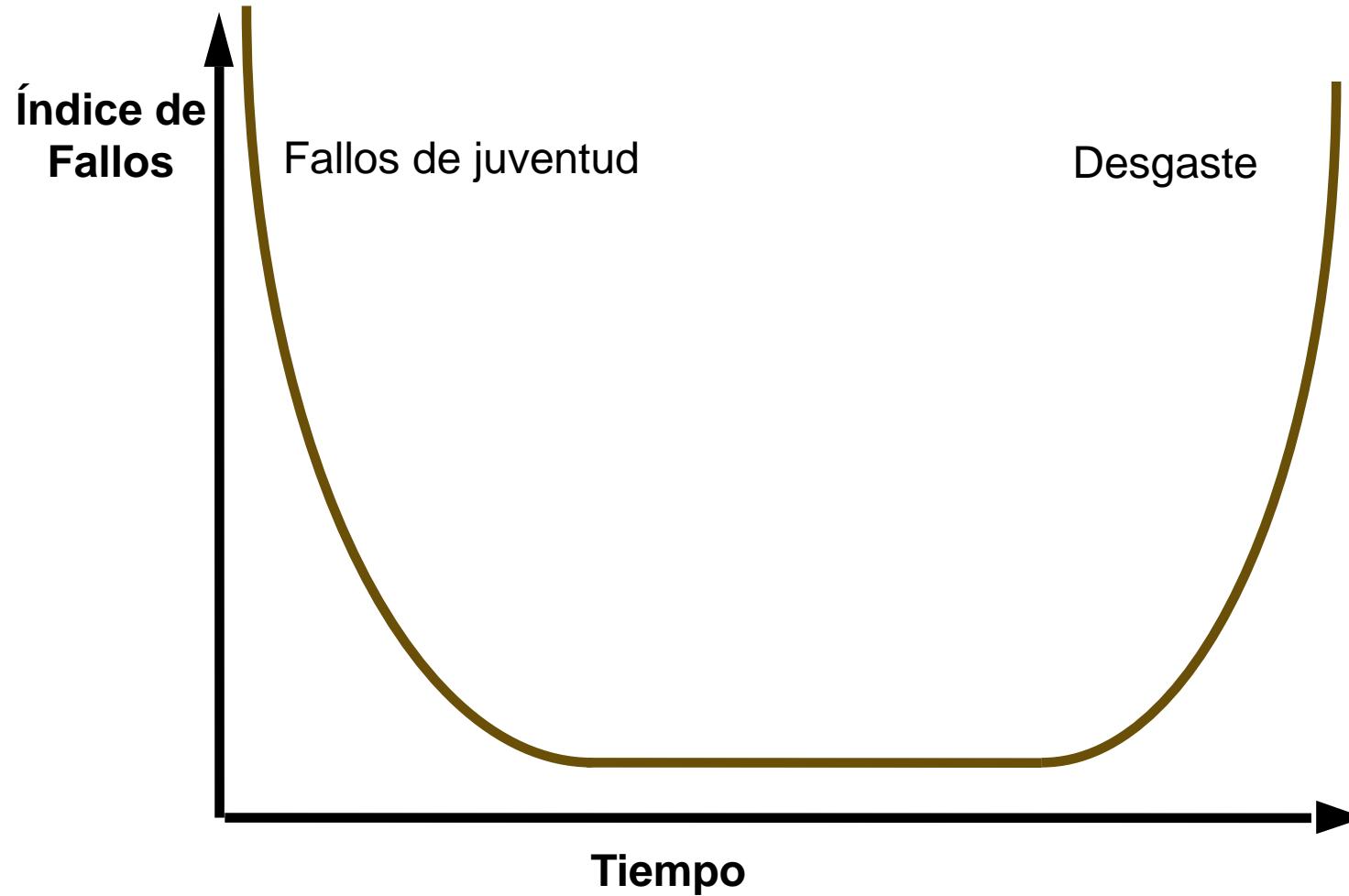


- El software no se estropea ni se desgasta por el uso
  - No existen repuestos



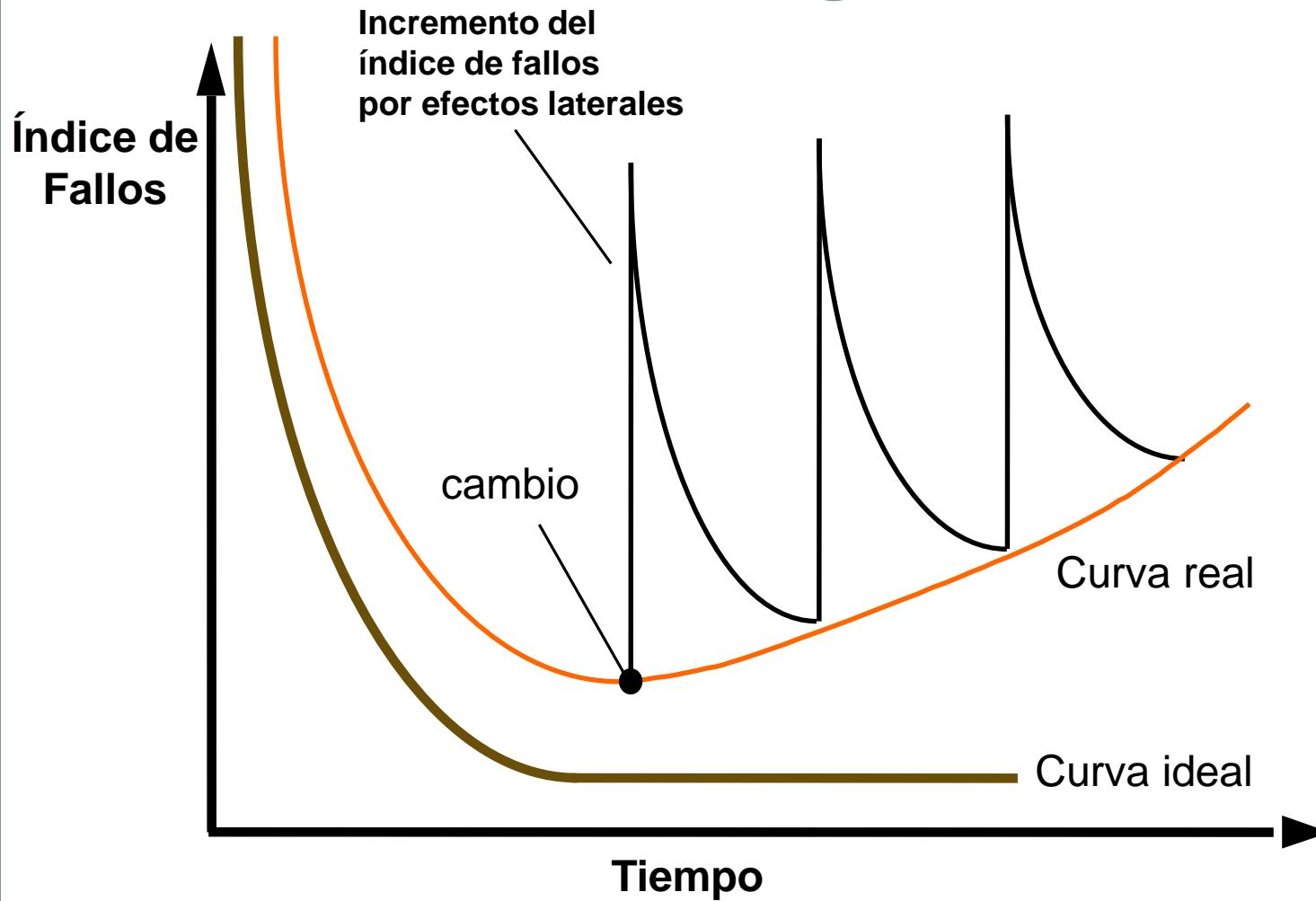
# Curva de fallos del hardware

15



# Curva de fallos del software

16



# Características del software

17

- La mayor parte del software existente se construye a medida
  - Aunque la industria software trata de fomentar el uso de componentes que se puedan reutilizar

“ anything  
is possible ”



# ¿Qué es la Ingeniería del Software?

18

- **Más definiciones:**
  - Es una disciplina de ingeniería que trata con todos los aspectos de la producción de software de calidad
  - *Is the establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines*
- **Los ingenieros software**
  - deben adoptar una forma de trabajo sistemática y organizada y utilizar herramientas y técnicas dependiendo del problema a resolver, las restricciones de desarrollo y los recursos disponibles

# ¿Qué es un proceso software?

19

- Es un conjunto de actividades cuyo objetivo es el desarrollo o la evolución de un sistema software
- Las actividades genéricas de un proceso software son (Pressman 2010)
  - Comunicación: para saber qué quiere el cliente
  - Planificación: descripción de las tareas técnicas a llevar a cabo
  - Modelado: creación de modelos para entender los requisitos software y el diseño que los llevará a cabo
  - Construcción: generación de código (programas) y su testeo para eliminar errores (*software testing*)
  - Implantación: entrega al cliente para su puesta en marcha

# ¿Qué es un modelo de proceso software?

20

- Los procesos software son complejos
  - Se suelen proponer modelos de procesos software
- Un modelo de proceso software es una representación simplificada de un proceso de software, presentado desde una perspectiva específica
- Ejemplos de perspectivas de proceso:
  - Perspectiva del flujo de trabajo – la secuencia de actividades
  - Perspectiva del flujo de datos – flujo de información
  - Perspectiva de rol/acción – quién hace qué

# ¿Qué es un modelo de proceso software?

21

- Ejemplos de modelos de procesos software:
  - Modelo en cascada
  - Desarrollo iterativo
  - Desarrollo software basado en componentes
  - Métodos ágiles
  - Proceso unificado

# ¿Cuáles son los costes de la ingeniería del software

22

- Aproximadamente el 60% de los costes son costes de desarrollo, el 40% son costes de pruebas
- Para software personalizado, los costes de evolución suelen superar a los costes de desarrollo
- Los costes varían dependiendo del tipo de sistema a desarrollar y de los requisitos para ciertos atributos del sistema como el rendimiento o la fiabilidad
- La distribución de los costes depende del modelo de desarrollo que se utilice

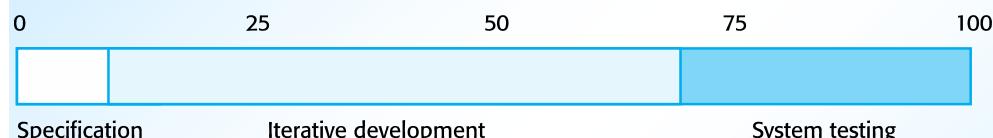
# Distribución de costes por actividad

23

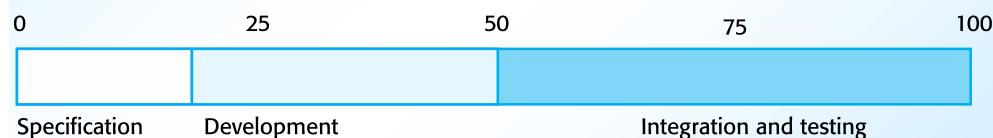
## Waterfall model



## Iterative development



## Component-based software engineering



## Development and evolution costs for long-lifetime systems



# ¿Qué son las herramientas CASE?

24

- *Computer-Aided Software Engineering*
- Son herramientas software destinadas a proporcionar soporte para las actividades de un proceso de software
- Dos tipos:
  - Upper-CASE: herramientas de soporte de las actividades iniciales del proceso (requisitos, diseño)
  - Lower-CASE: herramientas de soporte de actividades posteriores (programación, depuración, pruebas)

# Índice de contenidos

25

- Justificación de la Ingeniería del Software
- Definiciones
- **Características del Software**
- Mitos del Software
- Aspectos éticos del Software

# Características del software

26

- Un buen software debe proporcionar la funcionalidad requerida por el cliente y además tener las siguientes características:
  - Mantenibilidad
    - Para poder evolucionar para satisfacer nuevas necesidades
  - Fiabilidad
    - Debe estar libre de errores
  - Eficiencia
    - Los recursos no han de ser malgastados
  - Aceptación
    - Por los usuarios finales (no por los desarrolladores)

# Diversidad del software

27

- Aplicaciones autónomas
- Sistemas empotrados
- Sistemas de procesamiento por lotes
- Entretenimiento
- Simulación
- Aplicaciones Web
- Sistemas recolectores de datos
- Aplicaciones basadas en transacciones

# Índice de contenidos

28

- Justificación de la Ingeniería del Software
- Definiciones
- Características del Software
- **Mitos del Software**
- Aspectos éticos del Software

# Mitos del software

29

- Mitos de gestión de proyectos

- “Si vamos mal de tiempo, podemos añadir más programadores para avanzar más rápido”
- “Si decidimos subcontratar el proyecto a un tercero podemos confiar en que lo harán bien”
- “Para saber lo que hay que hacer podemos confiar en los libros que contienen estándares y procedimientos para construir software”

# Mitos del software

30

- **Mitos del cliente**

- “Una simple especificación de objetivos es suficiente para empezar a trabajar – los detalles se incluirán con posterioridad”
- “Los requisitos de un sistema software cambian continuamente, pero es fácil hacer cambios en el software porque éste es flexible”

# Mitos del software

31

- **Mitos de los profesionales del software**

- “Una vez que el programa compila y funciona, ya hemos terminado nuestro trabajo”
- “Hasta que no se tenga el programa en funcionamiento no se puede determinar su calidad”
- “El único resultado válido de un producto software es el programa resultante”
- “Los ingenieros software tienden a escribir grandes cantidades de documentación que no sirve para nada, excepto para ralentizar la marcha del proyecto”

# Índice de contenidos

32

- Justificación de la Ingeniería del Software
- Definiciones
- Características del Software
- Mitos del Software
- **Aspectos éticos del Software**

# Ética en Ingeniería del Software

33

- Aspectos éticos a considerar dentro de la Ingeniería del Software:
  - Confidencialidad
  - Competencia
  - Derechos de propiedad intelectual
  - Mal uso de los ordenadores
- ACM/IEEE-CS Joint Task Force on Software Engineering Ethics and Professional Practices
  - Software Engineering Code of Ethics and Professional Practice
  - <http://www.acm.org/about/se-code#short>

# Introducción a la Ingeniería del Software



## TEMA 3: PROCESOS SOFTWARE

Grado en Ingeniería Informática



# Índice de contenidos



- Introducción
- Actividades dentro de un proceso software
- Modelos de proceso software
- Modelos clásicos
- Modelos especializados
- Métodos ágiles

# Índice de contenidos

3

- **Introducción**
- Actividades dentro de un proceso software
- Modelos de proceso software
- Modelos clásicos
- Modelos especializados
- Métodos ágiles

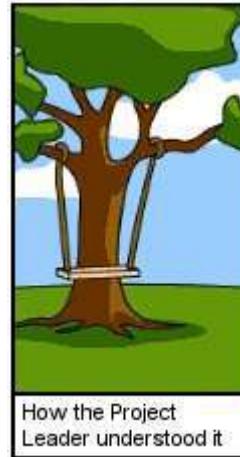
# La paradoja del columpio

4

- Desarrollo de un columpio según la Ing. Softw.



How the customer explained it



How the Project Leader understood it



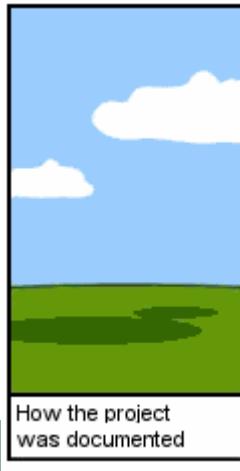
How the Analyst designed it



How the Programmer wrote it



How the Business Consultant described it



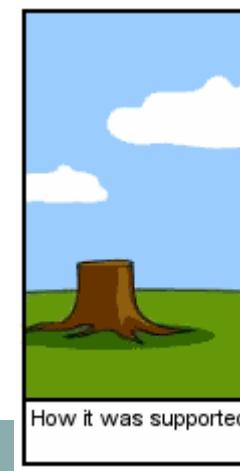
How the project was documented



What operations installed



How the customer was billed



How it was supported

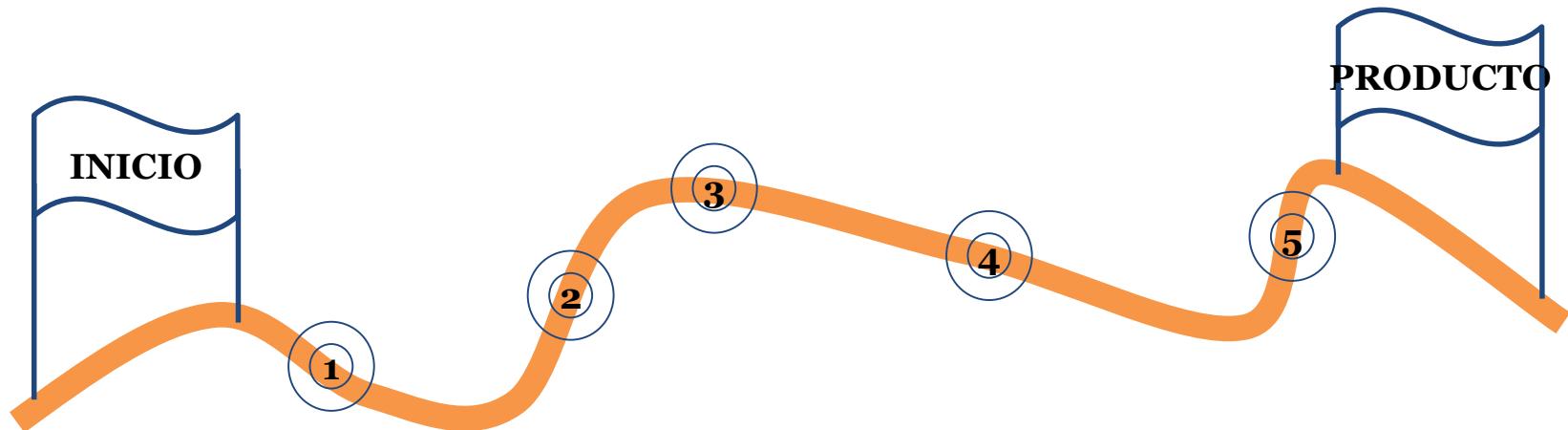


What the customer really needed

# Procesos del software

5

- Constituyen un conjunto de actividades cuya meta es el desarrollo del software (producto)
- De manera informal,
  - el proceso del software es como un “mapa de carreteras” o una “hoja de ruta” que marca los pasos que deben seguirse para la obtención del software



# Procesos del software

6

- Aunque existen muchos procesos diferentes de software
  - Algunas actividades básicas son comunes a todos ellos
- Para afrontar la resolución de cualquier problema hay que plantearse:
  - Entender el problema
  - Pensar una solución
  - Llevar a cabo un plan
  - Examinar el resultado y comprobar que el problema se ha resuelto de forma satisfactoria

# Procesos del software

7

- En el contexto de la ingeniería del software
  - Entender el problema es tarea de la INGENIERÍA DE REQUISITOS
- Aspectos a considerar:
  - Estudio de viabilidad
  - Obtención y análisis de requisitos
  - Especificación (modelado del análisis)
  - Validación de requisitos

# Procesos del software

8

- En el contexto de la ingeniería del software
  - Pensar una solución es tarea del DISEÑO SOFTWARE
- Aspectos a considerar:
  - La estructura del SW que se va a implementar
  - Los datos que forman parte del sistema
  - Las interfaces entre los componentes del sistema
  - Los algoritmos utilizados

# Procesos del software

9

- En el contexto de la ingeniería del software
  - Llevar a cabo un plan es tarea de la IMPLEMENTACIÓN DEL SOFTWARE
- Aspectos a considerar:
  - Se debe producir software que cumpla su especificación

# Procesos del software

10

- En el contexto de la ingeniería del software
  - Comprobar que se ha resuelto el problema es tarea de las fases de VALIDACIÓN Y EVOLUCIÓN
- Aspectos a considerar:
  - El software ha de hacer lo que el cliente desea
  - El software debe evolucionar para satisfacer las necesidades cambiantes del cliente

# Índice de contenidos

11

- Introducción
- Actividades dentro de un proceso software
- Modelos de proceso software
- Modelos clásicos
- Modelos especializados
- Métodos ágiles

# Actividades genéricas de un proceso software

12

- Las actividades genéricas de un proceso software son (Pressman 2010)
  - Comunicación: para saber qué quiere el cliente
  - Planificación: descripción de las tareas técnicas a llevar a cabo
  - Modelado: creación de modelos para entender los requisitos software y el diseño que los llevará a cabo
  - Construcción: generación de código (programas) y su testeo para eliminar errores (*software testing*)
  - Implantación: entre al cliente para su puesta en marcha

# Actividades genéricas de un proceso software

13

- Las actividades genéricas de un proceso software son (Sommerville 2011)
  - Especificación:
    - Qué debe hacer el sistema y las restricciones a considerar
  - Desarrollo:
    - Producir el sistema software
  - Validación:
    - Verificar que el software es lo que quiere el cliente
  - Evolución:
    - Modificar el software en respuesta a necesidades cambiantes

# Actividades complementarias

14

- **Otras actividades:**

- Gestión de proyectos
- Gestión de riesgos
- Control de calidad
- Revisiones técnicas
- Gestión de configuraciones

# Índice de contenidos

15

- Introducción
- Actividades dentro de un proceso software
- Modelos de proceso software
  - Modelos clásicos
  - Modelos especializados
  - Métodos ágiles

# Modelos de procesos software

16

- Constituyen una representación abstracta (simplificada) de un proceso software
- Debido a la amplia diversidad de tipos de aplicaciones software
  - No existe un proceso de software ideal
  - Hay modelos distintos que se adaptan al tipo de software
  - Si el modelo no es adecuado se presentan dificultades (burocracia innecesaria, problemas durante el desarrollo, etc.)

# Modelos de procesos software

17

- ¿Cómo saber si un modelo es apropiado?
  - Existen algunos indicadores:
    - Calidad del producto resultante
    - El tiempo de entrega se ajusta al previsto
    - La viabilidad a largo plazo del producto que se construye

# Modelos de procesos software

18

- En la ingeniería del software existen varios modelos genéricos
  - No son descripciones detalladas al 100%, sino abstracciones que se pueden usar para explicar diferentes enfoques de desarrollo
  - Proporcionan una guía útil
- Cuando se decide usar uno de estos modelos hay que:
  - Detallar las tareas a realizar para alcanzar las metas de desarrollo
  - Adaptar el modelo de proceso resultante y ajustarlo a la naturaleza específica del proyecto en cuestión

# Modelos de procesos software

19

- Modelos clásicos
  - Modelo en cascada
  - Prototipos
  - Desarrollo en espiral
  - Desarrollo incremental
  - El proceso unificado (RUP)
- Modelos especializados
  - Basado en componentes
  - Métodos formales
- Métodos ágiles
  - Programación extrema (XP)
  - Scrum

# Índice de contenidos

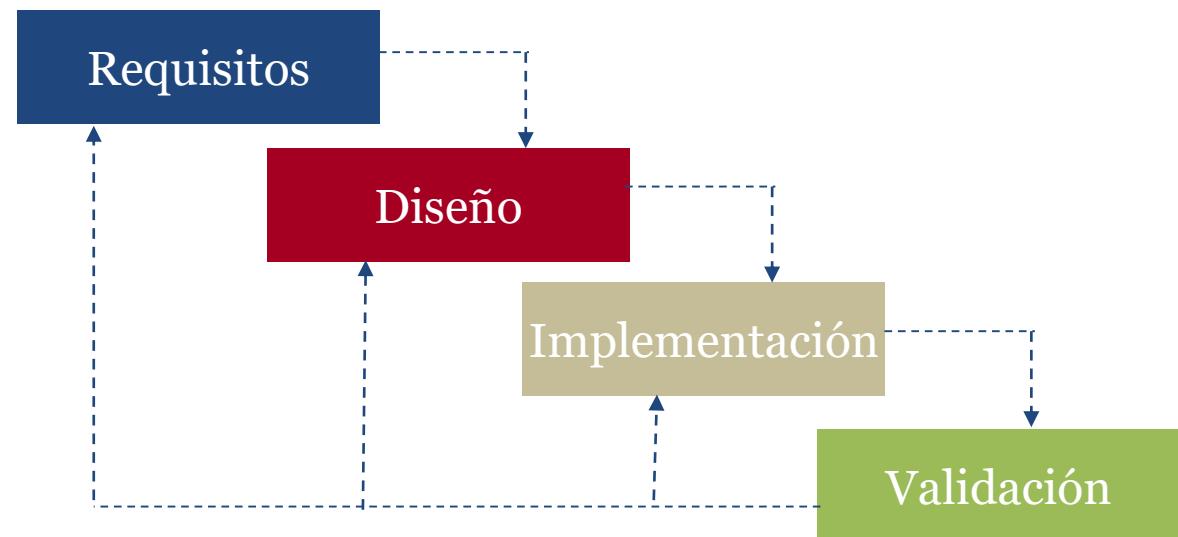
20

- Introducción
- Actividades dentro de un proceso software
- Modelos de proceso software
  - Modelos clásicos
  - Modelos especializados
  - Métodos ágiles

# Modelo en cascada

21

- Ciclo de vida clásico. Propuesto por Royce.
- Modelo secuencial que se aplica partiendo de la especificación hasta su finalización
- Las actividades fundamentales se suceden a lo largo del desarrollo



# Modelo en cascada

22

- **Ventajas**

- Adecuado cuando los requisitos están totalmente claros al inicio

- **Inconvenientes**

- Los proyectos reales raramente siguen el flujo secuencial del modelo
    - El cliente a veces no conoce todos los requisitos al inicio
    - Miembros del equipo deben esperar a que terminen tareas precedentes
  - El cliente debe tener paciencia
  - Poco flexible

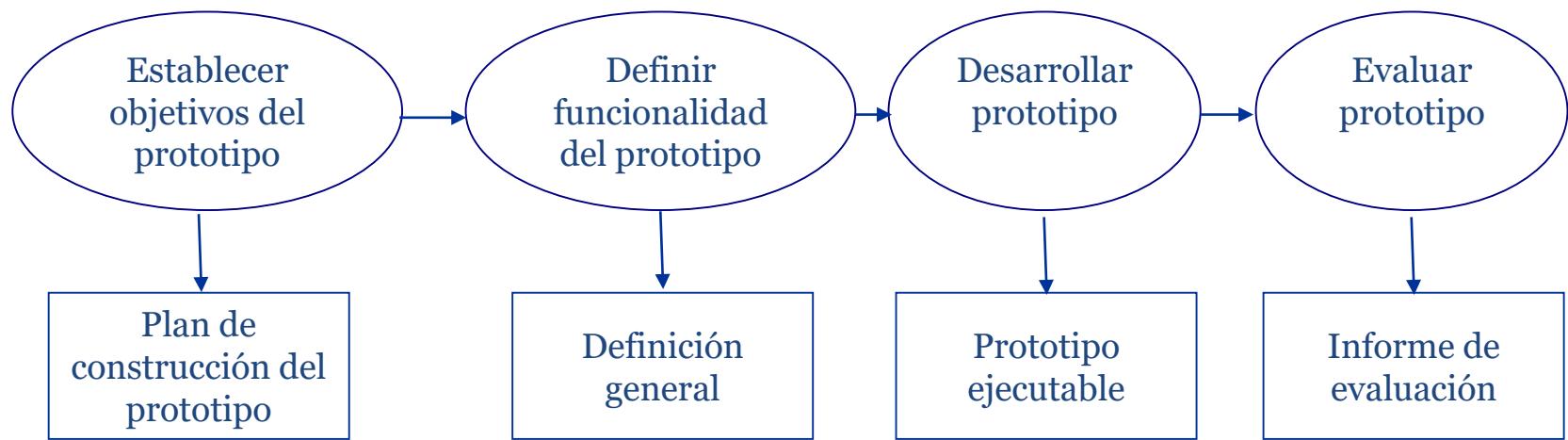
# Prototipos

23

- Un prototipo es una versión inicial del software que se utiliza para informarse más del problema y sus posibles soluciones
  - Es un “Diseño rápido” centrado en los aspectos visibles para el cliente
- Un prototipo se puede usar de varias maneras en un proceso de desarrollo de software:
  - Un prototipo está disponible de forma rápida y se puede enseñar al cliente
  - Obtención y validación de requisitos
  - Explorar soluciones de diseño
  - Ejecutar pruebas

# Prototipos

24



# Prototipos

25

## • Inconvenientes

- El prototipo no tiene por qué usarse como el sistema final
- Puede ser imposible hacer que el prototipo cumpla requisitos no funcionales (rendimiento, seguridad, fiabilidad, etc.)
- Los estándares de calidad se suelen relajar en un prototipo
- Los prototipos se suelen hacer rápido
  - ✖ Poca documentación
  - ✖ Puede dificultar el mantenimiento a largo plazo

# Desarrollo en Espiral

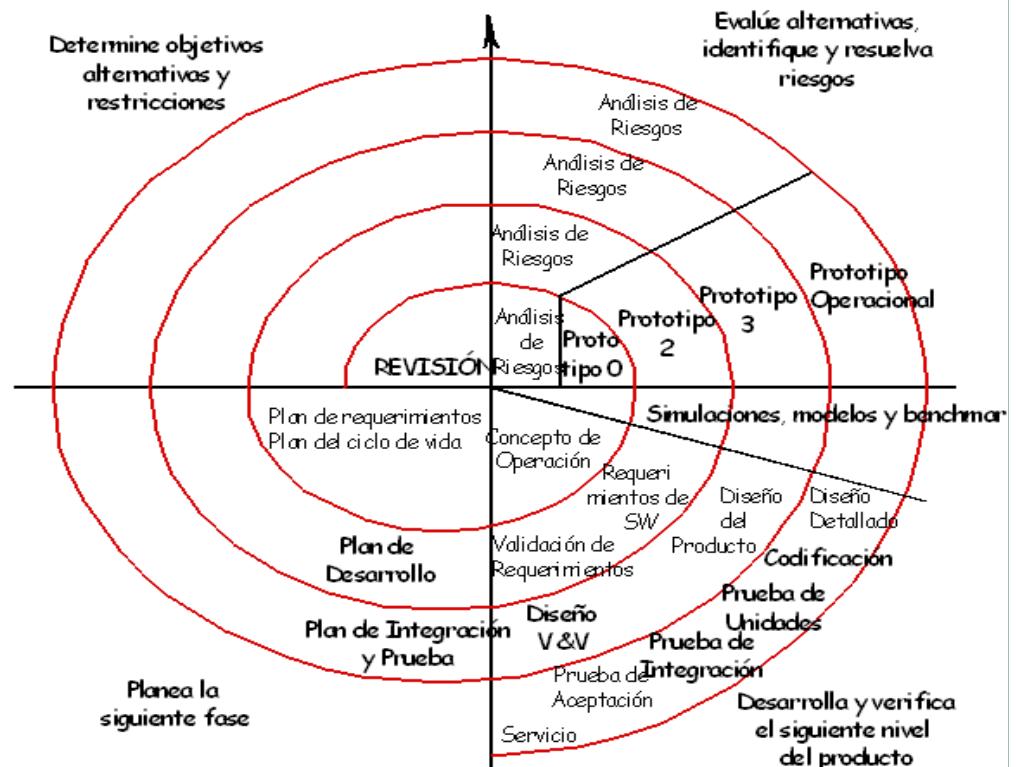
26

- Propuesto por Boehm 1986. A spiral Model of Software Development and Enhancement.
- Tiene en cuenta el riesgo en el desarrollo.
  - Primero mira las diferentes alternativas de desarrollo.
  - Se opta por la de riesgo más asumible.
  - Se hace un ciclo en espiral.
  - Si el cliente quiere hacer más mejoras volvemos a comenzar, hasta que no necesite mejorarse.
- Se puede interpretar que cada ciclo puede ser con el método en cascada.

# Desarrollo en espiral

27

- Es un modelo de tipo evolutivo
  - El proceso se presenta como una espiral y no como una secuencia de actividades
  - Cada ciclo en la espiral representa una iteración en el proceso
  - Los riesgos se evalúan de forma explícita a lo largo del proceso



# Desarrollo en Espiral. Ciclos

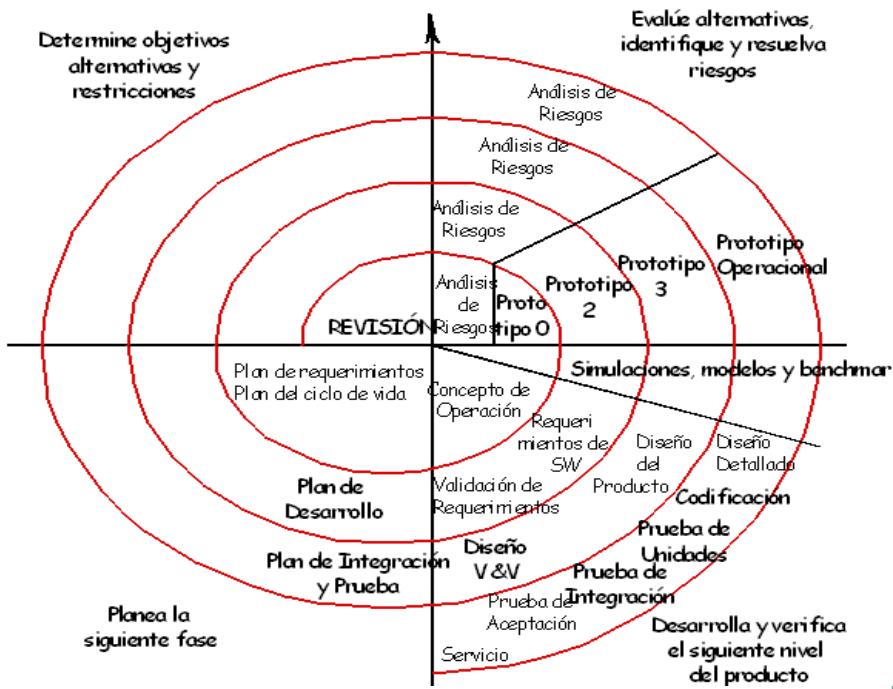
28

- En cada ciclo debemos tener en cuenta:
  - Objetivos: Necesidades que debe cubrir el producto.
  - Alternativas: Diferentes formas de conseguir los objetivos.
  - Desarrollar y Verificar.
  - Planificar: Si el resultado no es el adecuado, planificar siguientes pasos y comenzar.
- Medidas:
  - Dimensión Radial: Indica el aumento del coste. Cada ciclo implica más desarrollo.
  - Dimensión Angular: Avance del proyecto en un ciclo.

# Desarrollo en espiral

29

- Cada ciclo se divide en cuatro sectores:
  - Determinación de objetivos.
    - Productos, requisitos, especificaciones
  - Evaluación y reducción de riesgos.
    - Se analizan posibles amenazas, etc. Se evalúan alternativas y se hace un prototipo.
  - Desarrollo, verificar y validar.
    - Se elige el modelo para el desarrollo
  - Planificación



# Desarrollo en espiral

30

- **Ventajas**

- Modelo realista para el desarrollo de software a gran escala
    - El software evoluciona a la vez que lo hace el modelo
  - El uso de prototipos
    - Permite conocer mejor el riesgo
    - El cliente puede observar cómo va el producto

- **Inconvenientes**

- Requiere expertos en gestión de riesgos.

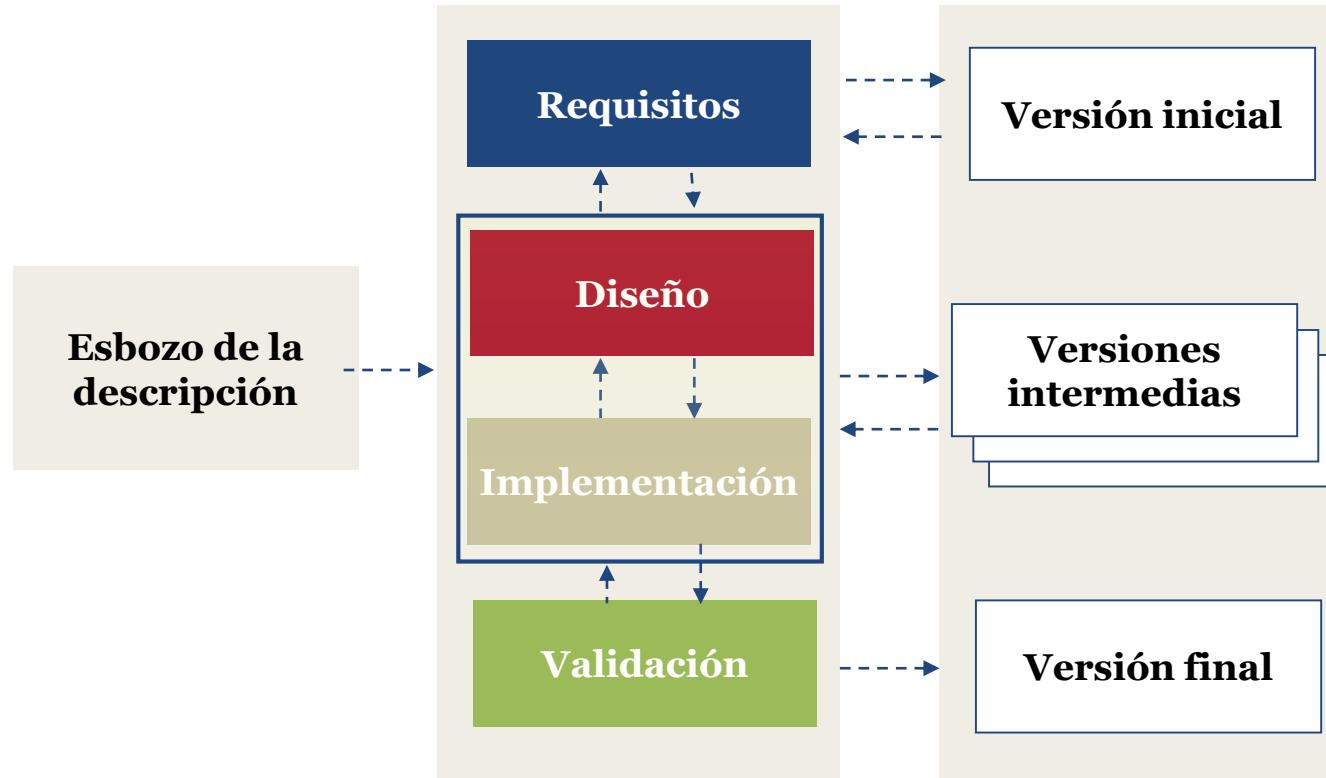
# Desarrollo incremental

31

- Se basa en la idea de desarrollar una implementación inicial, presentársela al usuario e ir refinándola a través de diferentes versiones.
- La actividades del proceso se mezclan, en vez de separarse como se hace en el modelo en cascada. En ocasiones, la especificación se desarrolla junto con el software.
- El propósito de estos modelos es desarrollar software de alta calidad de una manera *iterativa* o *incremental*.
- Los procesos ágiles se ajustan a este modelo

# Desarrollo incremental

32



# Desarrollo incremental

33

- Ventajas:

- La especificación se puede desarrollar de forma creciente
  - ✖ Ya que el cliente puede evaluar sobre la marcha cómo va el proyecto
- Puede satisfacer necesidades inmediatas de los clientes

# Desarrollo incremental

34

- Inconvenientes:

- Es difícil establecer a priori el número de ciclos que serán necesarios para construir el producto.
- Es difícil establecer una arquitectura estable del sistema
  - ✖ Inapropiado para sistemas grandes y complejos, con un periodo de vida largo en el que varios equipos desarrollan distintas partes del sistema.
- Si los sistemas se desarrollan rápidamente, no es rentable producir documentos que reflejen cada cambio del sistema
- Los cambios continuos tienden a corromper la estructura del software

# El Proceso Unificado de Rational

35

- Es un modelo de proceso software derivado de los trabajos sobre UML (Unified Modelling Language) y el proceso asociado
- Normalmente se describe desde tres perspectivas:
  - Perspectiva dinámica, que muestra las fases en el tiempo
  - Perspectiva estática, que muestra las actividades del proceso
  - Perspectiva práctica, que sugiere buenas prácticas de diseño

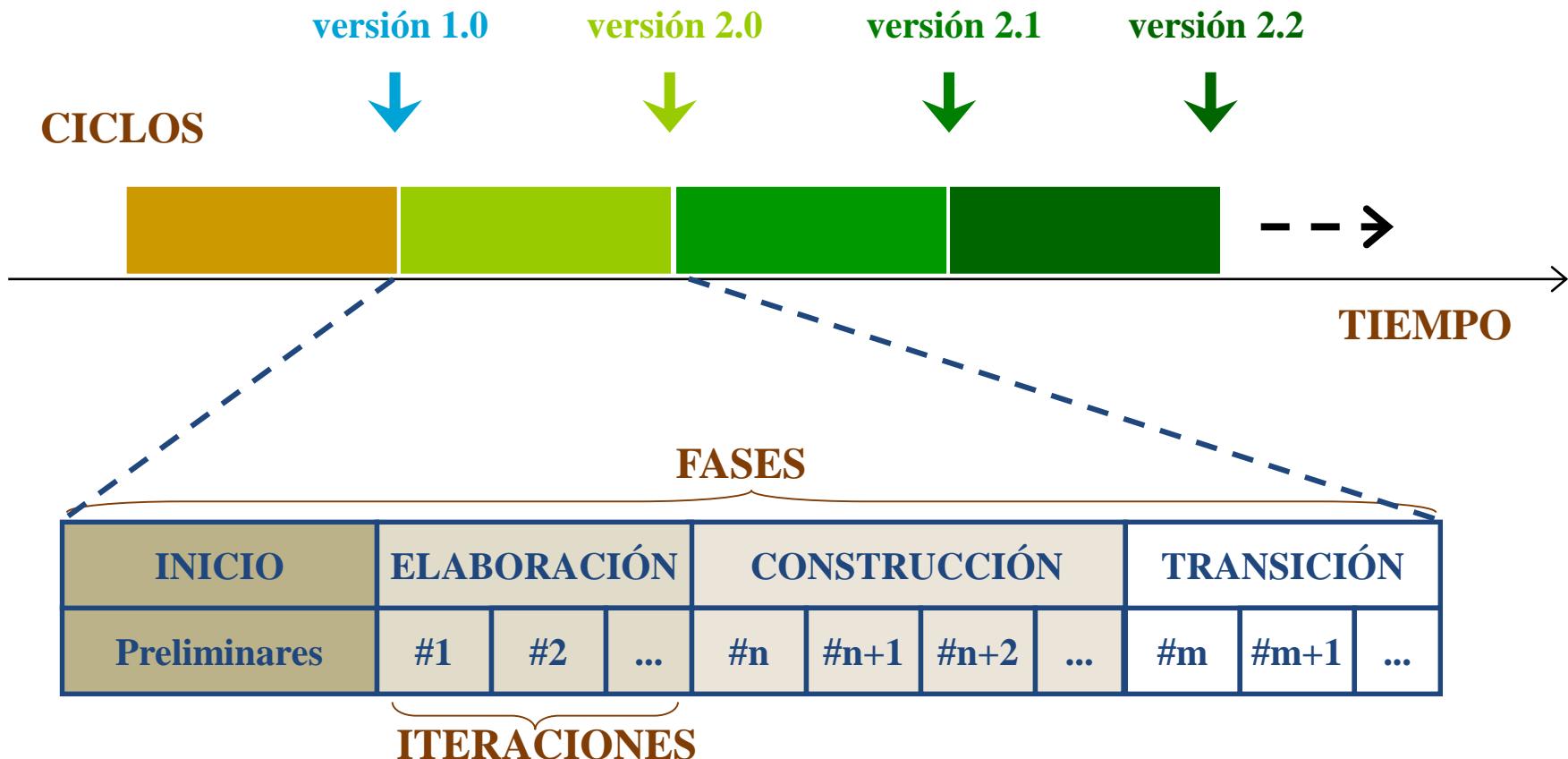
# El Proceso Unificado de Rational

36

- Es un modelo que se divide en ciclos
  - Cada ciclo identifica cuatro fases diferentes del proceso del software
- Las fases están relacionadas más con aspectos de negocio que técnicos
  - Al contrario que el modelo en cascada
  - Cada fase se divide a su vez en iteraciones
- Al finalizar un ciclo se produce una nueva versión del sistema:
  - Producto listo para su entrega
  - Código fuente + componentes + manuales + productos asociados

# El Proceso Unificado de Rational

37



# El Proceso Unificado de Rational

38

- **Inconvenientes (qué piensan algunos)**
  - Burocracia: hay un proceso para todo
  - Lento: se debe seguir todo el proceso para cumplir el modelo
  - Mucha sobrecarga: justificación, documentación, informes, reuniones, ... .

# Índice de contenidos

39

- Introducción
- Actividades dentro de un proceso software
- Modelos de proceso software
- Modelos clásicos
- **Modelos especializados**
- Métodos ágiles

# Modelos especializados

40

- Son enfoques concretos de ingeniería del software que se aplican en algunos contextos
- Ejemplos:
  - Desarrollo basado en componentes
    - Enfoque basado en la reutilización
    - Un programa se construye integrando componentes independientes débilmente acoplados
  - Métodos formales
    - Especificación matemática del software
    - A través del análisis matemático se pueden detectar errores e inconsistencias

# Índice de contenidos

41

- Introducción
- Actividades dentro de un proceso software
- Modelos de proceso software
- Modelos clásicos
- Modelos especializados
- **Métodos ágiles**

# Métodos ágiles

42

- Manifiesto del desarrollo ágil de software:
  - <http://agilemanifesto.org/iso/en/>

We are uncovering better ways of developing software by doing it and helping others do it.  
Through this work we have come to value:

**Individuals and interactions** over processes and tools  
**Working software** over comprehensive documentation  
**Customer collaboration** over contract negotiation  
**Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

# Métodos ágiles

43

- Los métodos ágiles combinan:

- Filosofía

- Fomento de la satisfacción del cliente
    - Desarrollo incremental
    - Equipos de pequeño tamaño y muy motivados
    - Métodos informales
    - Simplicidad

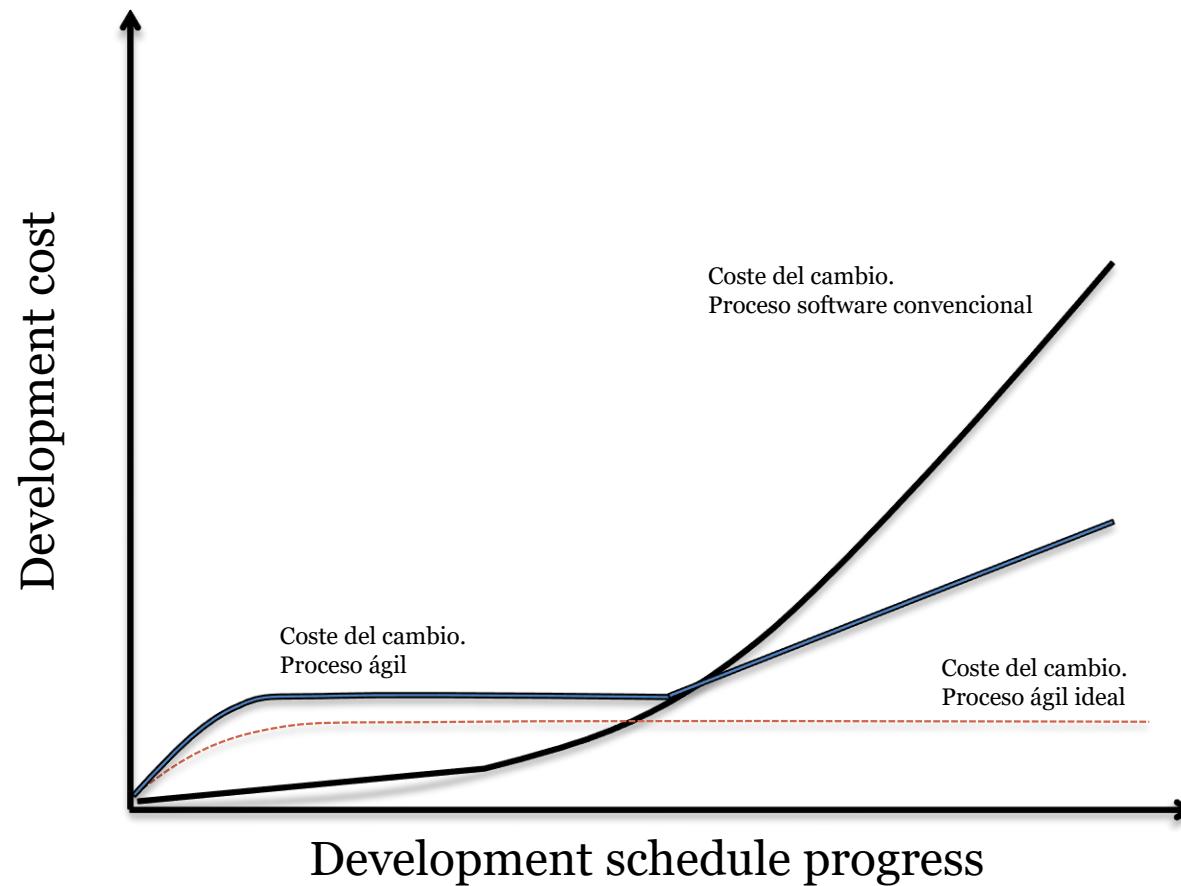
- Guías de desarrollo

- Comunicación continua y activa con el cliente
    - Fomento de la entrega de partes del producto frente a análisis y diseño

# Métodos ágiles

44

- Agilidad y el coste del cambio



# Métodos ágiles

45

- Ideas sobre las que sustentan los métodos ágiles:
  - Es difícil predecir inicialmente qué requisitos software se mantendrán y cuáles cambiarán
  - Para muchos tipos de software, diseño e implementación son tareas que se entremezclan
  - Análisis, diseño, implementación y pruebas son tareas con un grado de impredecibilidad desde un punto de vista de la planificación
- Solución para abordar la impredecibilidad
  - El proceso software debe ser adaptable

# Programación extrema (XP)

46

- Proceso software ágil más conocido
- Desarrollado en 1999
  - Por Kent Beck, Ward Cunningham and Ron Jeffries
- Indicado para proyectos
  - Con requisitos cambiantes
  - De alto riesgo
  - Con equipos de desarrollo pequeños (2 – 12 personas)
- La XP es la adopción de las mejores metodologías de desarrollo para un problema, aplicadas de manera dinámica.

# Programación extrema (XP)

47

- Las cuatro claves de XP

- Comunicación
  - Entre miembros del equipo, gestores de proyecto y clientes
- Simplicidad
  - “Do the simplest thing that could possibly work”
  - “Never implement a feature you don’t need now”
- Retroalimentación (feedback)
  - Por el sistema (tests unitarios), por el cliente, por el equipo
- Coraje
  - Perder el miedo a refactorizar, quitar código obsoleto, persistencia para resolver los problemas que se presentan, etc.

# Código Autocomentado

48

Mal comentario:

```
// set product to "base"

product = base;// loop from 2 to "num"

for ( int i = 2; i <= num; i++ ) {

    // multiply "base" by "product"

    product = product * base;

}

System.out.println("Product= product);
```

Buen comentario:

```
// compute the square root of Num using the Newton-Raphson approximation

x = num / 2;

while ( abs( x - (num/x) ) > TOLERANCE ) {

    x = 0.5 * ( x + (num/x) );

}

System.out.println( "x = " + x );
```

# Programación extrema (XP)

49

- Características principales
  - Planificación incremental
  - Pequeñas versiones (releases)
  - Diseño simple
  - Test-first development
  - Refactorización
  - Pair programming
  - Integración continua
  - El cliente es miembro del equipo



www.xprogramming.com

<http://xprogramming.com/what-is-extreme-programming/>

# Programación extrema (XP)

50

- Críticas al modelo

- Modelo poco realista: muy centrado en el programador.
- No se escriben las especificaciones con detalle.
- Refactorizaciones constantes (consume tiempo).
- Las actividades que caracterizan al proceso son demasiado interdependientes.

# Scrum

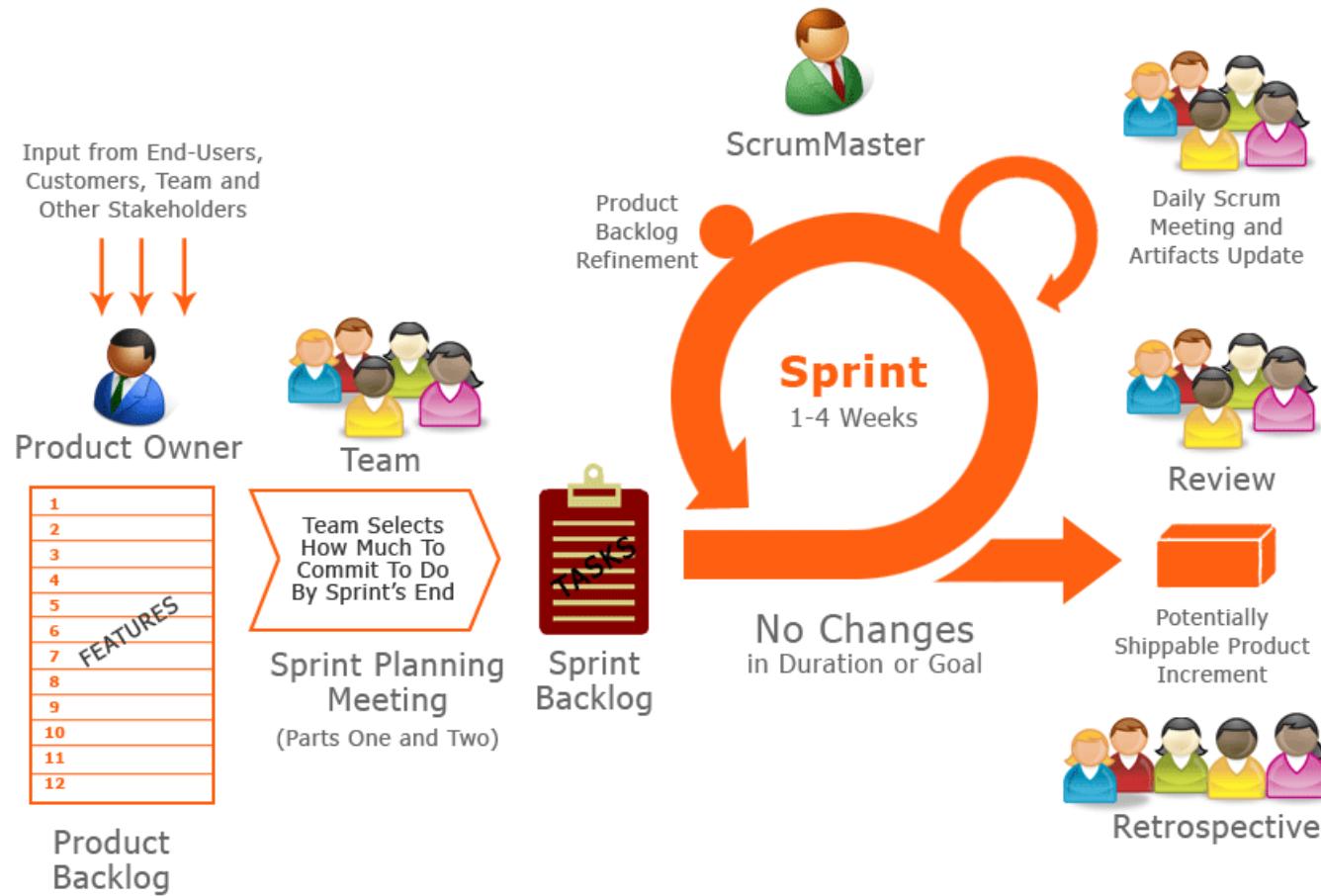
51

- Scrum (o melé) es una metodología ágil de gestión de proyectos que apareció en los 80-90.
- Requisitos inestables que requieren rapidez y flexibilidad.
- Desarrollo incremental, donde se solapan las diferentes fases del desarrollo.
- Se basa en un proceso iterativo que define un conjunto de prácticas y roles
  - Roles: scrum master, team, product owner
  - Prácticas: backlogs, sprints, meetings, demos
- Ha mostrado ser muy útil en proyectos de tamaño pequeño y mediano, principalmente web y apps.

<http://www.scrum.org/Resources/What-is-Scrum>

# Scrum

52



# Roles Principales

53

- **Product Owner.**
  - Representa al cliente.
  - Se asegura que el equipo Scrum trabaje de forma adecuada desde la perspectiva del negocio.
  - Escribe historias de usuario, las prioriza y las coloca en la pila de producto (Product Backlog).
- **ScrumMaster (o facilitador).**
  - El Scrum es facilitado por un ScrumMaster.
  - Su trabajo es eliminar los obstáculos que impiden que el equipo alcance el objetivo del sprint.
  - El Scrum no es el líder del equipo (el equipo se autoorganiza), actúa como protector del equipo para que no se distraiga.

# Roles Principales

54

- **Equipo de Desarrollo.**

- El equipo tiene la responsabilidad de entregar el producto.
- Equipo pequeño de 3 a 9 personas.
- Con habilidades transversales:
  - Análisis.
  - Diseño.
  - Desarrollo.
  - Pruebas.
  - Documentación.

# Roles Auxiliares

55

- Son aquellos que no tienen un rol formal y no se involucran frecuentemente en el proceso scrum.
- Pero deben ser tomados en cuenta.
- Usuarios, expertos del negocio, y otros públicos (clientes, proveedores, vendedores, etc).
- Administradores.
  - Es la gente que establece el ambiente para el desarrollo del producto.

# Reuniones en Scrum

56

- Daily Scrum o Stand-up meeting
  - Cada día de un sprint, se realiza una reunión sobre el estado del proyecto.
  - Guías para las reuniones:
    - Puntuales, a la misma hora y el mismo lugar.
    - Puede ir todo el mundo que lo deseé.
    - No hablan nada más que los involucrados en el proyecto.
    - 15 minutos de duración.
  - Cada miembro del equipo responde a tres preguntas:
    - ¿qué has hecho desde ayer?
    - ¿qué es lo que harás para mañana?
    - ¿has tenido algún problema que te haya impedido alcanzar tu objetivo? (Scrummaster gestiona los problemas?)

# Reunión de Sprint

57

- Al inicio de cada ciclo de Sprint se lleva a cabo una reunión (duración de 1 jornada laboral):
  - Seleccionar que trabajo se hará.
  - Preparar, con el equipo completo, el sprint Backlog que detalla el tiempo que llevará hacer el trabajo.
  - Identificar y comunicar el trabajo que se hará en el sprint.
- Sprint Review Meeting:
  - Revisar el trabajo que fue completado y no completado.
  - Presentar el trabajo a los interesados.
  - El trabajo incompleto no se presenta.
  - Media jornada de trabajo.
- Sprint Retrospective:
  - Despues de cada Sprint se hace un análisis del mismo.
  - El objetivo es mejorar para los sprint siguientes.

# Documentos de Scrum

58

- **Product Backlog**

- Documento de alto nivel del proyecto.
- Reúne los requisitos de proyecto, con descripciones genéricas de funcionalidades, priorizadas en función del retorno de inversión (ROI).
- Es abierto y solo puede ser modificada por el product owner.
- Contiene estimaciones del valor de negocio, esfuerzo de desarrollo.

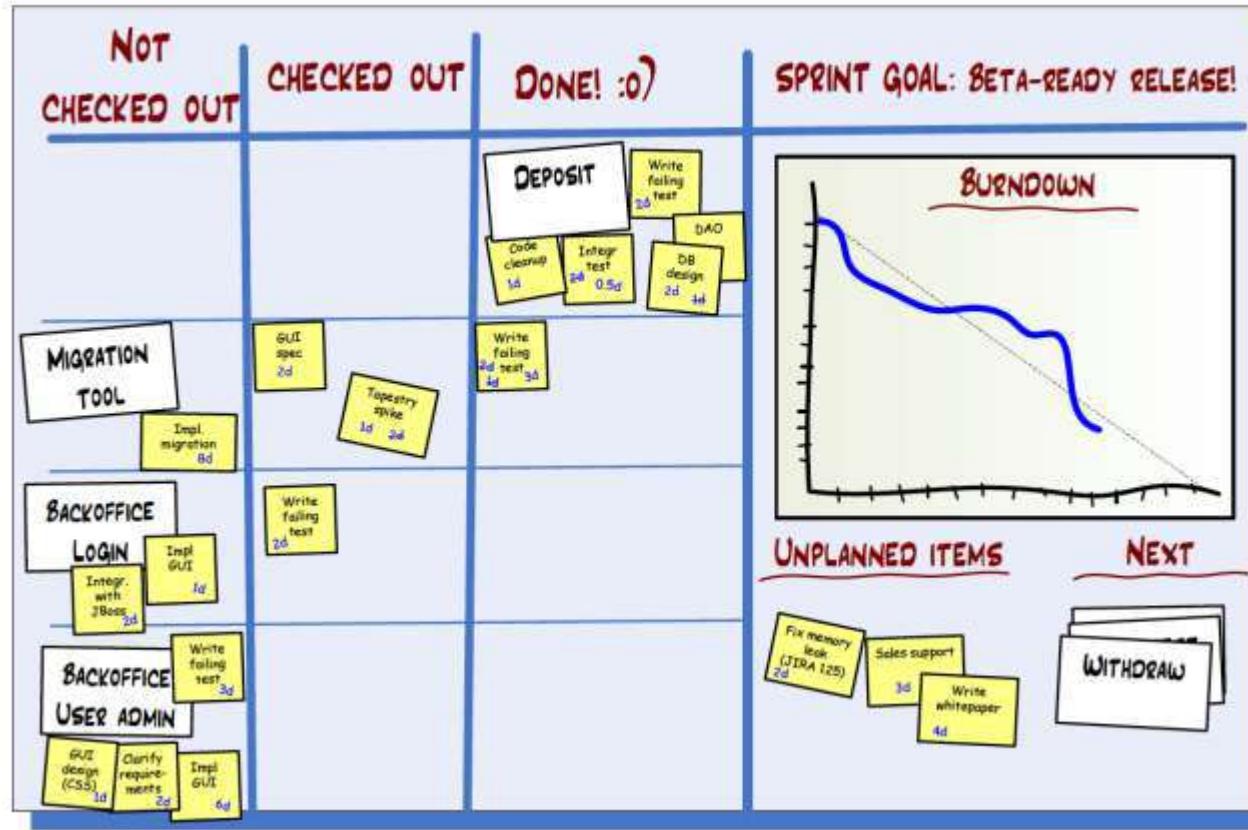
# Documentos de Scrum

59

- **Sprint Backlog:**
  - Es el subconjunto de requisitos que serán desarrollados en el siguientes sprint.
  - Los requisitos se subdividen en tareas con una duración de 16 horas de trabajo aproximadamente.
  - Las tareas no son asignadas, son tomadas por los miembros del equipo.
- **Burn down chart:**
  - Es una gráfica pública que mide la cantidad de requisitos en el Backlog del proyecto pendientes en cada sprint.
  - Ayuda a ver el progreso del proyecto.

# Scrum. Burndown chart

60



<http://www.crisp.se/bocker-och-produkter/scrum-and-xp-from-the-trenches>

# Ejercicio para Alumnos

61

- Estudiar el código autocomentado.

# Introducción a la Ingeniería del Software



**TEMA 4: INGENIERÍA DE REQUISITOS**

**REQUISITOS DEL SOFTWARE**

**Grado en Ingeniería Informática**

**Grado en Ingeniería del Software**

**Grado en Ingeniería de Computadores**

# Índice

2

- 1. Introducción**
- 2. Requisitos funcionales y no funcionales**
- 3. Especificación de requisitos**
- 4. Proceso de Ingeniería de Requisitos**
- 5. Obtención y análisis de requisitos**
- 6. Validación de requisitos**

# Índice

3

- 1. Introducción**
- 2. Requisitos funcionales y no funcionales**
- 3. Especificación de requisitos**
- 4. Proceso de Ingeniería de Requisitos**
- 5. Obtención y análisis de requisitos**
- 6. Validación de requisitos**

# ¿Qué es un requisito?

4

- Son descripciones de los servicios que un sistema debe proporcionar y las restricciones a su modo de operación.
- Reflejan las necesidades del cliente para el sistema.
- Pueden abarcar
  - desde una declaración abstracta de alto nivel de un servicio,
  - hasta la especificación matemática detallada y formal de una función del sistema.
- **Los requisitos tienen una doble función:**
  - **Pueden ser la base de la propuesta de un contrato** → deben estar abiertos a ser interpretados
  - **Pueden ser la base de un contrato** → deben estar definidos detalladamente.

# Ingeniería de Requisitos

5

- **Ingeniería de Requisitos (IR)**

- Es el proceso para determinar, analizar, documentar y comprobar los requisitos de un sistema.
- Los requisitos son, en sí mismos, la descripción de los servicios y las restricciones de un sistema que se descubren durante el proceso de IR.

# Tipos de Requisitos

6

- **Dos niveles de abstracción:**

- **Requisitos de Usuario** (especificación a alto nivel)
  - Declaraciones en lenguaje natural y en diagramas de los servicios que el sistema debe proporcionar y las restricciones bajo las que debe funcionar.
  - Escrito para los clientes.
- **Requisitos del Sistema** (especificación funcional)
  - Documento estructurado donde se establecen con detalle los servicios y restricciones del sistema.
  - Describe lo que se implementará y no se implementará.
  - Escrito como un contrato entre el comprador y el desarrollador software.
  - Deben ser una especificación completa y consistente del sistema.

# Ejemplo

7

## Requisitos de usuario

1. El software debe proporcionar un medio para representar y acceder a archivos externos creados por otras herramientas.

## Requisitos del sistema asociados

- 1.1. Se suministrará al usuario los recursos para definir el tipo de archivos externos.
- 1.2 Cada tipo de archivo externo tendrá una herramienta asociada que será aplicada al archivo.
- 1.3 Cada tipo de archivo externo se representará como un ícono específico sobre la pantalla del usuario.
- 1.4 Se proporcionarán recursos para que el usuario defina el ícono que representará un tipo de archivo externo.
- 1.5 Cuando un usuario selecciona un ícono que representa un archivo externo, el efecto de esa selección es aplicar la herramienta asociada con este tipo de archivo al archivo representado por el ícono seleccionado.

# Índice

8

1. Introducción
2. **Requisitos funcionales y no funcionales**
3. Especificación de requisitos
4. Proceso de Ingeniería de Requisitos
5. Obtención y análisis de requisitos
6. Validación de requisitos



# Definición

9

- **Requisitos funcionales (RF)**

- Describen la funcionalidad o los servicios que se espera que el sistema suministre.
- ¿Cómo reaccionará a determinadas entradas?
- ¿Cómo se comportará en determinadas situaciones?

- **Requisitos no funcionales (RNF)**

- Restricciones sobre los servicios o funciones ofrecidas por el sistema.
- Ej.: tiempo de respuesta, estándares, fiabilidad, capacidad E/S, etc.

- **Requisitos del dominio**

- Requisitos que vienen impuestos por el dominio de la aplicación del sistema y que recogen características de ese dominio.
- Ej.: requisitos legales

# Requisitos funcionales vs. no funcionales

10

- Los **requisitos funcionales** describen las capacidades del sistema (**¿QUÉ HACE?**).
- Los **requisitos no funcionales** describen las cualidades del sistema (**¿CÓMO HACE LO QUE HACE?**)
- **Muchos requisitos tienen una mezcla de ambos tipos**
  - Ej. Aspectos de seguridad.
  - Para facilitar su tratamiento los consideraremos como RNF.
- Los **RNFs son los que suelen influir más en la complejidad del proyecto.**
- Además son más difíciles de verificar que se cumplen.

# Requisitos funcionales

11

- Describen los servicios y la funcionalidad del sistema.
- Dependen del tipo de software, del tipo de sistema donde se usará y de los posibles usuarios.
- Los **RF del usuario** pueden ser declaraciones de alto nivel sobre lo que realizará el sistema.
- Pero los **RF del sistema** deben describir los servicios del sistema detalladamente.
  - Ej. funcionalidad del sistema, entradas/salidas y excepciones en detalle.
  - Son usados por los desarrolladores.

# Ejemplos de requisitos funcionales

12

1. El usuario tendrá la posibilidad de buscar en el conjunto de toda la base de datos o en cualquier subconjunto que seleccione de ella.
2. El sistema proporcionará visores adecuados que permitan leer los documentos almacenados en el repositorio de documentos.
3. A cada pedido se le deberá asignar un identificador único (ID\_PEDIDO) que el usuario podrá copiar al área de almacenamiento permanente de la cuenta.

# Imprecisión en los requisitos

13

- Aparecen problemas cuando los requisitos no están expresados con precisión.
  - Requisitos ambiguos pueden entenderse de forma distinta por los usuarios y por los desarrolladores.
  - Los desarrolladores tienden a entenderlos de forma que se simplifique la implementación:
- **Ejemplo:** “*visores adecuados*”
  - **Usuario:** existan visores especializados para cada tipo de documento.
  - **Desarrollador:** proporcionar un visor de texto que muestre el contenido de los documentos.

# Completitud y Consistencia de los Requisitos

14

- Los requisitos deben ser **completos** y **consistentes**.
  - **Compleitud**
    - Todos los servicios y funciones solicitadas por el cliente deben estar definidos.
  - **Consistencia**
    - No debe existir conflictos o contradicciones en la descripción de los servicios y funciones del sistema.
- En la práctica, para sistemas grandes y complejos, es imposible obtener un documento completo y consistente.

# Requisitos no funcionales

15

- **Definen propiedades y restricciones del sistema**
  - Fiabilidad, tiempo de respuesta, capacidad de almacenamiento, ...
- Algunos requisitos del proceso pueden obligar a utilizar estándares de calidad, determinadas herramientas CASE o lenguajes de programación.
- **Pueden ser más críticos que los RFs**
  - Si no se cumplen, invalidan el sistema.
  - Ej. No se satisfacen requisitos de seguridad sistemas críticos (bancarios, control de avión) → no se pueden certificar para uso.
- **Su implementación suele englobar a todo el sistema**
  - Suelen afectar la arquitectura global del sistema.
  - Un requisito no funcional puede generar varios requisitos funcionales.
  - Ej. Sistema seguro → transmisiones cifradas, login con password, etc.

# Clasificación de requisitos no funcionales

16

- **Requisitos del producto**

- Especifican el comportamiento del producto.
- Ej.: Velocidad de ejecución, tasa aceptable de fallos, seguridad, usabilidad, ...

- **Requisitos de la organización y del proceso**

- Son consecuencia de las políticas y procedimientos de la propia organización (del cliente o desarrollador).
- Ej.: estándares de proceso, métodos de diseño, lenguajes de programación, tipo de documentación a entregar.

- **Requisitos externos**

- Se derivan de factores externos al producto y a su proceso de desarrollo.
- Ej.: Interoperabilidad del sistema con otros, requisitos legales (LOPD), éticos, ...

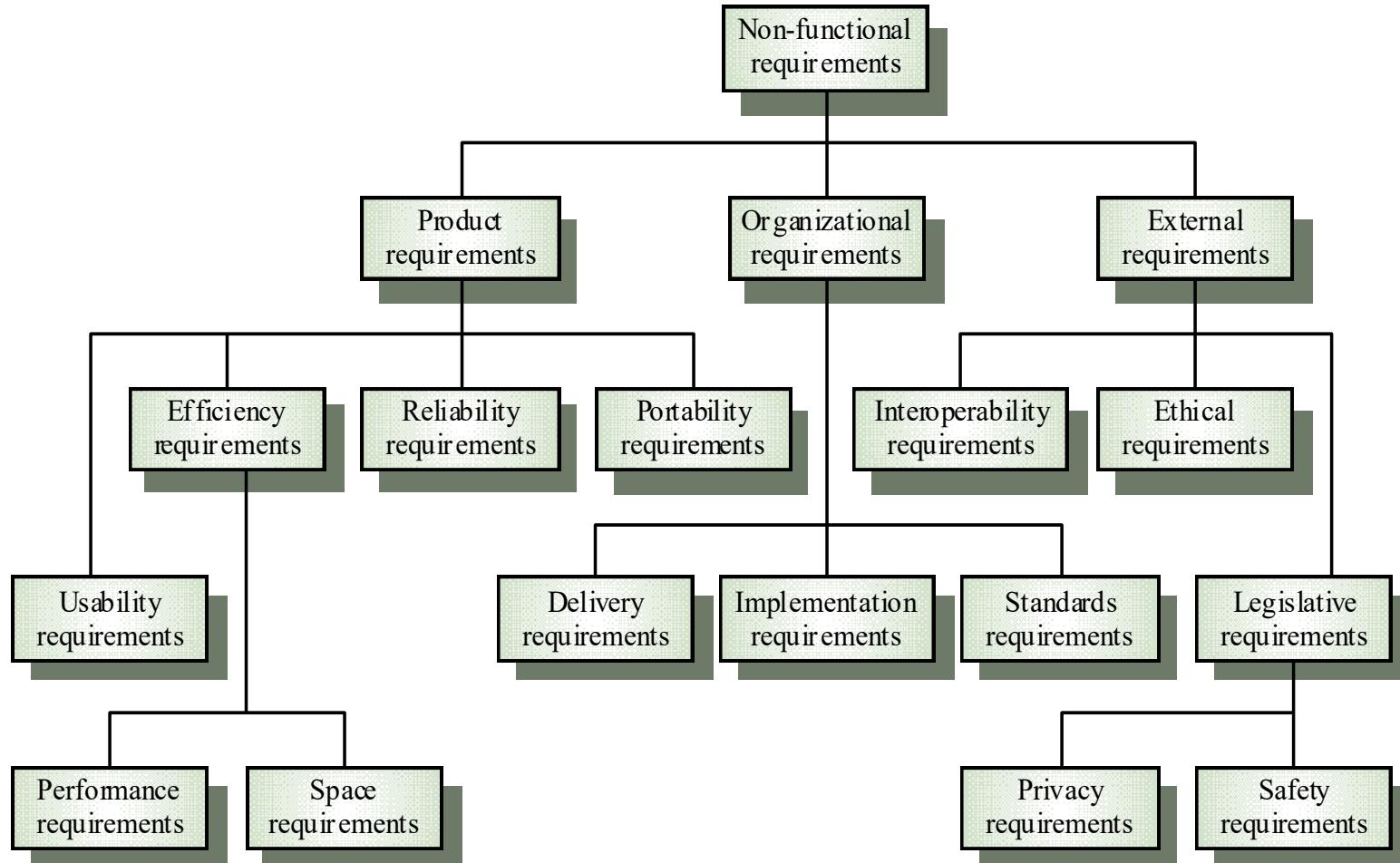
# Ejemplos de RNFs

17

- **Requisito del producto**
  - 4.C.8. Se utilizará en todas las comunicaciones el conjunto de caracteres ASCII estándar.
- **Requisito de la organización**
  - 9.3.2 El proceso de desarrollo y los documentos entregados serán conformes con el estándar ISO-XXXX.
- **Requisito externo**
  - 7.6.5 El sistema no revelará datos de carácter personal de los clientes a los usuarios con nivel 3 o inferior excepto el nombre y número de referencia.

# Tipos de requisitos no funcionales

18



# Metas y Requisitos

19

- Los RNFs pueden resultar difíciles de definir con precisión
  - Y los requisitos imprecisos son difíciles de verificar.
- **Meta**
  - Un propósito general para el sistema (p.ej. facilidad de uso).
- **Requisito no funcional verificable**
  - Aquellos que se pueden asociar a un objetivo medible (medibles cuantitativamente).
- Las metas son útiles porque trasmiten las prioridades de los clientes y usuarios.

# Ejemplos

20

- **Metas del sistema**

- El sistema debe ser fácil de usar para personas experimentadas.
- El sistema deberá minimizar los errores del usuario.

- **Requisitos verificables (detallados)**

- Los usuarios experimentados deberán poder utilizar todas las funciones del sistema después de dos horas de entrenamiento.
- Despues de un entrenamiento, el número medio de errores cometidos por los usuarios experimentados no excederá de 2 errores al día.

# Medidas para requisitos no funcionales

21

Atributo	Medida
Rapidez	Transacciones procesadas por segundo Tiempo de respuesta al usuario/eventos Tiempo de actualización de pantalla
Tamaño	K Bytes Número de chips de RAM
Facilidad de Uso	Tiempo de aprendizaje Número de ventanas de ayudas
Fiabilidad	Tiempo medio entre fallos Probabilidad de no disponibilidad Tasa de ocurrencias de fallos
Robustez	Tiempo de reinicio después de un fallo % de eventos que provocan un fallo Probabilidad de corrupción de datos por fallos
Portabilidad	% de sentencias dependientes del objetivo Número de sistemas objetivo

# Conflictos y Compromisos

22

- Debido a su naturaleza global, la mayoría de los RNFs interfieren con otros RNFs.
  - A veces pueden ser incompatibles: **Conflicto**
    - En estas situaciones debemos especificar en qué circunstancias qué RNF prevalece sobre los otros.
    - Ej. Seguridad vs. anonimato.
  - A veces se oponen pero no son incompatibles: **Compromiso**
    - Debemos especificar cuál es la relación óptima entre ellos.
  - Esto conlleva que la especificación de RNFs relacionados debe hacerse de manera conjunta o al menos referenciada

# Relación entre RNFs y RFs

23

- Los RNFs suelen conllevar la introducción de RFs adicionales en el sistema
  - Ej. Confidencialidad de los datos → Funciones de cifrado.
- Los RNFs también pueden limitar la manera de implementar los RFs o establecer restricciones de diseño
  - Ej. Transmitir datos confidenciales → Uso de determinados protocolos.
- La mayoría de los RNFs influyen en la arquitectura del sistema y su despliegue
  - Ej. Almacenamiento de datos personales → La BD debe estar físicamente situada en Europa.

# Relación entre RNFs y RFs

24

- Los RFs pueden ser incompatibles con los RNFs
  - Ej. Monitorización de cámaras de vigilancia en una escuela vs. privacidad.
- El cumplimiento de los RNFs puede implicar cambios en la manera en la que los RFs se llevan a cabo
  - Ej. El sistema debe permitir la generación de informes médicos de los empleados (RF)
  - Pero también debe mantener su privacidad (RNF)
  - → Anonimizar los informes.
- Casi cualquier otra relación (directa e indirecta) entre RFs y RNFs es posible.

# Definición

25

- **Requisitos funcionales (RF)**
  - Describen la funcionalidad o los servicios que se espera que el sistema suministre.
  - ¿Cómo reaccionará a determinadas entradas?
  - ¿Cómo se comportará en determinadas situaciones?
- **Requisitos no funcionales (RNF)**
  - Restricciones sobre los servicios o funciones ofrecidas por el sistema.
  - Ejs.: tiempo de respuesta, estándares, fiabilidad, capacidad E/S, etc.
- **Requisitos del dominio**
  - Requisitos que vienen impuestos por el dominio de la aplicación del sistema y que recogen características de ese dominio.
  - Ej.: requisitos legales

# Requisitos del Dominio

26

- Describen las características del dominio en el que se encuadra la organización.
- Pueden ser requisitos funcionales nuevos, restricciones sobre los existentes o definir cómo se deben realizar cálculos específicos.
- Si no se cumplen, el sistema puede no trabajar de forma satisfactoria.

# Ejemplo: Sistema de protección de trenes

27

- La desaceleración del tren se calculará como:
  - ★  $D_{tren} = D_{control} + D_{gradiente}$
- Donde
  - $D_{gradiente}$  es  $9.81 \text{ ms}^2 * \text{gradiente compensado}/\alpha$
  - Los valores de  $9.81 \text{ ms}^2/\alpha$  son conocidos para los diferentes tipos de trenes
- Es difícil de entender para alguien que no es un especialista en el tema (dominio).
- Sobre todo las implicaciones que puede tener con otros requisitos.



# Problemas con Requisitos del dominio

28

- **Comprendibilidad**
  - Se expresan en la “jerga” de la aplicación del dominio.
  - Esta suele ser desconocida para los ingenieros del software.
- **Requisitos implícitos**
  - Los especialistas entienden su área de tal forma que no creen necesario declarar de forma explícita los requisitos del dominio

# Índice

29

1. Introducción
2. Requisitos funcionales y no funcionales
- 3. Especificación de requisitos**
4. Proceso de Ingeniería de Requisitos
5. Obtención y análisis de requisitos
6. Validación de requisitos



# Especificación de requisitos

30

- Se refiere a la **elaboración de un documento** que puede ser sistemáticamente revisado, evaluado y aprobado.
- Para sistemas complejos se producen hasta tres tipos de documentos:
  - Definición del sistema
  - Requisitos del sistema
  - Requisitos de software
- Para productos de software simple, sólo se suele utilizar el tercero (requisitos de software)

# Documento de **definición** del sistema

31

- También conocido como documento de **Requisitos del Usuario**.
- Establece los requisitos de alto nivel del sistema desde la perspectiva de dominio del problema.
  - Especifica el comportamiento externo del sistema.
  - No debe incluir detalles de la arquitectura del sistema o su diseño.
- **Dirigido a:** usuarios y clientes
  - Se usa términos del dominio → Lenguaje natural.

# Documento de **requisitos** del sistema

32

- Para sistemas con componentes software y componentes se suele **separar**
  - la descripción de los requisitos del sistema.
  - descripción de los requisitos de software.
- Los requisitos del sistema se especifican y los requisitos de software se derivan de estos.
- ISO/IEC/IEEE 29148 proporciona una guía para el desarrollo de los requisitos del sistema (disponible en [jabega.uma.es](http://jabega.uma.es)).

# Especificación de Requisitos del Software (SRS)

33

- Establece las bases para un acuerdo entre clientes y contratistas o proveedores sobre:
  - Lo que el producto software tiene que hacer.
  - Lo que no se hará.
- En los proyectos dirigidos al mercado general estas funciones las realizan los departamentos de marketing y desarrollo.
- La SRS permite hacer una evaluación rigurosa de los requisitos antes de comenzar el diseño y reduce posteriores rediseños.
- Debe proporcionar una base realista para la estimación de costes, riesgos y tiempos.
- Los requisitos de usuario se suelen escribir en lenguaje natural, pero la SRS debe complementarse con descripciones formales o semi-formales.

# Requisitos y Diseño

34

- En principio, los **requisitos** deberían establecer qué hará el sistema y el **diseño** cómo se implementará.
- En la práctica, requisitos y diseño son inseparables
  - Se puede definir una arquitectura inicial del sistema para estructurar los requisitos.
  - Los sistemas pueden interoperar con otros ya existentes que obliguen a ciertos requisitos del diseño.
  - El uso de un determinado diseño puede ser un requisito del dominio.

# Alternativas para escribir la especificación de requisitos del sistema (SRS)

35

Notación	Descripción
Lenguaje Natural	<ul style="list-style-type: none"><li>Los requisitos se escriben con frases numeradas en lenguaje natural.</li><li>Cada frase debe expresar un requisito.</li></ul>
Lenguaje Natural Estructurado	<ul style="list-style-type: none"><li>Los requisitos se escriben en lenguaje natural en un formulario estándar o una plantilla.</li><li>Cada campo proporciona información sobre un aspecto de los requisitos.</li></ul>
Lenguajes de Descripción de Diseño	<ul style="list-style-type: none"><li>Se utiliza un lenguaje similar a un lenguaje de programación, pero con características más abstractas.</li><li>Este enfoque se utiliza raramente aunque puede ser útil para las especificaciones de interfaz.</li></ul>
Notaciones gráficas	<ul style="list-style-type: none"><li>Los modelos gráficos complementados con anotaciones de texto, se utilizan para definir los requisitos funcionales del sistema.</li><li>Ej. Diagramas de UML.</li></ul>
Especificaciones matemáticas	<ul style="list-style-type: none"><li>Basadas en conceptos matemáticos, tales como máquinas de estado finito o conjuntos.</li><li>Eliminan la ambigüedad.</li><li>Problema: la mayoría de los clientes no entienden una especificación formal.</li></ul>

# Especificación en lenguaje natural

36

- Los requisitos se escriben como frases en lenguaje natural complementadas con diagramas y tablas.
- Utilizado para definir los requisitos porque es expresivo, intuitivo y universal.
- Esto ayuda a que usuarios y clientes puedan comprender los requisitos.

# Pautas para escribir los requisitos

37

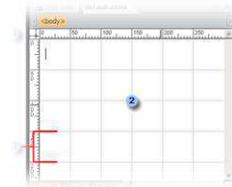
- Definir un formato estándar y usarlo de forma consistente para todos los requisitos.
- Utilizar el **futuro simple para los obligatorios** (deberá tener) y el **condicional para los deseables** (debería tener).
- Resaltar el texto con las **partes claves** del requisito.
- Evitar el uso de lenguaje técnico.

# Ejemplo: Cuadrícula de Edición

38

## 2.6 Recurso de Cuadrícula

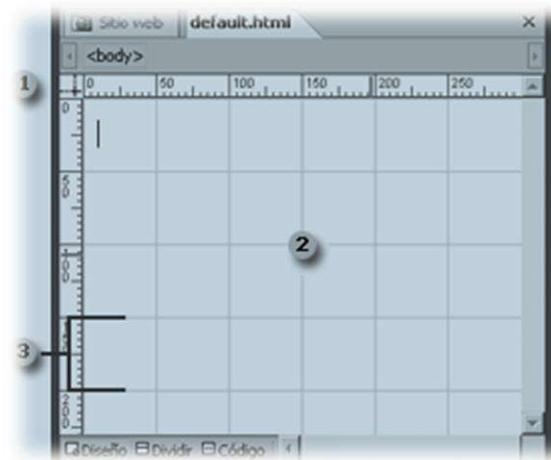
- Para ayudar a la colocación de entidades en un diagrama, el **usuario activará una cuadrícula en centímetros o en pulgadas**, mediante una opción del panel de control.
- **Inicialmente la cuadrícula estará desactivada.** Se podrá activar o desactivar en cualquier momento durante una sesión de edición y puesta en pulgadas o centímetros.
- La opción de cuadrícula se **mostrará en la vista de reducción de ajuste**, pero el número de líneas de cuadrícula a mostrar se reducirá para evitar la saturación del diagrama más pequeño con líneas de cuadrícula.



# Problemas

39

- Mezcla tres clases de requisitos
  - Requisito funcional conceptual
    - La necesidad de una cuadrícula.
  - Requisito no funcional
    - Unidades.
  - Requisito no funcional de interfaz de usuario
    - Activación / Desactivación.



# Más ejemplos

40

- *La aplicación mostrará el tiempo en 24 horas.*
- *Apagar el motor si el nivel del agua sube por encima de 10 metros durante 4 segundos.*

# Problemas con la especificación en lenguaje natural

41

- Ambigüedad
  - Los especificación de requisitos debería poder interpretarse de la misma forma por todo el mundo → difícil debido a ambigüedad del lenguaje natural.
- Demasiada flexibilidad
  - Lo mismo se puede decir de múltiples maneras.
- Falta de claridad
  - Es difícil ser preciso sin que el texto resulte difícil de leer.
- Confusión de requisitos
  - Se tiende a mezclar los requisitos funcionales y no funcionales, las metas y la información para el diseño.
  - Requisitos diferentes se expresan de forma conjunta.

# Especificaciones en lenguaje estructurado

42

- **Forma restringida del lenguaje natural**
  - Mantiene la expresividad y comprensión del lenguaje natural.
  - Limita la terminología utilizada y emplea plantillas.
  - Se reduce la ambigüedad y flexibilidad del lenguaje natural, incrementando de uniformidad.
- Incorporan construcciones de control derivadas de los lenguajes de programación
  - Ej. Para, si, entonces, ...

# Especificación basada en plantillas

43

- Definición de la función o entidad.
- Descripción de sus entradas y de donde provienen.
- Descripción de sus salidas y hacia donde van.
- Indicación de otras entidades utilizadas (si existen).
- Los efectos laterales (si hay).

# Ejemplo: Agregar nodo

44

ECLIPSE/Workstation/Tools/DE/FS/3.5.1

**Function** Add node

**Description** Adds a node to an existing design. The user selects the type of node, and its position. When added to the design, the node becomes the current selection. The user chooses the node position by moving the cursor to the area where the node is added.

**Inputs** Node type, Node position, Design identifier.

**Source** Node type and Node position are input by the user, Design identifier from the database.

**Outputs** Design identifier.

**Destination** The design database. The design is committed to the database on completion of the operation.

**Requires** Design graph rooted at input design identifier.

**Pre-condition** The design is open and displayed on the user's screen.

**Post-condition** The design is unchanged apart from the addition of a node of the specified type at the given position.

**Side-effects** None

*Definition: ECLIPSE/Workstation/Tools/DE/RD/3.5.1*

# Índice

45

1. Introducción
2. Requisitos funcionales y no funcionales
3. Especificación de requisitos
- 4. Proceso de Ingeniería de Requisitos**
5. Obtención y análisis de requisitos
6. Validación de requisitos



# Proceso de ingeniería de requisitos

46

- Los procesos utilizados por la I.R. varían ampliamente dependiendo de
  - Dominio de la aplicación.
  - Personal involucrado.
  - Organización que está elaborando los requisitos.
- Sin embargo, hay un número de actividades genéricas comunes a todos los procesos



# Estudio de viabilidad

47

- Permite decidir si el sistema propuesto es conveniente.
- Es un estudio rápido y orientado a conocer:
  - Si el sistema contribuye a los objetivos de la organización.
  - Si el sistema se puede realizar con la tecnología actual y con el tiempo y el coste previsto.
  - Si el sistema puede integrarse con otros existentes.

# Obtención y Análisis de Requisitos

48

- **Obtención de requisitos**

- El proceso mediante el cual se **captura el propósito y funcionalidades del sistema** desde la perspectiva del usuario.
- Técnicas: observación, entrevistas, herramientas CASE (UML).

- **Análisis de requisitos**

- El proceso de razonamiento sobre los requisitos obtenidos en la etapa anterior para **obtener una definición detallada** de los requisitos.
- Técnicas: representaciones gráficas (UML) y técnicas de revisión.

- **Especificación de requisitos**

- Proceso por el que se **documenta** el comportamiento requerido de un sistema software.
- Técnicas: notación de modelado y lenguajes de especificación.

# Validación de Requisitos

49

- **Validación de requisitos**

- Proceso de **confirmación**, por parte de los usuarios, de que los requisitos especificados son válidos, consistentes y completos.
- Deben definir el sistema que el cliente y los usuarios desean.
- Técnicas: listas de comprobación y técnicas de revisión.

- **Gestión de requisitos**

- Proceso de manejar los requisitos cambiantes durante el desarrollo del sistema.
- Técnicas: herramientas CASE.

# Índice

50

1. Introducción
2. Requisitos funcionales y no funcionales
3. Especificación de requisitos
4. Proceso de Ingeniería Requisitos
5. **Obtención y análisis de requisitos**
6. Validación de requisitos



# Obtención y Análisis de Requisitos

51

- Se trata de descubrir los requisitos.
- El personal técnico trabaja con los clientes y usuarios para
  - descubrir el dominio de la aplicación,
  - los servicios que el sistema debe proporcionar,
  - y las restricciones operativas del sistema.
- Puede involucrar a
  - usuarios finales,
  - gestores,
  - ingenieros de mantenimiento,
  - expertos del dominio,
  - sindicatos,
  - etc...
- Se utiliza el término “stakeholders” (participantes/interesados) para referirse a todos ellos.

# Problemas

52

- Los participantes no conocen realmente lo que desean.
- Expresan los requisitos con sus propios términos.
- Pueden aparecer requisitos de distintos grupos que entren en conflicto.
- Influencia de factores políticos y de la organización.
- Cambio de requisitos durante el proceso de análisis.
  - Pueden aparecer nuevos participantes o el entorno del negocio puede cambiar.

# La Espiral de los requisitos



# Actividades del proceso

54

## Descubrimiento de requisitos

- Interactuar con las partes interesadas para descubrir sus necesidades.
- Los requisitos del dominio también se descubren en esta etapa.

## Clasificación y organización de requisitos

- Agrupar requisitos relacionados y organizarlos en grupos coherentes.

## Asignación de prioridades y negociación

- Priorizar los requisitos y resolver conflictos entre requisitos (o personas).

## Documentación de los requisitos

- Los requisitos se documentan y serán la entrada para la siguiente ronda de la espiral.

# Descubrimiento de requisitos

55

- El proceso de **recopilación** de información sobre los sistemas propuestos o existentes y la **destilación** de esa información en los requisitos del usuario y del sistema.
- Las fuentes de información incluyen
  - la documentación,
  - los participantes interesados en el sistema,
  - y las especificaciones de sistemas similares.

# Fuentes de información (1/2)

56

## Los **objetivos generales** o de alto nivel del software

- Constituyen el motivo fundamental por el que se lleva a cabo el desarrollo del software.
- Un estudio de viabilidad es una forma relativamente barata de hacer esto.

## El **domino del problema**

- El ingeniero de software debe adquirir, o tener a su disposición, el conocimiento sobre el dominio de aplicación.
- Otros actores lo suelen conocer en profundidad.

## Los propios **participantes** o actores del proceso y sus diferentes puntos de vista

- Sobre la organización.
- Sobre el software bajo desarrollo.

# Fuentes de información (2/2)

57

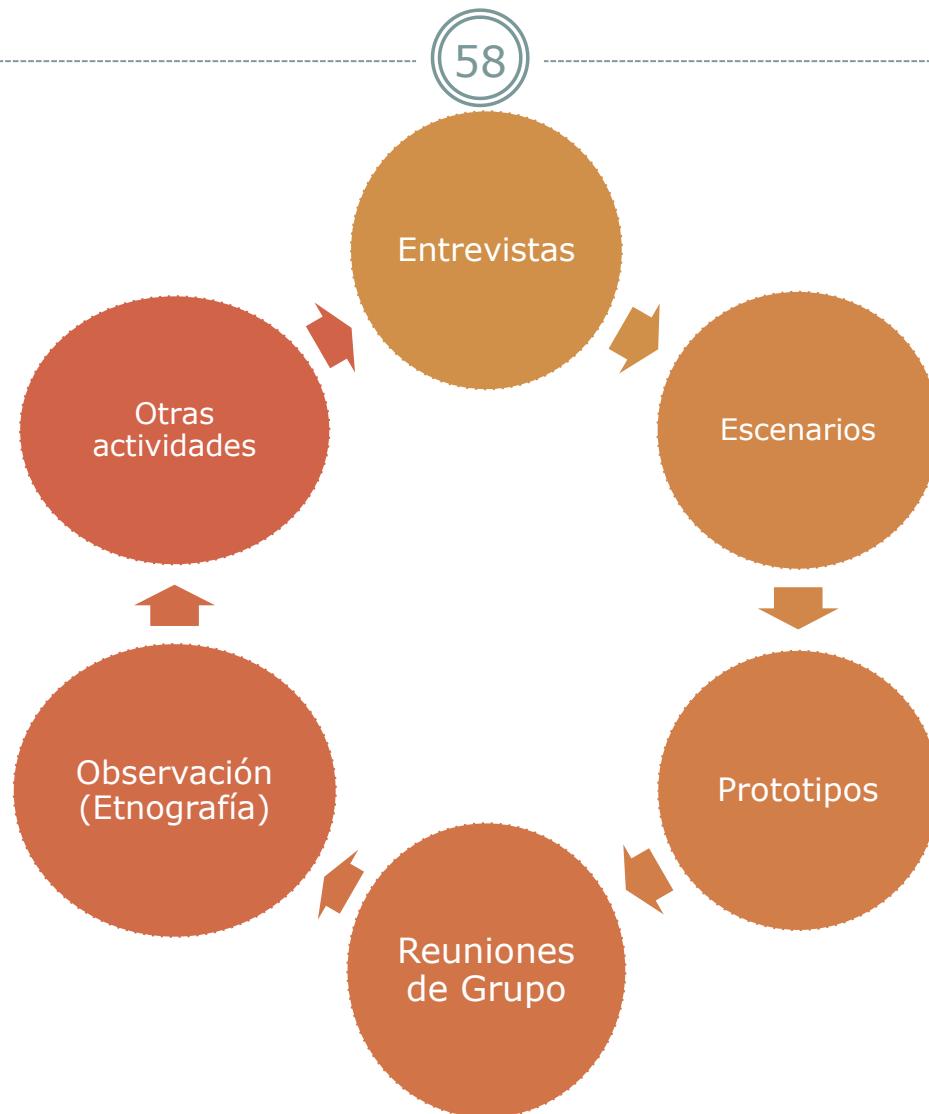
## El entorno operativo

- Requisitos derivados del entorno en el que se ejecutará el software.
- Ej.: Restricciones de tiempos para software de tiempo real o limitaciones de interoperabilidad para sistemas ofimáticos.

## El entorno de la organización

- Al que debe adaptarse el software.
- A menudo, el software es necesario para dar soporte a un proceso de negocio.
- Suele estar condicionado por la estructura, la cultura y la política interna de la organización.

# Técnicas de obtención de requisitos



# Entrevistas

59

- En las entrevistas (formales o informales) el equipo interroga a los interesados sobre el sistema que utilizan y el sistema a desarrollar.
- Hay dos tipos de entrevista
  - **Entrevistas cerradas** donde un conjunto predefinido de preguntas deben contestarse.
  - **Entrevistas abiertas**, donde no hay un programa predefinido y donde se analizan una serie de temas con los interesados.

# Entrevistas en la práctica

60

- Normalmente, una mezcla entrevistas cerradas y abiertas.
- Las entrevistas son buenas para conseguir una comprensión global de lo que los interesados hacen y cómo podrían interactuar con el sistema.
- Las entrevistas no son buenas para entender los requisitos de dominio
  - Los ingenieros software pueden no entender la terminología específica de un dominio.
  - Algunos de los conocimientos del dominio son tan familiares que los actores encuentran difícil articularlos o piensan que no vale la pena expresarlos.

# Entrevistas: Algunos tipos de preguntas

61

- Sobre detalles específicos.
- Sobre la visión de futuro que el entrevistado tiene sobre algo del sistema.
- Sobre ideas alternativas.
- Sobre una solución mínimamente aceptable.
- Acerca de otras fuentes de información.

# Escenarios

62

- Los escenarios son ejemplos reales de cómo se utilizará el sistema en la práctica.
- Proporcionan un contexto para la obtención de los requisitos del usuario.
- Permiten un marco de trabajo para realizar preguntas acerca de las tareas del usuario
  - ¿qué pasaría si?
  - ¿cómo se hace?

# Escenarios

63

- Deben incluir:
  - una descripción de la situación inicial,
  - una descripción del flujo normal de los acontecimientos,
  - una descripción de lo que puede salir mal,
  - información acerca de otras actividades concurrentes,
  - una descripción del estado del sistema cuando finalice el escenario.
- El tipo más común de escenario son los casos de uso.

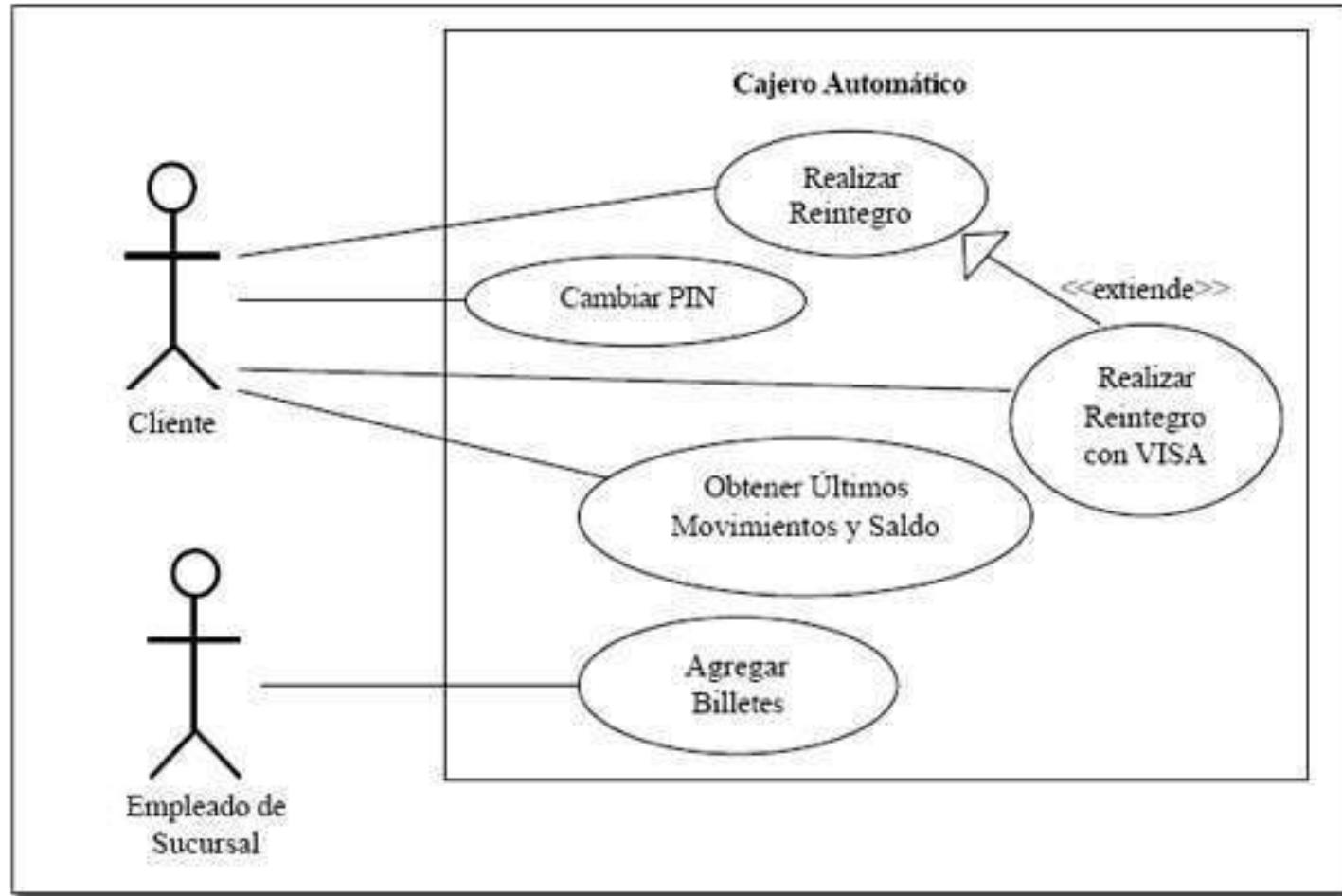
# Casos de uso

64

- Son una técnica **basada en escenarios** de **UML** que identifica a los actores involucrados en una interacción con el sistema y la describe.
- Un conjunto de casos de usos describiría todas las posibles interacciones con el sistema.
- Los diagramas de secuencia se pueden usar para añadir detalle a los casos de uso mostrando la secuencia de eventos procesados por el sistema.
- Los casos de uso se verán en profundidad en el **Tema 5**.

# Casos de Usos de un Cajero Automático

65



# Prototipos

66

Una herramienta valiosa para clarificar requisitos ambiguos.

Pueden actuar de manera similar a los escenarios, proporcionando a los usuarios un contexto dentro del cuál pueden comprender mejor cuál es la información que necesitan proporcionar.

Existe una amplia gama de técnicas de creación de prototipos,

- Desde maquetas en papel de los diseños de pantalla hasta versiones beta de los productos software

<http://www.balsamiq.com/products/mockups>

# Reuniones de grupo

67

Su propósito es tratar de unificar requisitos y obtener más información sobre las necesidades software de un grupo de participantes que trabajando individualmente.

- Pueden intercambiar y afinar ideas que pueden ser difíciles de sacar a la superficie a través de entrevistas.
- Aparecen los conflictos entre requisitos al principio.

Cuando funciona bien, esta técnica puede dar lugar a un conjunto más rico y más coherente de requisitos

Las reuniones se deben manejar con cuidado

- Necesidad de un facilitador
- Evitar que la capacidad crítica del equipo se resienta por la lealtad de grupo,
- Evitar que se favorezcan las necesidades de los que participan abiertamente (tal vez los superiores) en detrimento de otras personas.

# Análisis de requisitos

68



# Análisis de Requisitos

69

Detectar y resolver conflictos entre requisitos.

Descubrir los límites del software y  
cómo debe interactuar con su entorno.

Elaborar los requisitos del sistema para obtener,  
a partir de ellos, los requisitos del software a  
desarrollar.

# Análisis de requisitos

70

Clasificación  
de  
requisitos

Modelado  
conceptual

Negociación  
de los  
requisitos

# Clasificación de requisitos

71

Los requisitos se pueden clasificar según un numero de dimensiones:

- Si el requisito es **funcional** o **no funcional**
- Si el requisito se deriva de uno o más **requisitos de alto nivel** o de una **propiedad emergente** o lo **impone** directamente un participante o alguna otra fuente.
- Si el requisito es del **producto** o del **proceso**.

# Clasificación de requisitos

72

Los requisitos se pueden clasificar según un numero de dimensiones:

- **Ámbito del requisito**

- Medida en la cuál un requisito afecta al software y a sus componentes.
- Global (algún requisito no funcional) o particular

- **Volatilidad / estabilidad**

- Algunos requisitos cambian durante el ciclo de vida del software, e incluso durante el desarrollo
- Es útil tener alguna estimación de la probabilidad de cambio de los requisitos

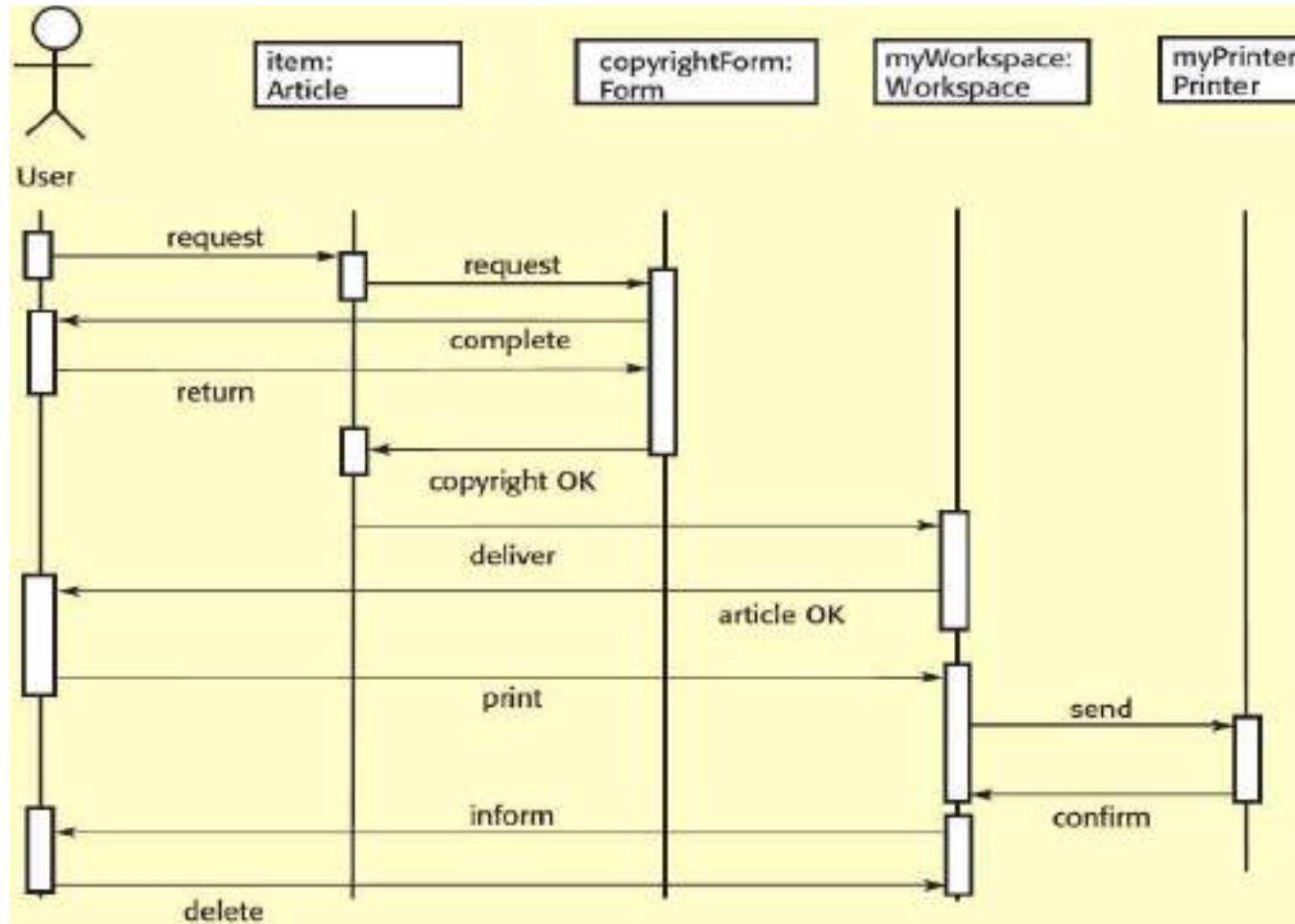
# Modelado conceptual

73

- Su propósito es ayudar a comprender el problema, en lugar de iniciar el diseño de la solución.
- El desarrollo de modelos de un problema del mundo real es clave para el análisis de requisitos del software.
- Se pueden desarrollar varios tipos de modelos:
  - Modelos de datos
  - Modelos de flujo de datos y de control
  - Modelos de comportamiento (estados)
  - Modelos de interacción
  - Etc...

# Imprimir un artículo (D. secuencia)

74



# Negociación de los requisitos

75

- O “resolución de conflictos”.
- Resolver los problemas con los requisitos cuando dos partes requieren características incompatibles entre sí.
- No es aconsejable que la decisión la tome el ingeniero software, debe conseguir un consenso entre las partes.
- Es importante, sobre todo por motivos contractuales, que estas decisiones se puedan “trazar” hasta el cliente.

# Índice

76

1. Introducción
2. Requisitos funcionales y no funcionales
3. Especificación de requisitos
4. Proceso de Ingeniería Requisitos
5. Obtención y análisis de requisitos
6. **Validación de requisitos**

# Validación de requisitos

77

- Se trata de demostrar que los requisitos definen el sistema que el cliente realmente quiere.
- El coste de los errores en los requisitos es alto por tanto la validación es muy importante
  - Arreglar un error de los requisitos después de la entrega puede costar hasta 100 veces el coste de arreglar un error de implementación.

# Validación de requisitos

## Métodos

78

### Revisión de los requisitos

- Un grupo de personas examina los requisitos buscando inconsistencias, malentendidos, puntos poco claros, conflictos y otros problemas similares.

### Prototipado

- Es una buena forma de probar cualquier producto.
- Gran ayuda para detectar problemas y clarificar requisitos.

### Validación del Modelo

- Si se ha utilizado notaciones de modelado es posible realizar ciertas comprobaciones automáticas.

### Pruebas de aceptación

- Elaborar un plan de aceptación mediante la verificación de requisitos.

# Cuestionario de validación de requisitos

79

<b>Conformidad con estándares</b>	La especificación en su conjunto, ¿es conforme a los estándares definidos? ¿es conforme a los estándares definidos cada uno de los requisitos?
<b>Seguimiento (Trazabilidad)</b>	¿Pueden identificarse únicamente los requisitos? ¿Incluyen referencias o enlaces a otros requisitos relacionados así como las razones para incluirlos?
<b>Estructuración</b>	¿Está estructurado el documento de requisitos? ¿Están agrupados los requisitos relacionados entre sí? ¿Con otra estructura serían más fáciles de entender?
<b>Ambigüedad</b>	¿Son ambiguos algunos requisitos? ¿Pueden darse distintas interpretaciones de algunos de los mismos?
<b>Comprensión</b>	¿Son comprensibles los requisitos? ¿Puede un lector entender lo que significa cada uno de ellos? ¿Están libres de detalles de diseño e implementación?
<b>Consistencia</b>	¿Son consistentes los requisitos? ¿Hay alguna contradicción entre algunos requisitos?
<b>Completitud</b>	¿Es completo el conjunto de requisitos? ¿Falta algún requisito? ¿Cada requisito de forma individual es completo? ¿Falta alguna información?
<b>Relevancia</b>	¿Individualmente es cada requisito pertinente para el problema y su solución?
<b>Gestionable</b>	¿Están expresados de forma que un cambio no afecte demasiado al resto? ¿Se han identificado las dependencias entre requisitos?
<b>Viabilidad</b>	¿Es posible implementar todos los requisitos con los recursos disponibles? ¿Son viables dadas las restricciones de coste y plazos?

# Validación de requisitos

## Revisiones

80

	Completo	Consistente	No ambiguo	Estructurado	Comprensible	Trazable
Req 1	X		X	X		X
Req 2		X		X	X	X
Req 3	X		X		X	X
...						
Req N	X	X	X			X

Tabla para validación de requisitos

# Introducción a la Ingeniería del Software



## TEMA 5. MODELADO DE SISTEMAS CON UML

Grado en Ingeniería Informática  
Grado en Ingeniería del Software  
Grado en Ingeniería de Computadores

# Índice

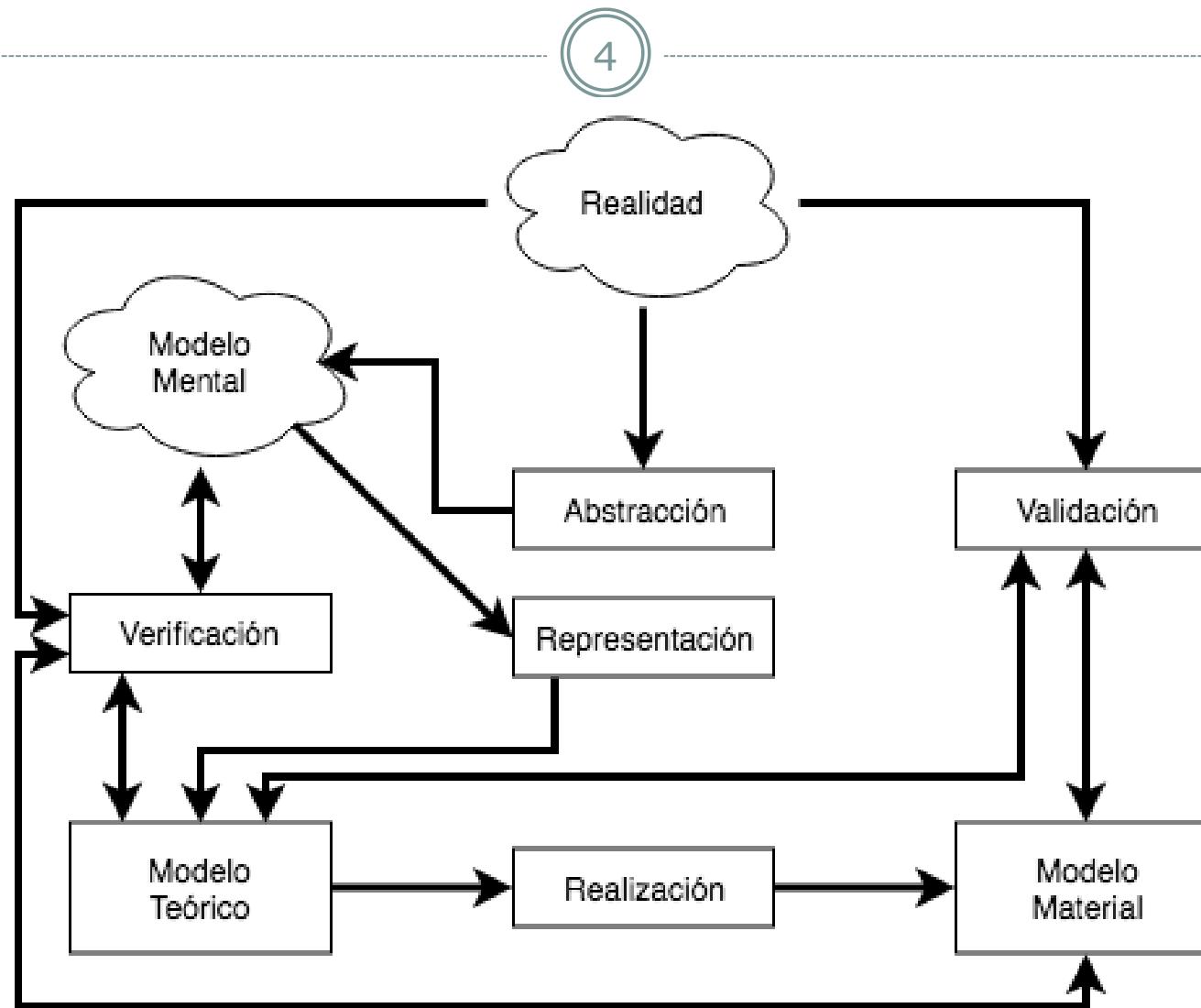
2

- 1. Introducción**
- 2. Modelos de contexto**
- 3. Modelos de interacción**
- 4. Modelos estructurales**
- 5. Modelos de comportamiento**

# 1. Introducción

3

# El proceso de modelado



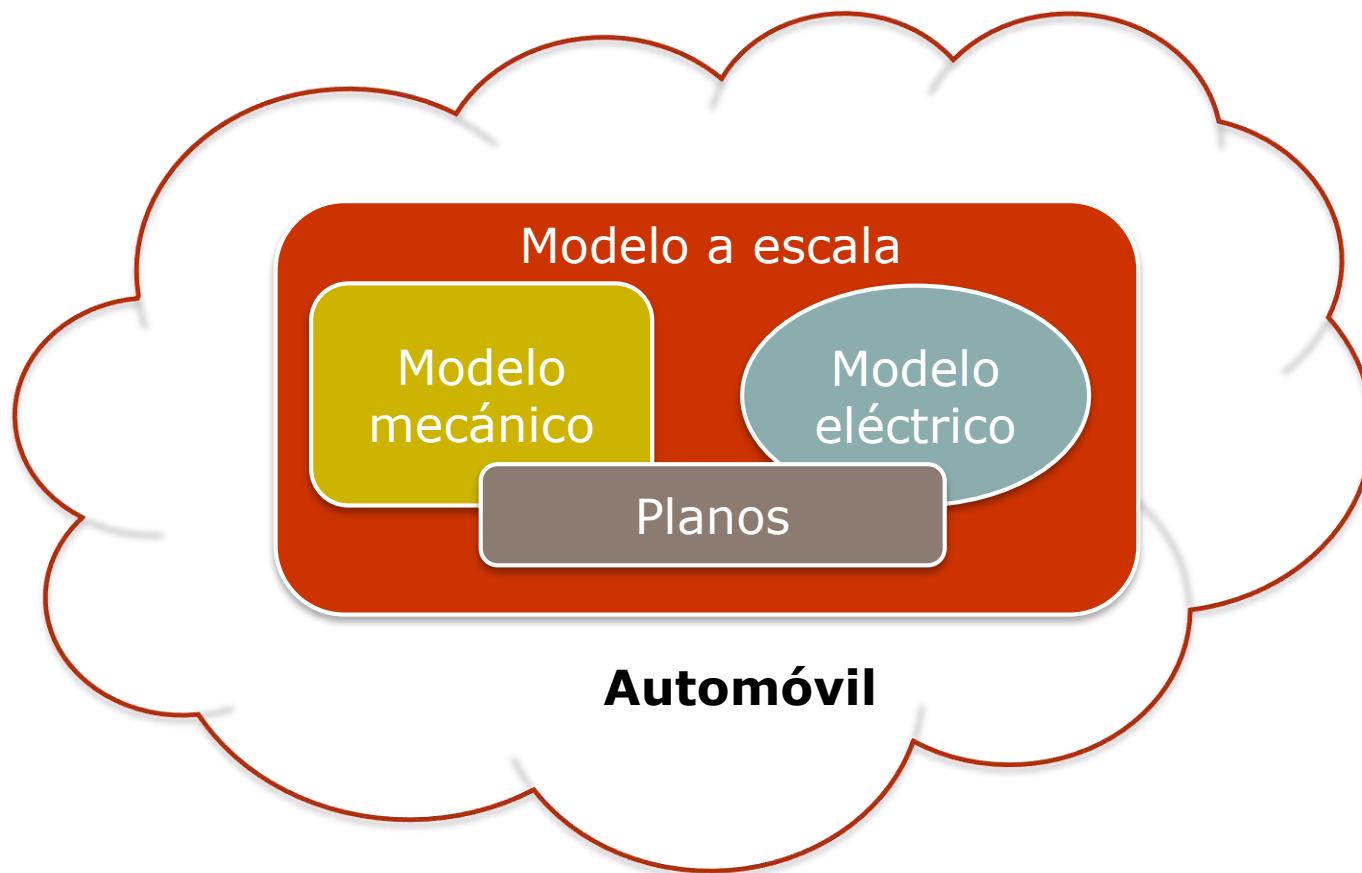
# ¿Qué es un modelo?

5

- Un modelo es una **simplificación de nuestra percepción de una realidad que no siempre existe** (p.ej. invención).
- El modelo es una **abstracción de la realidad** que nos **permite quedarnos con los aspectos más relevantes** de la misma.
  - El modelo es más simple de manejar y examinar.
  - Nos permite hacer predicciones (p.ej. meteorología).
- Validez de los modelos:
  - **Validación:** concordancia con la realidad.
  - **Verificación:** concordancia entre modelos.

# Vistas de un modelo

6



# Modelado de sistemas software

7

- **¿Para qué se usan?**

- El modelado del sistema permite al analista **entender la funcionalidad del sistema**.
- Los modelos son utilizados para **comunicarse con los clientes** y el equipo de desarrollo.
- Los diferentes modelos presentan diferentes **perspectivas (vistas) del sistema**.
- Representar un sistema usando algún tipo de notación gráfica, que actualmente está casi siempre basada en UML.

**“Lo más importante del modelado es obviar los detalles” Sommerville**

# Perspectivas del sistema

8

## Perspectiva externa

- Muestra el contexto o entorno donde debe funcionar el sistema.

## Perspectiva de interacción

- Muestra las interacciones entre el sistema y su entorno o entre los componentes internos del sistema.

## Perspectiva de comportamiento

- Modela el comportamiento dinámico del sistema y la forma en la que responde a los distintos eventos.

## Perspectiva estructural

- Modela la organización del sistema o la estructura de los datos que se procesan en el sistema.

# Uso de modelos gráficos

9

Son un medio para facilitar la discusión sobre el sistema propuesto (o que ya existe)

- Modelos **incorrectos** o **incompletos** son válidos si su papel es ayudar en la discusión

Pueden servir para documentar un sistema existente

- Los modelos deberían ser una representación precisa del sistema, pero **no** necesitan ser **completos**

Pueden servir para describir detalladamente el sistema

- Se puede utilizar para generar una implementación del sistema
- Los modelos tienen que ser **correctos** y **completos**

# Lenguaje UML

10

- El **lenguaje Unificado de Modelado (UML)** es un lenguaje de modelado gráfico para:
  - Describir y diseñar sistemas software.
  - Especialmente orientado a diseño OO.
- Desarrollado por Grady Booch, Ivar Jacobson y James Rumbaugh a mediados de los 90.
- UML está organizado en un conjunto de elementos de modelado:
  - Vistas.
  - Diagramas.
  - Bloques de modelado.
  - Mecanismos

# 2. Modelos de Contexto

12

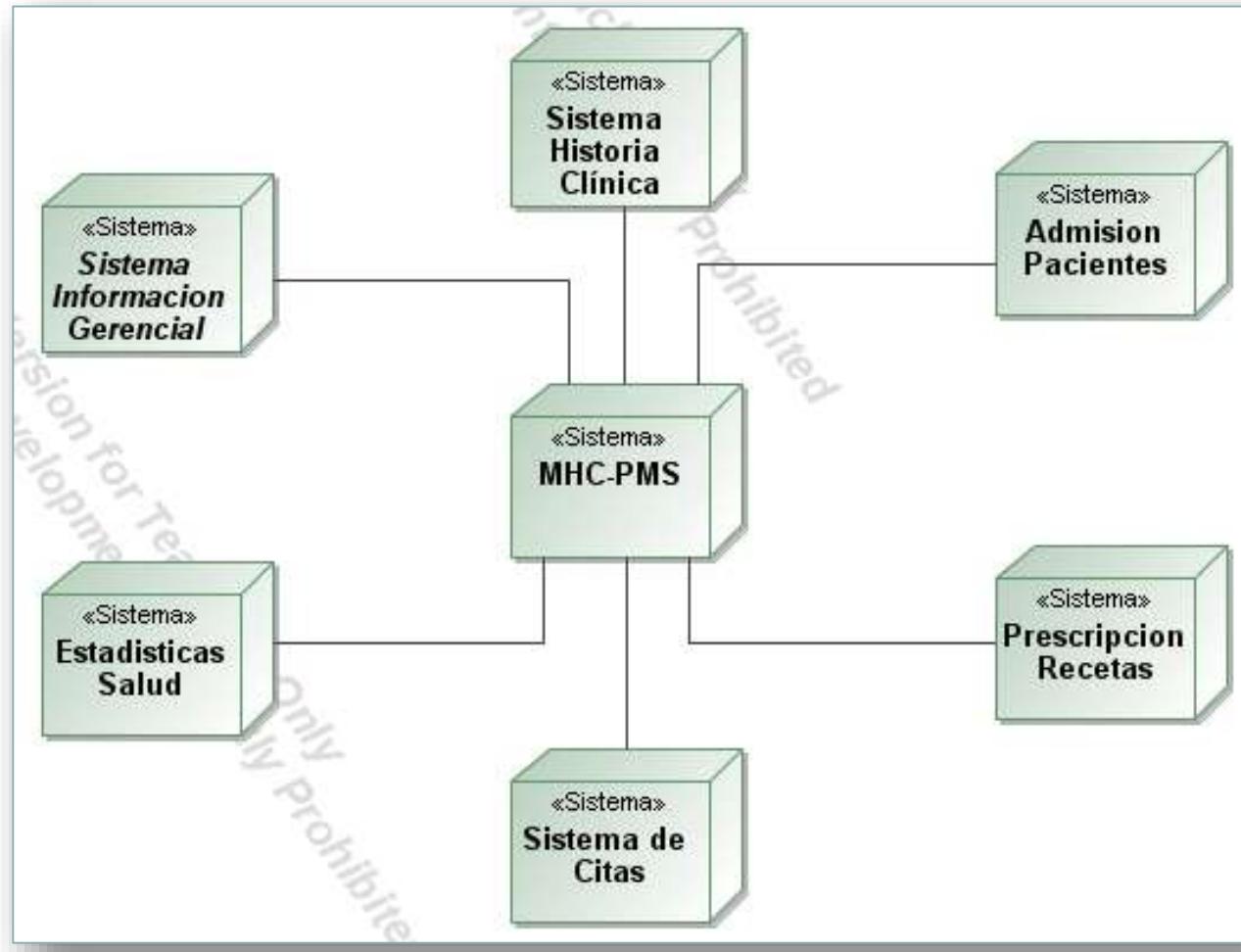
# Modelos de Contexto

13

- Se usan para mostrar el **contexto de funcionamiento del sistema** y para ilustrar sus **Límites**.
  - Los **Límites del sistema** definen qué está dentro y que está fuera del sistema.
  - Muestran otros sistemas que se utilizan o dependen del sistema en desarrollo.
- La posición de los límites del sistema tiene un efecto profundo sobre los requisitos del sistema.
- Definir un límite del sistema es una decisión *política*
  - Es posible que haya presiones para definir los límites del sistema que aumentarán o reducirán la influencia o la carga de trabajo de las diferentes áreas de una organización.
- Los **modelos arquitectónicos** muestran el sistema y sus relaciones con otros sistemas.

# Modelo de contexto de un sistema de gestión de pacientes MHC-PMS (Mental Health Care-Patient Management System)

14



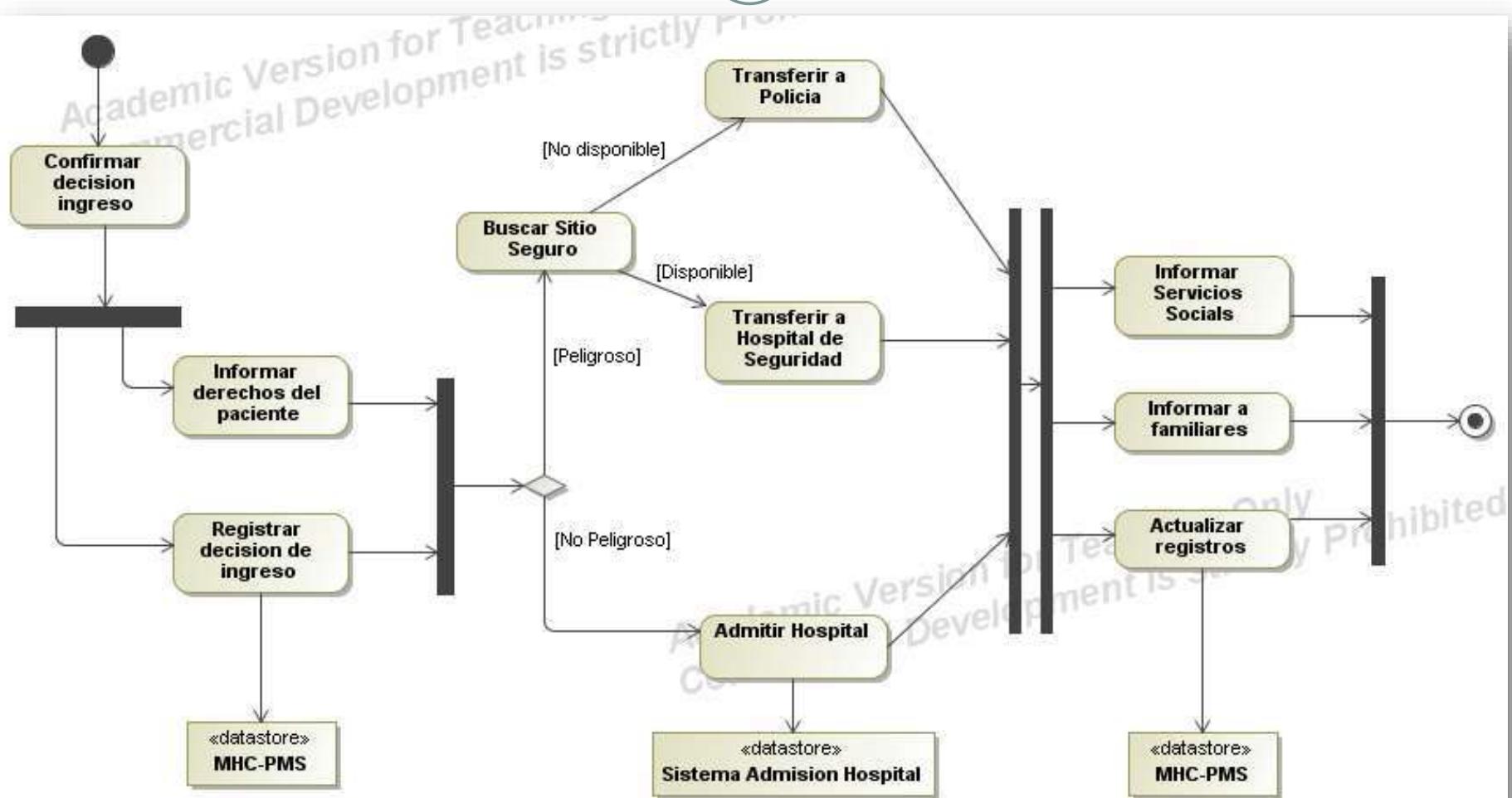
# Perspectiva de procesos

15

- Los modelos de contexto sólo muestran a otros sistemas que se relacionan con el nuestro.
- Pero no muestran cómo nuestro sistema se utiliza en dicho contexto.
  - Para esto se utilizan **modelos de proceso**.
- Los **diagramas de actividad de UML** se pueden utilizar para definir Modelos de Procesos de Negocio
  - BPM - Business Process Models

# Modelo de procesos para ingreso involuntario (diagrama de actividad)

16



# 3. Modelos de Interacción

17

# Modelos de Interacción

18

- Modelar las **interacciones del usuario** es importante ya que ayuda a identificar los requisitos de estos.
- Modelar las **interacciones de sistema a sistema** pone de relieve los problemas de comunicación que puedan surgir.
- Modelar la **interacción entre componentes del sistema** ayuda a entender si la estructura del sistema propuesta es probable que proporcione el rendimiento y la fiabilidad solicitada.
- Se pueden utilizar los **diagramas de casos de uso** y de **secuencia**.

# Modelos de Casos de uso

19

- Los casos de uso fueron desarrollados originalmente para **ayudar en la obtención de requisitos**.
- Cada caso de uso representa una tarea discreta que implica la interacción con un sistema externo.
- Los **actores** de un caso de uso pueden ser personas u otros sistemas.
- Se representan de forma gráfica mediante un **diagrama de casos de uso** (UML) y se detallan de forma textual mediante algún documento o plantilla

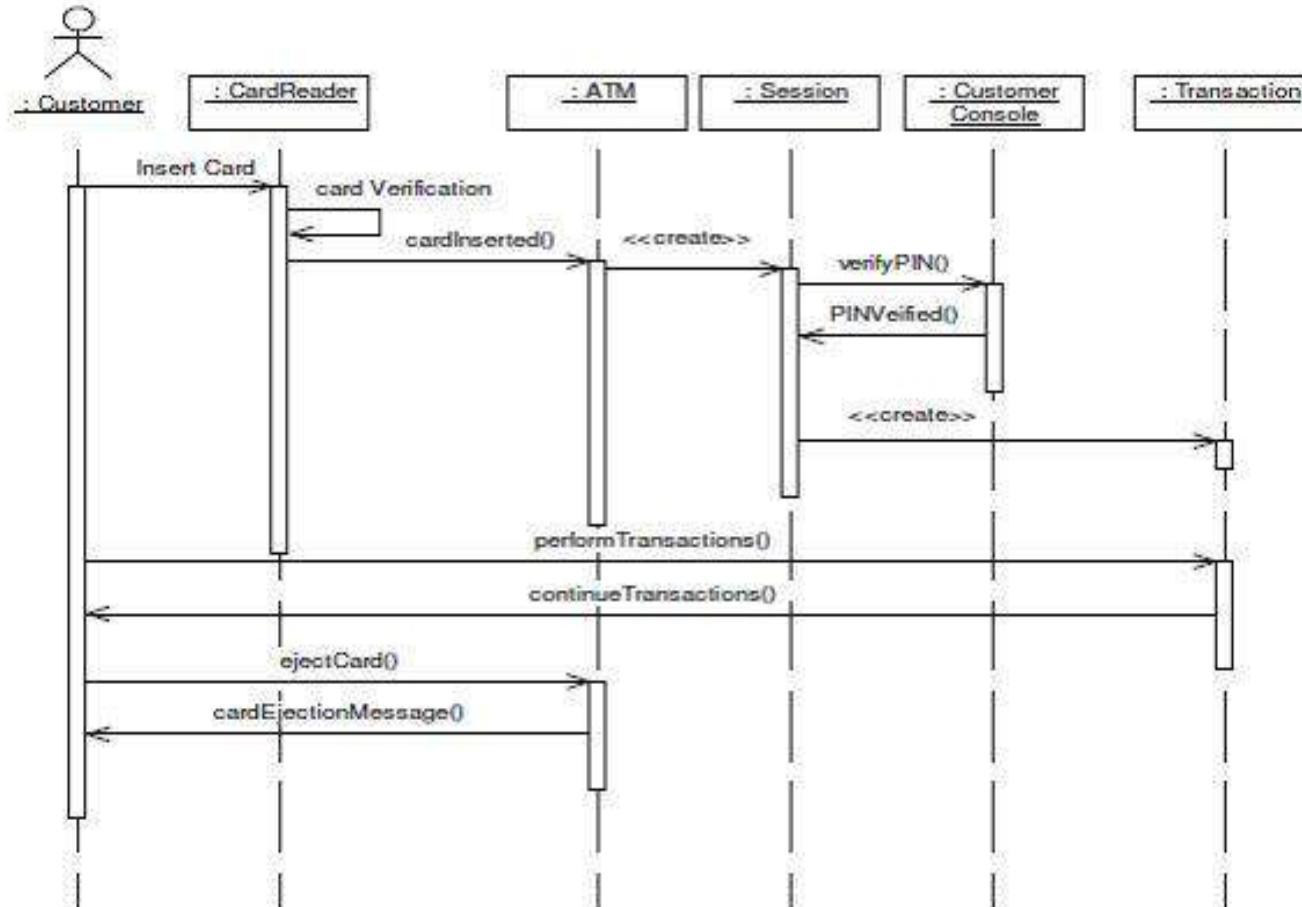
# Diagramas de secuencia

20

- Los **diagramas de secuencia** se utilizan para modelar las interacciones entre los actores y los objetos dentro de un sistema.
- Un diagrama de secuencia muestra la secuencia de interacciones que tienen lugar durante un caso de uso particular (**escenario**).
- Los objetos y los actores involucrados se muestran en la parte superior del diagrama, con una línea de puntos trazada verticalmente a partir de éstos
- Las interacciones entre los objetos se indica por las flechas con etiquetas

# Diagrama de secuencia para un cajero

21



# 4. Modelos Estructurales

22

# Modelos estructurales

23

- Los modelos estructurales muestran la organización de un sistema en términos de los componentes que conforman ese sistema y sus relaciones.
- Los modelos estructurales pueden ser
  - modelos **estáticos**, que muestran la estructura del diseño del sistema, o
  - modelos **dinámicos**, que muestran la organización del sistema cuando se está ejecutando.
- Se pueden crear modelos estructurales de un sistema para discutir el diseño de su arquitectura.

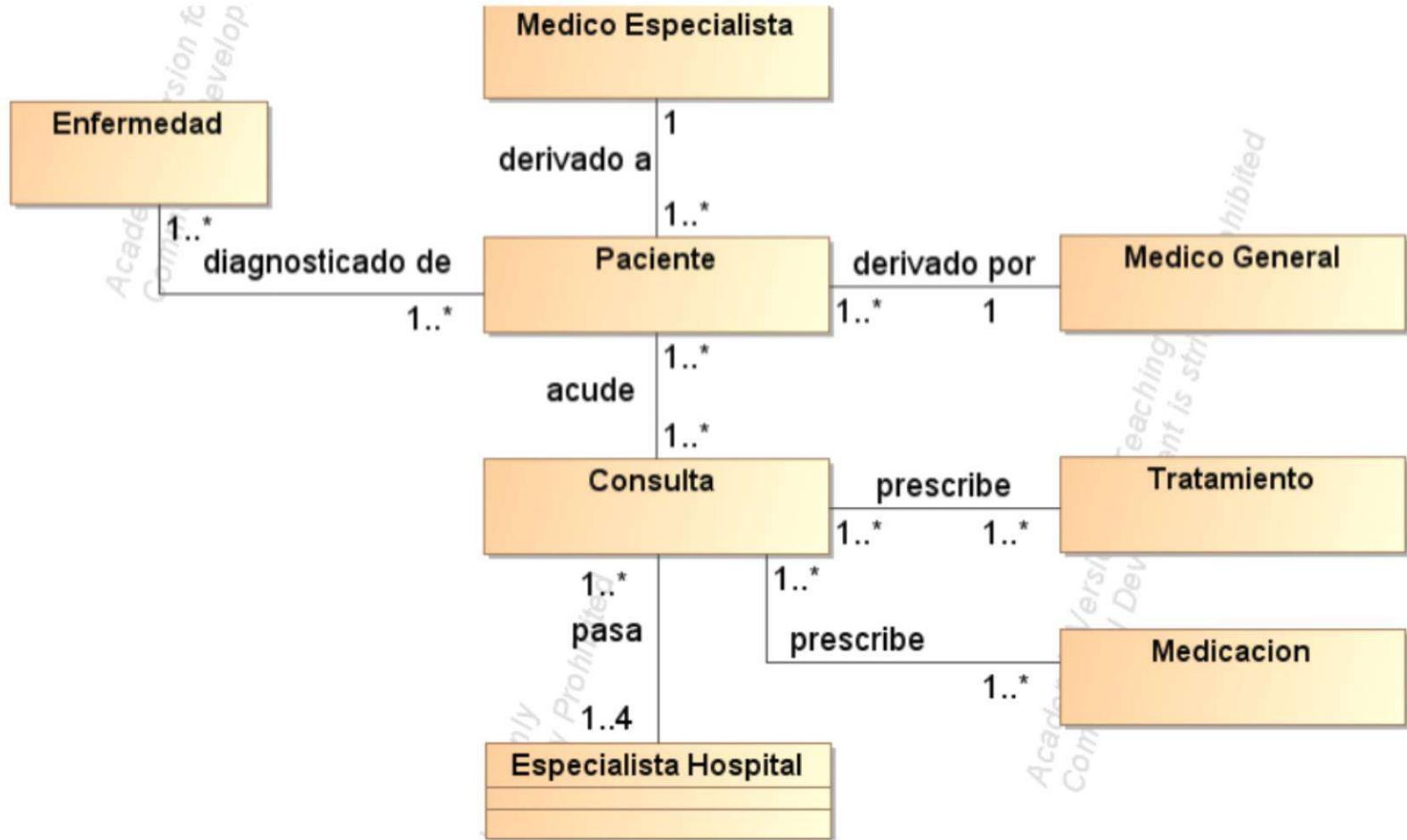
# Diagramas de clases

24

- Los **diagramas de clases** se utilizan para mostrar las clases del sistema bajo desarrollo y las asociaciones entre estas clases.
- Una **clase** se puede considerar como una definición general de un tipo de objeto del sistema.
- Una **asociación** es un enlace entre clases que indica que existe alguna relación entre esas clases.
- Durante las primeras etapas del proceso de ingeniería de software los objetos representan algo en el mundo real.
  - Ej.: un paciente, una receta médica, médico, etc. (Modelar el problema)

# Clases y asociaciones en el sistema MHC-PMS

25



# La clase consulta

26

## Consulta

- facultativos
- fecha
- hora
- centro
- motivo
- medicacion prescrita
- tratamiento prescrito
- notas verbales
- hoja de consulta

- +New()
- +recetar()
- +grabarNotas()
- +copiarHistoria()

# 5. Modelos de Comportamiento

27

# Modelos de Comportamiento

28

- Son modelos del comportamiento dinámico de un sistema que se está ejecutando.
- Muestran lo que sucede cuando el sistema responde a un estímulo de su entorno.
- Se puede pensar en estos estímulos como de dos tipos:
  - **Datos**. Algunos datos que llegan al sistema tienen que ser procesados.
  - **Eventos**. Ocurre algún evento que desencadena el procesamiento en el sistema. Los eventos pueden tener datos asociados.

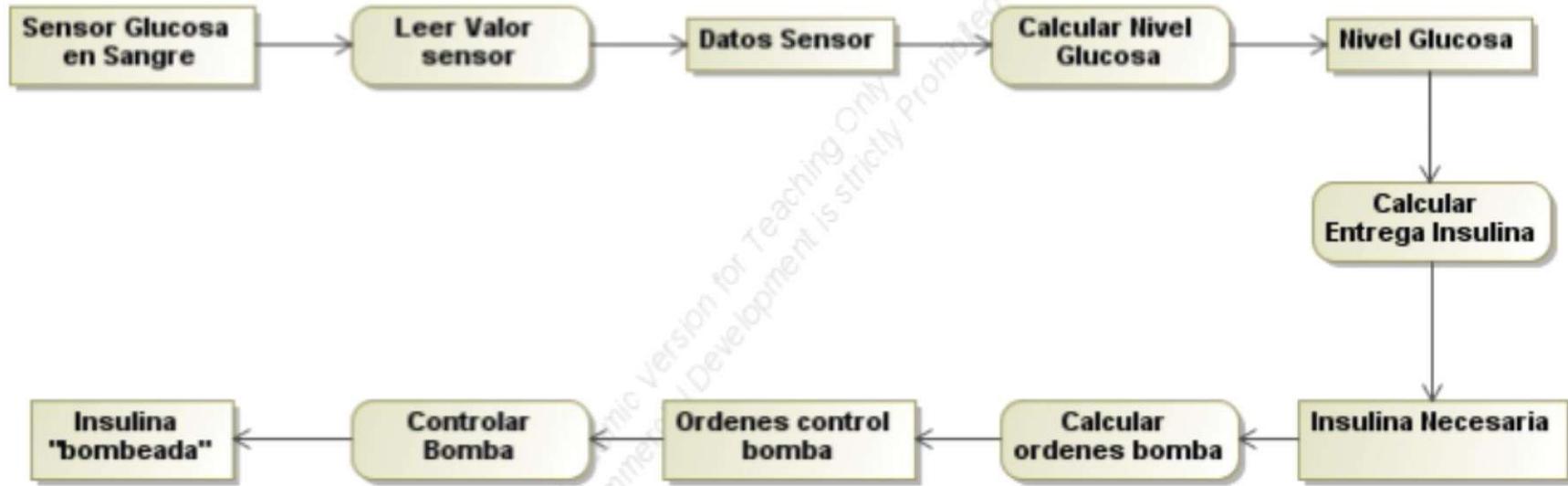
# Modelos dirigidos por Datos

29

- Muchos sistemas empresariales son sistemas de procesamiento de datos que están fundamentalmente basados en esos datos.
- Están controlados por la entrada de datos al sistema, con relativamente poco procesamiento de eventos externos.
- Los modelos dirigidos por datos muestran la secuencia de acciones implicadas en el procesamiento de los datos de entrada y en la generación de la salida asociada a ellos.
- Son particularmente útiles durante el análisis de requisitos, ya que se pueden utilizar para mostrar el resultado extremo a extremo del procesamiento de un sistema.

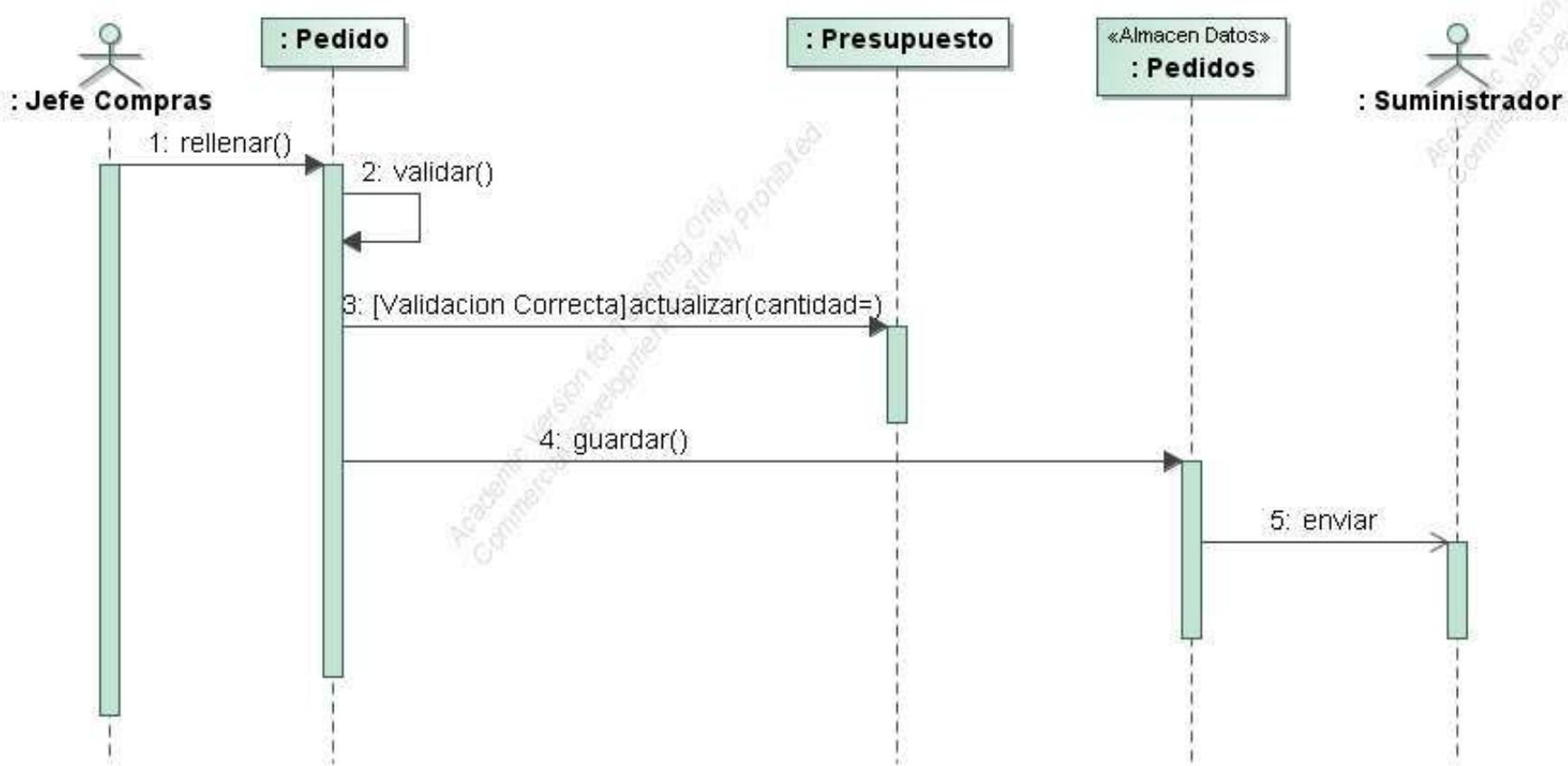
# Diagrama de actividad del funcionamiento de una bomba de insulina

30



# Procesamiento de un pedido (diagrama de secuencia)

31



# Modelos dirigidos por eventos

32

- Los sistemas de tiempo real están frecuentemente basados en eventos, con un mínimo procesamiento de datos.
  - Ej.: un sistema de conmutación de telefonía fija responde a eventos como "receptor descolgado", generando un tono de marcación.
- Los Modelos dirigidos por eventos muestran cómo responde el sistema a los eventos externos o internos.
- Se basa en la suposición de que un sistema tiene un número finito de **estados** y que los eventos (estímulos) pueden causar una **transición** entre estos estados

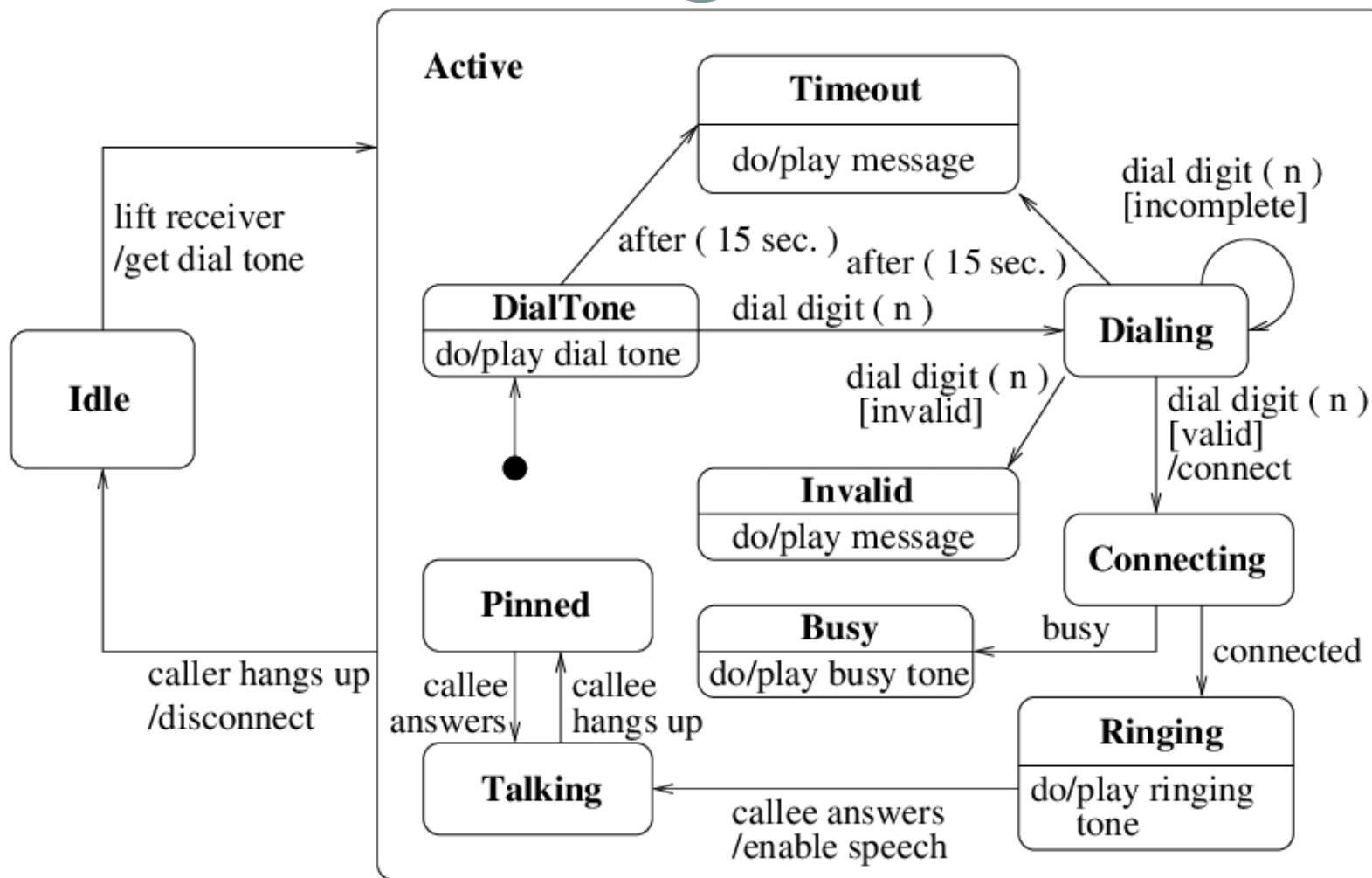
# Modelos de Máquinas de Estado

33

- Modelan el comportamiento del sistema en respuesta a eventos externos e internos.
- Usados a menudo para sistemas de tiempo real, muestran las respuestas frente a estímulos.
- Muestran los estados del sistema como nodos y los eventos como arcos entre nodos
  - Cuando ocurre un evento, el sistema se mueve de un estado a otro.
- Los **diagramas de estado** son parte integral de UML.

# Ejemplo de máquina de estados

34



# Modelado con UML



## 5.1. CASOS DE USO

# Introducción

2

- **¿Qué son?** Los diagramas de casos de uso muestran la funcionalidad del sistema desde el punto de vista de un observador externo.
  - **Qué hace el sistema** sin importar cómo lo hace.
- Capturan los requisitos funcionales del sistema.
- Definen el límite entre el sistema y los elementos externos.

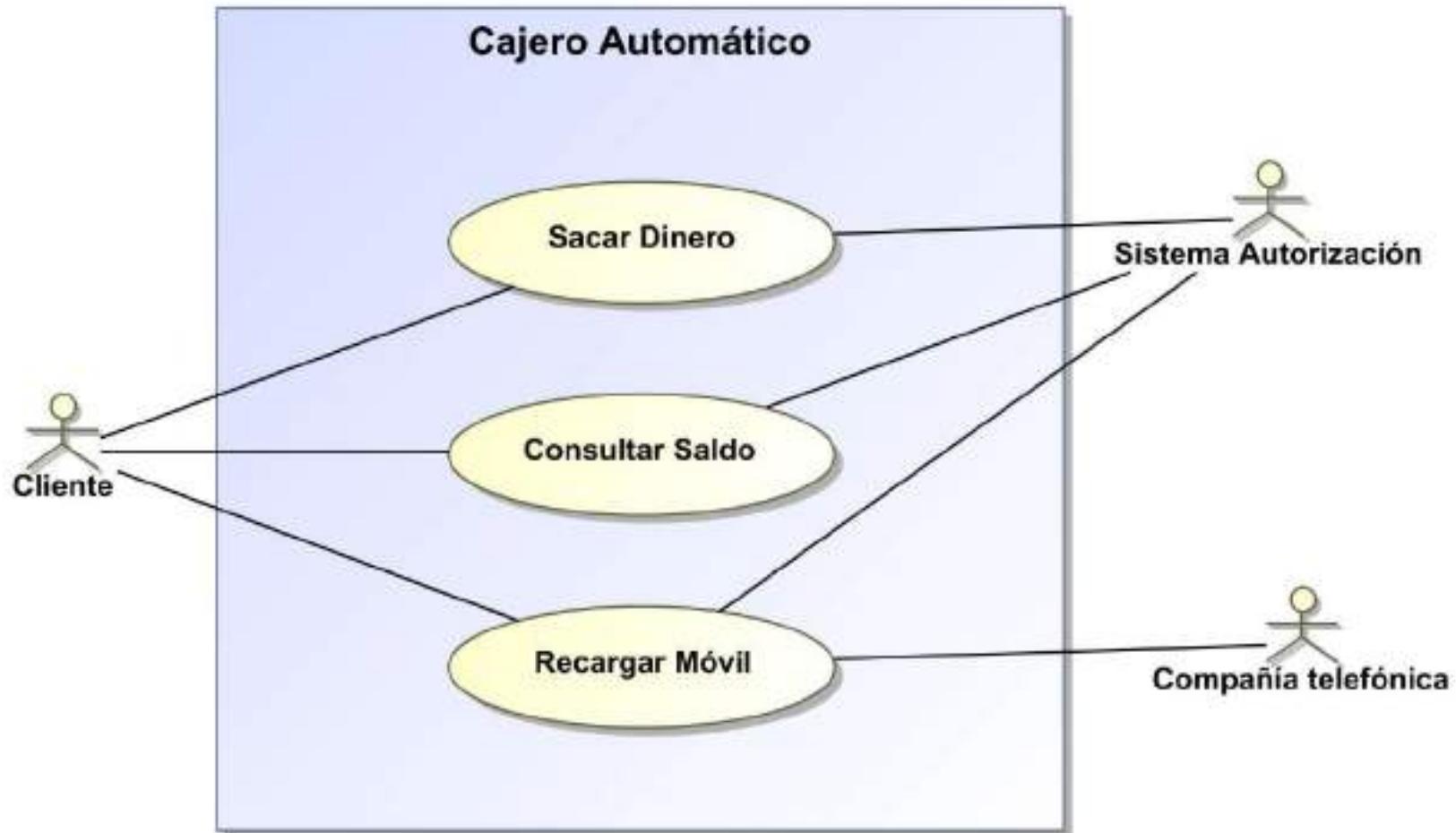
# Elementos del diagrama

3

- Un diagrama de casos de uso es un grafo constituido por
  - **Actor**
    - Representa a cualquier elemento que intercambia información con el sistema.
    - Son entidades externas al sistema.
  - **Caso de uso**
    - Conjunto de secuencias de acciones e interacciones entre los actores y el sistema objeto de estudio para obtener una función o capacidad (requisito funcional).
  - **Asociación**
    - Representa una relación entre los elementos del diagrama.
  - **Escenario**
    - Describe una secuencia concreta del caso de uso durante una ejecución del sistema.
    - Es decir, es una instancia del caso de uso.

# Ejemplo: Cajero Automático

4



# Actores

5

- Son quién/qué **inicia los eventos involucrados en una tarea (están fuera del sistema)**.
- Pueden ser:
  - **Usuarios con un rol en el sistema.**
    - Ej. En una agencia de viajes
      - Usuarios consultas posibles viajes.
      - Agentes añaden posibles viajes.
      - Administradores mantienen el sistema.
  - **Dispositivos de E/S**, como sensores y/o actuadores, siempre que sean independientes de la acción de un usuario.
    - Ej. En un regulador de temperatura, el sistema actúa en función de:
      - Usuario establece la temperatura ideal.
      - El sensor de temperatura se encarga de activar/desactivar el calefactor.

# Actores

6

- **Sistemas externos** con los que el sistema se tiene que comunicar.
  - Ej. En un cajero automático de un banco:
    - El cajero se comunica con un servicio de autorización para validar cliente, obtener dinero,...
- **Temporizador o reloj** en sistemas de tiempo real.
  - Ej. El sistema hace algo periódico como respuesta a un evento iniciado por un reloj (comprobar temperatura).

# Caso de Uso

7

- Describen una interacción entre el actor y el sistema
  - 1. Captura alguna función **visible** para el usuario.
  - 2. Logra un objetivo **discreto** para el usuario (de principio a fin, tiene que acabar).
  - 3. Especifica una funcionalidad **completa** (no se busca descomposición funcional del sistema).
  - 4. Puede ser pequeño o grande.
  - 5. Comparable a la definición clásica de “transacción”.
- Ejemplos:
  - Introducir un pedido.
  - Inicializar un cliente nuevo.
  - Generar una factura.



# Escenarios

8

- Un escenario es una secuencia de pasos que describe una interacción entre un usuario y el sistema.
- Proporciona un ejemplo de lo qué ocurre cuando alguien interactúa con el sistema.
- **Relación entre casos de uso y escenarios:**
  - Un escenario es una concreción de un caso de uso.
  - Dicho de otra forma, un caso de uso es un conjunto de escenarios que comparten un objetivo de usuario común.

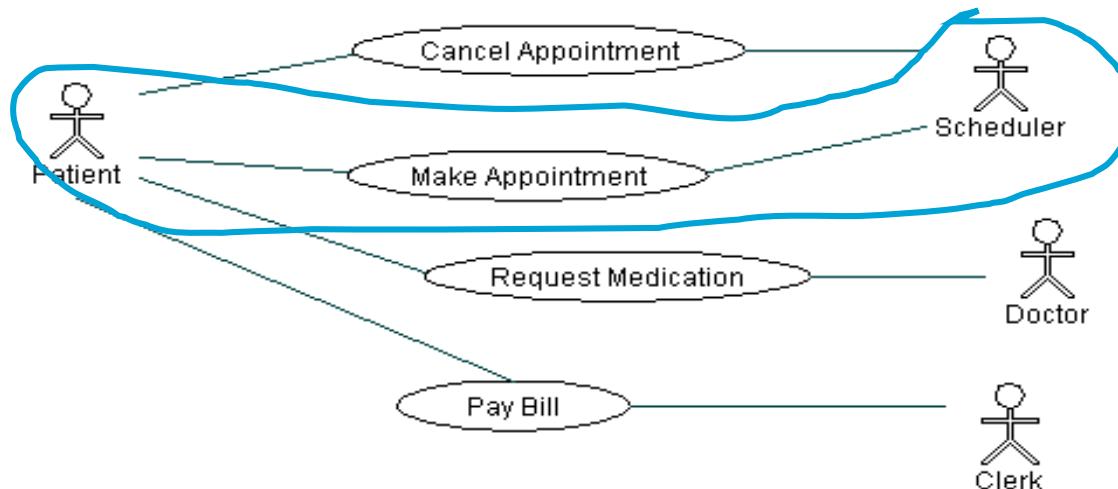
# Escenarios

9

**Ejemplo:** Un paciente llama a la clínica para concertar una cita para un chequeo anual.

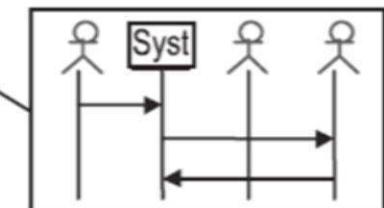
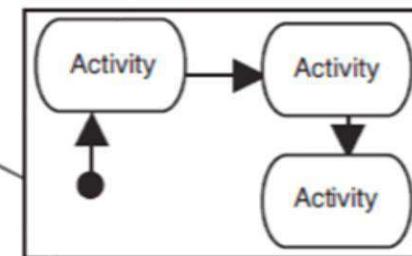
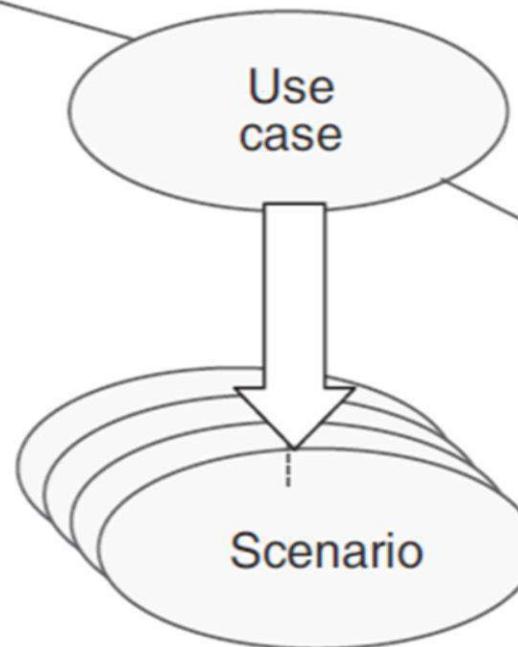
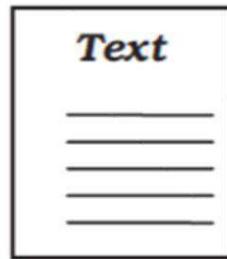
La recepcionista encuentra la cita libre más cercana en el registro de citas y programa la visita para esa cita.

**Flujo alternativo:** ¿y si no hay citas disponibles en la clínica?



# Descripción gráfica de casos de uso

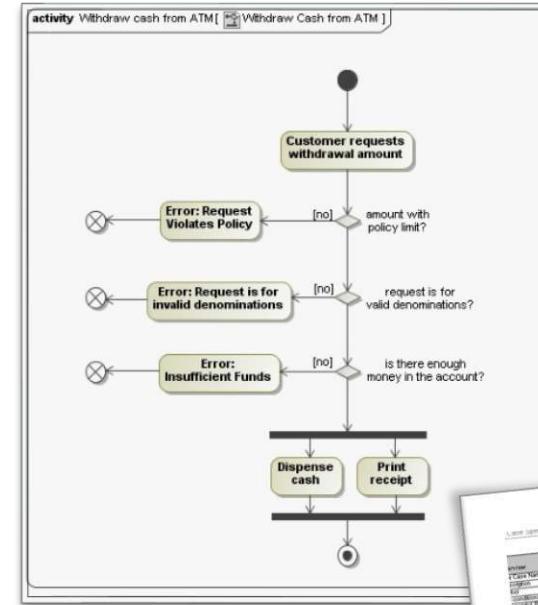
10



# Especificación de un Caso de uso

11

- Cada caso de uso debe tener una especificación verificable.
- Especificación se puede hacer usando:
  - Uno o varios **diagramas de actividad** o **de interacción**.
  - Un documento o **plantilla con texto**.



The document is a 'User Specification' for an ATM system. It includes sections for:

- Header: Project Name, Version, Date, Author, Description, Reviewer, Date Created.
- Complaints: Description, Origin, Date Created.
- Use Requirements: Title, Description, Author, Date Created.
- System Requirements: Description, Author, Date Created.
- Actor Filter: Actor, Action, System Action.
- Variables: Name, Type, Description, Author, Date Created.
- System Actions: Description, Author, Date Created.

# Especificación textual

12

- La descripción se centra en lo que debe hacerse, no en la manera de hacerlo.
- Definir varios escenarios posibles (escenario principal y alternativos)
  - Cada uno identifica diferentes secuencias de interacciones o eventos entre los actores y el sistema.
  - Detallar cada escenario narrativamente.
- Si el escenario alternativo es complejo, separar con una nueva descripción.
- La secuencia de eventos de los escenarios se representar mediante un **diagrama de secuencia**.

# Especificación Textual

13

- Documento de especificación de un caso de uso
  - **Identificador único**
  - **Declaración de objetivos** (propósito del caso de uso)
  - Autor
  - Prioridad
  - Riesgos
  - Supuestos
  - **Precondiciones y activación**
  - **Escenario principal**
  - **Escenarios alternativos**
  - Finalización
  - **Postcondiciones**
  - Problemas o temas a destacar
  - Requisitos funcionales relacionados
  - Requisitos no funcionales relacionados
  - Maquetas GUI

# Especificación Textual - Plantilla

14

Use Case Name	Register Loan
Current Status	Designed, but not implemented
Goal Statement	Register a book loan
Author	Edita
Priority	High
Pre-conditions	The librarian (user) must already be authenticated and authorized
Post-conditions	Reading item loan is registered to the customer



# Especificación Textual – Plantilla

15

Flow of Events	<ol style="list-style-type: none"><li>1. The reader asks to borrow a book</li><li>2. Librarian checks the reader's status to verify that the reader may borrow a book (alt 1)</li><li>3. Librarian checks verifies that the reading item is available (alt 2)</li><li>4. Verify that the book does not exceed the loan limit (alt 3)</li><li>5. Librarian marks the reading item as loaned to the reader</li><li>6. Librarian gives reading item to the reader</li></ol>
Alternative Flows	<ol style="list-style-type: none"><li>1) The reader cannot borrow reading items if he/she is penalized for not returning reading items – error message</li><li>2) The book is not available – error message</li><li>3) The reader cannot take more than the limit of items for the loan – error message</li></ol>
Non-Functional Requirements (Security, Performance, etc.)	The reader must not have to wait more than 30 seconds for the book loan to be registered
Notes	



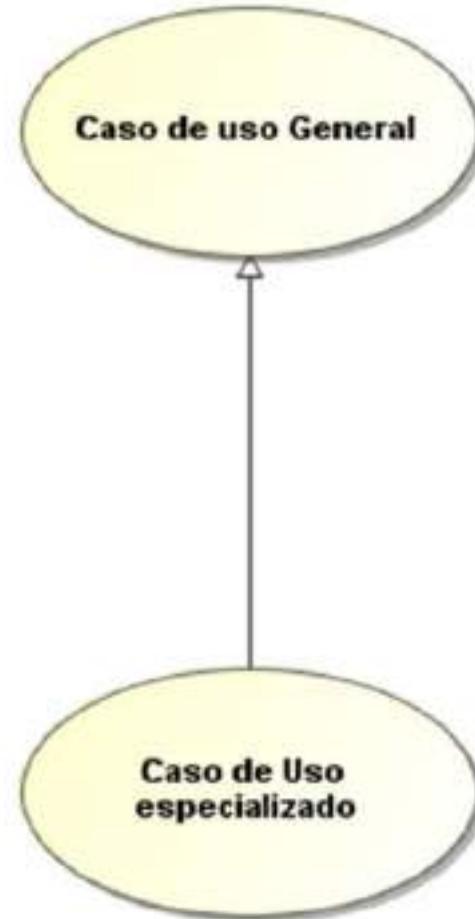
# Relaciones entre casos de uso

16

Relación	Función	Notación
<b>Asociación</b>	Interacción entre un actor y un caso de uso en el que participa.	—
<b>Generalización</b>	Relación entre un caso de uso general y otro más específico que hereda características y añade otras.	→
<b>Incluye</b>	Inserción de un fragmento de comportamiento dentro del caso de uso que lo incluye. Se usa para “factorizar”.	<<include>> ----->
<b>Extiende</b>	Inserción de comportamiento adicional a un caso de uso bajo determinadas condiciones.	<<extend>> ----->

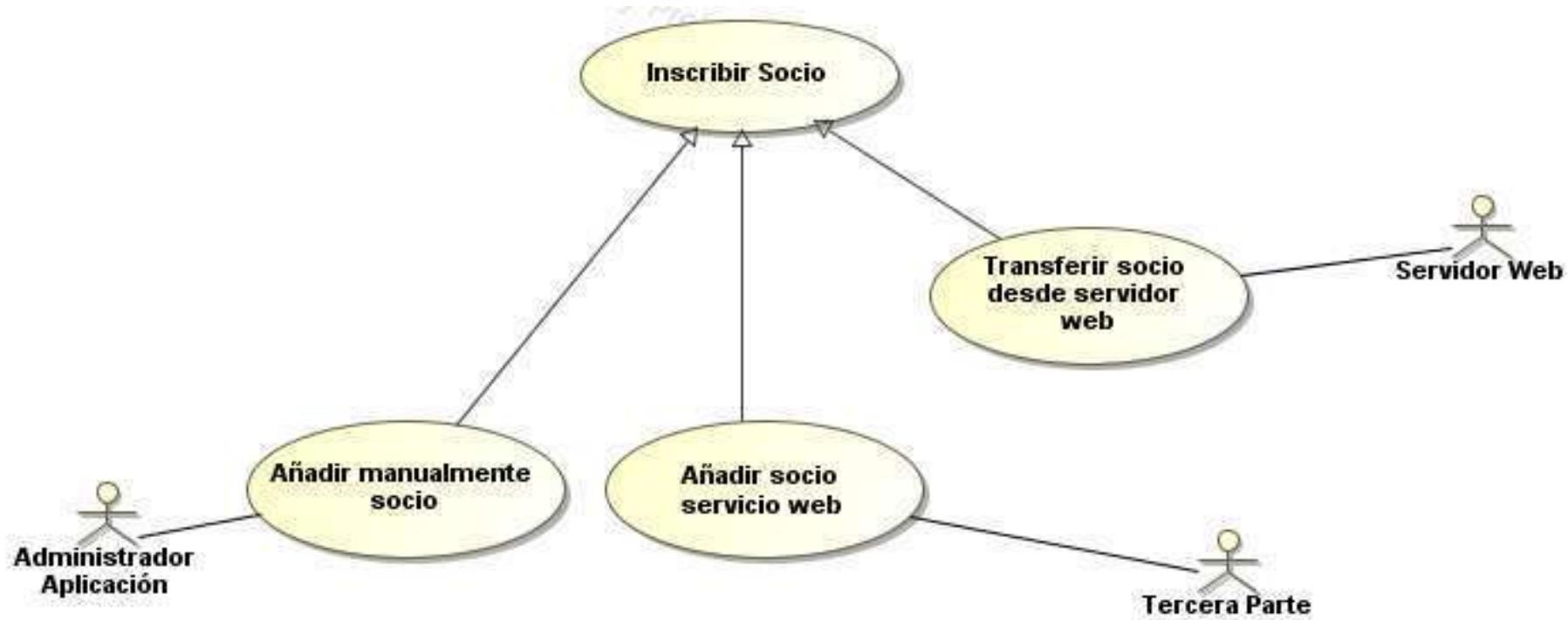
# Generalización de casos de uso

17



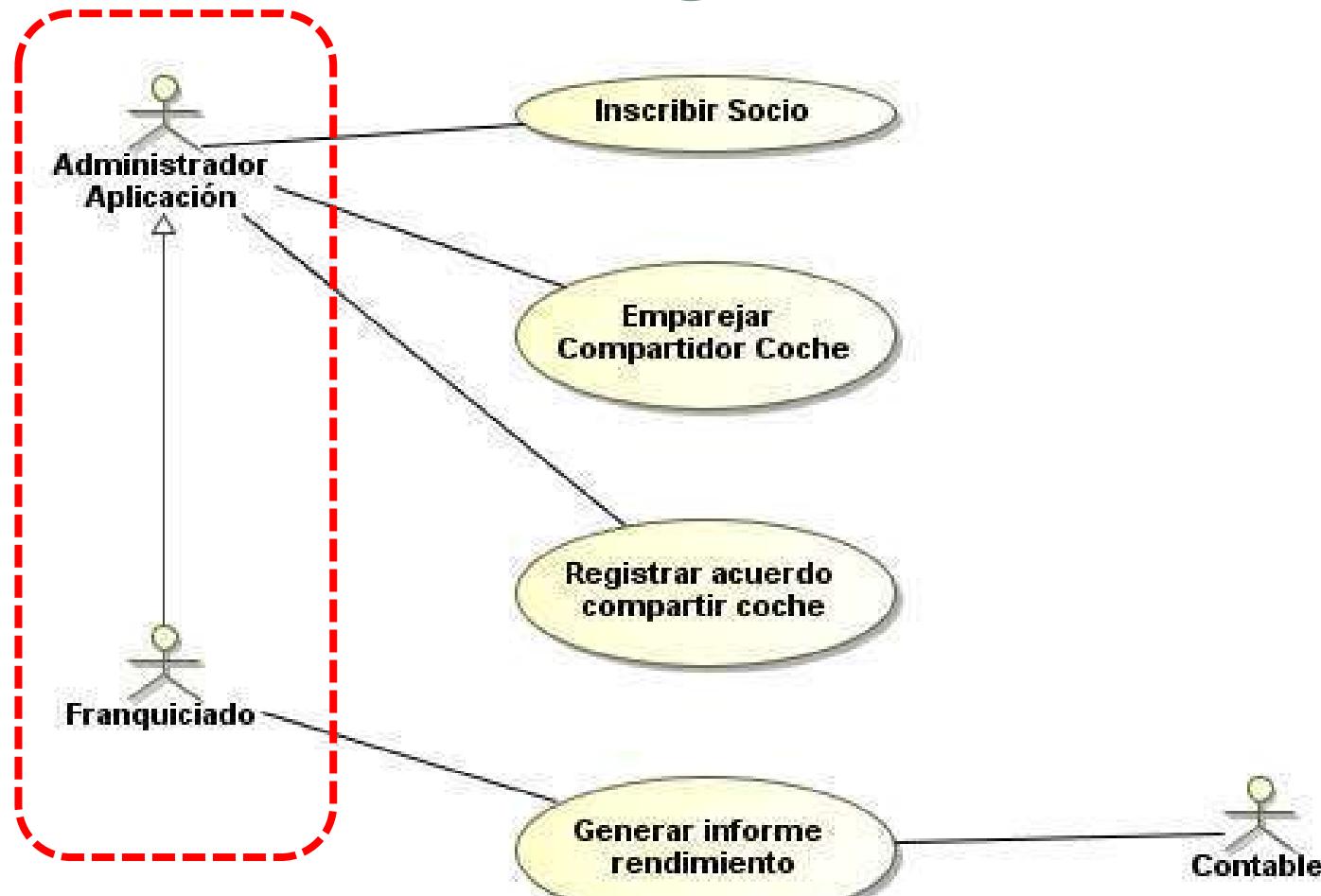
# Generalización de casos de uso

18



# Generalización de Actores

19

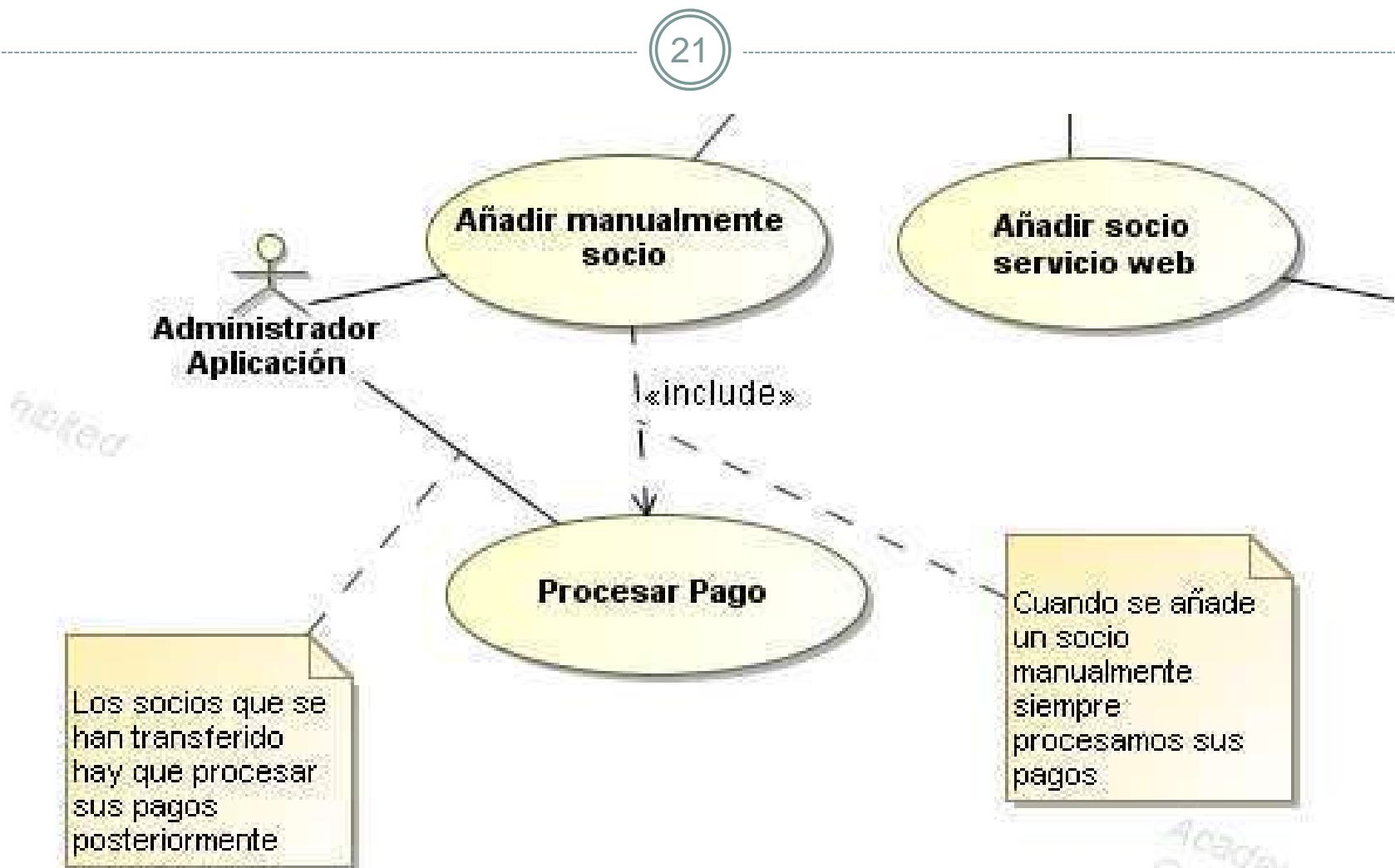


# Relación <<include>>

20

- Una relación de inclusión denota que un caso de uso está incluido en otro.
- Se da cuando
  - un caso de uso se utiliza por sí mismo,
  - y además otro casos de uso incluyen siempre esa funcionalidad.
- También aparecen cuando dos o más casos de uso comparten una funcionalidad.
- Denota que el caso de uso **SIEMPRE** está incluido en el otro caso de uso.

# Relación <<include>>



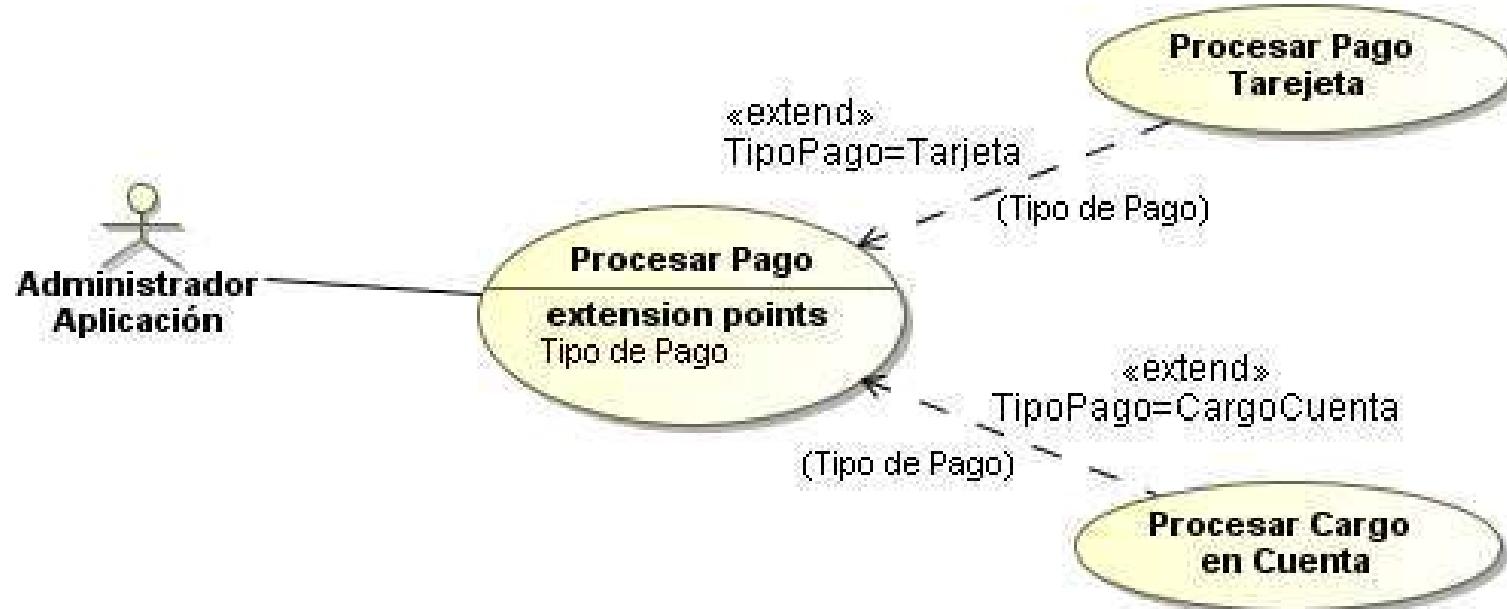
# Relación <<extended>>

22

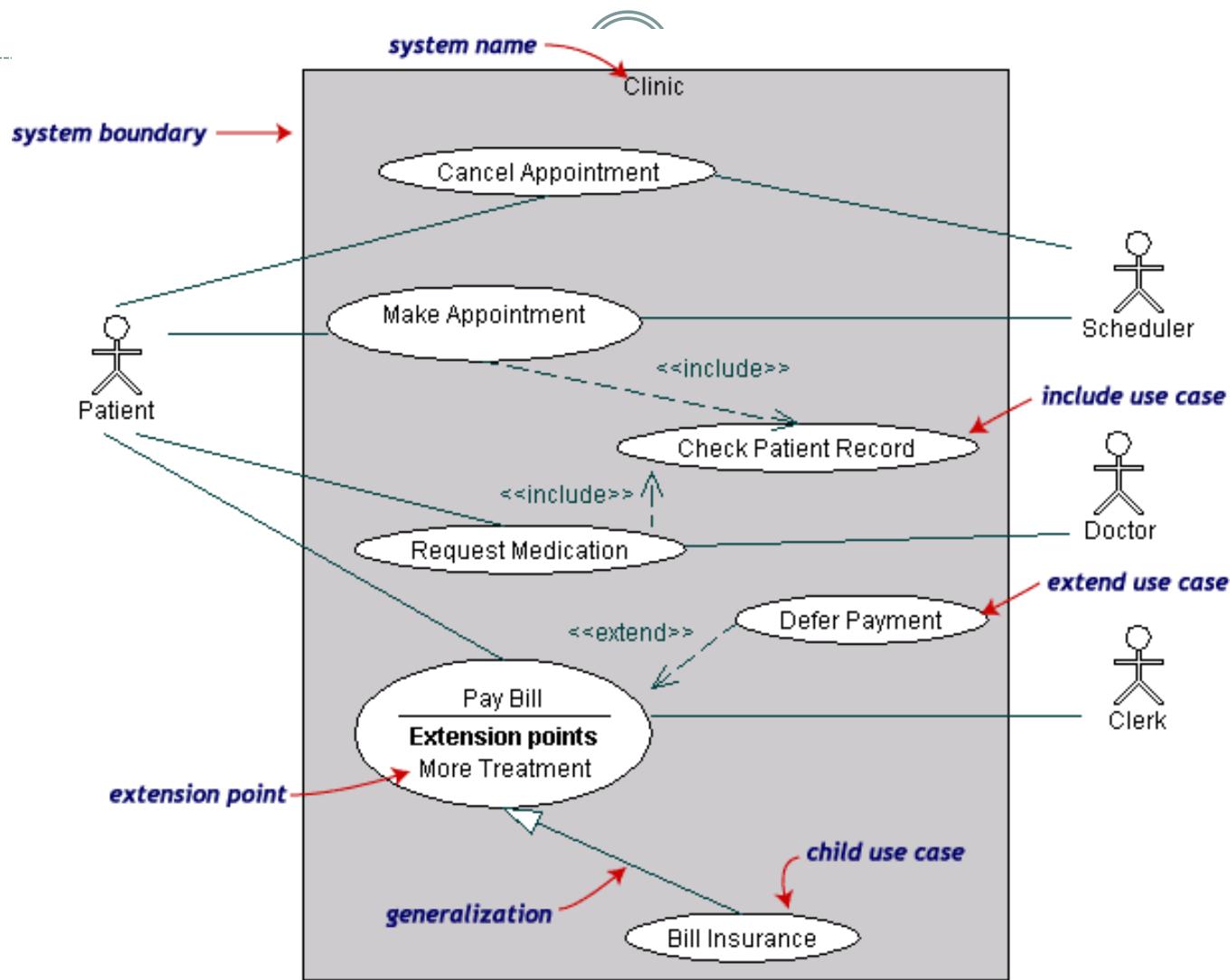
- Indica que **OPCIONALMENTE** un caso de uso es ampliado (extendido) por otro caso de uso
- El caso de uso base declara uno o más **puntos de extensión** (condiciones) cuando se amplia el caso de uso.

# Relación <<extended>>

23



# Ejemplo



# Ejercicio

25

- Dibujar el **diagrama de casos de uso** para una **máquina expendedora de café**.
- Se considerarán, al menos, los siguientes actores:
  - Cliente
  - Técnico (se encarga de reponer productos)
- Se permite el pago en metálico y con tarjeta.
- Describir el caso de uso “Comprar café”.
- **Software de dibujo:** MagicDraw o Draw.io

# Modelado con UML



## 5.2. DIAGRAMAS DE CLASES

# Contenido

2

- Clases y Objetos
- Diagrama de Clases – Relaciones
- Agregación y Composición
- Generalización
- Clases de Análisis
- Ejercicio

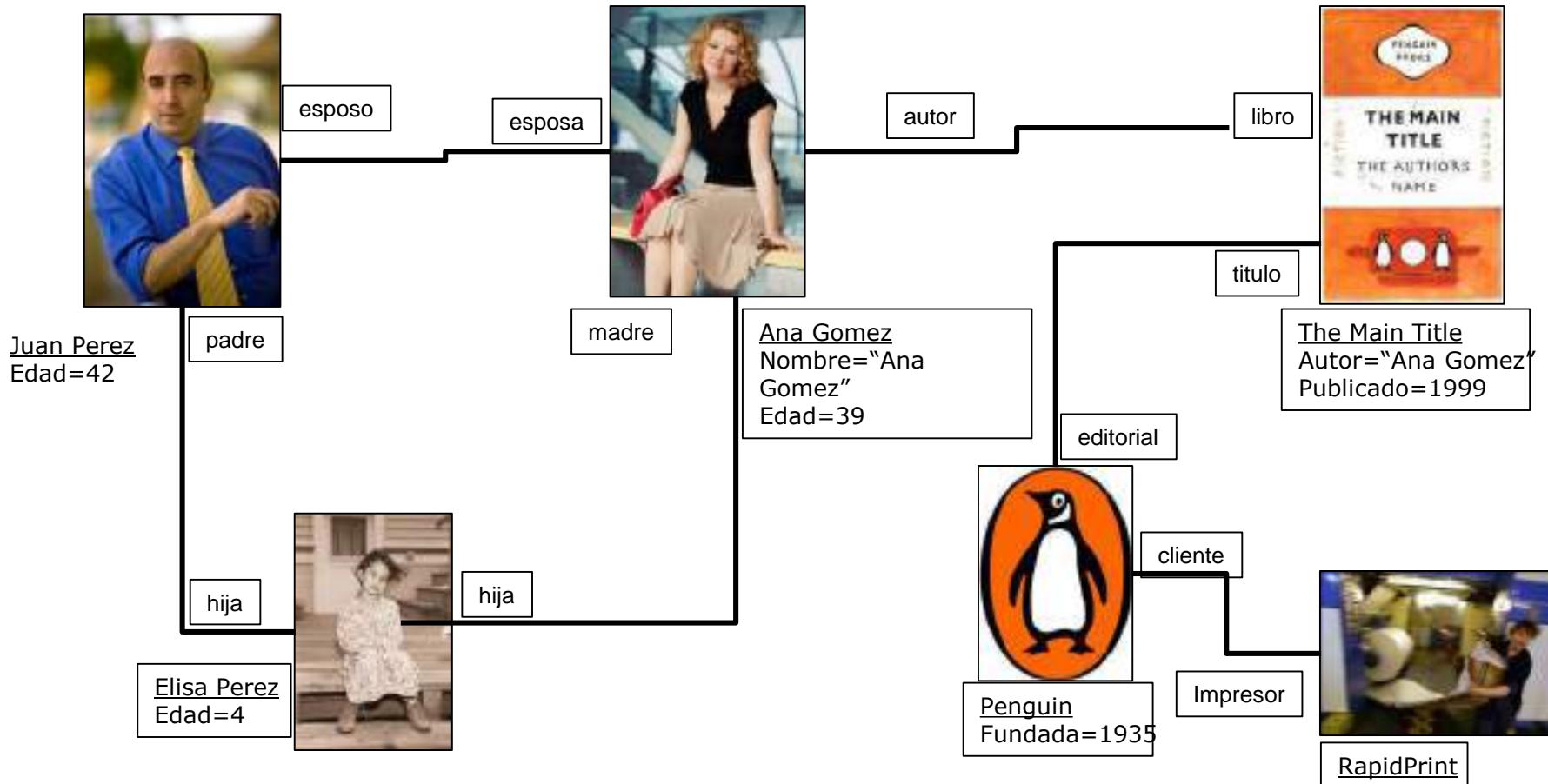
# Clases y Objetos

3

# Objetos

4

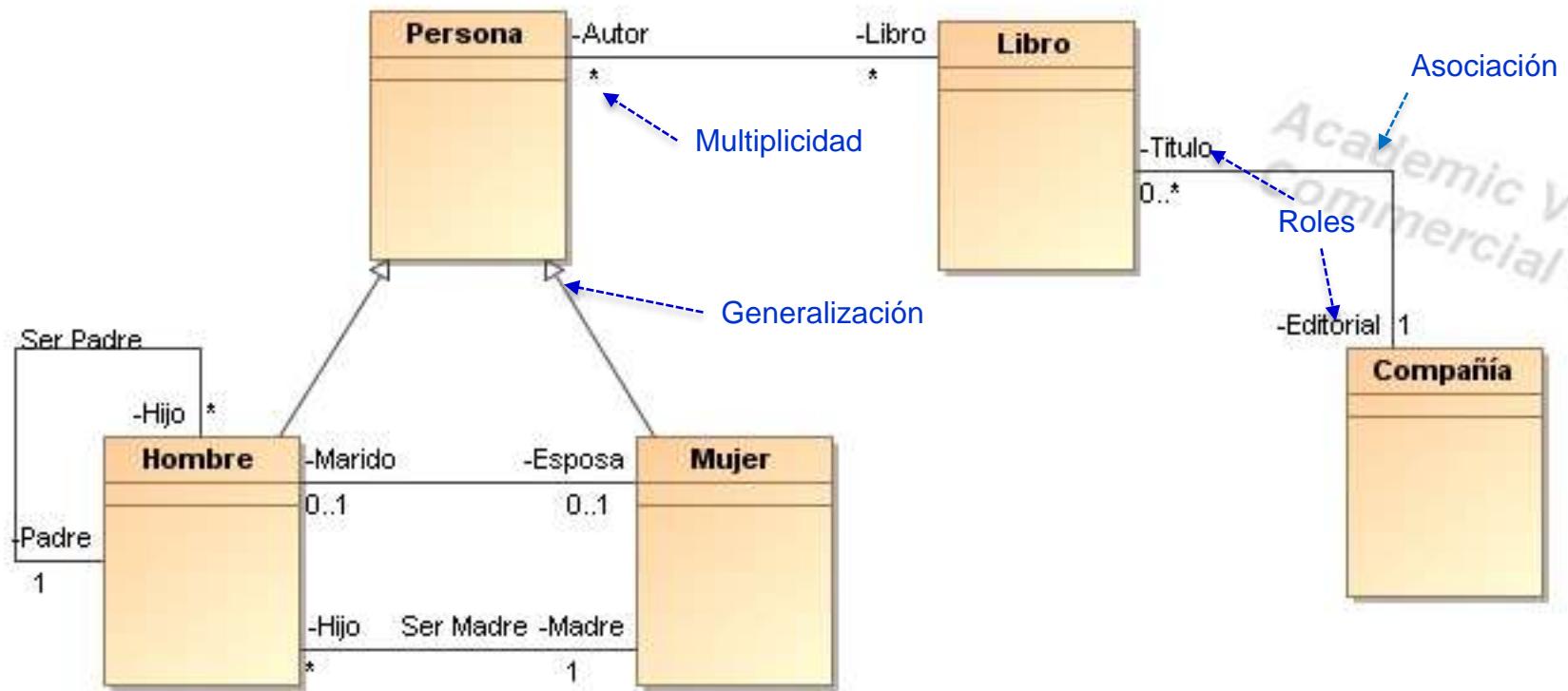
Un modelo de objetos, los valores de atributos reales y las relaciones entre ellos



# Clases

5

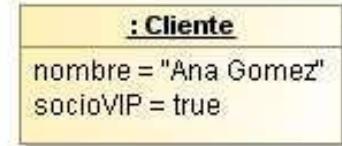
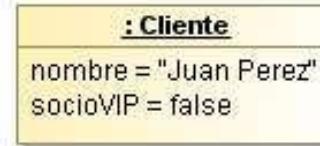
Un modelo de tipos de objetos y los atributos y relaciones permitidas



# Objetos y Clases

6

- Un **objeto** es un concepto o entidad única, inequívocamente identificable (a menudo denominado una *instancia* de una clase)
    - Objeto = Hechos
  - **Definición: Objeto : Clase**
- 
- Una **clase** es una plantilla que describe las propiedades de un concepto (a menudo referido como un *Tipo*)
    - Clase = Reglas



# Clase

7

- Describe la estructura y el comportamiento de objetos que tienen las mismas características y semántica.
- La estructura se describe mediante sus **atributos**.
- El comportamiento mediante sus **operaciones**.



# Atributos

8

- Representan la información almacenada en una clase (propiedad estructural de la clase).
- Representados como atributos integrados o por relación.
- **Especificación**

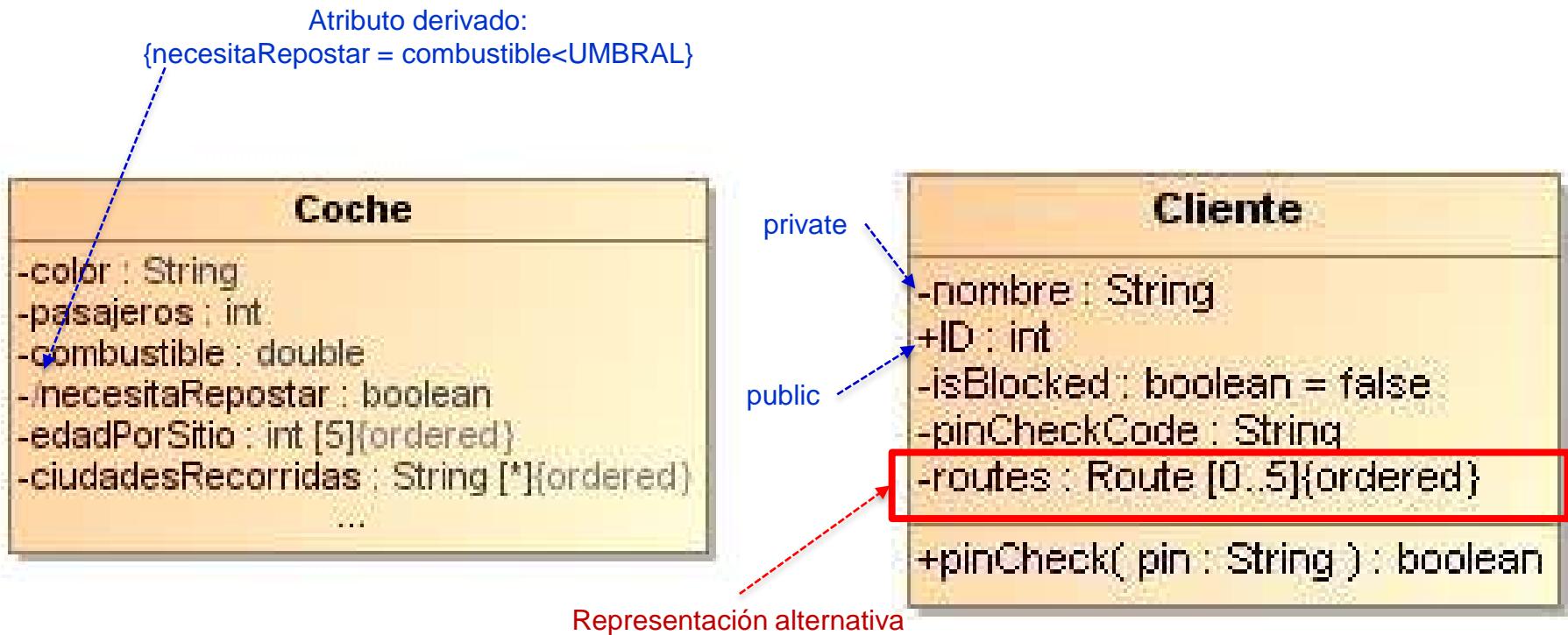
```
visibilidad nombre: tipo multiplicidad = valor_defecto {propiedad}
```

- **Visibilidad**

- Pública (+)
- Privada (-)
- Protected (#)

# Ejemplos

9



# Operación

10

- Una operación define una propiedad de comportamiento de una clase.
- Cada operación tiene un objeto destino implícito.
- Se puede aplicar la misma operación a distintas clases.
- **Especificación**

```
visibilidad nombre(lista_parámetros): tipo_devuelto {propiedad}
```

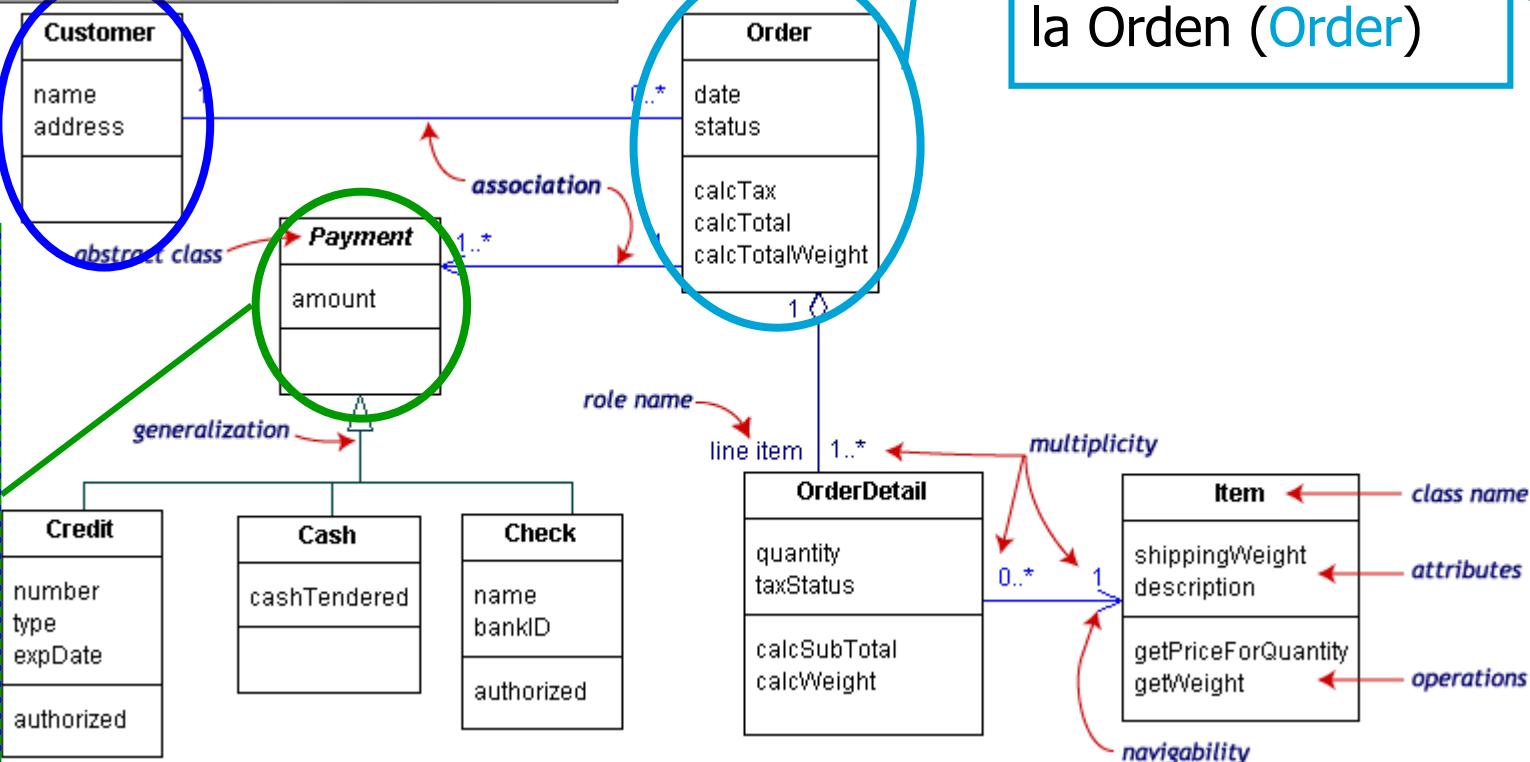
- **Ejemplos**
  - generarFactura
  - +concatenar(str1:String, str2:String): String

# Un ejemplo

11

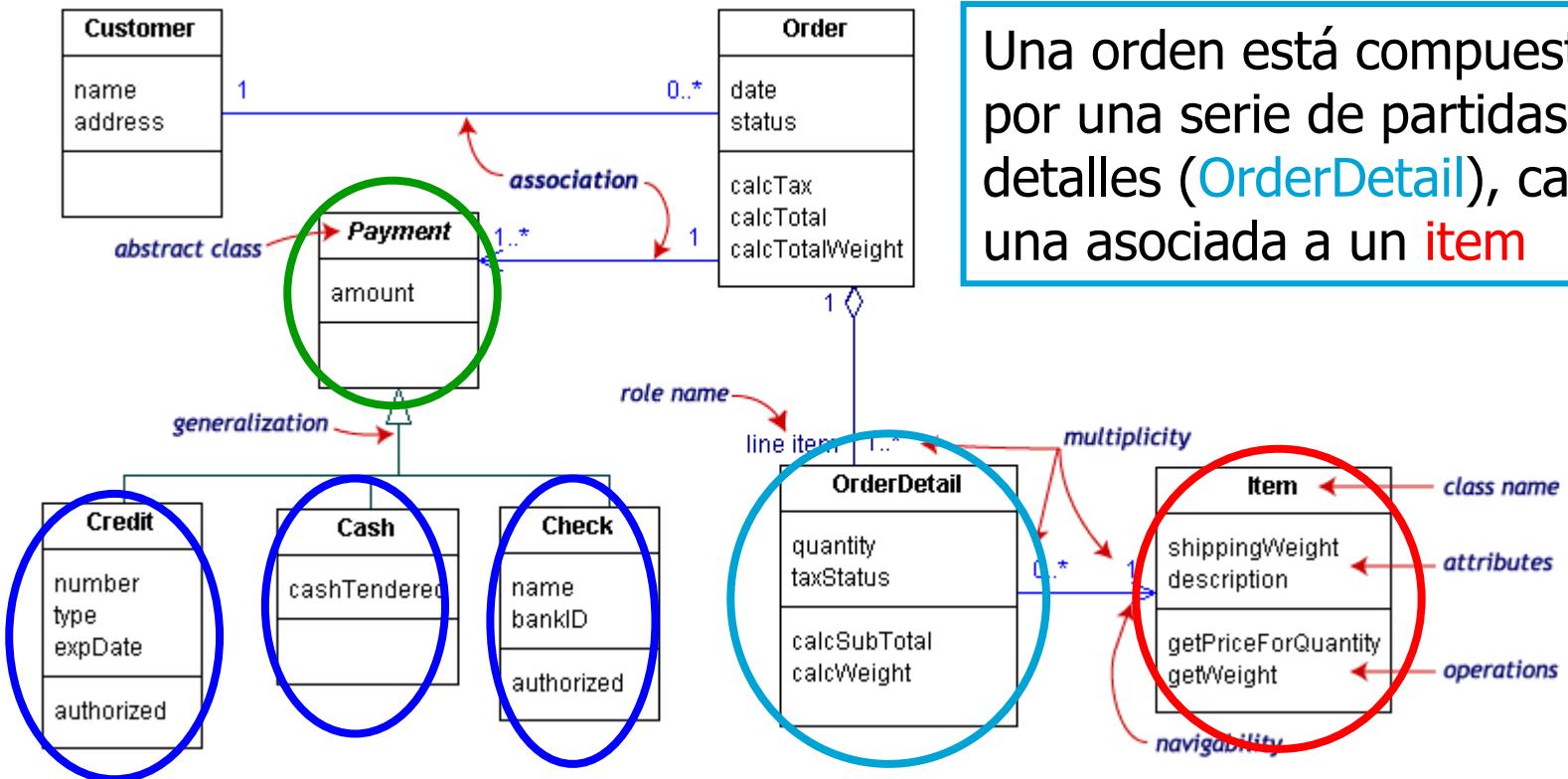
El diagrama de clase siguiente modela una orden de un cliente en un catálogo de venta al por menor.

Asociada a ella está el cliente (Customer) que realiza una compra y un pago (Payment)



# Un ejemplo

12



Una orden está compuesta por una serie de partidas o detalles (**OrderDetail**), cada una asociada a un **item**

El pago (**Payment**) puede ser de tres tipos: **Credit**, **Cash**, **Check**

# Diagrama de clases Conceptual y de Implementación

13

- Un diagrama de clases puede ilustrar varios niveles de detalles y diferentes objetivos, por ejemplo...

- **Diagrama de Clases Conceptual**

- Desarrollado por los analistas de sistema (negocio) para modelar los recursos del sistema (negocio)
- Contiene tipos de datos y operaciones del negocio
- Contiene solo clases del “negocio” – no clases de la implementación.

- **Diagrama de Clases de Implementación**

- Desarrollado por los diseñadores para modelar los requisitos del código.
- Contiene detalles de programación tales como tipos de datos del lenguaje o de la base datos y modificaciones de acceso.
- Se puede usar para generar el código.

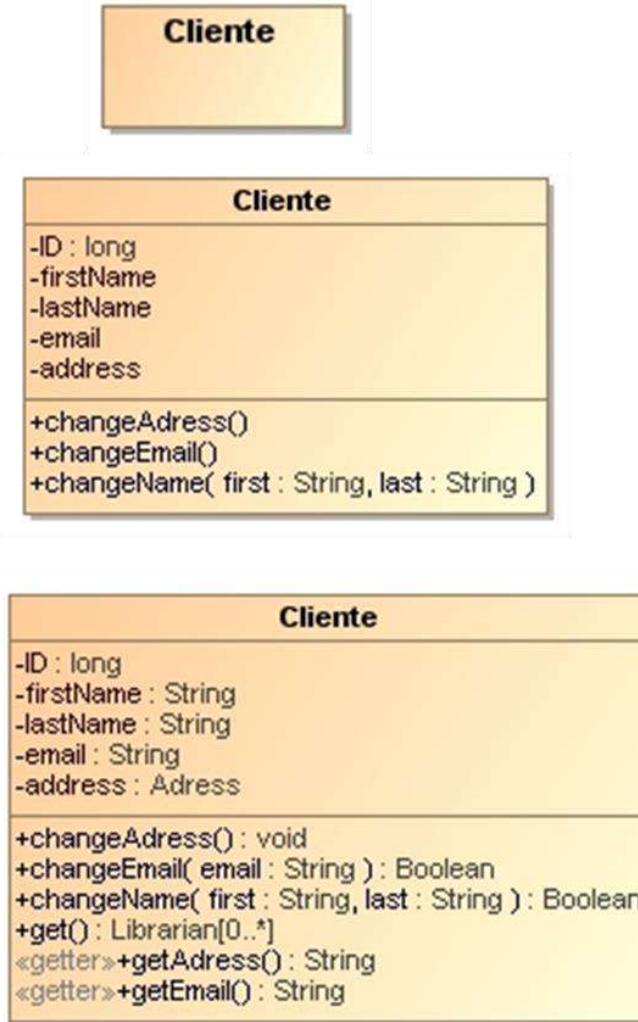
Modela el problema

Modela la solución

# Abstracción en UML

14

1. Conceptos de análisis inicial.
  2. Detalles del análisis totalmente especificados.
  3. Detalles del nivel de implementación.
- **Idea:** ocultar detalles para no complicar la visión de conjunto del diagrama (dejar sólo lo que necesitamos)



# Modelado OO: Actividades

15

1. Identificar los objetos y seleccionar las clases candidatas.
2. Obtener los atributos de las clases necesarios para satisfacer requisitos de información.
3. Buscar las relaciones entre clases.
4. Asociar los atributos con clases o relaciones entre clases.

# Buscando Clases

16

- **Clases del Modelo del Dominio**

- Información del negocio que se debe almacenar o analizar.
- **Sustantivos en la descripción del problema**, políticas y procedimientos del negocio, material de formación y productos en funcionamiento.
- Roles desempeñados por los actores.

- **Clases de Implementación**

- Servicios del negocio.
- Elementos del modelo de datos.
- Vistas gráficas.
- Controladores (Manejador eventos, flujo, y lógica de control).
- Bibliotecas de clases, componentes.

# Identificación de clases candidatas: Técnica de frases nominales

17

- Técnica para identificar componentes del modelo en las descripciones textuales del dominio (p.ej. casos de uso).

Partes del habla	Componente del modelo	Ejemplos
Sustantivo propio	Objeto	Alicia
Sustantivo común	Clase	Persona, Registro
Verbo de acción	Operación	Crea, Envía, Selecciona
Verbo de ser	Generalización	Es un tipo de, es alguno de
Verbo de tener	Agregación	Tiene, consiste en, incluye
Verbo modal	Restricciones	Debe ser
Adjetivo	Atributo	Descripción de incidente, color de coche

# Diagrama de Clases – Relaciones

18

# Asociaciones

19

- **Asociación:** Relación estructural entre dos clases.
- Por defecto bidireccionales, aunque pueden ser unidireccionales.
- Las asociaciones están caracterizadas por:
  - Nombre de la asociación.
  - Roles.
  - Multiplicidad.
  - Navegabilidad.

# Asociaciones

20

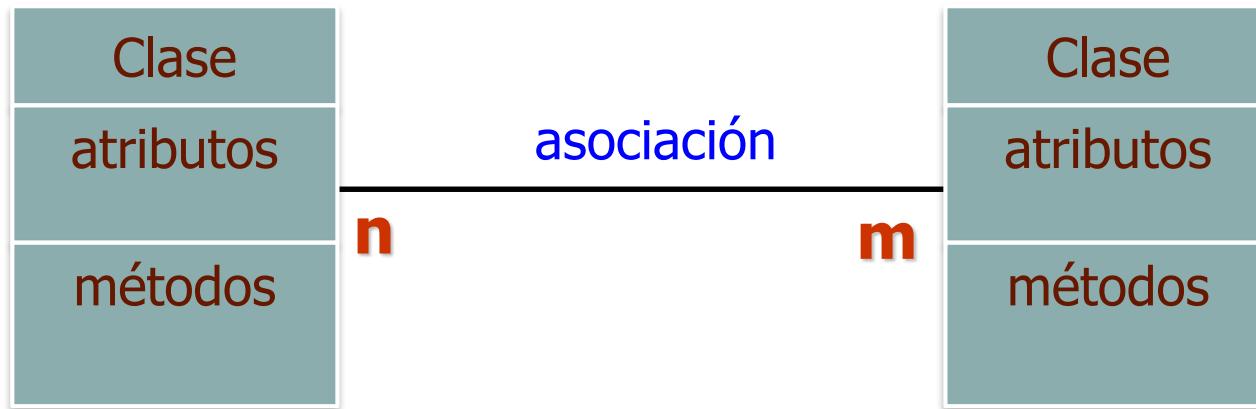
- Especifica que objetos de un tipo tienen una referencia a objetos de otro tipo.



- Permite a los objetos de una clase contactar los objetos de otra clase para acceder a sus datos y su comportamiento

# Multiplicidad

21



## Multiplicidad

**Multiplicidad:** indica cuántos objetos de un extremo de la asociación pueden conectarse con un objeto del otro extremo.

0..1	0 ó 1
*	Muchos
n	Exactamente n
m,n	m o n
m-n	Entre m y n
n+	Más de n
1	Exactamente 1

# Ejemplos de multiplicidad

22

Multiplicidad	Significado
0..1	El valor es opcional
1	Exactamente uno (por defecto)
2..4	Al menos 2 y como máximo 4
0..*	Número arbitrario de valores
*	Otra forma del anterior

# Navegabilidad

23

- Aparece en asociaciones unidireccionales.
- Una flecha muestra la dirección en la cual la asociación puede ser atravesada (o preguntada).
- La flecha nos permite conocer quien “**posee**” la implementación de la asociación.
- Las asociaciones sin flecha son bidireccionales.



# Navegabilidad Asociaciones

24

- Especifica qué objetos de un tipo tienen una **referencia** a objetos de otro tipo
- Permite a los objetos de una clase contactar con los objetos de otra clase para acceder a sus datos y su comportamiento.
- **Ejemplo anterior:**
  - Podríamos preguntarle a la persona sobre la empresa donde trabaja.
  - Pero la empresa no puede decírnos qué personas trabajan en ella.

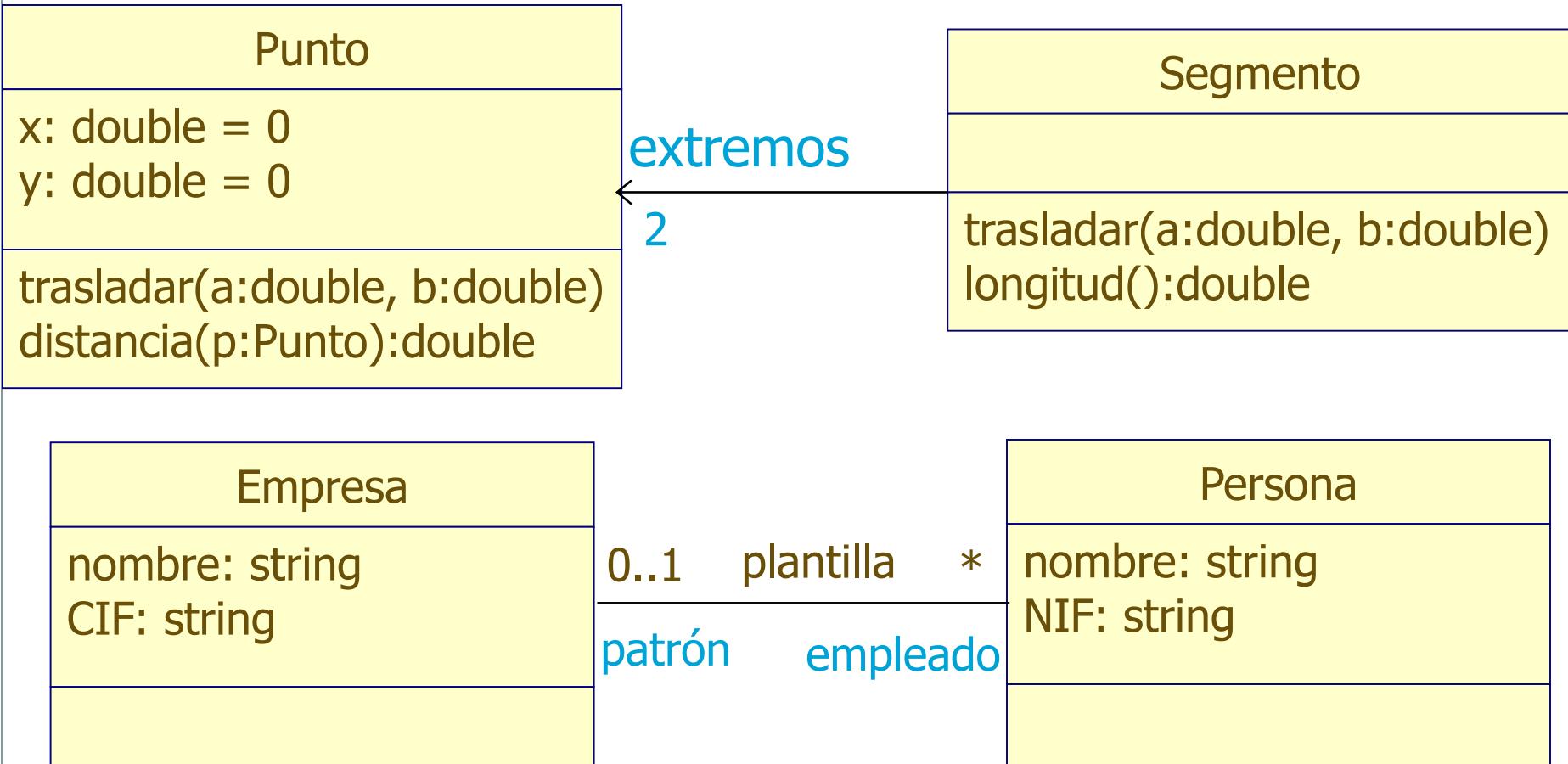
# Roles

25

- El **rol** indica el papel o la cara que la clase de un extremo de la asociación presenta a la clase del otro extremo.
- Una asociación binaria tiene dos roles.
- Se pueden nombrar explícitamente:
  - para recorrer asociaciones, y
  - para especificar direcciónalidad.
- Los nombres de los roles:
  - deben ser únicos en asociaciones que parten de una misma clase
  - son necesarios para asociaciones entre objetos de la misma clase

# Roles

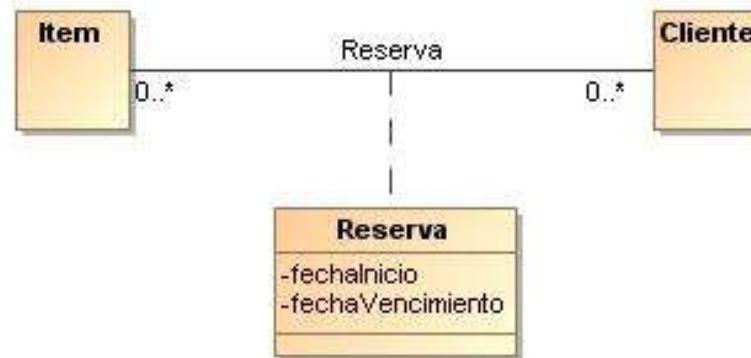
26



# Clases Asociación

(27)

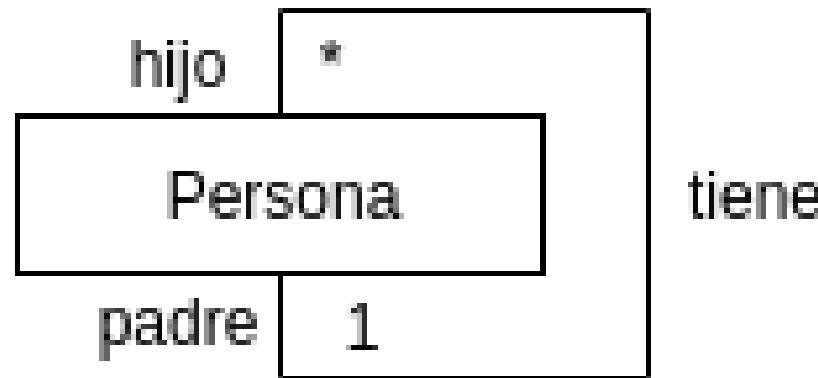
- **Clase asociación:** es una clase que se conecta a una asociación para definir propiedades de la asociación.
  - La clase asociación está ligada a la conexión formada entre las clases.
    - → Cada enlace conlleva una instancia de la clase asociación.
  - Común en asociaciones muchos a muchos y uno a muchos.



# Asociación reflexiva

28

- La ***misma clase*** puede ser principio y final
- Pero eso no implica un único objeto.



# Agregación y Composición

29

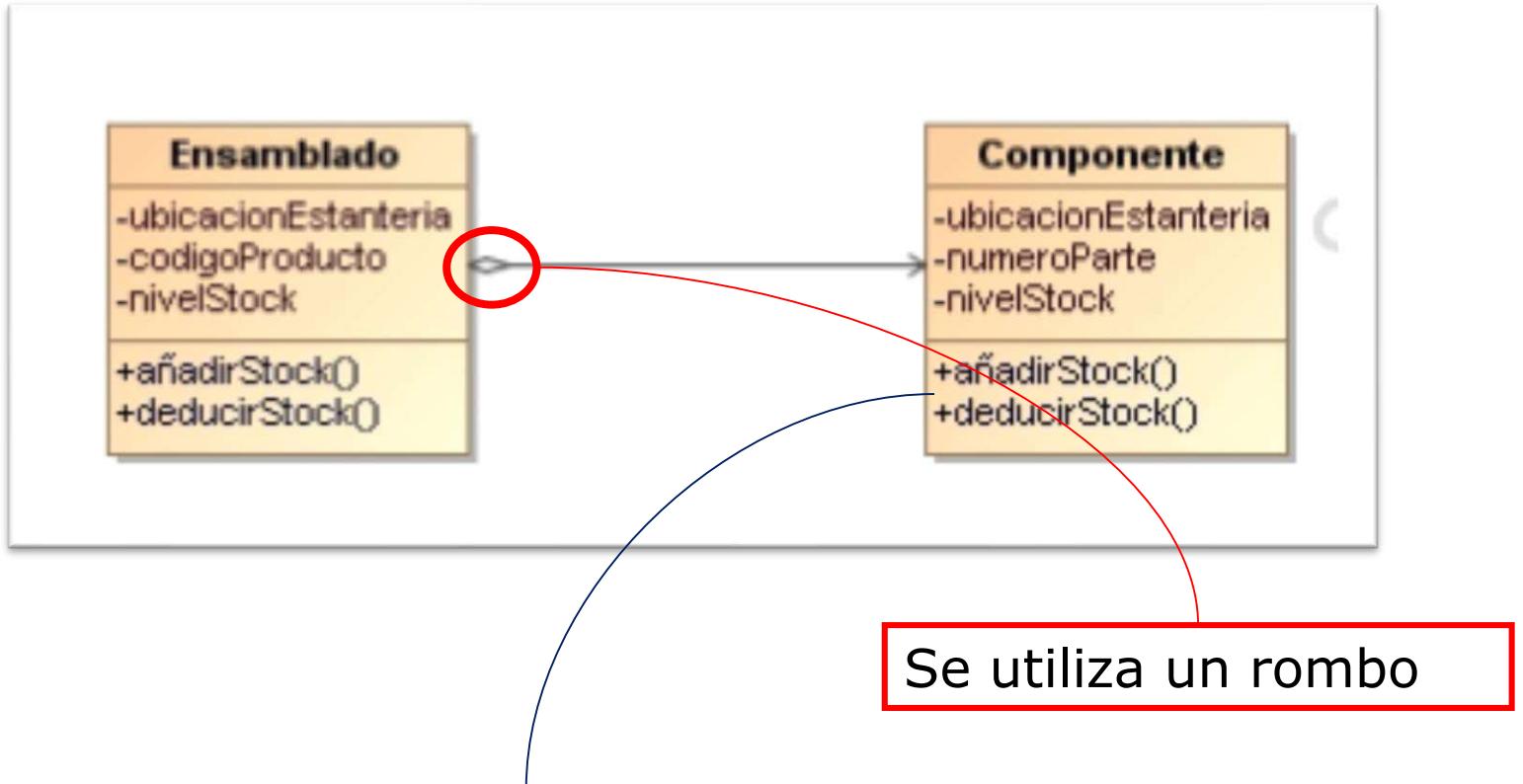
# Agregación

30

- Una **agregación** (relación “*es parte de*”) es una asociación en la que una de las partes representa el todo y la(s) otra(s) la(s) parte(s).
- Indica que una instancia de una clase,
  - además de tener sus propios atributos,
  - puede estar compuesta por (o incluir) instancias de otras clases .
- Es usual la **propagación** de operaciones entre un objeto y sus componentes.

# Agregación

31



- **Ensamblado (todo)** esta compuesto por Componentes (**partes**).
- Un Componente es parte de un Ensamblado.

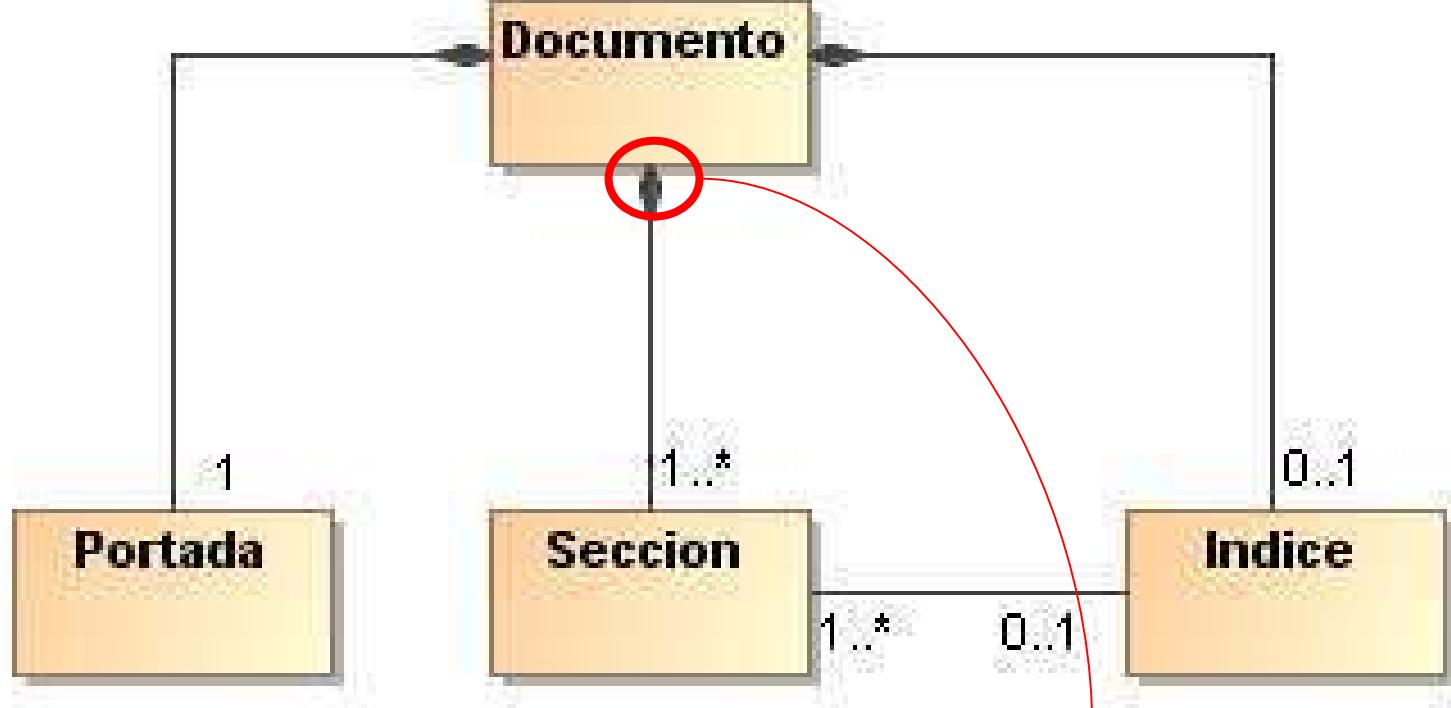
# Composición

32

- También modela una relación “ser-parte-de”.
- Pero además implica que el ciclo de vida de la parte está ligado al del todo.
- **En una composición una “parte” no puede existir sin ser parte de un “todo”**
  - En la agregación esto no es así (un componente podría existir sin estar ensamblado).
  - Un objeto que representa a la “parte” sólo puede pertenecer a un “todo”.

# Composición

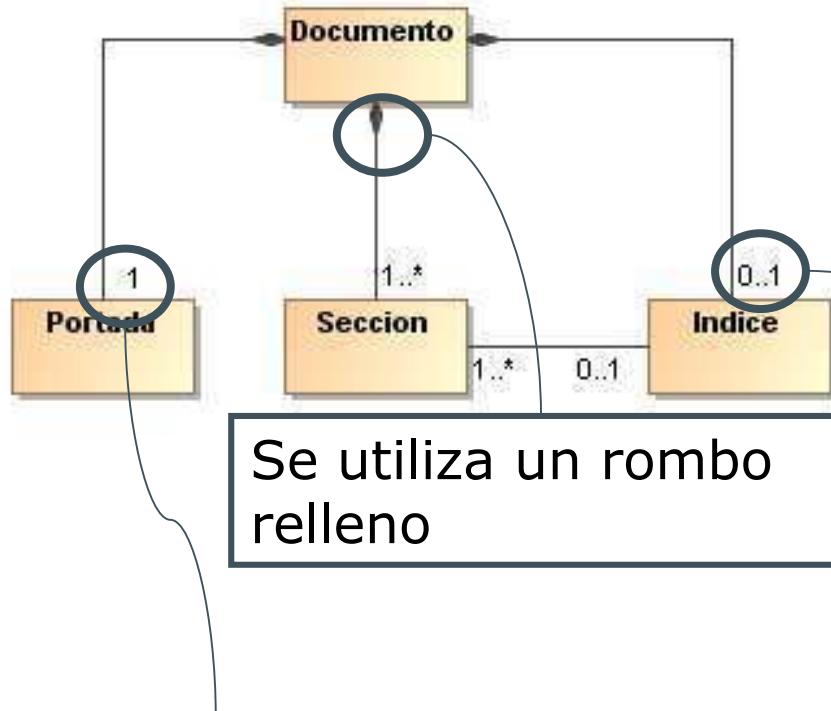
33



Se utiliza un rombo  
relleno

# Composición

34



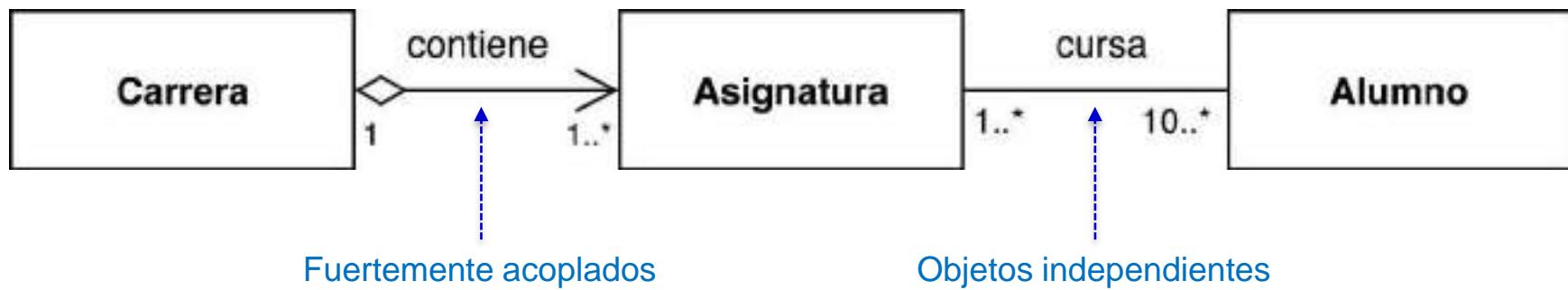
El uno implica que al crear Documento debemos crear la Portada

Si borramos un documento, además se deben borrar su portada, secciones e índice

# Agregación/Composición vs. Asociación

35

- Si los objetos instanciados de las clases están ligados por una relación todo/parte:
  - La relación es de agregación/composición.
- Si los objetos se consideran independientes aún cuando haya enlaces frecuentes entre ellos:
  - La relación es una asociación.



# Generalización

36

# Generalización

37

- Una **generalización** (relación “*es un tipo de*”) es una relación taxonómica desde una clase especializada hasta una clase general.
  - Relación entre un elemento general (superclase, padre o clase base),
  - y uno más específico (subclase, hijo o clase derivada).
- Cada subclase **hereda** todos los atributos y operaciones de la superclase y después los **especializa** de diferente formas.
- Las subclases **extienden** a las superclases con nuevos atributos y operaciones o **redefinen** la implementación de las operaciones heredadas.

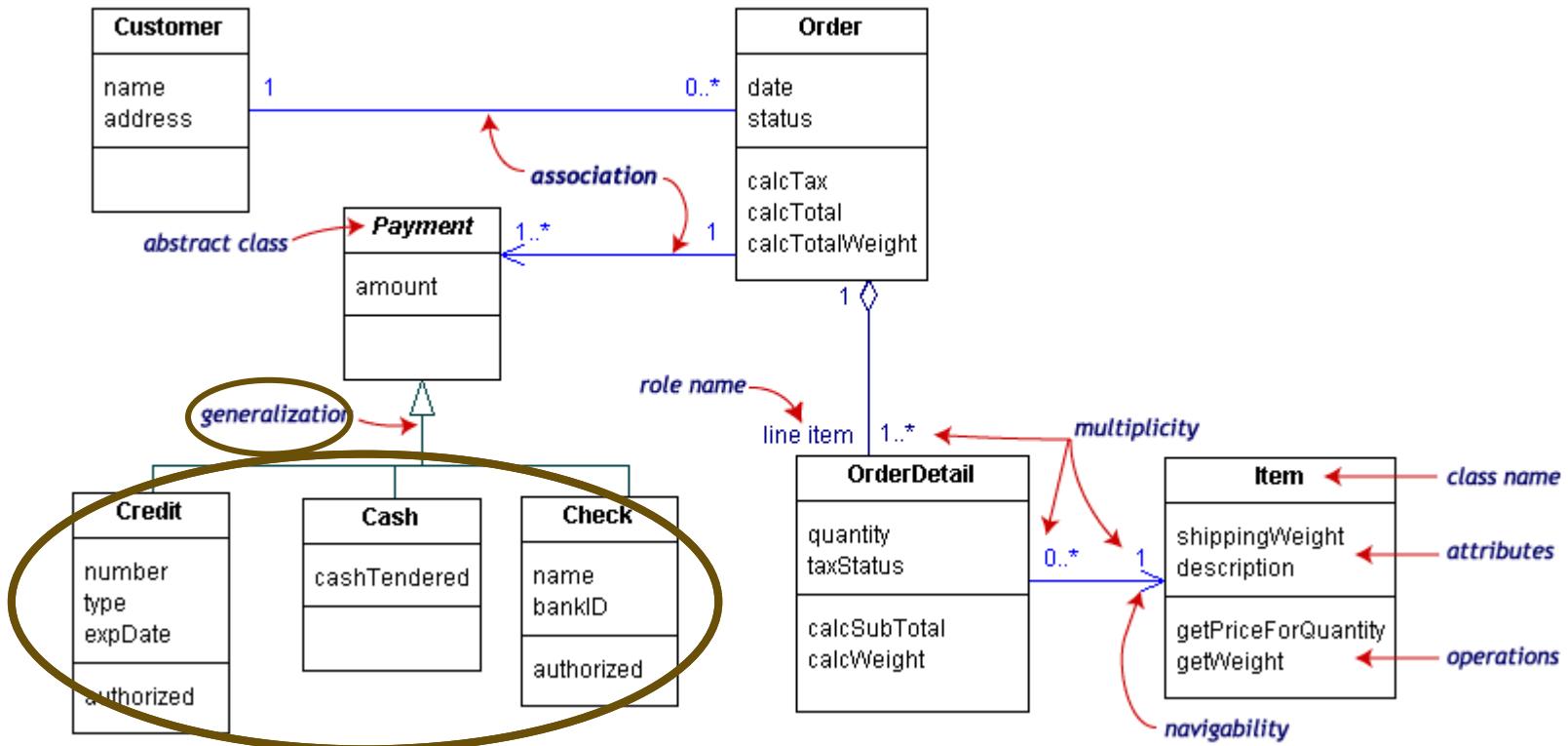
# Clase Abstracta

38

- Una **clase abstracta** es una clase no instanciable.
- Suele contener una o más **operaciones abstractas** (sin implementación) que las clases derivadas definen.
- Se especifica con el estereotipo <<abstract>> y el nombre en cursiva.

# Un ejemplo (Generalización)

39



Pago (payment) es una superclase de  
Crédito (credit),  
Efectivo (cash),  
Cheque (check).

# Clases de Análisis

40

# Tipos de objetos y el cambio

41

- Tener tres tipos de objetos lleva a modelos que son más resistentes al cambio.
  - El interfaz de un sistema cambia más que el control.
  - La forma en la que se controla al sistema cambia más que el dominio de la aplicación.
- Tipos de objetos, su origen es el lenguaje Smalltalk:
  - Modelo, Vista, Controlador (patrón MVC)
  - Objetos Entidad, Interfaz, Control

# Modelo de análisis.

## Las clases de análisis.

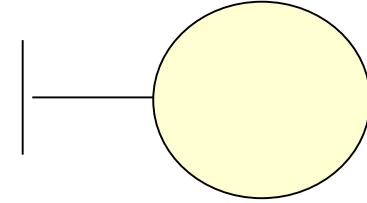
42

- Representan **abstracciones** de lo que serán una o varias clases en el diseño (y posiblemente subsistemas).
- Se centra en los requisitos funcionales.
- No contiene operaciones ni signaturas.
  - Su comportamiento se define mediante descripciones textuales llamadas responsabilidades.
- Define atributos de alto nivel.
- Normalmente pasan a ser clases en el modelo de diseño.

# Clases de análisis.

## Interfaz

43

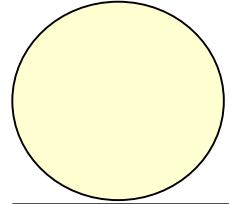


- Se usa para **modelar la interacción entre el sistema y los actores**. La interacción implica:
  - Peticiones de información del actor al sistema.
  - Recepción de información enviada por el sistema al actor.
- Representan la **interfaz del sistema**: ventanas, formularios, paneles, sensores, interfaces de comunicación...
- Deben ser abstractas, describiendo sólo la información y las peticiones que se intercambian entre el sistema y sus actores.

# Clases de análisis.

## Entidad.

44

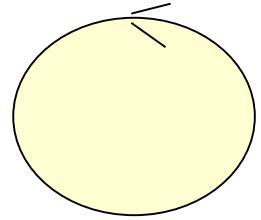


- Modelan la **información persistente**.
- Se derivan de una clase del modelo del negocio o del dominio.
- Pueden ser activas o pasivas y tener un comportamiento complejo correspondiente a la información que representan.

# Clases de análisis.

## Control.

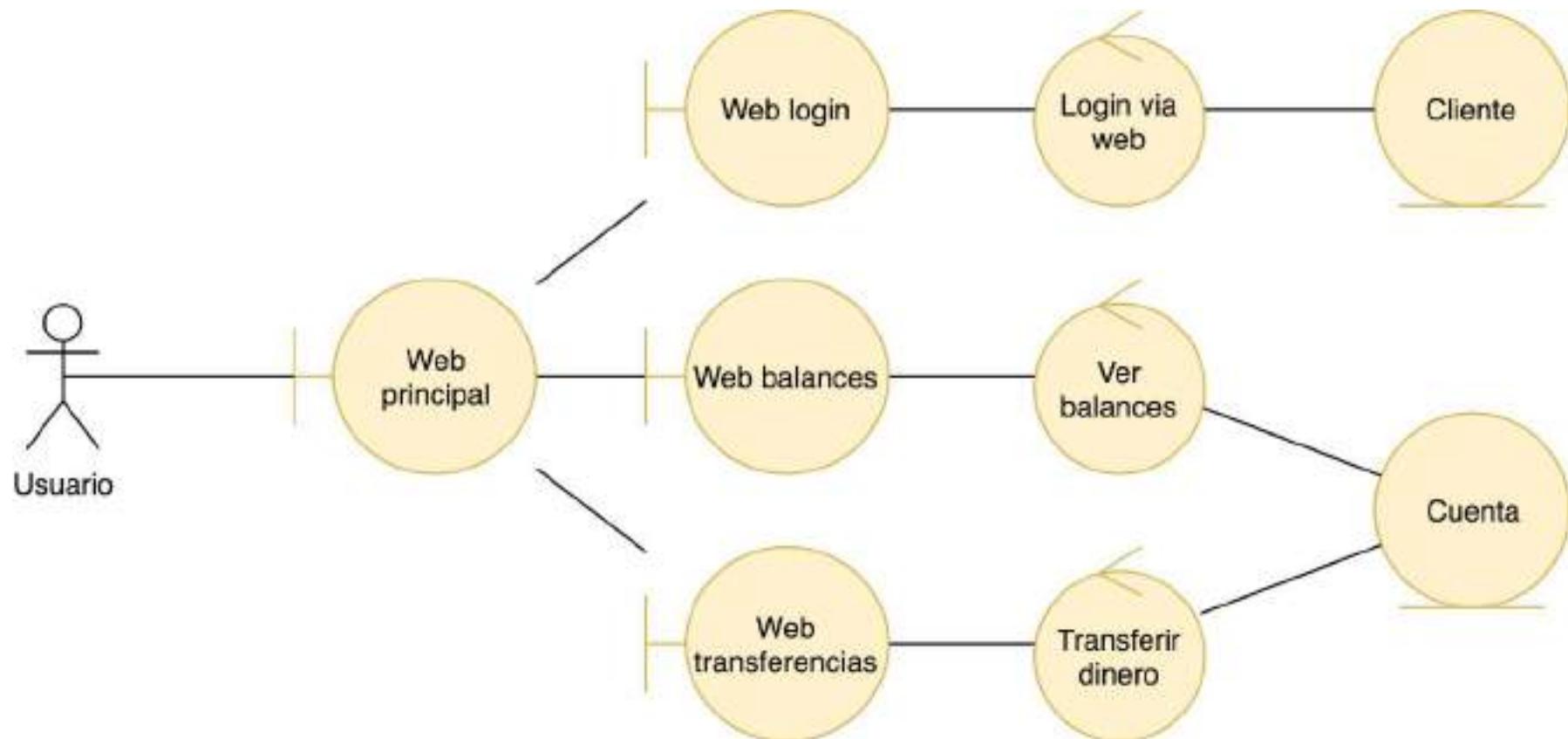
45



- Modelan los aspectos dinámicos internos del sistema.
- Se usan para representar el control de un caso de uso concreto.
- Representan cálculos, lógica de negocio.

# Ejemplo: Sistema de banca online

46



# Ejercicio

47

# Ejercicio: Aplicación de Dibujo

48

## Enunciado

- Se desea una aplicación de dibujo que puede manejar la agrupación de elementos de dibujo para facilitar la realización de gráficos.
- Cada documento se crea usando una o más páginas. Los objetos de dibujo se representan en estas páginas.
- Una vez colocado en una página los objetos de dibujo se pueden mover, copiar, cambiar de tamaño, rotar y borrar. Las páginas se pueden reordenar, insertar y eliminar.
- Todos los objetos en la página se mueven con ella. Cada página puede contener tantos (o tan pocos) objetos de dibujo como desee el cliente. Incluso algunas páginas pueden estar en blanco. Los clientes pueden dibujar una variedad de objetos tales como círculos, elipses, cuadrados, rectángulos, líneas o incluso texto en cualquier posición de la página.
- Los grupos de objetos se pueden mover, copiar, cambiar de tamaño, rotar y eliminar como una sola unidad. Cada grupo debe tener como mínimo dos objetos de dibujo. Sin embargo, un objeto de dibujo sólo puede ser un miembro directo de un grupo. Un grupo también puede contener otros grupos. Sin embargo, un grupo sólo puede ser un miembro directo de un grupo. El cliente también puede borrar el grupo y trabajar individualmente con los objetos de dibujo que habían estado en el grupo.

# Ejercicio: Identificamos clases candidatas

49

## Usando la técnica de frases nominales:

- Se desea una aplicación de dibujo que puede manejar la agrupación de elementos de dibujo para facilitar la realización de gráficos.
- Cada documento se crea usando una o más páginas. Los objetos de dibujo se representan en estas páginas.
- Una vez colocado en una página los objetos de dibujo se pueden mover, copiar, cambiar de tamaño, rotar y borrar. Las páginas se pueden reordenar, insertar y eliminar.
- Todos los objetos en la página se mueven con ella. Cada página puede contener tantos (o tan pocos) objetos de dibujo como desee el cliente. Incluso algunas páginas pueden estar en blanco. Los clientes pueden dibujar una variedad de objetos tales como círculos, elipses, cuadrados, rectángulos, líneas o incluso texto en cualquier posición de la página.
- Los grupos de objetos se pueden mover, copiar, cambiar de tamaño, rotar y eliminar como una sola unidad. Cada grupo debe tener como mínimo dos objetos de dibujo. Sin embargo, un objeto de dibujo sólo puede ser un miembro directo de un grupo. Un grupo también puede contener otros grupos. Sin embargo, un grupo sólo puede ser un miembro directo de un grupo. El cliente también puede borrar el grupo y trabajar individualmente con los objetos de dibujo que habían estado en el grupo.

# Ejercicio: Descartar clases candidatas

50

- **Clases descartadas**

Motivo para su descarte	Clases candidatas
Redundantes	Agrupación de elementos de dibujo (Grupo de objetos), objetos (objeto de dibujo)
Irrelevantes	Aplicación de dibujo, cliente
Imprecisas	Gráficos, Posición (de página), unidad

- **Clases finales:**

- Documento
- Página
- Grupo de Objetos
- Objeto de dibujo
- Elipse, círculo, rectángulo, cuadrado, línea, texto

# Ejercicio: Atributos y operaciones

51

Clase	Atributos	Operaciones
Documento		
Página		reordenar, insertar, eliminar
Grupo de objetos		mover, copiar, cambiarTamaño, rotar, eliminar, desagrupar
Objeto de dibujo		mover, copiar, cambiarTamaño, rotar, borrar, dibujar
Elipse, círculo, ..., texto		mover, copiar, cambiarTamaño, rotar, borrar, dibujar

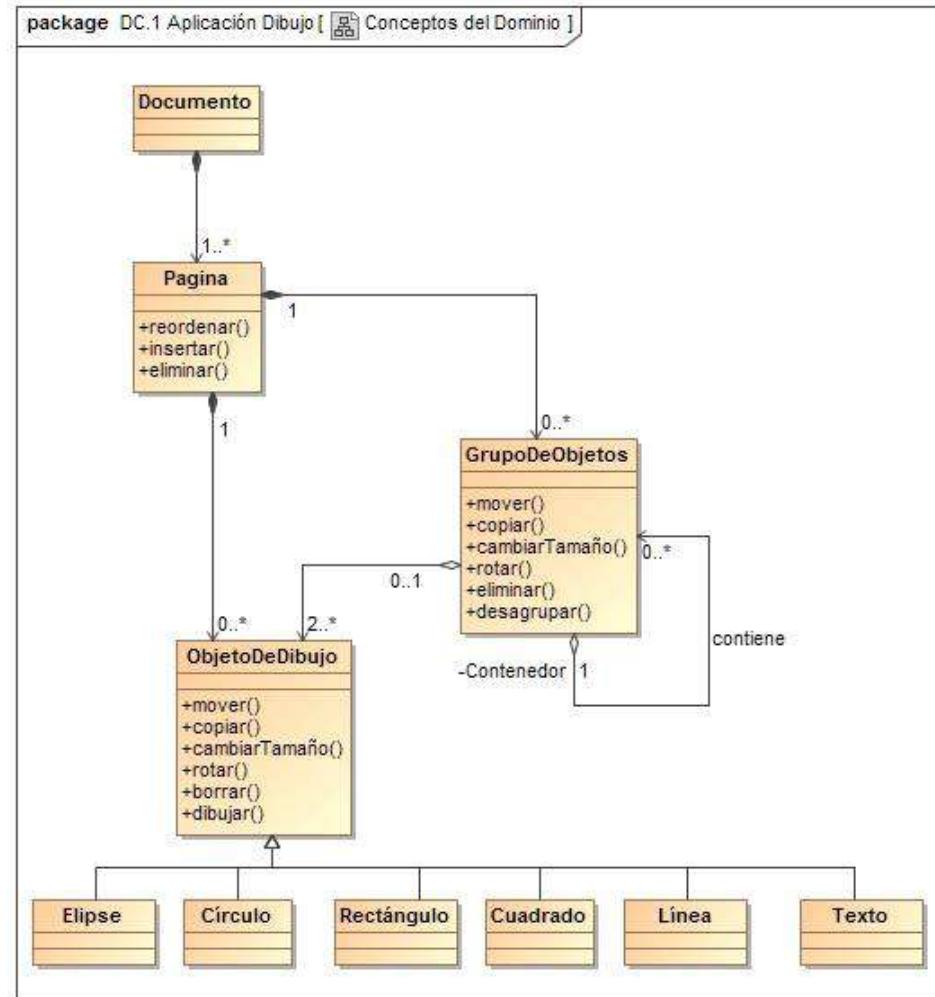
# Ejercicio: Determinar relaciones

52

- Documento se crea usando *una o más páginas*.
- Cada página puede contener *tantos objetos de dibujo* como se desee.
- Cada grupo debe tener como *mínimo dos objetos de dibujo*.
- Un objeto de dibujo sólo puede ser miembro directo de un grupo.
- Un grupo también *puede contener otros grupos*.
- Una variedad de objetos tales como círculos, elipses, cuadrados, rectángulos, líneas o incluso texto.

# Ejercicio: Diagrama de clases

53



# Modelado con UML



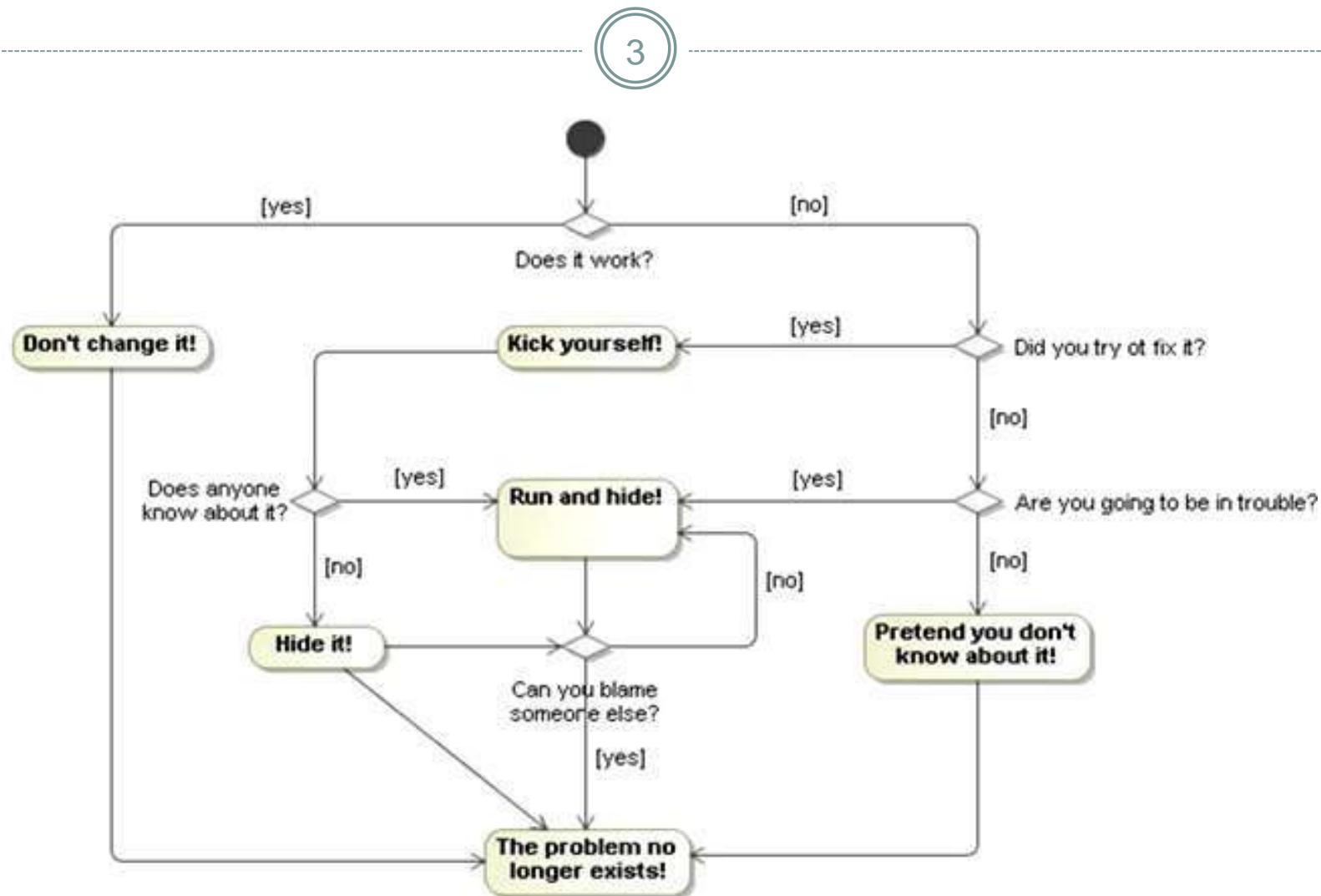
## 5.3. DIAGRAMAS DE ACTIVIDAD

# Diagrama de actividad

2

- Un **diagrama de actividad** muestra el flujo de control general de un sistema, modelando su lógica.
- Modela una secuencia de acciones y condiciones tomadas dentro de un proceso.
- Se puede utilizar uno para cada operación o **caso de uso**.
- Diferencia principal con **diagramas de flujo**: **soportan paralelismo**.

# Ejemplo - ¿Cómo abordar un problema?



# Diagrama de Actividad

4

## • **Cuando usarlos**

- Para modelar negocios o flujos de trabajo (workflows) inter/intra sistemas.
- Para modelar el **flujo de eventos en un caso de uso**.
- Para modelar la implementación de operaciones complejas de una clase (describir un algoritmo complejo).

## • **Cuando NO usarlos**

- Para ver cómo se comunican los objetos
  - Usar **Diagramas de secuencia**.
- Para ver cómo se comporta un objeto
  - Usar **Diagramas de máquinas de estado**.

# Elementos

5

- Actividades
- Acciones
  - Acciones de eventos de aceptación
- Flujos de control
- Inicio y final de actividad
- Nodos objeto
  - Nodos de parámetro de actividad
- Flujos de objetos
  - Pines de entrada y salida
- Precondición y postcondición
- Nodos de control
  - Nodos de decisión/mezcla
  - Nodos fork/join
- Excepciones
- Nodos final de flujo
- Calles (Swimlanes)

# Actividades y Acciones

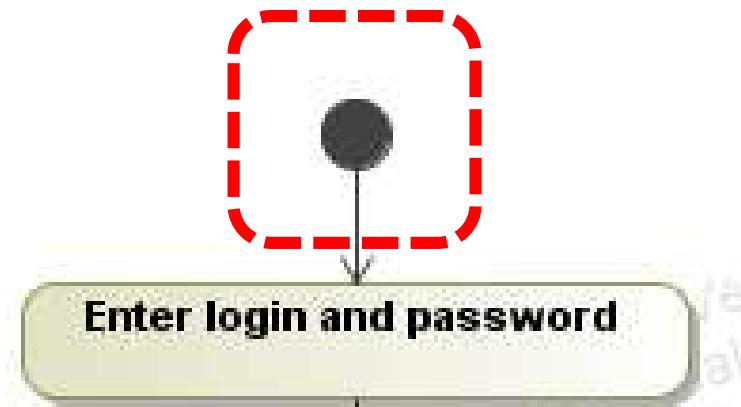
6

- Una **actividad** es un trabajo manual o computarizado que se realiza para producir un resultado.
  - Es compleja y puede desglosarse en actividades o acciones.
  - Puede ser interrumpida por un evento.
  - Se representa mediante un rectángulo redondeado que engloba las acciones y flujos de control de la que se compone.
- Una **acción** modela un paso en la realización de la actividad:
  - Es simple y no puede ser desglosada.
  - Es atómica, por tanto no puede ser interrumpida.
  - Se representa mediante un rectángulo redondeado.
- Las **transiciones** entre acciones se representan mediante una flecha.
  - Se producen en cuanto finaliza la acción origen.

# Acciones y transiciones

7

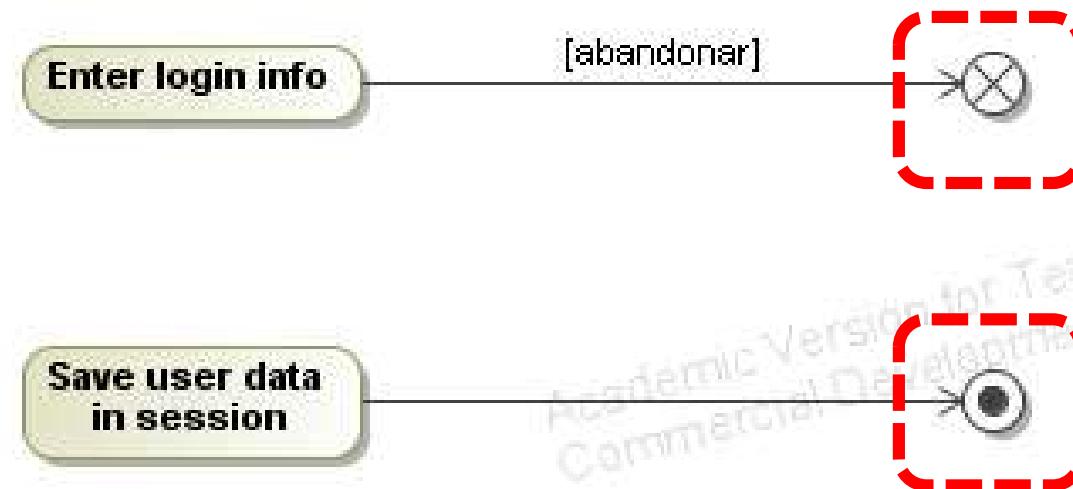
- El Nodo Inicial muestra el comienzo de la colaboración.
- Solo puede tener transiciones de salida.
- Solo habrá un único nodo inicial.



# Nodo final y Nodo final de flujo

8

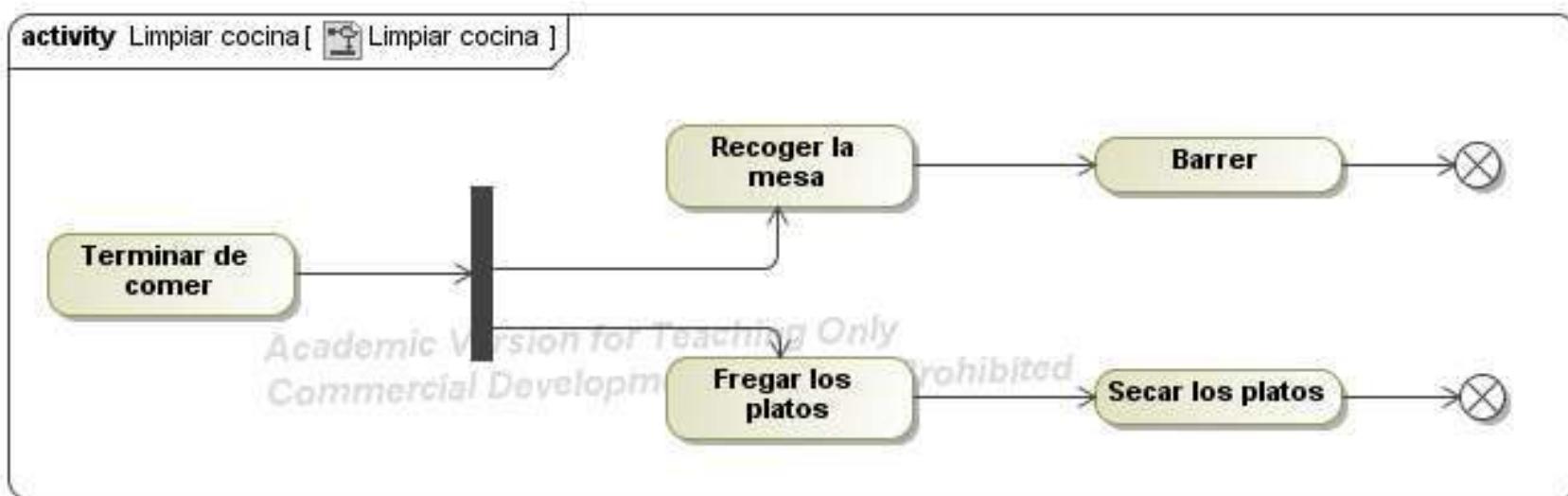
- **Nodo final de flujo** es un final en un flujo o secuencia lógica dentro de un diagrama de actividad.
- **Nodo final** indica la finalización de todo el diagrama de actividad.



# Nodo de final de flujo

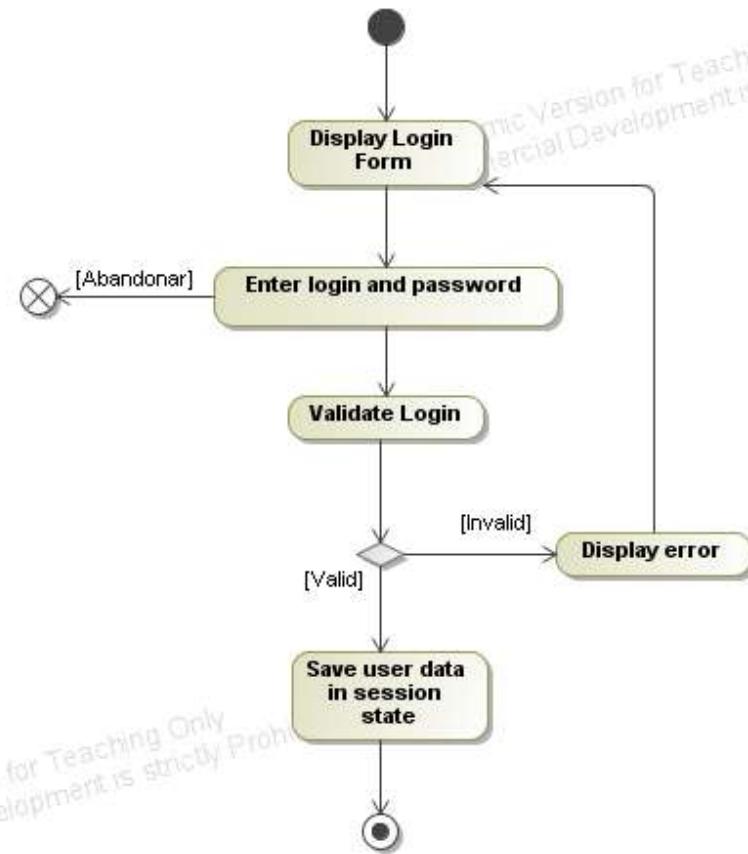
9

- Diferencia con el nodo de final de actividad
  - El nodo de final de actividad termina la ejecución de la actividad cuando se alcanza
    - ★ Interrumpe todos los flujos
  - Los nodos de final de flujo permiten que los demás flujos continúen su actividad



# Ejemplo

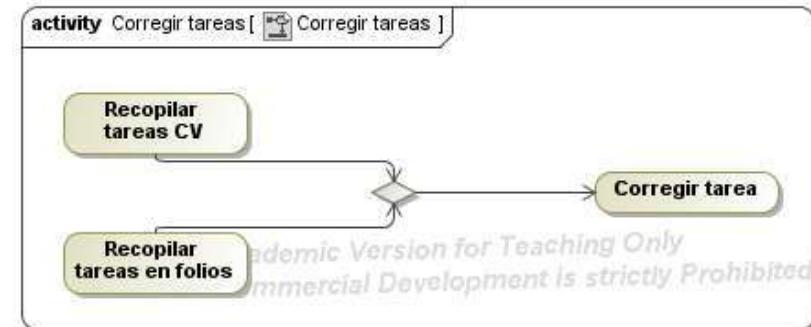
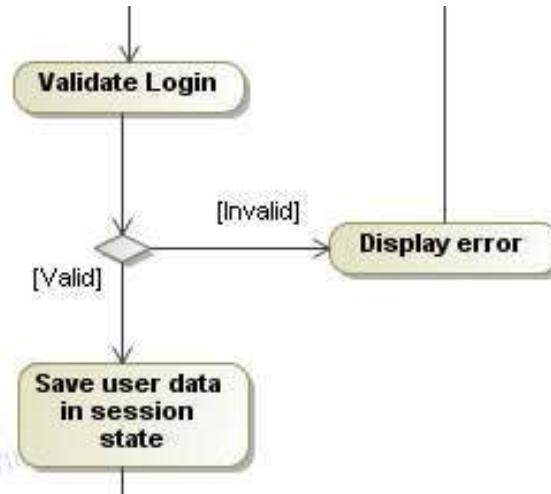
10



# Nodo de decisión

11

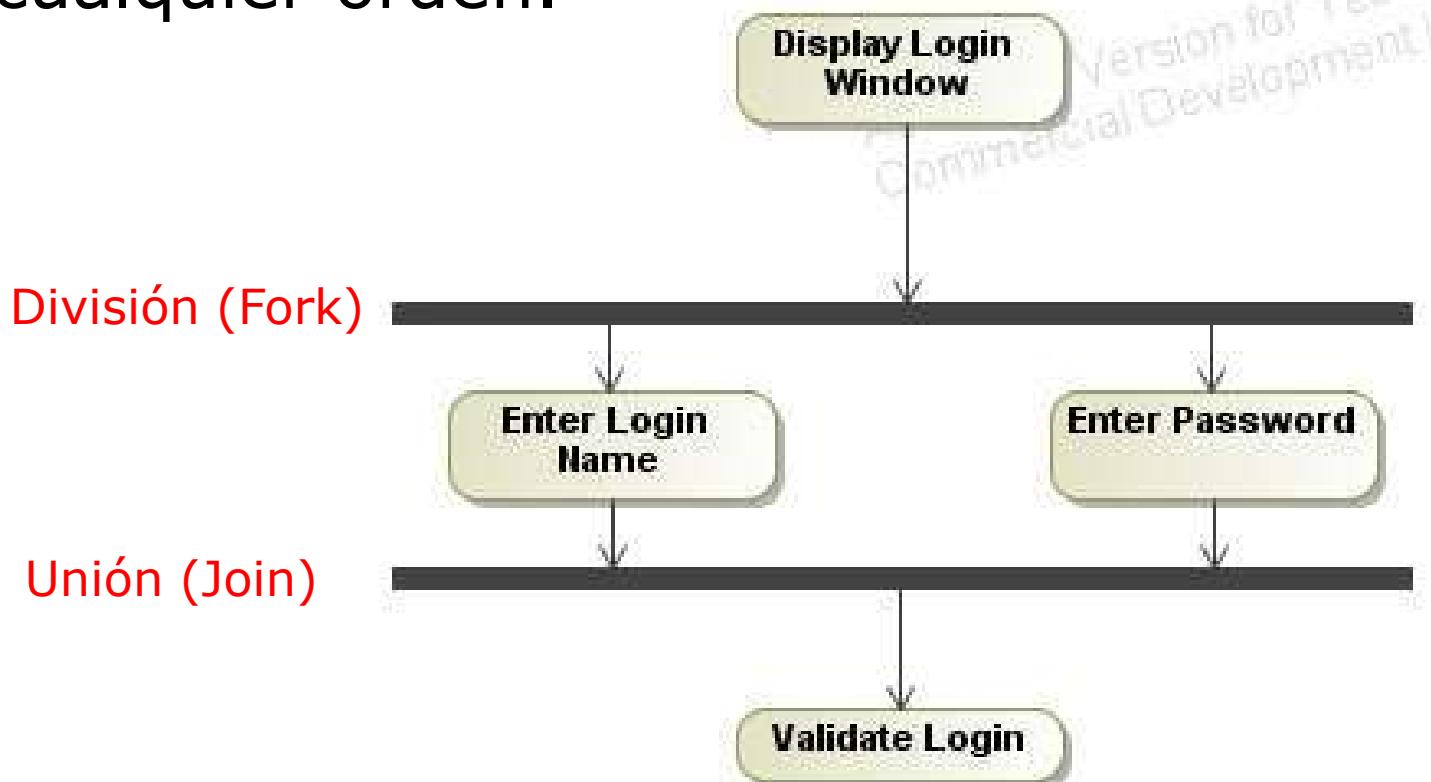
- Un **nodo de decisión** utiliza un rombo para mostrar donde divergen dos (o más) transiciones alternativas.
- Los rombos también se usan para unir en un solo flujo varios flujos alternativos.
- Las **guardas** son condiciones entre corchetes cuadrados que especifican cuando el flujo sigue la transición con guarda.



# Sincronización

12

- Nodos de unión/división modelan actividades que pueden ocurrir simultáneamente o en cualquier orden.



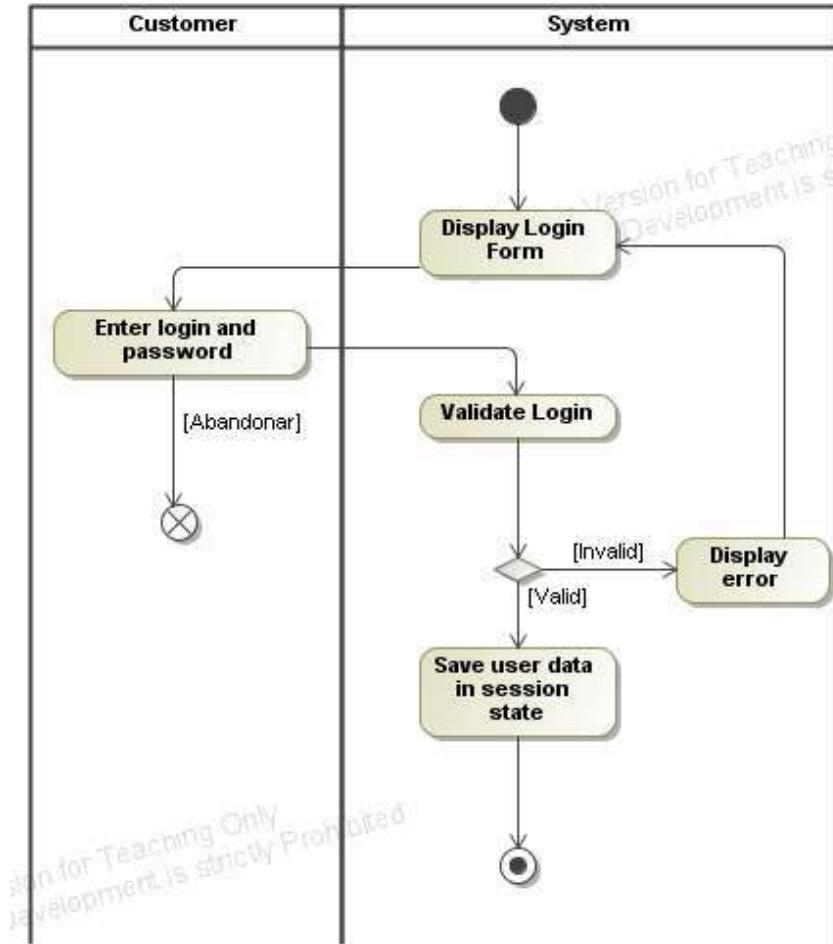
División (Fork)

Unión (Join)

# Calles

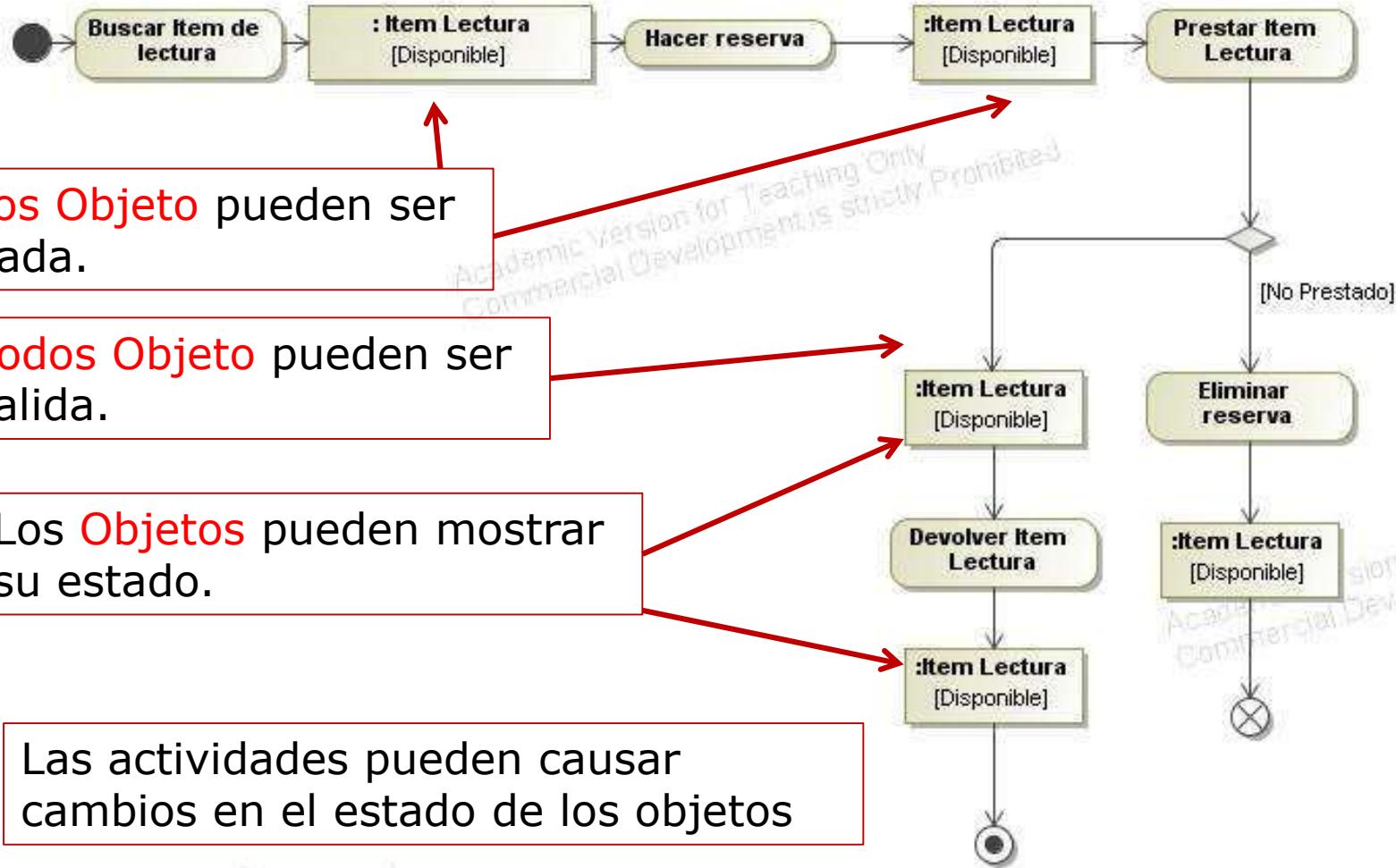
13

- Calles permiten **agrupar las actividades** según un criterio.
- Asignan la responsabilidad de un comportamiento a usuarios, sistemas, objetos o capas dentro de una arquitectura.



# Nodos Objeto

14

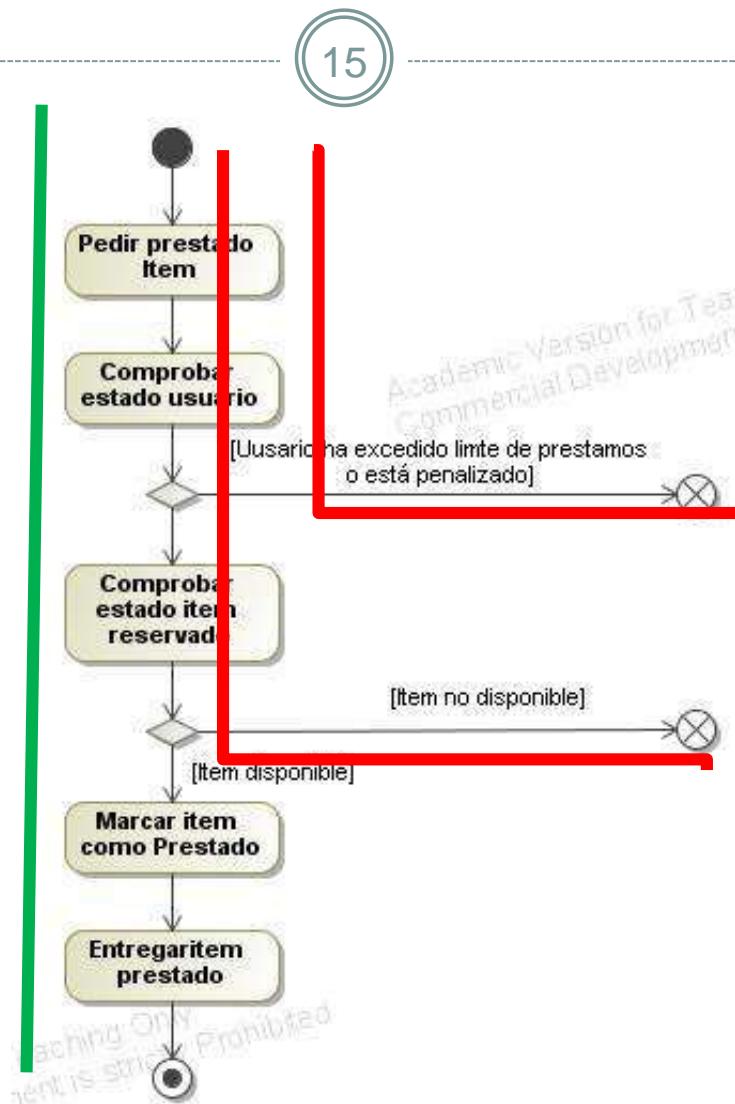


# Usar un diagrama de actividad

15

**Un escenario principal o exitoso**

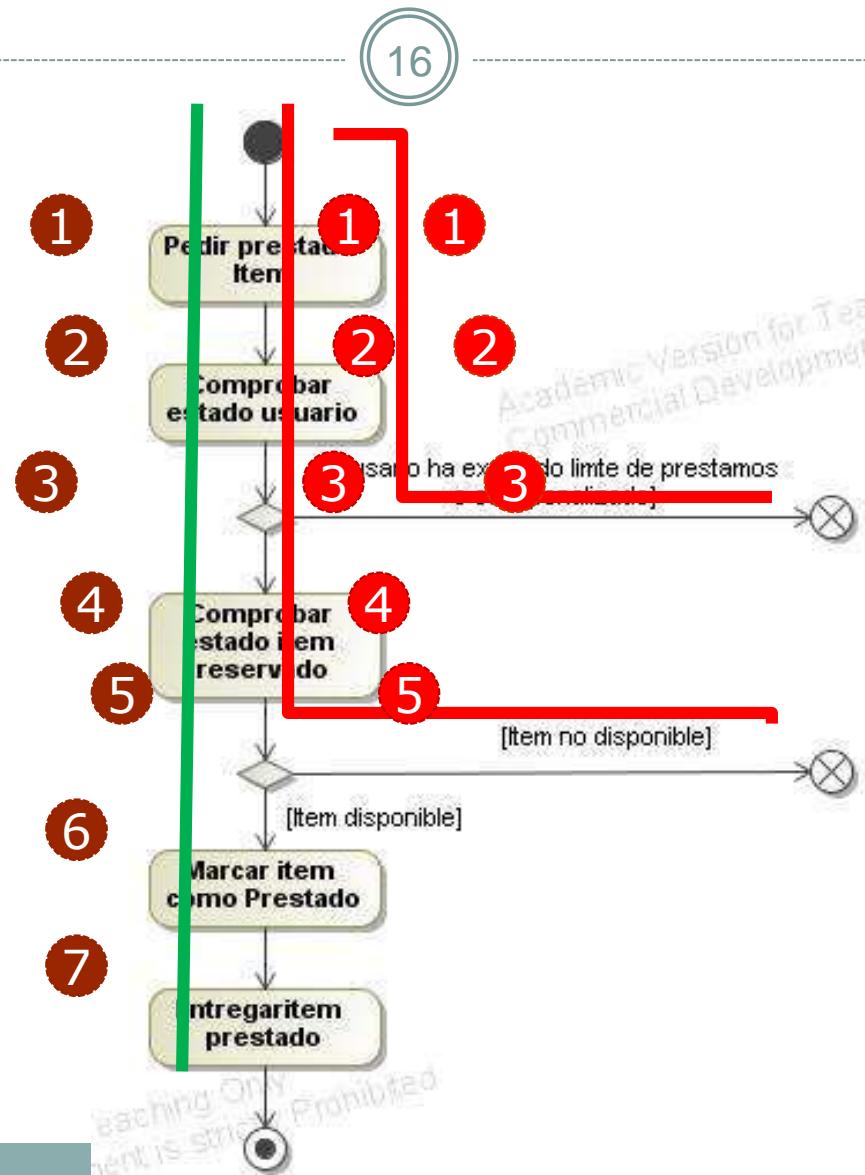
**Dos escenarios alternativos**



# Crear casos de prueba

16

**Crear datos de prueba que fuercen el caso de uso en cada paso de cada escenario**



**Asociar el plan de pruebas con el caso de uso**

# Ejemplo 1

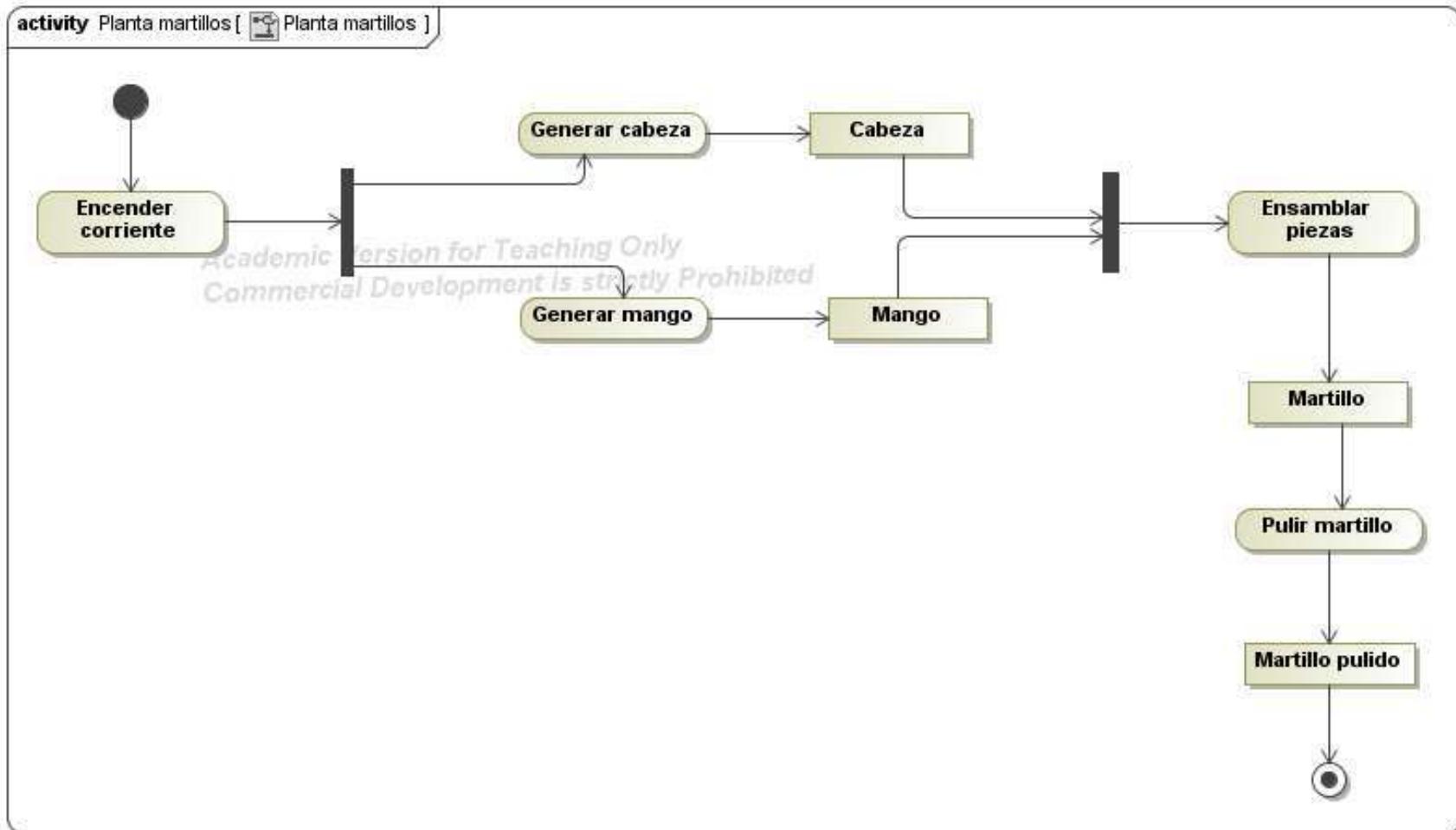
17

## **Cadena de montaje de martillos**

- En una planta de montaje se generan simultáneamente cabezas y mangos de martillos tras encender la corriente de la fábrica
  - Cuando se tiene una cabeza y un mango, se ensamblan ambas partes para generar un martillo
  - El martillo es luego pulido y termina el flujo

# Ejemplo 1: Diagrama de actividad

18



# Ejemplo 2

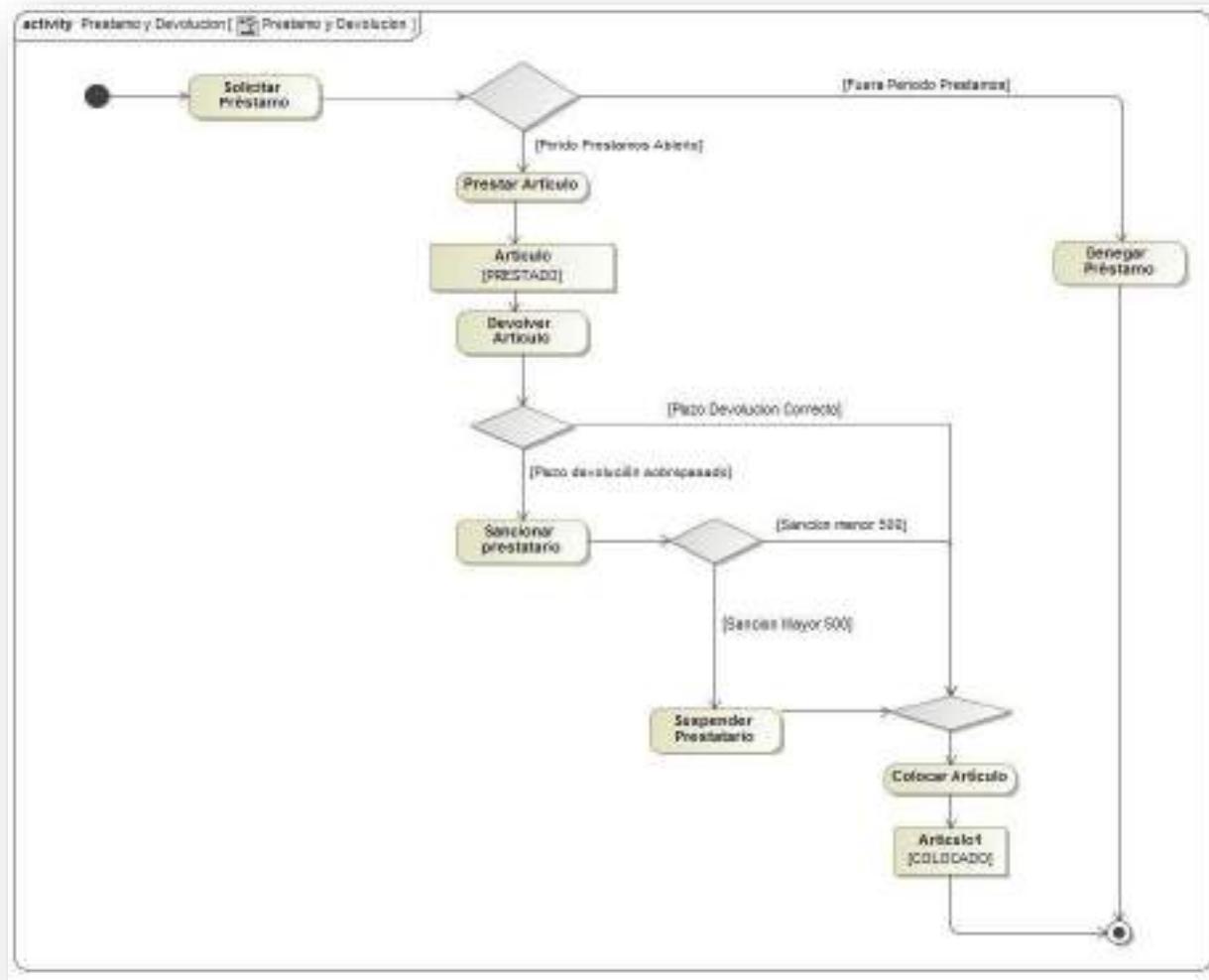
19

## **Biblioteca Universitaria**

- Vamos a modelar el préstamo y devolución de artículos en una misma actividad
- Proceso de solicitud de préstamos de artículos:
  - Si se está fuera del periodo de préstamos, el préstamo se deniega
  - Si no, se presta el artículo.
- Luego, hay que devolver el artículo
  - Si está dentro del plazo de devolución, no pasa nada, se coloca el artículo en su sitio
  - Si está fuera de plazo, se sanciona al prestatario
    - Si la suma de sus sanciones es mayor que 500, se suspende al prestatario y se coloca el artículo en su sitio
    - Si la suma es menor que 500, no se suspende al prestatario y se coloca el artículo en su sitio

# Ejemplo 2: Diagrama de actividad

20

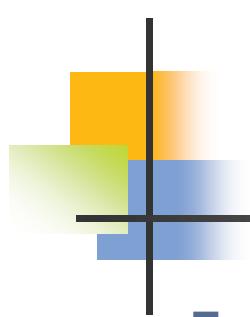




# Modelado con UML

## 5.4. Diagramas de Secuencia



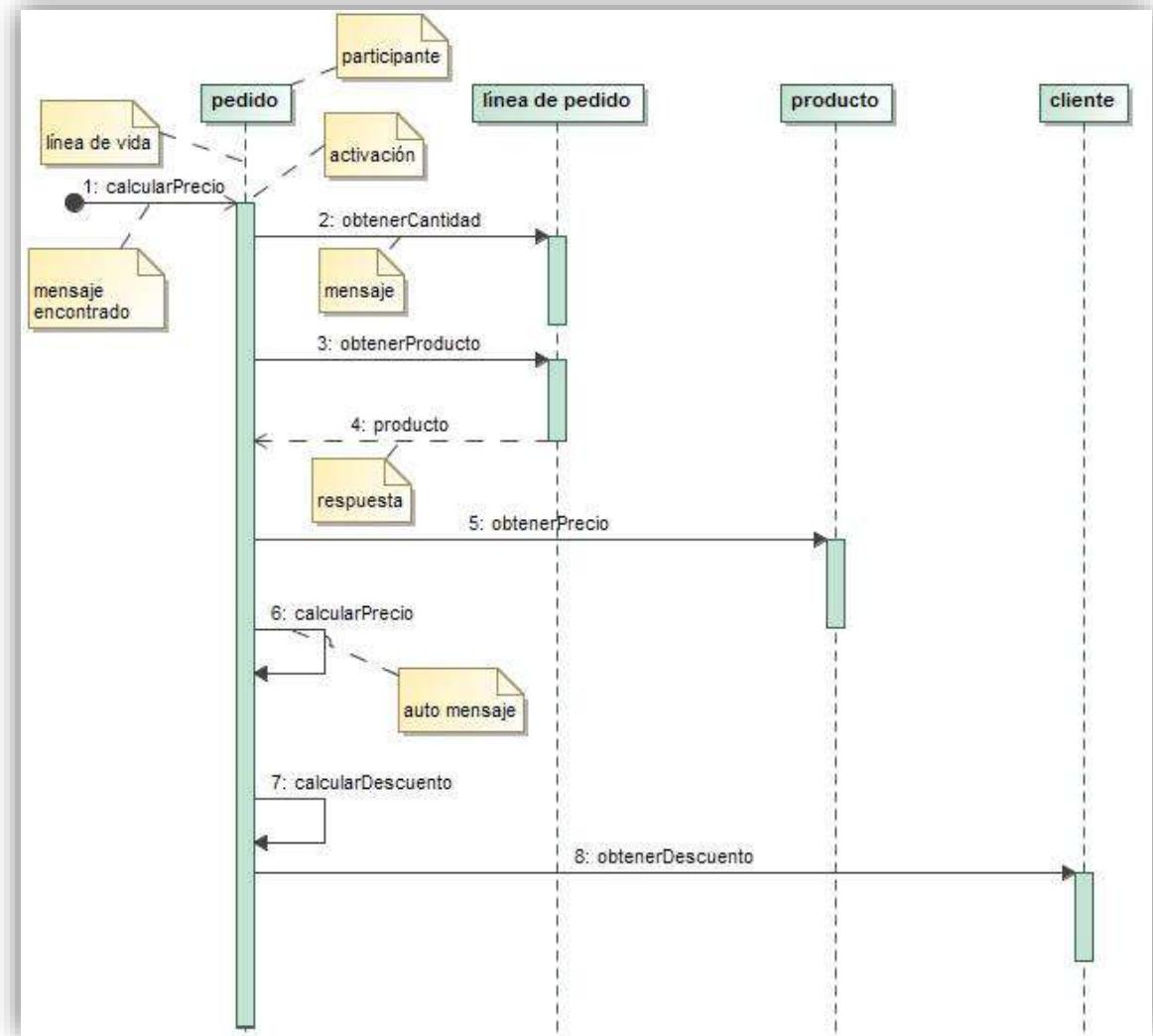


# Diagrama de Secuencia

- Muestran la **interacción** de un conjunto de **objetos** enfatizando el **orden en el tiempo** de los mensajes
  - las interacciones entre objetos en el orden secuencial en el cual las interacciones ocurren.
- En la **fase de requisitos**, los analistas suelen refinrar los **casos de usos** en uno o mas diagramas de secuencia
  - Contiene detalles de implementación del escenario
    - Objetos y clases usados para la implementación
    - Mensajes intercambiados entre los objetos
- No están pensados para mostrar lógicas de procedimientos complejos

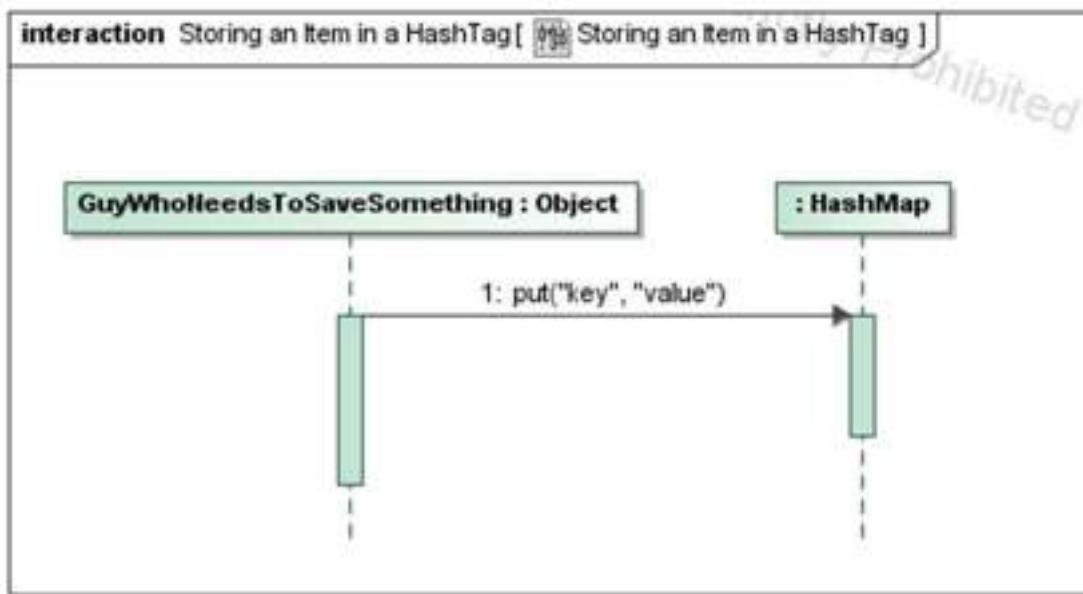
# Ejemplo: Calcular Precio de un Pedido

- Tiempo progresia hacia abajo.
- Los objetos involucrados en la operación aparecen de izquierda a derecha en función de cuándo toman parte en la secuencia de mensajes.



# Elementos de los Diagramas: Participantes

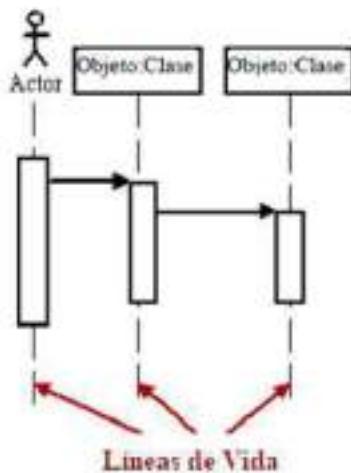
- Los **participantes** son instancias de clases (objetos)
  - Pueden tener nombre o ser anónimos
  - El siguiente diagrama tiene dos participantes, uno llamado *GuyWhoNeedsToSaveSomething* y el otro sin nombre



# Elementos de los Diagramas: Líneas de vida y Activación

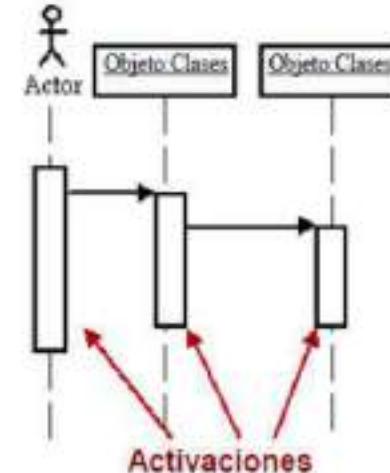
## ■ Líneas de vida

- Indican la presencia del objeto en el tiempo.
- Los objetos pueden crearse/destruirse.



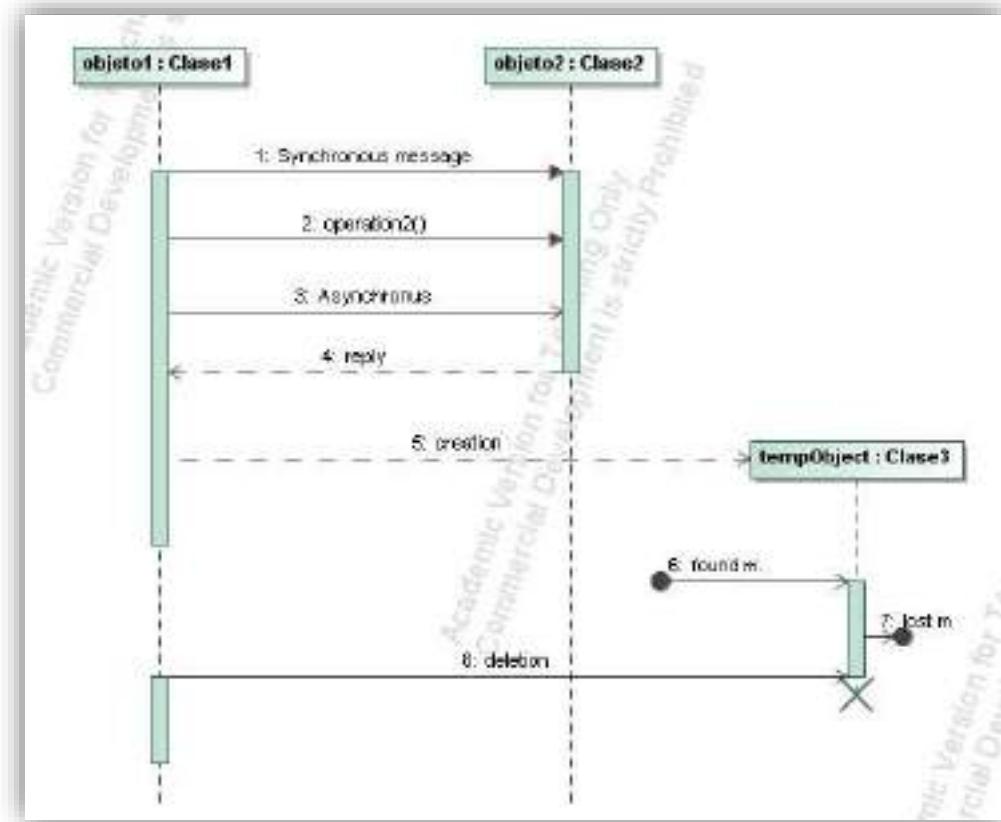
## ■ Activación

- Representa cuándo el participante está activo en la interacción.



# Elementos de los Diagramas: Mensajes

- Muestran una serie **interacciones** entre los participantes para llevar a cabo el proceso modelado.
- El primer mensaje de un diagrama de secuencia siempre se inicia en la parte superior.
- El mensaje que se envía al objeto receptor representa una **operación** que la clase del objeto implementa.

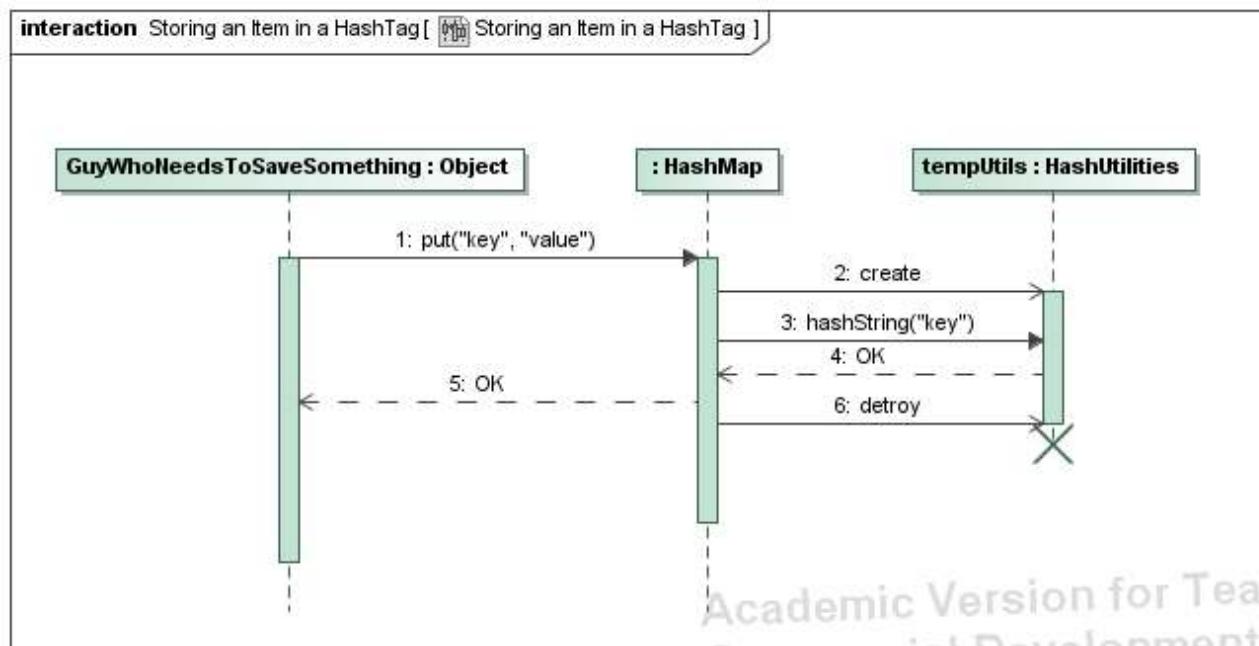


**Tipos de mensajes**



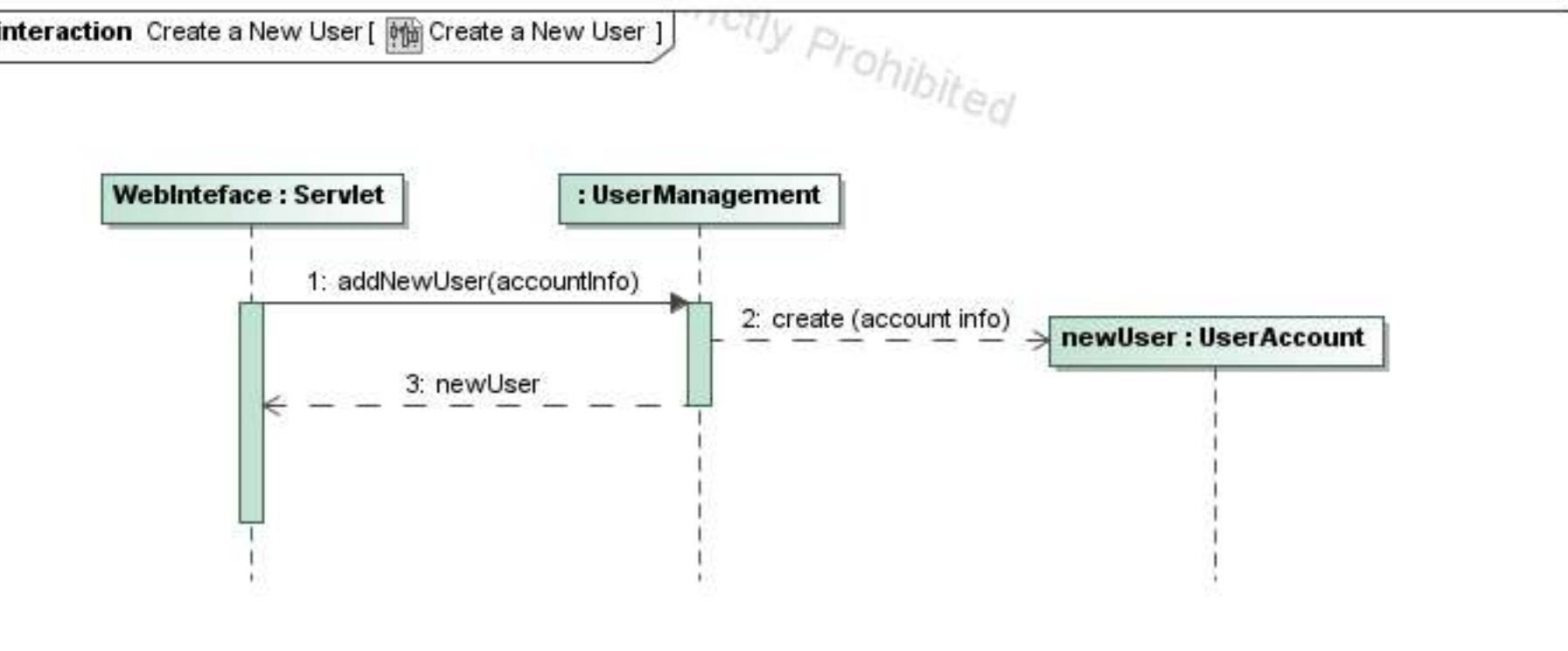
# Creación y destrucción (I)

- Los participantes (instancias) se pueden **crear y destruir de forma dinámica** durante la ejecución.



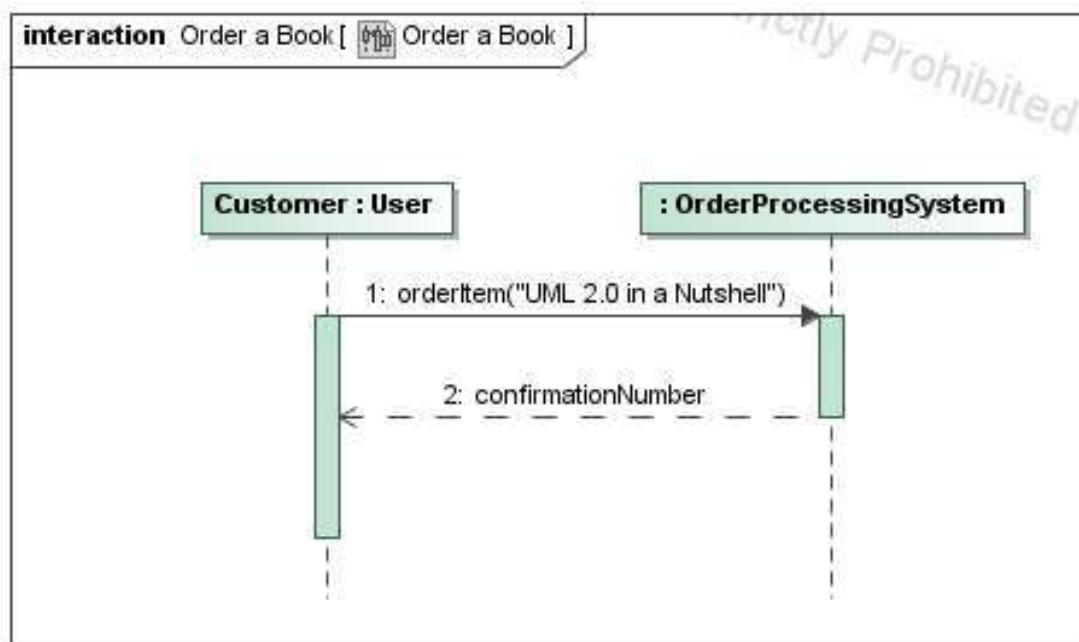
# Creación y destrucción (II)

interaction Create a New User [  Create a New User ]



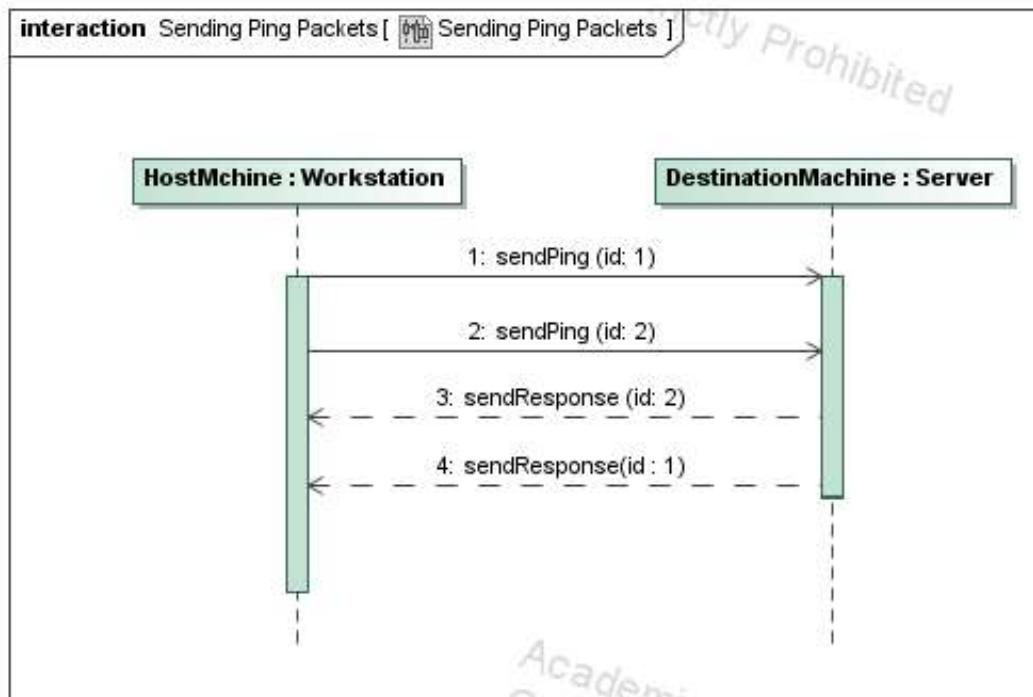
# Comunicación síncrona

- El objeto que envía el mensaje queda bloqueado hasta que recibe la respuesta
  - Los mensajes se representan con flechas con la cabeza llena
  - Los mensajes de respuesta se dibujan con línea discontinua



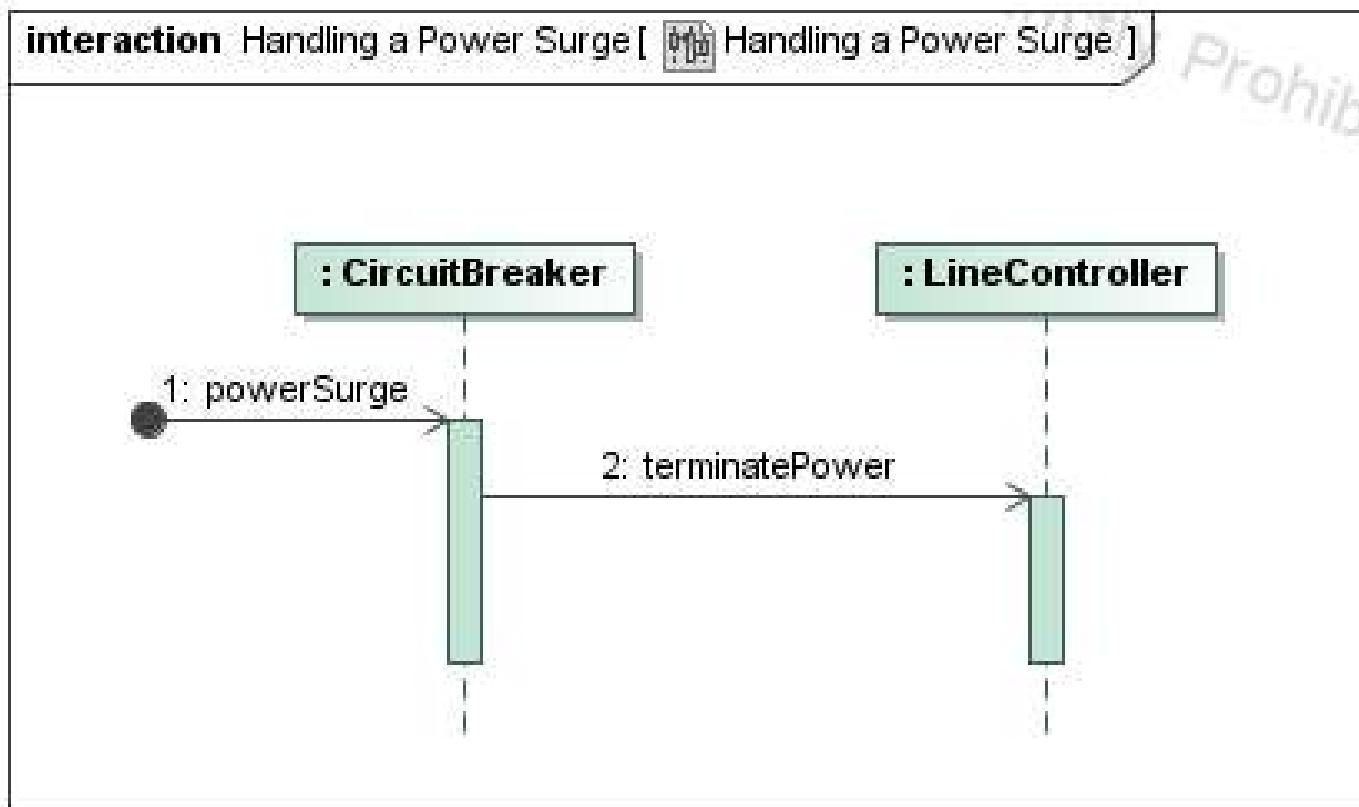
# Comunicación asíncrona

- Los mensajes asíncronos terminan inmediatamente
  - Crean un nuevo hilo de ejecución dentro de la secuencia
  - Se representan con flechas con la cabeza abierta



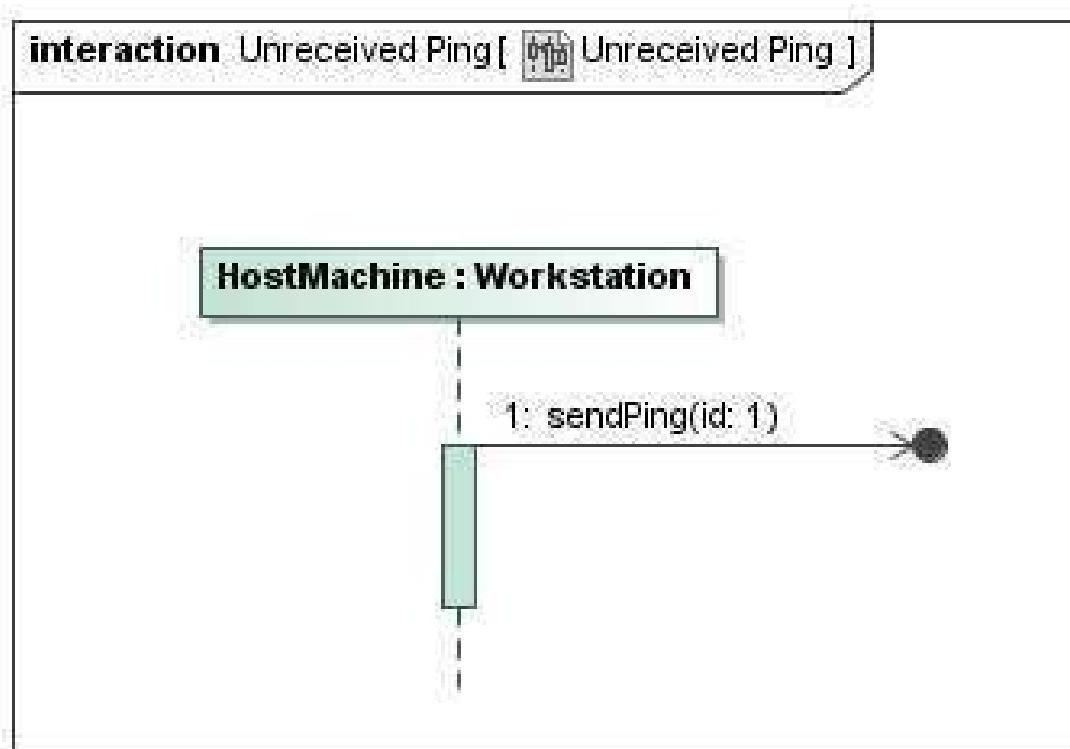
# Mensajes encontrados

- Son aquellos cuyo origen no importa



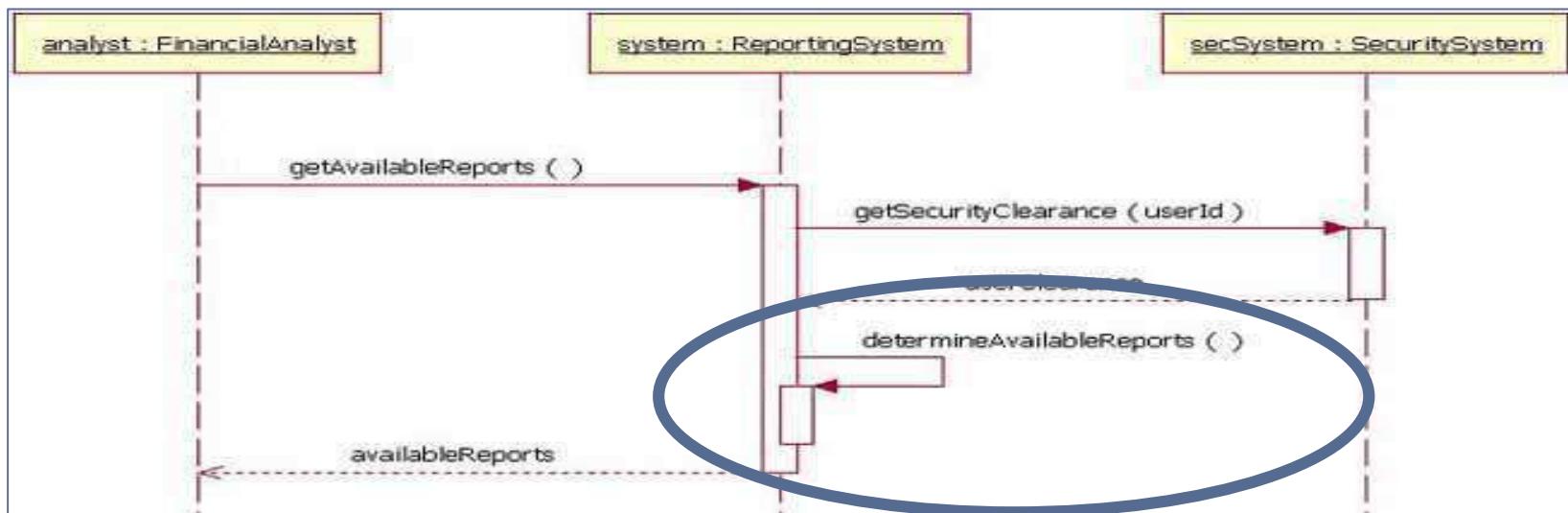
# Mensajes perdidos

- Son aquellos que no llegan a ningún participante



# Auto-mensajes

- Un objeto se manda mensaje a sí mismo
  - La flecha parte y termina en el mismo objeto

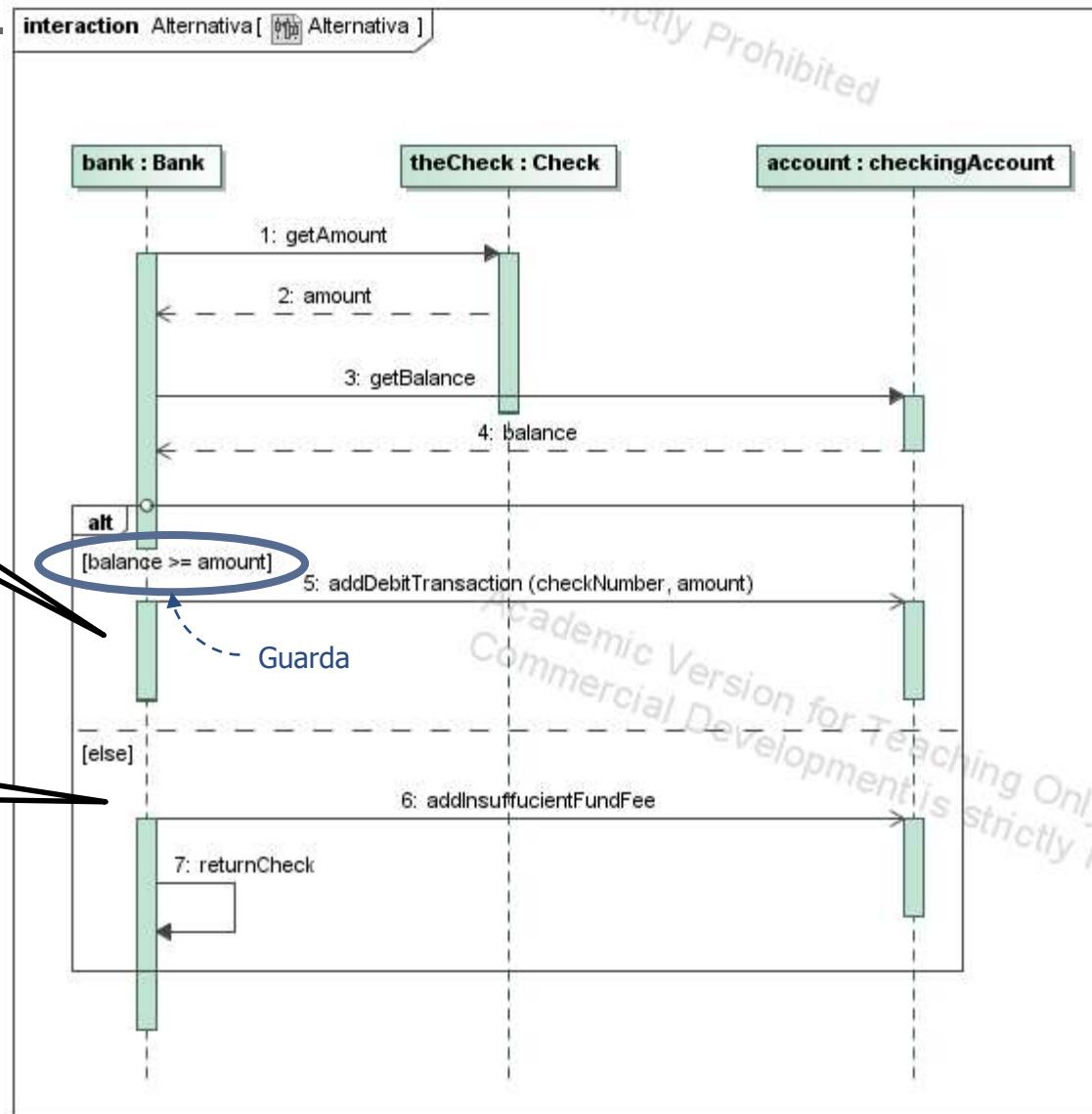




# Fragmentos combinados

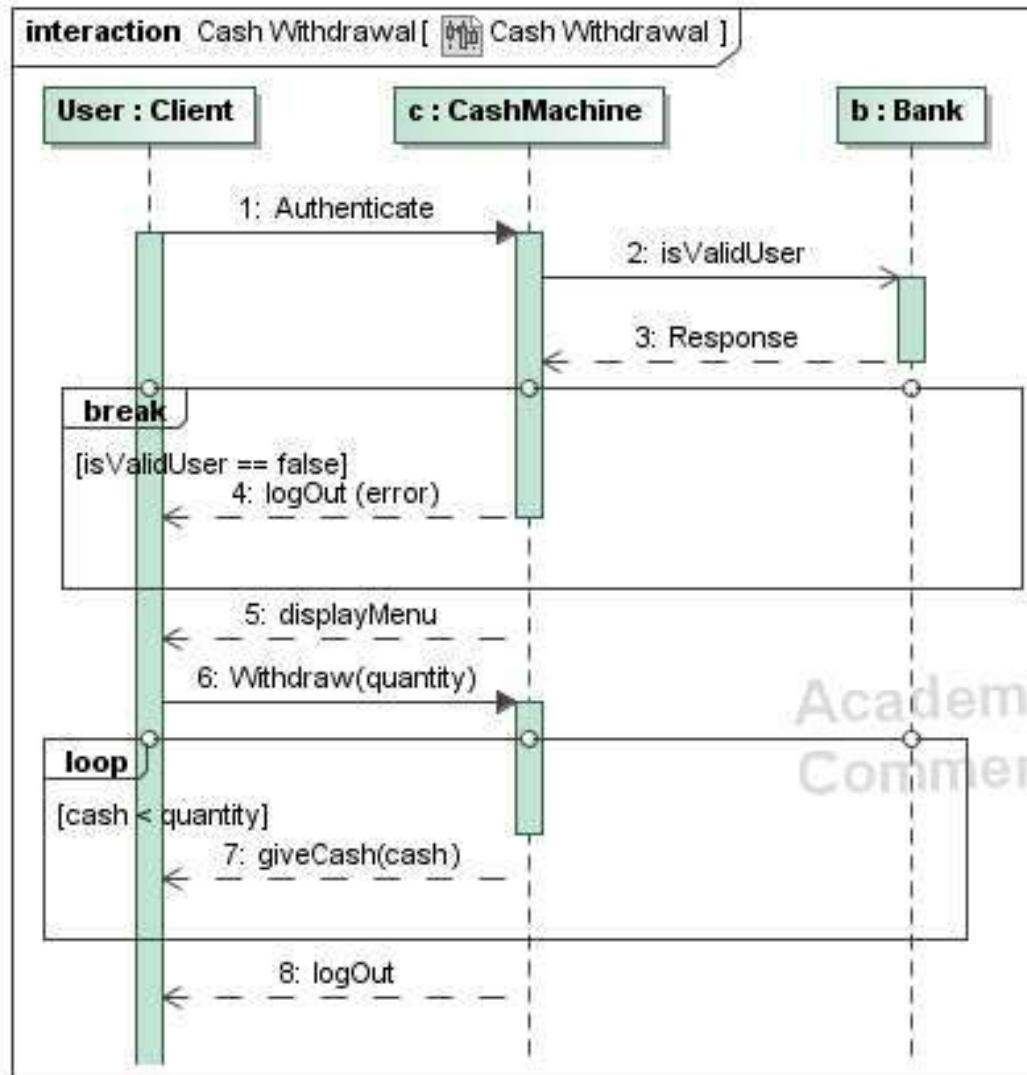
- Permiten definir una interacción compleja mediante componentes más simples
  - **Alternativa**
  - **Opción**
  - **Paralelas**
  - **Break**
  - **Bucle**
  - Etc.
- Usar con mesura: tienden a complicar el diagrama.

# Alternativas



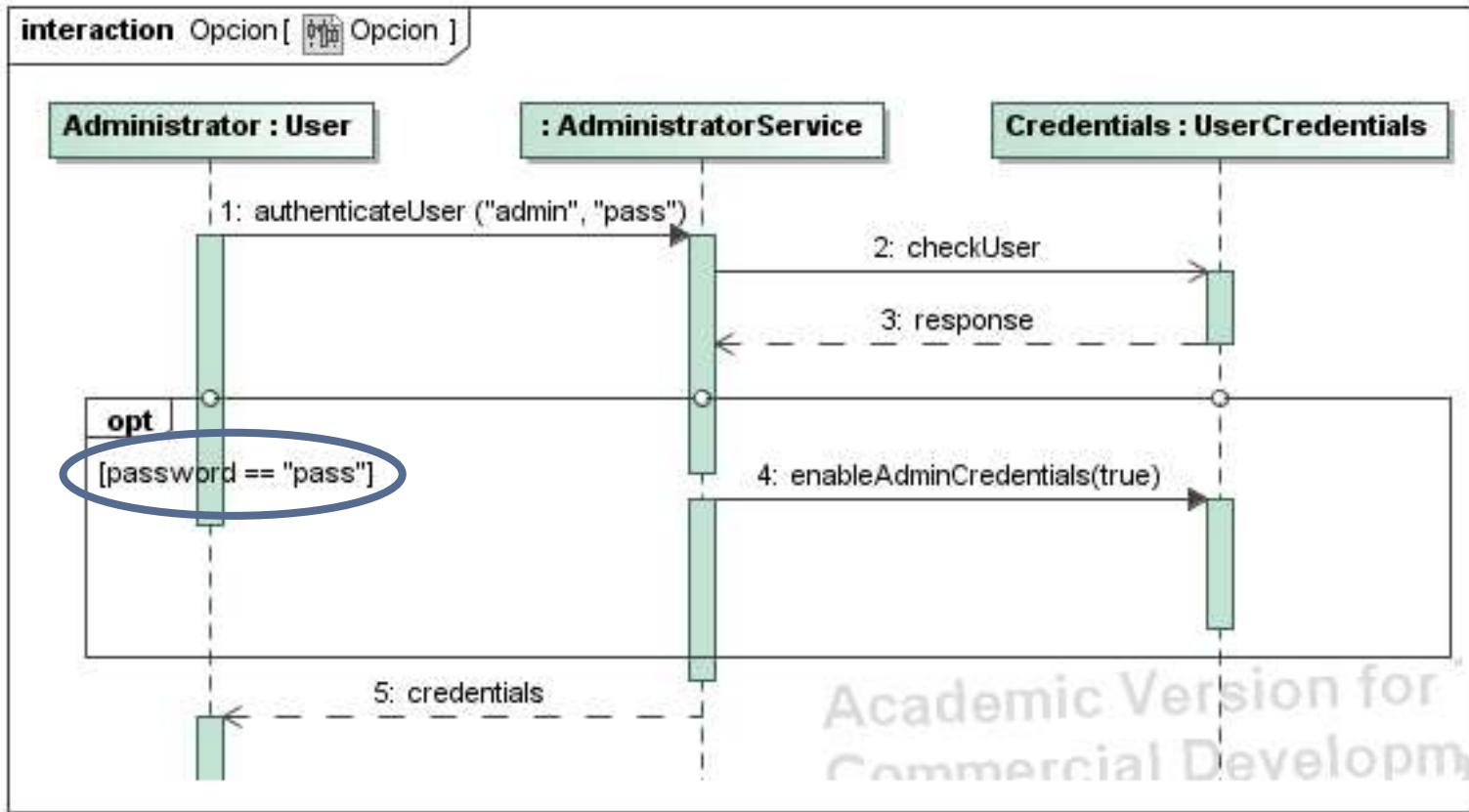
5.4. Diagramas de Secuencia

# Bucles



# Opción

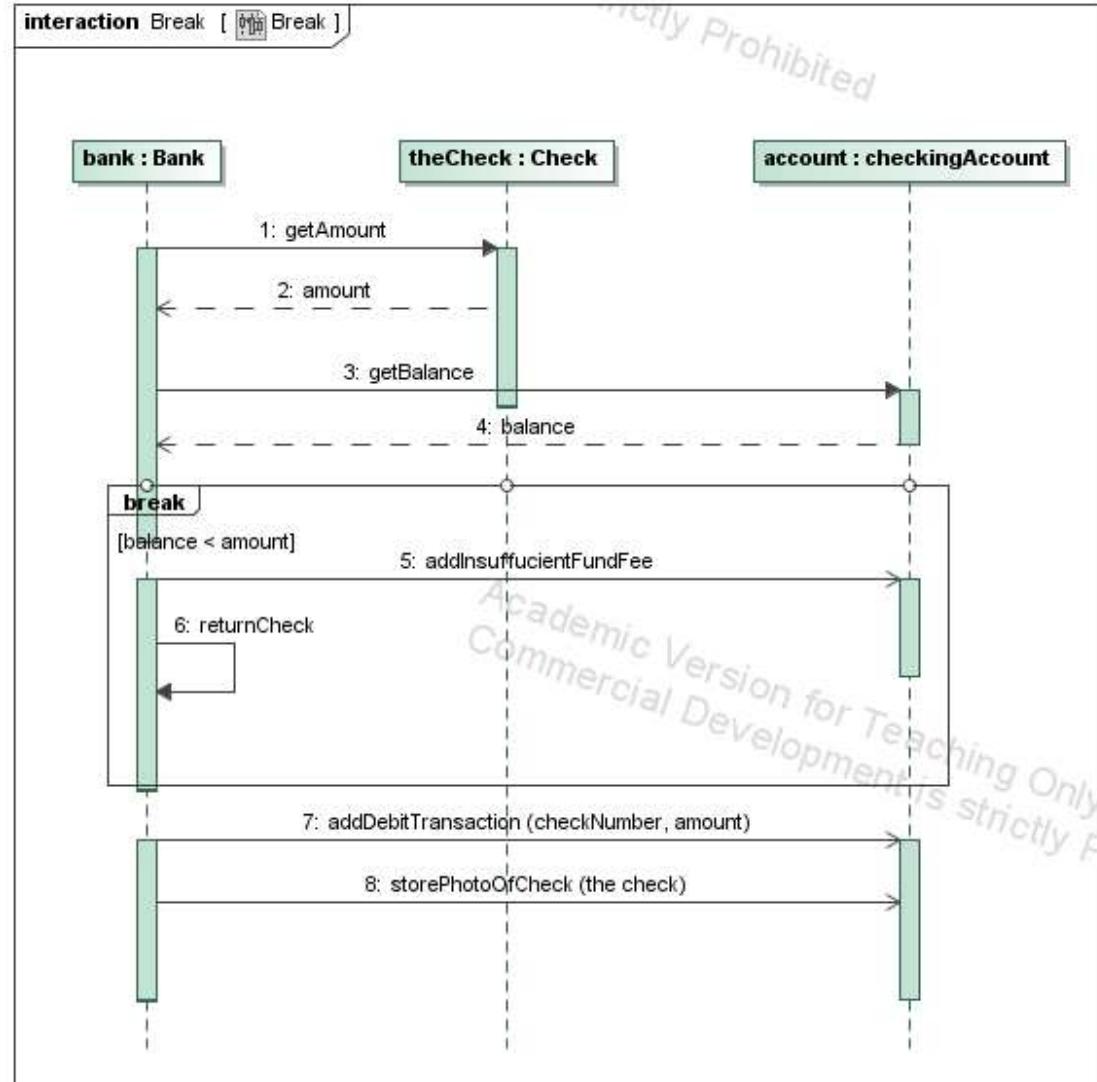
- Ciertas acciones se ejecutan sólo si se cumple la condición



Academic Version for  
Commercial Development

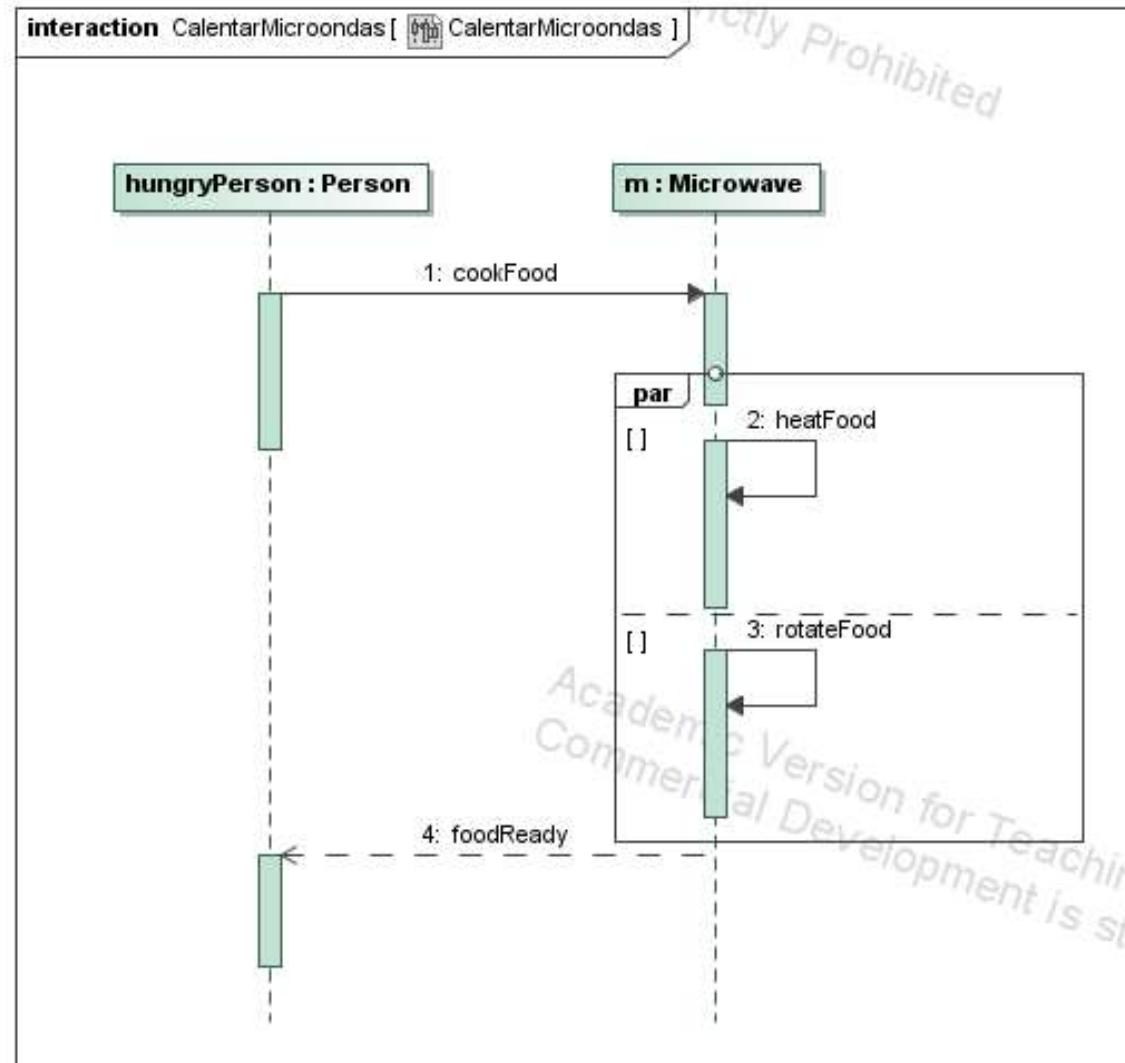
# Break

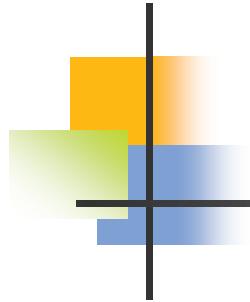
- Similar la opción, pero lo que sigue no se ejecuta si entra en el *break*.
- Ej.: Si [balance<amount]
  - Se ejecutan mensajes del fragmento break.
  - Se detiene la ejecución del resto de interacciones.



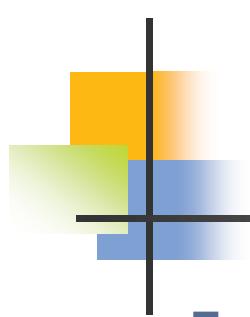
# Paralelo

- ▶ Especifica que varias acciones se realizan a la vez, cada una en un hilo de ejecución





# EJEMPLOS

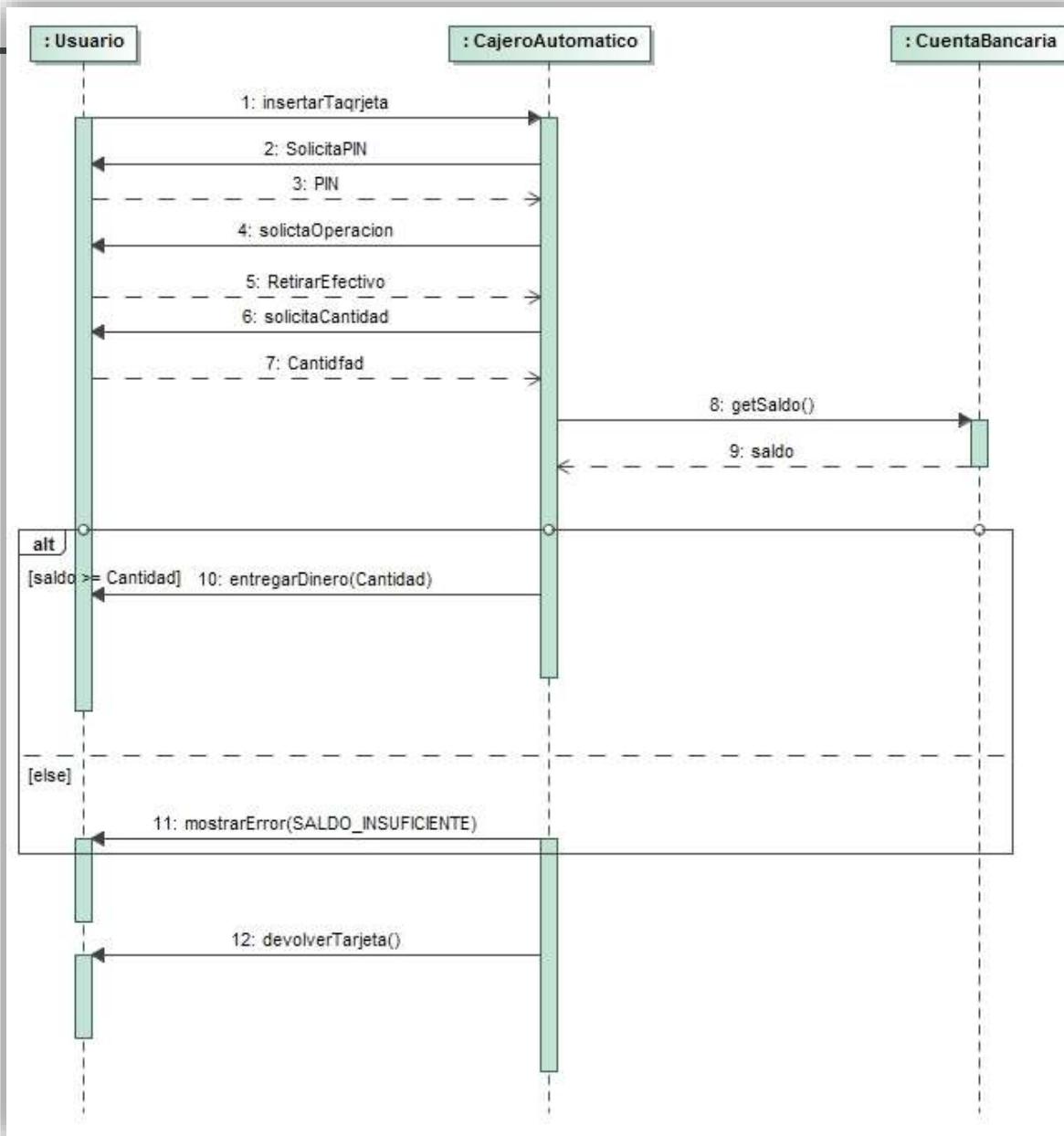


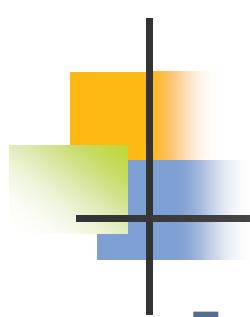
# Ejemplo Cajero Automático

## Retirada Efectivo

- Crear un diagrama de secuencia para la retirada de efectivo de un cajero automático de un banco.
- Los actores son:
  - Usuario,
  - Cajero Automático,
  - Cuenta Bancaria
- El Cajero debe
  - Solicitar el PIN que supondremos correcto
  - Solicitar la operación a realizar (retirada de efectivo en este caso)
  - Comprobar que hay suficiente dinero antes de permitir la retirada del efectivo

# Ejemplo Cajero Automático Retirada Efectivo

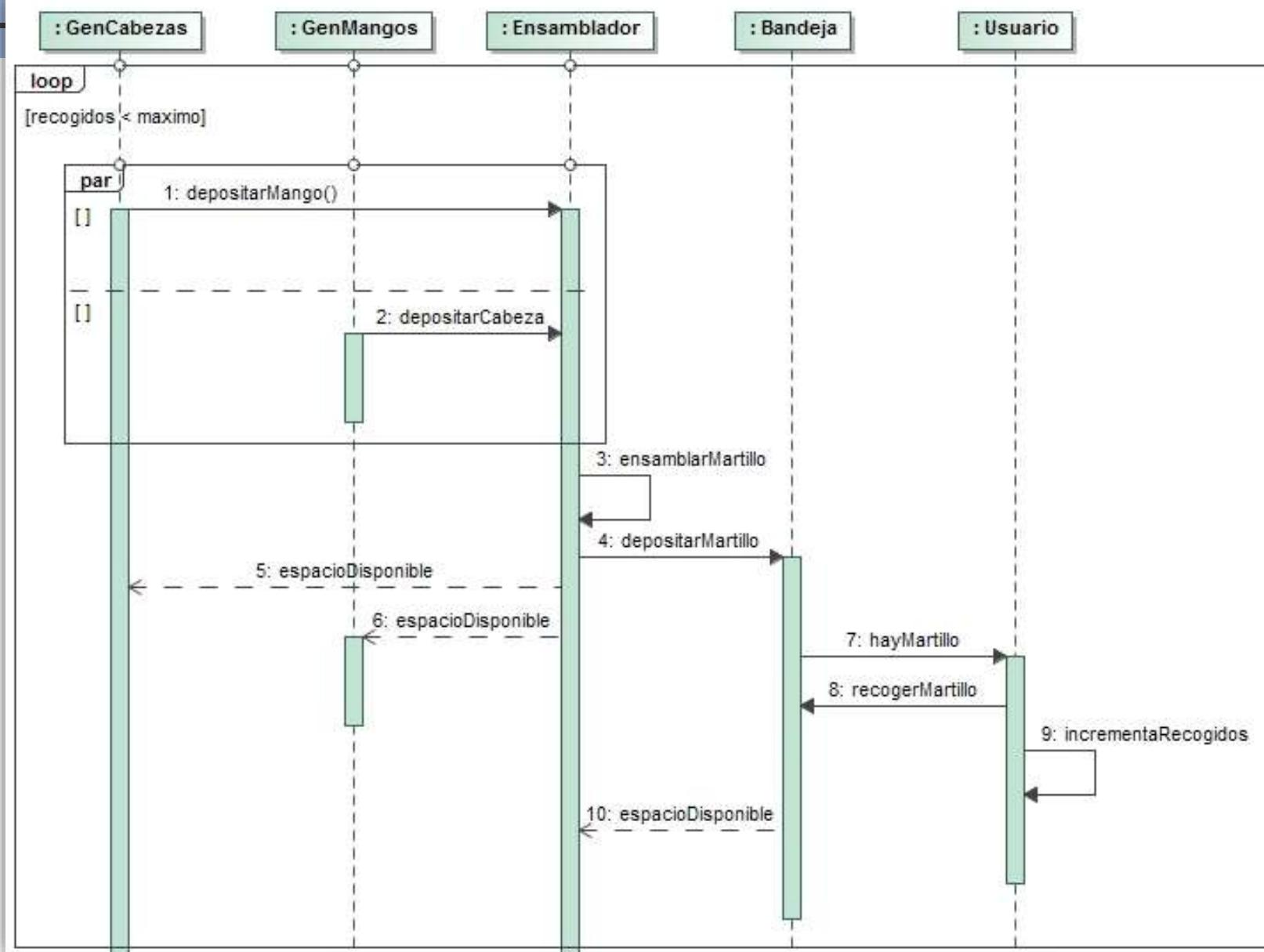


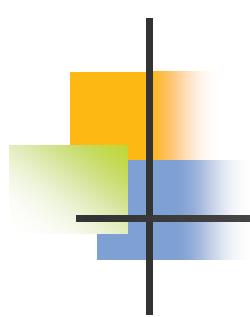


# Ejemplo: Factoría de Martillos

- El siguiente proceso se repite mientras que el número de martillos recolectados por el Usuario sea menor que un *máximo*:
  - El Generador de mangos y el Generador de cabezas depositan un mango y una cabeza, respectivamente, en el Ensamblador. Ambas máquinas trabajan en paralelo.
  - A continuación, el Ensamblador deposita un martillo en la Bandeja, y hace saber a ambos Generadores que tiene espacio disponible
  - La Bandeja le hace saber al Usuario que hay un martillo disponible, y el Usuario lo recoge. La Bandeja le hace saber al Ensamblador que tiene espacio disponible

# Ejemplo: Factoría de Martillos

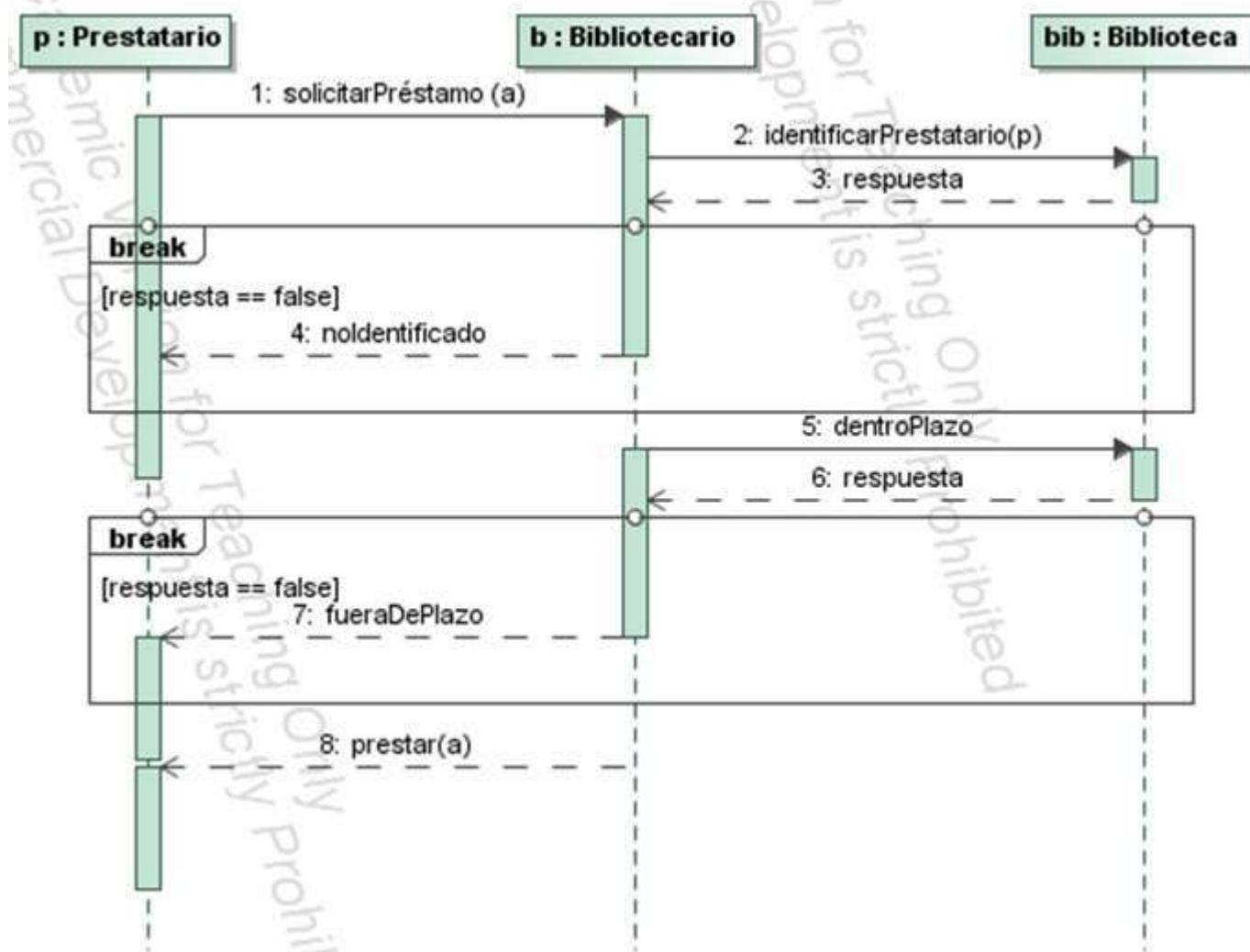


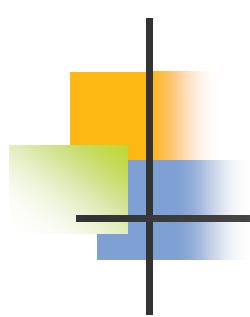


# Ejercicio: Préstamo de artículo

- El Prestatario solicita un préstamo al Bibliotecario
  - El Bibliotecario consulta con la Biblioteca si el usuario está identificado
    - Si no lo está, el Bibliotecario le dice al Prestatario que no está identificado y se acaba el proceso
    - Si está identificado, el Bibliotecario consulta a la Biblioteca si se está dentro de plazo para préstamos
      - Si se está dentro de plazo, el Bibliotecario presta el libro al Prestatario
      - Si se está fuera de plazo, no le presta el libro

# Ejercicio: Préstamo de artículo (II)

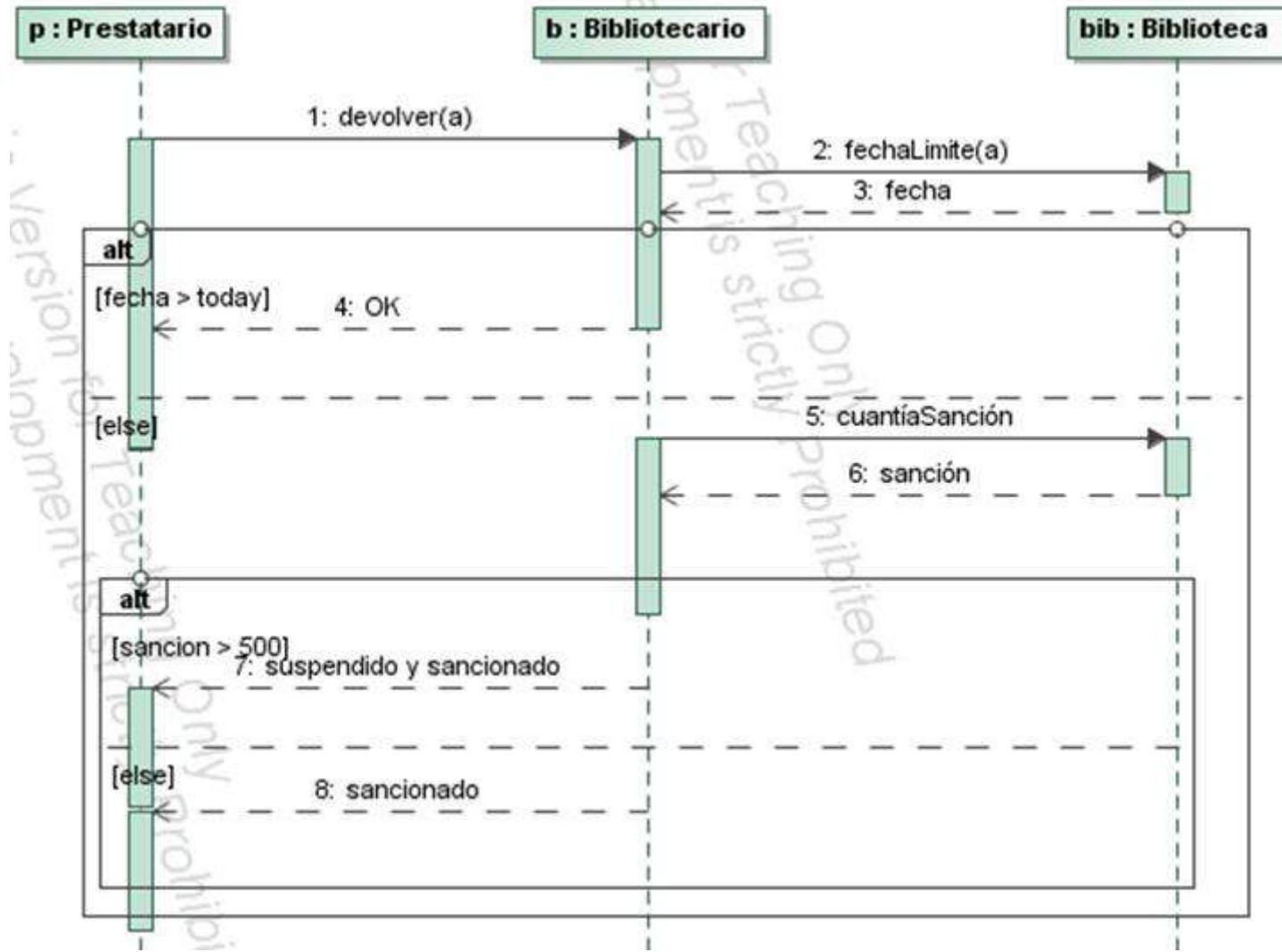




# Ejercicio: Devolución de artículo

- El Prestatario solicita al Bibliotecario la devolución de un artículo
  - El Bibliotecario comprueba la fecha límite de devolución del artículo con la Biblioteca
    - Si la fecha es posterior al día de hoy, el Bibliotecario acepta la devolución del artículo
    - Si no, el Bibliotecario le pregunta a la Biblioteca la cuantía de la sanción al Prestatario
      - Si la sanción es mayor que 500, el Bibliotecario suspende y sanciona al Prestatario
      - Si es menor, sólo le sanciona

# Ejercicio: Devolución de artículo (II)



5.4. Diagramas de Secuencia



# Ejercicio para casa

- Crear un diagrama de secuencia para el ejemplo de la máquina expendedora de café.
- Describir el caso de uso “Comprar café”.
- Suponed que
  - Puede haber o no existencias del producto seleccionado.
  - El cliente paga con dinero en efectivo.
  - Inserta la cantidad exacta (no hay que dar cambio).
- Discutid brevemente la solución aportada.

# Diagramas UML



**DIAGRAMA DE MÁQUINA DE ESTADOS**

## 5.5. Diagrama de Maquinas de estado



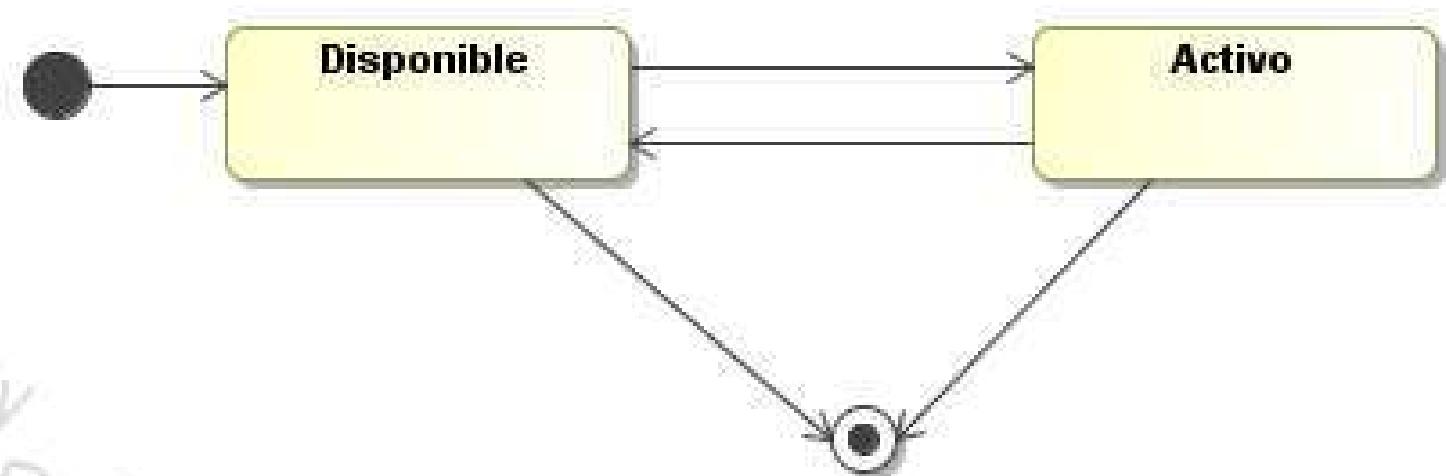
# Maquinas de estado

3

- Otro forma de describir el comportamiento de elementos del modelo dinámico.
  - Muy relacionados con los D. de actividad.
- Se usan fundamentalmente para describir el comportamiento de una clase (objeto).
  - Otros elementos como: casos de uso, actores, subsistemas u operaciones.

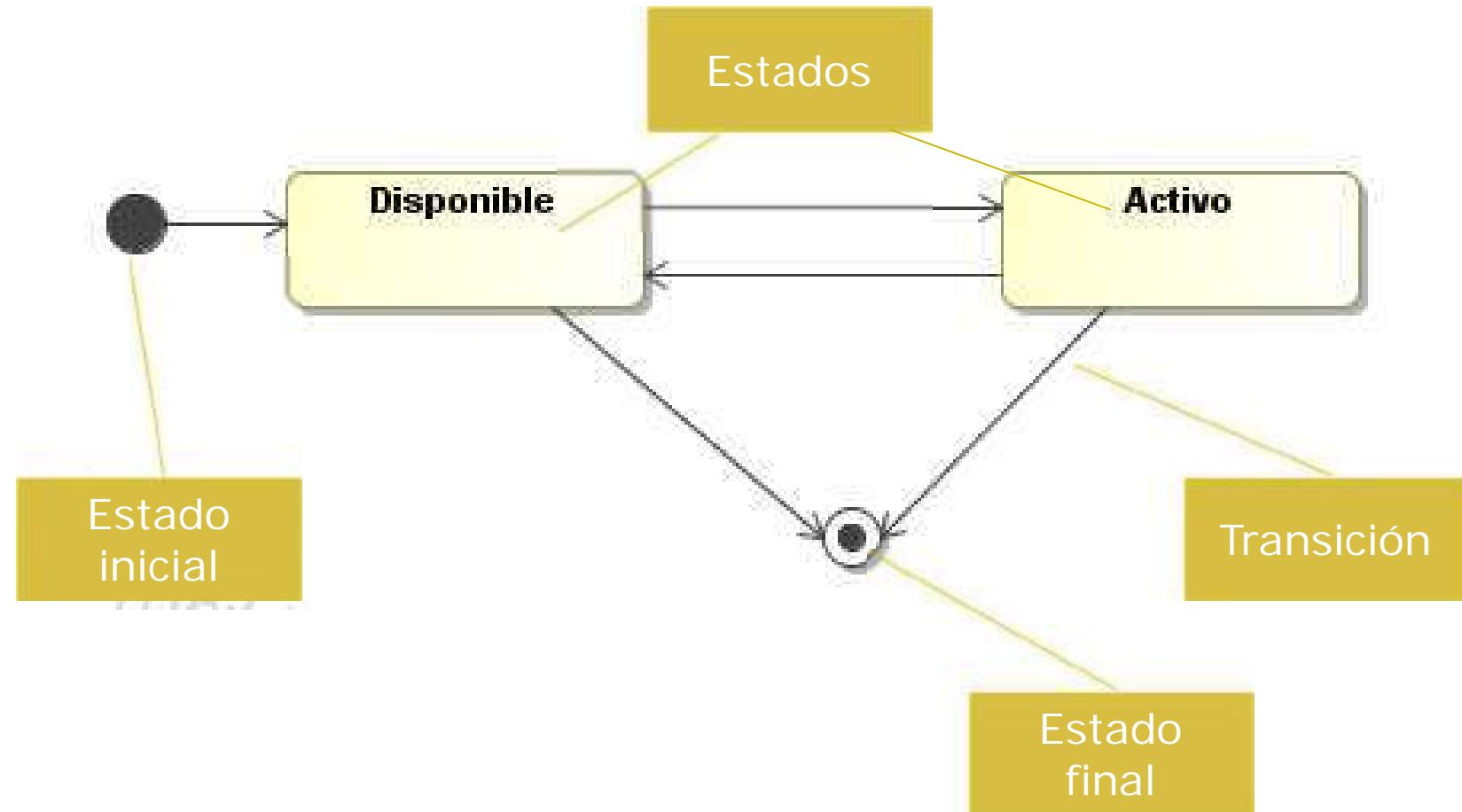
# Un maquina de estados simple describiendo la clase Viaje

4



# Un maquina de estados simple describiendo la clase Viaje

5



# Eventos

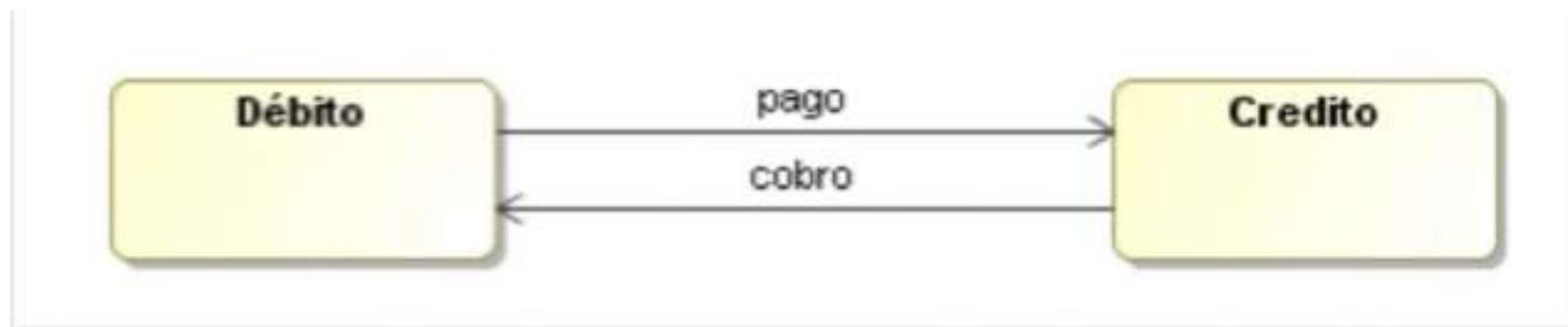
6

- ¿Cómo se producen las transiciones entre estados?
- Los sistemas están dirigidos por eventos (externos e internos)
- El tiempo también es una fuente de eventos.

# Transiciones

7

- Una transición es el movimiento entre estados.
- Una transición es siempre en respuesta a algún evento.



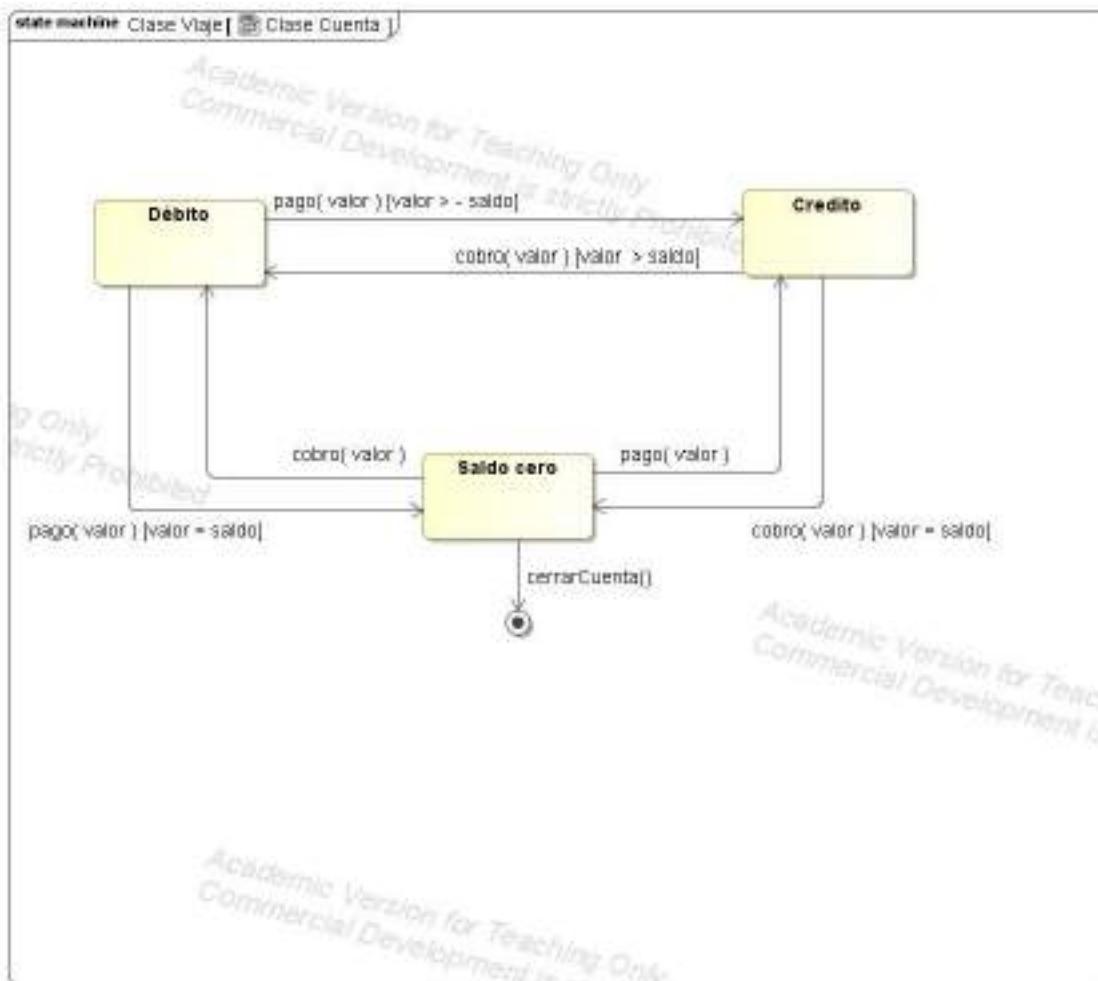
# Transiciones

8

- De un estado se puede salir por más de una transición
  - Añadimos un estado de “saldo 0” a cuenta
  - 3 eventos pueden provocar la salida de este estado
    1. Un pago
    2. Un cobro
    3. El cierre de la cuenta
- Se pueden poner condiciones (guardas) a las transiciones.
  - entre corchetes cuadrados [ ]
- La transición sólo se produce si la condición es verdadera.

# Clase Cuenta

9



# Acciones internas

10

- Un estado puede tener acciones “disparadas”.

**Entry**: la acción se realiza en cuanto se entra en el estado.

**Exit**: la acción se realiza justo antes de salir del estado.

**Do**: estas acciones tiene lugar mientras el estado está activo.

**Event**: estas acciones se realizan en respuesta a un evento, y no producen una transición a otro estado.

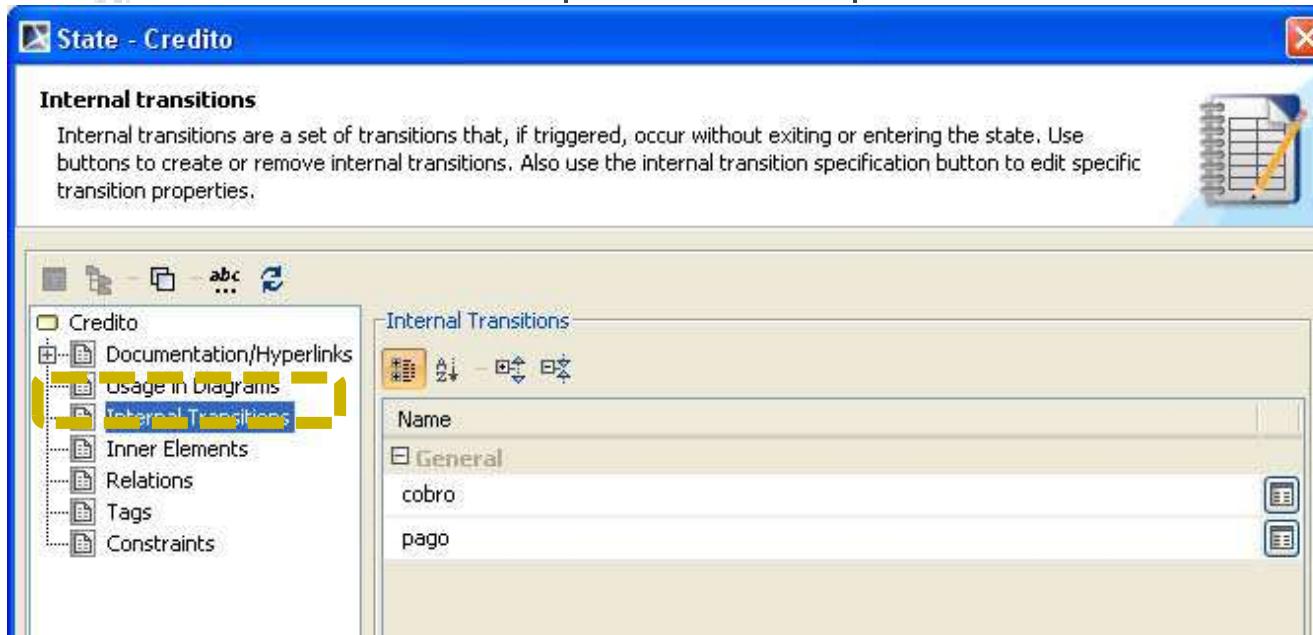
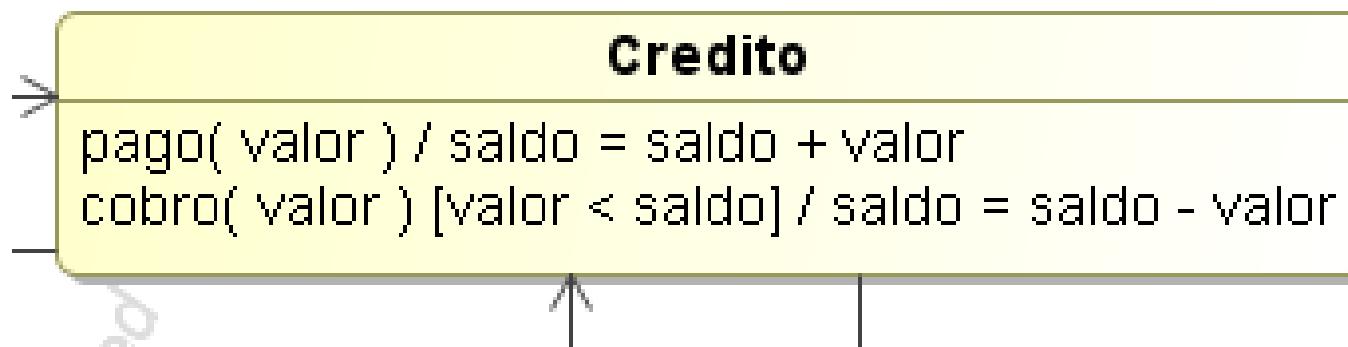
# Acciones

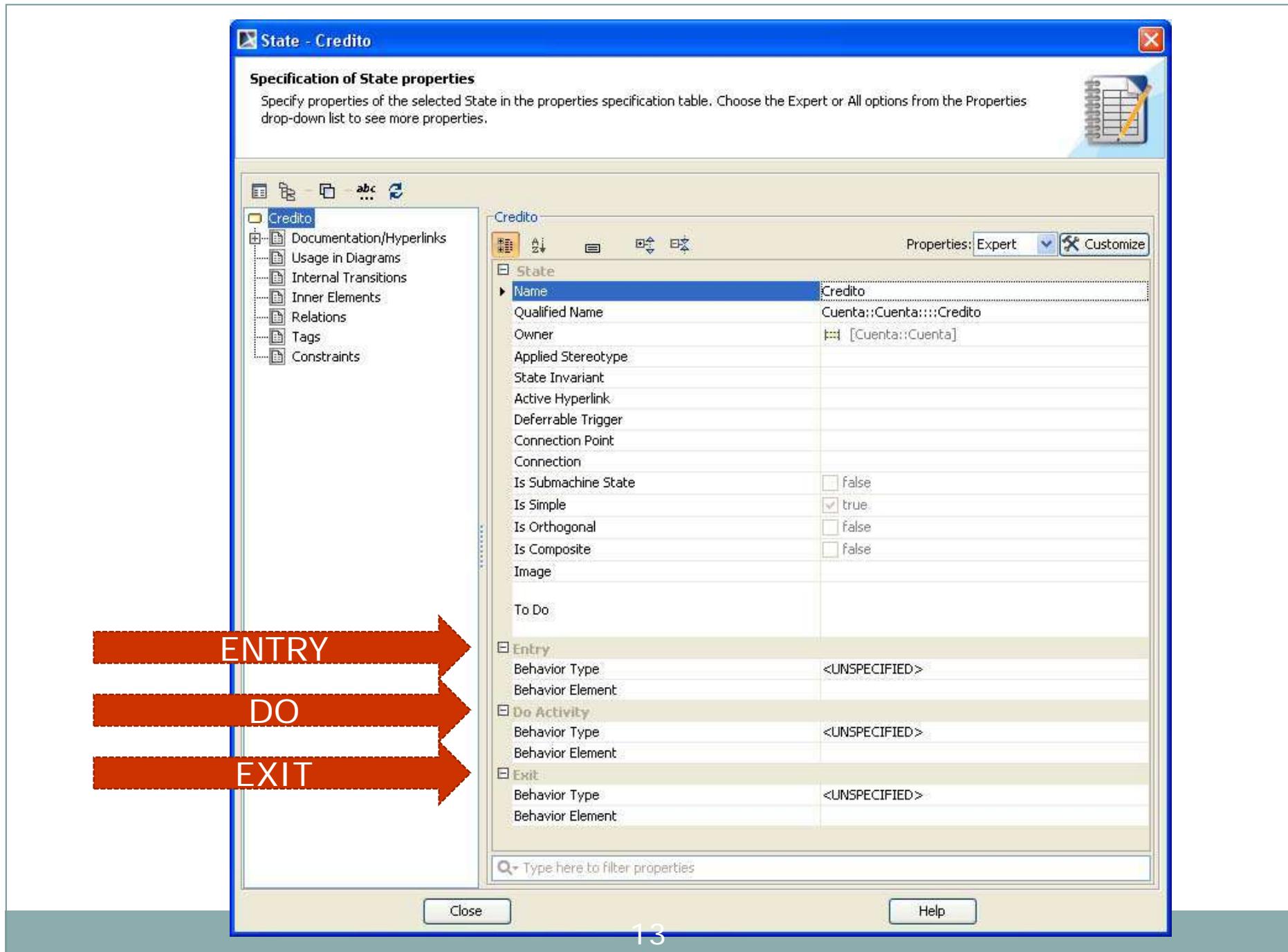
11

- Sintaxis: *etiquetaAccion / accion*
  - *etiquetaAccion* puede ser: entry, exit, do  
do / addCompartidor  
entry / resetCount  
exit / count = count + 1
- Sintaxis-2:
  - nombreEvento (argumentos) [condicion] / accion*  
pago(valor) / saldo=saldo+valor  
cobro(valor)[valor<saldo] / saldo=saldo-valor

# Acciones internas

12





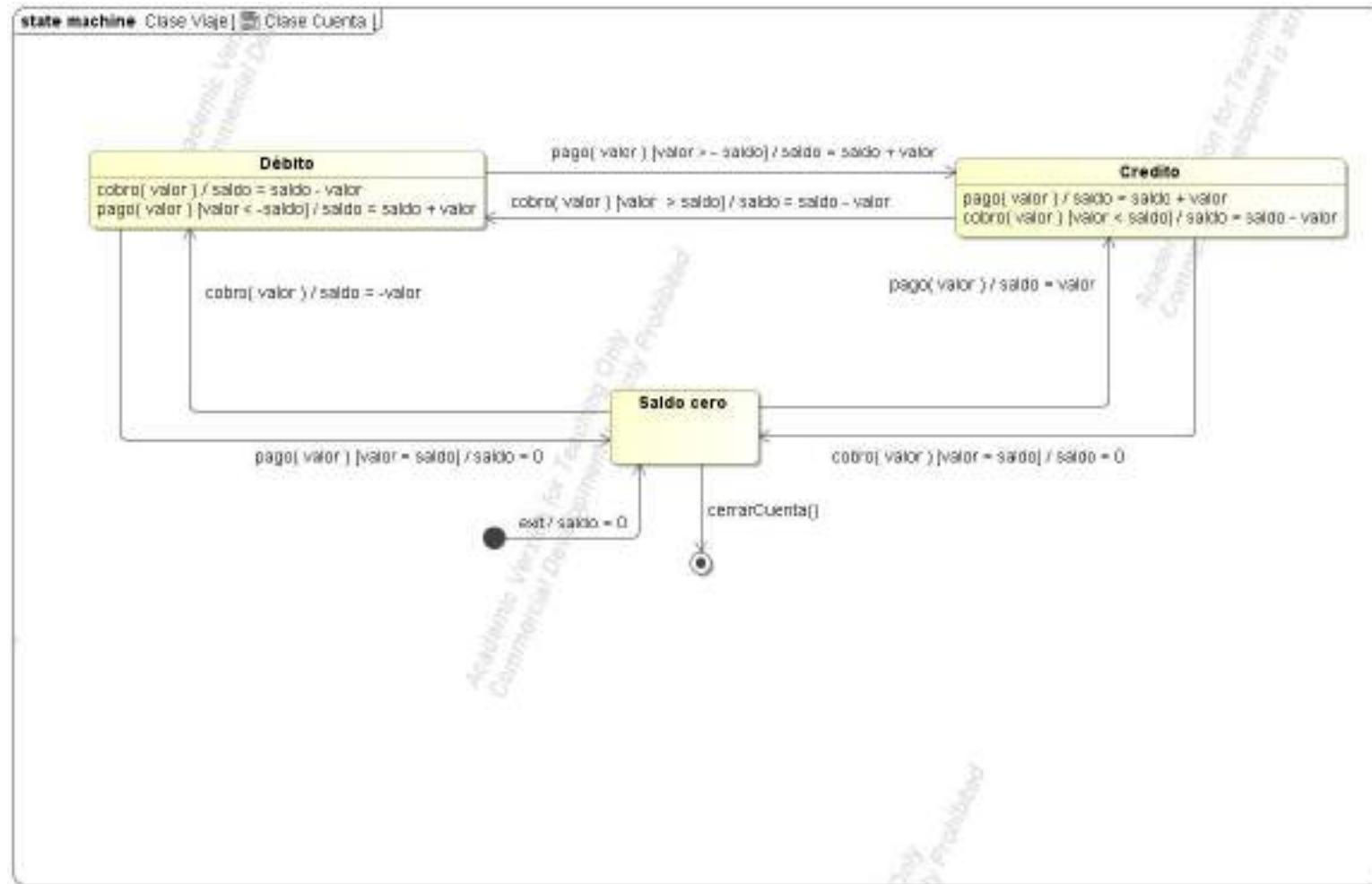
# Acciones en las transiciones

14

- Un transición puede disparar una acción.
- Sintaxis es similar a las acciones internas
  - *nombreEvento (argumentos) [condicion] / acción*
- Ejemplo completo de la cuenta para ComparteTuCoche.

# Ejemplo completo de la cuenta para ComparteTuCoché

15



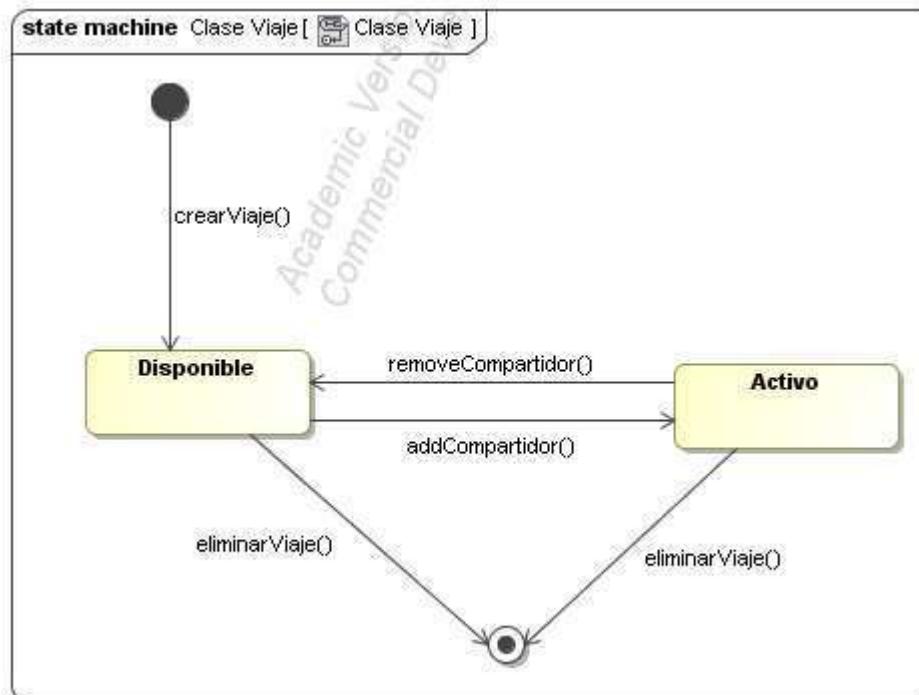
# Estados compuestos

16

- Un estado compuesta se puede dividir en sub-estados:
  - Se pueden dibujar dentro del estado compuesto.
  - O en un diagrama a parte.

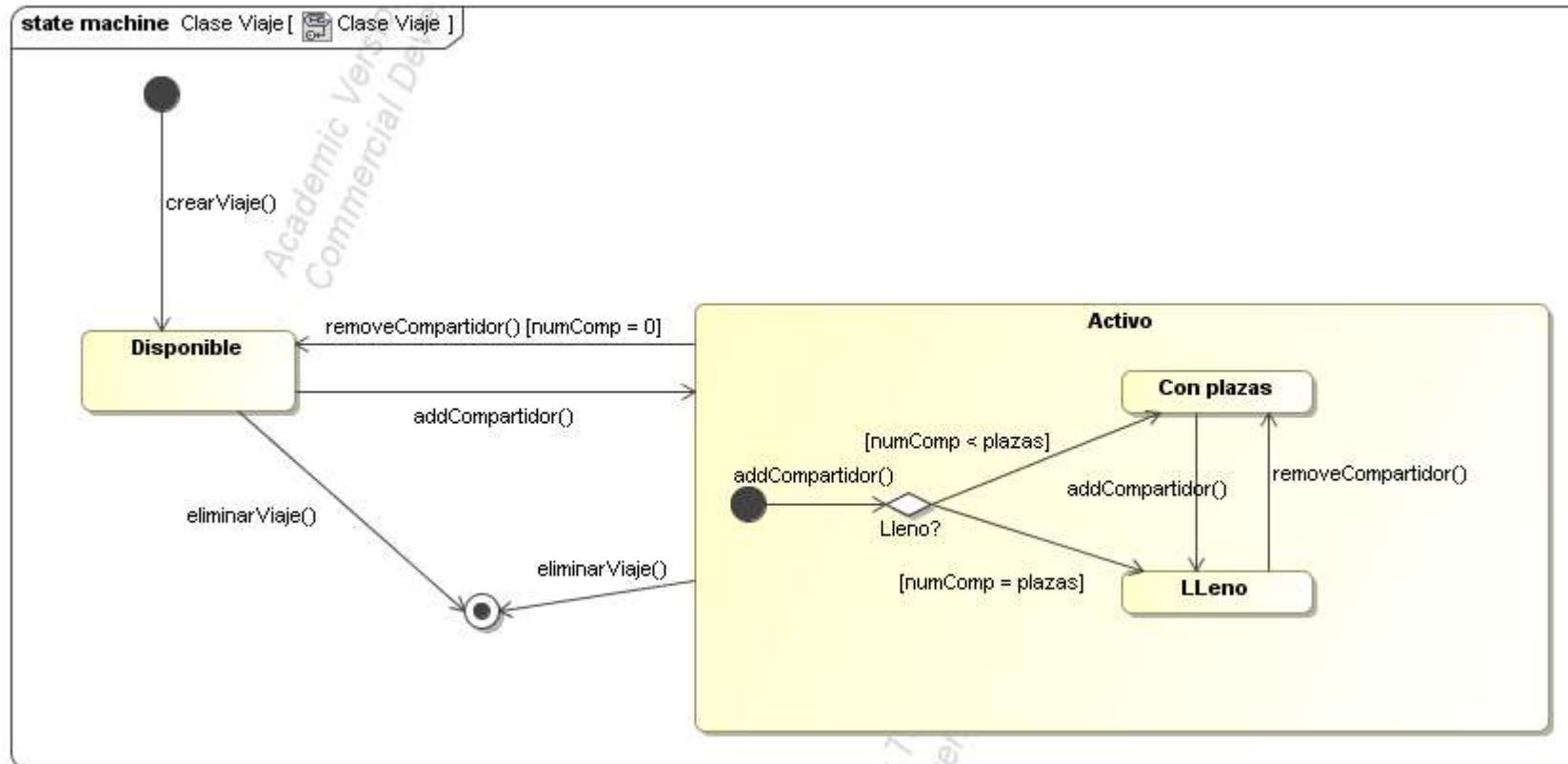
# Clase Viaje

17



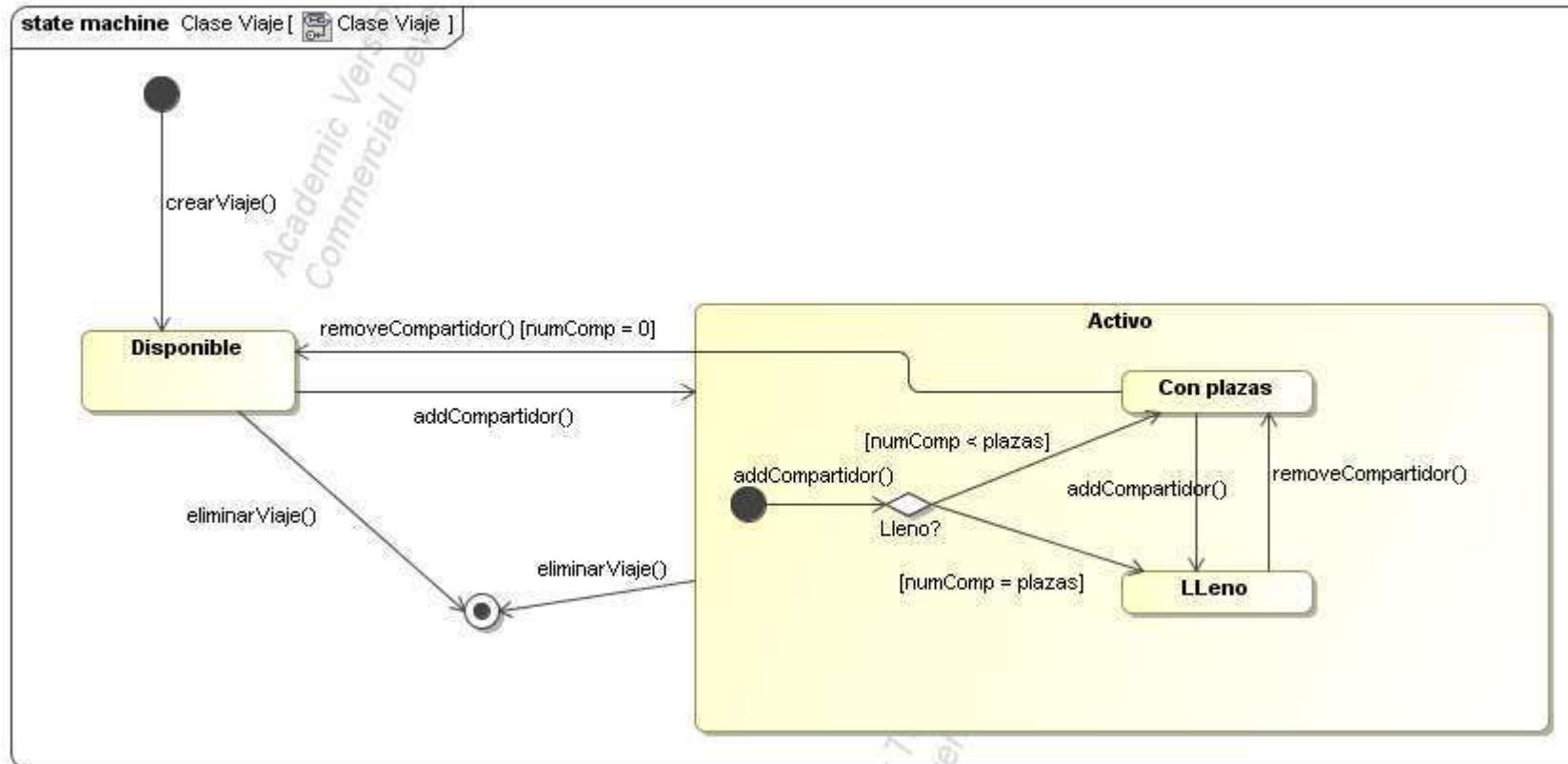
# Clase Viaje con estado compuesto

18



# Clase Viaje con estado compuesto

19





1

# PLANIFICACIÓN DE PROYECTOS SOFTWARE

Grado en Ingeniería Informática



# Contenido de la Presentación

2

- Visión General.
- Creación de tareas.
- Duración.
- Vinculación de tareas.
- Creación de calendarios.
- Creación Hoja de Recursos.
- Asignación de recursos.
- Informes.

# Introducción

3

- Microsoft Project una herramienta para planes de proyectos.
- **Crear** planes de proyectos, **comunicarlos** a otros usuarios y **adaptarse** a los cambios a medida que éstos se van produciendo.
- 100% Microsoft.



# Administración de un Proyecto

4

- Planificación, organización y administración de tareas y recursos necesarios para llevar a cabo un objetivo definido, normalmente con limitación de tiempo y costos
- 3 Fases:

- **Planificación del proyecto y creación de una programación:**

- Definición de las tareas y duraciones,
    - Establecimiento de relaciones entre tareas
    - Seguimiento del uso de recursos y la asignación de los mismos.

Todas las fases posteriores del proyecto se basan directamente en la información que se les proporciona cuando se planifica el proyecto.

- **Adaptación a los cambios:**

- Seguimiento y ajuste de la programación ® reflejar cambios durante el desarrollo.

- **Comunicación de la información del proyecto:**

- Comunicar información del proyecto a los clientes, personal y administración ® mostrar la información deseada en el formato conveniente para cada usuario.

# Nuestro Objetivo

5

- Crear una Planificación de Proyecto.
- Pasos a seguir:
  - La introducción de tareas y duraciones.
  - La organización de la lista de tareas en una estructura de esquema.
  - La vinculación de tareas y el ajuste de relaciones entre las mismas.
  - La introducción y asignación de recursos.
  - Seguimiento y gestión de la programación.

# Nuestro Puzzle

6

- 4 Piezas principales:
  - Tareas, recursos, seguimiento e informe.



TAREAS	RECURSOS	SEGUIMIENTO
Definir el proyecto Creas lista Organizar Fases Programar	Personas y Equipamiento Costos Asignar a Tareas	Plan Previsto Progreso de Tareas Cambios

# Vocabulario

7

- **Tareas:** hechos necesarios para que el proyecto avance en el orden supuesto para su realización.
- **Recursos:** personal y/o equipo necesario para completar el proyecto.
- **Costos:** rapidez con la que se llevarán a cabo las tareas y cómo se emplean los recursos (equipamiento y trabajadores). El costo de completar cada tarea incluye el costo de todos los recursos que trabajan en una tarea y de todos los gastos adicionales, como el costo de los materiales y los costos por uso.
- **Margen de demora:** tiempo que una tarea puede retrasarse sin demorar a ninguna otra. Se puede ver también como la cantidad de tiempo que puede retrasarse una tarea sin afectar a la fecha de fin del proyecto (margen de demora total).
- **Hito:** tarea sin duración que se utiliza para identificar sucesos significativos en la programación, como la finalización de una fase importante.

# Vocabulario

8

- **Método de Ruta Crítica:**

- Estudio de procesos, estructuras y sistemas mediante gráficos. Busca el camino dentro del grafo de mayor duración denominado camino crítico que condiciona el tiempo total del proyecto. Las tareas dentro de este camino son críticas y deben ser las más controladas. Y en caso de desear acelerarlo se debe comenzar por ellas.

- **Diagrama de Gantt:**

- Gráfico en 2 escalas, la horizontal mide el tiempo, la vertical los elementos que intervienen en la programación. Establece una relación cronológica entre cada elemento productor o tarea. Las subdivisiones horizontales del espacio en el gráfico representan tres cosas: transcurso de una unidad de tiempo, trabajo programado para ese intervalo y trabajo realizado.

# Pasos

9

- Definir el proyecto: Fecha comienzo y conectividad.
- Definir periodos laborales: Calendario laboral y jornada.
- Lista de tareas: Tareas, hitos y duración.
- Organizar tareas en fases.
- Programar tareas. Dependencia entre tareas
- Lista de recursos.
- Asignar recursos a tareas.
- Seguimiento:
  - Plan previsto.
  - Seguimiento:
    - Varias vías (Web, Access, ...).
    - Varios métodos (porcentual, trabajo realizado, trabajo por periodo).
    - Comprobar progreso respecto de fecha.

# Pasos

10

- **Informes:**

- Información general, como resúmenes del proyecto.
- Información acerca de las tareas, como los diagramas de Gantt.
- Información acerca del recurso, como la programación de cada recurso.
- Información acerca de los costos, como el costo de los recursos con presupuestos sobrepasados.
- Información acerca del seguimiento, como una lista de tareas que lleven retraso con respecto a la programación.

# Informes

11

PRESENTACIÓN	DESCRIPCIÓN
Diagrama de Grantt	Muestran tareas y duraciones en el tiempo. Usar para introducir y planificar una lista de tareas.
Diagrama de Pert	Un diagrama de red que muestra todas las tareas y sus relaciones.
Calendario	Muestra las tareas y sus duraciones. Se utiliza para mostrar las tareas programadas para una semana o semanas.
Hoja de Tareas	Una lista de tareas e información relacionada. Introducir y programar tareas en un formato de tipo hoja Calculo.
Formulario de Tareas	Se utiliza para introducir y editar la información acerca de una tarea determinada.
Formulario Detalles de Tareas	Se utiliza para revisar y editar el seguimiento detallado Y la programación de la formación de una tarea determinada.
Formulario Nombre de Tarea	Se utiliza para introducir y editar el nombre de la tarea y otra información acerca de la misma.

# Informes

12

PRESENTACIÓN	DESCRIPCIÓN
Diagrama de Pert de Tareas	Muestra las predecesoras y las sucesoras de una tarea.
Gantt de Retrasos	Muestran el retraso añadido a las tareas durante la Redistribución.
Gantt Detallado	Una lista de tareas e información relacionada y un diagrama de barras que muestran las barras de lo Planificado y lo programado para cada tarea.
Entrada de Tarea	Presentación combinada con el diagrama de Gantt encima y el formulario de tareas abajo.
Gráficos de Recursos	Muestra la asignación de recursos, los costos o las Horas Extras.
Hoja de Recursos	Una lista de recursos e información relacionada.
Uso de Recursos	Muestran la asignación, costo, información del trabajo de cada recurso de trabajo en el tiempo.
Formulario de Recursos	Se utiliza para introducir y editar la acerca de un recurso determinado.
Formulario Nombre Recurso	Un formulario simplificado que se utiliza para introducir y editar el nombre del los recursos.
Asignación de Recursos	Una presentación combinada que se utiliza para resolver las sobreasignaciones de recursos.

# Calendarios

113



# Calendarios

14

- Tres tipo de calendarios, y cada uno con una función determinada:
  - Calendario del Proyecto: Programación de trabajo predeterminada de todas las tareas de un proyecto.
  - Calendario Base: Project facilita tres calendarios base.
  - Calendario de recursos: Tiempos de trabajo de cada persona (u objeto) que trabajo en un proyecto. Usar este calendario para cambiar la programación de un solo recurso y los días de vacaciones.

# Definición de Calendarios

15

- Hay que definir los calendarios bajo los cuales nuestro proyecto se va a desarrollar. Franjas horarias, días, etc.
- Ir a Ficha **proyecto** y seleccionar **Cambiar tiempo de Trabajo**.
- **Cambiar tiempo de trabajo** permite definir los calendarios laborales.
- Existen tres calendarios predefinidos.
  - Cada calendario muestra las franjas horarias y días de trabajo.
- Si seleccionamos **opciones** (parte inferior), nos permite definir entre otros:
  - Franjas horarias, horas, etc.

Project - Project Reference

**HERRAMIENTAS DE DIAGRAMA DE GANTT**

ARCHIVO TAREA RECURSO CREAR UN INFORME PROYECTO VISTA FORMATO

Indicador de progreso: 0% Re: vie 13/03/15

ESCALA DE TIEMPO: Comienzo: vie 13/03/15

Modo de: Nombre de tarea:

Cambiar calendario laboral...

Para calendario: Estándar (Calendario del proyecto)

El calendario 'Estándar' es un calendario base.

Legends:

- Laborable
- No laborable
- Horas laborables modificadas
- En este calendario
- Día de excepción
- Semana laboral no predefinida

Haga clic en un día para ver sus períodos laborables: Períodos laborables del 13 marzo 2015: marzo 2015

L	M	M	J	V	S	D
1						
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

-9:00 a 13:00  
-15:00 a 19:00

Buscado en:  
Semana laboral predefinida  
del calendario 'Estándar'.

Excepciones Semanas laborales

Nombre	Comienzo	Fin

Aceptar Cancelar Opciones Ayuda

LISTO NUEVAS TAREAS PROGRAMADA MANUALMENTE

pdf Acceso a Internet

Project - Project Professional

FERRAMENTAS DE DIAGRAMA DE GANTT

Opciones de Project

**General**

**Programación** (selecciónada)

**Revisión**

**Guardar**

**Avanzado**

Personalizar cinta de opciones

Barras de herramientas de acceso rápido

Centro de confianza

Cambie opciones relacionadas con la programación, los calendarios y los cálculos.

Opciones de calendario para este proyecto: **Proyecto**

La semana comienza en: Lunes

El año fiscal comienza en: Enero

Usar el año inicial para la numeración de los años fiscales

Hora de comienzo predeterminada: 9:00

Hora de fin predeterminada: 19:00

Horas por día: 8

Horas por semana: 40

Días por mes: 20

Se asignan estos horarios a las tareas cuando se escribe una fecha de comienzo a fin sin especificar una hora. Si cambia esta configuración, es conveniente hacerla coincidir con el calendario del proyecto mediante el comando Cambiar tiempo de trabajo de la ficha Proyecto en la cinta.

Programación:

Mostrar mensajes de programación

Mostrar los unidades de asignación como: Porcentaje

Opciones de programación de este proyecto: **Proyecto**

Nuevas tareas creadas: Programado manualmente

Tareas programadas automáticamente programadas en la: Fecha de comienzo del proyecto

Mostrar duración en: Días

Mostrar trabajo en: Horas

Tipo de tarea predeterminada: Unidades fijas

Las tareas nuevas están condicionadas por el esfuerzo

Las tareas siempre respetan las fechas de restricción

Vincular automáticamente las tareas

Mostrar las tareas programadas que tienen duraciones estimadas

Aceptar Cancelar

Inicio sesión

Fin

30 mar '15 06 abr '15

L M M V S D L M J

100% NUEVAS TAREAS: PROGRAMADA MANUALMENTE

Proyecto - Project Professional

HERRAMIENTAS DE DIAGRAMA DE GANTT

ARCHIVO TAREA RECURSO CREAR UN INFORME PROYECTO VISTA FORMATO

Iniciar sesión

ESCALA DE TIEMPO

Carrera: Inicio 13/03/15 Fin 13/03/15

Detalles de [Predeterminado]

Establecer el periodo laborable para esta semana laboral.

Seleccionar días:

- Utilizar los períodos predefinidos del proyecto para estos días.
- Etiquetar días como periodo no laborable.
- Establecer día(s) con períodos laborables específicos.

	Domingo	Lunes	Martes	Miércoles	Jueves	Viernes	Sábado
Horas	08:00 - 13:00	08:00 - 13:00	08:00 - 13:00	08:00 - 13:00	08:00 - 13:00	08:00 - 13:00	08:00 - 13:00
Fin	13:00	13:00	13:00	13:00	13:00	13:00	13:00

Aceptar Cancelar

Agenda en un día para gerar sus períodos laborables: Períodos laborables del 13 marzo 2015.

marzo 2015

L	M	M	J	V	S	D
1						
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

+ 9:00 a 13:00  
+ 13:00 a 19:00

Basada en:  
Semana laboral predeterminada  
del calendario 'Estándar'.

Detalles... Eliminar

GANTT DE SEGUIMIENTO

Nombre: (Predeterminado) Carrera: MOD Fin: MOD

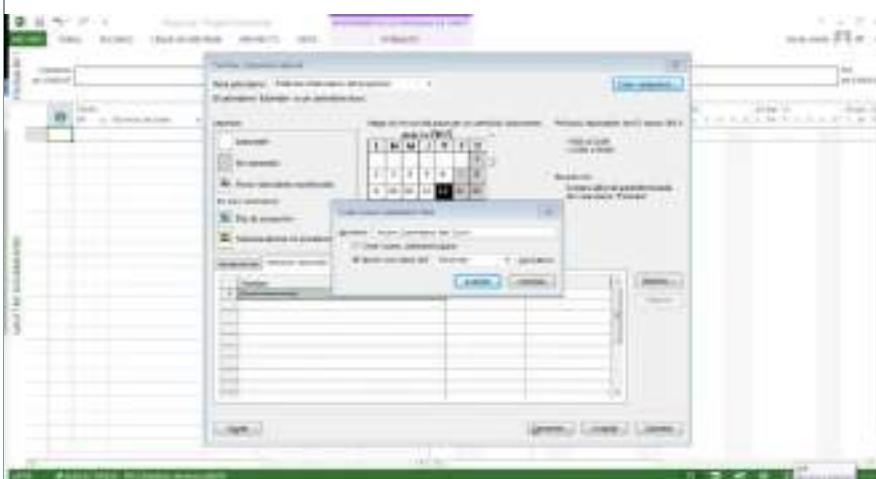
Opciones... Aceptar Cancelar

NUEVAS TAREAS | PROGRAMADA MANUALMENTE

# Crear un Nuevo Calendario

19

- Asignar un nombre.
- Posibilidad crear un calendario desde cero, o utilizar un nuevo calendario.



# Crear un Nuevo Calendario

20

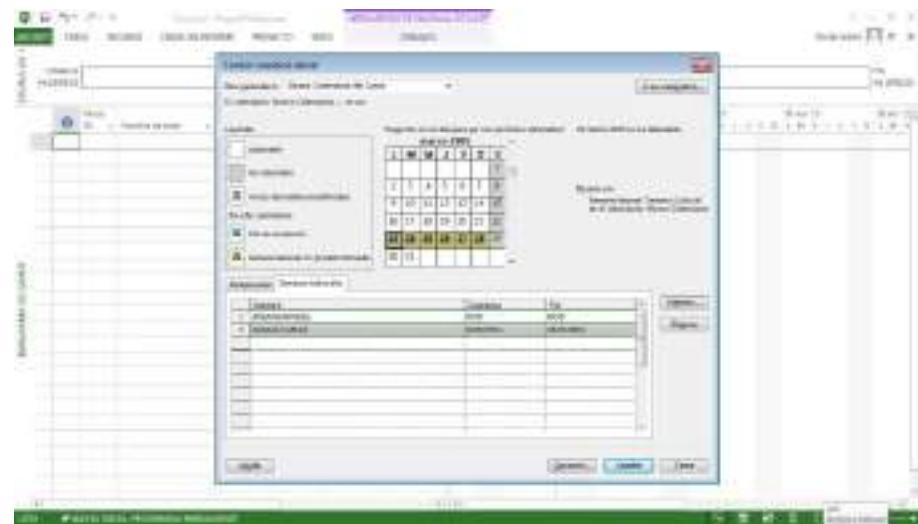
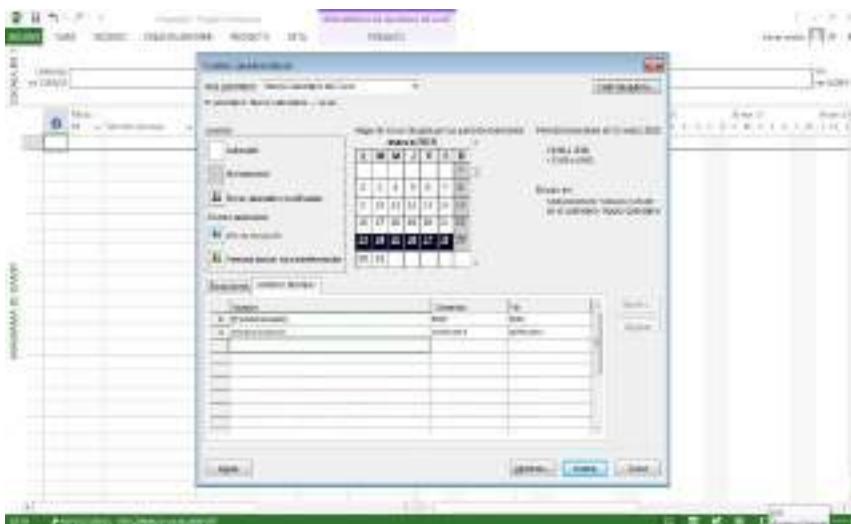
- Vamos a cambiar los horarios semanales.
  - Seleccionamos la ficha **semanas laborales**.
    - NOD comienzo y Fin, implica que utilizamos los mismos detalles para todos los días de todo el calendario.



# Crear un Nuevo Calendario

21

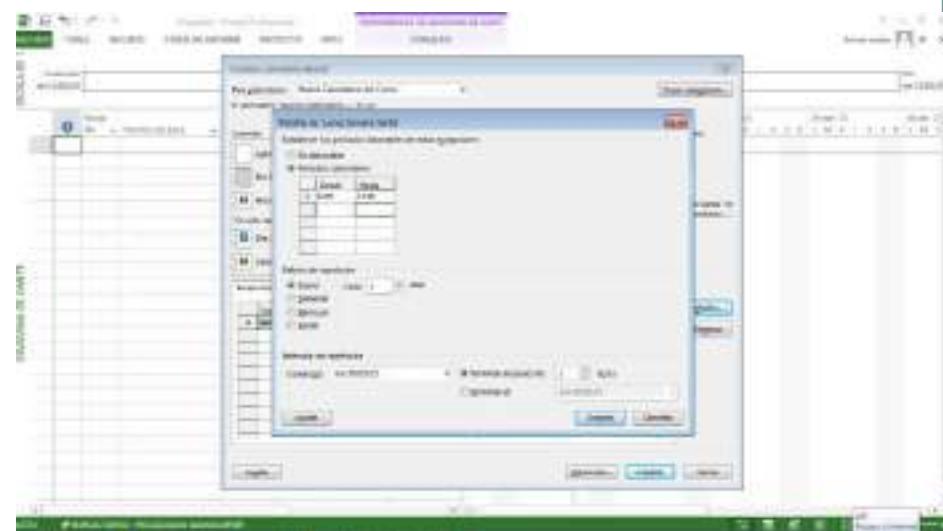
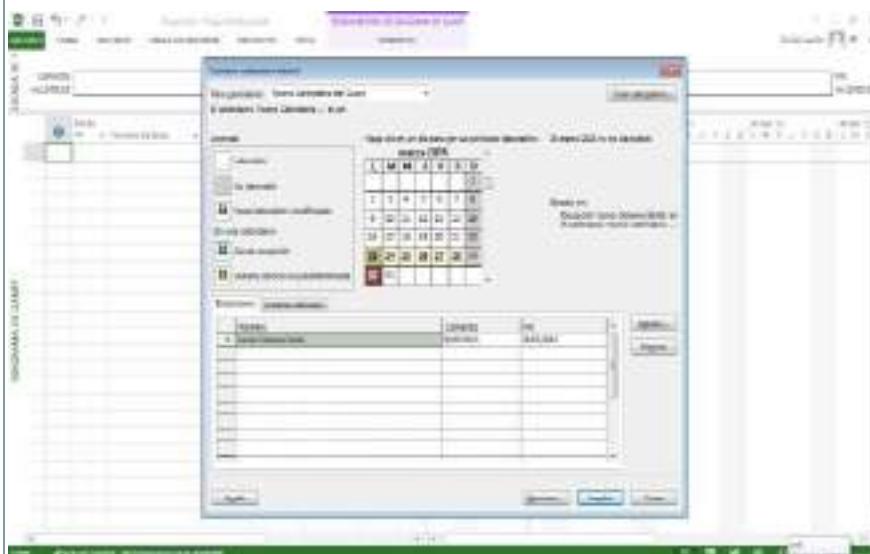
- Podemos definir una semana no laborable o con distinto horario. Por ejemplo podemos definir la Semana del 23 de Marzo de 2015 como no laborable.
  - Seleccionamos días.
  - Les damos nombre (Semana cultural).
  - Vamos a detalles.
  - Seleccionamos no laboral.



# Crear un Nueva Calendario

22

- Podemos crear excepciones para días concretos, los cuales pueden ser individuales o que se repitan en el tiempo.
  - Seleccionamos día. Por ejemplo 30 de Marzo de 2015, y el horario será de 8 a 12 horas.



Comienzo  
vía 13/03/15Fin  
vía 13/03/15

## Cambiar calendario laboral.

Para calendario: Nuevo Calendario del Curso

Crear calendario...

El calendario 'Nuevo Calendario ...' es un

## Leyenda:

- Laborable
- No laborable
- Horas laborables modificadas
- Día de excepción
- Semana laboral no predeterminada

Haga clic en un día para ver sus períodos laborables:

marzo 2015						
L	M	M	J	V	S	D
1						
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

Períodos laborables del 02 marzo 2015:

- 9:00 a 13:00
- 15:00 a 19:00

## Basado en:

Semana laboral predefinida  
del calendario 'Nuevo Calendario'

## Excepciones | Semanas laborales

Nombre	Comienzo	Fin
1 Lunes Semana Santa	30/03/2015	30/03/2015

Detalles...

Duplicar...

Ayuda

Opciones...

Aceptar

Cancelar



Información

Nuevo

Abrir

Guardar

Guardar como

Imprimir

Compartir

Exportar

Cerrar

Cuenta

Opciones

## Información



### Cuentas de Project Web App

No está conectado a Project Web App



### Organizar plantilla global

Mueve vistas, informes y otros elementos del proyecto entre archivos del proyecto y la plantilla global.

### Información del proyecto -

Fecha de comienzo	Hoy
Fecha de finalización	Hoy
Progresar a partir de	Comienzo
Fecha actual	Hoy
Fecha de estado	Hoy
Calendario del proyecto	Nuevo Calendario...
Prioridad	500

**Asignamos a nuestro proyecto el nuevo calendario**

## Información

[Nuevo](#)[Alert](#)[Guardar](#)[Guardar como](#)[Imprime](#)[Compartir](#)[Exportar](#)[Cuenta](#)[Cuenta](#)[Opciones](#)

### Cuentas de Project Web App

No está conectado a Project Web App

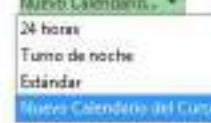


### Organizar plantilla global

Mueve vistas, informes y otros elementos del proyecto entre archivos del proyecto y la plantilla global.

### Información del proyecto +

Fecha de comienzo	Hoy
Fecha de finalización	Hoy
Programar a partir de	Comienzo
Fecha actual	Hoy
Fecha de estado	Hoy
Calendario del proyecto	<a href="#">Nuevo Calendario...</a>
Prioridad	



**Asignamos a nuestro proyecto el nuevo calendario**

# Tareas

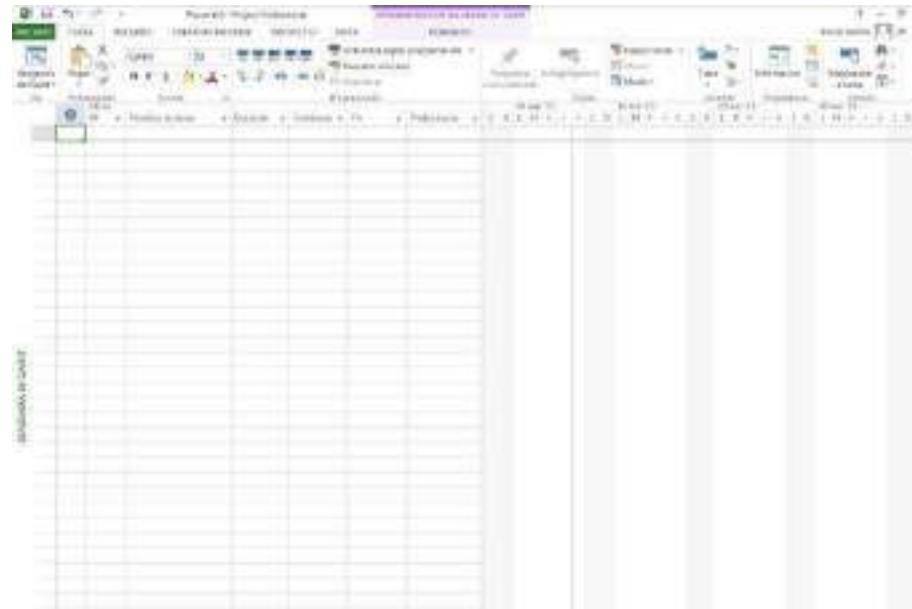
26



# Introducir Tareas

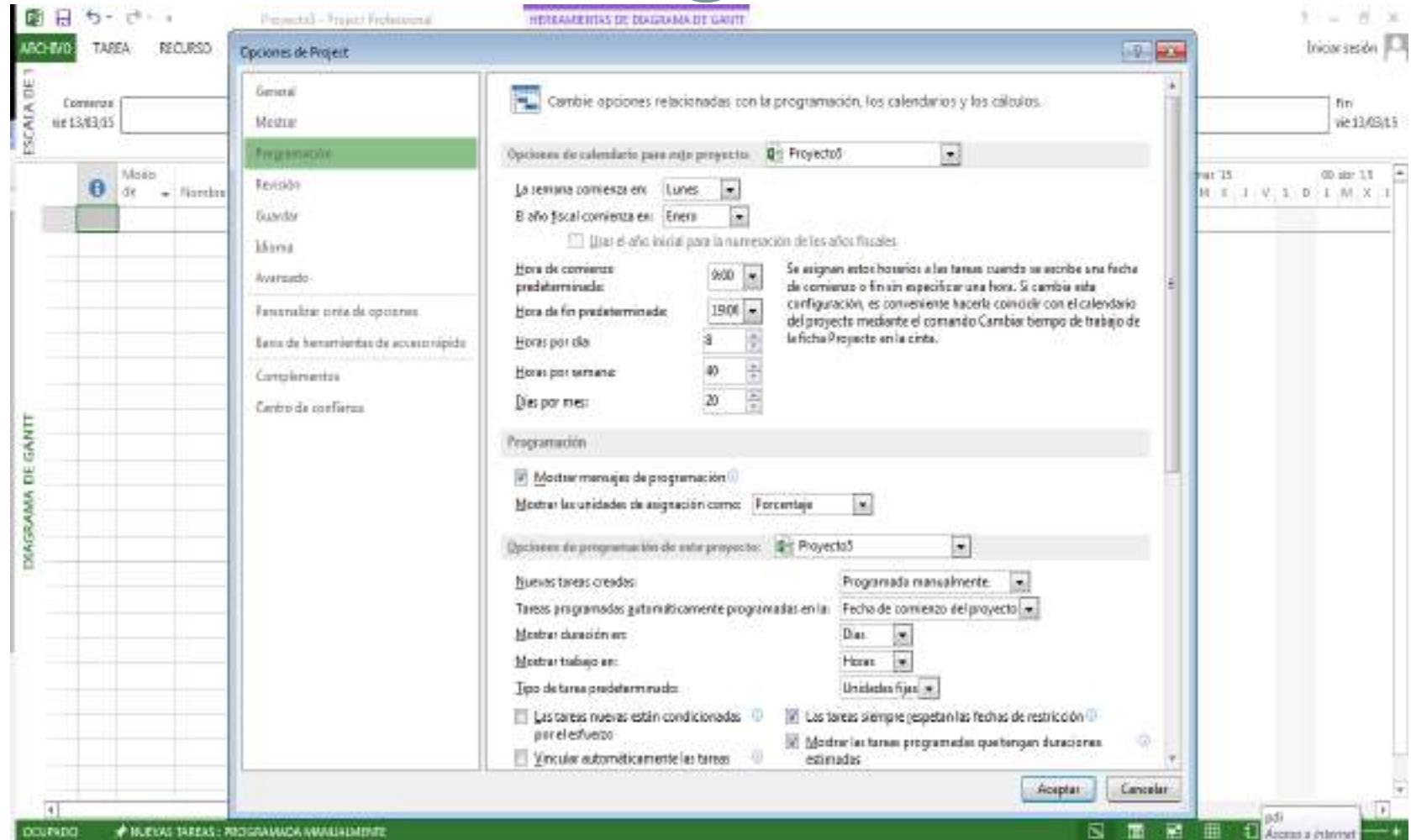
27

- Ficha **Tarea** y hacer Click en **Diagrama de Gannt**.
- Aparece una cuadricula donde ingresar las tareas.
- Las tareas pueden introducirse de manera automática o manual.
  - En modo automático ajusta las fechas del resto de tareas de manera automática.



# Parametrizar Microsoft Project

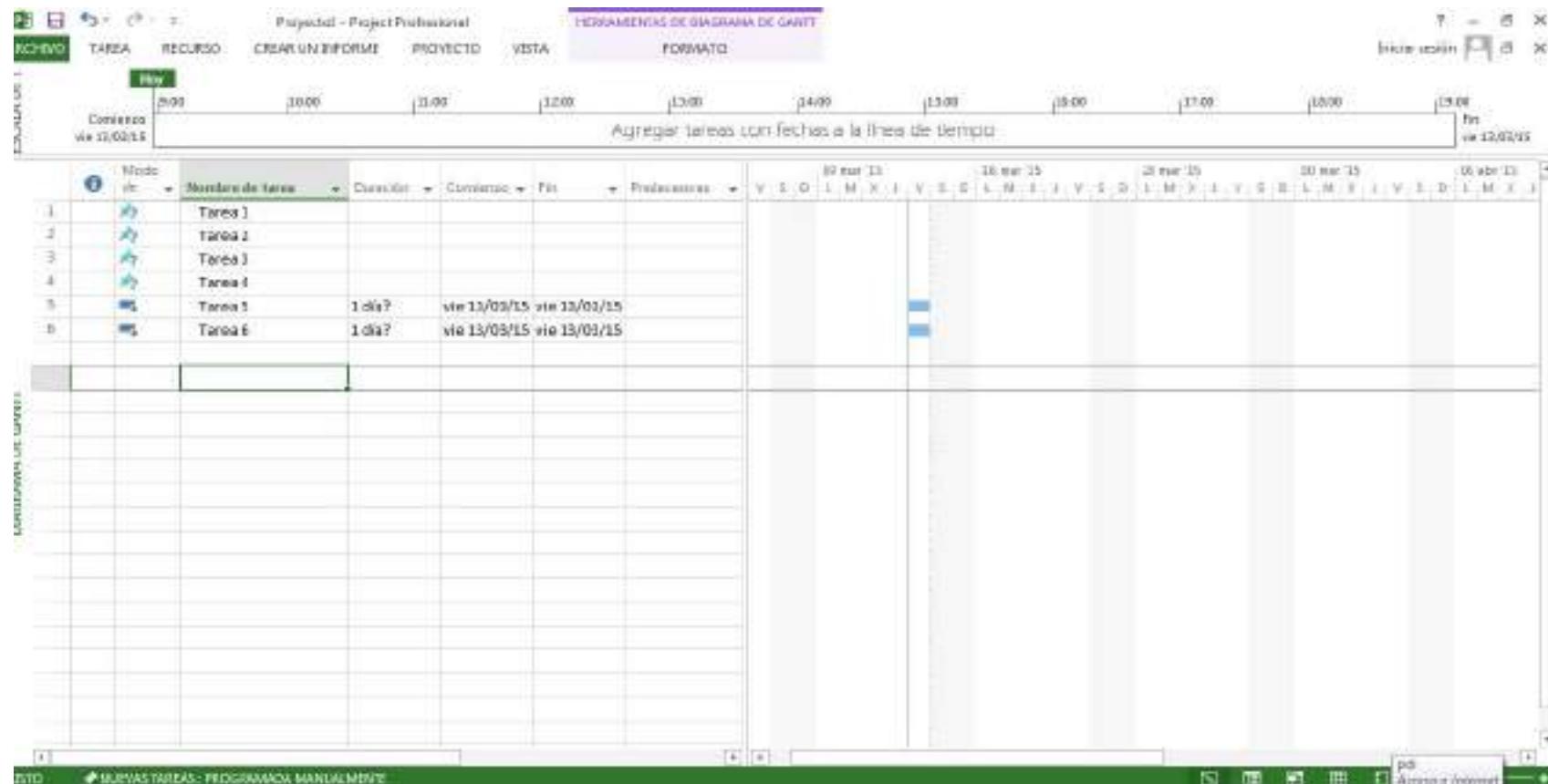
28



# Introducción de tareas

29

- Las tareas 1 a 4 son manuales y el resto automáticas.



# Duración de Tareas

30

- La duración de una tarea puede ser:

- Las siglas ms significan meses.
- Las siglas s significa semanas.
- Las siglas d significa días.
- Las siglas h significa horas.
- Las siglas m significa minutos.
- Las siglas m? significa duración estimada.
- Las siglas 2dt significa tiempo transcurrido y no se ajusta al calendario asociado al proyecto, sino a jornadas de 24 horas.
- También se pueden indicar duraciones con formas del tipo:
  - Lo que cueste.

# Duración de Tareas

31

Projecto - Project Professional

ARCHIVO TARSA RECURSO CREAR UN INFORME PROYECTO VISTA HERRAMIENTAS DE DIAGRAMA DE GANTT FORMATO Iniciación

ESCALA DE TIEMPO Hoy 9:00 10:00 11:00 12:00 13:00 14:00 15:00 16:00 17:00 18:00 19:00 Comienzo Fin viernes 13/03/15 viernes 13/03/15 Agregar tareas con fechas a la línea de tiempo

	Modo de	Nombre de tarea	Duración	Comienza	Fin	Predecesoras	Non	mar'15	26 mar'15	27 mar'15	28 mar'15
1	■	Tarea 1	lo que quieras				recurrir	M X J V S D L	M X J V S D L	M X J V S D L	
2	■	Tarea 2									
3	■	Tarea 3									
4	■	Tarea 4									

DIAGRAMA DE GANTT

LISTO NUEVAS TAREAS : PROGRAMADA MANUALMENTE

pdf Acceso Internet

# Inicio Aproximado

32

Proyecto5 - Project Professional

HERMANITAS DE DIAGRAMA DE GANTT

ARCHIVO TAREA RECURSO CREAR UN INFORME PROYECTO VISTA FORMATO

Hoy  
Comienzo: vie 13/03/15 Fin: vie 13/03/15

Agregar tareas con fechas a la línea de tiempo

ESCALA DE TIEMPO

	Modo de	Nombre de tarea	Duración	Comienzo	Fin	Préde-	M	F	J	V	S	D	L	M	X	J	V	S	D
1.		Tarea 1	/o que cueste	La semana de la constitución															
2.		Tarea 2																	
3.		Tarea 3																	
4.		Tarea 4																	

DIAGRAMA DE GANTT

LISTO NUEVAS TAREAS : PROGRAMADA ANUALMENTE

pdf Acceso e Internet

# Hitos

33

- Son momentos especiales en un proyecto, donde queremos establecer unos puntos de control, para saber cuando se llega a un determinado punto.
- Tienen una duración de 0 días normalmente, aunque una tarea puede ser un hito y tener una duración mayor.
  - Para ello seleccionar la tarea, botón derecho, información, avanzada y hacer click en tarea es un hito.
- Pueden ser trasladados a la escala de tiempos para ver donde se ubican.

# Hitos

34

Project 2013 - Project Professional

ARCHIVO TARDA RECURSO CREAR UN INFORME PROYECTO VISTA FORMATO HERRAMIENTAS DE DIAGRAMA DE GANTT Iniciar sesión ? - & X

ESCALA DE TIEMPO Hoy lal 14/03 dom 15/03 lun 16/03 mar 17/03 mié 18/03 jue 19/03 vie 20/03 sáb 21/03 dom 22/03 lun 23/03 mar 24/03 mié 25/03 Fin mié 25/03/15

Agregar tareas con fechas a la línea de tiempo

	Modo de	Nombrar tarea	Duración	Comienzo	Fin	Predecesoras	Nom recur	09 mar'15	10 mar'15	11 mar'15	12 mar'15	13 mar'15	14 mar'15	15 mar'15	
1	■	1 FASE 1	9 días	vie 13/03/15	mié 25/03/15										
2	■	1.1 Tarea 1	2 días	vie 13/03/15	lun 16/03/15										
3	■	1.2 Tarea 2	3 días	mar 17/03/15	jue 19/03/15	2									
4	■	1.3 Tarea 3	4 días	vie 20/03/15	mié 25/03/15	3									
5	■	1.4 FIN DE FASE 1	0 días	mié 25/03/15	mié 25/03/15	4									
6	■	2 FASE 2	7 días	vie 13/03/15	lun 23/03/15										
7	■	2.1 Tarea 4	3 días	vie 13/03/15	mar 17/03/15										
8	■	2.2 Tarea 5	4 días	mié 18/03/15	lun 23/03/15	7									
9	■	2.3 FIN DE FASE 2	0 días	lun 23/03/15	lun 23/03/15	8									

Diagrama de Gantt

LISTO NUEVAS TAREAS : PROGRAMADA MANUALMENTE pdf Acessos Interiores

The Gantt chart displays the timeline for the project. Phase 1 tasks (1.1 to 1.4) are scheduled from March 13 to 25. Phase 2 tasks (2.1 to 2.3) are scheduled from March 13 to 23. All tasks start on March 13 and end on their respective dates.

# Escala de Tiempos

35

Proyecto5 - Project Professional

HERMANAS DE DIAGRAMA DE GANTT

ARCHIVO TARJA RECURSO CREAR UN INFORME PROYECTO VISTA FORMATO

Hoy

Comienzo: vie 13/03/15 Fin: vie 03/04/15

ESCALA DE TIEMPO

FIN DE FASE 1  
vie 25/03/15

FIN DE FASE 2  
vie 03/04/15

1. Tarea 1 9 días comienza: vie 13/03/15 fin: mié 25/03/15

1.1 Tarea 1 2 días comienza: vie 13/03/15 fin: lun 16/03/15

1.2 Tarea 2 3 días comienza: mar 17/03/15 fin: jue 19/03/15

1.3 Tarea 3 4 días comienza: vie 20/03/15 fin: mié 25/03/15

1.4 FIN DE FASE 1 0 días comienza: mié 25/03/15 fin: mié 25/03/15

2. Tarea 4 3 días comienza: jue 26/03/15 fin: lun 30/03/15

2.1 Tarea 4 3 días comienza: jue 26/03/15 fin: lun 30/03/15

2.2 Tarea 5 4 días comienza: mar 31/03/15 fin: vie 03/04/15

2.3 FIN DE FASE 2 0 días comienza: vie 03/04/15 fin: vie 03/04/15

ESQUEMA DE GANTT

The Gantt chart displays the project timeline from March 13 to April 3. Task 1 (9 days) starts on March 13 and ends on March 25. Task 2 (7 days) starts on March 26 and ends on April 2. Task 1.1 (2 days) and Task 1.2 (3 days) are part of Task 1. Task 1.3 (4 days) follows Task 1.4 (0 days). Task 2.1 (3 days) and Task 2.2 (4 days) are part of Task 2. Task 2.3 (0 days) follows Task 2.4 (0 days).

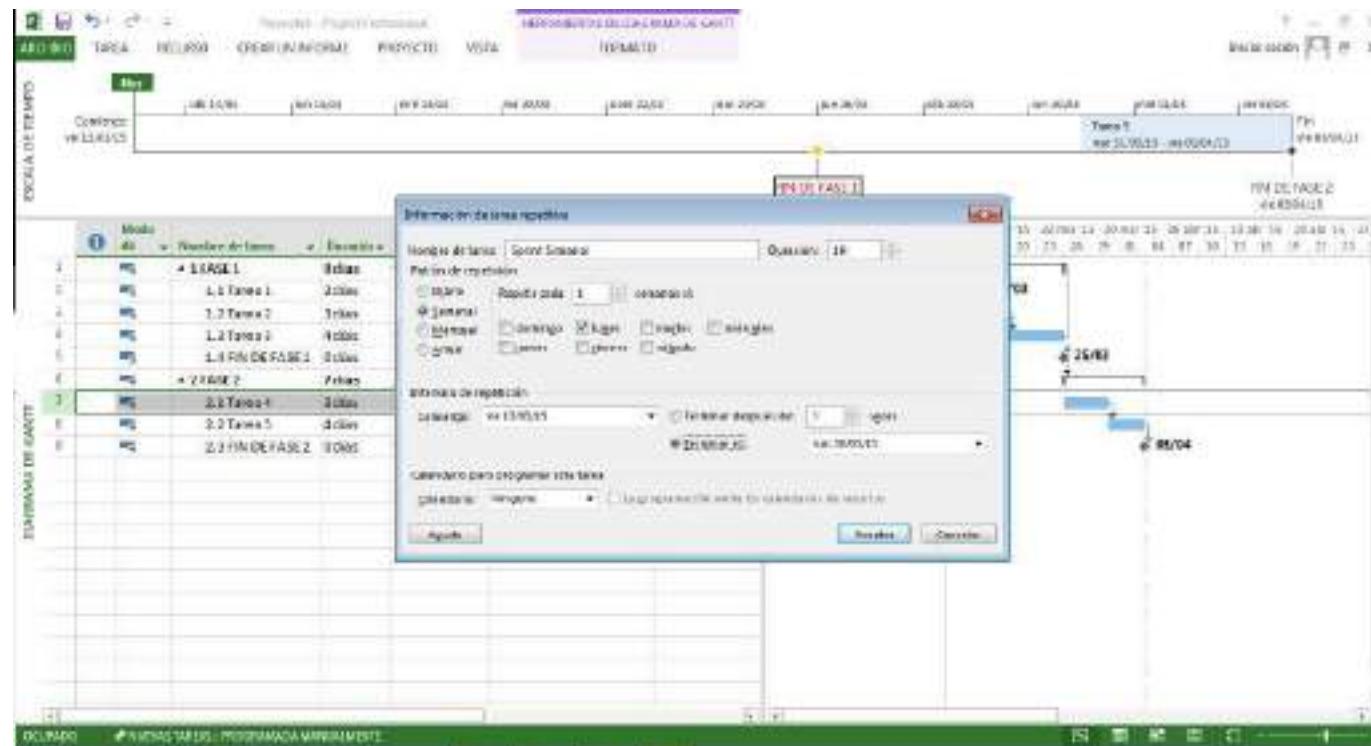
Ítem	Modo de	Nombre de tarea	Duración	Comienza	Fin	Predecesores
1.	■	1 FASE 1	9 días	vie 13/03/15	mié 25/03/15	
2.	■	1.1 Tarea 1	2 días	vie 13/03/15	lun 16/03/15	
3.	■	1.2 Tarea 2	3 días	mar 17/03/15	jue 19/03/15	2
4.	■	1.3 Tarea 3	4 días	vie 20/03/15	mié 25/03/15	3
5.	■	1.4 FIN DE FASE 1	0 días	mié 25/03/15	mié 25/03/15	4
6.	■	2 FASE 2	7 días	jue 26/03/15	vie 03/04/15	1
7.	■	2.1 Tarea 4	3 días	jue 26/03/15	lun 30/03/15	
8.	■	2.2 Tarea 5	4 días	mar 31/03/15	vie 03/04/15	7
9.	■	2.3 FIN DE FASE 2	0 días	vie 03/04/15	vie 03/04/15	8

(LISTO) NUEVAS TAREAS / PROGRAMADA MANUALMENTE

# Tareas Repetitivas

36

- Tareas que se repiten durante un tiempo en mismo día. Por ejemplo, sprint los lunes una hora.



# Tareas Repetitivas

37

Project 2013 - Project Professional

HERMAMIENTAS DE DIAGRAMA DE GANTT

ARCHIVO TAREA RECURSO CREAR UN INFORME PROYECTO Vista FORMATO Iniciar sesión

May

Comienzo: vie 13/03/15 Fin: vie 03/04/15

FIN DE FASE 1  
vie 25/03/15

FIN DE FASE 2  
vie 03/04/15

ESCALA DE TIEMPO

Modo de vista: Nombre de tareas Duración Comienzo Fin Predecesoras

	Nombre de tareas	Duración	Comienzo	Fin	Predecesoras
1	+ 1 FASE 1 9 días	9 días	vie 13/03/15	mié 25/03/15	
2	1.1 Tarea 1 2 días	2 días	vie 13/03/15	lun 16/03/15	
3	1.2 Tarea 2 3 días	3 días	mar 17/03/15	jue 19/03/15	2
4	1.3 Tarea 3 4 días	4 días	vie 20/03/15	mié 25/03/15	3
5	1.4 FIN DE FASE 1 0 días	0 días	mié 25/03/15	mié 25/03/15	4
6	+ 2 FASE 2 15 días	15 días	lun 16/03/15	vie 03/04/15	1
7	+ 2.1 Sprint Semanal 10,33 días	10,33 días	lun 16/03/15	lun 30/03/15	
8	2.1.1 Sprint Sem 1 hora	1 hora	lun 16/03/15	lun 16/03/15	
9	2.1.2 Sprint Sem 1 hora	1 hora	lun 23/03/15	lun 23/03/15	
10	2.1.3 Sprint Sem 1 hora	1 hora	lun 30/03/15	lun 30/03/15	
11	2.2 Tarea 4 3 días	3 días	jue 26/03/15	lun 30/03/15	
12	2.3 Tarea 5 4 días	4 días	mar 31/03/15	vie 03/04/15	11
13	2.4 FIN DE FASE 2 0 días	0 días	vie 03/04/15	vie 03/04/15	12

DIAGRAMA DE GANTT

The Gantt chart displays the project timeline from March 13 to April 3, 2015. Task 1 (1 FASE 1) starts on March 13 and ends on March 25. Task 2 (1.1 Tarea 1) follows Task 1 and ends on March 16. Task 3 (1.2 Tarea 2) follows Task 2 and ends on March 19. Task 4 (1.3 Tarea 3) follows Task 3 and ends on March 25. Task 5 (1.4 FIN DE FASE 1) follows Task 4 and ends on March 25. Task 6 (2 FASE 2) follows Task 5 and ends on April 3. Task 7 (2.1 Sprint Semanal) follows Task 6 and ends on March 30. Task 8 (2.1.1 Sprint Sem 1 hora) follows Task 7 and ends on March 16. Task 9 (2.1.2 Sprint Sem 1 hora) follows Task 8 and ends on March 23. Task 10 (2.1.3 Sprint Sem 1 hora) follows Task 9 and ends on March 30. Task 11 (2.2 Tarea 4) follows Task 10 and ends on March 30. Task 12 (2.3 Tarea 5) follows Task 11 and ends on April 3. Task 13 (2.4 FIN DE FASE 2) follows Task 12 and ends on April 3.

LISTO NUEVAS TAREAS: PROGRAMADA MANUALEMENTE

# Estructurar Tareas

38

- **Primer forma:**
  - Seleccionar las tareas.
  - Ficha tarea. Seleccionar resumen.
- **Segunda Forma:**
  - Insertar tarea.
  - Sangrar.
- Utilizar la opción EDT para numerar.

# Estructurar Tareas

39

Project 2013 - Project Professional

HERMAMIENTAS DE DIAGRAMA DE GANTT

ARCHIVO TAREA RECURSO CREAR UN INFORME PROYECTO Vista FORMATO Iniciar sesión

May

Comienzo: vie 13/03/15 Fin: vie 03/04/15

FIN DE FASE 1  
vie 25/03/15

FIN DE FASE 2  
vie 03/04/15

ESCALA DE TIEMPO

Modo de vista: Nombre de tareas Duración Comienzo Fin Predecesoras

	Nombre de tareas	Duración	Comienzo	Fin	Predecesoras
1	+ 1 FASE 1 9 días	9 días	vie 13/03/15	mié 25/03/15	
2	1.1 Tarea 1 2 días	2 días	vie 13/03/15	lun 16/03/15	
3	1.2 Tarea 2 3 días	3 días	mar 17/03/15	jue 19/03/15	2
4	1.3 Tarea 3 4 días	4 días	vie 20/03/15	mié 25/03/15	3
5	1.4 FIN DE FASE 1 0 días	0 días	mié 25/03/15	mié 25/03/15	4
6	+ 2 FASE 2 15 días	15 días	lun 16/03/15	vie 03/04/15	1
7	+ 2.1 Sprint Semanal 10,33 días	10,33 días	lun 16/03/15	lun 30/03/15	
8	2.1.1 Sprint Sem 1 hora	1 hora	lun 16/03/15	lun 16/03/15	
9	2.1.2 Sprint Sem 1 hora	1 hora	lun 23/03/15	lun 23/03/15	
10	2.1.3 Sprint Sem 1 hora	1 hora	lun 30/03/15	lun 30/03/15	
11	2.2 Tarea 4 3 días	3 días	jue 26/03/15	lun 30/03/15	
12	2.3 Tarea 5 4 días	4 días	mar 31/03/15	vie 03/04/15	11
13	2.4 FIN DE FASE 2 0 días	0 días	vie 03/04/15	vie 03/04/15	12

DIAGRAMA DE GANTT

LISTO NUEVAS TAREAS: PROGRAMADA MANUALEMENTE.



# Proyecto de Ejemplo

41

Ejemplo.mpp - Project Professional

TIERRAMENTAS DE DIAGRAMA DE GANTT

ARCHIVO TAREA RECURSO CEAR UN INFORME PROYECTO VISTA FORMATO

ESCALA DE TIEMPO

May

16 mar '15 23 mar '15 30 mar '15 06 abr '15 13 abr '15 20 abr '15 27 abr '15 04 may '15 11 may '15 18 may '15

Comienzo dom 15/03/15 Fin jue 21/05/15

Agregar tareas con fechas a la línea de tiempo

Modo de Número de tarea Duración Comienzo Fin Predecesores Número recurso

1 1 PROYECTO DE EJEMPLO 48,25 días dom 15/03/1 jue 21/05/15

2 1.1 Primer Grupo de Tareas 38,5 días dom 15/03/1 jue 30/04/15

3 1.1.1 Inicio de Tareas Grupo 1 0 días dom 15/03/1 dom 15/03/1

4 1.1.2 Tarea1 2 sem. lun 16/03/15 vie 27/03/15 3

5 1.1.3 Tarea2 3 días lun 30/03/15 mié 01/04/15 4

6 1.1.4 Tarea3 4 horas jue 02/04/15 jue 02/04/15 5

7 1.1.5 Tarea4 1 ms jue 02/04/15 jue 30/04/15 6

8 1.1.6 Final de Tareas Grupo 1 0 días jue 30/04/15 jue 30/04/15 7

9 1.2 Segundo Grupo de Tareas 14,75 días jue 30/04/15 jue 21/05/15 8

10 1.2.1 Inicio de Tareas Grupo 2 0 días jue 30/04/15 jue 30/04/15 8

11 1.2.2 Tarea5 4 días jue 30/04/15 mié 06/05/15 10

12 1.2.3 Tarea6 2 sem. mié 06/05/15 mié 20/05/15 11

13 1.2.4 Tarea7 6 horas mié 20/05/15 jue 21/05/15 12

14 1.2.5 Final de Tareas Grupo 2 0 días jue 21/05/15 jue 21/05/15 13

15 1.3 Reuniones de Coordinación 25,38 días lun 16/03/15 lun 20/04/15

DIAGRAMA DE GANTT

15/03 30/04 21/05

ESTADO NUEVAS TAREAS: PROGRAMADA ANTERIORMENTE

# Crear Hoja de Recurso

42

- Ir a **vistas** y hacer click en **hoja de recurso**.

The screenshot shows the Microsoft Project application interface. The title bar reads "Proyecto2 - Project Professional". The ribbon menu is visible with tabs like ARCO-IRIDI, TAREA, RECURSO, CREAR UN INFORME, PROYECTO, VISTA, and FORMATO. A yellow highlighted section of the ribbon says "INTERFAZ DE HOJA DE RECURSOS". The main workspace is titled "HOJA DE RECURSOS" and contains a table with one row of data:

	Nombre del	Tipo	Días para la	Salvo que	Grupo	Capacidad	Tasa	Tasa hora	Precio/h	Acumula	Calendario	Costo	Unidad ejecución
1	José María	Trabajo	20		AB	100%	8,00 €/hora	0,00 €/hora	8,00 €	Acumula	Calendario	8,00 €	Unidad ejecución

The left sidebar has sections for "ESTRUCTURA DEL PROYECTO" and "HOJA DE RECURSOS". The bottom navigation bar includes buttons for "LEER", "NUEVA TAREA", "PROGRAMAR MANUALMENTE", and other standard project management tools.

# Crear Hoja de Recursos

443

- **Campos:**

- Nombre de recurso.
- Tipo: costo, material o trabajo.
- Etiqueta: es para los tipo material.
- Iniciales: Una forma de identificarlo de manera más simple.
- Grupo: Indica al grupo al que pertenece, p.e: programadores.
- Capacidad: Disponibilidad del recurso.
- Tasa: precio estándar por hora.
- Tasa horas: precio hora extra.
- Costo/Unidad: costo por uso, por ejemplo, maquinaria que requiere un coste adicional por usarla, aparcarla, etc.
- Acumular: Forma de pago, inicio, final, prorrateo.
- Calendario: Calendario con el que trabaja este recurso: 24 horas, turno de noche, estándar, definido por el usuario.

# Recursos Tipo Trabajo

44

- Los recursos de trabajo son el equipo y las personas que completan tareas dedicando tiempo o trabajo a las mismas.
- Cuando se crea un recurso de trabajo, se hace en un contexto de tiempo. Puede definir un nuevo recurso de trabajo indicando cuánto tiempo (o la capacidad máxima) tendrá que dedicar al proyecto en su conjunto: tiempo completo (100%), tiempo parcial (por ejemplo, 50%) o múltiple (por ejemplo 300% para tres programadores que trabajen en el mismo proyecto)

# Recursos Tipo Material

45

- Los recursos materiales son suministros, existencias u otros artículos consumibles que se utilizan para completar las tareas de un proyecto.
- Algunos ejemplos de recursos materiales son el paper, toner, cables de red.
- Cuando crea un recurso material, establece que es un material y no un recurso de trabajo.
- Asimismo, define la etiqueta del material, o unidad de medida, para ese material. Por ejemplo, metros, cajas.

# Recursos Tipo Coste

46

- Son los recursos que nos permiten contratar, arrendar o comprar recursos que pueden ser humanos, instalaciones, equipos o materiales.
- El valor o coste del recurso de indica, cuando se asigna a una tarea.

# Asignar Recursos a las Tareas

47

- **Tenemos 4 pasos:**

1. Tener en cuenta la disponibilidad de los recursos cuando se estima la duración de las tareas.
  - Los recursos deben considerarse durante la construcción del cronograma, no después.
  - Estimar calculando el número de horas/hombre para terminar la tarea. No hacerlo en días de trabajo.
2. Crear y organizar todos los recursos.
  - Utilizar la hoja de recursos para tener organizados los recursos.
  - Recursos con nombres claros y el tipo de recurso.
    - Recursos de trabajo: Para categorizar a las personas. Utilizar el nombre de la persona o la subcontrata en trabajos de gran tamaño.
    - Recursos materiales: Para categorizar los materiales: madera, clavos, ordenadores, etc.
    - Recursos de Costo: Para categorizar recursos con una tasa de costo definida: alquiler de una máquina, etc.
3. Asignar cada tarea a un recurso.
4. Verificar la asignación de recursos en la vista gráfica de recursos.

# Asignar Recursos a Tarea

48

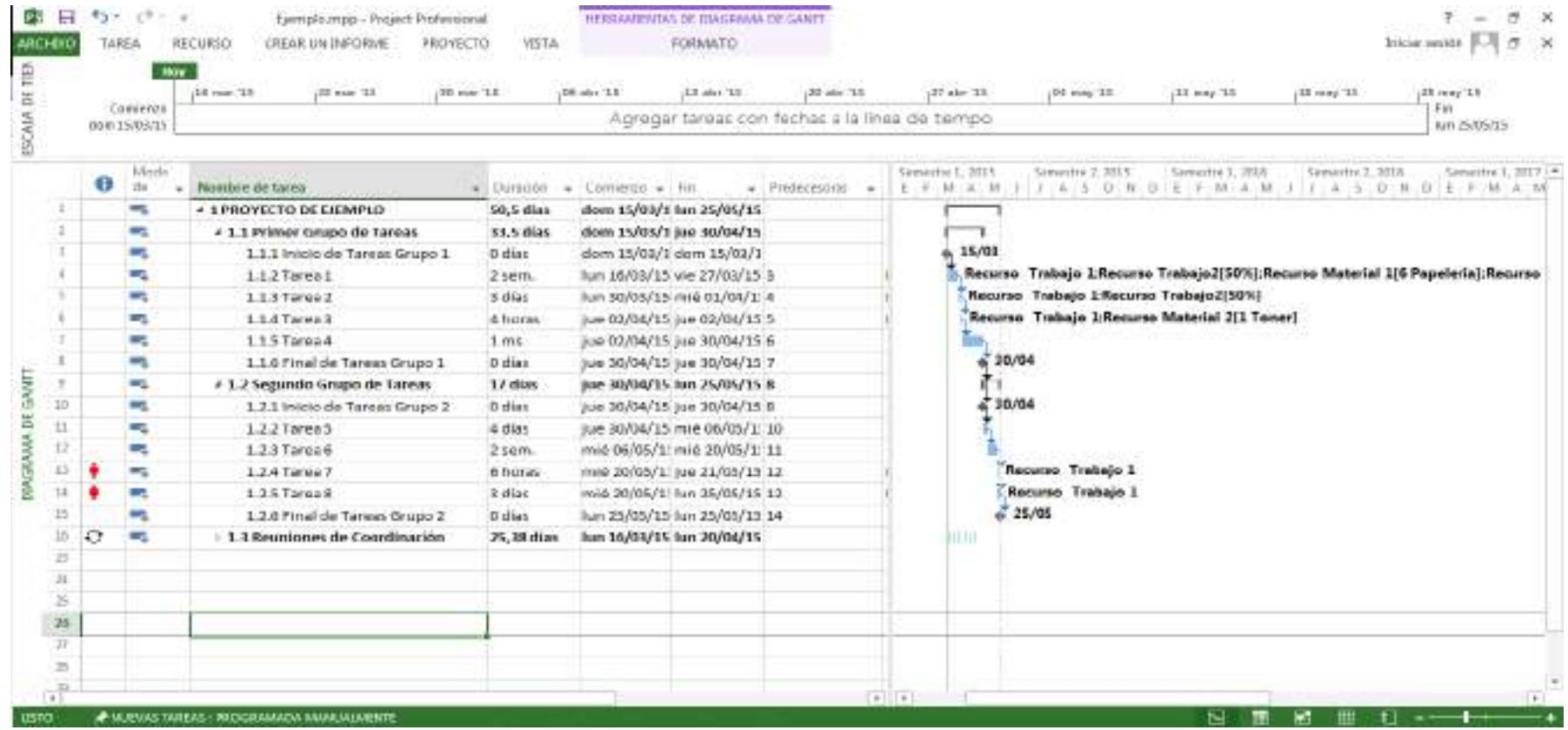
- Ficha de Recurso. Asignar Recurso.

The screenshot shows the Microsoft Project application interface. The ribbon at the top has tabs: ARCHIVO, TAREA, RECURSO, CREAR UN INFORME, PROYECTO, Vista, and FORMATO. The 'RECURSO' tab is currently selected. Below the ribbon is a Gantt chart showing tasks over time. A specific task, 'Tarea 1', is selected and highlighted in green. An 'Información de la tarea' dialog box is open over the Gantt chart. The dialog box has tabs: General, Predecesores, Recursos, Avanzado, Notas, and Comentarios. The 'Recursos' tab is selected. Inside the dialog, the task name 'Tarea 1' is listed, along with its duration of '2 días'. There is one resource assignment listed: 'Recurso Trabajo 1' assigned at 100% cost. The Gantt chart shows the task starting on '16 mar '15' and ending on '09/05/15'. The status bar at the bottom indicates 'OCUPADO' and 'NUEVAS TAREAS: PROGRAMADA MANUALMENTE'.

# Tareas Sobre Asignadas

49

- El recurso de trabajo 1 ha sido asignado a las tareas 7 y 8, con un uso del 100%.





# Informes

# *Informes del Estado del Proyecto*

51

- Un informe es una representación gráfica de los datos de las tablas que componen el proyecto.
- Representaciones gráficas para interpretar de manera más rápida los datos.
- Obtener información del estado del proyecto.
- Por ejemplo, informes que indiquen cuando el coste previsto y el coste real .

# Acceso a informes

52

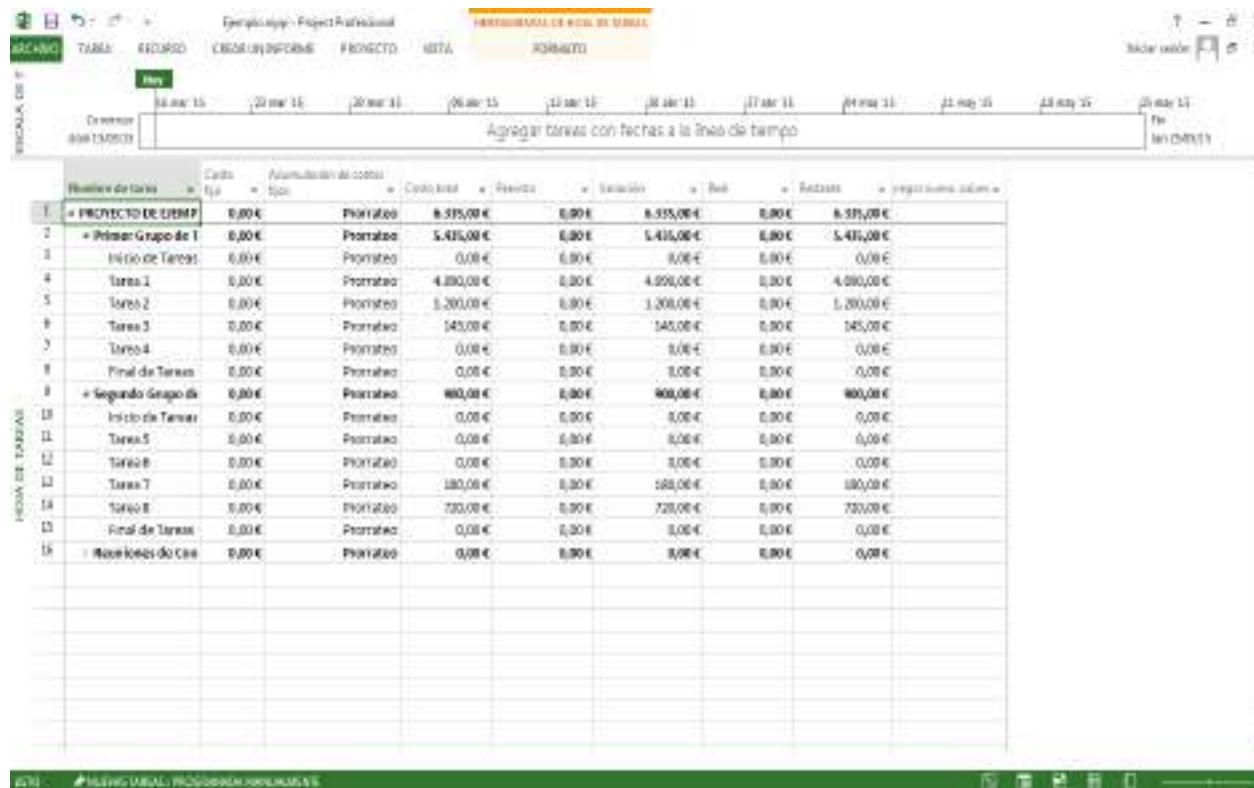
- Ficha Vista + Otras vistas + más vistas

	Duración	Comenzó	Fin	Priorización	Nombre de los recursos	Ajustar nuevo ítem
1	50,5 días	dom 15/03/15	jue 25/05/15			
2	33,5 días	dom 15/03/15	jue 30/04/15			
3	0 días	dom 15/03/15	dom 15/03/15			
4	2 sem.	lun 16/03/15	vie 27/03/15	3	Recurso Trabajo 1:Re	
5	3 días	lun 30/03/15	mié 01/04/15	4	Recurso Trabajo 1:Re	
6	4 horas	jue 02/04/15	jue 02/04/15	5	Recurso Trabajo 1:Re	
7	Tarea 4	1 ms	jue 02/04/15	jue 30/04/15	6	
8	Final de Tareas Grupo 1	0 días	jue 30/04/15	jue 30/04/15	7	
9	Segundo Grupo de Tareas	17 días	jue 30/04/15	jue 25/05/15	8	
10	Inicio de Tareas Grupo 2	0 días	jue 30/04/15	jue 30/04/15	9	
11	Tarea 5	4 días	jue 30/04/15	mié 06/05/15	10	
12	Tarea 6	2 sem.	mié 06/05/15	mié 20/05/15	11	
13	Tarea 7	6 horas	mié 20/05/15	jue 21/05/15	12	Recurso Trabajo 1:Re
14	Tarea 8	3 días	mié 20/05/15	jue 25/05/15	12	Recurso Trabajo 1:Re
15	Final de Tareas Grupo 2	0 días	jue 25/05/15	jue 25/05/15	14	
16	Reuniones de Coordinación	25,38 días	jue 16/03/15	jue 20/04/15		

# Informe

53

- Por ejemplo podemos ver el costo del proyecto:
  - Vista + otras vistas + más vistas + hoja de tareas + << aplicar>> + vista + tablas + costo



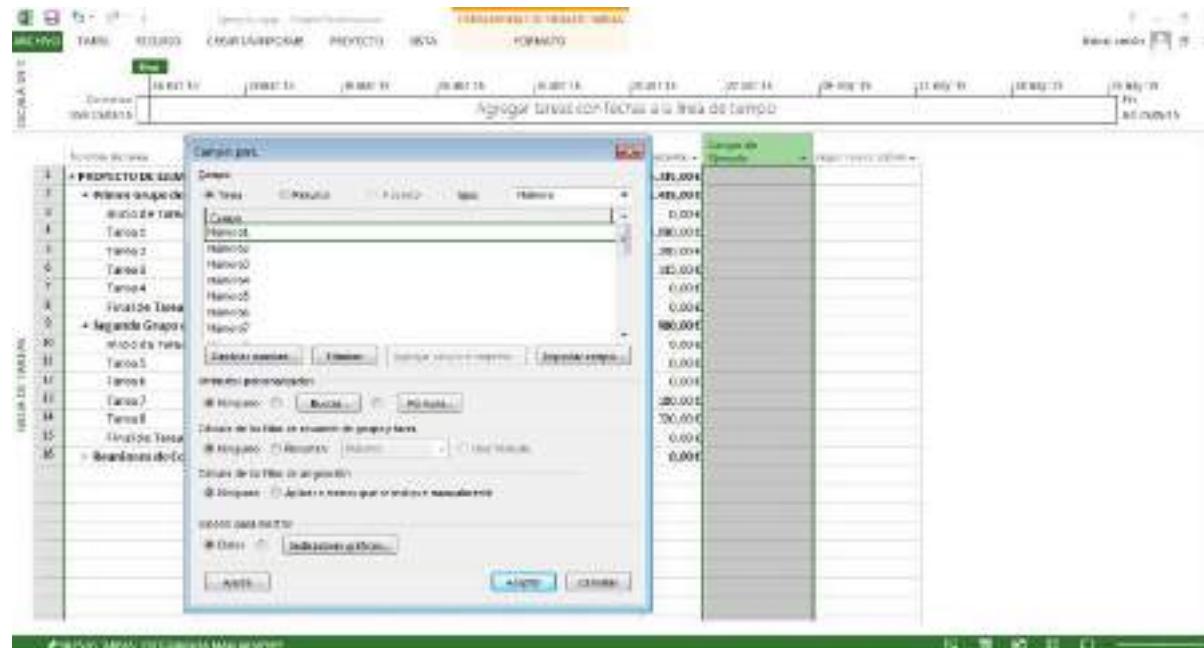
The screenshot shows the Microsoft Project application window with the 'Cost' table selected. The table displays the cost accumulation for various tasks, including overhead costs and resource costs.

Resumen de tarea	Costo fijo + tipo	Acumulación del costo					
		= Costo total	= Fijo	= Variable	= Total	= Recursos	= Impresión saliente
1 a PROYECTO DE TIEMPO	0,00 €	Prioritario	8.375,00 €	0,00 €	8.375,00 €	0,00 €	8.375,00 €
2 + Primer Grupo de T.	0,00 €	Prioritario	5.475,00 €	0,00 €	5.475,00 €	0,00 €	5.475,00 €
3 Inicio de Tareas	0,00 €	Prioritario	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €
4 Tarea 1	0,00 €	Prioritario	4.000,00 €	0,00 €	4.000,00 €	0,00 €	4.000,00 €
5 Tarea 2	0,00 €	Prioritario	1.200,00 €	0,00 €	1.200,00 €	0,00 €	1.200,00 €
6 Tarea 3	0,00 €	Prioritario	145,00 €	0,00 €	145,00 €	0,00 €	145,00 €
7 Tarea 4	0,00 €	Prioritario	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €
8 Final de Tareas	0,00 €	Prioritario	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €
9 + Segundo Grupo de T.	0,00 €	Prioritario	800,00 €	0,00 €	800,00 €	0,00 €	800,00 €
10 Inicio de Tareas	0,00 €	Prioritario	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €
11 Tarea 5	0,00 €	Prioritario	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €
12 Tarea 6	0,00 €	Prioritario	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €
13 Tarea 7	0,00 €	Prioritario	180,00 €	0,00 €	180,00 €	0,00 €	180,00 €
14 Tarea 8	0,00 €	Prioritario	720,00 €	0,00 €	720,00 €	0,00 €	720,00 €
15 Final de Tareas	0,00 €	Prioritario	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €
16 Recorridos de casa	0,00 €	Prioritario	0,00 €	0,00 €	0,00 €	0,00 €	0,00 €

# Informes Personalizados

54

- Se puede añadir una columna a la figura anterior y utilizar la opción de campo personalizado para añadir una formula de calculo.



# Más Información

55

- Manual Project.
- Videos en Youtube. Los siguiente videos son adecuados.

<https://www.youtube.com/watch?v=bimyMkKgyDw>

# Introducción a la Ingeniería del Software



## TEMA 6.1: ARQUITECTURAS SOFTWARE

Grado en Ingeniería Informática  
Grado en Ingeniería del Software  
Grado en Ingeniería de Computadores



# Índice de contenidos

2

- Concepto de Arquitectura Software
- Estilos de Arquitectura Software

# Índice de contenidos

3

- Concepto de Arquitectura Software
- Estilos de Arquitectura Software

# Necesidad de la arquitectura software

4

*"A medida que el tamaño y la complejidad de los sistemas de software aumenta, el problema del diseño va más allá de los algoritmos y las estructuras de datos de la implementación: el diseño y la especificación de la estructura general del sistema surge como un nuevo tipo de problema ... Este es la arquitectura de software a nivel de diseño."*

Garland y Shaw, "An introduction to Software Architecture", 1994

# Concepto de arquitectura software

5

- **Arquitectura software:** descripción de los subsistemas de un sistema software, sus propiedades y las relaciones entre ellos.
- **Definiciones alternativas:**
  - *"Estructura de los componentes de un programa o sistema, sus interrelaciones, y los principios y reglas que gobiernan su diseño y evolución en el tiempo."* Garlan y Perry, 1995
  - *"Estructura o estructuras de un sistema, lo que incluye sus componentes de software, las propiedades observables de dichos componentes y las relaciones entre ellos."* Bass, Clements y Kazman, 1998
  - *"Para mí el término arquitectura conlleva la noción de los elementos que forman el núcleo del sistema, las piezas que son difíciles de cambiar. Los cimientos sobre los que se edifica el resto."* Martin Fowler, 2001

# Importancia de la arquitectura

6

- ¿Por qué es necesario desarrollar un modelo arquitectónico?
  - Facilita la comprensión a cualquier miembro del equipo.
  - Permite que cada miembro pueda trabajar en los subsistemas de forma individual.
  - Prepara al sistema para su extensión.
  - Facilita la reutilización del sistema.

# Elementos de una arquitectura software

7

- **Componentes**

- Elementos computacionales donde se realiza el “trabajo” (subsistemas software)
- Pueden ser de grano grueso (e.g. un servidor web) o fino (e.g. un módulo)

- **Conectores**

- Comunican componentes
- Pueden ser explícitos (e.g. invocación) o implícitos (e.g. evento)

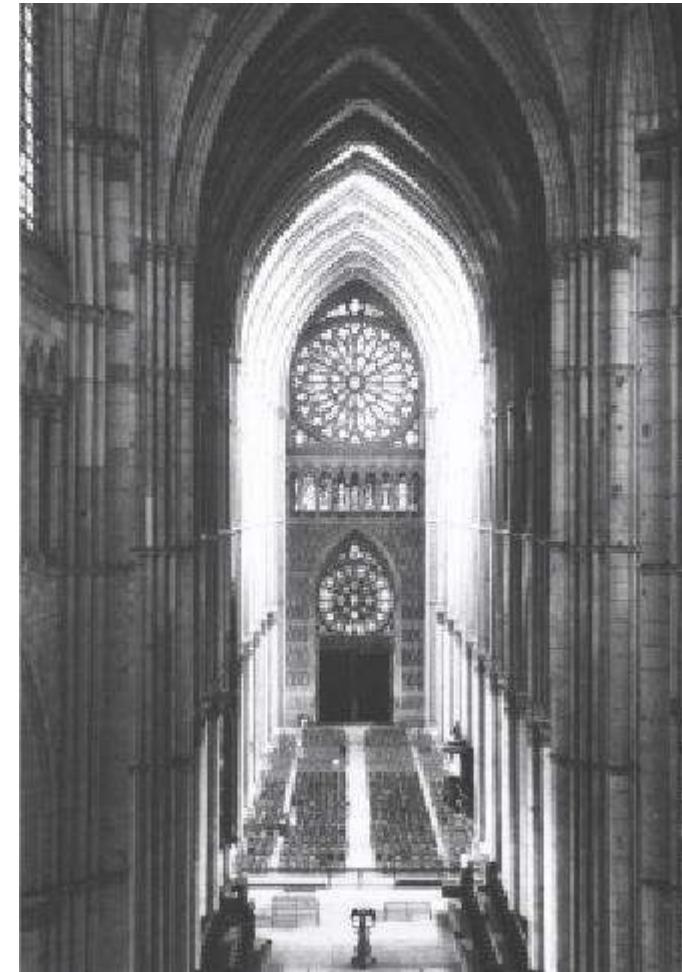
- **Configuración**

- Disposición concreta de los componentes y conectores que forman una arquitectura software

# ¿Qué caracteriza a la buena arquitectura?

8

- *La catedral de Reims*  
*(Estilo gótico, siglo XIII)*
- Ocho generaciones de constructores
- Micro-arquitecturas que contribuyen a la arquitectura global
- Sigue un **estilo** (patrón)



# Estilo o patrón arquitectónico

9

- Expresa la **organización estructural** para sistemas software.
- Cada estilo arquitectónico define la organización general en base a:
  - El tipo de componentes y sus responsabilidades
    - ✖ capas, servidores, bases de datos, componentes software...
  - El tipo de conectores
    - ✖ tuberías, invocaciones, difusión de eventos...
  - La configuración adecuada
    - ✖ Cómo se conectan los componentes entre sí mediante conectores
- Decidir el estilo arquitectónico apropiado para un sistema no es trivial

# Integridad conceptual

10

- Una buena arquitectura sigue un estilo o patrón arquitectónico que asegura la **integridad conceptual**:

*El patrón general de diseño de un sistema se refleja en cada parte del mismo*

“conceptual integrity is the **most important** consideration in system design”

Brooks, The Mythical Man Month, 1975

“the **same set of laws** determines the structure of a city; a building; or a single room”

Alexander, A Pattern Language, 1977

# Índice de contenidos

11

- Concepto de Arquitectura Software
- Estilos de Arquitectura Software

# Ejemplos de estilos arquitectónicos

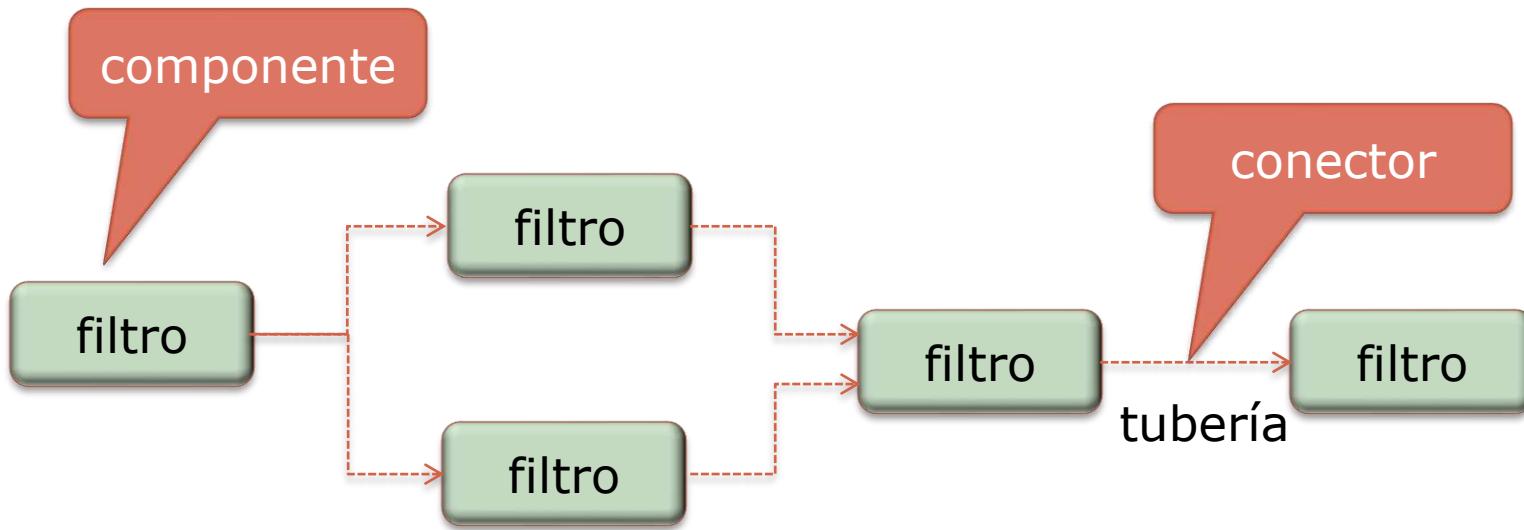
12

- Basados en flujos de datos
  - Tuberías y filtros
  - MapReduce
- Basados en llamada y retorno
  - Funcional
  - Orientado a objetos
- Arquitectura de capas
- Cliente-Servidor
- Modelo-Vista-Controlador.

# Tuberías y filtros (1)

13

- Dirigido por el flujo de datos (*dataflow*)
  - Los datos se procesan incrementalmente conforme llegan
  - Se puede generar la salida antes de consumir toda la entrada



# Tuberías y filtros (2)

14

- **Componentes: filtros**

- Leer flujos de entrada
- Transforman localmente los datos
- Escriben flujos de salida
- Son independientes
  - ✖ no tienen estado compartido
  - ✖ se desconocen entre sí

- **Conectores: tuberías**

- Flujos de datos (e.g. buffer FIFO, socket, etc.)

# Ejemplos de tuberías y filtros

15

- Shell de Unix

- `$ ls -l | grep "Aug"`

```
-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02  
-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07  
-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro  
-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros
```

- ✖ El comando “ls” muestra en formato largo los archivos contenidos en una dirección determinada y a continuación, con esa lista el comando “grep” filtra los que tienen coincidencia con “Aug”

- `$ ps -ef | grep httpd | wc -l`

```
727
```

- ✖ El comando “ps” lista los procesos activos del sistema operativo, “grep” selecciona aquellos con la coincidencia “httpd” y cuenta las líneas de estos ficheros, devolviendo ese valor.

- Composición en lenguajes funcionales (p.ej. Haskell)
- Cauce de un procesador, fases de un compilador, etc.

# Ventajas de tuberías y filtros

16

- **Semántica compositiva:** el comportamiento global es la composición de los comportamientos de los filtros
- **Reutilización:** dos filtros existentes se pueden conectar si soportan el tipo de datos adecuado
- **Mantenimiento:** sustitución de filtros por otros (más eficientes, ...)
- **Ejecución paralela o distribuida**

# Inconvenientes de tuberías y filtros

17

- No es apropiado para **procesamiento interactivo**, es decir, con participación del usuario
- Coste de la transformación de los datos para leerlos y escribirlos en las tuberías
- **Gestión de errores complicada**: ¿qué pasa si falla un filtro intermedio?
- Compartir un **estado global** es difícil

# MapReduce (1)

18

- Introducido por Google para el **procesamiento distribuido masivo sobre clusters o grids**
  - Implementaciones libres alternativas (**Hadoop**)
- Inspirado en programación funcional:
  - **map**: aplica una función sobre todos los componentes de una lista
  - **reduce**: extrae un resultado con los componentes de una lista
- API disponible en varios lenguajes
  - API oculta detalles complicados (balanceo de carga, paralelización automática, gestión de errores, etc.)



# MapReduce (2)

19

- **Paralelismo masivo:** cluster de Yahoo! para ejecutar Hadoop.



# Idea clave de MapReduce

20

- **Problema:** contar las apariciones de una palabra en un conjunto de documentos
- Solución con tuberías y filtros
  - \$ `cat *.docs | grep palabra | wc -l`  
Concatenar documentos, seleccionar la palabra y contarlas
- Idea clave: grep se puede **paralelizar**
  - aplica grep en paralelo:  
`counts = map (grep palabra) *.docs`
  - acumula resultados:  
`reduce (+) 0 counts`

# Esquema de MapReduce

21

- MapReduce trabaja con pares (**key, value**)
  1. map:  $(k, v) \mapsto [(k', v')]$   
El par de entrada dará lugar a una lista de pares
  2. Las salidas de los map se agrupan por claves ( $k'$ ,  $[v']$ ) y se pasan a reduce
  3. reduce:  $(k', [v']) \mapsto [v'']$   
El par (clave, lista) generará una nueva lista
- El esquema es estable, para cada problema se decide qué hacen map y reduce

# Diagrama de MapReduce

22

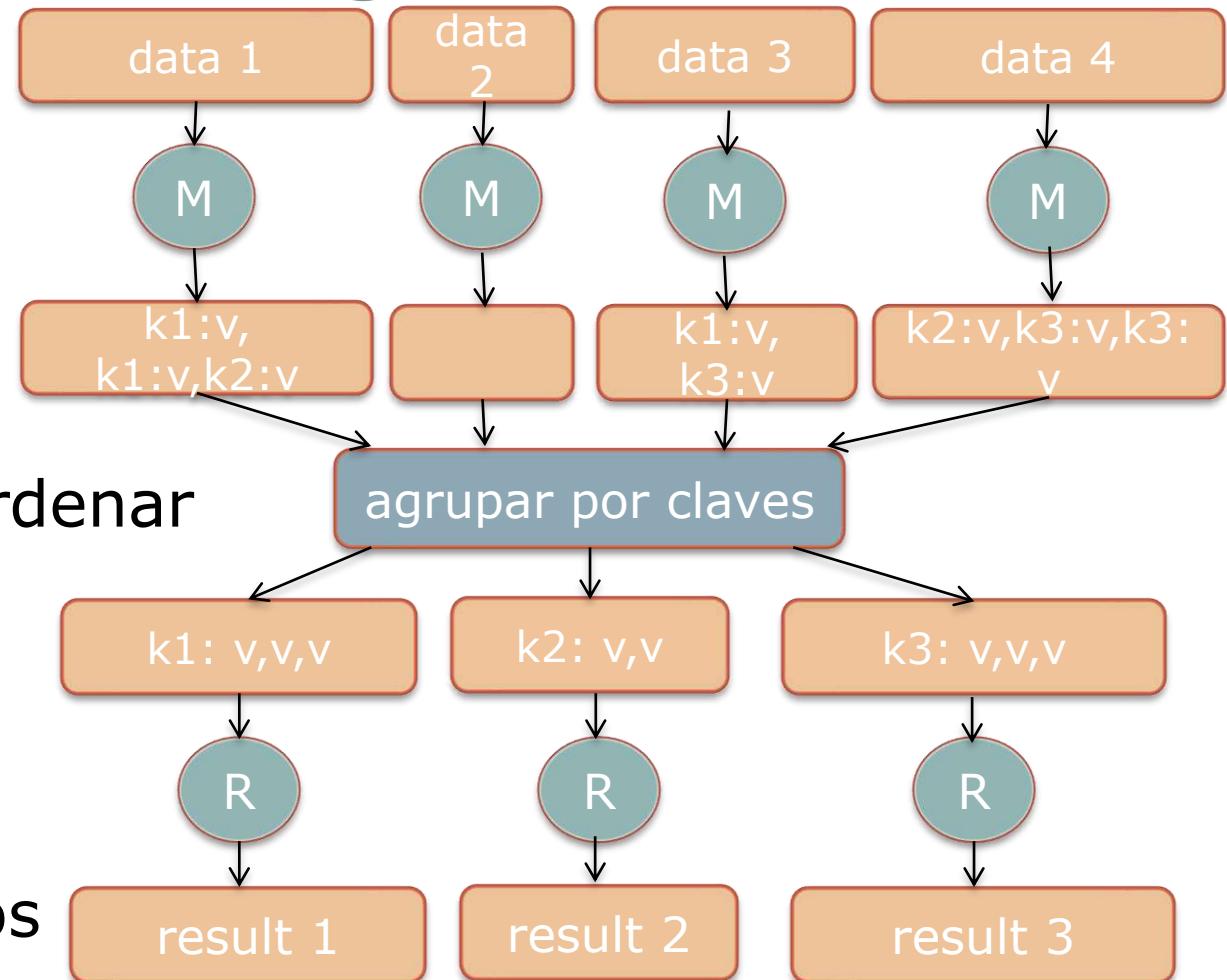
1. leer datos

2. map

3. agrupar y ordenar

4. reduce

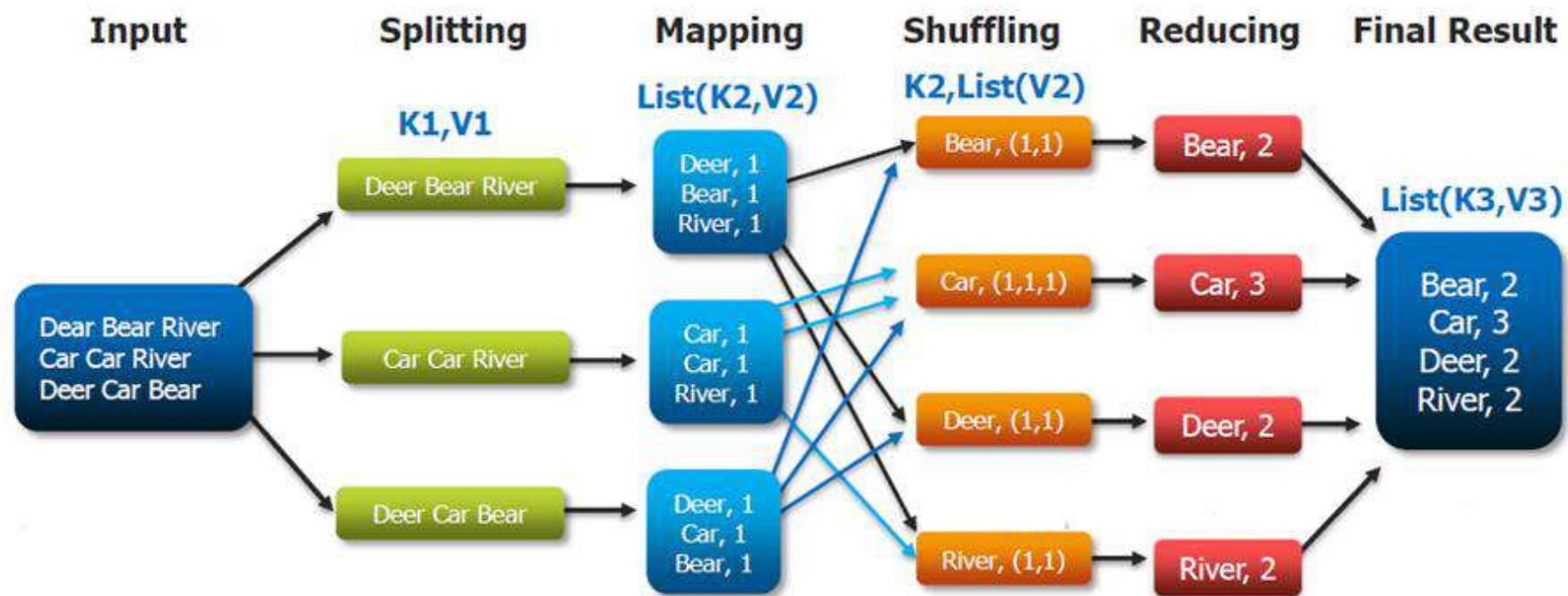
5. escribir datos



# Ejemplo

23

## The Overall MapReduce Word Count Process



# Otro ejemplo de MapReduce

24

- **Temperatura máxima:** Partimos de un conjunto de ficheros con columnas para el nombre de la ciudad y su temperatura, tomada en distintos momentos de un periodo de tiempo determinado.

- **Aplicando Map:**

(Madrid, 20), (Barcelona, 23), (Bilbao, 17), (Valencia, 24),  
(Málaga, 27), (Bilbao, 19), (Madrid, 22), (Valencia, 22)

- **La reagrupación obtendría:**

(Madrid, [20,22]), (Barcelona, [23]), (Bilbao, [17,19]),  
(Valencia, [22,24]), (Málaga, [27])

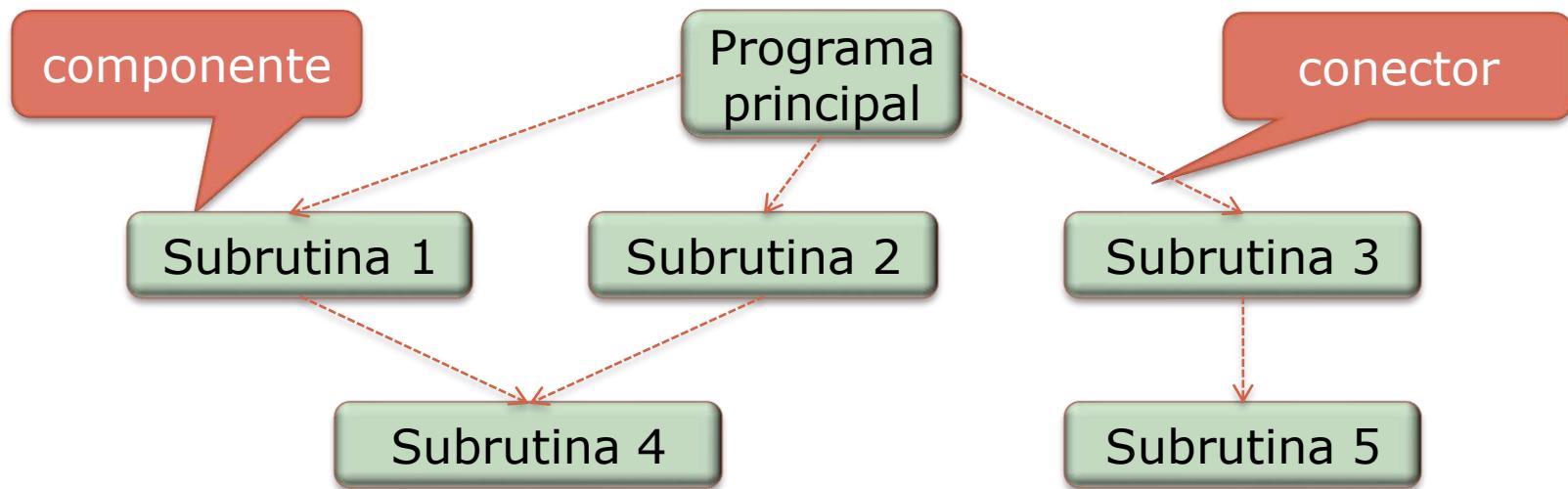
- **Aplicamos reduce para obtener la T<sup>a</sup> máxima:**

(Madrid, 22), (Barcelona, 23), (Bilbao, 19), (Valencia, 24),  
(Málaga, 27)

# Llamada y retorno funcional (1)

25

- Basado en la descomposición funcional del sistema
  - Las subrutinas corresponden a las tareas a realizar
  - Se combinan según el interfaz y el flujo de control



# Llamada y retorno funcional (2)

26

- **Componentes: subrutinas**

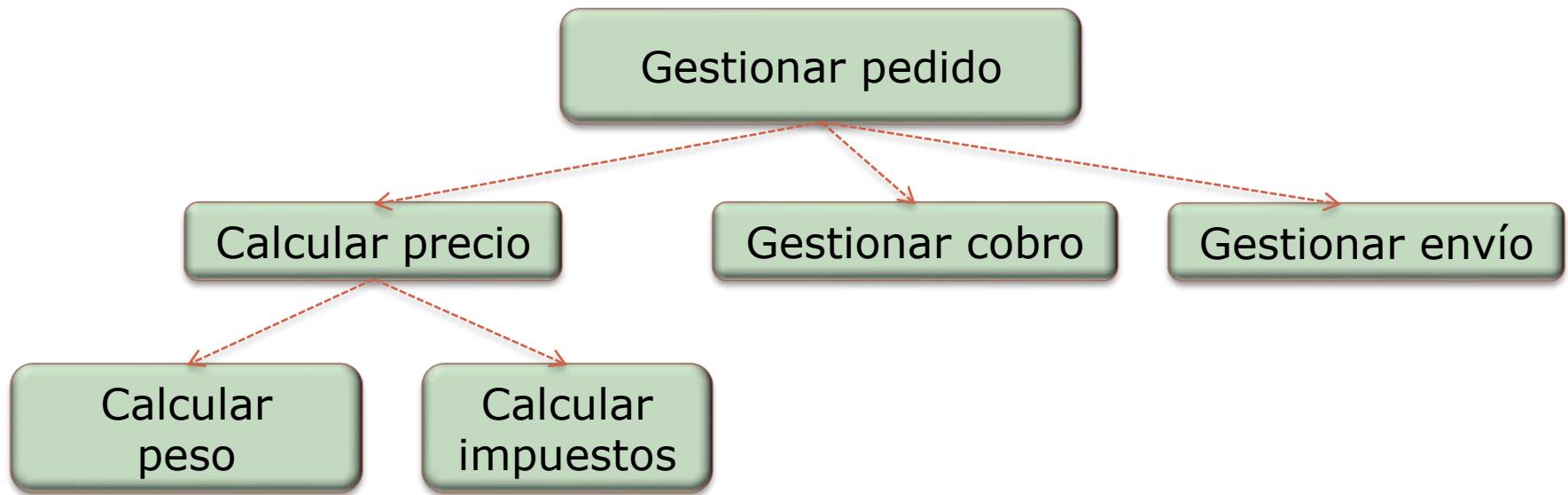
- Implementan tareas o subtareas
- Ofrecen una interfaz al exterior (argumentos)
- Combinadas a través del flujo de control

- **Conectores: llamadas**

- De acuerdo con la interfaz (argumentos)

# Ejemplo de llamada y retorno funcional

27



```
gestionarPedido(args) {  
    calcularPrecio(args);  
    gestionarCobro(args);  
    gestionarEnvio(args);  
}
```

# Ventajas de llamada y retorno funcional

28

- Se basa en partes bien identificadas de la tarea a realizar
- Se puede cambiar una subrutina sin que afecte a los clientes
- Se pueden reutilizar operaciones concretas

# Inconvenientes de llamada y retorno funcional

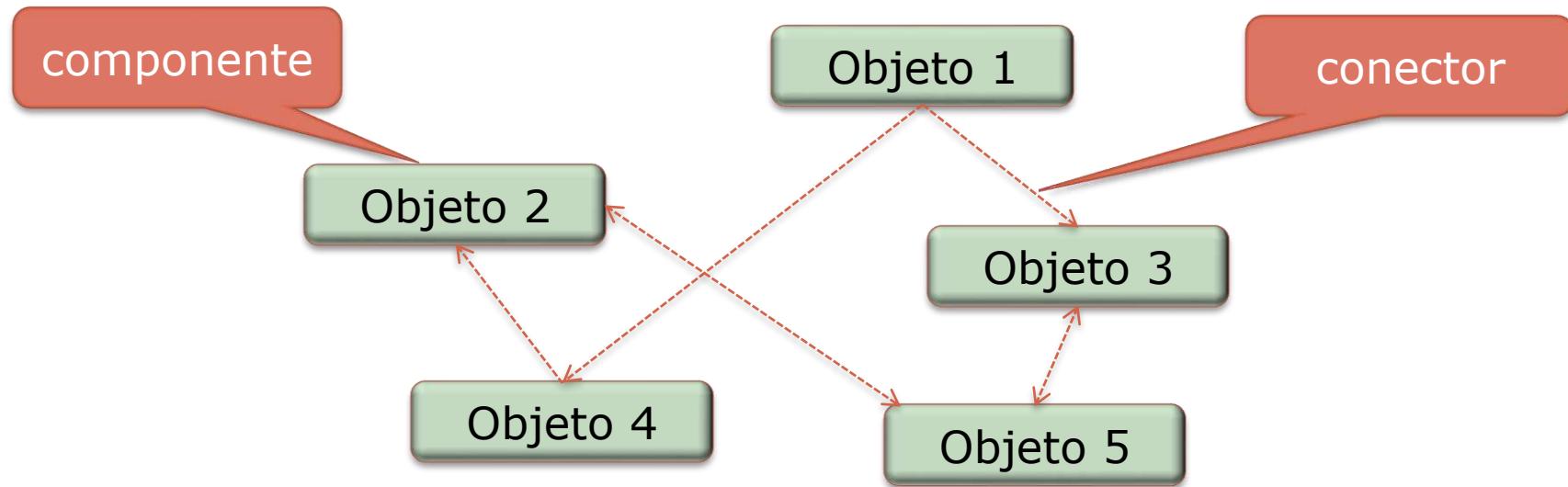
29

- Enfoque operacional que oculta el papel de los datos
- Dependencias entre subrutinas
- Es difícil de extender y adaptar a nuevas situaciones

# Llamada y retorno orientada a objetos (1)

30

- Basado en la descomposición en objetos
  - Los objetos modelan entidades reales (diseño antropomórfico)
  - Colaboran intercambiando mensajes



# Llamada y retorno orientada a objetos (2)

31

- **Componentes: objetos**

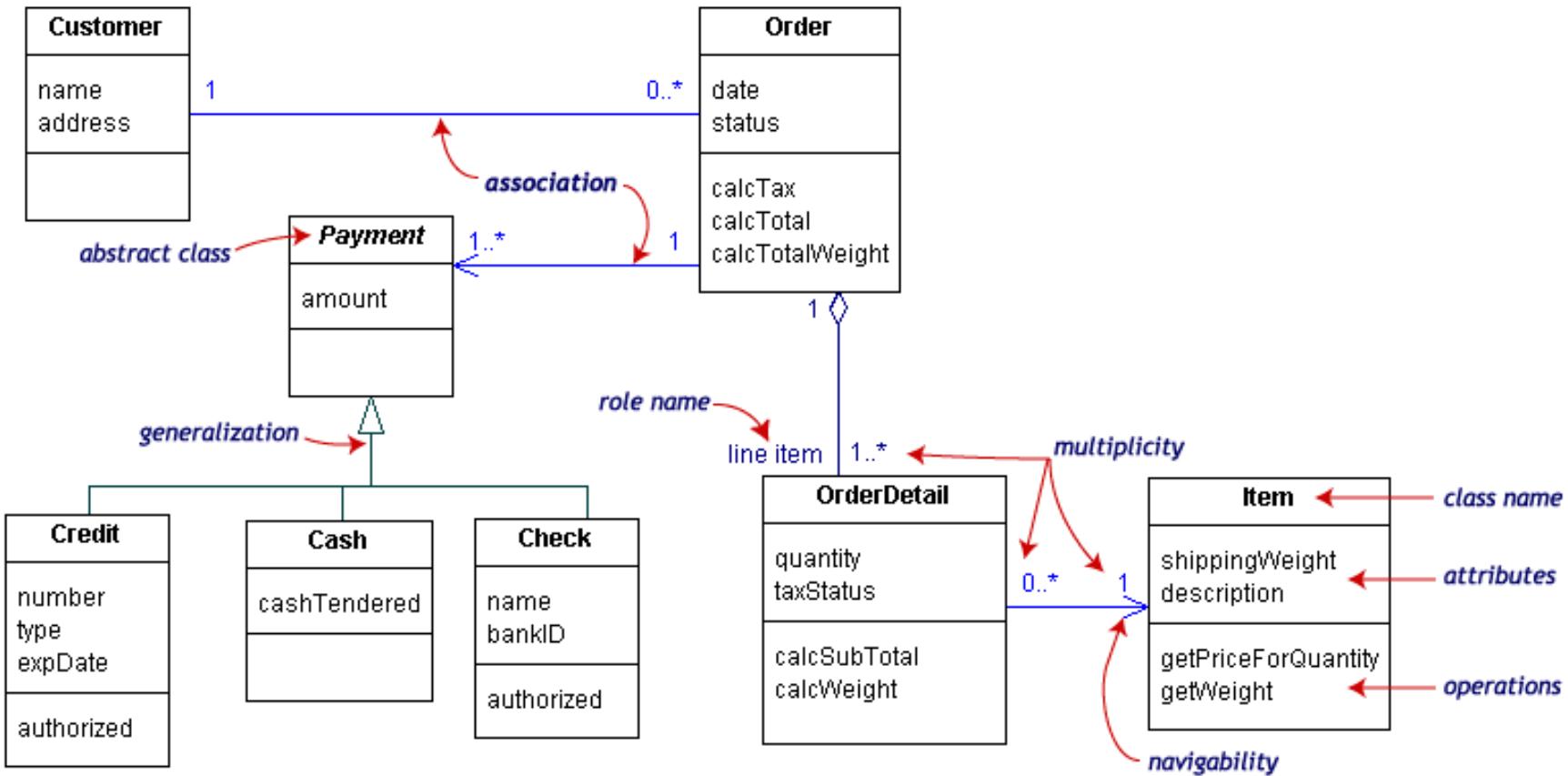
- Representan una entidad del dominio del problema (análisis) o de la solución (diseño)
- Encapsulan un estado privado y mantienen su integridad
- Exhiben un comportamiento público (interfaz)

- **Conectores: mensajes**

- Se pueden resolver en tiempo de ejecución (vinculación dinámica)

# Ejemplo de llamada y retorno con objetos

32



# Ventajas de llamada y retorno con objetos

33

- Se reducen las dependencias
- Se facilitan la reutilización, extensión y el mantenimiento
- Se puede modificar la implementación sin afectar a los clientes
- Se puede distribuir sobre varias máquinas o redes

# Inconvenientes de llamada y retorno con objetos

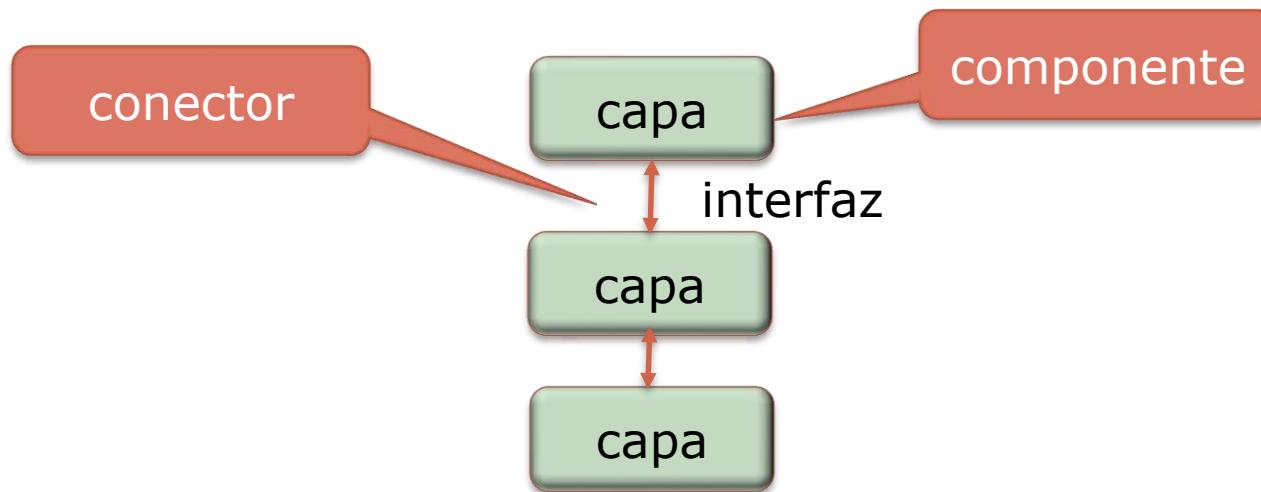
34

- Los objetos deben conocerse para cooperar
- Efectos secundarios de la colaboración
  - Si los objetos A y B colaboran con un objeto C es posible que los efectos de la colaboración entre A y C afecten a la colaboración con B o viceversa

# Arquitecturas de capas (1)

35

- Organizado en niveles de abstracción (capas)
  - Cada capa se comunica exclusivamente con las adyacentes
  - Las **interfaces** entre capas están claramente definidas



# Arquitecturas de capas (2)

36

- **Componentes: capas**

- Grupo de tareas que implementan un nivel de abstracción
- Presta servicios a la capa inmediatamente superior
- Demanda servicios de la capa inmediatamente inferior
- Son completamente independientes de las capas superiores
- Capas más bajas proporcionan servicios de bajo nivel (ej. Protocolos de comunicación, acceso a datos, etc.).

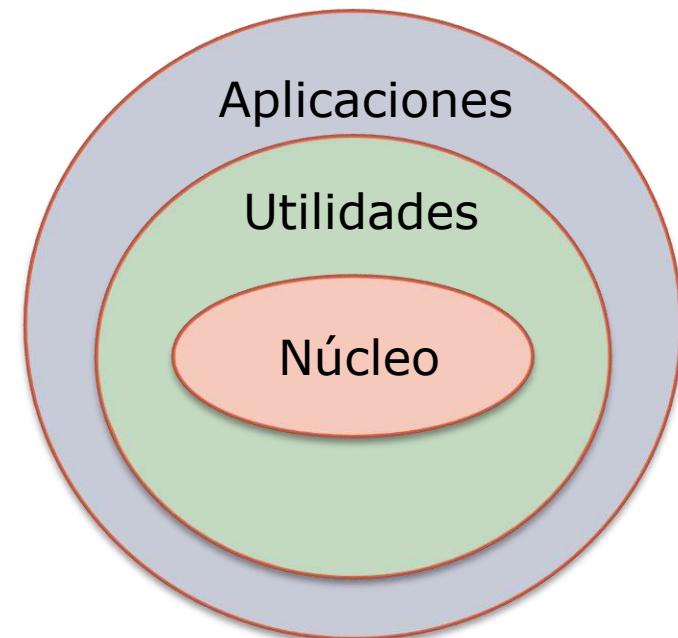
- **Conectores: interfaces**

- Protocolos que definen la interacción entre capas (oferta de servicios)

# Ejemplos de arquitecturas de capas

37

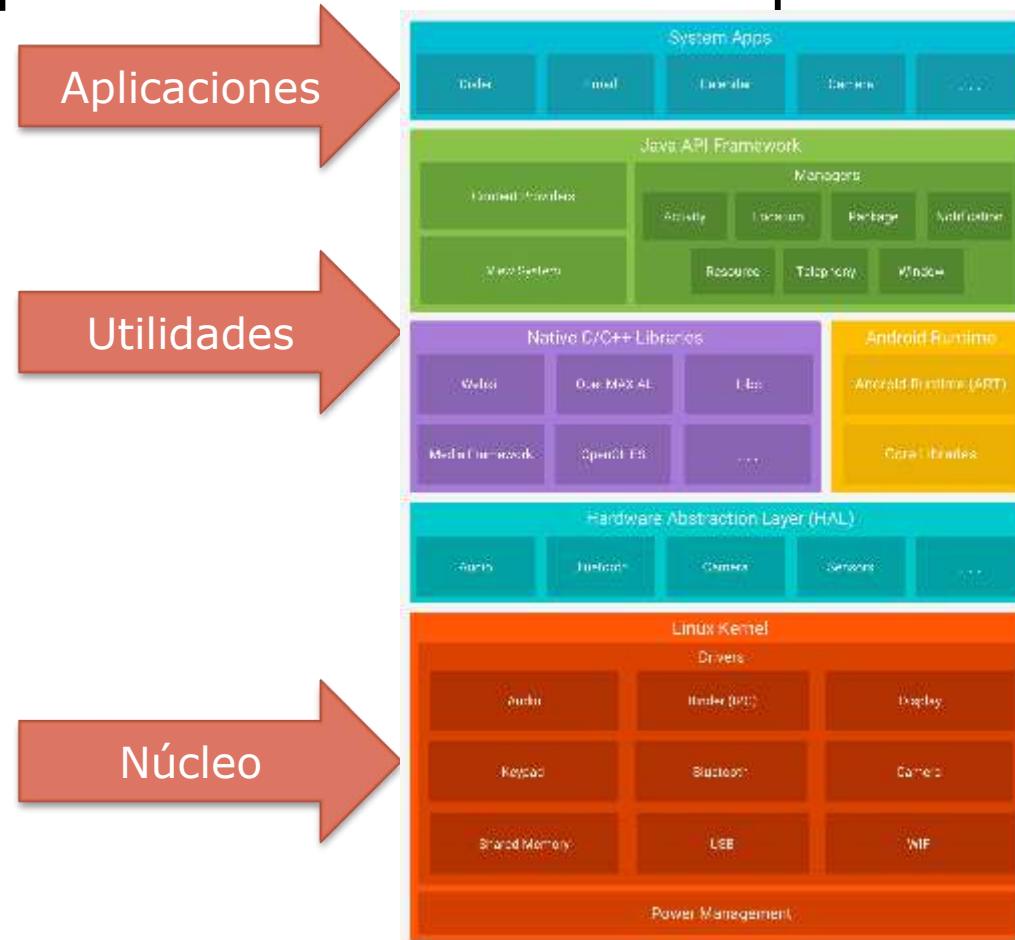
- Protocolos de comunicación (modelo OSI)
- Sistemas operativos



# Ejemplos de arquitecturas de capas

38

- Arquitectura del Sistema Operativo Android



# Ventajas de las arquitecturas de capas

39

- Nivel de abstracción creciente conforme ascendemos por las capas
- **Reutilización:** preservando la interfaz, una capa se puede reemplazar por otra.
- **Mantenimiento:** el cambio de las interfaces de las capas sólo afecta a la capa adyacente (cambios localizados)

# Inconvenientes de las arquitecturas de capas

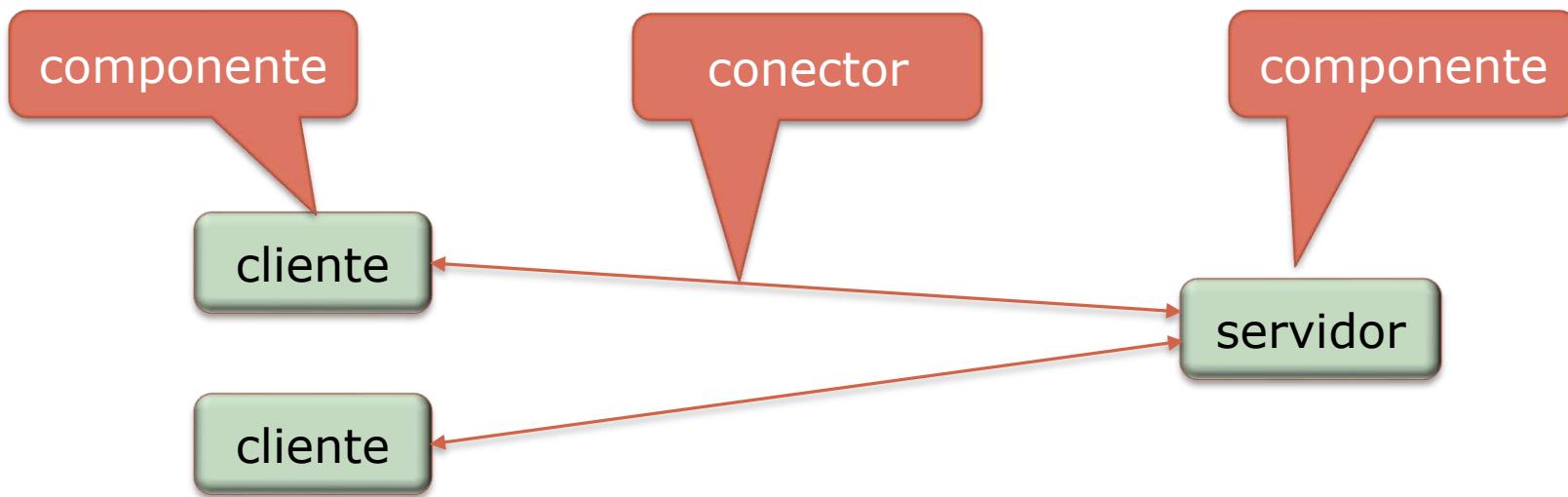
40

- Puede ser difícil identificar las capas.
- **Rendimiento:** descenso y ascenso a través de las capas
  - Es tentador acceder a capas no adyacentes

# Cliente-Servidor (1)

41

- Modelo de sistema distribuido que muestra cómo datos y procesamiento se distribuyen a lo largo de varios procesadores
  - Centraliza la gestión de la información
  - Protocolo de petición-respuesta



# Cliente-Servidor (2)

42

- **Componentes: servidor y clientes**

- El servidor oferta servicios
  - Gran capacidad de cómputo o almacenamiento.
  - Ej. Servidor de impresión, de almacenamiento, Web, etc.
- Los clientes demandan servicios
  - Cliente delgado: poco procesamiento, se centra en la interfaz
  - Cliente grueso: tanto procesamiento como sea posible, el servidor almacena y comunica datos

- **Conectores: flujos de datos**

- Remotos: a través de red
- Locales: cliente y servidor residen en la misma máquina

# Ejemplos de Cliente-Servidor

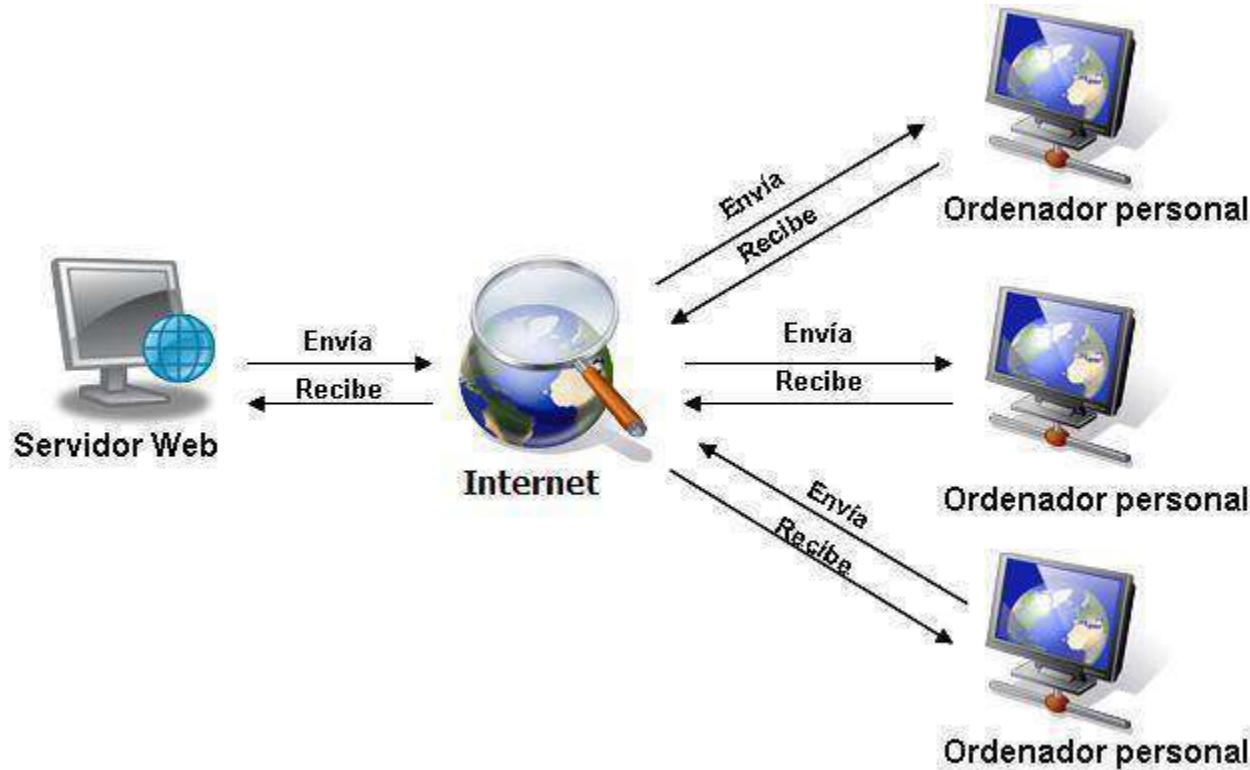
43

- **Bases de datos**
  - Cliente: interfaz personalizado, acceso remoto
  - Servidor: gestión de datos centralizada (integridad, consistencia, seguridad, ...), accesos concurrentes, ...
- **Subversion** (herramienta de control de versiones open source basada en un repositorio)
  - Cliente: actualización, resolución de conflictos
  - Servidor: almacenamiento de revisiones
- **Otros servidores:** web, impresión, X WindowSystem, ...

# Ejemplos de Cliente-Servidor

44

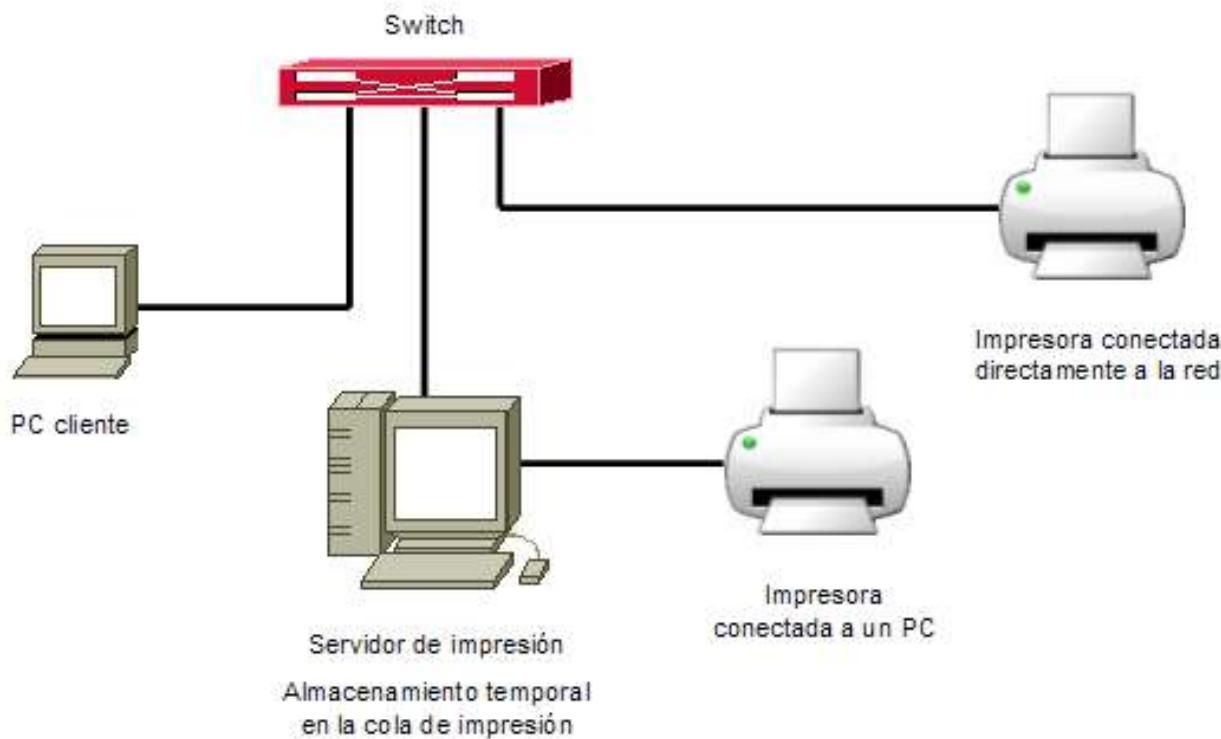
- Servidor Web:



# Ejemplos de Cliente-Servidor

45

- Servidor de Impresión:



# Ventajas de Cliente-Servidor

46

- Control centralizado
  - Facilita el mantenimiento
  - Refuerza seguridad
- Escalabilidad
  - Es fácil aumentar la capacidad del servidor o los clientes
- Uso racional del presupuesto hardware
  - Concentra el gasto en el servidor

# Inconvenientes de Cliente-Servidor

47

- Congestión del tráfico
  - Cuello de botella, denegación de servicio, etc.
- Robustez
  - ¿qué pasa si falla el servidor?
- Coste de la transformación de los datos
  - Cada cliente puede utilizar un formato distinto

# Arquitectura Modelo-Vista-Controlador

48

- **Modelo-Vista-Controlador (MVC)**: ayuda a separar la capa de interfaz de usuario de otras partes del sistema.
- Adecuada cuando:
  - Aplicaciones con interfaces de usuario interactivas.
  - Hay múltiples formas de mostrar o interactuar con los datos.
  - Los requisitos de la interacción y la presentación son desconocidos.
- **Idea clave**: separar el modelo de negocio y la interfaz.
  - Modelo de negocio: datos y lógica de negocio que implementa la funcionalidad básica de la aplicación.
  - Interfaz: interacción con el usuario.
  - Se intenta no contaminar el modelo con la interfaz.

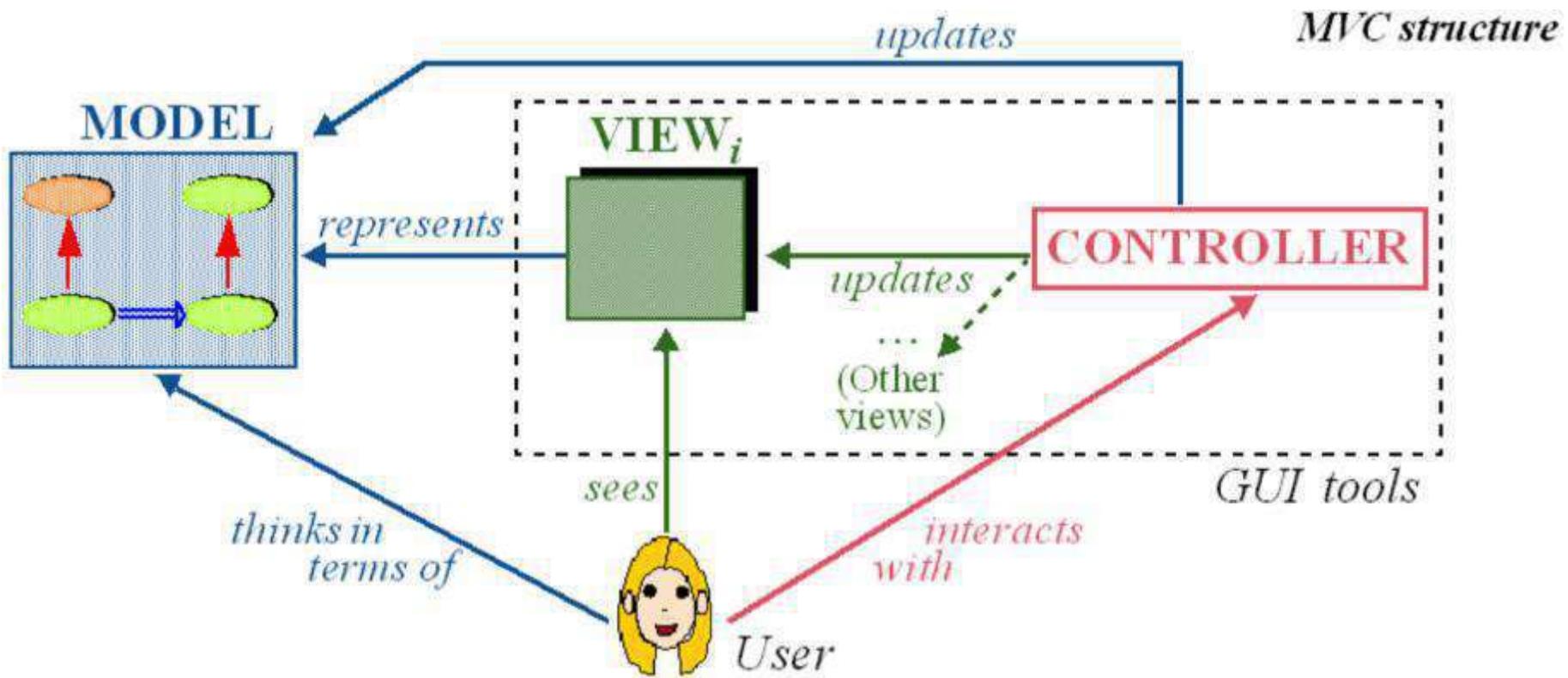
# Arquitectura MVC (2)

49

- Tres tipos de componentes:
  - **Modelo**: responsable de gestionar los datos del sistema y las operaciones asociadas a los datos (datos del negocio).
  - **Vista**: define y gestiona cómo los datos se muestran al usuario (representación visual del modelo).
  - **Controlador**: responsable de la interacción con el usuario (p.ej. clicks de ratón, teclas pulsadas) y de pasar esas interacciones a la Vista y al Modelo.
- A un modelo le pueden corresponder múltiples vistas y controladores
- Interfaz = Vista + Controlador.
  - Controlador y vista están muy relacionados.

# Arquitectura MVC

50



# El Modelo

51

- Representa el problema a resolver.
- Encapsula el estado de la aplicación.
- Contiene los datos y la lógica del negocio.
- No contiene operaciones de entrada/salida.
- Es independiente de la vista y el controlador.
  - No contiene referencias ni a la vista ni al controlador.
  - Pero facilita métodos para que estos lo usen.

# La Vista

52

- Ofrece una visión total o parcial del estado del modelo.
- Contiene controles interactivos que lanzan eventos, pero no los maneja.
- No debe suponer el estado del modelo, debe reflejarlo fielmente.
  - No utiliza estados almacenados.
- Es un observador pasivo
  - No afecta al modelo.

# El Controlador

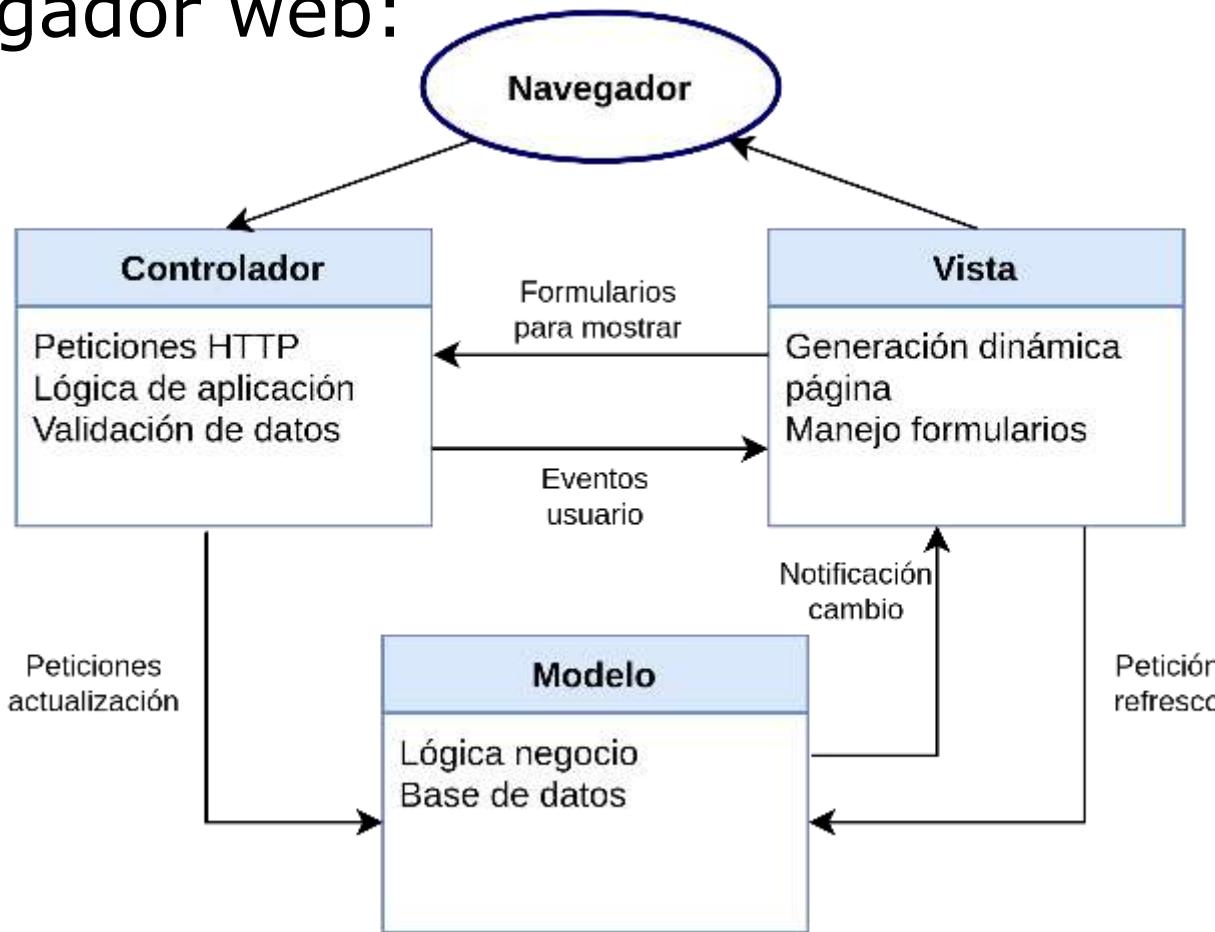
53

- Gestiona la interacción usuario-modo.
- Responde a los eventos de la vista, indicando qué acciones debe ejecutar el modelo y cómo actualizar la vista.
  - Traduce los eventos de la vista a acciones sobre el modelo y actualizaciones de la vista.
- Contiene referencias al modelo y la vista.

# Ejemplo MVC

54

- Navegador web:



# Ventajas e Inconvenientes de MVC

55

- **Ventajas:**

- Robustez:
  - Permite que los datos cambien con independencia de su representación y viceversa.
- Flexibilidad
  - Los mismos datos se pueden presentar de múltiples formas
  - Se pueden reutilizar componentes.
- Mantenimiento
  - Desarrollo, prueba y mantenimiento por separado

- **Inconvenientes:**

- Sobrecarga extra:
  - Puede requerir código adicional (o más complejidad) para modelos de datos o interacciones simples.

# Introducción a la Ingeniería del Software



## TEMA 6.2: DISEÑO SOFTWARE

Grado en Ingeniería Informática  
Grado en Ingeniería del Software  
Grado en Ingeniería de Computadores



# Índice de contenidos

2

- Concepto de Diseño Software
- Principios de Diseño
- Principios de Diseño Orientado a Objetos
- Patrones de Diseño
- Refactorización

# Índice de contenidos

3

- Concepto de Diseño Software
- Principios de Diseño
- Principios de Diseño Orientado a Objetos
- Patrones de Diseño
- Refactorización

# El problema del diseño software

4

*"Hay dos formas de construir un **diseño de software**: una forma es **hacerlo tan simple** que es evidente que **no hay deficiencias**, y la otra forma es **hacerlo tan complicado** que no haya deficiencias obvias. El primer método es mucho más difícil".*

C.A.R. Hoare

# Concepto de diseño (1)

5

- El **diseño** es un proceso de **resolución de problemas** cuyo objetivo es encontrar y describir una manera de:
  - Implementar los requisitos funcionales (servicios que se esperan del sistema)
  - Respetando las restricciones impuestas por los requisitos de calidad, plataforma y proceso software
  - Ajustado al presupuesto y plazo de ejecución
  - Siguiendo principios generales de calidad

Lethbridge y Laganiere

# Concepto de diseño (2)

6

- **Diseño** es el proceso creativo de transformar un **problema** en una **solución**
- La descripción de una solución también se conoce como diseño
- La fase de **análisis** plantea el problema (**qué**)
- La fase de **diseño** especifica una solución particular para el problema (**cómo**)

Pfleeger y Atlee

# Propiedades de diseño

7

- El diseño es un proceso de refinamiento que implica una **propuesta de solución**
  - Integra el paso del ¿Qué? al ¿cómo?
- Es una **actividad creativa**, apoyada por principios, técnicas, herramientas, ...
- Requiere adoptar **decisiones de diseño** explícitas
  - Éstas luego podrán ser corregidas o modificadas
- Es clave para la calidad posterior del software

# Espacio y decisiones de diseño

8

El diseñador se enfrenta a una serie de **cuestiones de diseño**

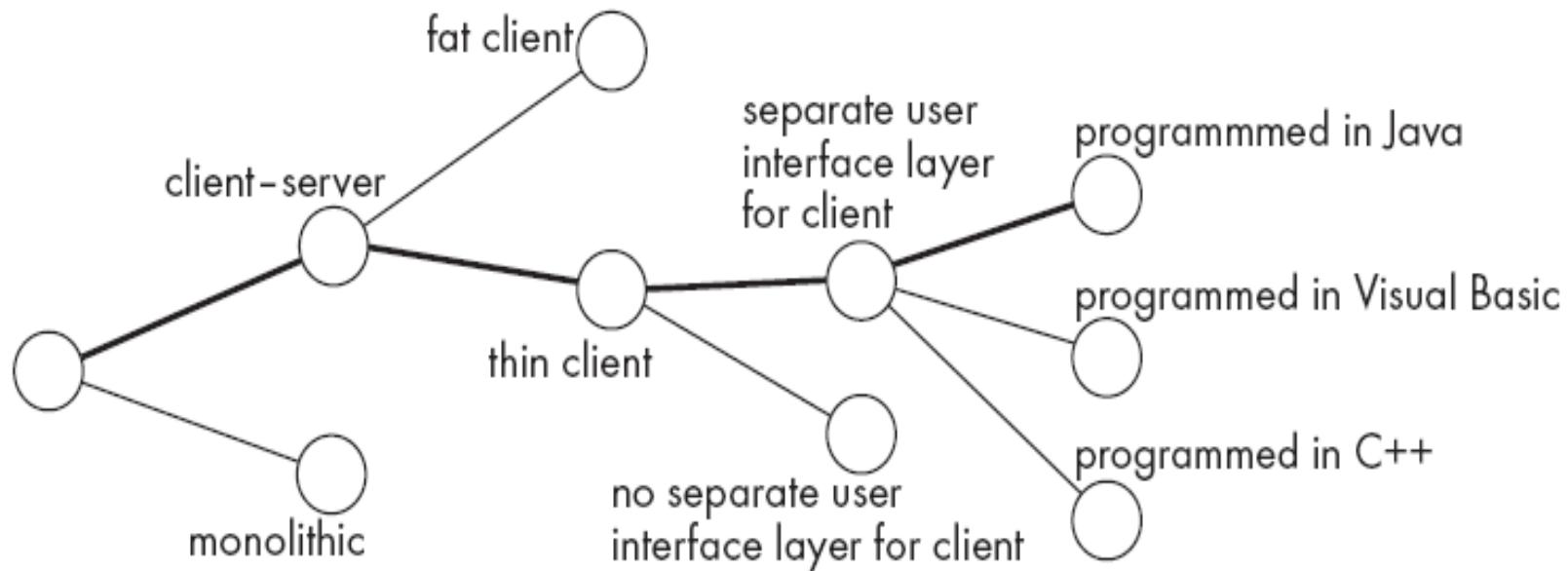
Cada cuestión tiene varias soluciones alternativas u **opciones de diseño**

El diseñador toma **decisiones de diseño** para cada una de las cuestiones

**Espacio de diseño:** es el conjunto de diseños que se puede obtener según las diferentes opciones de diseño

# Ejemplo simple de espacio de diseño

9



# Tomando decisiones de diseño

10

- Para tomar una decisión de diseño nos guiamos por:
  - Los requisitos
  - La parte del diseño ya creada
  - La tecnología disponible
  - Los principios de diseño
  - La experiencia

# Índice de contenidos

11

- Concepto de Diseño Software
- **Principios de Diseño**
- Principios de Diseño Orientado a Objetos
- Patrones de Diseño
- Refactorización

# Algunos objetivos de un buen diseño

12

- **Satisfacer los requisitos** (funcionales y no funcionales)
- Asegurar la **calidad del software**
- Ajustarse al **presupuesto y plazo de entrega**

# ISO/IEC 25010 – Product Quality Model

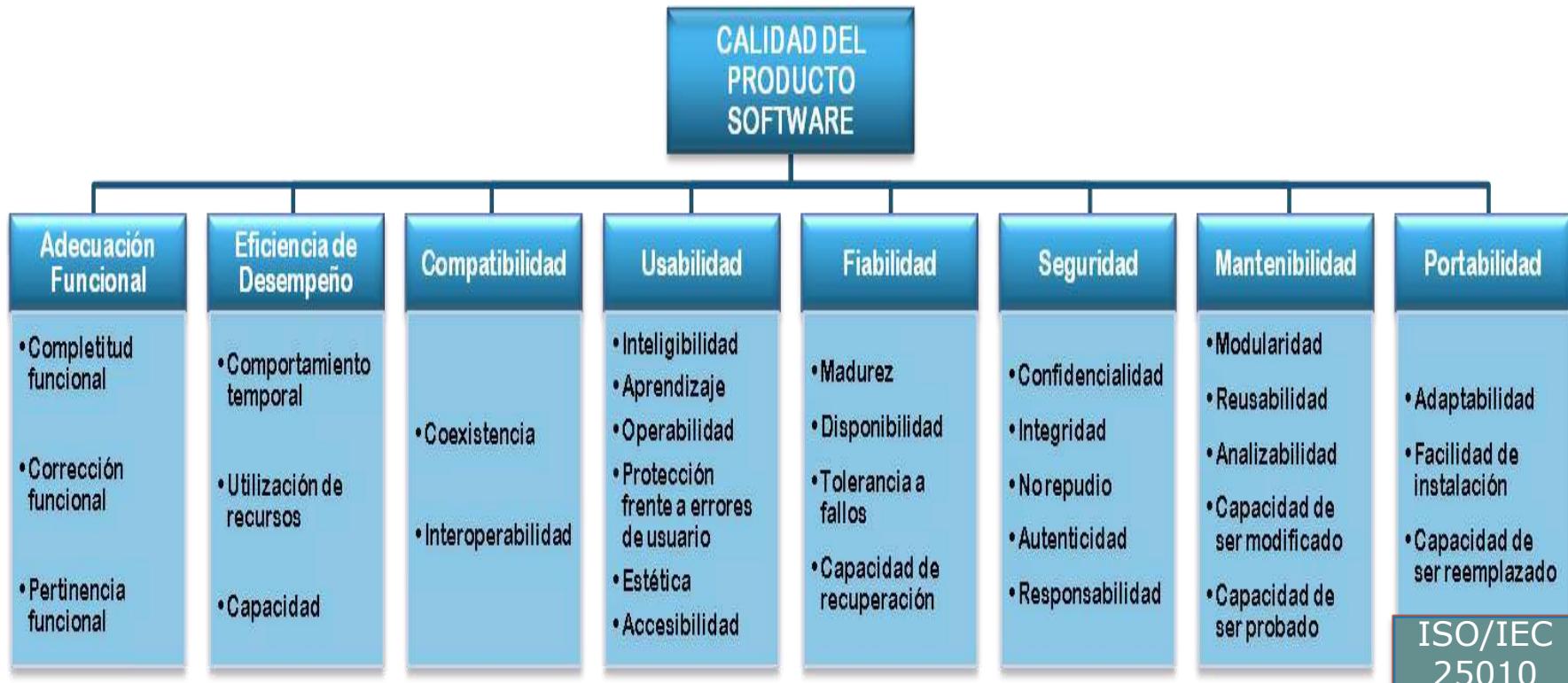
13

- ISO: International Organization for Standardization
- IEC: International Electrotechnical Commission
- El **Modelo de Calidad del Producto Software** ISO/IEC 25010 establece el sistema para la **evaluación de la calidad** de un producto **software**.
- **Calidad**: grado en que el producto satisface los requisitos de los usuarios aportando de esta forma un valor a la organización.
- **URL:** <http://iso25000.com/index.php/normas-iso-25000/iso-25010>

# ISO/IEC 25010 – Product Quality Model

14

- El modelo de calidad del producto ISO/IEC 25010 (**en su vista interna**) se compone de ocho características:



# ISO/IEC 25010 – Product Quality Model

15

- **Adecuación Funcional:** capacidad de proporcionar funciones que satisfacen las necesidades declaradas
- **Eficiencia de desempeño:** cantidad de recursos utilizados bajo determinadas condiciones
- **Compatibilidad:** capacidad de los sistemas para intercambiar información y/o realizar sus funciones compartiendo entorno
- **Usabilidad:** capacidad del producto para ser entendido, aprendido, usado y resultar atractivo
- **Fiabilidad:** capacidad de realizar las funciones especificadas bajo unas condiciones determinadas
- **Seguridad:** protección de la información de manera que personas o sistemas no autorizados no puedan leerlos o modificarlos
- **Mantenibilidad:** capacidad del producto para ser modificado efectiva y eficientemente, debido a las necesidades
- **Portabilidad:** capacidad del producto o componente de ser transferido de forma efectiva y eficiente de un entorno a otro

# ISO/IEC 25010 – Quality in Use Model

16

- El modelo de calidad del producto ISO/IEC 25010 (**en su vista en uso**) contempla la productividad y efectividad del usuario final al utilizar el software:



23

# ISO/IEC 25010 – Quality in Use Model

17

- **Efectividad:** capacidad de lograr los resultados deseados
- **Eficiencia:** capacidad de lograr dicho resultado con el mínimo de recursos posible
- **Libertad de Riesgo:** minimización de los riesgos económicos, de salud y ambientales
- **Satisfacción:** aceptación plena de los usuarios del sistema
- **Contexto de Uso:** circunstancias en las cuales el producto va a ser usado

# Algunos principios de diseño

18

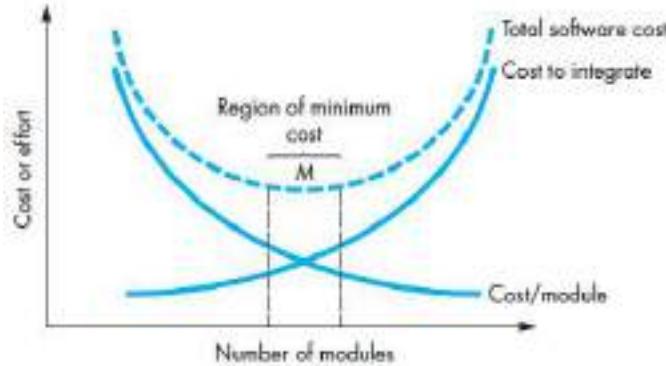
Principios que conducen a un buen diseño software:

- **Modularidad:** divide y vencerás
- **Aumentar la cohesión:** agrupar las unidades software que están relacionadas.
  - Dicha unidad será mas sencilla de diseñar, implementar, probar y mantener.
- **Reducir el acoplamiento:** grado de relación de un módulo con los demás.
  - A menor acoplamiento el módulo será más sencillo diseñar, implementar, probar y mantener.

# Modularidad: divide y vencerás

19

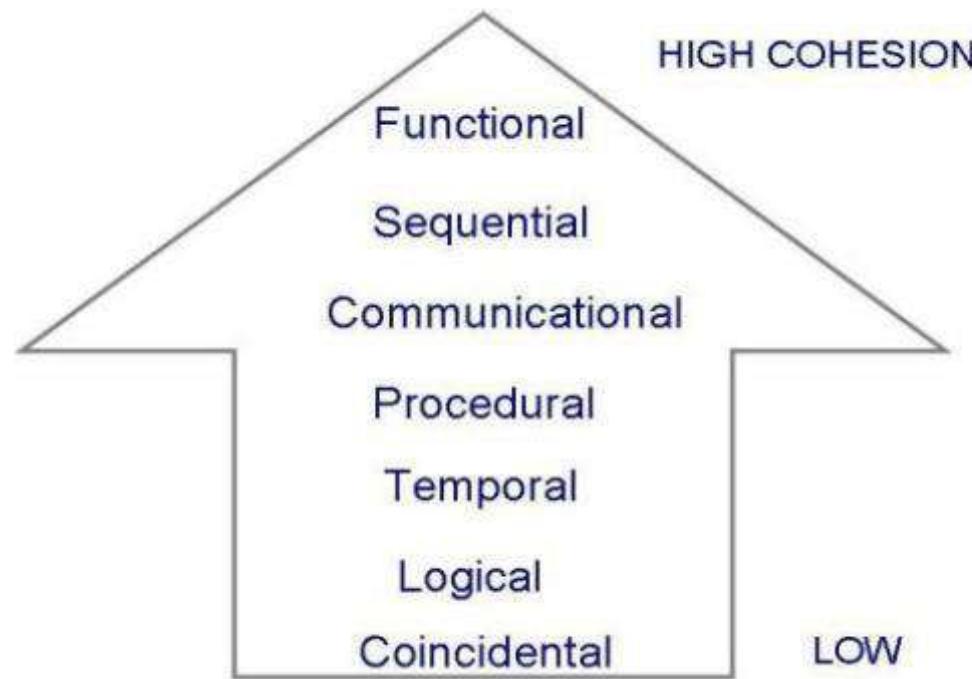
- El **sistema se divide en componentes** identificables y tratables por separado que se integran para satisfacer los requisitos del sistema
- **Ventajas:**
  - Cada componente es más pequeño que el total; más fácil de entender y diseñar
  - Los componentes se pueden cambiar o reemplazar
  - Diferentes personas pueden trabajar en diferentes partes
- **Problema:** llegar a un balance en el nº de módulos



# Aumentar la cohesión

20

- **Cohesión:** medida del **grado de relación** de los **contenidos** de un componente entre sí
- Existen varios **grados de cohesión**



# Tipos de Cohesión

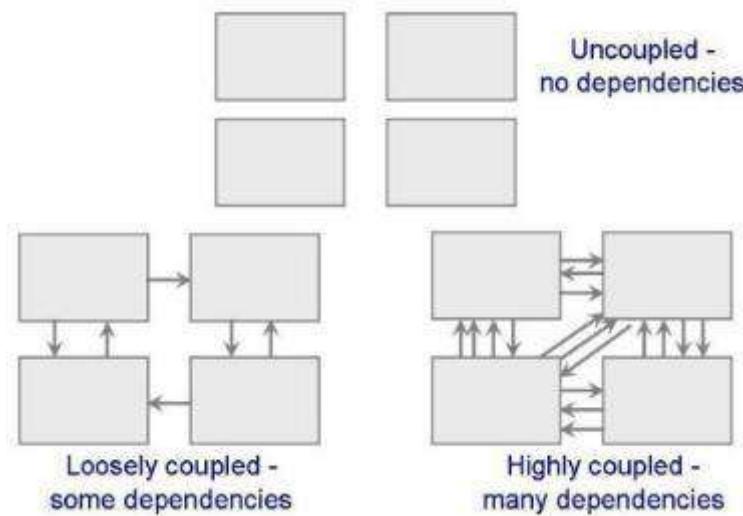
21

- **Cohesión funcional:** se agrupan elementos que desarrollan una única función sin efectos colaterales (p.ej. funciones matemáticas, ordenamiento de arrays).
- **Cohesión secuencial:** un módulo realiza distintas tareas en secuencia, de forma que las entradas de cada tarea son las salidas de la tarea anterior.
- **Cohesión comunicacional:** se agrupan los elementos que operan sobre los mismos tipos datos. Un buen diseño de clases tiene este tipo de cohesión.
- **Cohesión procedimental:** similar a la secuencial, pero la salida de una tarea no tiene por qué ser la entrada a la siguiente. En este caso se agrupan tareas diferentes y posiblemente no relacionadas (p.ej. 1er año universidad: matricularse en universidad, buscar piso, salir de fiesta, etc).
- **Cohesión temporal:** las operaciones que se realizan durante la misma fase de la ejecución del programa se mantienen juntas (p.ej. agrupación de operaciones de inicialización o terminación).
- **Cohesión lógica:** distintas operaciones se agrupan porque están relacionadas de forma lógica, esto es, realizan acciones parecidas (p.ej. agrupar rutinas para gestión del teclado y/o ratón).
- **Cohesión casual o coincidente:** las operaciones se agrupan de forma arbitraria (p.ej. clase de utilidades o dividir programa en N subrutinas de igual longitud).

# Reducir el acoplamiento

22

- **Acoplamiento:** medida del **grado de dependencia** entre los **componentes** de un sistema
  - Sistemas muy acoplados son difíciles de probar, implementar y mantener.
- Hay varios **grados de acoplamiento**



# Tipos de Acoplamiento

23

- **Acoplamiento normal:** una unidad de software llama a otra de un nivel inferior y tan solo intercambian datos (p.ej. parámetros de entrada / salida).
- **Acoplamiento externo:** las unidades de software están ligadas a componentes externos (p.ej. dispositivos de entrada / salida, librerías comunes, protocolos de comunicaciones, etc.)
- **Acoplamiento común:** dos unidades de software acceden a un mismo recurso común, generalmente memoria compartida, una variable global o un fichero.
- **Acoplamiento de contenido:** ocurre cuando una unidad de software necesita acceder a una *parte interna* de otra unidad de software.

# Índice de contenidos

24

- Concepto de Diseño Software
- Principios de Diseño
- **Principios de Diseño Orientado a Objetos**
- Patrones de Diseño
- Refactorización

# Principios **SOLID**

25

Acrónimo acuñado por Robert C. Martin para resumir los 5 **principios básicos del Diseño Orientado a Objetos**:

- **S**ingle Responsibility Principle (SRP)
- **O**pen-Closed Principle (OPC)
- **L**iskov Substitution Principle (LSP)
- **I**nterface Segregation Principle (ISP)
- **D**evelopment Dependency Inversion Principle (DIP)

# Principio de Responsabilidad Única (SRP)

26



## Single Responsibility Principle

Just because you *can* doesn't mean you *should*.

**Simplemente porque puedes no quiere decir que debas**

# Principio de responsabilidad única (SRP)

27

*"Una clase debería tener un solo motivo para cambiar"*

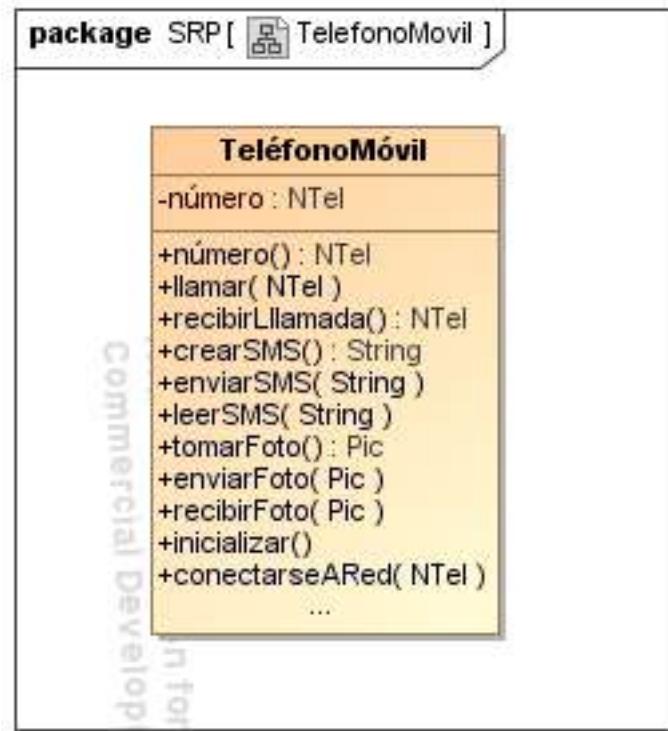
Robert C. Martin

*"Una clase debe hacer una cosa y hacerla bien"*

- Responsabilidad es la razón para cambiar.
- Reinterpreta la **cohesión** desde el punto de vista de los motivos que fuerzan los cambios en una clase.
- Los servicios de la clase se deben alinear para proveer esa responsabilidad.
- Si una clase asume más de una responsabilidad: más sensible al cambio y las responsabilidades se acoplan

# Violación de SRP

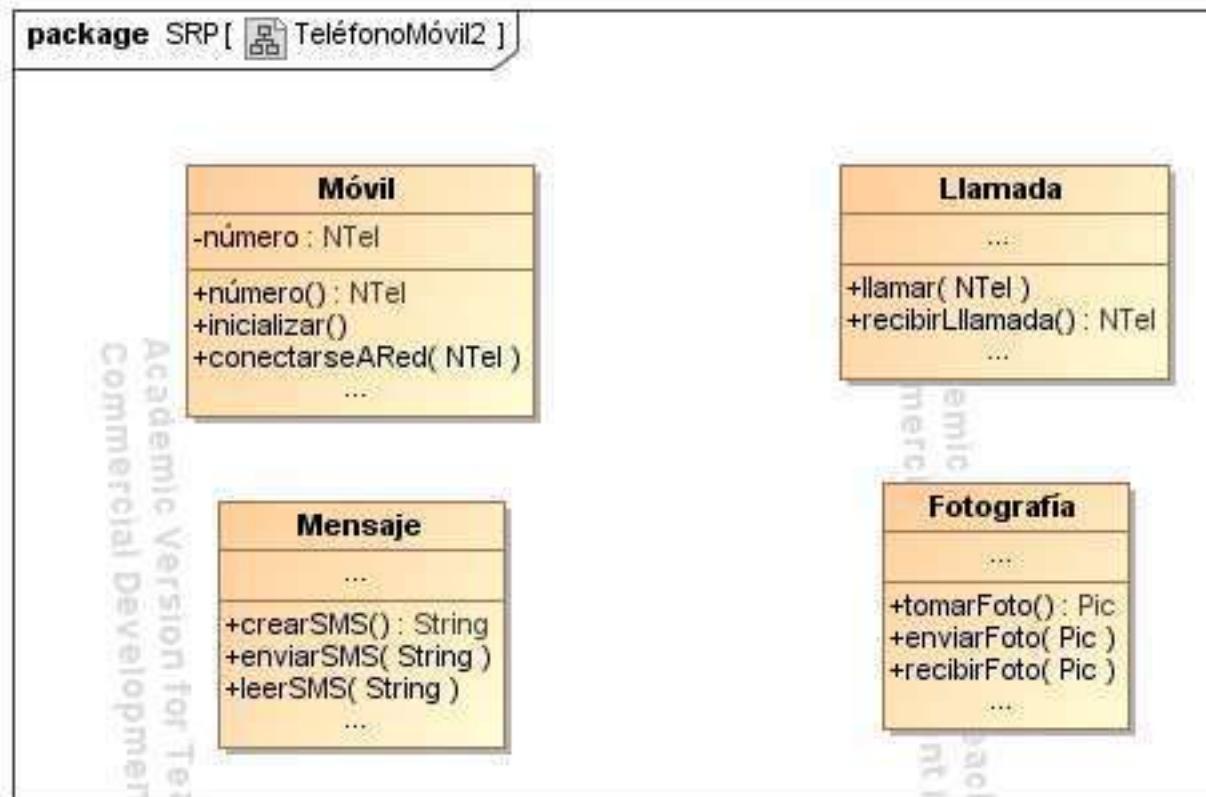
28



**Clase que asume múltiples responsabilidades**

# Aplicación de SRP

29



**Clases donde cada una asume una responsabilidad**

# Principio abierto-cerrado (OCP)

30



## Open-Closed Principle

Open-chest surgery isn't needed when putting on a coat.

**La cirugía a tórax abierto no es necesaria para ponerse un abrigo**

# Principio abierto-cerrado

31

*"Las clases deben estar abiertas a la extensión  
y cerradas a la modificación"*

Bertrand Meyer

- Adaptarse a nuevas situaciones:
  - sin modificar el código existente que funciona bien (cerrado para modificación)
  - añadiendo nuevo código (abierto para extensión)
- Se puede conseguir mediante polimorfismo:
  - Dinámico: sin especificación del tipo de datos (herencia, vinculación dinámica)
  - Estático: tipos de datos explícitos (genericidad, plantillas)

# Violación de OCP

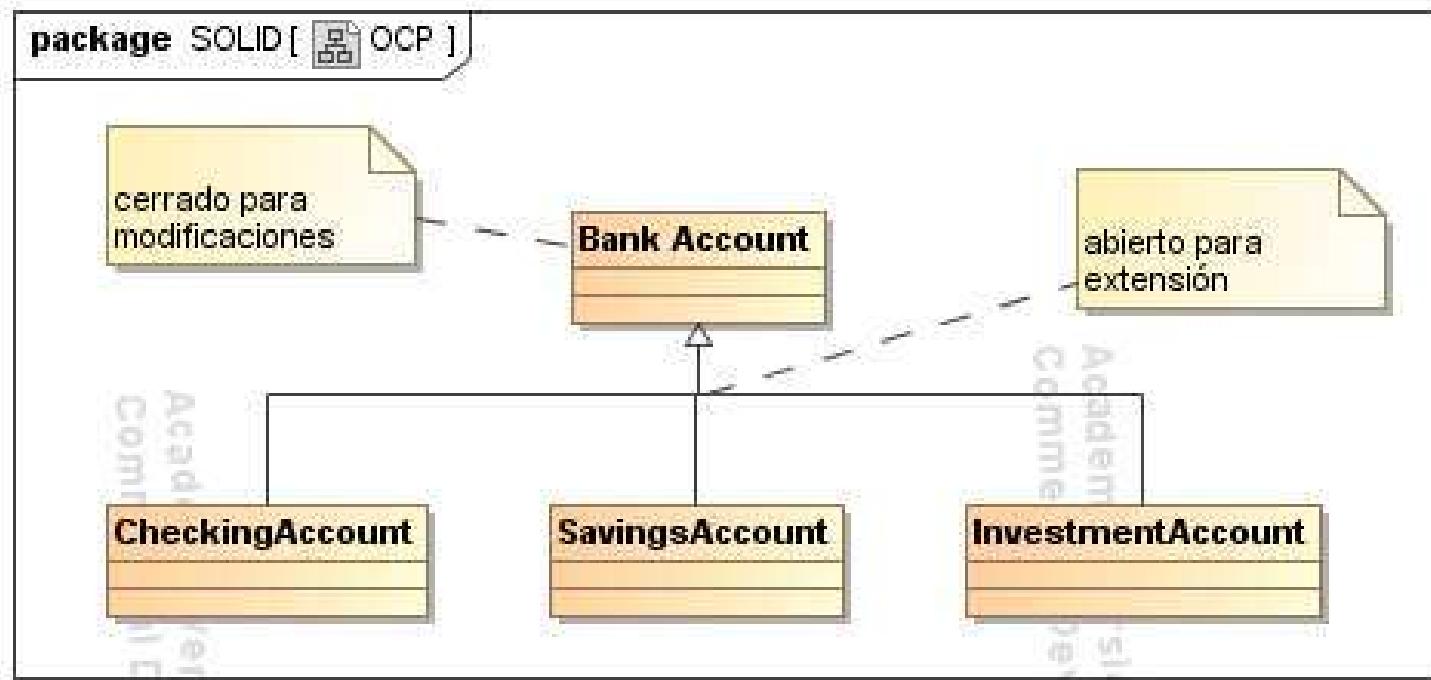
32

```
switch (a->account_type) {  
    case checkingAcc : foo(a); break;  
    case savingsAcc : bar(a); break;  
    case investmentAcc : qux(a); break;  
    default : hmmm(a);  
}
```

- **Problema:** es imposible añadir un nuevo tipo de cuenta sin modificar el código

# Aplicación de OCP

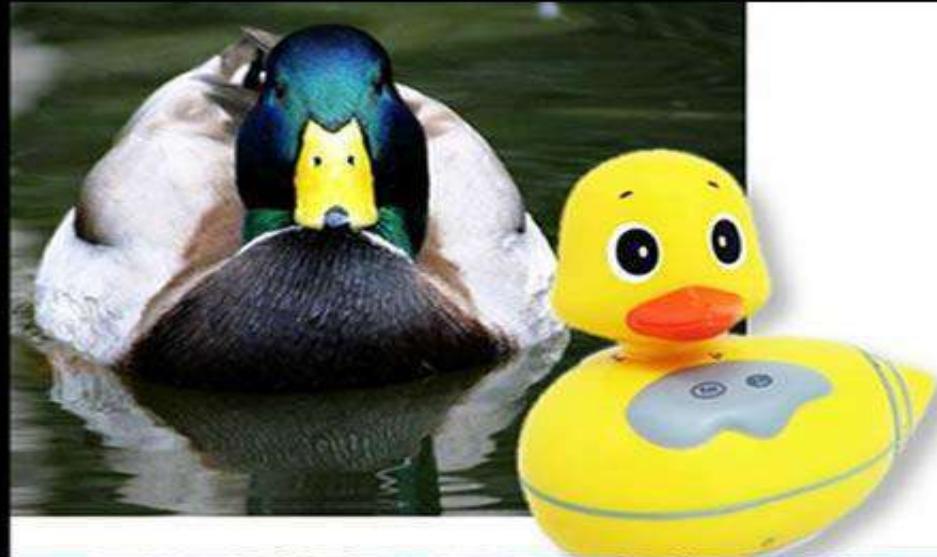
33



- Cambiar no es modificar lo que ya funciona
- Cambiar es extender

# Principio de sustitución de Liskov

34



## Liskov Substitution Principle

If it looks like a duck and quacks like a duck but needs batteries,  
you probably have the wrong abstraction.

**Si parece un pato y grazna como un pato pero necesita pilas, probablemente tienes mal la abstracción**

# Principio de sustitución de Liskov

35

*"Las subclases deben poder sustituir a las clases base sin que el código cliente lo note"*

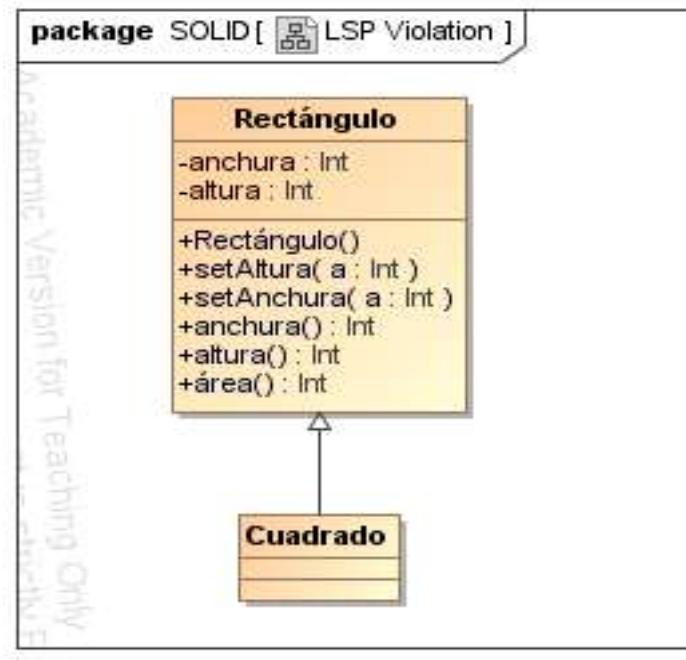
Barbara Liskov

- *es-un = puede-sustituir-a*
- Usar bien la herencia
- Sustituir sin comprometer propiedades deseables
- Asegurar la interoperabilidad semántica de los subtipos
- Clases derivadas deben respetar los contratos (precondiciones y postcondiciones) definidas en las clases base.

# Violación de LSP (1)

36

- Un cuadrado es *un* rectángulo



**La clase Cuadrado hereda de Rectángulo, pero no puede sustituir a ésta**

# Violación de LSP (2)

37

- Un cuadrado es un rectángulo si altura y anchura coinciden:

```
class Cuadrado extends Rectángulo {  
    public int setAnchura(int a) {  
        anchura = altura = a;  
    }  
  
    public int setAltura(int a) {  
        anchura = altura = a;  
    }  
}
```

# Violación de LSP (3)

38

- Un cuadrado no puede sustituir a un rectángulo: el cliente nota la diferencia

```
int pruebaÁrea(Rectángulo r) {  
    r.setAnchura(2);  
    r.setAltura(5);  
    assert(r.área() == 2*5);  
}
```

# Relación entre OCP y LSP

39

- “Arreglo” mediante violación de OCP:

```
int pruebaÁrea(Rectángulo r) {  
    if (r instanceof Rectángulo) {  
        r.setAnchura(2);  
        r.setAltura(5);  
        assert(r.área() == 2*5);  
    }  
}
```

- Violación de LSP = Violación latente de OCP
- El no cumplimiento de LSP conlleva el no cumplimiento de OCP

# Síntomas de la violación del LSP

40

- Ocultar el comportamiento heredado:
  - Redefiniendo métodos con implementaciones vacías o lanzando excepciones indicando “no soportado”
- Modificar el comportamiento heredado:
  - Redefiniendo métodos con semánticas incompatibles
  - Lanzando excepciones nuevas, inesperadas por cliente
- Modificar la clase base:
  - Para adaptarla a las necesidades de la heredera
- Si redefines más que extiendes, deberías usar delegación, agregación o composición

# Diseño por contrato (DbC)

41

- Técnica para asegurar el LSP (B. Meyer)
- Una clase expone un contrato entre implementador y cliente
- El contrato contiene de forma explícita:
  - Un **invariante de clase**: se verifica antes y después de invocar un método (no durante la ejecución)
  - Para cada método:
    - ✖ **Una precondición**: obligación para el cliente, beneficio para el implementador
    - ✖ **Una postcondición**: obligación para el implementador, beneficio para el cliente

# Diseño por contrato y herencia

42

- Las clases herederas tienen que respetar el contrato de la clase base
- Una clase heredera:
  - Debe preservar el invariante de la clase
  - Para cada método, puede:
    - ✖ Debilitar la precondición (menos obligaciones para cliente)
    - ✖ Fortalecer la postcondición
      - *"derived methods should expect no more and provide no less"*
- Soporte en los lenguajes del DbC:
  - nativo: Eiffel, D, Spec#, Fortress, Vala, ...
  - externo: Java, C#, C/C++, Ada, Python, Ruby, ...

# Ejemplo: diseño por contrato en Java

43

```
public class orderedList{
    /* lista ordenada sin repeticiones
     * @invariant forall Node n in elements()
     *   n.prev() != null
     *   implies
     *   n.value().compareTo(n.prev().value())>0
     */
    // @pre contains(aNode) == false
    // @post contains(aNode) == true
    public void insertNode(final Node aNode){
        //..
```

# Principio de segregación de interfaces

44



**Interface Segregation Principle**  
You want me to plug this in *where?*

**Quieres que conecte esto, ¿dónde?**

# Principio de segregación de interfaces

45

*"Los clientes no deben depender de métodos que no utilizan"*

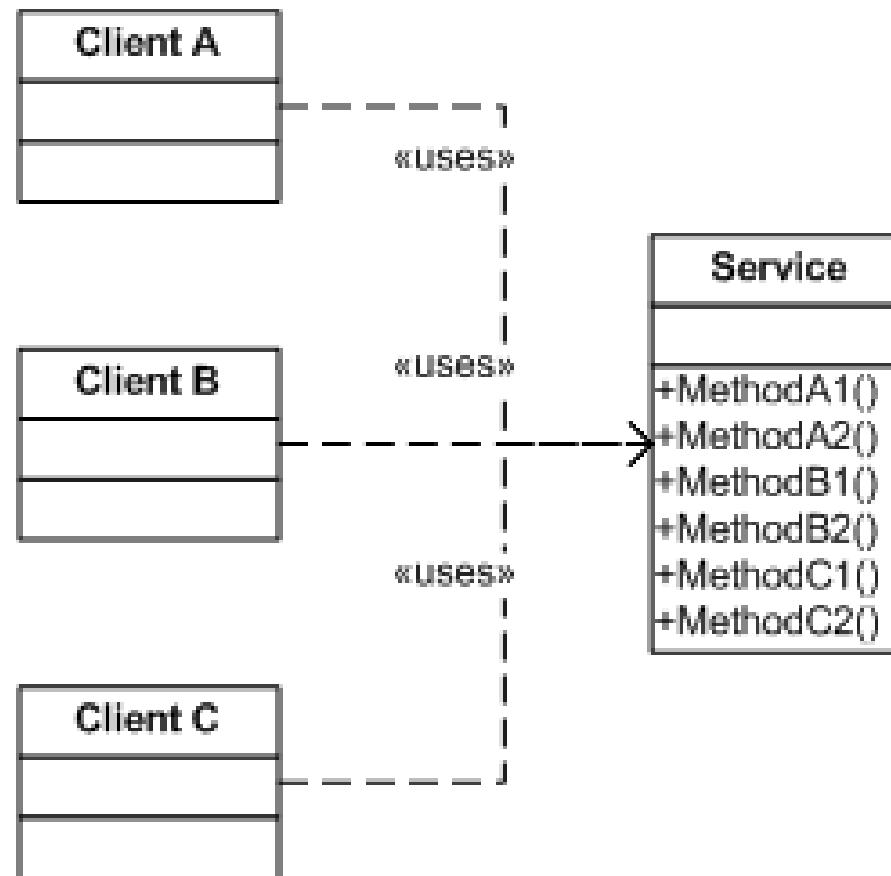
Robert C. Martin

- Algunas clases tienen **interfaces amplias**
- Los clientes solo requieren algunos servicios
- Algunos clientes hacen crecer la interfaz
- Agrupar los clientes según necesidades y segregar la interfaz en función de los clientes
  - Las interfaces reducidas son conocidas como interfaces de rol.

# Violación de ISP

46

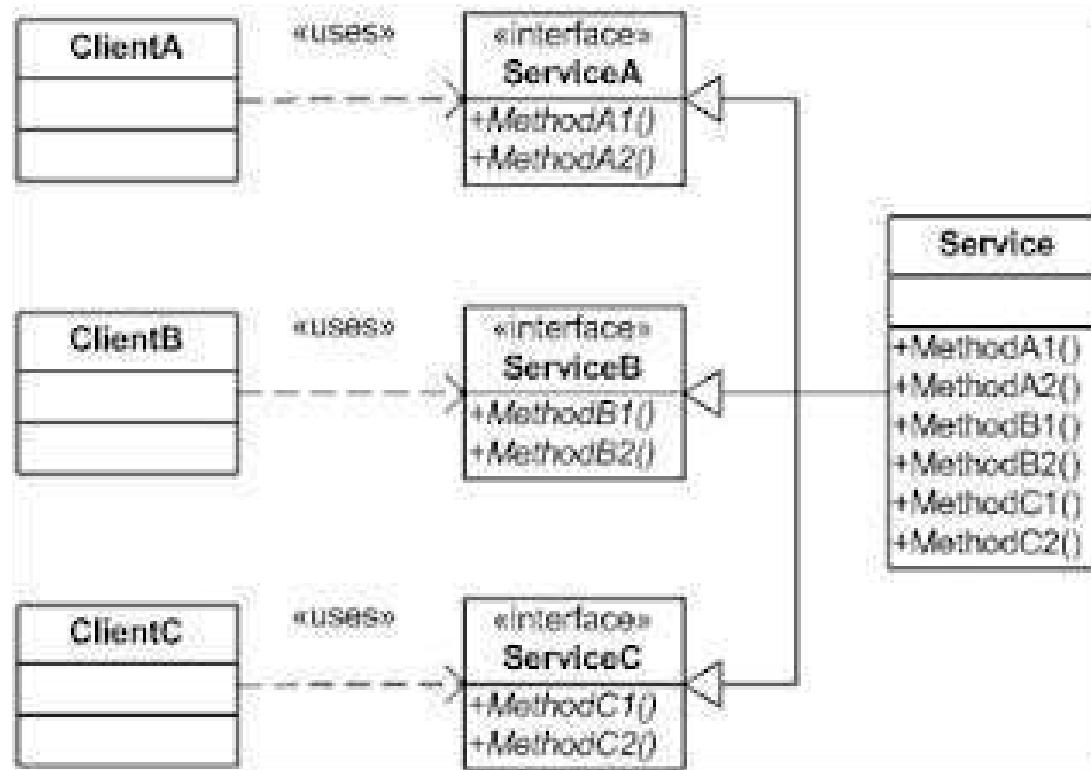
- Todos los clientes usan la misma interfaz



# Aplicación de ISP

47

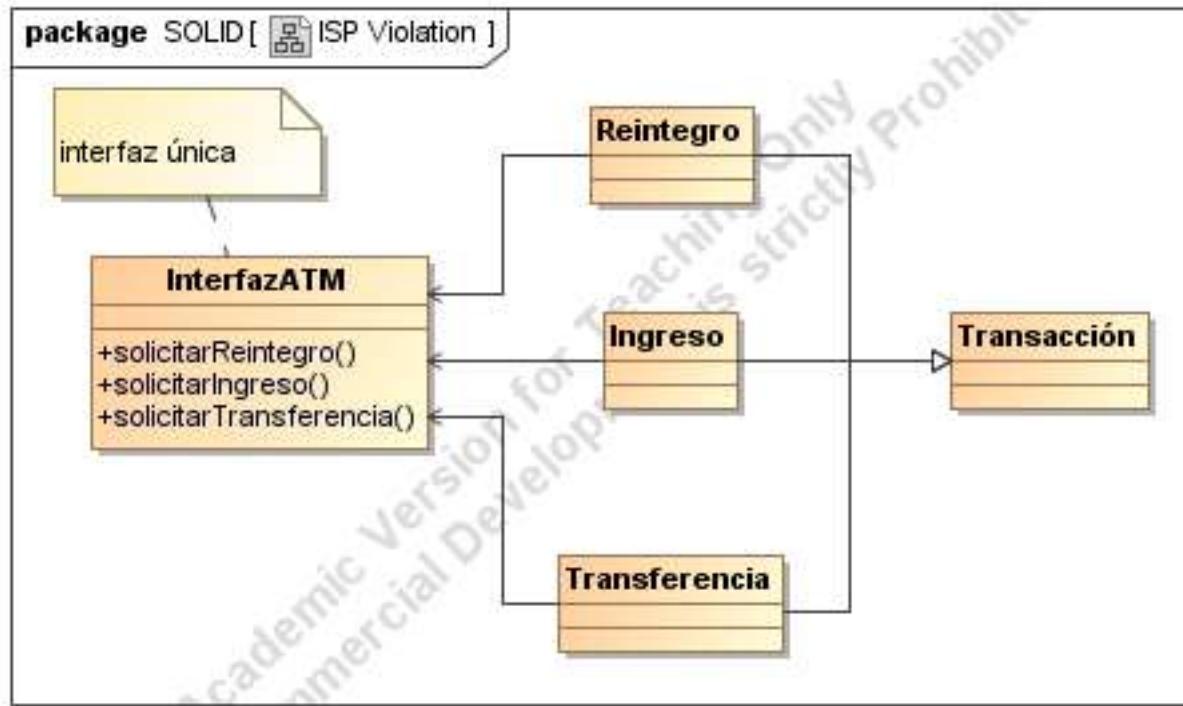
- Segregamos la interfaz según el cliente



# Violación de ISP (1)

48

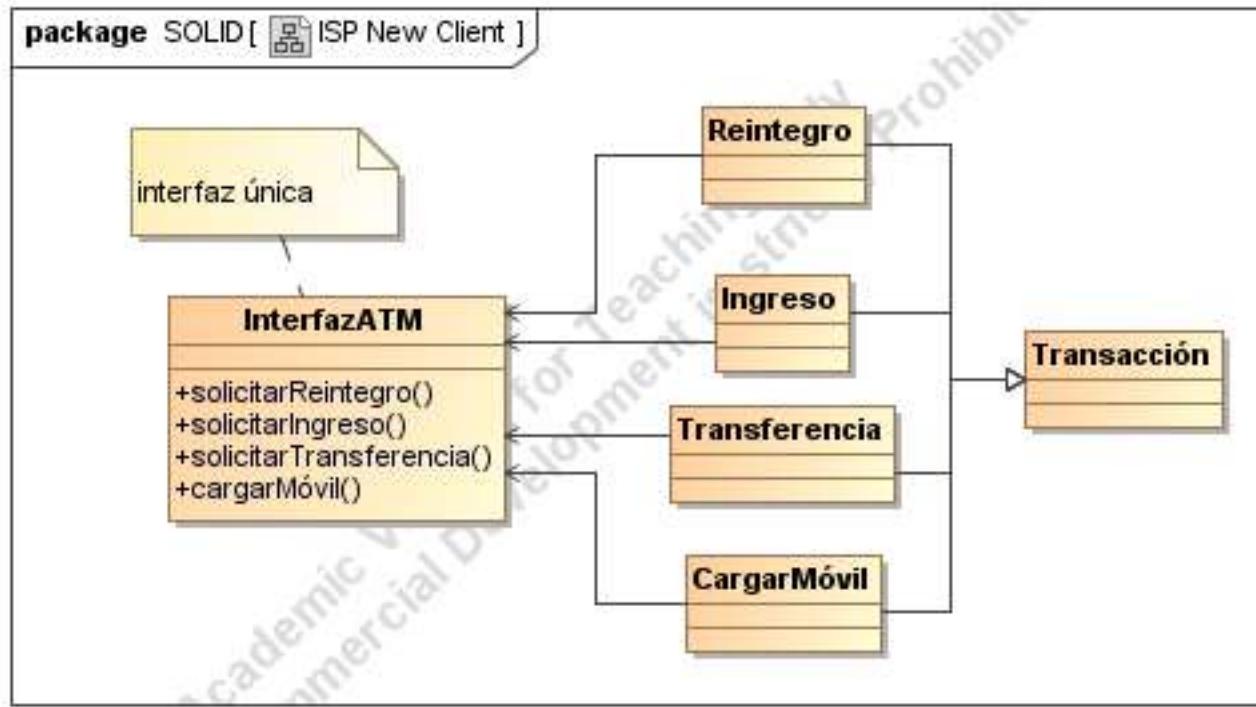
- Todos los clientes usan la misma interfaz



# Violación de ISP (2)

49

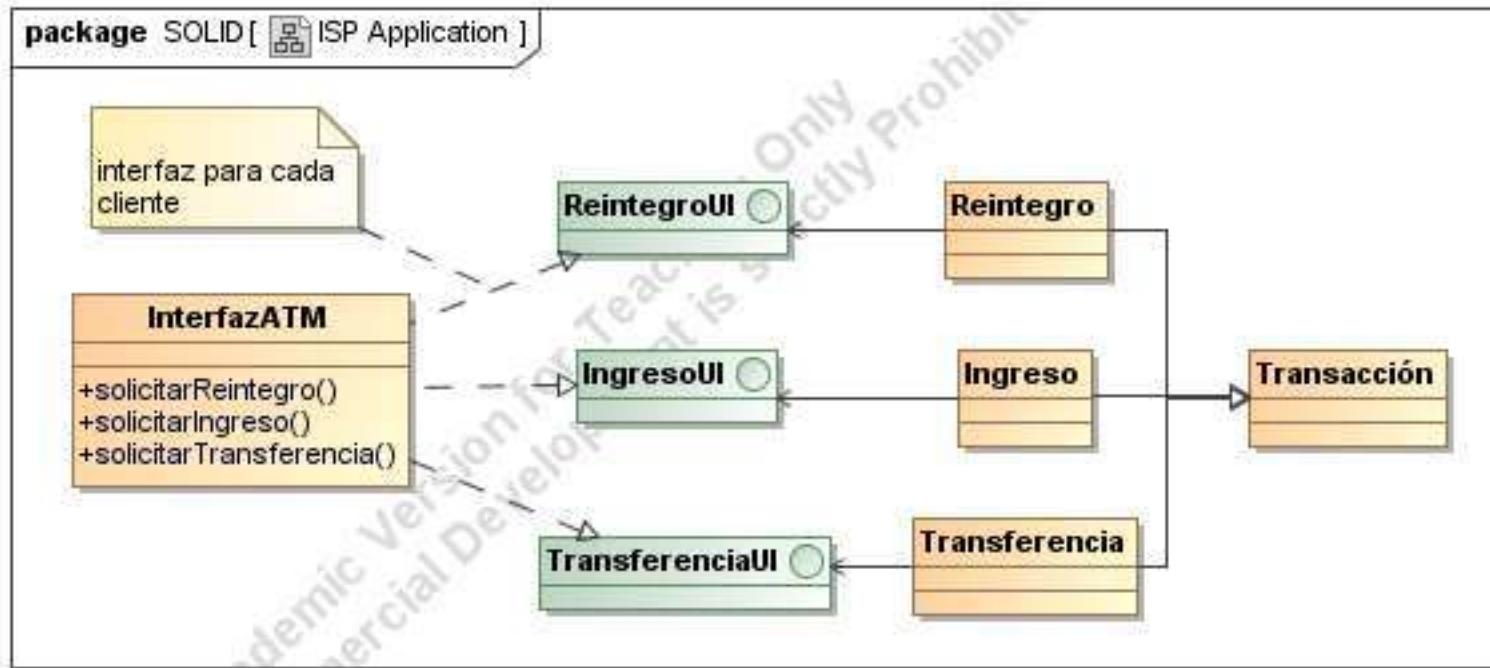
- Añadir un cliente puede modificar la interfaz



# Aplicación de ISP (1)

50

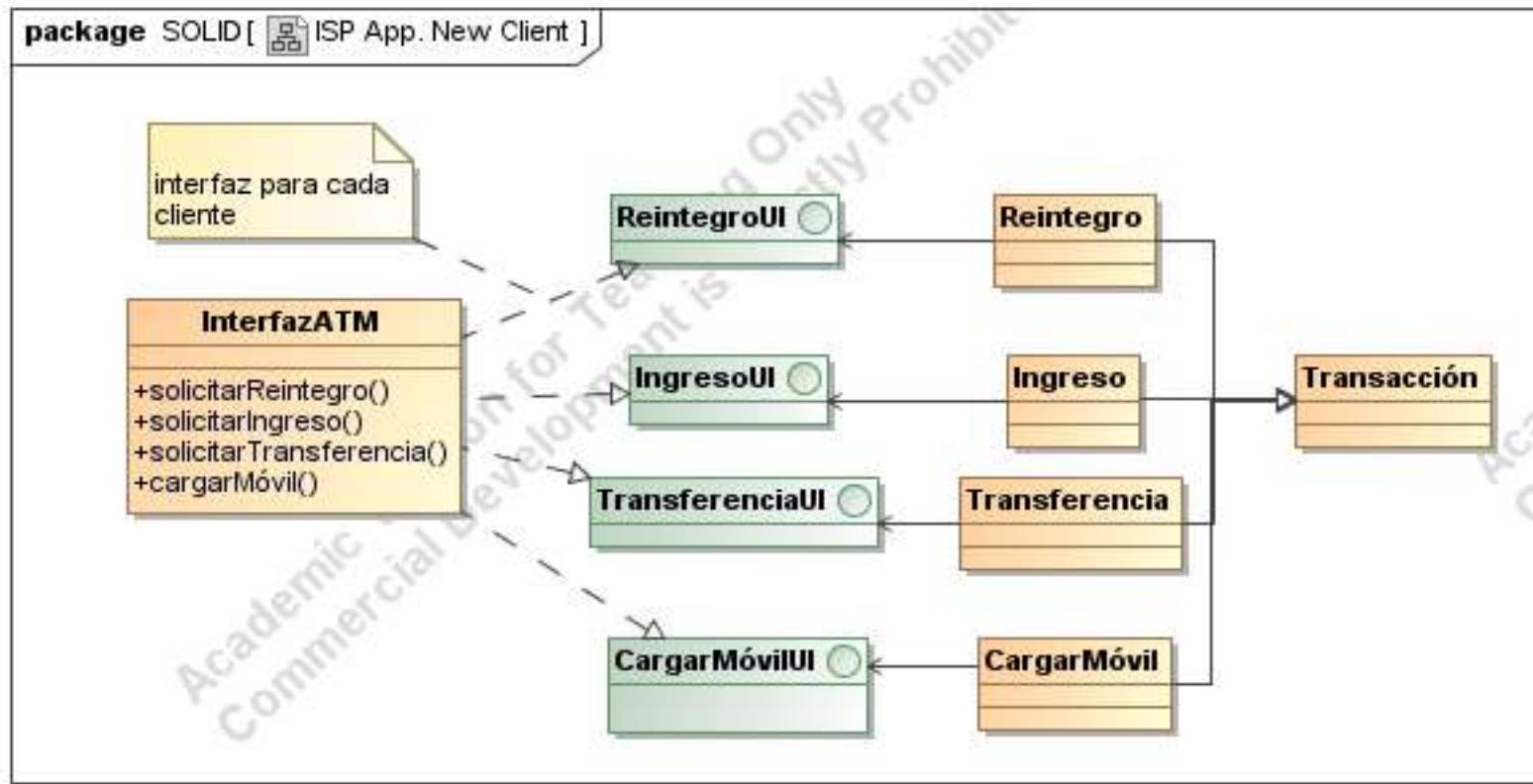
- Segregamos la interfaz según el cliente



# Aplicación de ISP (2)

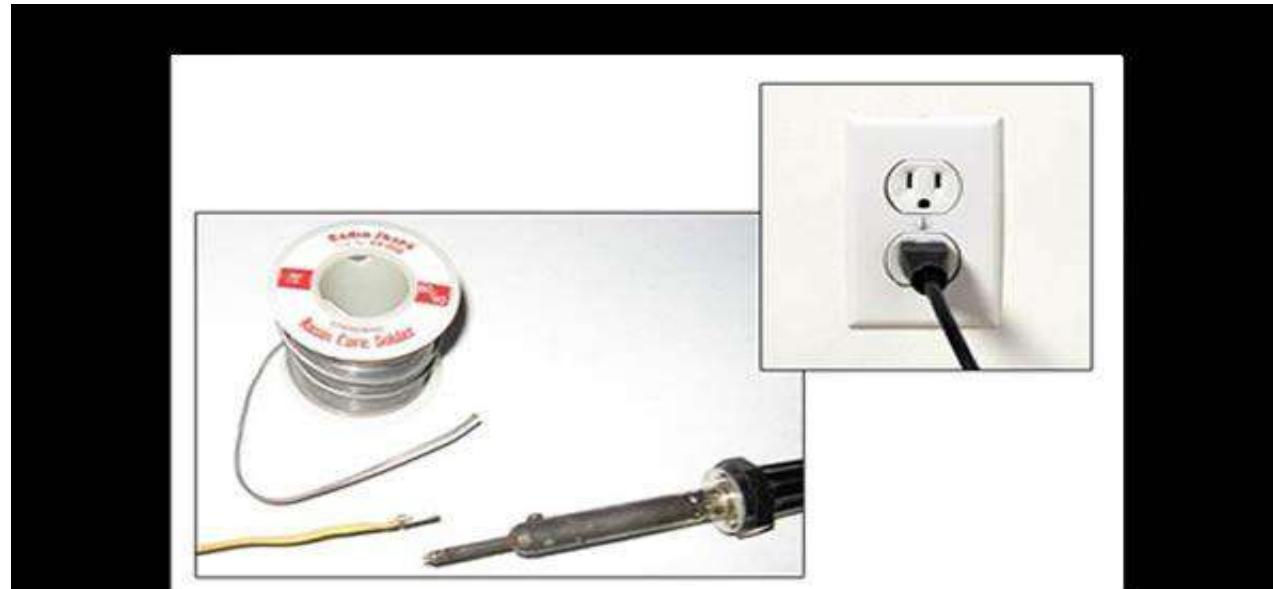
51

- Añadir un nuevo cliente no afecta a las interfaces de los demás clientes



# Principio de inversión de dependencias

52



## Dependency Inversion Principle

Would you solder a lamp directly  
to the electrical wiring in a wall?

**¿Soldarías una lámpara directamente al cableado eléctrico de una pared?**

# Principio de inversión de dependencias

53

*"Depende de abstracciones; no dependas de implementaciones"*

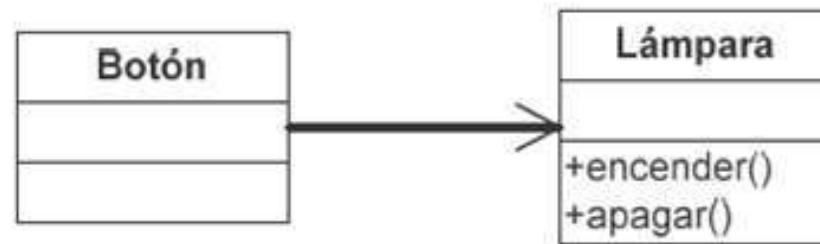
Robert C. Martin

- Los módulos de alto nivel no deben depender de módulos de menor nivel. Ambos deben depender de abstracciones
- Las abstracciones no deben depender de detalles. Los detalles deben depender de las abstracciones
- Lo concreto cambia más que lo abstracto

# Violación de DIP

54

- La clase Botón depende de Lámpara para su implementación

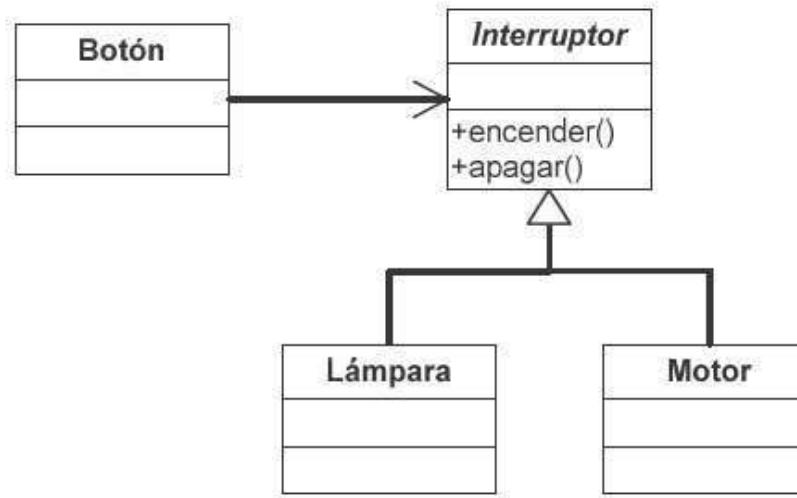


- **Problema:** ¿y si queremos añadir otro dispositivos controlado por un botón?
- **Solución:** crear una clase que abstraiga los dispositivos controlados por un interruptor.

# Aplicación de DIP

55

- La interfaz Interruptor es propiedad del módulo superior
- Ahora es posible añadir nuevas clases



# Otros principios de diseño OO

56

- Encapsula lo que probablemente cambiará
- Codifica respecto a interfaces, no respecto a implementaciones
- No te repitas; evita duplicaciones
- Principio de conocimiento mínimo o ley de Demeter:
  - Cada unidad debe tener un limitado conocimiento sobre otras unidades
  - Cada unidad debe hablar con sus amigos y no con extraños
  - Sólo hablar con amigos inmediatos

# Índice de contenidos

57

- Concepto de Diseño Software
- Principios de Diseño
- Principios de Diseño Orientado a Objetos
- **Patrones de Diseño**
- Refactorización

# A Pattern Language

58

*"Cada **patrón** **describe** un **problema** que **ocurre una y otra vez** a nuestro alrededor, y además **describe** el núcleo de la **solución a ese problema**, de tal manera que se puede utilizar esta solución un millón de veces, sin tener que hacer lo mismo dos veces. "*



Christopher Alexander *et al.*

# Concepto de patrón

59

- Los **patrones de diseño** plantean **soluciones** parciales a **problemas de diseño recurrentes**:
  - No hay que reinventar la rueda → reutilizar soluciones buenas
  - Cada patrón describe a un conjunto de objetos y clases comunicadas.
  - El conjunto se ajusta para resolver un problema en un contexto específico.
  - Proporcionan un vocabulario compartido por los diseñadores
- Son descripciones de **clases y objetos relacionados** para resolver un problema de **diseño general en contexto concreto**.
- Se basan en la **experiencia** previa
- Son un esqueleto básico que cada diseñador **adapta** a las peculiaridades de su situación
- Permiten construir sistemas con propiedades específicas

# Patrones de diseño y reutilización

60

- Los lenguajes orientados a objetos facilitan la **reutilización de código**, pero un **buen diseño** es la **clave** para una **reutilización** efectiva
- Un diseñador experimentado producirá diseños más simples, robustos y generales; fácilmente adaptables a cambios
- Los patrones de diseño pretenden explotar soluciones efectivas a determinados problemas

# Elementos de un patrón de diseño

61

Cada patrón tiene cuatro elementos:

- **Nombre**
  - Describe el problema y la solución en una o dos palabras.
- **Problema**
  - Explica el problema y su contexto: cuándo usar el patrón (en qué condiciones)
- **Solución**
  - No describe un diseño/solución concretos, sino una plantilla que se aplica a situaciones diferentes.
  - Elementos, responsabilidades, relaciones,..., sin implementación particular
- **Consecuencias**
  - Resultados y ventajas/inconvenientes de aplicar el patrón: espacio, tiempo, flexibilidad, portabilidad, críticas, costos y beneficios
  - Permite evaluar las alternativas de diseño.

# Clasificación de patrones

62

Según su **propósito** se clasifican en:

- Patrones de creación
  - Cuándo y cómo crear objetos
- Patrones estructurales
  - Cómo combinar objetos en otros objetos mayores
- Patrones de comportamiento
  - Cómo distribuir las responsabilidades y cómo comunicar los objetos

# Clasificación de patrones

63

Según su **ámbito** se clasifican en:

- **Patrones de Clases:** tratan relaciones entre las clases y sus subclases. Las relaciones se establecen por herencias y son estáticas en tiempo de compilación
- **Patrones de Objetos:** tratan las relaciones entre los objetos. Dichas relaciones pueden cambiar en el tiempo de ejecución y son dinámicas

# Catálogo de patrones de diseño (Gamma, 1995)

64

Propósito				
		<i>De creación</i>	<i>Estructurales</i>	<i>De comportamiento</i>
Ámbito	<i>Clase</i>	Factory Method	Adapter	Interpreter Template Method
	<i>Objeto</i>	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

# Patrones de creación

65

- Unitario (*Singleton*)
- Fábrica abstracta (*Abstract Factory*)
- Método de Fábrica (*Factory Method*)
- Constructor (*Builder*)
- Prototipo (*Prototype*)

# El patrón *Singleton* - Problema

66

- A veces necesitamos clases que tengan una única instancia. Por ejemplo:
  - contadores para asignar identificadores únicos
  - controladores de colas de impresión, dispositivos, etc.
- ¿Cómo aseguramos que sólo se crea una instancia?
- ¿Cómo proporcionamos un mecanismo de acceso global a esa única instancia?

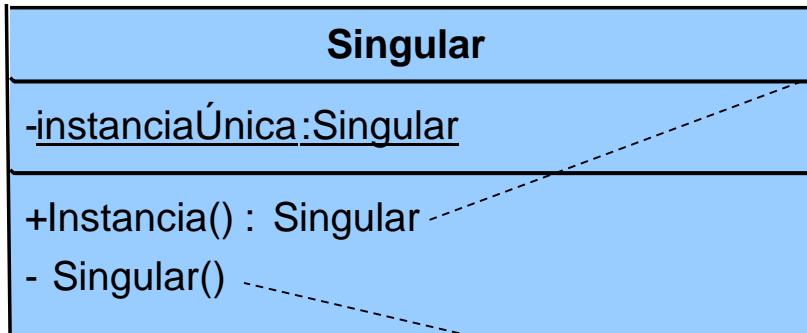
# El patrón *Singleton* - Solución

67

- Para asegurar que se crea una sola instancia
  - el constructor de la clase no debe ser público
- El almacenamiento de la instancia
  - se hará en una variable de clase (estática)
- El acceso a dicha instancia
  - se hará a través de un método de clase (estático) que devolverá una referencia a la instancia
- La creación de la instancia
  - se hará la primera vez que se invoque este método

# El patrón *Singleton* - Diagrama

68



```
Instancia (...) {  
    if (instanciaÚnica == null)  
        instanciaÚnica = new Singular(...);  
    return instanciaÚnica;  
}
```

El constructor debe ser privado.

# El patrón *Singleton* - Java

69

```
class Singular {  
    private static Singular instanciaÚnica;  
    ... // declaración de atributos  
  
    private Singular(...) {  
        ... // inicialización de atributos  
    }  
  
    public static Singular Instancia() {  
        if (instanciaÚnica == null)  
            instanciaÚnica = new Singular(...);  
        return instanciaÚnica;  
    }  
    // métodos de instancia  
}
```

# El patrón *Singleton* - Consecuencias

70

- No debe usarse en exceso, pues es similar a una variable global
  - pero no contamina el espacio global de nombres
- Se puede adaptar para que permita la creación y acceso de un *número acotado* de instancias
- La creación y acceso a la instancia *Singleton* deben serializarse en aplicaciones multihilo
  - Por ejemplo usando un monitor
- Introduce un estado global que dificulta el testeo unitario (*unit testing*)

# El patrón *Factory Method* - Problema

71

- A veces necesitamos crear instancias de clases que aún no conocemos:
  - Los *frameworks* y *toolkits* trabajan con clases abstractas de las que derivan clases concretas
  - Las clases concretas dependen de cada aplicación
  - El *framework* o *toolkit* no conoce las clases concretas que usará una aplicación
- ¿Cómo creamos instancias de las clases concretas que aún no conocemos?
- **Ejemplo:** framework para crear aplicaciones
  - El framework define 2 clases abstractas: Aplicacion y Documento.
  - Los clientes heredan de ellas para llevar a cabo su implementación específica.
  - La clase Aplicacion no conoce la subclase específica de Documento a instanciar para una aplicación determinada

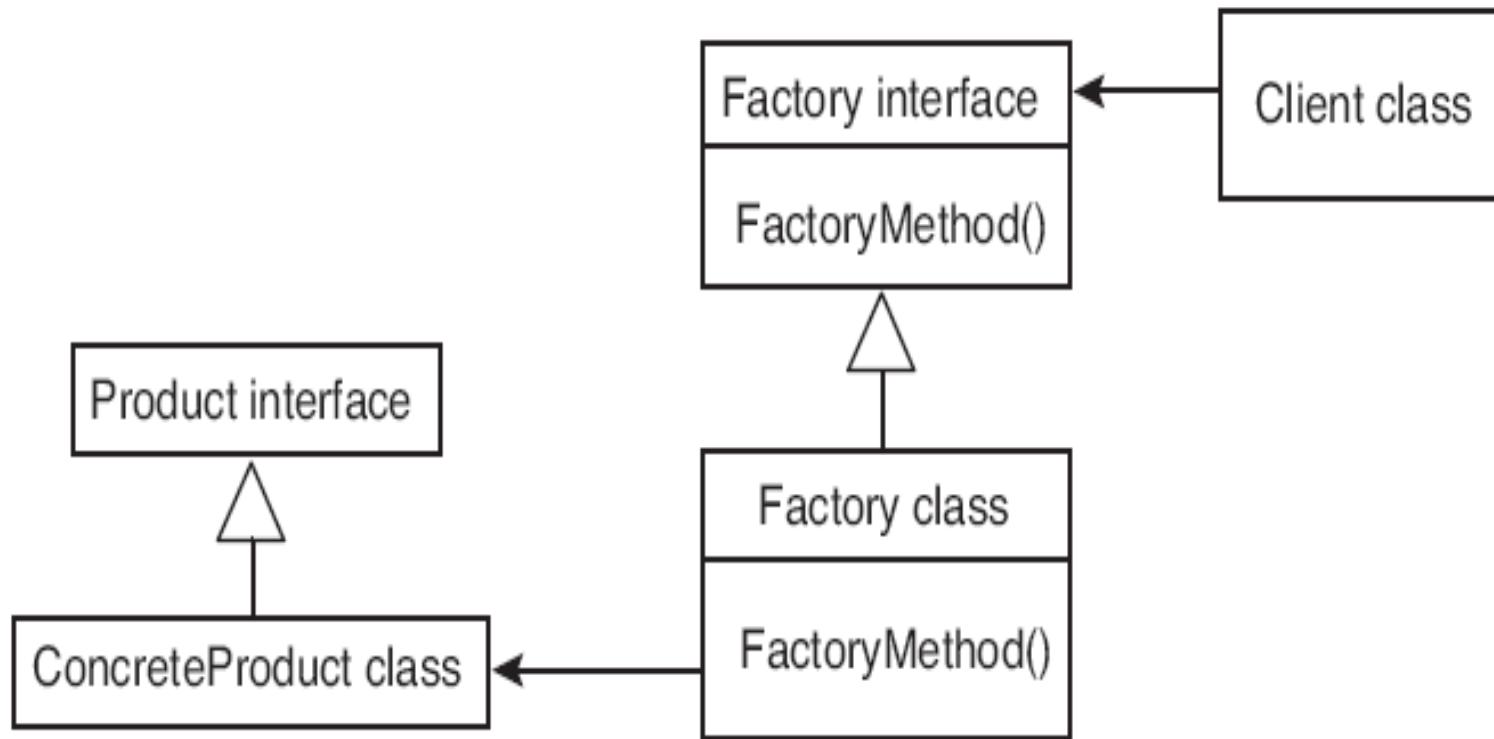
# El patrón *Factory Method* - Solución

72

- Creamos instancias a través de una interfaz de fabricación que delega en sus subclases:
  1. Definimos una interfaz de fabricación
    - incluye el *factory method*
  2. Las subclases implementan la interfaz de fabricación
    - implementan el *factory method*
- Se comporta como un constructor virtual
- **Ejemplo:** subclases de Aplicacion redefinen método CrearDocumento()
  - Devuelve la subclase de Documento adecuada para esa aplicación

# El patrón *Factory Method* - Diagrama

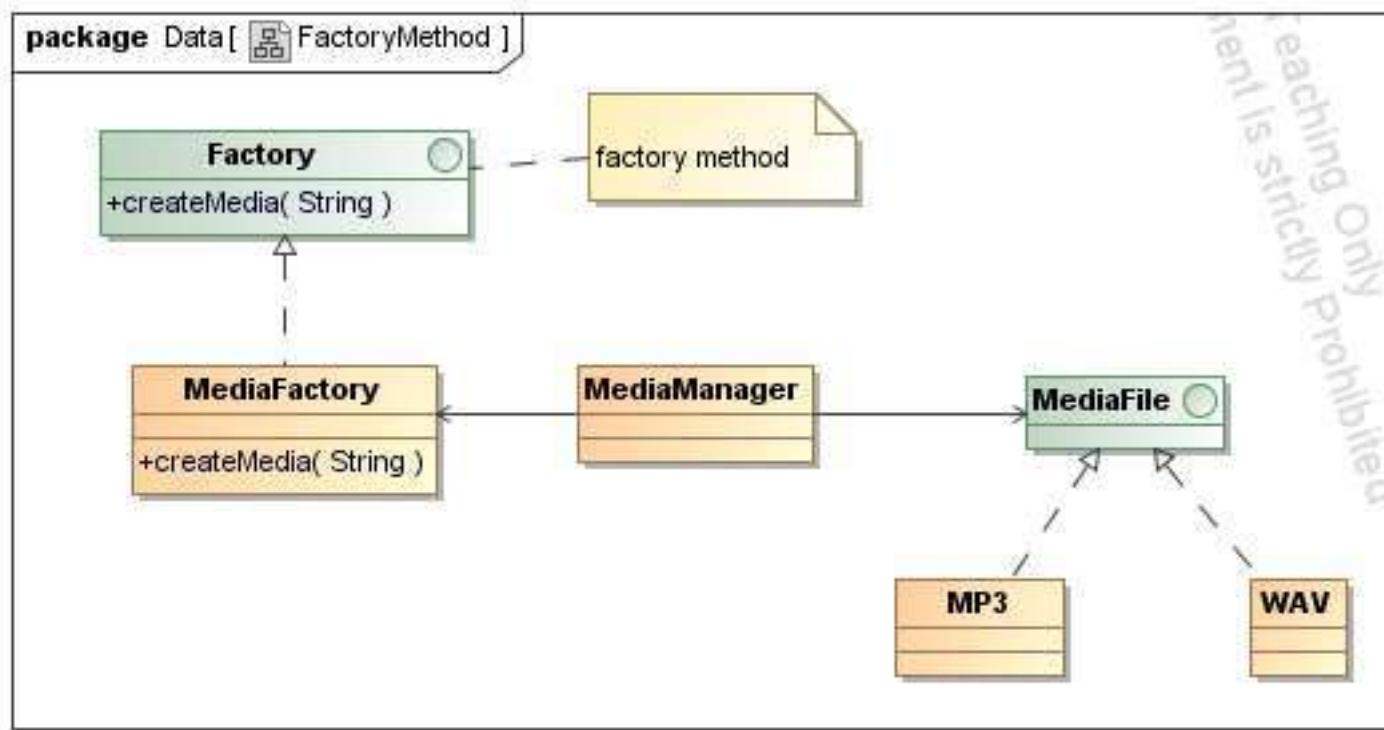
73



# El patrón *Factory Method* – Ejemplo

74

- La aplicación **MediaManager** sólo declara **MediaFile** y no usa **new**, sino **createMedia()**



# El patrón *Factory Method* - Java

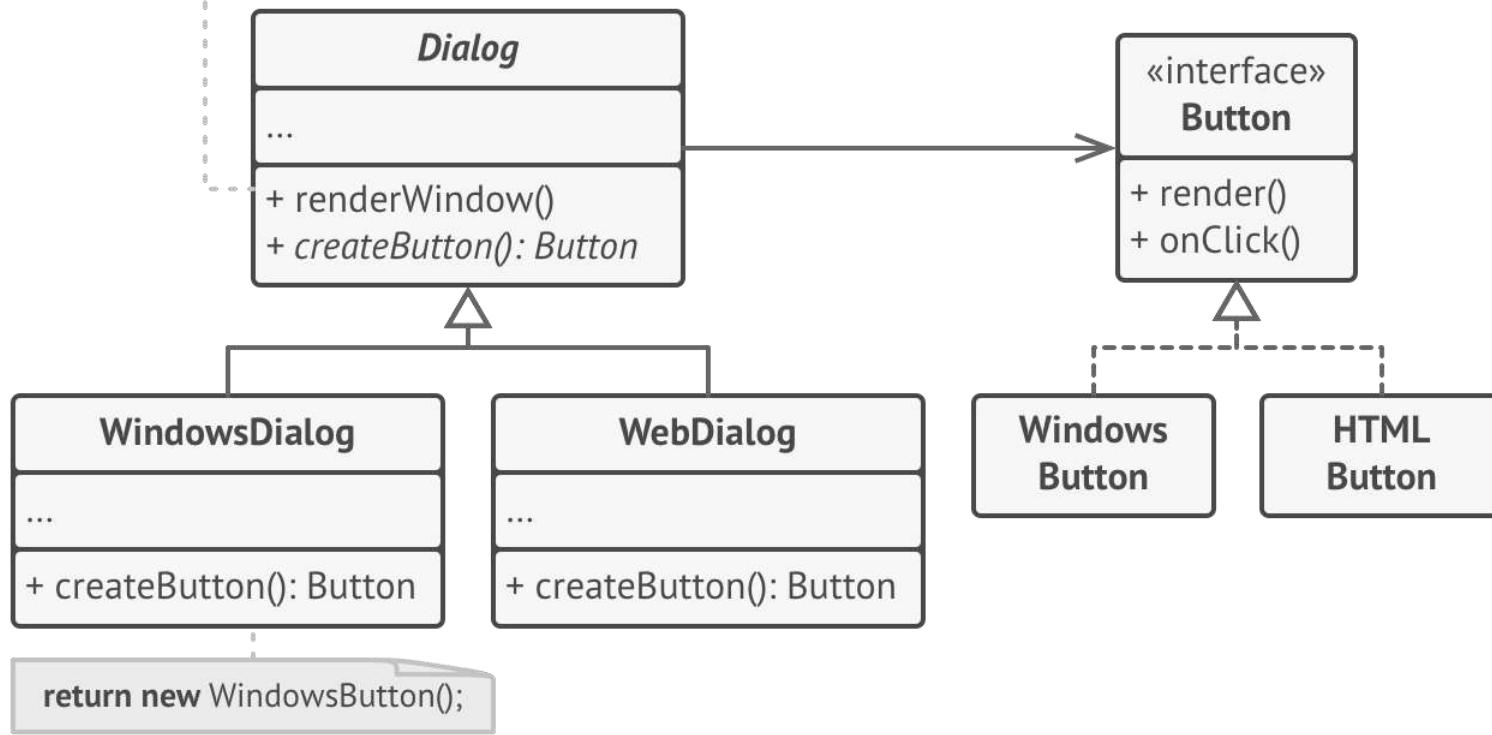
75

```
public interface Factory {  
    public MediaFile createMedia(String type);  
}  
  
Public class MediaFactory implements Factory {  
    public MediaFile createMedia(String type) {  
        if (type.equals("mp3")) return new MP3();  
        if (type.equals("wav")) return new WAV();  
        return null;  
    }  
}
```

# El patrón *Factory Method* – Ejemplo 2

76

```
Button okButton = createButton();
okButton.onClick(closeDialog);
okButton.render();
```



# El patrón *Factory Method* - Consecuencias

77

- El código cliente no crea objetos (`new`) de clases concretas
- El código cliente no trabaja con clases concretas
- Los objetos devueltos por un *factory method* no tienen por qué ser nuevos
- Se comporta como un constructor, pero es polimórfico, con nombres más descriptivos y comportamientos enriquecidos

# Patrones estructurales

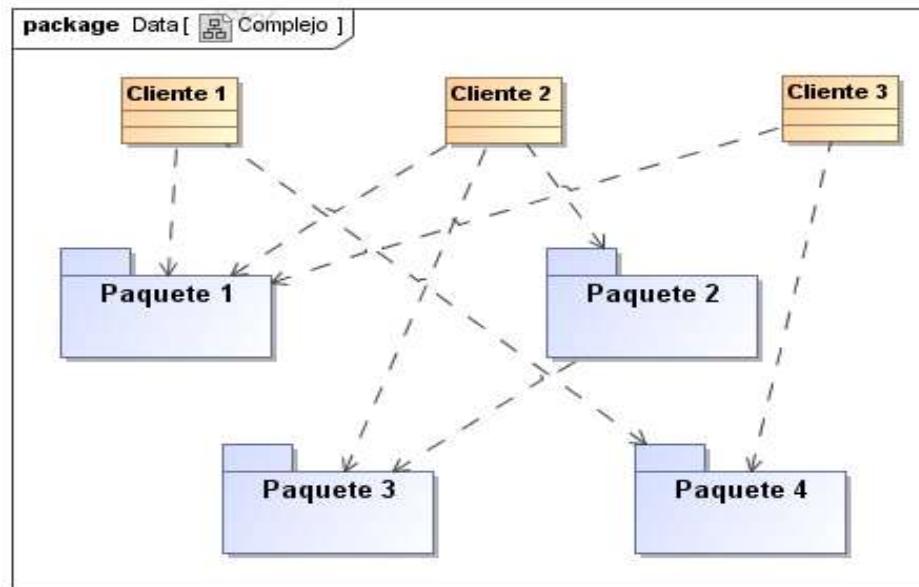
78

- Adaptador (*Adapter*)
- Puente (*Bridge*)
- Compuesto (*Composite*)
- Decorador (*Decorator*)
- Fachada (*Façade*)
- Peso mosca (*Flyweight*)
- Representante (*Proxy*)

# El patrón *Façade* - Problema

79

- A veces necesitamos trabajar con sistemas (*APIs, frameworks, toolkits*) muy complejos:

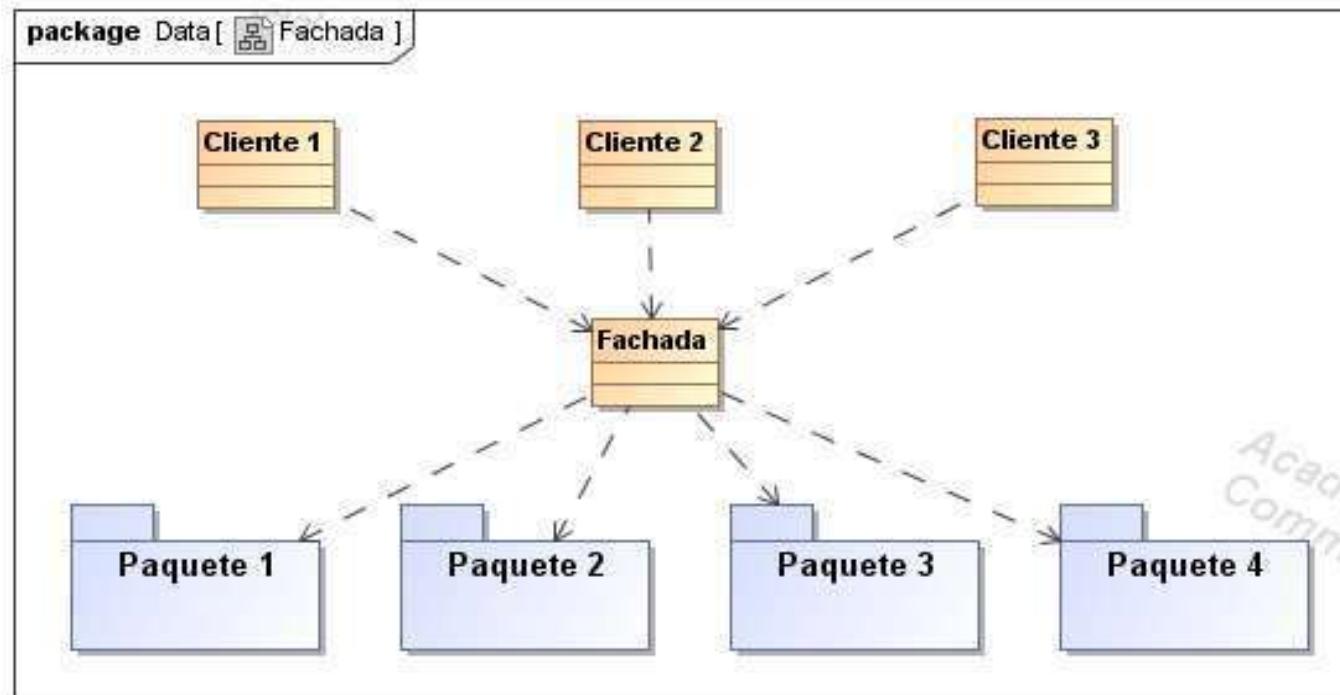


- ¿Cómo usar la funcionalidad básica ignorando la complejidad del sistema?

# El patrón Façade - Solución

80

- Definir una clase Fachada que facilite una **interfaz de alto nivel**, unificada y simplificada a todas las interfaces del sistema



# El patrón *Façade* - Consecuencias

81

- Facilita una interfaz simple a un sistema complejo
- Permite disponer el sistema en capas
  - Cada fachada es un nivel de abstracción o capa
- Reduce el acoplamiento entre los clientes y los subsistemas
  - Aumenta la portabilidad, facilita mantenimiento
- Limita la funcionalidad
  - Pero se puede seguir accediendo al subsistema
- No añade nueva funcionalidad

# Patrones de comportamiento

82

- Cadena de responsabilidad (*Chain of responsibility*)
- Orden (*Command*)
- Intérprete (*Interpreter*)
- Iterador (*Iterator*)
- Mediador (*Mediator*)
- Memento (*Memento*)
- Observador (*Observer*)
- Estado (*State*)
- Estrategia (*Strategy*)
- Método plantilla (*Template Method*)
- Visitante (*Visitor*)

# El patrón *Strategy* - Problema

83

- A veces necesitamos elegir un algoritmo o comportamiento adecuado:

```
datos comprimir(datos f) {  
    switch(tipoCompresion) {  
        case zip : return ZIP(f);  
        case rar : return RAR(f);  
    }  
}
```

- ¿Cómo añadir nuevos algoritmos sin modificar el código?

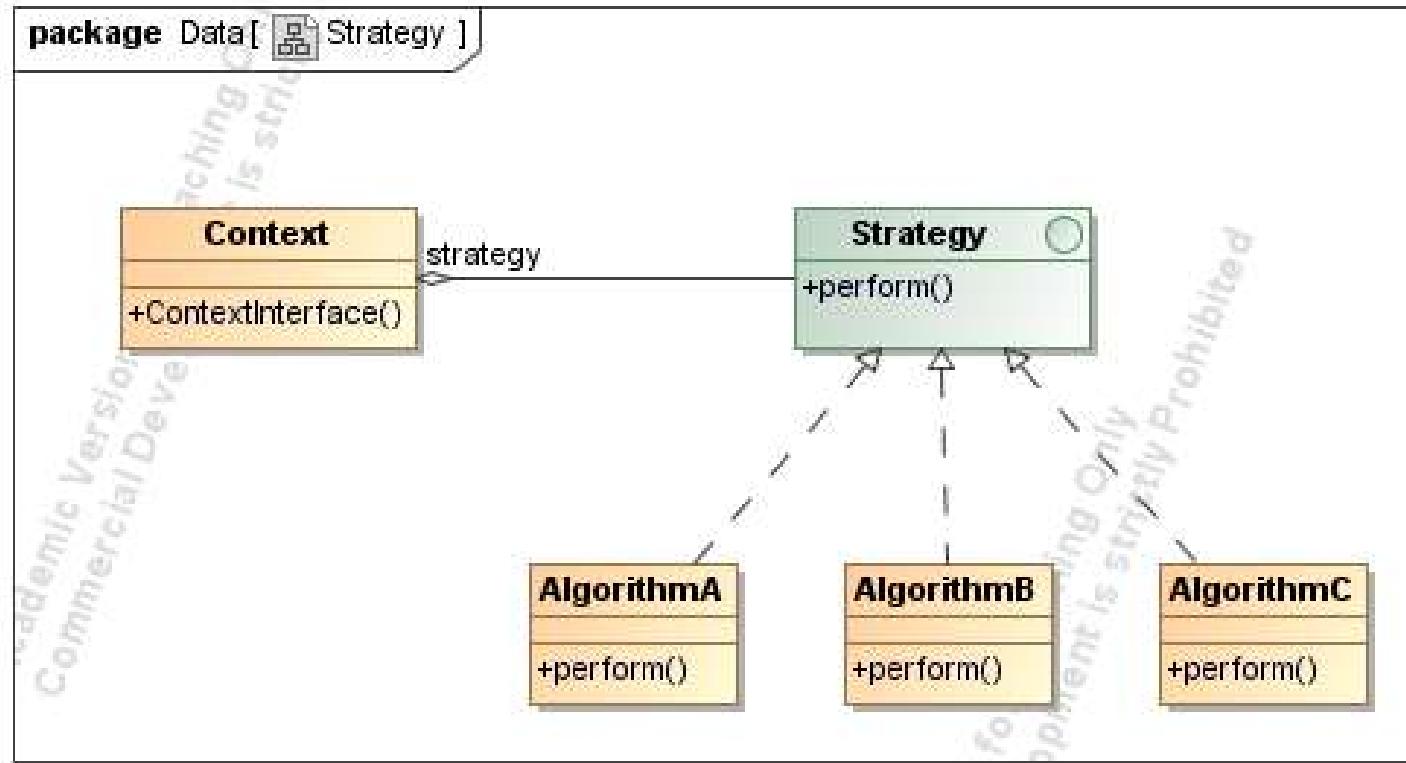
# El patrón *Strategy* - Solución

84

- Definir una **interfaz estrategia**
  - Todos los algoritmos implementan esta interfaz (estrategias concretas)
- Definir una **clase contexto**
  - Contiene una referencia a una estrategia concreta
  - Usa esta referencia para invocar el algoritmo elegido
- El código **cliente**
  - Crea la estrategia concreta (elige el algoritmo)
  - Configura el contexto (estrategia concreta y datos de entrada)
  - Se comunica con el contexto exclusivamente

# El patrón *Strategy* - Diagrama

85



# El patrón *Strategy* - Estrategias

86

```
public interface Compresor {  
    public datos comprimir(datos f);  
}  
  
public class ZIP implements Compresor {  
    public datos comprimir(datos f) {...}  
}  
  
public class RAR implements Compresor{  
    public datos comprimir(datos f) {...}  
}
```

# El patrón *Strategy* - Contexto

87

```
public class Contexto {  
    private Compresor estrategia;  
    public Contexto(Compresor comp){  
        estrategia= comp;  
    }  
    public setEstrategia(Compresor comp){  
        estrategia= comp;  
    }  
    public datos aplicar(datos f) {  
        return estrategia.comprimir(f)  
    }  
}
```

# El patrón *Strategy* - Cliente

88

```
public class Cliente {  
  
    datos original=...;  
    Contexto compresor;  
    compresor = new Contexto(new ZIP());  
    datos d1 = compresor.aplicar(original);  
    compresor.setEstrategia(new RAR());  
    datos d2 = compresor.aplicar(original);  
    ...  
}
```

# El patrón *Strategy* - Consecuencias

89

- Familia de algoritmos e implementaciones
- Alternativa a las construcciones condicionales
- Alternativa a la herencia
- El cliente debe conocer las estrategias
  - El cliente crea la estrategia concreta (`new`)
  - Solución: puede hacerlo el contexto (*factory method*)
- Comunicación entre contexto y estrategia
  - El contexto debe pasar la entrada a la estrategia
  - La estrategia debe devolver resultados al contexto
  - La estrategia puede tener una referencia al contexto

# Ventajas de los patrones de diseño (1)

90

- Son soluciones concretas:
  - Un catálogo de patrones es un conjunto de recetas de diseño
  - Aunque se pueden clasificar, cada patrón es independiente del resto
- Son soluciones técnicas:
  - Dada una determinada situación, los patrones indican cómo resolverla mediante un buen diseño
  - Existen patrones específicos para un lenguaje determinado, y otros de carácter más general
- Se aplican en situaciones muy comunes:
  - Proceden de la experiencia
  - Han demostrado su utilidad para resolver problemas que aparecen frecuentemente en el diseño

# Ventajas de los patrones de diseño (2)

91

- Son soluciones simples:
  - Indican cómo resolver un problema particular utilizando un pequeño número de clases relacionadas de forma determinada
  - No indican cómo diseñar un sistema completo, sino sólo aspectos puntuales del mismo
- Facilitan la reutilización de las clases y del propio diseño:
  - Los patrones favorecen la reutilización de clases ya existentes y la programación de clases reutilizables
  - La propia estructura del patrón es reutilizada cada vez que se aplica

# Inconvenientes de los patrones de diseño

92

- El uso de un patrón no se refleja claramente en el código:
  - A partir de la implementación es difícil determinar qué patrón de diseño se ha utilizado
  - No es posible hacer ingeniería inversa
- Es difícil reutilizar la implementación de un patrón:
  - Las clases del patrón son roles genéricos, pero en la implementación aparecen clases concretas
- Los patrones suponen cierta sobrecarga de trabajo a la hora de implementar:
  - Se usan más clases de las estrictamente necesarias
  - A menudo un mensaje se resuelve mediante delegación de varios mensajes a otros objetos

# Índice de contenidos

93

- Concepto de Diseño Software
- Principios de Diseño
- Principios de Diseño Orientado a Objetos
- Patrones de Diseño
- Refactorización

# Concepto de refactorización

94

- Cambios realizados en la estructura interna de un producto software
  - para facilitar su comprensión
  - para hacer menos costosa su modificación
  - sin cambiar su comportamiento observable
- Suele involucrar una modificación pequeña del software
- El proceso de refactorización puede implicar varios cambios consecutivos

# Ventajas de la refactorización

95

- **La refactorización:**

- mejora el diseño del software
- facilita su comprensión
- ayuda a encontrar errores
- ayuda a pasar del diseño a la implementación de manera más “ágil”

# ¿Cuándo refactorizar?

96

- Aplicaremos refactorización cuando:
  - se añada funcionalidad al sistema
  - se necesite arreglar un error
  - se haga una revisión de código
- Después de refactorizar se deben volver a pasar las pruebas

# Limitaciones de la refactorización

97

- Puede implicar cambios en la interfaz
  - mantener las interfaces originales junto con las nuevas
- Puede disminuir las prestaciones
  - eficiencia vs. legibilidad y mantenibilidad
- A veces es mejor empezar desde cero
  - mantener la integridad del diseño
- Ejemplo:
  - Las bases de datos son difíciles de refactorizar
    - ✖ cambios en el esquema fuerzan migraciones

# Candidatos a la refactorización (1)

98

- **Duplicación de código:**
  - en la misma clase
  - en distintas subclases de una clase dada
  - en clases no relacionadas directamente
- **Métodos excesivamente grandes:**
  - con muchos parámetros
  - con muchas variables temporales
  - con muchos bucles
- **Clases grandes:**
  - con muchas variables de instancia
  - con métodos duplicados

# Candidatos a la refactorización (2)

99

- Si un simple cambio produce muchos pequeños cambios en clases distintas
- Métodos que acceden en exceso a datos de clases ajenas
- Grupos cohesionados de datos
  - pueden constituir clases independientes
- Instrucciones de selección de casos
  - Vinculación dinámica
- Jerarquías de herencia paralelas
- Cambios divergentes
  - si una clase necesita modificarse de distintas formas, atendiendo a razones independientes
    - ✖ suele ser conveniente dividir la clase en varias

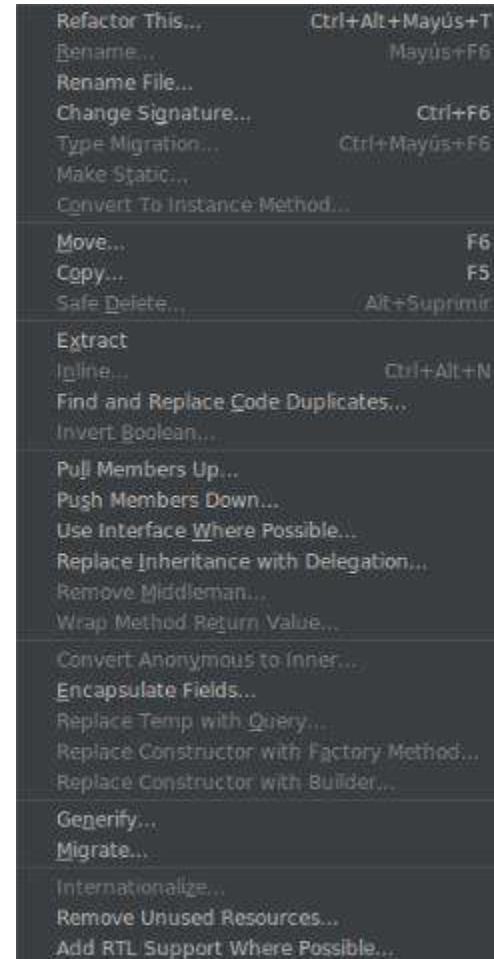
# Candidatos a la refactorización (3)

100

- Generalización especulativa
  - previsión de métodos que quizá puedan necesitarse en el futuro
- Uso excesivo de objetos intermediarios
- Alto grado de acoplamiento entre clases
  - a menudo motivado por un mal uso de la herencia
- Bibliotecas de clases incompletas
- Clases innecesarias

# Refactorizaciones en Android Studio

101



# Un catálogo de refactorizaciones (1)

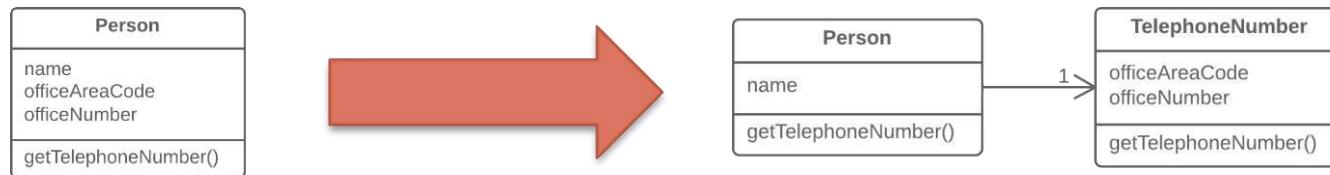
102

- Renombrar método
  - el nombre no ofrece información sobre su propósito
- Mover método
  - un método es usado más en otra clase que en su propia clase
- Extraer método
  - un fragmento de código se puede agrupar según algún criterio: definir método que lo encapsule
- Introducir asertos
  - una sección de código supone alguna propiedad sobre el estado del programa: hacerlo explícito con un assert

# Un catálogo de refactorizaciones (2)

103

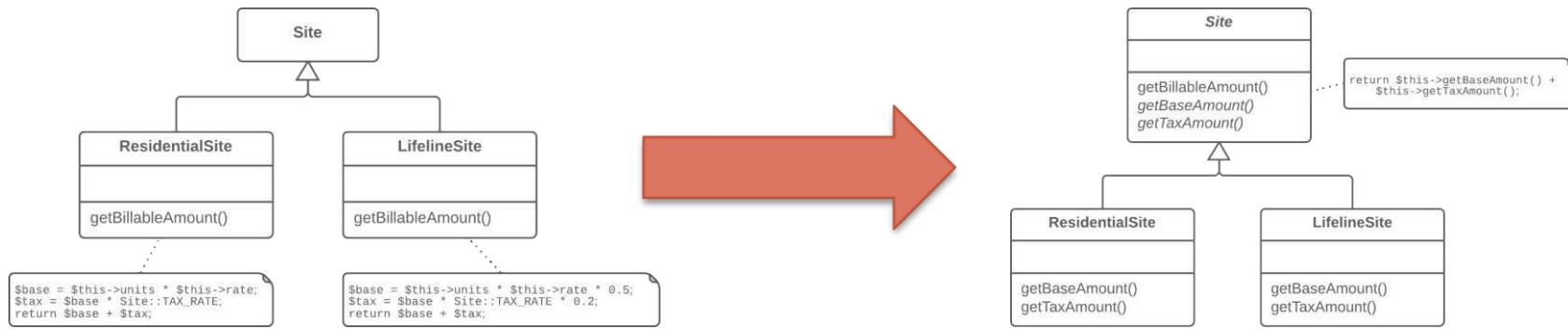
- Encapsular atributo
  - un atributo público debe definirse privado y proporcionar funciones de acceso (setter/getter)
- Preservar la unidad de los objetos
  - si se pasan como parámetros en una invocación datos sobre un mismo objeto, es conveniente enviar el objeto completo
- Extracción de clases
  - si una clase hace el trabajo que deberían hacer dos, debe crearse una nueva clase asociada a ella y trasladar allí los campos y métodos relevantes



# Un catálogo de refactorizaciones (3)

104

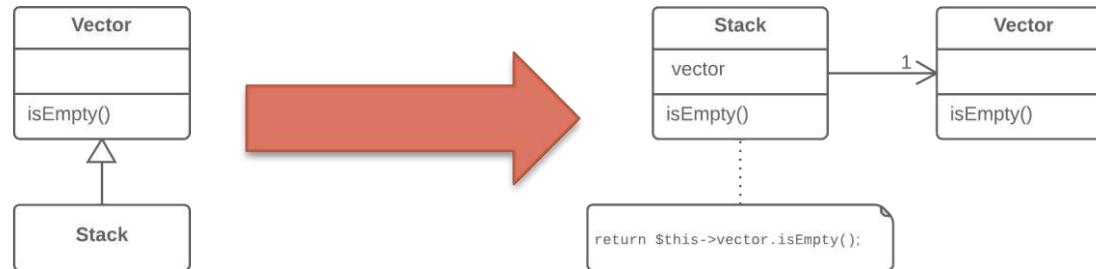
- Colapsar jerarquías de herencia
  - si una clase y su subclase no difieren significativamente, fundirlas en una sola
- Promoción de métodos hacia arriba
  - si dos métodos están definidos de forma similar en dos subclases, agruparlos en la clase padre común
- Construcción de una plantilla de método
  - si métodos en dos subclases realizan pasos similares en el mismo orden, pero los pasos son diferentes:
    - cada paso se encapsula en un método con igual firma, de forma que los métodos originales llegan a ser idénticos
    - entonces, dichos métodos pueden subir en la jerarquía



# Un catálogo de refactorizaciones (4)

105

- Cambiar asociación bidireccional a unidireccional
  - si se tiene una asociación bidireccional pero una clase no necesita características de la otra
    - ✖ eliminar el extremo innecesario de la asociación
- Reemplazar herencia con delegación
  - si una subclase utiliza solo parte de la interfaz de una superclase, o no quiere heredar datos
    - ✖ crear un campo para la superclase, ajustar métodos y delegar a la superclase, eliminando la subclasicación



# Un catálogo de refactorizaciones (5)

106

- Ocultar delegados

- si un cliente invoca una clase delegada de un objeto
    - ✖ crear métodos sobre el servidor, para ocultar el delegado (desacopla clases)



- Reemplazar condicional con polimorfismo

- si una estructura condicional selecciona el comportamiento de un objeto dependiendo del tipo
    - ✖ cambiar cada rama del condicional a un método redefinido en una subclase

- Introducir objeto nulo

- si se hacen comprobaciones repetitivas sobre un valor "null",
    - ✖ remplazar "null" con un objeto nulo.
    - ✖ ayuda a que el código sea más legible y corto.

# Para ampliar información

107

- Refactoring guru: <https://refactoring.guru/>

# Ejercicio para casa

108

- Describir alguno de los patrones de diseño que no hayamos visto en clase.
- Para el patrón seleccionado indicar:
  - Nombre
  - Problema que aborda
  - Solución propuesta
  - Consecuencias
  - Ejemplo: código fuente y/o diagrama de clases
- Subir el ejercicio al CV como un documento PDF.
- Bibliografía:
  - Gamma y cols., "Patrones de diseño", Addison Wesley.
  - Gamma y cols., "Design patterns: Elements of reusable object-oriented software", Addison Wesley.

# Introducción a la Ingeniería del Software



## TEMA 7: VERIFICACIÓN Y PRUEBAS

Grado en Ingeniería Informática  
Grado en Ingeniería del Software  
Grado en Ingeniería de Computadores



# Índice de contenidos

2

- Introducción
- Tipos de pruebas
- Pruebas de caja blanca
- Pruebas de caja negra
- *Test Driven Development* (TDD)

# Algunos errores de software famosos

3

- **NASA Mars climate orbiter, 1999 (\$125 millones)**
  - Bug: discrepancia entre unidades del sistema métrico y el sistema imperial.
- **Explosión del cohete Ariane 5 de la ESA, 1997 (\$500 millones)**
  - Bug: un *float* de 64 bits relacionado con la velocidad horizontal del cohete se convirtió en un entero de 16 bit causando un *overflow*.
- **Error de cálculo en los Intel Pentium, 1994 (\$500 millones)**
  - Bug: bug en la división de floats que provocó que Intel tuviese que reemplazar las CPUs afectadas
- **Error en los frenos de los Toyota, 2010 (\$3 mil millones)**
  - Bug: Toyota tuvo que retirar >400.000 vehículos debido a un bug que provocaba un retraso en el sistema anti-bloqueo de frenos.

# Conceptos básicos

4

- **Fallo software (bug):** El software no se comporta como se esperaba
- **Causas:**
  - *Requerimientos* incompletos, imprecisos o imposibles
  - *Especificación, diseño o código* incompletos o incorrectos.
- Las pruebas de software (**software testing**) se definen como:
  - El proceso que permite **identificar la corrección**, completitud, seguridad y calidad **del producto software desarrollado** antes de que éste sea usado.
  - **NO SON**
    - ✖ Un método para demostrar que NO hay errores
    - ✖ Un método para demostrar que el software funciona correctamente
- Una prueba en sí es:
  - El proceso de ejecutar ciertos componentes software con el fin de **encontrar errores**

# Conceptos básicos

5

- **Un caso de prueba es:**

- Conjunto de condiciones bajo las cuales se puede determinar si los requisitos del software se cumplen parcial o totalmente, o bien no se cumplen
- Un buen caso de prueba es aquel que tiene una **alta probabilidad de encontrar un error** no descubierto hasta entonces

- **Definición de error:**

- Discrepancia entre un valor o condición calculado, observado o medido y el valor o condición específica teóricamente correctos

# Depuración de errores

6

- **Tradicionalmente**

- La fase de pruebas de un proyecto tenía lugar después de la codificación y antes de la entrega del producto al cliente
- Inconvenientes: errores en etapas tempranas (requisitos, modelado) son muy costosos de reparar

- **Alternativa**

- Realizar pruebas en paralelo a la fase de desarrollo

- **Alternativa extrema**

- ***Test Driven Development***: las pruebas pasan a ser el centro del proceso de desarrollo

# Principios a seguir

7

- Saber el resultado esperado del programa es fundamental en un caso de prueba
- Cuanto más se modifique un programa, más hay que probarlo
- Es fundamental documentar bien los casos de prueba
- Definir los casos de prueba en la fase de diseño
- Un programador debe evitar probar su programa

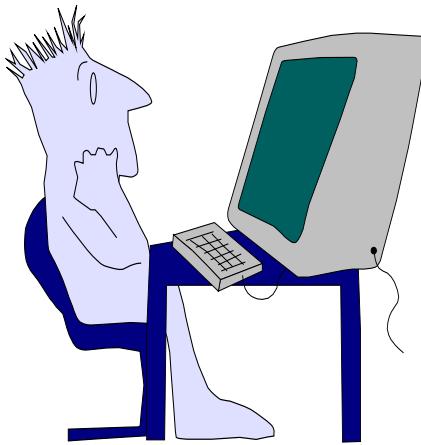
# ¿Quién prueba el software?

8

- Video

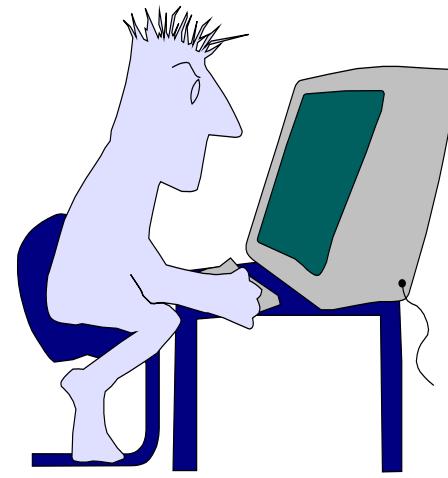
# ¿Quién prueba el software?

9



**Desarrollador**

Comprende el sistema pero probará “amablemente”. Está condicionado por la entrega



**Probadores independientes**

Debe comprender el sistema, pero intentará “romperlo”. Está dirigido por la calidad

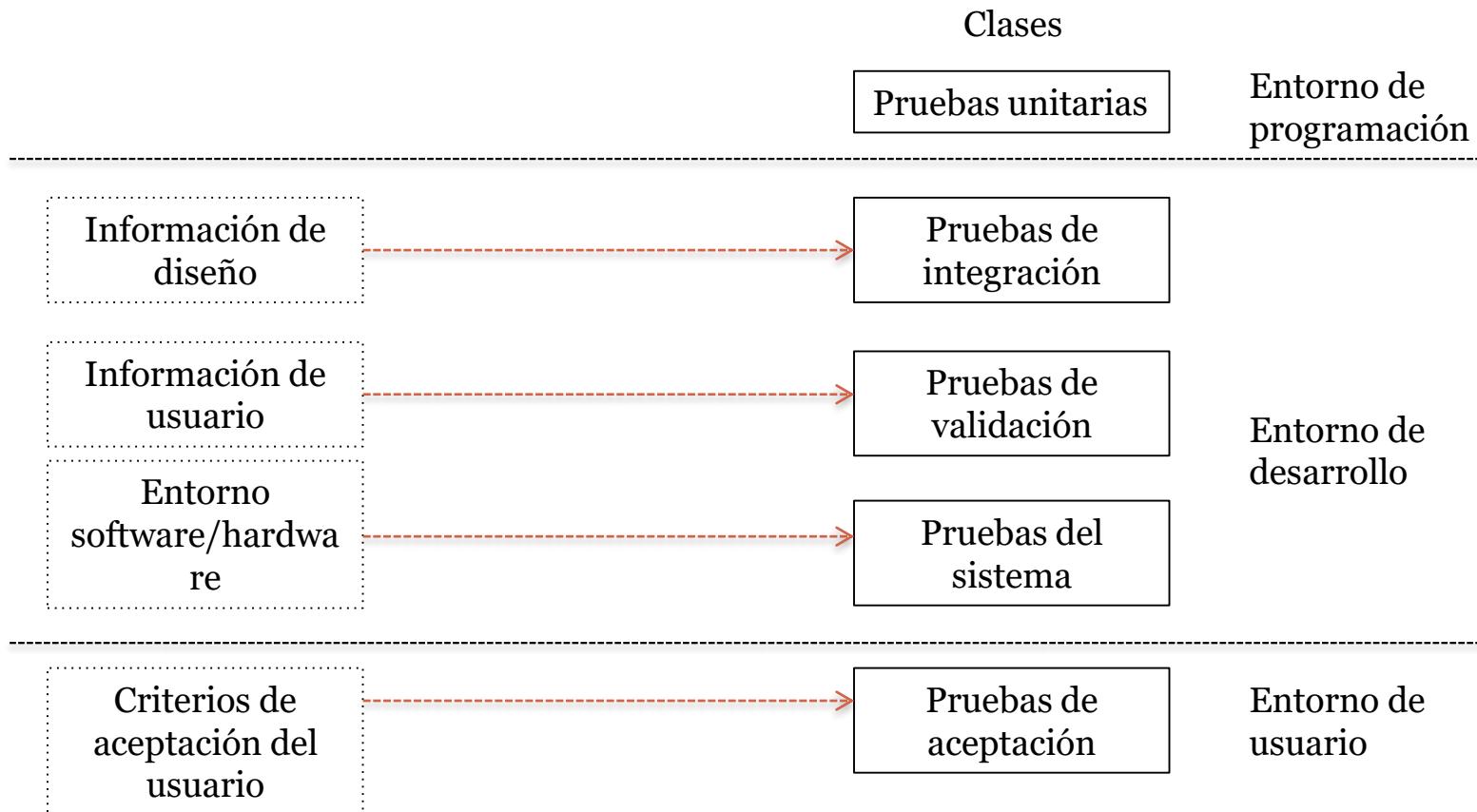
# Clasificación de tipos de pruebas

10

- **Pruebas unitarias** (de componentes aislados)
- **Pruebas de integración** (varios componentes a la vez)
- **Pruebas de validación** (¿se cumplen los requisitos? Pruebas alfa y beta)
- **Pruebas de sistema** (sistema completo)
- **Pruebas de aceptación** (validación por el usuario)
- **Pruebas de regresión** (después de un cambio, verificar que no se han introducido errores)
- **Pruebas de carga** (prueba bajo carga normal)
- **Pruebas de estrés** (prueba bajo alta carga)

# Relación entre los tipos de pruebas

11



# Herramientas

12

- Multitud de herramientas existentes:
  - Pruebas de unidad:
    - jUnit (Java): <http://www.junit.org/>
    - nUnit (.Net): <http://www.nunit.org/>
  - Creación de objetos Mock para pruebas:
    - jMock: <http://www.jmock.org/>
    - Mockito: <http://code.google.com/p/mockito/>
  - Generación automática de casos de prueba:
    - PEX (.Net): <http://research.microsoft.com/en-us/projects/pex/>
  - ...

# Pruebas de caja blanca y de caja negra

13

- Son dos tipos de pruebas relacionadas con la ejecución de código
- **Pruebas de caja blanca:**
  - Comprueban el **funcionamiento interno** y la lógica del código
- **Pruebas de caja negra:**
  - Son pruebas guiadas por los datos de entrada y de salida, sin tener en cuenta los detalles de implementación
  - No se dispone de acceso al código fuente

# Pruebas de caja blanca

14

- **Idea básica:**

- Asegurar **que todas las sentencias y condiciones** se han ejecutado al menos una vez

- Hay que diseñar casos de prueba que garanticen que:

- Se ejecuten por lo menos una vez todos los **caminos independientes** de cada módulo
  - Se comprueben los **bucles** en sus límites
  - Se comprueben todas las **decisiones lógicas** (V y F)
  - Se analicen **estructuras internas de datos**

Pruebas complementarias

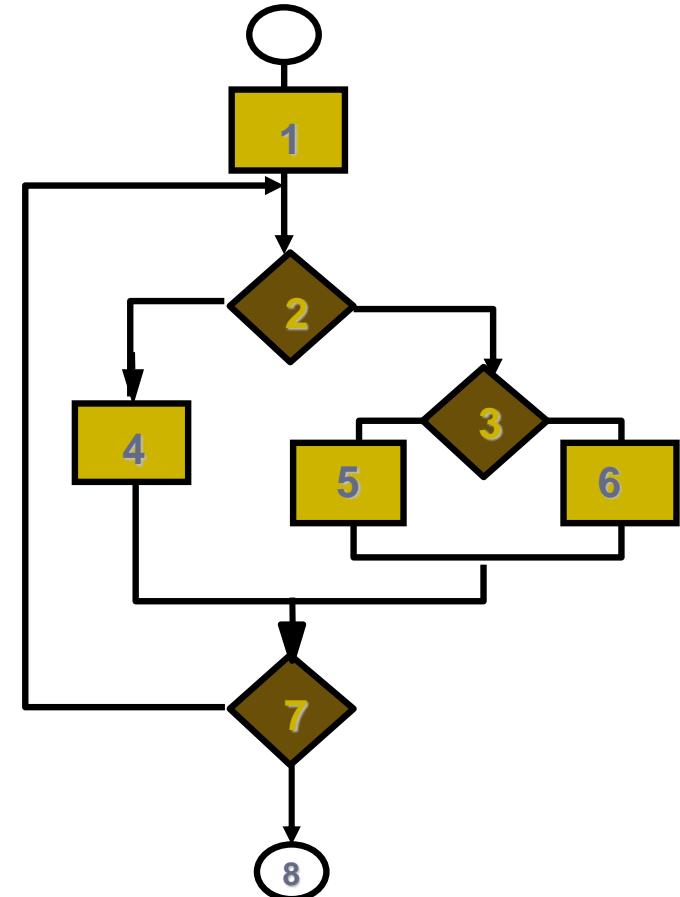
- **Cobertura de código:** Porcentaje del código sometido a pruebas

- Los caminos menos frecuentemente visitados son los que suelen contener mayor proporción de errores.

# Prueba de los caminos base

15

- **Idea:** diseñar casos de prueba que ejecuten cada **camino independiente** del código al menos una vez.
- **Pregunta:** ¿cómo determinamos el nº de caminos independientes? ➔ complejidad ciclomática del grafo de flujo.
- **Pasos:**
  1. Construir el *grafo de flujo*
  2. Determinar la **complejidad ciclomática**
  3. Usar esta medida para la definición de un conjunto de **caminos base** de ejecución.
  4. Elaborar **casos de prueba** que fuercen cada uno de los caminos base



# Prueba de los caminos base

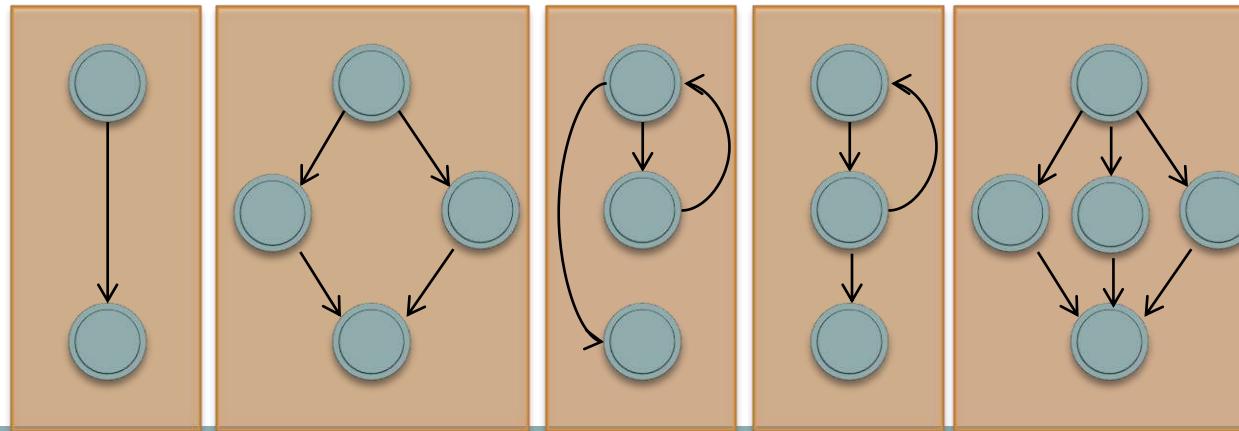
16

## • Complejidad ciclomática

- Es una métrica que proporciona una medición cuantitativa de la complejidad lógica de un programa
- Indica el número de caminos base de un grafo de ejecución de un programa

## • Camino base

- Camino linealmente independiente dentro del código



Notación de  
grafo de flujo

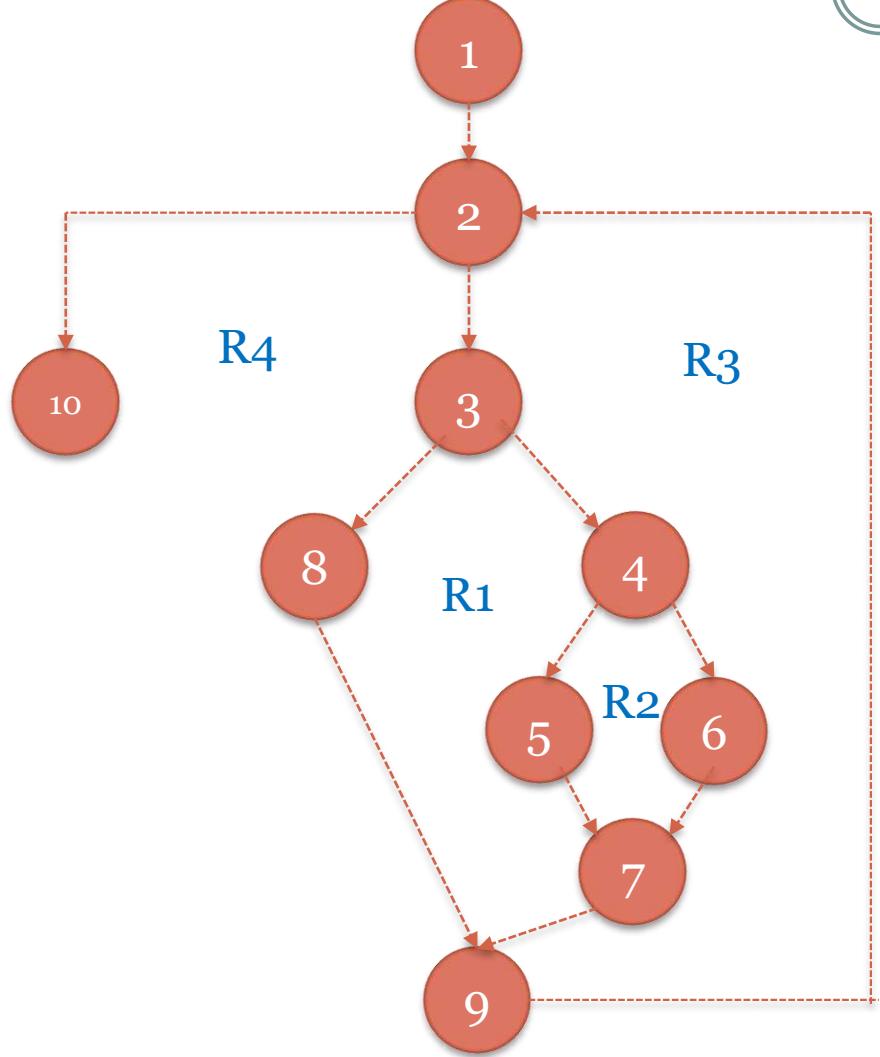
# Prueba de los caminos base

17

- **Complejidad ciclomática:** límite superior para el número de pruebas que se deben realizar para asegurar que se ejecutan todas las sentencias al menos una vez
- Se puede calcular de tres formas:
  - $V(G) = \text{Nº de regiones}$
  - $V(G) = \text{Aristas} - \text{Nodos} + 2$
  - $V(G) = P + 1$ , donde  $P$  es el número de nodos predicados (nodos que contienen una condición, por lo que dos o más aristas parten de ellos)
  - El número de caminos base coincide con la complejidad ciclomática
- **A mayor valor de complejidad ciclomática**
  - mayor probabilidad de errores

# Prueba de los caminos base

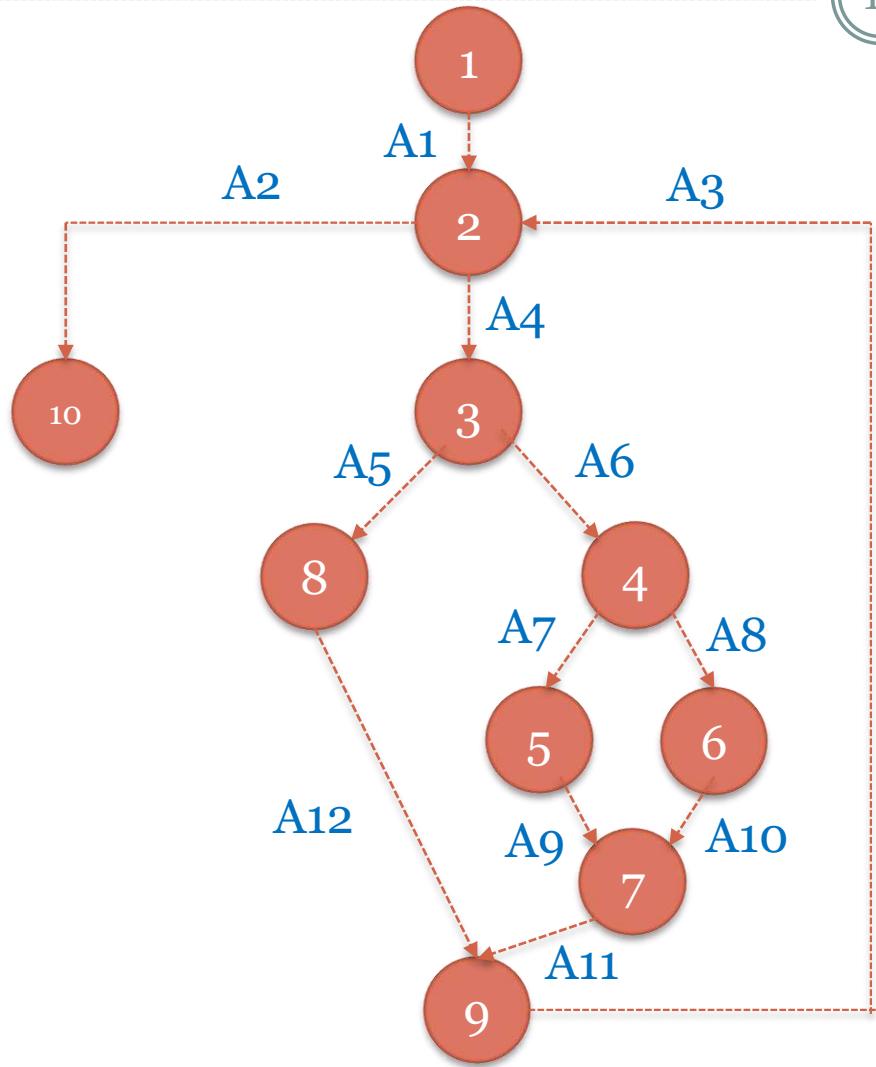
18



$$V(G) = \text{nº regiones} = 4$$

# Prueba de los caminos base

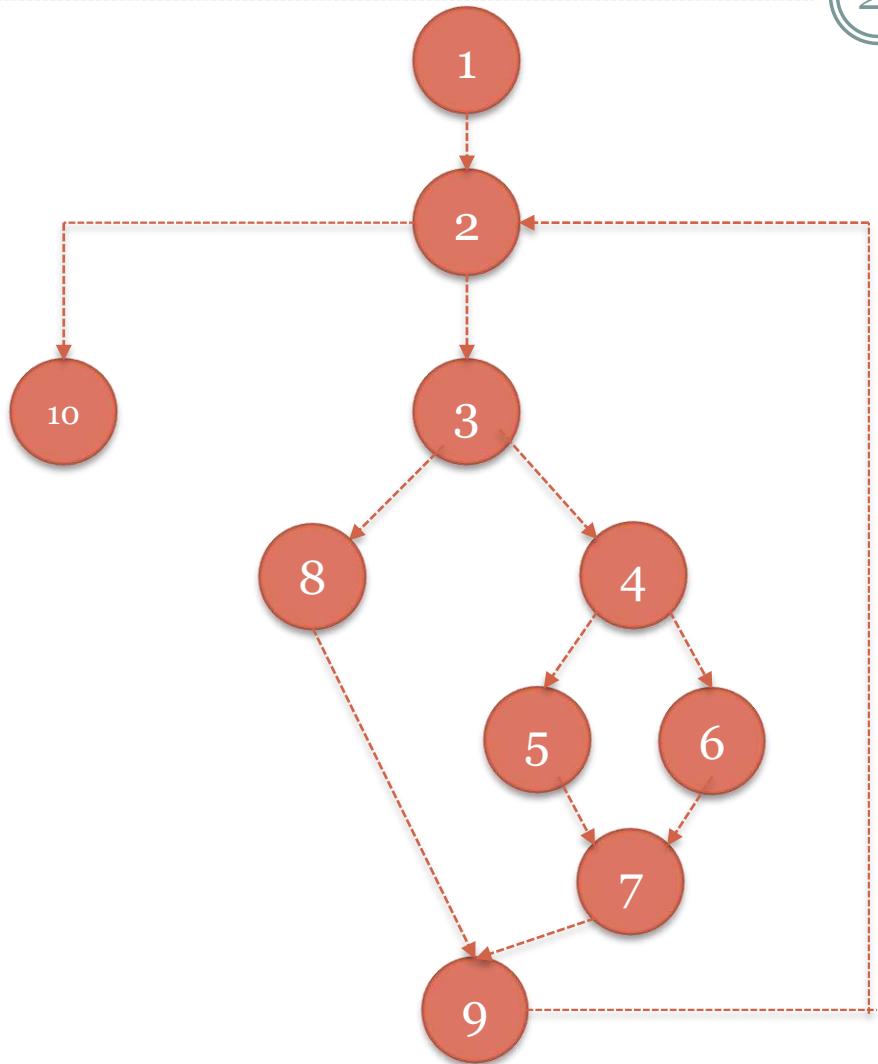
19



$$\begin{aligned}V(G) &= A - N + 2 \\N &= \text{nº nodos} = 10 \\A &= \text{nº Aristas} = 12 \\V(G) &= 12 - 10 + 2 = 4\end{aligned}$$

# Prueba de los caminos base

20



$$V(G) = |P| + 1$$

P = nodos predicados

$$P = \{2, 3, 4\}$$

$$V(G) = 3 + 1 = 4$$

- **Caminos base:**

- 1, 2, 10
- 1, 2, 3, 8, 9, 2, 10
- 1, 2, 3, 4, 6, 7, 9, 2, 10
- 1, 2, 3, 4, 5, 7, 9, 2, 10

- Los casos de prueba han de cubrir todos estos caminos
- Un analizador dinámico podría comprobar que estos caminos han sido ejecutados

# Ejemplo

21

Dada la siguiente función, que calcula el MCD de dos números, obtener el **grafo** correspondiente, la **complejidad ciclomática** y los **caminos base**:

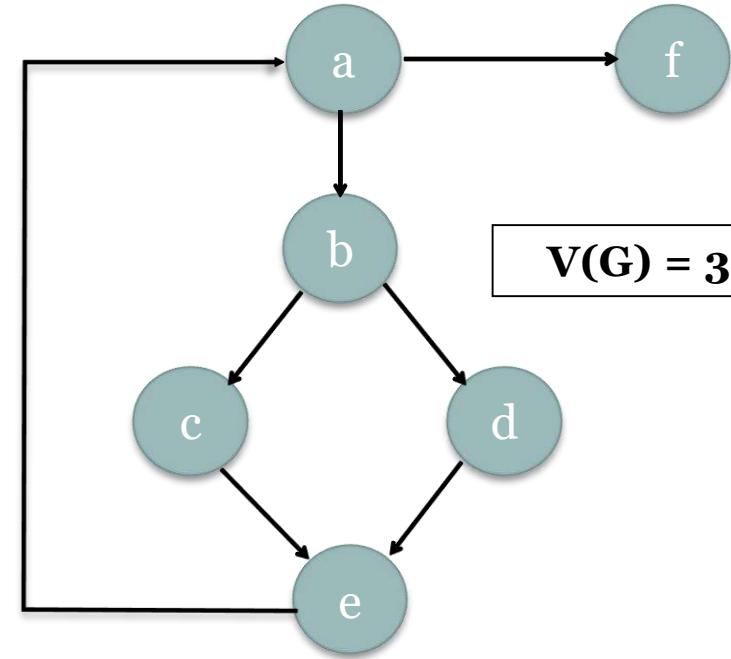
```
public int MCD (int x, int y) {  
    while (x != y) // a  
    {  
        if (x > y) // b  
            x = x - y; // c  
        else  
            y = y - x; // d  
    } // e  
    return x; // f  
}
```

# Ejemplo

22

## • Solución:

```
public int MCD (int x, int y) {  
    while (x != y) // a  
    {  
        if (x > y) // b  
            x = x - y; // c  
        else  
            y = y - x; // d  
    } // e  
    return x; // f  
}
```



### Caminos base

a, f

a, b, c, e, a, f

a, b, d, e, a, f

### Casos de prueba

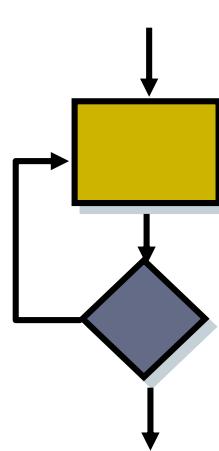
$\text{mcd}(x=1, y=1) \Rightarrow 1$

$\text{mcd}(x=4, y=2) \Rightarrow 2$

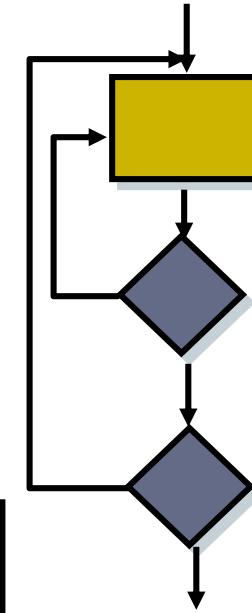
$\text{mcd}(x=2, y=4) \Rightarrow 2$

# Pruebas de bucles

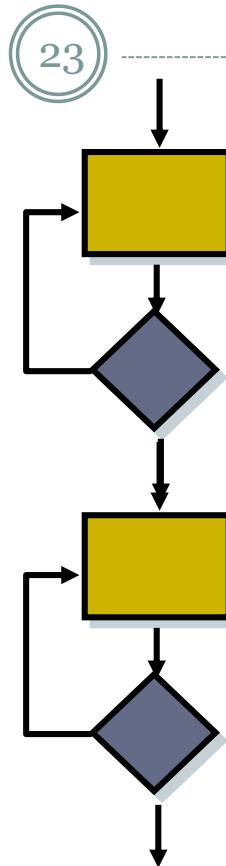
23



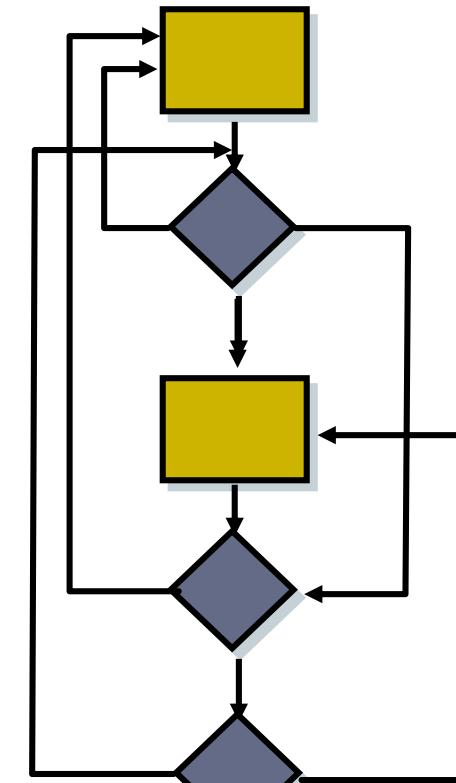
# Bucle Simple



# Bucles Anidados



# Bucles Concatenados

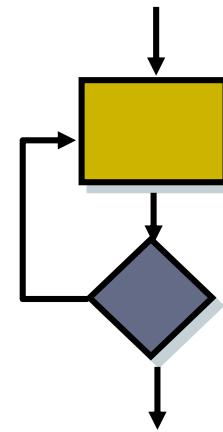


# Bucles no estructurados

# Pruebas de bucles

24

- Bucles simples
  - Siendo n el número máximo de pasos
    - Saltar el bucle
    - Pasar una sola vez
    - Pasar dos veces
    - Hacer m pasos, siendo  $m < n$
    - Hacer  $n - 1$ ,  $n$  y  $n+1$  pasos (**casos límite**)
  - Total: 7 pruebas



# Pruebas de bucles

25

## • **Bucles anidados**

- Impráctico seguir el esquema de bucles simples: el número de pruebas aumentaría geométricamente conforme el nivel de anidamiento
- Aproximación práctica:
  - Comenzar por el bucle más interior y los demás al mínimo
  - Pruebas de bucles simples para el bucle más interior, resto a mínimos
  - Progresar hacia fuera y para cada bucle realizar pruebas de bucles simples manteniendo los bucles externos en mínimos y los bucles internos en sus valores medios o típicos

# Pruebas de bucles

26

- **Bucles concatenados**
  - Si son independientes, se prueba como bucles simples
  - Si no, se prueban como bucles anidados
- **Bucles no estructurados**
  - Son bucles mal diseñados: rediseñar de forma correcta

# Pruebas de condición

27

- Las condiciones de una sentencia puede ser:
  - **Simples:**
    - Una variable lógica que se evalúa a verdadero o falso
    - Una expresión relacional del tipo `a op b`, donde `op` puede ser `>`,  
`>=`, `<`, `<=`, `=`, `<>`
  - **Compuestas:**
    - Varias condiciones simples, operadores lógicos (AND, OR, NOT) y paréntesis

# Pruebas de condición

28

- Tipos de pruebas relacionadas con condiciones y decisiones:
  - De cobertura de **condición** (condiciones simples)
  - De cobertura de **decisión** (condiciones compuestas)
  - De cobertura de **decisión/condición**
- En las decisiones hay que probar las ramificaciones:
  - Probar la rama verdadera
  - Probar la rama falsa
  - Probar cada condición simple

# Pruebas de condición. Ejemplo

29

```
public boolean comprobarHora(int hora, int minuto, int segundo)
{
    boolean horaTieneFormatoCorrecto = false;

    if ((hora >= 0) && (hora <= 23)) {
        if ((minuto >= 0) && (minuto <= 59)) {
            if ((segundo >= 0) && (segundo <= 59)) {
                horaTieneFormatoCorrecto = true
            }
        }
    }
    return horaTieneFormatoCorrecto ;
}
```

# Pruebas de condición. Ejemplo

30

- **Tres decisiones:**

- D1:  $((h \geq 0) \ \&\& \ (h \leq 23))$
- D2:  $((m \geq 0) \ \&\& \ (m \leq 59))$
- D3:  $((s \geq 0) \ \&\& \ (s \leq 59))$

- **Casos de prueba:** ejemplos de valores

	Verdadero	Falso
D1	h=10	h=24
D2	m=25	m=65
D3	s=12	s=70

# Pruebas de condición. Ejemplo

31

- **Tres decisiones:**

- D1: ( ( h >= 0 ) && ( h <= 23 ) )
- D2: ( ( m >= 0 ) && ( m <= 59 ) )
- D3: ( ( s >= 0 ) && ( s <= 59 ) )

- **Casos de prueba para cubrir las decisiones:**

- Caso 1: D1 = verdadero, D2 = verdadero, D3 = verdadero
- Caso 2: D1 = verdadero, D2 = verdadero, D3 = falso
- Caso 3: D1 = verdadero, D2 = falso
- Caso 4: D1 = falso

# Pruebas de condición. Ejemplo

32

- Tres decisiones, dos condiciones por decisión:
  - D1:  $( ( h \geq 0 ) \ \&\& \ ( h \leq 23 ) )$ 
    - C1.1:  $( h \geq 0 )$
    - C1.2:  $( h \leq 23 )$
  - D2:  $( ( m \geq 0 ) \ \&\& \ ( m \leq 59 ) )$ 
    - C2.1:  $( m \geq 0 )$
    - C2.2:  $( m \leq 59 )$
  - D3:  $( ( s \geq 0 ) \ \&\& \ ( s \leq 59 ) )$ 
    - C3.1:  $( s \geq 0 )$
    - C3.2:  $( s \leq 59 )$

# Pruebas de condición. Ejemplo

33

- Tres decisiones, dos condiciones por decisión
  - Hay que garantizar que cada condición tome al menos una vez el valor verdadero y otra el falso, garantizando la cobertura de la decisión

	<b>Verdadero</b>	<b>Falso</b>
C1.1	$h=10$	$h = -1$
C1.2	$h=10$	$h=24$
C2.1	$m=30$	$m=-1$
C2.2	$m=30$	$m=60$
C3.1	$s=50$	$s=-1$
C3.2	$s=50$	$s=70$

# Pruebas de condición. Ejemplo

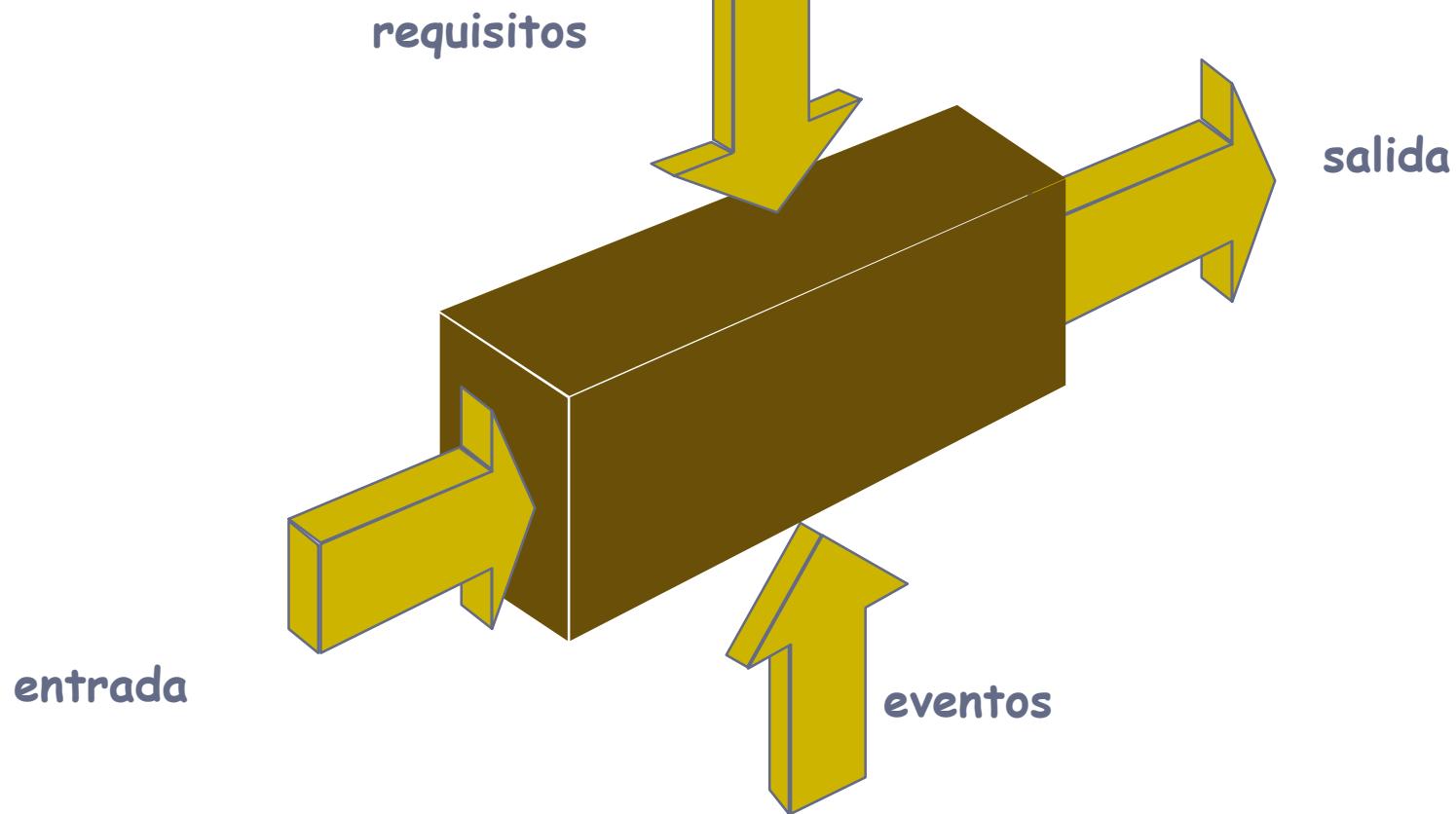
34

- Casos de prueba:

- Caso 1: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = V, C3.1 = V, C3.2 = V
- Caso 2: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = V, C3.1 = V, C3.2 = F
- Caso 3: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = V, C3.1 = F, C3.2 = V
- Caso 4: C1.1 = V, C1.2 = V, C2.1 = V, C2.2 = F
- Caso 5: C1.1 = V, C1.2 = V, C2.1 = F, C2.2 = V
- Caso 6: C1.1 = V, C1.2 = F
- Caso 7: C1.1 = F, C1.2 = V

# Pruebas de caja negra

35



# Pruebas de caja negra

36

- Se centran en los **requisitos funcionales** del software
  - Obtener conjuntos de condiciones de entrada que prueben todos los requisitos funcionales del programa
  - No es una alternativa a las pruebas de caja blanca
    - Es un **enfoque complementario**
    - Suelen descubrir tipos de errores diferentes a los obtenidos con las técnicas de caja blanca.

# Pruebas de caja negra

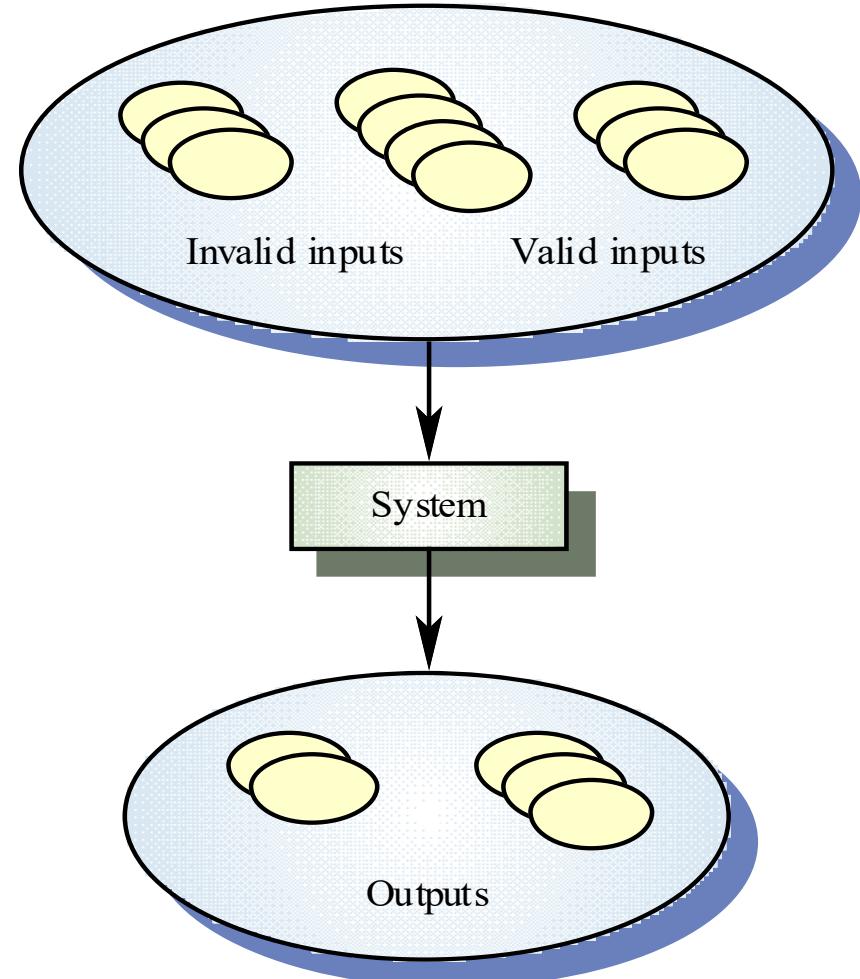
37

- Trata de encontrar errores en:
  - Funciones incorrectas o ausentes
  - Errores de interfaz
  - Errores en estructuras de datos o en accesos a bases de datos externas
  - Errores de rendimiento
  - Errores de inicialización y de terminación
- Ignora intencionadamente la estructura de control
- Se suele aplicar durante las fases finales de las pruebas

# Pruebas de caja negra

38

- Están conducidas por los datos de entrada y de salida
- Hay que conocer de antemano las **salidas correctas**
- **Dos técnicas:**
  - Partición de equivalencia
  - Análisis de valores límites



# Partición de equivalencia

39

- **Idea:**
  - Hay que definir casos que descubran clases de errores
  - Dividir el dominio de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba
- Evaluar clases de equivalencia para una condición de entrada
  - Una clase de equivalencia representa un conjunto de estados válidos o inválidos para condiciones de entrada

# Partición de equivalencia

40

- Se pueden definir de acuerdo con las siguientes directrices:
  - Un rango: define una clase de equivalencia válida y dos inválidas
  - Un valor específico: una válida y dos no válidas
  - Un miembro de un conjunto: una válida y una no válida
  - Lógica: una válida y una no válida

# Partición de equivalencia

41

- **Ejemplo 1:**

- Una entrada es un entero de 5 dígitos entre 10.000 y 99.999.
- La particiones equivalentes son
  - <10.000
  - 10.000 – 99.999
  - > 100.000
- Luego hay que elegir casos de prueba en los límites de estos conjuntos:
  - 00000, 09999, 10000, 99999, 100000, 100001

- **Ejemplo 2:** aplicación que acepta un nº de mes e imprime su nombre
  - La particiones equivalentes son
    - <1 (inválido)
    - 1 – 12 (válido)
    - > 12 (inválido)
- **Ejemplo 3:** formulario de autenticación con usuario y password
  - **Usuario:** puede ser verdadero o falso. Particiones equivalentes:
    - Vacío
    - Válido
    - Inválido
  - **Password:** puede ser verdadero o falso. Particiones equivalentes:
    - Vacío
    - Válido
    - Inválido

# Análisis de valores límite

43

- Los errores tienden a darse más en los límites del campo de entrada que en el centro
  - Consecuentemente, hay que plantear casos de pruebas que ejerciten los valores límite
- Esta estrategia es complementaria a la partición equivalente

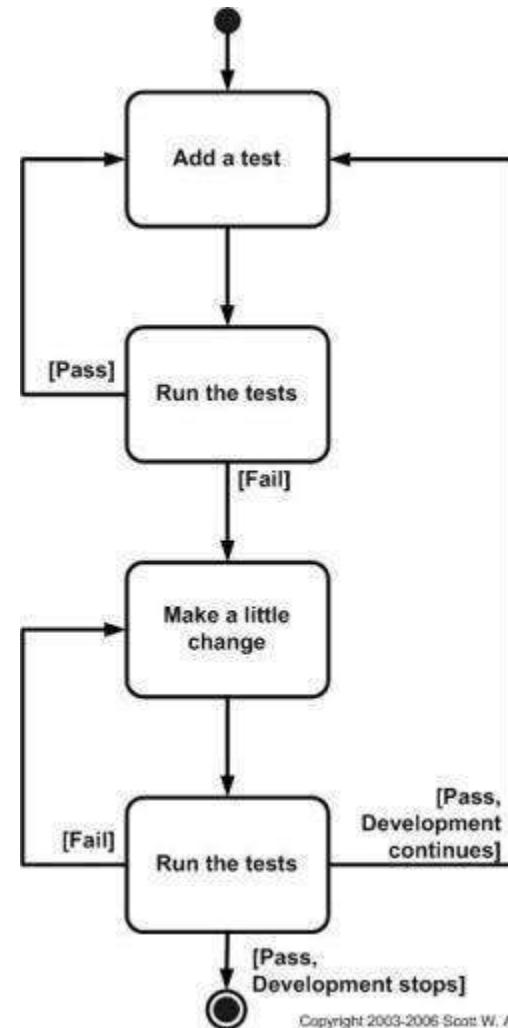
# Test Driven Development (TDD)

44

- **Desarrollo dirigido por pruebas**

- Característica clave de la **Extreme Programming**
- Proceso de desarrollo software basado en el siguiente ciclo:
  1. Escritura de un caso de prueba para una funcionalidad a incluir en el software
  2. Implementar el código que pase el test
  3. Refactorizar el código

- **Herramientas:** JUnit



Copyright 2003-2006 Scott W. Ambler

# Test Driven Development (TDD)

45

- **Ventajas**

- Al centrarse en las pruebas, el código tiene menos errores y se aumenta la productividad
- Separación entre el comportamiento esperado y la implementación para conseguirlo
- Tests de regresión: al desarrollar de forma incremental los tests de regresión permiten encontrar errores
- Se reduce la necesidad de depurar código

- **Inconvenientes**

- Frecuentemente el diseñador de los tests implementa el código
- Sensación de seguridad que puede ser irreal

# Introducción a la Ingeniería del Software



**TEMA 7.2: VERIFICACIÓN Y PRUEBAS**

**MOCKING**

**Grado en Ingeniería Informática  
Grado en Ingeniería del Software  
Grado en Ingeniería de Computadores**



# Mocking

---



- Las **pruebas unitarias** se deberían aplicar a **clases aisladas**
- Pero lo habitual es que unas **clases dependan de otras clases o recursos**
  - Una base de datos
  - Datos de un servidor remoto
  - Datos de un sensor
- ¿Cómo eliminar las dependencias con esas clases/recursos y sus potenciales efectos colaterales?
- *La idea del mocking es usar objetos que reemplacen a este tipo de entidades*

# Mocking



- Un objeto mock<sup>(1)</sup> es un sustituto de un objeto real
  - Para poder hacer pruebas sin tener que usar el objeto real
  - El objeto real
    - Puede no estar implementado todavía
    - Puede ser costoso de usar
    - Puede ser muy lento
    - El objeto es una interface
    - El objeto puede no existir todavía
  - Se pretende comprobar que el código probado funciona, independientemente de sus dependencias

## **(1) Mock**

*adjective [ attrib. ]*

*not authentic or real, but without the intention to deceive*

Mock: simulado, fingido.

Mock file: Archivo simulados o un sustituto de archivo.

# Mocking



- Varias herramientas disponibles en Java:
  - Easymock: <http://www.easymock.org/>
  - jMock: <http://www.jmock.org/>
  - jmockit: <http://code.google.com/p/jmockit/>
  - PowerMock: <https://github.com/powermock/powermock>
  - Mockito: <http://code.google.com/p/mockito/>

# Mockito



- **Pasos para trabajar con Mockito:**
  - Descargar el jar de su página Web:
    - <https://search.maven.org/artifact/org.mockito/mockito-core>
  - Añadirlo al CLASSPATH de proyecto
  - **Alternativa:** añadirlo como dependencia en Maven.
- **Características principales**
  - Se pueden crear mocks de interfaces y de clases concretas
  - Verificación de invocaciones (cantidad exacta, al menos una vez, ...)

# Tests con Mockito



- Un método de prueba ha de tener tres partes:
  - Inicializar el entorno del test (p.ej. crear objetos mocks e insertar los mocks en el código a probar)
  - Ejecutar el método a probar
  - Verificar que el código se ejecutó correctamente

```
public class MockitoTest {  
  
    @Test  
    public void probarAlgo() {  
        Estrategia estrategia = mock(Estrategia.class);  
        AlgunaClase objetoAProbar = new AlgunaClase(estrategia);  
  
        objetoAProbar.ejecutaMetodo();  
  
        verify(estrategia).hacerAlgoConcreto();  
    }  
}
```

# Lo básico



```
import static org.mockito.Mockito.*;  
  
//Let's import Mockito statically so that the code looks clearer  
  
//mock creation  
List<String> mockedList = mock(List.class);  
  
//using mock object  
mockedList.add("one");  
mockedList.clear();  
  
//verification  
verify(mockedList).add("one");  
verify(mockedList).clear();
```

# Método verify()



- Mockito controla todas **las llamadas a los métodos** y sus **parámetros** al objeto mock.
- **Comprobaciones:**
  - El método se ha llamado una única vez con el argumento “one”  
`verify(mockedList).add("one")`
  - El método se ha llamado una única vez (sin importar el argumento)  
`verify(mockedList).add(anyString())`
  - El método se ha llamado cierto número de veces  
`verify(mockedList, times(2)).add("one")`
  - El método no se ha llamado nunca (`times(0)`)  
`verify(mockedList, never()).add("one")`
  - El método se ha llamado como mucho/como poco  
`atLeastOnce(), atLeast(2), atMost(3), etc.`
  - No se ha llamado a ningún otro método aparte de los verificados anteriormente  
`verifyNoMoreInteractions(mockedList)`

# Stubbing



- Los *mocks* pueden **devolver valores diferentes dependiendo de los argumentos** pasados a un método.
- Se usa `when(...).thenReturn(...)` para especificar el **valor de retorno** de un **método con parámetros predefinidos**.
  - También se puede usar `anyInt()`, `anyString()`, `any(Clase.class)`
- Se usa `when(...).thenThrow(...)` para especificar **excepciones lanzadas** por un **método con parámetros predefinidos**

# Ejemplo



```
// creamos el objeto mock usando mock
List<String> mockedList = mock(List.class);

// Definimos el funcionamiento del objeto mock
when(mockedList.get(0)).thenReturn("primer elemento");
when(mockedList.get(1)).thenReturn("segundo elemento");
when(mockedList.get(-1)).thenThrow(new
IndexOutOfBoundsException());

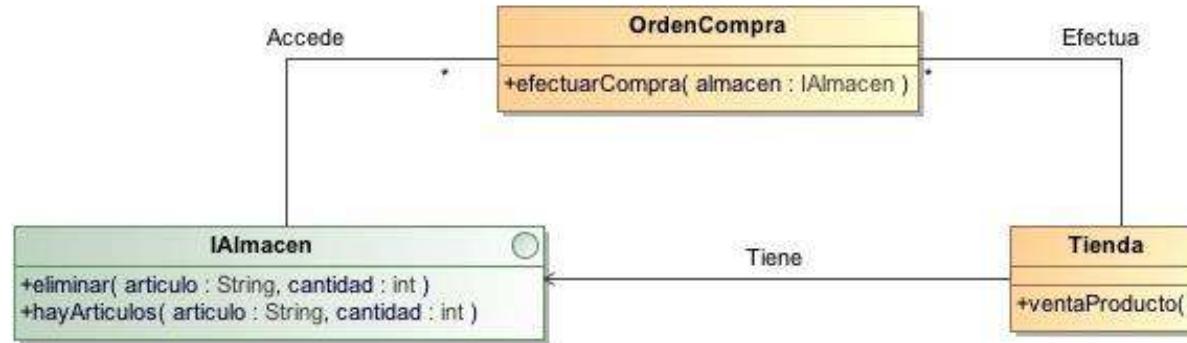
// Usamos el objeto mock
System.out.println(mockedList.get(1));
System.out.println(mockedList.get(-1));
// La siguiente llamada devuelve null porque
// no se ha definido el mock
System.out.println(mockedList.get(999));
```

# Tests con Mockito. Ejemplo



- Ejemplo:

- Una tienda tiene un almacén.
- Cada orden de compra en la tienda implicar ver si hay cantidad suficiente del producto en el almacén.
- Una vez efectuada la compra, la cantidad vendida del producto se debe eliminar del almacén.



# Ejemplo de Mockito



```
public interface IAlmacen {  
  
    boolean hayArticulos(String string, int i);  
  
    void eliminar(String producto, int cantidad);  
}
```

- Tenemos esta Interfaz que estará implementada en una Base de datos, en un fichero, nos da igual.
- No la tenemos accesible físicamente.
- Pero tenemos que probar nuestra clase que la usará.

# Tests con Mockito. Ejemplo



- Asumiendo que el almacén es una interfaz
  - Comprobar que se ha eliminado del almacén de una tienda un producto que se ha vendido

```
public class TiendaTest{  
  
    @Test  
    public void ventaProductoTest() {  
        // Inicializar  
        IAlmacen almacen= mock(IAlmacen.class);  
        when (almacen.hayArticulos( "Cerveza" , 20 )).thenReturn(true) ;  
  
        // Ejecutar  
        OrdenCompra orden = new OrdenCompra( "Cerveza" , 20) ;  
        orden.efectuarCompra(almacen) ;  
  
        // Comprobar  
        verify(almacen).eliminar( "Cerveza" , 20 );  
    }  
}
```

# Introducción a la Ingeniería del Software



CONTROL DE VERSIONES CON

# GIT

Grado en Ingeniería Informática  
Grado en Ingeniería del Software  
Grado en Ingeniería de Computadores



# Índice de contenidos



- Sistemas de control de versiones
- Tipos de sistemas de control de versiones
- Caso de estudio: git

# Conceptos generales

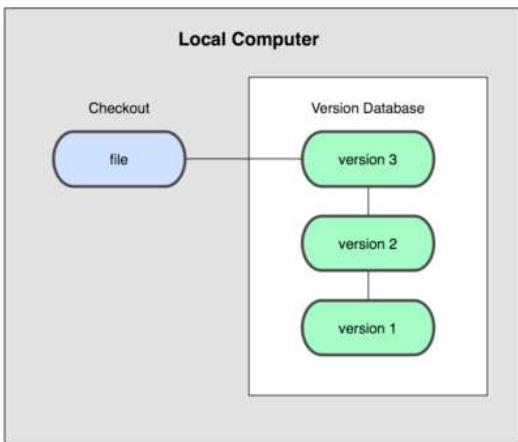


- **¿Qué es un sistema de control de versiones (SCV)?**
  - Herramienta que registra los cambios realizados sobre un archivo o conjunto de archivos a lo largo del tiempo, de modo que puedas recuperar versiones específicas más adelante.
  - Cualquier tipo de archivo que encuentres en un ordenador puede ponerse bajo control de versiones.
- **Propósito de un SCV**
  - Permite revertir archivos a un estado anterior.
  - Comparar cambios a lo largo del tiempo.
  - Ver quién modificó por última vez algo que puede estar causando un problema.
  - Quién introdujo un error y cuándo, y mucho más.

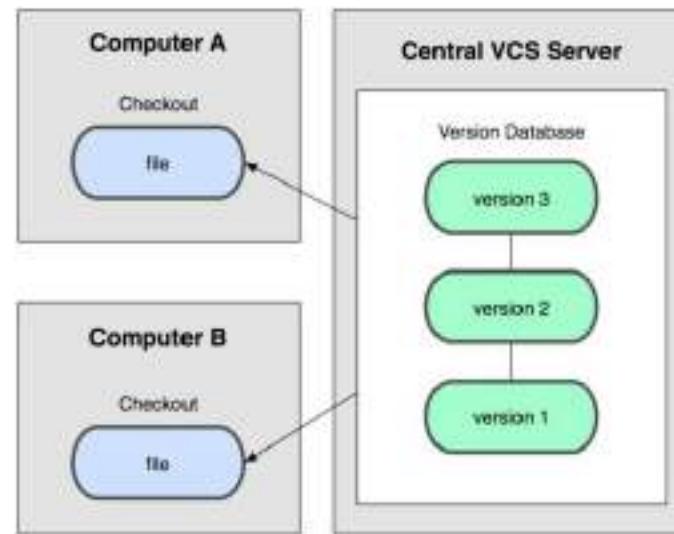
# Tipos de sistemas de control de versiones



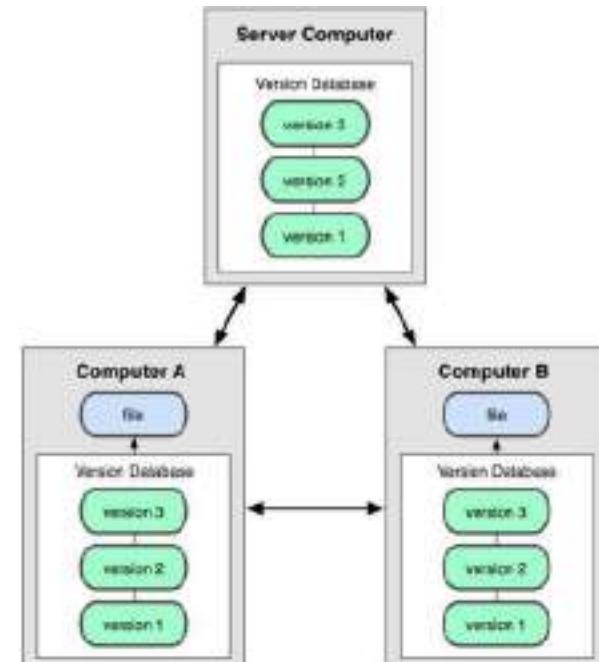
SCV Local (rcs)



SCV centralizado  
(cvs, subversion)



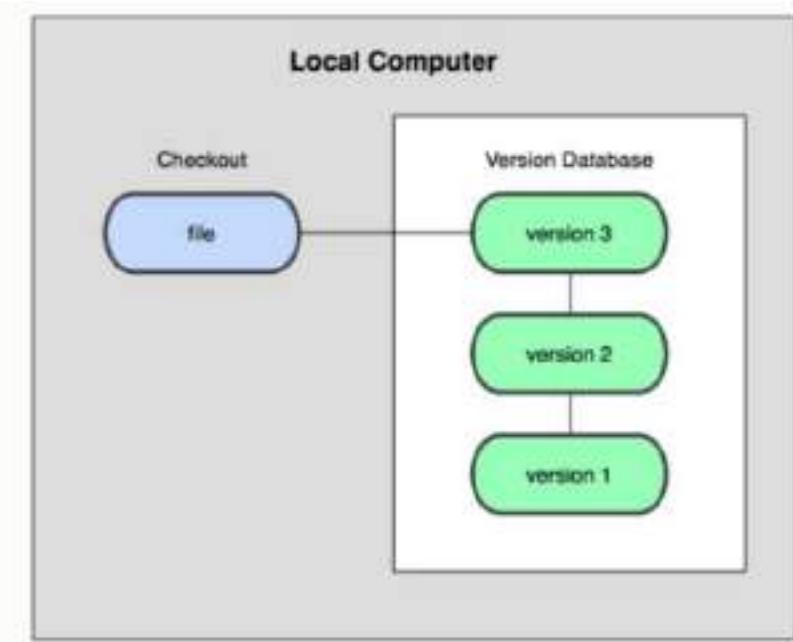
SCV distribuido  
(git, mercurial)



# SCV Locales



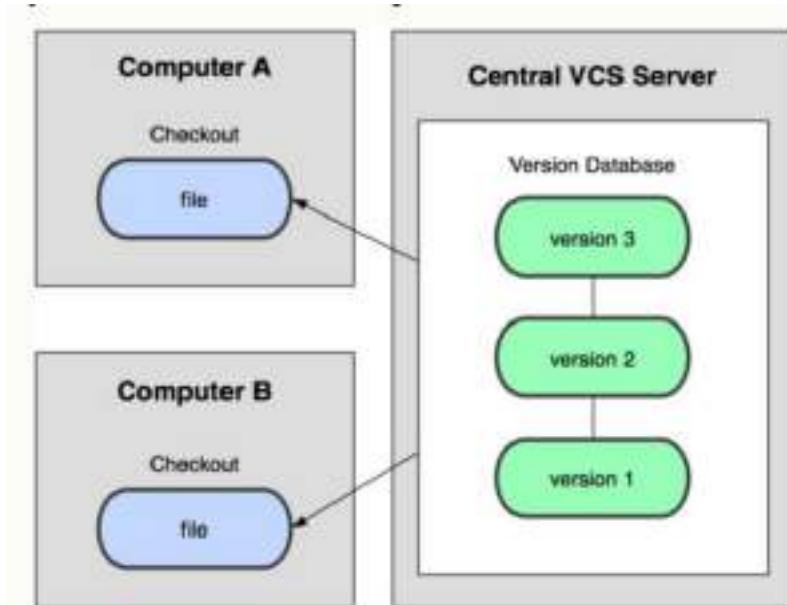
- **Método más simple:** copiar los archivos a otro directorio, indicando la fecha y la hora.
- Método propenso a errores:
  - Es fácil olvidar en qué directorio te encuentras, y guardar accidentalmente en el archivo equivocado o sobrescribir archivos que no querías.
- **SCV locales:** tienen una base de datos en la que se registra todos los cambios realizados sobre los archivos.
- **Problema:** ¿cómo colaboran los desarrolladores?



# SCV Centralizados



- **Arquitectura cliente-servidor**
  - Un único servidor que contiene todos los archivos versionados.
  - Los clientes descargan los archivos desde ese lugar central.
- Sistema más común para SCV durante muchos años.
- **Ejemplos:** CVS, Subversion, y Perforce,



# SCV Centralizado



- Ventajas respecto al SCV Local
  - Todo el mundo puede saber en qué trabajan los otros colaboradores del proyecto.
  - Los administradores tienen control detallado de qué puede hacer cada uno.
  - Es más fácil administrar un SCV central que tener que lidiar con bases de datos locales en cada cliente.
- Desventajas:
  - El servidor es un punto crítico ante los fallos.

# SCV Distribuido



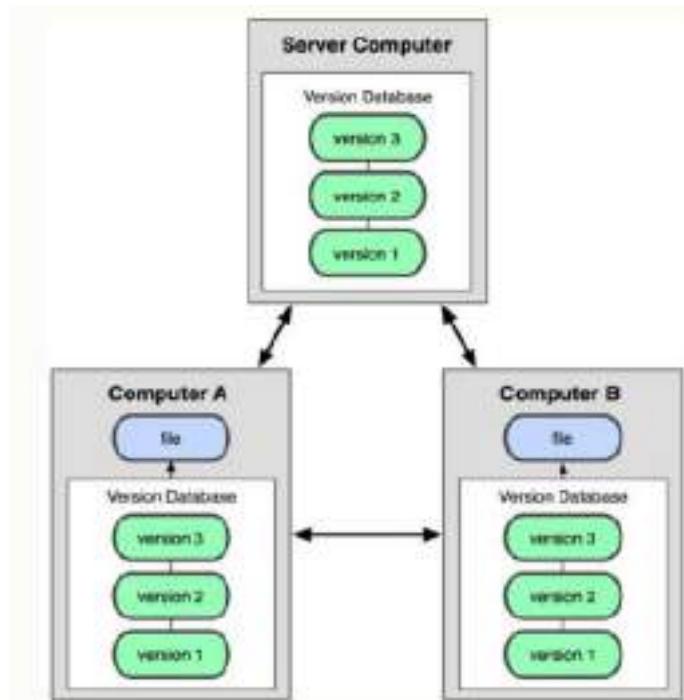
- **Arquitectura peer-to-peer**

- Los clientes tienen una copia legítima del repositorio.
- Pueden subir/bajar archivos al repositorio central.
- Clientes pueden trabajar incluso cuando no hay conexión al servidor.

- **Ventajas:**

- Si un servidor falla cualquiera de los repositorios de los clientes puede copiarse en el nuevo servidor para restaurarlo.
- Permiten varios repositorios de trabajo.

- **Ejemplos:** Git, Mercurial, Bazaar, Darcs



# Caso de estudio: Git

---



- Ideado por Linus Torvalds para el desarrollo del kernel de Linux (1<sup>a</sup> versión en abril de 2005).
- Características principales
  - SCV distribuido de código abierto
  - Existen numerosos clientes gráficos (GitKraken, SourceTree, TortoiseGit,...)
  - Existen *plugins* para muchos IDEs (Eclipse, NetBeans, ...)
- Existen servicios de hosting gratuitos para alojar proyectos (repositorios) Git
  - GitHub (<http://github.com>)
  - Bitbucket (<https://bitbucket.org/>)

# GitHub



GitHub, Inc. (US) https://github.com ⌂ R plot 3D

Search or type a command ⌂ Explore Gist Blog Help ajnebro ⌂ X ⌂

ajnebro News Feed

News Feed Pull Requests Issues Stars

**GitHub Bootcamp** If you are still new to things, we've provided a few walkthroughs to get you started. ⌂

A black cat character is shown pouring colorful commits from a box into another cat's mouth.

**Set Up Git** 1  
A quick guide to help you get started with Git.

A black cat character is shown inside a computer monitor screen, which is itself inside a cube.

**Create A Repository** 2  
Create the place where your commits will be stored.

Two cat characters, one black and one blue, are shown interacting with each other.

**Fork a Repository** 3  
Copy a repo to create a new, unique project from its contents.

A group of black cat characters are shown interacting with each other in a social network-like setting.

**Be social** 4  
Follow a friend. Watch a project.

# Bitbucket



Screenshot of a web browser showing the Bitbucket interface for a repository named 'SRAM\_Demo\_Practical\_Control'. The user is 'joseandres'.

The left sidebar shows navigation links: Overview, Source, **Commits**, Branches, Full requests, Pipelines, Deployments, Downloads, Boards, and Settings.

The main content area displays the 'Commits' page for the 'master' branch. It includes a search bar labeled 'Find commits' and filters for 'Date' and 'Builds'.

Author	Commit	Message	Date	Builds
joseeng	ee7eade	Merge branch 'master' of https://bitbucket.org/joseeng/sram_demo_practical_control	2017-10-12	
joseeng	686f587	fixed bug in background cancellation	2017-10-12	
jose A. Gonzal...	6d3458e	added background_model.json example file in resources	2017-10-09	
joseeng	c7c5f87	added background cancellation support	2017-10-09	
joseeng	83e1b42	optimized audio latency	2017-07-19	
joseeng	5c28857	implemented audio resampling in the native library	2017-07-18	
joseeng	d030095	Disable on-the-fly data plotting and logging for speech purposes	2017-07-18	
José A. Gonzal...	4882437	Disable on-the-fly data plotting and logging for speech purposes	2017-06-29	
José A. Gonzal...	38c5622	Initial commit	2017-06-23	
José A. Gonzal...	ac4380e	Initial commit	2017-06-23	

# Instalación de Git

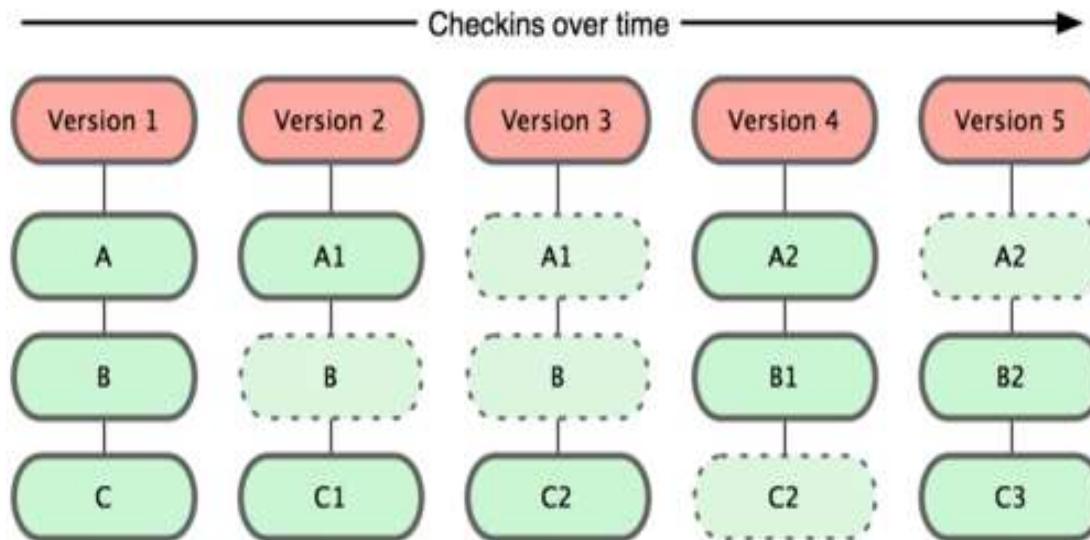


- Herramienta en línea de comandos:
  - Linux/Mac: ya suelen tener preinstalada una versión de git.
  - Windows: descargar desde <https://git-scm.com/downloads>
- Clientes gráficos
  - Ver: <https://git-scm.com/downloads/guis>
- Plugins para IDEs
  - Ya suelen venir preinstalados en los principales IDEs (Visual Studio, Android Studio, etc).

# Fundamentos de Git



- **Git** modela los datos como un conjunto de instantáneas de un mini sistema de archivos.
- Cada vez que se confirma un cambio, Git hace “hace una foto” de los datos del proyecto y guarda una referencia a esa instantánea.
- Si los archivos no se han modificado, Git sólo almacena un enlace al archivo anterior idéntico que ya tiene almacenado.



# Fundamentos de Git



- Operaciones en modo local
  - Sólo precisa archivos y recursos locales. Velocidad.
  - Si no estás conectado a una Red, puedes hacer casi todas las operaciones. No precisa un servidor.
  - Actúa como un SCV local.
- Integridad
  - Guarda el contenido no por el nombre, lo hace por el valor de hash.
  - Todas las comprobaciones se conocen como hash SHA-1. Se trata de una cadena de 40 caracteres hexadecimales y se calcula respecto a los contenidos del archivo o estructura de directorio.

**24b9da6552252987aa493b52f8696cd6d3b00373**

# Fundamentos de Git



- Los cuatro estados de los archivos:
  - No seguido (untracked)
    - El archivo no está siendo seguido por Git (sus versiones no se controlan).
  - Modificado (modified).
    - Se ha modificado el archivo pero no ha sido confirmado en la base de datos.
  - Preparado (staged).
    - El archivo ha sido marcado en su versión actual para que vaya en la próxima confirmación.
  - Confirmado (committed).
    - Los datos están almacenados de manera segura en la base de datos local.

# Fundamentos de Git



- Secciones de un proyecto de Git:

- Directorio de trabajo (Working directory).

- Es una copia de una versión del proyecto (no tiene por qué ser la última).
    - Los archivos se extraen de la base de datos del directorio de Git y se colocan en disco para usarlos.

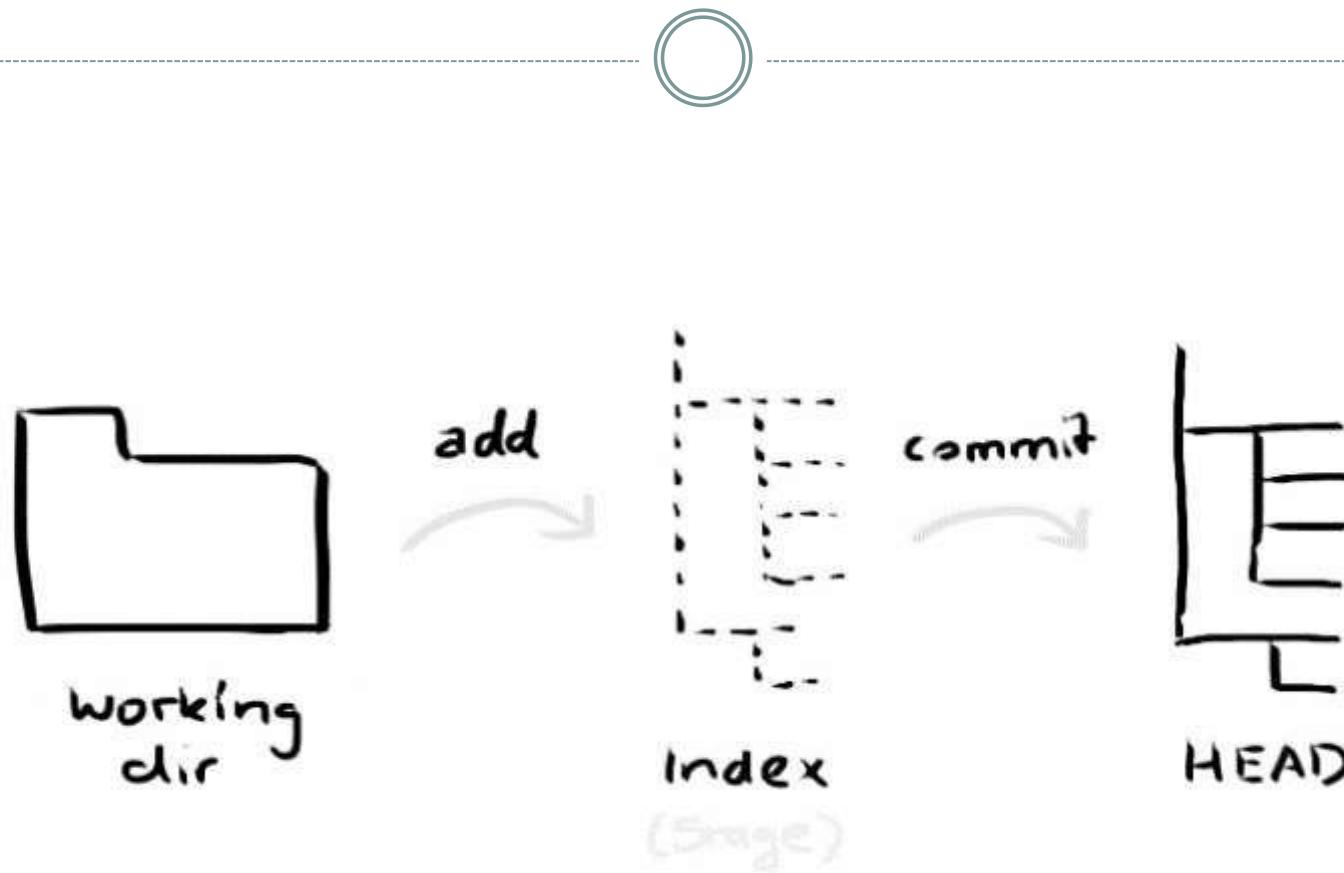
- Área de preparación (Staging area).

- Almacena información sobre lo que va a ir en la próxima confirmación.
    - Es un área intermedia entre el Directorio de trabajo y el Directorio de Git.

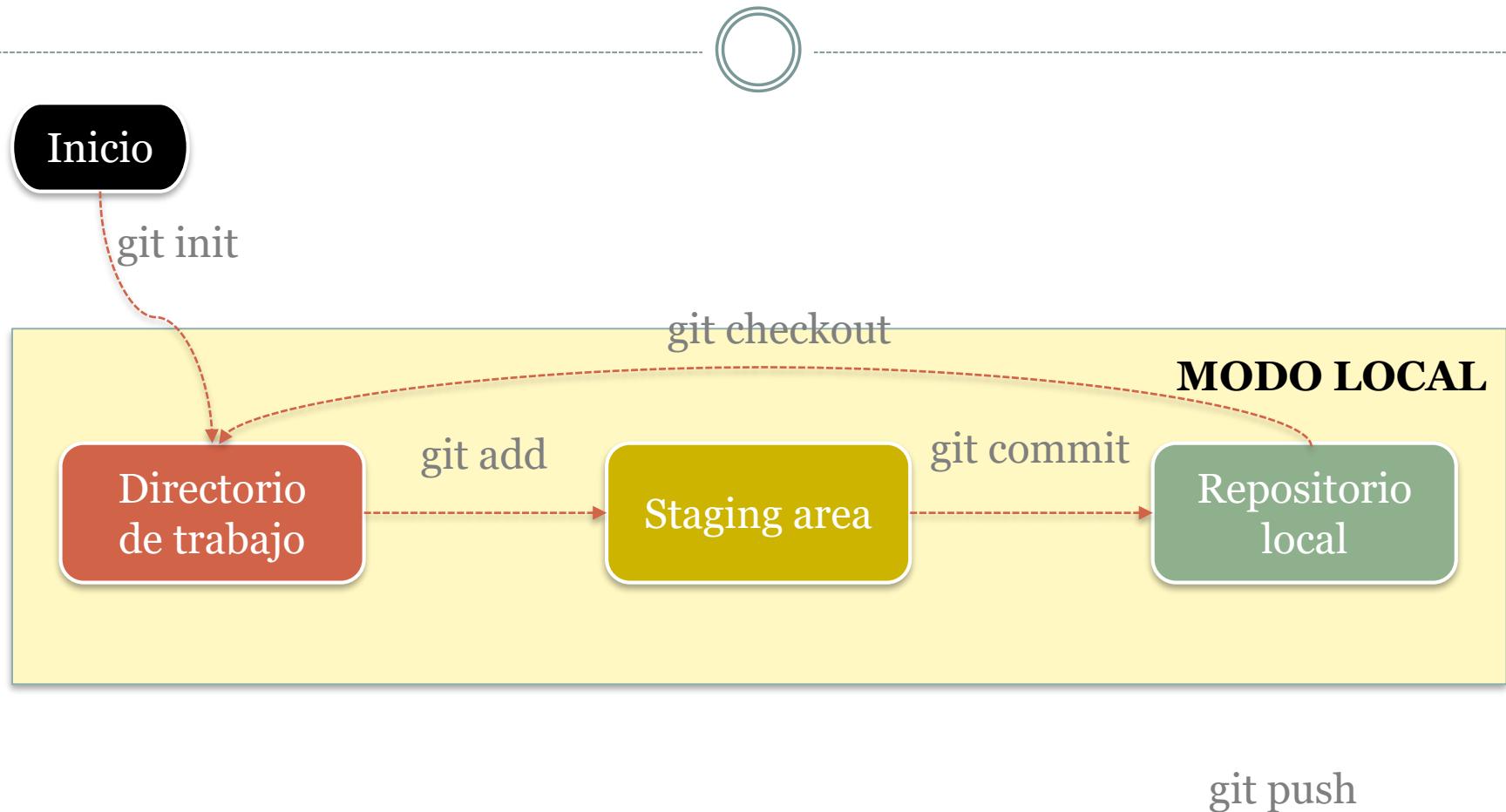
- Directorio Git (repositorio).

- Es donde Git almacena los metadatos y la base de datos de objetos para el proyecto.
    - Es lo que se copia cuando se clona un repositorio.

# Secciones de un proyecto



# Flujo básico de trabajo en Git



# Configuración inicial

---



- Configurar el nombre de usuario y dirección de correo electrónico
  - `git config --global user.name "jgonzalez"`
  - `git config --global user.email j.gonzalez@uma.es`
- Es importante porque las confirmaciones de cambios (commits) en Git usan esta información.
  - Si se usa --global esta opción siempre estará en el sistema.
  - Si se quiere reescribir esta información con otro nombre o dirección de correo para proyectos específicos se puede ejecutar el comando sin la opción --global cuando estemos en ese proyecto.

# Creando un repositorio nuevo

---



- Crear un nuevo repositorio local desde cero
  - Nos situamos en el directorio desde donde cuelga el proyecto.
  - Ejecutamos
    - ✖ `git init`
  - Se crea de forma automática una carpeta .git
  - A partir de ahora git llevará un control de las versiones de los archivos del proyecto.

# Añadiendo archivos a la Staging Area

---



- Agregar un fichero a la *staging area*
  - Si el archivo se ha modificado o queremos que git empiece a llevar un control de sus versiones.
  - `git add <nombre del archivo>`
  - Esto añade el archivo a la *staging area* para que cuando hagamos un commit se confirmen sus cambios.
- Agregar todos los ficheros
  - `git add .`
- Ver el estado de los ficheros
  - `git status`

# Ejemplo



```
1. zsh
joseangl@192 ~/git_project $ nano Hola.cpp
joseangl@192 ~/git_project $ cat Hola.cpp
#include <iostream>

int main()
{
    std::cout << "Hola mundo!!" << std::endl;
    return 0;
}
joseangl@192 ~/git_project $ git status
On branch master

Initial commit

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    Hola.cpp

nothing added to commit but untracked files present (use "git add" to track)
joseangl@192 ~/git_project $ git add Hola.cpp
joseangl@192 ~/git_project $
```

# Ejemplo



```
1. zsh
joseangl@192 ~/git_project $ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   Hola.cpp

joseangl@192 ~/git_project $
```

# Ejemplo



```
1. zsh
joseangl@192 ~/git_project $ nano Hola.cpp
joseangl@192 ~/git_project $ cat Hola.cpp
#include <iostream>

int main()
{
    //Incluimos un comentario en el codigo
    std::cout << "Hola mundo!!" << std::endl;
    return 0;
}
joseangl@192 ~/git_project $ git status
On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   Hola.cpp

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   Hola.cpp

joseangl@192 ~/git_project $
```

# Consolidar los cambios



- Los cambios hay que confirmarlos para añadirlos al directorio de git.
  - `git commit -m "comentario"`
- El comentario da información de lo que hace un commit.
  - ¿Por qué has hecho este cambio?
  - ¿Qué cambios has hecho?
  - ¿Qué consecuencias esperas que tenga tu cambio?
- Añadir al *staging area* y commit a la vez:
  - `git commit -a -m "comentario"`

# Ejemplo



```
1. zsh
joseangl@192 ~/git_project $ git commit -am "Actualizado Hola.cpp"
[master (root-commit) aa94a3b] Actualizado Hola.cpp
 1 file changed, 8 insertions(+)
 create mode 100644 Hola.cpp
joseangl@192 ~/git_project $ git status
On branch master
nothing to commit, working tree clean
joseangl@192 ~/git_project $
```

# Commits al Kernel de Linux en GitHub



The screenshot shows the GitHub interface for the repository 'torvalds/linux'. The repository has 8,142 stars and 20,507 forks. The 'Code' tab is selected, showing the commit history for the 'master' branch. Two groups of commits are visible: one from February 23, 2018, and another from February 22, 2018. Each commit includes the author, commit message, date, and a copy/paste link.

Date	Commit Message	Author	Commit Hash
Feb 23, 2018	MIPS: boot: Define __ASSEMBLY__ for its .S build	kees authored and torvalds committed 14 hours ago	8f9eda4
Feb 23, 2018	Merge branch 'siginfo-linus' of git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/siginfo-linus	torvalds committed 14 hours ago	b4e6bcf
Feb 23, 2018	Merge branch 'linus' of git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linus	torvalds committed 15 hours ago	8bb7816
Feb 22, 2018	fs/signalfd: fix build error for BUS_MCEERR_AR	Randy Dunlap authored and akpm committed 11 days ago	9826e32
Feb 22, 2018	Merge tag 'usb-4.18-rc3' of git://git.kernel.org/pub/scm/linux/kernel...	torvalds committed 19 hours ago	e638ef9
Feb 22, 2018	Merge tag 'staging-4.18-rc2' of git://git.kernel.org/pub/scm/linux/ke...	torvalds committed 19 hours ago	77f892e
Feb 22, 2018	Merge tag 'char-misc-4.18-rc3' of git://git.kernel.org/pub/scm/linux/...	torvalds committed 19 hours ago	bb17186
Feb 22, 2018	Merge tag 'for-linus' of git://git.kernel.org/pub/scm/linux/kernel/gi...	torvalds committed 19 hours ago	004e390

# Viendo los cambios

---



- Se pueden ver los cambios de un fichero en el directorio de trabajo respecto a su última versión confirmada (committed).
- Para ello se usa la orden:
  - `git diff [archivo]`
- Salida del comando:
  - Líneas borradas en rojo (con un - delante)
  - Líneas añadidas en verde (con un + delante)

# Ejemplo



```
1. zsh
joseangl@192 ~/git_project $ nano Hola.cpp
joseangl@192 ~/git_project $ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   Hola.cpp

no changes added to commit (use "git add" and/or "git commit -a")
joseangl@192 ~/git_project $ git diff
diff --git a/Hola.cpp b/Hola.cpp
index 9889c00..7fac266 100644
--- a/Hola.cpp
+++ b/Hola.cpp
@@ -3,6 +3,7 @@
 int main()
 {
     //Incluimos un comentario en el codigo
-     std::cout << "Hola mundo!!" << std::endl;
+     std::cout << "Hello world!!" << std::endl;
+     //Incluimos otro segundo comentario
     return 0;
 }
joseangl@192 ~/git_project $
```

# Otras órdenes útiles



- Reemplazar cambios locales
  - `git checkout -- <nombre del archivo>`
  - Esto reemplaza el archivo con su última versión confirmada.
- Para quitar un archivo del control de Git (dejar de seguirlo)
  - `git rm <nombre del archivo>`
- Para no controlar un archivo pero dejarlo en el directorio de trabajo usamos el fichero
  - `.gitignore`
  - Se pueden especificar extensiones (p.ej. .exe, .o, .jar)
- Ver historial y los commits que se han realizado
  - `git log --oneline`

# Ejemplo



```
1. zsh
joseangl@192 ~/git_project $ git commit -am "Nuevo cambio a Hola.cpp"
[master 56e8f5f] Nuevo cambio a Hola.cpp
 1 file changed, 2 insertions(+), 1 deletion(-)
joseangl@192 ~/git_project $ git log --oneline
56e8f5f Nuevo cambio a Hola.cpp
aa94a3b Actualizado Hola.cpp
joseangl@192 ~/git_project $
```

# Gitk



The screenshot shows the gitk graphical interface for a project named 'git\_project'. The main window displays a commit history for the file 'Hola.cpp'. The first commit, highlighted in green, is titled 'Nuevo cambio a Hola.cpp' and was made by 'Actualizado Hola.cpp' on '2018-02-22 10:52:18'. The commit message includes the SHA1 hash: 50e8757960006d38c0108327d7475d91b79c94b. Below the commit details, there are search and filter options: 'Buscar' (Search), 'revisión' (Revision) set to 'que contiene:' (contains:), and dropdown menus for 'exento' (Exempt) and 'Todos los campos' (All fields). There are also buttons for 'Parche' (Patch) and 'Árbol' (Tree). The commit message itself is displayed in a large text area:

```
Author: Jose A. Gonzalez <j.gonzalez@sheffield.ac.uk> 2018-02-22 10:52:18 +0000
Committer: Jose A. Gonzalez <j.gonzalez@sheffield.ac.uk> 2018-02-22 10:52:18 +0000
Patch: https://github.com/jggonzalez/Hola/commit/50e8757960006d38c0108327d7475d91b79c94b (Actualizado Hola)
Renames:
Sigue a:
Precede a:

Nuevo cambio a Hola.cpp
```

Below the commit message, the file content 'Hola.cpp' is shown in a diff view:

```
diff --git a/Hola.cpp b/Hola.cpp
index 9889e80..21ae25b 100644
--- a/Hola.cpp
+++ b/Hola.cpp
@@ -3,6 +3,7 @@
 int main()
 {
     //Incluimos un comentario en el codigo
-    std::cout << "Hola mundo!" << std::endl;
-    std::cout << "Hasta luego!" << std::endl;
-    //Incluimos otro segundo comentario
+    std::cout << "Hola mundo!" << std::endl;
+    std::cout << "Hasta luego!" << std::endl;
     return 0;
 }
```

# Examinar commits previos



- Volver a un commit anterior.
  - `git checkout <hash del commit>`
  - Ejemplo: `git checkout f704f4a`
- Esta orden **reemplaza los archivos del directorio de trabajo** por la copia almacenada en el commit.
- Para volver al commit donde estuviésemos:
  - `git checkout <nombbre de la rama>`
  - Ejemplo: `git checkout master`

# Ejemplo



```
1. zsh
joseangl@192 ~/git_project $ nano Readme.md
joseangl@192 ~/git_project $ git add .; git commit -m "Añadido Readme.md"
[development 82c96e0] Añadido Readme.md
 1 file changed, 5 insertions(+)
 create mode 100644 Readme.md
joseangl@192 ~/git_project $ ls
Hola.cpp  Readme.md
joseangl@192 ~/git_project $ git checkout aa94a3b
Note: checking out 'aa94a3b'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

git checkout -b <new-branch-name>

HEAD is now at aa94a3b... Actualizado Hola.cpp
joseangl@192 ~/git_project $ ls
Hola.cpp
joseangl@192 ~/git_project $
```

# Creando tags

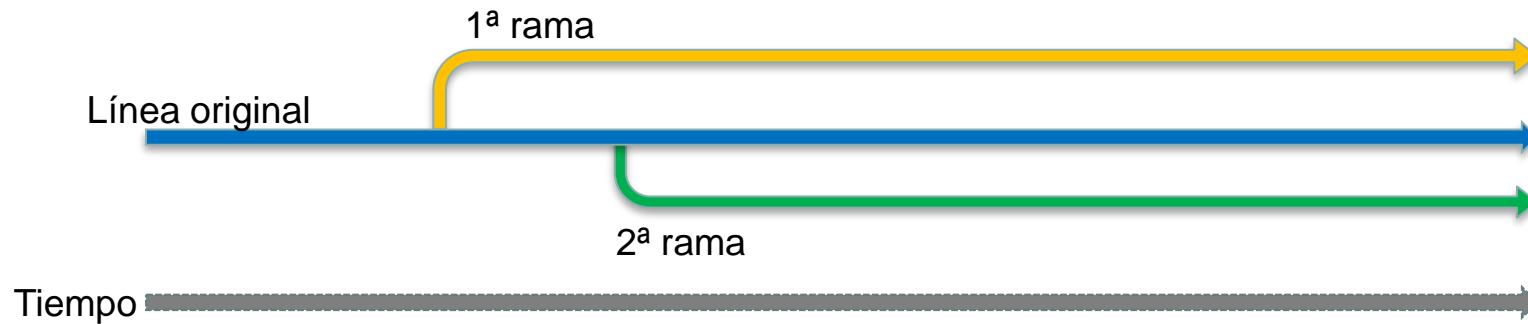


- **Tags (etiquetas)**: sirven para crear alias a commits concretos con un nombre más *amigable*.
  - En lugar de referirnos a un commit con su hash lo hacemos usando su tag.
  - Se suelen usar para nombrar las diferentes versiones del proyecto.
- Creación:
  - `git tag <alias> <código del commit>`
  - Ejemplo (commit actual): `git tag v1.0`
  - Ejemplo (commit antiguo): `git tag v1.0 f704f43`
- Ver la lista de tags:
  - `git tag`

# Ramas



- Una rama (*branch*) es una línea de desarrollo que existe de forma independiente de una anterior
  - Pero que parte de ella
  - Y posiblemente se fusionen posteriormente
- Se suelen usar para probar nuevas funcionalidades en el proyecto sin modificar la rama estable (master)



# Ramas de TensorFlow en GitHub



The screenshot shows the GitHub repository page for `tensorflow/tensorflow`. The top navigation bar includes links for Features, Business, Explore, Marketplace, Pricing, and Sign In or Sign up. The main header displays the repository name and its statistics: 7,058 Watchers, 90,130 Stars, and 58,524 Forks. Below the header, there are tabs for Code, Issues (1,187), Pull requests (173), Projects (0), and Insights. A search bar for branches is located on the right. The page is divided into sections: Default branch, Active branches, and Stale branches.

### Default branch

**master** Updated 10 hours ago by gunesi Default

### Active branches

Branch	Last Update	Commits	Pulls	Action
v1.5	Updated 19 hours ago by Mekkaoui	2050	11	Compare
v1.6	Updated 2 days ago by swembit	456	7	Compare
timeseries-head	Updated 3 days ago by terrytangyuan	250	2	#17134 Open
terrytangyuan-patch-1	Updated 6 days ago by terrytangyuan	227	2	#17100 Open
mertimwicke-patch-1	Updated 8 days ago by mertimwicke	428	2	#17051 Merged

[View more active branches >](#)

### Stale branches

Branch	Last Update	Commits	Pulls	Status
0.6.8	Updated 2 years ago by vvv	25950	4	#16516 Closed
0.8.0	Updated 2 years ago by brandan	25721	2	#2964 Closed

# Ramas



- La rama principal en git se denomina por defecto **master** (se crea al crear el repositorio)
- Las ramas se muestran con el comando
  - **git branch**
- La rama activa se marca con un asterisco

```
joseangl@192 ~/git_project $ git branch
* master
joseangl@192 ~/git_project $
```

# Creación de nuevas ramas



- Creación de una rama:
  - `git branch <nombre de la rama>`
- Cambiar a una rama:
  - `git checkout <nombre de la rama>`
  - Los siguientes *commits* afectarán a esa rama y no a otras.
- Crear la rama y cambiar a ella:
  - `git checkout -b <nombre de la rama>`

```
joseangl@192 ~/git_project $ git checkout -b development
Switched to a new branch 'development'
joseangl@192 ~/git_project $ git branch
* development
  master
joseangl@192 ~/git_project $
```

The screenshot shows a terminal window titled "1. zsh". It displays a command-line session where the user runs "git checkout -b development", which creates a new branch named "development". Then, the user runs "git branch" to list the branches, showing "development" as the current active branch (indicated by an asterisk) and "master" as another branch. The terminal has a dark background with light-colored text and standard OS X window controls at the top.

# Añadiendo archivos a *development*

---



- Creamos dos nuevos archivos en *development*:
  - print.h
  - print.cpp
- El código de Hola.cpp queda de la siguiente forma:

```
#include "print.h"

int main()
{
    //Llamada a la función que imprime "Hello world!!"
    print_hello_world();

    return 0;
}
```

# Añadiendo archivos a *development*

---



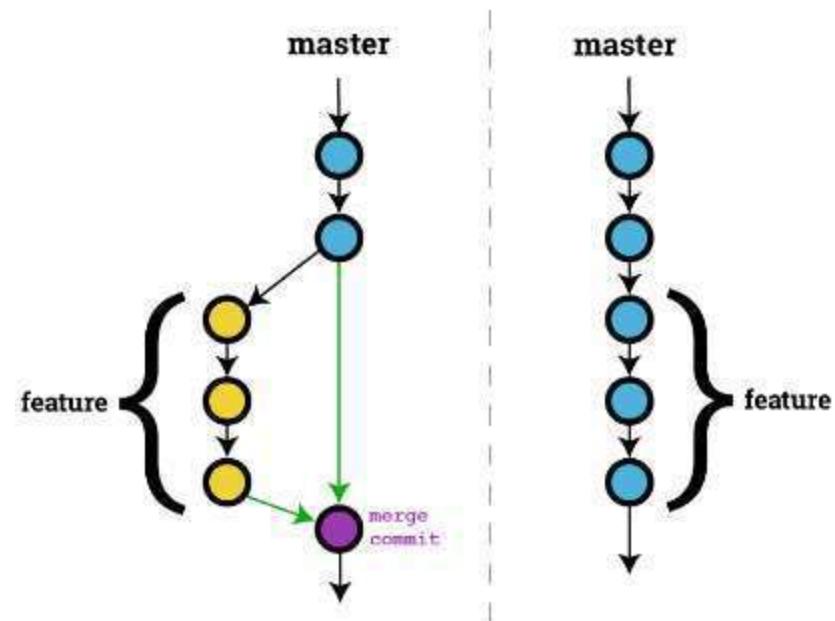
- Comprobamos que los nuevos ficheros sólo están en la rama *development*:

```
1. zsh
joseanol@192 ~/git_project $ git checkout development
Switched to branch 'development'
joseanol@192 ~/git_project $ ls
Hola.cpp print.cpp print.h
joseanol@192 ~/git_project $ git checkout master
Switched to branch 'master'
joseanol@192 ~/git_project $ ls
Hola.cpp Readme.md
joseanol@192 ~/git_project $ 
```

# Fusionar ramas



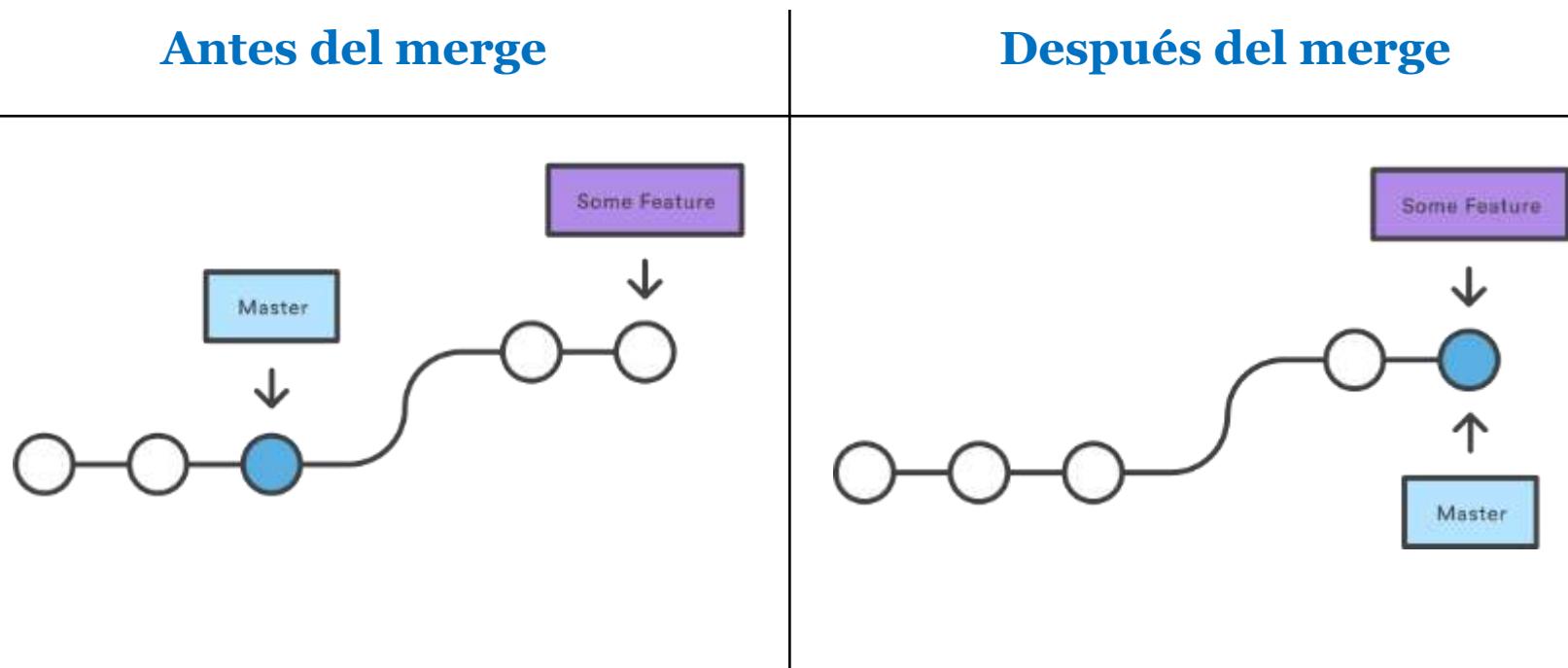
- Cuando fusionamos dos ramas los cambios de una rama se añaden a la otra rama.
- **Ejemplo:** fusionar development y master
- La forma más sencilla de hacerlo es:
  - `git checkout master`
  - `git merge development`



# Conflictos al fusionar ramas



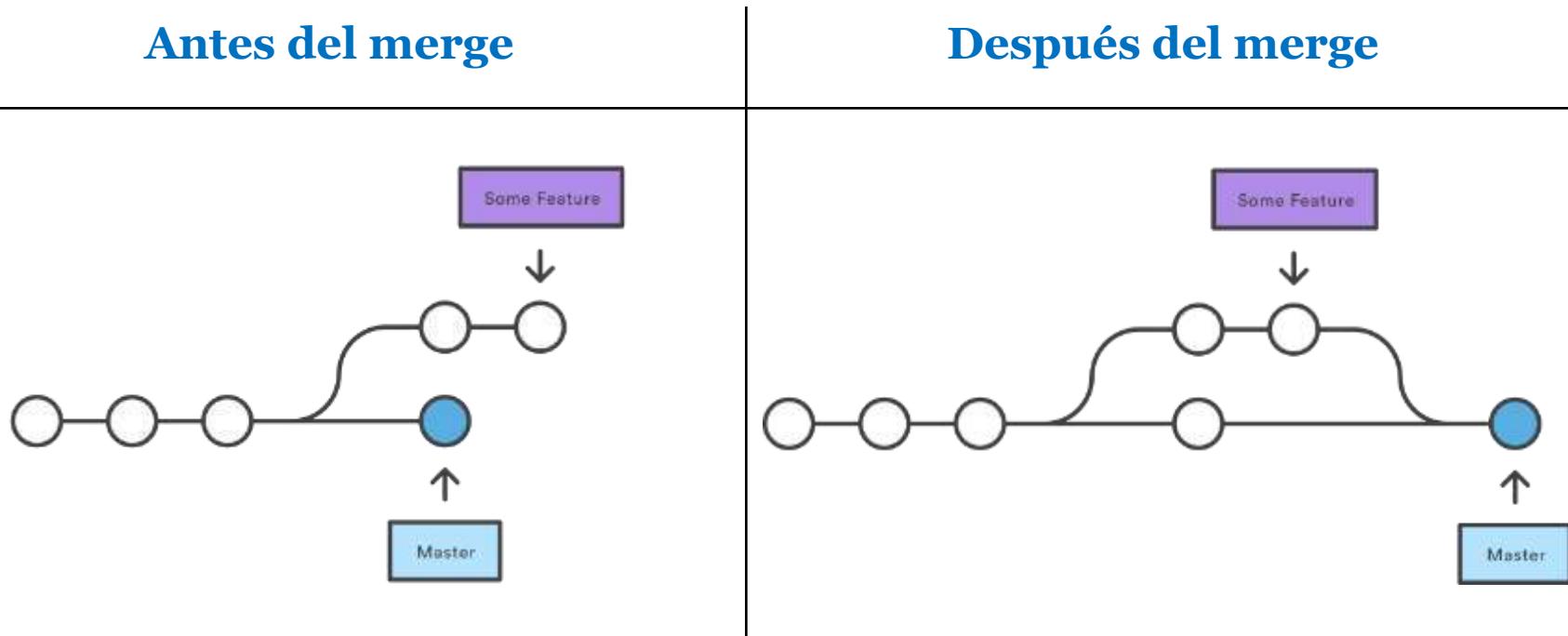
- Situación más simple (*fast-forward merge*)



# Conflictos al fusionar ramas



- Situación más compleja (*3-way merge*):



# Conflictos al fusionar ramas



- Varias situaciones posibles:
  - Merge de una rama en otra sin haber tocado los mismos archivos
    - Los cambios se hacen bien.
  - Se ha tocado el mismo archivo en ambas ramas pero editando partes separadas
    - Git detecta que no has tocado la misma región y se suelen integrar bien.
  - Se ha modificado el mismo archivo en ambas ramas y se ha modificado la misma región
    - Git genera un conflicto porque no sabe con qué modificación quedarse.
    - Git nos indica que reparemos estos problemas manualmente.
    - Una vez arreglados se vuelve a enviar.

# Ejemplo



- Modificamos Hola.cpp en la rama master:

```
#include <iostream>

void print ()
{
    std::cout << "Hello world!!" << std::endl;
}

int main()
{
    print();
    return 0;
}
```

- Fusionamos las ramas:
  - `git checkout master`
  - `git merge development`

# Ejemplo



```
1. zsh
joseangl@192 ~/git_project $ git merge development
Auto-merging Hola.cpp
CONFLICT (content): Merge conflict in Hola.cpp
Automatic merge failed; fix conflicts and then commit the result.
joseangl@192 ~/git_project $ git status
HEAD detached from 6c47001
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:

  new file:  print.cpp
  new file:  print.h

Unmerged paths:
  (use "git add <file>..." to mark resolution)

  both modified:  Hola.cpp

joseangl@192 ~/git_project $
```

# Conflictos al fusionar ramas

---



- Git edita el contenido de los archivos con conflictos con las siguientes marcas:
  - <<<<<
  - =====
  - >>>>>
- El contenido antes de ===== se suele referir a la rama que recibe los cambios (master).
- **Resolución del conflicto:** editar manualmente el fichero con conflictos y ejecutar `git add <fichero>`.
- Finalmente se ejecuta un `git commit` para mezclar las ramas.

# Ejemplo



- Fichero Hola.cpp después del **git merge**:

```
#include "print.h"

<<<<< HEAD
void print ()
{
    std::cout << "Hello world!!" << std::endl;
}

int main()
{
    print();
=====

        //Llamada a la funcion que imprime "Hello world!!"
        print_hello_world();
>>>>> development
        return 0;
}
```

# Ejemplo



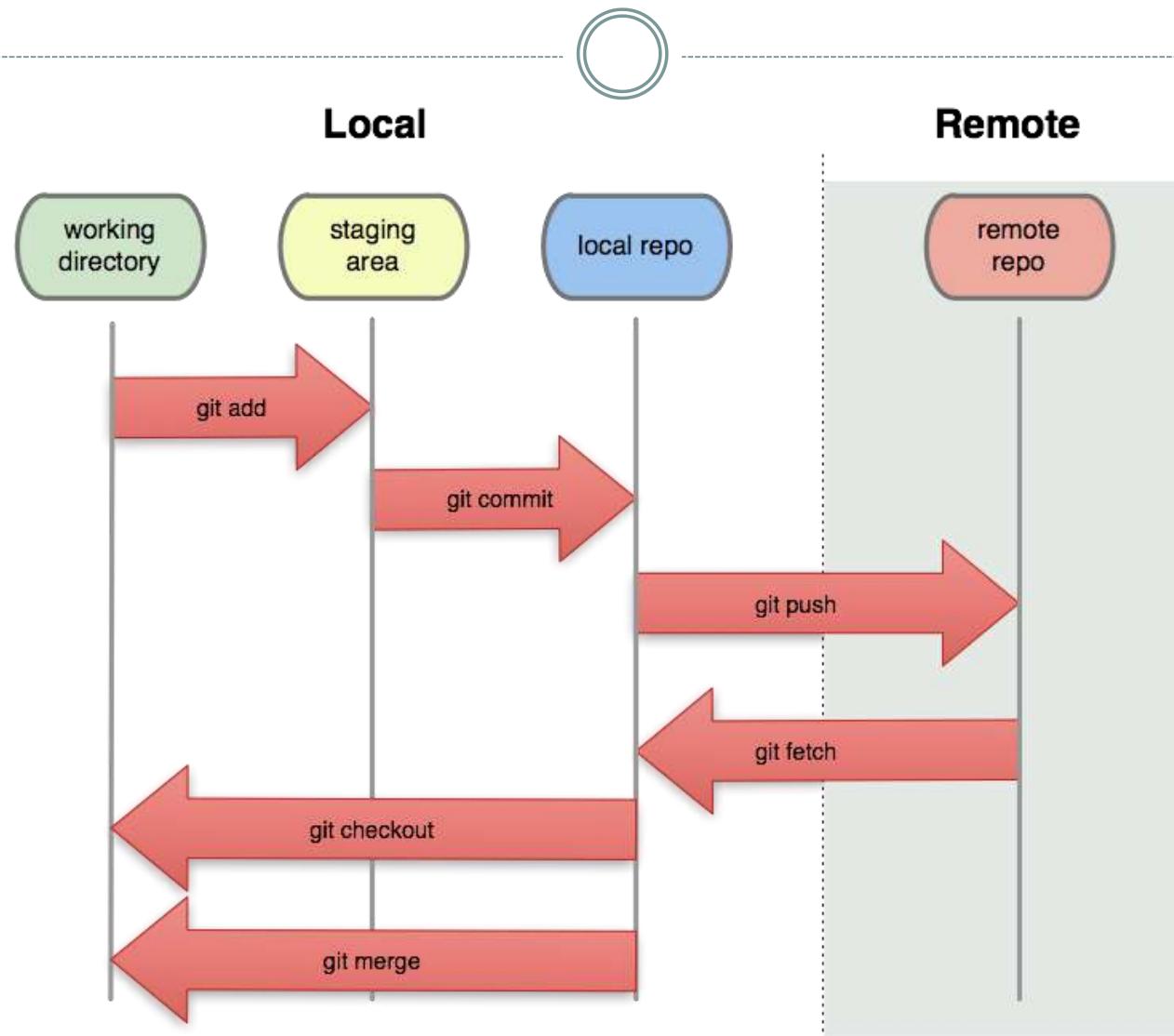
- Editamos el fichero

```
#include "print.h"

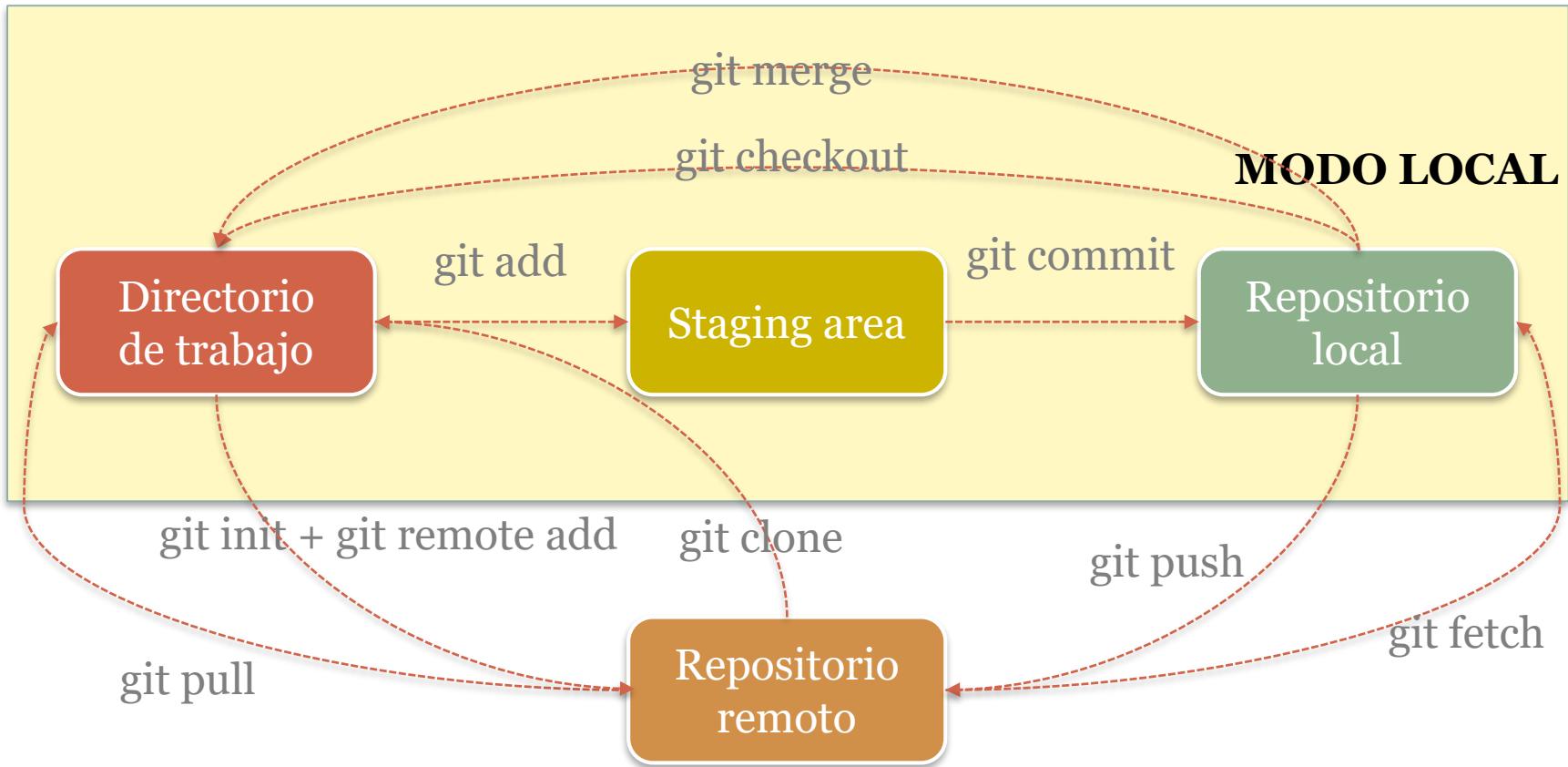
int main()
{
    //Llamada a la funcion que imprime "Hello world! !"
    print_hello_world();
    return 0;
}
```

- Resolvemos el conflicto con:
  - **git add Hola.cc**
  - **git commit -m “Merge de master y development”**
- Para eliminar la rama development:
  - **git branch -d development**

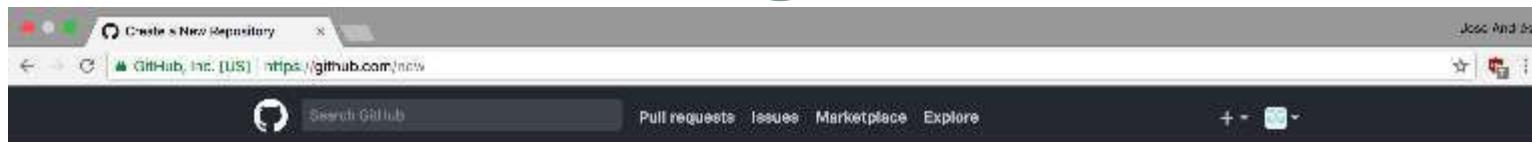
# Git en modo remoto



# Flujo básico de trabajo remoto en Git



# Creando el repositorio en GitHub



## Create a new repository

A repository contains all the files for your project, including the revision history.

Owner	Repository name
joseangl	/github-project

Great repository names are short and memorable. Need inspiration? How about [special-winner](#).

Description (optional)

Proyecto de ejemplo para introducción a la Ingeniería del Software

Public  
Anyone can see this repository. You choose who can commit.

Private  
You choose who can see and commit to this repository.

Initialize this repository with a README  
This will let you immediately clone the repository to your computer. Skip this step if you're importing an existing repository.

Add a license: None ▾ Add a license: Apache License 2.0 ▾ ⓘ

**Create repository**

# Contenido del repositorio



The screenshot shows a GitHub repository page. At the top, there's a navigation bar with links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the navigation, the repository name 'joseangl/github-project' is displayed, along with statistics: 1 commit, 1 branch, 0 releases, 1 contributor, and Apache-2.0 license. A 'Clone or download' button is highlighted in red. The main content area shows a file list with 'LICENSE' and 'README.md' files. The 'README.md' file is expanded, showing its content: 'Proyecto de ejemplo para Introducción a la Ingeniería del Software'.

## github-project

Proyecto de ejemplo para Introducción a la Ingeniería del Software

# Creando un repositorio nuevo desde local

---



- Crear un nuevo repositorio local desde cero
  - Nos situamos en el directorio desde donde cuelga el proyecto.
  - Creamos el repositorio local:
    - ✖ `git init`
- Creamos un repositorio remoto vacío.
- En el directorio de trabajo ejecutamos:
  - `git remote add <nombre remoto> <URL GitHub>`
  - Al repositorio remoto se le suele denominar `origin`.
  - La URL nos la da GitHub en el botón `Clone or download`.

# Crear un repositorio nuevo desde remoto

---



- Crear un nuevo repositorio remoto no vacío.
- Clonar un repositorio existente
  - Para clonar un repositorio ejecutamos
    - ✖ `git clone <URL> [directorio]`
  - El repositorio local es una copia del repositorio remoto.

# Actualización de datos entre repositorio local y remoto

---



- Subir datos al repositorio remoto:
  - `git push <nombre remoto> <nombre rama>`
  - Ejemplo: `git push -u origin master`
- Descargar datos del repositorio remoto:
  - `git pull <nombre remoto> <nombre rama>`
  - Ejemplo: `git pull origin master`
  - Esta orden también fusiona los cambios del repositorio remoto en el local (merge) → puede generar conflictos.

# Ejemplo



```
1. zsh
joseangl@192 ~/git_project $ git remote add origin https://github.com/joseangl/practica-git.git
joseangl@192 ~/git_project $ git push -u origin master
Counting objects: 6, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (5/5), done.
Writing objects: 100% (6/6), 641 bytes | 0 bytes/s, done.
Total 6 (delta 0), reused 0 (delta 0)
To https://github.com/joseangl/practica-git.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
joseangl@192 ~/git_project $
```

# Otras órdenes útiles



- Sobreescribir el repositorio local con los datos del remoto:
  - `git fetch origin`
  - `git reset --hard origin/master`