



DISEÑO DE SOFTWARE

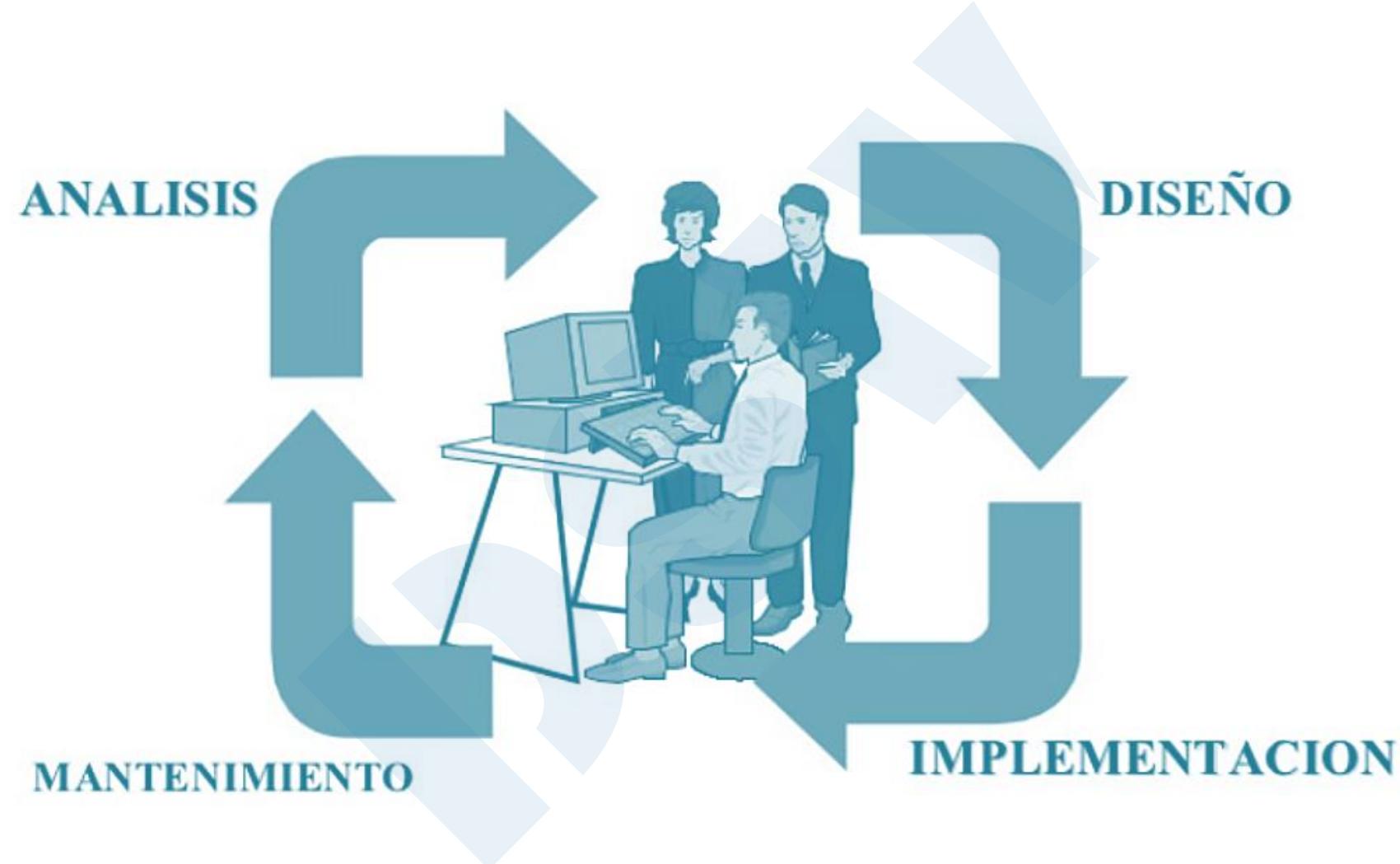
Conceptos del Diseño

Sesión S1

Agenda

- Conceptos del Diseño
- Procesos de Desarrollo de Software.
- Metodologías de desarrollo de software,
- Metodologías Agiles.
- Ciclo de vida del desarrollo

Introducción



Análisis Orientado a Objetos

Cual es el objetivo del análisis orientado a objetos ?



Diseño Orientado a Objetos

Que es el Diseño orientado a objetos ?



Conceptos del Diseño



SOLID



Abstracción

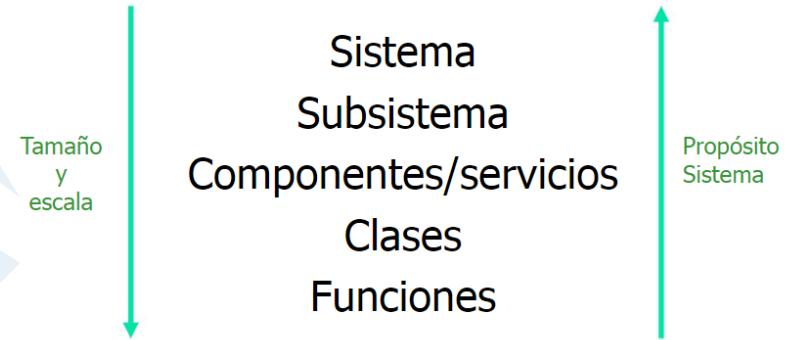


Proceso



Metodología

Método

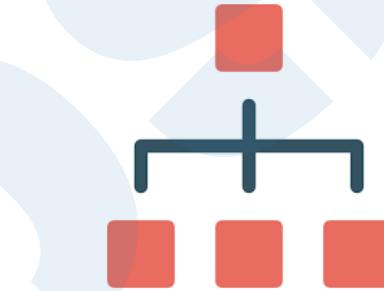
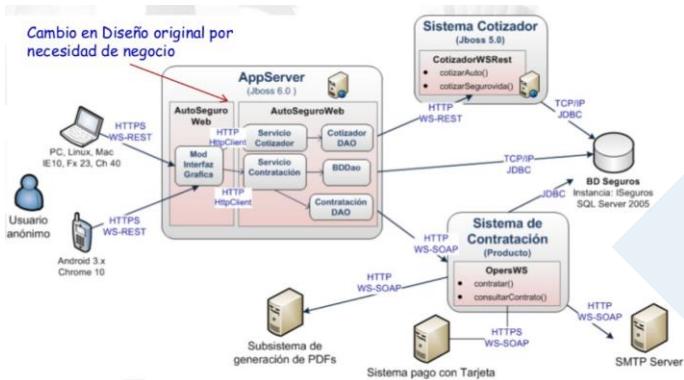
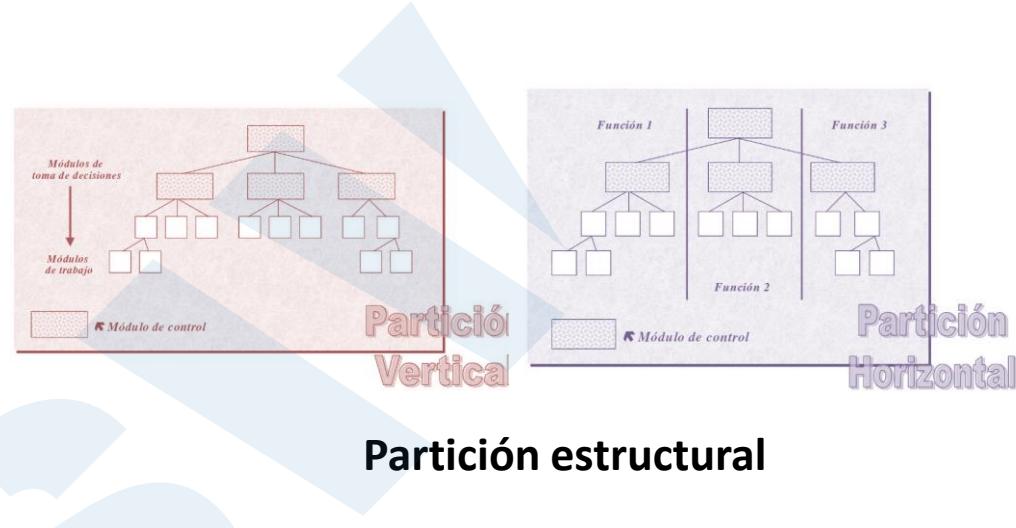


Refinamiento

Conceptos del Diseño



Modularidad

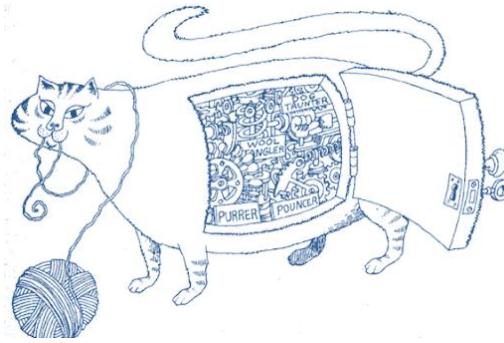


Jerarquía de control



Estructura de datos

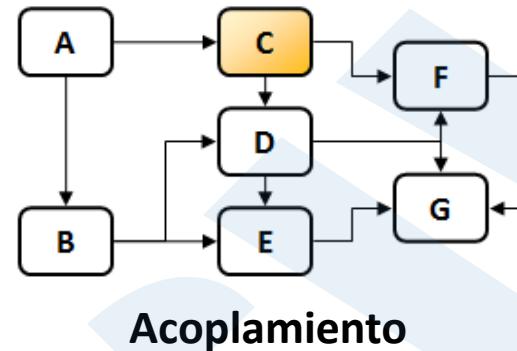
Conceptos del Diseño



Ocultación de información



Independencia funcional



Acoplamiento



Cohesión

Malo™
Baja cohesión
Alto acoplamiento

Cohesión

Baja cohesión
Bajo acoplamiento



¿?
Alta cohesión
Alto acoplamiento

Alta cohesión
Bajo acoplamiento

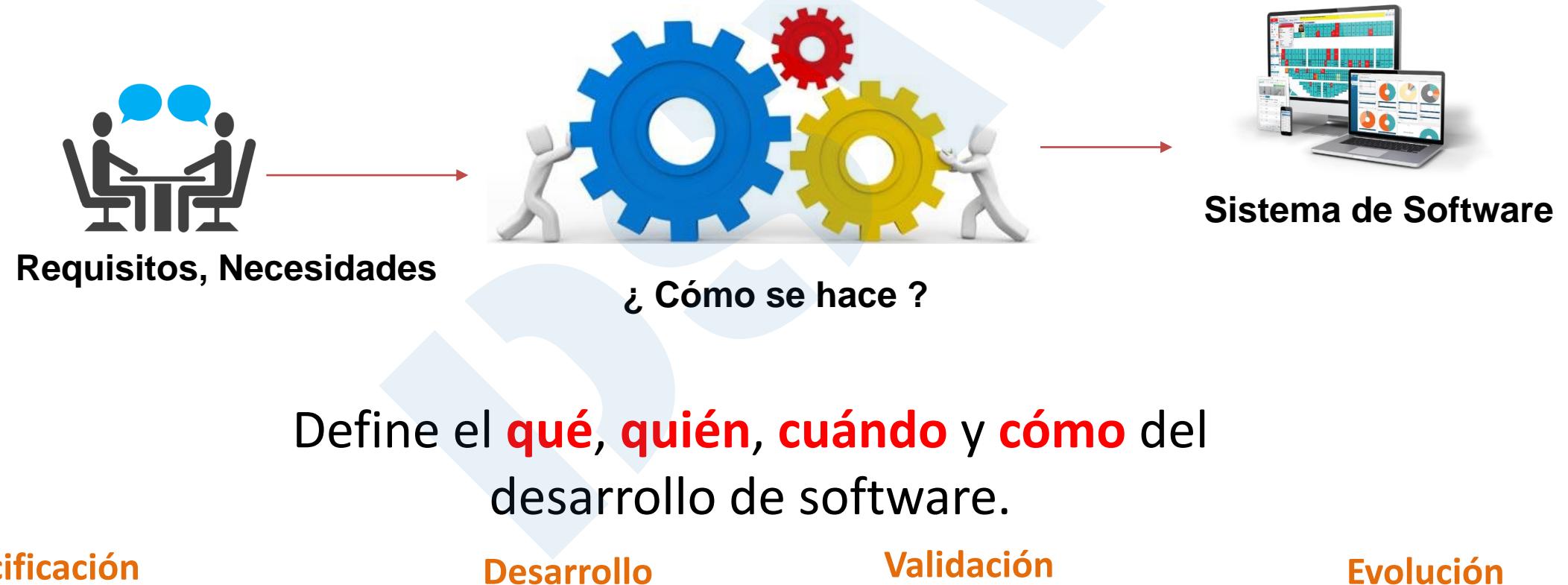
Bueno™

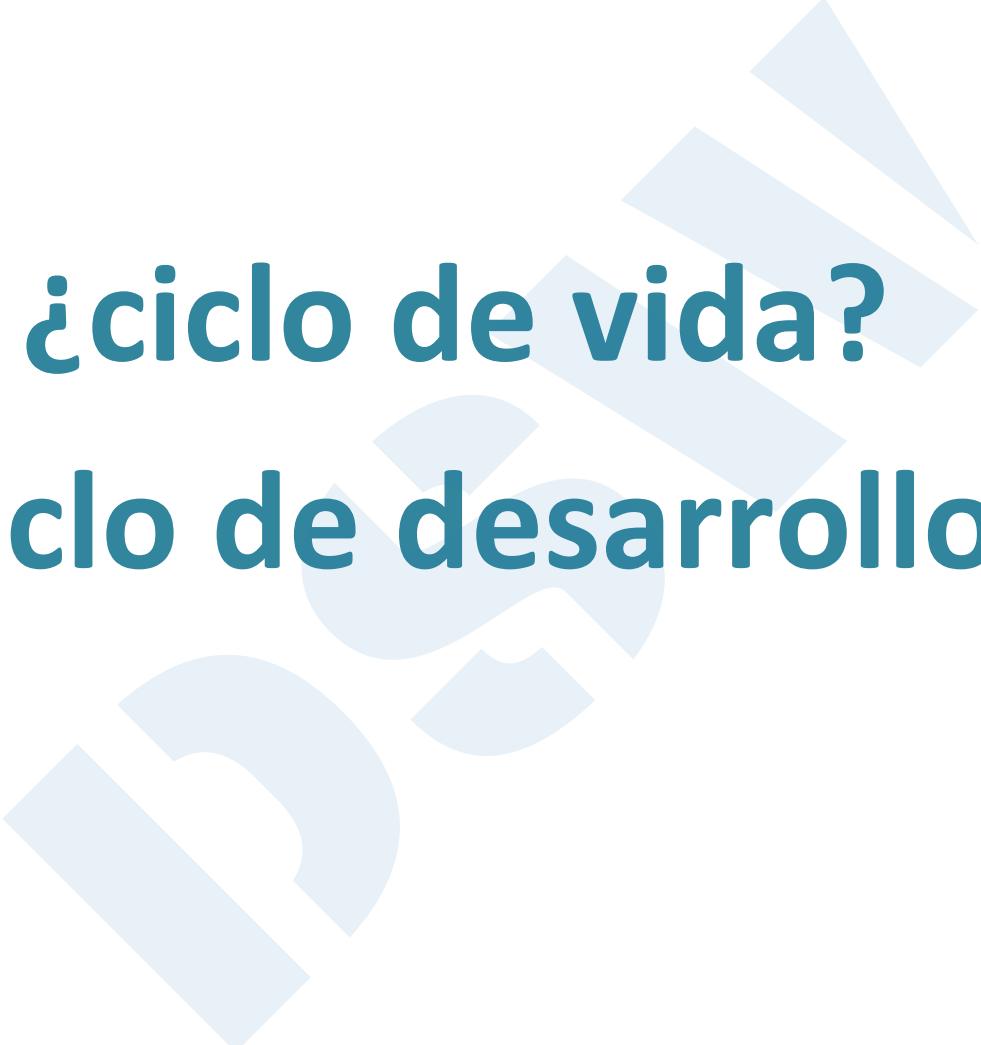
Conceptos del Diseño

El diseño combina

-
- The diagram illustrates the components of design through three main categories, each represented by a red bracketed label and a corresponding list of elements:
- Del ingeniero del software**:
 - Creatividad
 - Intuición
 - Experiencia
 - Del proceso de diseño**:
 - Guías
 - Métodos
 - Heurísticas
 - Diseño Final**:
 - Criterios de calidad
 - Proceso iterativo

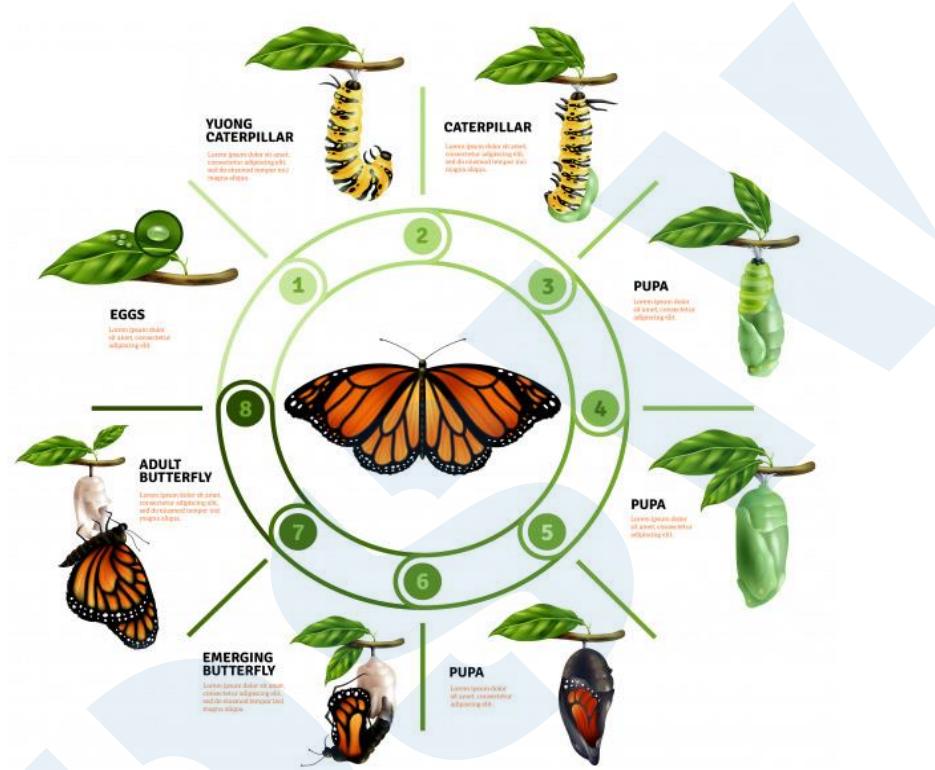
Procesos de Desarrollo de Software





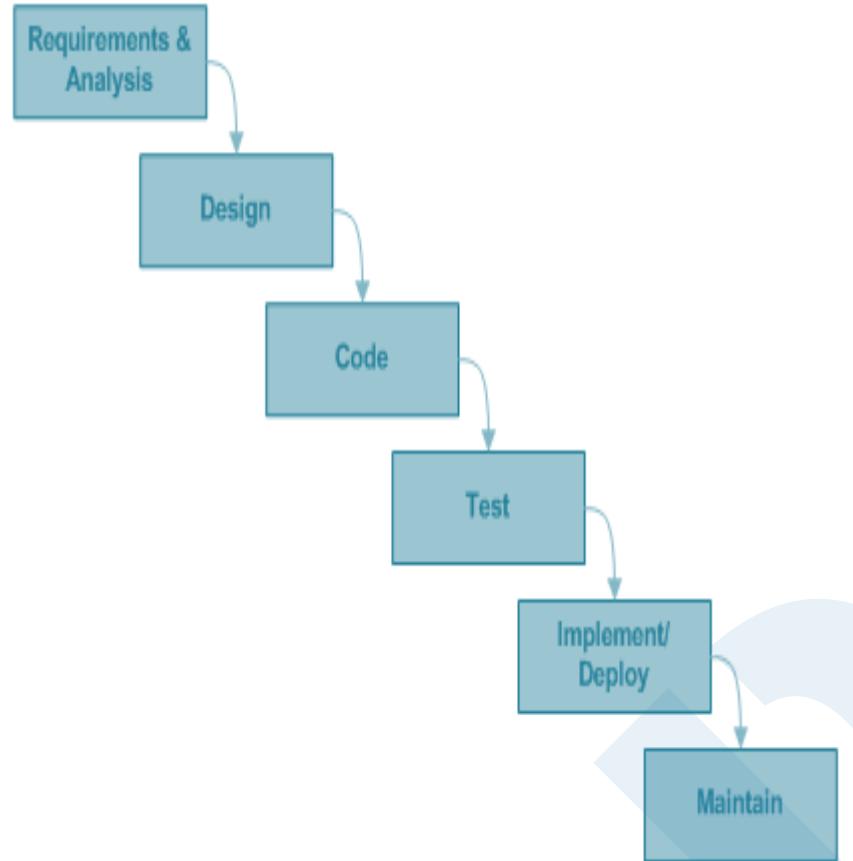
¿ciclo de vida?
¿ciclo de desarrollo?

Procesos de Desarrollo de Software

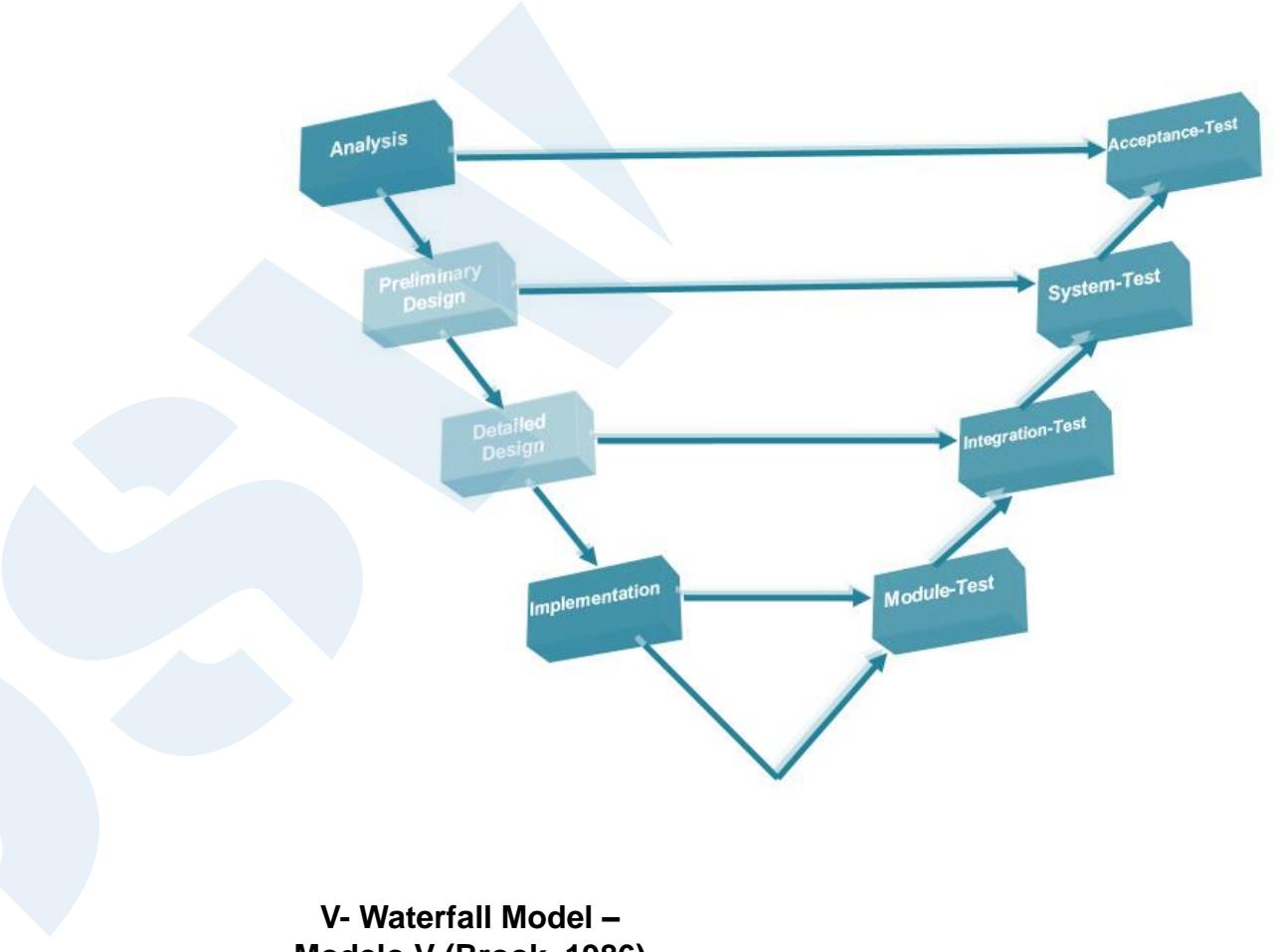


Describe la vida de un producto de software desde su definición, pasando por su diseño, implementación, verificación, validación, entrega, y hasta su operación y mantenimiento

Tipos de modelos de procesos

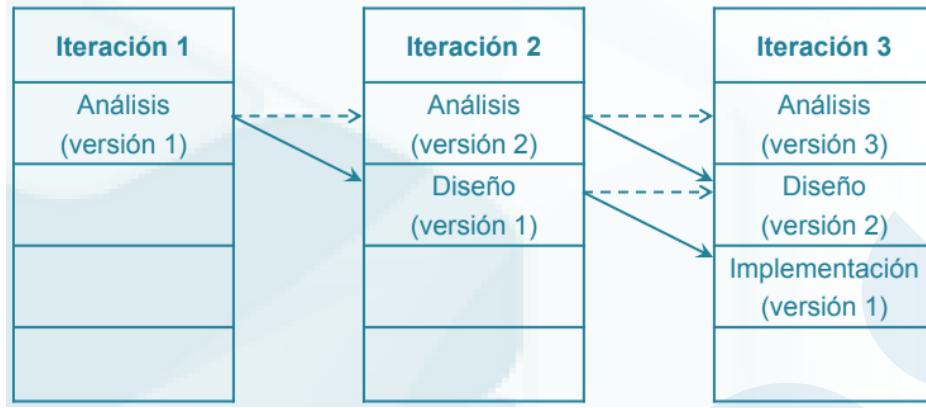


Waterfall Model –
Modelo cascada (Royce, 1970)
<http://www.waterfall-model.com/>

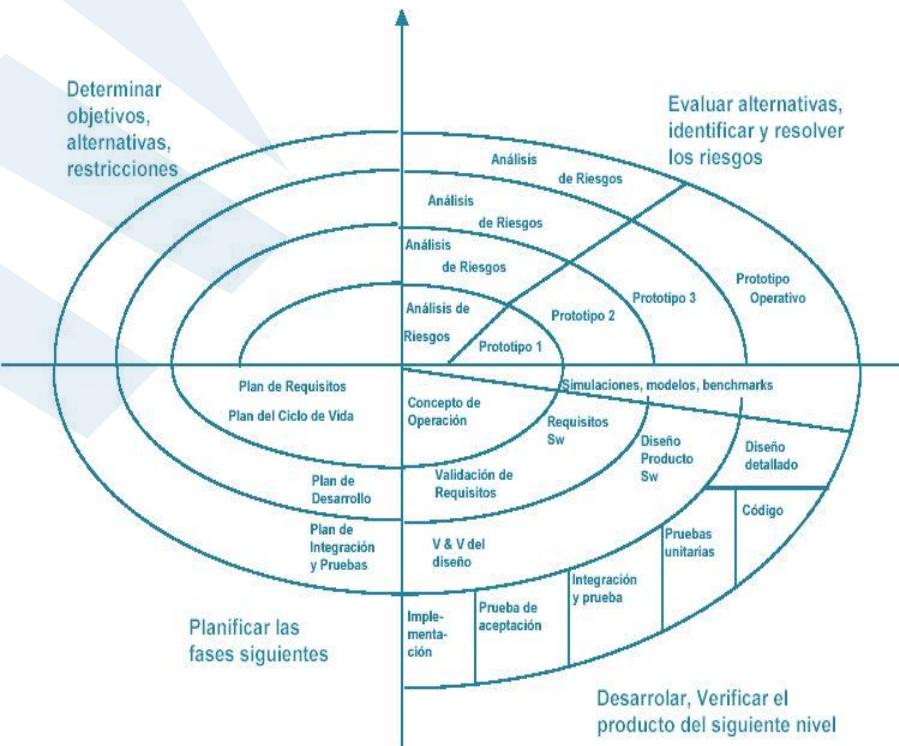


V- Waterfall Model –
Modelo V (Brook, 1986)
<http://www.waterfall-model.com/>

Tipos de modelos de procesos

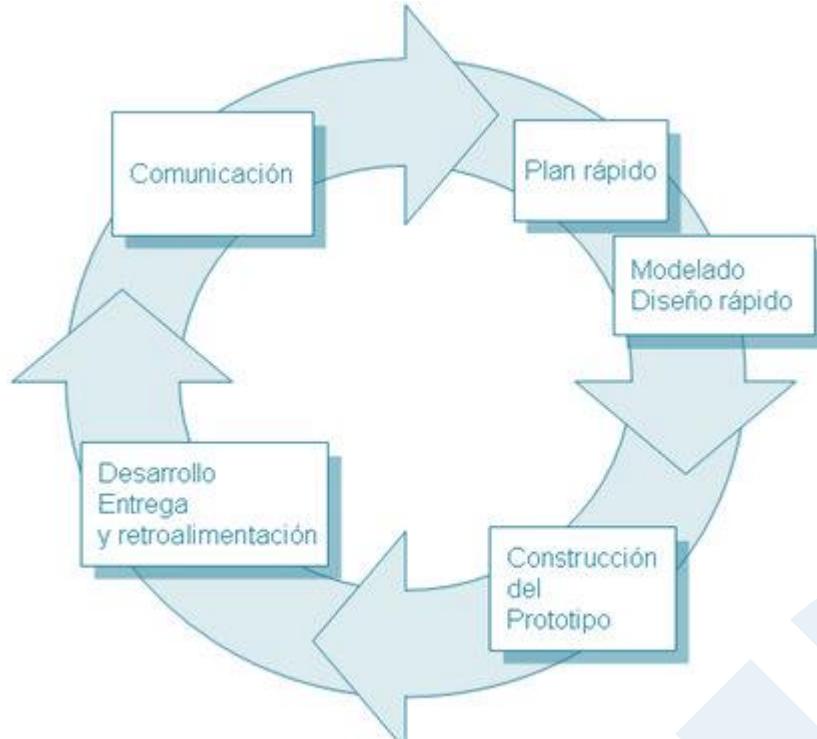


Modelo evolutivo
(Boehm en 1988)

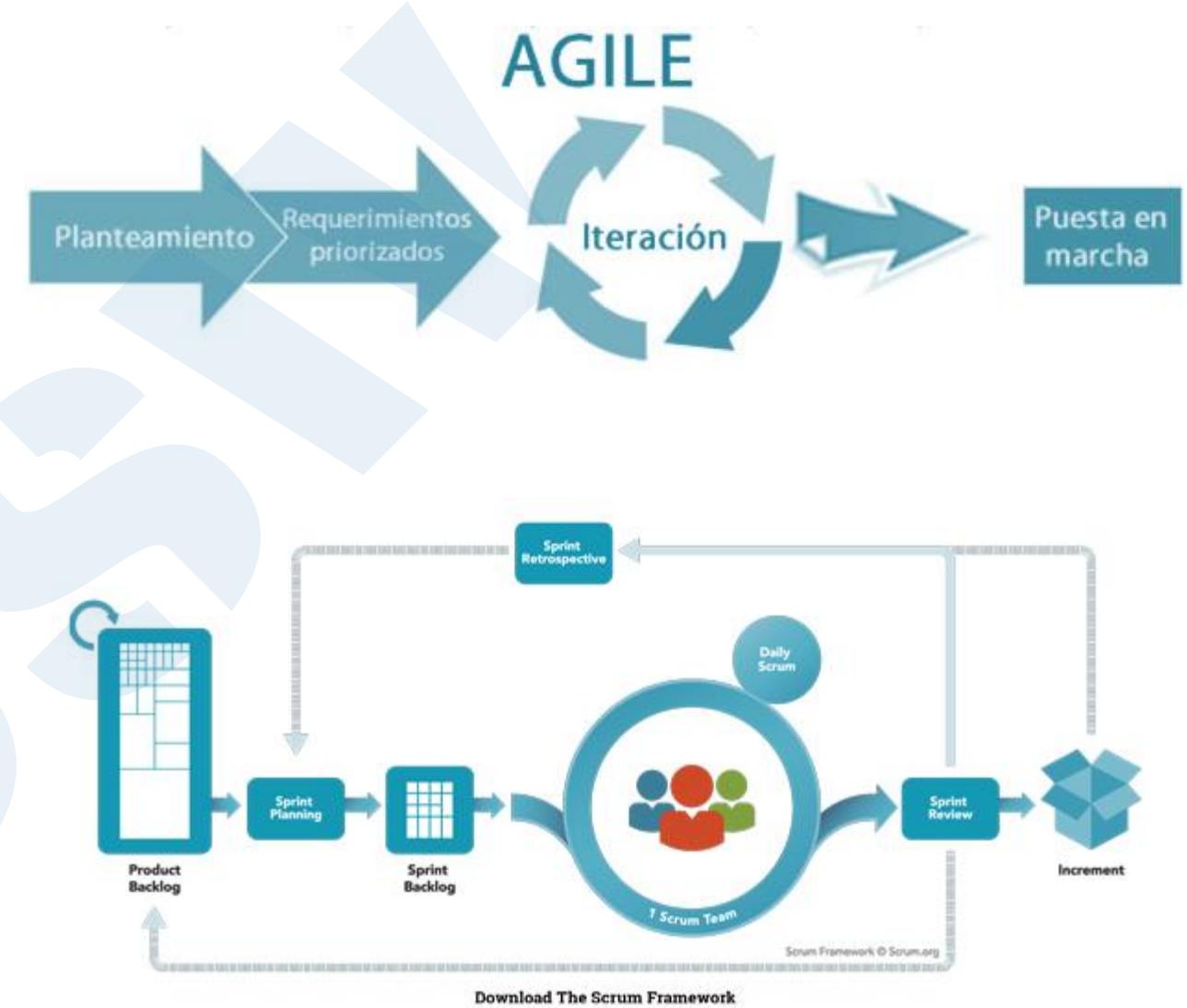


Espiral
(Boehm en 1988)

Tipos de modelos de procesos



Modelo de Prototipos
(Gomma en 1984)



Download The Scrum Framework





Referencias bibliográficas

Pressman, R. Ingeniería del software: un enfoque práctico. McGraw-Hill, 1998.



UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.



DISEÑO DE SOFTWARE

Metodología Agil
Sesión S1 - Laboratorio

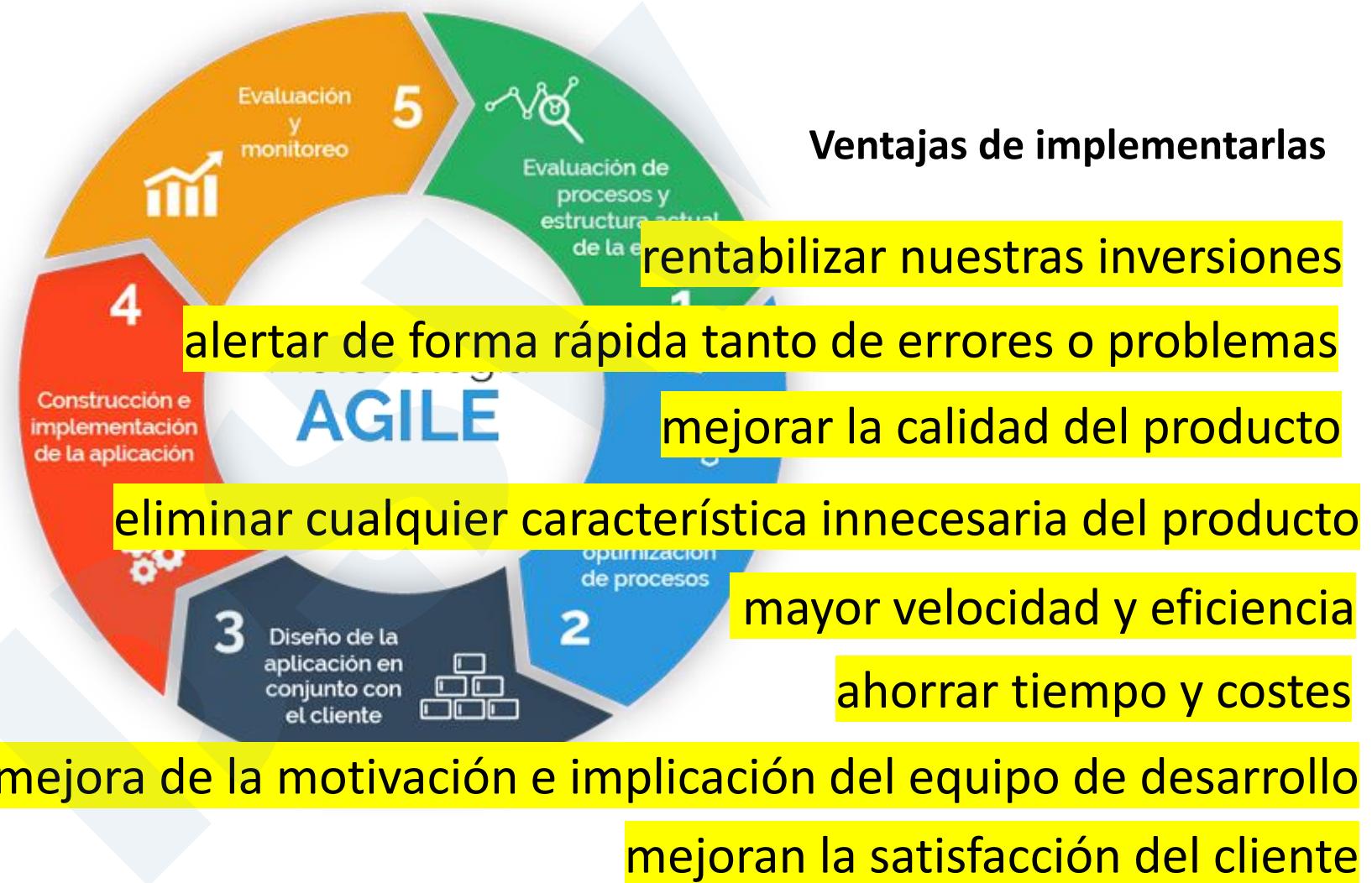
Agenda

- Metodología Ágil
- Manifiesto Ágil y principios
- Scrum
- XP

Metodologías Agiles.

Manifiesto por el Desarrollo
Ágil de Software
(4)

Principios del Manifiesto Ágil
(12)



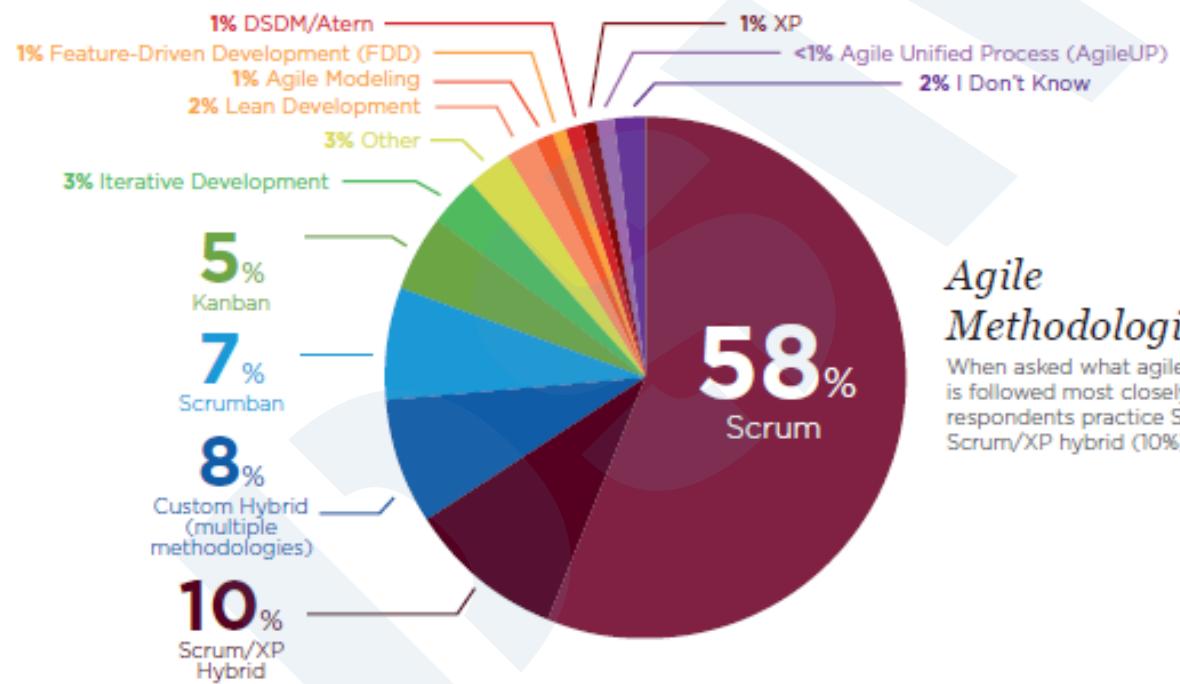
Manifiesto Agil

1. Se prioriza al Individuo y las Interacciones del Equipo
2. Software funcional en lugar de demasiada documentación
3. Colaboración con el Cliente en lugar de Contratos
4. Posibilidad de hacer cambios de planes a medio proyecto



Tendencia Ágil

AGILE METHODS AND PRACTICES



Agile Methodologies Used

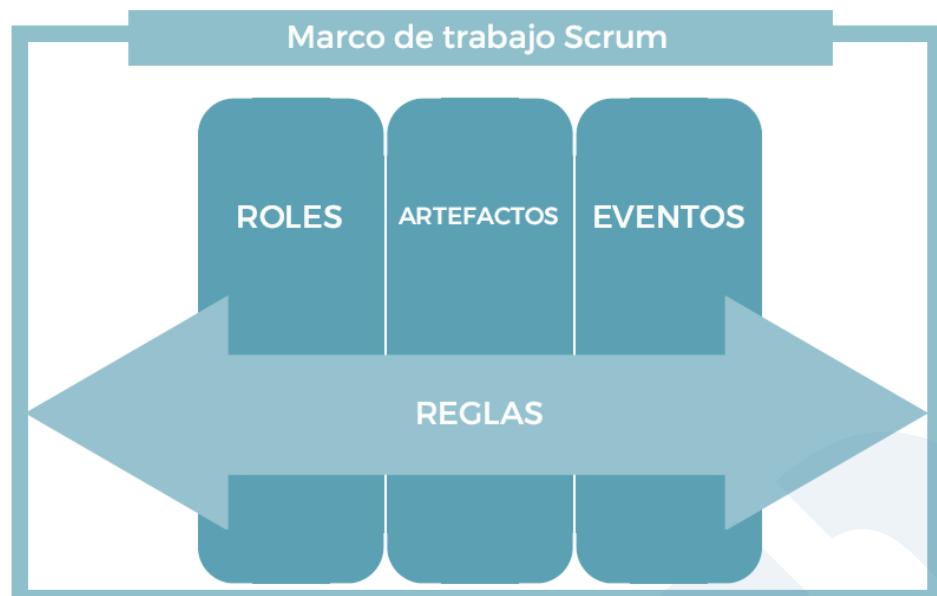
When asked what agile methodology is followed most closely, nearly 70% of respondents practice Scrum (58%) or Scrum/XP hybrid (10%).

En el gráfico siguiente, se puede observar la tendencia (2017) en cuanto al uso de metodologías ágiles.

SCRUM

Qué es SCRUM?

Por qué interesarse por SCRUM?



Product Owner (PO)

- Mantiene la comunicación con el cliente.
- Es el responsable del product backlog.
- Se asegura de que las descripciones están claras.
- Mide el desempeño del producto.

Development team (DT)

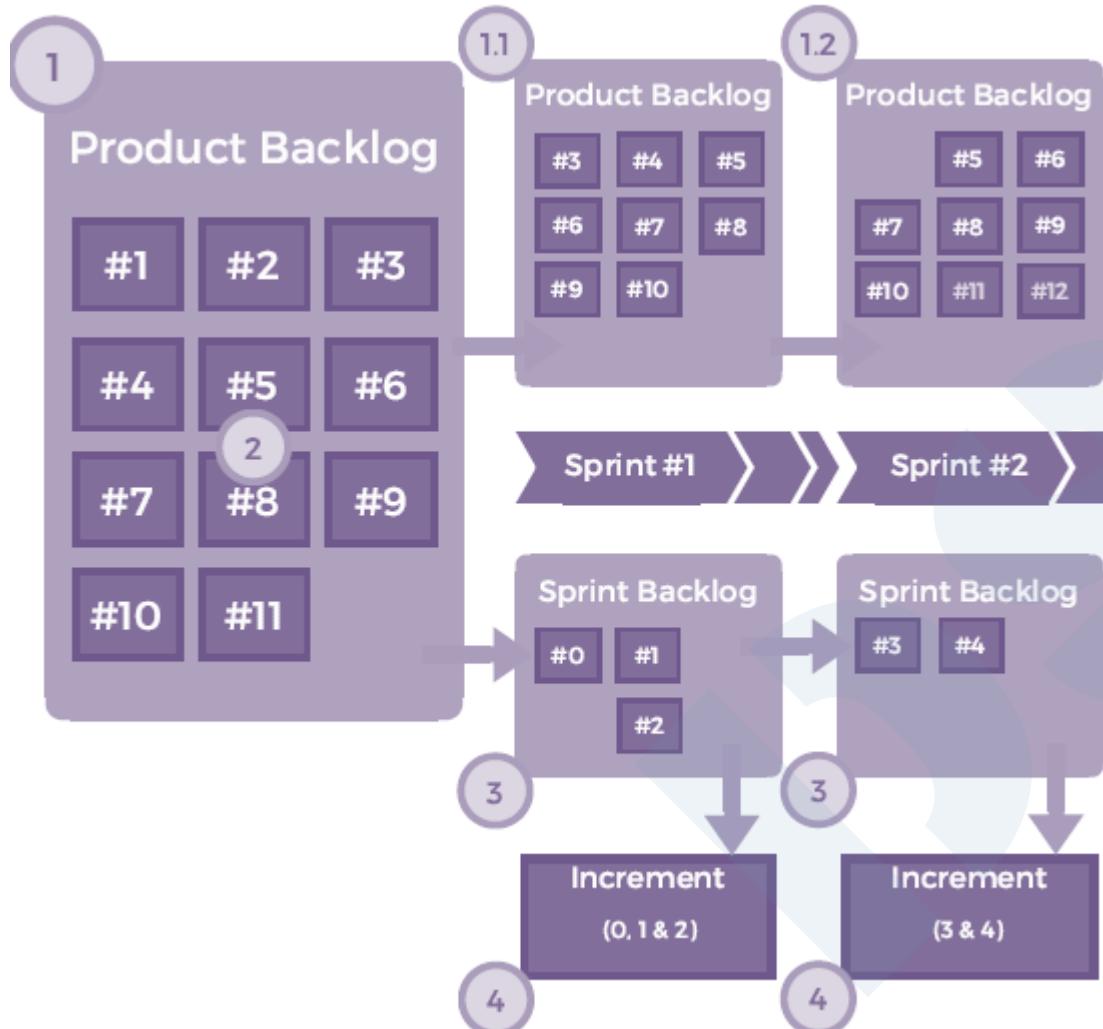
- Responsable de producir el producto o la solución en desarrollo.
- Responsables de gestionar su propio esfuerzo y desempeño.
- Todo el DT es igualmente responsable y rinde cuentas por igual.

Scrum master (SM)

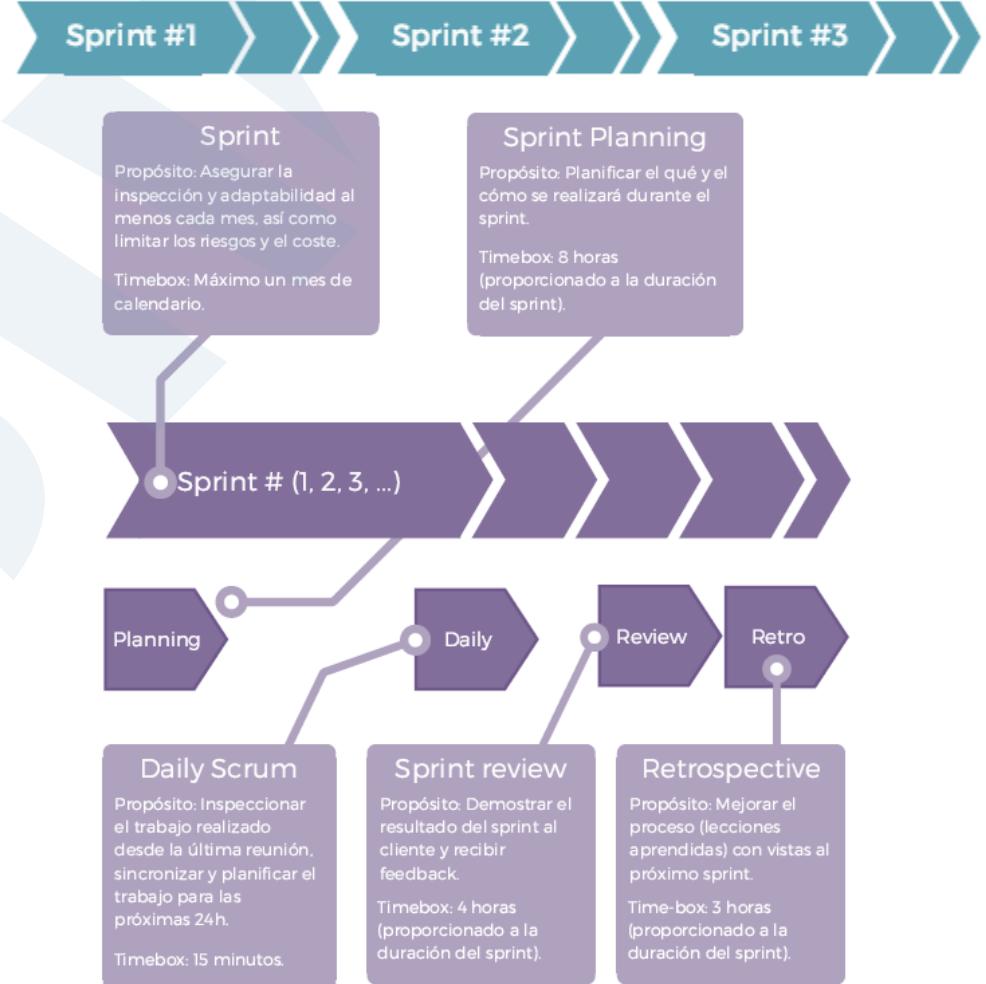
- Experto en la metodología Scrum.
- Responsable de que Scrum sea comprendido y adaptado correctamente.
- Lidera la implantación de Scrum en la organización.
- Elimina los obstáculos que encuentre el DT.

SCRUM

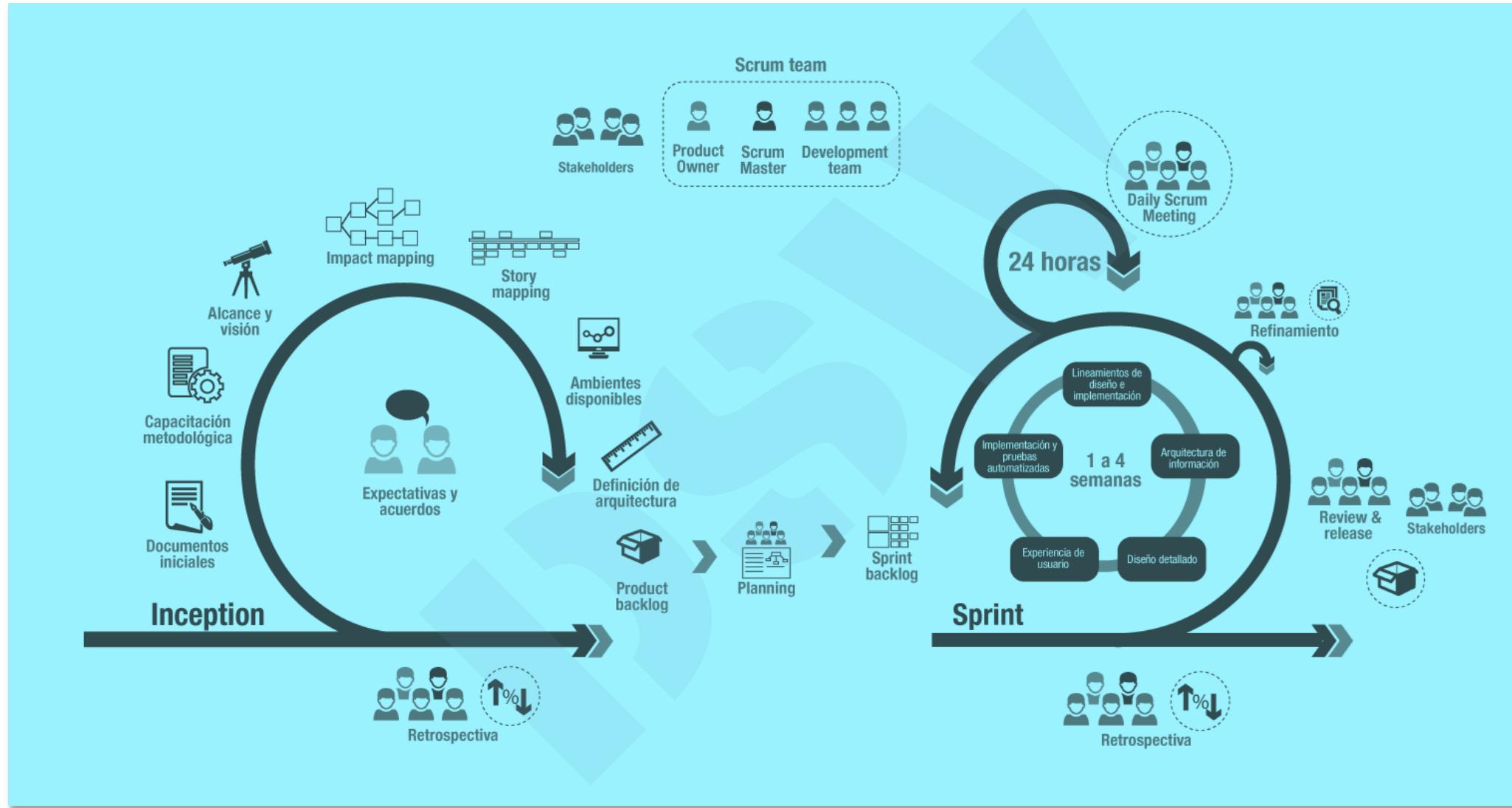
Artefactos



Eventos



El proceso de Scrum





Referencias bibliográficas

Pressman, R. Ingeniería del software: un enfoque práctico. McGraw-Hill, 1998.



UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.



DISEÑO DE SOFTWARE

Arquitectura de Software

Sesión S2

Introducción

Actividades del Análisis

Obtención y modelado de requerimientos

Diagramas inicial de clases

Diagramas inicial de interacción

Actividades del Diseño

Diseño de alto nivel

Diagramas de interacción (refinado)

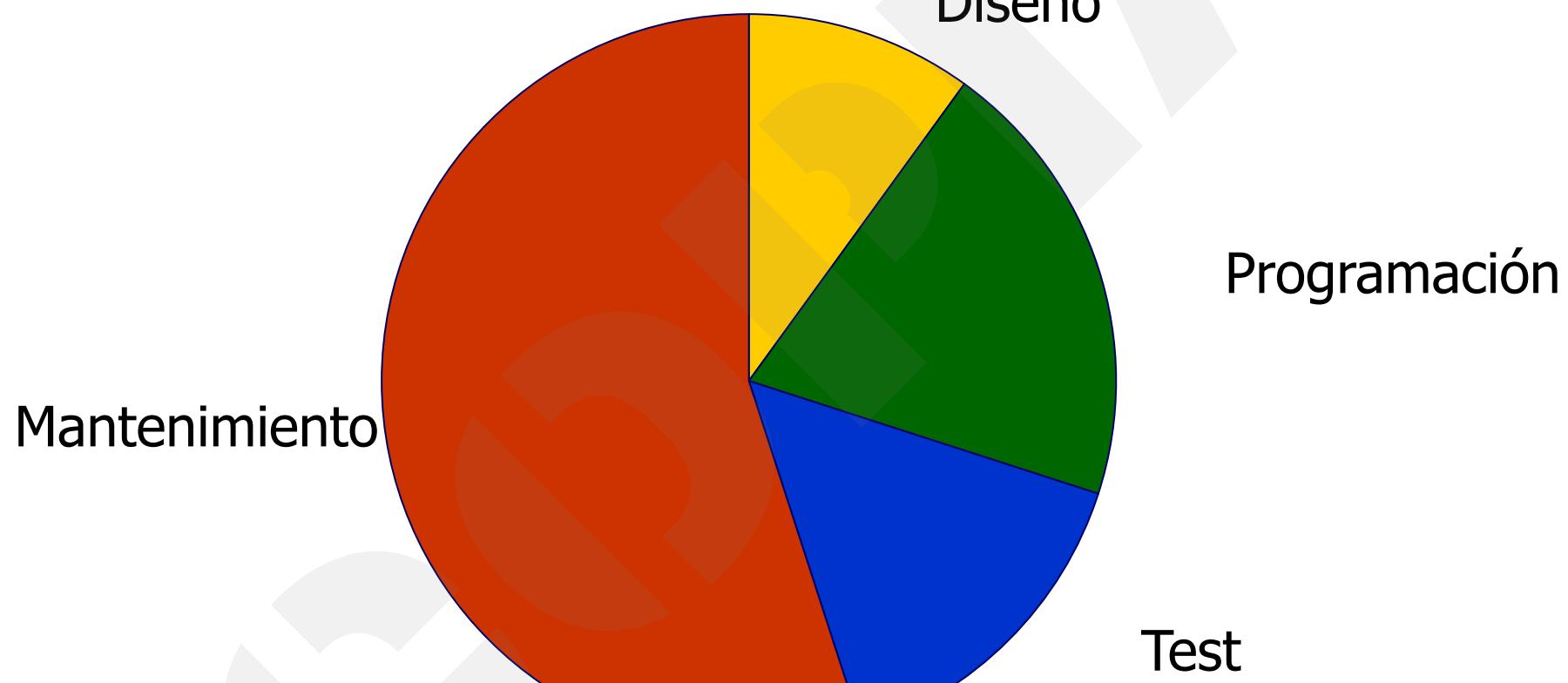
Diagramas de clases de análisis (refinado)

Diagramas de componentes

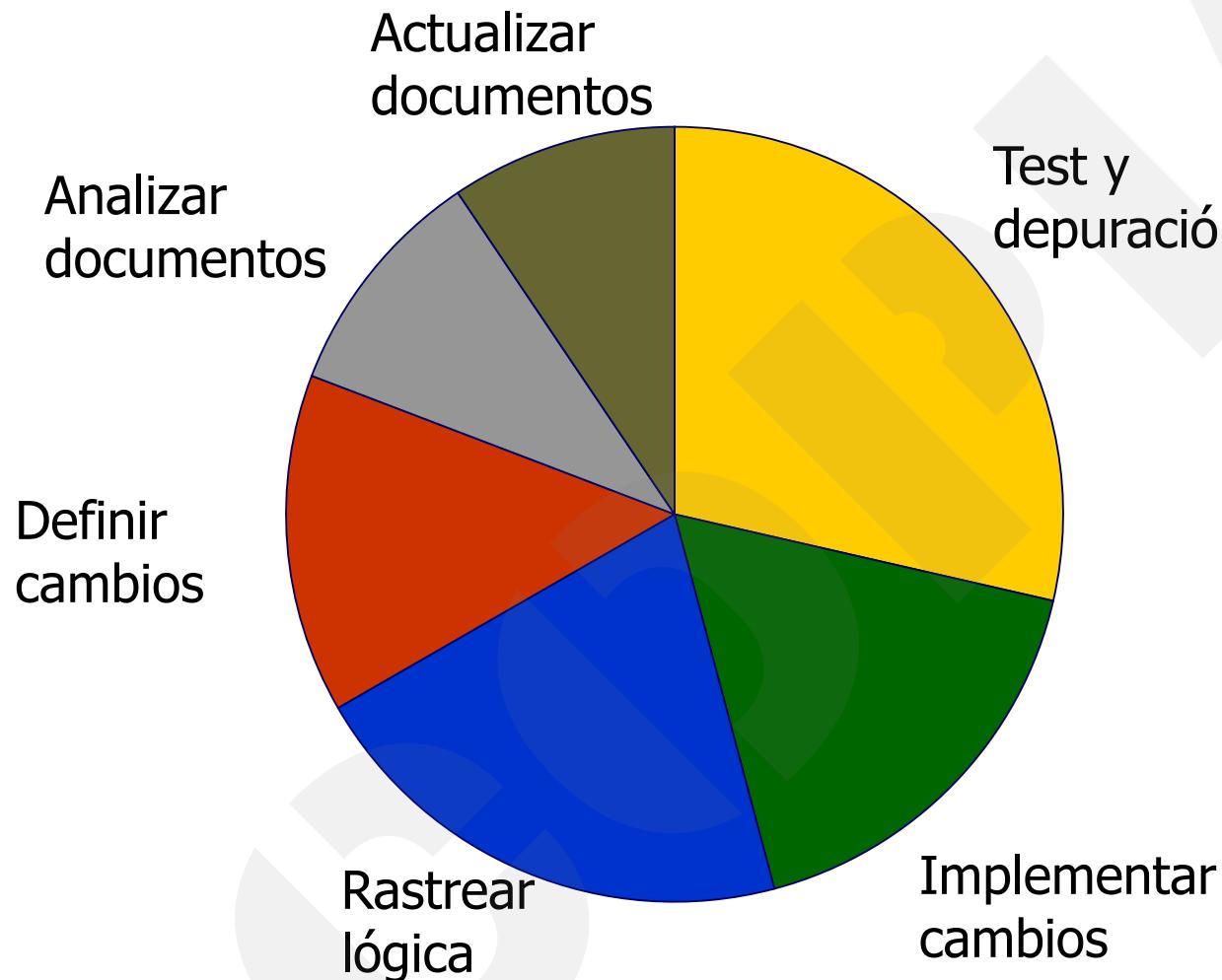
Diagramas de distribución



Distribución del Costo del Software

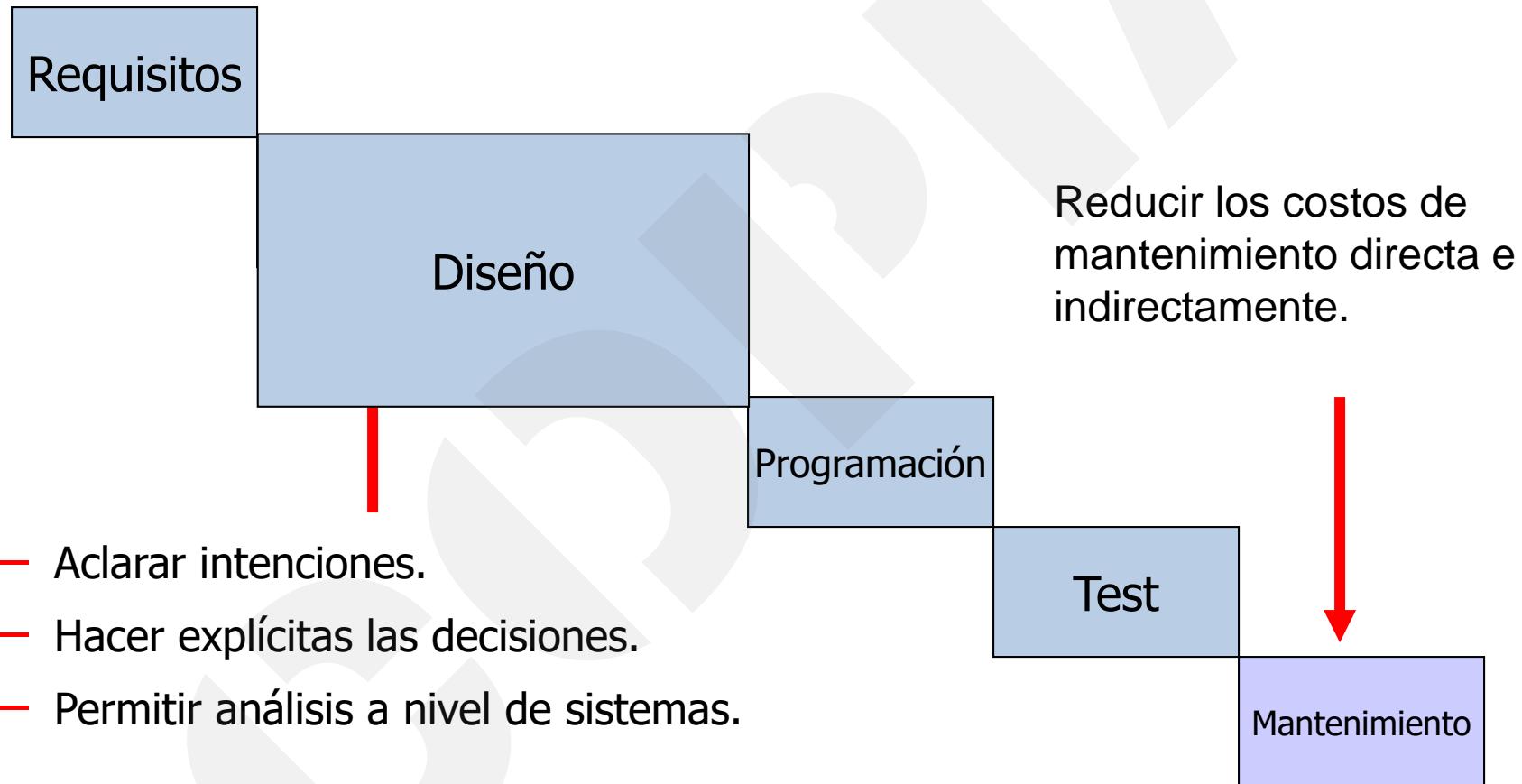


Costos de Mantenimiento del Software



Más del 50% del tiempo del programador encargado de mantenimiento está dedicado a comprender el código y la documentación del sistema.

Arquitectura en el proceso de desarrollo



Rol de la arquitectura



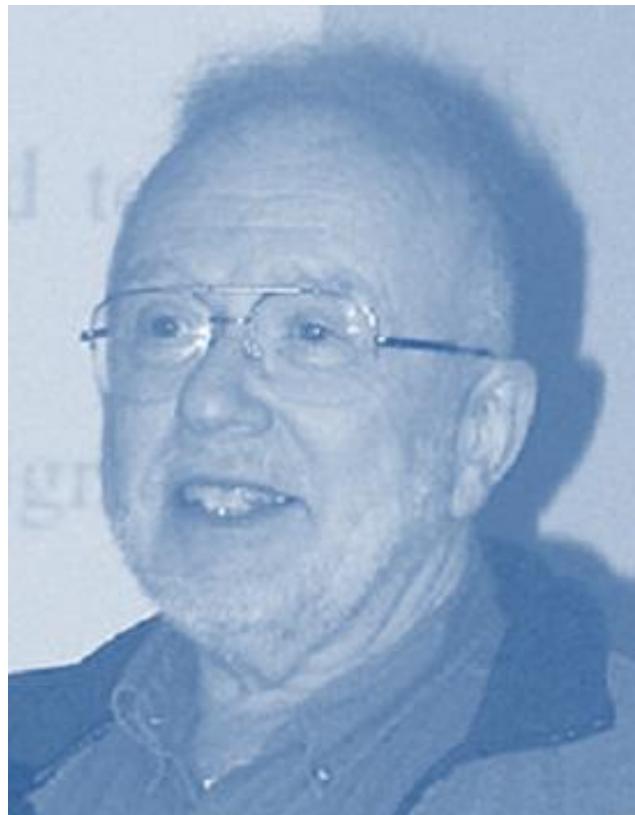
Un poco de historia

El estudio de arquitectura del software se empezó en 1968 cuando Edsger Wybe Dijkstra propuso que se establezca una estructuración correcta de los sistemas de software antes de lanzarse a programar, escribiendo código de cualquier manera.

Dijkstra fue el primero en definir la noción de la estructura de capas.

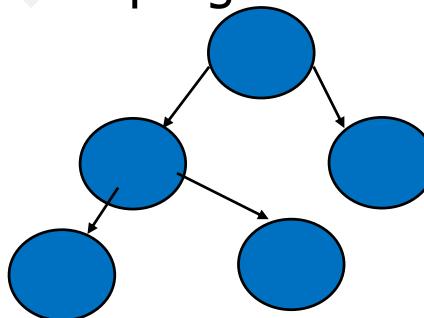


Un poco de historia



David Parnas hizo sus contribuciones acerca de los módulos de información-oculta (1972), la estructura del software (1974), y las familias de programas (1975)

Familia de
programas



Arquitectura de Software

“Una arquitectura es el conjunto de decisiones significativas sobre la organización de un sistema de software que define los principios que guían el desarrollo, los componentes principales del sistema, sus responsabilidades y la forma en que se interrelacionan”

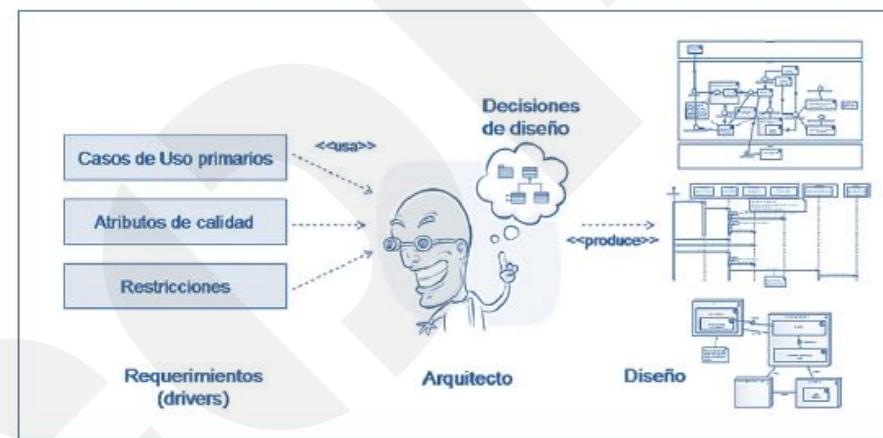
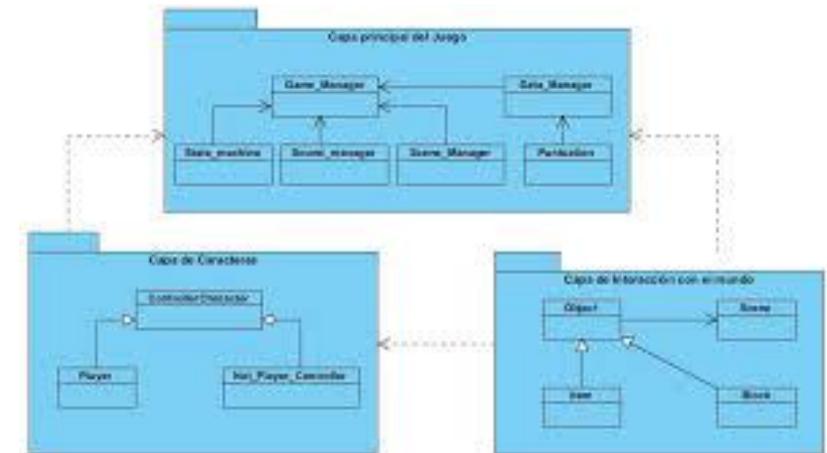


Figura 1: Durante el diseño de la arquitectura, el arquitecto toma como entrada los requerimientos que influyen la arquitectura (drivers) y produce un diseño arquitectónico.

Arquitectura de SW

La Arquitectura del Software es el **diseño de más alto nivel** de la estructura de un sistema, programa o aplicación y tiene la responsabilidad de:

- Definir los módulos principales
- Definir las responsabilidades que tendrá cada uno de estos módulos
- Definir la interacción que existirá entre dichos módulos.

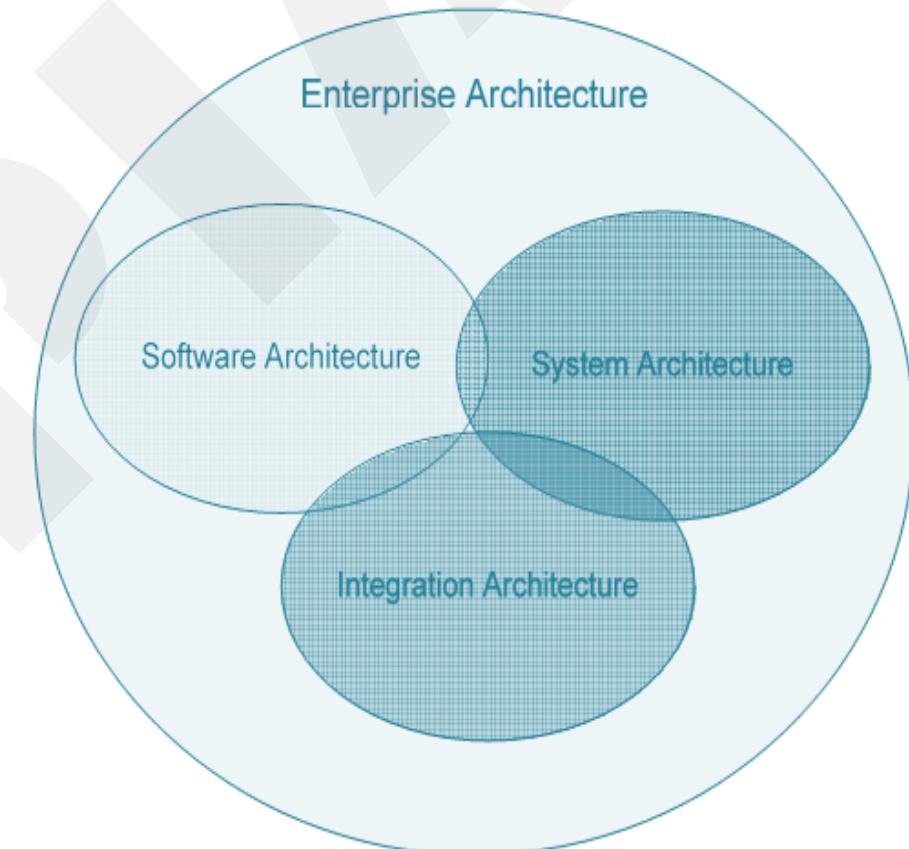


Arquitectura de Software es..

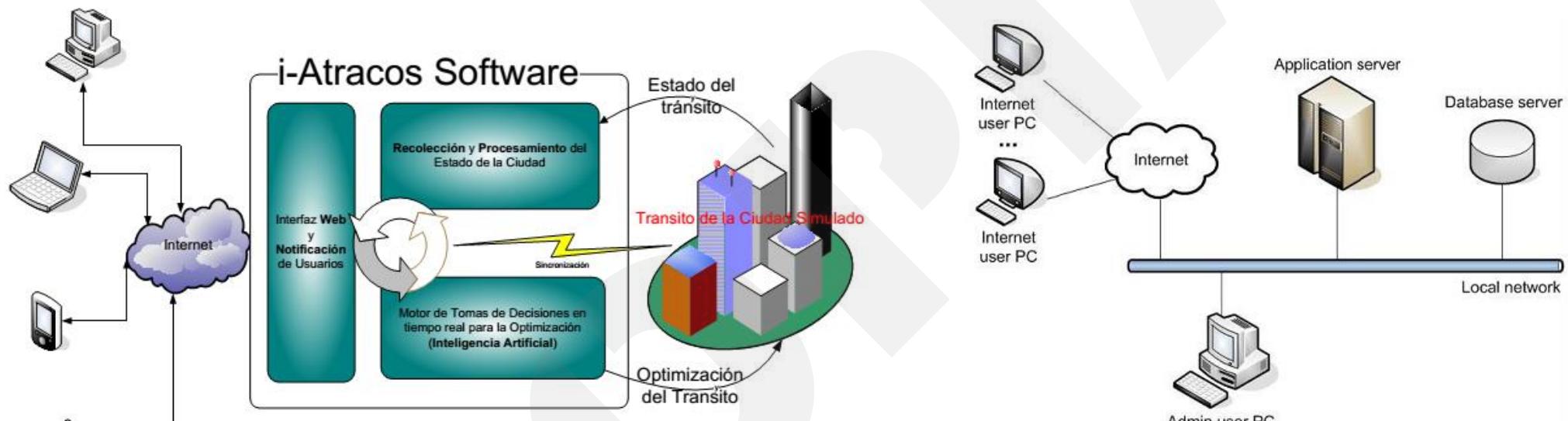
Características de la Arquitectura de SW

Principios de la Arquitectura

- Abstracción
- Encapsulamiento
- Separación de responsabilidades
- Acoplamiento y Cohesión
- No Duplicación
- Parametrización y Configurabilidad
- Claridad y simplicidad
- Separación de interfaz e implementación



Ejemplo



Arquitectura
no es...
Arquitectura
estructural

Los elementos estructurales incluyen:

- la organización y el control globales,
- los protocolos de comunicación,
- la distribución física,
- la composición de elementos de diseño,
- la escalabilidad y el rendimiento, y
- la elección entre distintas alternativas de diseño.

Arquitectura de software

Una arquitectura no es inherentemente “buena” o “mala”

ventajas del diseño de la arquitectura de software

“tener los atributos de calidad”

Comunicación

Análisis del sistema RNF

Reutilización a gran escala

Atributos de calidad

Disponibilidad

Usabilidad

Performance

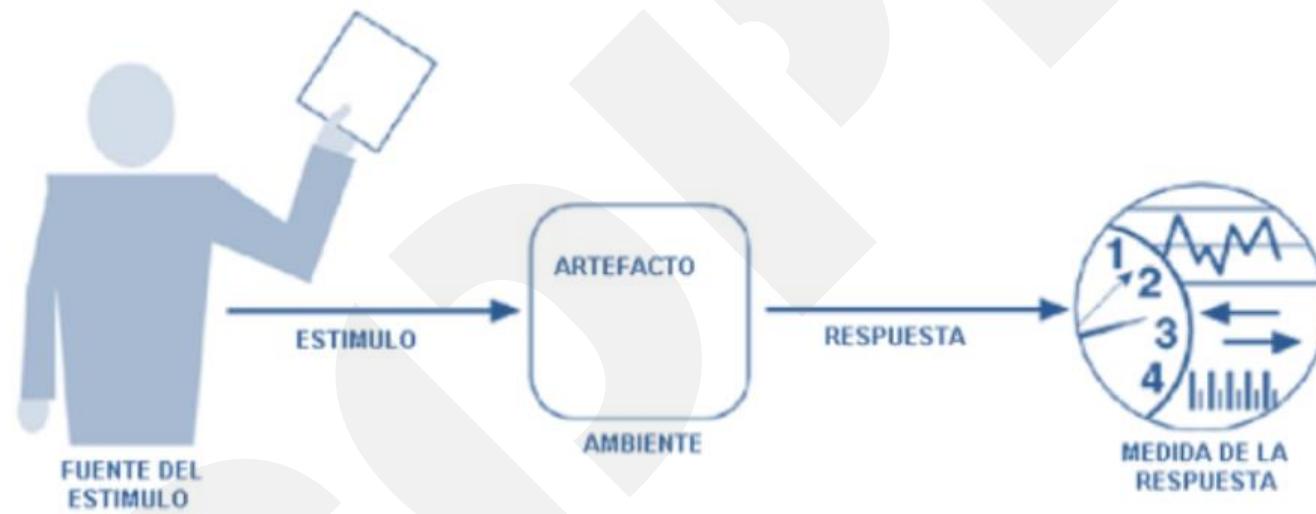
Modificabilidad

Seguridad

Testeabilidad

Escenarios de calidad

Los escenarios de calidad son una forma de representar, de una manera formal, los QARs de un sistema. Un escenario consiste de seis parte.



Escenarios

<u>Elemento</u>	<u>Descripción</u>
Origen del Estímulo.	Cualquier actor que interactúa con el sistema.
Estimulo.	Es una condición que necesita ser considerada cuando arriba al sistema.
Ambiente.	Son las condiciones en la cual se encuentra el sistema en el momento que se recibe el estímulo.
Componentes.	Son los componentes del sistema que son afectados.
Respuesta.	La respuesta es la actividad que debe realizar el sistema.
Medida de la Respuesta.	Es un tipo de medida con al cual debe cumplir la respuesta para que el requerimiento pueda ser testeado.

Ejemplo :Performance

Ej. Los usuarios inician 1.000 transacciones X por minuto (probabilidad) bajo condiciones normales, de 9 a 18:00hs, el sistema debe procesarlas (resultado en pantalla) en una latencia menor a 3 segundos.

<u>Elemento</u>	<u>Descripción</u>
Origen del Estímulo.	Usuarios. Definir el tipo de Usuario si es necesario.
Estímulo.	Inicio de Tx X. Probabilístico. 1.000 tx por minuto
Ambiente.	Procesamiento Normal. De 9 a 18. Carga Normal.
Componentes.	Todo el Sistema
Respuesta.	Transacción procesada
Medida de la Respuesta.	la latencia debe ser menor a 3 segundos

Organización de Sistema

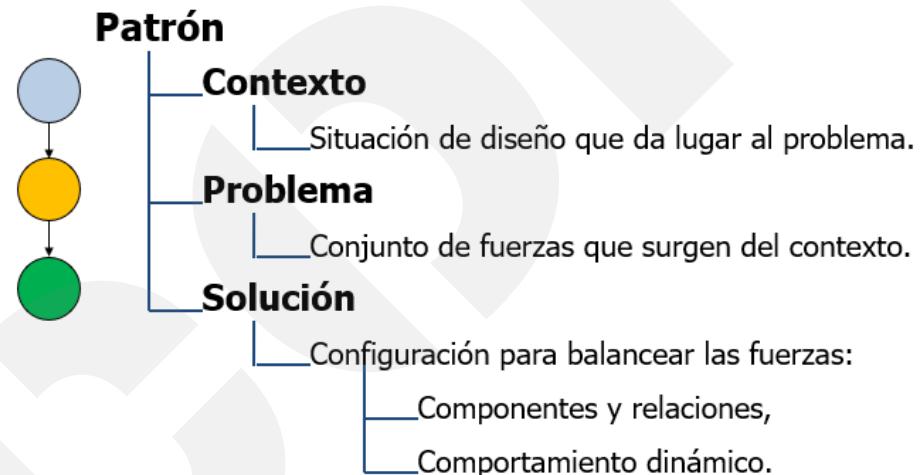
Según Shaw y Garlan una arquitectura de SW incluye 3 cuestiones de diseño:

1. Estructura más adecuada o estilo arquitectónico.
2. Estrategia para descomponer el sistema.
3. Control de la ejecución de los subsistemas.

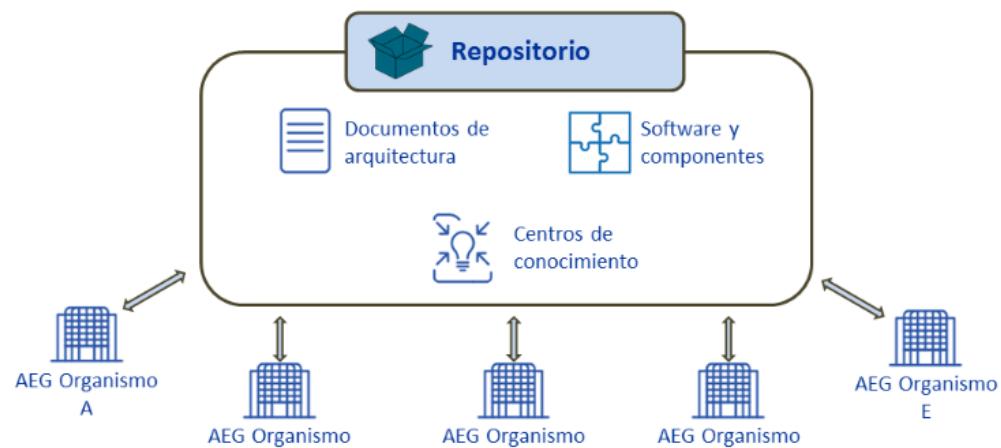
Estilos arquitectónicos

Un estilo arquitectónico define un conjunto de familias de patrones de software con una determinada estructura y restricciones

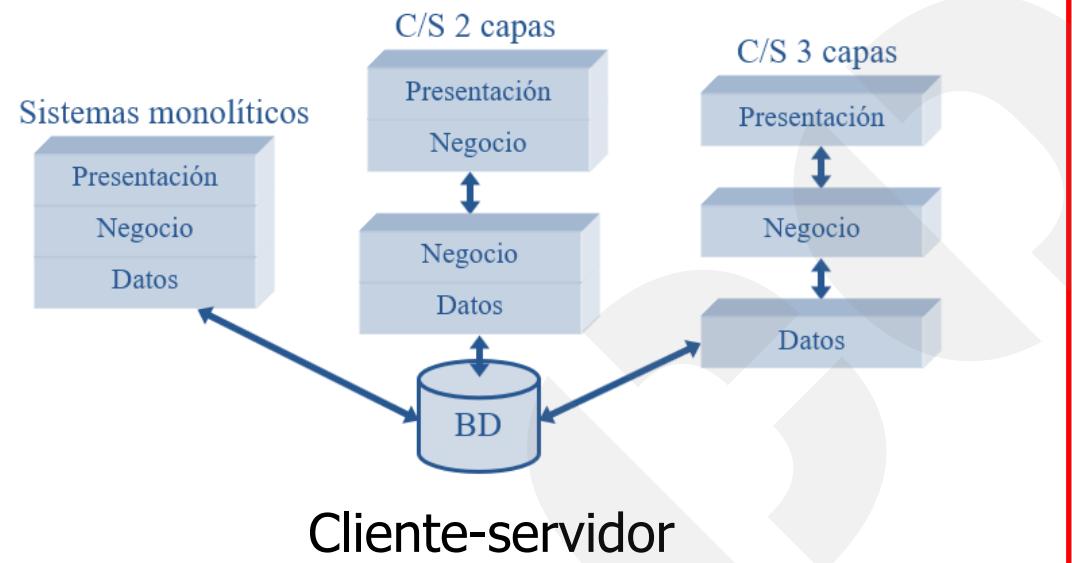
Un patrón de arquitectura de software es: un esquema genérico probado para solucionar un problema particular recurrente que surge en un cierto contexto.



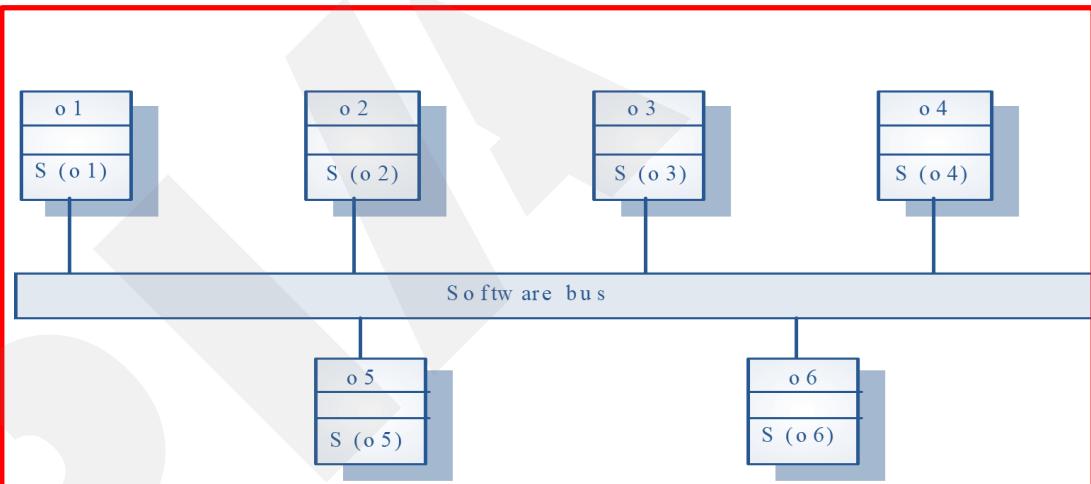
1. Estilos arquitectónicos



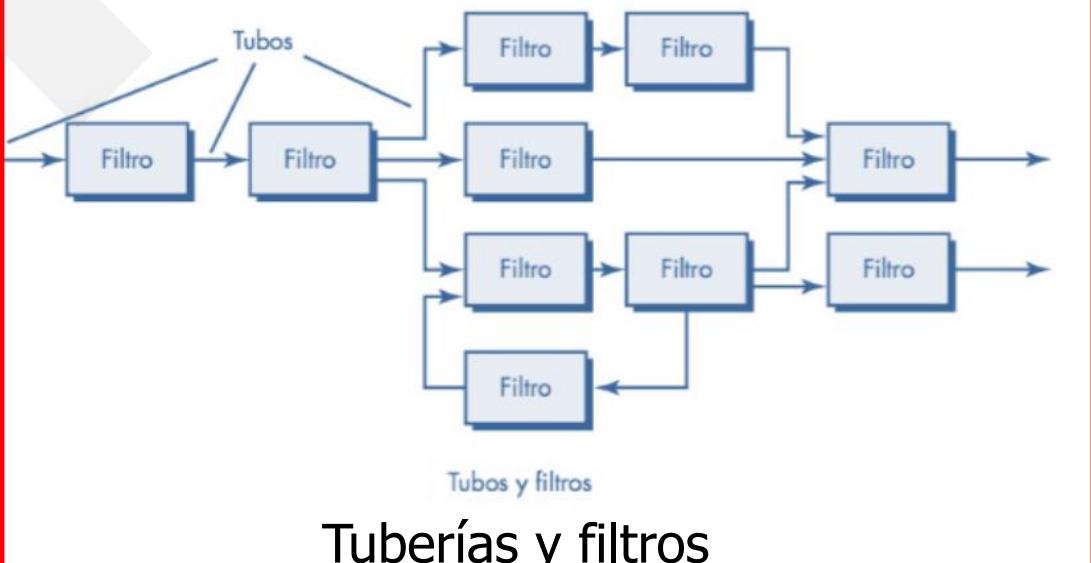
El modelo de repositorio



Cliente-servidor

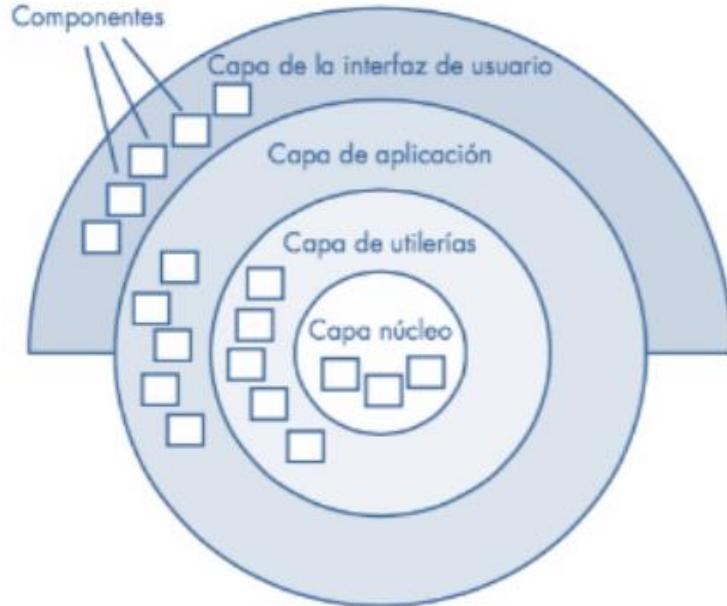


Objetos Distribuidos

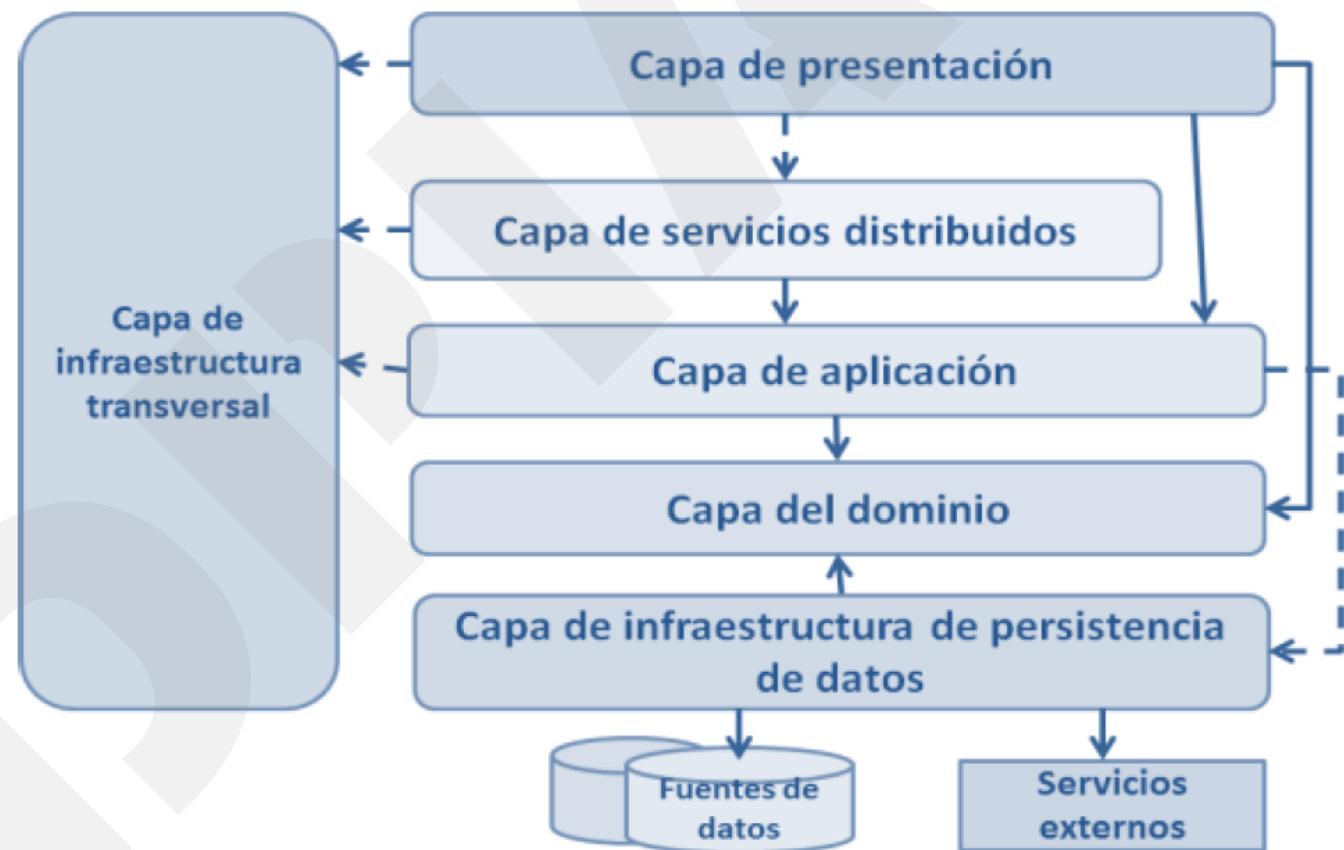


Tuberías y filtros

Estilos arquitectónicos

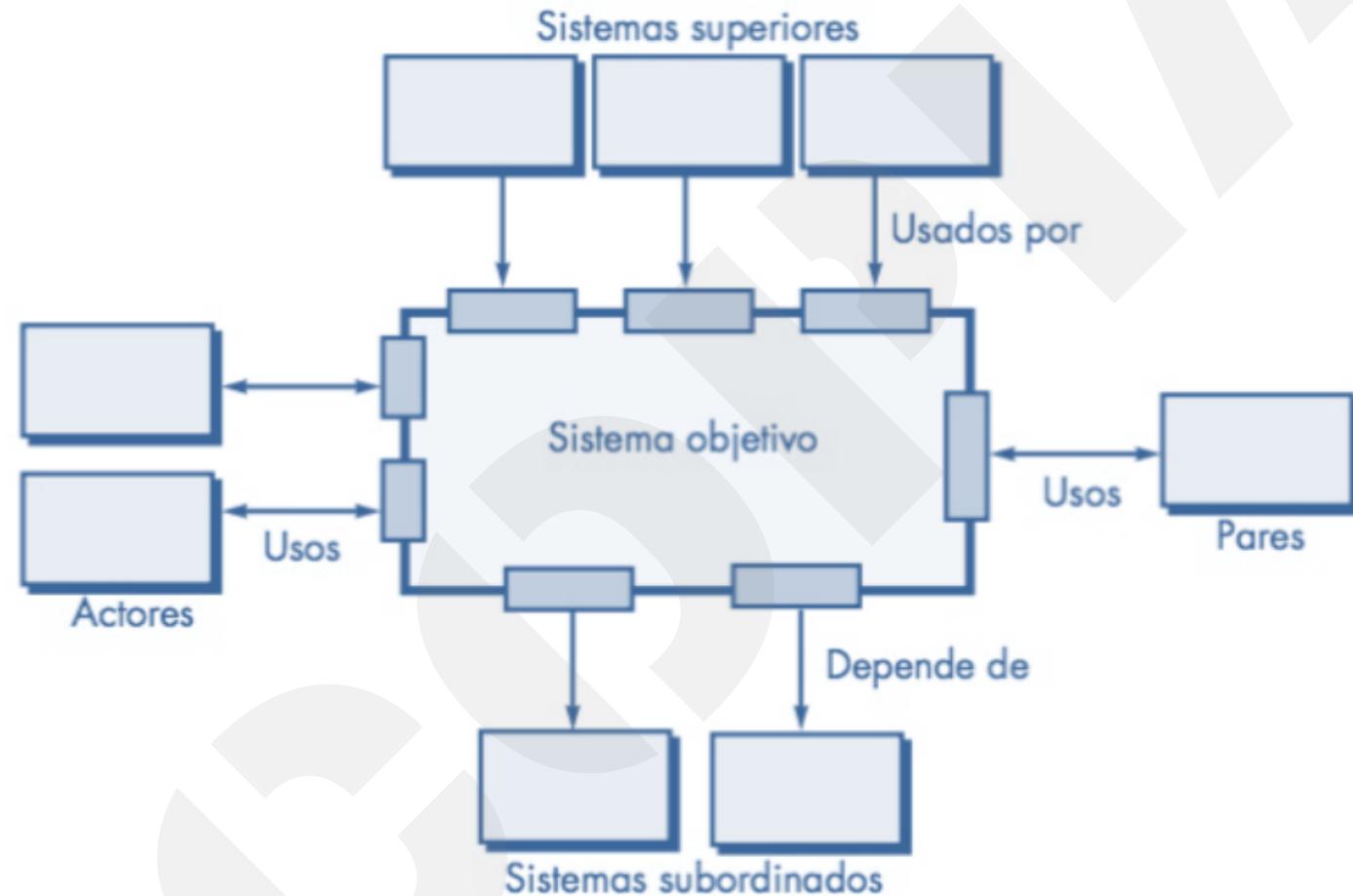


Arquitectura n de Capas

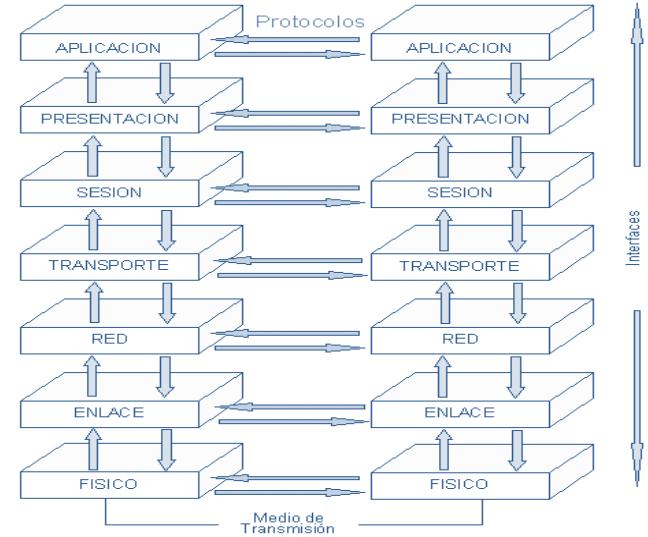
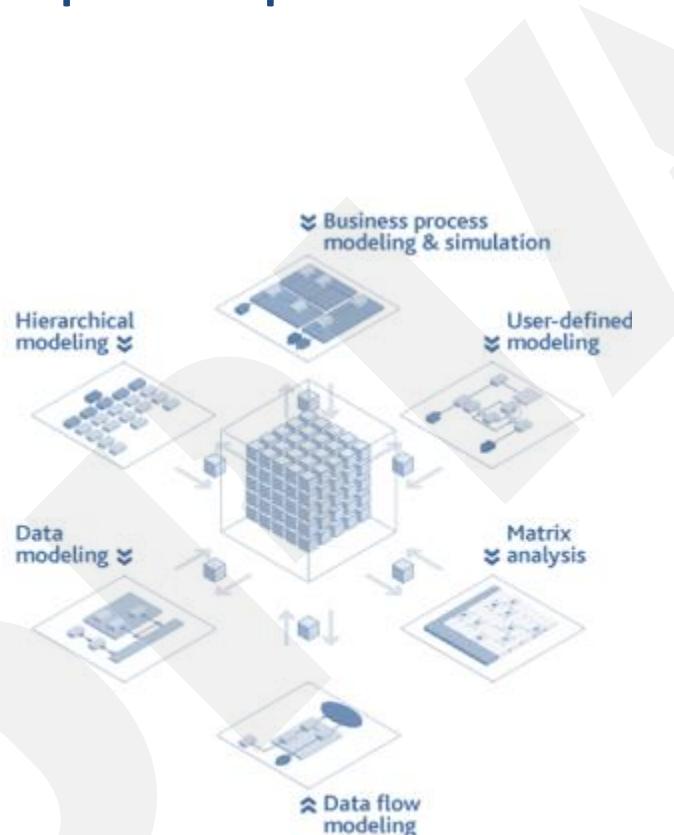
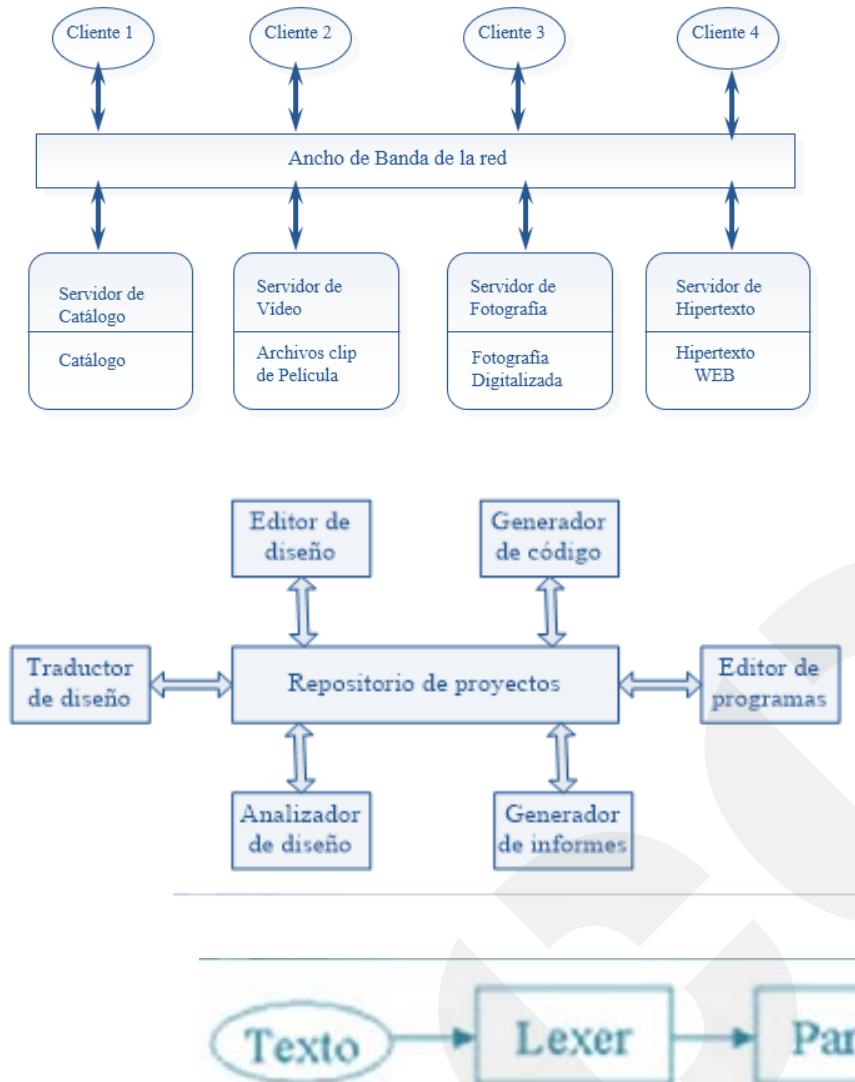


Arquitectura N-Capas
orientada al Dominio

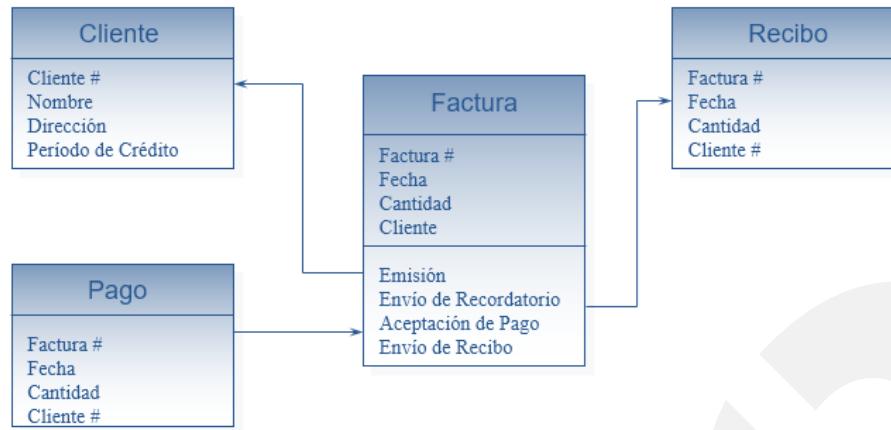
Diagrama de contexto de Arquitectura



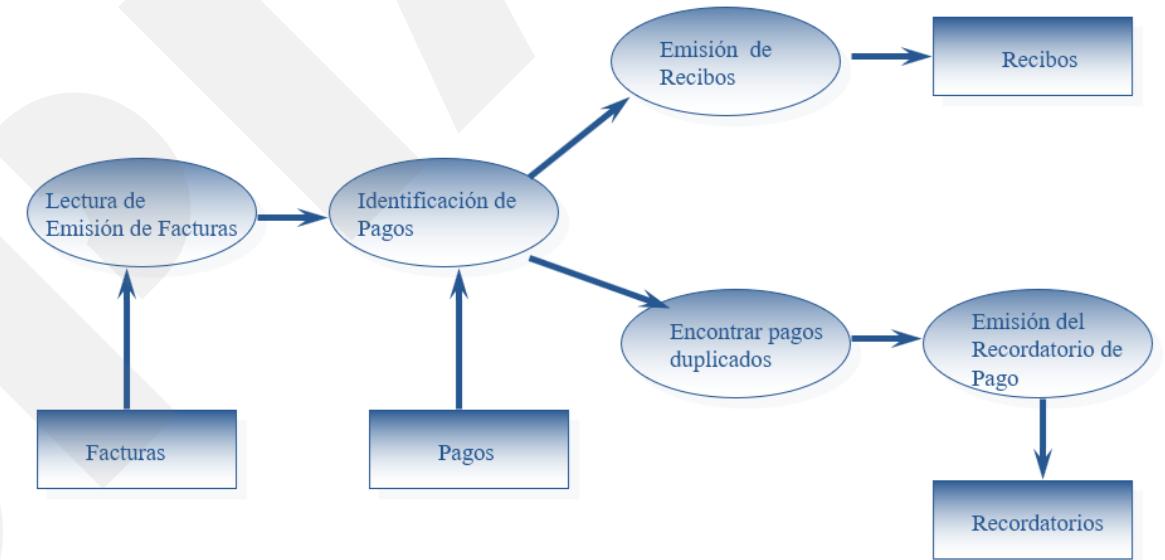
Ejemplos de Aplicación



2. Estilos de descomposición modular



Descomposición OO

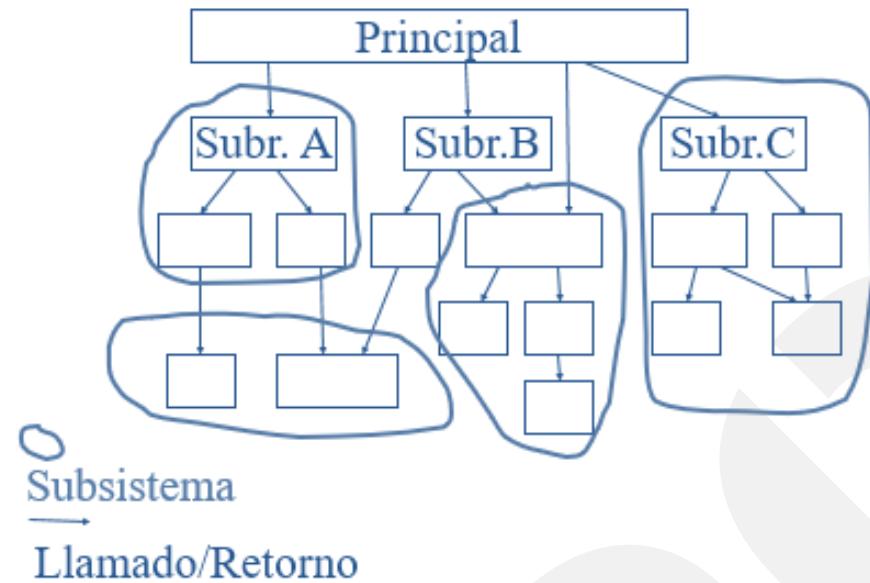


Descomposición funcional

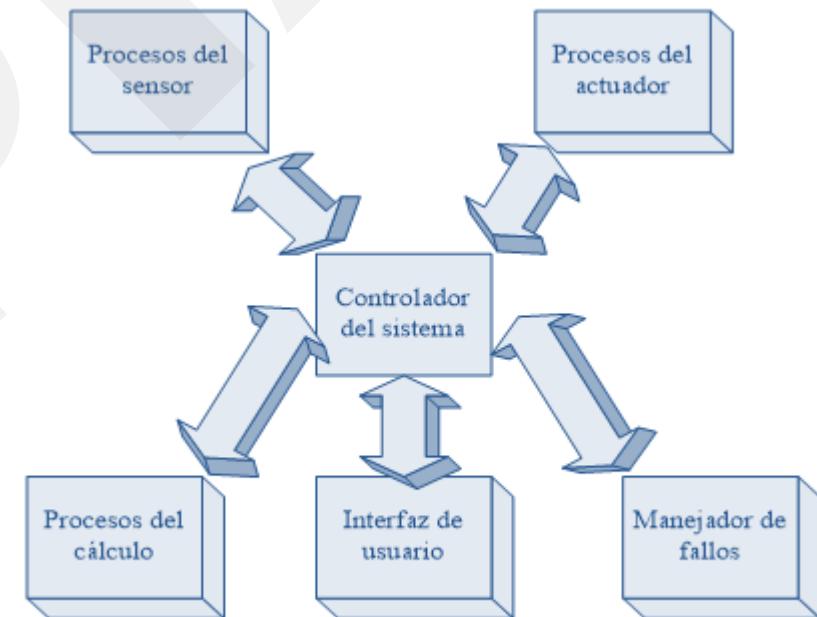
3. Estilos de control

Los sistemas deben ser controlados para que sus servicios se entreguen en el lugar y en el momento correcto.

Control centralizado



Modelos de llamada retorno



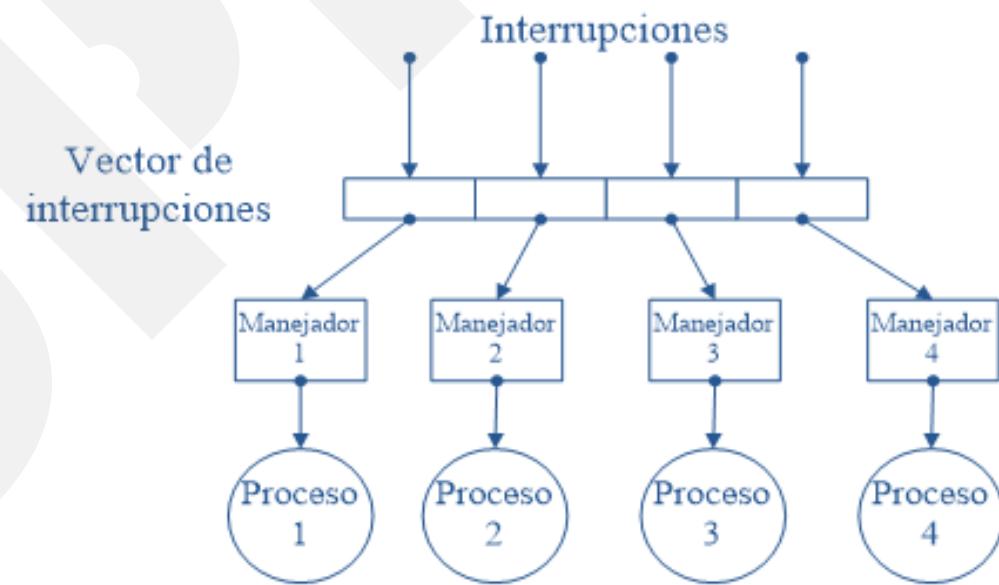
El modelo del gestor:

3. Estilos de control



Modelos de transmisión

Control basado en eventos



Modelos dirigidos por interrupciones



Diagrama de contexto de Arquitectura



Modalidad Asincrónica

<< ACTIVIDAD >>

- **Formación de grupos de Proyectos**
- **Elaboración Project Chárter del proyecto**
- **Calendario de Actividades del proyecto**





UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.



DISEÑO DE SOFTWARE

Modularidad
Acoplamiento y Cohesión

Sesión S3

Introducción

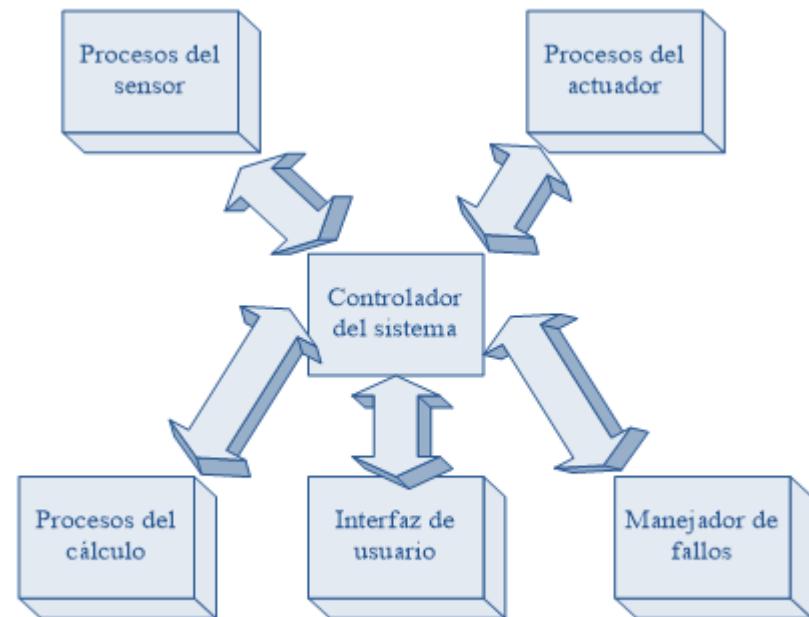
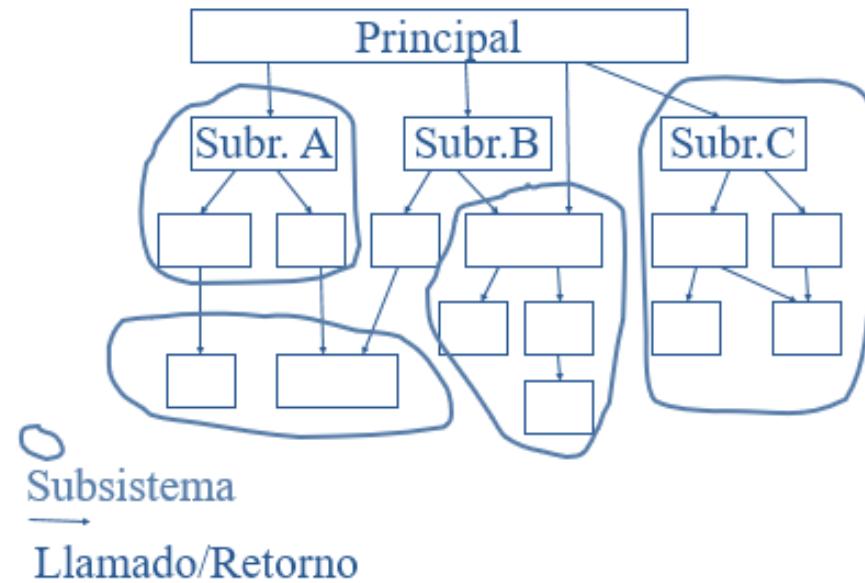
- Control centralizado.
- Control basado en eventos.
- Modularidad
- Acoplamiento y Cohesión



3. Estilos de control

Los sistemas deben ser controlados para que sus servicios se entreguen en el lugar y en el momento correcto.

Control centralizado



Modelos de llamada retorno

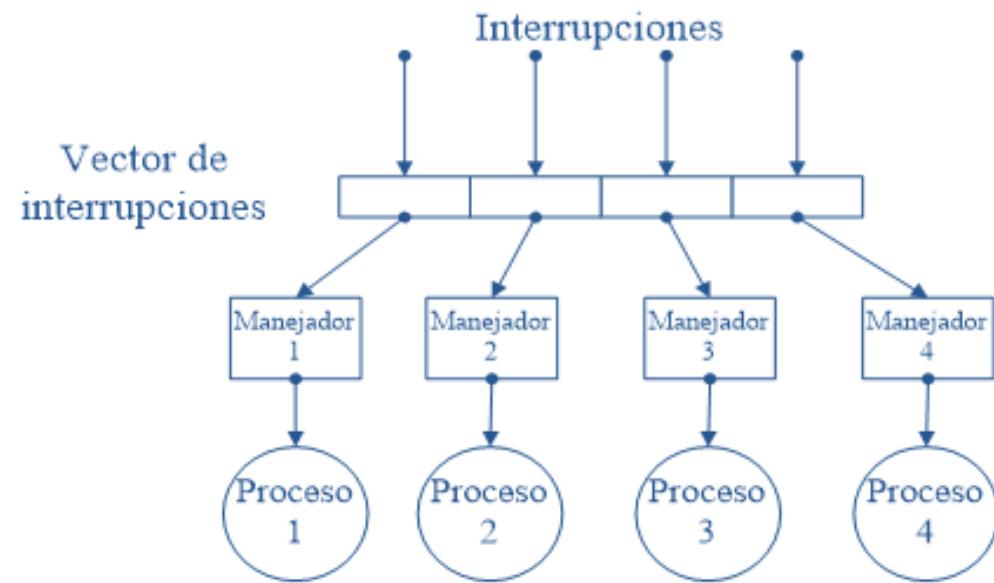
El modelo del gestor:

3. Estilos de control

Control basado en eventos



Modelos de transmisión



Modelos dirigidos por interrupciones

Diseño modular



¿Por qué modularidad?



¿Por qué modularidad?

- Minimiza la complejidad
- Reusabilidad
- Extensibilidad
- Portabilidad
- Mantenibilidad

¿Qué es un módulo?

Visión común: un fragmento de código, unidad de compilación, incluidas las declaraciones relacionadas e interfaz

David Parnas: una unidad de trabajo, Colección de unidades de programación (procedimientos, clases, etc.)

- con una interfaz bien definida y un propósito dentro del sistema entero,
- que puede asignarse independientemente a un desarrollador

¿Por qué modularizar un sistema?

- Gestión
- Evolución
- Entendimiento
- Ocultar Información

¿Qué es una interfaz?

La Interfaz es un contrato publicado por un módulo mediante el cual:

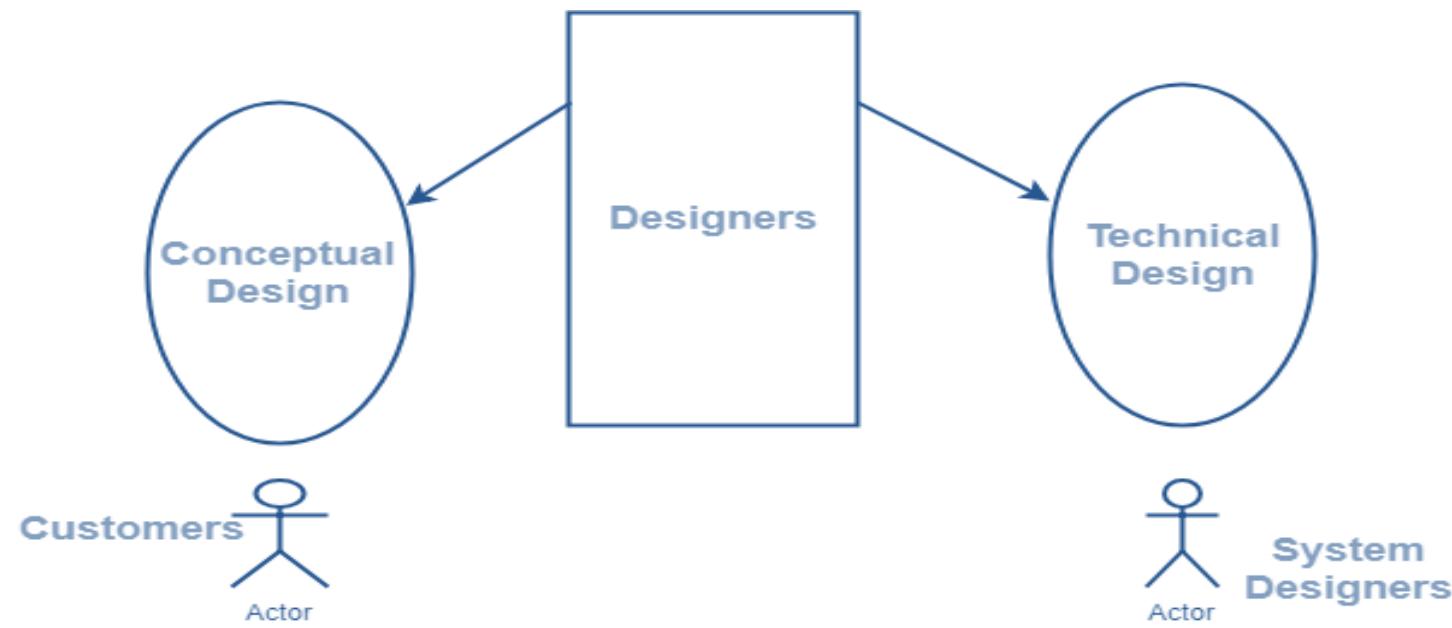
- *Interfaz proporcionada* : a los clientes del módulo pueden depender y
 - *Interfaz requerida* : el módulo puede depender de otros módulos
-
- **Interfaces sintácticas**
 - **Interfaces semánticas**

Principios adicionales

- interfaces explícitas
- Bajo acoplamiento
- interfaces pequeñas
- alta cohesión

El propósito de la fase de Diseño

El propósito de la fase de Diseño en el Ciclo de Vida del Desarrollo de Software es producir una solución a un problema dado en el documento SRS, El resultado de la fase de diseño es Sofware Design Document (SDD).



Diseño conceptual del sistema:

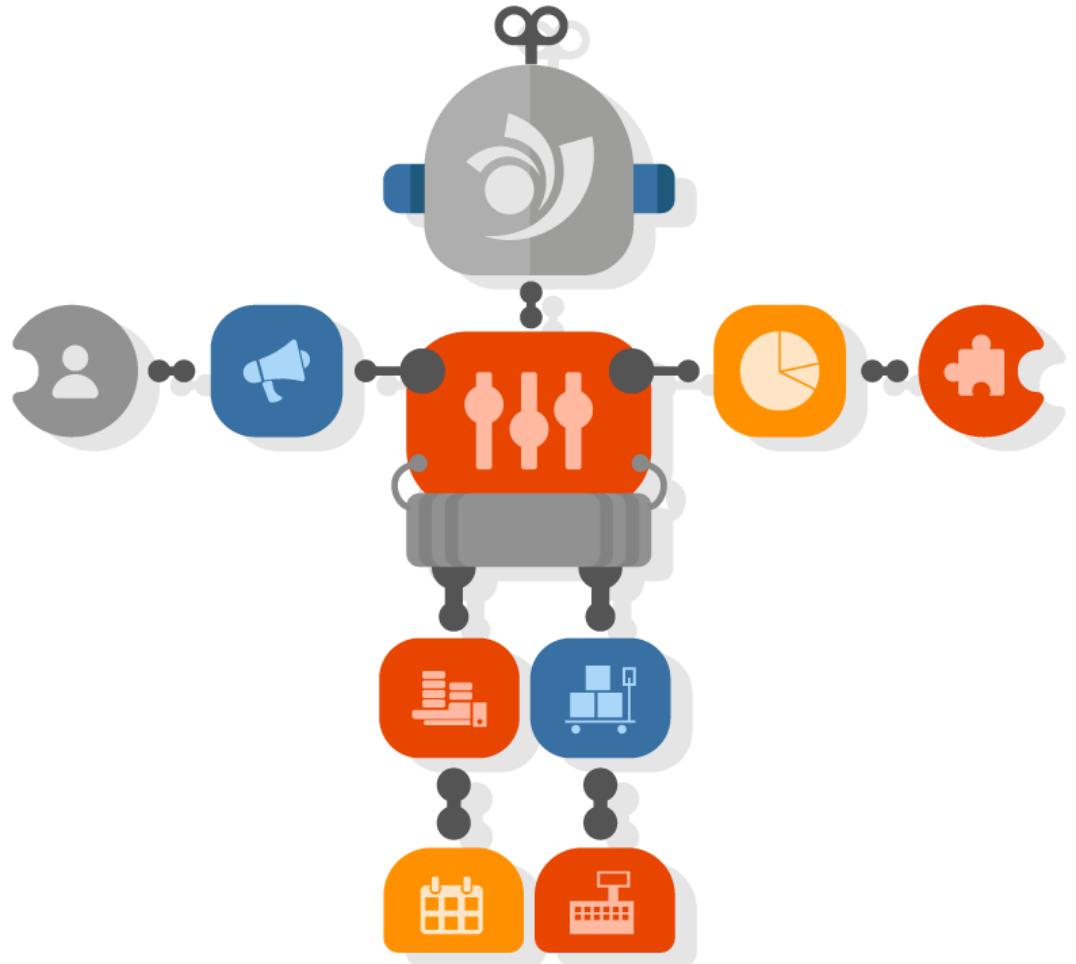
- Escrito en lenguaje simple
- Explicación detallada sobre las características del sistema.
- Describe la funcionalidad del sistema.
- Es independiente de la implementación.
- Vinculado con el documento de requisitos.

Diseño técnico del sistema

- Componente de hardware y diseño.
- Funcionalidad y jerarquía del componente de software.
- Arquitectura de software
- Red de arquitectura
- Estructura de datos y flujo de datos.
- Componente de E / S del sistema.
- Muestra la interfaz.

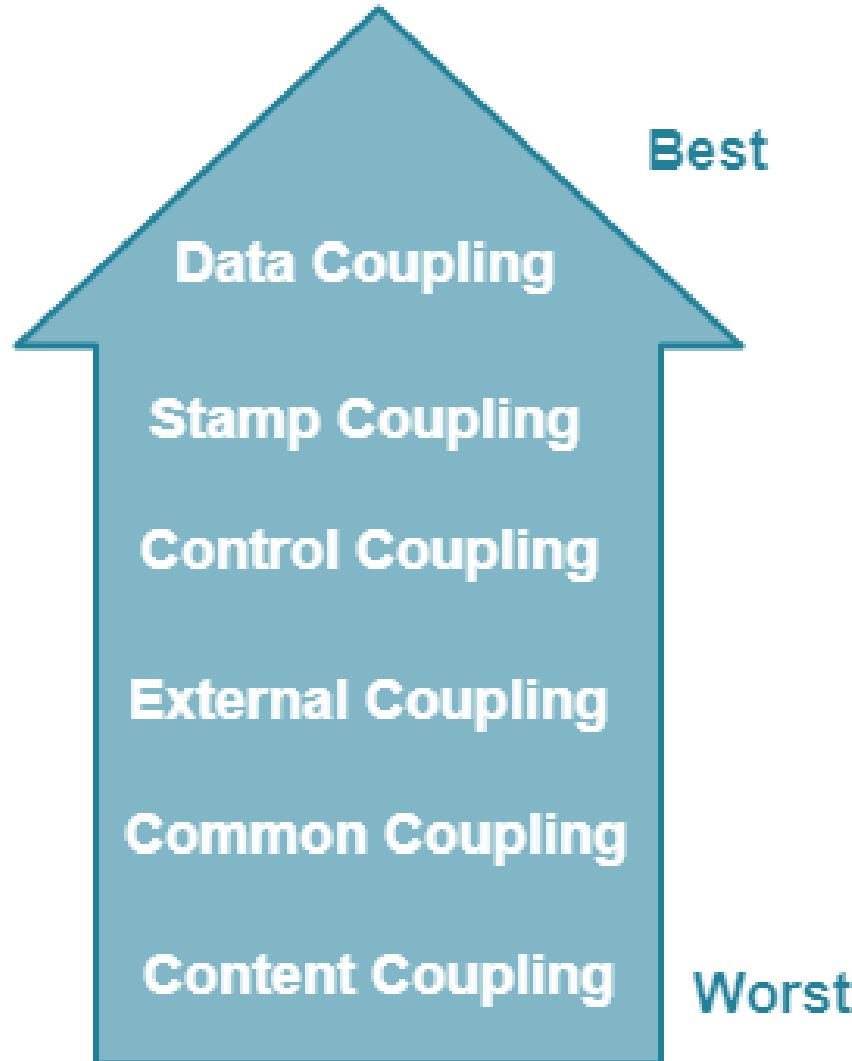
Modularización

La modularización es el proceso de dividir un sistema de software en múltiples módulos independientes donde cada módulo funciona de forma independiente.



Acoplamiento

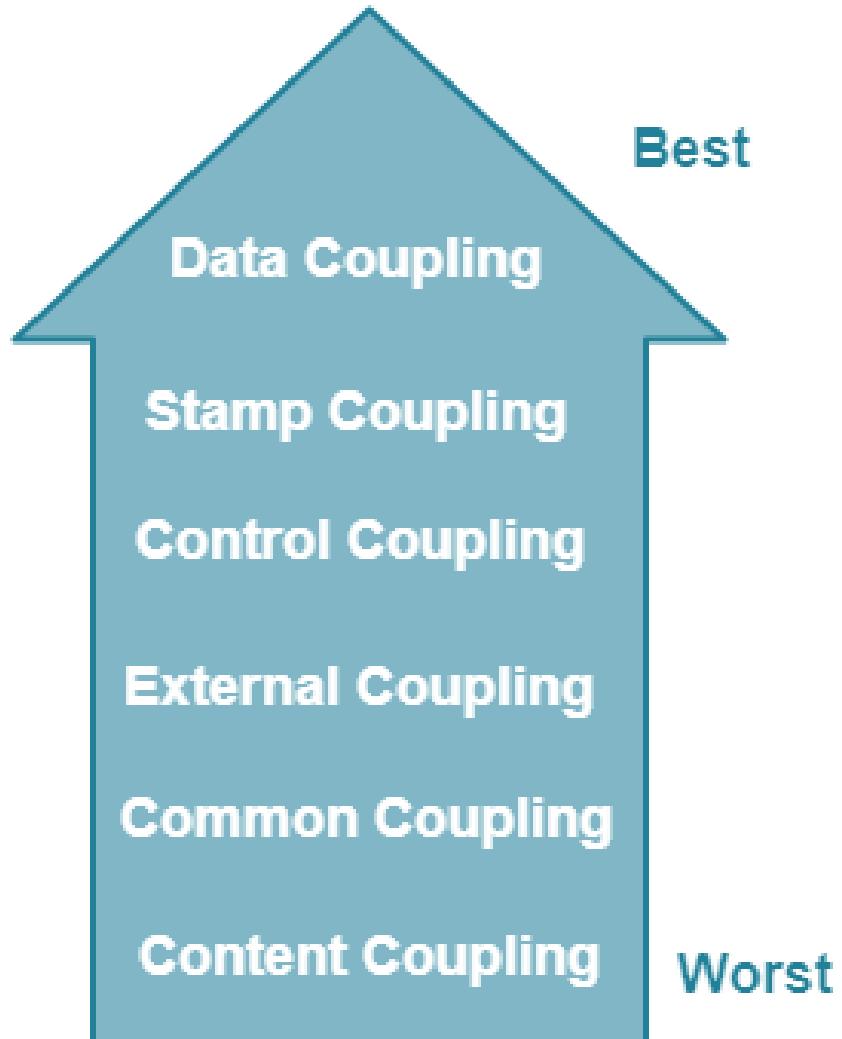
El acoplamiento es la medida del grado de interdependencia entre los módulos. Un buen software tendrá un bajo acoplamiento.

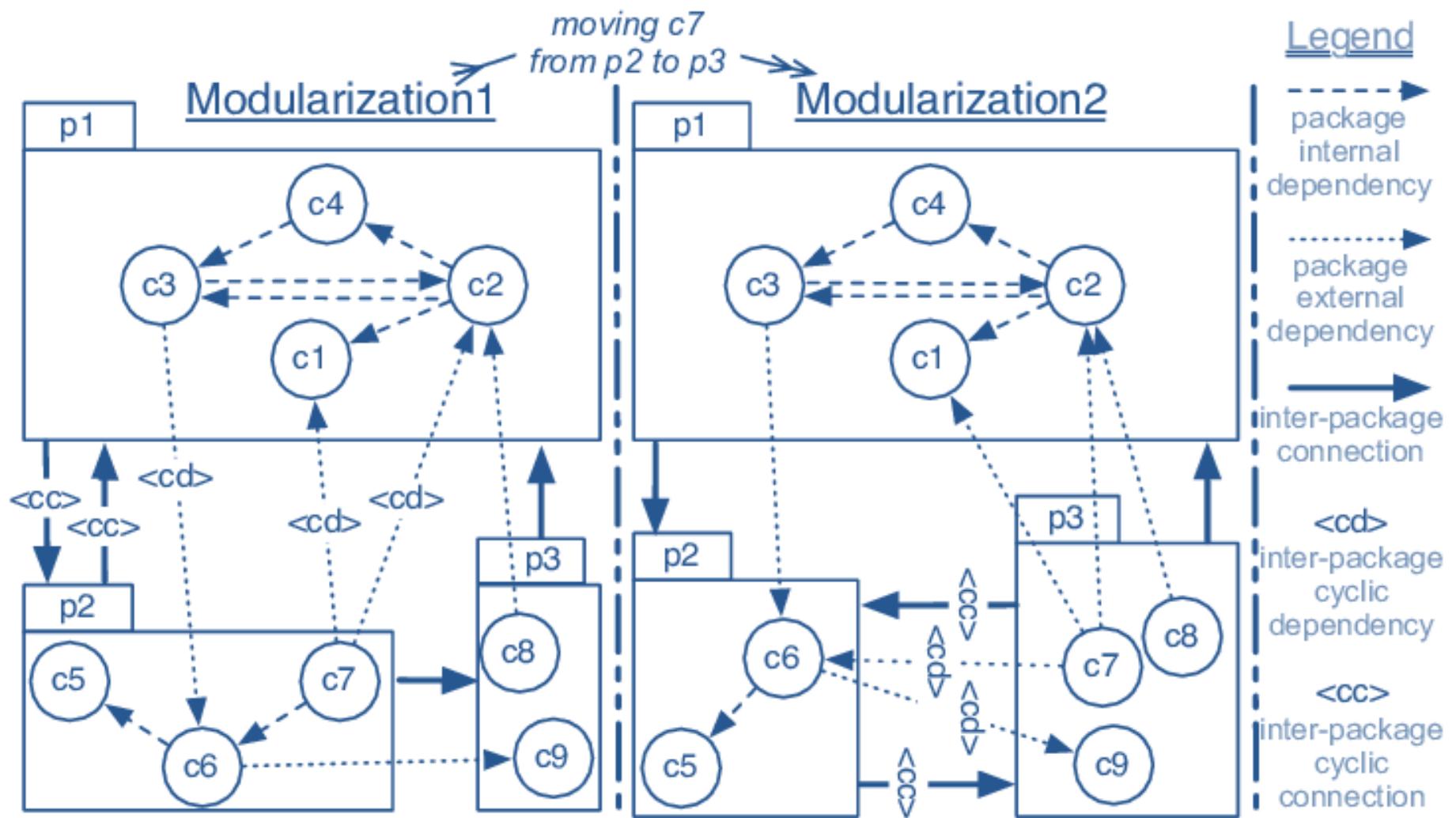


Acoplamiento

La cohesión es una medida del grado en que los elementos del módulo están funcionalmente relacionados.

Es el grado en que todos los elementos dirigidos a realizar una sola tarea están contenidos en el componente.







Modalidad Asincrónica

<< ACTIVIDAD >>

- Diseño de Prototipos



Mockup

Prototipado

Para poder realizar evaluaciones de la usabilidad en las etapas iniciales necesitamos **prototipos**.

Son documentos, diseños o sistemas que simulan o tienen implementadas partes del sistema final



Prototipado

Un prototipo es una **implementación parcial** pero concreta del diseño de un sistema.

Los prototipos pueden ser **creados para explorar** muchas cuestiones acerca del sistema durante el desarrollo

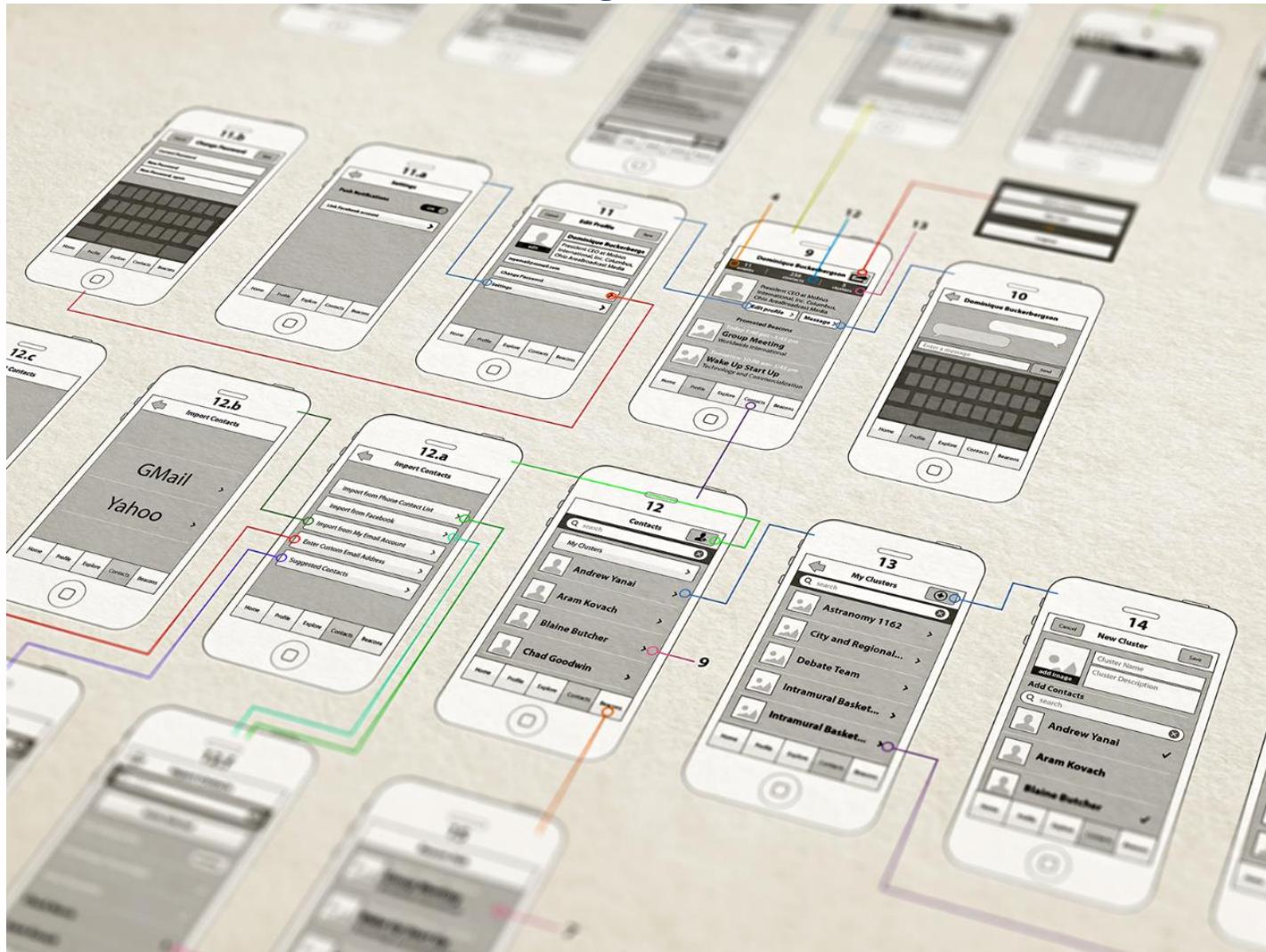
Un **prototipo de una interfaz de usuario** es un prototipo realizado con la finalidad de **explorar los aspectos interactivos del sistema**, incluyendo la usabilidad, la accesibilidad y/o funcionalidad del mismo



Prototipado



Prototipado



Mockup

Un prototipo o *mockup* en inglés es una **maqueta** o **modelo** de un diseño o dispositivo para que **tener una idea** de cómo será el producto final

Maquetas

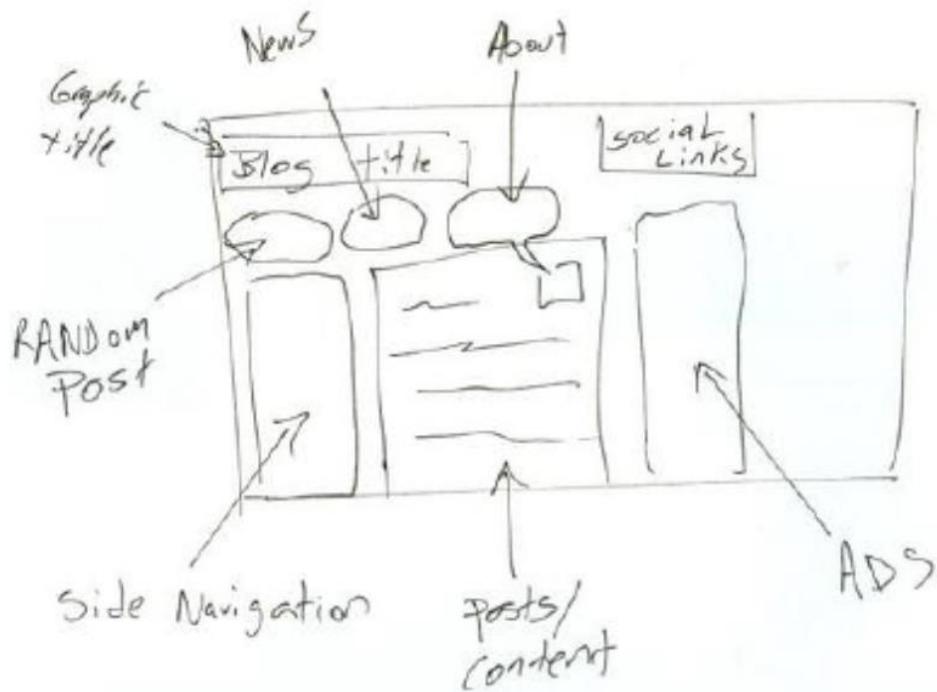
Prototipo de baja fidelidad y de alta fidelidad Sketching, wireframes, storyboard, prototipos funcionales

The image shows two versions of a website layout. On the left is a hand-drawn sketch labeled "Boceto" (Sketch) in blue ink. It features a header with a logo and menu, a main content area with sections for "Proyecto" and "Relaciones", and a footer labeled "PIE". On the right is a high-fidelity wireframe labeled "Maqueta" (Mockup) in blue ink. This version includes a header with a logo and navigation links (Inicio, Promociones, Oportunidades, Asociados, Servicio, Contacto), a main content area with a title, date, and text, and a sidebar with a search bar and a "Noticias recientes" section.

Sketching

Son prototipos de baja fidelidad

Los voceros son maneras de representar la **primeras ideas**.



Sketching

Sketch



Sitio real

 **usable** accesible *des de cero*

Blog Sobre mí Consultoría Recursos

Martes, 12 de febrero de 2013

Actitudes y aptitudes de un arquitecto de información

A lo largo del artículo [Arquitectura de información: Planteamiento](#) me preguntaron cuáles son los conocimientos o actitudes que califican a un arquitecto de información.

Este ensayo intenta dar una reflexión personal sobre cuáles son esas **actitudes y aptitudes que apoyan con su apoyo en las personas** con las que se da trabajo a lo largo de los años, y que son inherentes en la calidad de cualquier profesional.

Experiencia

Es innegable que el talento y complejidad de los proyectos en los que haya participado anteriormente te permitirán acompañar con más garantías de éxito la arquitectura de información de nuevo diseño.

Capacidad de autocritica

Sin embargo, la experiencia, al irse acompañada de capacidad de autocritica, trae consigo la calidad del resultado, que posiblemente se centrará en el siguiente portal los mismos errores que en los anteriores.

Por consiguiente de que no se case raro, cuenta desde la humildad, aceptar y analizar las críticas que se lesotén, aprender de los errores y de los demás, son aspectos importantes. Debe aprender a trabajar con personas con capacidad de autocritica, que saben decir "esto no lo sé" o "estoy equivocado".

Capacidad de autoaprendizaje

Por algo dicen que un experto es el que no conoce todos los errores que posee conocer y ha sucedido en cada fracaso aquello que debía aprender. Cuanto más se pasa por alto, peor la resulta es que la teoría es analítica y la práctica no, y hay ciertas déns que solo se adquieren después de haberse repasado.



Olga Camerón Montoto
Consultora de accesibilidad web: PDF, accesibilidad, experiencia de usuario (UX) y ergonomía de interacción (ADI)

Contacto: olgam@polipost.com

Historico de artículos: [Portafolio](#) | [Páginas](#) | [Glosario](#)



Lo más leído sobre Accesibilidad:

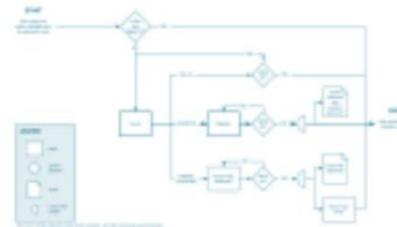
1. [CSSAccesible](#)
2. [Accesibilidad web: PDF, accesibilidad...](#)
3. [SEO: Accesibilidad web: PDF, accesibili...](#)
4. [HTML: Accesibilidad](#)
5. [Reflexión sobre la accesibilidad web:...](#)
6. [Metodologías, certificaciones y estándares:...](#)
7. [Resposta a 100000 consultas sobre accesibilidad web](#)
8. [PDFAccesible](#)
9. [Importancia de usar para realizar una mejor forma de accesibilidad de acuerdo a los WCAG 2.0](#)
10. [Correspondencia entre Norma UNE 106620 y WCAG 2.0](#)
11. [WCAG 2.0](#)
12. [Un recurso para pasarse a las WCAG 2.0](#)
13. [Diferencias de prioridad entre los requisitos de la Norma UNE 106620 y los puntos de control de las WCAG 2.0](#)
14. [Cuestiones avanzadas de accesibilidad web](#)
15. [Consultores accesibles: regla las WCAG 2.0](#)
16. [AVAccesible](#)

The screenshot displays a website template for a news section. At the top, there is a header with a logo placeholder labeled "LOGOTIPO". Below the logo is a horizontal navigation bar with links: Inicio, Promociones, Oportunidades, Actualidad (which is highlighted), Servicios, and Contacto. The main content area has a title "Actualidad". Underneath, there is a breadcrumb trail "Inicio / Actualidad" and an RSS feed icon. The main article is titled "Título de la noticia" and is dated "27 febrero de 2012". It contains a placeholder image labeled "Imagen" and three columns of text. Below the article is a link "Seguir leyendo >>". A sidebar on the right contains a search form with "Buscar noticias" and "Buscar" button, and a link "Búsqueda avanzada". There is also a "Zona versátil" box with the text "(podcast, diarios, twitters, destacado)". At the bottom, there is a newsletter sign-up form with fields for "Tu email" and "Enviar".

Niveles

Planos

Blueprint, diagramas de contenido o flujo, o mapa web



Maquetas

Baja fidelidad

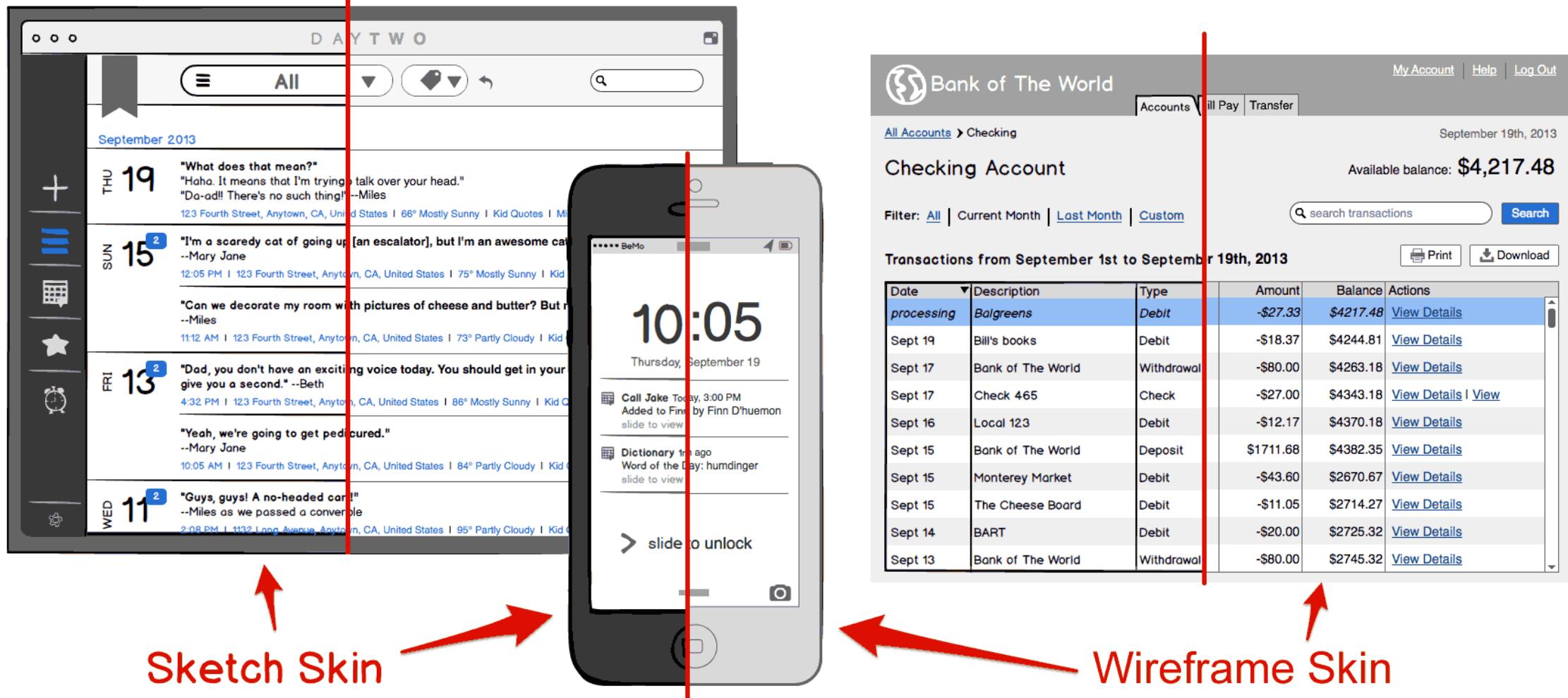
Sketch

Wireframes



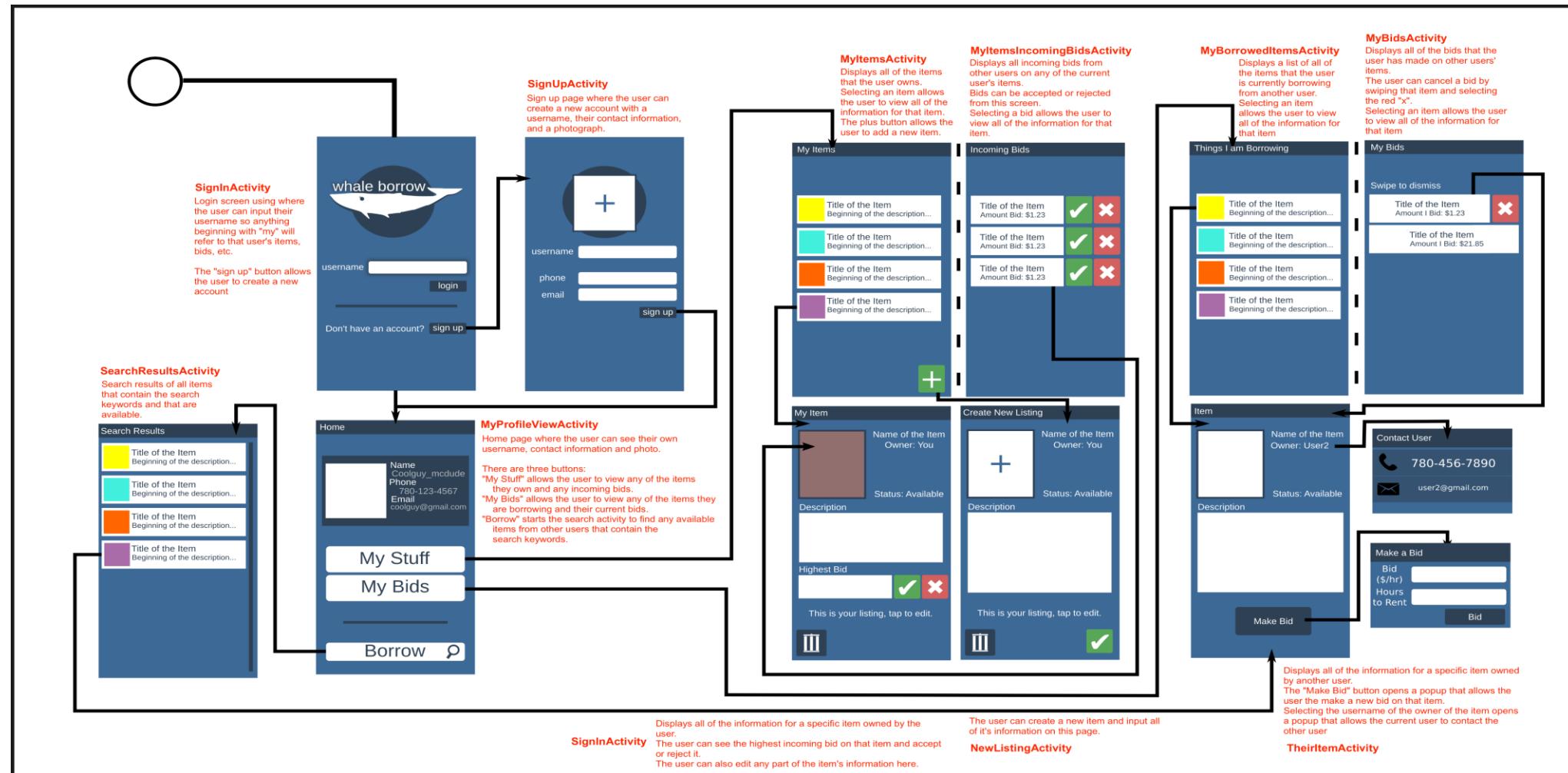
Alta fidelidad → Prototipo funcional

Ejemplo



Storyboard

Secuencia de wireframes



Herramientas mockup

Pencil (free)

Balsamiq (paga)

Mockingbird (free online)

Moqups (pago/free online) *

MockFlow (pago/free online) *

ninjamock.com (pago/free online) *

Mockup Builder pago/free online)





UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.



DISEÑO DE SOFTWARE

MOCKUP

Sesión S3

Prototipado

Para poder realizar evaluaciones de la usabilidad en las etapas iniciales necesitamos **prototipos**.

Son documentos, diseños o sistemas que simulan o tienen implementadas partes del sistema final



Prototipado

Un prototipo es una **implementación parcial** pero concreta del diseño de un sistema.

Los prototipos pueden ser **creados para explorar** muchas cuestiones acerca del sistema durante el desarrollo

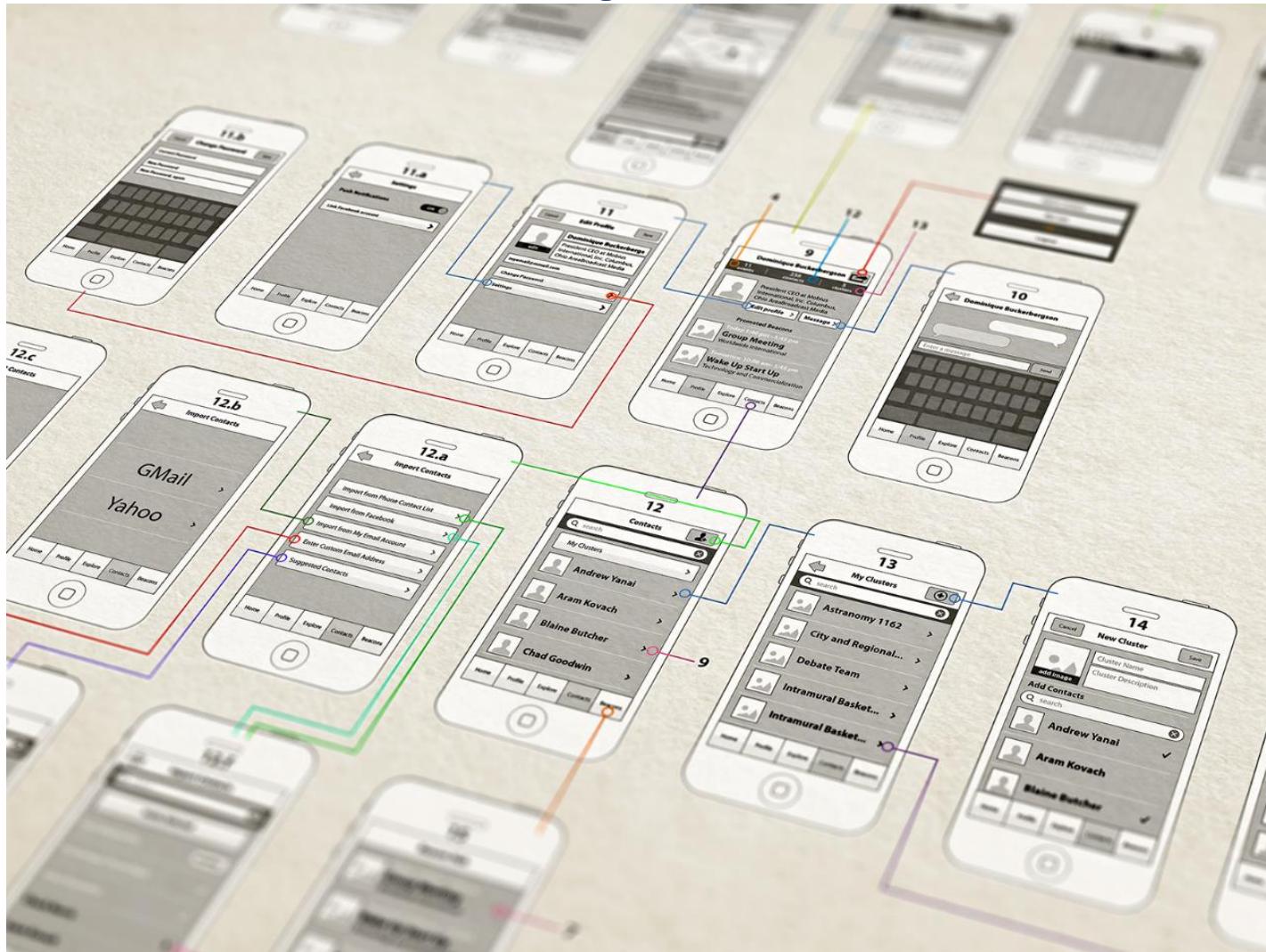
Un **prototipo de una interfaz de usuario** es un prototipo realizado con la finalidad de **explorar los aspectos interactivos del sistema**, incluyendo la usabilidad, la accesibilidad y/o funcionalidad del mismo



Prototipado



Prototipado



Mockup

Un prototipo o *mockup* en inglés es una **maqueta** o **modelo** de un diseño o dispositivo para que **tener una idea** de cómo será el producto final

Maquetas

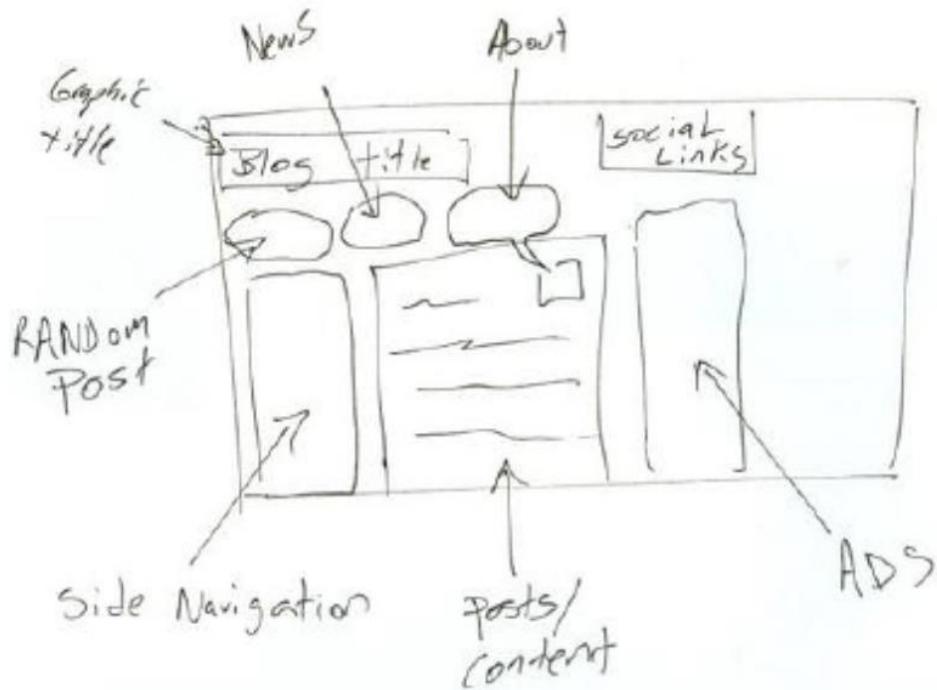
Prototipo de baja fidelidad y de alta fidelidad Sketching, wireframes, storyboard, prototipos funcionales

The image shows two versions of a website layout. On the left is a hand-drawn sketch labeled "Boceto" (Sketch) in the top right corner. It features a header with a logo and menu, a main content area with sections for "Proyecto" and "Relaciones", and a footer labeled "PIE". On the right is a high-fidelity wireframe labeled "Maqueta" (Mockup) in the top right corner. This version includes a header with a logo and navigation links (Inicio, Promociones, Oportunidades, Asociados, Servicio, Contacto), a main content area with a "Actualidad" section, and a sidebar with a search bar and a "Noticias" section. Both versions include placeholder text and icons.

Sketching

Son prototipos de baja fidelidad

Los voceros son maneras de representar la **primeras ideas**.



Sketching

Sketch



Sitio real

 **usable** accesible *des de cero*

Blog Sobre mí Consultoría Recursos

Martes, 12 de febrero de 2013

Actitudes y aptitudes de un arquitecto de información

A lo largo del artículo [Arquitectura de información: Planteamiento](#) me preguntaron cuáles son los conocimientos o actitudes que califican a un arquitecto de información.

Este entorno indica que lo que califica a un arquitecto de información no se limita a unos conocimientos teóricos y metodológicos, aunque evidentemente son importantes, sino que tiene mucho ver con una serie de actitudes y aptitudes.

En este artículo hago una reflexión personal sobre cuáles son esas **actitudes y aptitudes que apoyan con su apoyo en las personas** con las que se da trabajo a lo largo de los años, y que son características de realidad a cualquier profesional:

Experiencia

Es innegable que el número y complejidad de los proyectos en los que haya participado anteriormente te permitirán acompañar con más garantías de éxito la arquitectura de información de nuevo diseño.

Capacidad de autocritica

Sin embargo, la experiencia, al no ir acompañada de capacidad de autocritica, temporalmente asegura la calidad del resultado, pero posteriormente se centrará en el siguiente portal los mismos errores que en los anteriores.

Por consiguiente de que no se casi todo, contar desde la humildad, aceptar y analizar las críticas que se reciben, aprender de los errores y de los demás, son aspectos importantes. Debe aprender a trabajar con persona con capacidad de autocritica, para saber decir "esto no lo sé" o "estoy equivocado".

Por algo dicen que un experto es el que no conoce todos los errores que posee conocer y ha sucedido en su vida ni hace aquello que desea aprender. Cuanto más se posa exagerado, peor la resulta es que la teoría es analítica y la práctica no, y hay ciertas ideas que solo se adquieren después de haberse repitiendo.

Capacidad de autoaprendizaje

Usable y accesible en Facebook

Olga Camerón Montoto Consultora de accesibilidad web: PDF, accesibilidad, experiencia de usuario (UX) y ergonomía de interacción (ADI)

Contacto: olgam@usabilypdf.com

Histórico de artículos

Parte uno | Parte dos | Glosario

Lo más leído sobre Accesibilidad:

1. [CSSAccesible](#)
2. [Accesibilidad web: PDF, accesibilidad...](#)
3. [SEO: Accesibilidad web: PDF, accesibili...](#)
4. [HTML: Accesibilidad](#)
5. [Reflexión sobre la accesibilidad web: con la accesibilidad web](#)
6. [Metodologías, certificaciones y estándares: retroalimentación de la accesibilidad web en España](#)
7. [Resposta a 1000 dudas básicas sobre accesibilidad web](#)
8. [PDF: accesibilidad](#)
9. [Materia prima de prueba para realizar una memoria de accesibilidad de acuerdo a las WCAG 2.0](#)
10. [Correspondencia entre Norma UNE 106620 y WCAG 2.0](#)
11. [WCAG 2.0](#)
12. [Un recurso para pasarse a las WCAG 2.0](#)
13. [Diferencias de prioridad entre los requisitos de la Norma UNE 106620 y los puntos de control de las WCAG 2.0](#)
14. [Cuestiones avanzadas de accesibilidad web](#)
15. [Formularios accesibles según las WCAG 2.0](#)
16. [AVAccesible](#)

Wireframes

The wireframe illustrates a news website's layout. At the top is a header bar with a logo placeholder, navigation links for Inicio, Promociones, Oportunidades, Actualidad (which is highlighted), Servicios, and Contacto, and social media icons for RSS and Facebook.

The main content area features a large title "Actualidad" and a breadcrumb trail "Inicio / Actualidad". To the right of the title is a search bar labeled "Buscar noticias" with a "Buscar" button and a link to "Búsqueda avanzada".

The central column contains a news article with a title "Título de la noticia", a date "27 febrero de 2012", and a snippet of text: "Entradilla. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris.". Below this is a placeholder for an image labeled "Imagen". The main text of the article is divided into two sections: "Cuerpo de la noticia. Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris." and "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris.". A third section at the bottom contains the text "Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris." followed by a "Seguir leyendo >>" link and three tags: "etiqueta 1", "etiqueta 2", and "etiqueta 3".

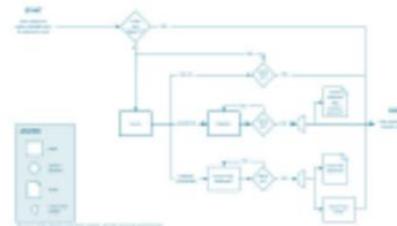
At the bottom of the article are sharing options: "Compartelo:" followed by icons for Google+, Twitter, Facebook, LinkedIn, and Email, and a link "Enviar a un amigo".

A footer section at the bottom allows users to "Recibe semanalmente todas nuestras novedades y oportunidades" via email, with an "Enviar" button.

Niveles

Planos

Blueprint, diagramas de contenido o flujo, o mapa web

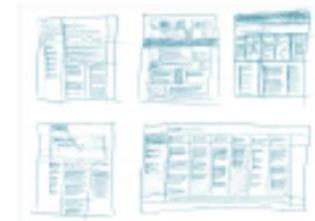


Maquetas

Baja fidelidad

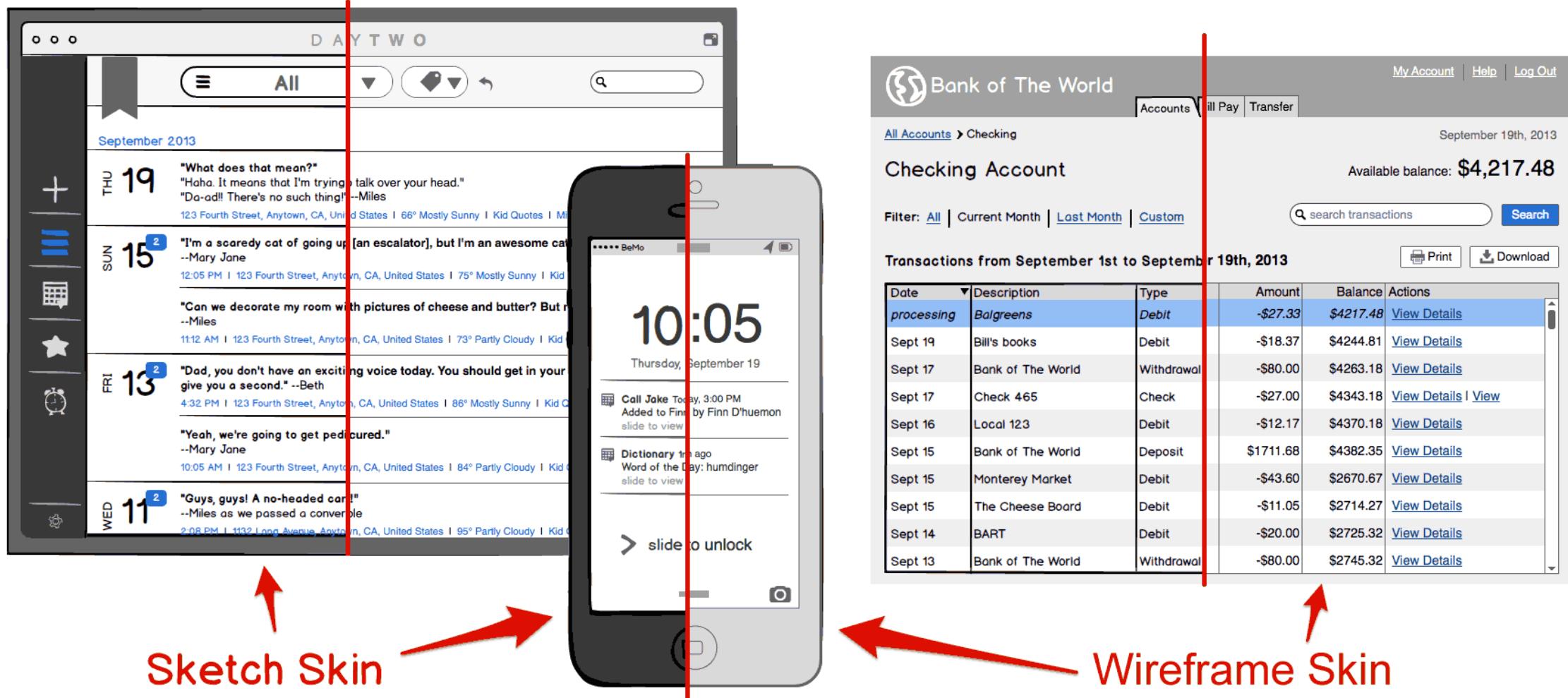
Sketch

Wireframes



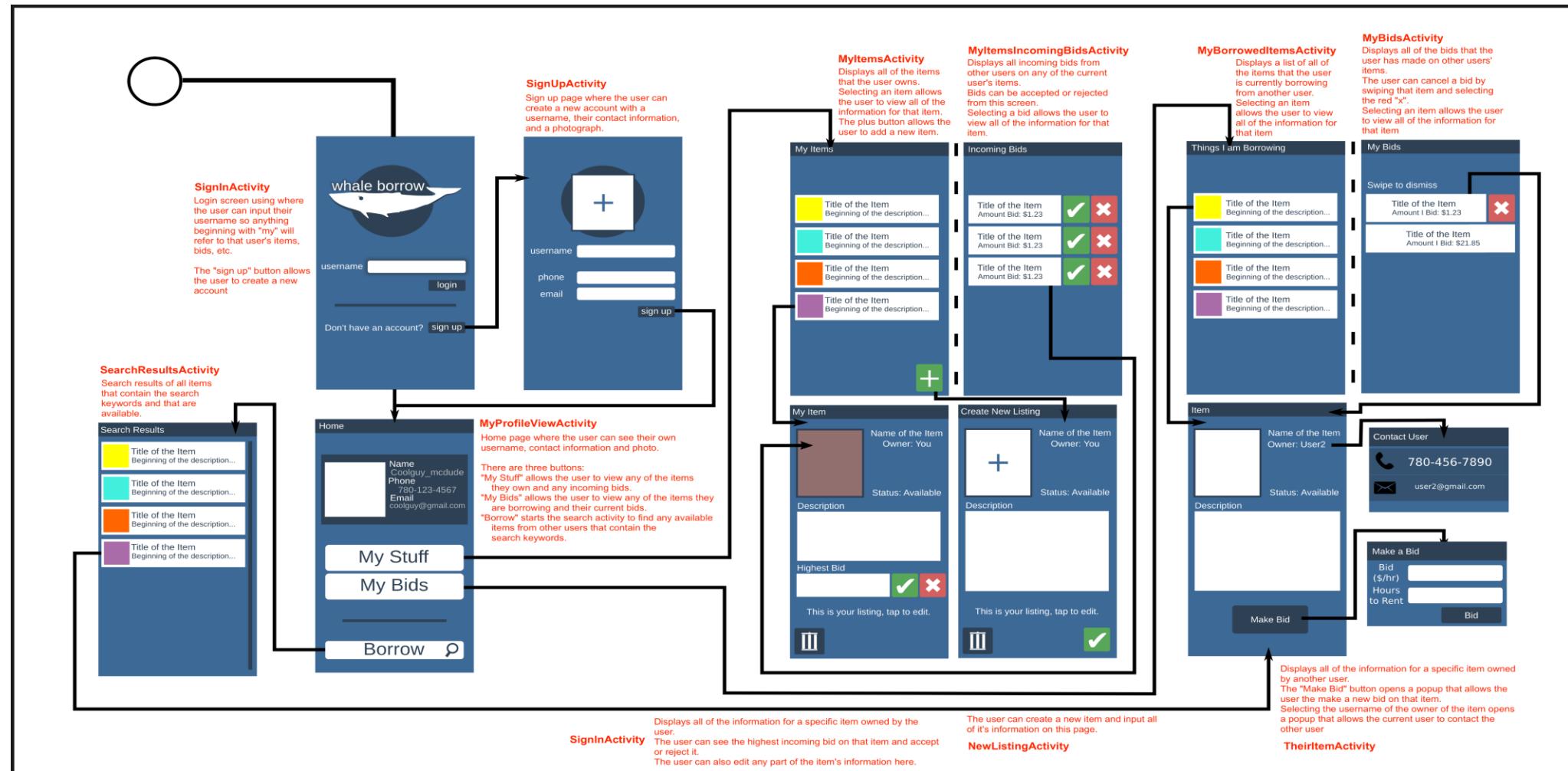
Alta fidelidad → Prototipo funcional

Ejemplo



Storyboard

Secuencia de wireframes



Herramientas mockup

Pencil (free)

Balsamiq (paga)

Mockingbird (free online)

Moqups (pago/free online) *

MockFlow (pago/free online) *

ninjamock.com (pago/free online) *

Mockup Builder pago/free online)





UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.



DISEÑO DE SOFTWARE

Caso de Uso

Modelado de Casos de Uso

- Un caso de uso especifica un comportamiento deseado del sistema.
- Representan los **requisitos funcionales** del sistema.

*“Un caso de uso especifica un **conjunto de secuencias de acciones**, incluyendo **variantes**, que el **sistema puede ejecutar** y que produce un **resultado observable** de valor para un **particular actor.**”* Definición en UML)

- Describen qué hace el sistema, no cómo lo hace.

Actor de Sistema

Un actor representa el rol jugado por una persona o cosa que actúa con el sistema.

“Cliente, Administrador, Usuario no Registrado (Autenticado), Usuario Registrado (Autenticado), Jefe de Compras, Jefe de Personal, Moderador, Jefe de Departamento, Obrero de Planta, Supervisor...”

Casos de Uso -Observaciones

- los casos de uso son de vital importancia en los proyectos de software (*Procesos Guiados por Casos de Uso*)
- Describen bajo la forma de **acciones y reacciones** el comportamiento de un sistema **desde el punto de vista de un usuario**
- Se puede considerar que hasta cierto nivel, cada **caso de uso es independiente** de los demás
- Permiten **definir los límites**

¿Qué es un Escenario?

¿Qué es un Escenario?

Es una **secuencia** de **acciones** e interacciones (pasos) entre los usuarios (**actores**) y el sistema

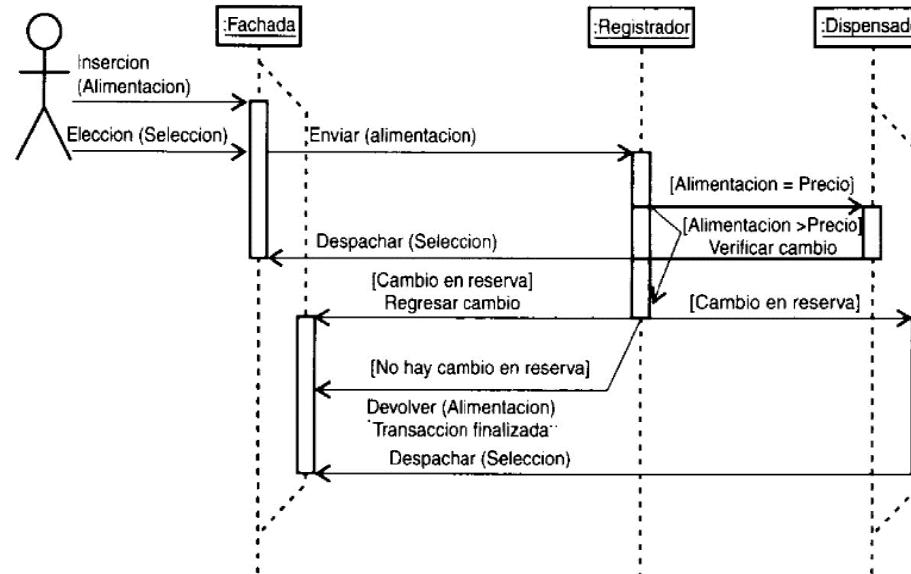
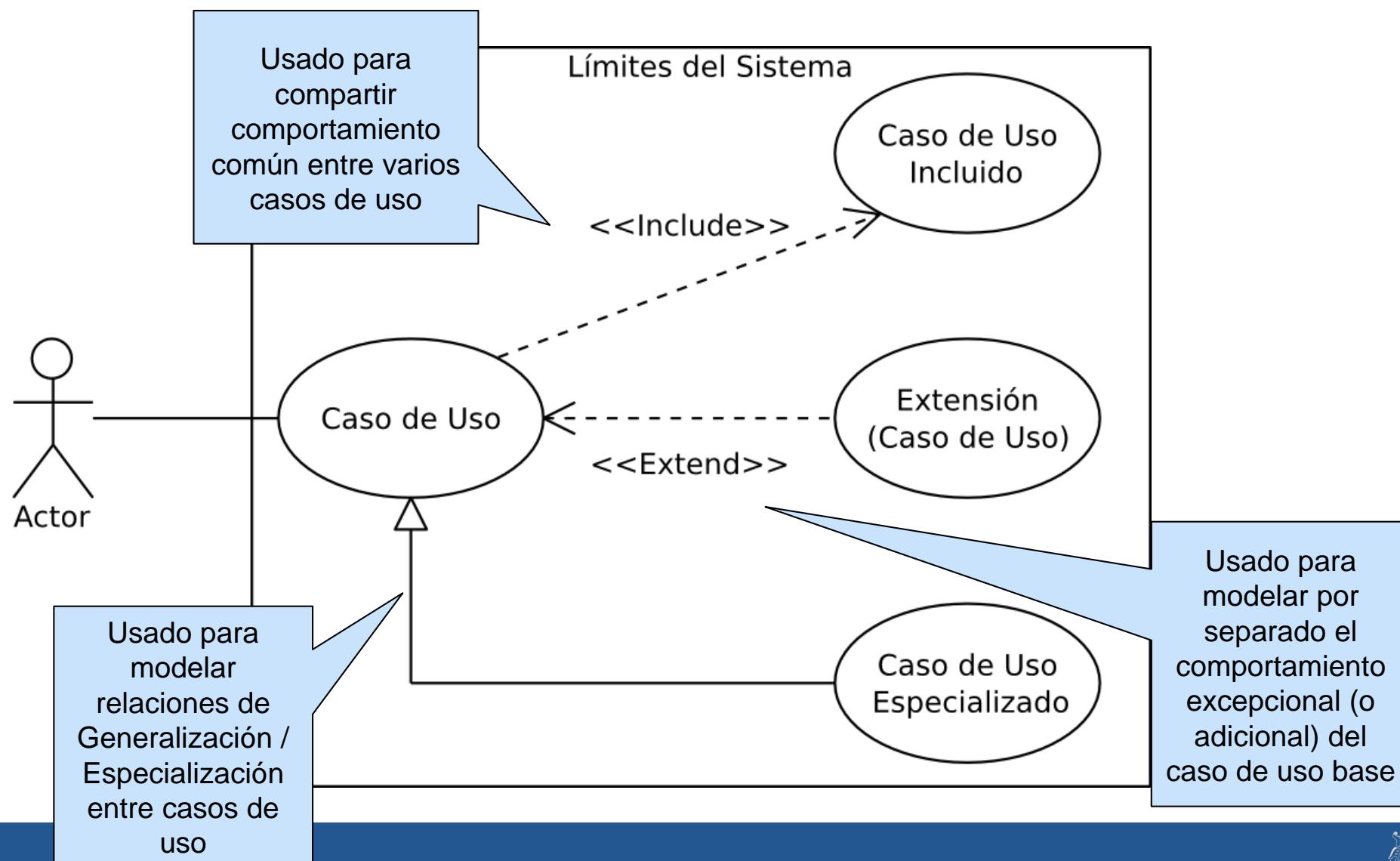


Diagrama de Casos de Usos



Actores

Dos tipos de actores:

- **Principal:**

Requiere al sistema el cumplimiento de un objetivo.

- **Secundarios:**

El sistema necesita de ellos para satisfacer un objetivo.

Propiedades de los casos de uso

1. Son iniciados por un actor
2. Puede incluir secuencias **alternativas**
3. El sistema es considerado como una “**caja negra**
4. **El conjunto completo** de casos de uso especifica todas las posibles formas de usar el sistema.

Descripción de un caso de uso

- Texto estructurado informal
- Texto estructurado formal (**plantillas**)
- Pseudocódigo
- Notaciones gráficas: **diagramas de secuencia**

•

Nombre:	<i><nombre del actor></i>
Descripción: <i><descripción del actor></i>	
Nombre:	Usuario no Autenticado
Descripción:	<p>Representa a un usuario que no se ha identificado frente al sistema. Generalmente estos usuarios deberían poder registrarse (crear un nuevo usuario) o ingresar al sistema para transformarse en usuarios autenticados, en moderadores o en administradores del sistema</p>

Descripción Textual de un Caso de Uso

Nombre:	<i><nombre del caso de uso></i>
Autor:	<i><nombre del autor (o autores) del caso de uso></i>
Fecha:	<i><fecha de creación del caso de uso></i>
Descripción: <i><breve descripción del caso de uso></i>	
Actores: <i><actores participantes en el caso de uso></i>	
Precondiciones: <i><condiciones que deben cumplirse para poder ejecutar el caso de uso></i>	
Flujo Normal: <i><flujo normal (feliz) de ejecución del caso de uso></i>	
Flujo Alternativo: <i><flujos alternativos de ejecución del caso de uso></i>	
Poscondiciones: <i><condiciones que deben cumplirse al finalizar la ejecución del caso de uso></i>	

Descripción Textual de un Caso de Uso

Nombre:	Crear mensaje foro
Autor:	Pedro Pérez
Fecha:	21/04/09
Descripción:	
Permite crear un nuevo mensaje (hilo) en el foro de discusión.	
Actores:	
Usuario / Moderador	
Precondiciones:	
El usuario debe de estar autenticado en el sistema.	

Descripción Textual de un Caso de Uso

Flujo Normal:

- 1.- El actor pulsa sobre el botón para crear un nuevo mensaje.
- 2.- El sistema muestra una caja de texto para introducir el título del mensaje y una zona de mayor tamaño para introducir el cuerpo del mensaje.
- 3.- El actor introduce el título del mensaje y el cuerpo del mismo.
- 4.- El sistema comprueba la validez de los datos y los almacena.
- 5.- El moderador recibe una notificación de que hay un nuevo mensaje.
- 6.- El moderador acepta y el sistema publica el mensaje si éste fue aceptado por el moderador.

Flujo Alternativo:

- 4.A.- El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa al actor de ello permitiéndole que los corrija.
- 7.B.- El moderador rechaza el mensaje, de modo que no es publicado sino devuelto al usuario.

Poscondiciones:

El mensaje ha sido almacenado en el sistema y fue publicado.

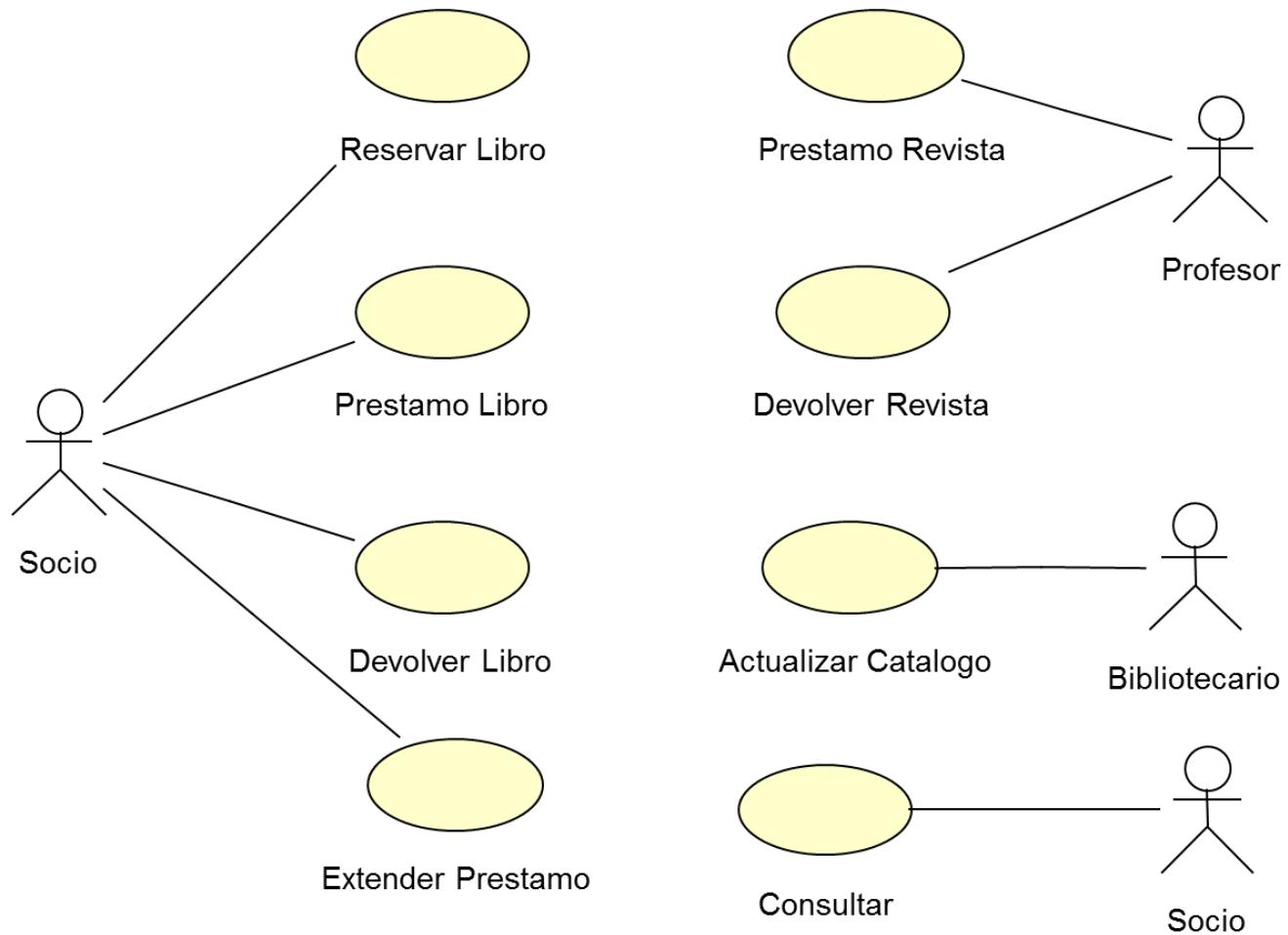
Reglas de Estilo - DCUS

- **nombre único**
- Los nombres de los actores deben **representar roles**
- El nombre de un caso de uso debe **indicar acción** y
- **Verbo (Infinitivo) + Predicado**
- mismo nivel de abstracción
- Evitar el **cruce de líneas**
- **Evite tener demasiados** casos de uso en el mismo diagrama
-

Reglas de Estilo - Para la Descripción Textual de CU

- Narrar el flujo de eventos usando **voz activa, en tiempo presente y desde la perspectiva del actor:**
- Evitar el uso de la **voz pasiva:**
- Preferir la **voz activa**
- Exprese cada paso del flujo usando la forma **llamada y respuesta:**
- El caso de uso que se describe **debe expresar un solo requisito funcional**
- Sin embargo, un caso de uso **puede expresar más de un requisito NO funcional**

Ejemplo : DCU



CU y Colaboraciones

- Con un caso de uso se describe un comportamiento esperado del sistema, pero ***no se especifica cómo se implementa.***
- Una caso de uso se implementa a través de una ***colaboración:***

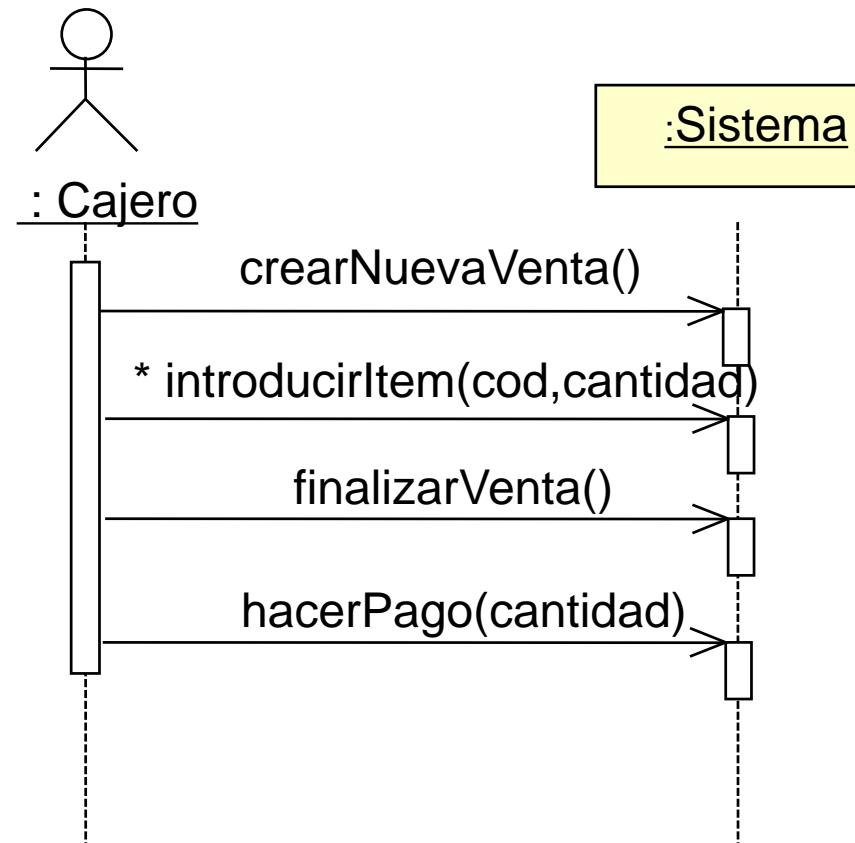
“Sociedad de clases y otros elementos que colaborarán para realizar el comportamiento expresado en un caso de uso”

- Una colaboración tiene una **parte estática** (diagramas de clases) y una **parte dinámica** (diagramas de secuencia).

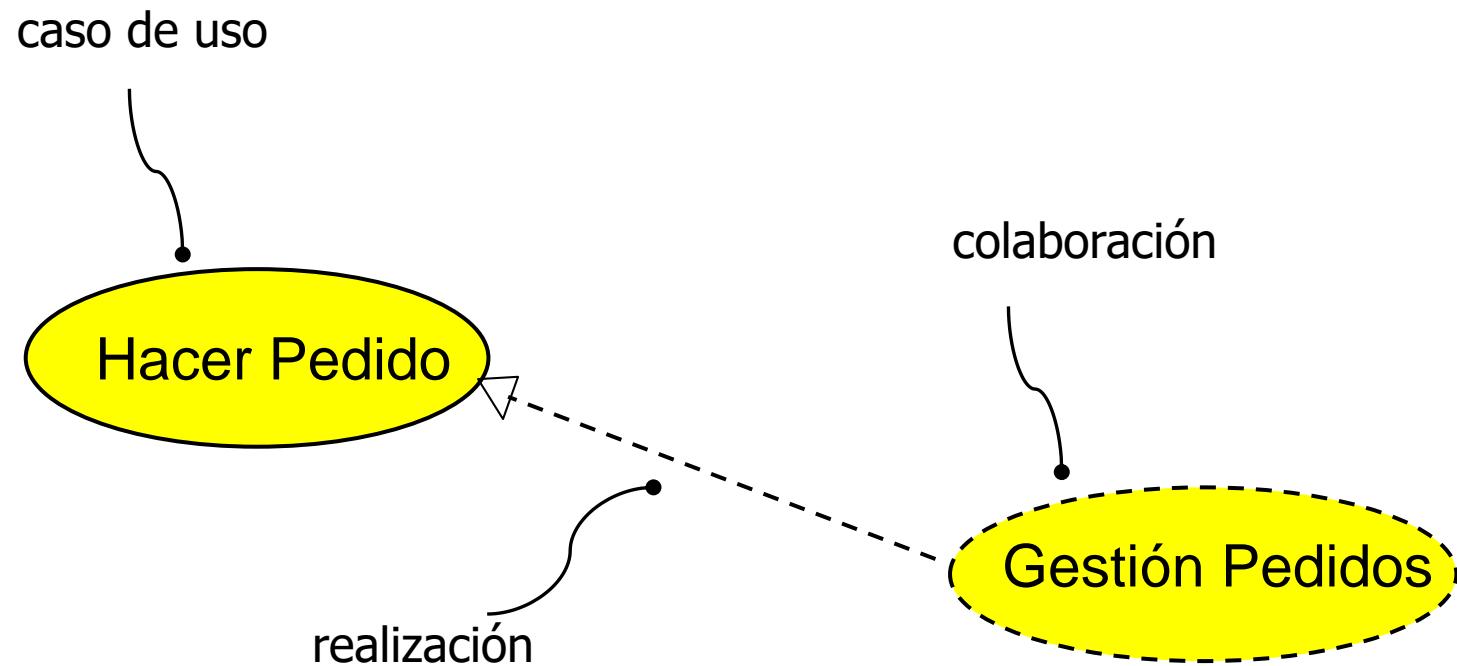
Descripción de un caso de uso: gráfica

Realizar Venta

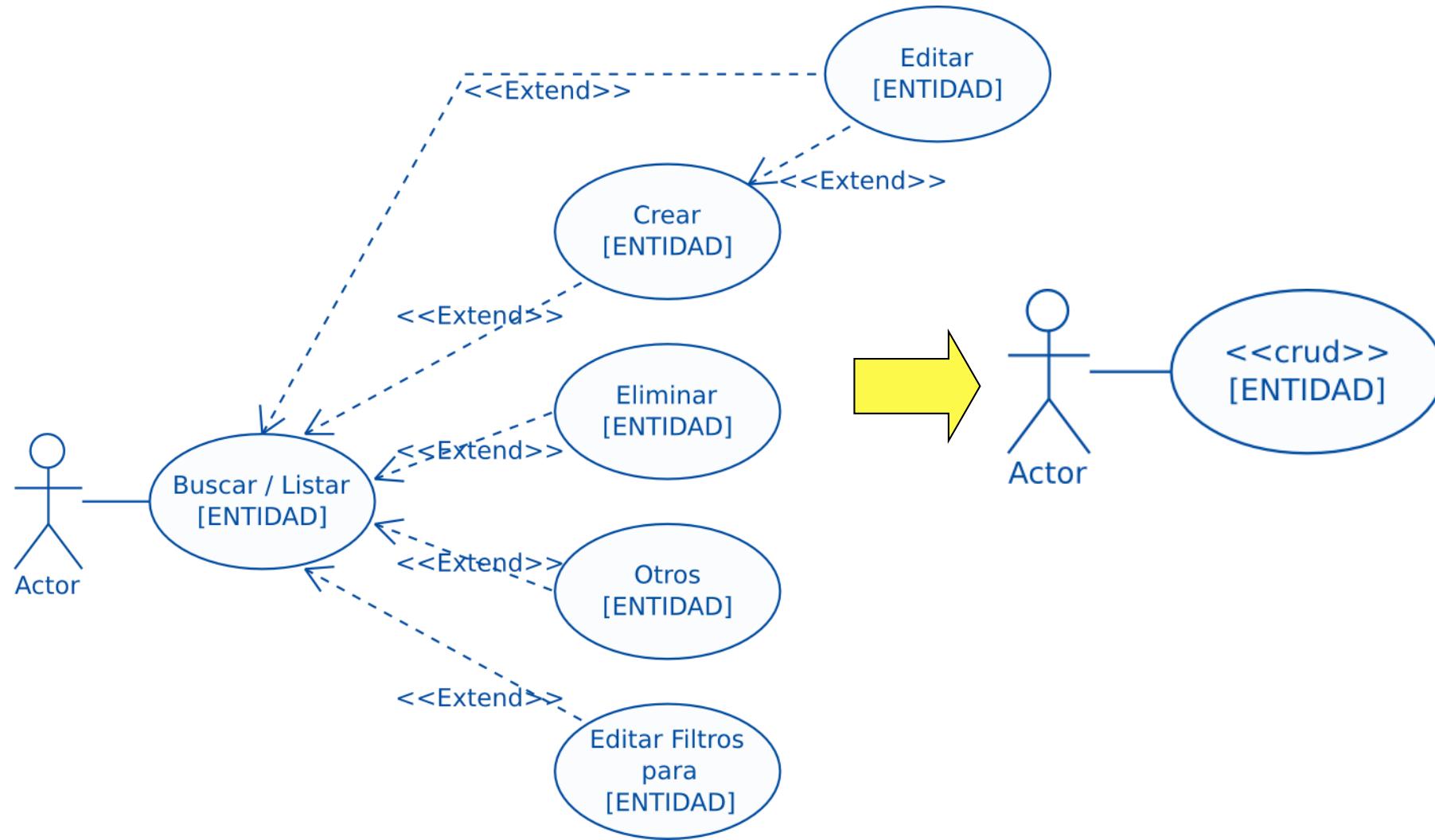
Diagrama de secuencia



Casos de uso y Colaboraciones

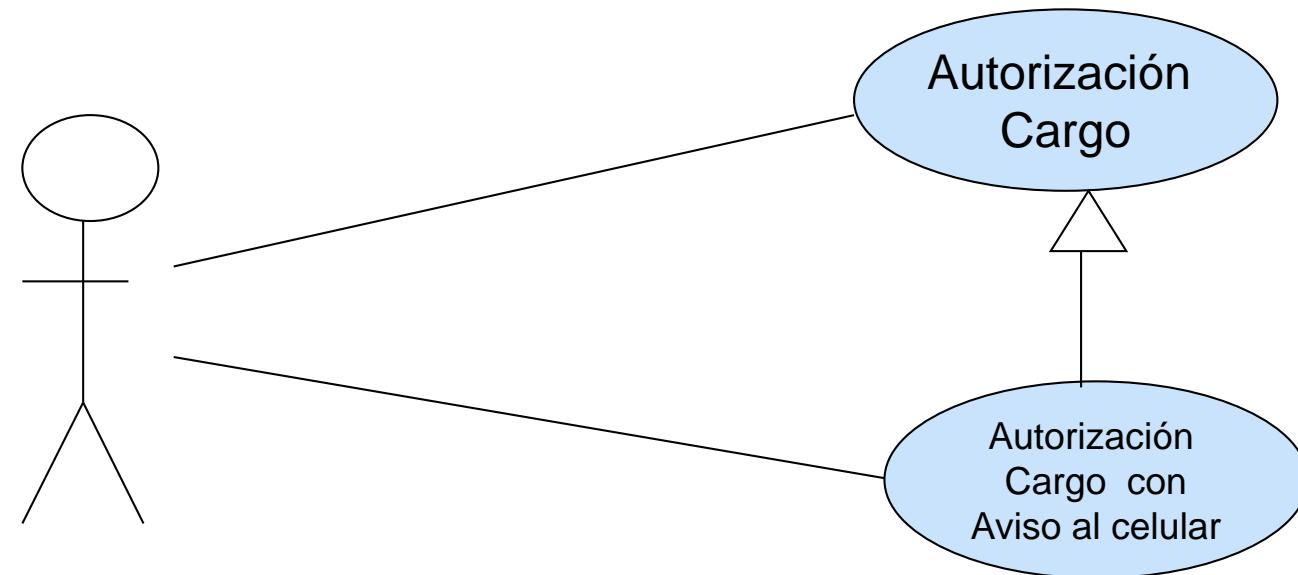


Estereotipos



Generalización

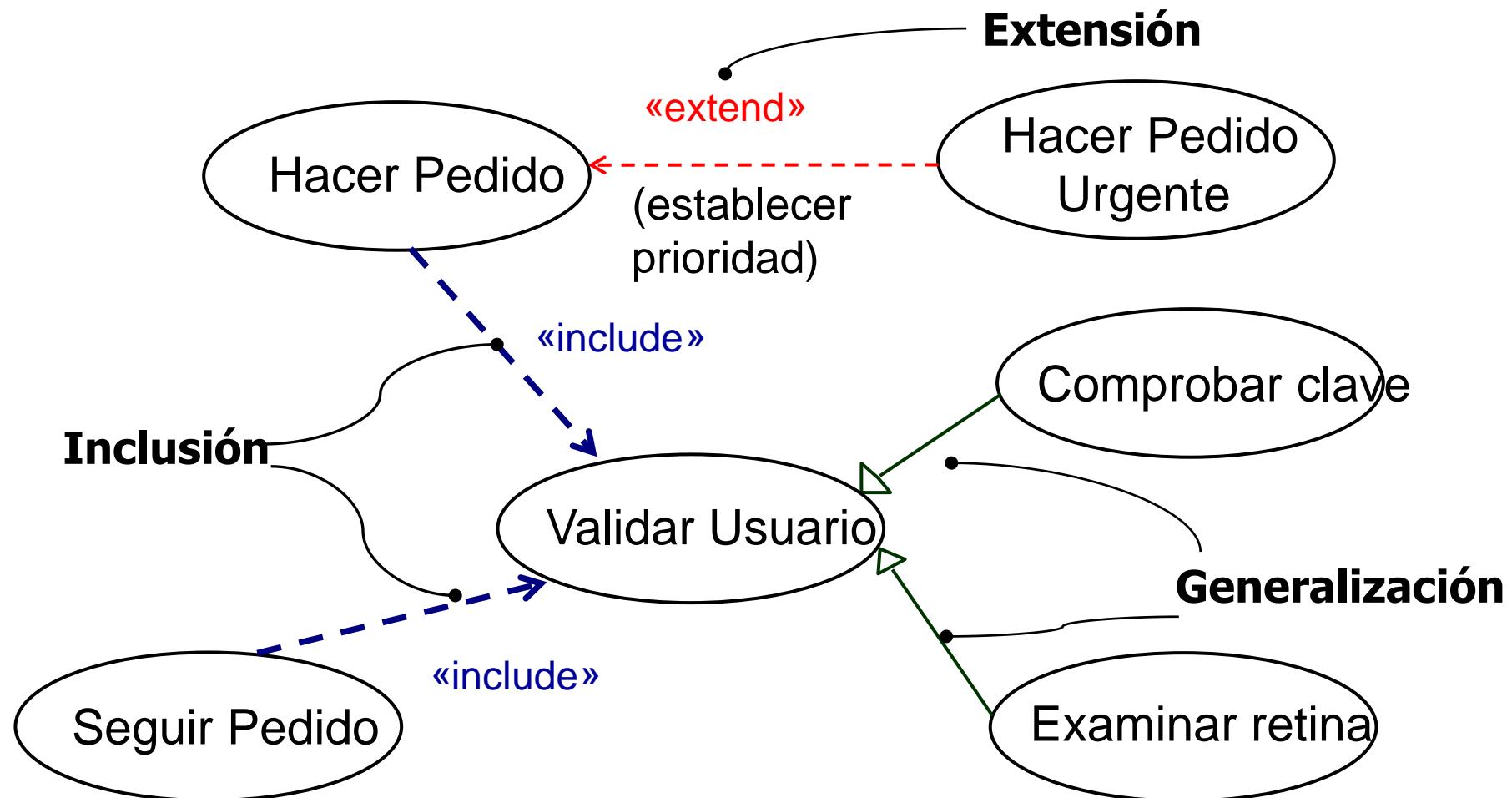
- La herencia indica que un objeto tiene desde el momento de su creación, acceso a todas las propiedades de otra clase.
- Esto mismo se aplica a los actores y a los Casos de Uso, se conoce como generalización y a veces se especifica con una **relación “es un”**



Organización de Casos de uso

- Tres tipos de relaciones:
 - **Generalización**
 - *Un cdu hereda el comportamiento y significado de otro.*
 - **Inclusión**
 - *Un cdu base incorpora explícitamente el comportamiento de otro en algún lugar de su secuencia.*
 - **Extensión**
 - *Un cdu base incorpora implícitamente el comportamiento de otro cdu en el lugar especificado indirectamente por este otro cdu.*

Ejemplo



Relación de inclusión

- **Permite factorizar un comportamiento en un caso de uso aparte y evitar repetir un mismo flujo en diferentes casos de uso.**
- Ejemplo:

Hacer Pedido:

Obtener y verificar el número de pedido;

Incluir "Validar usuario";

Recoger los ítem del pedido del usuario;

...

Relación de extensión

- El caso de uso base incluye una serie de puntos de extensión.
- Sirve para modelar:
 - la **parte opcional** del sistema, o
 - un subflujo que sólo **se ejecuta bajo ciertas condiciones**.

Relación de extensión

- Ejemplo:

Hacer Pedido:

Incluir “*Validar usuario*”;

Recoger los ítem del pedido del usuario;

Establecer prioridad: punto de extensión

Enviar pedido para ser procesado según la prioridad.

Plantilla *usecases.org* (Larman)

- Resumen
- Actores Principales y Secundarios
- Personas involucradas e Intereses
- Precondiciones
- Poscondiciones
- Escenario Principal (Flujo Básico)
- Extensiones (Flujos Alternativos)
- Requisitos de Interfaz de Usuario
- Requisitos No-Funcionales
- Cuestiones Pendientes

Granularidad

- Diferente granularidad
 - **Casos de uso del negocio**
 - Procesos de Negocio: Objetivo estratégico de la empresa
 - Ej. Vender productos
 - **Casos de uso del sistema**
 - Objetivo de un usuario
 - Ej. Realizar una compra
 - **Casos de uso de inclusión**
 - Forman parte de otro, son como subfunciones
 - Ej. Buscar, Validar, Login

Ejemplo

Una empresa gestiona un conjunto de inmuebles, que administra en calidad de propietaria.

Cada inmueble puede ser bien un local (local comercial, oficinas, etc.), un departamento o bien un edificio que a su vez tiene departamento y locales. Como el número de inmuebles que la empresa gestiona no es un número fijo, la aplicación debe permitir tanto introducir inmuebles nuevos, así como darlos de baja, modificarlos y consultarlos.

Asimismo, que una empresa administre un edificio determinado no implica que gestione todos sus departamento y locales, por lo que la aplicación también deberá permitir introducir nuevos pisos o locales, darlos de baja, modificarlos y hacer consultas sobre ellos.

Cualquier persona que tenga una nómina, un aval bancario, un contrato de trabajo o venga avalado por otra persona puede alquilar el edificio completo o alguno de los departamentos o locales que no estén ya alquilados, y posteriormente desalquilarlo.

Por ello, deberán poder ser dados de alta, si son nuevos inquilinos, con sus datos correspondientes (nombre, DNI, edad, sexo, ...), poder modificarlos, darlos de baja, consultarlos, etc. La aplicación ofrece acceso web para que un inquilino puede modificar o consultar sus datos, pero no darse de baja o de alta. Para la realización de cualquiera de estas operaciones es necesaria la identificación por parte del inquilino.





UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.



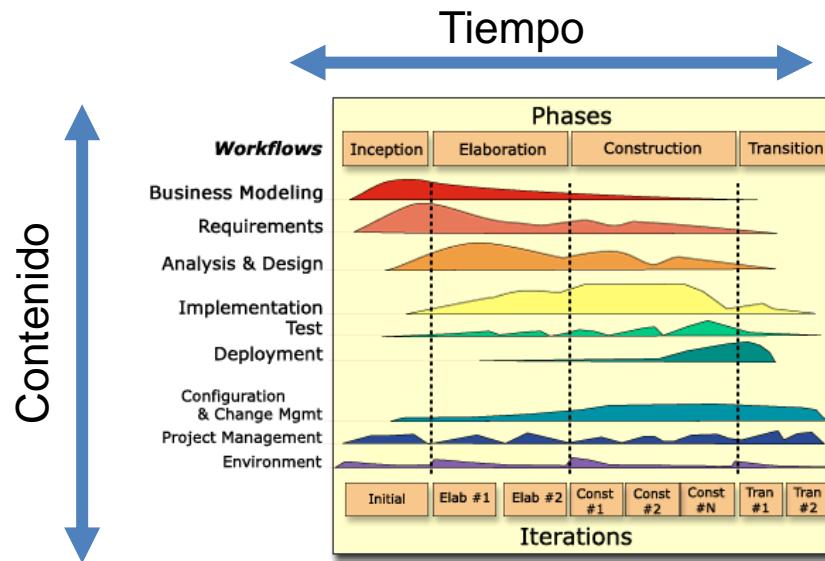
DISEÑO DE SOFTWARE

Diseño según el Proceso Unificado

Sesión S4

El Proceso Unificado de Desarrollo

Es un proceso de ingeniería del software que agrupa las **6 mejores prácticas** de desarrollo software que existen en el mercado.



Características

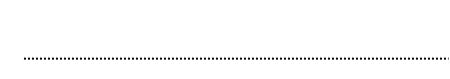
- Está basado en *componentes* e *interfaces* bien definidas
- Utiliza el Lenguaje Unificado de Modelado (UML)
- Aspectos característicos:
 - **Dirigido por casos de uso**
 - **Centrado en la arquitectura**
 - **Iterativo e incremental**

Fase Requisitos: Artefacto

Pieza de **información utilizada o producida por** un proceso de desarrollo de software

Artefactos implicados durante la captura de requisitos

Modelo de Casos de Uso



Actor



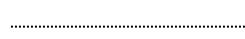
Glosario



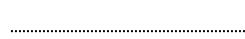
Caso de Uso



Prototipo de Interfaz de Usuario

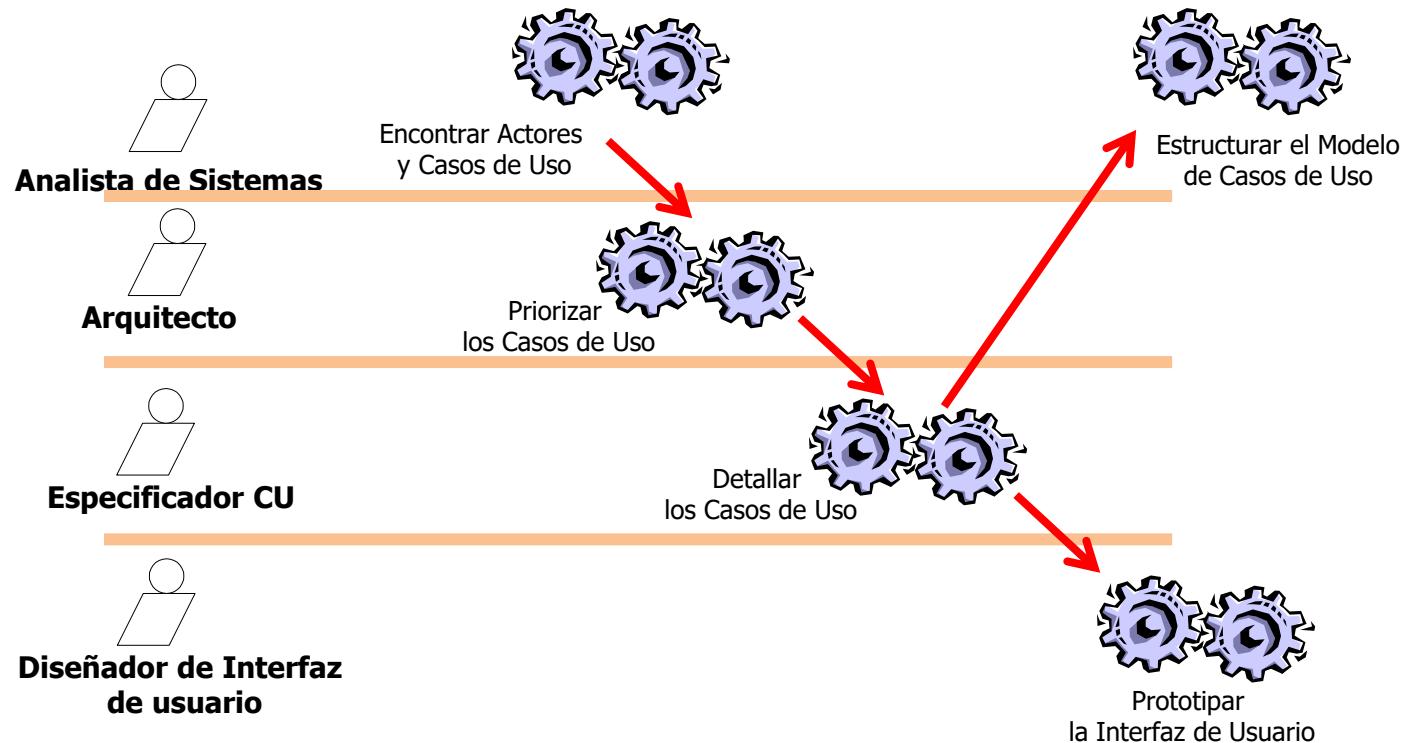


Descripción de la Arquitectura



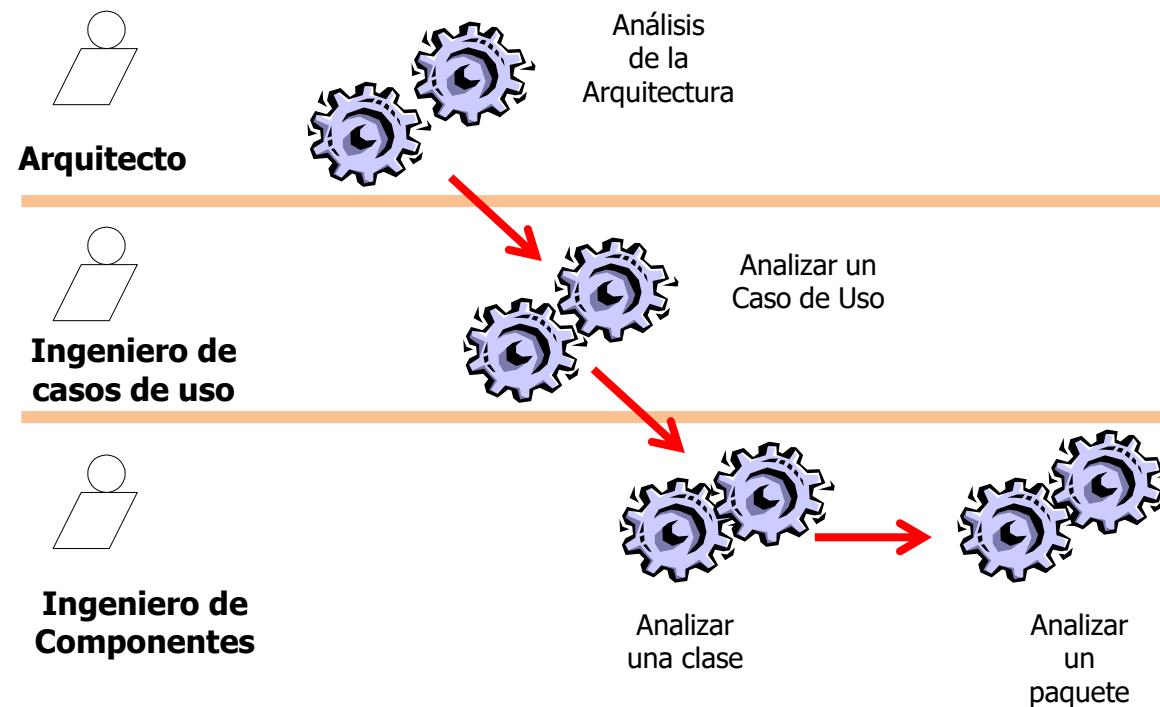
Work Flow

Flujo de Trabajo de Requisitos

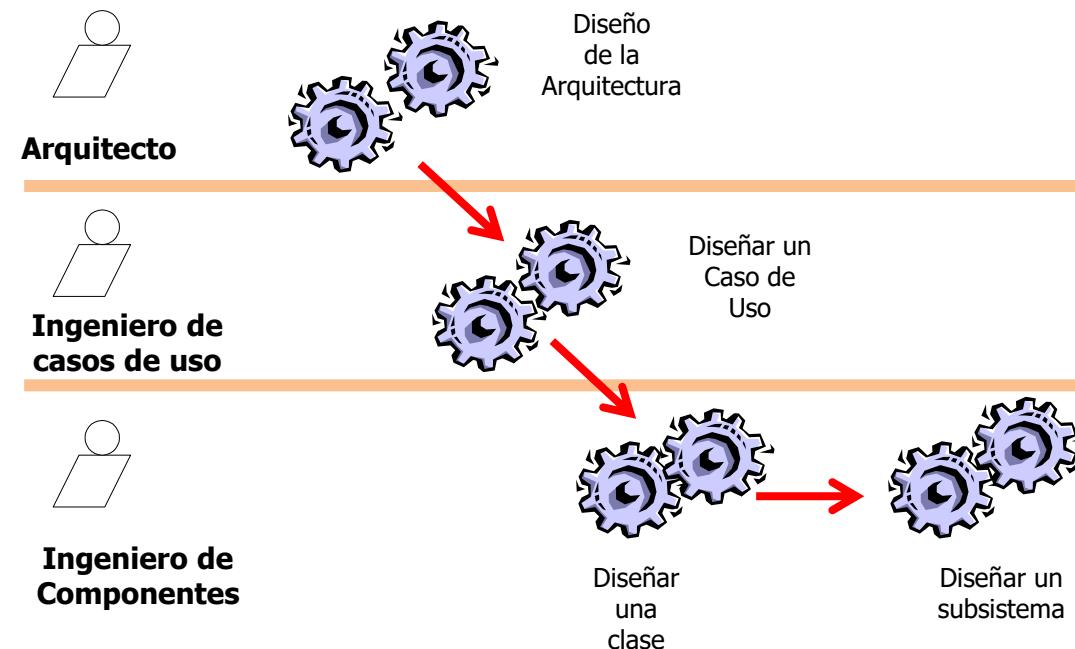


Work Flow de Análisis

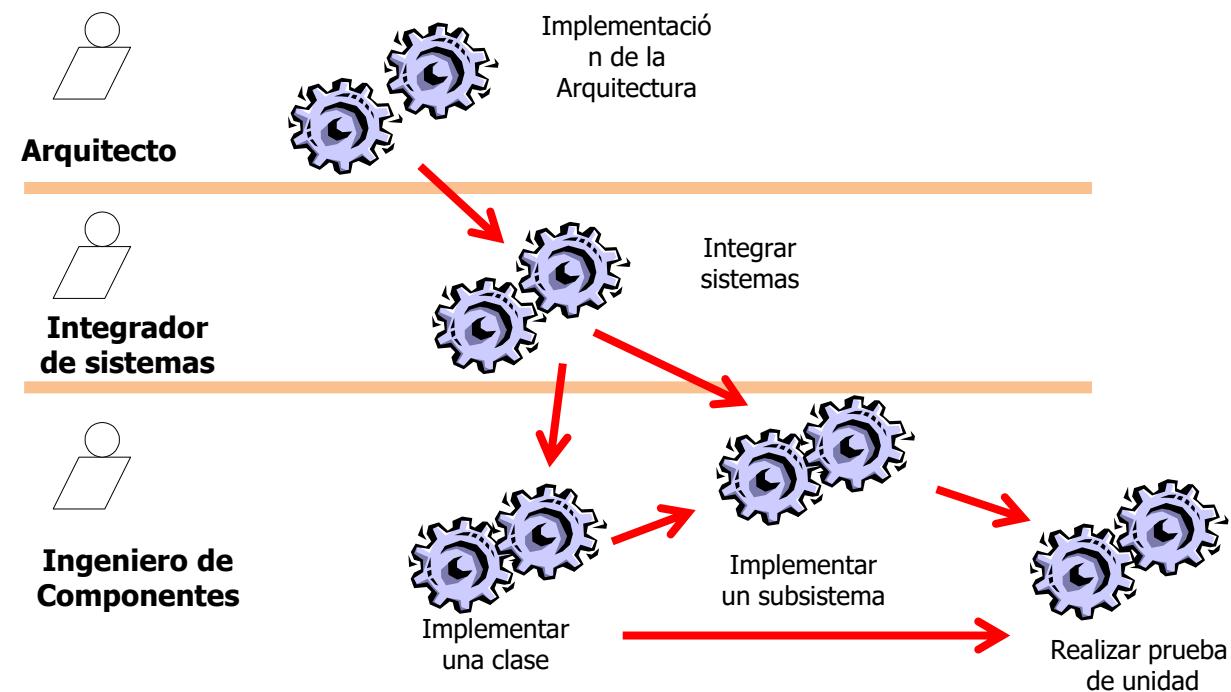
Ofrecer una especificación más precisa de los requisitos que la que tenemos como resultado de los requisitos.



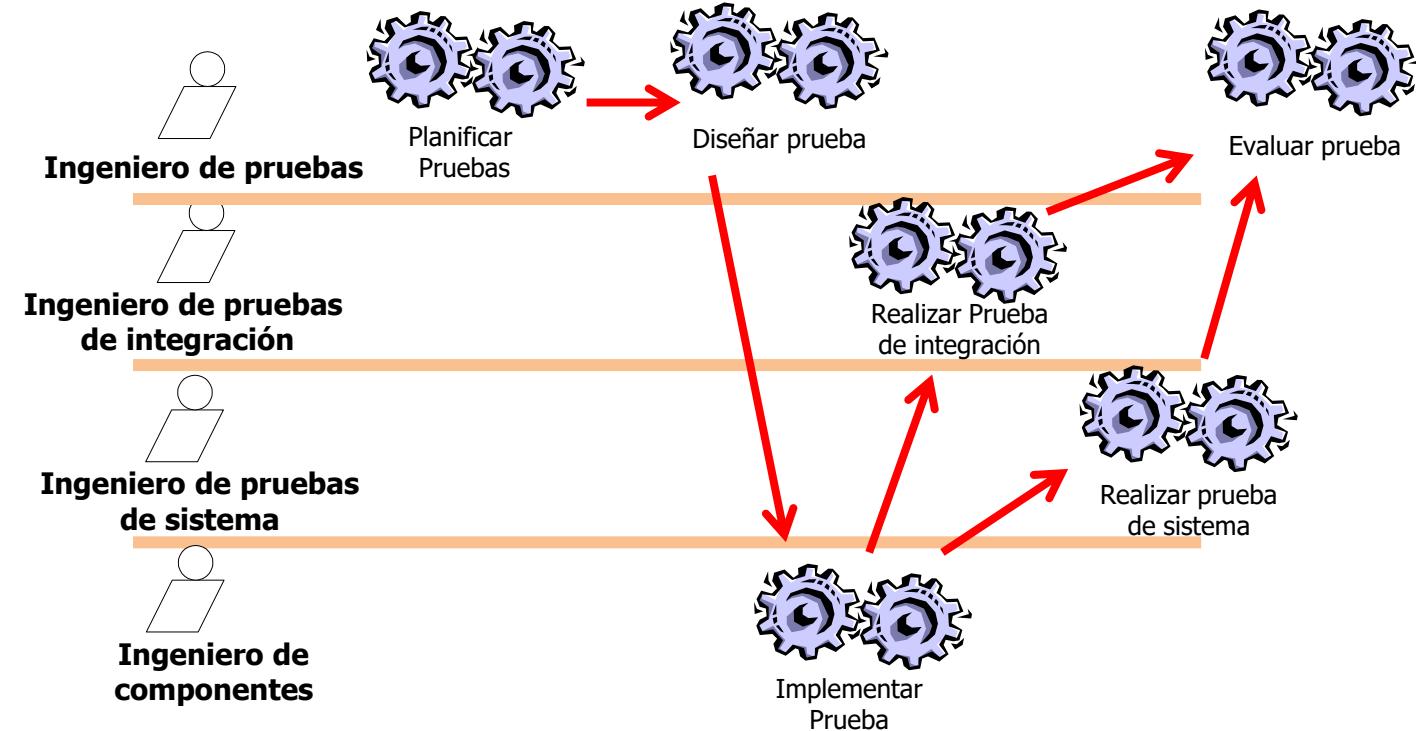
Work Flow de Diseño



Work Flow de Implementación



Work Flow de Pruebas



Fases del proceso

Fase de inicio

Objetivo: Establece la viabilidad del proyecto

Se fundamenta el análisis de negocio inicial:

- Se delimita el ámbito del sistema
- Se propone o esboza una arquitectura del sistema
- Se identifican riesgos críticos
- Se demuestra a los usuarios la utilidad del sistema propuesto

Productos de la fase:

- Lista de características
- Primera versión del modelo del negocio
- Primera versión del modelo de casos de uso, de análisis y de diseño
- Descripción de la arquitectura candidata
- Prototipo exploratorio
- Lista inicial de riesgos y clasificación de casos de uso
- Plan para el proyecto
- Primer borrador del análisis del negocio

	Modelo negocio	Casos de uso identificados	Casos de uso descritos	Casos de uso analizados	Casos de uso diseñados, implementados y probados
Fase inicio	50% -70%	50%	10%	5%	Lo suficiente para el prototipo
Fase elaboración	Cerca del 100%	>80%	40% - 80%	20% - 40%	<10%
Fase construcción	100%	100%	100%	100% si se mantiene	100%

Hito : Objetivos (visión)

Fase de elaboración

Objetivos: Elaborar una arquitectura estable, Conocer suficientemente el sistema como para planificar en detalle la fase de construcción

Tareas básicas:

- Crear una línea base para la arquitectura
- Identificar riesgos significativos
- Especificar atributos de calidad
- Estudiar 80% de los requisitos funcionales

Productos

- Modelo del negocio completo
- Versión de los modelos
- Línea base de la arquitectura
- Lista de riesgos actualizada
- Plan de proyecto para construcción y transición
- Manual de usuario (opcional)
- Análisis del negocio completo

Hito : Arquitectura

Fase de construcción

Objetivo: La capacidad de operación inicial.

Versión beta, requiere mayor número de iteraciones

Tareas básicas:

- Extensión a todos los casos de uso
- Finalización del análisis, diseño, implementación y prueba
- Mantenimiento de la integridad de la arquitectura
- Monitorización de los riesgos críticos y significativos.

Productos

- El plan de proyecto para la fase de transición
- El sistema software ejecutable
- Todos los artefactos
- La descripción de la arquitectura actualizada
- Versión preliminar del manual de usuario
- Análisis del negocio actualizado

Hito : Funcionalidad operativa

Fase de transición

- **Objetivos:** Despliegue del producto en el cliente

Tareas:

- Completa la versión del producto
- Se gestionan los aspectos relativos al entorno del cliente
- Se corrigen los defectos de la versión beta
- Se terminan los manuales de usuario y cursos de formación
- La atención se desplaza a la corrección de defectos

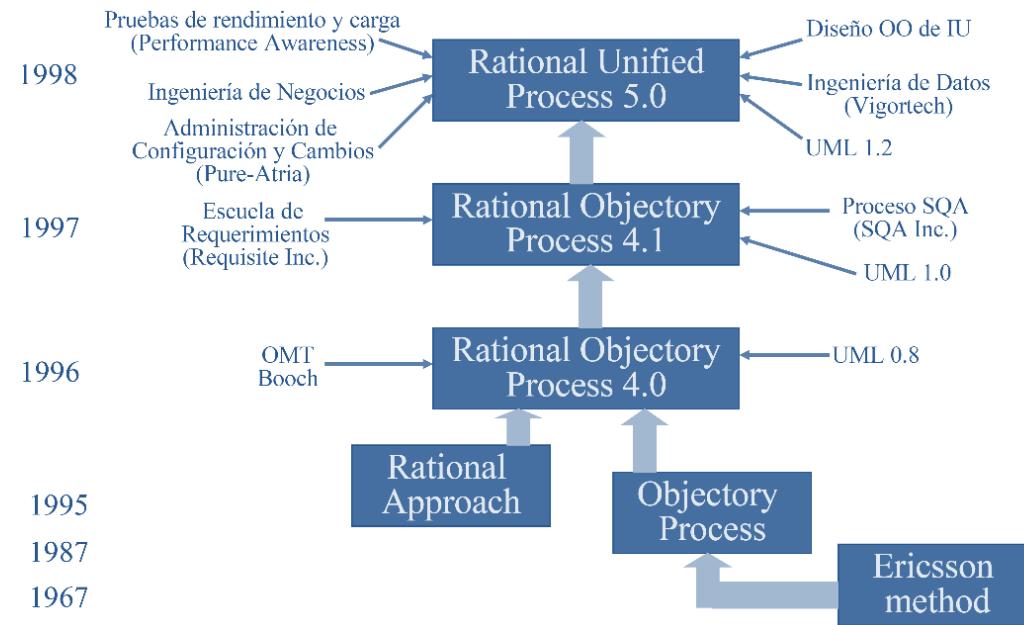
Productos:

- El sistema software ejecutable + software instalación
- Documentos legales, contratos, licencias, garantías
- Versión completa y corregida del producto, incluyendo los modelos del sistema
- La descripción de la arquitectura completa y actualizada
- Manuales y material de formación del usuario, del operador y del administrador
- Referencias para la ayuda del cliente, cómo informar de defectos

Hito : Release del producto

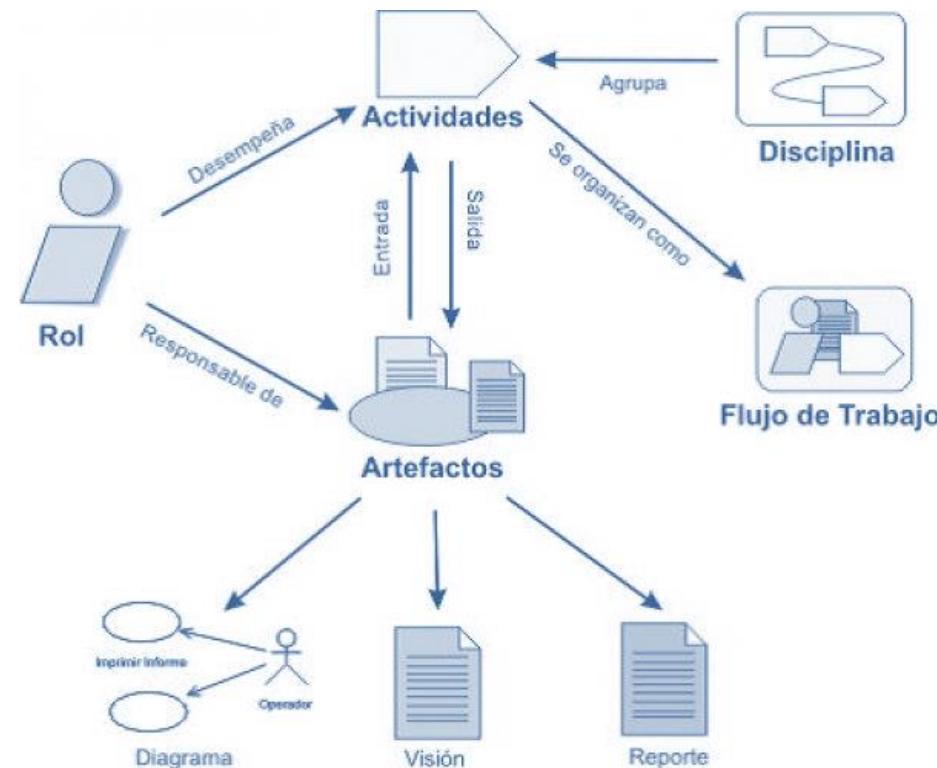
El Proceso Unificado de Rational - RUP

El **Rational Unified Process** fue el resultado de una convergencia de Rational Approach y Objectory (el proceso de la empresa Objectory AB). El primer resultado de esta fusión fue el Rational Objectory Process, la primera versión de RUP, fue puesta en el mercado en 1998, siendo el arquitecto en jefe Philippe Kruchten.



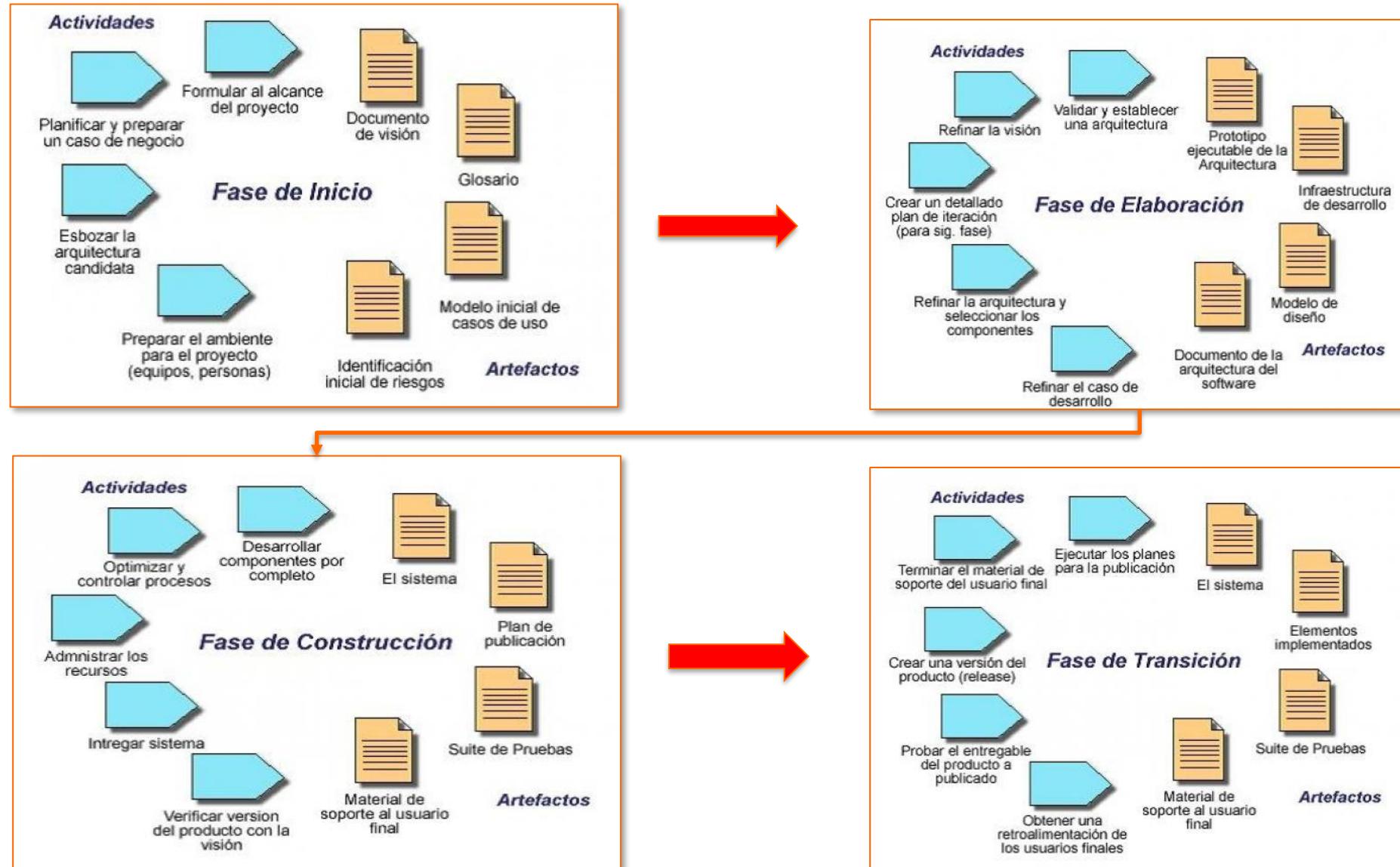
Elementos de RUP

Con RUP, un proceso de desarrollo es representado usando un conjunto de elementos de modelado. En RUP se encuentran 4 elementos básicos, los roles **el quién**, las actividades **el cómo**, los artefactos **el qué** y los flujos de trabajo **el cuándo**.

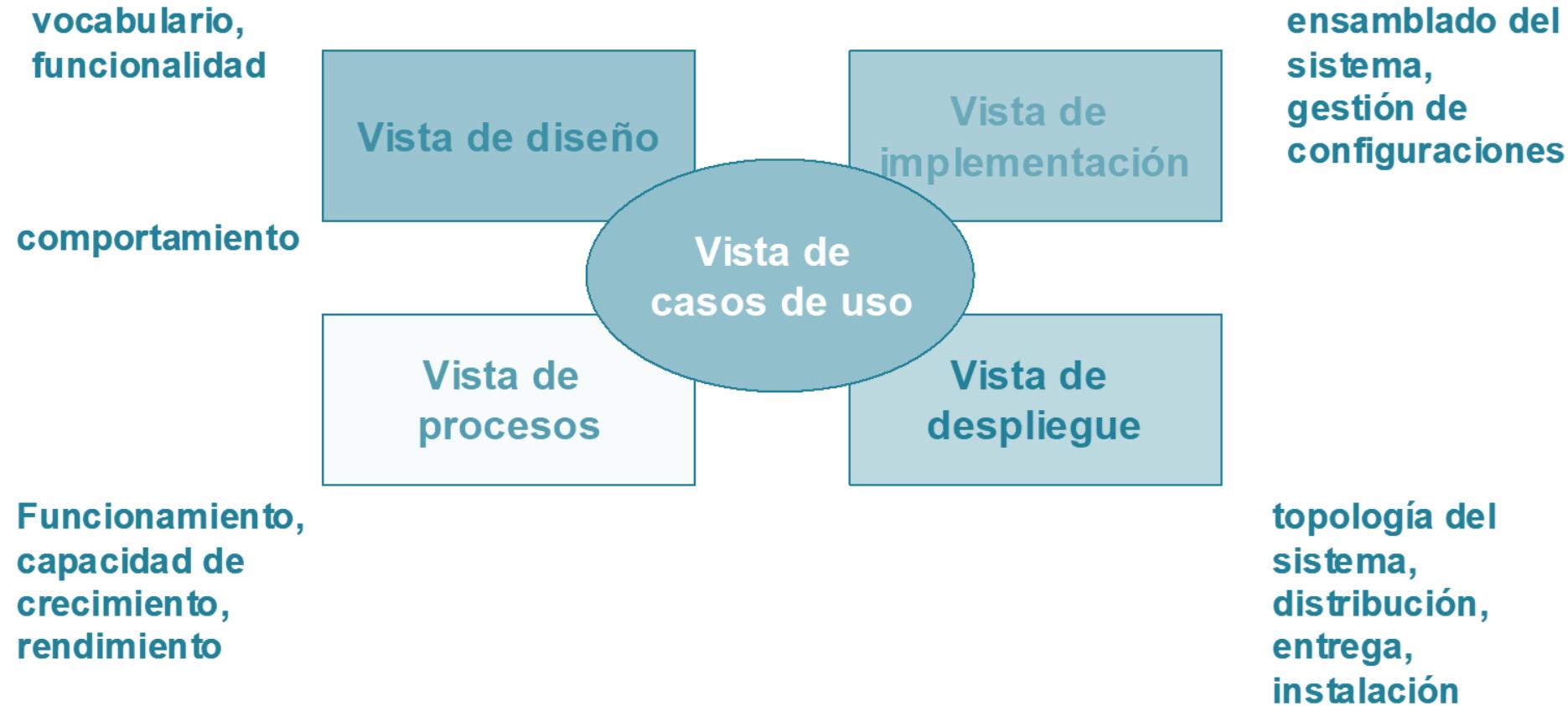


Elementos Básicos De RUP

Elementos de RUP



Vistas de la arquitectura de RUP



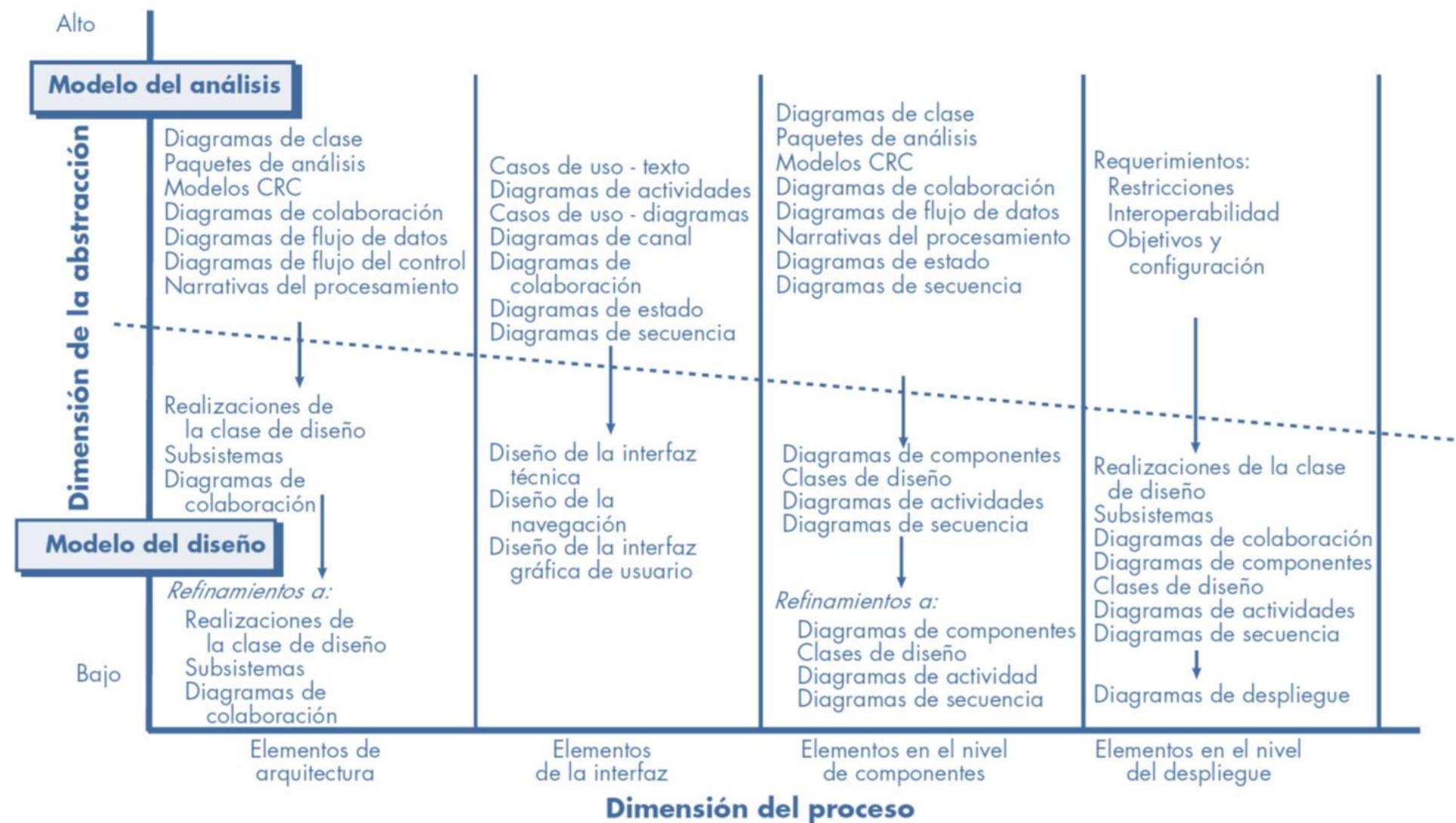
El modelo de diseño

El modelo de diseño

La dimensión del proceso

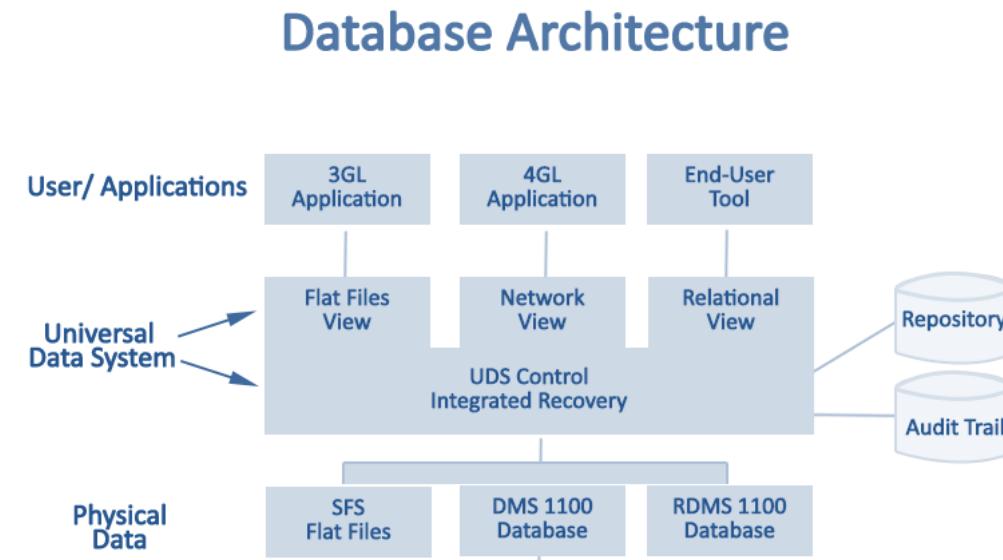
La dimensión de Abstracción

Dimensiones del modelo de diseño



Elementos del diseño de datos

El diseño de datos (algunas veces llamado arquitectura de datos), crea un **modelo de datos o información** que se representa con un alto grado de abstracción (la visión de los datos del cliente/usuario).



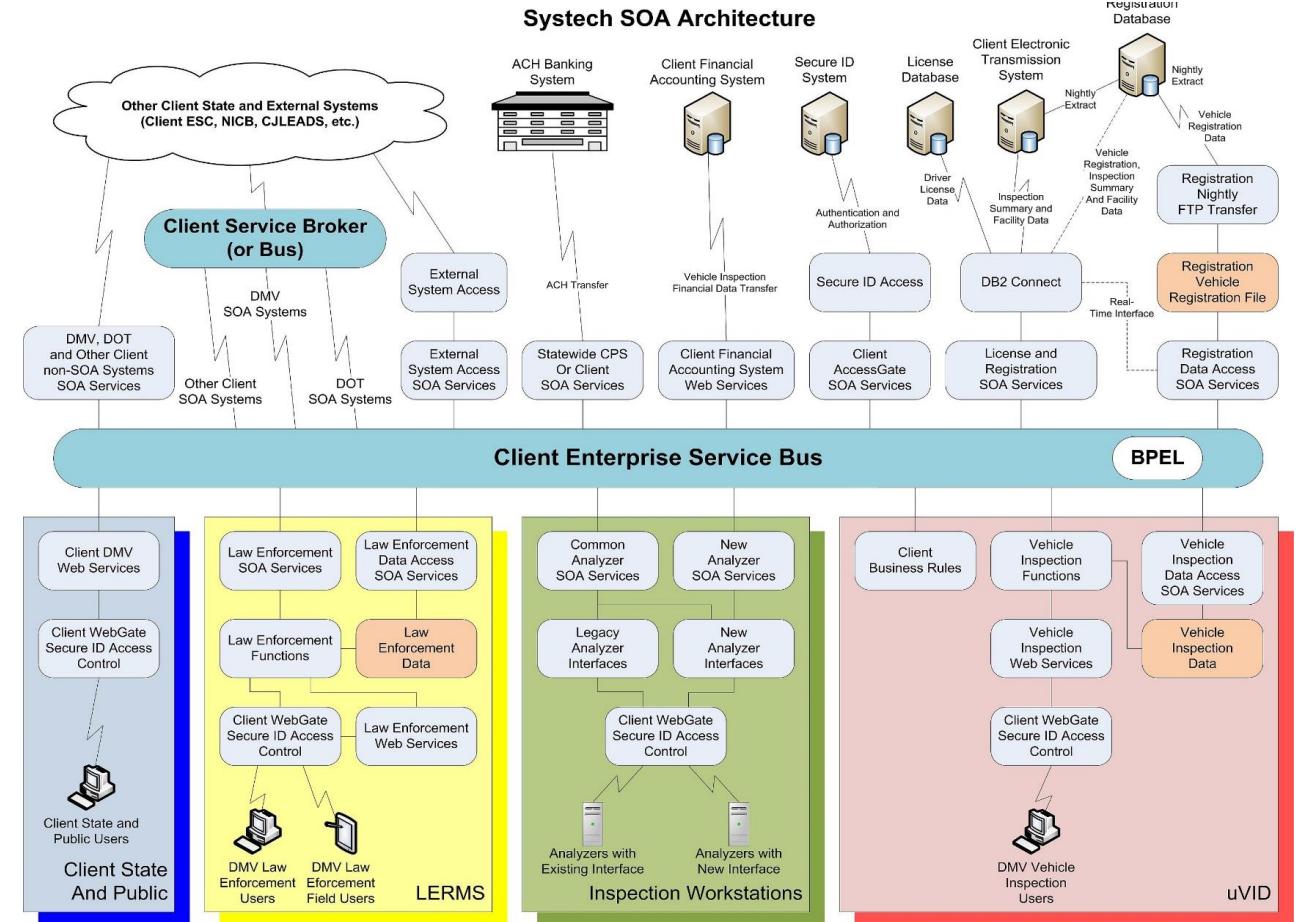
Elementos del diseño de datos

Importancia de la estructura de datos

A nivel de los componentes del sistema

A nivel de aplicación

A nivel de negocios



2. Elementos del diseño arquitectónico

- El diseño arquitectónico para el SW es como el **plano de planta de una casa**.
- Los elementos del diseño arquitectónico dan una visión general de SW.
- El modelo arquitectónico se obtiene de 3 fuentes:
 - La información acerca del dominio
 - Los elementos del modelo de análisis
 - patrones y estilos arquitectónicos

3. Elementos del diseño de interfaces

Los elementos del diseño de interfaz para SW **muestran como fluyen la información hacia o fuera** del sistema y cómo éste está comunicado entre los componentes definidos como parte de la arquitectura.

Elementos importantes del diseño:

- La **interfaz con el usuario**.
- Las **interfaces externas** a otros sistemas, artefactos, redes u otros productores o consumidores de información.
- **Interfaces internas** entre varias componentes de diseño.

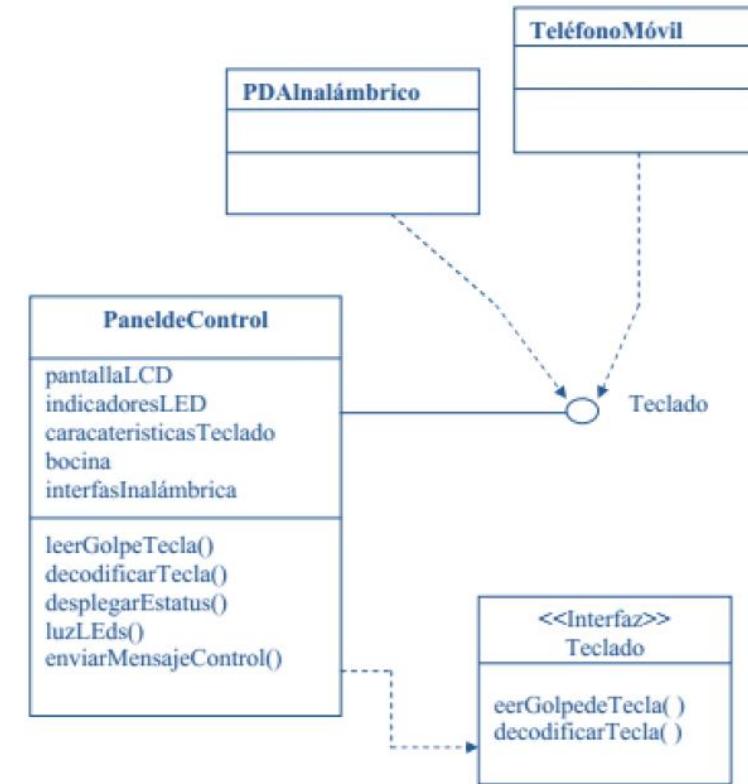
3. Elementos del diseño de interfaces

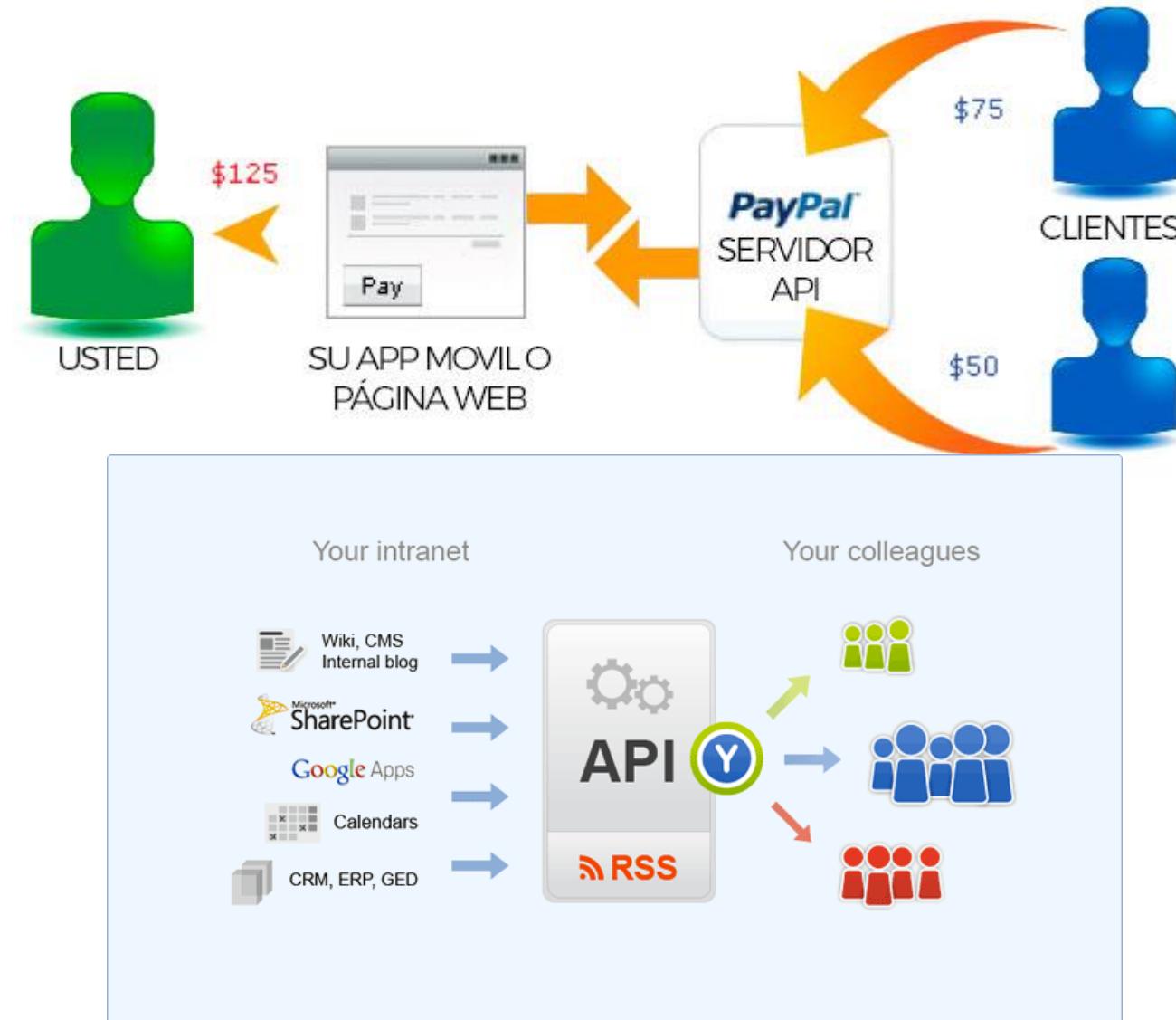
El diseño de las interfaces externas requiere información **definitiva** acerca de la entidad hacia donde se manda o recibe información.

El diseño de interfaces internas está cercanamente **alineado** con el diseño al nivel de los componentes.

Las realizaciones del diseño de clases de análisis representan todas las operaciones y esquemas de mensajes requeridos para permitir la comunicación y colaboración entre las operaciones de varias clases.

Figura 9.5 Representación en UML de la interfaz para el panel de control





4. Elementos del diseño a nivel de componentes

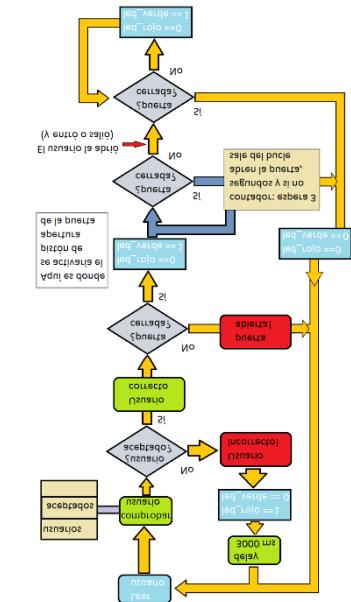
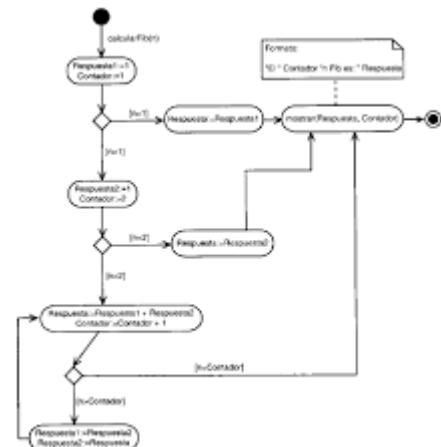
Para el SW: Describe por completo el detalle interno de cada componente del SW.

Los detalles de diseño a nivel de componentes se pueden modelar a muchos grados distintos de abstracción.

```

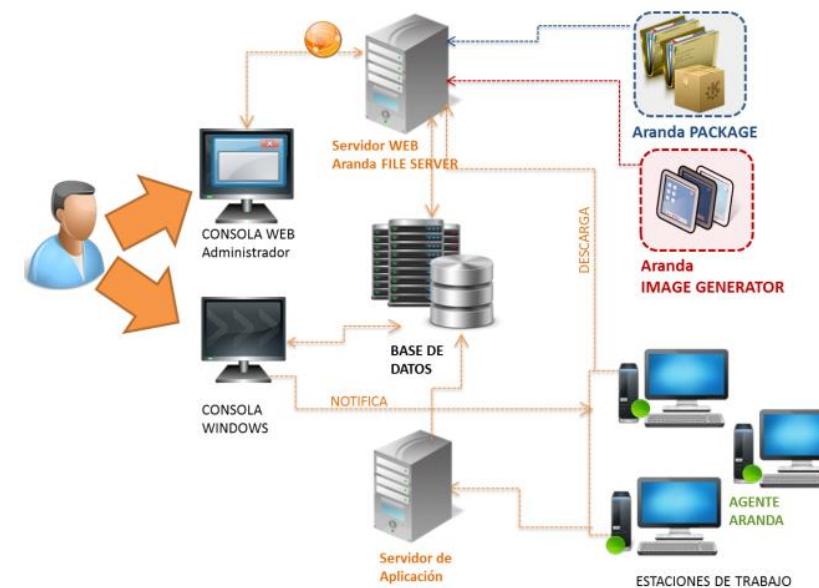
Procedimiento Ordenar (L)
  //Comentario : L = (L1, L2, ..., Ln) es una lista con n elementos/
  k ← 0;
  Repetir
    intercambio ← Falso;
    k ← k + 1;
    Para i ← 1 Hasta n - k Con Paso 1 Hacer
      Si Li > Li+1 Entonces
        intercambiar (Li, Li+1);
        intercambio ← Verdadero;
      Fin Si
    Fin Para
  Hasta Que intercambio = Falso;
Fin Procedimiento

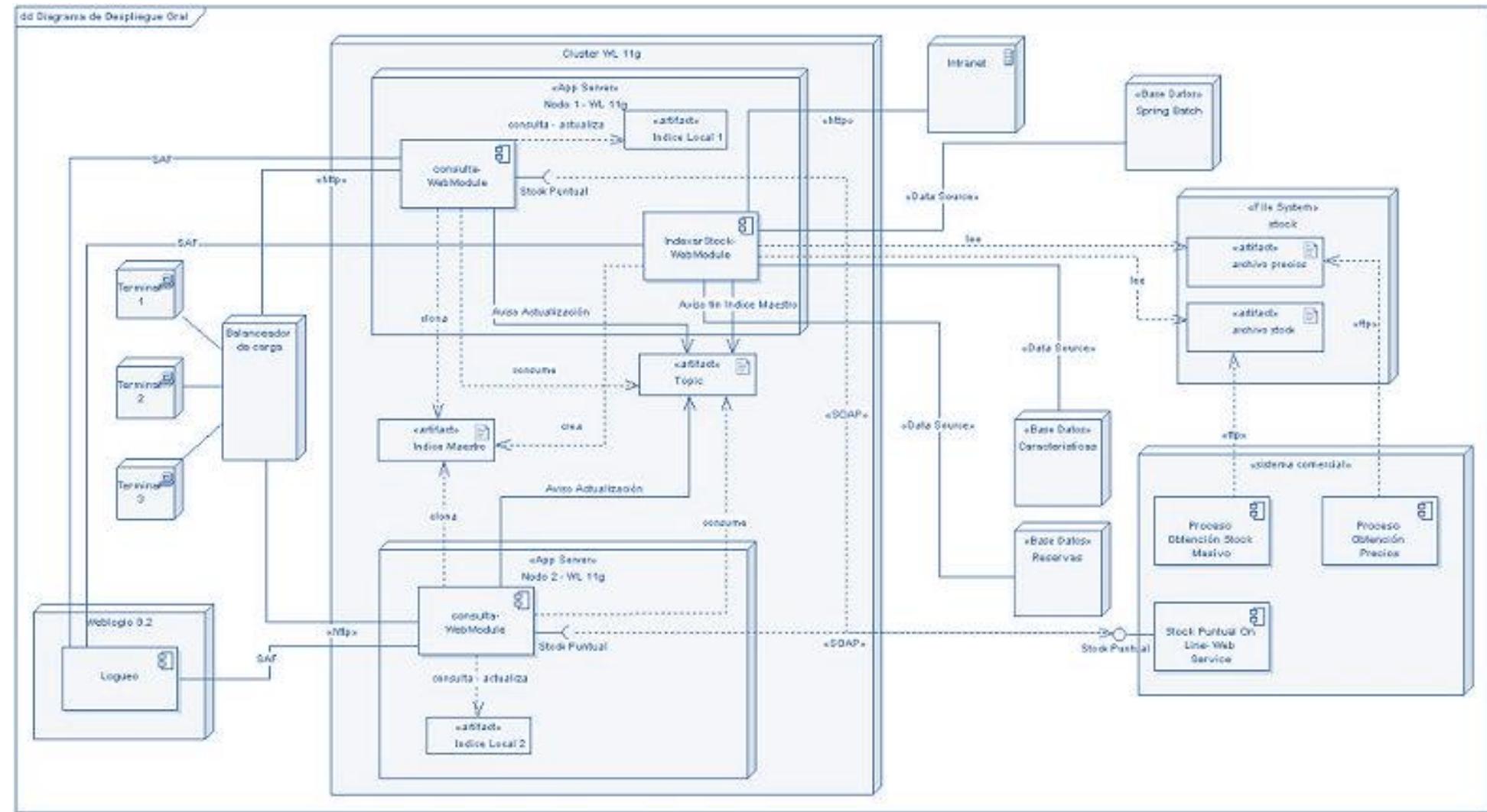
```

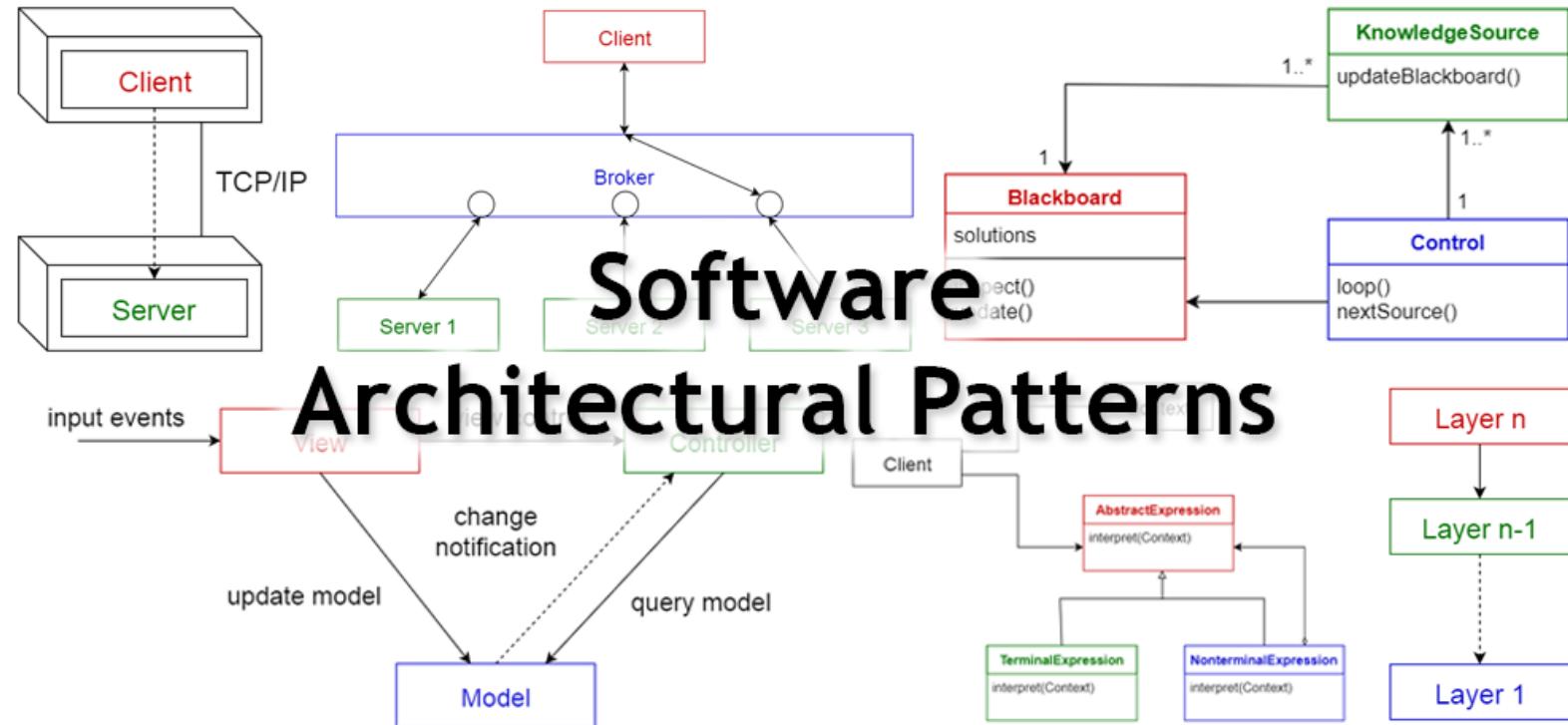


5. Elementos de diseño a nivel del despliegue

- Los elementos de diseño a nivel de despliegue indican **como se ubicarán la funcionalidad y los subsistemas dentro del entorno computacional físico que soportará al SW.**
- Durante el diseño se desarrolla un diagrama de despliegue en UML y después se refina

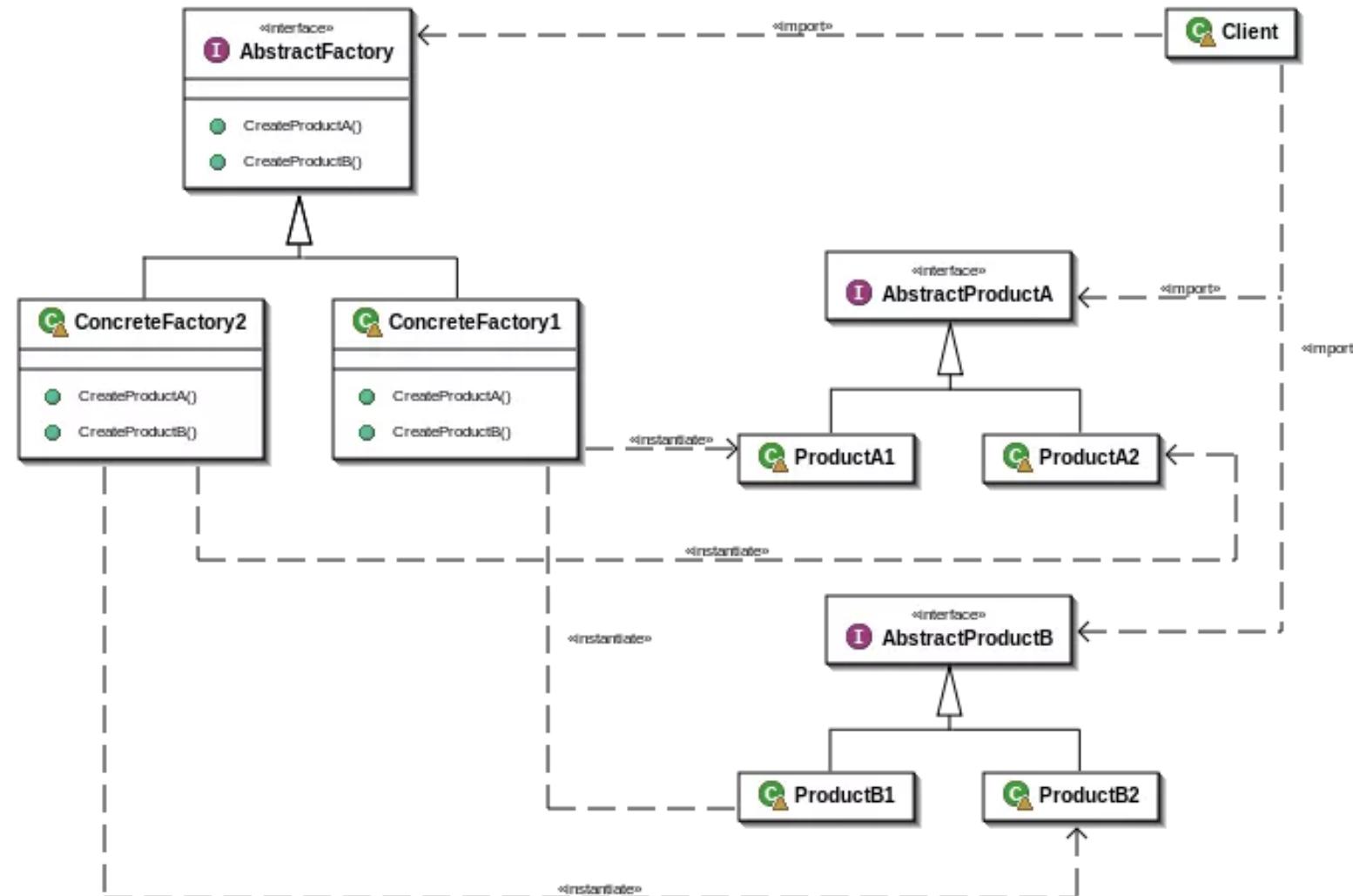






Diseño del SW basado en Patrones

- Los mejores diseñadores en cualquier campo tienen la misteriosa habilidad de vislumbrar **patrones que caracterizan un problema y los patrones correspondientes que pueden combinarse para crear una solución.**
- A través del proceso de diseño un **Ingeniero de SW debe buscar toda oportunidad para reutilizar patrones de diseños existentes** (cuando cumplen la necesidad de un diseño) en vez de crear nuevos.







UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.



DISEÑO DE SOFTWARE

Del Modelo del Análisis al Diseño
Arquitectura 4 + 1

Sesión S5

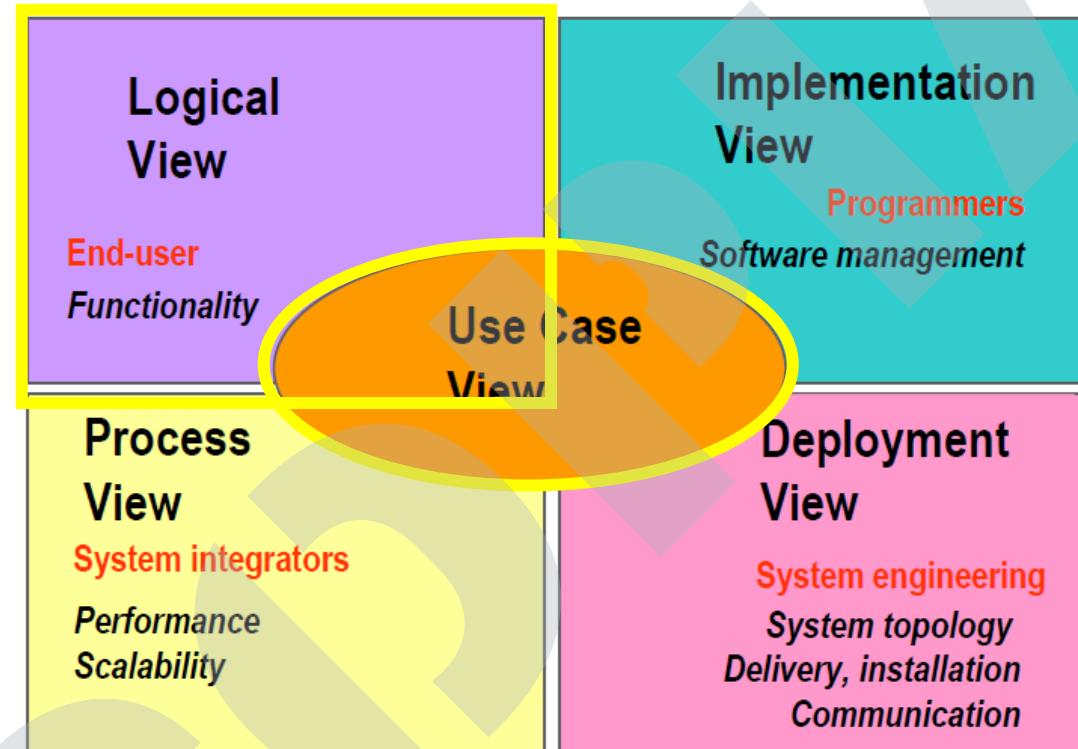
Vistas Arquitectónicas

Vista Lógica

Diseño de Funcionalidades del Sistema

Vista de Procesos

Aspectos de concurrencia y sincronización



Vista de Implementación

Organización de elementos físicos que se implementa en el sistema (código)

Vista de Despliegue

Arquitectura física del sistema

Vista de Casos de Uso

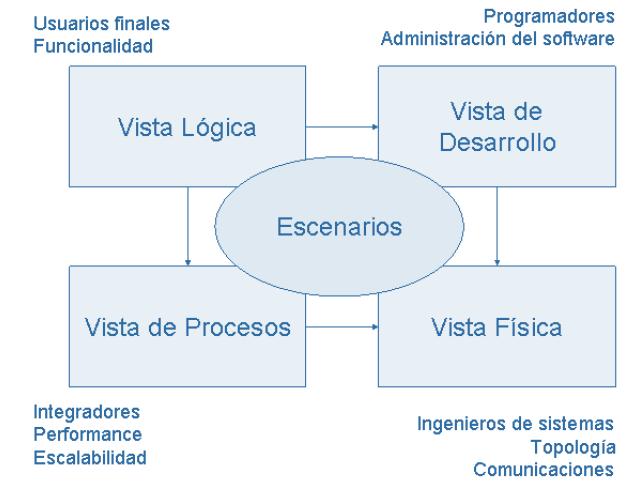
Comportamiento del Sistema percibido por usuarios finales, analistas y encargados de las pruebas



Vista de Casos de Uso

Vista De Caso de Uso o de Escenarios:

- Muestra los requisitos del sistema tal como es percibido por el usuario final, analista y encargado de pruebas. Los CU mostrados tienen relevancia para la arquitectura.
- Uso los diagramas de:
 - Casos de Uso.



Vista de Lógica

Describe las clases y su organización en paquetes y subsistemas y la organización de estos en capas

usa los diagramas de:

- Diagrama de clases
- Diagrama de Estados
- Diagramas de Interacción
- Diagrama de actividades.

Vista de Proceso

En esta vista **se describe las tareas sus interacciones y la asignación de objetos del diseño a las tareas.**

Muestra elementos **relacionados con el desempeño, escalabilidad, concurrencia**

Se usa solo si el sistema tiene alta concurrencia, por lo que es una vista **OPCIONAL**

Modela la Distribución de los procesos e hilos

Emplea los mismos diagramas de las vistas de diseño, pero poniendo atención en las clases activas y los objetos que presentan hilos y procesos.

Vista de Implementación

Modela los componentes y archivos que utilizan para ensamblar y hacer disponible el sistema físico

Contiene visión general del Modelo de Implementación y su organización en módulos en paquetes y capas .

Emplea los diagramas :

Para modelar **aspectos estáticos** : **Diagrama de Componentes**.

Para modelar **aspectos dinámicos**:

Diagrama de Interacción

Diagrama de estados

Diagrama de actividades

Vista de Despliegue.

Modela los nodos de la topología de hardware sobre la que se ejecutan los sistemas.

Para modelar **aspectos estáticos** : **Diagrama de Despliegue**

Para modelar **aspectos dinámicos**.

Diagramas de Interacción

Diagramas de Estados

Diagrama de Actividades

Mapeo del Framework 4 +1 a la propuesta de Arquitectura

Framework 4 +1	Arquitectura
Vista de CU o Escenarios	CU, Restricciones y QoS (Requerimientos no Funcionales)
Vista Lógica	Lógica
Vista de Procesos	Procesos
Vista de Implementación	Implementación, Datos
Vista de Despliegue	Despliegue

Resumen

- **Vista de casos de uso o historias de usuario**

- Audiencia: todas las partes interesadas del sistema, incluidos los usuarios finales.
- Área: describe el conjunto de escenarios y / o casos de uso / o historias de usuario que representan alguna funcionalidad central significativa del sistema. Describe los actores y casos de uso para el sistema, esta vista presenta las necesidades del usuario y se desarrolla más a nivel de diseño para describir flujos y restricciones discretos con más detalle. Este vocabulario de dominio es independiente de cualquier modelo de procesamiento o sintaxis de representación (es decir, XML).
- Artefactos relacionados: modelo de caso de uso, documentos de caso de uso

- **Vista lógica**

- Audiencia: Diseñadores.
- Área: Requisitos funcionales: describe el modelo de objetos del diseño. También describe las realizaciones de caso de uso más importantes y los requisitos comerciales del sistema.
- Artefactos relacionados: modelo de diseño

- **Vista de datos**

- Audiencia: especialistas en datos, administradores de bases de datos
- Área: Persistencia: describe los elementos persistentes arquitectónicamente significativos en el modelo de datos, así como la forma en que los datos fluyen a través del sistema.
- Artefactos relacionados: modelo de datos.

- **Vista de implementación**

- Audiencia: administradores de implementación.
- Área: Topología: describe el mapeo del software en el hardware y muestra los aspectos distribuidos del sistema. Describe las posibles estructuras de implementación, al incluir escenarios de implementación conocidos y anticipados en la arquitectura, permitimos a los implementadores hacer ciertas suposiciones sobre el rendimiento de la red, la interacción del sistema, etc.
- Artefactos relacionados: modelo de implementación.

VISTA DE CASOS DE USO

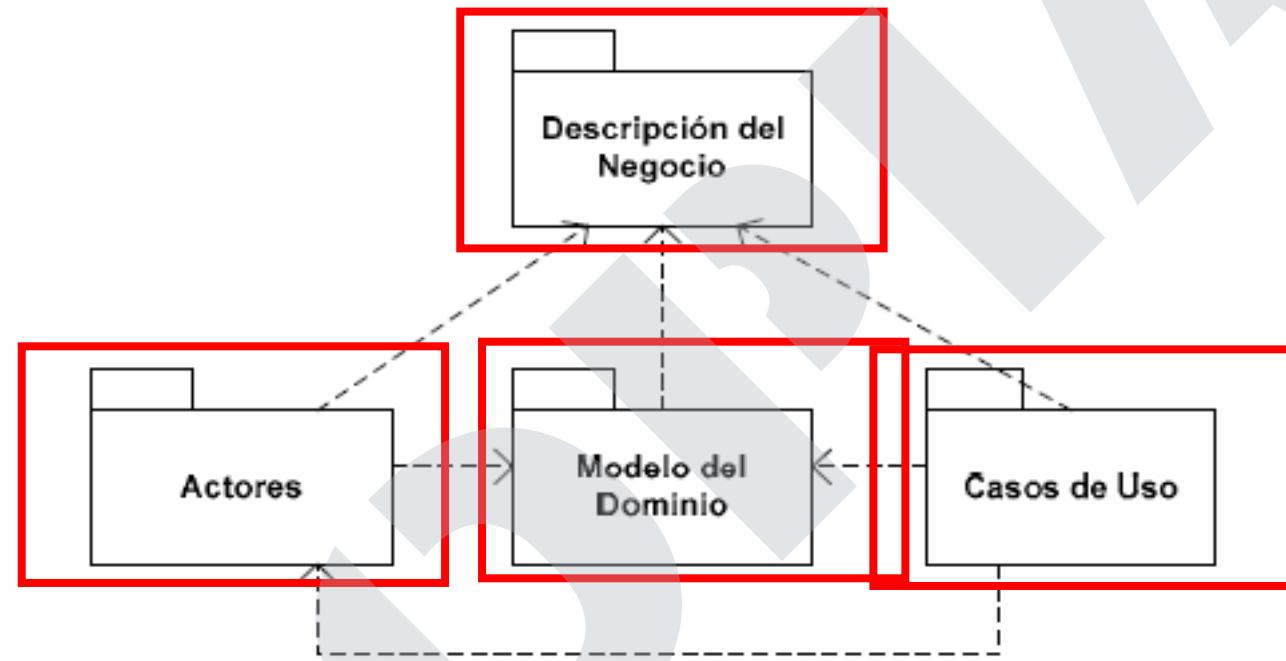
Vista de Casos de Uso

Se toma en cuenta todos **los casos de uso que tienen impacto en la arquitectura.**

Los diagramas de caso de uso **representan la funcionalidad del sistema.**

Las Restricciones y los requerimientos no funcionales.

Organización de la Vista de CU



En esta vista se presenta primero una sección de la descripción del Negocio, se presenta el modelo del Dominio los actores y los Cu relevantes para la Arquitectura.

Descripción del Negocio

Primero se describe el Sistema y se identifica los procesos más importantes. La descripción del Sistema es a través de un texto, preciso y claro ejemplo:

- Sistema de Gestión Hotelera:
Una cadena Hotelera desea automatizar los servicios prestados por sus Hoteles.....
- Identificación de los PN (procesos de Negocio).
Se determina los PN relativos al sistema, por ejemplo en el caso de la Gestión Hotelera:
 - PN Gerenciamiento de la Cadena Hotelera.
 - Reserva de Habitaciones.
 - Check Out y Facturación.
 - Consultas estadísticas.

Descripción del Negocio

Se determina los PN de mas **impacto en la Arquitectura**, y se empieza la descripción de la Arquitectura por el PN de mayor impacto.

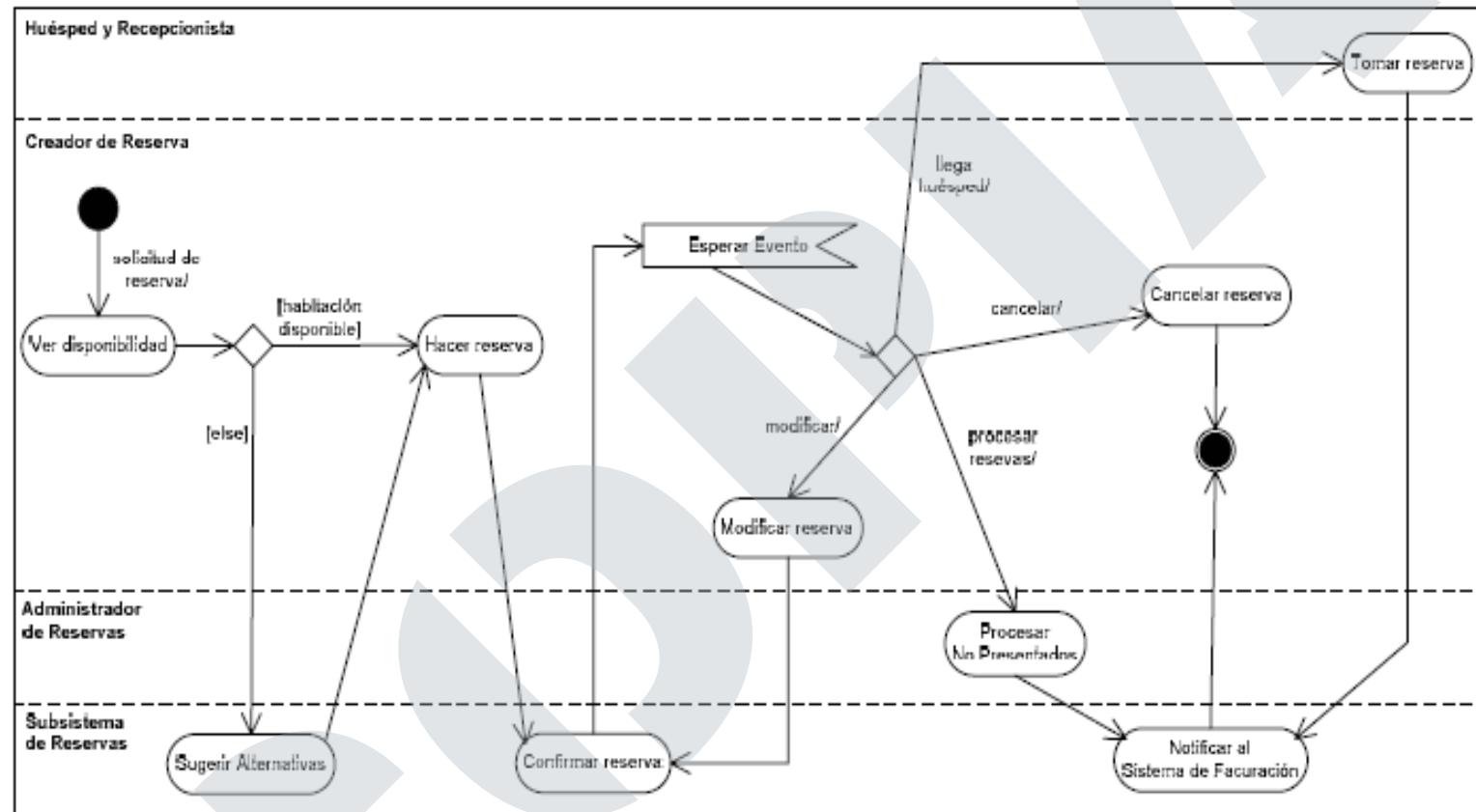
Por ejemplo:

En nuestro ejemplo tenemos 4 PN Gerenciamiento de la Cadena (PN1), Reserva de Habitaciones (PN2), Check Out y Facturación (PN3), Consulta Estadística (PN4).

Los PN PN1 y PN4, son procesos genéricos y pueden enmarcarse en otros subsistemas de gestión hotelera.

Los Procesos PN2 y PN3 son **el corazón del sistema**, estos procesos exigen buena performance, las reservas hechas por internet deben durar a lo mas 5 segundos, cuando hace la reserva en el hotel o telefónicamente debe durar 3 minutos.

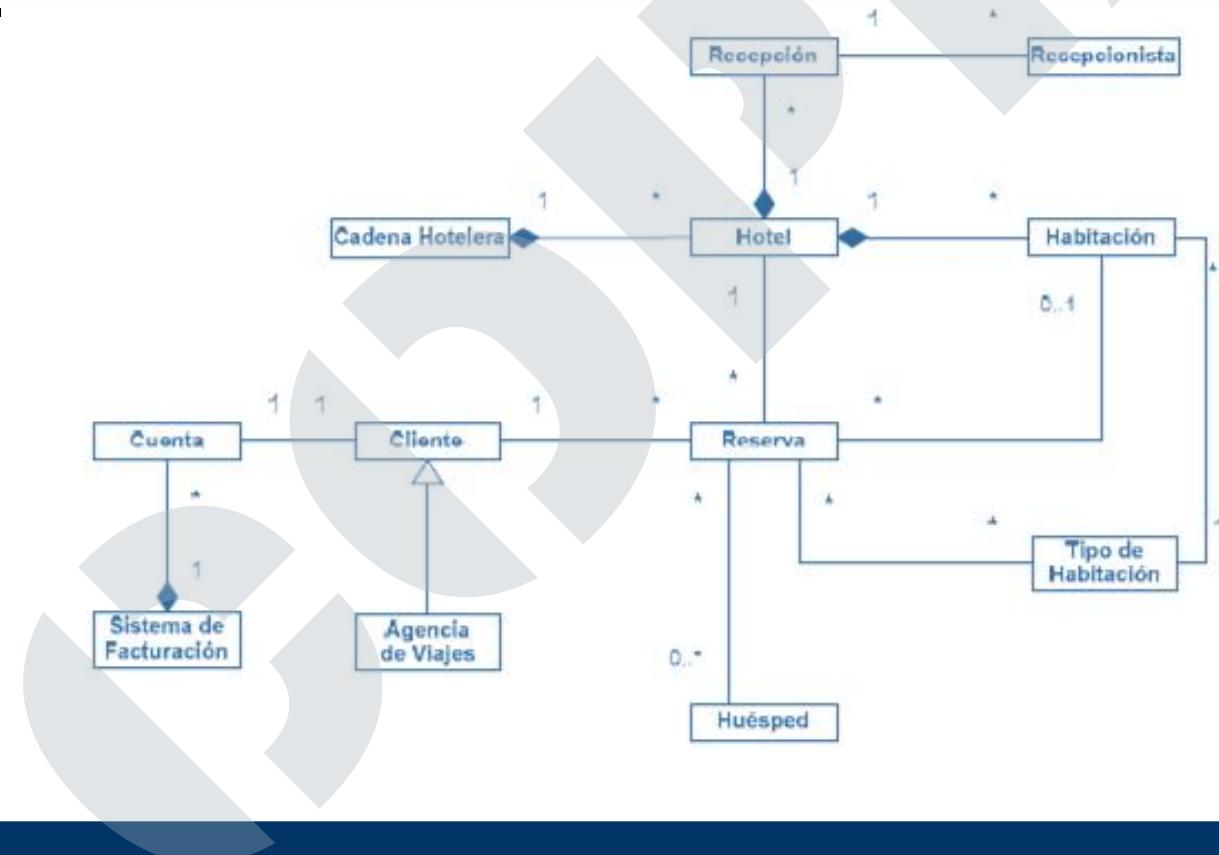
Descripción del Negocio.



Ambos procesos tienen impacto en la Arquitectura, y se representa los PN con diagramas de Actividades. Ejemplo el PN2 con su diagrama de Actividades.

Modelo del Dominio

El modelo de dominio incluye aquel vocabulario del dominio **significativo desde el punto de vista de la arquitectura**, aquel que ayude al entendimiento de la misma.

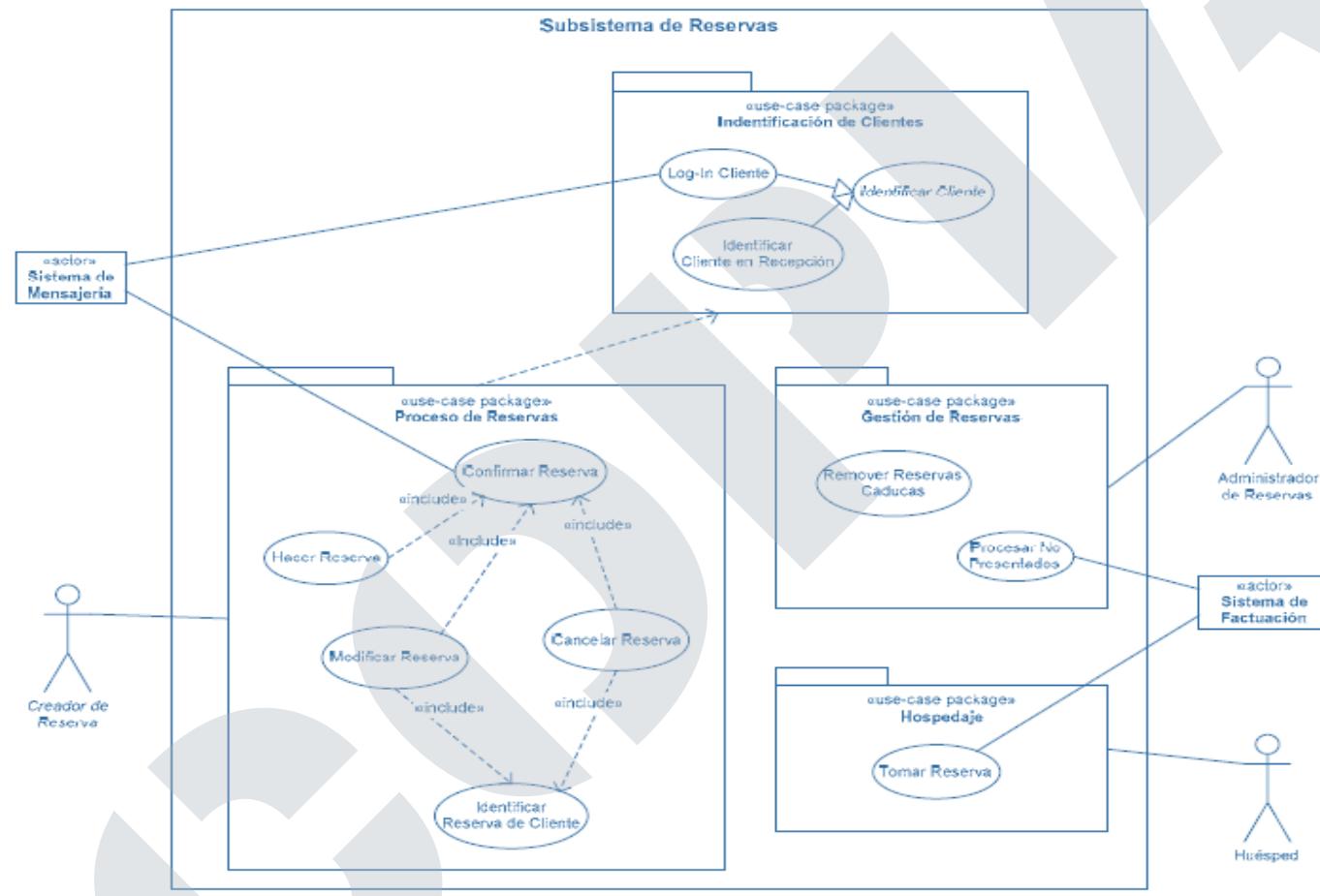


Actores



Los actores que se muestran son los que **interactúan con el sistema**, en el ejemplo .

CU relevantes para la Arquitectura



Descripción de los CU Relevantes.

Nombre	Hacer Reserva (CU1)
Actores	Creador de Reserva, Sistema de Mensajería
Actividades	Ver Disponibilidad, Sugerir Alternativas, Hacer Reserva, Confirmar Reserva
Sinopsis	Este caso de uso comienza cuando el Creador de Reserva solicita crear una reserva. El sistema chequea la disponibilidad de una habitación en un hotel solicitado. Si hay disponibilidad el Sistema hace la reserva y le confirma la misma al cliente. Si no hay disponible una habitación, el sistema sugiere hoteles alternativos.
Curso Típico de Eventos	
	<ol style="list-style-type: none">1. Incluir Identificar Cliente (CUS/CU9).2. Creador de Reserva indica hotel (en caso de estar en la Recepción de un hotel esta información se provee automáticamente), tipo de habitación y duración de la estadia.3. Sistema confirma disponibilidad.4. Sistema registra la reserva.5. Incluir Confirmar Reserva (CU10).
Extensiones	
	<ol style="list-style-type: none">1a. No existe el cliente:<ol style="list-style-type: none">1. Incluir Alta de Cliente.2. Resúme 2.3a. No hay disponibilidad:<ol style="list-style-type: none">1. Sistema busca disponibilidad en otros hoteles.<ol style="list-style-type: none">1a. No hay disponibilidad en ningún hotel:<ol style="list-style-type: none">1. Sistema notifica a Creador de Reserva.2. Resúme 2.2. Creador de Reserva indica un hotel de su conveniencia.2a. Creador de Reserva prefiere cambiar datos de la reserva:<ol style="list-style-type: none">1. Resúme 2.3. Resúme 4.

Se procede a describir todos los CU relevantes para la Arquitectura.

Interfaces de Usuario

Ingrese los datos de la reserva



Cliente:

ID	<input type="text" value="1"/>
Nombre	<input type="text" value="Daniel Perovich"/>
Usuario	<input type="text" value="dperovich"/>
E-mail	<input type="text" value="perovich@fing.edu.uy"/>
Teléfono	<input type="text" value="+598 (2) 711 4244, 114"/>
Fax	<input type="text" value="+598 (2) 711 0469"/>

Ciudad:

 ▼

Hotel:

 ▼

Tipo de Habitación:

 ▼

Check-in

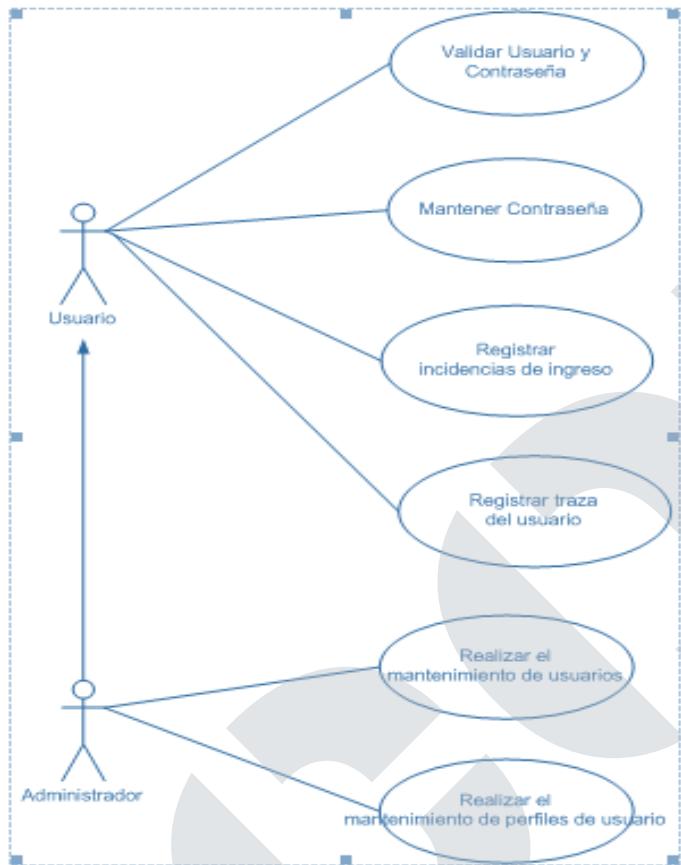
May Junio de 2003 Jul						
L	M	M	J	V	S	D
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

Check-out

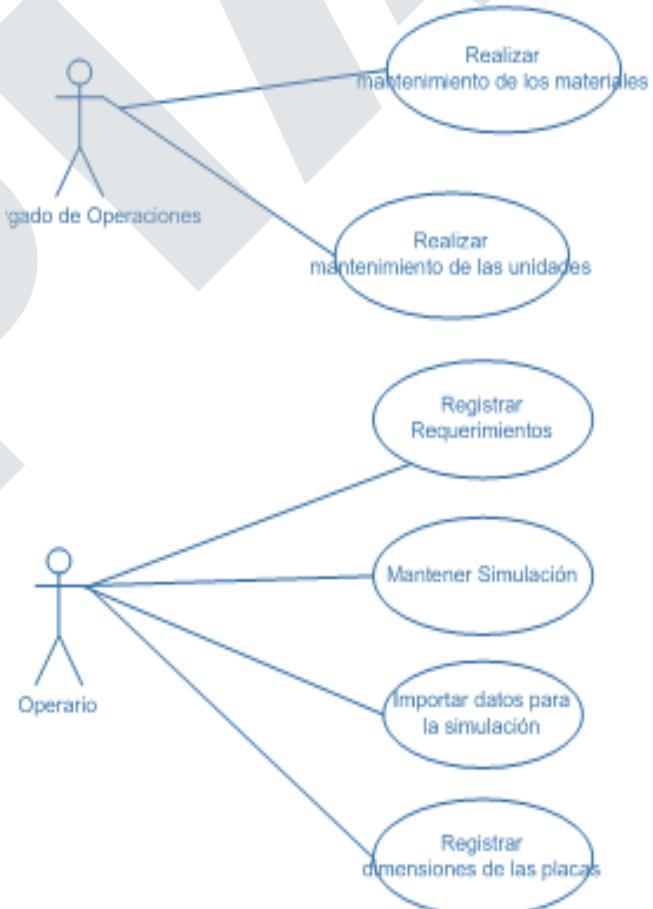
May Junio de 2003 Jul						
Lun	Mar	Mié	Jue	Vie	Sáb	Dom
26	27	28	29	30	31	1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	1	2	3	4	5	6

Se presenta las pantallas de los Cu **relevantes para la arquitectura**

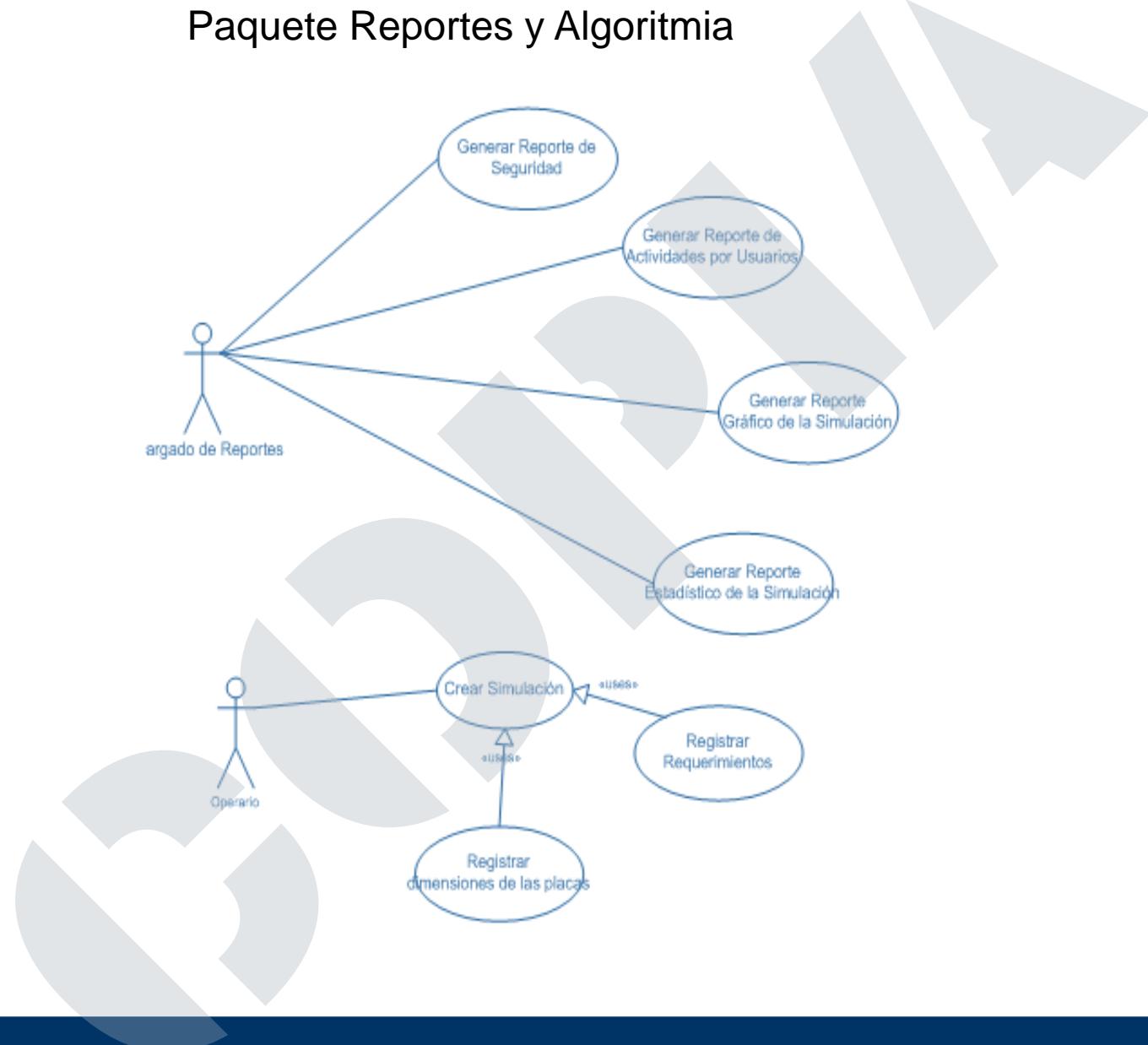
Paquete de Seguridad



Paquete de Maestros



Paquete Reportes y Algoritmia



Vista CU: Sección Restricciones

En esta sección se presentan las **restricciones normativas, de estándares y de tecnologías**.

Normativas:

Existen restricciones normativas de organismos gubernamentales / no gubernamentales.

Como los licenciamientos, formas de pago, registro impositivo, etc...

Estándares:

Como pueden ser el UML, Interfaz Web, Web Services

Vista CU: Sección Restricciones

Tecnología:

Se debe indicar la plataforma tecnológica en que se desarrollara el sistema ej.” el sistema será desarrollado en plataforma Microsoft .Net 2010. ADO.Net, ASPE Net, el leguaje es C# .Net..”

Sistemas Existentes:

Se debe tener en cuanta los sistemas heredados o aquellos con los que el SI tenga relación. Ej. “ el sistema se relacionara con un Software de Facturación ”
...”

Soporte:

Se debe indicar el tipo de mantenimiento a usar (evolutivo, adaptativo) , puede ser ambos, indicando las partes del SI a los cuales se aplicara.

Vista CU: Sección QoS

En esta sección se describe los requerimientos no funcionales del sistema, dentro de las categorías de Usabilidad, confiabilidad y performance descritas en el FURPS+.

Usabilidad: “la interfaz de usuario orientada a web , para los actores humanos disminuye la necesidad de capacitación ...”

Confiabilidad:” el sub sistema de reserva no debe fallar , pues es critico para el sistema..”

Performance: “El sub sistema de reserva tiene fuertes restricciones en el CU1 y CU4...”

VISTA LÓGICA



Vista Lógica

Permite **organizar el modelo de diseño en piezas mas manejables.**
Subsistemas e Interfaces.

No todos los subsistemas se implementan.

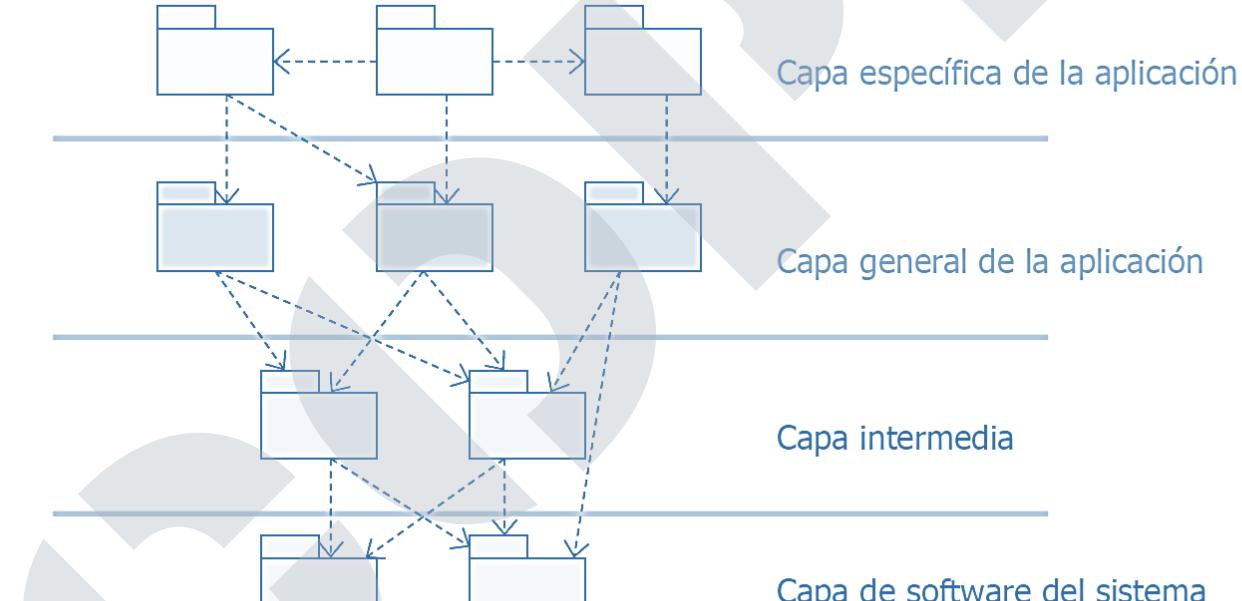
Identificación de Subsistemas de la aplicación:

Se hacen a partir de los paquetes del análisis, los que se refinan si es necesario.

Vista Lógica

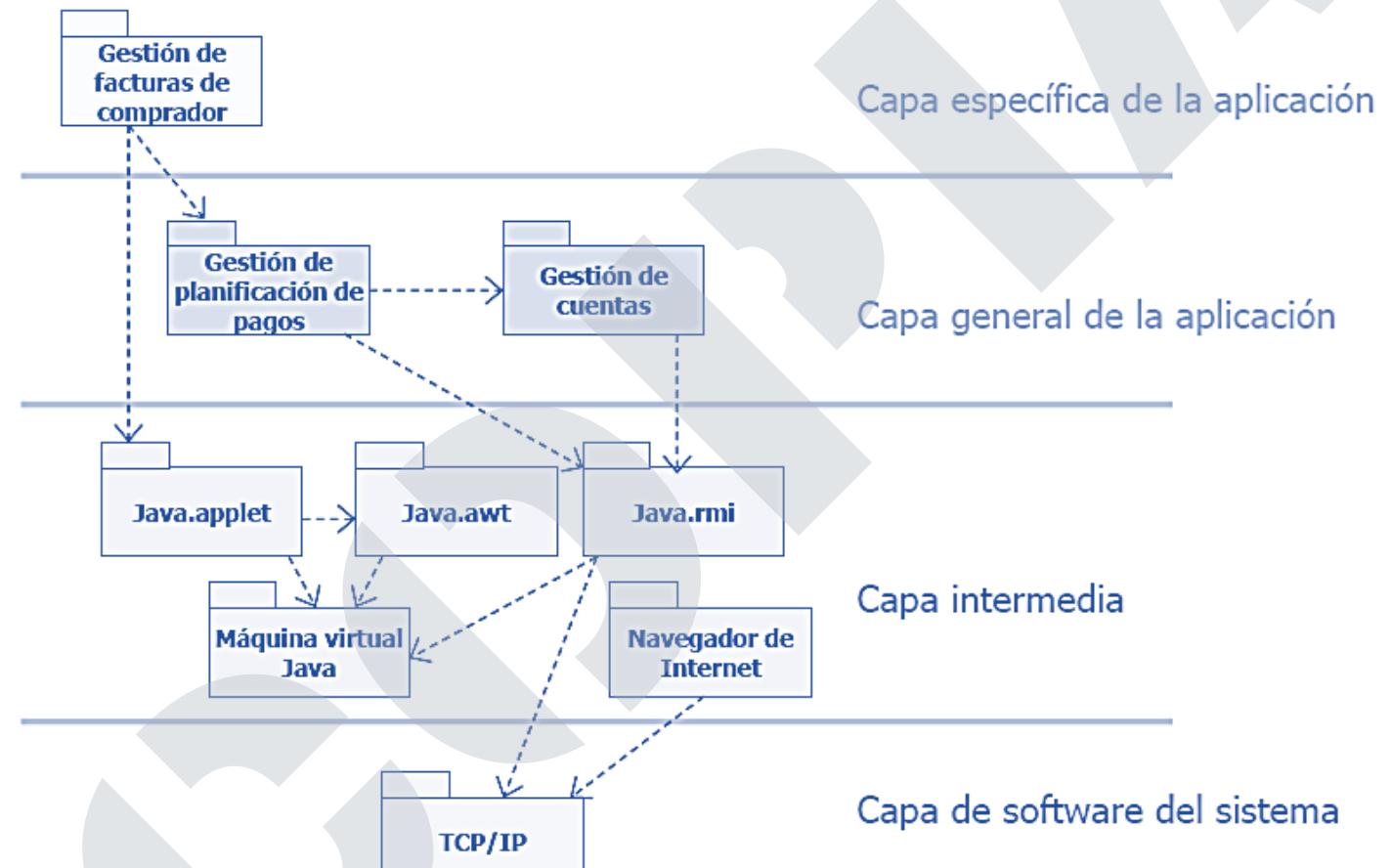
Identificación de Subsistemas de Aplicación

Se identifican a partir de los paquetes del análisis, pero puede ser necesario un refinamiento para tratar temas relativos al diseño, implementación y distribución



Vista Lógica

Identificación de Subsistemas de Aplicación



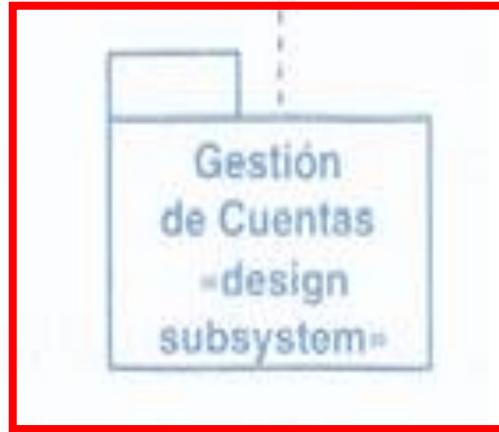
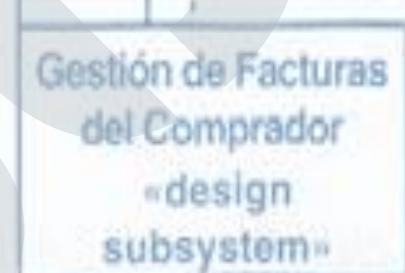
Vista Lógica

Identificación de Subsistemas de Aplicación

Modelo de análisis

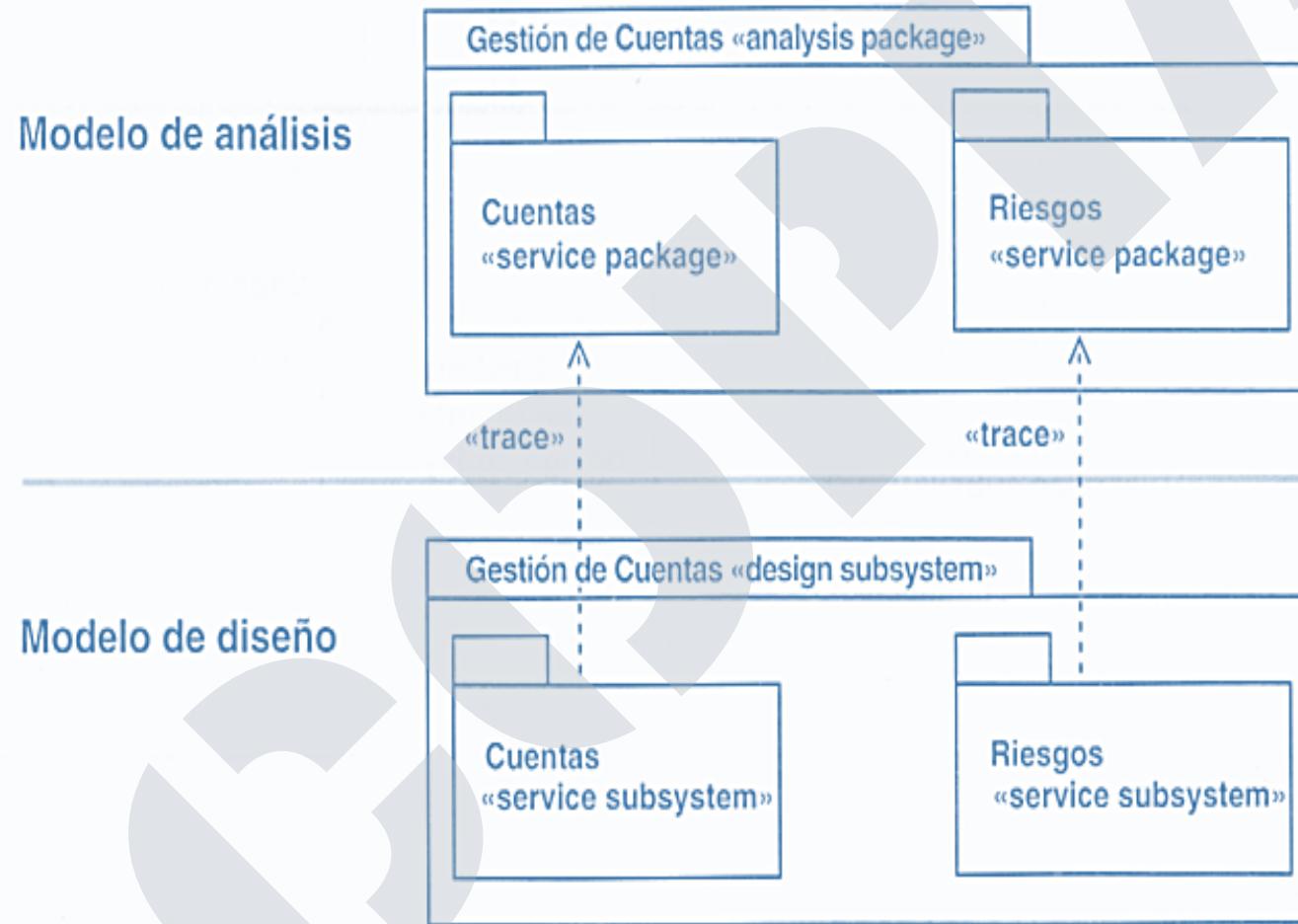


Modelo de diseño



Vista Lógica

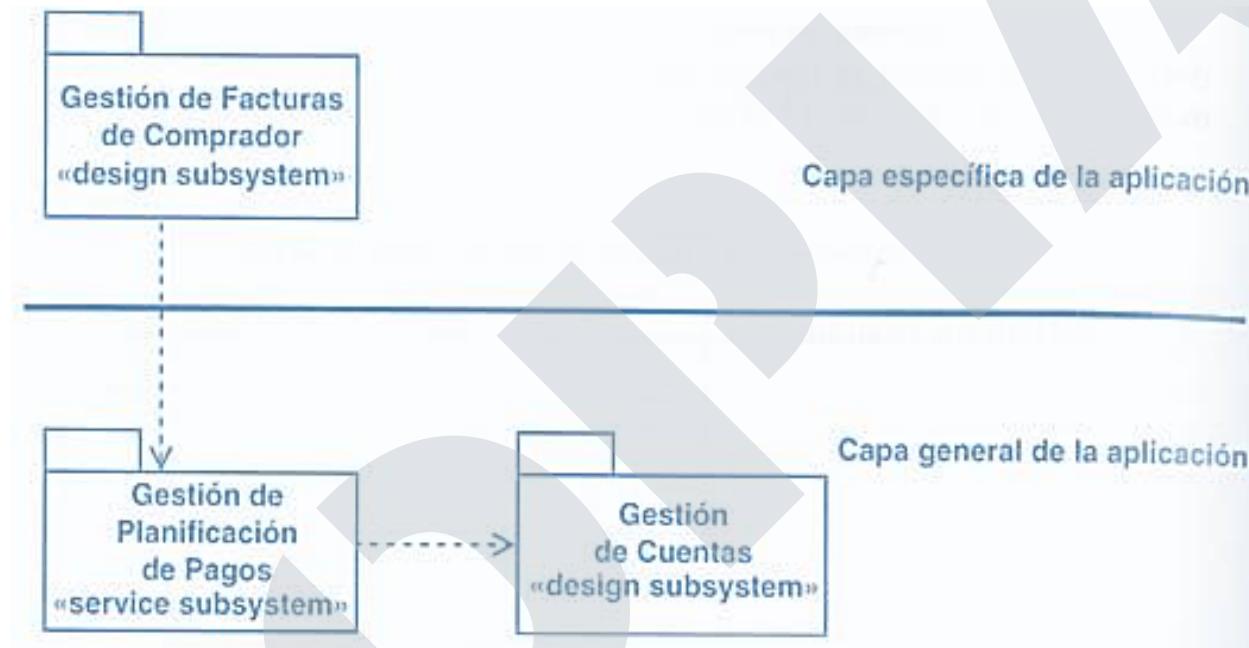
Identificación de Subsistemas de Aplicación



Refinamiento de los subsistemas para tratar funcionalidades compartidas

Se considera implementar todos los paquetes de servicio para el pago de facturas en el subsistema “Gestión de Facturas de Comprador”, donde parecía encajar tras el análisis de los CU relacionados con la gestión de las facturas

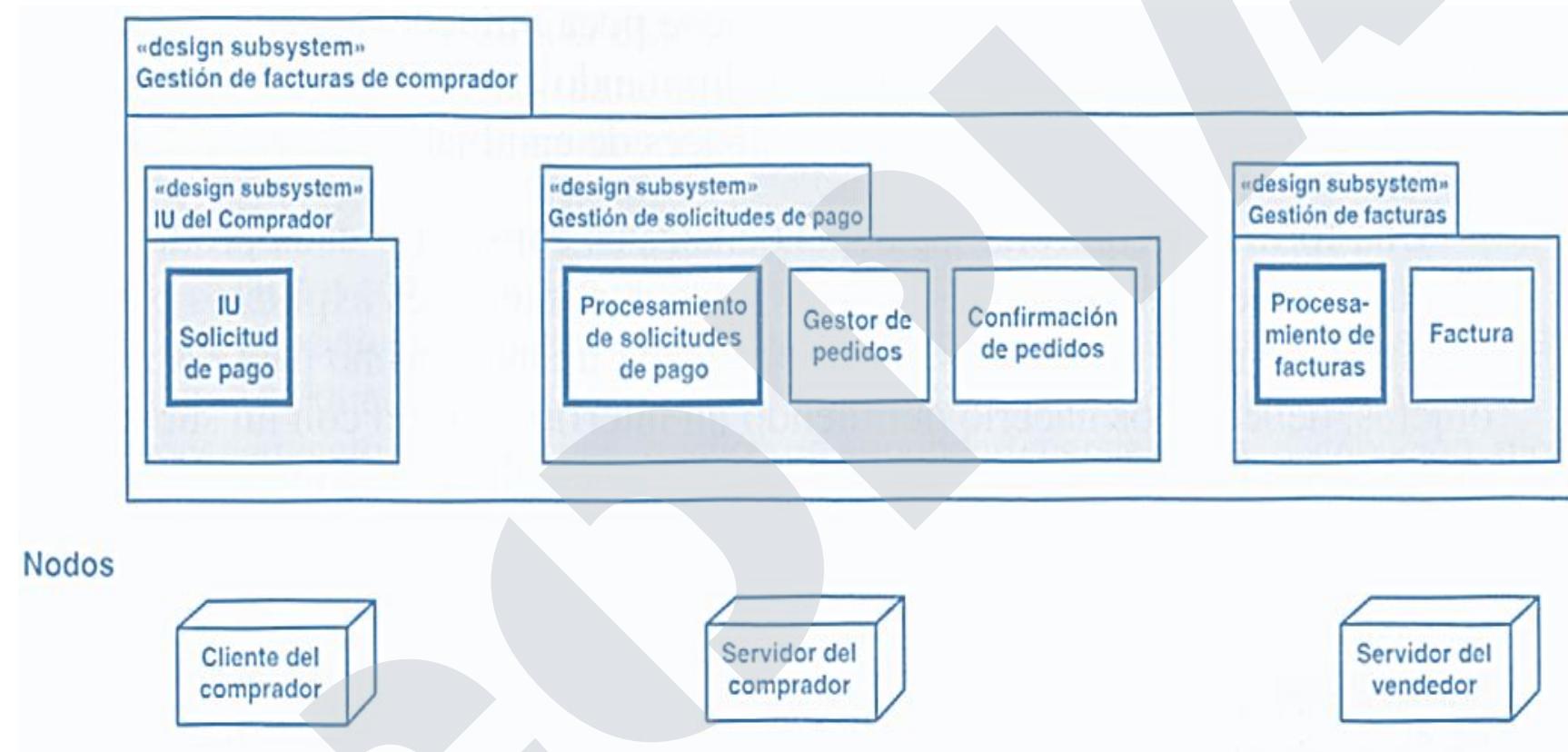
Refinamiento de los subsistemas para tratar funcionalidades compartidas



Los desarrolladores **agrupan la funcionalidad** de los pagos en un subsistema de servicio llamado “Gestión de Planificación de Pagos”.

El Sub Sistema “Gestión de facturas de comprador” **usa este nuevo** subsistema para planificar facturas. Cuando se hace efectivo el pago el subsistema “Gestión de Planificación de pagos” usa el subsistema “Gestión de Cuenta” para transferir el dinero

Distribución de un subsistema entre los nodos : el subsistema Gestión de Facturas de Comprador debe distribuirse en varios nodos

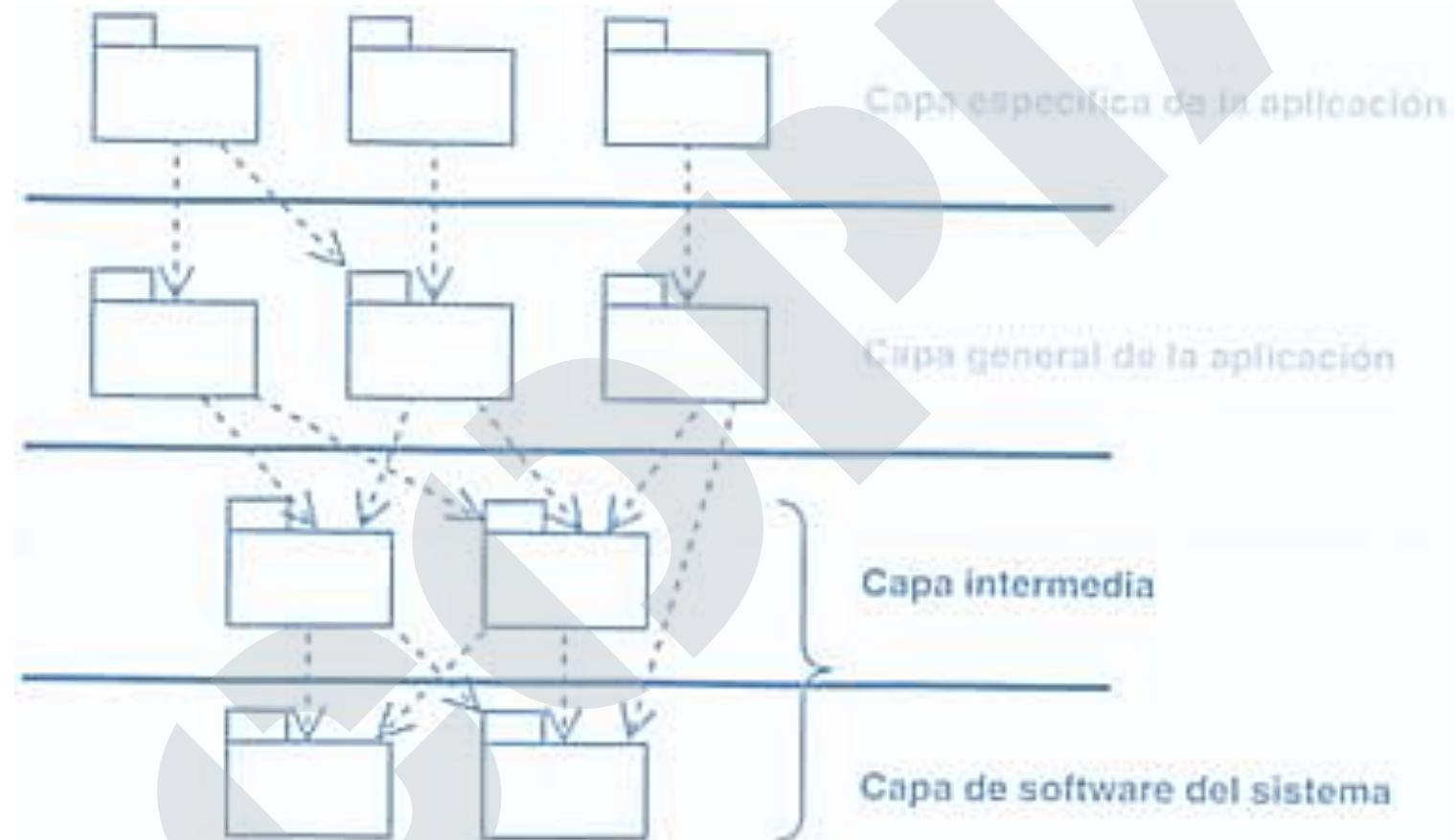


Se descompone el subsistema en tres: "IU de Comprador", "Gestión de Solicitud de Pago", "Gestión de Factura". Sus componentes pueden distribuirse en los nodos : "Cliente del Comprador", "Servidor del Comprador", "Servidor del Vendedor"

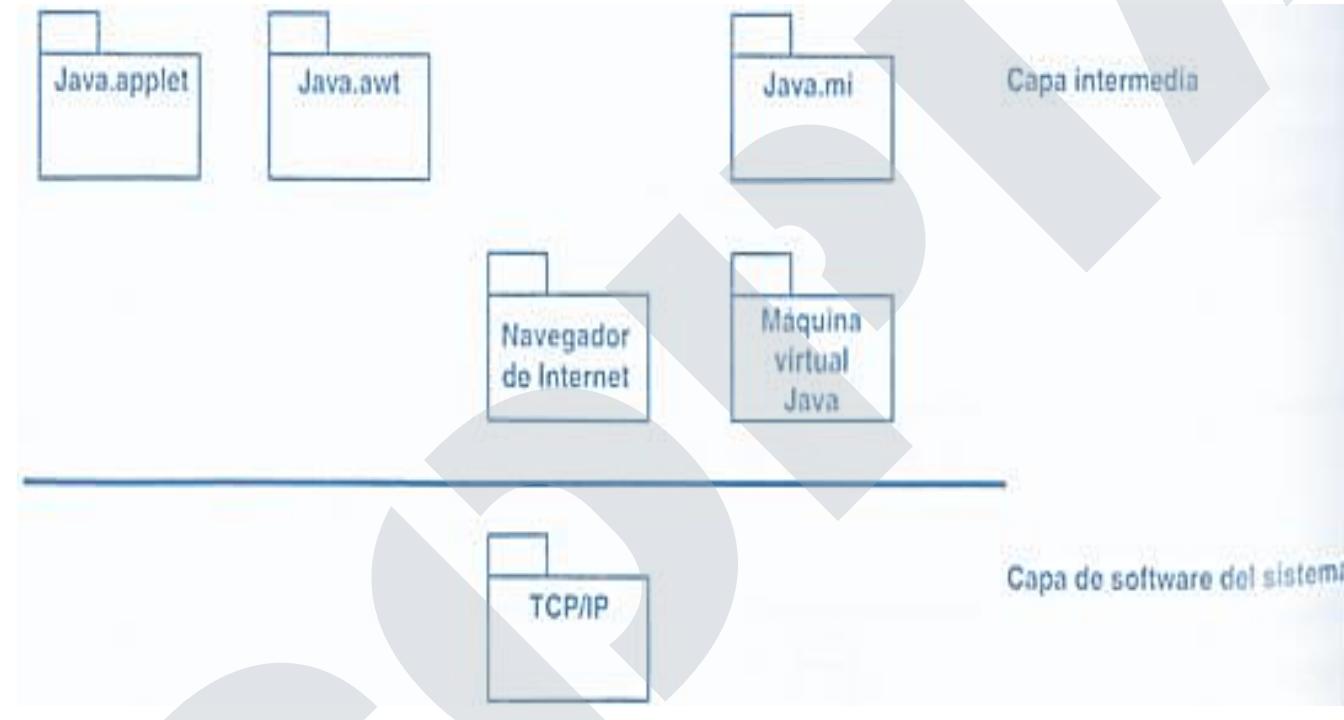
Identificación de subsistemas intermedios y de Software del sistema

El SW del sistema y la capa intermedia constituye los cimientos de un sistema, ya que toda la funcionalidad descansa sobre SO, SGBD, SW de comunicaciones, tecnología de distribución de objetos, kit de diseño de IGU

Refinamiento de los subsistemas para tratar funcionalidades compartidas



Uso de Java para construir la capa intermedia

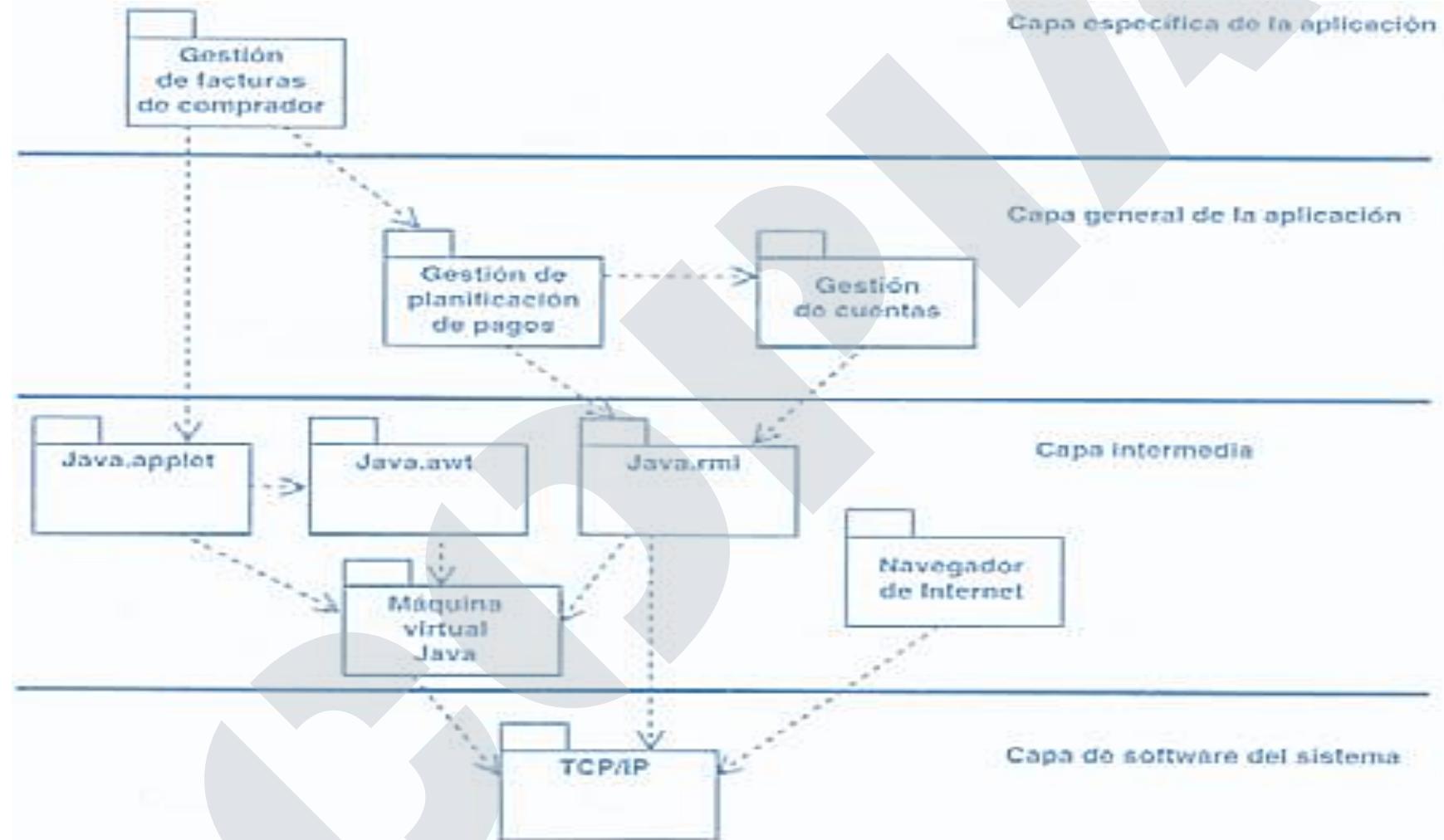


Se implementa mediante **middleware** con paquetes Java AWT, RMI y Applet. Estos paquetes se representan como subsistemas que se ejecutan sobre la maquina virtual Java. Se usa un navegador Internet.
A bajo nivel el sistema se construirá sobre SW de sistema, como protocolos TCP/IP

Definición de dependencia entre subsistemas

- Si sus contenidos tienen relación unos con otros.
- Se consideran las dependencias de los paquetes de análisis que se corresponden con las del diseño.
- Si se consideran interfaces entre subsistemas, las dependencias deberán ir hacia las interfaces.

Dependencias y capas iniciales del sistema ejemplo.



Identificación de Interfaces entre capas

Las interfaces definen operaciones que son accesibles desde fuera del subsistema. **Esta interfaces las proporcionan clases o subsistemas.**

Cuando un subsistema tiene una dependencia que apunta hacia el, es probable que **deba proporcionar una interfaz**

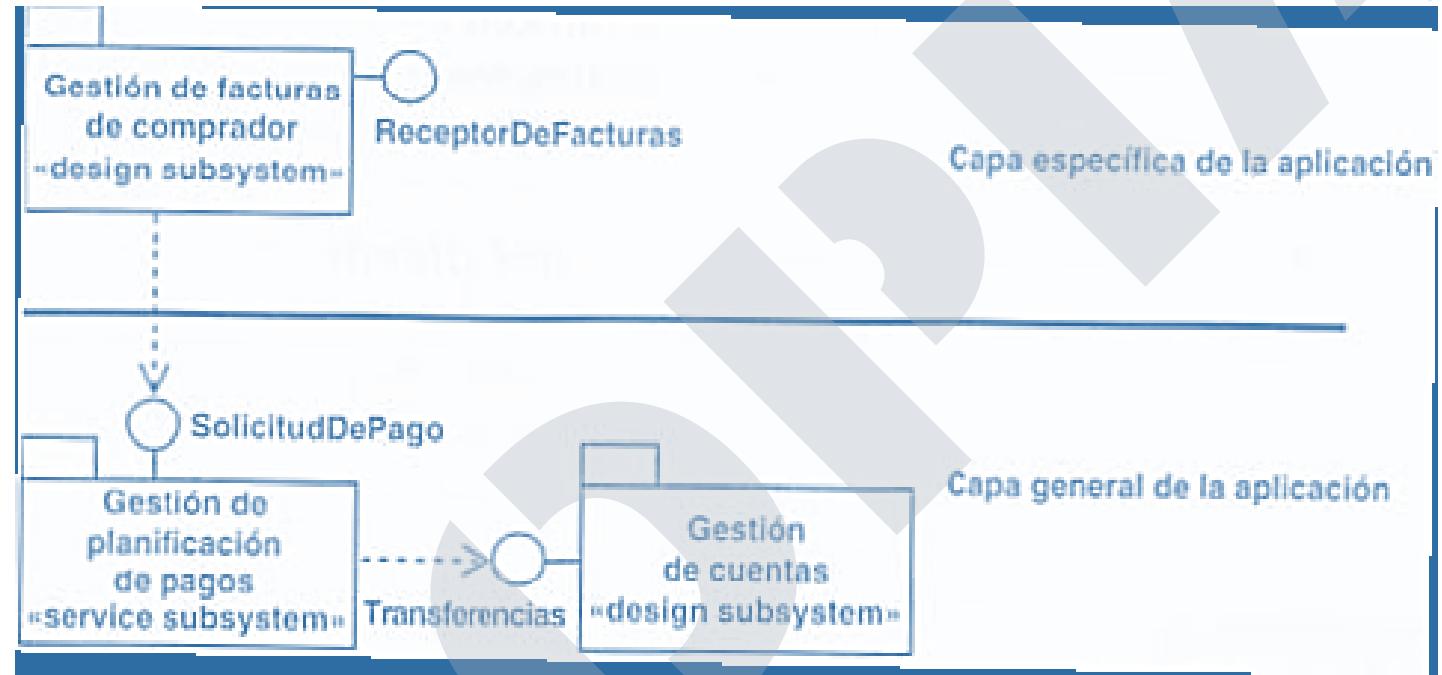
Consideraciones:

- Las interfaces permiten refinar las dependencias entre los subsistemas.
- En las capas inferiores la identificación de interfaces es mas sencilla.
- Es necesario identificar las operaciones de la interfaces.

Identificación de interfaces candidatas a partir del modelo de análisis



Identificación de interfaces candidatas a partir del modelo de análisis



El subsistema “Gestión de Cuentas” proporciona la interfaz “Transferencias”, para transferir dinero entre cuentas.

El Subsistema “Gestión de Planificación de Pagos” proporciona la interfaz “Solicitud de Pago” que es para planificar pagos.

El subsistema “Gestión de Facturas del Comprador” proporciona la interfaz “Receptor de Facturas” para recibir nuevas facturas

Vista Lógica

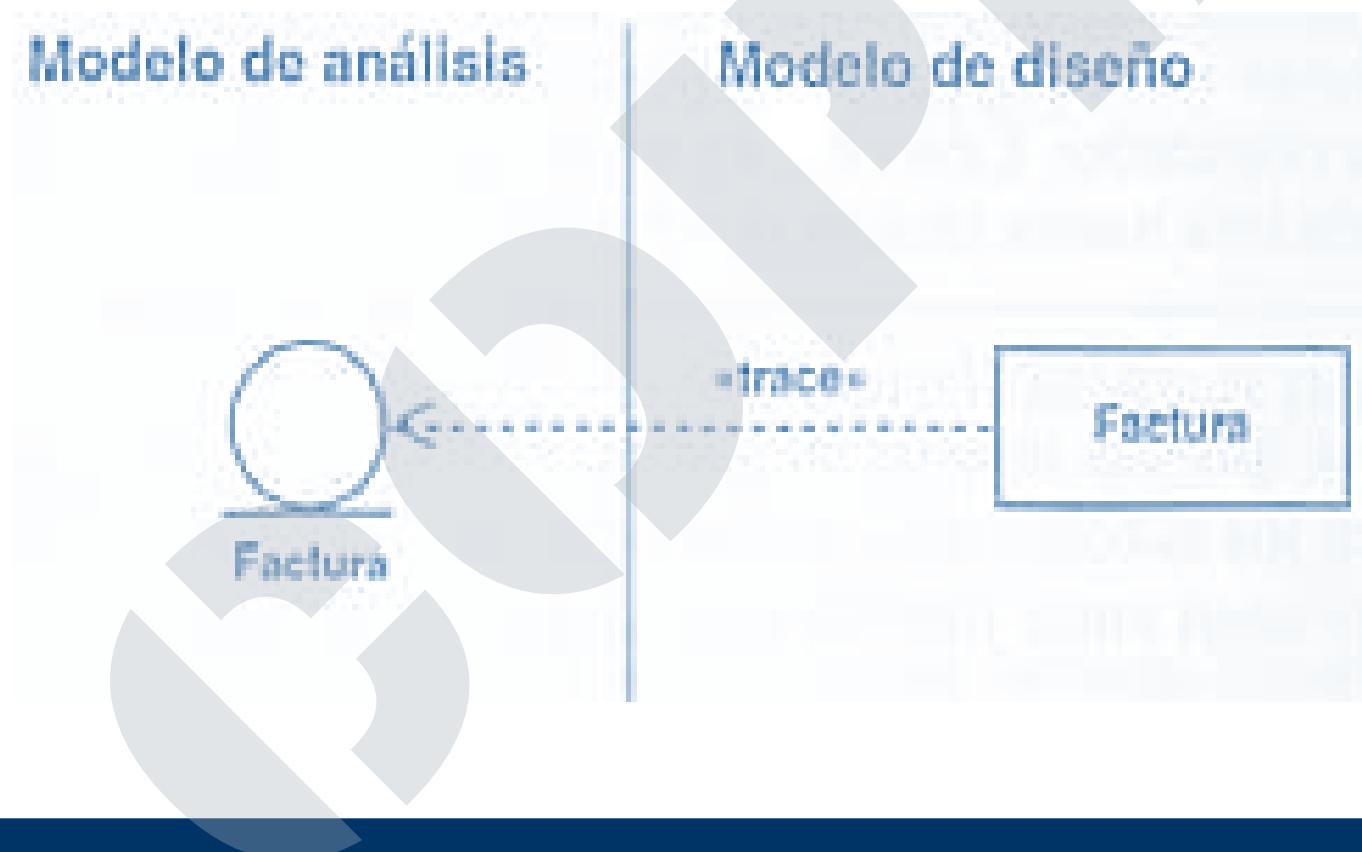
Actividad 2 : Identificación de Clases de Diseño

- Estudia las clases de análisis que participan en la realización de CU análisis. Identifica las clases de diseño que poseen una traza hacia esa clase.
- Estudia requisitos especiales de la realización de CU análisis. Identifica clases de diseño que lo realizan
- Identifica las clases de diseño que faltan.
- Las clases de diseño se recogen en un diagrama de clases .

Vista Lógica

Actividad 2 : Identificación de Clases de Diseño

La clase de diseño **factura** se esboza a partir de la clase entidad de análisis **factura**



Vista Lógica

Actividad 2 : Identificación de Clases de Diseño

Artefacto: Clase del Diseño

Una clase de diseño es una abstracción sin costuras con una clase o construcción similar en la implementación del sistema.

Se especifica en el lenguaje de implementación (Java, C#, etc).

Se especifica visibilidad de atributos y operaciones.

Se implementan las relaciones de generalización via herencia del lenguaje de programación.

Se implementan las relaciones de asociación y agregación via atributos de las clases que implementan las referencias.

Se especifican los métodos con la sintaxis del lenguaje de programación.

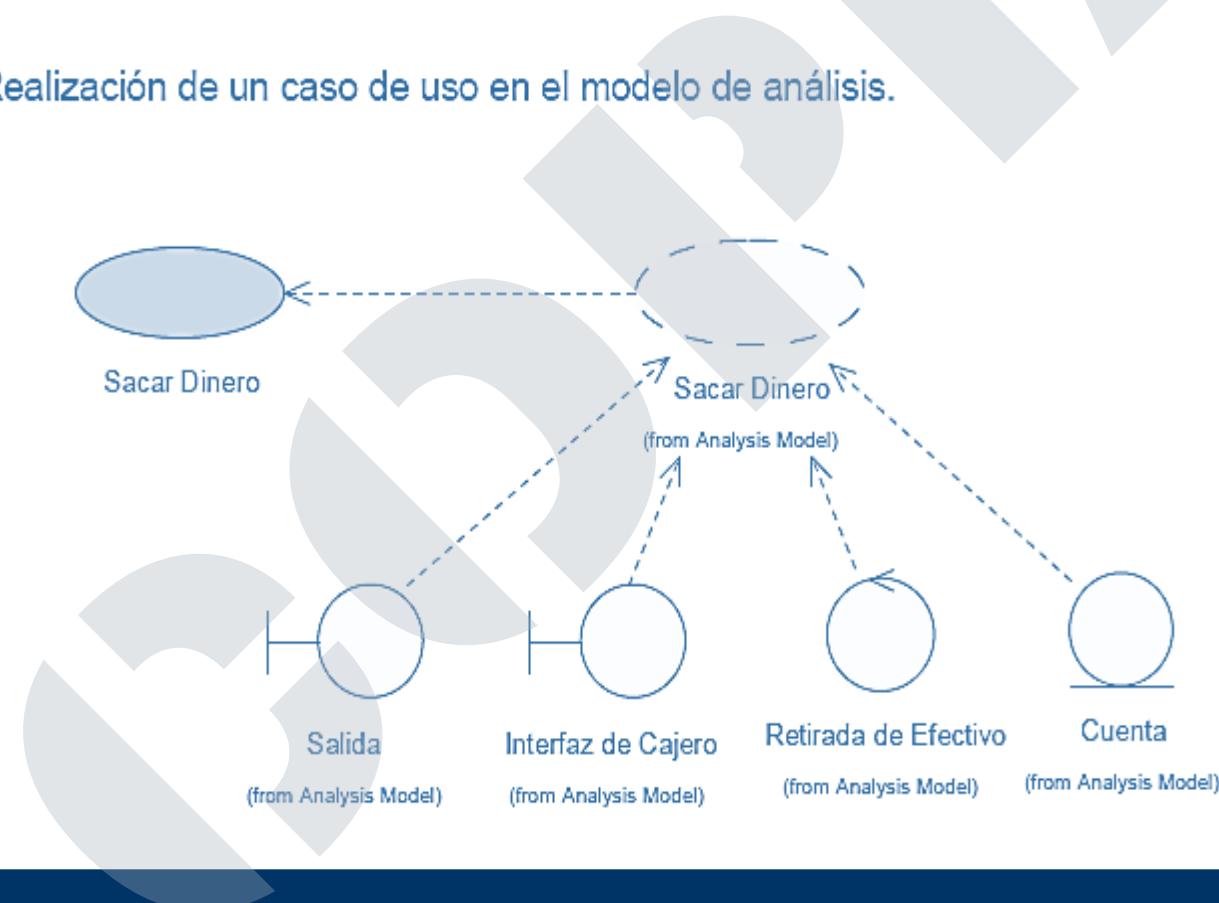
Se usan estereotipos que se corresponden con el lenguaje de programación, por ejemplo en VB se puede utilizar <<class module>>, <<form>>, etc.

Vista Lógica

Actividad 2 : Identificación de Clases de Diseño

En el modelo de Análisis desde el CU se construye la Realización de CU Análisis

Ej.: Realización de un caso de uso en el modelo de análisis.

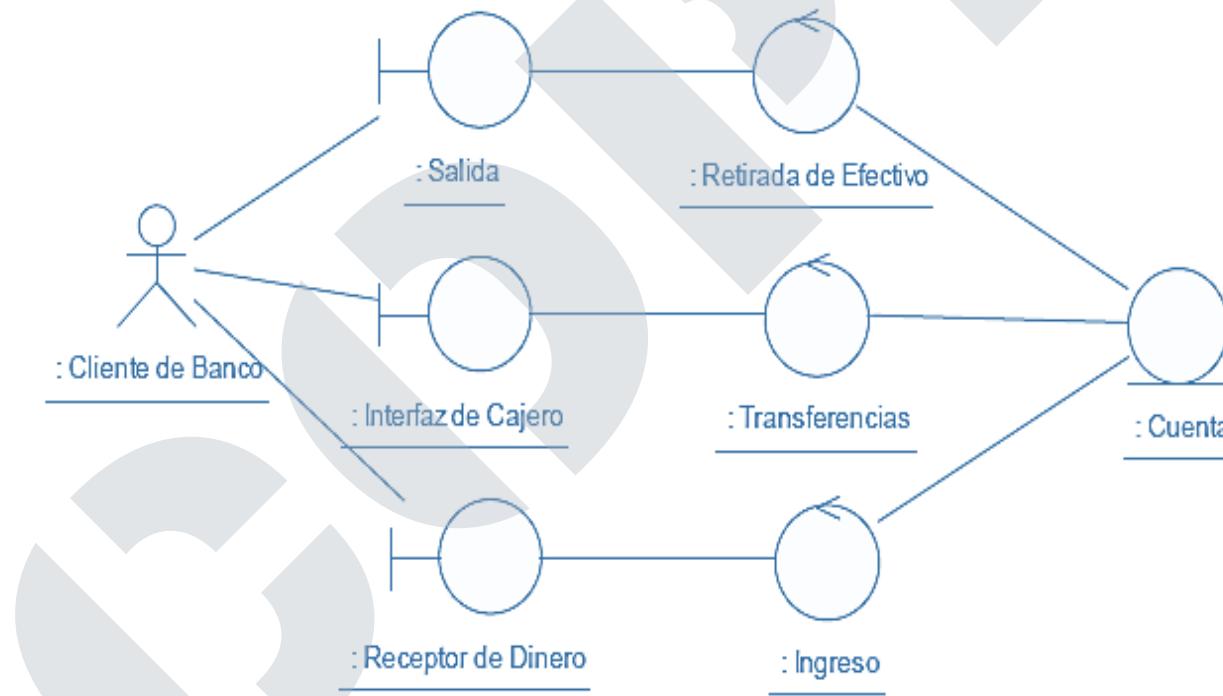


Vista Lógica

Actividad 2 : Identificación de Clases de Diseño

Una clase del Análisis puede participar en varias realizaciones de CU análisis

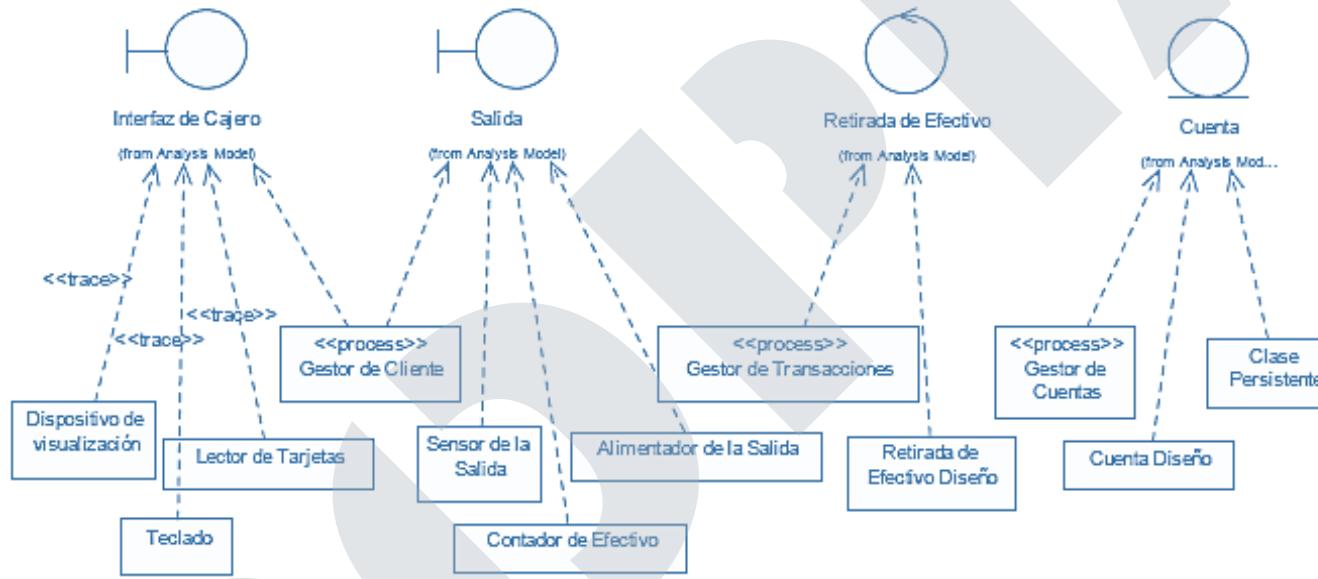
Ej.: Una clase puede participar en varias realizaciones.



Vista Lógica

Actividad 2 : Identificación de Clases de Diseño

Ej.: Las clases del diseño refinan las clases del análisis.



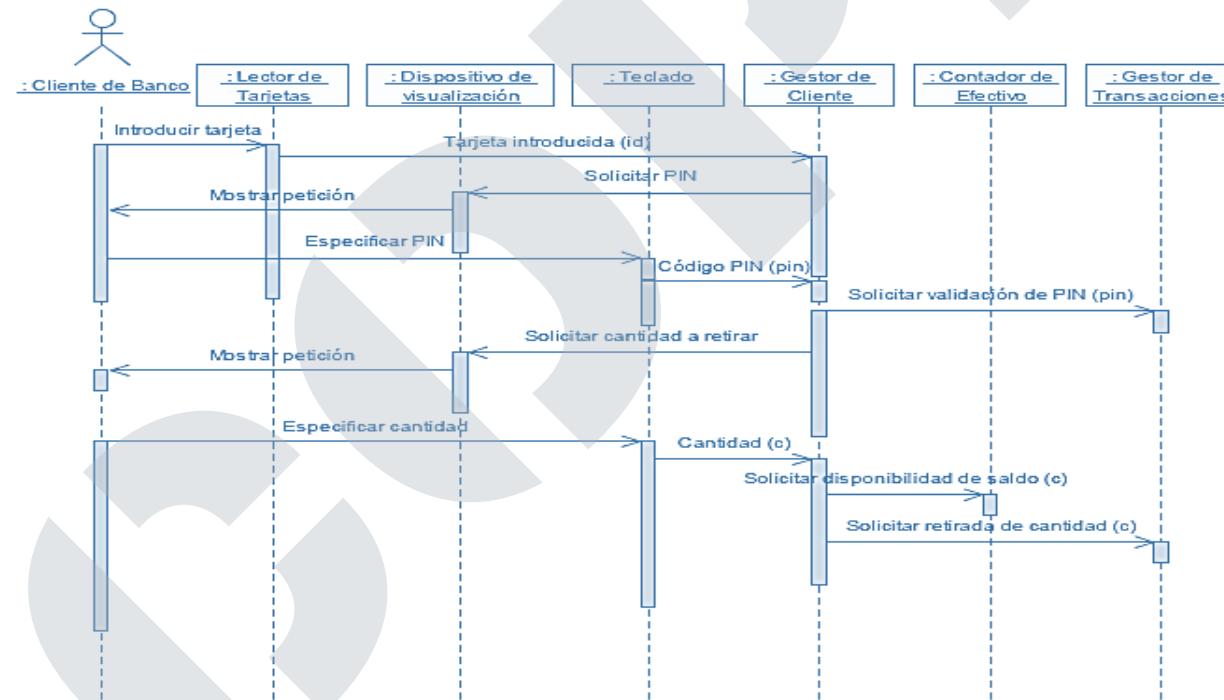
La mayoría de las clases de diseño normalmente tienen una sola traza a una clase de análisis. Esto es habitual para las clases de diseño que son específicas de la aplicación, diseñadas para dar soporte a una aplicación o a un conjunto de ellas.

Vista Lógica

Actividad 2 : Identificación de Clases de Diseño

En el Diseño se identifica las interacciones entre los objetos del diseño, esto se hace con un diagrama de secuencias.
Llamado Realización de CU Diseño

Ej.: Diagrama de secuencia para representar la realización del caso de uso Sacar Dinero en el modelo de diseño.



Agrupación de las clases en Subsistemas

- Es **imposible usar solo clases** para realizar los CU en un sistema grande.
- Es preciso **agrupar las clases** en Subsistemas.
- Un subsistema es una **agrupación útil** de clases o de otro subsistema.
- Los **subsistemas de bajo nivel** se denominan de **servicios**, que son una unidad manejable de funcionalidad opcional
- Los subsistema pueden diseñarse de forma **Ascendente** (se hace a partir de clases ya identificadas) o **Descendente** (se identifica el Subsistema antes que las clases)

Ejemplo:

En un sistema de CA, las clases se agrupan en 3 subsistemas:

Subsistema de Interfaz del CA:

- Lector de Tarjeta
- Dispositivo de visualización
- Teclado
- Alimentador de Salida
- Sensor de salida
- Contador de efectivo

Gestor de Cliente

- **Subsistema Gestión de Transacciones:**

- Gestión de Transacciones
- *Subsistema de Servicios Gestión de Retirada de Efectivo*
 - Retirada de efectivo

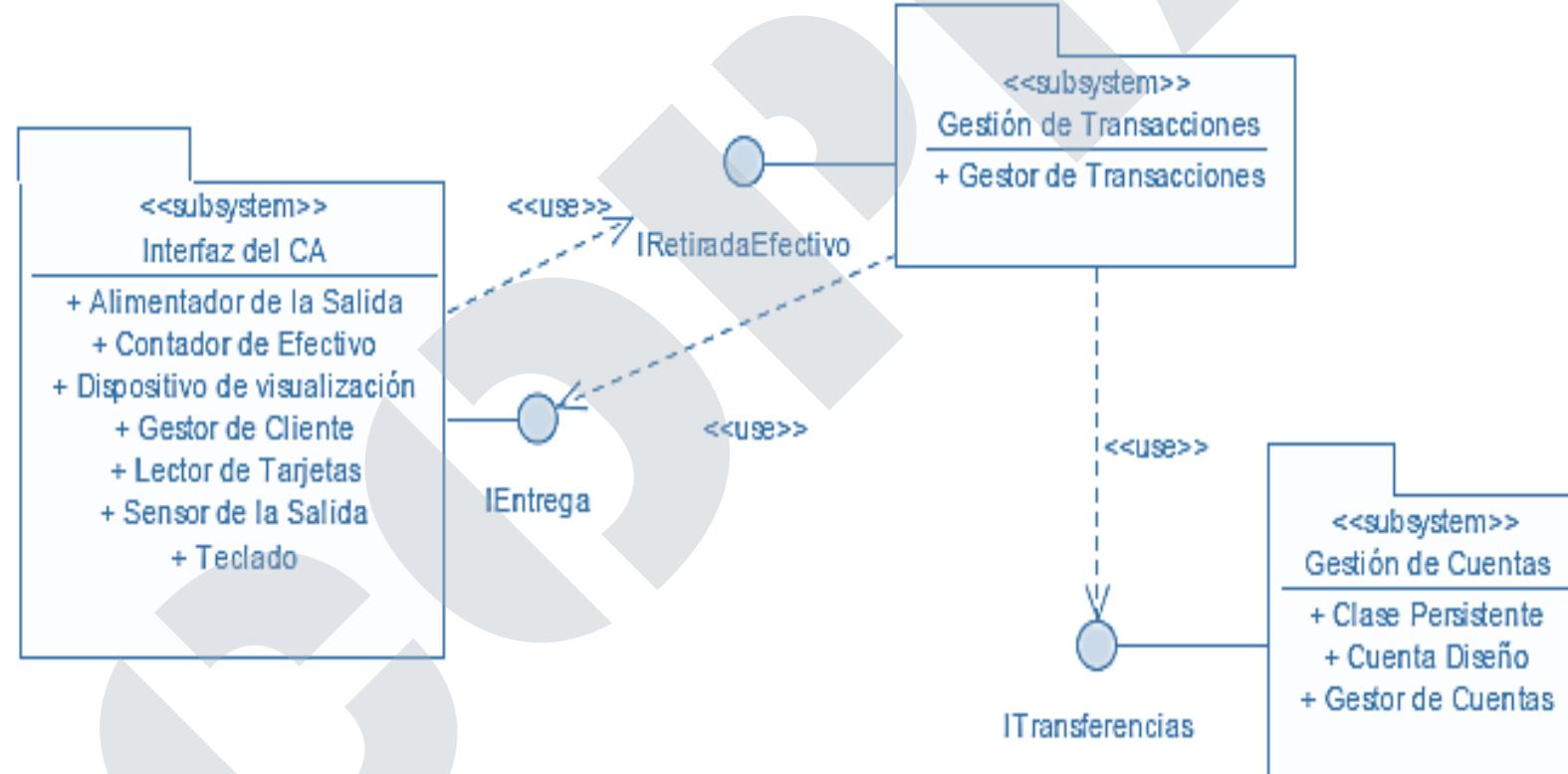
- **Subsistema de Gestión de Cuentas**

- Clases Persistentes
- **Gestor de Cuentas**
- Cuenta.

Agrupación de las clases en Subsistemas -interfaces.

- La ventaja de colocar todas las clases de interfaz en un subsistema permitiría **reemplazar el subsistema completo** para adecuarlo a otra interfaz sin mayores cambios en el resto del sistema.
- **Los subsistemas implementan interfaces.**
 - Las interfaces se representan por un círculo vinculado con una línea de trazo continuo a la clase dentro del subsistema que proporciona la interfaz.
 - Una línea de trazo discontinuo de una clase a una interfaz representa que la clase usa la interfaz.

Agrupación de las clases en Subsistemas -interfaces.



Las Clase del Diseño :Métodos y atributos

- Durante el workflow del diseño se **debe especificar el formato exacto de cada atributo** del diagrama de clases.
- Esto es así pues el Proceso Unificado es iterativo. En cada iteración se puede cambiar parte o todo, de lo que se haya hecho.
- Esto se mostrara en las siguientes secuencias de acciones para para determinar las clases de diseño desde al análisis.

Diagrama de Clase Inicial en el Workflow del Análisis

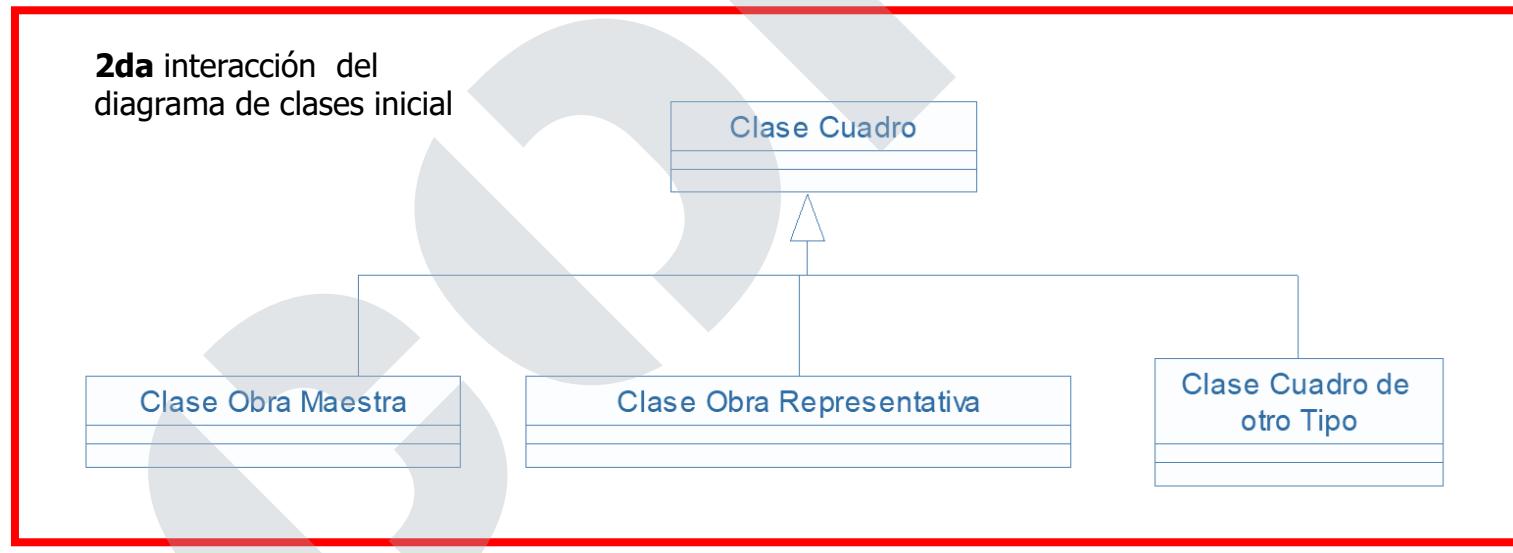


Diagrama de Clase Inicial en el Workflow del Análisis

3ra interacción del diagrama de clases inicial



4ta interacción del diagrama de clases inicial

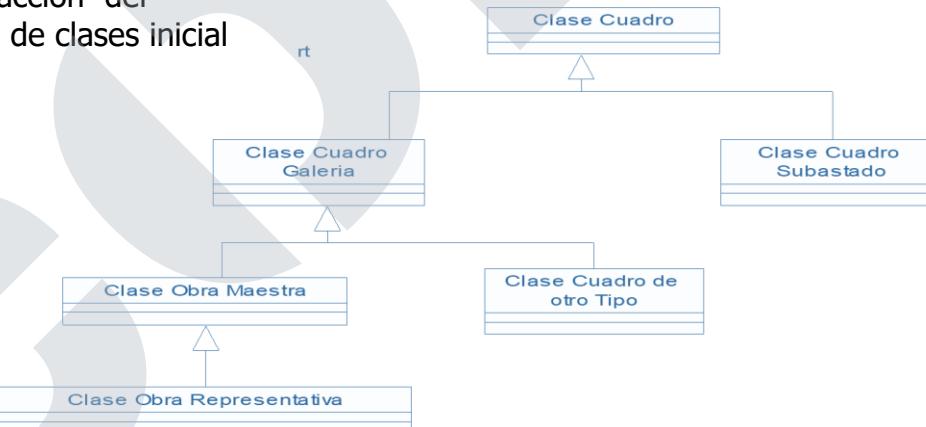


Diagrama de Clase Inicial en el Workflow del Análisis

5ta interacción del
diagrama de clases inicial

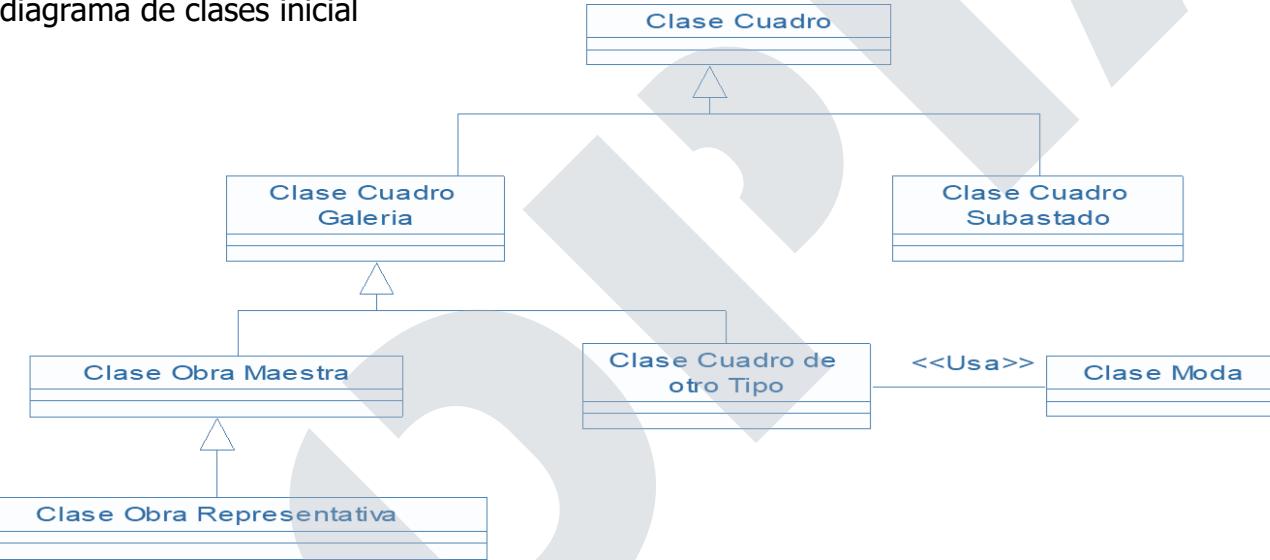
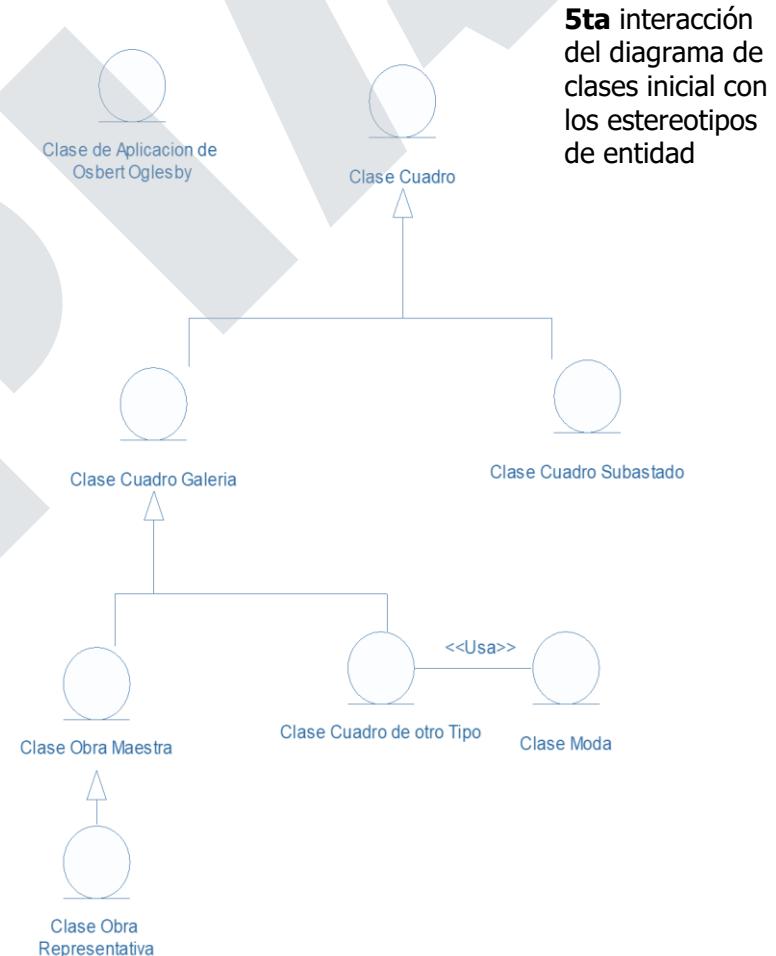
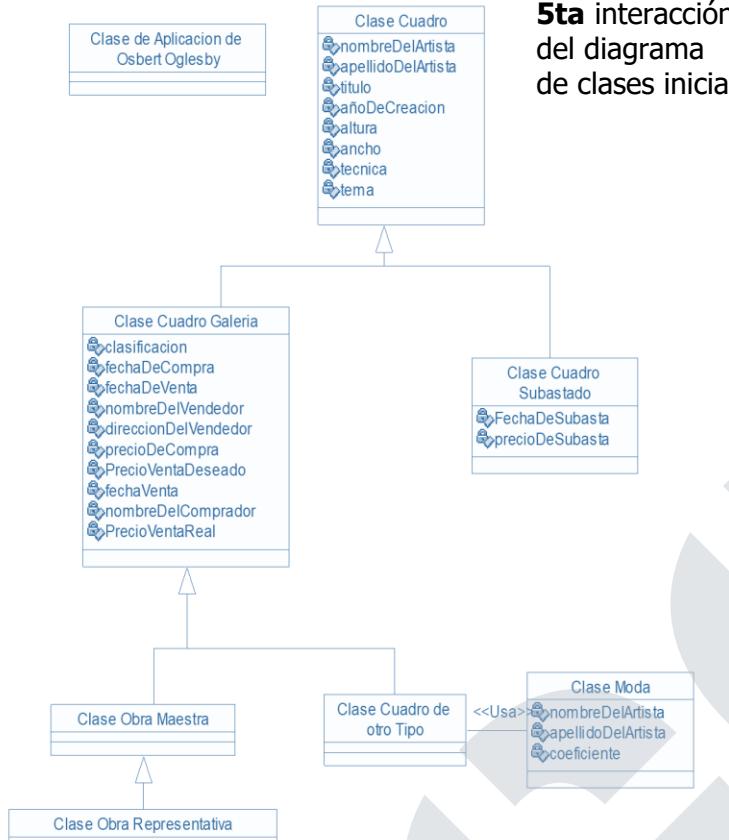


Diagrama de Clase Inicial en el Workflow del Análisis



En el Workflow del Diseño se especifica el formato exacto de los atributos



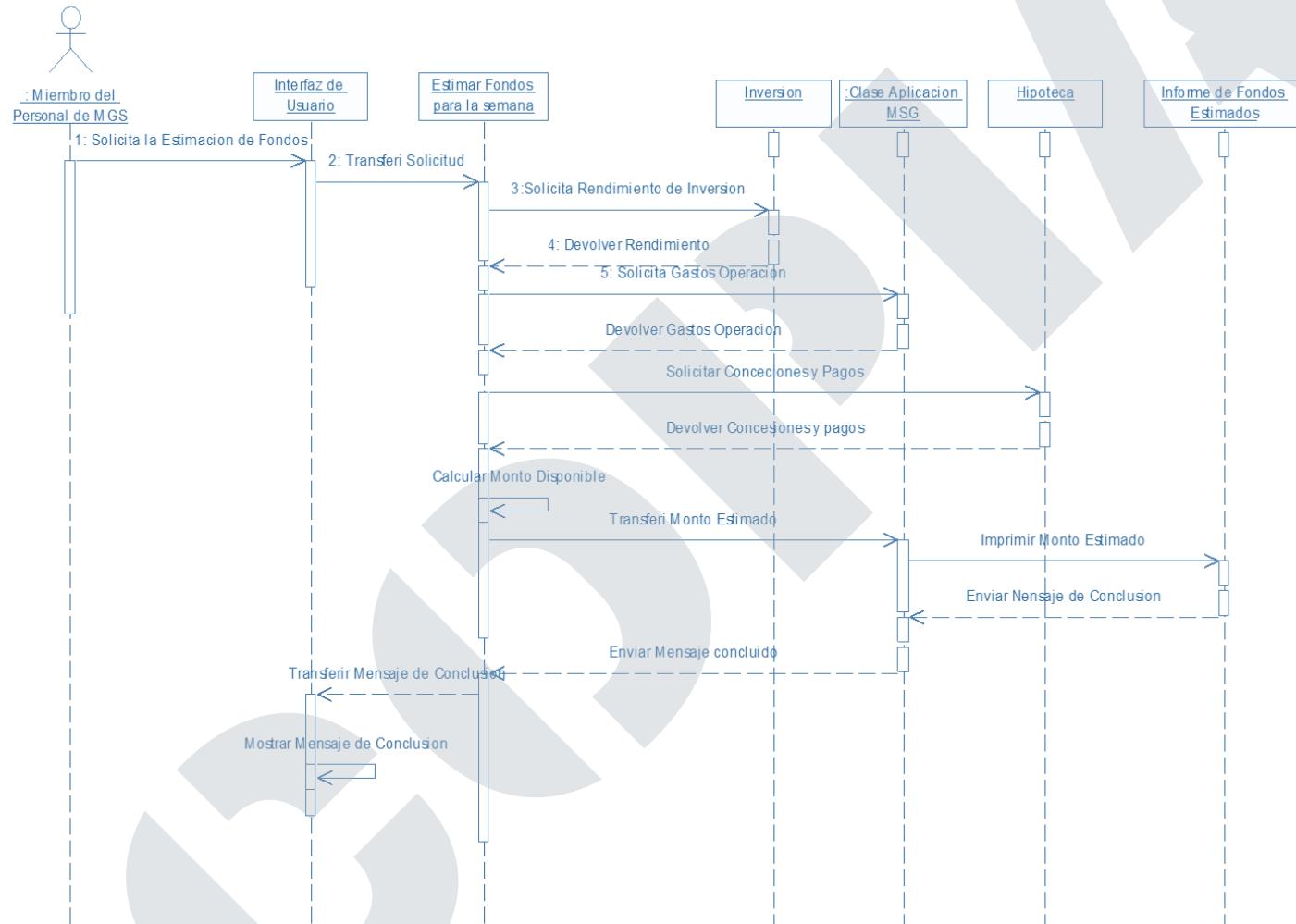
Asignación de Operaciones

Para establecer las operaciones de las clases del diseño se tendrá en cuenta los **diagramas de Secuencias** de la realización de CU diseño.

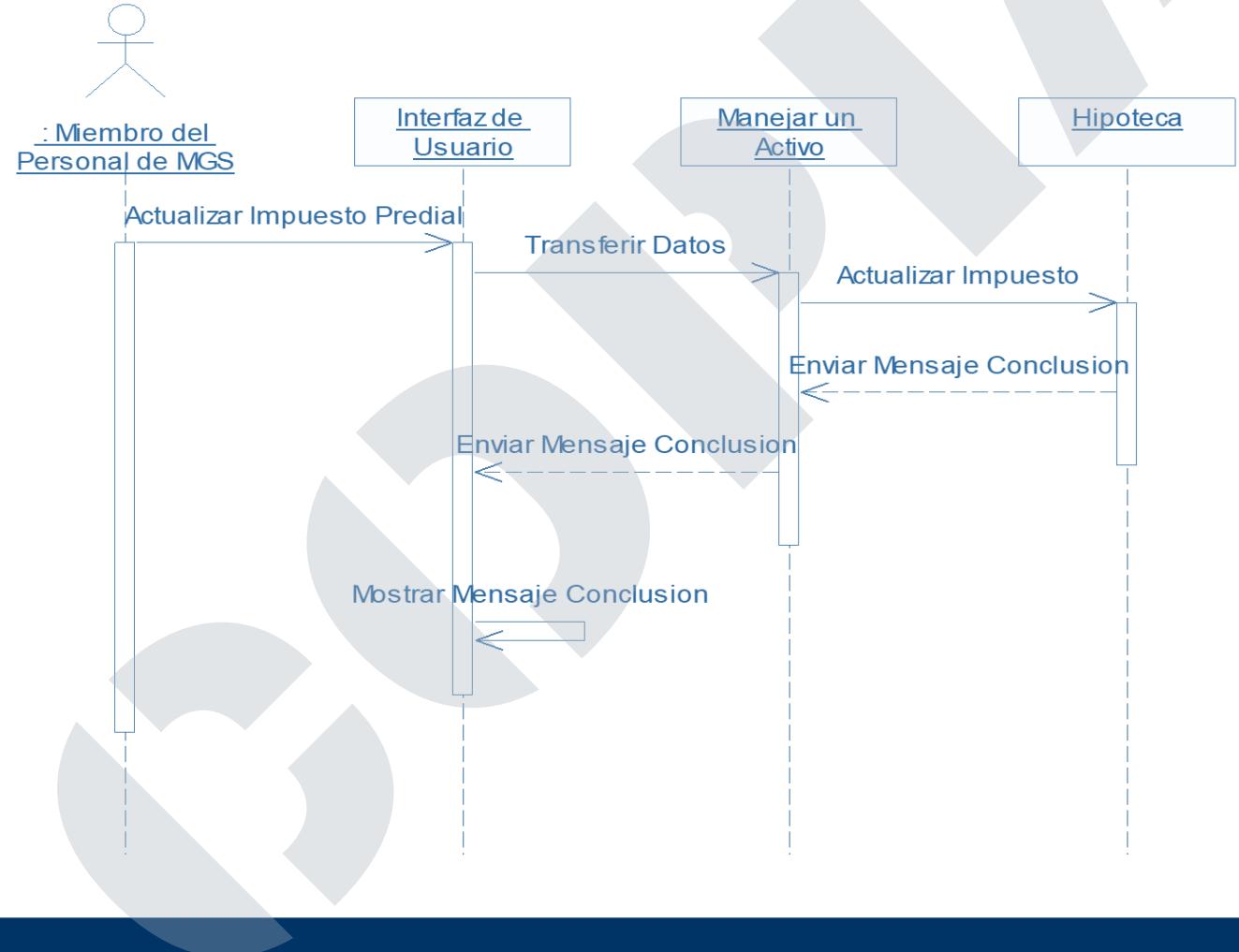
Los **mensajes** que la clase del diseño **recibe**, serán **candidatas para transformarse en Operaciones** de la clase.

También puede **tomarse los contratos establecidos en la realización de CU análisis**.

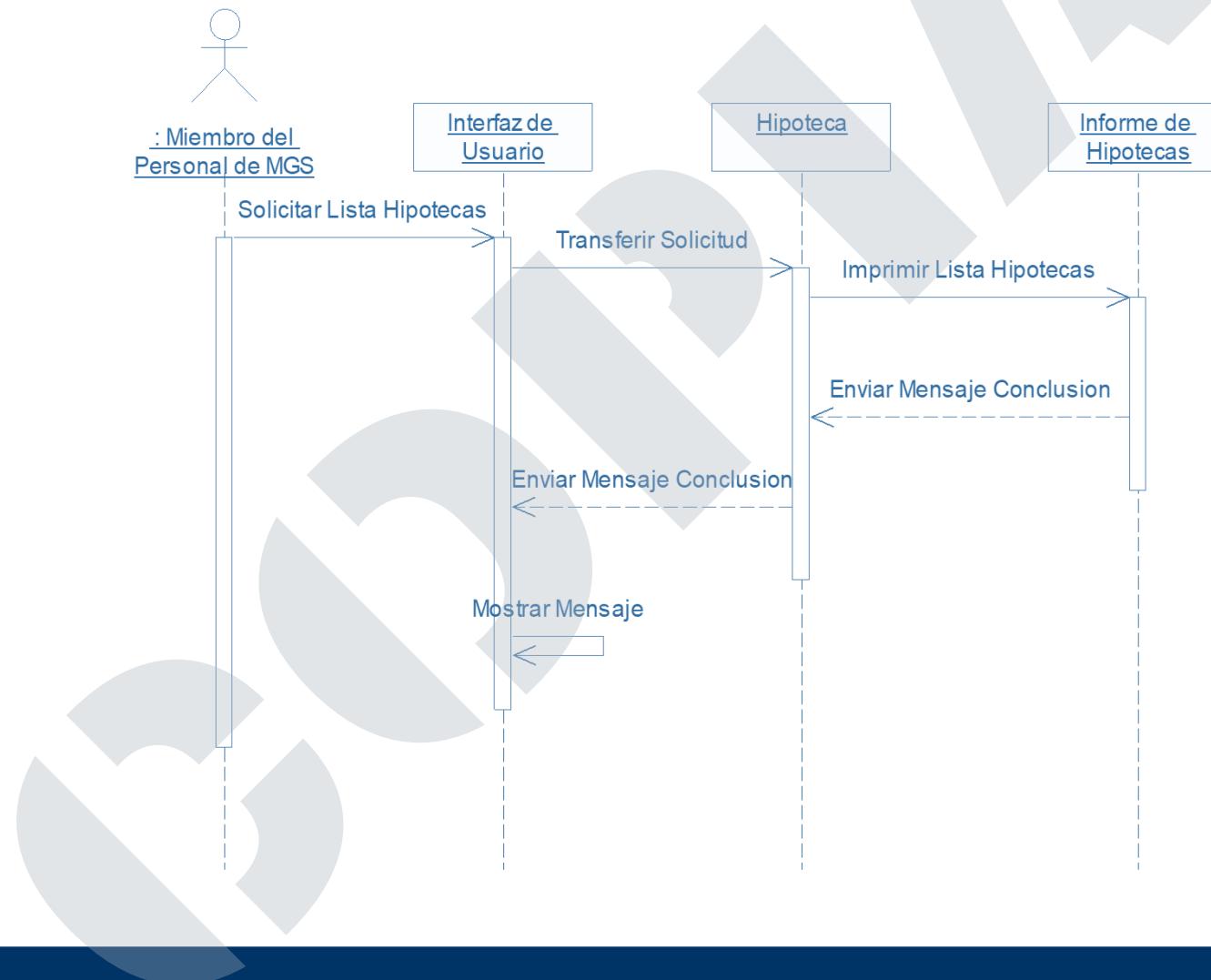
Tomando el diagrama de secuencia de la realización de CU diseño.



Tomando el diagrama de secuencia de la realización de CU diseño.



Tomando el diagrama de secuencia de la realización de CU diseño.



Tomando el diagrama de secuencia de la realización de CU diseño.

- Tomando en cuenta los diagramas de secuencia se **analizan los mensajes para determinar las operaciones que genera**.
- Se debe tener en cuenta que la **operación corresponde a la clase que recibe el mensaje**.
- Puede para esto se construir tarjetas CRC (clase responsabilidad colaboración), para todas las clases, tomaremos el ejemplo de la clase Hipoteca , que participa en variar realizaciones de CU diseño.

Tomando el diagrama de secuencia de la realización de CU diseño.

Clase : Hipoteca	
RESPONSABILIDADES	COLABORACIONES
Calcular pagos y concesiones estimados	Clase: Estimar Fondos para la semana
Iniciar, Actualizar y Eliminar Hipotecas	Clase: Manejar un activo
Generar Lista de Hipotecas	Clase: Interfaz de Usuario
Imprimir lista de hipotecas	Clase: Informe de Hipotecas







DISEÑO SOFTWARE

Diagrama de Paquetes

Sesión S6

Paquetes

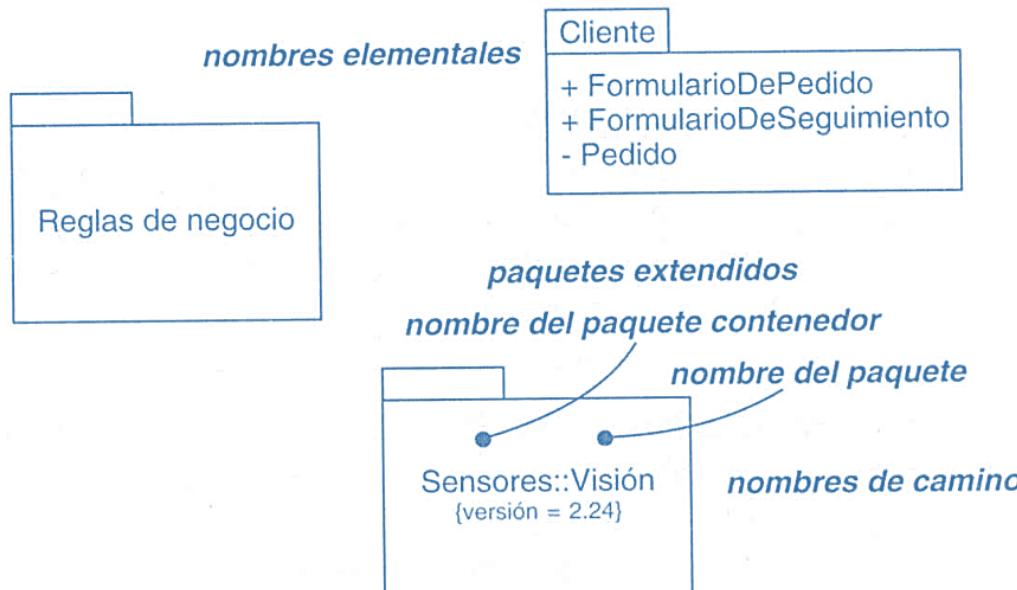
Un **paquete** es un mecanismo de propósito general para organizar un modelo de manera jerárquica.

Uso:

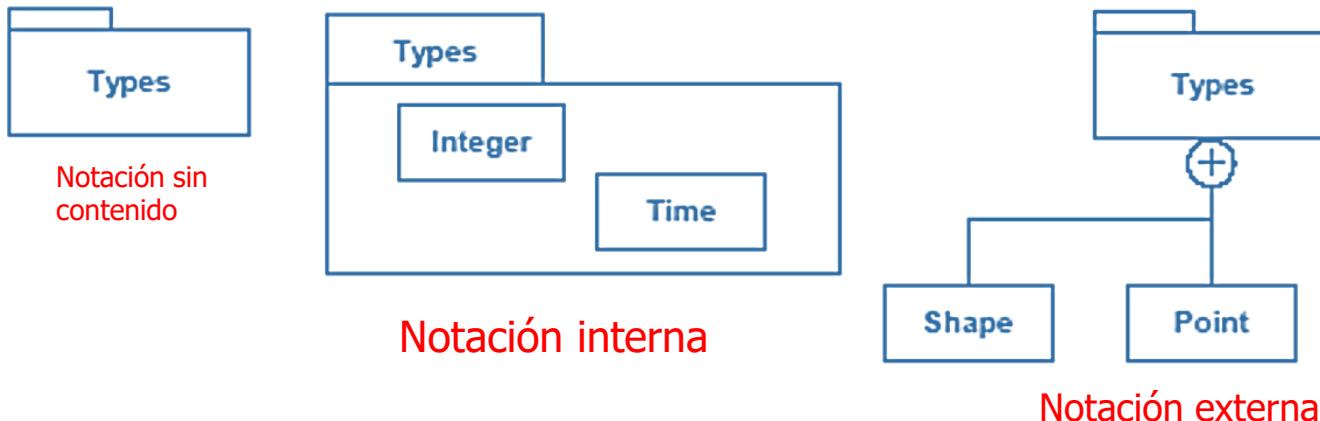
- Organizar los elementos en los modelos para comprenderlos más fácilmente.
- Controlar el acceso a sus contenidos para controlar las líneas de separación de la arquitectura del sistema



Notación Paquetes



Representación de Paquetes

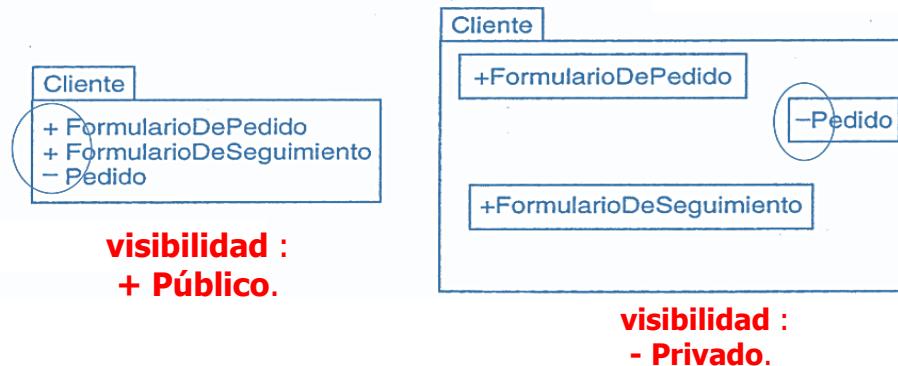


- Las clases son abstracciones de aspectos del problema o la solución.
- Los paquetes son mecanismos para organizar, pero no tienen identidad (no puede haber instancias de paquetes)

Paquetes - Contenido

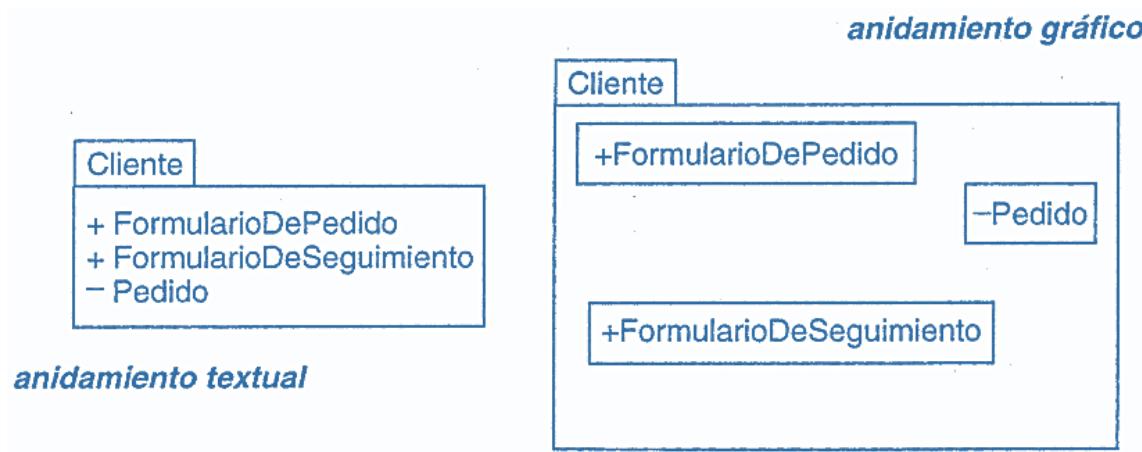
Los paquetes pueden contener **elementos** como clases, interfaces, componentes, nodos, colaboraciones, casos de uso e incluso otros paquetes.

- Entre un paquete y sus elementos existe una **relación de composición**
- Un paquete forma un **espacio de nombres** (namespace)



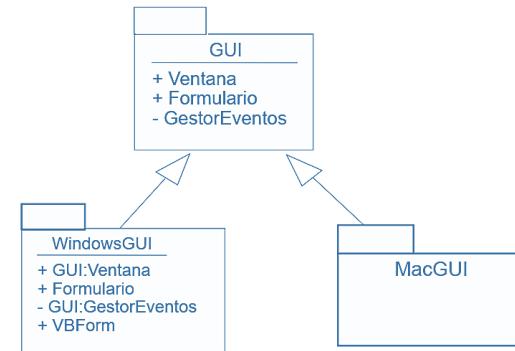
Paquetes - Contenido

Se puede mostrar de forma explícita el contenido de un paquete, de forma textual o gráfica.



Paquetes - Relaciones

Generalización: Un paquete especializado puede usarse en sustitución de un paquete más general, del cual hereda.



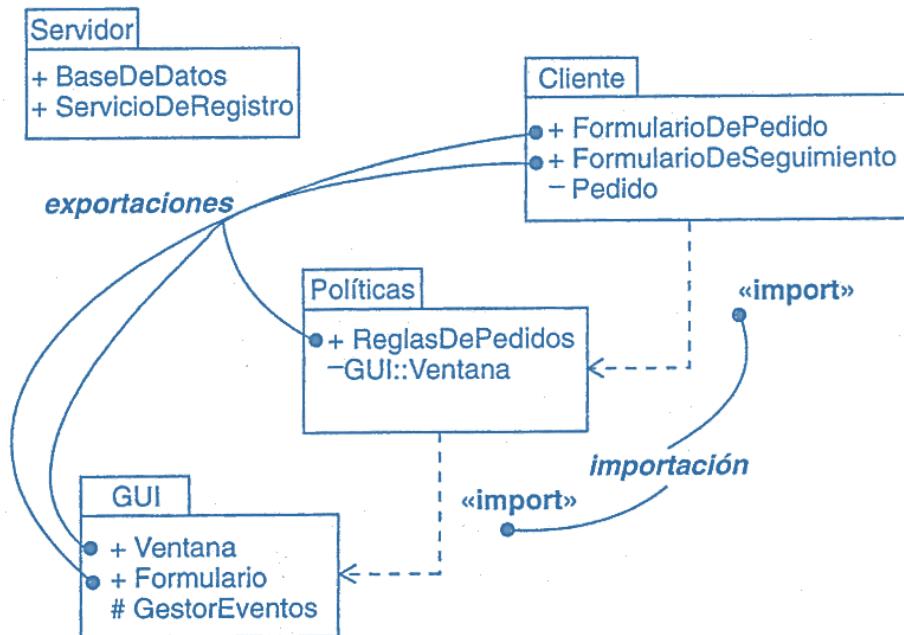
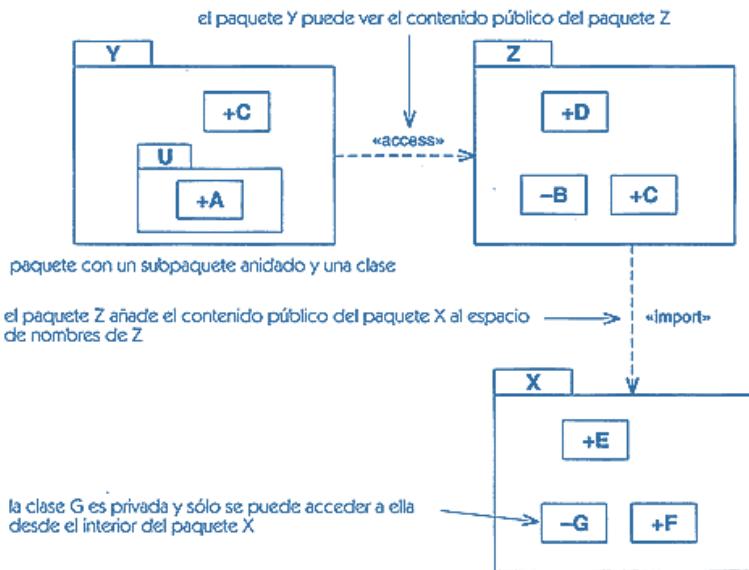
Dependencias: Muestran que algún elemento de un paquete depende de los elementos en otro paquete. (3 tipos Importación, Exportación, Acceso)



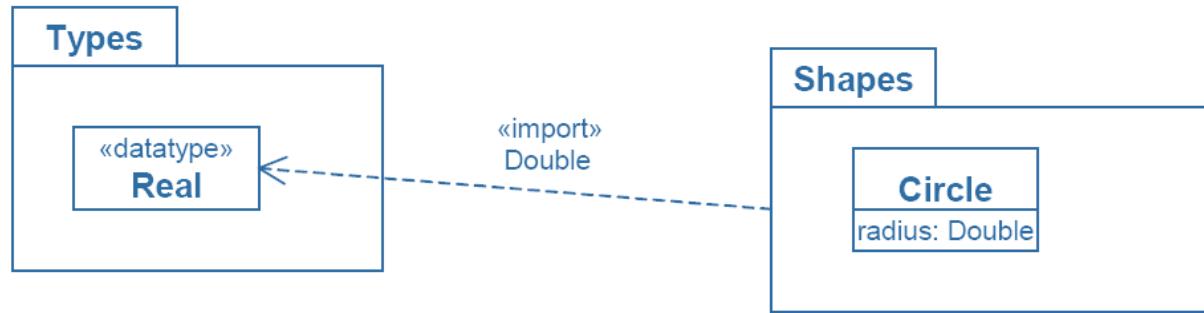
Paquetes - Relaciones



Paquetes - Relaciones

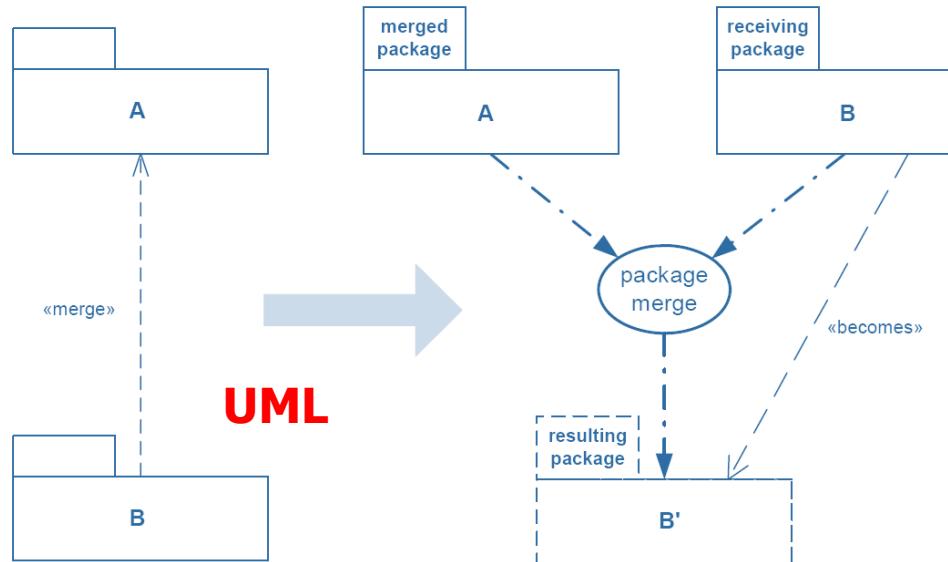


Paquetes - Relaciones

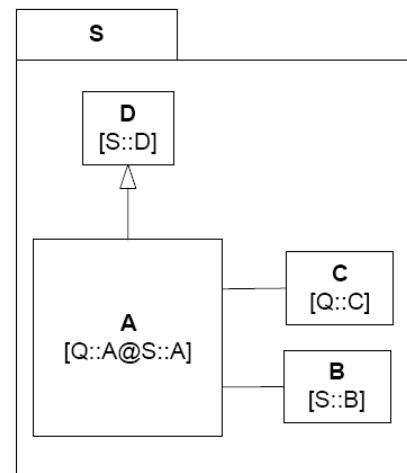
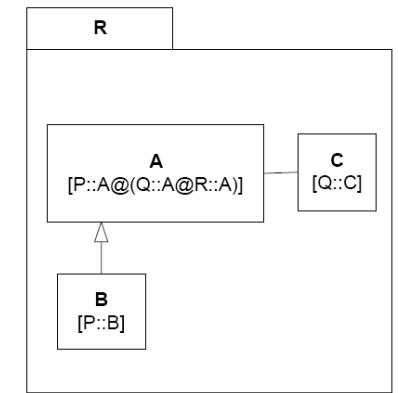
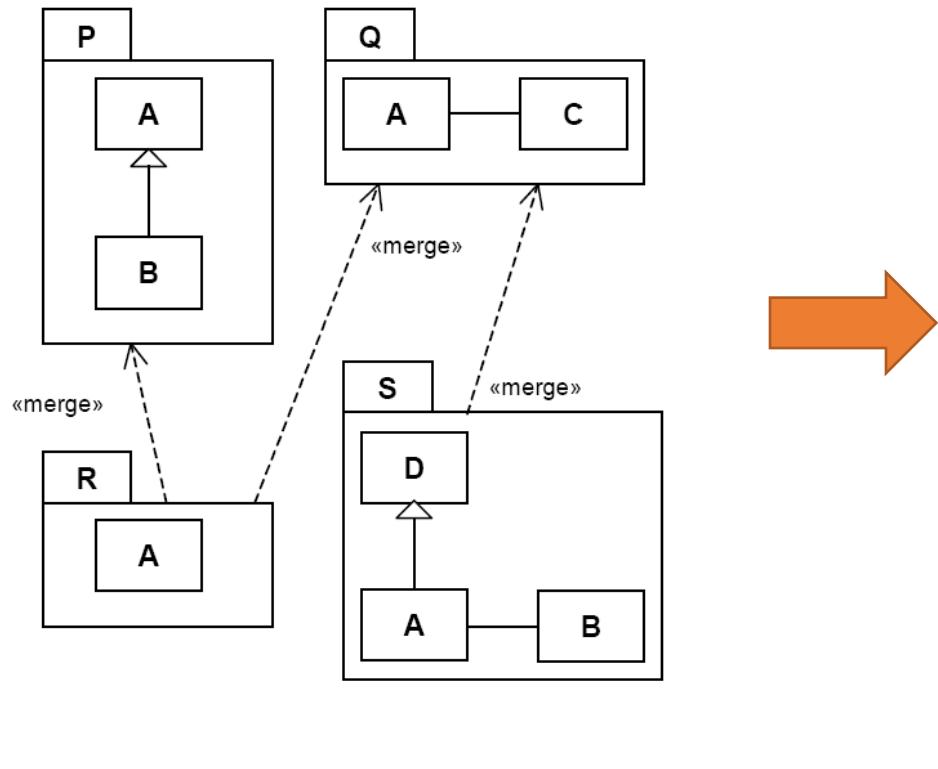


Paquetes - Fusión

Una relación de **fusión** (merge) entre dos paquetes especifica que el contenido del paquete origen (receptor) se extiende con el contenido del paquete destino .



Paquetes - Fusión



Paquetes - Tipos

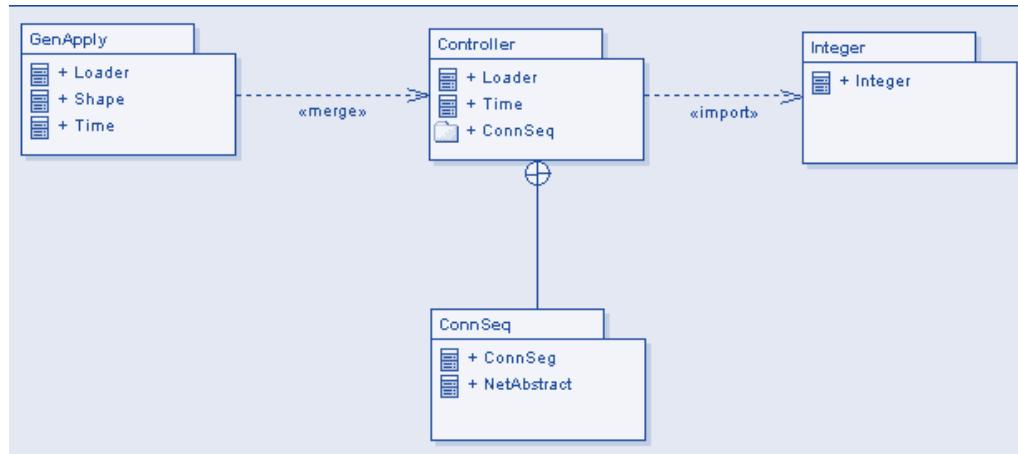
Existen varios **estereotipos** que definen nuevas categorías de paquetes (además del ya conocido “profile”):

- **System** - Paquete que representa el sistema completo que se está modelando.
- **Subsystem** - Expresa que el paquete es independiente del sistema completo que se modela.
- **Framework** – Contiene elementos reusables como clases, patrones y plantillas.
- **Facade** (fachada) - Proporciona una vista simplificada del paquete.
- **Stub** - Un paquete que sirve de sustituto para el contenido público de otro paquete.

Otros: ModelLibrary, Perspective, ..

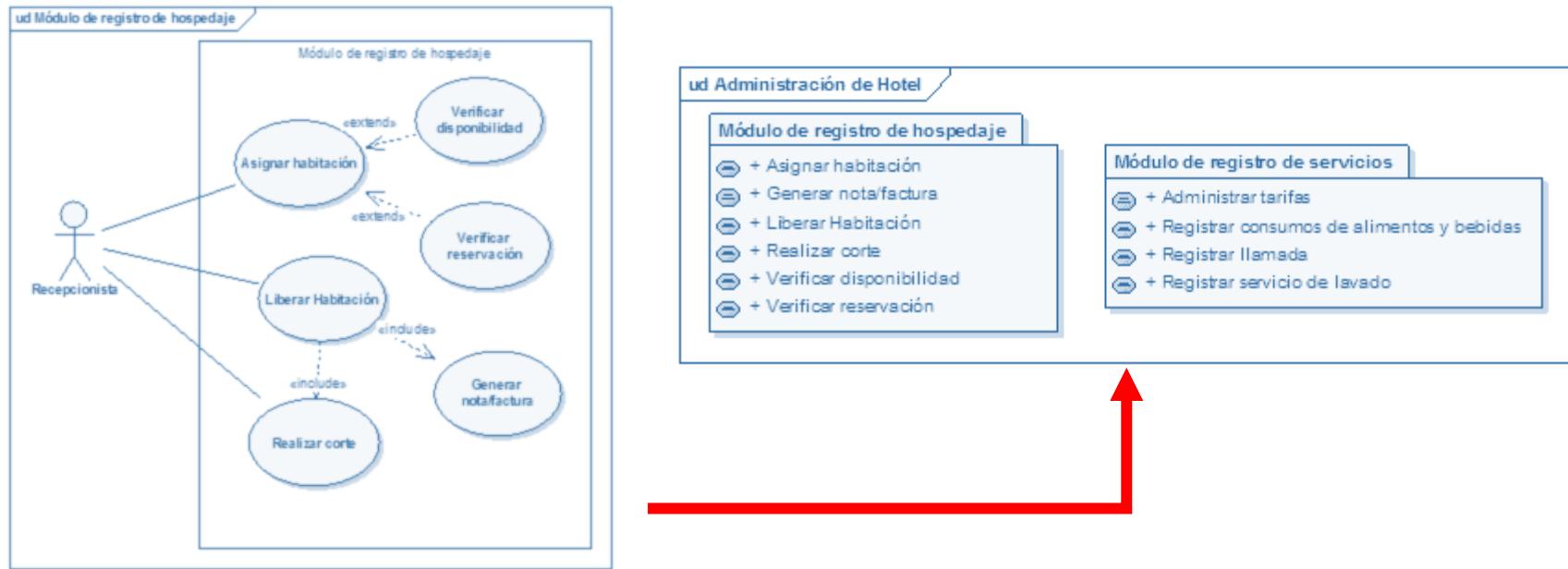
Diagrama de Paquetes

Un **diagrama de paquetes** es un diagrama de estructura cuyo contenido es, principalmente, paquetes y sus relaciones.



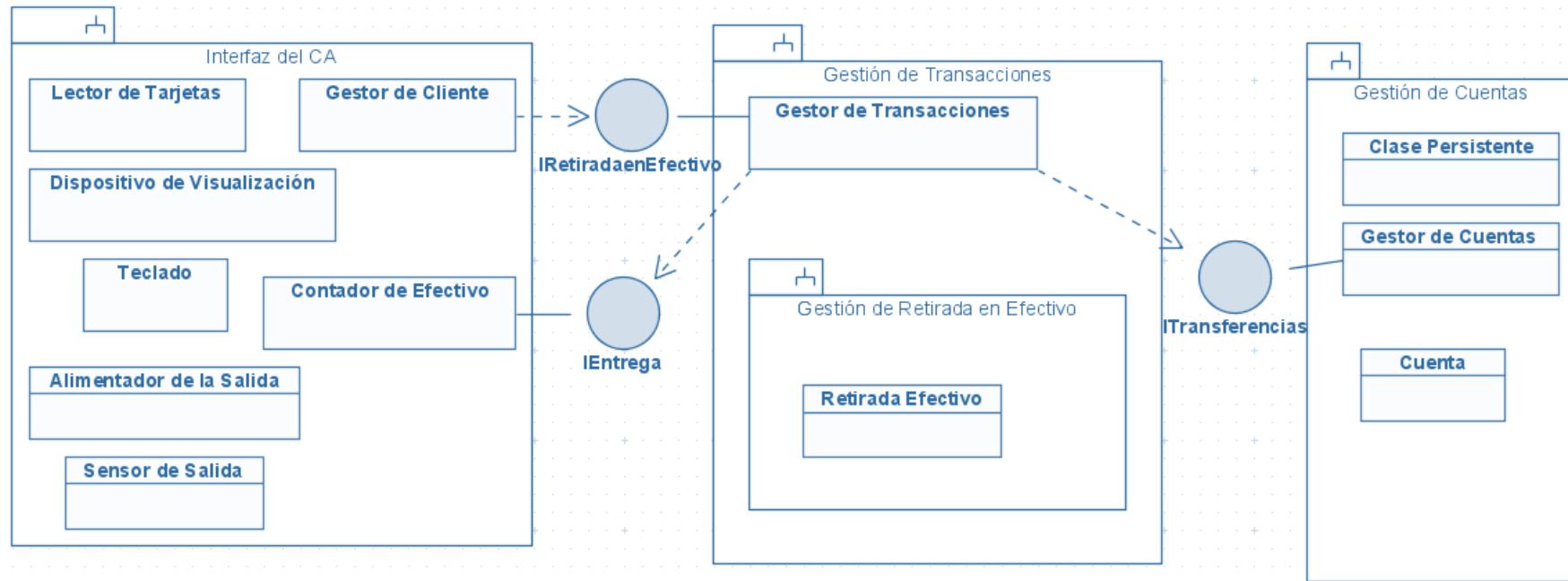
Ejemplo:

Diagrama de Casos de Uso simplificado con el uso de paquetes.



Ejemplo:

Arquitectura detallada.





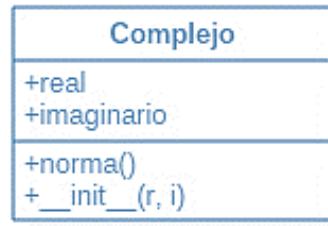
DISEÑO SOFTWARE

Diagrama de clases

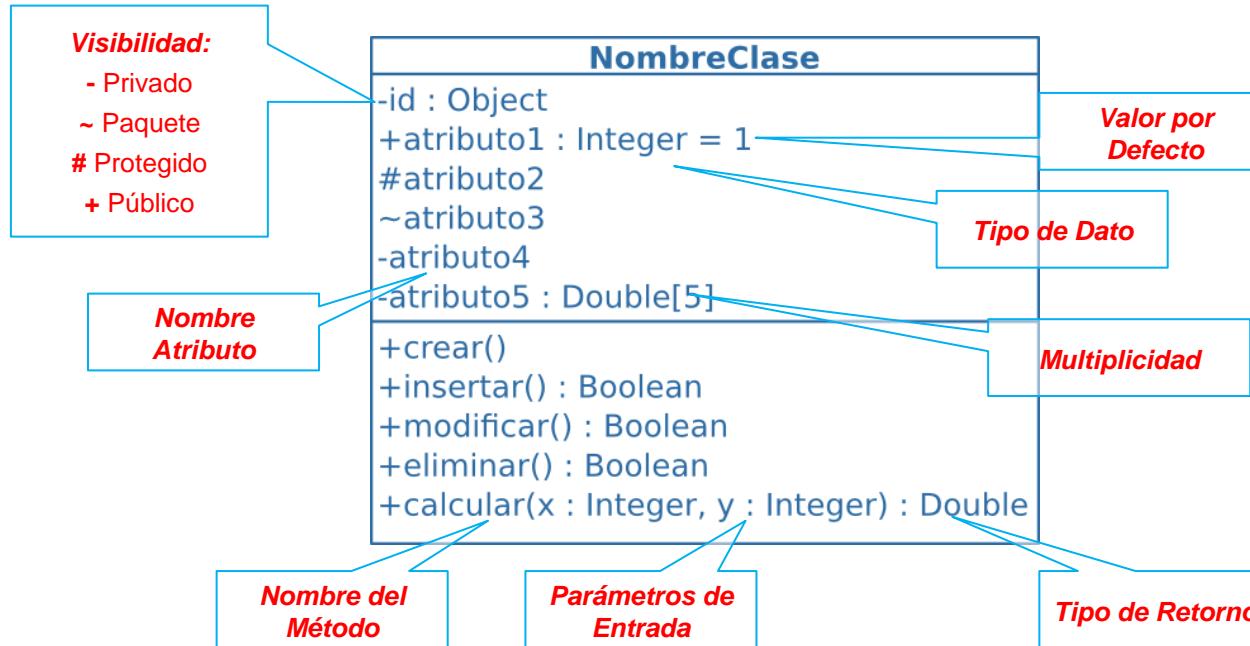
Sesión S6

Clase de Diseño

```
class Complejo:  
    def __init__(self, r, i):  
        self.__real = r  
        self.__imaginario = i  
  
    def norma(self):  
        return math.sqrt(self.__real**2 + self.__imaginario**2)
```



Clase de Diseño

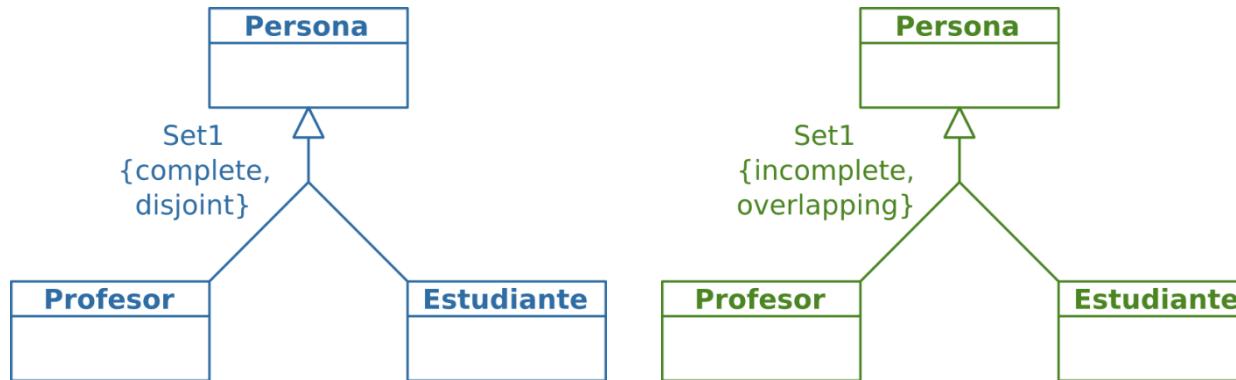


Especialización / Generalización / Herencia

Herencia:

Disjunta / Traslapada

Total / Parcial



```
class Figura:  
    def calcArea(self):  
        def dibujar(self, canvas):
```

```
class  
Rectangulo(Figura  
):  
  
    pass
```

```
class Cuadrado(Rectangulo):  
  
    pass
```

```
class Elipse(Figura):  
    pass
```

```
class  
Circulo(Elipse):  
  
    pass
```

Diagramas de Clases :(Asociaciones)

```
class Departamento:  
    def __init__(self):  
        self.ListaProfesores = []  
  
    def listarProfesor(self):  
        for i in range(len(self.ListaProfesores)):  
            print(self.ListaProfesores[i].nombre)  
    def adicionar(self, profesor):  
        self.ListaProfesores.append(profesor)  
  
class Profesor:  
    def __init__(self, nombre, departamento):  
        self.nombre=nombre  
        self.departamentoRef=departamento  
        departamento.adprofesor(self)
```



Asociaciones

```
class Estudiante:  
    def __init__(self):  
        self.ListaAsignaturas = []  
  
class Asignaturas:  
    def __init__(self):  
        self.ListaEstudiantes = []
```

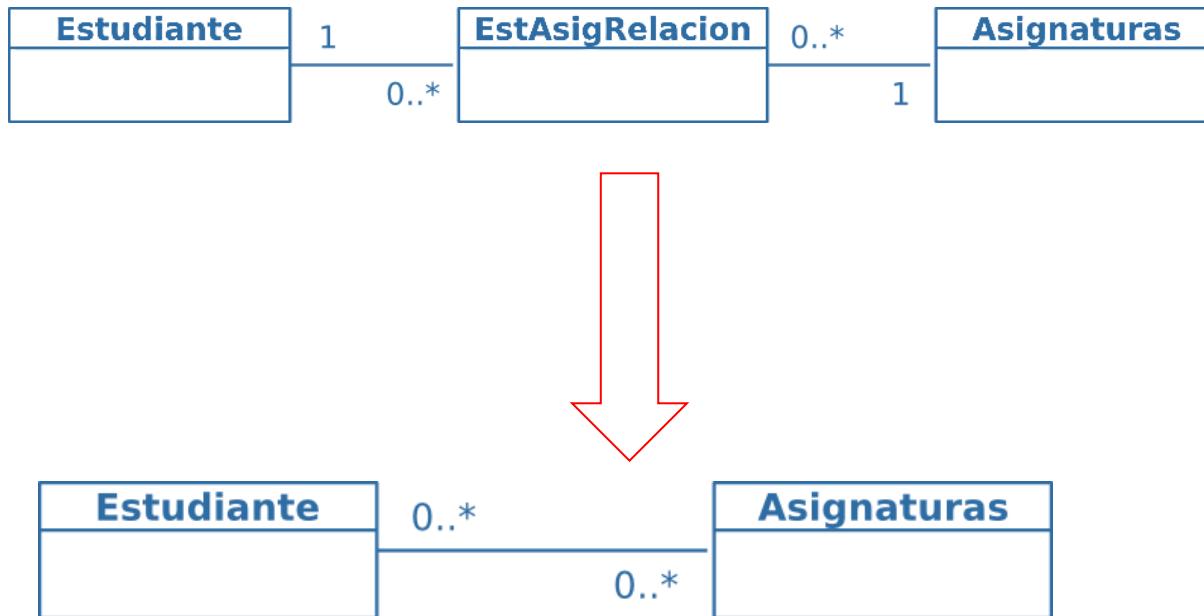


Asociaciones

```
class Estudiante:  
    def __init__(self):  
        self.ListaEstAsigRelacion = []  
  
class EstAsigRelacion:  
    def __init__(self, estudiante, asignatura):  
        self.estudianteRef=estudiante  
        self.asignaturaRef=asignatura  
  
class Asignaturas:  
    def __init__(self):  
        self.ListaEstAsigRelacion = []
```



Asociaciones



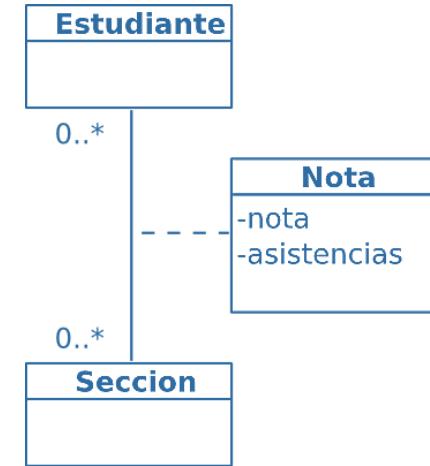
Asociaciones

```
class Departamento:  
    def __init__(self, secretaria):  
        self.secretariaRef = secretaria  
  
class Secretaria:  
    def __init__(self, departamento):  
        self.departamentoRef=departamento
```



Asociaciones

```
class Estudiante:  
    def __init__(self):  
        self.ListaNotas= []  
  
class Nota:  
    def  
        __init__(self,estudiante,seccion,nota,asistencia):  
            self.nota=nota  
            self.asistencias=asistencia  
            self.estudianteRef=estudiante  
            self.SeccionRef=seccion  
  
class Seccion:  
    def __init__(self):  
        self.ListaNotas = []
```





DISEÑO DE SISTEMAS DE INFORMACIÓN

Diagrama de estado

Sesión S6

Definición:

Un Diagrama de Estado es una técnica que *describe todos los estados posibles de un objeto.*

Un diagrama de estado representa el ciclo de vida de un objeto: los eventos que le ocurren, sus transiciones, y los estados que median entre estos eventos.

Máquina de estados

Especifica la **secuencia de estados** por las que pasa un objeto a lo largo de su vida en respuesta a eventos, junto con sus respuestas a esos eventos.

Útiles para **objetos reactivos**, las instancias de su clase

deben **responder a eventos** externos o internos

tienen un **comportamiento que depende** de su historia

tienen un ciclo de vida modelado como una **progresión de estados**, transiciones y eventos.

Se representa mediante un **diagrama de estados**.

Conceptos Importantes:

Estado: condición de un objeto en un momento determinado.

Elementos de un estado:

Nombre

Acciones entrada/salida

Transiciones internas

Subestados

Eventos diferidos

Ejemplo: un teléfono se encuentra en estado “ocioso” una vez que el auricular es puesto en su sitio y mientras no lo levantemos.

Conceptos Importantes:

Evento: Un evento es algo que ocurre en el ambiente que afecta el comportamiento del objeto analizado ocasionando que cambie a un nuevo estado..

Ejemplo: levantar el auricular telefónico.

Conceptos Importantes:

Transición: relación entre dos estados, indica que, cuando ocurre un evento el objeto pasa del estado anterior al siguiente. (*Simple*)

Elementos de una transición:

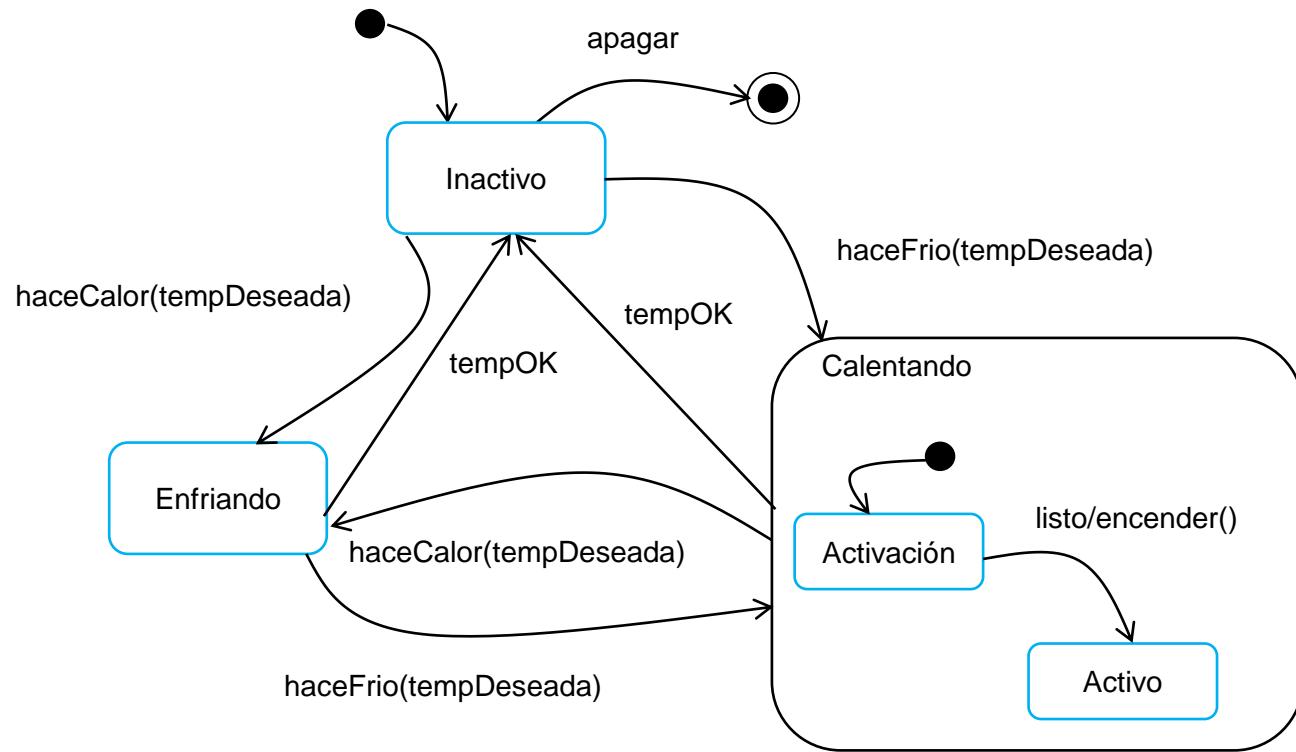
Estados origen y destino

Evento de disparo (cero o más)

Condición de guarda (cero o más)

Acción (cero o más)

Ejemplo: cuando ocurre el evento “levantar el auricular”, el teléfono realiza la transición del estado “ocioso” al estado “activo”.



Tipos de Transiciones

Internas: Es una transición que permanece en el mismo estado, en vez de involucrar dos estados distintos.

Compleja: Relaciona tres o más estados en una transición de múltiples fuentes y/o múltiples destinos.

Temporizada: Las esperas son actividades que tienen asociada cierta duración. Un evento esperado puede ocasionar una transición que permita salir de la espera.

Definiciones

Subestados: Hace referencia a que dentro de un estado puede haber otros estados con sus transiciones.

Generalización de Estados:

Podemos reducir la complejidad de los diagramas usando la generalización de estados y distinguimos así entre superestado y subestados. Los subestados heredan las variables de estado y las transiciones externas.

Acción: Se puede especificar la ejecución de una acción como consecuencia de una transición. Puede venir acompañada de una condición para que se ejecute tal acción

Actividad: Es similar a una acción pero tienen duración y se ejecutan dentro de un estado del objeto. Además puede interrumpirse en todo momento, cuando ocurre la operación de salida de un estado.

Dependencias:

Antecedentes:

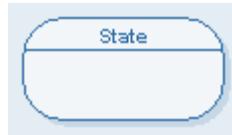
- =>Descripción de Casos de Uso en
Formato Expandido.
- =>Diagrama de Clases.

Precedentes:

- =>Diagrama de Despliegue.
- =>Diagrama de Componentes

Notación

ESTADO:



INICIO:



FIN:



TRANSICIÓN:

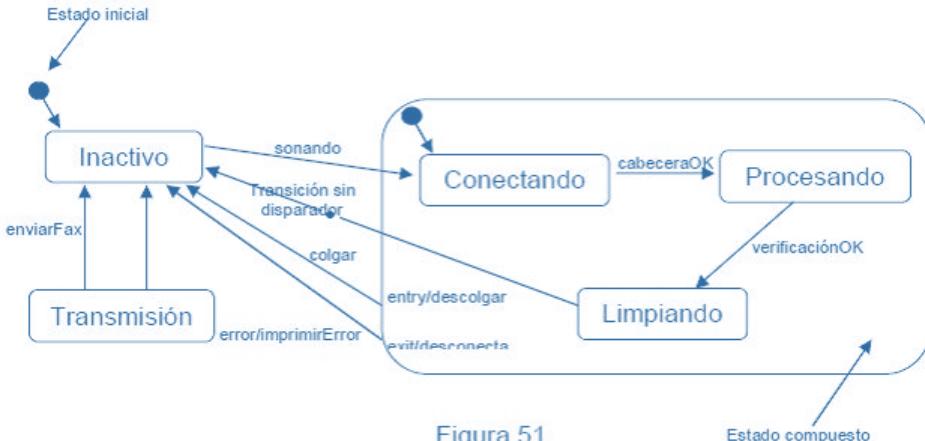
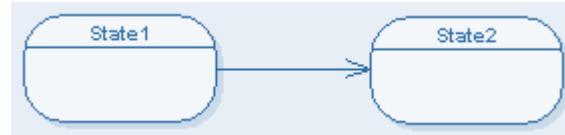
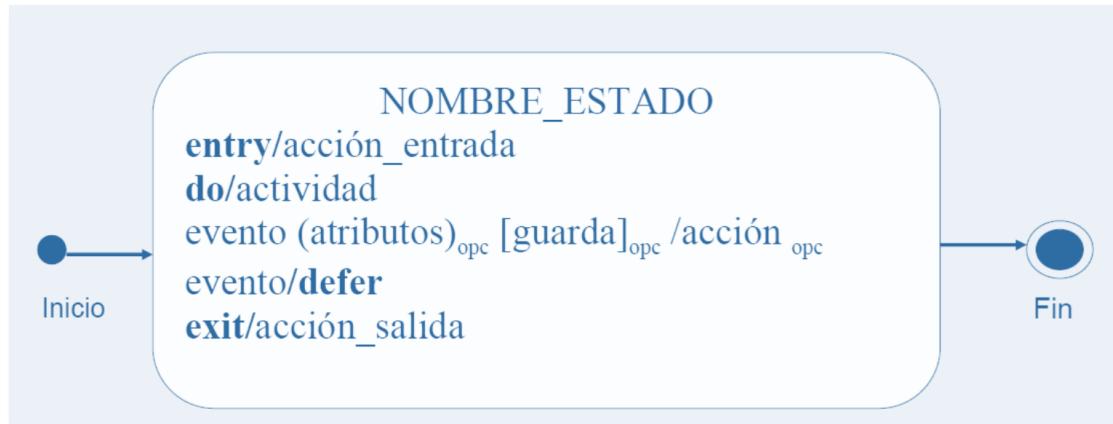


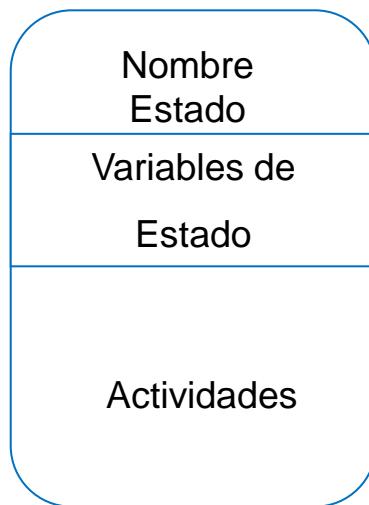
Figura 51.

Partes de un estado



Notación

Más información en los Estados

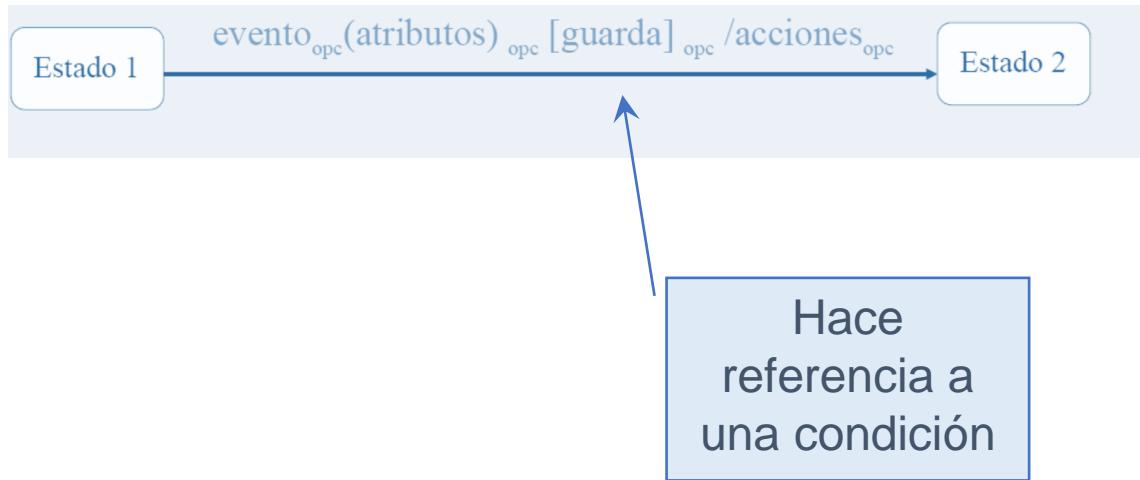


Son atributos que
definen posibles
estados iniciales



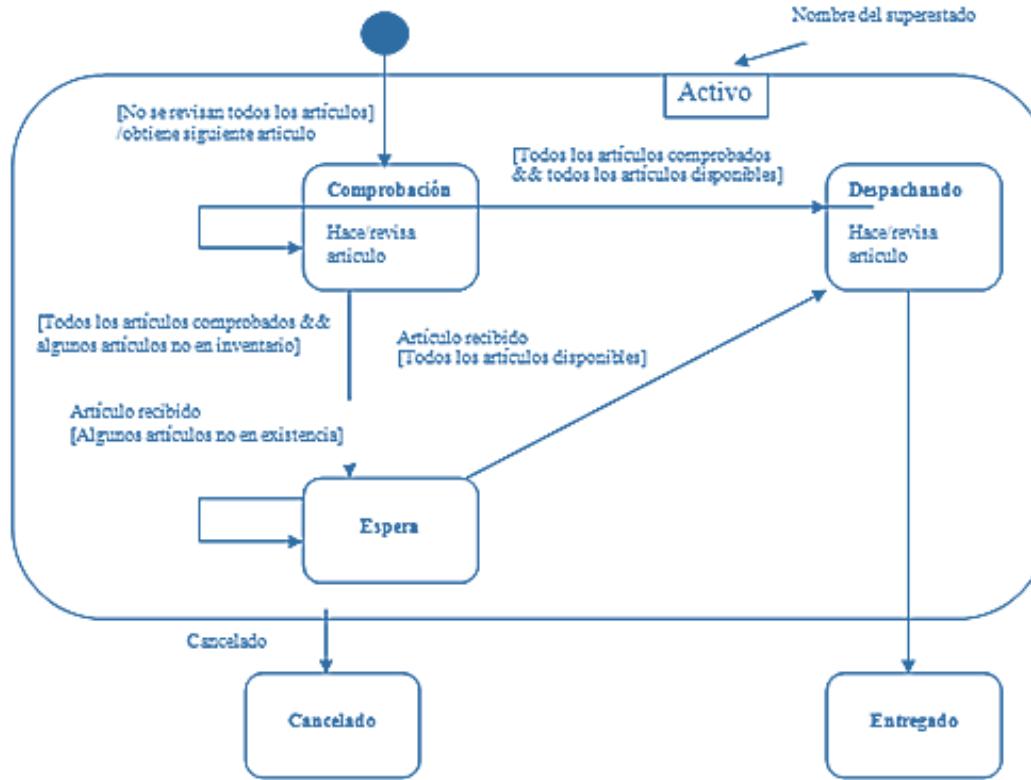
entry: al entrar
exit: al salir
do: en el estado

Transición:



Pasos que se siguen en su construcción

1. Identificar todos los sucesos y estados analizando los casos de uso u otros artefactos disponibles.
2. Seleccionar aquellos **objetos** que sean de mayor **relevancia** para el desarrollo del Sistema.
3. Construir el Diagrama de Estado.



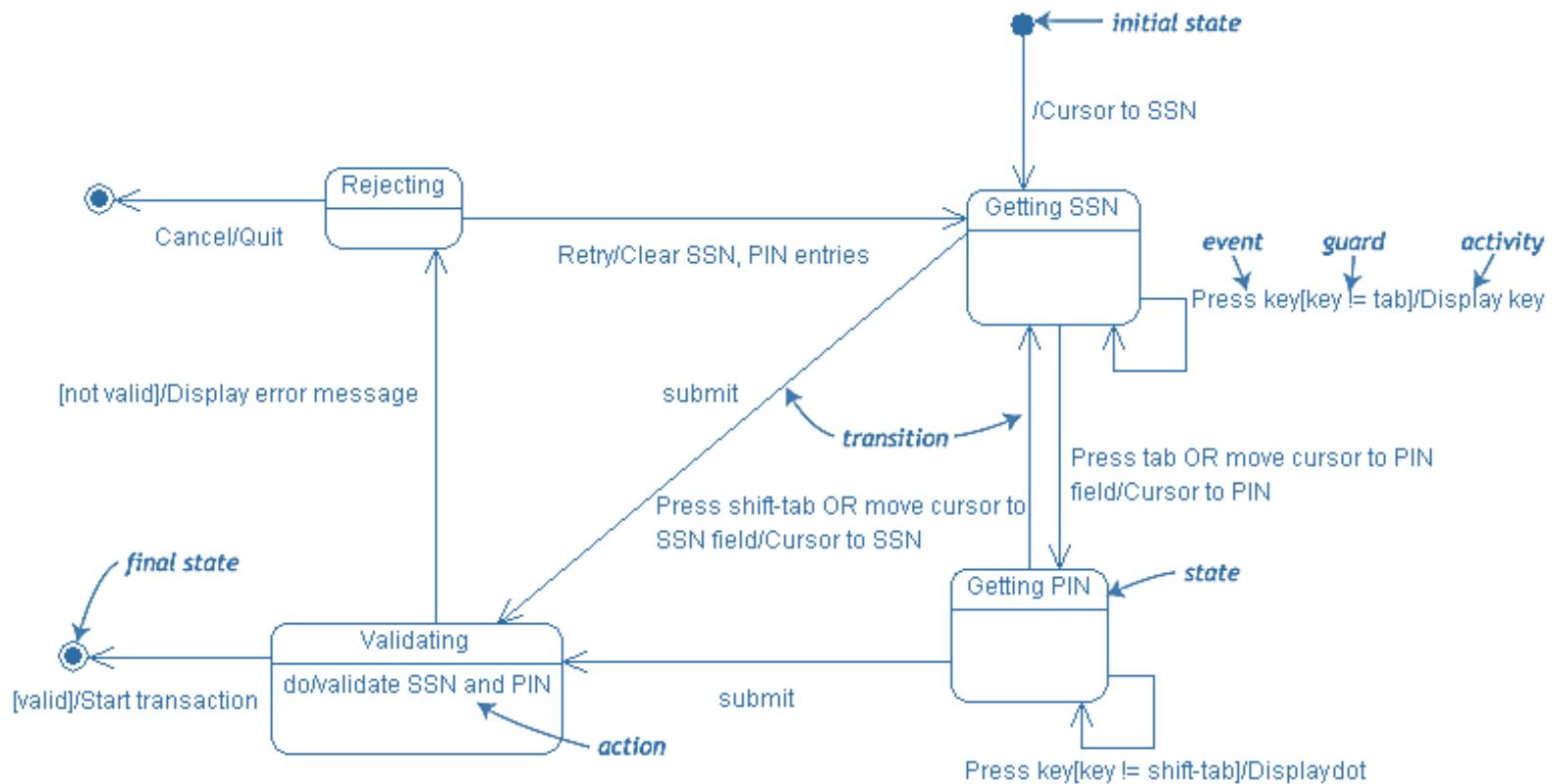
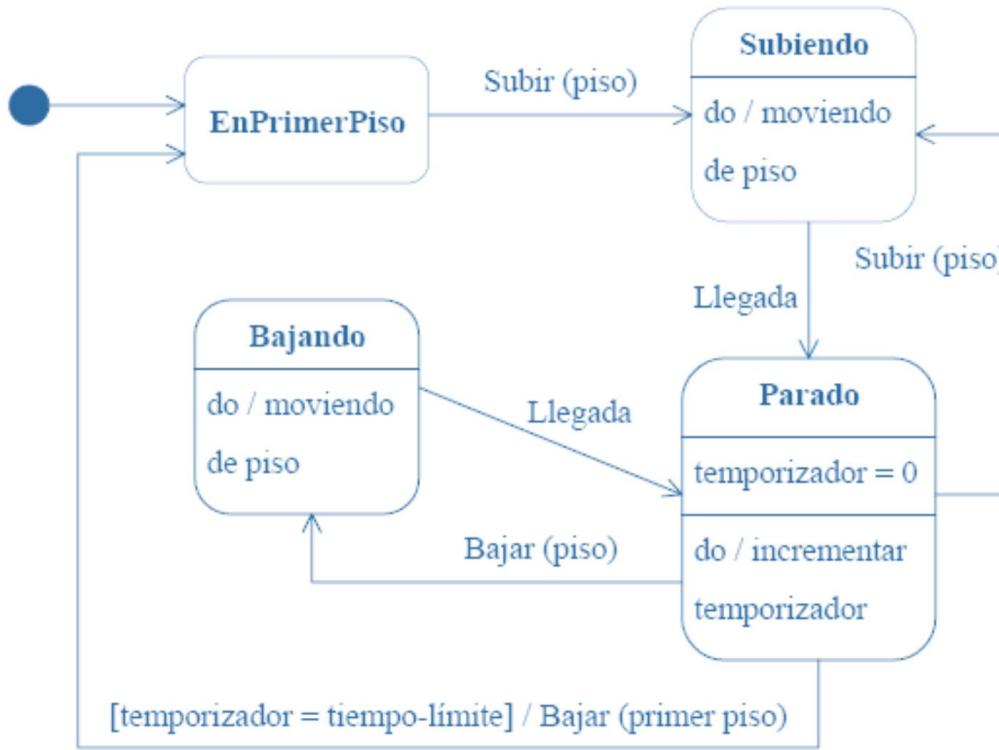
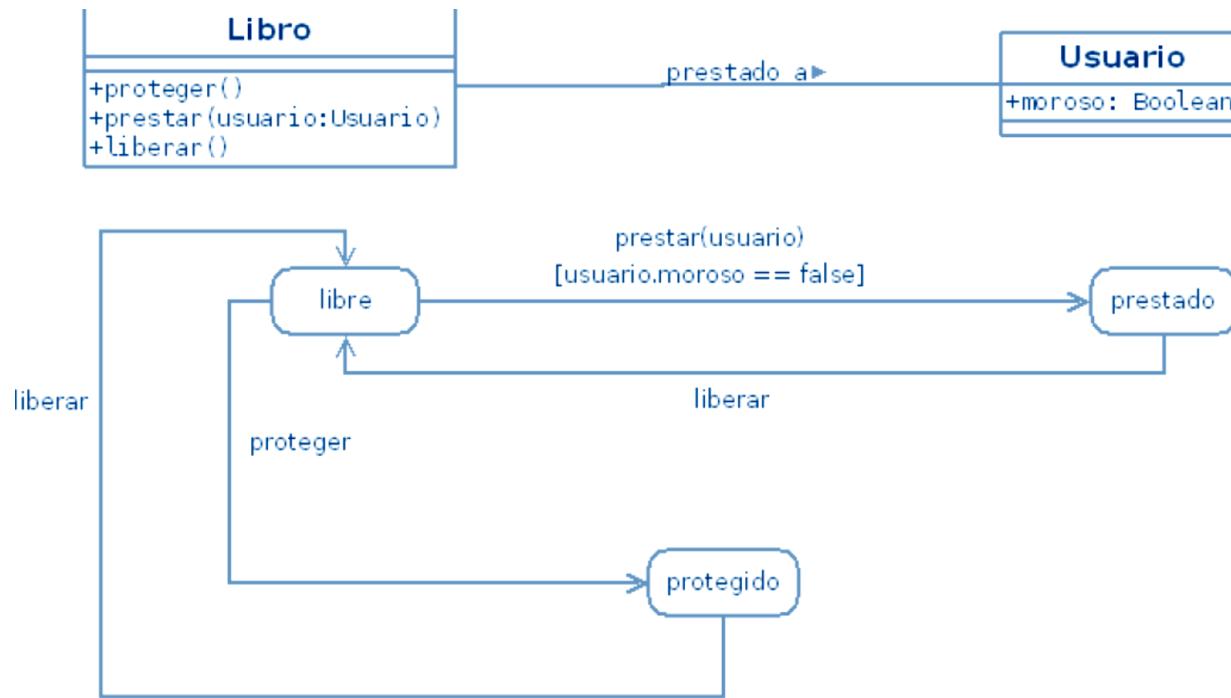
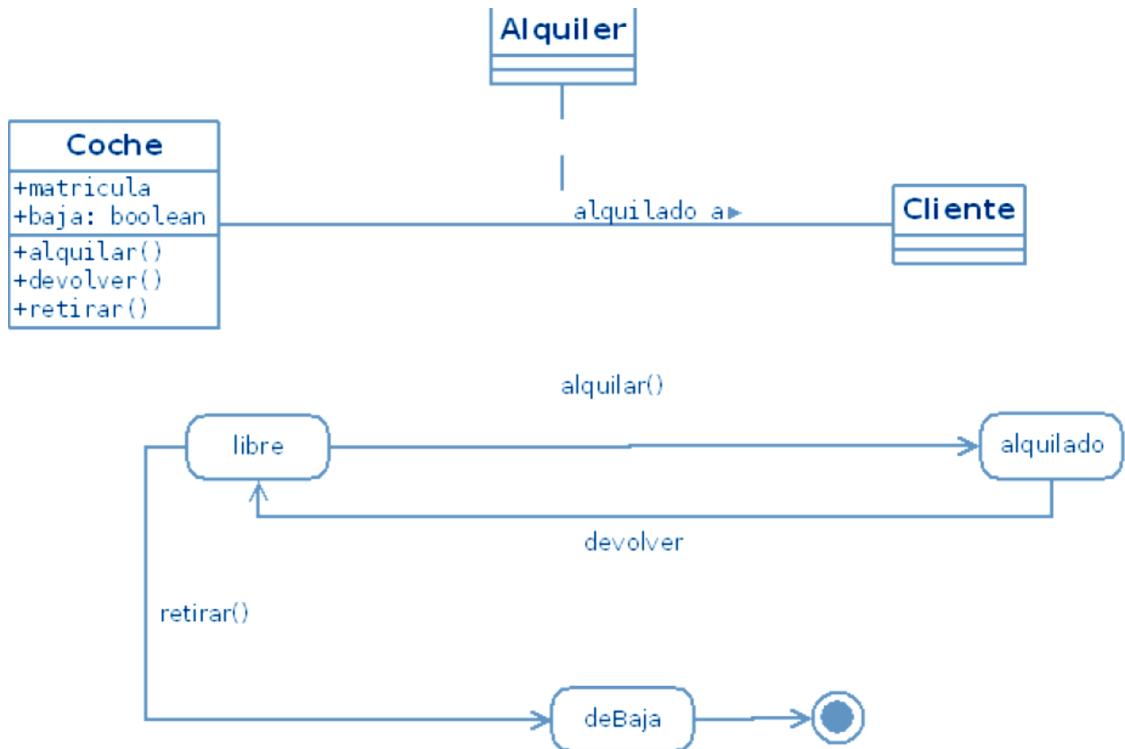


Diagrama de estado - Ascensor









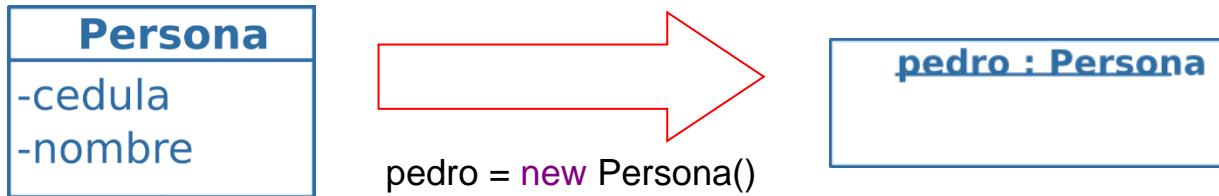
DISEÑO DE SISTEMAS DE INFORMACIÓN

Diagrama de secuencia

Sesión S6

Diagramas de Secuencia

- Los Diagramas de Secuencias muestran la forma en que *un grupo de objetos se comunican* (interactúan) entre sí a lo largo del *tiempo*



Elementos

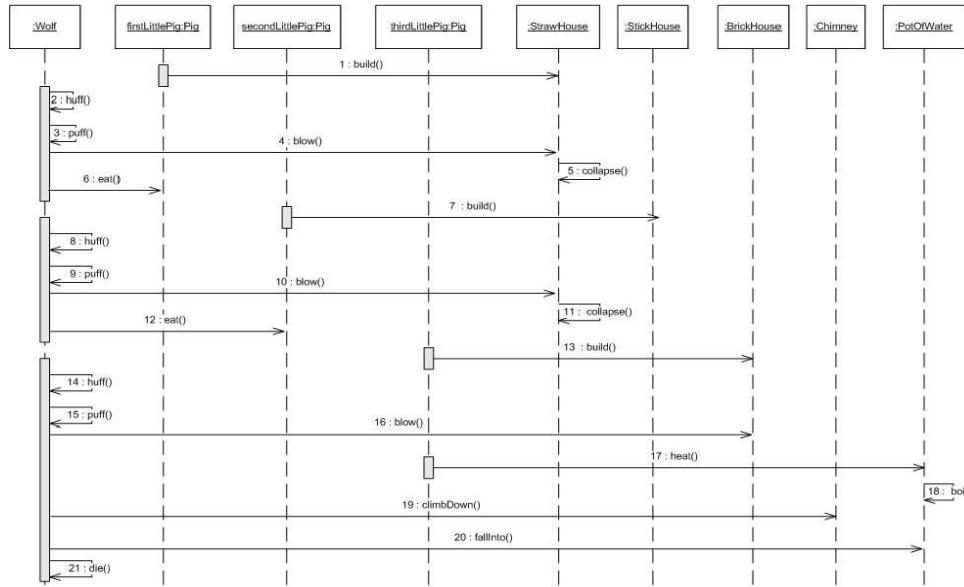
Tipo de Nodo	Notación	Descripción
Marco		Provee un borde visual para el diagrama de secuencias
Línea de Vida		Representa un participante individual en una interacción
Actor		Representa el papel desempeñado por un usuario
Mensaje		Define una comunicación particular entre líneas de vida de una interacción
Fragmento combinado		Describe una interacción reutilizable.

Mensajes



- **Un mensaje síncrono** implica que el transmisor espera hasta que el mensaje es recibido y recibe una confirmación satisfactoria de la recepción desde el receptor. Por lo tanto, el transmisor se detiene esperando por una operación completa y satisfactoria.
- **Un mensaje asíncrono** no necesita esperar por una confirmación o recepción satisfactoria. Entonces, el transmisor continúa su ejecución.

Diagramas de Secuencia



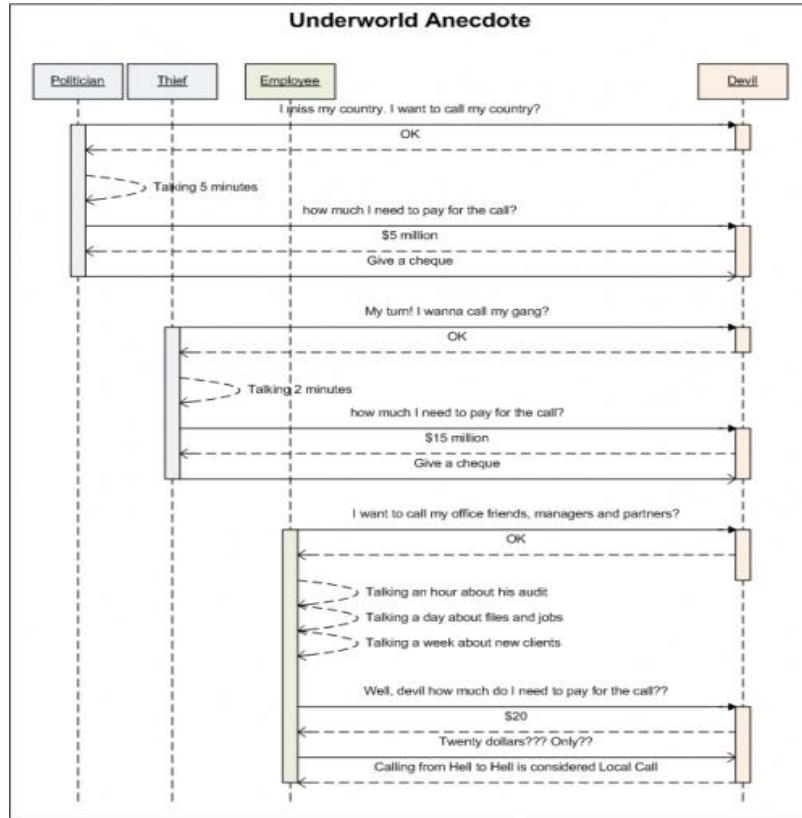
¿diagrama de secuencias
con la fábula de los tres cerditos?

(Gracias Ken Howard)

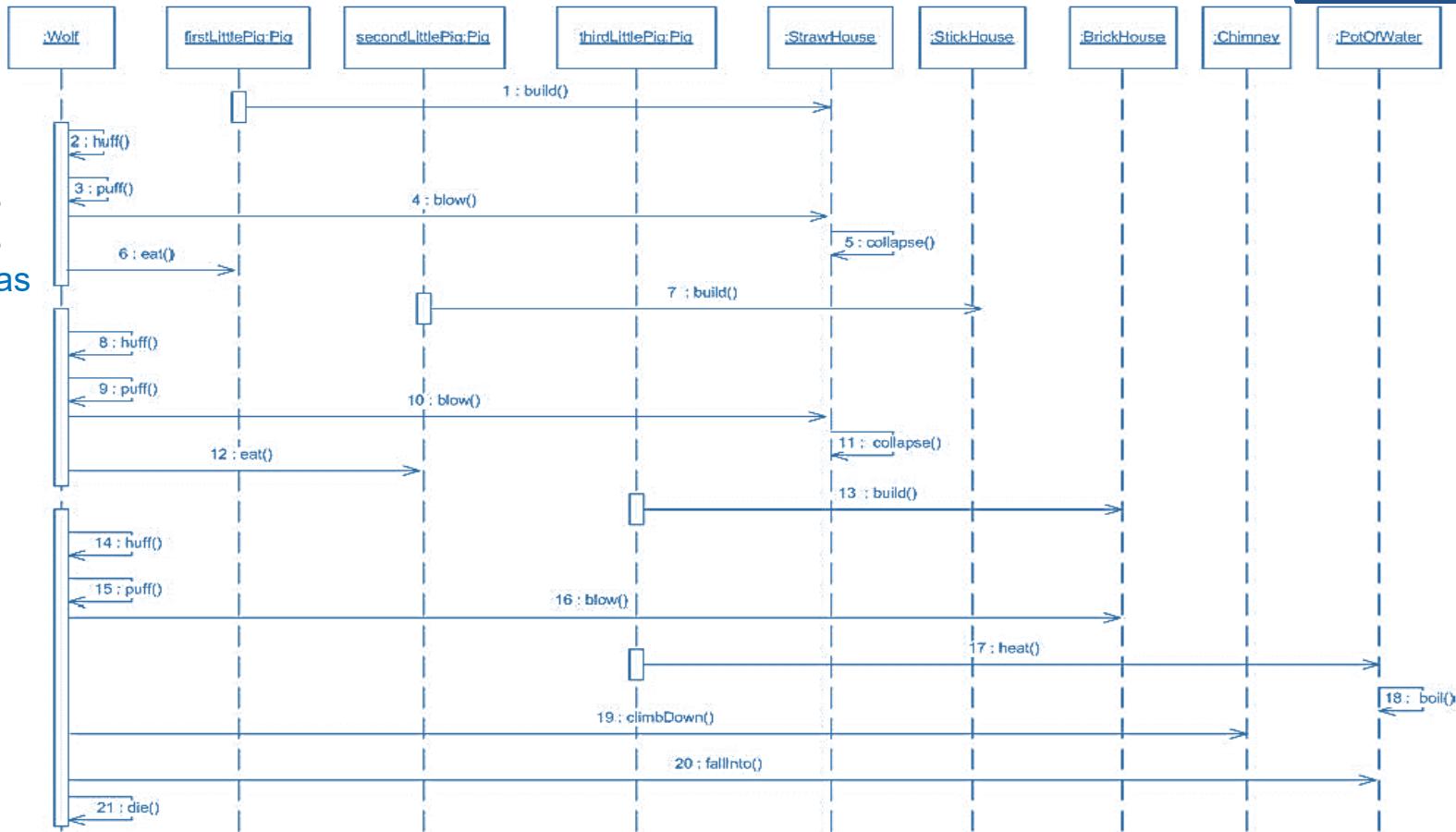
<http://kenhoward01.blogspot.com/2008/06/three-little-pigs-in-uml.html>

Diagramas de Secuencia

Los diagramas
de Secuencias
“cuentan” historias



Diagramas de Secuencia

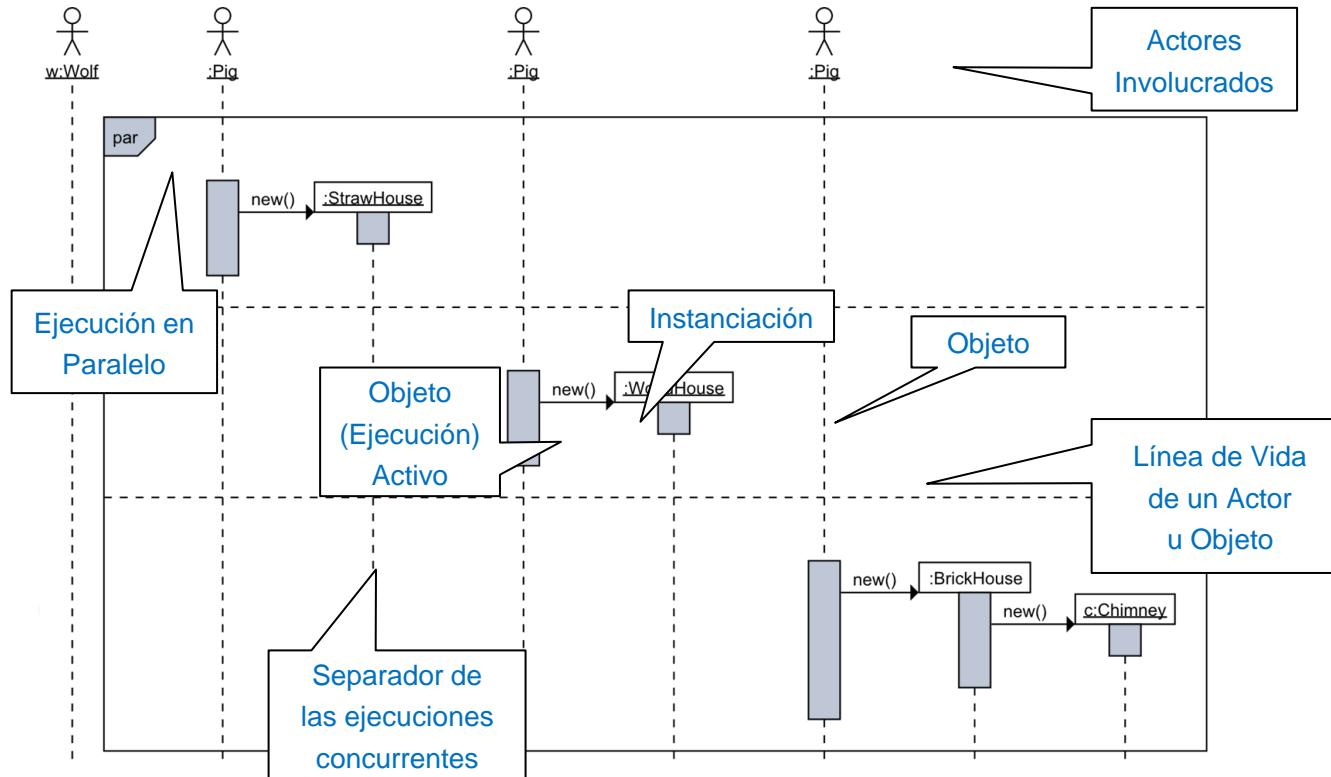


Los diagramas
de Secuencias
“cuentan” historias

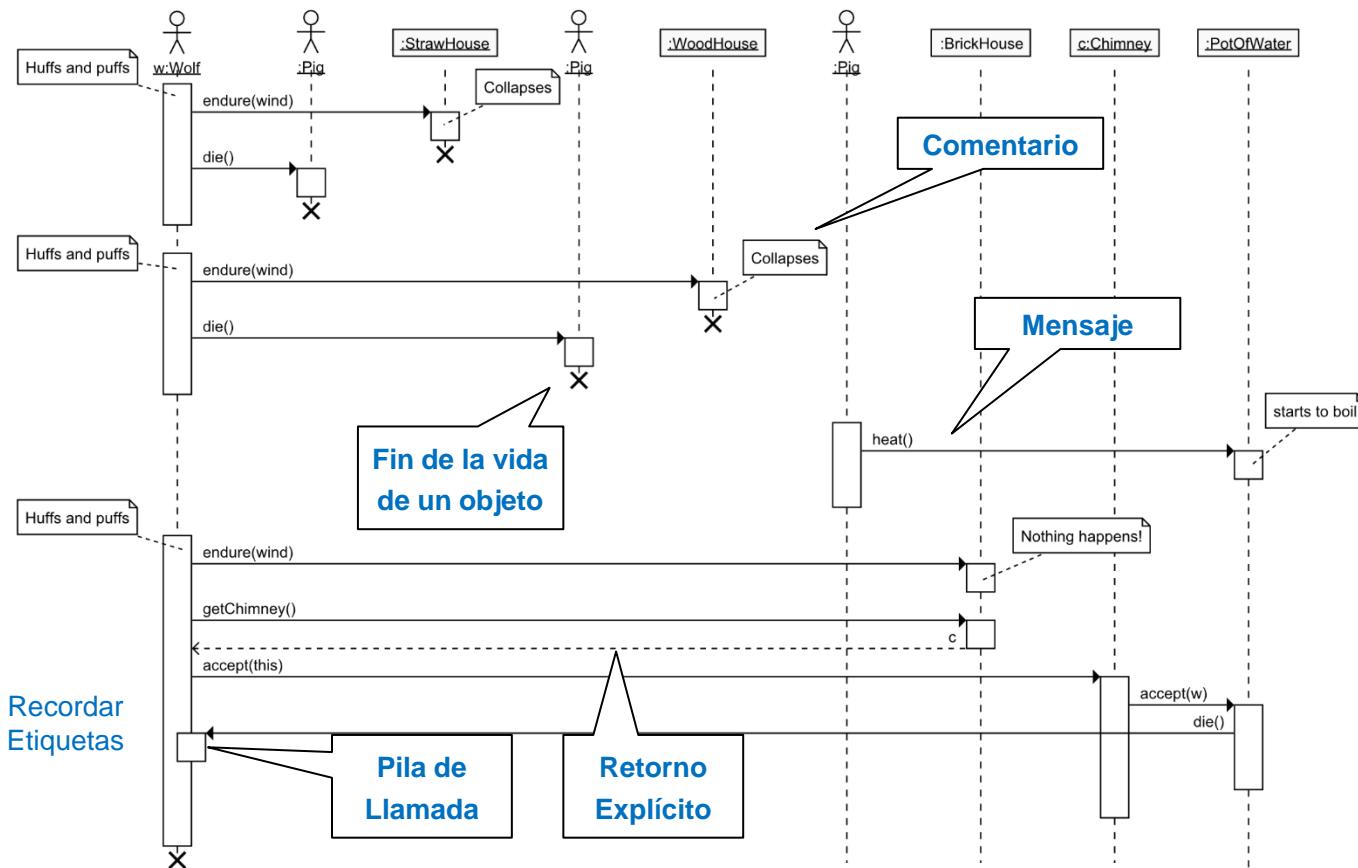


Fuente: <http://kenhoward01.blogspot.com/2008/06/three-little-pigs-in-uml.html>

Diagramas de Secuencia



Fuente: <http://www.tracemodeler.com/articles/pimp-my-diagram-three-little-pigs/>



Diagramas de Secuencia

Flujo Normal:

- 1.- El actor pulsa sobre el botón para crear un nuevo mensaje.
- 2.- El sistema muestra una caja de texto para introducir el título del mensaje y una zona de mayor tamaño para introducir el cuerpo del mensaje.
- 3.- El actor introduce el título del mensaje y el cuerpo del mismo.
- 4.- El sistema comprueba la validez de los datos y los almacena.
- 5.- El moderador recibe una notificación de que hay un nuevo mensaje.
- 6.- El moderador acepta y el sistema publica el mensaje si éste fue aceptado por el moderador.

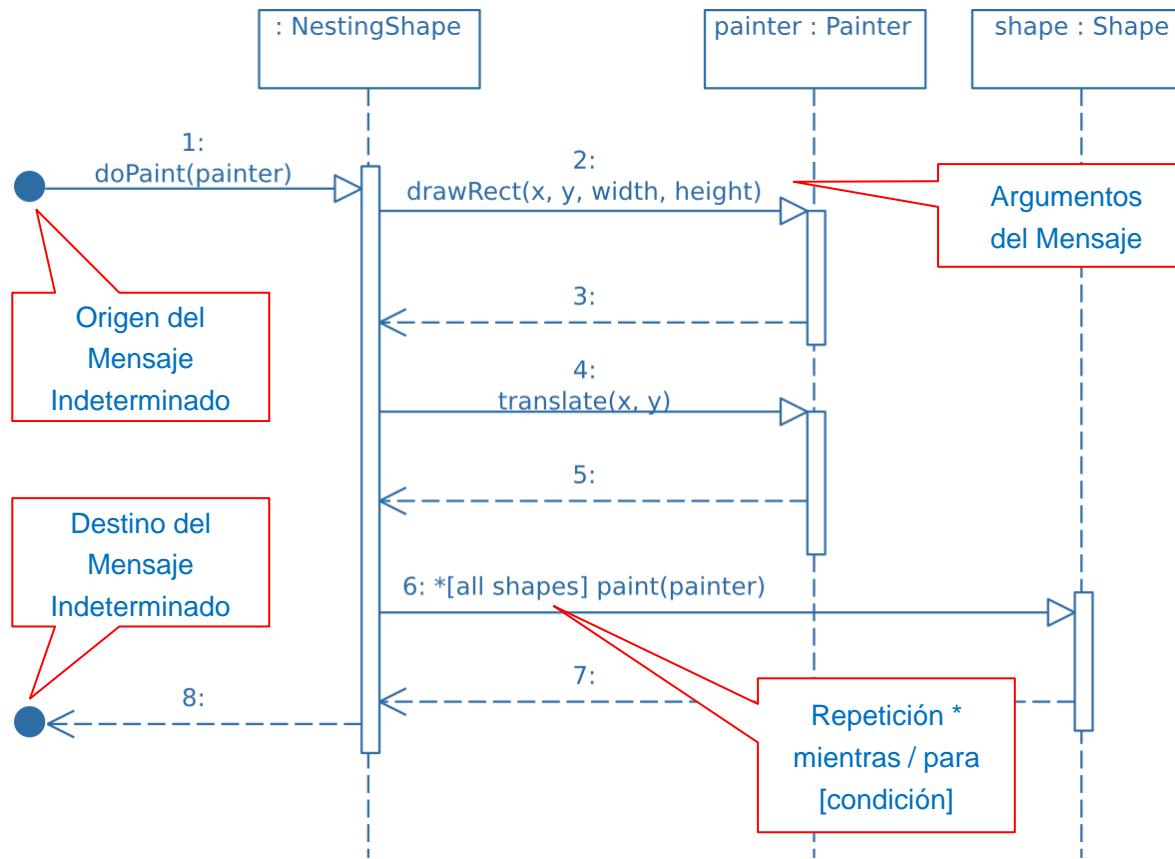
Flujo Alternativo:

- 4.A.- El sistema comprueba la validez de los datos, si los datos no son correctos, se avisa al actor de ello permitiéndole que los corrija.
- 7.B.- El moderador rechaza el mensaje, de modo que no es publicado sino devuelto al usuario.

Diagramas de Secuencia - Implementación

```
protected void doPaint(Painter painter) {  
    painter.drawRect(x, y, width, height);  
  
    // Cause painting of shapes to be relative to this shape  
    painter.translate(x, y);  
  
    for (Shape s : shapes) {  
        s.paint(painter);  
    }  
}
```

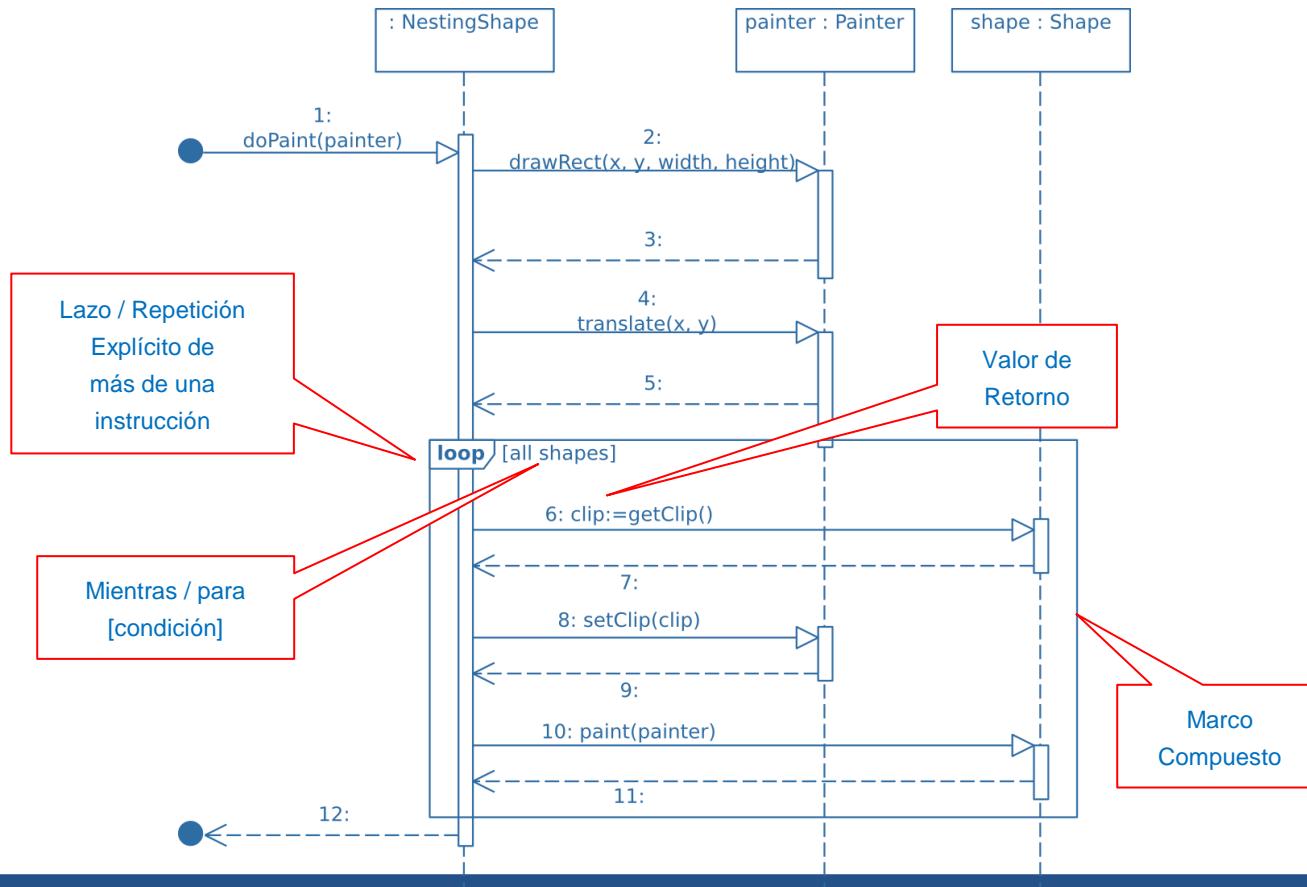
Diagramas de Secuencia - Implementación



Diagramas de Secuencia - Implementación

```
protected void doPaint(Painter painter, Config config) {  
    painter.drawRect(x, y, width, height);  
  
    // Cause painting of shapes to be relative to this shape  
    painter.translate(x, y);  
  
    for (Shape s : shapes) {  
        Rectangle clip = s.getClip();  
        painter.setClip(clip);  
        s.paint(painter);  
    }  
  
    // Restore graphics origin  
    painter.translate(-x, -y);  
}
```

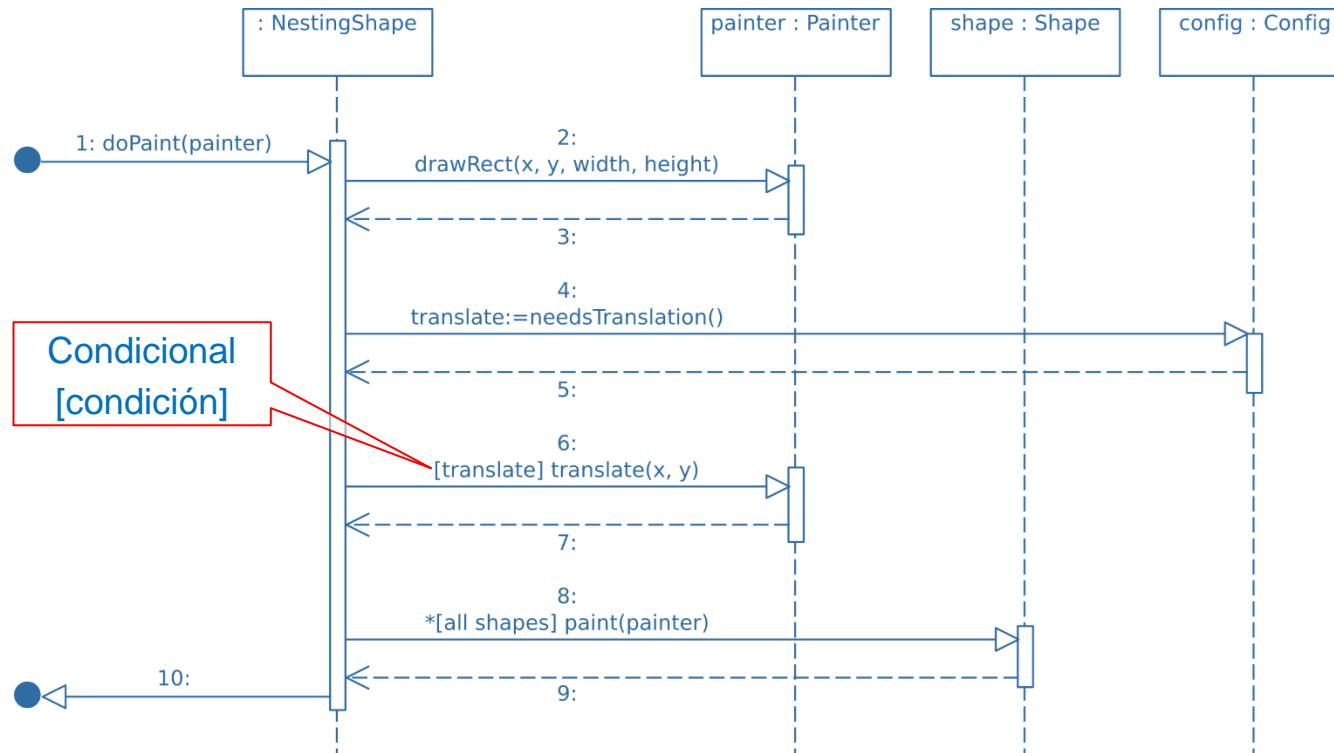
Diagramas de Secuencia - Implementación



Diagramas de Secuencia - Implementación

```
protected void doPaint(Painter painter, Config config) {  
    painter.drawRect(x, y, width, height);  
  
    // Cause painting of shapes to be relative to this shape  
    boolean translate = config.needsTranslation();  
  
    if (translate) {  
        painter.translate(x, y);  
    }  
  
    for (Shape s : shapes) {  
        s.paint(painter);  
    }  
}
```

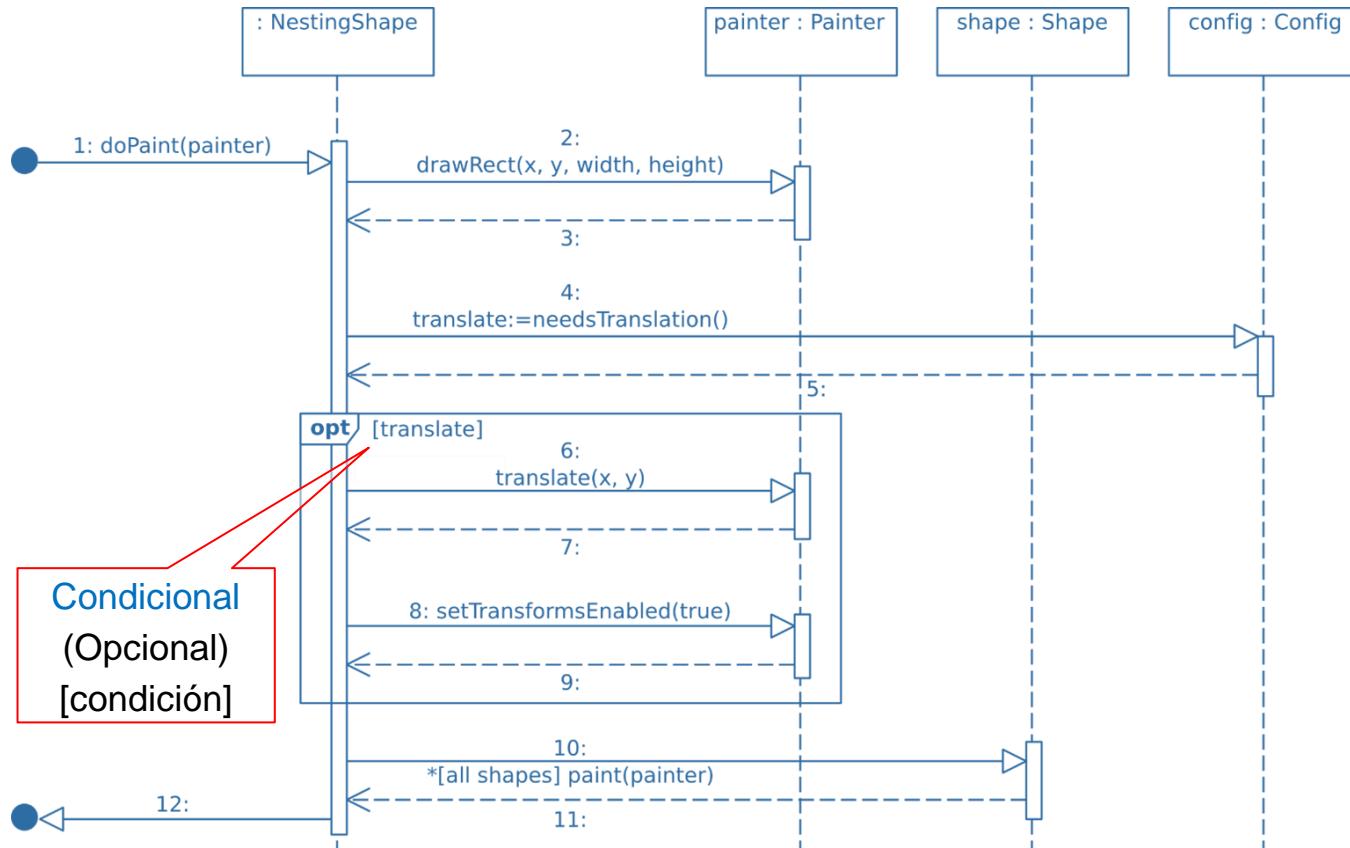
Diagramas de Secuencia - Implementación



Diagramas de Secuencia - Implementación

```
protected void doPaint(Painter painter, Config config) {  
    painter.drawRect(x, y, width, height);  
  
    // Cause painting of shapes to be relative to this shape  
    boolean translate = config.needsTranslation();  
  
    if (translate) {  
        painter.setTransformsEnabled(true);  
        painter.translate(x, y);  
    }  
  
    for (Shape s : shapes) {  
        s.paint(painter);  
    }  
}
```

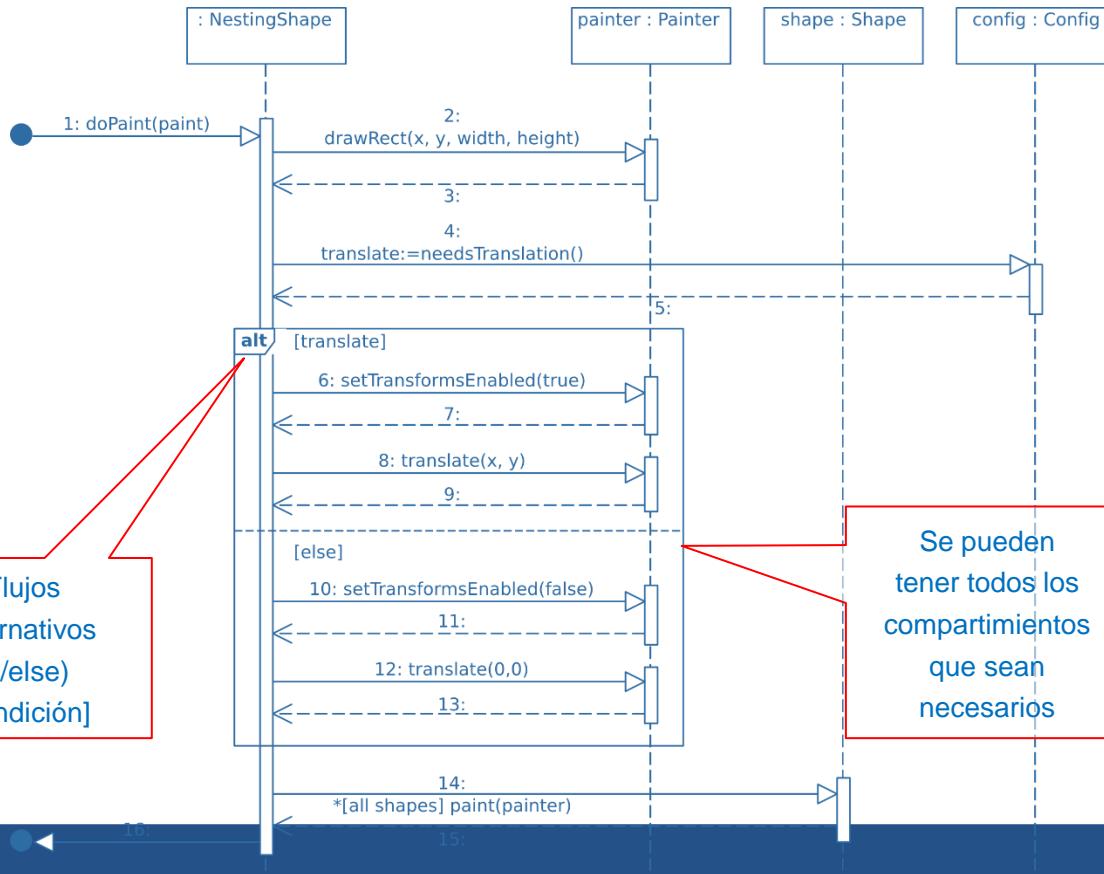
Diagramas de Secuencia - Implementación



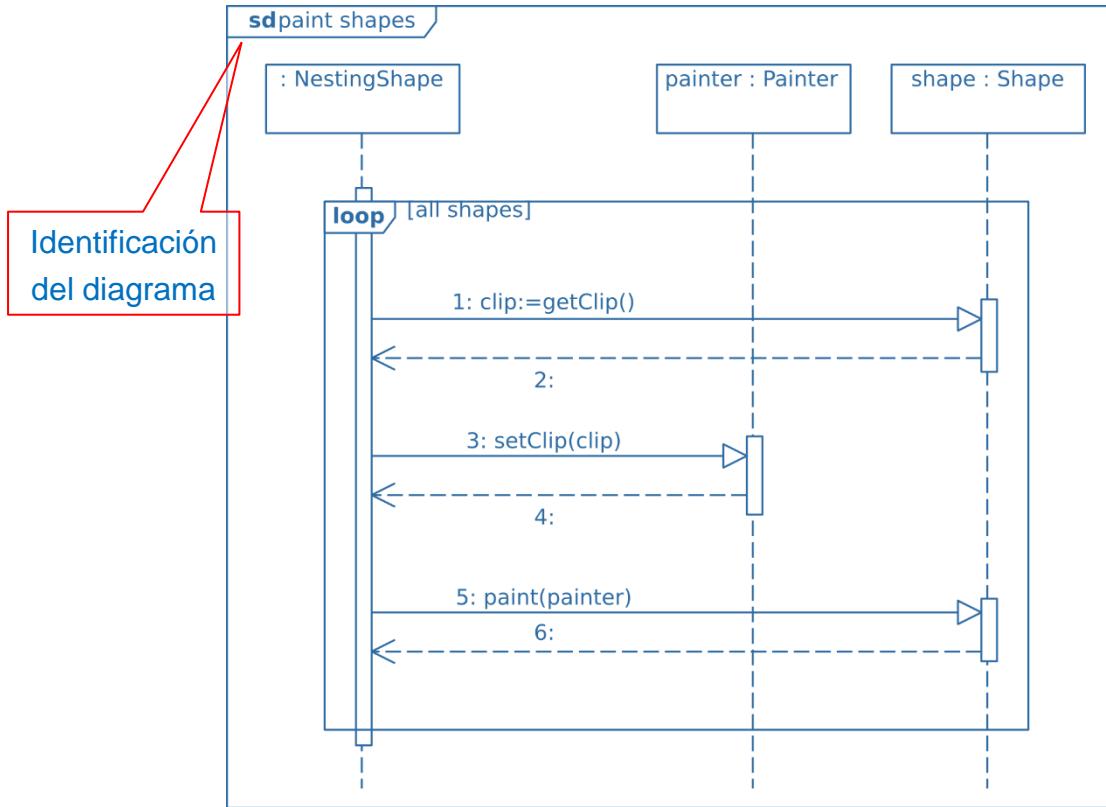
Diagramas de Secuencia - Implementación

```
protected void doPaint(Painter painter, Config config) {  
    painter.drawRect(x, y, width, height);  
  
    // Cause painting of shapes to be relative to this shape  
    boolean translate = config.needsTranslation();  
  
    if (translate) {  
        painter.setTransformsEnabled(true);  
        painter.translate(x, y);  
    } else {  
        painter.setTransformsEnabled(false);  
        painter.translate(0, 0);  
    }  
  
    for (Shape s : shapes) {  
        s.paint(painter);  
    }  
}
```

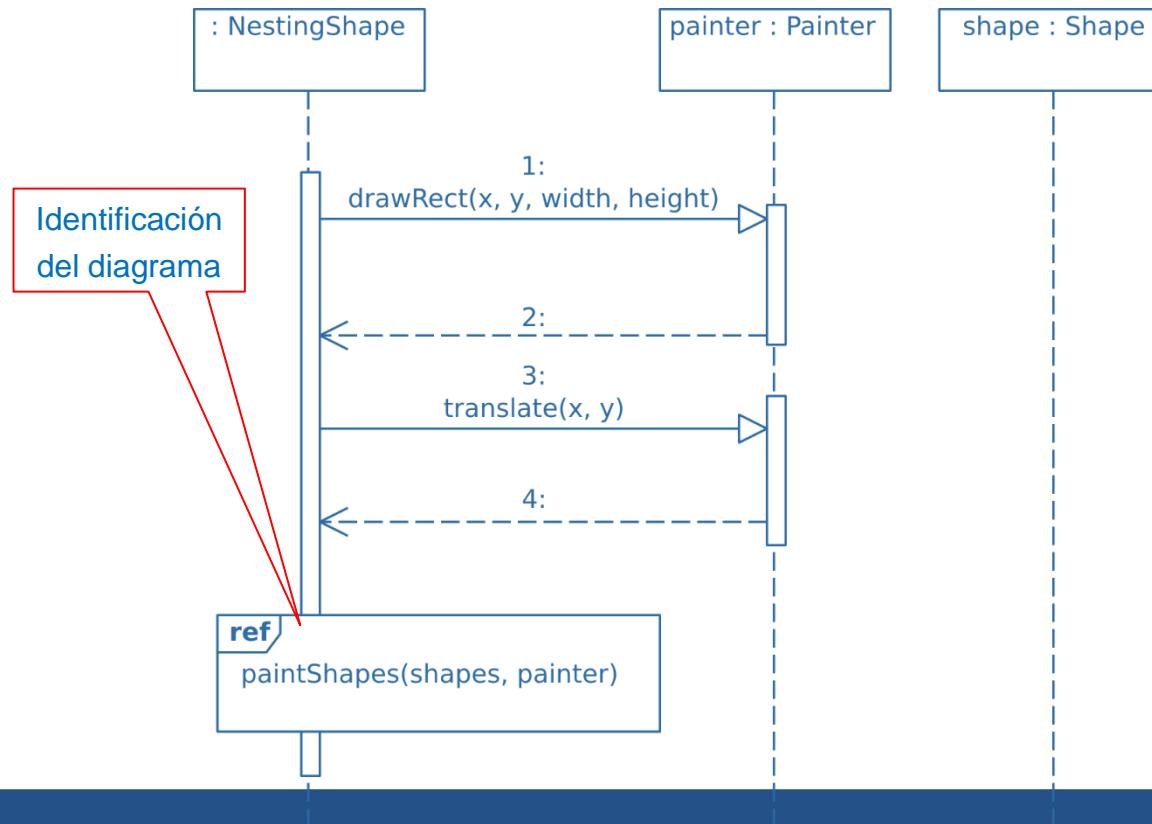
Diagramas de Secuencia - Implementación



Diagramas de Secuencia - Implementación



Diagramas de Secuencia - Implementación





DISEÑO DE SOFTWARE

Diagrama de Componente

Sesión S7

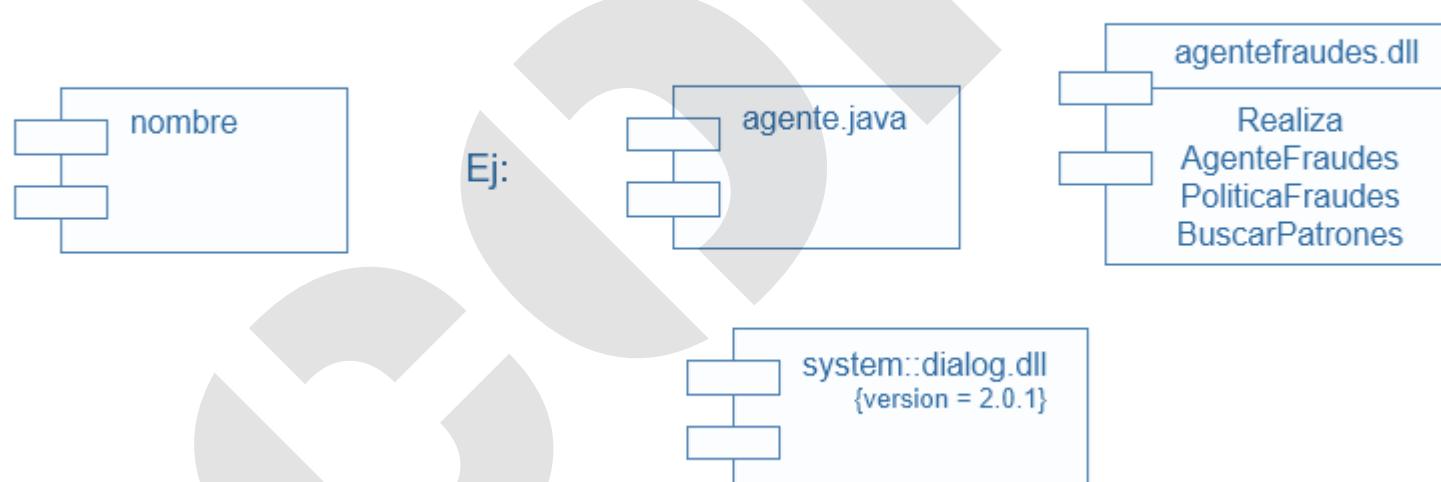
Diagramas de Componentes

- El modelo de componentes se centra en la organización física del sistema.
- Primero, se decidirá como las clases identificadas serán organizadas en librerías.
- Luego, se identificarán las librerías ejecutables, librerías de enlace dinámico y otros archivos necesarios en tiempo de ejecución.
- El único tipo de relación que existe entre componentes es la dependencia.
- Ejemplos:
 - .cpp, .exe, .java, .class

Diagramas de Componentes

Definición

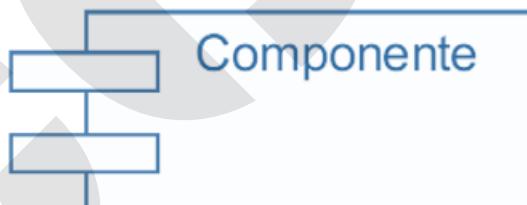
Un *componente* es una parte física y reemplazable de un sistema.



Diagramas de Componentes

Componente Genérico

- Este ícono representa un módulo de software dentro de una interfase bien definida. Dentro del tab de especificación se debe identificar el tipo de componente: ActiveX, Applet, Application, DLL)

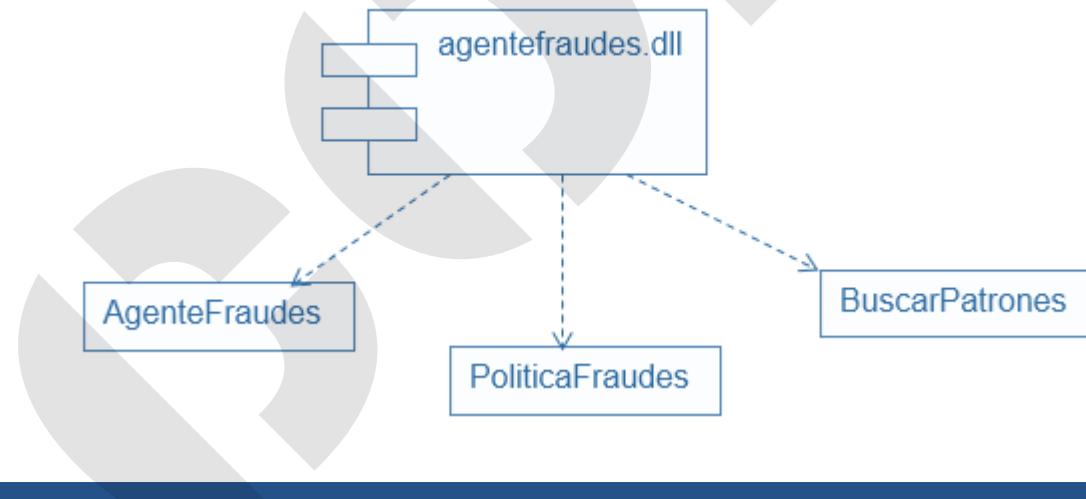


Diagramas de Componentes

Componentes y clases

Las clases representan abstracciones lógicas. Los componentes son elementos físicos del mundo real. Un componente es la implementación física de un conjunto de otros elementos lógicos, como clases y colaboraciones.

Ejemplo de un componente y las clases que implementa:



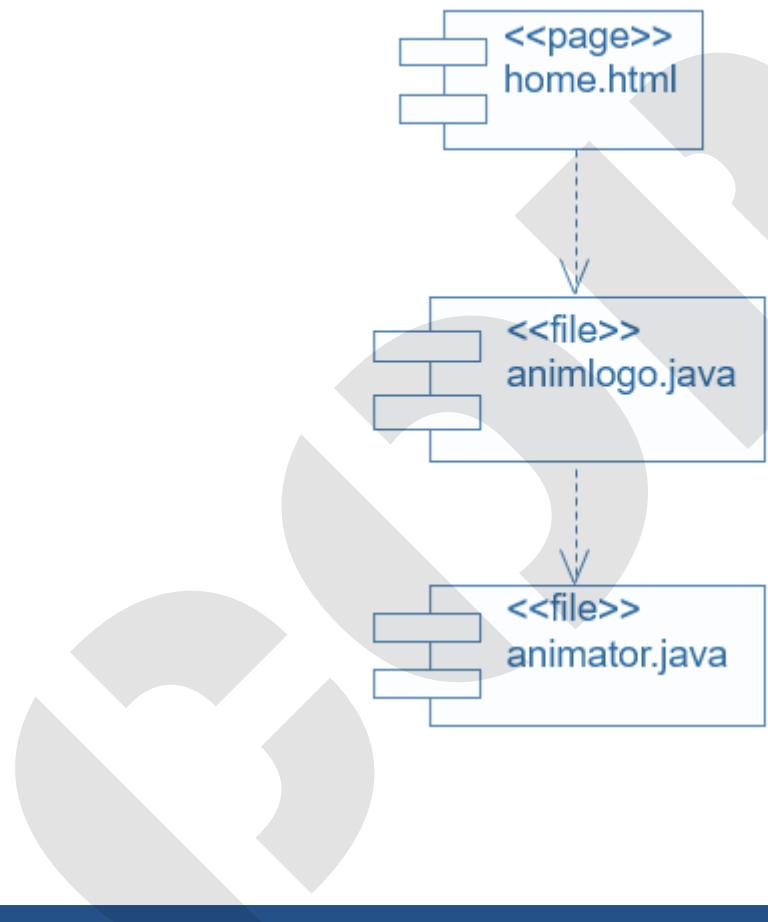
Diagramas de Componentes

Dependencias entre componentes

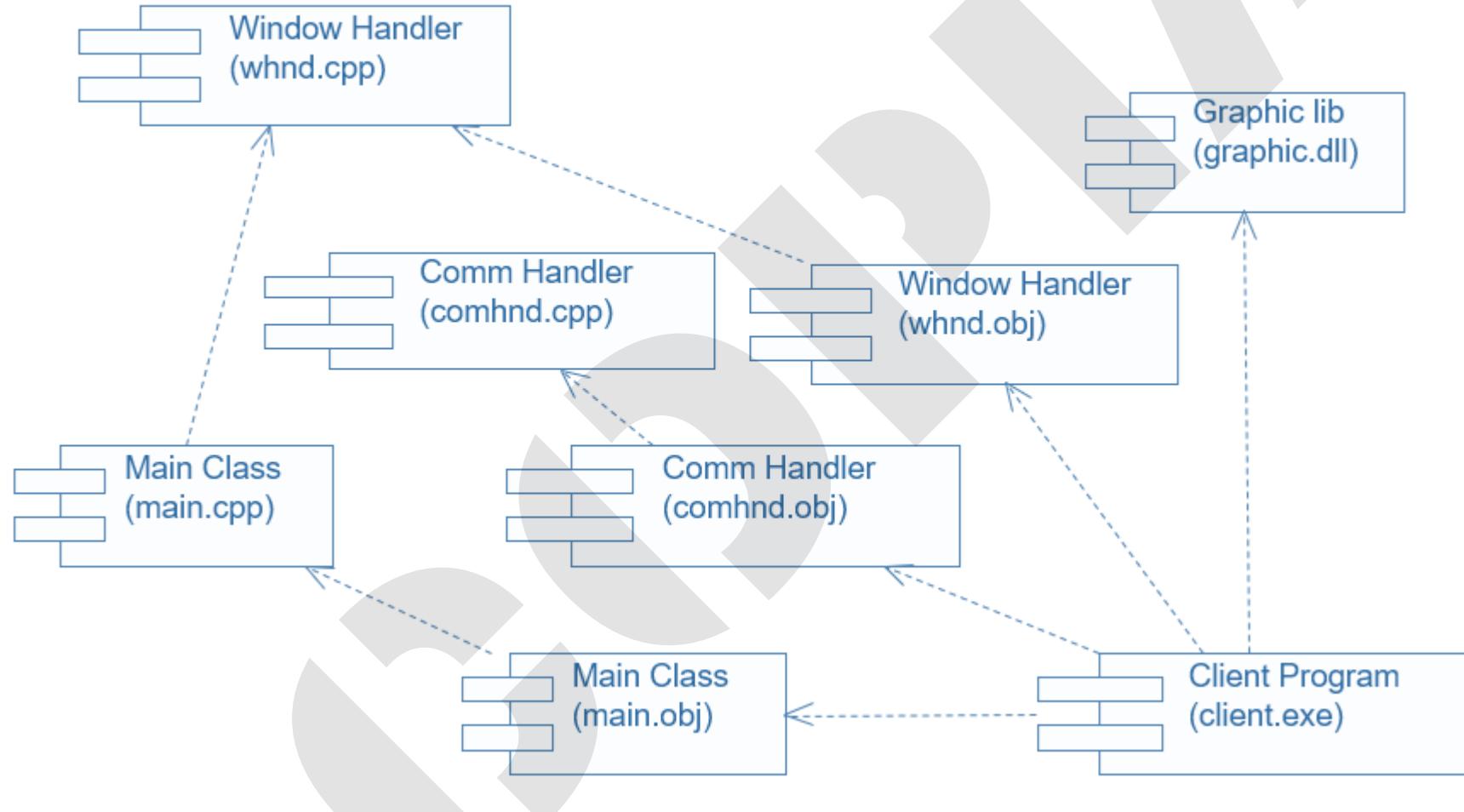
La *dependencia* entre dos componentes se muestra como una flecha punteada. La *dependencia* quiere decir que una componente necesita de la otra para completar su definición.

Ejemplos:

Diagramas de Componentes



Diagramas de Componentes



Diagramas de Componentes

Subprogramas

- Estos íconos representan las partes visibles de un subprograma y su implementación. Un subprograma es típicamente un conjunto de subrutinas y no contienen definición de clases.



Diagramas de Componentes

Programa principal

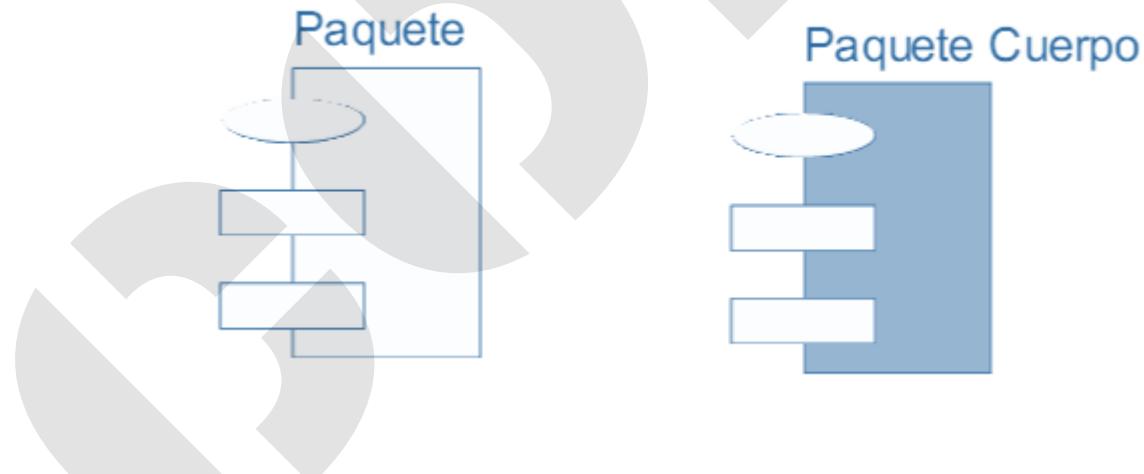
- Este ícono representa el programa principal. Contiene la raíz del sistema:



Diagramas de Componentes

Paquetes

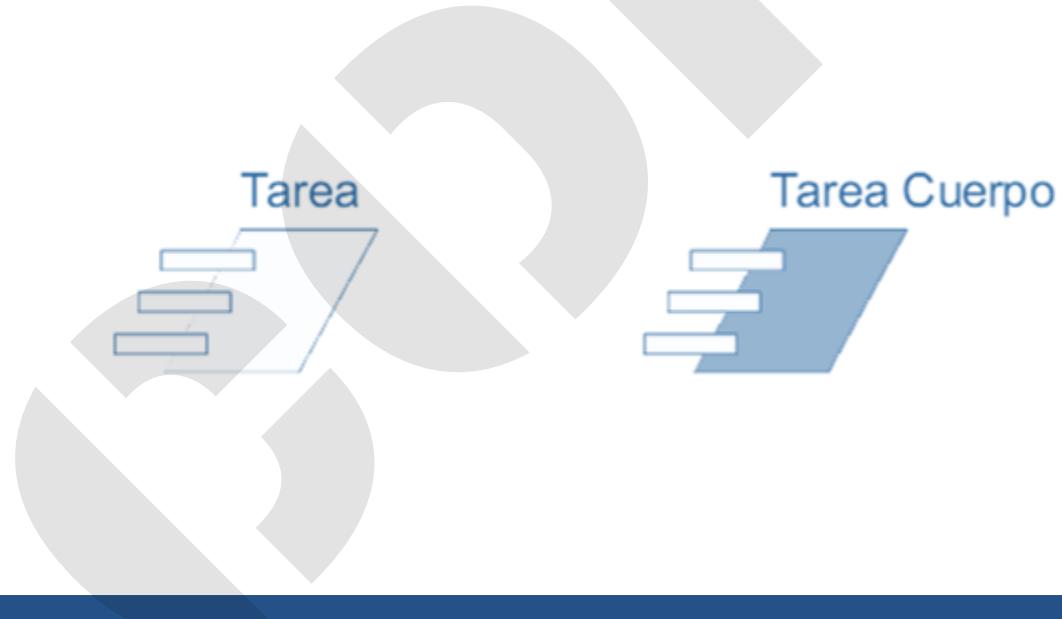
- Es la implementación de una clase. Es el archivo de cabecera que contiene las definiciones. En C++ son los archivo .h, en java representan los .java



Diagramas de Componentes

Tareas

- Representa a aquellos paquetes que tienen un hilo independiente de ejecución. Una archivo ejecutable .exe, es representado por el ícono de tarea:



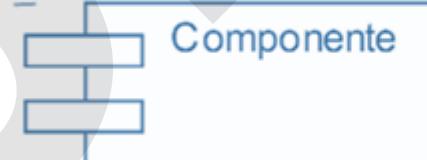
Diagramas de Componentes

Base de Datos

base datos

Interface

Interface



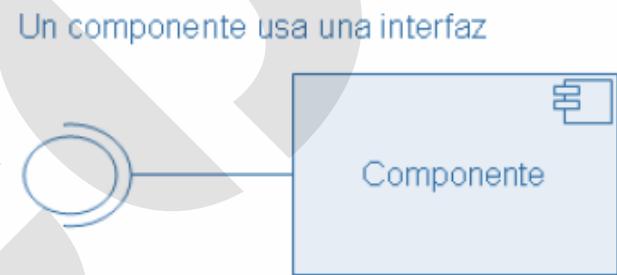
Interfaces

Es el lazo de unión entre varios componentes.



Interfaces

Las interfaces pueden representarse de varias formas, como vemos en la grafica:



Interfaces

Además se pueden representar de dos maneras de forma icónica y expandida.

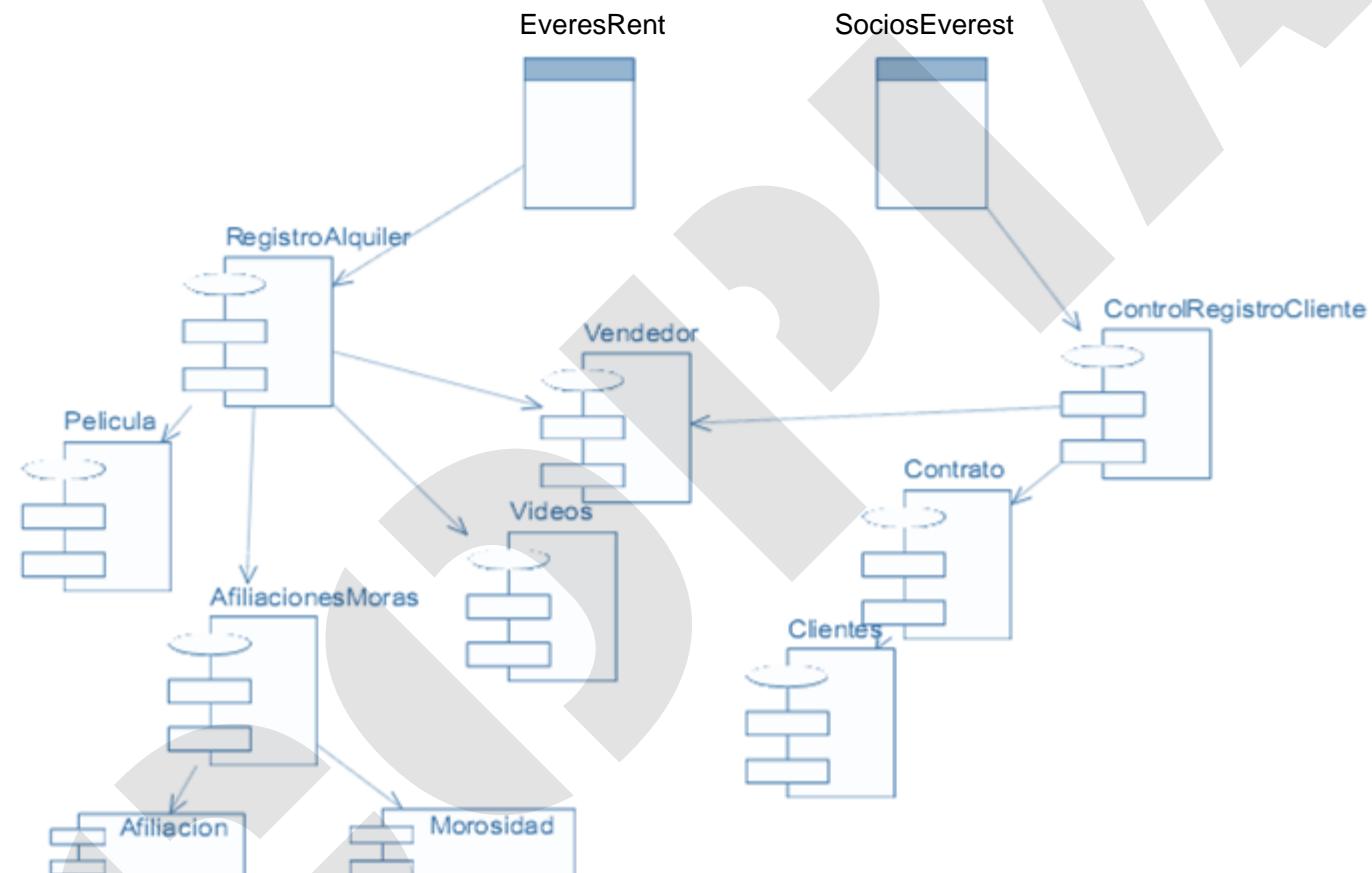
Forma icónica



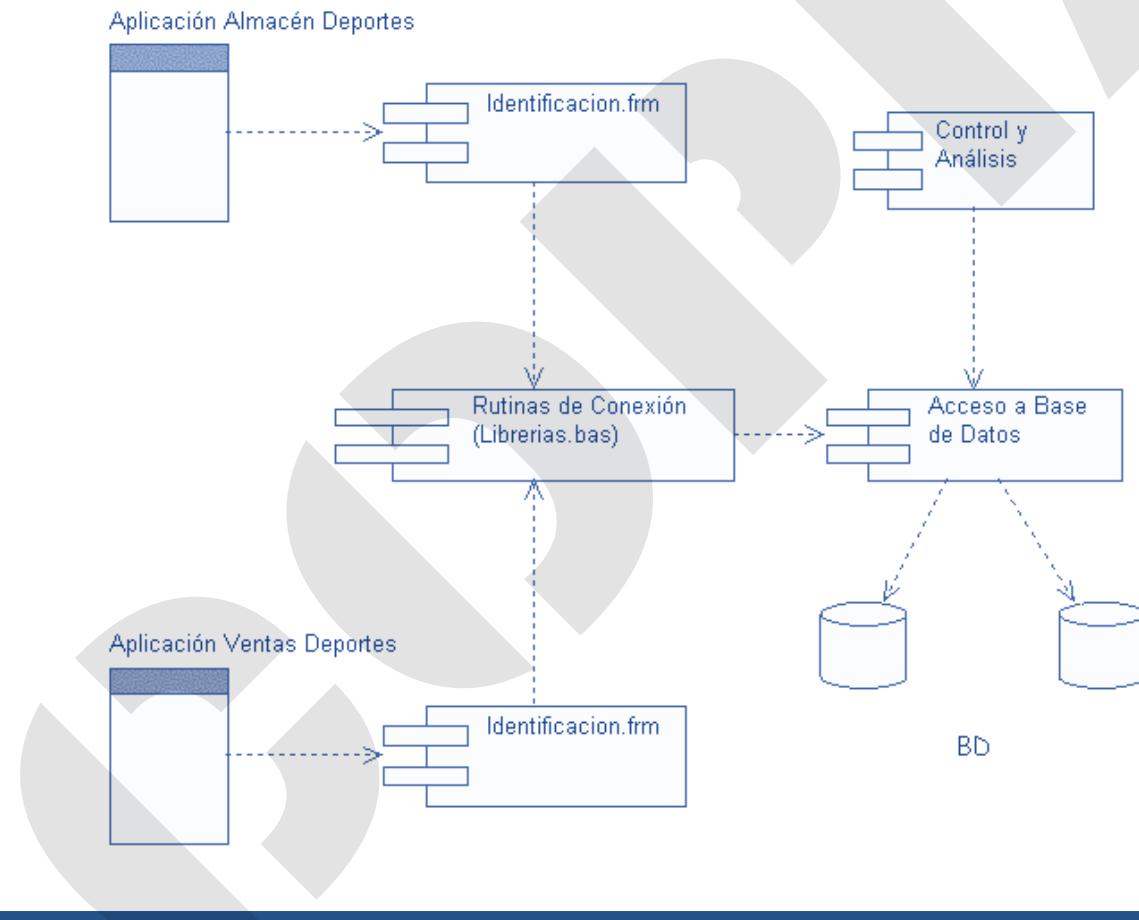
Forma expandida

<<Interface>>
ObservImagen
abortar: int
error: int
actI: Boolean

Ejemplo de Diagrama de componentes



Ejemplo de Diagrama de componentes



¿En que fase del ciclo de vida se encuentra?

Se presenta en el diseño que da paso a la implementación

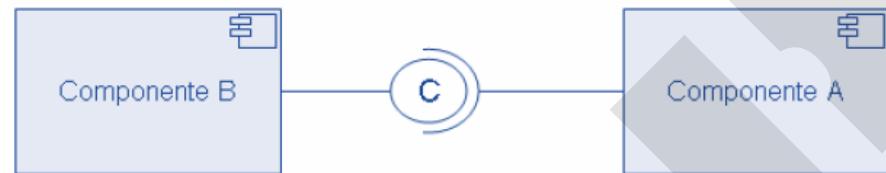
El diagrama de Componentes se genera a partir del diagrama de clases

Pasos para la elaboración de un diagrama de componentes

- previamente al diagrama de componentes debemos de tener hecho el diagrama de clases.
- Se debe identificar a todos las clases que participaran en el sistema o subsistema a desarrollar.
- Una vez identificado las clases, se procede a identificar sus métodos.
- Estos métodos pasaran a ser módulos con líneas de código independientes.
- Estos módulos serán los componentes de nuestro diagrama.
- Estos componentes se relacionan entre si por medio de sus interfaces.

Uso del Diagrama de Componentes

- Nos permite ver el modelado de un sistema o subsistema
- permite especificar un componente con interfaces bien definidas.



El componente A usa la interfaz que provee el componente B



El componente D puede reemplazar al componente B,
porque utiliza la misma interfaz que B.

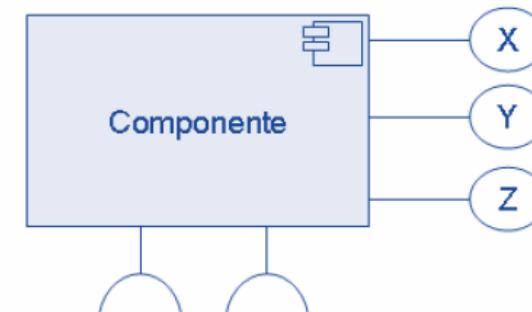


Diagrama de Componentes

Relación con otros diagramas

- Diagrama de despliegue

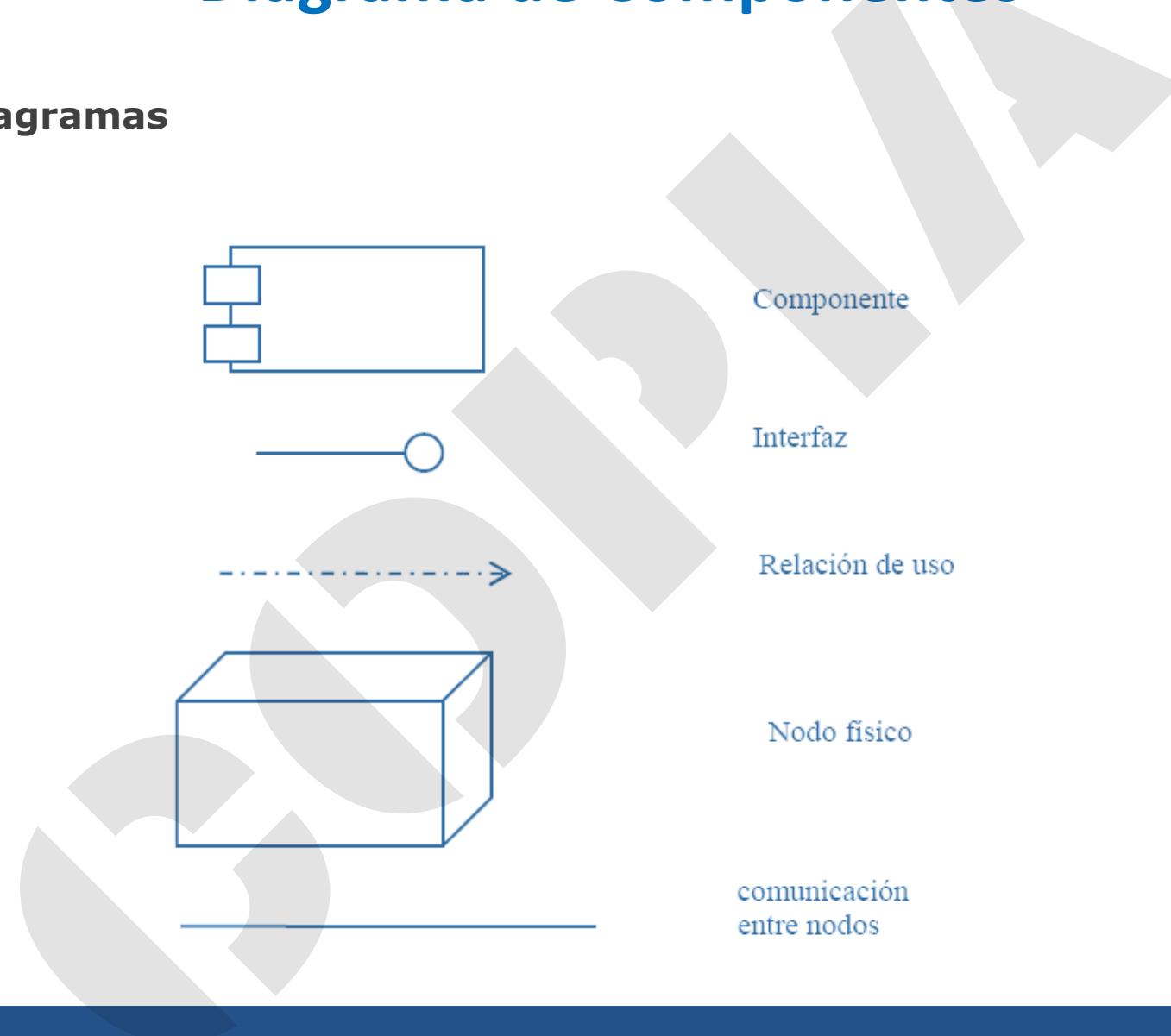
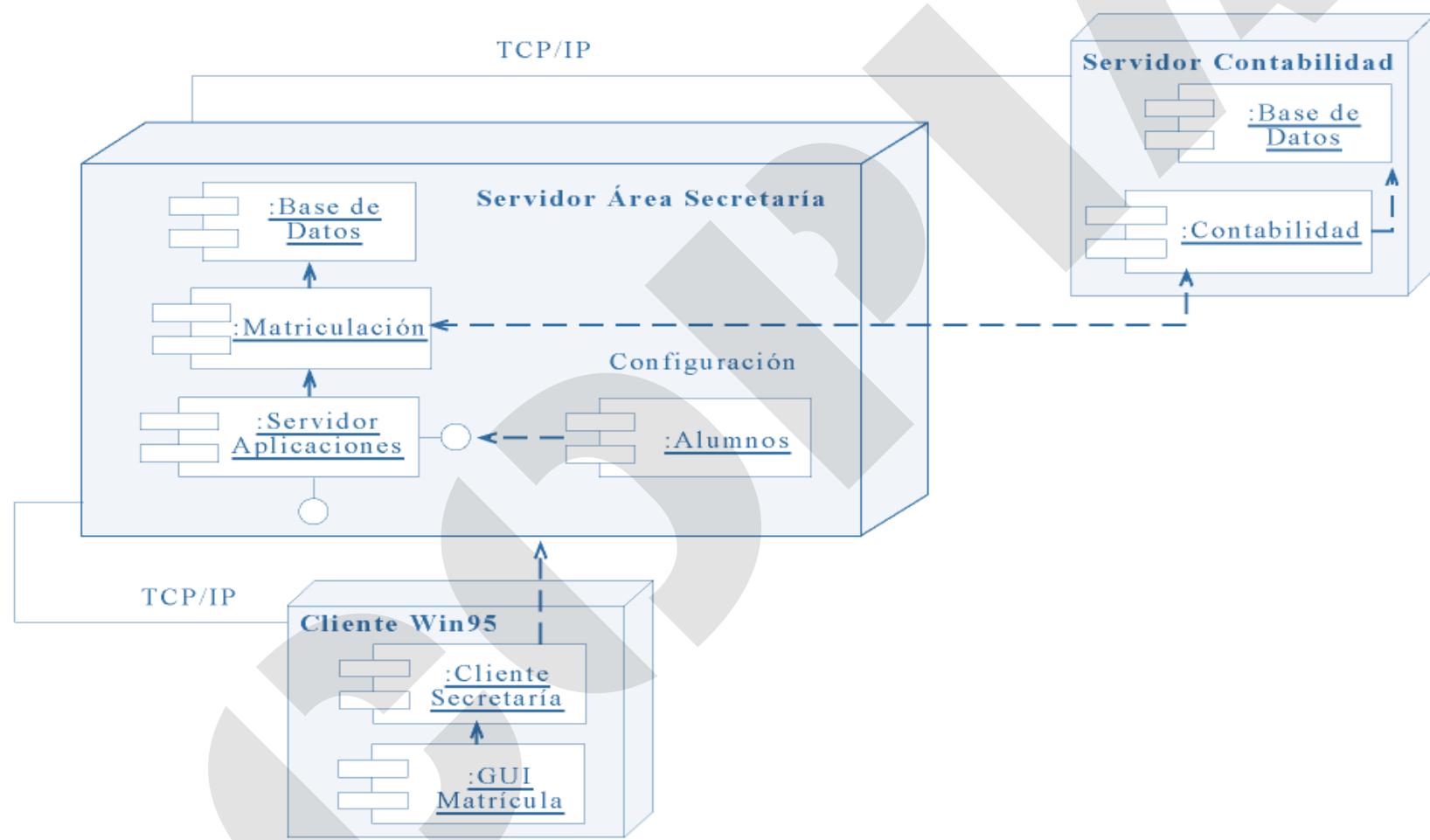


Diagrama de Componentes y despliegue



Relación con diagrama de clases

Métodos de la clase pasan a ser módulos

Módulos pasan a ser componentes.

Nombre
Atributo
Métodos

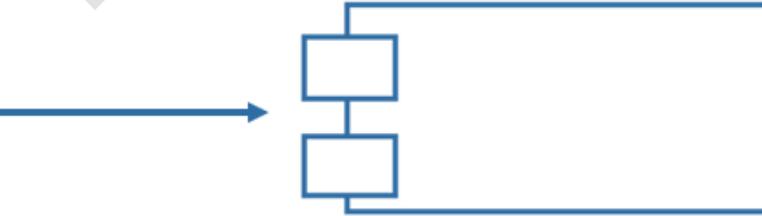


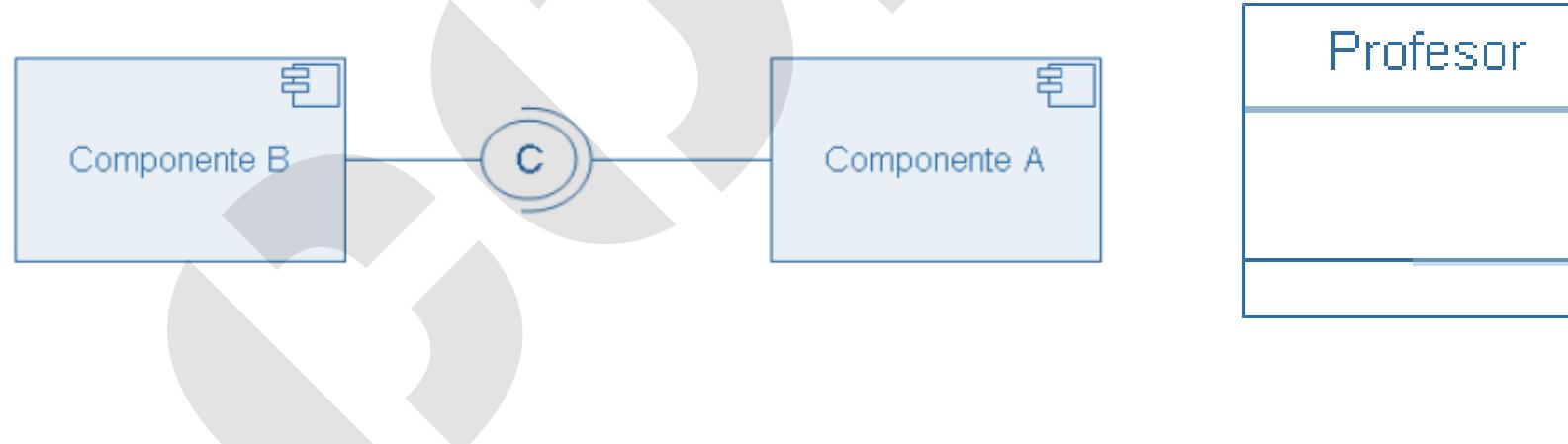
Diagrama de Componentes

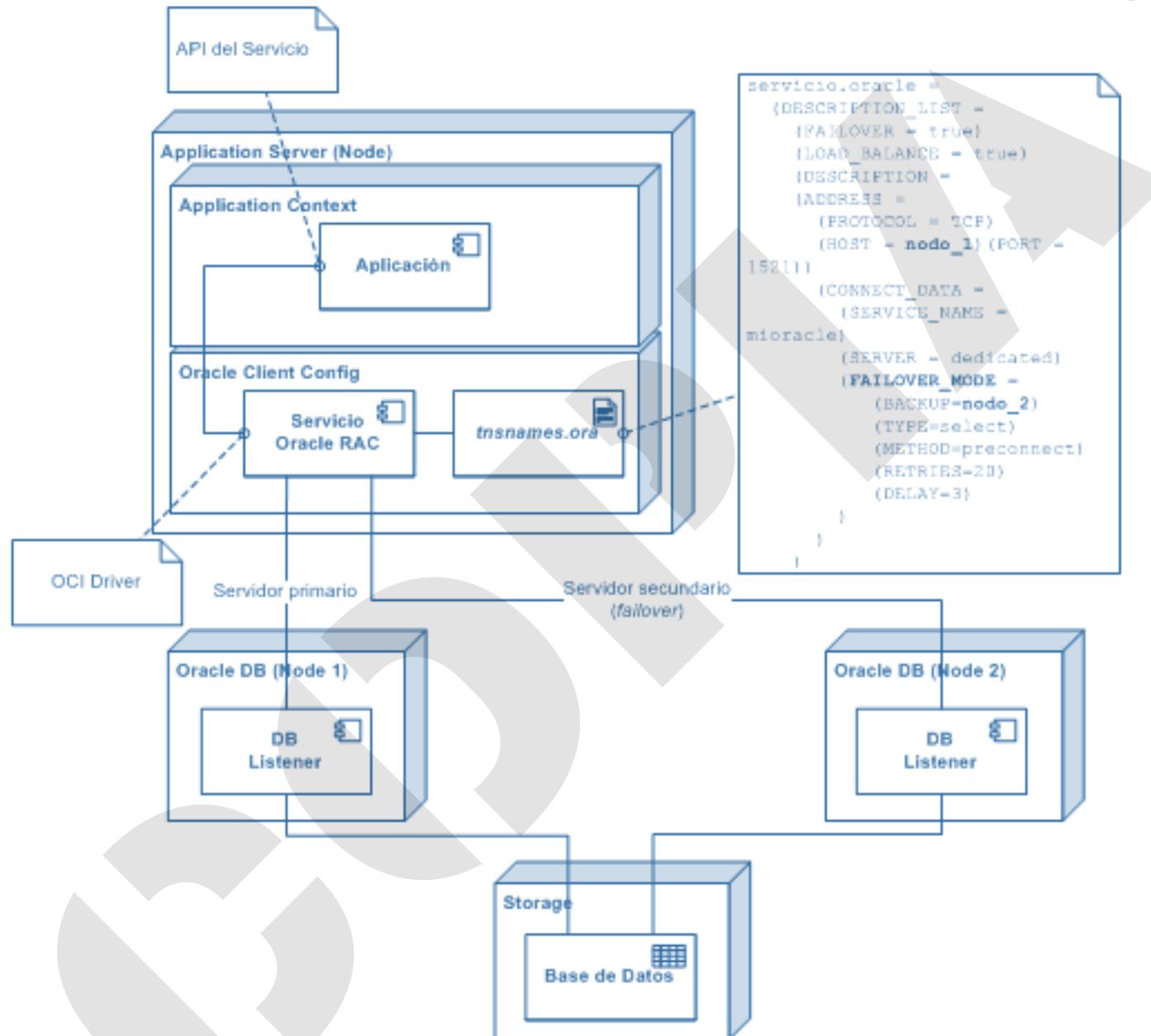
Diferencias:

Un componente representa un elemento físico (bits). Una clase es una abstracción lógica.

El componente se puede representar en nodos físicos, la clase no.

Las operaciones de un componente solo se alcanzan a través de interfaces. Las de una clase podrían ser accesibles directamente.







DISEÑO DE SISTEMAS DE INFORMACIÓN

Diagrama de Despliegue

Sesión S7

Diagrama de Distribución

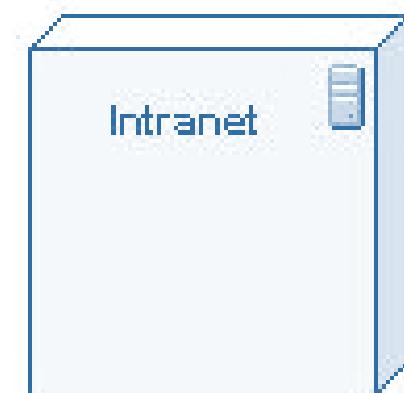
- Muestra los recursos físicos de un sistema incluyendo componentes, nodos y conexiones
- Presenta la distribución de componentes de software en nodos donde ejecutan, y las asociaciones entre los nodos
- Las asociaciones representan conexiones físicas entre nodos estereotipadas por protocolos de la comunicación, ej. TCP/IP, HTTP.

Diagramas de Despliegue

- Describen la arquitectura física del sistema durante la ejecución, en términos de:
 - procesadores
 - dispositivos
 - componentes de software
- Describen la topología del sistema: la estructura de los elementos de hardware y el software que ejecuta cada uno de ellos.

Elementos del Diagrama de Despliegue

- Los **nodos** son objetos físicos que existen en tiempo de ejecución, y que representan algún tipo de recurso computacional (capacidad de memoria y procesamiento):
 - Computadores con procesadores
 - Otros dispositivos
 - impresoras
 - lectoras de códigos de barras
 - dispositivos de comunicación



Elementos del Diagrama de Despliegue

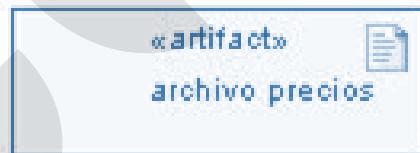
Componentes

Los componentes son las aplicaciones que se ejecutan en los nodos.



Artefactos

Un artefacto es un producto del proceso de desarrollo de software.



Elementos del Diagrama de Despliegue

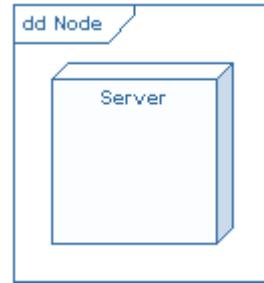
Asociaciones

Las asociaciones son las conexiones que unen los nodos y representan la comunicación entre los nodos.

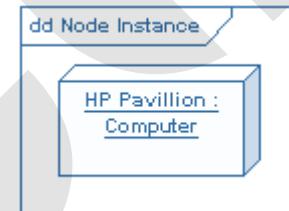
- Estas asociaciones indican:
 - Algún tipo de ruta de comunicación entre los nodos
 - Los nodos intercambian objetos o envían mensajes a través de esta ruta
- El tipo de comunicación se identifica con un estereotipo que indica el protocolo de comunicación o la red.

Elementos del Diagrama de Despliegue

Nodo:

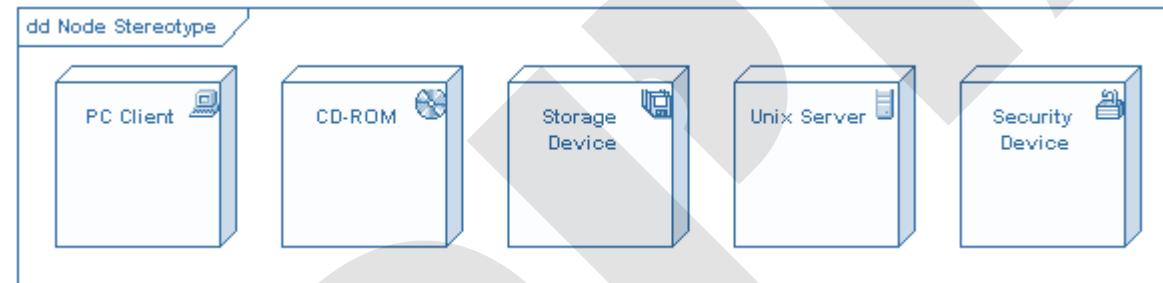


Instancia de Nodo: Se puede distinguir desde un nodo por el hecho de que su nombre esta subrayado y tiene dos puntos antes del tipo de nodo base. Una instancia puede o no tener un nombre antes de los dos puntos.

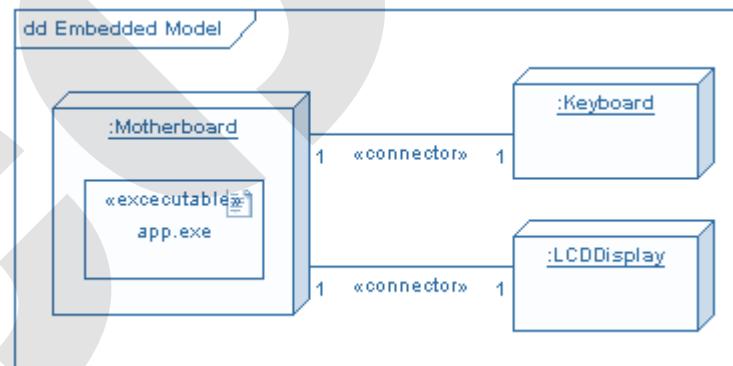


Elementos del Diagrama de Despliegue

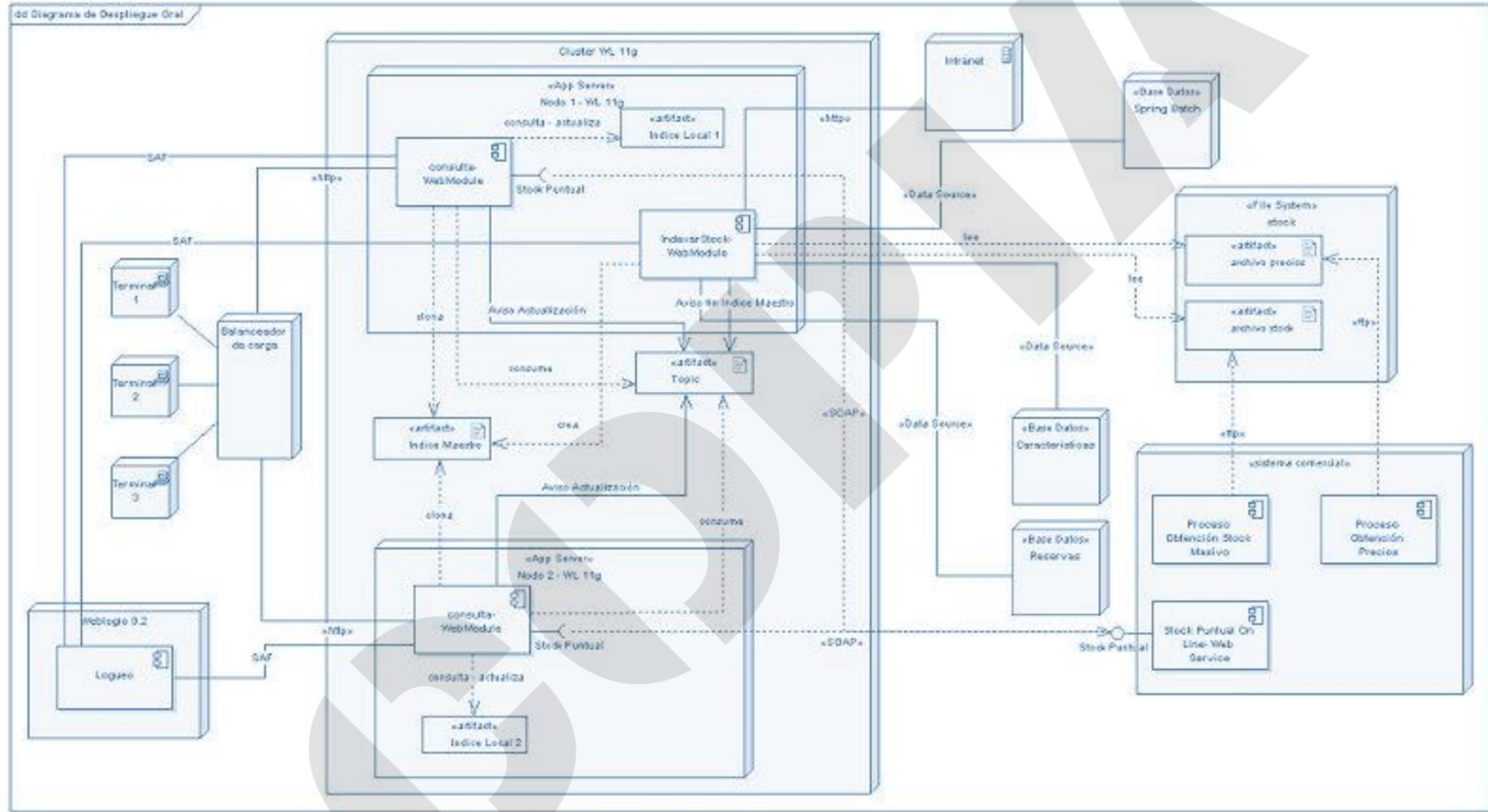
Estereotipo de Nodo: Un número de estereotipos estándar se proveen para los nodos. Estos mostrarán un ícono apropiado en la esquina derecha arriba del símbolo nodo.



Nodo como contenedor: Un nodo puede contener otros elementos, como componentes o artefactos.



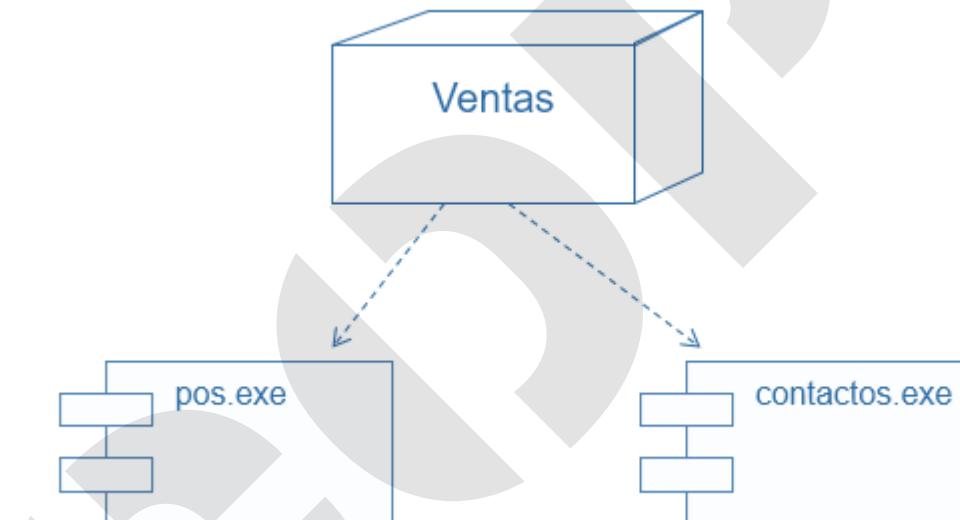
Diagramas de Despliegue



Diagramas de Despliegue

Nodos y componentes

Los nodos son los elementos donde se ejecutan los componentes.

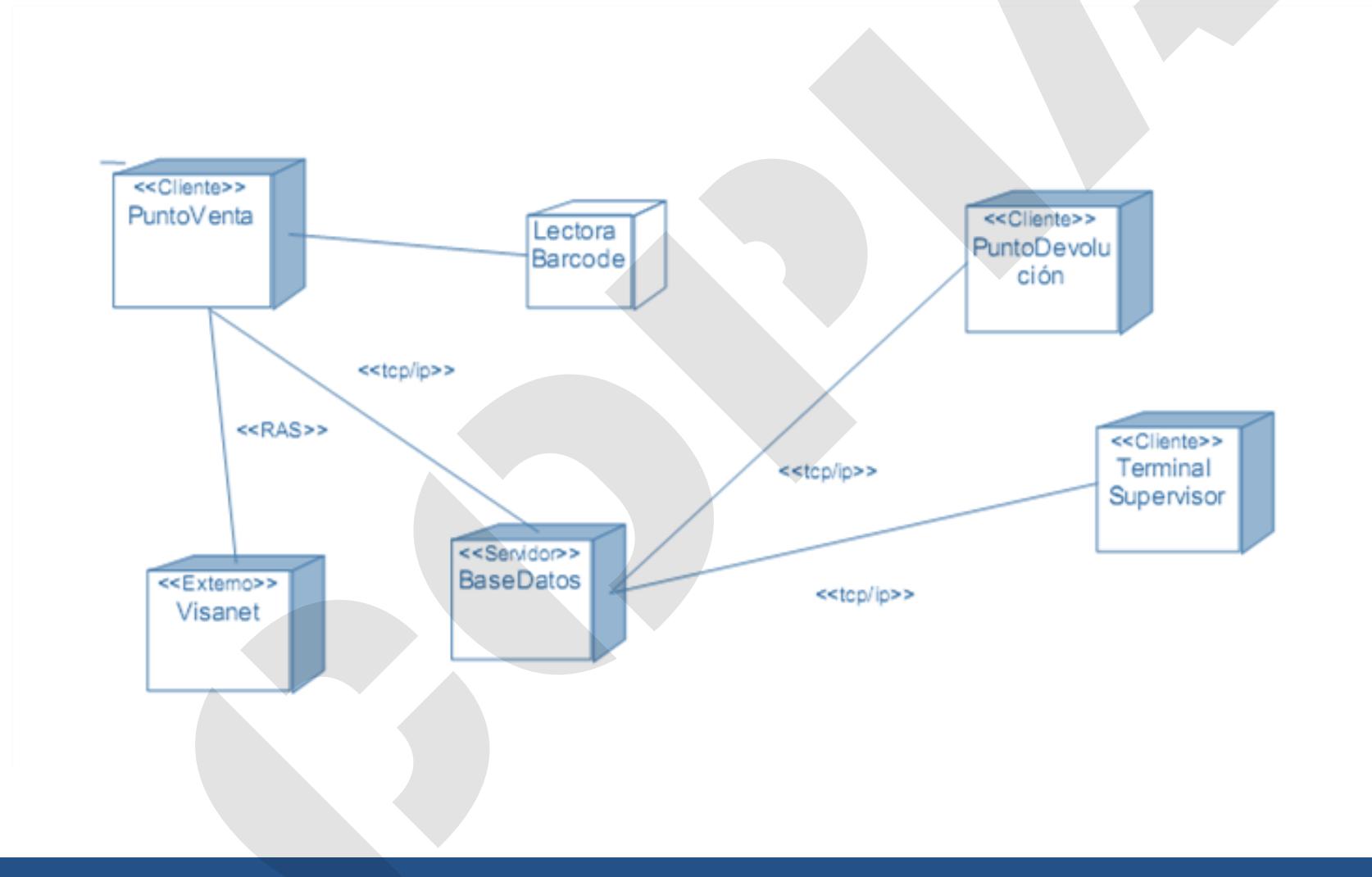


Diagramas de Despliegue

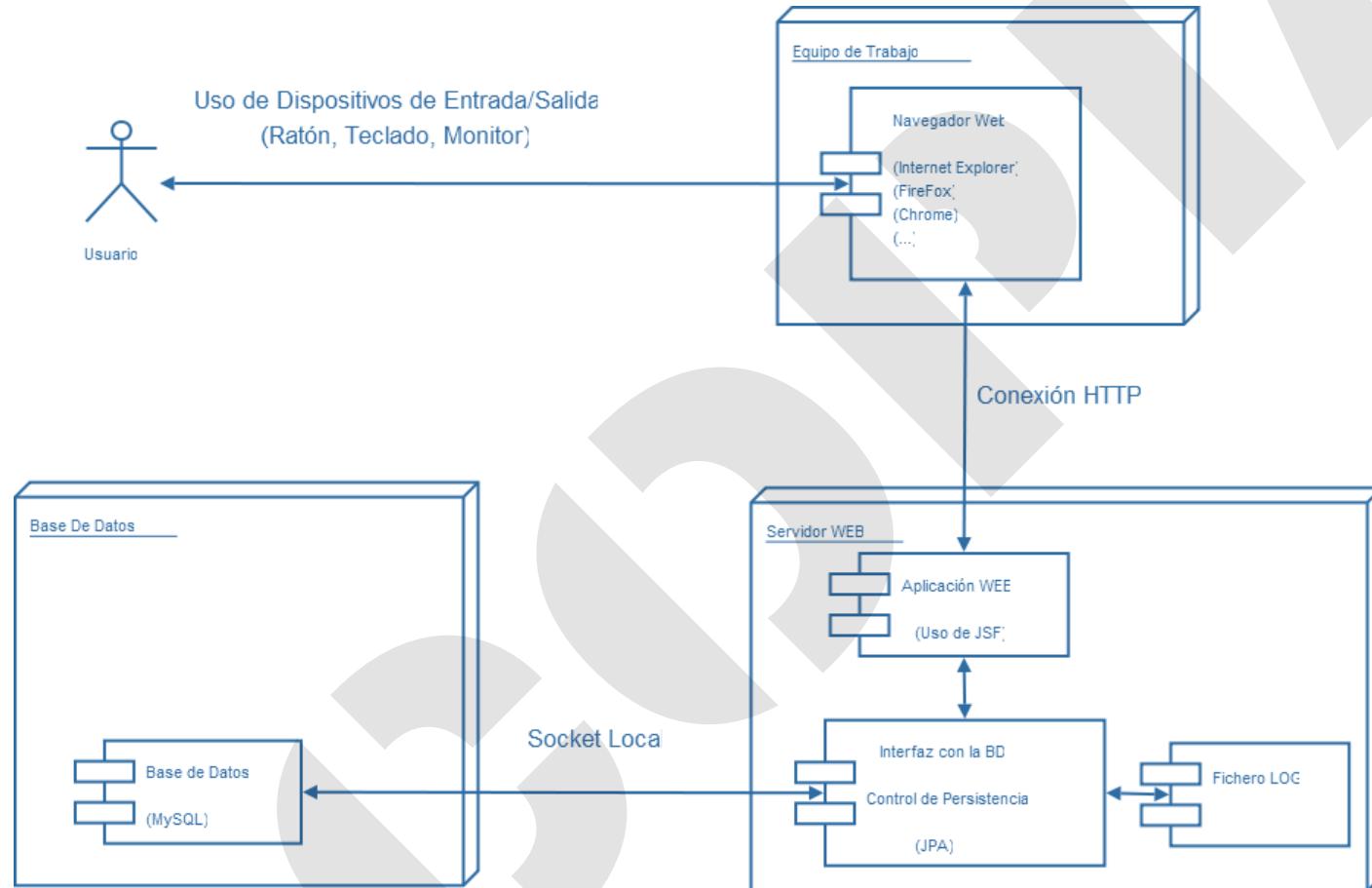
- Si un tipo de componente puede ejecutarse en un tipo de nodo, se crea una dependencia con el estereotipo <<supports>>
 - Una instancia de la componente podría localizarse en una instancia de ese nodo.



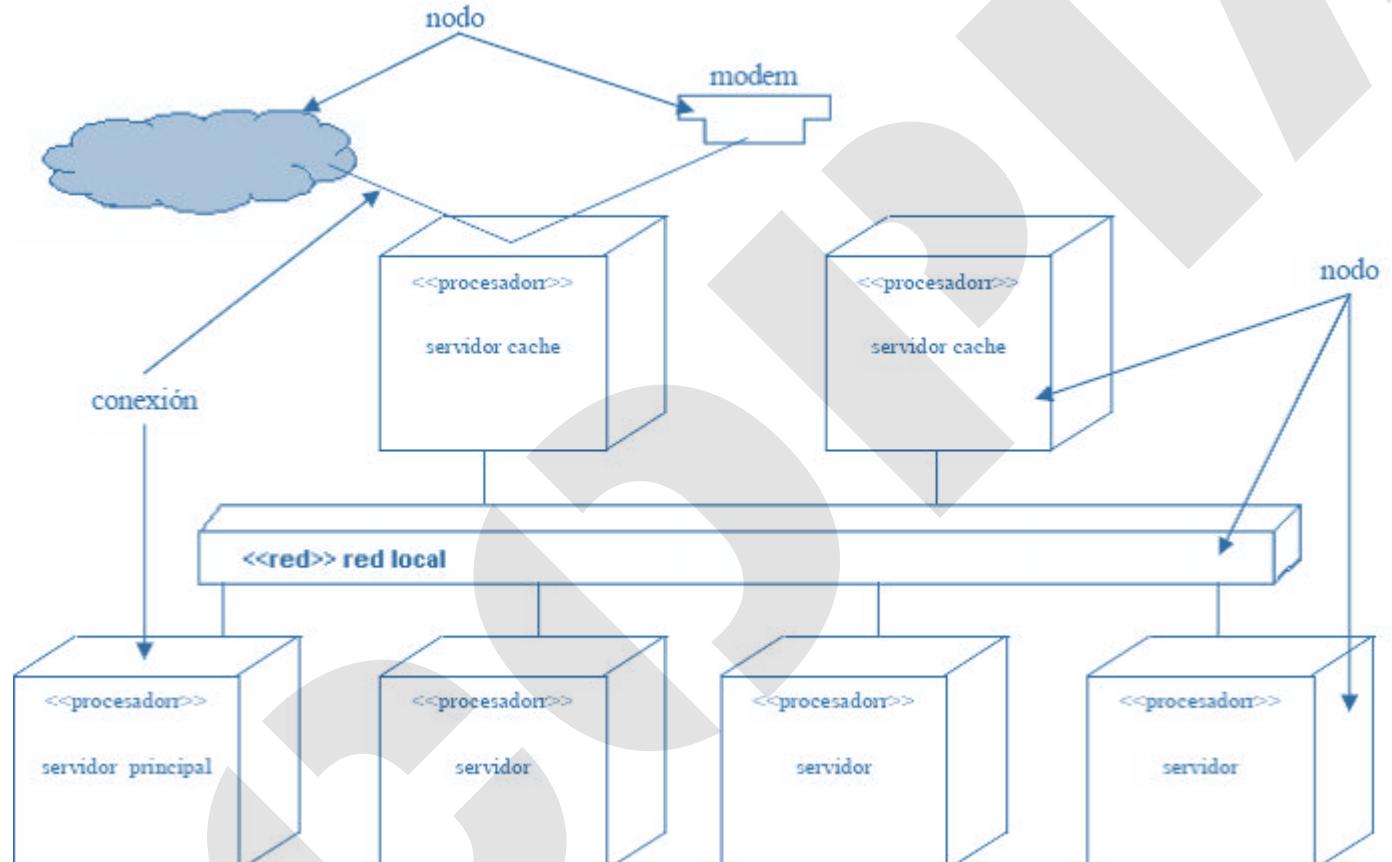
Ejemplo



Ejemplo



Ejemplo



Implementación de componentes por medio de artefactos

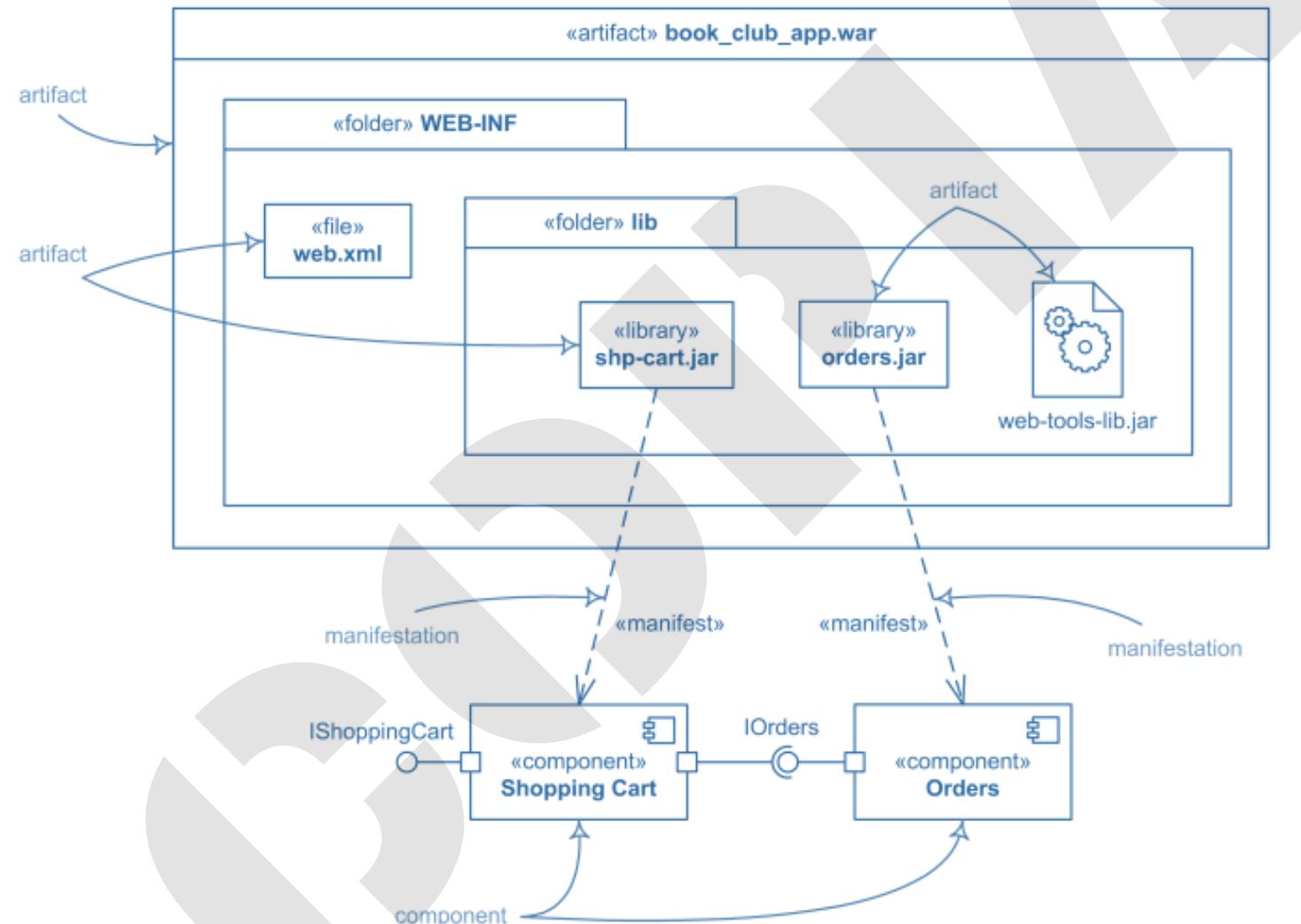


Diagrama de despliegue de nivel de especificación

Los diagramas de despliegue de nivel de especificación muestran una visión general del despliegue de los artefactos hacia los destinos de despliegue, sin hacer referencia a casos concretos de artefactos o nodos.

Diagrama de despliegue de nivel de especificación

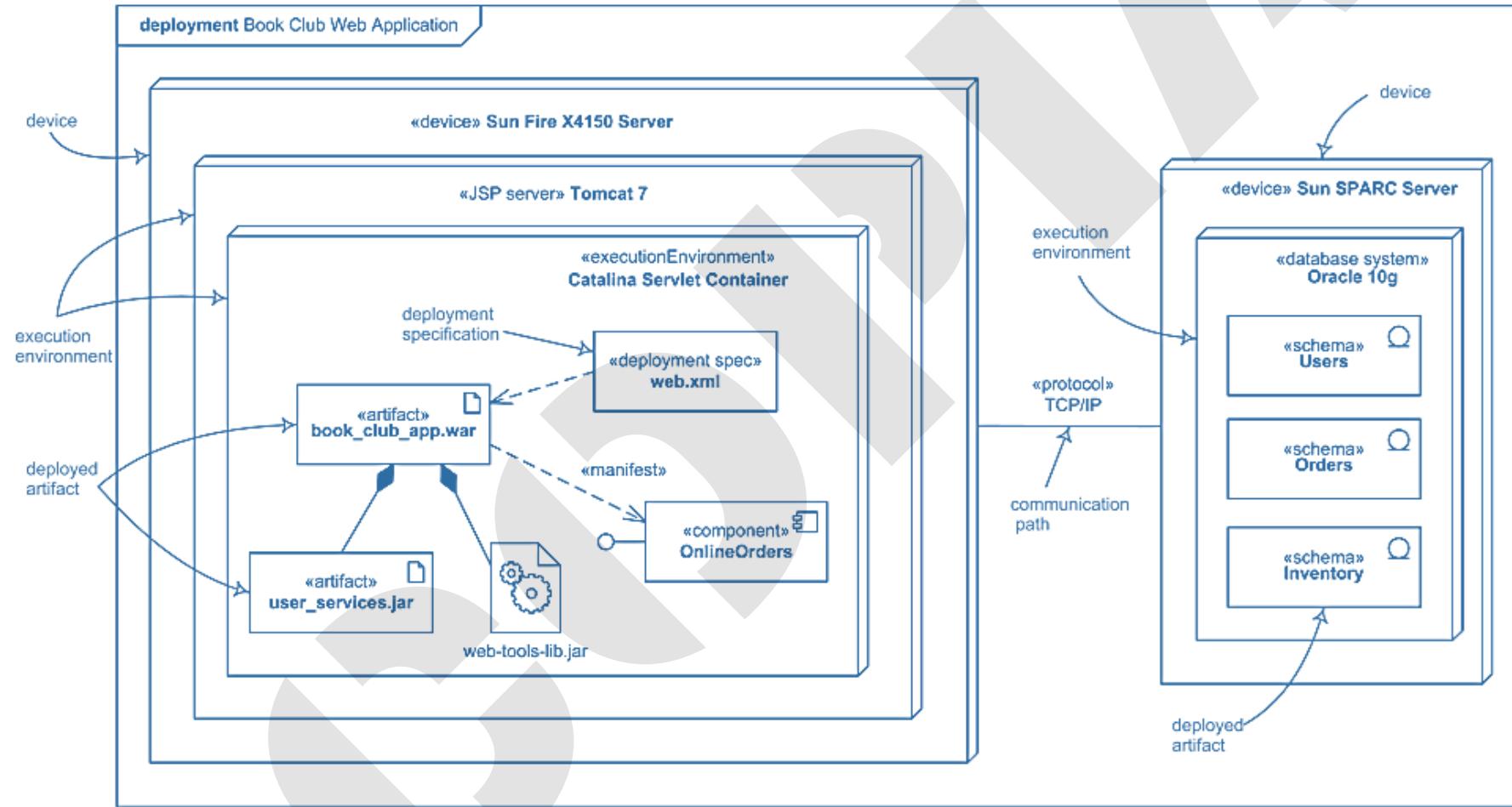
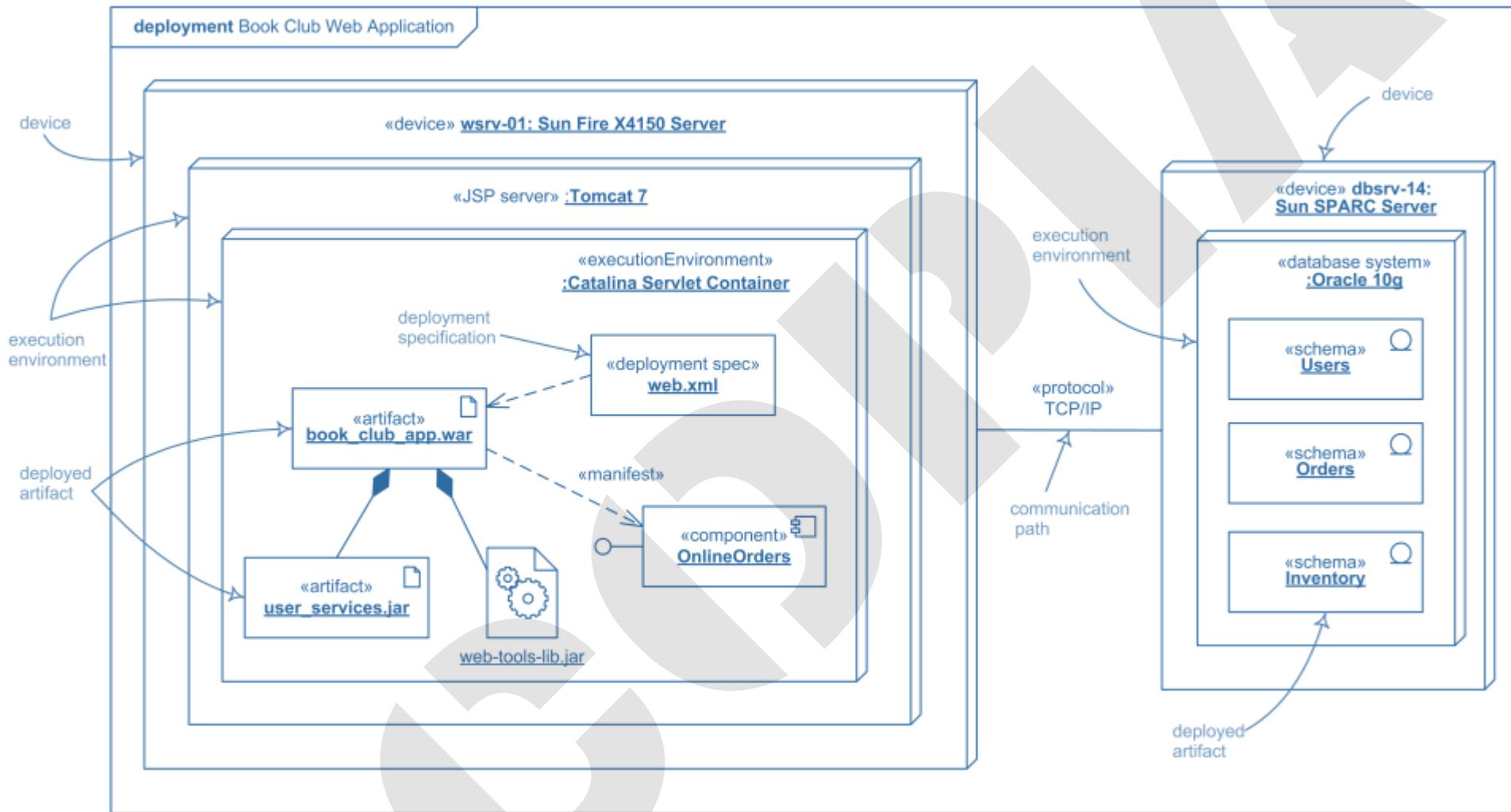


Diagrama de despliegue de nivel de instancia

Los diagramas de despliegue nivel de instancia muestran el despliegue de instancias de artefactos en instancias específicas de los destinos de despliegue.

Se pueden utilizar por ejemplo para mostrar las diferencias existentes en nombres/identificaciones en ambientes de despliegue a desarrollo, de "staging" o de producción, entre construcciones específicas o servidores de despliegue o dispositivos.

Diagrama de despliegue de nivel de instancia





DISEÑO DE SOFTWARE

Patrones de Diseño

Evidencias de diseño erróneo

Rigidez: problemas para insertar algún cambio.

Fragilidad: el software falla en muchos lugares al insertar un cambio.

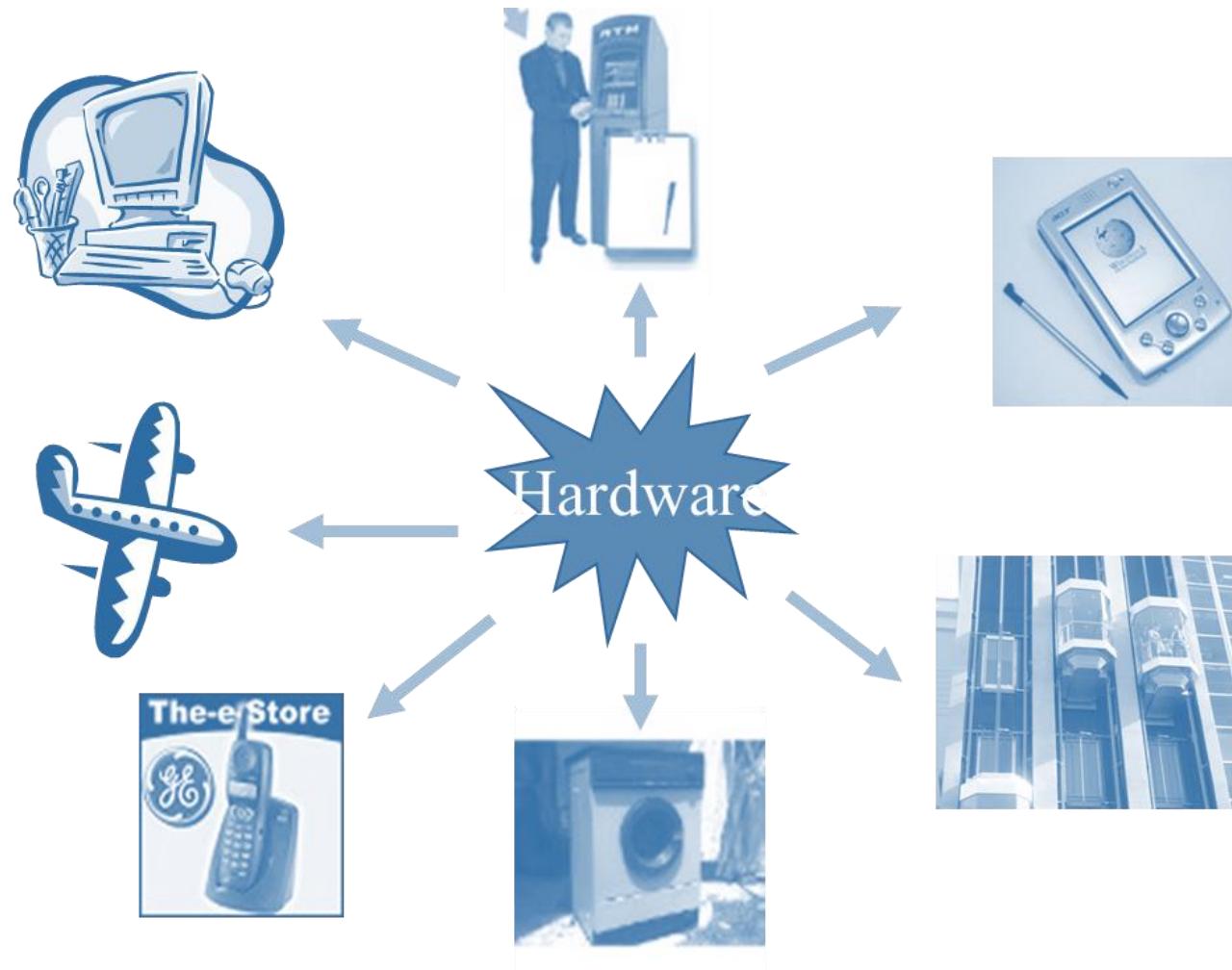
Inmovilidad: no se pueden rehusar partes del proyecto.

Viscosidad: De diseño: cuando se deben hacer cambios, es más fácil hacer cosas mal, que bien. De entorno: entorno de desarrollo ineficiente

Cambios de requerimientos

- Los cambios en un diseño de software, si no fueron cambios previstos en el diseño original, degradan el mismo. Incluyen dependencias.
- Generalmente lo hacen ingenieros que no estaban relacionados con la filosofía de diseño original.

Patrones



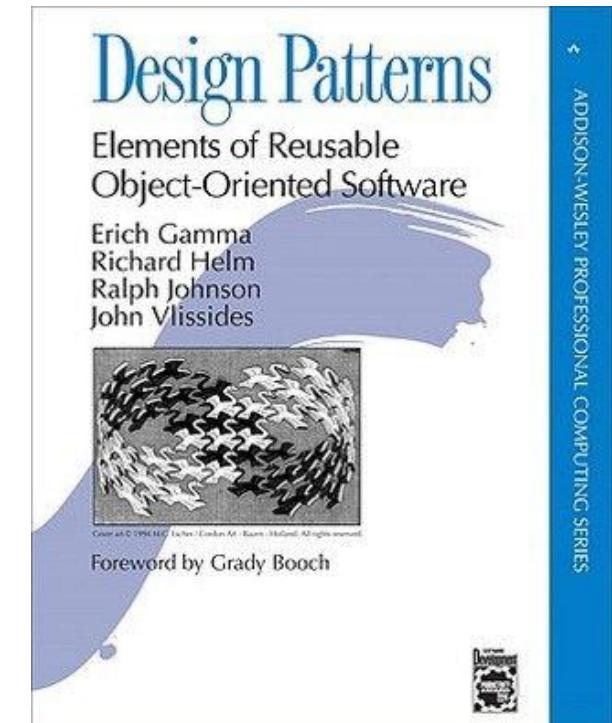
Patrones



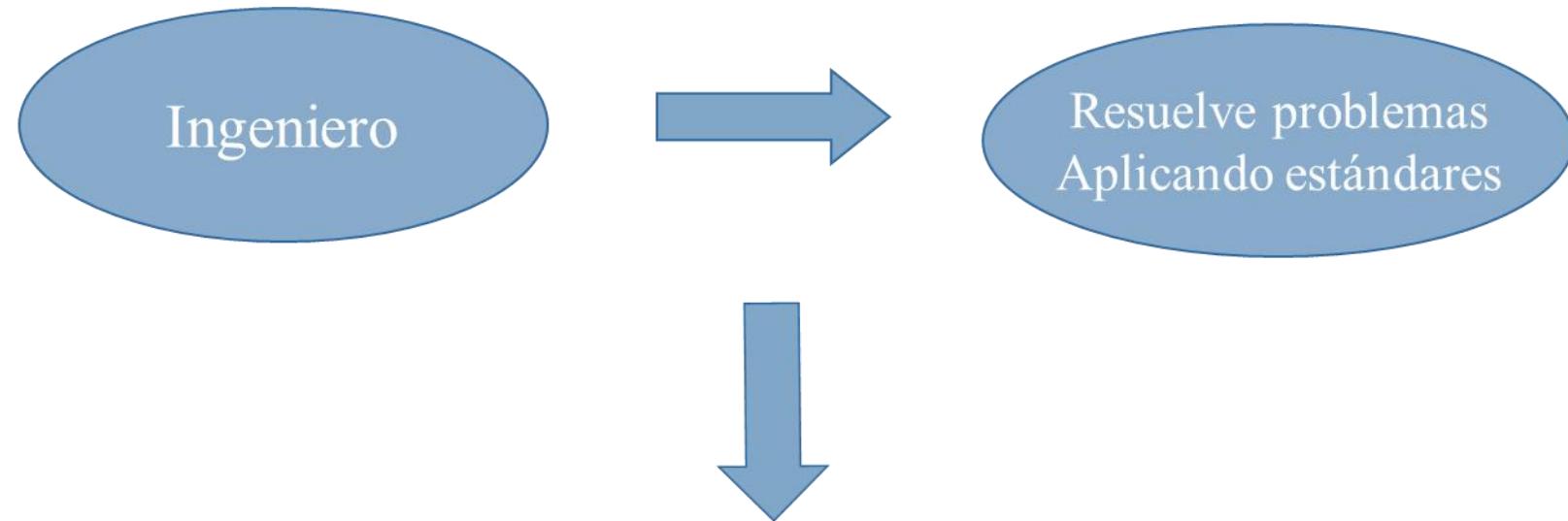
Patrones de diseño

“Son descripciones de clases y objetos relacionados que están adaptados para resolver un problema de diseño general en un contexto determinado”.

Erich Gamma, Richard Helm, John Vlissides y Ralph Johnson



Patrones de diseño



*Las buenas soluciones permanecen, las malas se rechazan.
Los ingenieros deben conocer y saber aplicar los estándares conocidos*

Patrones de diseño

- Se definen con un alto nivel de abstracción.
- Son independientes de los lenguajes de programación y de los detalles de implementación.
- Los patrones promueven y facilitan la reutilización de arquitecturas y diseños de software que han demostrado su validez en muchas aplicaciones.

Arquitectura

- Define la forma y la estructura de la aplicación de software.
- Relacionada al propósito de la aplicación.
- Arquitectura de módulos y sus interconexiones

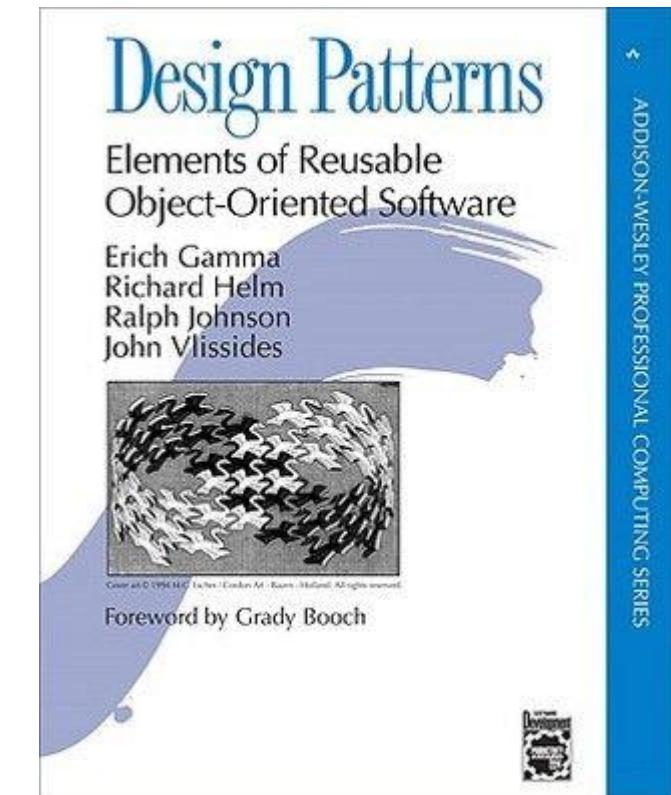


Patrones de diseño

Erich Gamma, Richard Helm, John Vlissides y Ralph Johnson

Design Patterns.

1994



Patrones de diseño

- Describe una estructura dentro de la cual catalogar y describir patrones
- Cataloga 23 patrones
- Destaca estrategias y aproximaciones basadas en el diseño de patrones
- No crearon los patrones descriptos en el libro.
- Los descubrieron como existentes dentro de la comunidad del software

Porque estudiar patrones de diseño

- Reúso de soluciones de diseño.
- Establecer terminología común.
- Dan una perspectiva de alto nivel sobre el análisis y diseño.
- Los Patrones de diseño proporciona un esquema para **refinar los subsistemas o componentes** de un sistema de software, o las relaciones entre ellos. Describe estructuras repetitivas de *comunicación de componentes* que resuelven un problema de diseño en un contexto particular

Patrones de diseño

- Programe para una interfaz, no para una implementación.

Comience cualquier jerarquía que necesite para solucionar su problema con una clase abstracta, sin implementación de métodos. Que solo describa los métodos que debe soportar.

- Favorecer la composición frente a la herencia de clases.

Construir objetos que contengan otros objetos. No cargue con todo el peso de heredar métodos que no necesita

- Favorecer la composición frente a la herencia de clases.

Construir objetos que contengan otros objetos. No cargue con todo el peso de heredar métodos que no necesita.

- Encuentre lo que varía y encapsúlelo.

Lo que puede ser cambiado en su diseño encapsúlelo en una clase , para no tener necesidad de rediseñar.

Patrones de diseño

1. Aceptarlos
2. Reconocerlos
3. Internalizarlos

Clasificación de patrones (GoF)

Patrones de creación: Tratan de la inicialización y configuración de clases y objetos.

Patrones estructurales: Tratan de desacoplar interfaz e implementación de clases y objetos.

Patrones de comportamiento: Tratan de las interacciones dinámicas entre sociedades de clases y objetos

Patrones de creación

The Factory Method *retorna una de las posibles subclases de una clase abstracta dependiendo de los datos que se le provee.*

The Abstract Factory Method *retorna una de las varias familias de objetos.*

The Builder Pattern *separa la construcción de un objeto complejo de su representación. Varias representaciones se pueden crear dependiendo de las necesidades del programa.*

The Prototype Pattern *inicializa e instancia una clase y luego copia o clona si necesita otras instancias, mas que crear nuevas instancias.*

The Singleton Pattern *es una clase de la cual no puede existir mas de una instancia.*

Patrones estructurales

Adapter: *cambia la interfaz de una clase a la de otra.*

Bridge: *permite mantener constante la interfaz que se presenta al cliente, cambiando la clase que se usa*

Composite: *una colección de objetos*

Decorator: *una clase que envuelve a una clase dándole nuevas capacidades.*

Facade: *reúne una jerarquía compleja de objetos y provee una clase nueva permitiendo acceder a cualquiera de las clases de la jerarquía.*

Flyweight: *permite limitar la proliferación de pequeñas clases similares.*

Patrones de comportamiento

Cadena de responsabilidad: *permite que un conjunto de clases intenten manejar un requerimiento.*

Interprete: *define una gramática de un lenguaje y usa esa gramática para interpretar sentencias del lenguaje.*

Iterator: *permite recorrer una estructura de datos sin conocer detalles de cómo están implementados los datos*

Observer: *algunos objetos reflejan un cambio a raíz del cambio de otro, por lo tanto se le debe comunicar el cambio de este último.*

Strategy: *cantidad de algoritmos relacionados encerrados en un contexto a través del cual se selecciona uno de los algoritmos.*

Otros tipos de patrones

Patrones de programación concurrente

Patrones de interfaz gráfica

Patrones de organización de código

Patrones de optimización de código

Patrones de robustez de código

Patrones de fases de prueba

Plantilla GoF

Nombre *Un buen nombre es vital porque será parte del vocabulario de diseño*

Nombres Alternativos *Otros nombres de uso común para el patrón.*

Propósito *Qué problema pretende solucionar.*

Motivación *Descripción del problema y su contexto*

Puede consistir en un ejemplo (un escenario) que ilustre la clase de problemas que el patrón intenta resolver.

En general se entienden mejor los problemas concretos que los abstractos.

Plantilla GoF

Estructura *Representación gráfica de las clases de los objetos que participan en el patrón y de sus relaciones estructurales (estáticas)*

Actualmente, lo más común es usar UML.

Aplicabilidad *¿En qué situaciones puede/debe aplicarse el patrón?*

Participantes *Las clases y objetos que participan en el patrón y sus responsabilidades o roles*

Consecuencias *¿Qué efectos positivos y negativos implica el uso del patrón?*

¿Cuáles son los compromisos de diseño que implica su uso?

¿Qué aspectos de la estructura del sistema pueden variar de forma independiente?

Colaboración *Cómo colaboran los participantes para llevar a cabo sus responsabilidades y proporcionar el comportamiento deseado*

Plantilla GoF

Usos conocidos *Al menos dos ejemplos de uso del patrón en aplicaciones reales.*

Implementación *¿Cómo puede implementarse el patrón en un lenguaje de programación?*

¿Qué dificultades implica?

¿Hay aspectos dependientes del lenguaje de programación?

Código de ejemplo *Fragmentos de código que ilustren cómo se implementa el patrón en uno o varios lenguajes de programación*

Patrones relacionados *¿Cuáles son los patrones más estrechamente relacionados con el dado?*

¿Se usa en conjunción con otros patrones? ¿De qué manera?

Beneficios de los patrones

- Los patrones favorecen la reutilización de diseños y arquitecturas a gran escala.
- Capturan el conocimiento de los expertos y lo hacen accesible a toda la comunidad software.
- Proporcionan un cuerpo de conocimiento utilizable por toda la comunidad software.
- Favorecen la transmisión de conocimiento entre profesionales y entre clientes y desarrolladores
- Proporcionan un lenguaje común. Los nombres de los patrones forman parte del vocabulario técnico del ingeniero software.

Problema de los patrones

- Los patrones, no llevan de forma directa a la reutilización del código, aunque dicha reutilización se facilita mediante su uso.
- La integración de los patrones en el proceso de desarrollo se hace todavía de forma manual.
- El número de patrones identificados es cada vez más grande. Las clasificaciones actuales no siempre sirven de guía para decidir cual usar.

Problema de los patrones

- El número de combinaciones patrones estilos y atributos que se dan en la práctica son incontables.
- Los patrones **se validan por la experiencia y el debate**, no mediante la aplicación de técnicas formales

Cómo usar un patrón de diseño

1. Leer el patrón una vez para tener una visión general
2. Volver y estudiar la estructura, los participantes y las colaboraciones
3. Ver un ejemplo concreto codificado del patrón
4. Elegir nombres para los participantes del patrón que sean significativos en el contexto de la aplicación
5. Definir las clases
6. Definir nombres específicos de la aplicación para las operaciones en el patrón.
7. Implementar las operaciones que realizarán las responsabilidades y colaboraciones del patrón.

Singleton

Singleton

Métodos y Variables estáticas

Antes de nada es importante entender que son los métodos y variables estáticas en el desarrollo orientado a objetos.

Método estático (o método de clase)

- Un método estático es un método perteneciente a una clase que puede ser invocado sin necesidad de instanciar esa clase.
- Un método estático sólo puede acceder a miembros estáticos de la clase a la cual pertenece.

Variable estática (o variable de clase)

- Es una variable propia de la clase, lo cual significa que tiene el mismo valor para todas las instancias de la misma. Dicho de otra forma, no existirán dos instancias de una misma clase con diferentes valores almacenados en una variable estática, sino que el valor será el mismo para ambas instancias.

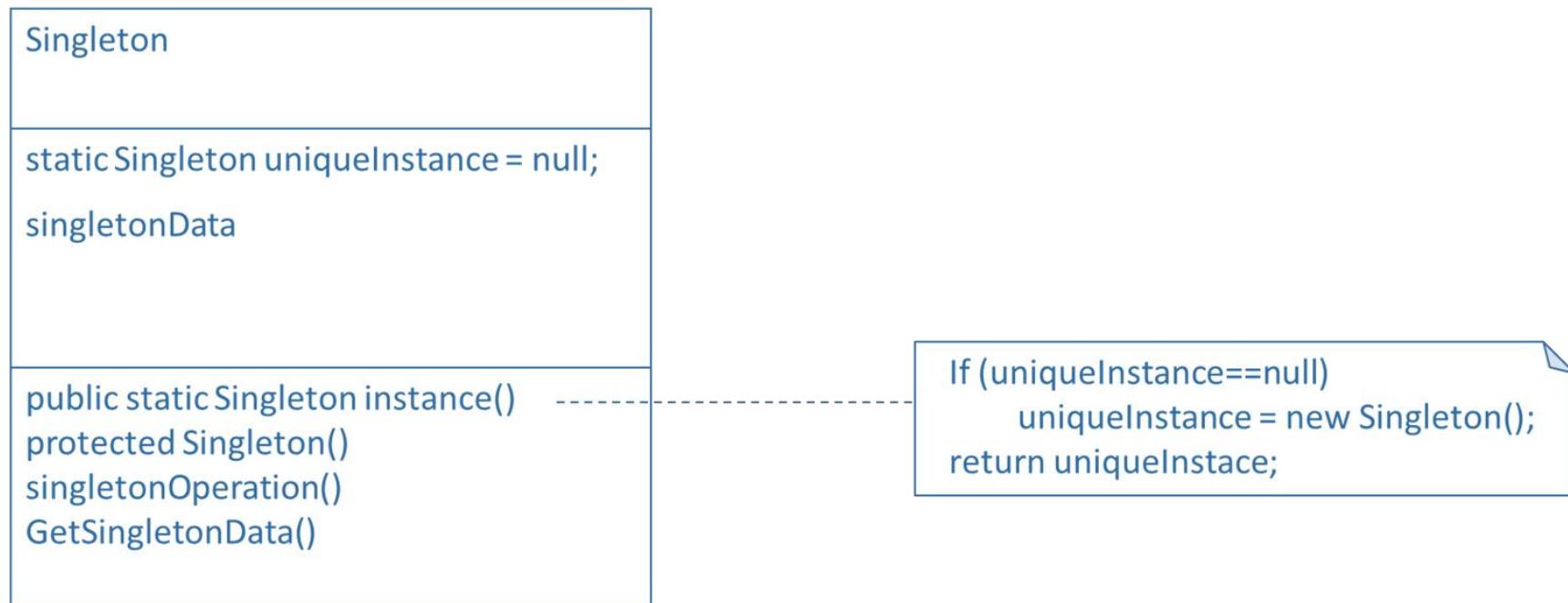
Singleton – Problema a resolver

Asegurar que una clase tiene una única instancia y proveer un punto de acceso global a la misma, es decir, que pueda accederse fácilmente a dicha instancia desde cualquier lugar del sistema.

Debe usarse cuando:

- Debe existir una única instancia de una clase y ésta debe poder ser accesible desde un punto de acceso bien conocido.
- La única instancia debería ser extensible por subclases y los clientes deberían poder usar una instancia que herede de la clase sin modificar su código.

Singleton – Estructura



Singleton – Participantes

Singleton

- Define una **operación “Instance”** que permite a los clientes acceder a la única instancia. Esta operación es una operación de clase (ver método de clase o estático).
- Debería ser **responsable** de crear su propia única instancia.
- El constructor de la clase Singleton debe ser **protegido**, de modo tal que sea la clase la única que puede usarlo y así controlar la creación de instancias de la clase.

Singleton – Colaboraciones

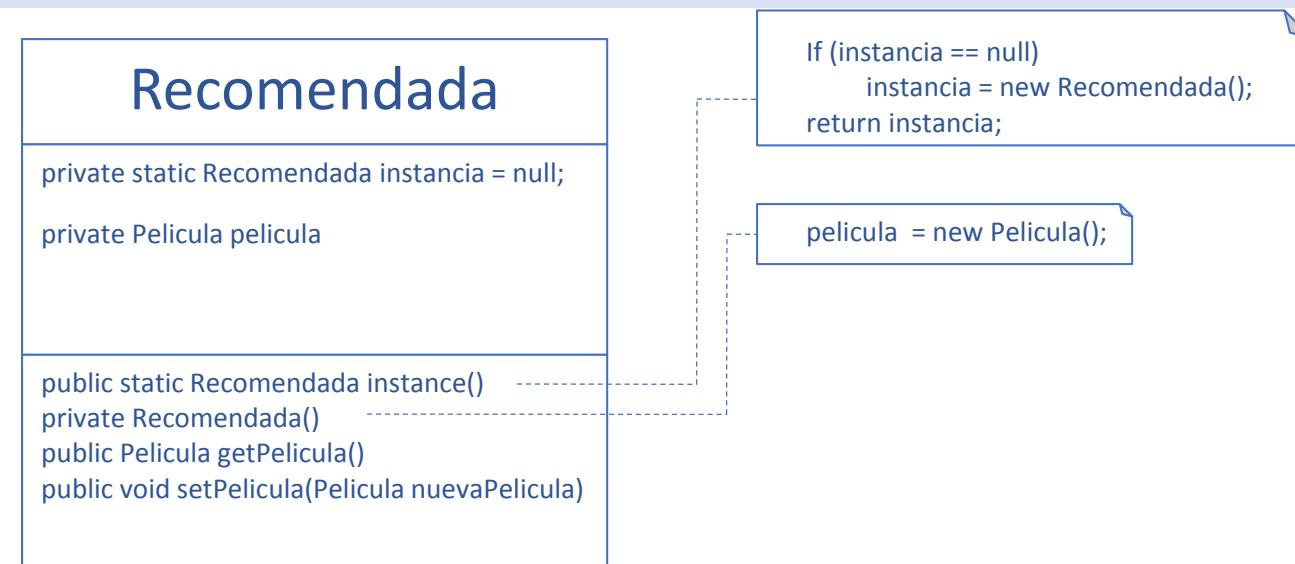
Los clientes acceden una instancia Singleton únicamente a través de la operación de instancia Singleton.

Singleton - Consecuencias

- **Acceso controlado a la única instancia.** Dado que una clase Singleton encapsula su única instancia, tiene un control estricto de cómo y cuando los clientes acceden a ella.
- **Espacio de nombre reducido.** El patrón Singleton es una mejora de las variables globales. Evita la contaminación del espacio de nombres con variables globales de que almacenan una sola instancia.
- **Permite el refinamiento de operaciones y de la representación.** La clase Singleton puede ser extendida y resulta fácil configurar una aplicación para que utilice esta subclase. Se puede configurar una aplicación con la instancia que sea necesaria en tiempo de ejecución.
- **Permite un número variable de instancias.** El patrón permite cambiar de ideas y empezar a usar más de una instancia con facilidad. Es más, se puede usar el mismo enfoque para control el número de instancias que la aplicación usa. Para esto lo único necesario es modificar la operación que garantiza el acceso a la instancia Singleton.

Singleton – Ejemplo

Consideremos una aplicación para un video club que desea administrar la venta y alquiler de películas. Supongamos que deseamos agregarle una funcionalidad a dicho sistema la cual permita seleccionar diariamente una película recomendada para ese día. Esto podría dar lugar a la existencia de una clase **Recomendada**. Diseñando la misma como Singleton nos asegura de que a lo largo de todo el sistema (el cual podría estar distribuido entre sucursales) se determine una única vez cuál será la película recomendada para ese día y **que ésta sea una sola**.



Factory Method

Factory Method - Introducción

Es un conjunto de clases cooperando entre si con el fin de obtener un diseño reusable para ciertas clases de software específico. Estas clases son abstractas y a partir del framework en general pueden desarrollarse varias aplicaciones creando clases que implementen esas clases abstractas. Esto conlleva a crear aplicaciones mucho más rápidamente y todas las aplicaciones desarrolladas a partir de un mismo framework tienen estructuras muy similares.

Factory Method – Problema a resolver

Definir una interface para crear un objeto, pero permitir que las subclases decidan que clase instanciar. **Permite delegar la instanciación a las subclases.**

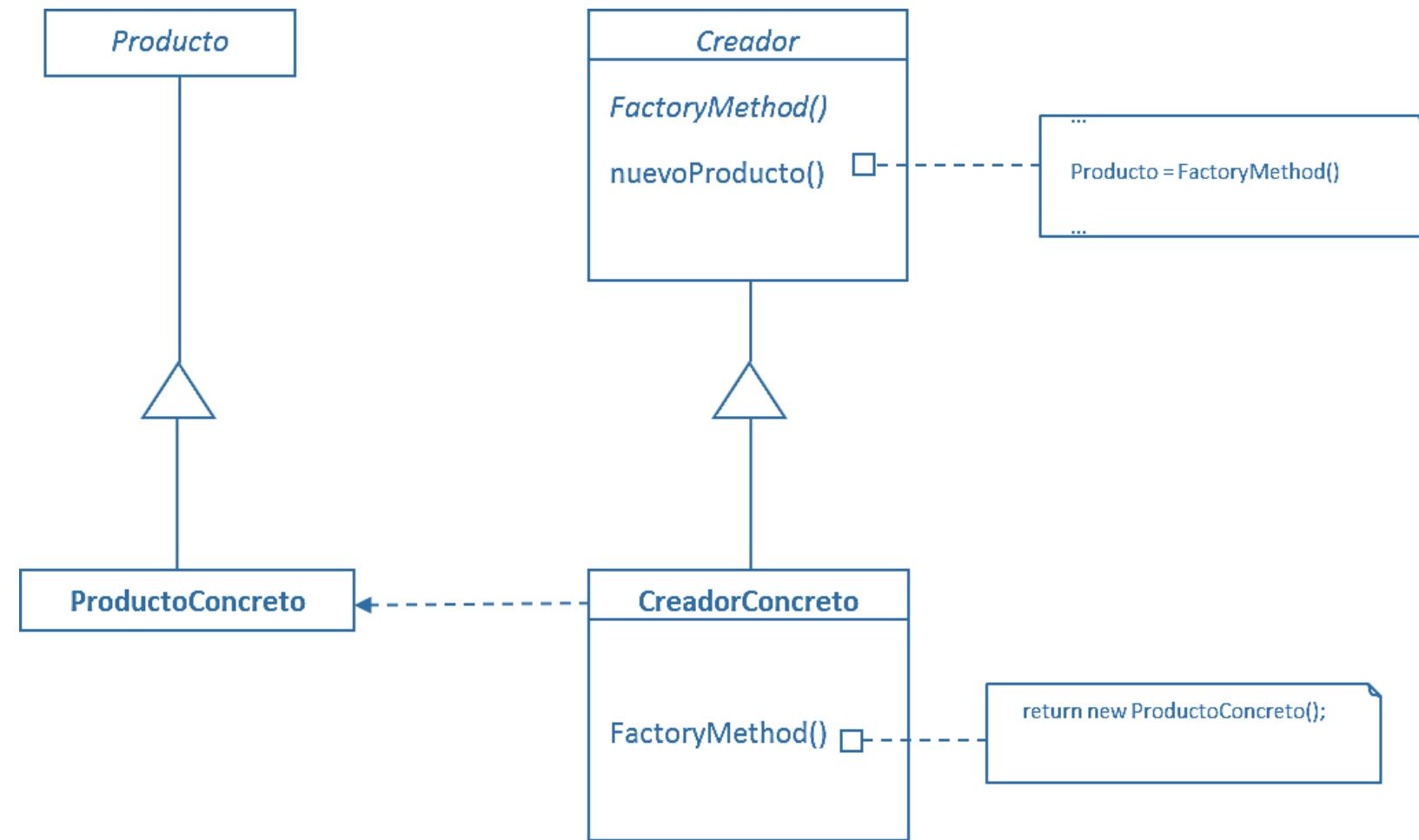
Los frameworks utilizan clases abstractas para definir y mantener relaciones entre objetos. Generalmente el framework también es responsable de crear estos objetos.

Factory Method - Solución

Este patrón se usa cuando:

- La clase no puede saber de antemano que tipo de objetos creará.
- La clase quiere que sean sus subclases las que especifiquen que objeto crear.
- Clases que necesitan delegar esta responsabilidad a una de varias subclases y se desea saber cuál de ellas es la delegada.

Factory Method - Estructura



Factory Method - Participantes

- **Producto:** Define la interface de los objetos que el método FactoryMethod() crea.
- **ProductoConcreto:** implementa la interface de Producto.
- **Creador:** declara el método factory, el cual retorna un objeto del tipo Producto. A su vez podría definir una implementación por defecto para el Factory Method que retorne un objeto de ProductoConcreto por defecto. A su vez debería llamarse al FactoryMethod para crear instancias de Producto.
- **CreadorConcreto:** Implementa el método FactoryMethod de la clase Creador de tal forma que este cree instancias de la clase ProductoConcreto correspondiente. Si en Creador se le dio a FactoryMethod una implementación por defecto, CreadorConcreto puede hacer un override de la misma.

Factory Method - Colaboraciones

La clase Creador cuenta con que sus subclases implementen el factory method de forma tal de que este retorne una instancia de la clase ProductoConcreto correspondiente.

Factory Method - Consecuencias

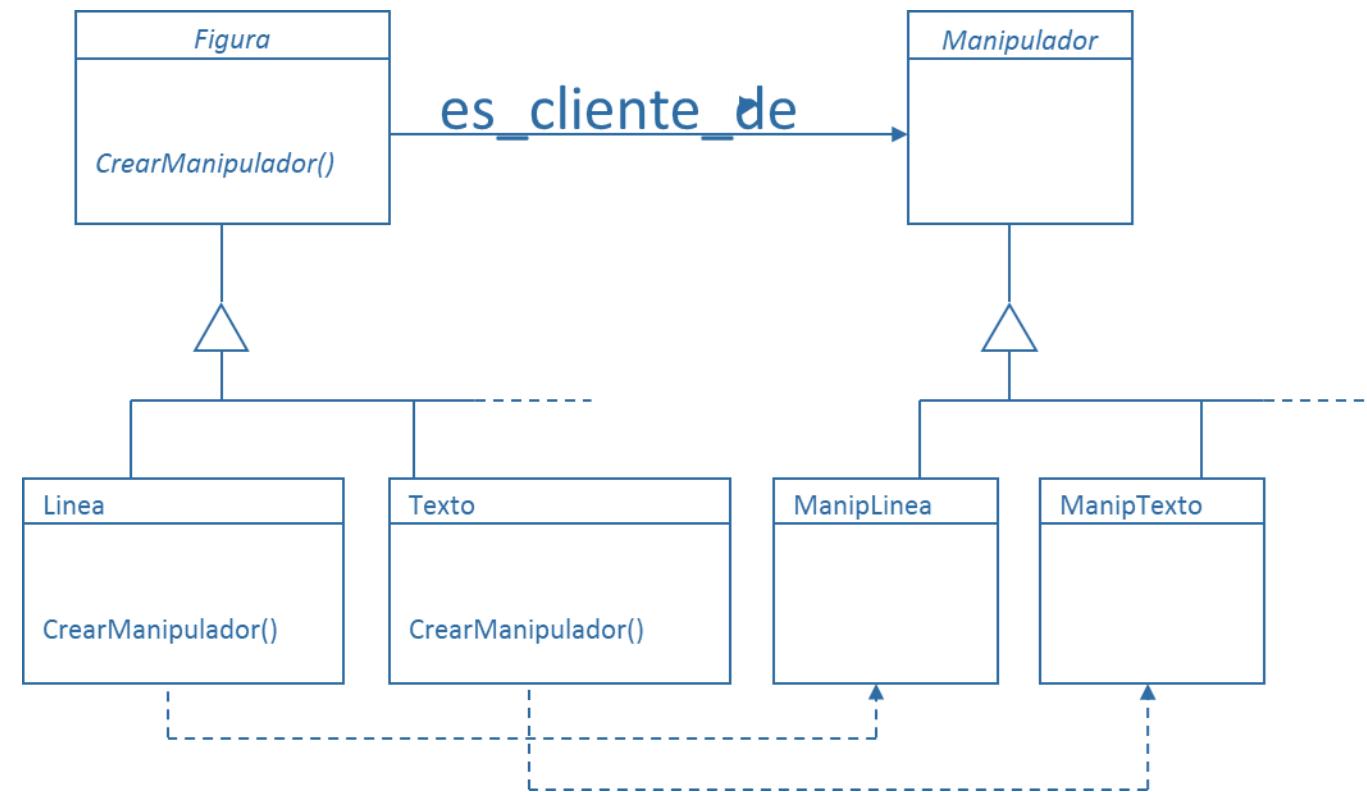
Una de las desventajas más importantes de este patrón es que los clientes deben extender si o si la clase Creador sólo para poder crear objetos de la clase ProductoConcreto. Esto no sería un problema si de todos modos para la aplicación era necesario extender aquella clase, pero en caso de que esto no hubiera sido necesario de antemano el patrón obliga a crear subclases.

Otras dos consecuencias son:

- Facilita a las subclases la creación de objetos desde ellas mismas, lo cuál es mucho más flexible que crear un objeto directamente, dado que las clases padres tiene un factory method para crear objetos. Como las subclases pueden hacer override del factory method, éstas podrán crear objetos adecuados a sus necesidades.
- Conecta jerarquías paralelas de clases. Supongamos tener una clase abstracta que hace uso de otra clase como cliente. Esta clase tendrá una serie de subclases las cuales podrían necesitar usar como clientes una serie de subclases de aquella de la que la clase padre es cliente. Colocando un factory method en la clase padre y permitiendo que cada subclase implemente este método de acuerdo a la subclase correspondiente de la clase de la cual es cliente nos lleva a tener bien definido el paralelismo entre las clases clientes y las clases servidor.

Factory Method - Solución

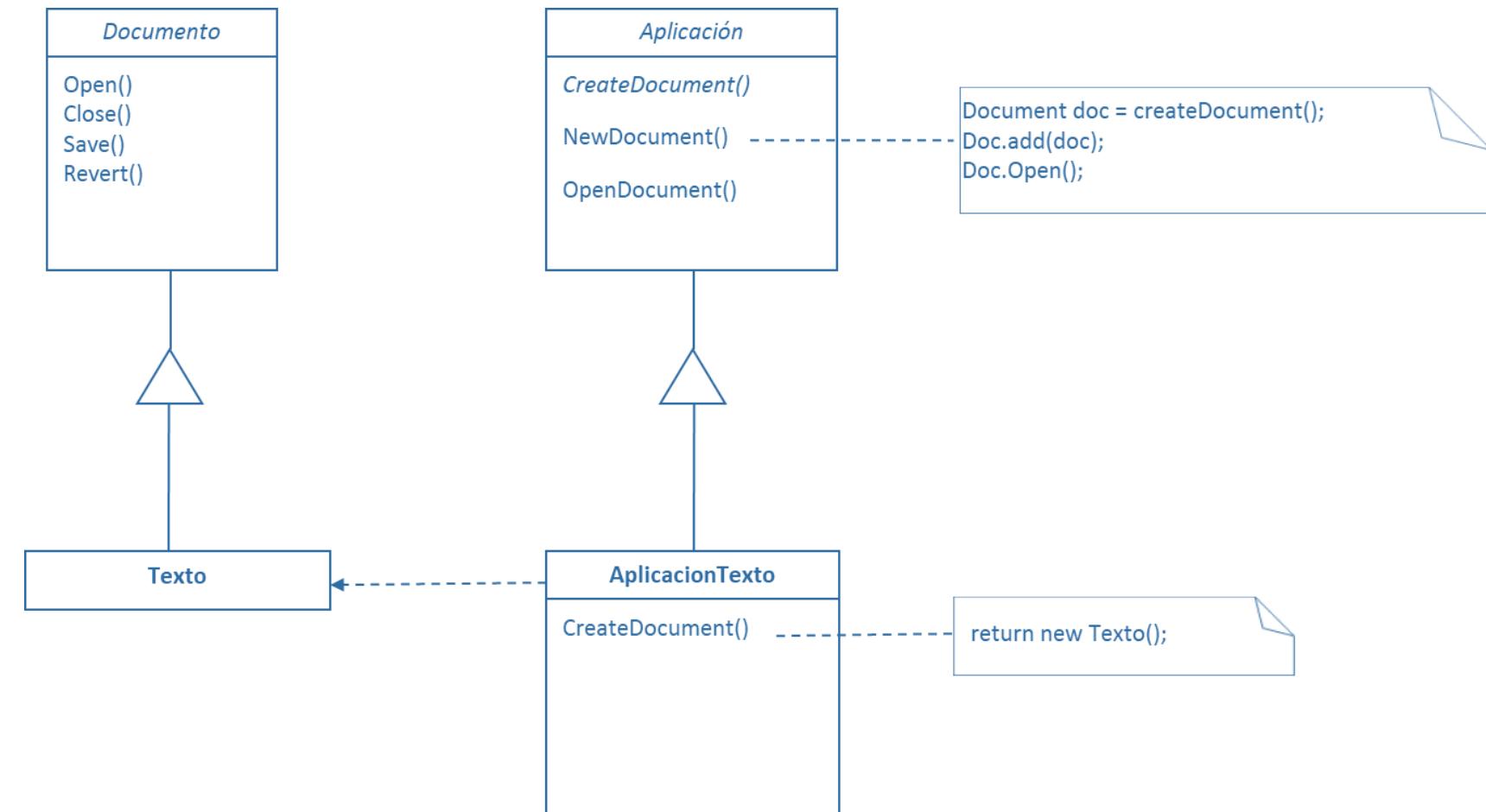
Ejemplo de conexión de jerarquías paralelas.

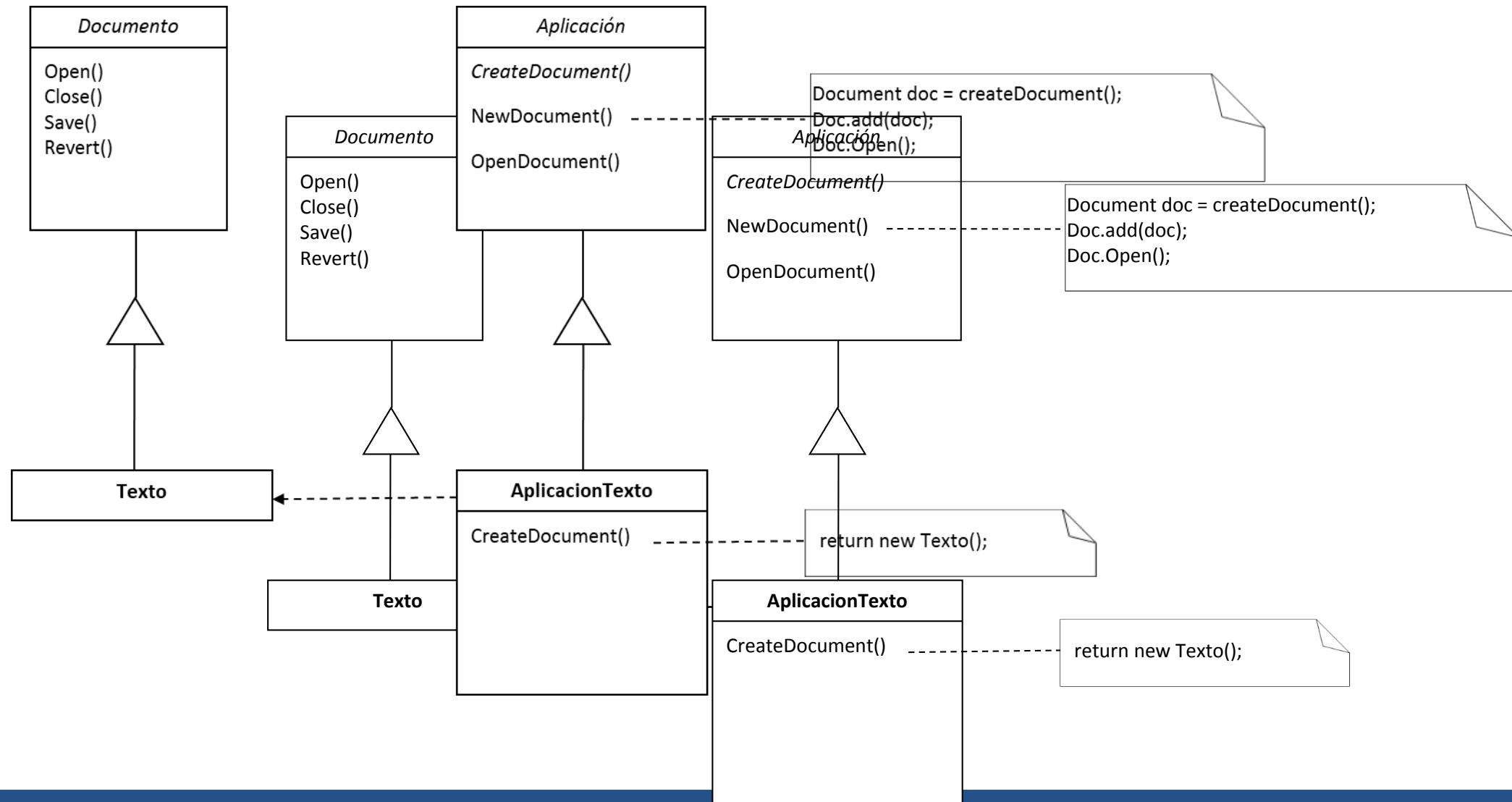


Factory Method - Ejemplo

Consideremos el ejemplo ya dado en el cual se desea tener un marco con el que crear distintas aplicaciones que manejan distintos tipos de documentos. En aquel caso la aplicación creadora particular creaba, valga la redundancia, documentos de tipo imagen. Para ver el uso general de este patrón, supongamos ahora, que basándonos en las mismas clases abstractas queremos crear una aplicación que trabaje con documentos de texto. Lo único necesario será crear una clase que herede de la clase Aplicación y que implemente adecuadamente el método abstracto *CreateDocumento()* para que este instancie objetos de tipo texto, los cuales, a su vez, son subclases de la clase Documento.

El diagrama correspondiente con su implementación sería:





Abstract Factory

Abstract Factory: Definición

El patrón **Abstract Factory** proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas. Permite trabajar con objetos de distintas familias de manera que las familias no se mezclen entre sí y haciendo transparente el tipo de familia concreta que se esté usando.

¿Qué queremos decir con familias de productos? Imaginemos que tenemos una aplicación y queremos que pueda mostrarse en múltiples sistemas de ventanas. Por ejemplo, en una ventana no queremos que se mezclen botones tipo Windows 95 con barras de desplazamiento de estilo Mac. Por esto, debemos asegurarnos que todos los objetos de interfaz de usuario que creemos pertenezcan a la misma familia.

Abstract Factory: *Definición*

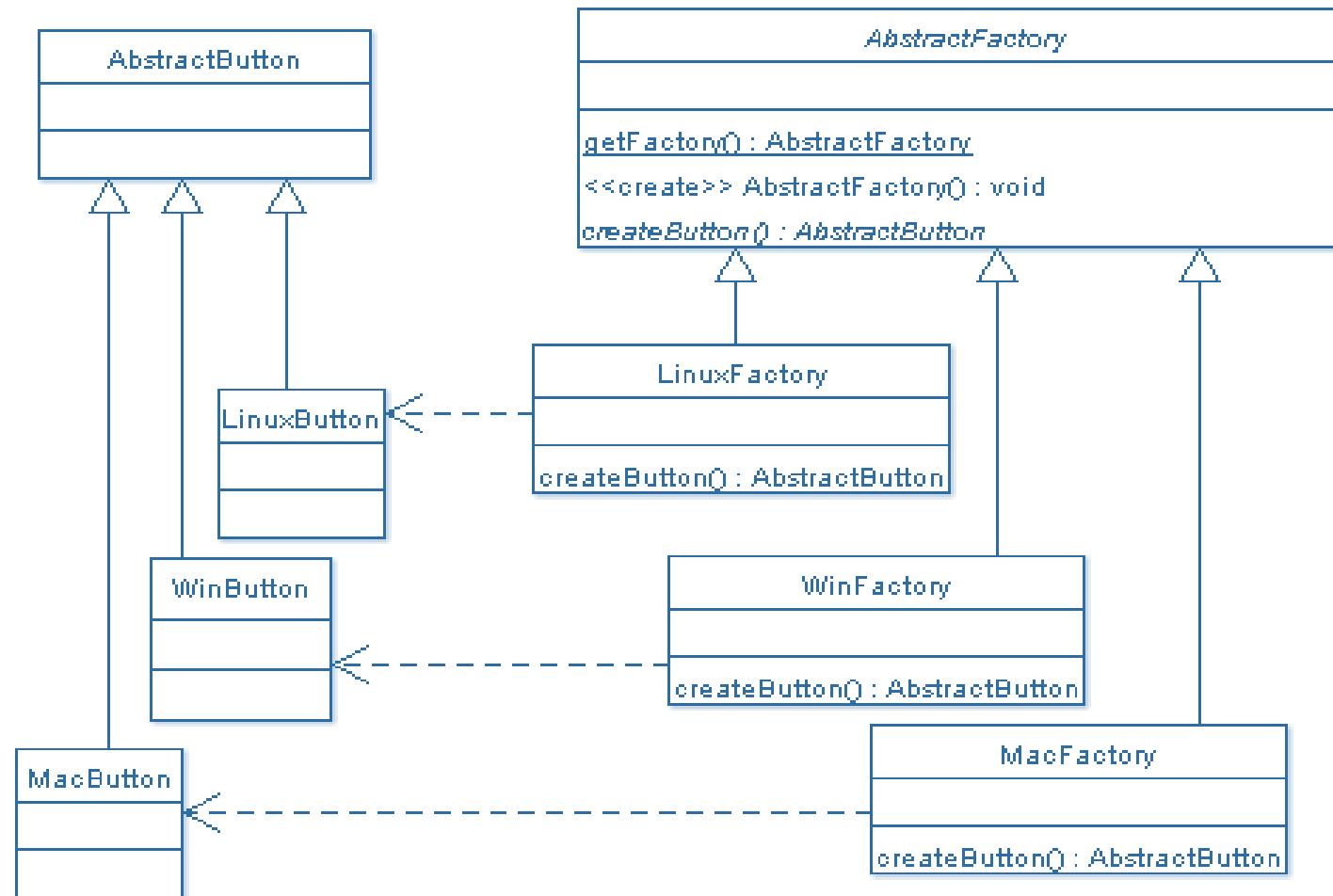
Propósito

Proporciona una interfaz para la creación de distintos tipos de objetos relacionados **sin necesidad de especificar a qué clase concreta pertenecen.**

Otros nombres

Kit o Toolkit

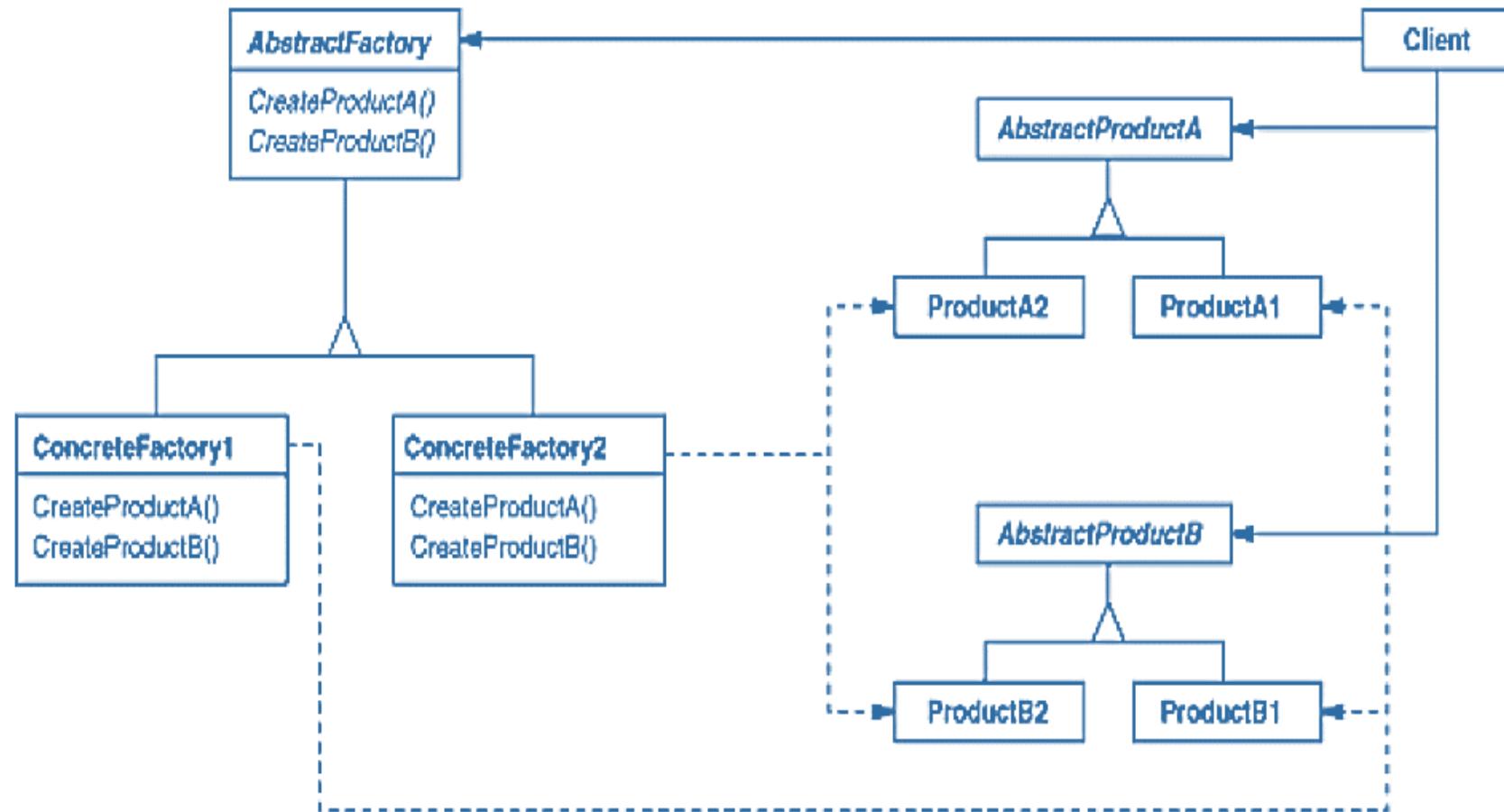
Abstract Factory: Ejemplo



Abstract Factory: *Aplicaciones de uso*

- Un sistema debe ser independiente de los procesos de creación, composición y representación de sus productos
- El uso de este patrón está recomendado para situaciones en las que tenemos una familia de productos concretos y prevemos la inclusión de distintas familias de productos en un futuro.
- Permite que los clientes sólo conozcan las interfaces de los productos y no sus implementaciones concretas.
- Se quiere proporcionar una librería de productos y solo se quiere revelar sus interfaces

Abstract Factory: Estructura



Abstract Factory: *Participantes*

- **AbstractFactory** define la interfaz para la creación de los productos y las **ConcreteFactory** implementan dicha interfaz.
- **AbstractProduct** declara la interfaz para un cierto tipo de producto y los **ConcreteProduct** implementan dicha interfaz y definen un tipo de producto que será creado por la correspondiente **ConcreteFactory**.
- **Los clientes** tan solo dependen de las interfaces **AbstractFactory** y **AbstractProduct**.

Abstract Factory: *Colaboraciones*

- Normalmente se crea una instancia de alguna ConcreteFactory para la creación de los objetos ConcreteProduct. Esto es en tiempo de inicialización.
- AbstractFactory delega la creación de productos concretos (ConcreteProduct) en las ConcreteFactory.

Abstract Factory: *Consecuencias - Ventajas*

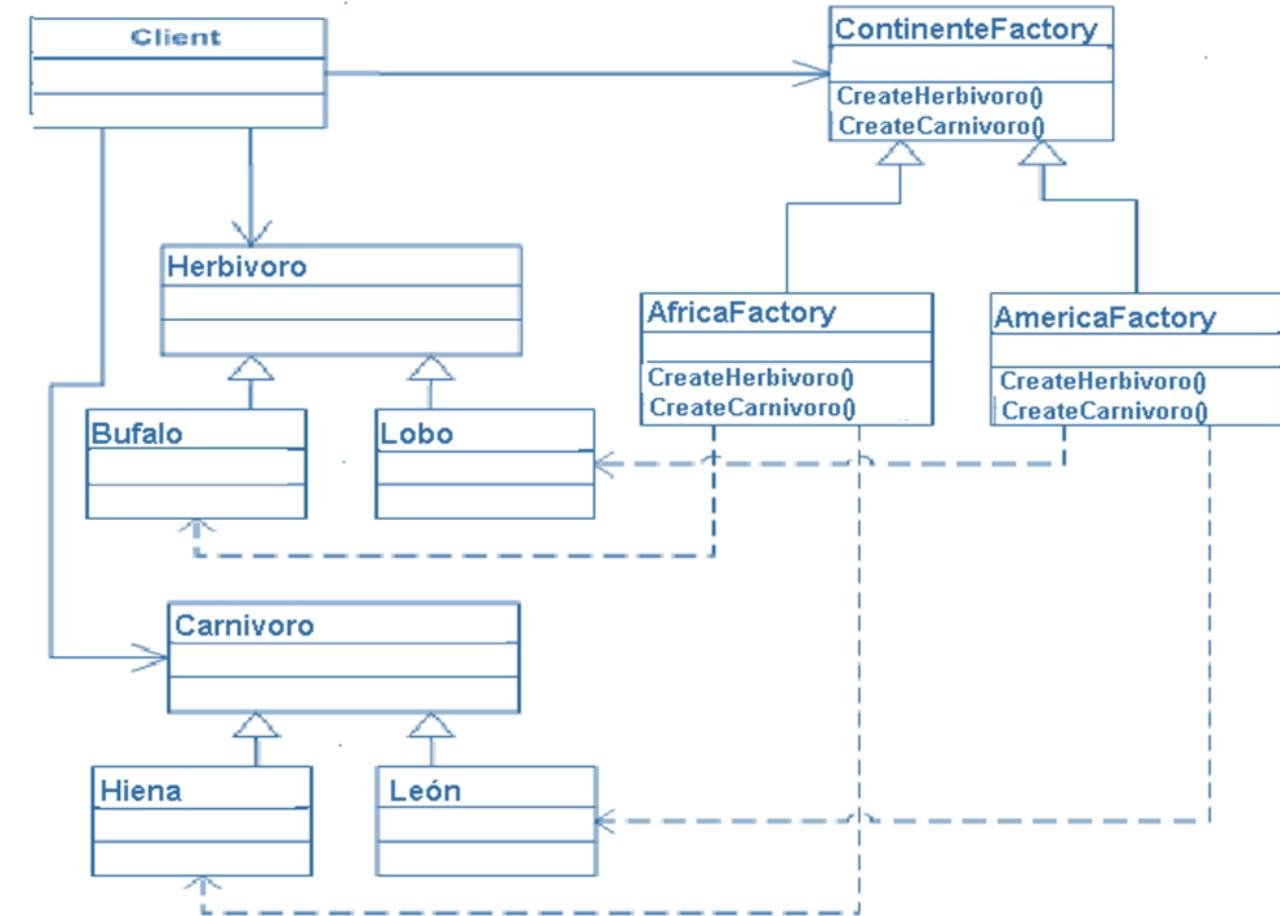
- Aísla las clases de implementación
- Ayuda a controlar los objetos que se crean
- Encapsula la responsabilidad de creación
- Hace fácil el intercambio de familias de productos
 - Cambio de factory -> Cambio de familia
- Fomenta la consistencia entre productos

Abstract Factory: *Consecuencias -Desventajas*

- Puede ser difícil incorporar nuevos tipos de productos (cambiar AbstractFactory y sus factorías concretas)

Abstract Factory: Ejemplo

Un caso de uso de este patrón se da en la creación de familias de Continentes en las cuales los elementos (productos) del interfaz se mantienen constantes (por ejemplo Herbívoro, Carnívoro) pero el dibujado de dichos elementos puede delegarse en distintas familias de forma que, en función de la fábrica seleccionada obtenemos unos animales u otros.





DISEÑO DE SOFTWARE

Gestión de Proyectos

©Ian Sommerville
Software Engineering, 7th edition. Chapter 5

Sesión S10

Objetivos

Conocerá las tareas principales de los gestores de proyectos de software

Comprenderá por qué la naturaleza del software hace mas difícil la gestión de proyectos de software que la gestión de los proyectos de otras ingenierías.

Comprenderá por qué planificar proyectos es esencial en todos los proyectos de software.

Conocerá la forma en que las representaciones gráficas (gráficos de barras y redes de actividades) son utilizadas por los gestores de proyectos para representar las agendas del proyecto.

Conocerá el proceso de gestión de riesgos y algunos de los riesgos que surgen en los proyectos de software.

Contenidos

- Actividades de Gestión.
- Planificación del proyecto.
- Calendarización del proyecto.
- Gestión de riesgos.
- Gestión de Personal

¿Que es la Gestión de proyectos?

¿Que es la Gestión de proyectos?

- La gestión de proyectos de software es una parte esencial de la ingeniería de software.
- Los proyectos necesitan administrarse porque la ISW está sujeta siempre a:
 - **restricciones organizacionales**
 - **presupuesto**
 - **fecha**.
- Administrador del Proyecto es el responsable



¿Qué es la Gestión de proyectos?

Se preocupa en las actividades implicadas en garantizar que el software se :

Entregará a tiempo según la calendarización prevista de acuerdo con los requerimientos de los organismos internacionales de desarrollo y adquisición de software.

La Gestión de proyectos es necesaria porque el desarrollo de software **está siempre sujeto a las limitaciones de presupuesto y el calendario** que son fijados por la organización el desarrollo del software.

Las metas importantes de un proyectos:

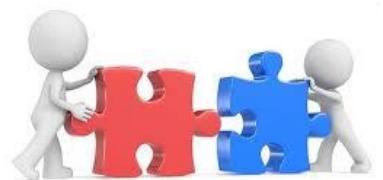
1. Entregar el software al cliente en el tiempo acordado.
2. Mantener costos dentro del presupuesto general.
3. Entregar software que cumpla con las expectativas del cliente.
4. Mantener un equipo de desarrollo óptimo y con buen funcionamiento.



Distinciones de proyectos de software

La ingeniería de Software no es reconocida como una disciplina de ingeniería con el mismo estatus de la mecánica, eléctrica.

- El producto es intangible.
- El producto de software es único.
- Los procesos de software son variables y específicos de la organización
- El proceso de desarrollo de software no está estandarizado.



Actividades de gestión

- Planificación y calendarización del proyecto.
- Redacción y presentación de informes.
- Gestión del riesgo
- Gestión de personal
- Redacción de la propuesta.



Similitudes de Gestión de Proyectos

- Esta gestión de actividades no es exclusiva del software.
- Muchas técnicas de proyectos de ingeniería son igualmente aplicables a la gestión de proyectos de software.
- Los sistemas de ingeniería técnicamente complejos tienden a sufrir los mismos problemas que los sistemas de software.

Staff del Proyecto

Quizas no se podría determinar de manera idónea a la gente que trabajará en el proyecto.

- El presupuesto del proyecto no podría permitir la incorporación de personal altamente remunerado;
- Podríamos no contar con personal con la experiencia adecuada;
- Una organización debería desarrollar las habilidades de sus empleados en proyectos de software.

Los administradores tienen que trabajar sin restricciones especialmente cuando hay escasez de personal capacitado.

Staff del Proyecto

El administrador de proyectos:

Asigna a las personas adecuadas para el proyecto con base en sus áreas de especialización.



Se centra intensamente en el aspecto de la seguridad, para que todos los elementos clave que integran el proyecto estén adecuadamente dirigidos.

Planificación del proyecto

Probablemente la actividad que mayor tiempo consume en la gestión de proyectos.

Actividades continuas desde la concepción inicial hasta la entrega del sistema. Los planes deberían ser revisados periódicamente cuando se disponga de nueva información.

Distintos tipos de planes deben ser desarrollados para apoyar el plan de proyecto principal de software como calendarios y presupuestos.

Tipos de planes de proyectos

Plan	Descripción
Plan de calidad	Describe los procedimientos y estándares de calidad que se utilizarán en un proyecto.
Plan de validación	Describe el enfoque, los recursos y la programación utilizados para la validación del sistema.
Plan de gestión de configuraciones	Describe los procedimientos para la gestión de configuraciones y las estructuras a utilizar.
Plan de mantenimiento	Predice los requerimientos del mantenimiento del sistema, los costes del mantenimiento y el esfuerzo requerido.
Plan de desarrollo del personal	Describe como desarrollan las habilidades y experiencia de los miembros del equipo del proyecto.

Project planning process

Establecer las restricciones del proyecto

Hacer la valoración inicial de los parámetros del proyecto

Definir los hitos del proyecto y productos a entregar

Mientras el proyecto no se haya completado o cancelado **repetir**

Bosquejar la programación el tiempo del proyecto

Iniciar actividades acordes con la programación

Esperar (por un momento)

Revisar el progreso del proyecto

Revisar la estimaciones de los parámetros del proyecto

Actualizar la programación del proyecto

Renegociar las restricciones del proyecto y los productos a entregar

Si (surgen problemas) **entonces**

 Iniciar la revisión técnica y la posible revisión

fin de si

fin de repetir

El plan de proyecto

El plan de proyecto fija:

- Los recursos disponibles del proyecto;
- Divide el trabajo;
- Calendario de trabajo.

Estructura del plan de proyecto

- Introducción
- Organización del proyecto.
- Análisis del riesgo.
- Requerimientos de recursos de hardware y software.
- División del trabajo.
- Programa del proyecto.
- Mecanismos de supervisión e informe.

Organización de actividades

- Las **actividades** en un proyectos deben ser organizadas para producir resultados tangibles para evaluar el progreso de la gestión.
- Los **hitos** son los puntos finales de una actividad del proyecto..
- **Entregables** son los resultados de los proyectos entregados a los clientes.
- El **proceso en cascada** permite la sencilla definición de los hitos de progreso.

Milestones in the RE process



Figura 5.3 Hitos del proceso de especificación de requerimientos.

Calendarización del proyecto

Divida el proyecto en tareas y calcule el tiempo y los recursos necesarios para completar cada tarea.

Organice las tareas simultáneamente para hacer un uso óptimo de la fuerza laboral.

¿**Minimizar las dependencias** de tareas para evitar demoras? Causadas por una tarea que espera a que se complete otra.

Dependientes de la intuición y experiencia de los gerentes de proyecto.

El proceso de calendarización del proyecto

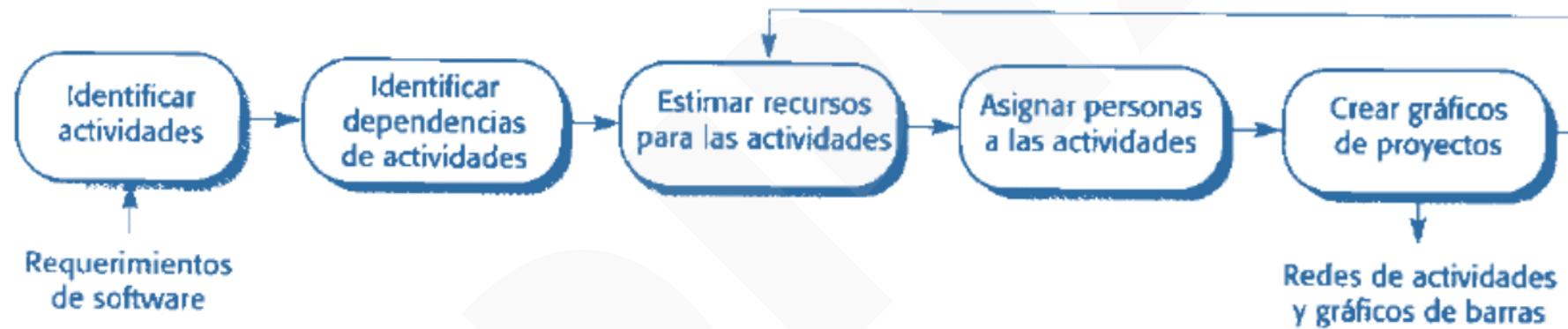


Figura 5.4 El proceso de calendarización del proyecto.

Problemas de calendarización

- La estimación de la dificultad y por lo tanto el costo de desarrollo de la solución es difícil.
- La productividad no es proporcional al número de gente trabajando en una tarea.
- La incorporación tardía de gente al proyecto hace que se atrace mas por problemas de comunicación.
- Lo inesperado siempre sucede. Se deben considerar planes de contingencia.

Gráficos de Barras y Redes de actividades

Notaciones gráficas para ilustrar el calendario del proyecto.

Muestra los hitos en las tareas. Las tareas no deben ser demasiado pequeñas. Ellas deben tomar una o dos semanas.

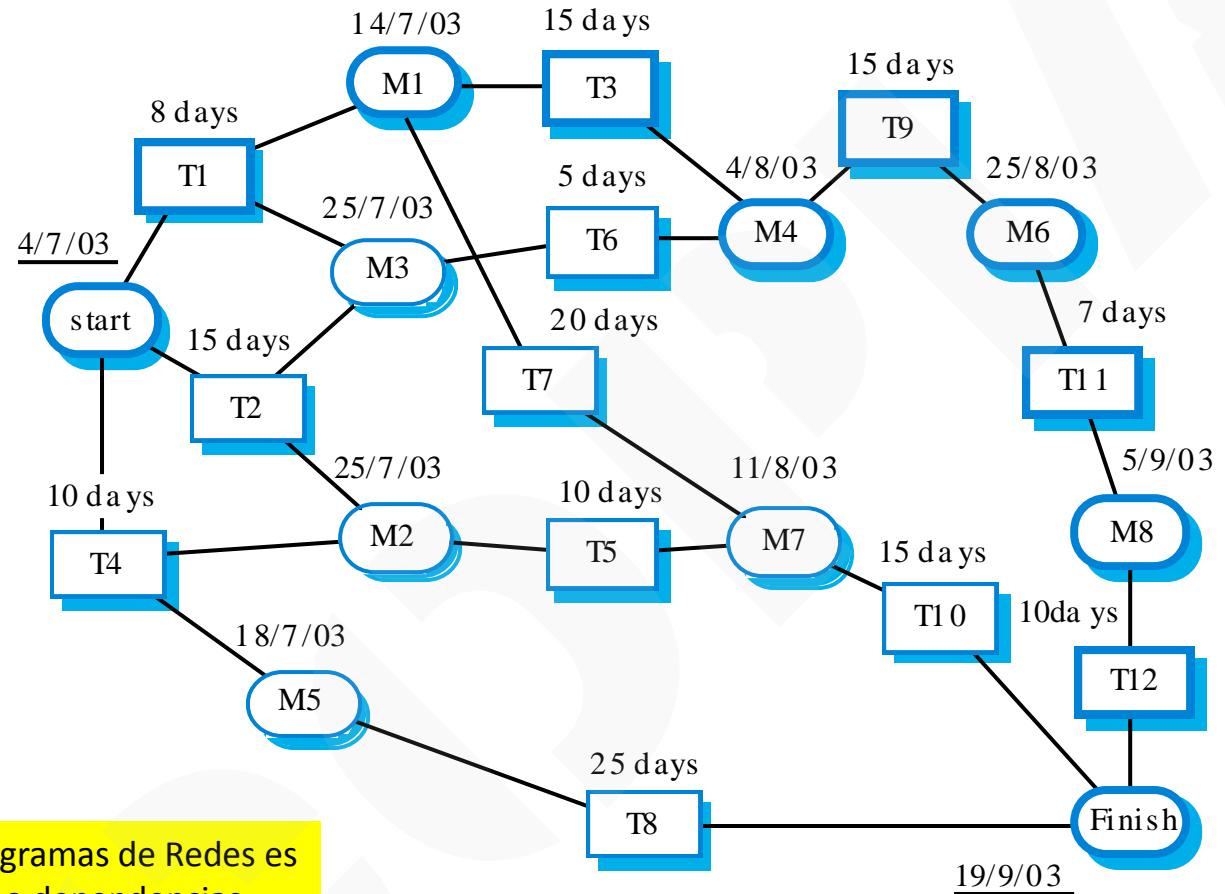
Las redes de actividades muestran las dependencias de tareas y la ruta crítica.

Los gráficos de barras muestran la programación en el tiempo.

Duraciones y dependencias de tareas.

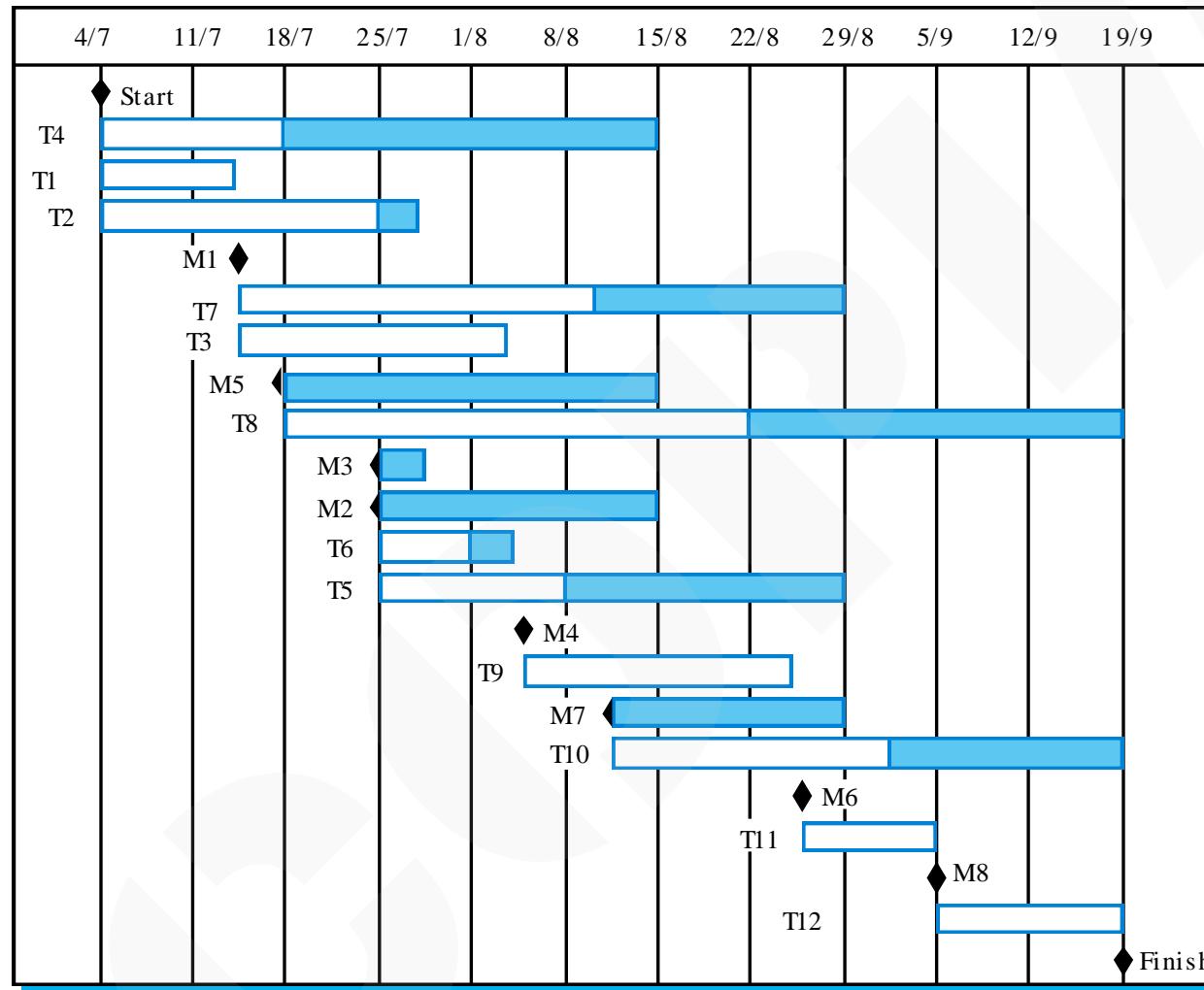
Actividad	Duración (días)	Dependencias
T1	8	
T2	15	
T3	15	T1 (M1)
T4	10	
T5	10	T2, T4 (M2)
T6	5	T1, T2 (M3)
T7	20	T1 (M1)
T8	25	T4 (M5)
T9	15	T3, T6 (M4)
T10	15	T5, T7 (M7)
T11	7	T9 (M6)
T12	10	T11 (M8)

Activity network

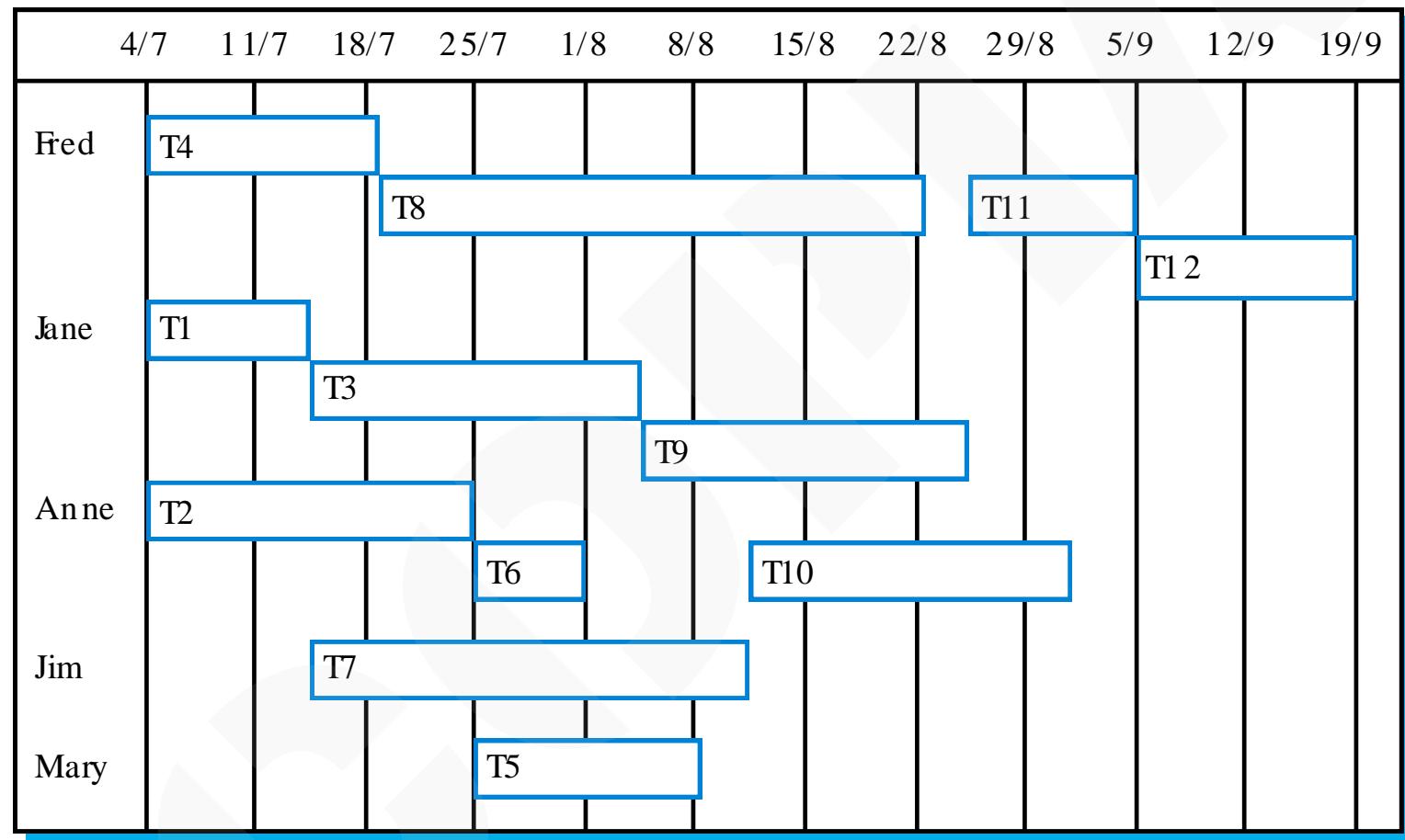


La principal particularidad de los Diagramas de Redes es que permiten mostrar las relaciones o dependencias entre las actividades de un Proyecto utilizando, como elementos básicos de la Programación con ellos, los Eventos y las Actividades

Activity timeline



Staff allocation



Gestión del riesgo

- La gestión del riesgo se refiere a la **identificación** de riesgos y la **elaboración de planes** para reducir al mínimo su efecto sobre un proyecto.
- El riesgo es la probabilidad de que algunas circunstancias adversas se produzcan



Categorías de los riesgos

1. **Riesgos del proyecto** afectan a la programación y los recursos;
2. **Riesgos del producto**, afectan la calidad o rendimiento del software que se esta desarrollando;
3. **Riesgos del negocio o empresariales** estos afectan a la organización que desarrolla o suministra el software.

Gestión del riesgo

Riesgo	Repercute en	Descripción
Rotación de personal	Proyecto	Personal experimentado abandonará el proyecto antes de que éste se termine.
Cambio administrativo	Proyecto	Habrá un cambio de gestión en la organización con diferentes prioridades.
Indisponibilidad de hardware	Proyecto	Hardware, que es esencial para el proyecto, no se entregará a tiempo.
Cambio de requerimientos	Proyecto y producto	Habrá mayor cantidad de cambios a los requerimientos que los anticipados.
Demoras en la especificación	Proyecto y producto	Especificaciones de interfaces esenciales no están disponibles a tiempo.
Subestimación del tamaño	Proyecto y producto	Se subestimó el tamaño del sistema.
Bajo rendimiento de las herramientas CASE	Producto	Las herramientas CASE, que apoyan el proyecto, no se desempeñan como se anticipaba.
Cambio tecnológico	Empresa	La tecnología subyacente sobre la cual se construye el sistema se sustituye con nueva tecnología.
Competencia de productos	Empresa	Un producto competitivo se comercializa antes de que el sistema esté completo.

El proceso de gestión de riesgos

1. Identificación de riesgos

Identifica los riesgos del proyecto, producto y negocio;

2. Análisis de riesgos

Valorar las probabilidades y consecuencias de estos riesgos;

3. Planificación de riesgos

Trazar planes para abordar los riesgos, ya sea para evitarlos o minimizar sus efectos en el proyecto;

4. Supervisión de riesgos

Valorar los riesgos de forma constante y revisar los planes para la mitigación de riesgos tan pronto como la información de los riesgos esté disponible;

El proceso de gestión de riesgos



1. Identificación de riesgos

Se ocupa de identificar los riesgos que pudieran plantear una mayor amenaza al proceso de ingeniería de software, al software a desarrollar, o a la organización que lo desarrolla.

1. **Riesgos tecnológicos** Se derivan de las tecnologías de software o hardware usadas para desarrollar el sistema.
2. **Riesgos personales** Se asocian con las personas en el equipo de desarrollo.
3. **Riesgos organizacionales** Se derivan del entorno organizacional donde se desarrolla el software.
4. **Riesgos de herramientas** Resultan de las herramientas de software y otro software de soporte que se usa para desarrollar el sistema.
5. **Riesgos de requerimientos** Proceden de cambios a los requerimientos del cliente y del proceso de gestionarlos.
6. **Riesgos de estimación** Surgen de las estimaciones administrativas de los recursos requeridos para construir el sistema.

Riesgos y tipos de riesgos

Tipo de riesgo	Riesgos posibles
Tecnológico	<p>La base de datos que se usa en el sistema no puede procesar tantas transacciones por segundo como se esperaba. (1)</p> <p>Los componentes de software de reutilización contienen defectos que hacen que no puedan reutilizarse como se planeó. (2)</p>
Personal	<p>Es imposible reclutar personal con las habilidades requeridas. (3)</p> <p>El personal clave está enfermo e indisponible en momentos críticos. (4)</p> <p>No está disponible la capacitación requerida para el personal. (5)</p>
De organización	<p>La organización se reestructura de modo que diferentes administraciones son responsables del proyecto. (6)</p> <p>Problemas financieros de la organización fuerzan reducciones en el presupuesto del proyecto. (7)</p>
Herramientas	<p>El código elaborado por las herramientas de generación de código de software es ineficiente. (8)</p> <p>Las herramientas de software no pueden trabajar en una forma integrada. (9)</p>
Requerimientos	<p>Se proponen cambios a los requerimientos que demandan mayor trabajo de rediseño. (10)</p> <p>Los clientes no entienden las repercusiones de los cambios a los requerimientos. (11)</p>
Estimación	<p>Se subestima el tiempo requerido para desarrollar el software. (12)</p> <p>Se subestima la tasa de reparación de defectos. (13)</p> <p>Se subestima el tamaño del software. (14)</p>

2. Análisis de riesgos

- Evaluar la probabilidad y gravedad de cada riesgo.
- ***La probabilidad del riesgo*** se puede valorar como :
muy bajo (<10%), bajo (10-25%), moderado (25-50%),
alto (50-75%) o muy alto (>75%).
- ***Los efectos del riesgo*** pueden ser valorados como:
catastrófico, serio, tolerable o insignificante.

Análisis de riesgos (i)

Riesgo	Probabilidad	Efectos
Problemas financieros de la organización fuerzan reducciones en el presupuesto del proyecto. (7)	Baja	Catastrófico
Es imposible reclutar personal con las habilidades requeridas. (3)	Alta	Catastrófico
El personal clave está enfermo e indisponible en momentos críticos. (4)	Moderada	Grave
Los componentes de software de reutilización contienen defectos que hacen que no puedan reutilizarse como se planeó. (2)	Moderada	Grave
Se proponen cambios a los requerimientos que demandan mayor trabajo de rediseño. (10)	Moderada	Grave
La organización se reestructura de modo que diferentes administraciones son responsables del proyecto. (6)	Alta	Grave
La base de datos que se usa en el sistema no puede procesar tantas transacciones por segundo como se esperaba. (1)	Moderada	Grave
Se subestima el tiempo requerido para desarrollar el software. (12)	Alta	Grave
Las herramientas de software no pueden trabajar en una forma integrada. (9)	Alta	Tolerable
Los clientes no entienden las repercusiones de los cambios a los requerimientos. (11)	Moderada	Tolerable
No está disponible la capacitación requerida para el personal. (5)	Moderada	Tolerable
Se subestima la tasa de reparación de defecto. (13)	Moderada	Tolerable
Se subestima el tamaño del software. (14)	Alta	Tolerable
El código elaborado por las herramientas de generación de código de software es ineficiente. (8)	Moderada	Insignificante

3. Planificación de riesgos

Considera cada riesgo y desarrolla una estrategia para gestionar ese riesgo.

Para cada uno de los riesgos, se debe **considerar las acciones** que puede tomar para **minimizar** la perturbación del proyecto si se produce el problema identificado en el riesgo.

3. Planificación de riesgos

Estrategias de gestión del riesgo :

1. *Estrategias de prevención*

La probabilidad de que el riesgo aparezca se reduce;

2. *Estrategias de minimización*

Se reduce el impacto del riesgo en el proyecto o producto;

3. *Planes de contingencia*

Si se concreta el riesgo, los planes de contingencia son necesarios para hacer frente a ese riesgo;

Estrategias de gestión de riesgos

Riesgo	Estrategia
Problemas financieros de la organización	Prepare un documento informativo para altos ejecutivos en el que muestre cómo el proyecto realiza una aportación muy importante a las metas de la empresa y presente razones por las que los recortes al presupuesto del proyecto no serían efectivos en costo.
Problemas de reclutamiento	Alerte al cliente de dificultades potenciales y de la posibilidad de demoras; investigue la compra de componentes.
Enfermedad del personal	Reorganice los equipos de manera que haya más traslape de trabajo y, así, las personas comprendan las labores de los demás.
Componentes defectuosos	Sustituya los componentes potencialmente defectuosos con la compra de componentes de conocida fiabilidad.
Cambios de requerimientos	Obtenga información de seguimiento para valorar el efecto de cambiar los requerimientos; maximice la información que se oculta en el diseño.
Reestructuración de la organización	Prepare un documento informativo para altos ejecutivos en el que muestre cómo el proyecto realiza una aportación muy importante a las metas de la empresa.
Rendimiento de la base de datos	Investigue la posibilidad de comprar una base de datos de mayor rendimiento.
Subestimación del tiempo de desarrollo	Investigue los componentes comprados; indague el uso de un generador de programa.

4. Supervisión de riesgos

- Valora cada uno de los riesgos identificados para decidir si éste es más o menos probable y si han cambiado sus efectos.
- Cada factor de riesgo debe ser discutido en las reuniones de gestión de progreso del proyecto.

Factores de riesgo

Tipo de riesgo	Indicadores potenciales
Tecnológico	Entrega tardía de hardware o software de soporte; muchos problemas tecnológicos reportados.
Personal	Baja moral de personal; malas relaciones entre miembros del equipo; alta rotación de personal.
De organización	Chismes en la organización; falta de acción de los altos ejecutivos.
Herramientas	Renuencia de los miembros del equipo para usar herramientas; quejas acerca de las herramientas CASE; demandas por estaciones de trabajo mejor equipadas.
Requerimientos	Muchas peticiones de cambio de requerimientos; quejas de los clientes.
Estimación	Falla para cumplir con el calendario acordado; falla para corregir los defectos reportados.

Gestión de personal

Las personas que trabajan en una organización de software son los activos más importantes.

- Cuesta mucho dinero reclutar y retener al buen personal, así que depende de los administradores de software garantizar que la organización obtenga el mejor aprovechamiento posible por su inversión.



Factores críticos en la gestión de personal

1. **Consistencia** Todas las personas en un equipo de proyecto deben recibir un trato similar..
2. **Respeto** Las personas tienen distintas habilidades y los administradores deben respetar esas diferencias. Todos los miembros del equipo deben recibir una oportunidad para aportar.
3. **Inclusión** Las personas contribuyen efectivamente cuando sienten que otros las escuchan y que sus propuestas se toman en cuenta.
4. **Honestidad** Como administrador, siempre debe ser honesto acerca de lo que está bien y lo que está mal en el equipo. También debe ser honesto respecto a su nivel de conocimiento técnico y voluntad para comunicar al personal más conocimiento cuando sea necesario. Si trata de encubrir la ignorancia o los problemas, con el tiempo, éstos saldrán a la luz y perderá el respeto del grupo.

GP- Motivación del personal

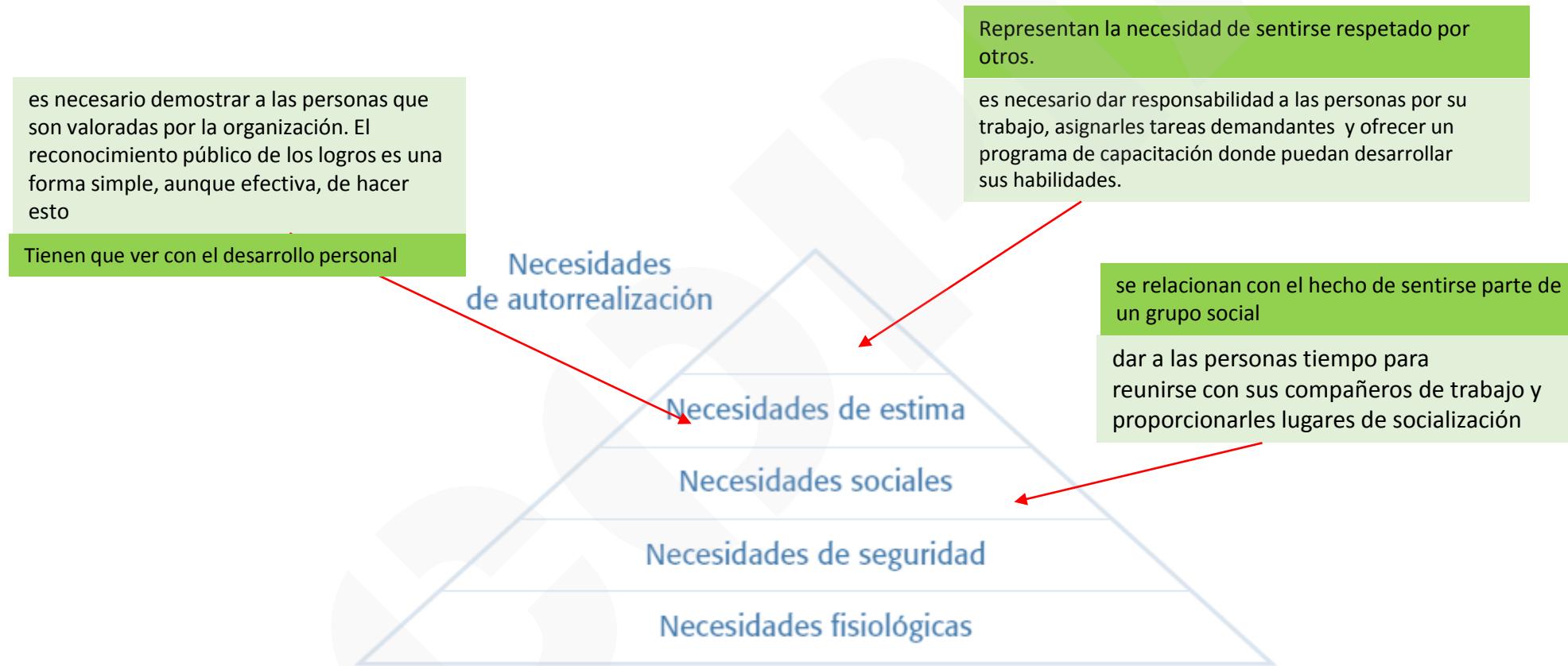
Como administrador de proyecto, usted necesitará motivar a las personas con quienes trabaja, de manera que éstas contribuyan con lo mejor de sus habilidades.

- Motivación significa organizar el trabajo y el ambiente laboral para alentar a los individuos a desempeñarse tan efectivamente como sea posible.
- Si las personas no están motivadas, no estarán interesadas en la actividad que realizan.



GP- Motivación del personal

Para fomentar este ánimo, hay que saber un poco acerca de qué motiva a la gente. Se sugiere que las personas se sienten ***motivadas para cubrir sus necesidades***, las cuales se ordenan en una serie de niveles:



Gestión de personal

El tipo de personalidad también influye en la motivación

1. **Personas orientadas a las tareas**, quienes están motivadas por el trabajo que realizan En la ingeniería de software se trata de personas que están motivadas por el reto intelectual de desarrollar software.
2. **Personas orientadas hacia sí mismas**, quienes están motivadas principalmente por el éxito y el reconocimiento personales. Están interesadas en el desarrollo del software como un medio para lograr sus propias metas.
3. **Personas orientadas a la interacción**, quienes están motivadas por la presencia y las acciones de los compañeros de trabajo. Conforme el desarrollo de software se vuelve más centrado en el usuario, los individuos orientados a la interacción se involucran más en la ingeniería de software.



Gestión de personal

- Las personas orientadas a la interacción comúnmente disfrutan al trabajar como parte de un grupo,
- mientras que quienes están orientadas a las tareas o hacia sí mismas prefieren actuar individualmente.
- Las mujeres tienen más probabilidad que los hombres de estar orientadas a la interacción.

Gestión de personal - Trabajo en equipo

La mayor parte del software profesional se desarrolla mediante equipos de proyecto, cuyo número de miembros varía entre dos y varios cientos de personas.

- Conformar un grupo que tiene el equilibrio justo de habilidades técnicas, experiencia y personalidades es una tarea administrativa fundamental
- Un buen equipo es cohesivo y tiene espíritu de grupo

Gestión de personal

En un grupo cohesivo, los miembros piensan que el equipo es más importante que los individuos que lo integran. Los miembros de un grupo cohesivo bien liderado son leales al equipo. Se identifican con las metas del grupo y con los demás miembros



* Esto hace que el grupo sea sólido y pueda enfrentar problemas y situaciones inesperadas.

Gestión de personal

Los beneficios de crear un grupo cohesivo son:

1. ***El grupo puede establecer sus propios estándares de calidad*** Puesto que dichos estándares se establecen por consenso, éstos tienen más probabilidad de respetarse que los estándares externos impuestos sobre el grupo.
2. ***Los individuos aprenden de los demás y se apoyan mutuamente*** Las personas en el grupo aprenden de los demás. Las inhibiciones causadas por la ignorancia se minimizan mientras se promueve el aprendizaje mutuo.
3. ***El conocimiento se comparte*** Puede mantenerse la continuidad si sale un miembro del grupo. Otros en el grupo pueden tomar el control de las tareas críticas para asegurar que el proyecto no se altere en forma considerable.
4. ***Se alientan la refactorización y el mejoramiento continuo*** Los miembros del grupo trabajan de manera colectiva para entregar resultados de alta calidad y corregir problemas, sin importar quiénes crearon originalmente el diseño o programa.

Gestión de personal

factores genéricos que afectan el trabajo en equipo:

1. ***Las personas en el grupo*** Se necesita una combinación de personas en un grupo de proyecto, puesto que el desarrollo de software implica diversas actividades, como negociación con clientes, programación, pruebas y documentación.
2. ***La organización grupal*** Un grupo debe organizarse de forma que los individuos puedan contribuir con sus mejores habilidades y completar las tareas como se esperaba.
3. ***Comunicaciones técnicas y administrativas*** Es esencial la óptima comunicación entre los miembros del grupo, y entre el equipo de ingeniería de software y otras partes interesadas en el proyecto.



GP- Selección de los miembros del grupo

La labor de un administrador o líder de equipo es crear un grupo cohesivo y organizar a los miembros del grupo para que puedan trabajar en conjunto de manera efectiva.

- Esto implica crear un grupo con el equilibrio correcto de habilidades técnicas y personalidades, así como organizarlo para que los miembros trabajen adecuadamente en conjunto
- los administradores pocas veces tienen absoluta libertad en la selección del equipo.
- Con frecuencia deben recurrir a las personas que estén disponibles en la compañía, aun cuando no sean ideales para el puesto.
- Un grupo con personalidades complementarias puede trabajar mejor que un grupo seleccionado exclusivamente por la habilidad técnica.
- Es probable que las personas que están **motivadas por el trabajo** sean las más fuertes técnicamente
- Las personas que son **orientadas hacia sí mismas** tal vez serán mejores para impulsar el trabajo hacia delante para terminar la tarea.
- Las personas **orientadas a la interacción** ayudan a facilitar las comunicaciones dentro del grupo

GP -Organización del grupo

La forma en que se organiza un grupo influye en las decisiones que toma dicho grupo, las maneras como se intercambia la información y las interacciones entre el grupo de desarrollo y los participantes externos del proyecto

Las preguntas organizacionales importantes para los administradores de proyecto incluyen:

- ¿El administrador del proyecto debe ser el líder técnico del grupo?
- ¿Quién se encargará de tomar las decisiones técnicas críticas, y cómo se tomarán? ¿Las decisiones las tomará el arquitecto del sistema, el administrador del proyecto o se llegará a un consenso entre un rango más amplio de miembros del equipo?
- ¿Cómo se manejarán las interacciones con los participantes externos y los altos directivos de la compañía?
- ¿Cómo es posible que los grupos logren integrar a personas que no se localizan en el mismo lugar?.
- ¿Cómo puede compartirse el conocimiento a través del grupo?

GP -Organización del grupo

Los grupos de programación pequeños, por lo general, están organizados en una forma **bastante informal**. Los grupos de programación extrema siempre son grupos informales.

- ***Los grupos informales*** pueden ser muy exitosos, en particular cuando la mayoría de los miembros del grupo **son experimentados y competentes**.
- ***Los grupos jerárquicos*** son grupos que comparten una estructura jerárquica con el líder del grupo en la parte superior del escalafón. El líder tiene autoridad más formal que los miembros del grupo y así puede dirigir el trabajo

GP –Comunicacion del grupo

Es absolutamente esencial que los miembros del grupo se comuniquen efectiva y eficientemente entre sí y con otras partes interesadas en el proyecto.

- Los miembros del grupo deben intercambiar información acerca del estatus de su trabajo, las decisiones de diseño que se tomaron y los cambios a las decisiones de diseño previas.
- Tienen que resolver los problemas que surjan con otros interesados en el proyecto e informar a éstos sobre los cambios al sistema, grupo y planes de entrega.

GP- Comunicaciones grupales

La efectividad y la eficiencia de las comunicaciones están influidas por:

1. **Tamaño del grupo** Conforme el grupo crece, se hace más difícil que los miembros se comuniquen de manera efectiva. El número de vínculos de comunicación de un canal es $n * (n - 1)$, donde n es el tamaño del grupo.
2. **Estructura del grupo** Las personas en los grupos estructurados de manera informal se comunican más efectivamente que los individuos en grupos con una estructura jerárquica formal. En los grupos jerárquicos, las comunicaciones tienden a fluir hacia arriba y abajo de la jerarquía. Las personas en el mismo nivel tal vez no se comuniquen entre sí.
3. **Composición del grupo** Las personas con los mismos tipos de personalidad pueden chocar y, como resultado, las comunicaciones se inhiben. Además, por lo regular, la comunicación es mejor en los grupos integrados por personas de uno y otro género que en los grupos formados por miembros de un solo género.
4. **El ambiente laboral físico** La organización del centro de trabajo es un factor importante para facilitar o inhibir las comunicaciones.
5. **Los canales de comunicación disponibles** Existen muchas formas diferentes de comunicación: cara a cara, correo electrónico, documentos formales, teléfono y tecnologías Web 2.0, como las redes sociales y los wikis. Conforme los equipos de proyecto se distribuyen cada vez más, con miembros de equipo que trabajan en lugares remotos, es necesario utilizar varias tecnologías para facilitar las comunicaciones.





UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.



DISEÑO DE SOFTWARE

Patrones de Diseño de Estructurales - Comportamiento

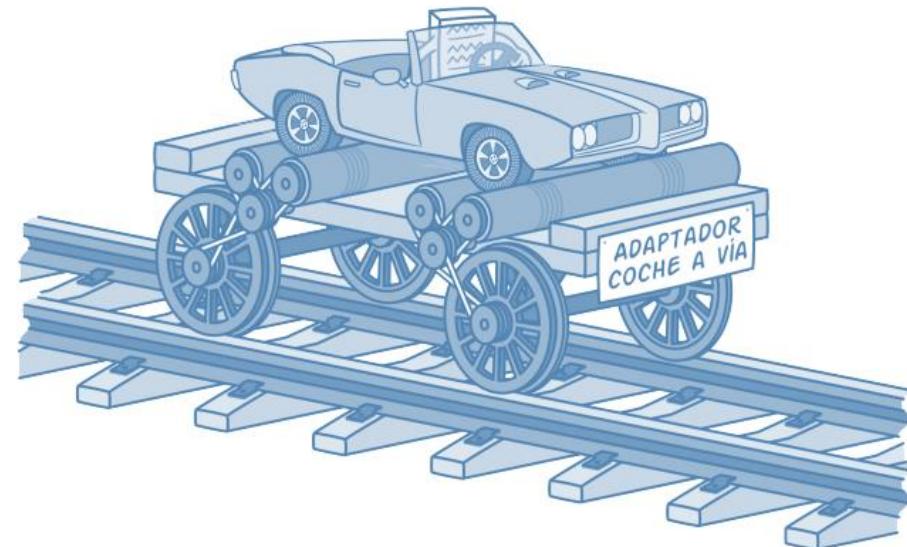
Patrones estructurales

Los patrones estructurales explican cómo **ensamblar** objetos y clases en estructuras más grandes, a la vez que se mantiene la flexibilidad y eficiencia de estas estructuras.

- **Adapter:** *cambia la interfaz de una clase a la de otra.*
- **Bridge:** *permite mantener constante la interfaz que se presenta al cliente, cambiando la clase que se usa*
- **Composite:** *una colección de objetos*
- **Decorator:** *una clase que envuelve a una clase dándole nuevas capacidades.*
- **Facade:** *reúne una jerarquía compleja de objetos y provee una clase nueva permitiendo acceder a cualquiera de las clases de la jerarquía.*
- **Flyweight:** *permite limitar la proliferación de pequeñas clases similares.*

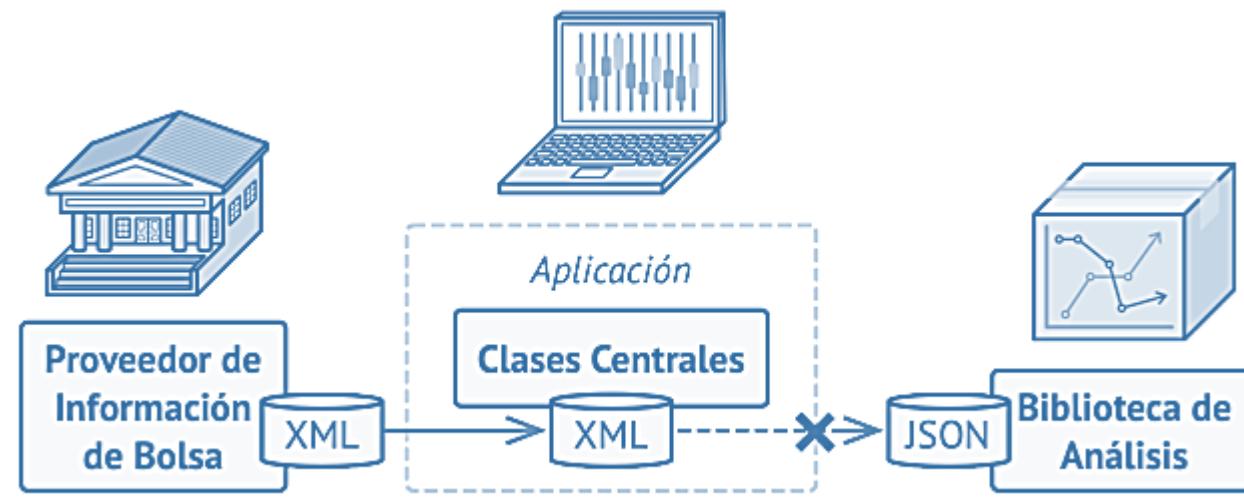
Patrón Adapter

Adapter es un patrón de diseño estructural que permite la colaboración entre objetos con interfaces incompatibles.



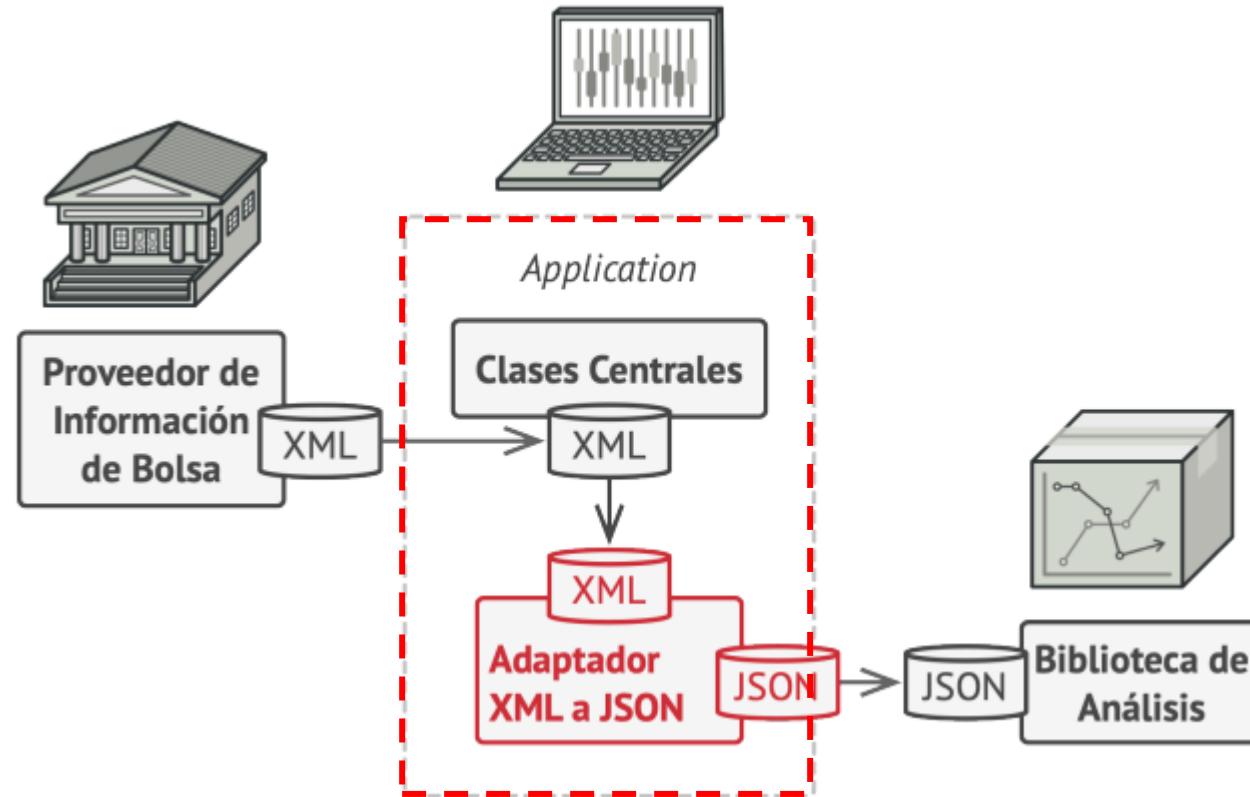
Patrón Adapter

Problema:



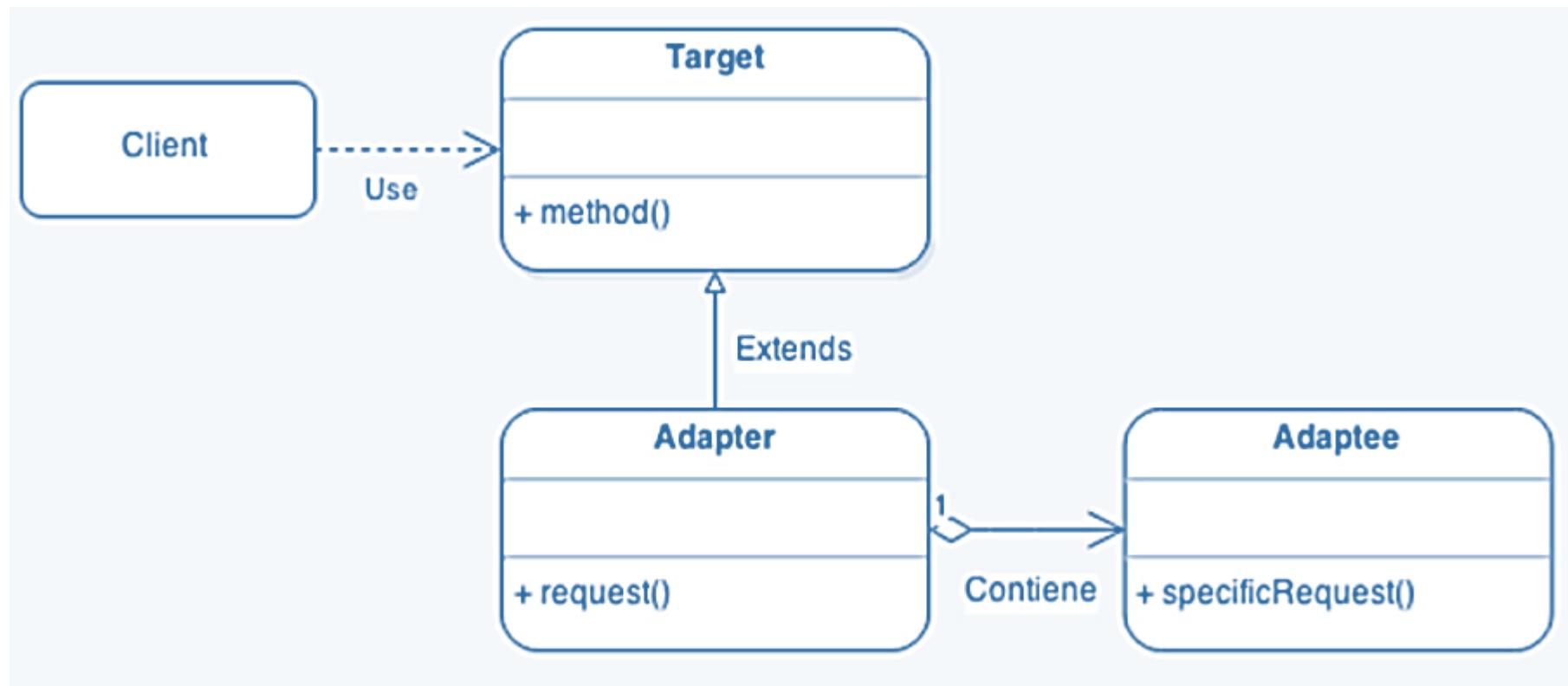
Patrón Adapter

Solución:



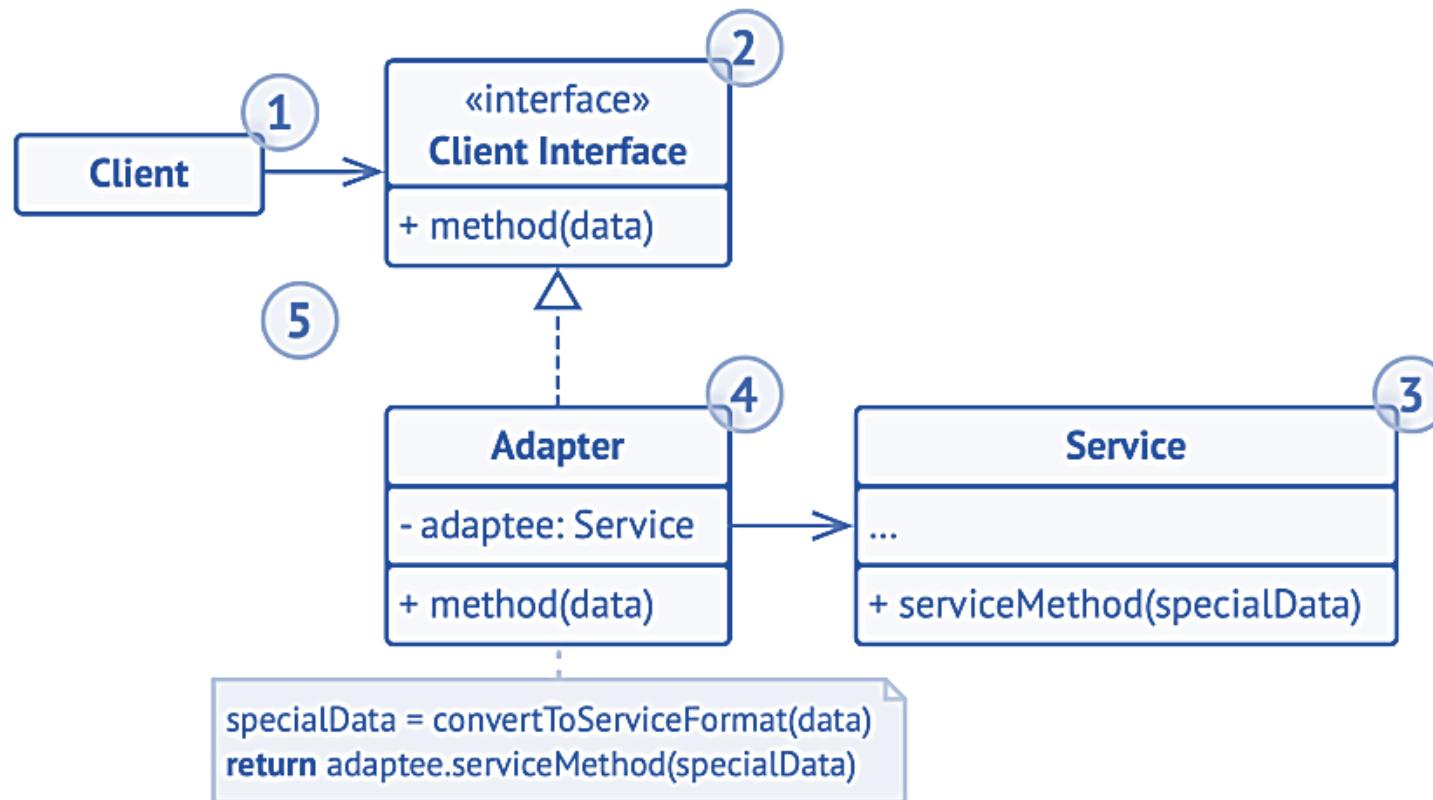
Patrón Adapter

Estructura:



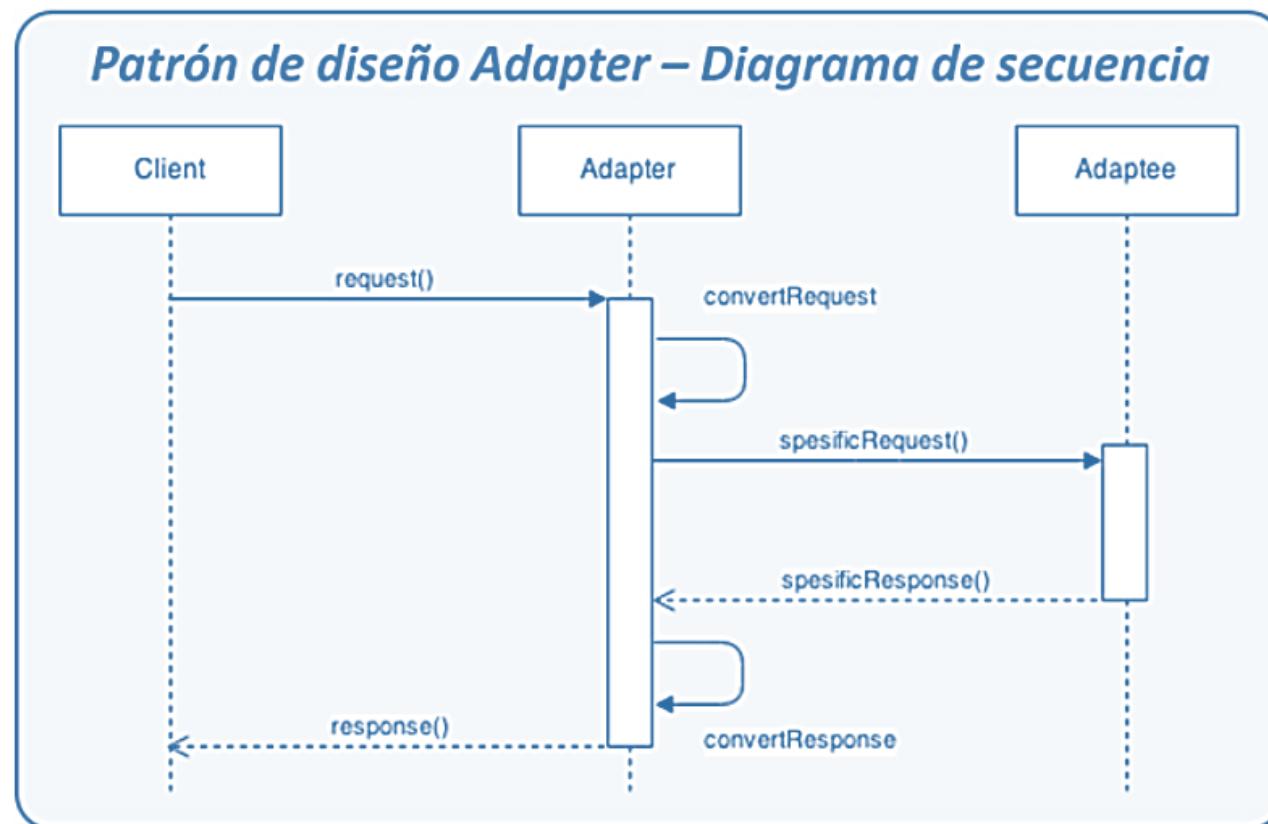
Patrón Adapter

Estructura:



Patrón Adapter

Colaboración:



Patrón Adapter

Cuando Aplicar:

- Utiliza la clase adaptadora cuando quieras usar una clase existente, pero cuya interfaz no sea compatible con el resto del código.
- Utiliza el patrón cuando quieras reutilizar varias subclases existentes que carezcan de alguna funcionalidad común que no pueda añadirse a la superclase.

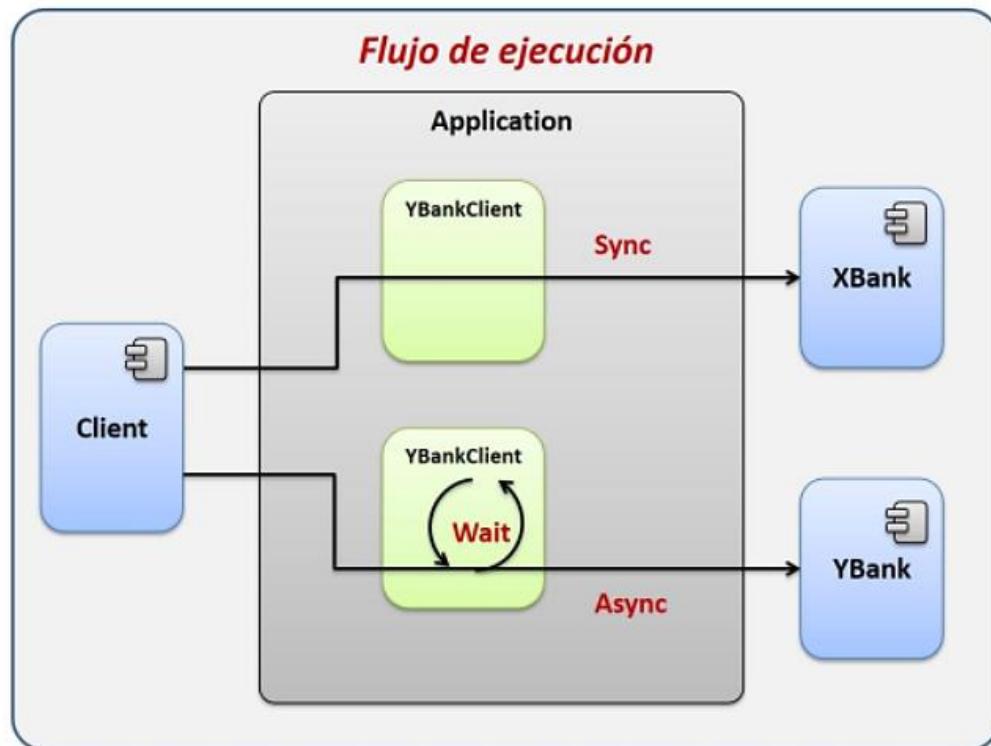
Patrón Adapter

Cómo implementarlo:

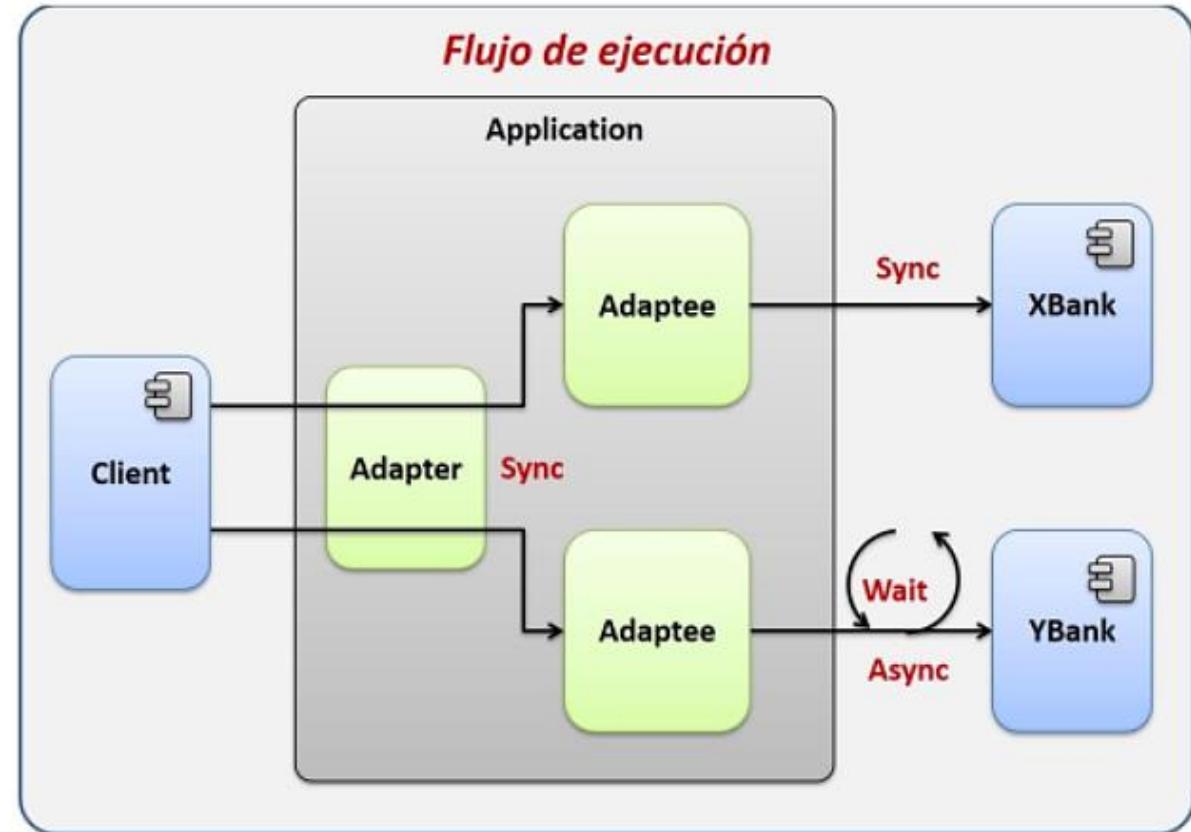
1. Asegure de que tiene al menos dos clases con interfaces incompatibles:
 - Una útil clase servicio que no puedes cambiar
 - Una o varias clases cliente que se beneficiarían de contar con una clase de servicio.
2. Declara la interfaz con el cliente y describe el modo en que las clases cliente se comunican con la clase de servicio.
3. Crea la clase adaptadora y haz que siga la interfaz con el cliente.
4. Añade un campo a la clase adaptadora para almacenar una referencia al objeto de servicio.
5. Uno por uno, implementa todos los métodos de la interfaz con el cliente en la clase adaptadora.
6. Las clases cliente deberán utilizar la clase adaptadora a través de la interfaz con el cliente.

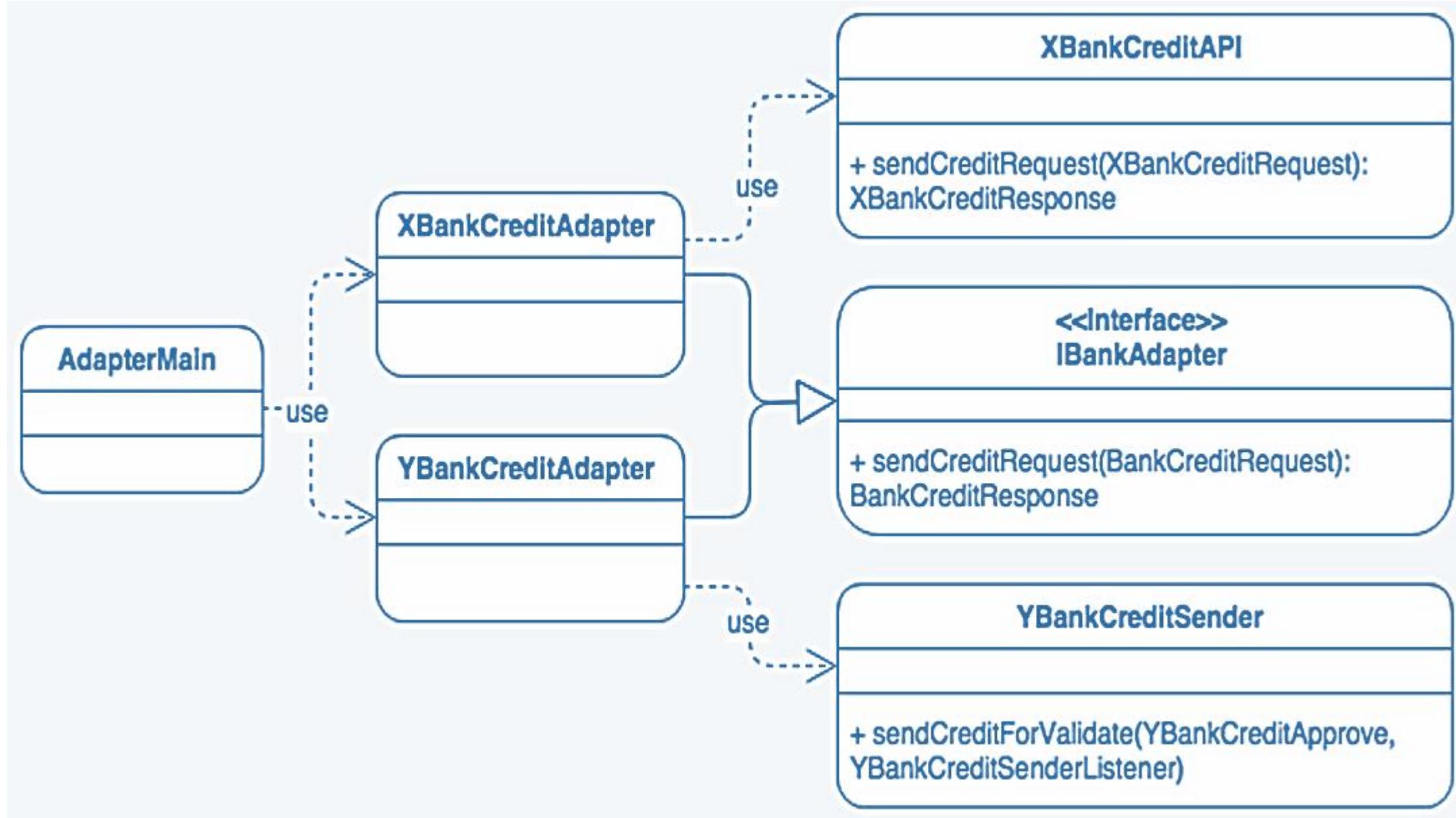
Patrón Adapter

El escenario:



La solución:





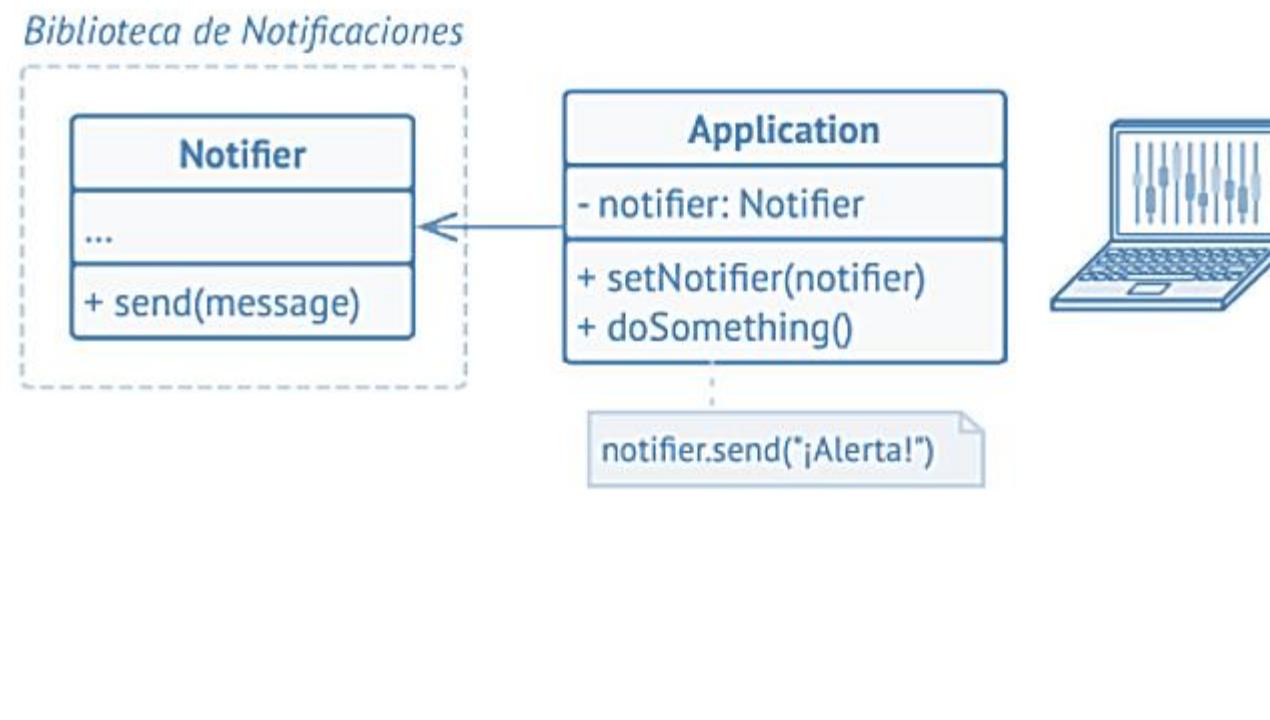
Patrón Decorator

Decorator es un patrón de diseño estructural que te permite añadir funcionalidades a objetos colocando estos objetos dentro de objetos encapsuladores especiales que contienen estas funcionalidades.



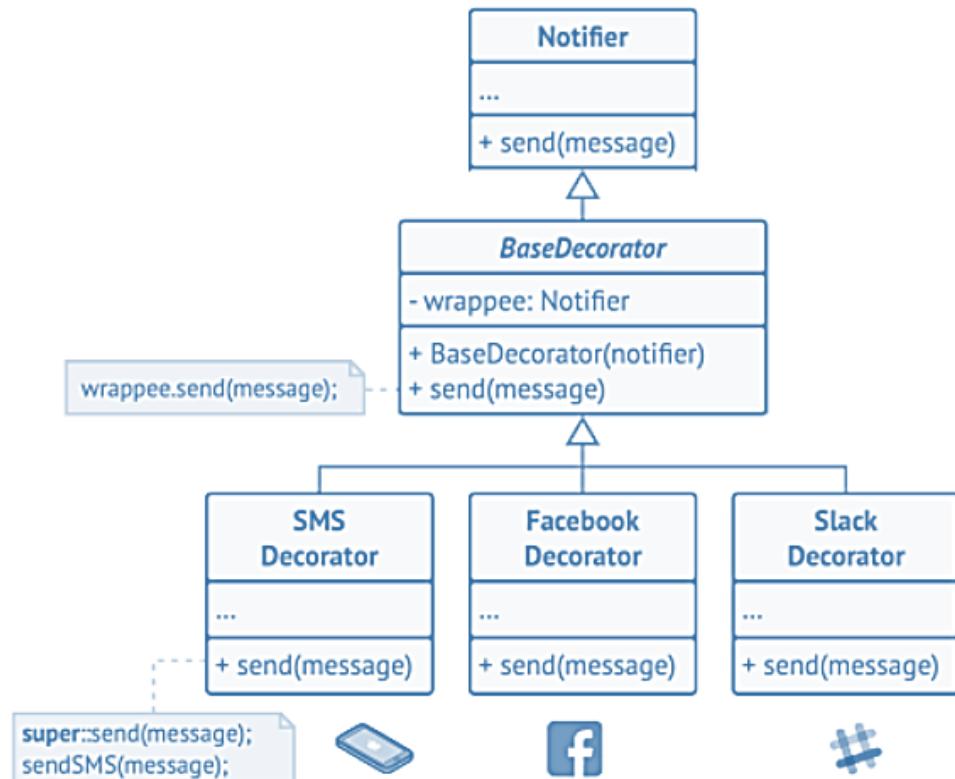
Patrón Decorator

Problema:



Patrón Decorator

Solución:

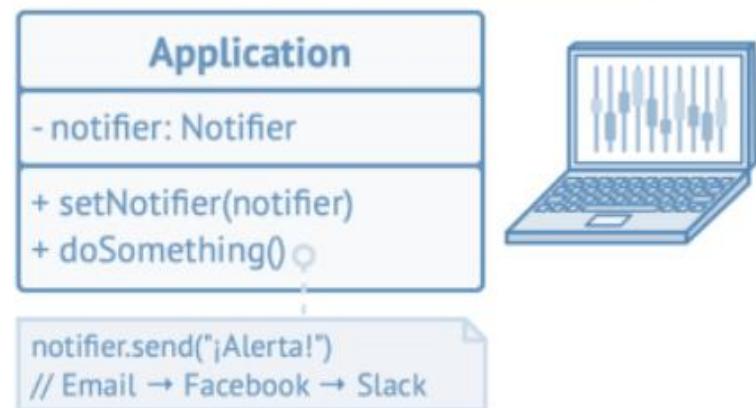


Varios métodos de notificación se convierten en decoradores.

Patrón Decorator

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

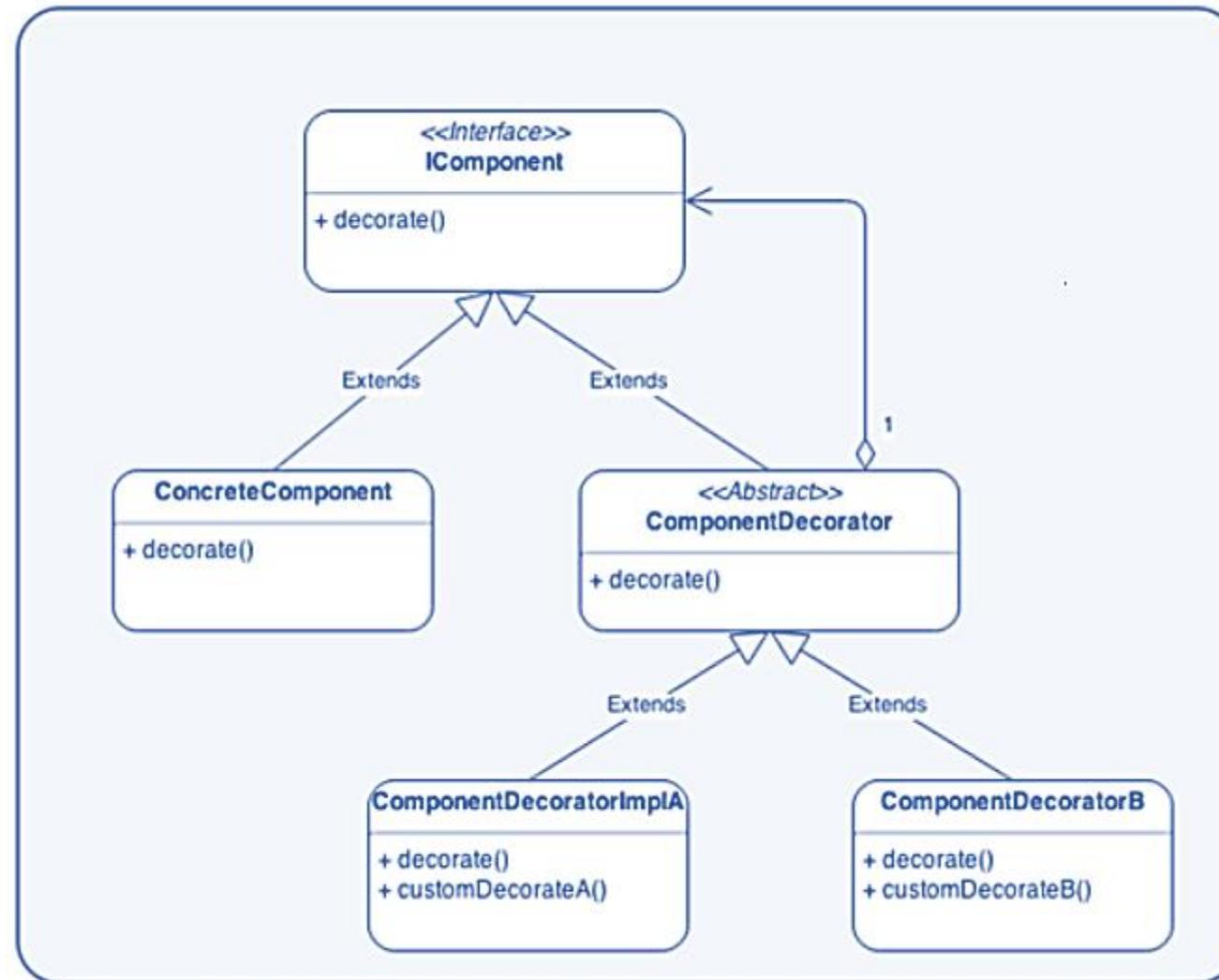
app.setNotifier(stack)
```



Las aplicaciones pueden configurar pilas complejas de decoradores de notificación.

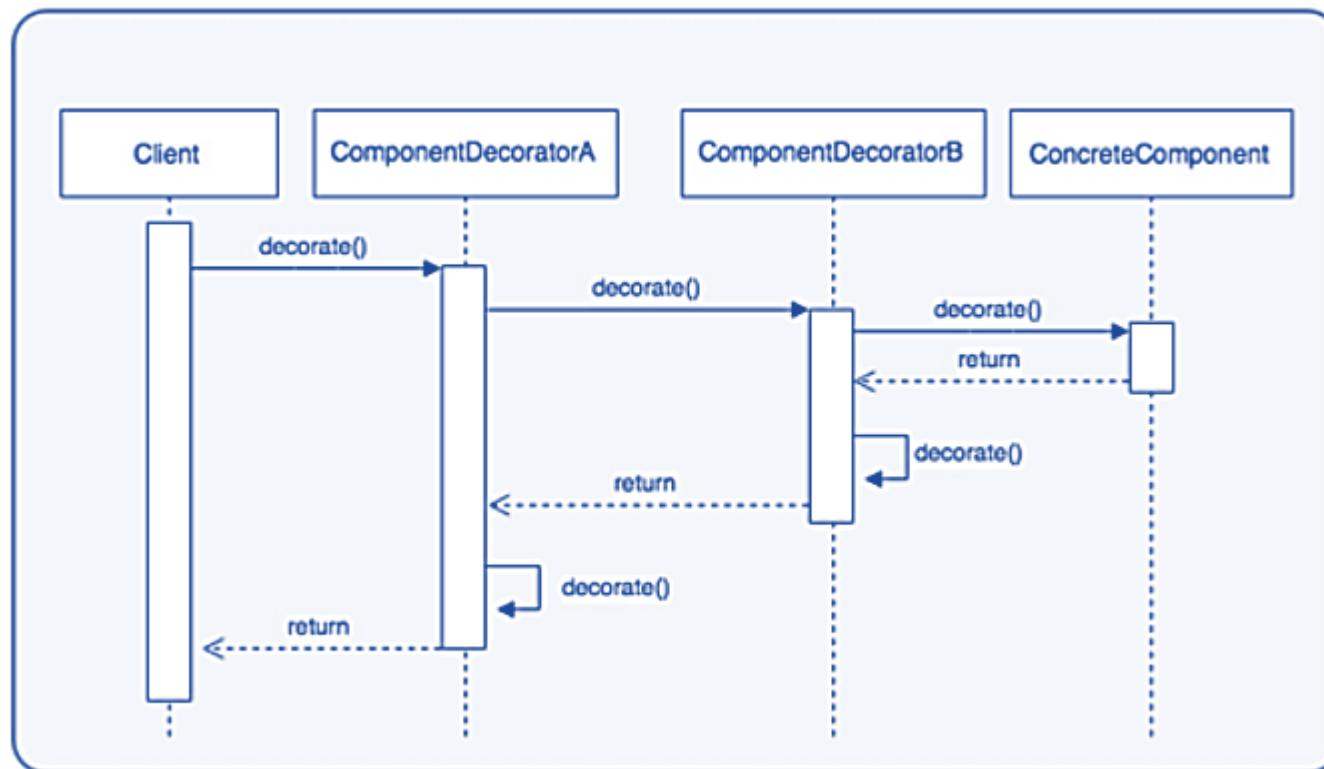
Patrón Decorator

Estructura:



Patrón Decorator

Estructura:



Patrón Decorator

Cuando Aplicar:

- Utiliza el patrón Decorator cuando necesites asignar funcionalidades adicionales a objetos durante el tiempo de ejecución sin descomponer el código que utiliza esos objetos
- Utiliza el patrón cuando resulte extraño o no sea posible extender el comportamiento de un objeto utilizando la herencia.

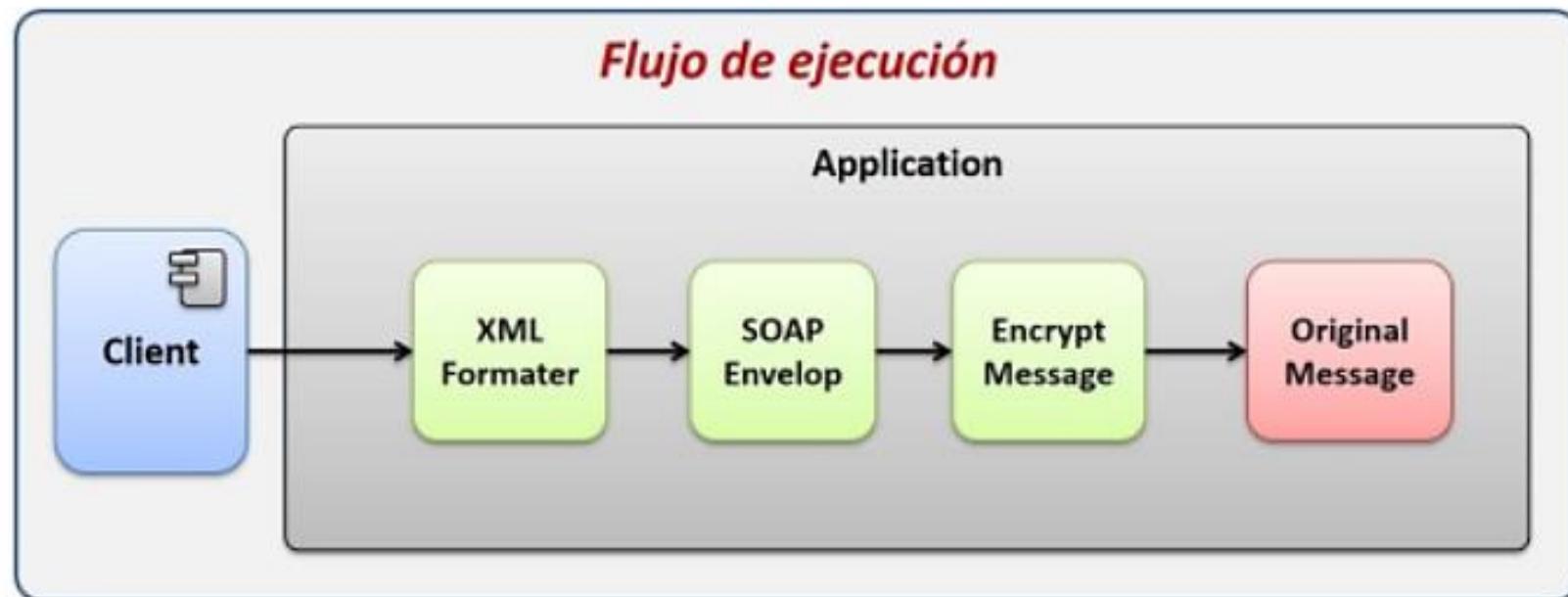
Patrón Decorator

Cómo implementarlo:

1. Asegúrate de que tu dominio de negocio puede representarse como un componente primario con varias capas opcionales encima.
2. Decide qué métodos son comunes al componente primario y las capasopcionales.
3. Crea una clase concreta de componente y define en ella el comportamiento base.
4. Crea una clase base decoradora.
5. Asegúrate de que todas las clases implementan la interfaz de componente.
6. Crea decoradores concretos extendiéndolos a partir de la decoradora base.
7. El código cliente debe ser responsable de crear decoradores y componerlos del modo que el cliente necesite.

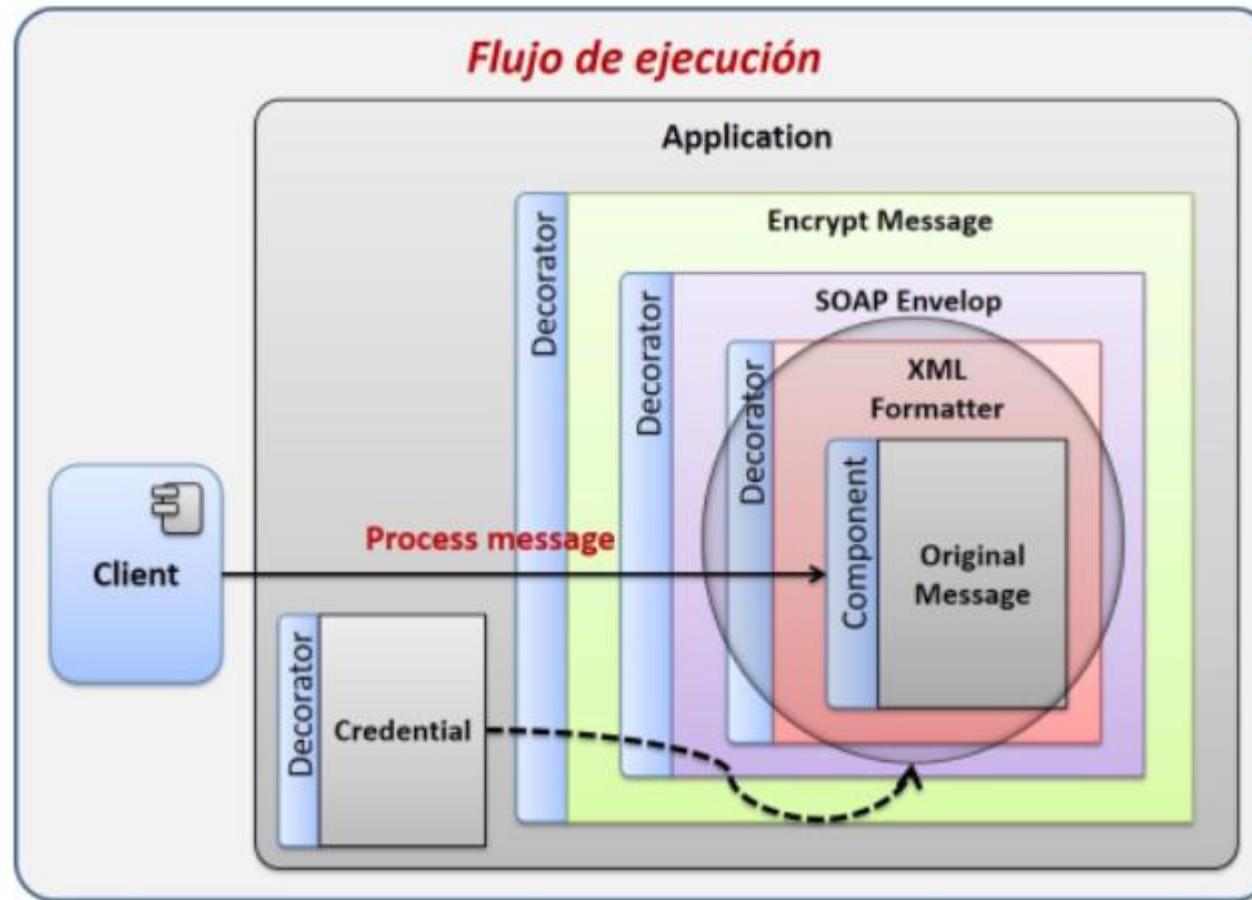
Patrón Decorator

El escenario:

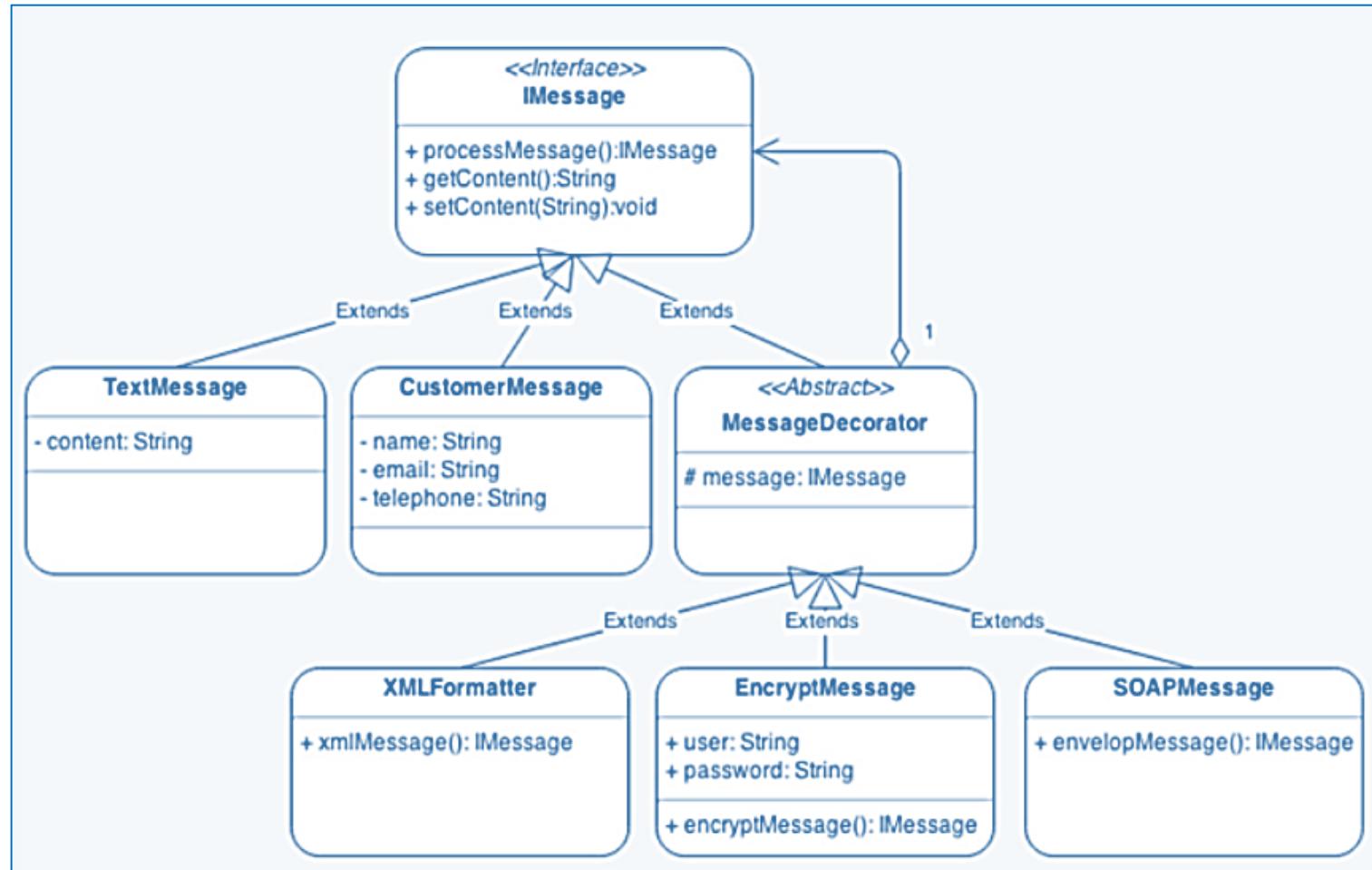


Patrón Decorator

La solución:

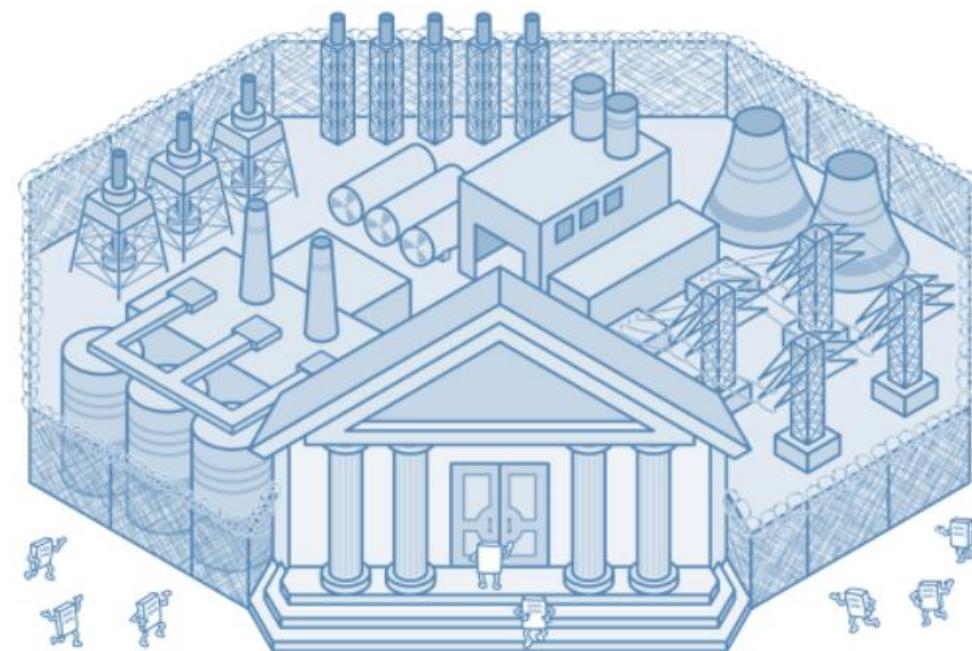


Patrón Decorator



Patrón Facade

Facade es un patrón de diseño estructural que proporciona una interfaz simplificada a una biblioteca, un framework o cualquier otro grupo complejo de clases



Patrón Facade

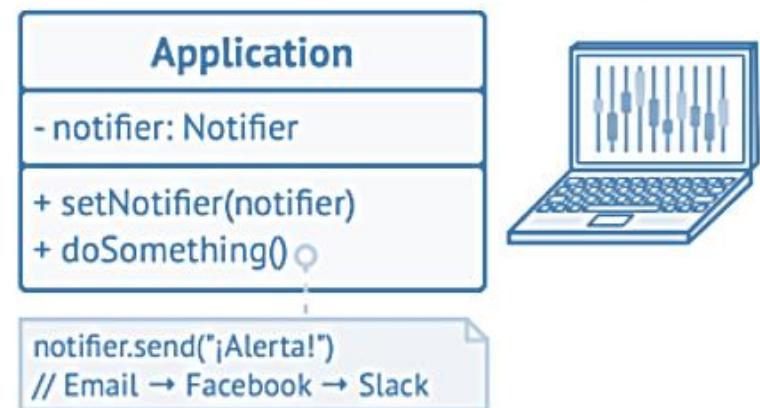
Problema:

- El objetivo del patrón Facade es agrupar las interfaces de un conjunto de objetos en una interfaz unificada volviendo a este conjunto más fácil de usar por parte de un cliente.
- El patrón Facade encapsula la interfaz de cada objeto considerada como interfaz de bajo nivel en una interfaz única de nivel más elevado. La construcción de la interfaz unificada puede necesitar implementar métodos destinados a componer las interfaces de bajo nivel.

Patrón Facade

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

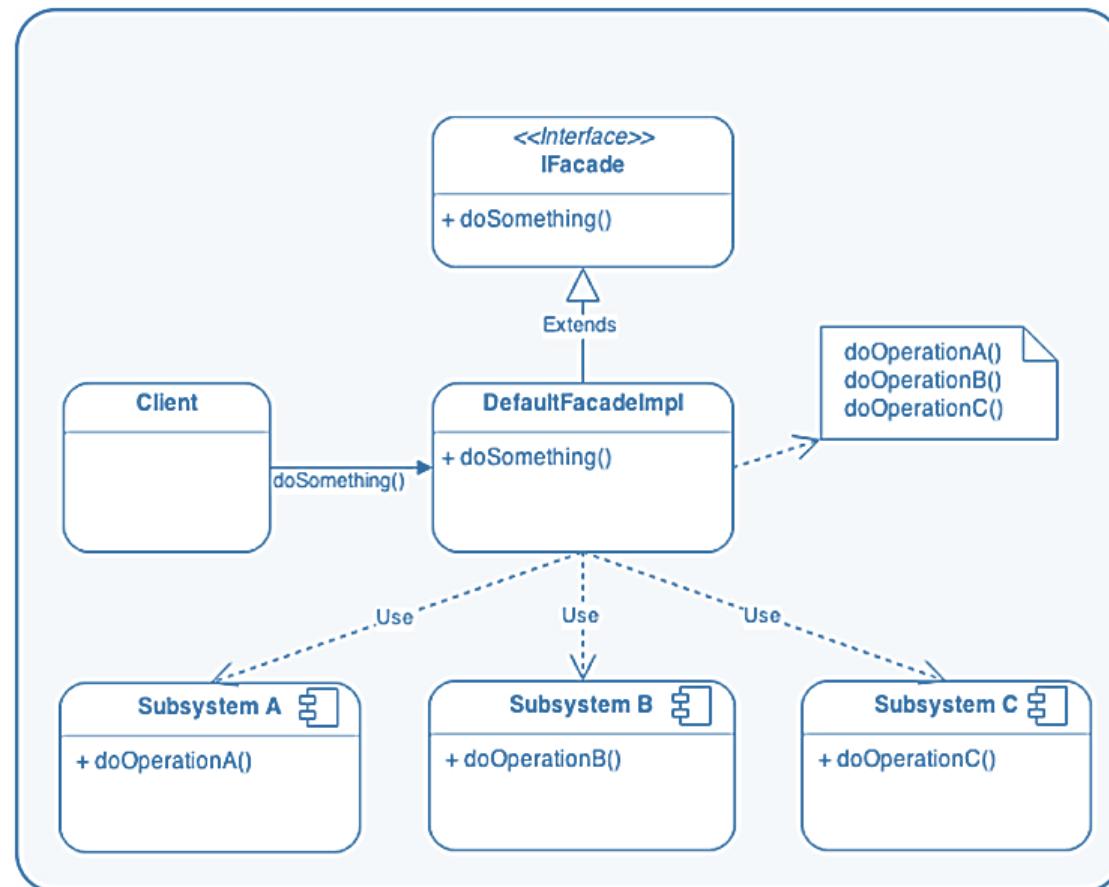
app.setNotifier(stack)
```



Las aplicaciones pueden configurar pilas complejas de decoradores de notificación.

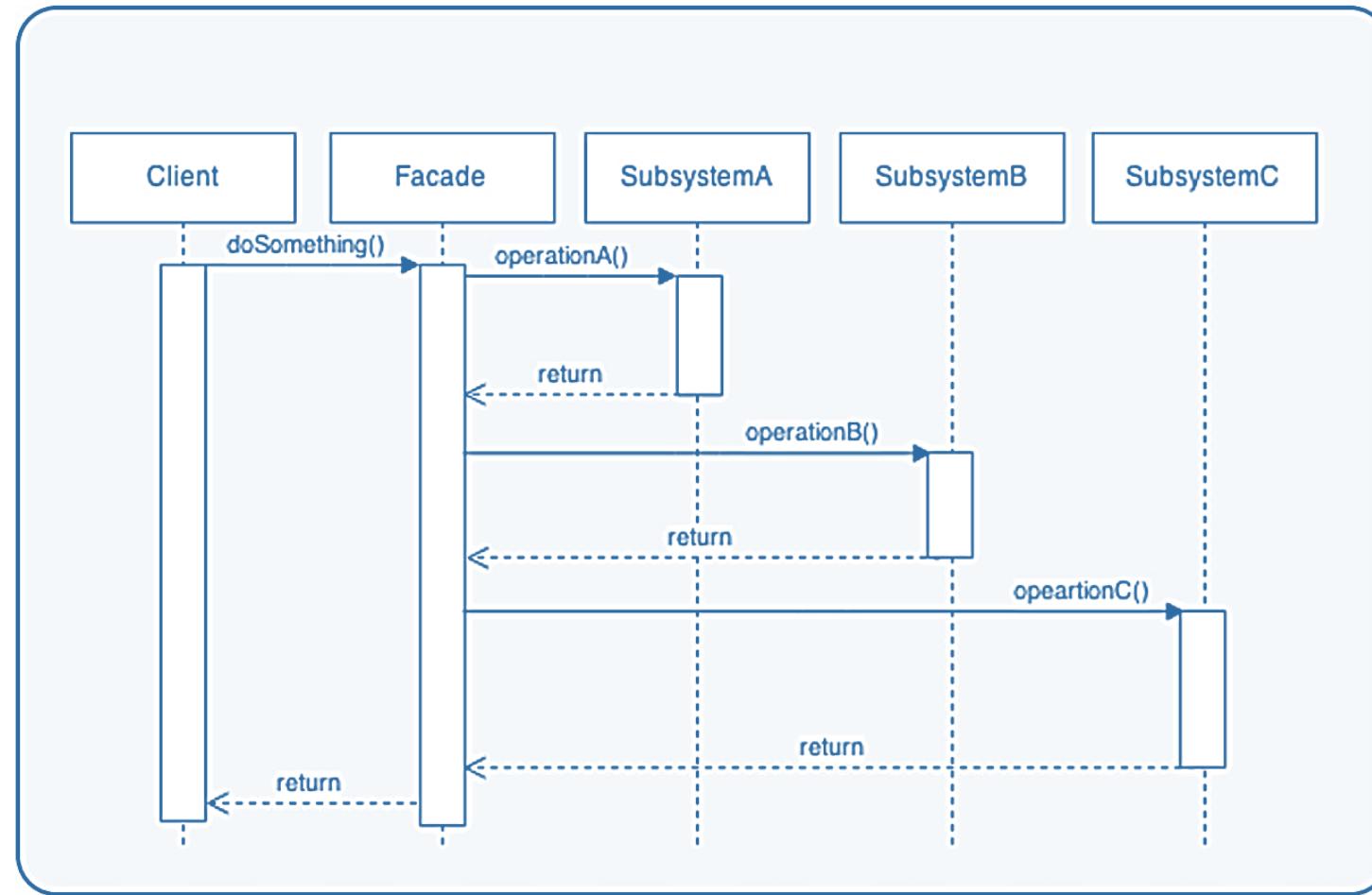
Patrón Facade

Estructura:



Patrón Facade

Colaboración:



Patrón Facade

Cuando Aplicar:

- Utiliza el patrón Facade cuando necesites una interfaz limitada pero directa a un subsistema complejo.
- Utiliza el patrón Facade cuando quieras estructurar un subsistema en capas.

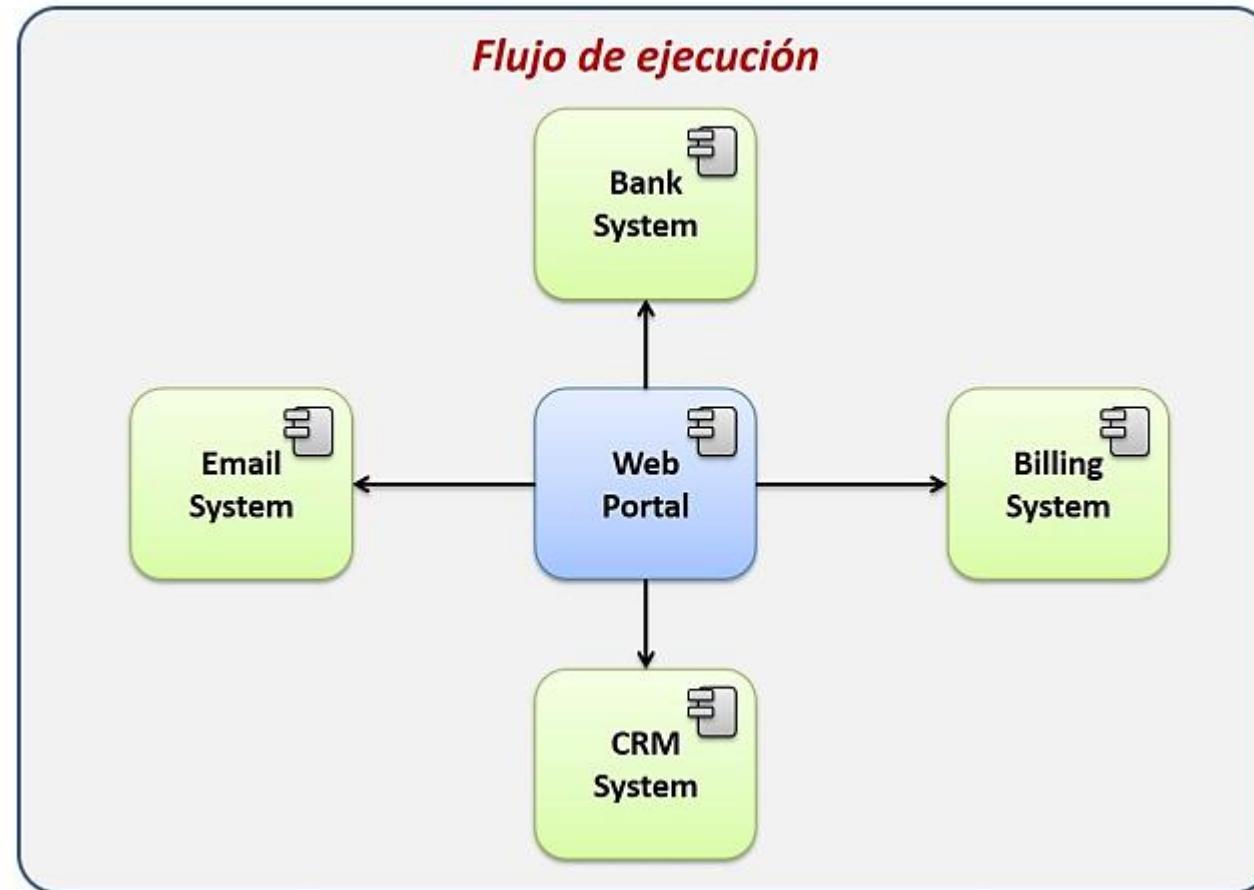
Patrón Facade

Cómo implementarlo:

1. Comprueba si es posible proporcionar una interfaz más simple que la que está proporcionando un subsistema existente.
2. Declara e implementa esta interfaz en una nueva clase fachada.
3. Para aprovechar el patrón al máximo, haz que todo el código cliente se comunique con el subsistema únicamente a través de la fachada.
4. Si la fachada se vuelve **demasiado grande**, piensa en extraer parte de su comportamiento y colocarlo dentro de una nueva clase fachada refinada.

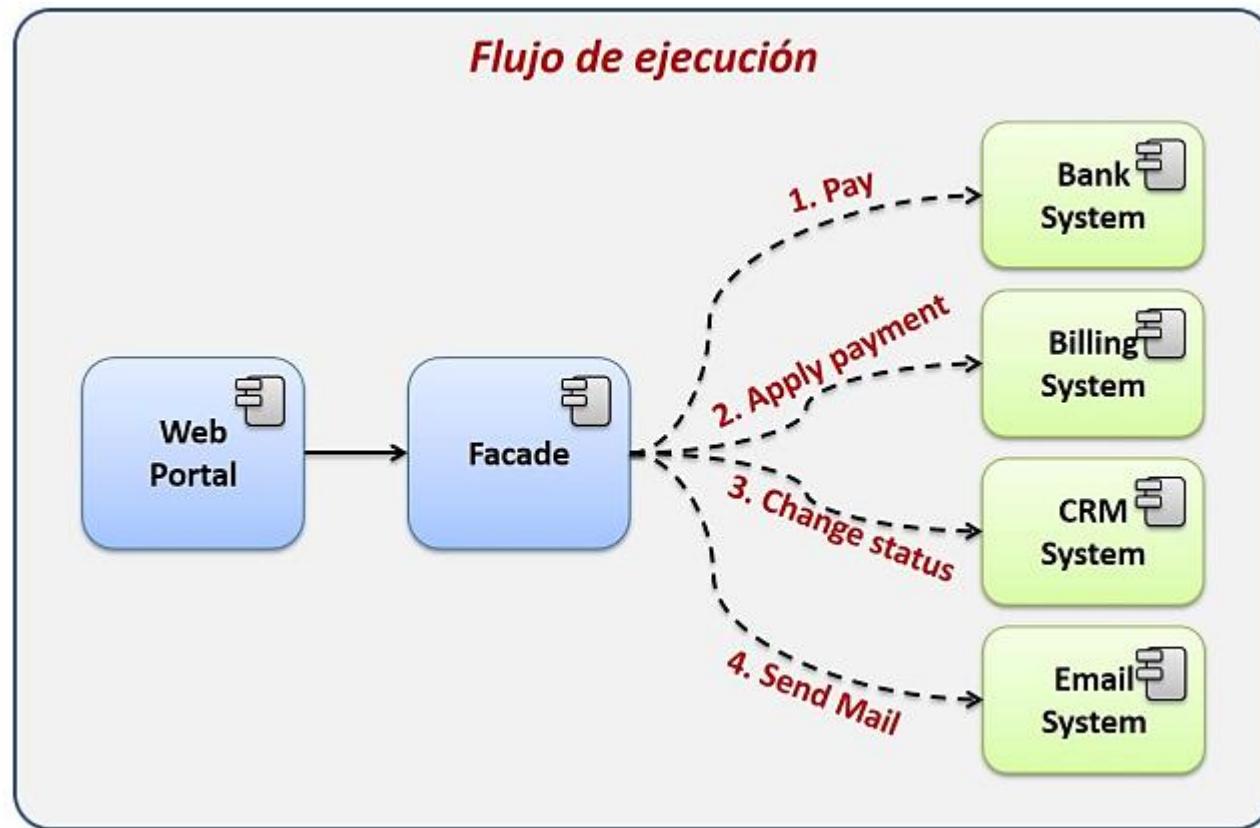
Patrón Facade

El escenario:

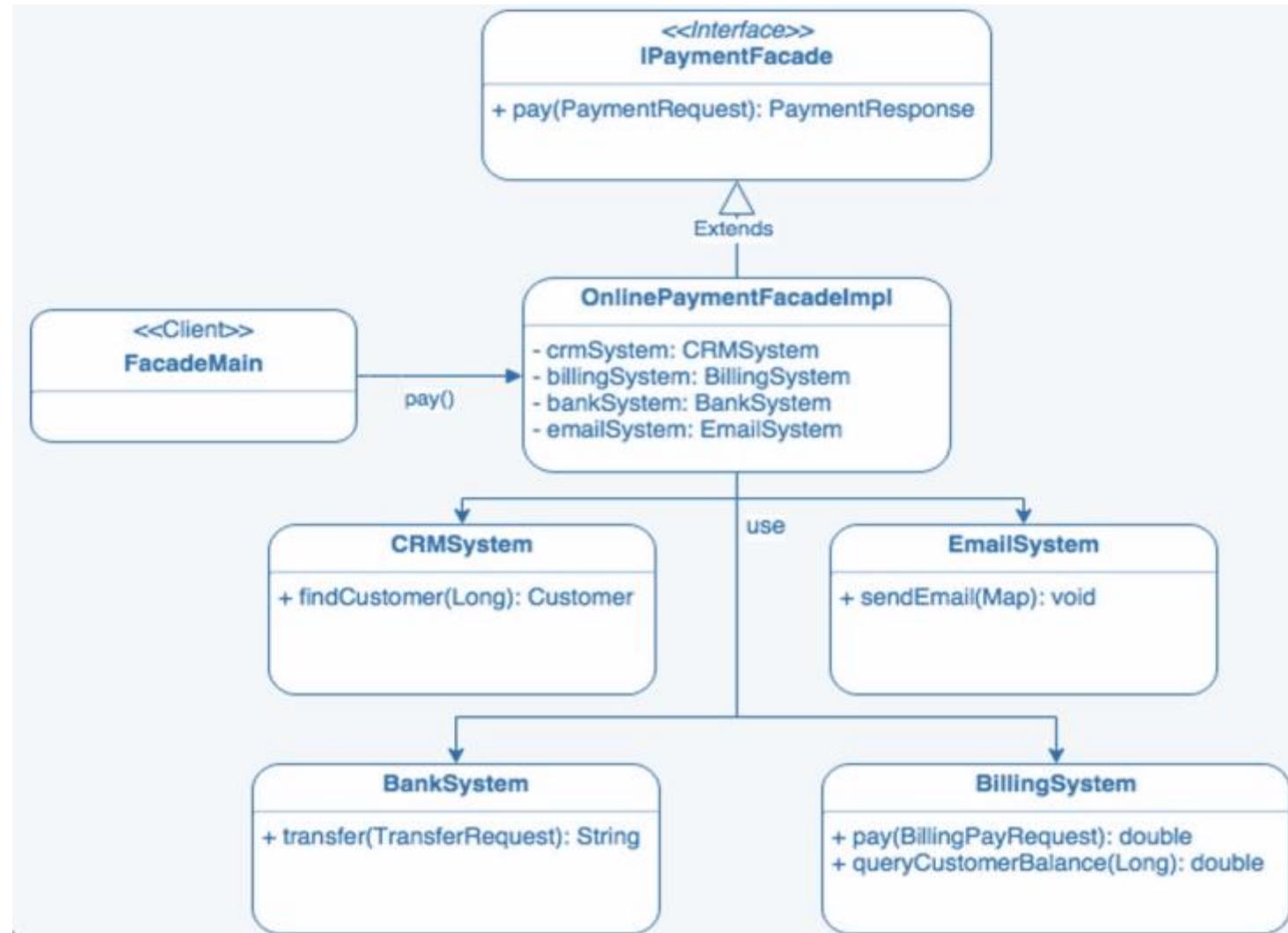


Patrón Facade

La solución:



Patrón Facade



Patrones de comportamiento

Los patrones de comportamiento tratan con algoritmos y la asignación de responsabilidades entre objetos.

Cadena de responsabilidad: *permite que un conjunto de clases intenten manejar un requerimiento.*

Interprete: *define una gramática de un lenguaje y usa esa gramática para interpretar sentencias del lenguaje.*

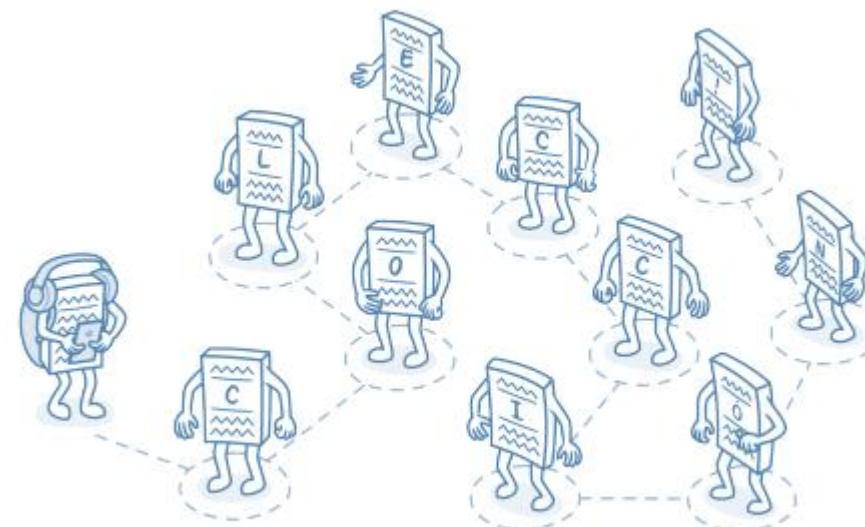
Iterator: *permite recorrer una estructura de datos sin conocer detalles de cómo están implementados los datos*

Observer: *algunos objetos reflejan un cambio a raíz del cambio de otro, por lo tanto se le debe comunicar el cambio de este último.*

Strategy: *cantidad de algoritmos relacionados encerrados en un contexto a través del cual se selecciona uno de los algoritmos.*

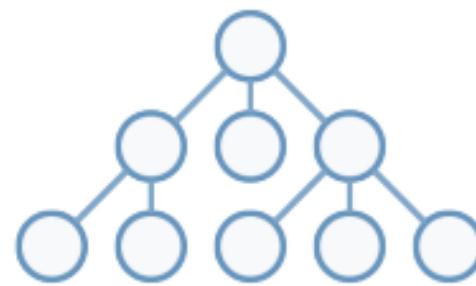
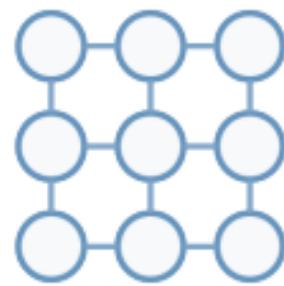
Patrón Iterator

Iterator es un patrón de diseño de comportamiento que te permite recorrer elementos de una colección sin exponer su representación subyacente (lista, pila, árbol, etc.)

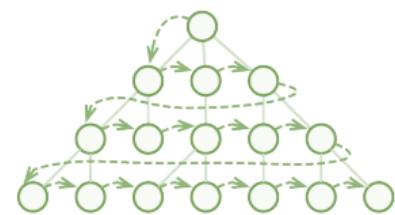
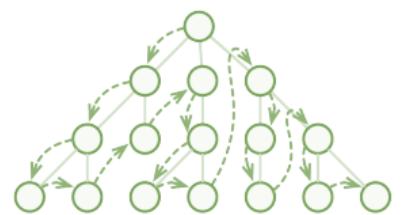


Patrón Iterator

Problema:



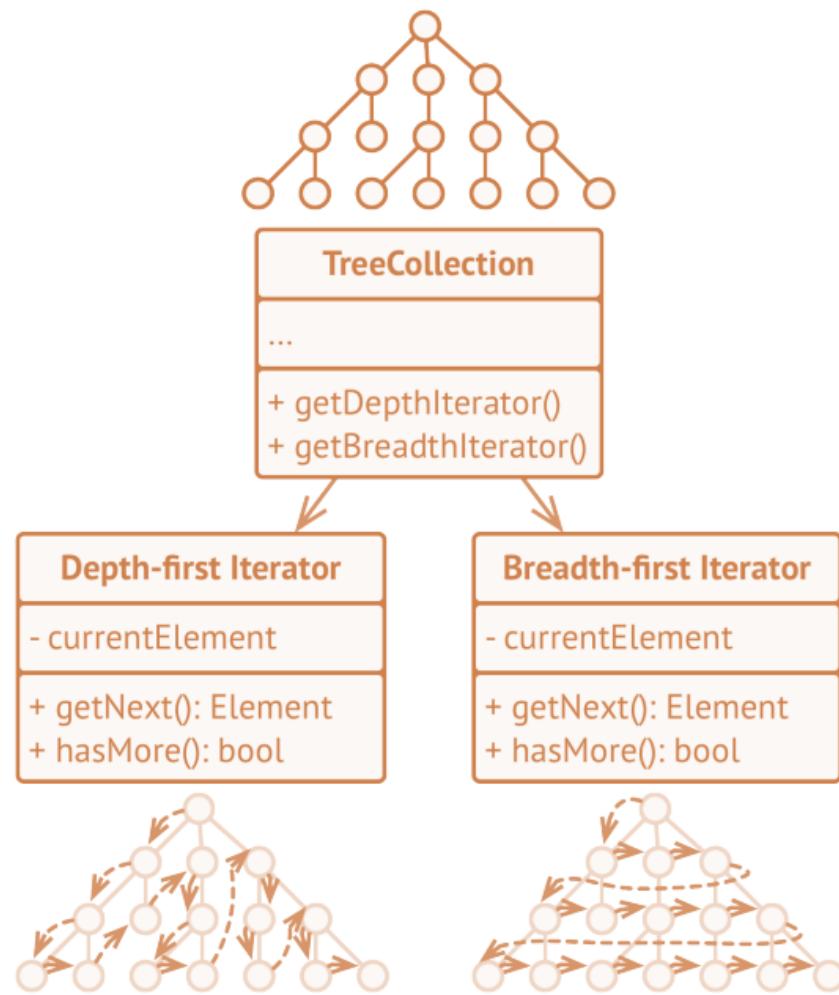
Varios tipos de colecciones.



La misma colección puede recorrerse de varias formas diferentes.

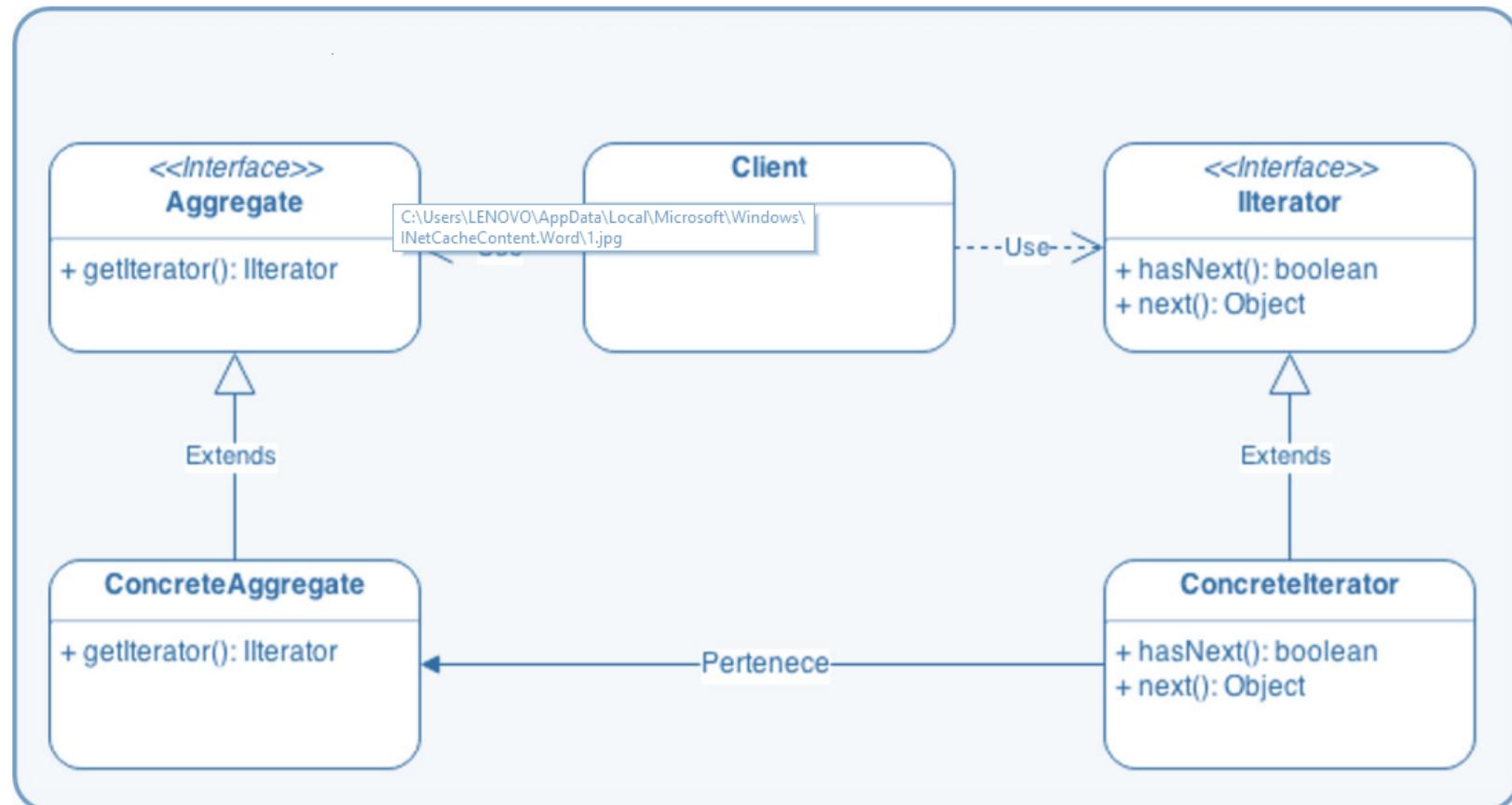
Patrón Iterator

Solución:



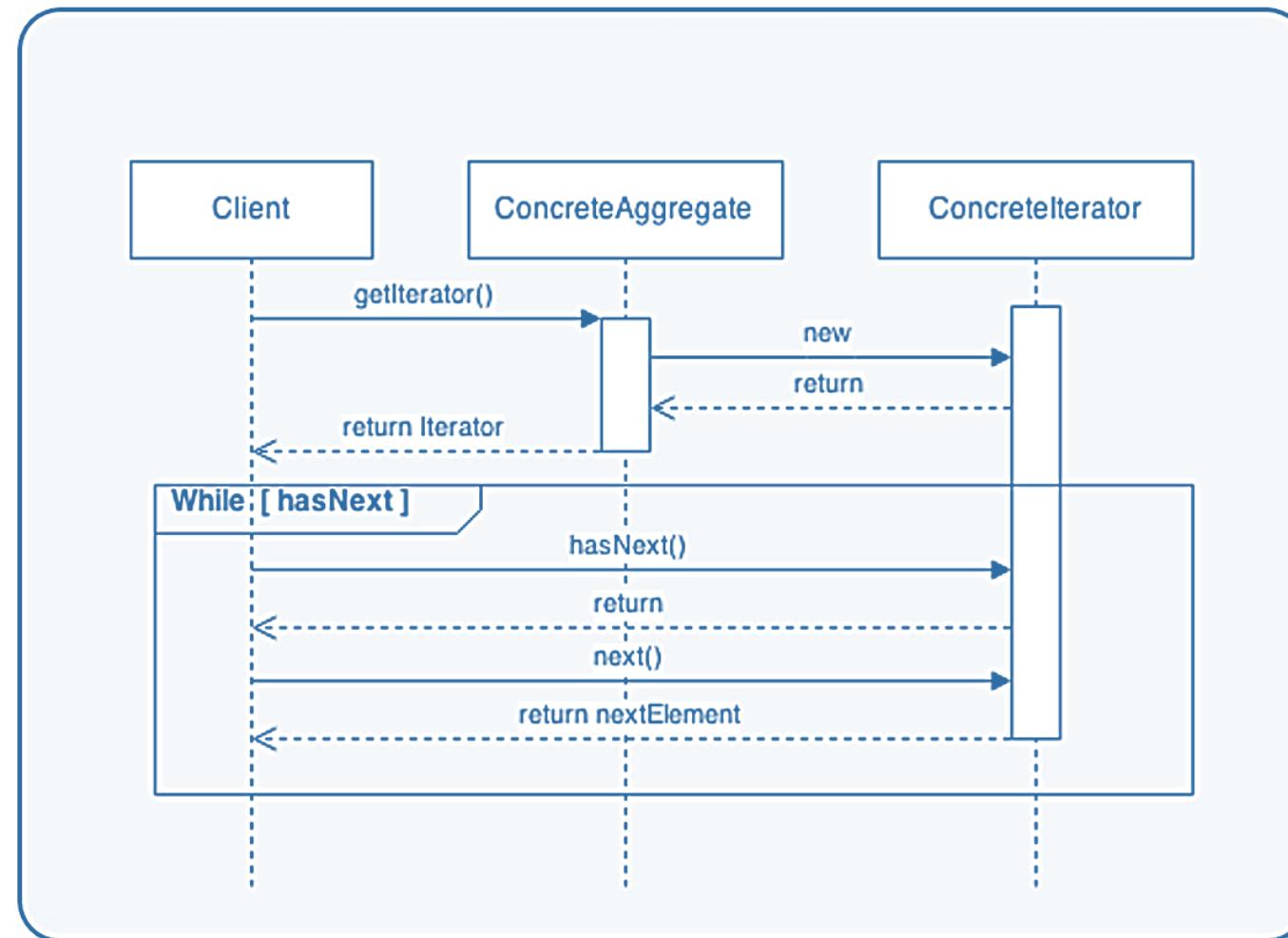
Patrón Iterator

Estructura:



Patrón Iterator

Colaboración:



Patrón Iterator

Cuando Aplicar:

- Utiliza el patrón Iterator cuando tu colección tenga una estructura de datos compleja a nivel interno, pero quieras ocultar su complejidad a los clientes (ya sea por conveniencia o por razones de seguridad).
- Utiliza el patrón para reducir la duplicación en el código de recorrido a lo largo de tu aplicación.
- Utiliza el patrón Iterator cuando quieras que tu código pueda recorrer distintas estructuras de datos, o cuando los tipos de estas estructuras no se conozcan de antemano.

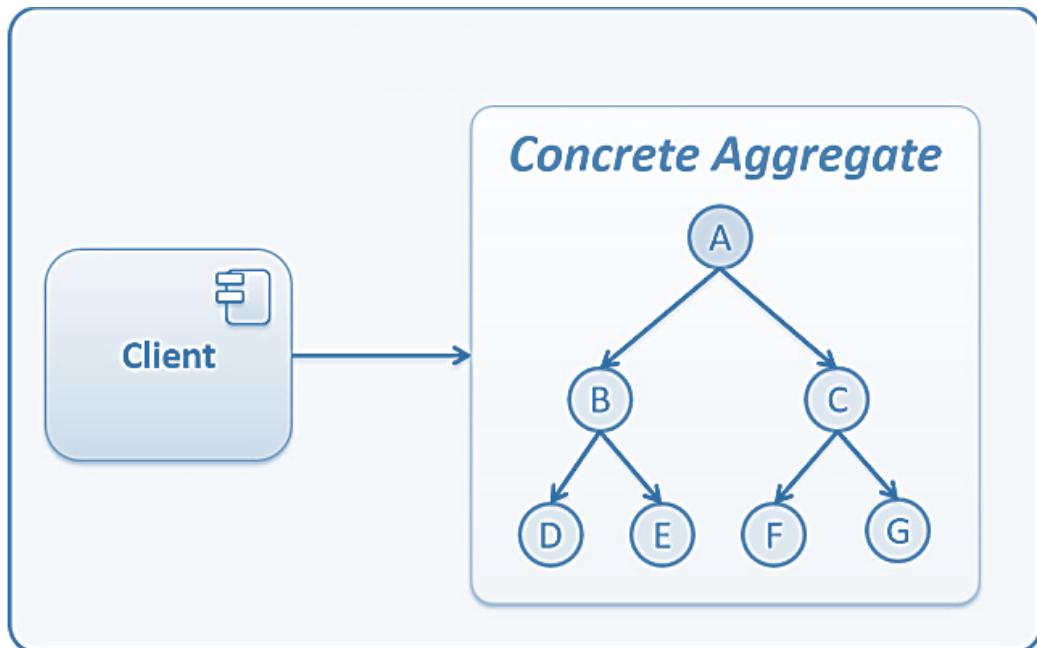
Patrón Iterator

Cómo implementarlo:

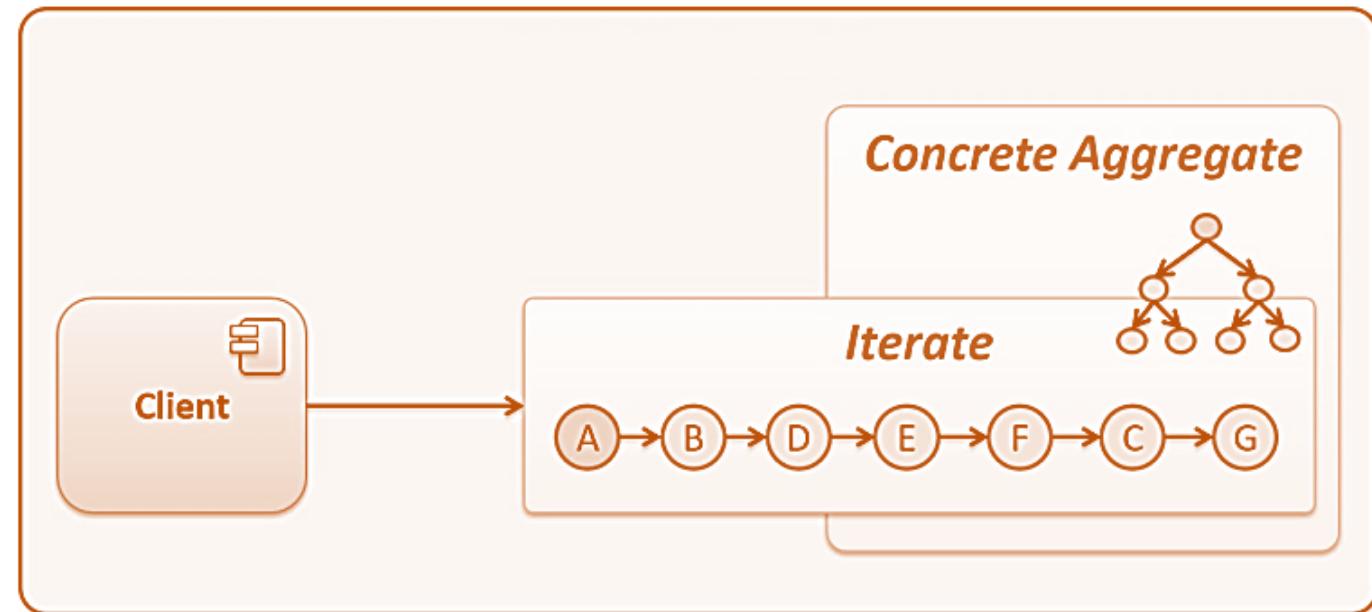
1. Declara la interfaz iteradora. Como mínimo, debe tener un método para extraer el siguiente elemento de una colección.
2. Declara la interfaz de colección y describe un método para buscar iteradores.
3. Implementa clases iteradoras concretas para las colecciones que quieras que sean recorridas por iteradores.
4. Implementa la interfaz de colección en tus clases de colección.
5. Repasa el código cliente para sustituir todo el código de recorrido de la colección por el uso de iteradores.

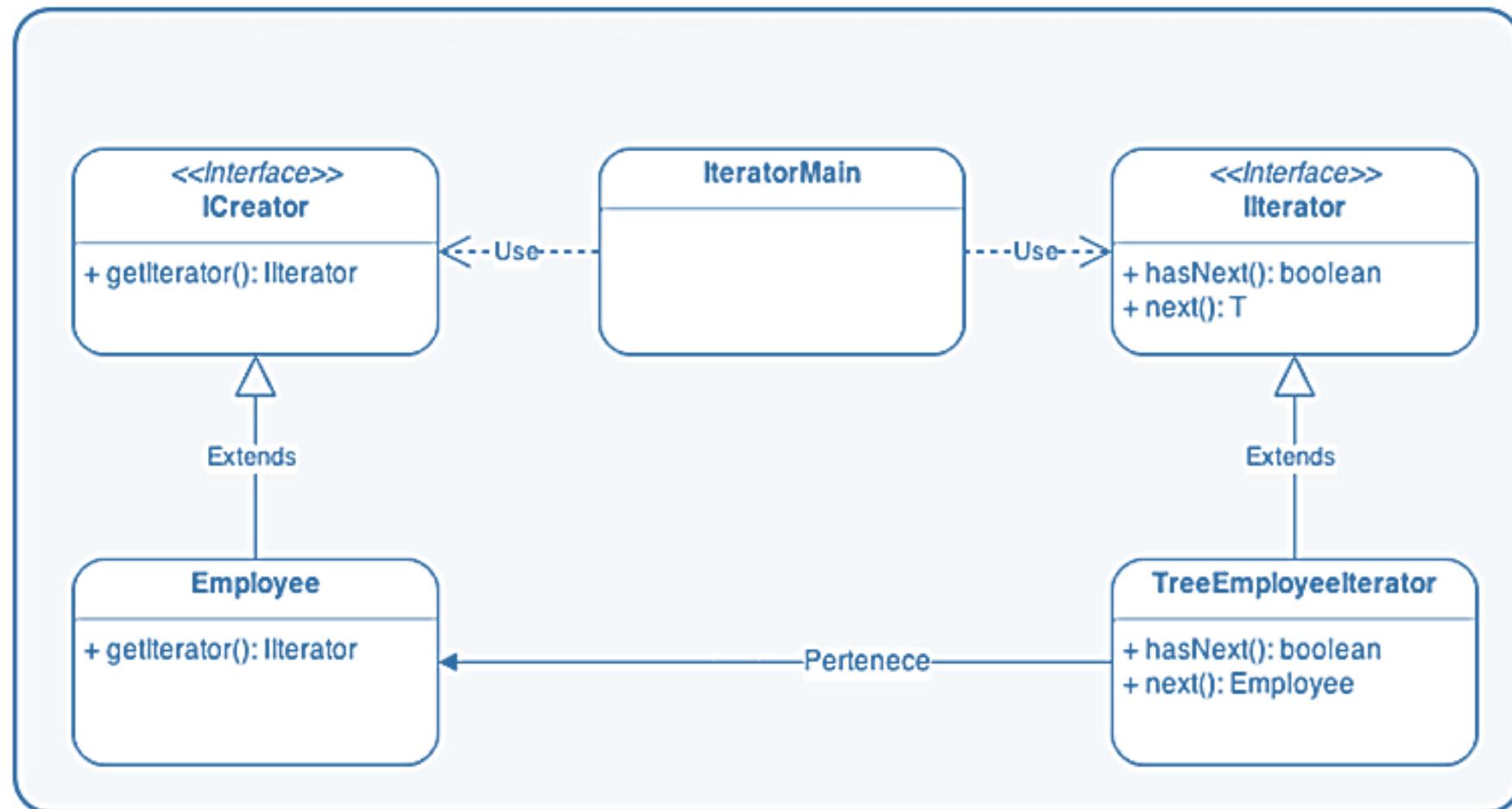
Patrón Iterator

El escenario:



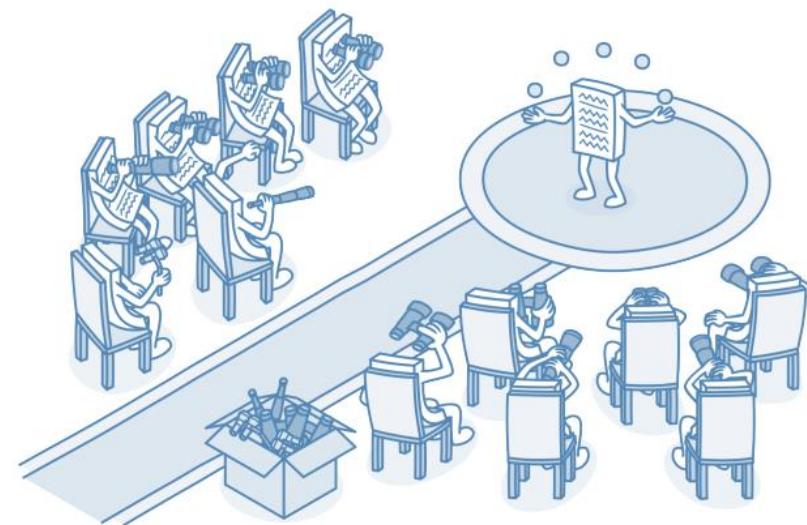
La solución:





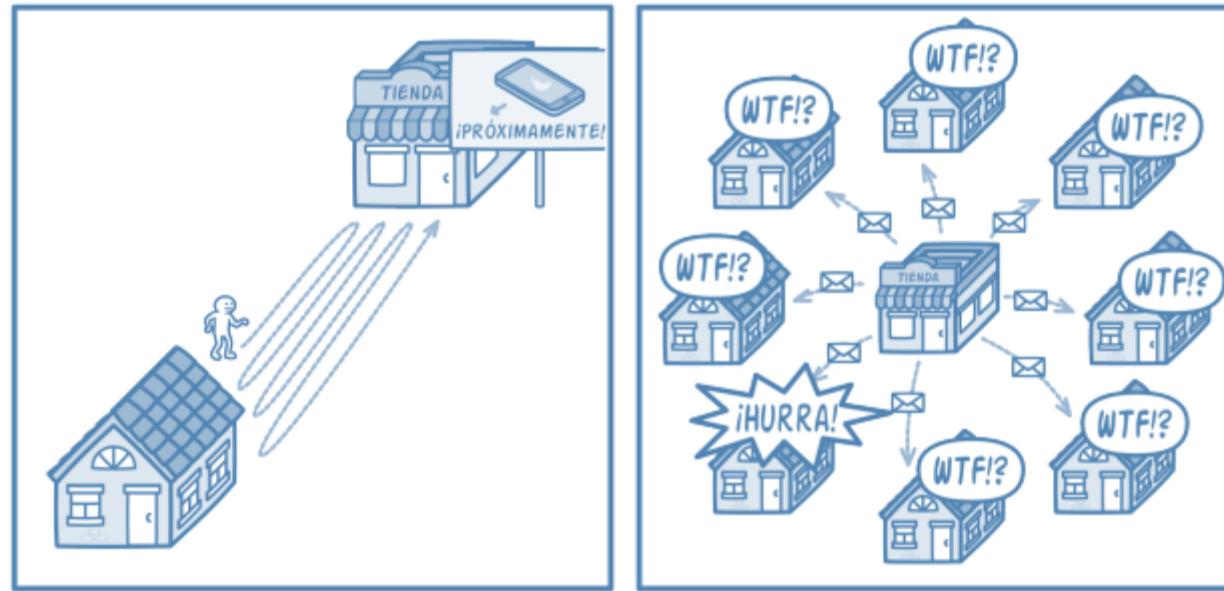
Patrón Observer

Observer es un patrón de diseño de comportamiento que te permite definir un mecanismo de suscripción para notificar a varios objetos sobre cualquier evento que le suceda al objeto que están observando.



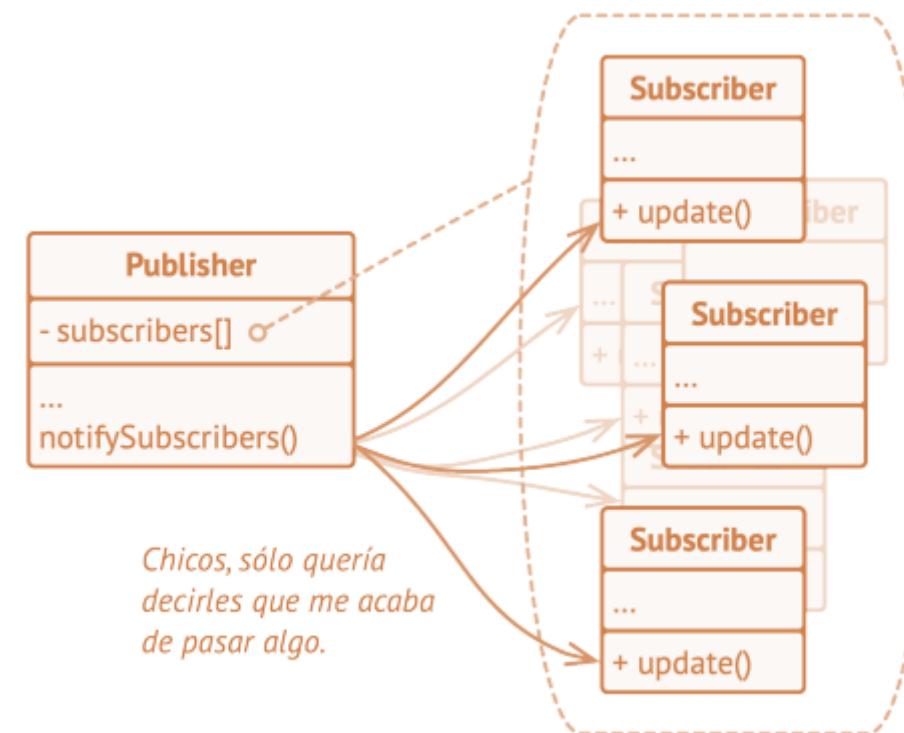
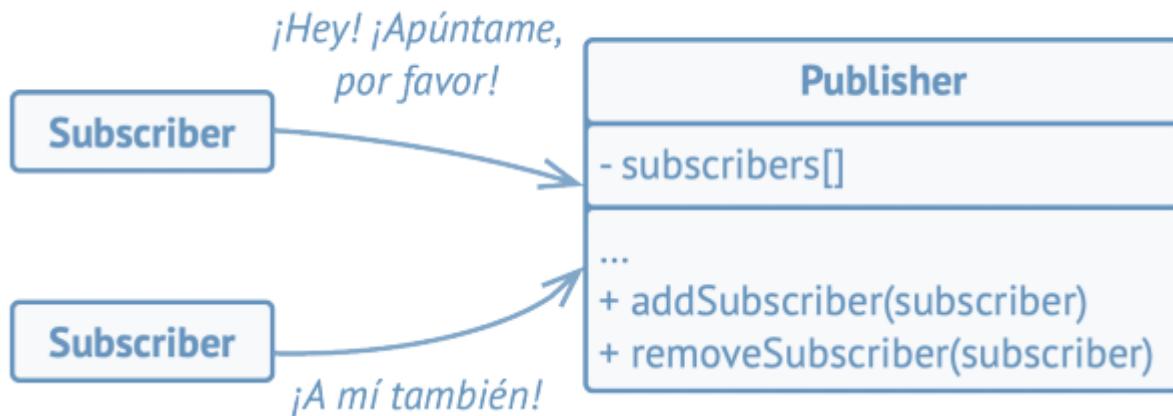
Patrón Observer

Problema:



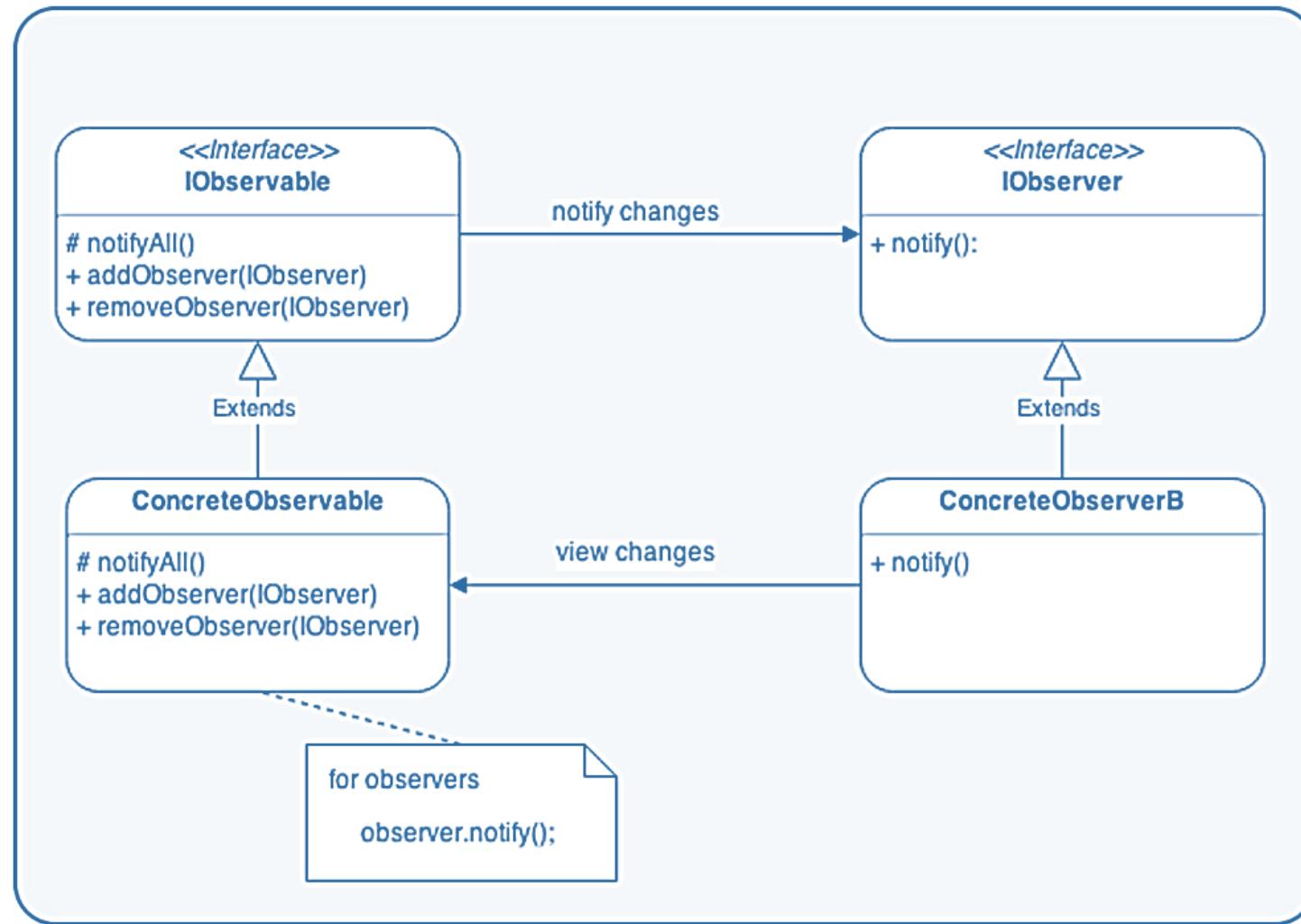
Patrón Observer

Solución:



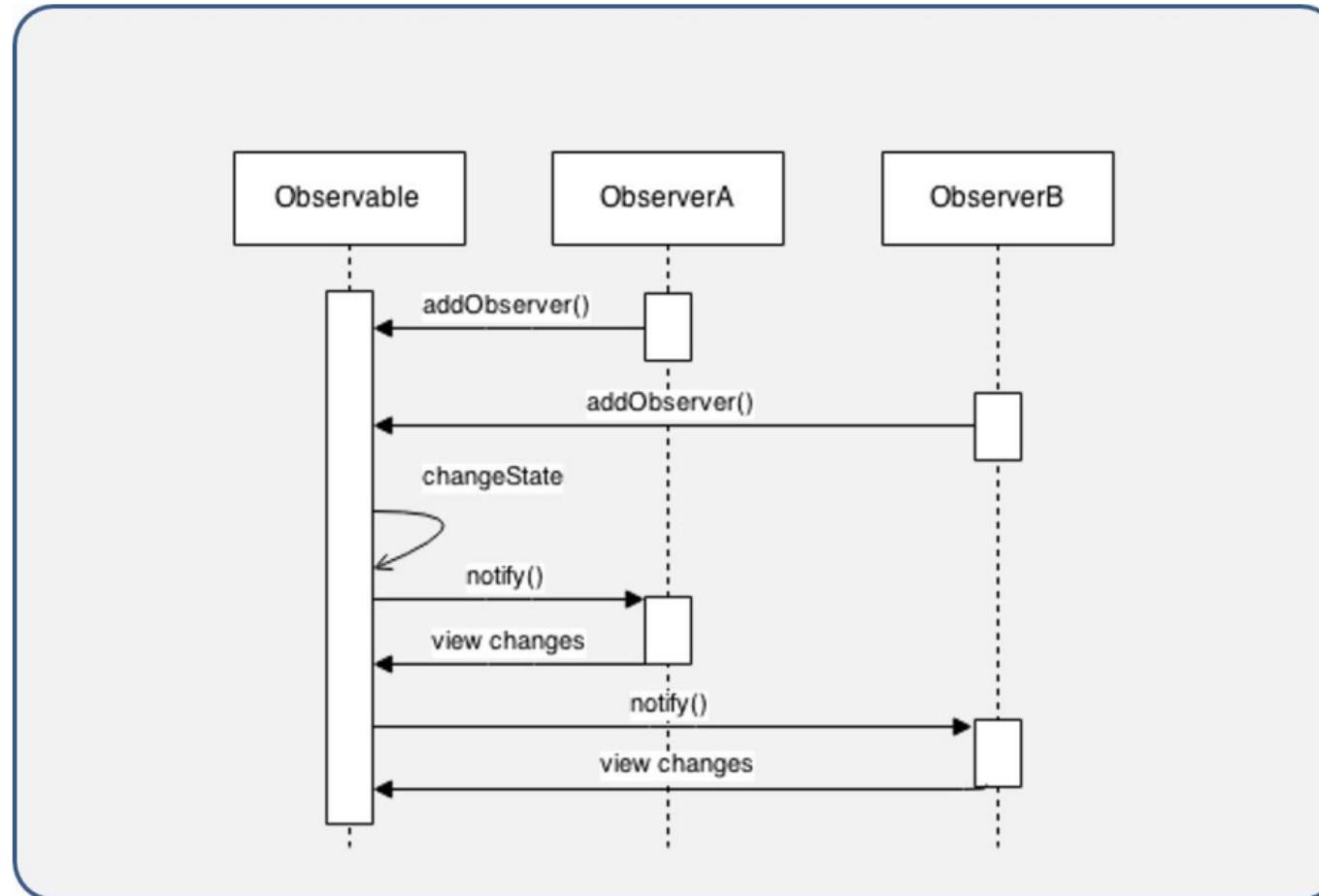
Patrón Observer

Estructura:



Patrón Observer

Estructura:



Patrón Observer

Cuando Aplicar:

- Utiliza el patrón Observer cuando los cambios en el estado de un objeto puedan necesitar cambiar otros objetos y el grupo de objetos sea desconocido de antemano o cambie dinámicamente.
- Utiliza el patrón cuando algunos objetos de tu aplicación deban observar a otros, pero sólo durante un tiempo limitado o en casos específicos.

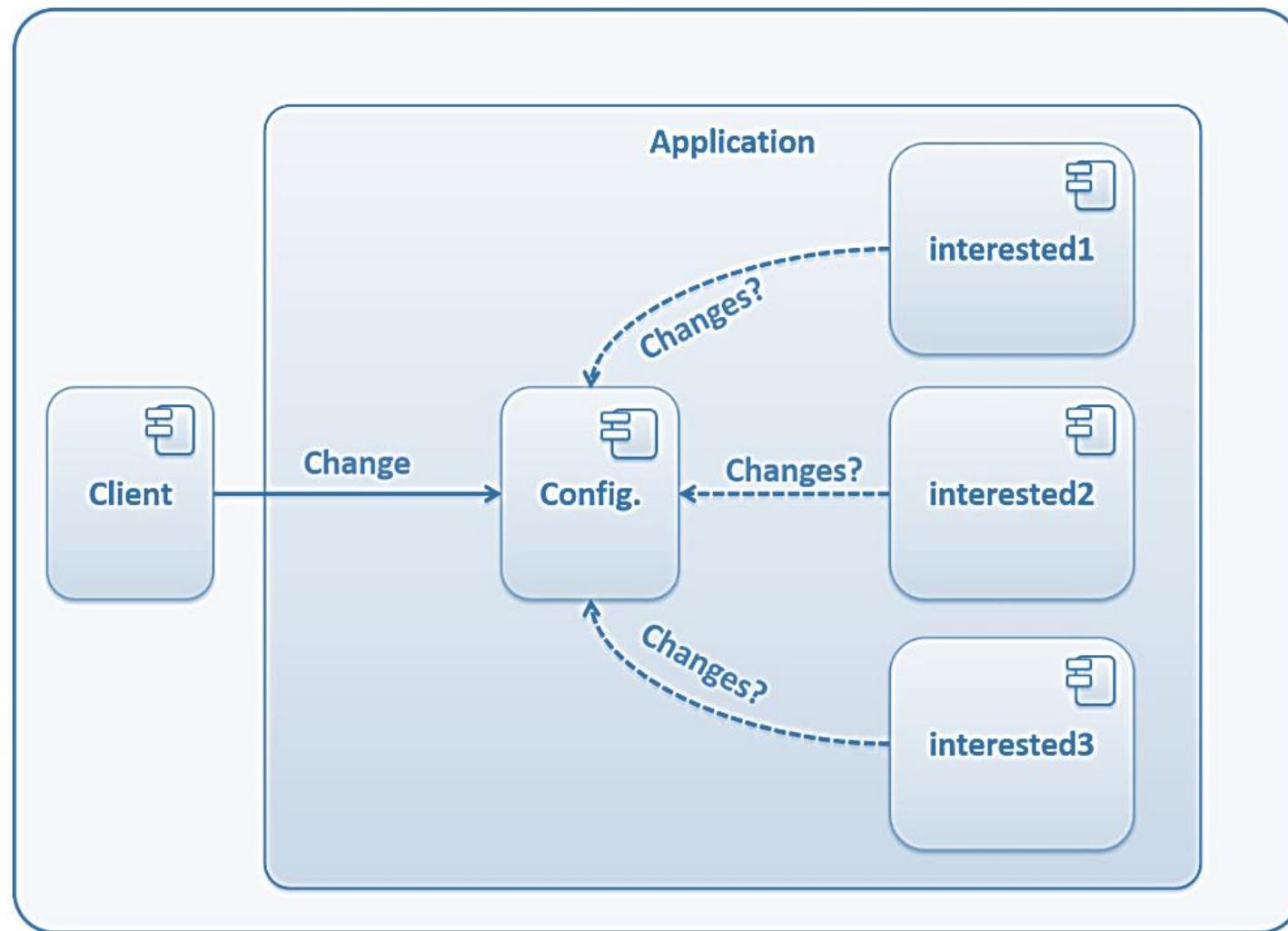
Patrón Observer

Cómo implementarlo:

1. Repasa tu lógica de negocio e intenta dividirla en dos partes .
2. Declara la interfaz suscriptora.
3. Declara la interfaz notificadora y describe un par de métodos
4. Decide dónde colocar la lista de suscripción y la implementación de métodos de suscripción.
5. Crea clases notificadoras concretas
6. Implementa los métodos de notificación de actualizaciones en clases suscriptoras concretas.
7. El cliente debe crear todos los suscriptores necesarios y registrarlos con los notificadores adecuados.

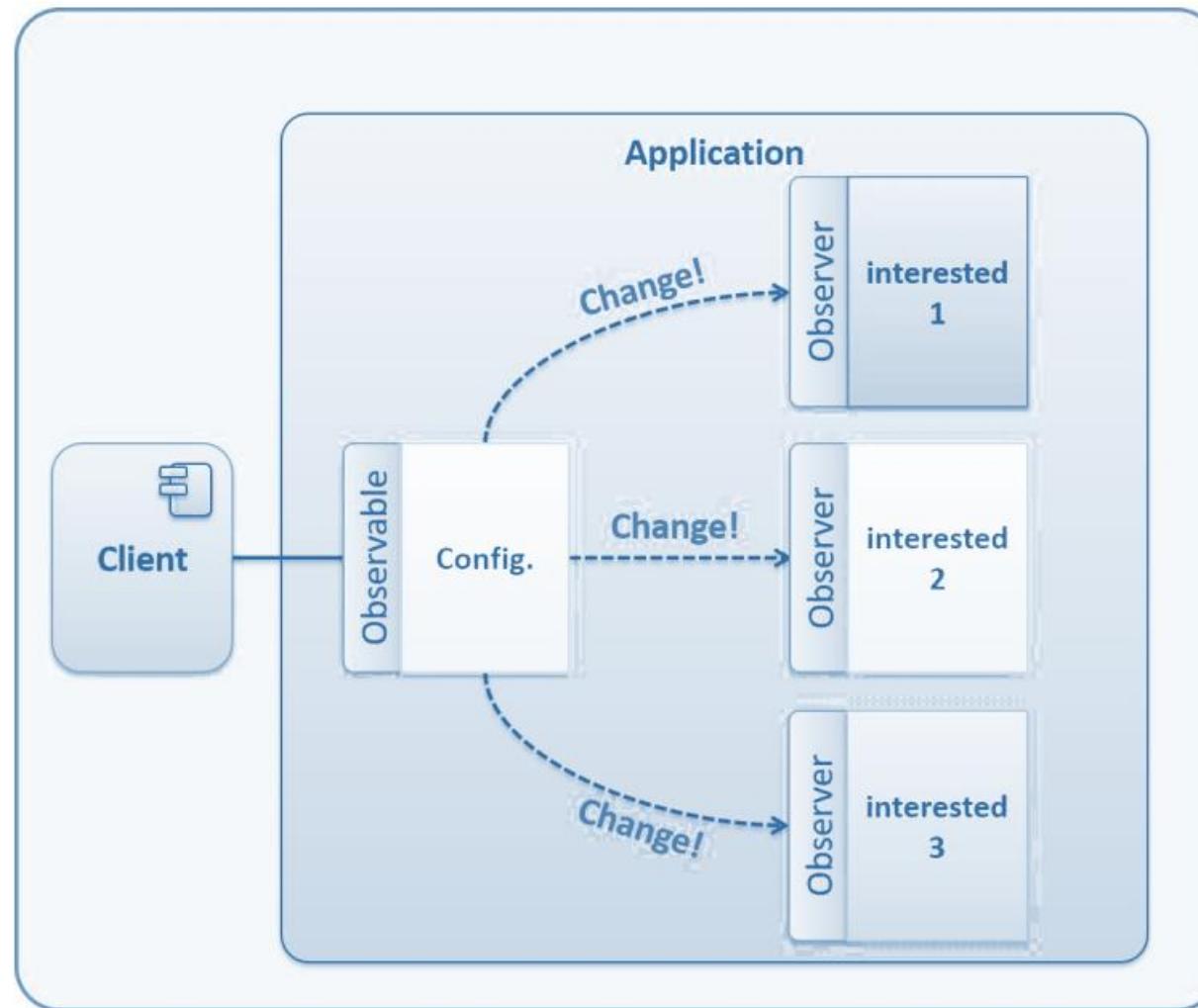
Patrón Observer

El escenario:

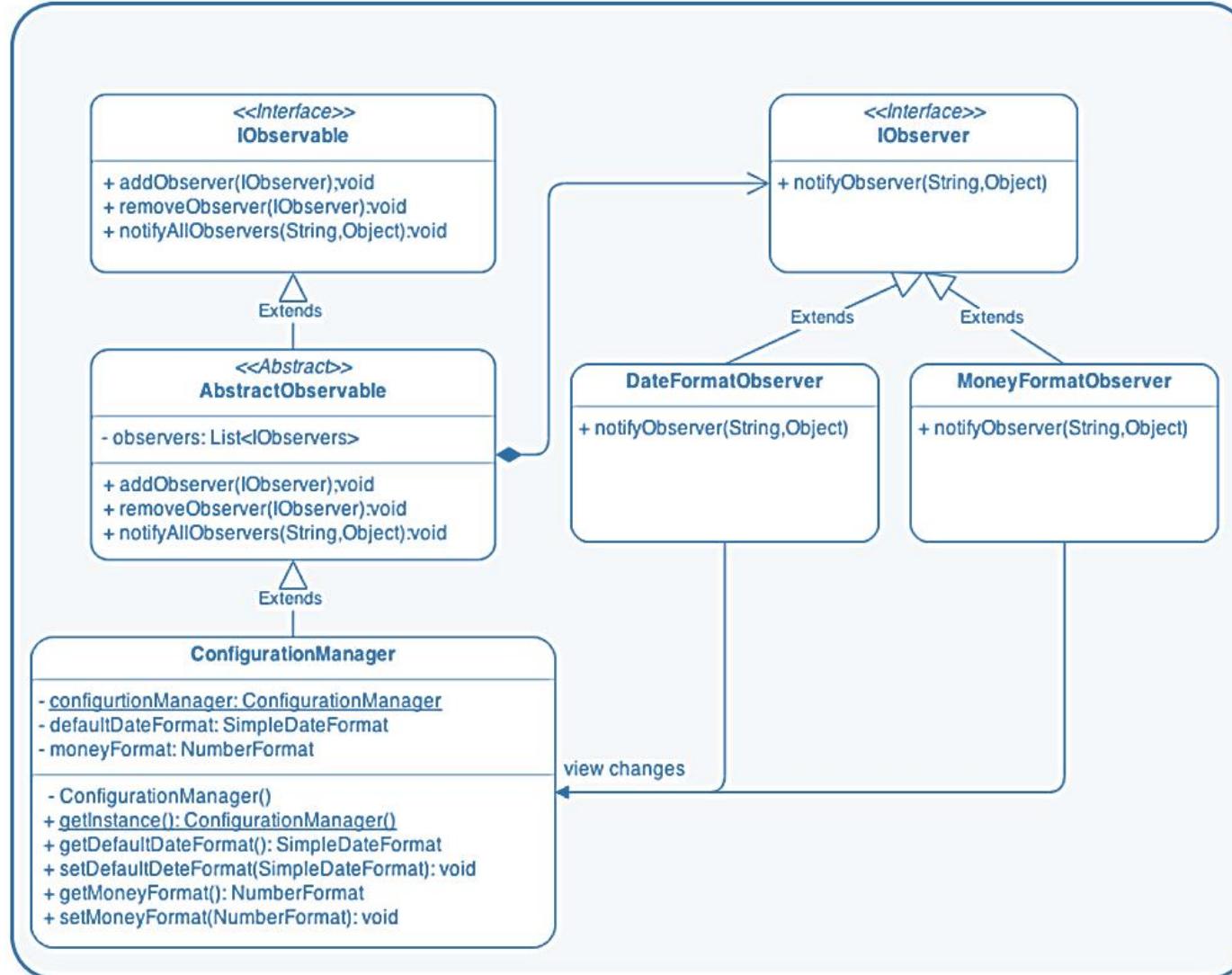


Patrón Observer

La solución:



Patrón Observer



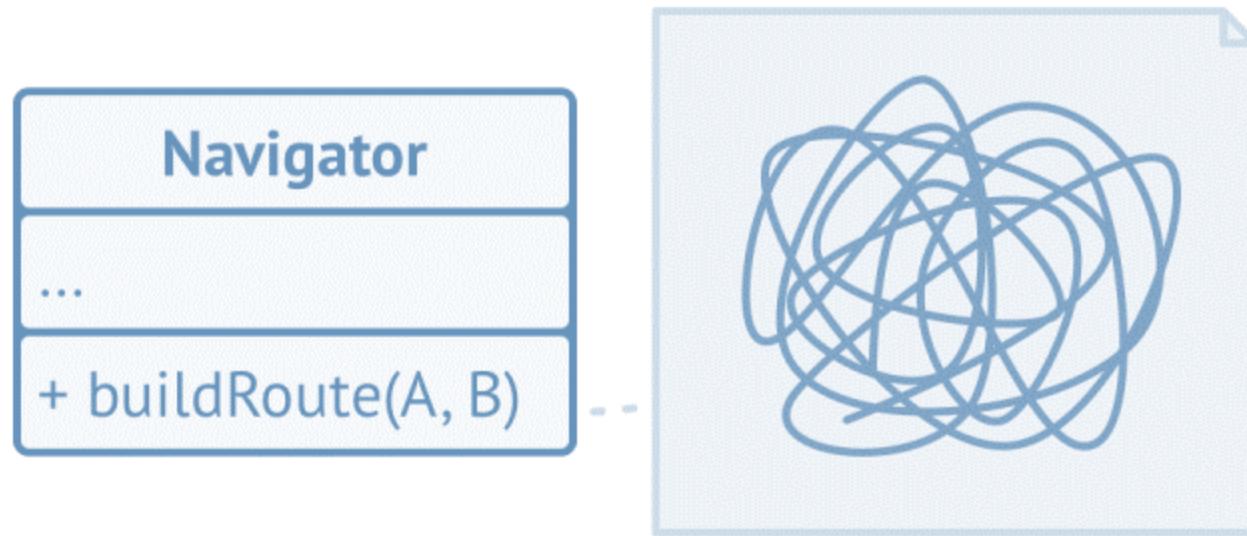
Patrón Strategy

Strategy es un patrón de diseño de comportamiento que te permite definir una familia de algoritmos, colocar cada uno de ellos en una clase separada y hacer sus objetos intercambiables.



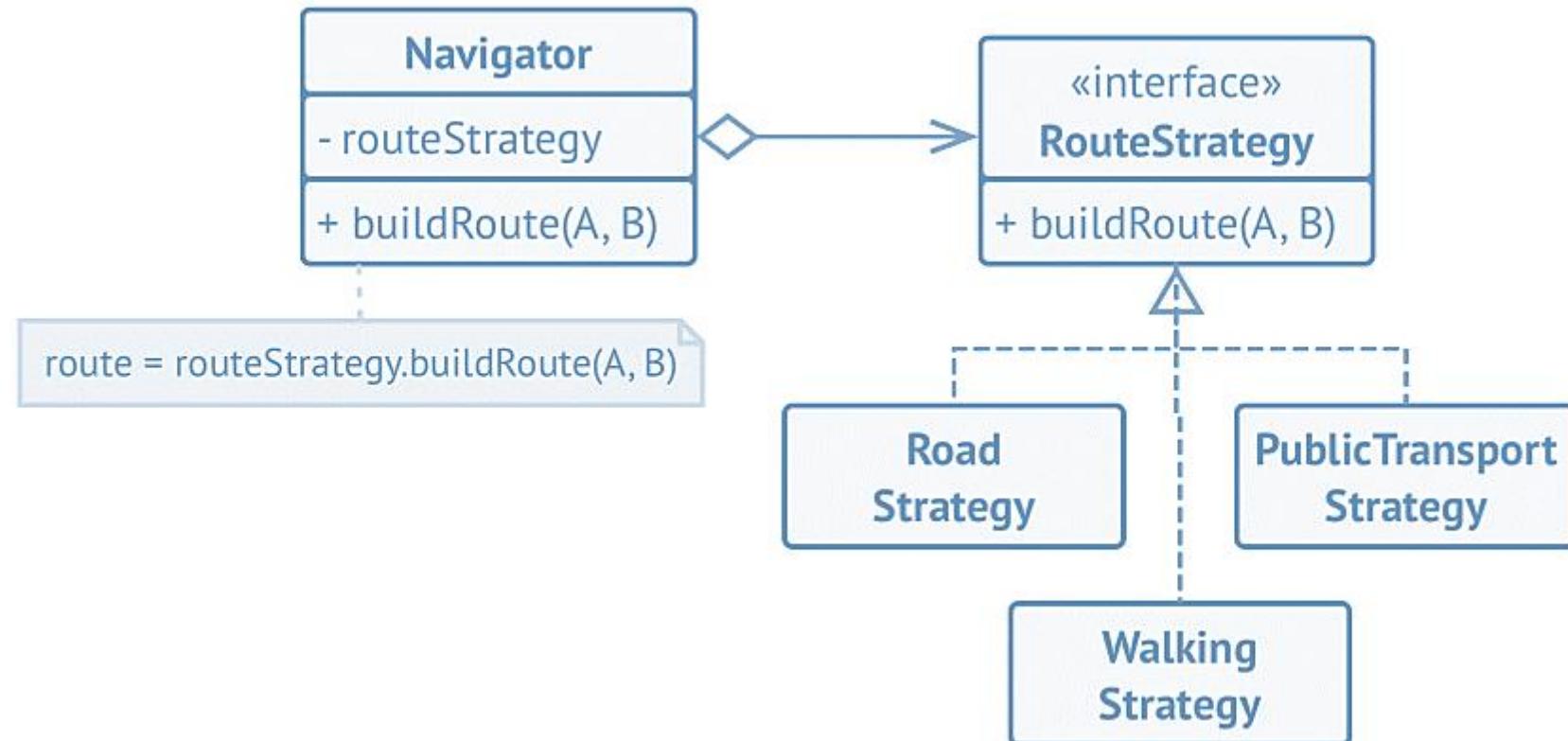
Patrón Strategy

Problema:



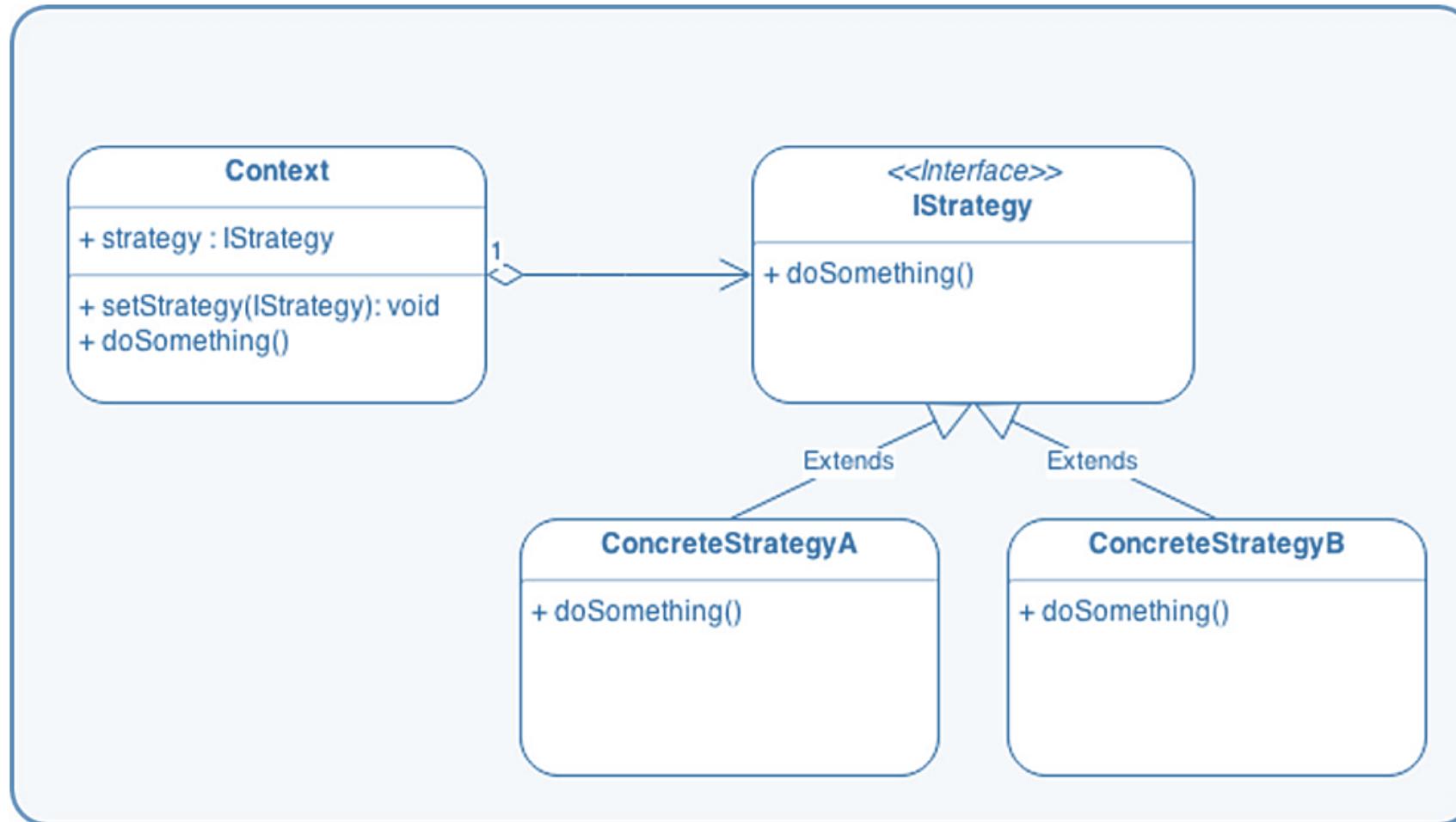
Patrón Strategy

Solución



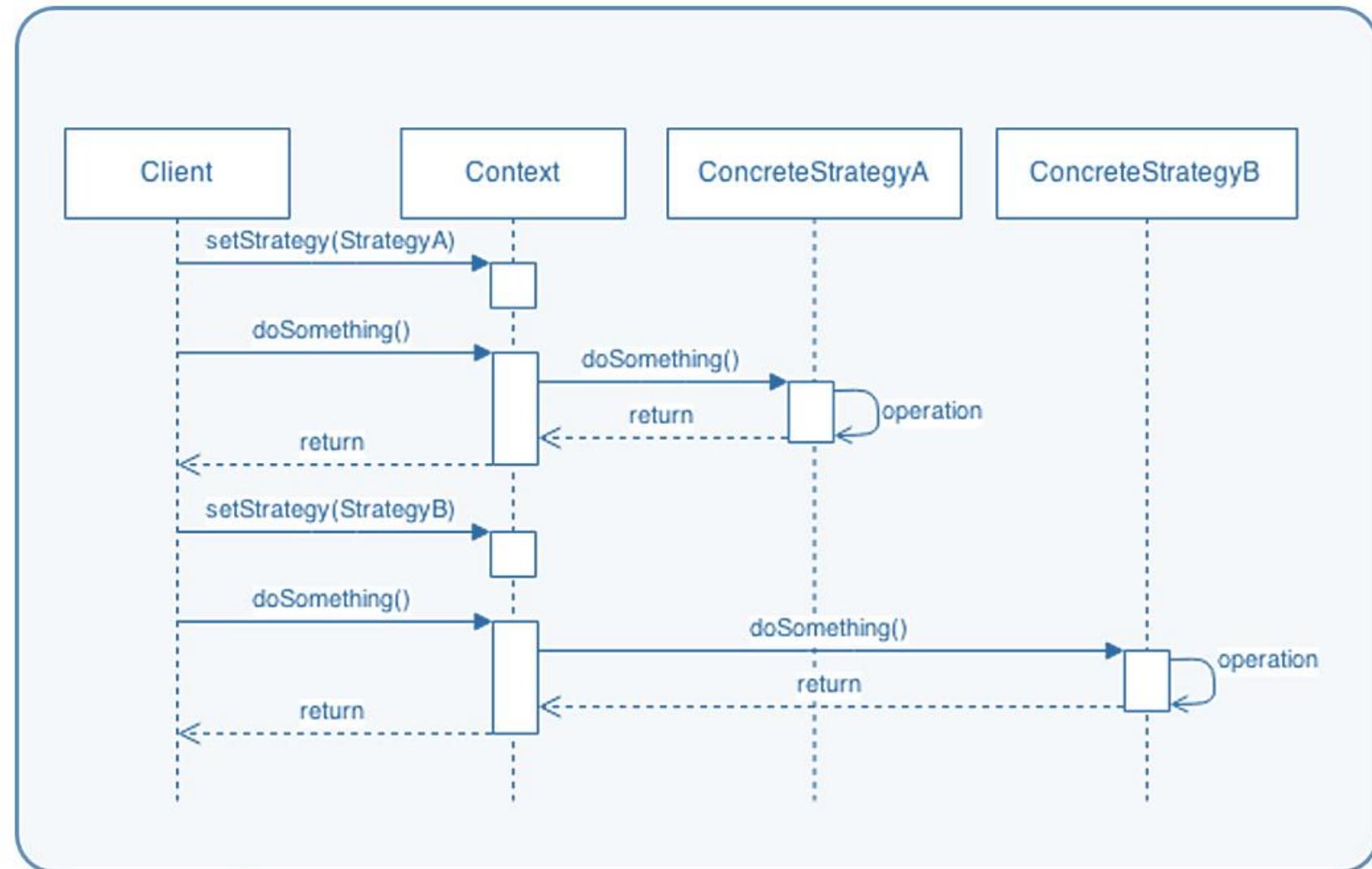
Patrón Strategy

Estructura:



Patrón Strategy

Colaboración:



Patrón Strategy

Cuando Aplicar:

- Utiliza el patrón Strategy cuando quieras utilizar distintas variantes de un algoritmo dentro de un objeto y poder cambiar de un algoritmo a otro durante el tiempo de ejecución.
- Utiliza el patrón Strategy cuando tengas muchas clases similares que sólo se diferencien en la forma en que ejecutan cierto comportamiento.
- Utiliza el patrón para aislar la lógica de negocio de una clase, de los detalles de implementación de algoritmos que pueden no ser tan importantes en el contexto de esa lógica.
- Utiliza el patrón cuando tu clase tenga un enorme operador condicional que cambie entre distintas variantes del mismo algoritmo.

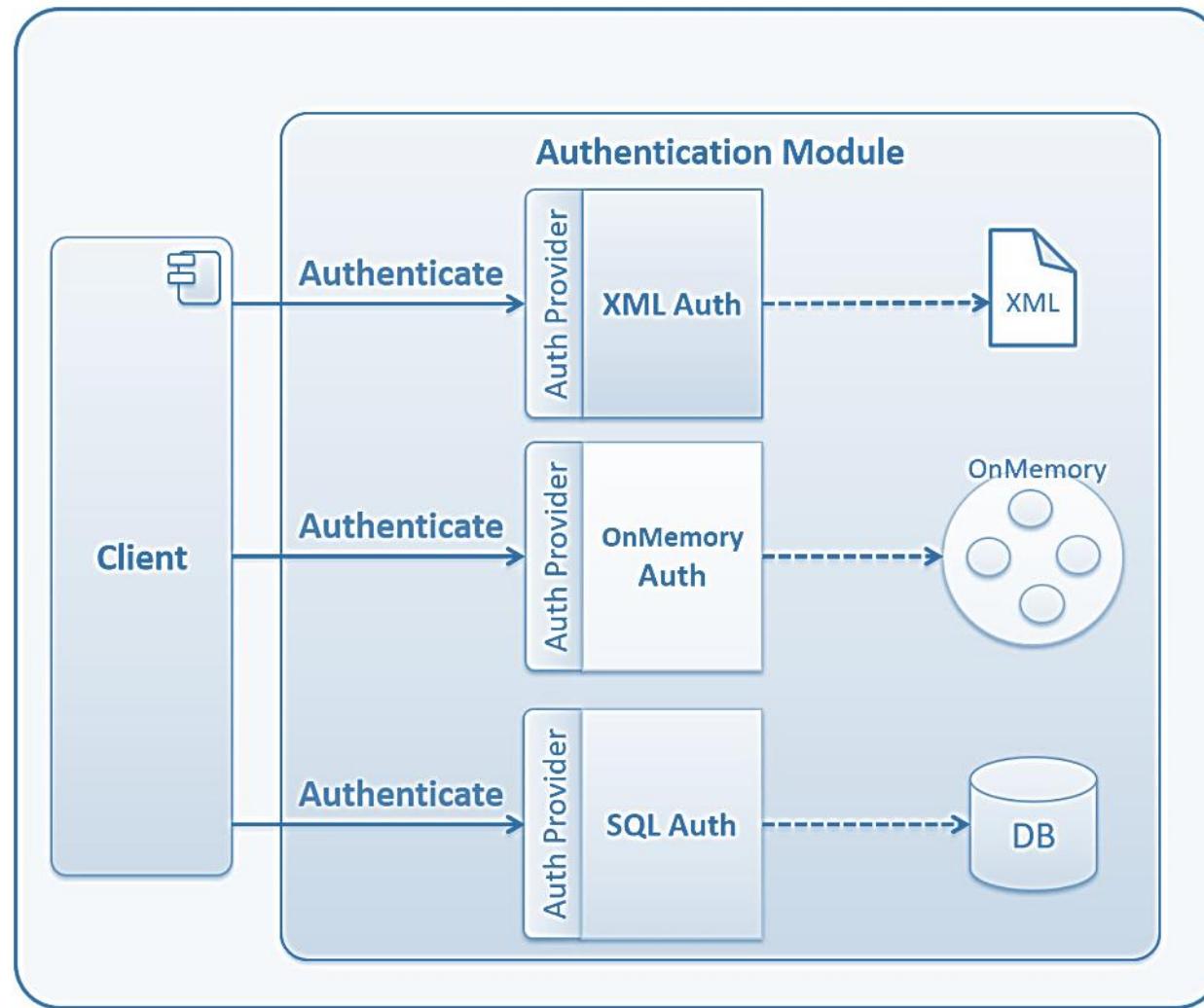
Patrón Strategy

Cómo implementarlo:

1. En la clase contexto, identifica un algoritmo que tienda a sufrir cambios frecuentes.
2. Declara la interfaz estrategia común a todas las variantes del algoritmo.
3. Uno a uno, extrae todos los algoritmos y ponlos en sus propias clases.
4. En la clase contexto, añade un campo para almacenar una referencia a un objeto de estrategia.
5. Los clientes de la clase contexto deben asociarla con una estrategia adecuada que coincida con la forma en la que esperan que la clase contexto realice su trabajo principal.

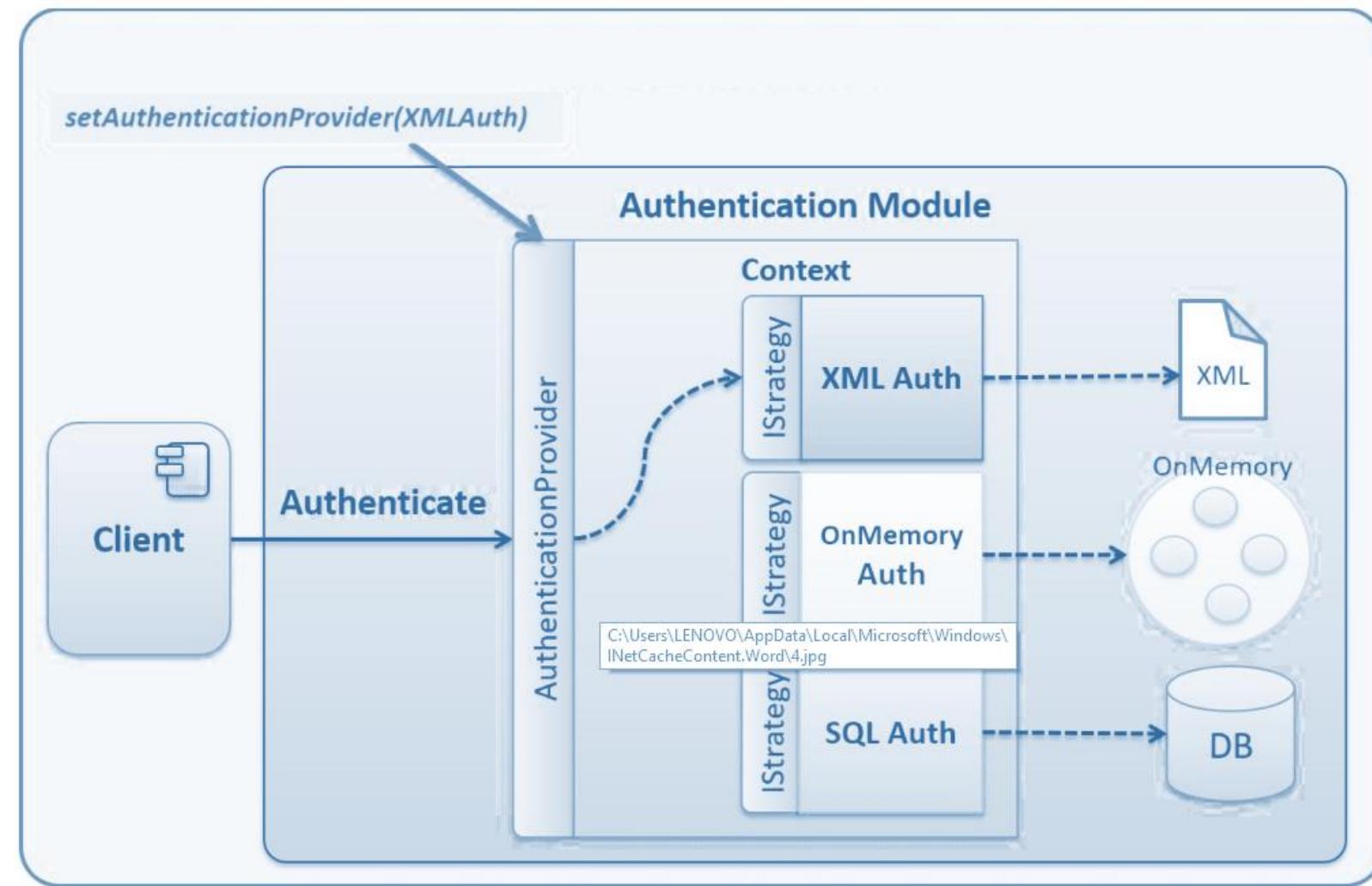
Patrón Strategy

El escenario:

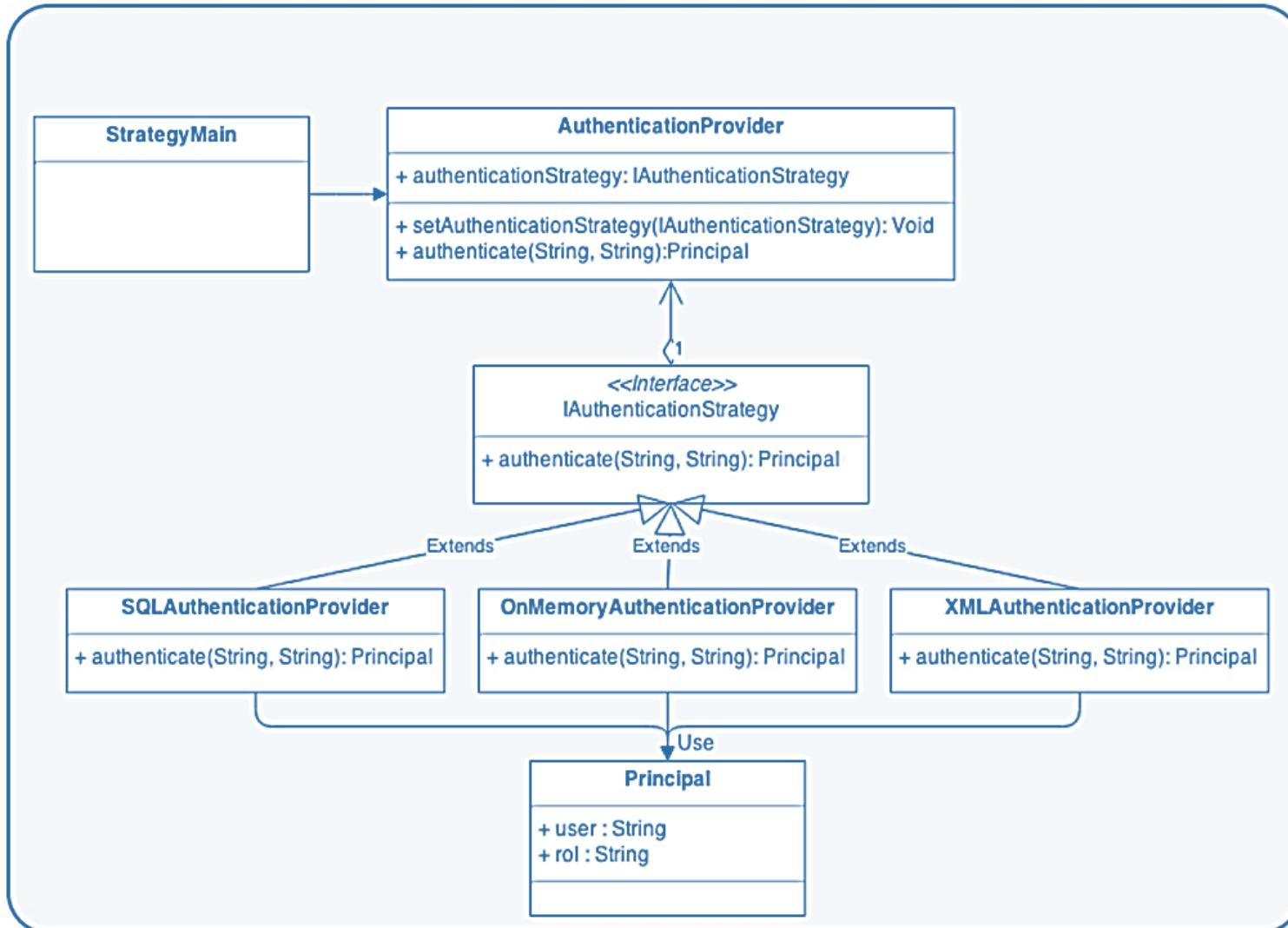


Patrón Strategy

La solución:



Patrón Strategy



Gracias.



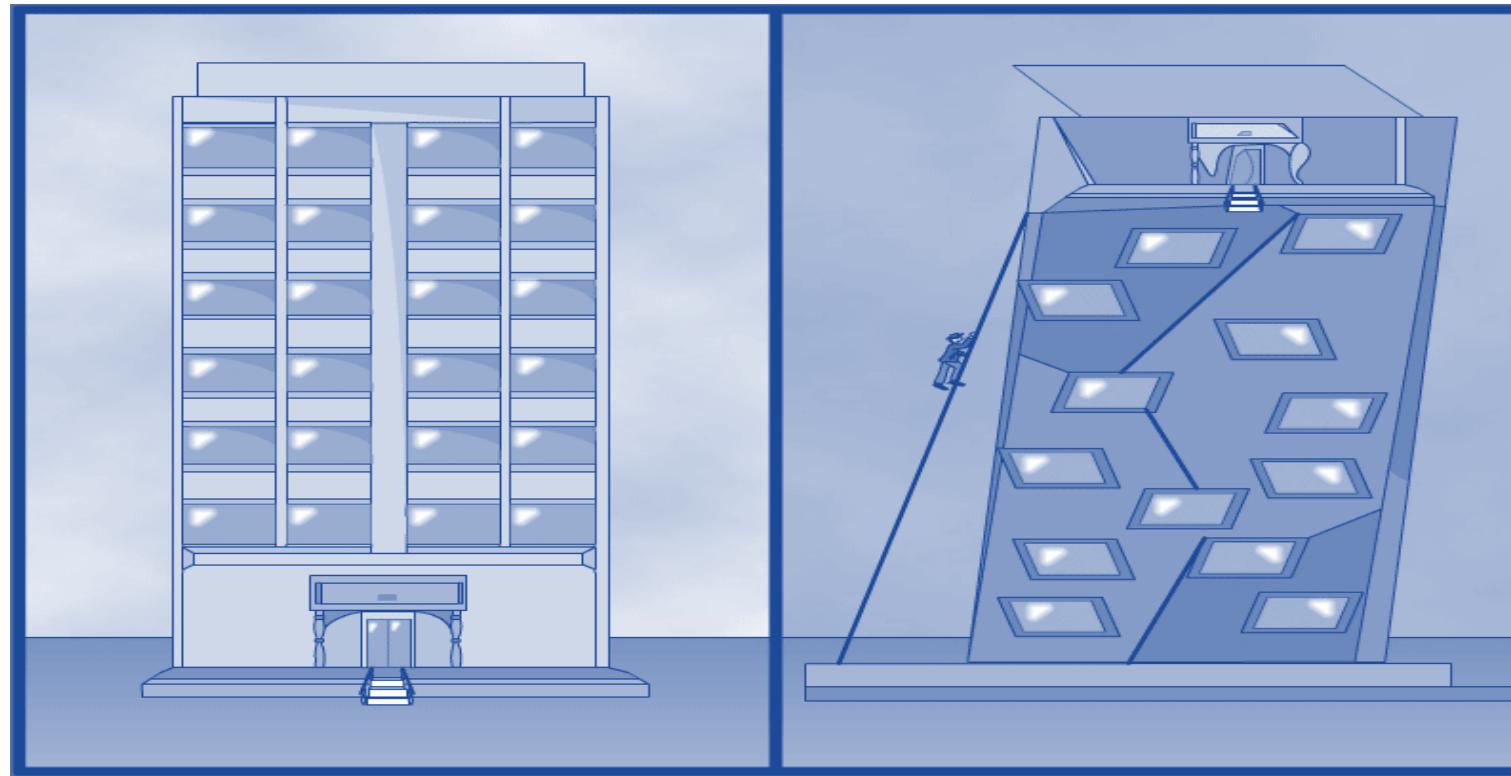
DISEÑO DE SOFTWARE

Prueba de Software

Sesión S13

El Software Actual tiene edor

¿Si su software fuera un edificio, se parecería mas a uno de la izquierda o de la derecha?

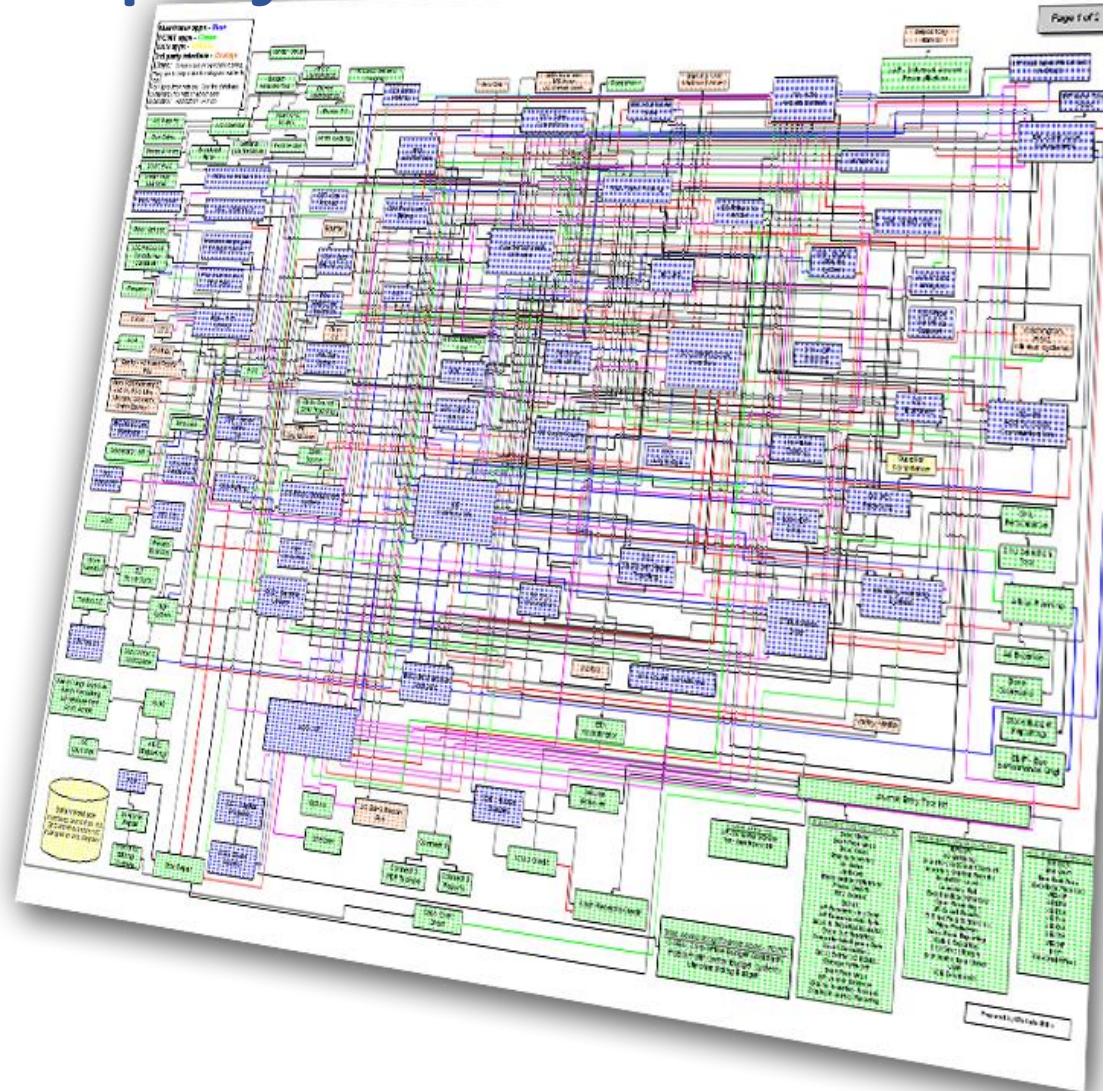


Software más Complejo

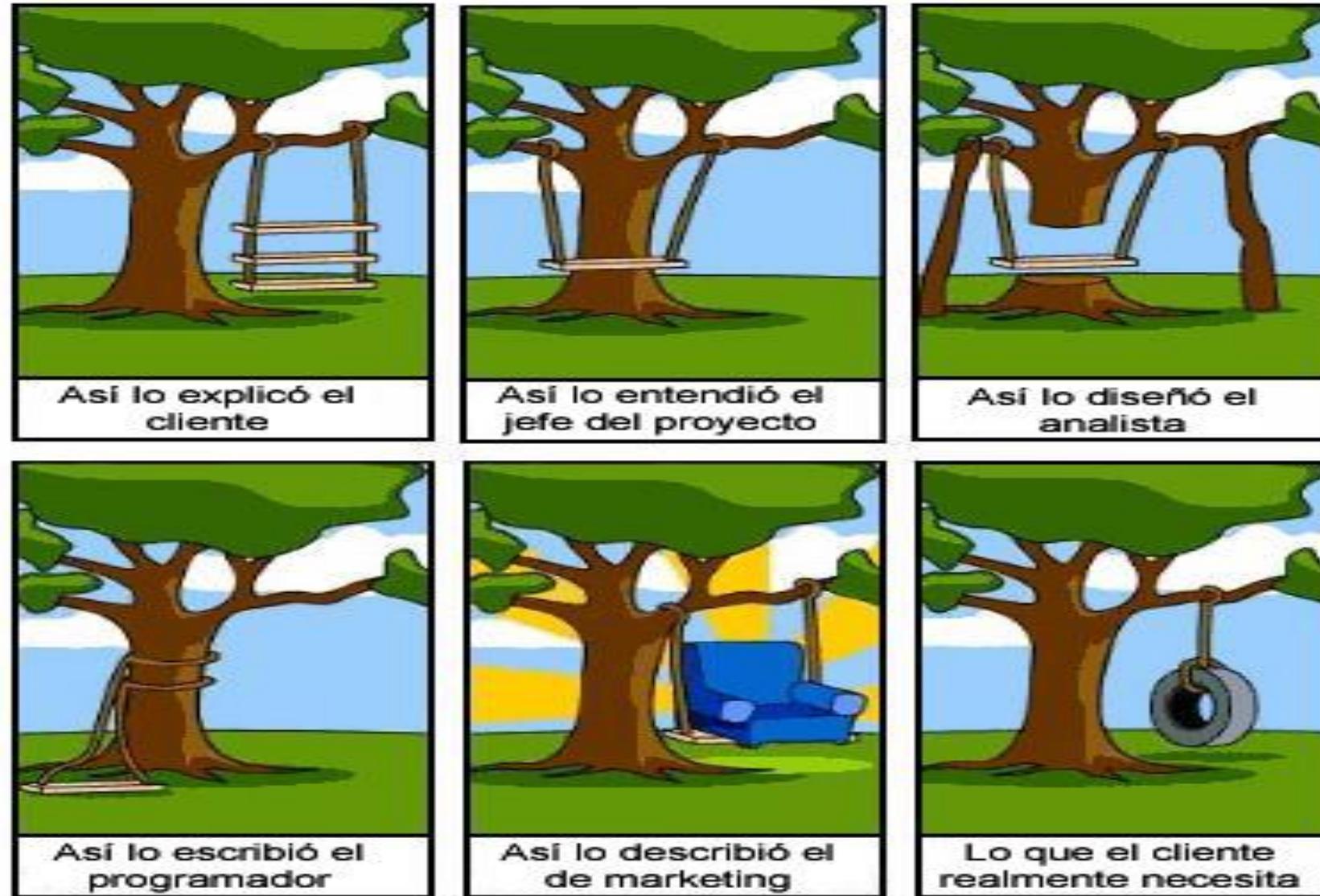
Mito: los programadores de ahora ya no programan como los de antes.

Herramientas más fáciles y productivas

El software es cada día más complejo



Problema de Entendimiento



Tipos de Desarrollo



“Casas de Perros”
Proyectos Escolares
Poco \$



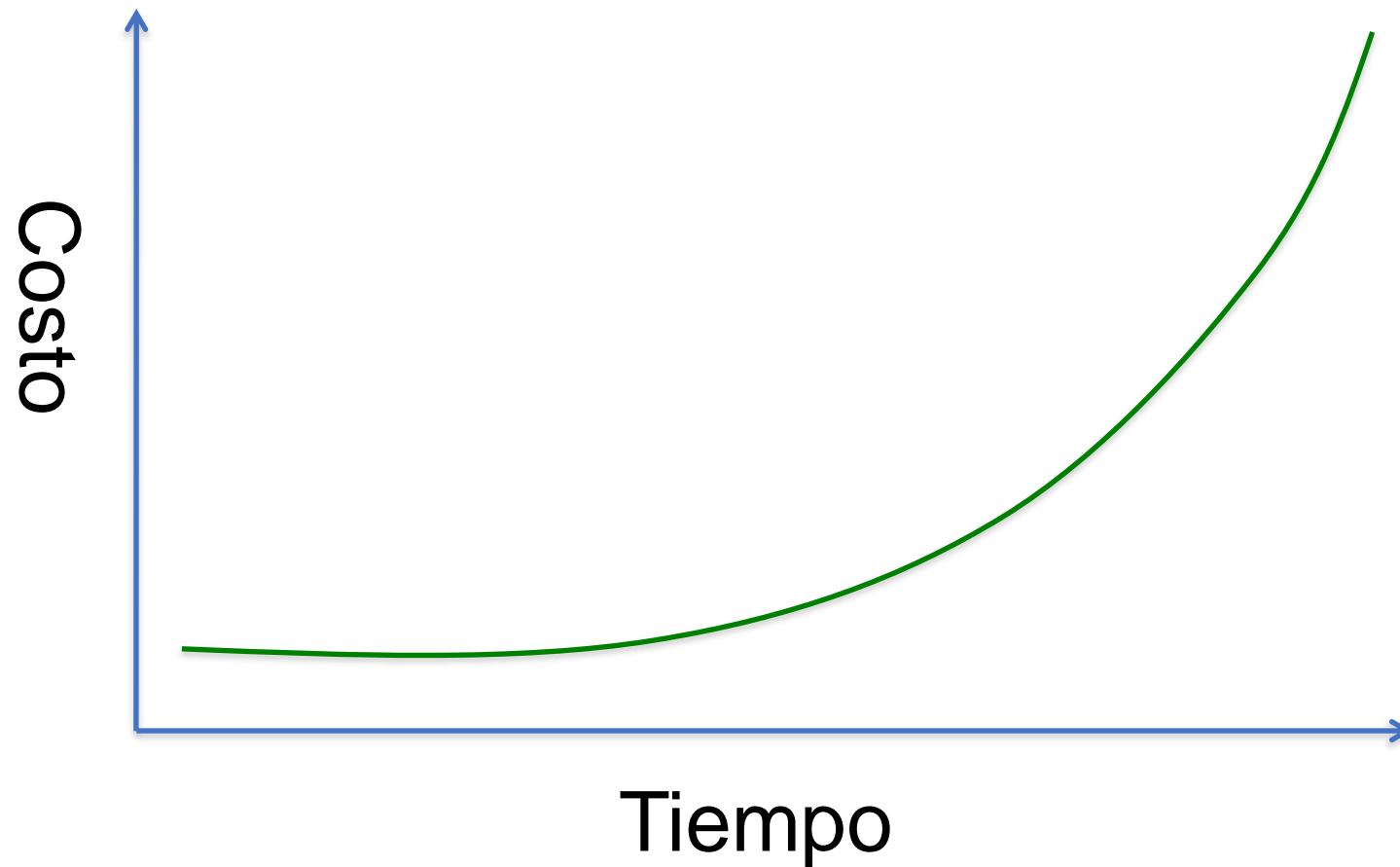
Casas
Proyecto de PyMES
Rentable \$



Edificios
Grandes Corporativos
Mucho \$\$\$\$

Costos

40% de los Costos de Desarrollo son para Pruebas



Objetivos

- **Comprender** las dificultades asociadas a la validación y verificación del software.
- **Conocer** las técnicas básicas de prueba de programas: caja blanca y caja negra.
- Ser capaces de **diseñar** casos de prueba para un módulo o función utilizando las técnicas del camino básico y de la partición equivalente.

Introducción

Pruebas: factor crítico para determinar la calidad del software

La Prueba de Software puede definirse como una actividad en la cual un sistema o uno de sus componentes se ejecuta en circunstancias previamente especificadas, los resultados se observan y registran, y se realiza una evaluación de algún aspecto: corrección, robustez, eficiencia, etc.

El objetivo de una prueba es descubrir algún error.

Caso de prueba (test case): «un conjunto de entradas, condiciones de ejecución y resultados esperados desarrollados para un objetivo particular»

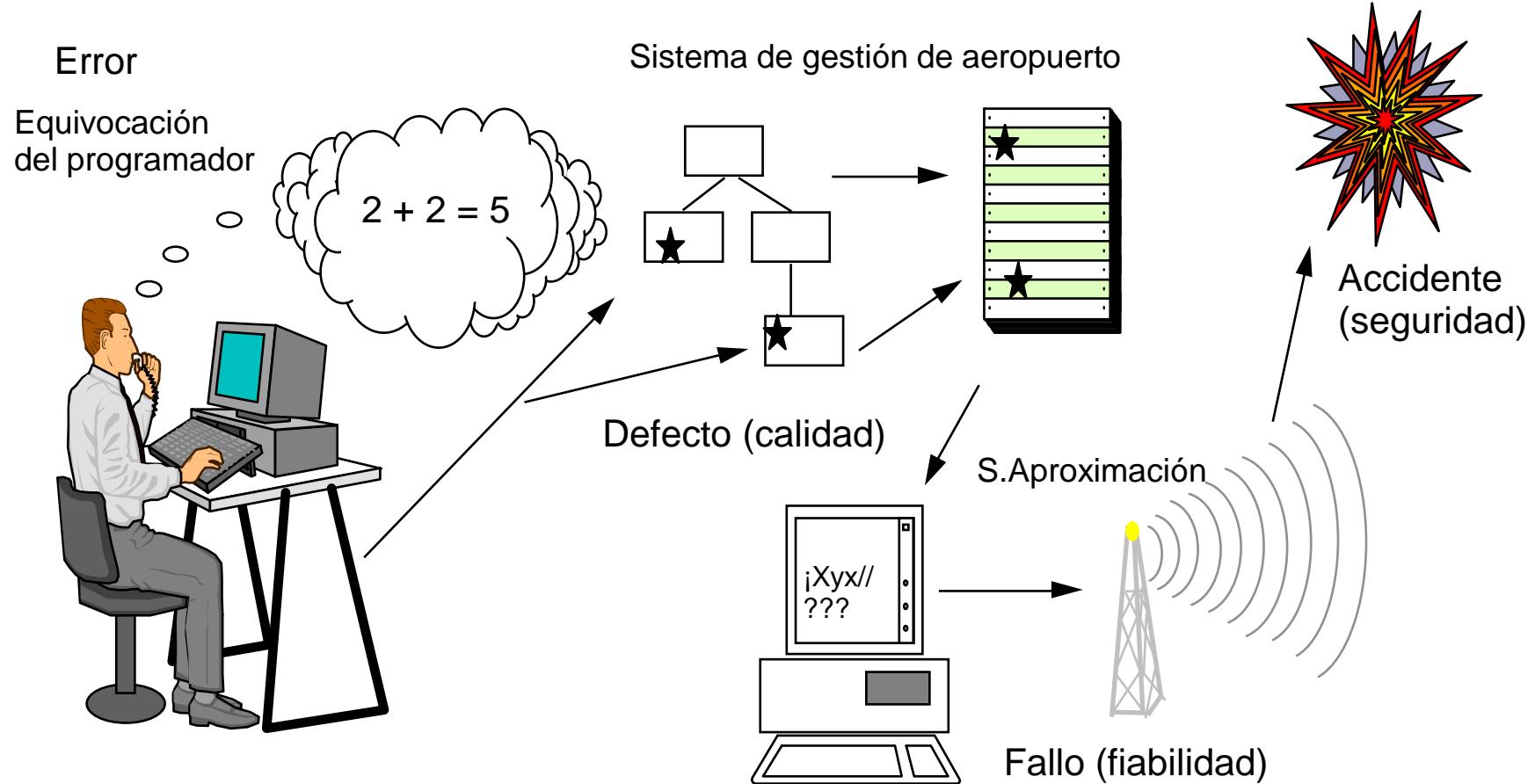
Un caso de prueba es bueno cuando su ejecución conlleva una probabilidad elevada de encontrar un error.

El éxito de la prueba se mide en función de la capacidad de detectar un error que estaba oculto.

Definiciones

- **Verificación**
- **Validación**
- **Defecto**
- **Fallo**
- **Error**

Relación entre error, defecto y fallo



Recomendaciones para la prueba

- Cada caso de prueba debe definir el resultado de salida esperado.
- El programador debe evitar probar sus propios programas, ya que desea (consciente o inconscientemente) demostrar que funcionan sin problemas.
- Se debe inspeccionar a conciencia el resultado de cada prueba, y así, poder descubrir posibles síntomas de defectos.
- Al generar casos de prueba, se deben incluir tanto datos de entrada válidos y esperados como no válidos e inesperados.

Recomendaciones para la prueba

Las pruebas deben centrarse en dos objetivos (es habitual olvidar el segundo):

Probar que el programa **hace lo que debe hacer**.

Probar que el programa **no hace lo que no debe hacer**.

Todos los casos de prueba deben documentarse y diseñarse con cuidado.

No deben hacerse planes de prueba suponiendo que, prácticamente, no hay defectos en los programas y, por lo tanto, dedicando pocos recursos a las pruebas.

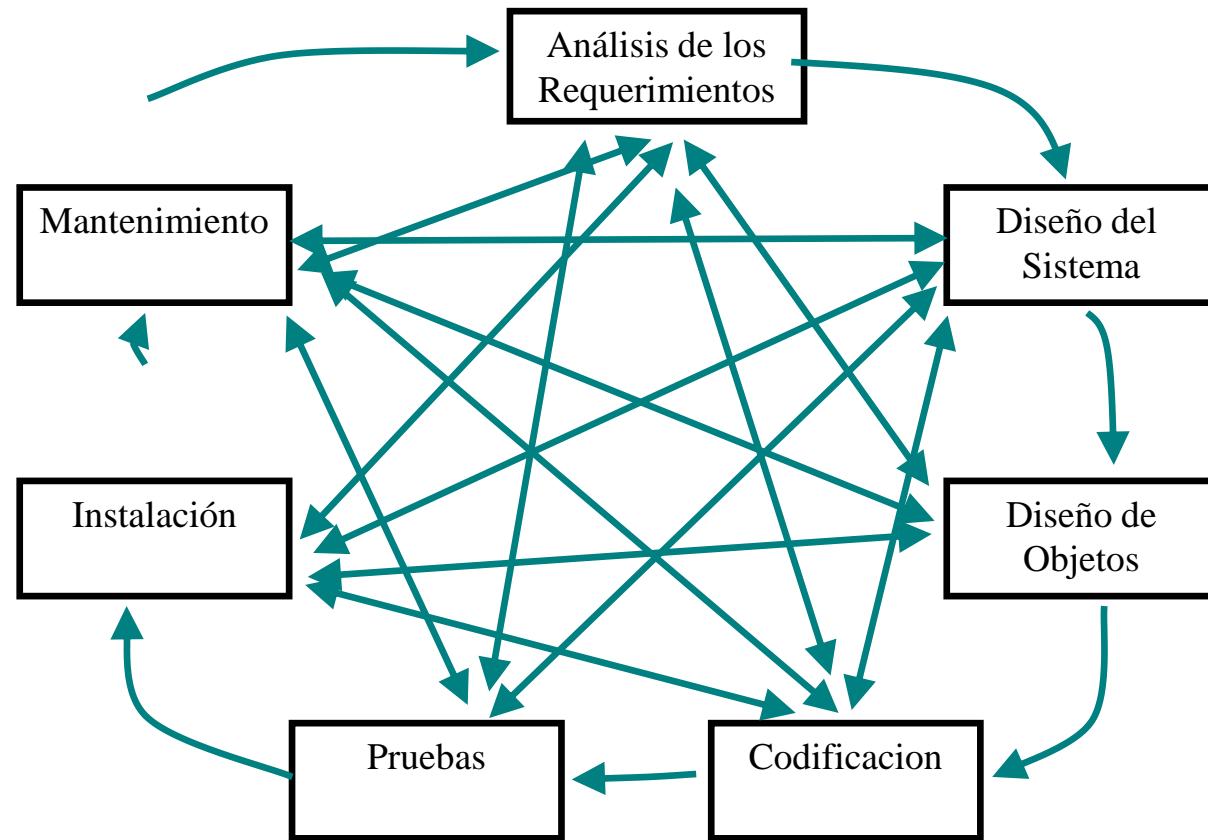
La experiencia indica que donde hay un defecto pueden aparecer otros.

Pruebas

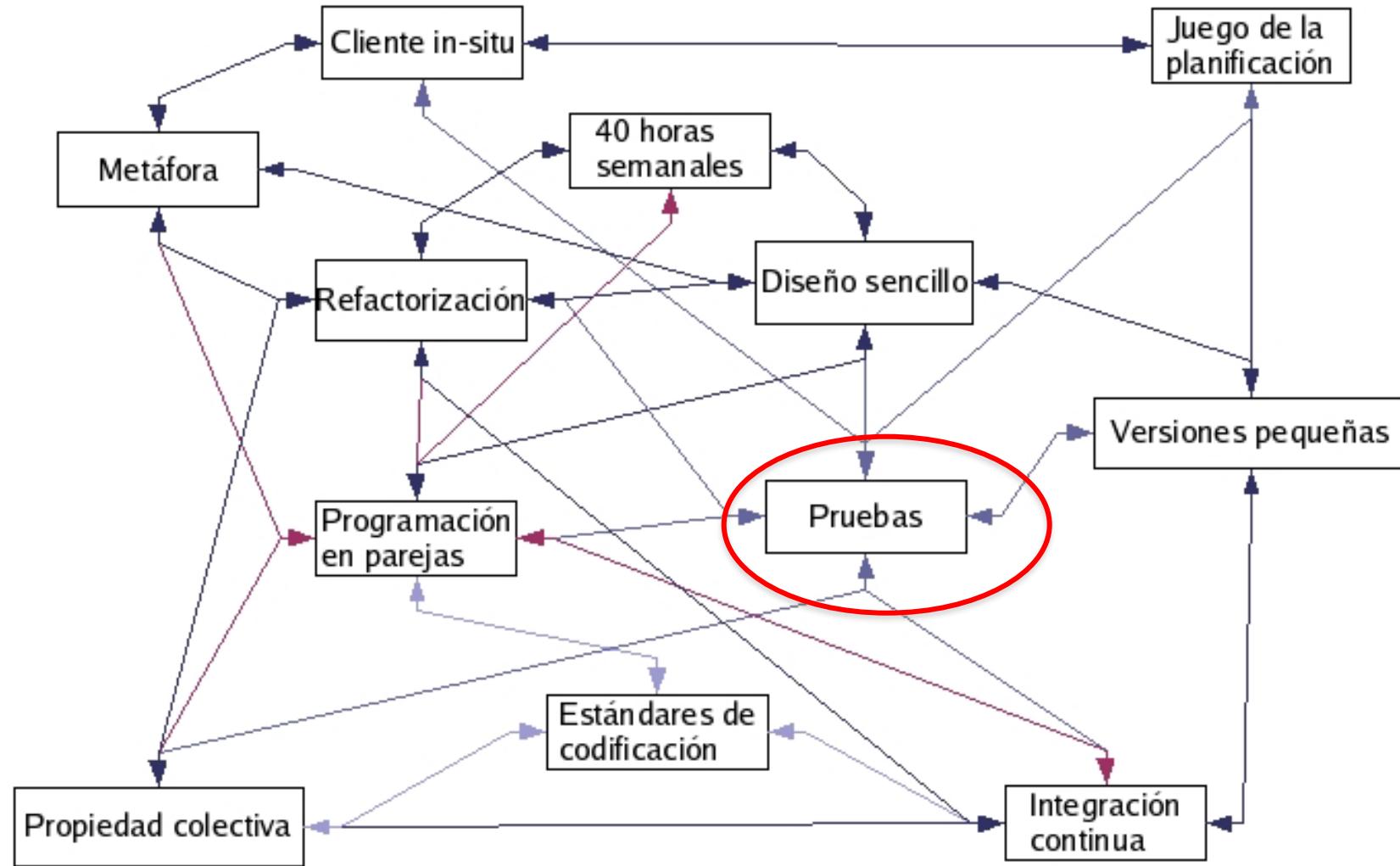
“El testing puede probar la presencia de errores pero no la ausencia de ellos” Edsger Dijkstra

Las pruebas deben de hacerse en todas las fases del desarrollo

Pruebas Desarrollo de Software

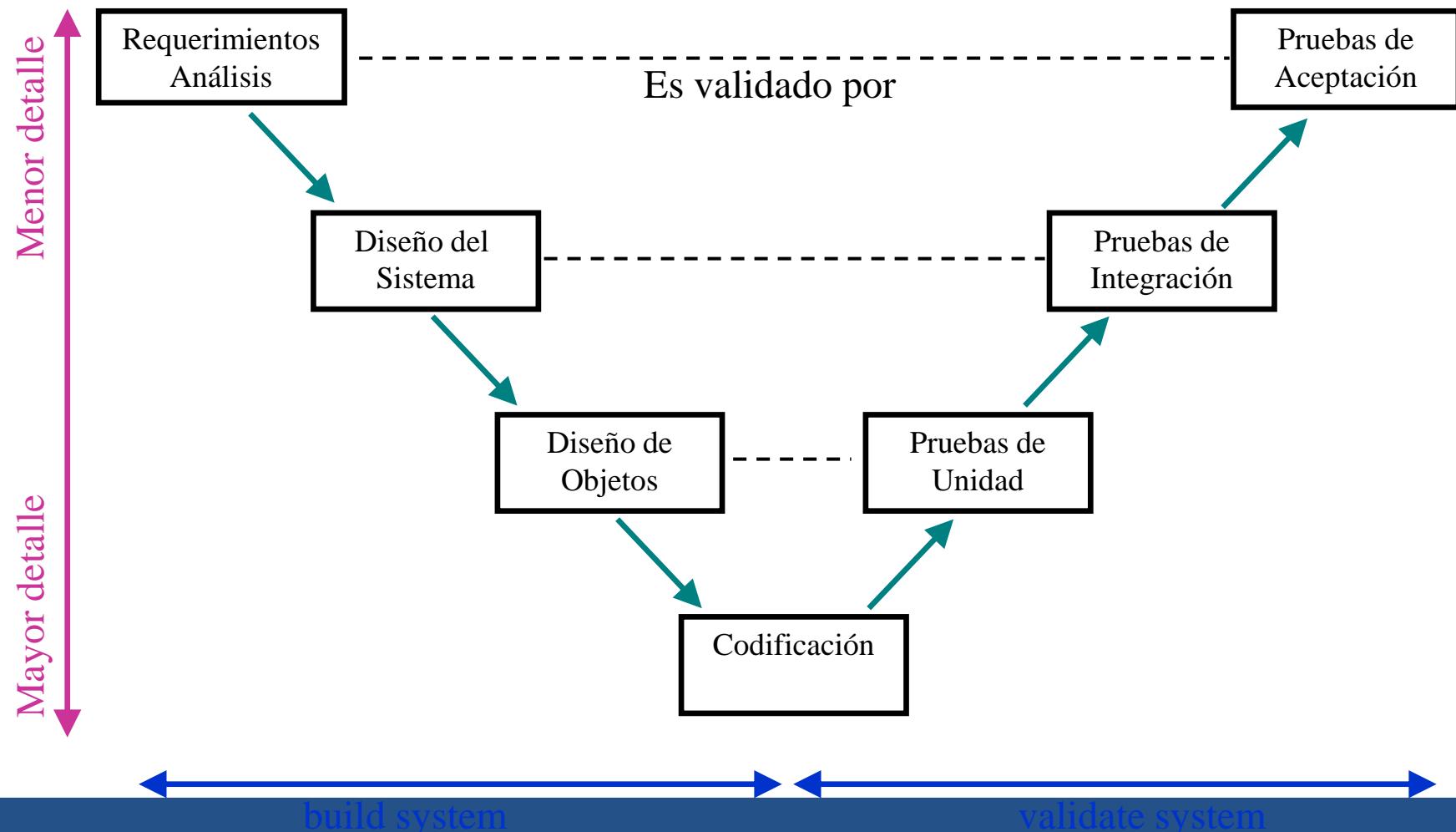


Pruebas en XP

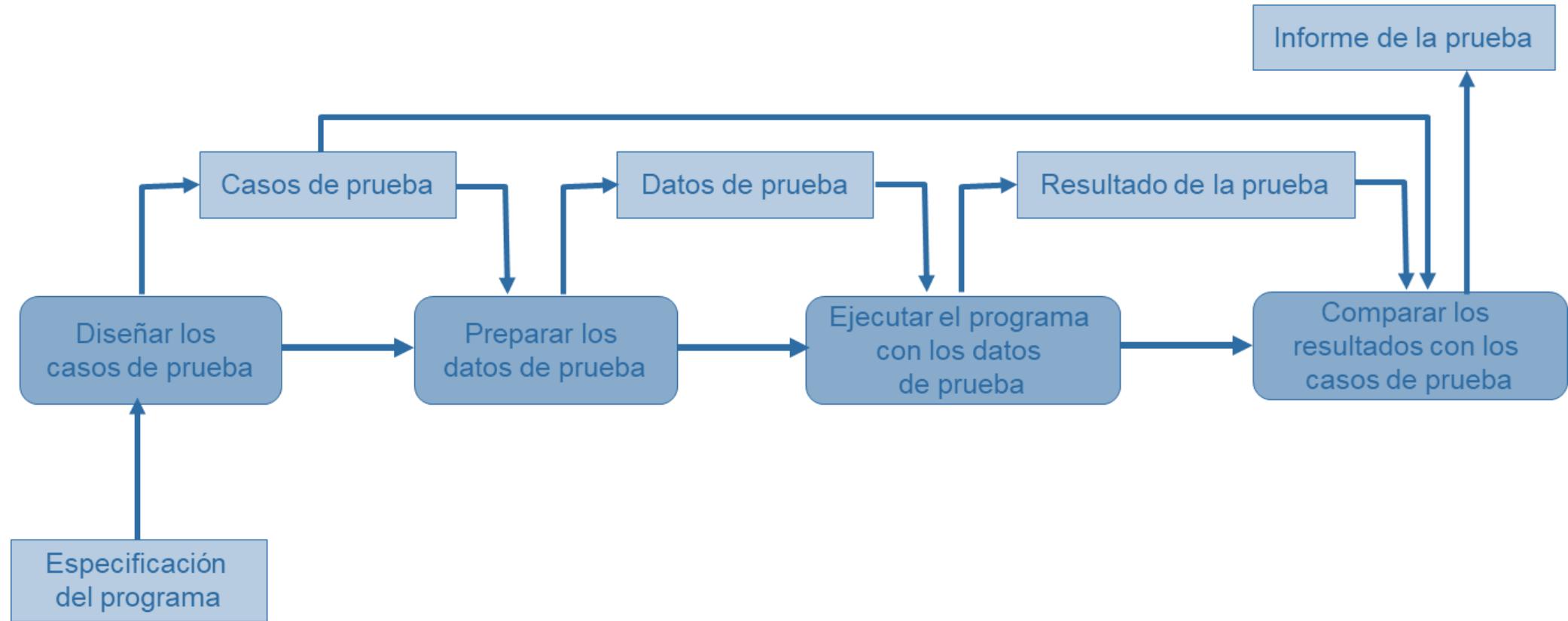


Modelo de Pruebas

Modelo V



Flujo de Información de la Prueba



Diseño de casos de Prueba

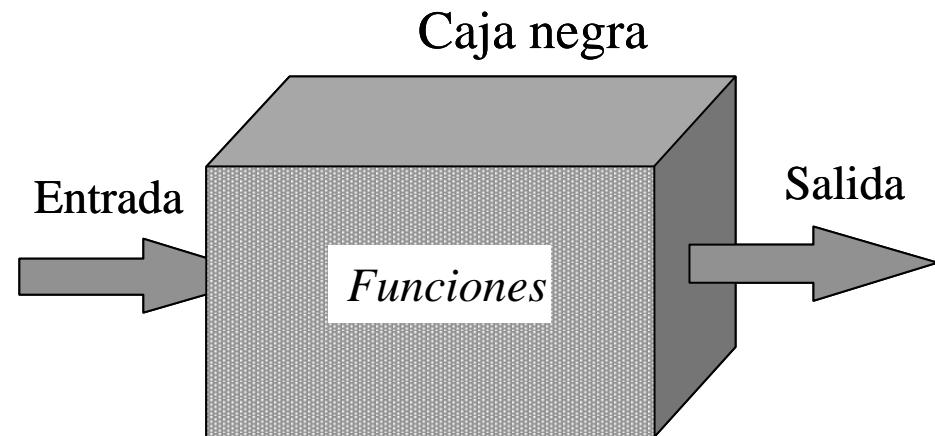
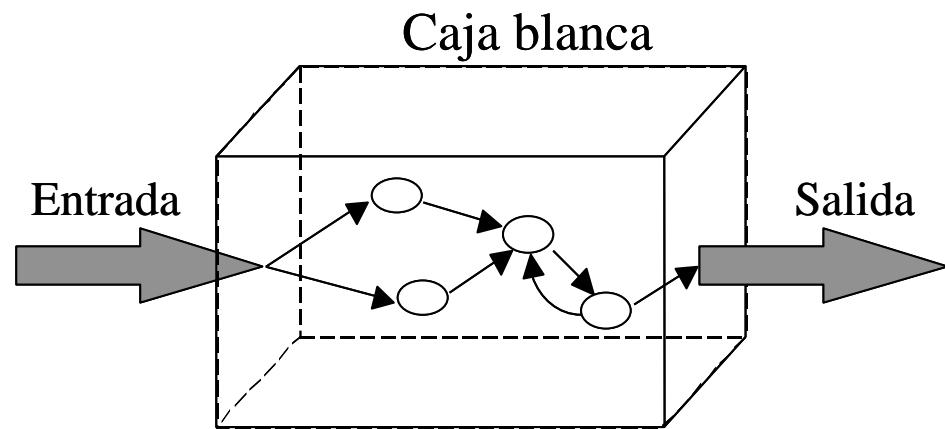
El diseño de casos de prueba para la verificación del software puede significar un esfuerzo considerable (cerca del 40% del tiempo total de desarrollo)

Existen fundamentalmente tres enfoques:

- Prueba de caja blanca
- Prueba de caja negra
- Pruebas aleatorias

Combinar ambos enfoques permite lograr mayor fiabilidad.

Diseño de casos de Prueba



Formato Plan de Pruebas

ID: 1	Nombre: Enviar artículo
	Probado por: Juan Perez
	Descripción: Se introducen los datos del artículo y de los autores.
	Condiciones de Entrada: nombreArticulo="Calidad del Sw" ... emailAutor="joseC@unmsm.edu.pe"
	Resultado Esperado: El sistema confirma la correcta recepción del artículo enviando un e-mail al autor de contacto con un userid y password para que el autor pueda posteriormente acceder al artículo.
	Resultado Obtenido: Se generan bien userid y password pero el correo no llegó.
	Criterio de Aceptación: No.

Diseño de casos de Prueba

La prueba de caja blanca o prueba estructural se basa en el estudio minucioso de toda la operatividad de una parte del sistema, considerando los detalles procedimentales.

Se plantean distintos caminos de ejecución alternativos y se llevan a cabo para observar los resultados y contrastarlos con lo esperado.

En principio se podría pensar que es viable la verificación mediante la prueba de la caja blanca de la totalidad de caminos de un procedimiento.

Esto es prácticamente imposible en la mayoría de los casos reales dada la exponencialidad en el número de combinaciones posibles.

Diseño de casos de Prueba

- La **prueba de caja negra** o prueba funcional principalmente analiza la compatibilidad en cuanto a las interfaces de cada uno de los componentes software entre sí.
- La **prueba aleatoria** consiste en utilizar modelos que representen las posibles entradas del programa para crear a partir de ellos los casos de prueba.

Pruebas de caja blanca

La prueba de la caja blanca usa la estructura de control del diseño procedural para derivar los casos de prueba.

Idea: no es posible probar todos los caminos de ejecución distintos, pero sí confeccionar casos de prueba que garanticen que se verifican todos los caminos de ejecución llamados independientes.

Verificaciones para cada camino independiente:

Probar sus dos facetas desde el punto de vista lógico, es decir, verdadera y falsa.

Ejecutar todos los bucles en sus límites operacionales

Ejercitar las estructuras internas de datos.

Prueba del camino básico

La **prueba del camino básico** es una técnica de prueba de la caja blanca propuesta por Tom McCabe.

La idea es **derivar** casos de prueba a partir de un conjunto dado de **caminos independientes** por los cuales puede circular el flujo de control.

Camino independiente es aquel que introduce por lo menos una sentencia de procesamiento (o condición) que no estaba considerada en el conjunto de caminos independientes calculados hasta ese momento.

Para obtener el conjunto un conjunto de caminos independientes se construirá el Grafo de Flujo asociado y se calculará su Complejidad Ciclomática.

Prueba del camino básico: Grafos de Flujo

El flujo de control de un programa puede representarse por un grafo de flujo.

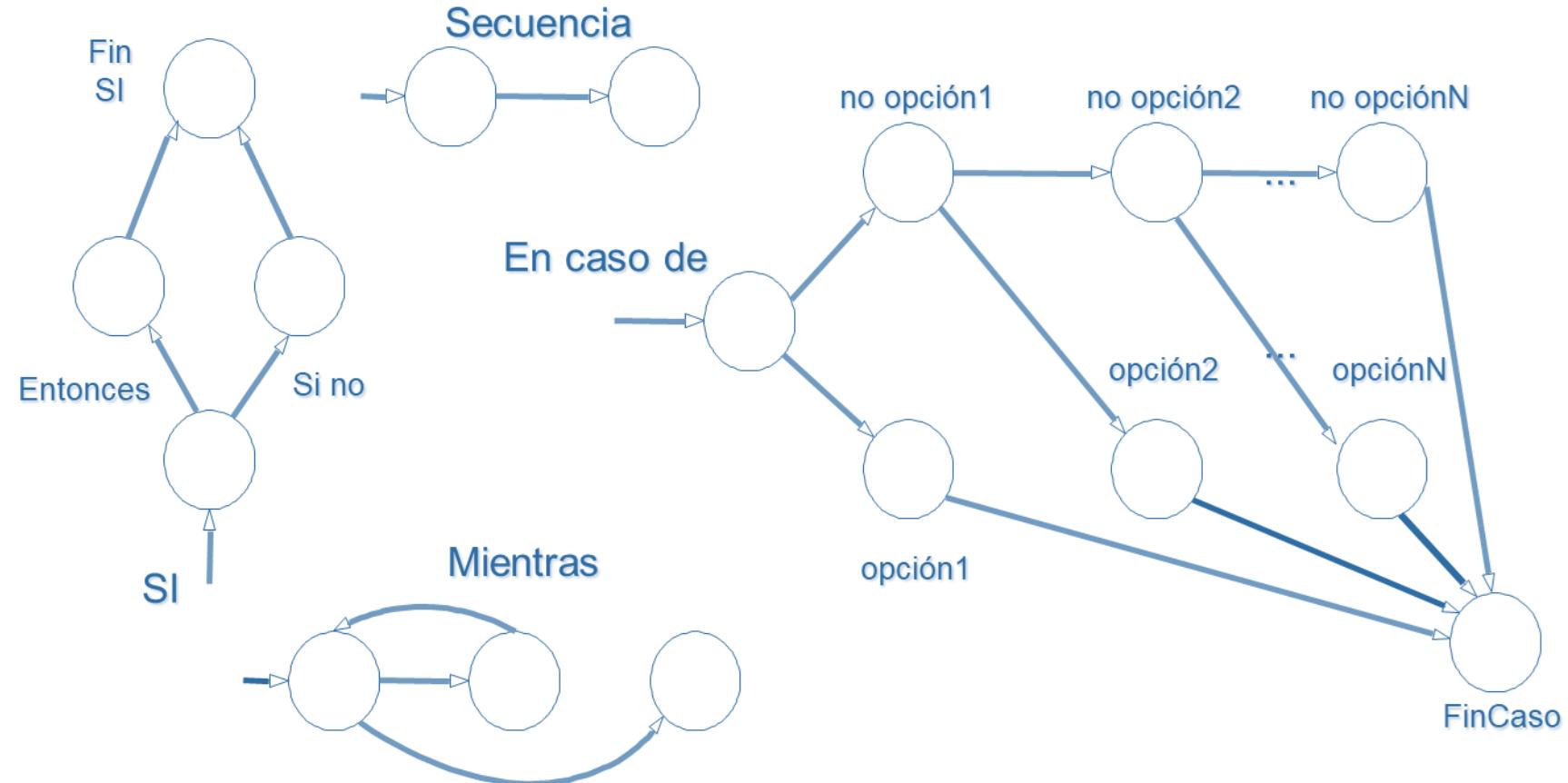
Cada nodo del grafo corresponde a una o más sentencias de código fuente.

Cada nodo que representa una condición se denomina nodo predicado.

Cualquier representación del diseño procedural se puede traducir a un grafo de flujo.

Un camino independiente en el grafo es el que incluye alguna arista nueva, es decir, que no estaba presente en los caminos definidos previamente.

Prueba del camino básico: Grafos de Flujo



Prueba del camino básico: Grafos de Flujo

Si se diseñan casos de prueba que cubran los caminos básicos, se garantiza la ejecución de cada sentencia al menos una vez y la prueba de cada condición en sus dos posibilidades lógicas (verdadera y falsa).

El conjunto básico para un grafo dado puede no ser único, depende del orden en que se van definiendo los caminos.

Cuando aparecen condiciones lógicas compuestas la confección del grafo es más compleja.

Prueba del camino básico: Grafos de Flujo

Ejemplo:

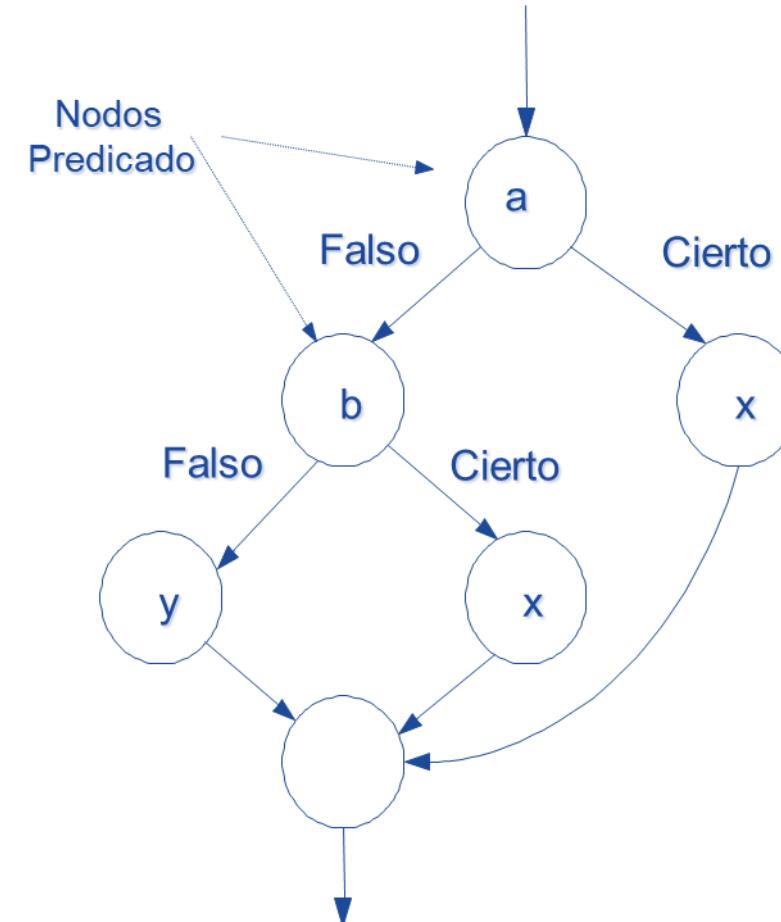
SI a O b

Entonces
hacer x

Si No

hacer y

FinSI



Prueba del camino básico: Complejidad Ciclomática

Complejidad ciclomática de un grafo de flujo, $V(G)$, indica el número máximo de caminos independientes en el grafo.

Puede calcularse de tres formas alternativas:

El **número de regiones** en que el grafo de flujo divide el plano.

$V(G) = A - N + 2$, donde A es el número de aristas y N es el número de nodos.

$V(G) = P + 1$, donde P es el número de nodos predicado.

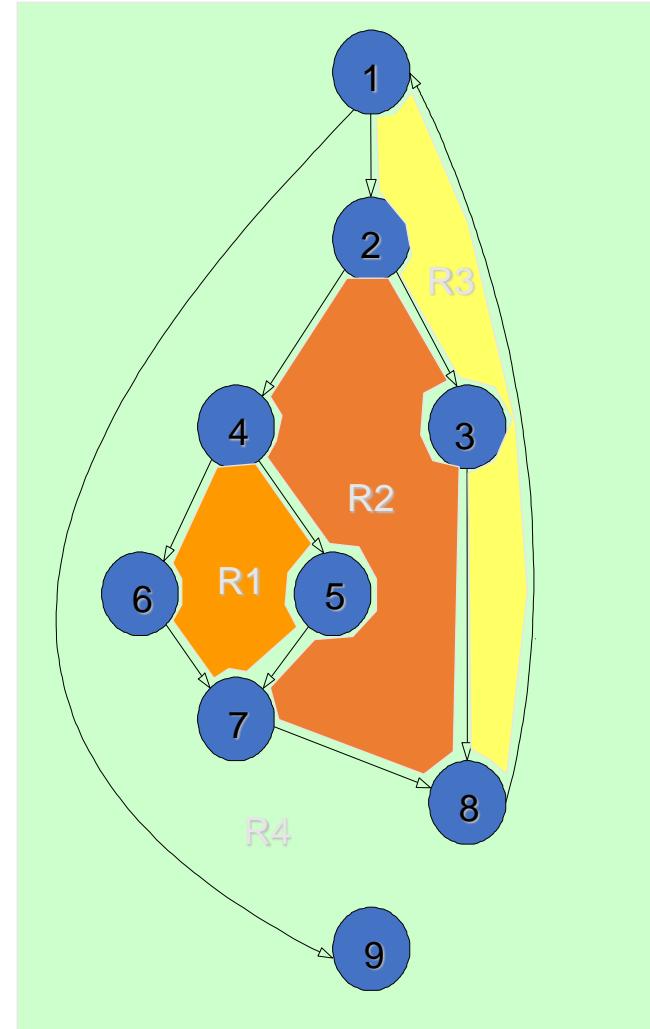
Prueba del camino básico: Complejidad Ciclomática

$$V(G) = 4$$

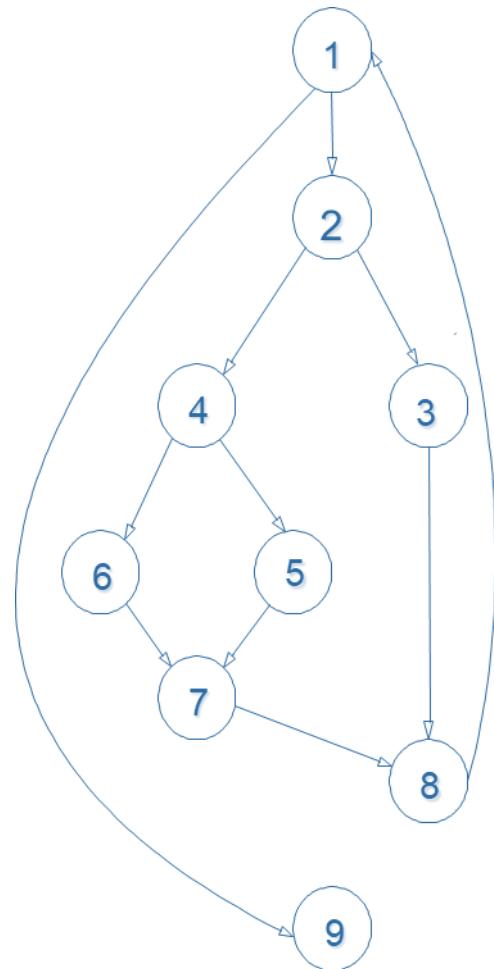
El grafo de la figura delimita cuatro regiones.

$$11 \text{ aristas} - 9 \text{ nodos} + 2 = 4$$

$$3 \text{ nodos predicado} + 1 = 4$$



Prueba del camino básico: Complejidad ciclomática



El conjunto de caminos independientes del grafo será 4.

Camino 1: 1-9

Camino 2: 1-2-4-5-7-8-1-9

Camino 3: 1-2-4-6-7-8-1-9

Camino 4: 1-2-3-8-1-9

Cualquier otro camino no será un camino independiente, p.e.,

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

ya que es simplemente una combinación de caminos ya especificados

Los cuatro caminos anteriores constituyen un conjunto básico para el grafo

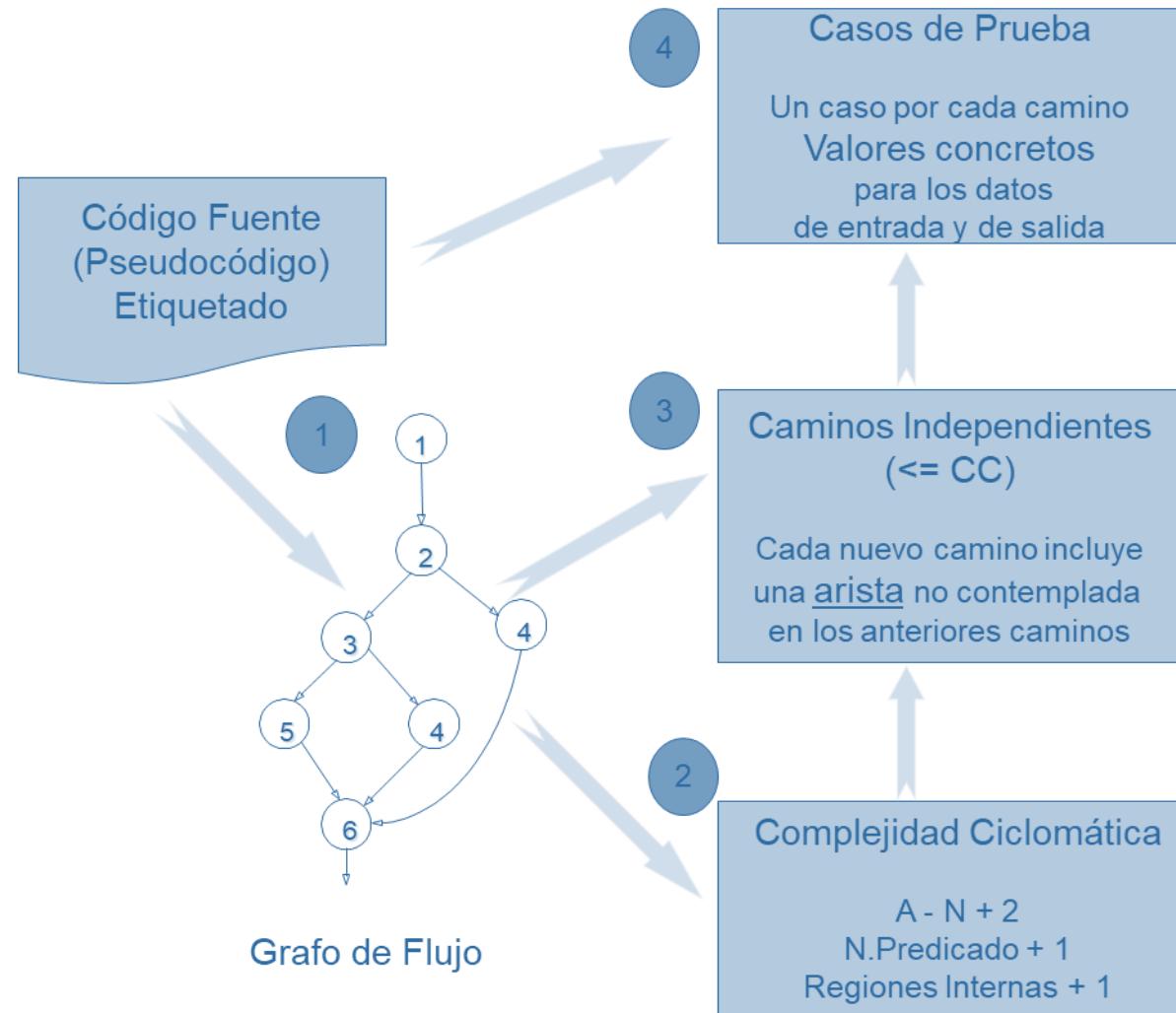
Prueba del camino básico: Derivación de casos de prueba

El método de prueba del camino básico se puede aplicar a un diseño procedural detallado (pseudocódigo) o al código fuente de la aplicación.

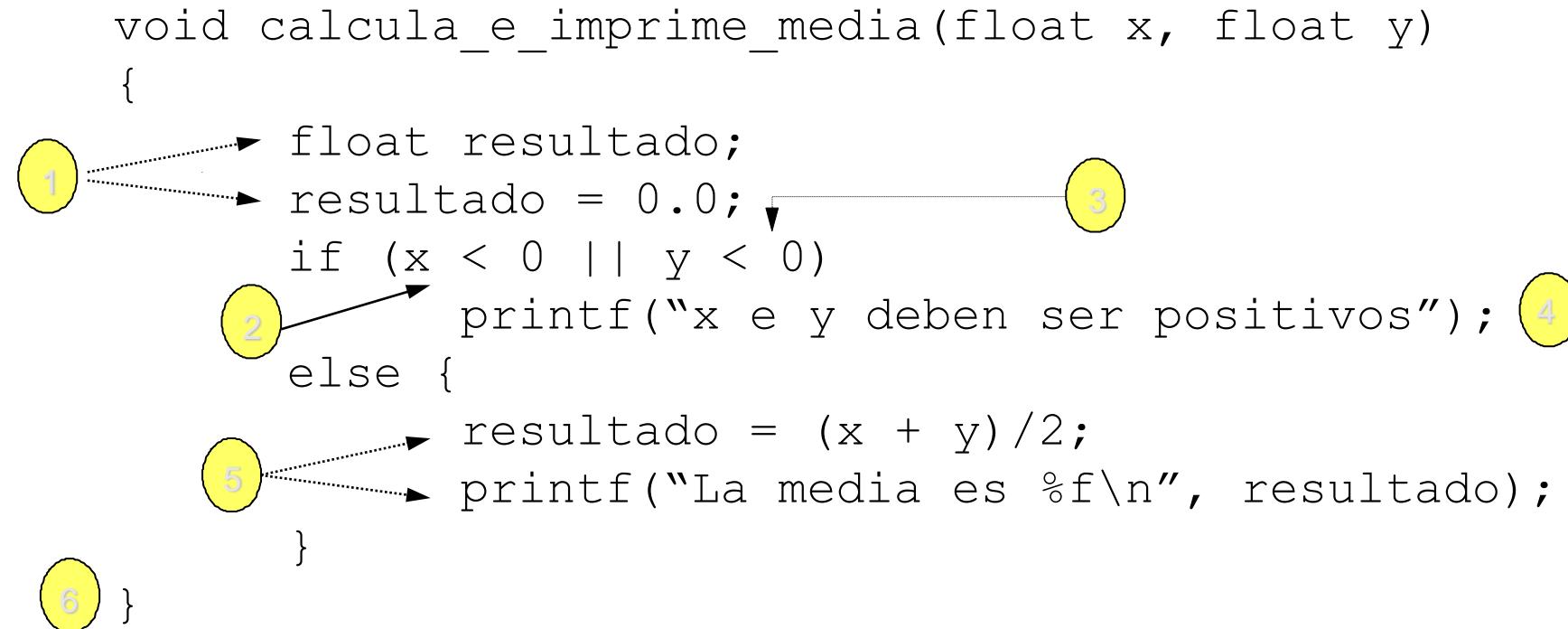
Pasos para diseñar los casos de prueba:

0. Se etiqueta el código fuente, o el pseudocódigo, enumerando cada instrucción y cada condición simple.
1. A partir del código fuente etiquetado, se dibuja el grafo de flujo asociado.
2. Se calcula la complejidad ciclomática del grafo.
3. Se determina un conjunto básico de caminos independientes.
4. Se preparan los casos de prueba que obliguen a la ejecución de cada camino del conjunto básico.

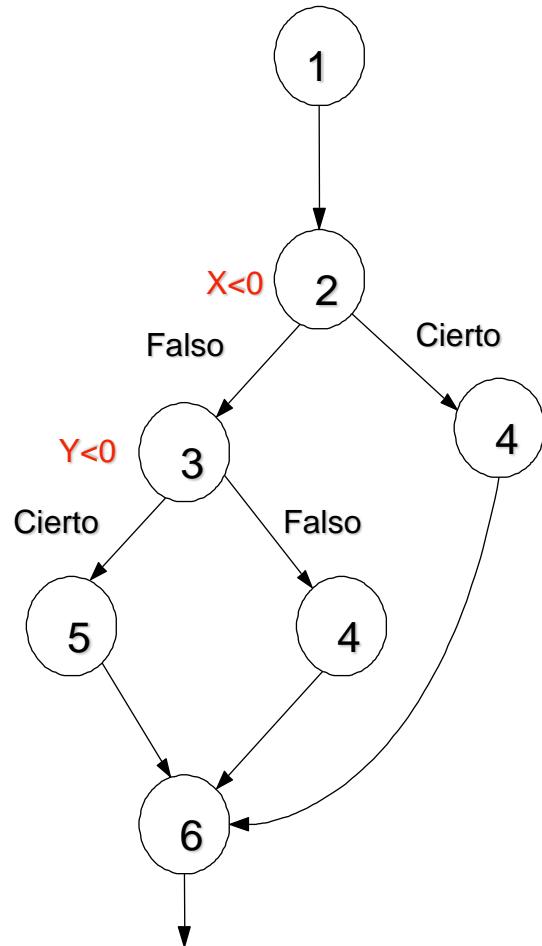
Prueba del camino básico: Derivación de casos de prueba



Prueba del camino básico: Ejemplo



Prueba del camino básico: Ejemplo



$V(G) = 3$ regiones. Por lo tanto, hay que determinar tres caminos independientes.

- Camino 1: 1-2-3-5-6
- Camino 2: 1-2-4-6
- Camino 3: 1-2-3-4-6

Casos de prueba para cada camino:

Camino 1: $x=3, y=5, rdo=4$

Camino 2: $x=-1, y=3, rdo=0$, error

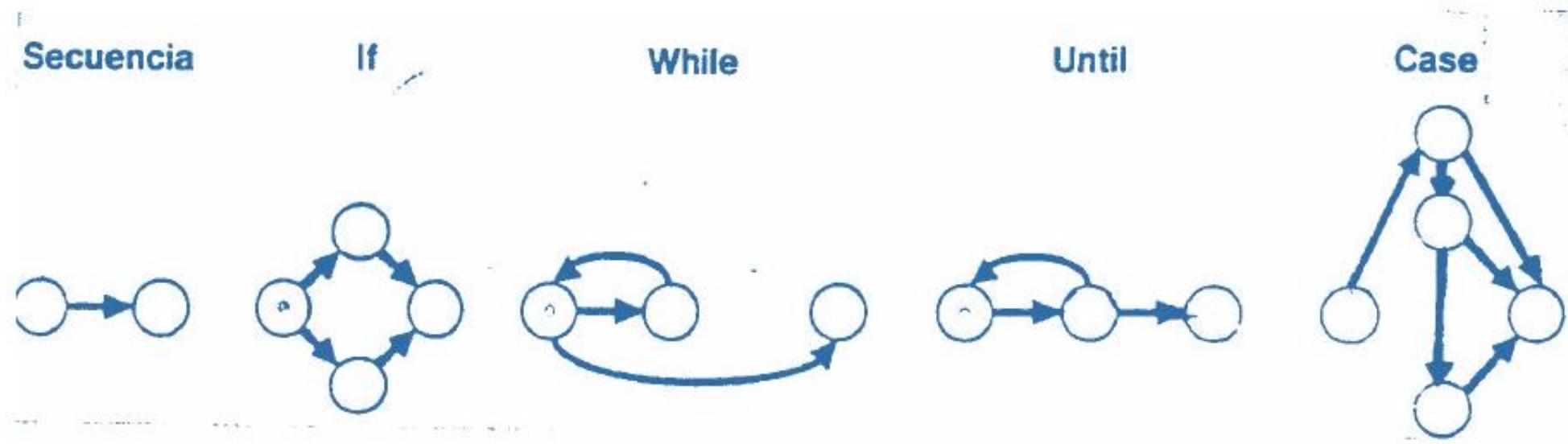
Camino 3: $x=4, y=-3, rdo=0$, error

Pruebas de estructura de control

La técnica de prueba del camino básico del punto anterior es una de las muchas existentes para la pruebas de la estructura de control.

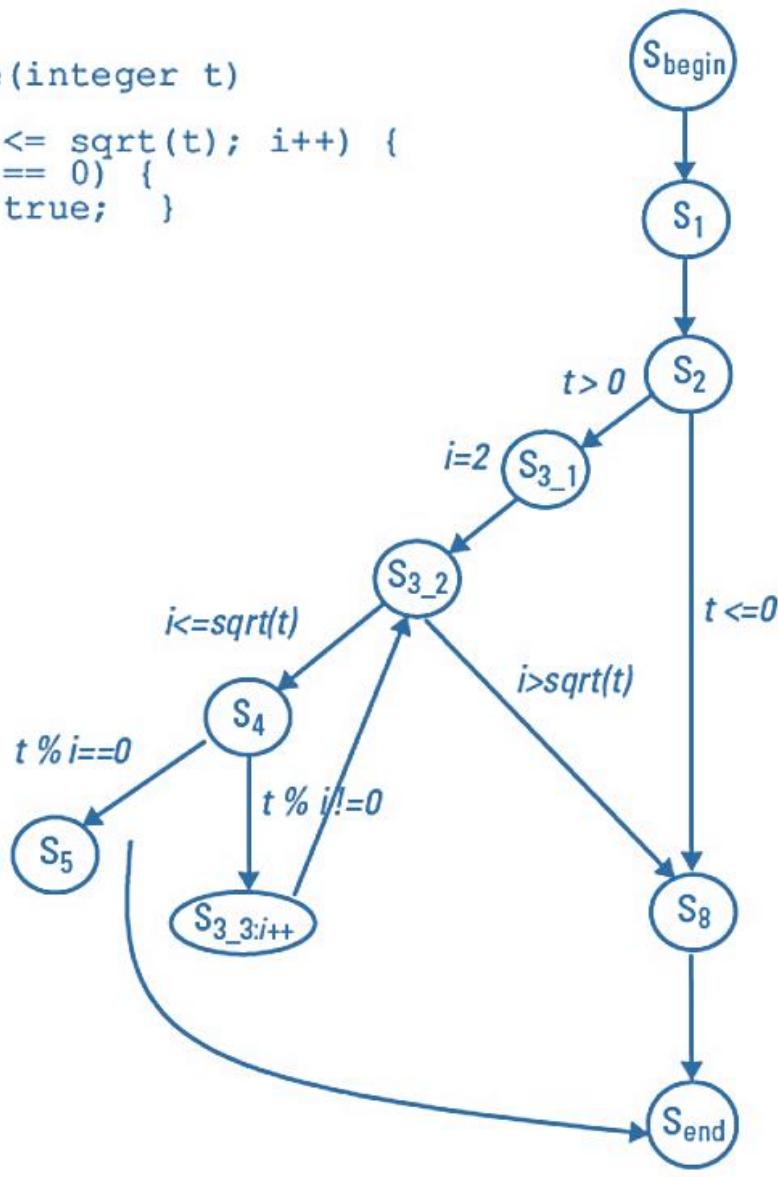
Aunque sea alta la efectividad de la prueba del camino básico, no nos asegura completamente la corrección del software.

Veremos a continuación otras variantes de la prueba de la estructura de control. Estas variantes amplían el abanico de pruebas mejorando la fiabilidad de las pruebas de caja blanca.



```
S1  Boolean Composite(integer t)
S2  if (t > 0) {
S3      for( i = 2; i <= sqrt(t); i++) {
S4          if ( t % i == 0) {
S5              return true;  }
S6      }
S7  }
S8  return false;
```

```
S1 Boolean Composite(integer t)
S2 if (t > 0) {
S3     for( i = 2; i <= sqrt(t); i++) {
S4         if ( t % i == 0) {
S5             return true; }
S6     }
S7 }
S8 return false;
```



```
public static boolean capicua(char[] list) {  
    int index = 0;  
    int l = list.length;  
    while (index<(l-1)) {  
        if (list[index] != list[(l-index)-1]) {  
            return false;  
        }  
        index++;  
    }  
    return true;  
}
```

Pruebas de estructura de control - Prueba de Condiciones

La prueba de condiciones es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa. Los tipos de errores que pueden aparecer en una condición son los siguientes:

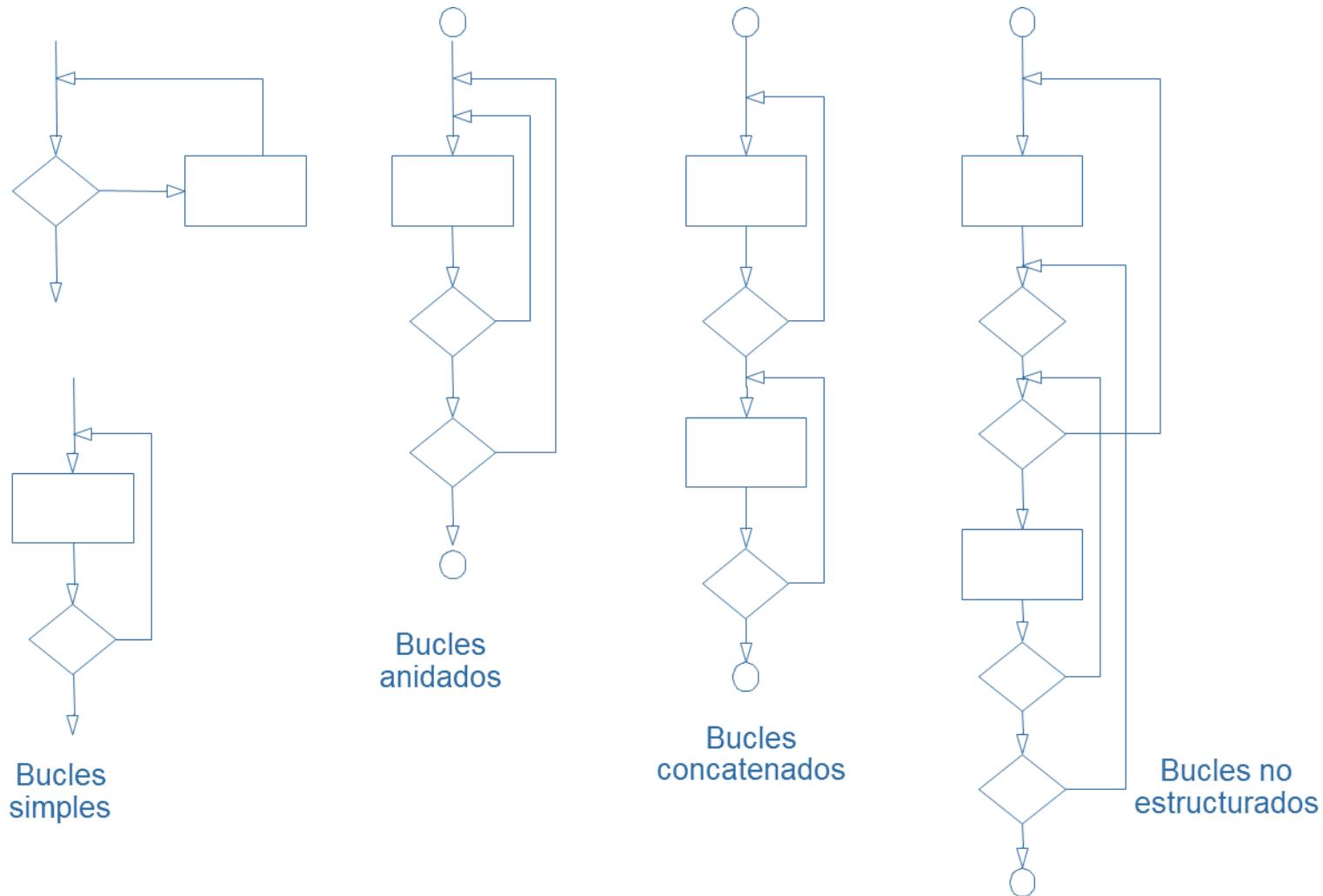
Existe un error en un operador lógico (que sobra, falta o no es el que corresponde).

Existe un error en un paréntesis lógico (cambiando el significado de los operadores involucrados).

Existe un error en un operador relacional (operadores de comparación de igualdad, menor o igual, etc.).

Existe un error en una expresión aritmética.

Pruebas de estructura de control - Prueba de Bucles



Pruebas de estructura de control - Prueba de Bucles

- **Pruebas para Bucles simples.** (n es el número máximo de iteraciones permitidos por el bucle)

Pasar por alto totalmente el bucle

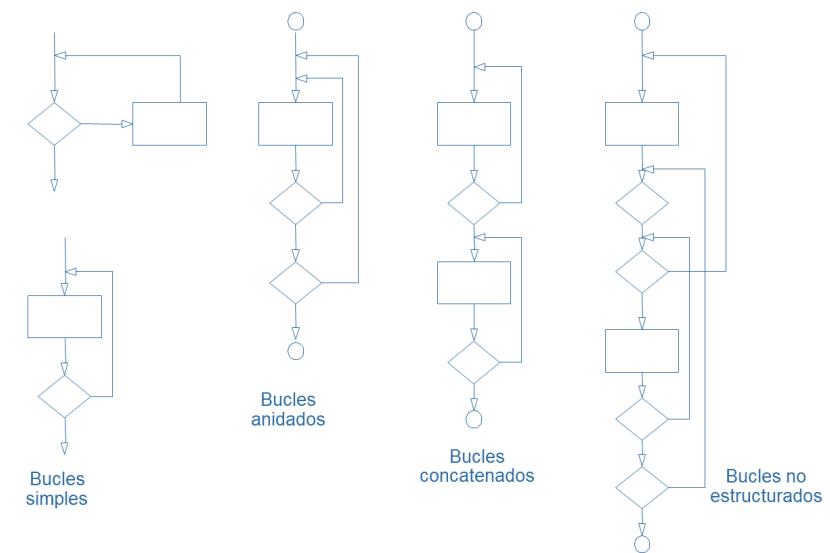
Pasar una sola vez por el bucle

Pasar dos veces por el bucle

Hacer m pasos por el bucle con $m < n$

Hacer $n-1$, n y $n + 1$ pasos por el bucle

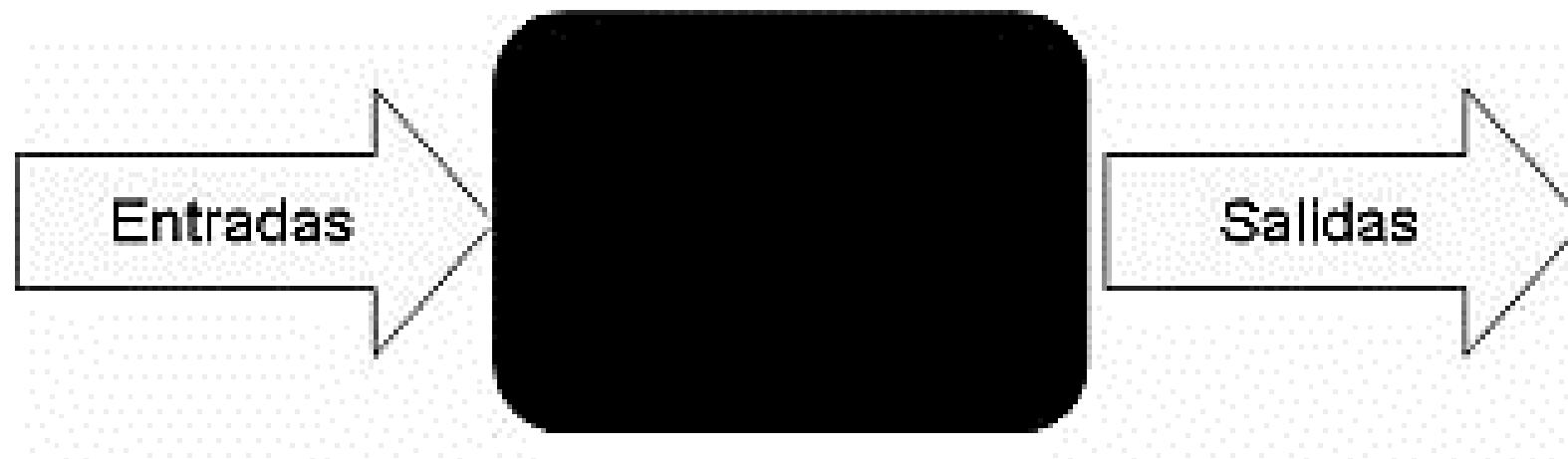
- **Pruebas para Bucles anidados**



- **Pruebas para Bucles concatenados**

Pruebas de “caja negra”

La prueba de caja negra o prueba funcional principalmente analiza la compatibilidad en cuanto a las interfaces de cada uno de los componentes software entre sí.



Partición equivalente

La partición equivalente es un método que divide el campo de entrada de un programa en clases de datos a partir de los cuales se pueden derivar casos de prueba.

La partición equivalente define casos de prueba para descubrir clases de errores.

Se define una condición de entrada para cada dato de entrada (valor numérico específico, rango de valores, conjunto de valores relacionados o condición lógica).

Partición equivalente

Las clases de equivalencia se pueden definir de acuerdo a las siguientes directrices:

- Si una condición de entrada especifica un rango, se define una clase de equivalencia válida y dos no válidas.
- Si la condición de entrada es un valor específico, se define una clase de equivalencia válida y dos no válidas.
-
- Si la condición de entrada especifica un miembro de un conjunto, se define una clase de equivalencia válida y otra no válida.
- Si la condición de entrada es lógica, se define una clase válida y otra no válida.

Ejemplo Partición equivalente

Datos de entrada a una aplicación bancaria

- **Código de área:** en blanco o número de 3 dígitos.
- **Prefijo:** número de 3 dígitos que no comience por 0 o 1.
- **Sufijo:** número de 4 dígitos.
- **Contraseña:** en blanco o valor alfanumérico de 6 caracteres.
- **Ordenes:** “comprobar”, “depositar”, “pagar factura”, etc.

Ejemplo Partición Equivalente

Condiciones de entrada

Código de área: número de 3 dígitos o no es número

lógica: el código puede ser un número o no.

valor: número de 3 dígitos.

Prefijo: número de 3 dígitos que no comience por 0 o 1.

rango: valores entre 200 y 999.

Sufijo: número de 4 dígitos.

valor: número de 4 dígitos.

Contraseña: en blanco o valor alfanumérico de 6 caracteres.

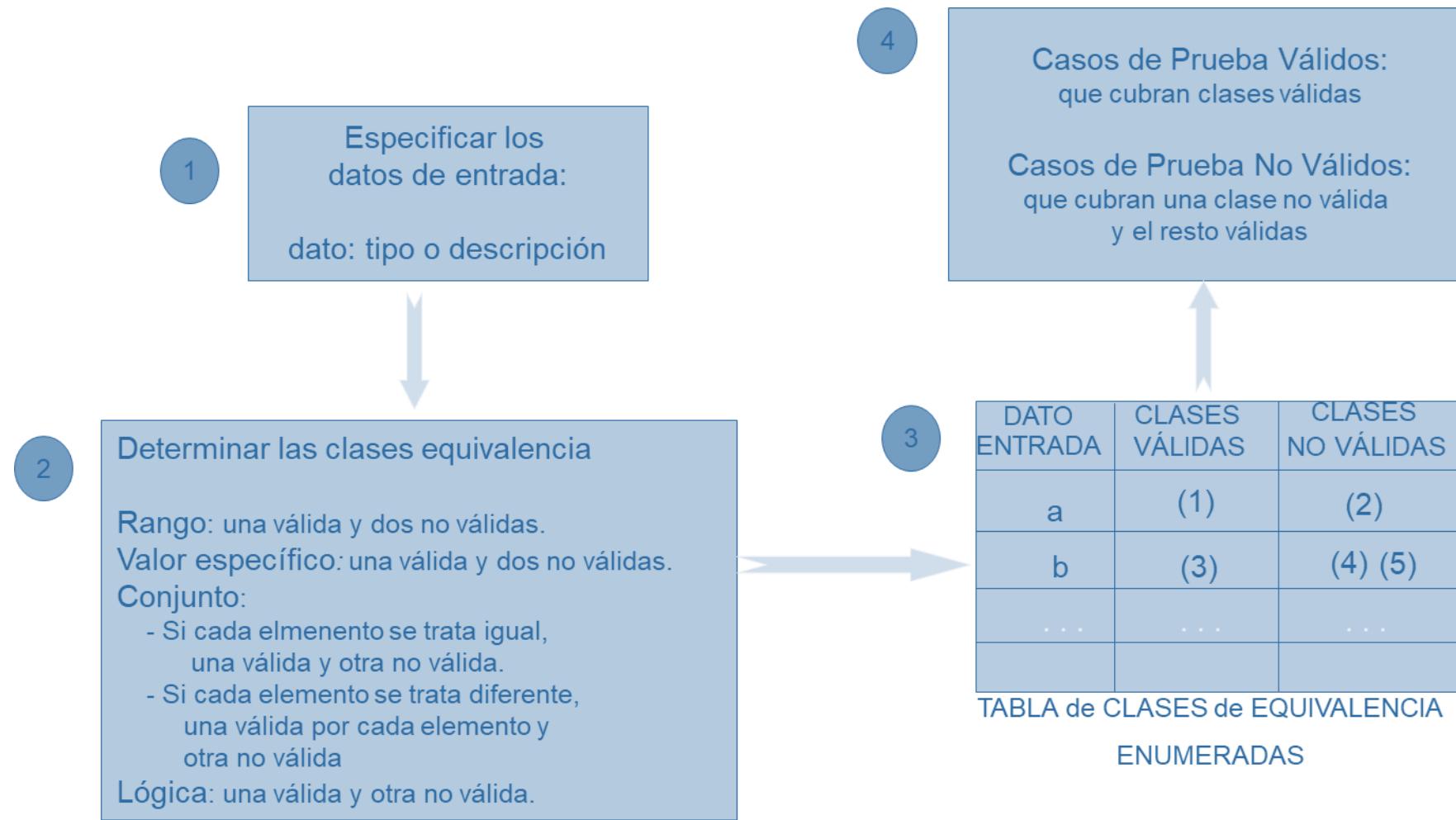
lógica: la contraseña puede estar presente o no.

valor: cadena de 6 caracteres.

Ordenes: “comprobar”, “depositar”, “pagar factura”, etc.

conjunto: el de todas las ordenes posibles.

Esquema prueba de la partición equivalente



Ejemplo Partición Equivalente

Aplicación bancaria. Datos de entrada:

Código de área: número de 3 dígitos que no empieza por 0 ni por 1

Nombre de identificación de operación: 6 caracteres

Órdenes posibles: “cheque”, “depósito”, “pago factura”, “retirada de fondos”

Ejemplo Partición Equivalente

Determinar las clases de equivalencia

Código de área:

Lógica:

1 clase válida: número

Rango:

1 clase válida: $200 < \text{código} < 999$

2 clases no válidas: $\text{código} < 200$; $\text{código} > 999$

1 clase no válida: no es número

Nombre de identificación:

Valor específico:

1 clase válida: 6 caracteres

2 clases no válidas: más de 6 caracteres; menos de 6
caracteres

Órdenes posibles:

Conjunto de valores:

1 clase válida: 4 órdenes válidas

1 clase no válida: orden no válida

Ejemplo Partición Equivalente : Tabla de clases de equivalencia

Datos de Entrada	Clases Válidas	Clases No Válidas
Código de área	(1) $200 \leq \text{código} \leq 999$	(2) código < 200 (3) código > 999 (4) no es numérico
Identificación	(5) 6 caracteres	(6) menos de 6 caracteres (7) más de 6 caracteres
Orden	(8) “cheque” (9) “depósito” (10) “ pago factura” (11) “retirada de fondos”	(12) ninguna orden válida

Ejemplo Partición Equivalente : Casos de Prueba Válidos

Código	Identificación	Orden	Clases Cubiertas
300	Nómina	“Depósito”	(1) ^C (5) ^C (9) ^C
400	Viajes	“Cheque”	(1) (5) (8) ^C
500	Coches	“Pago-factura”	(1) (5) (10) ^C
600	Comida	“Retirada-fondos”	(1) (5) (11) ^C

Ejemplo Partición Equivalente: Casos de Prueba NO Válidos

Código	Identificación	Orden	Clases Cubiertas
180	Viajes	“Pago-factura”	(2) ^C (5) (10)
1032	Nómina	“Depósito”	(3) ^C (5) (9)
XY	Compra	“Retirada-fondos”	(4) ^C (5) (11)
350	A	“Depósito”	(1) (6) ^C (9)
450	Regalos	“Cheque”	(1) (7) ^C (8)
550	Casa	&%4	(1) (5) (12) ^C

Análisis de valores límite

La técnica de Análisis de Valores Límites selecciona casos de prueba que ejerciten los valores límite dada la tendencia de la aglomeración de errores en los extremos.

Complementa la de la partición equivalente. En lugar de realizar la prueba con cualquier elemento de la partición equivalente, se escogen los valores en los bordes de la clase.

Se derivan tanto casos de prueba a partir de las condiciones de entrada como con las de salida.

Análisis de valores límite

Directrices:

Si una condición de entrada especifica un rango delimitado por los valores a y b, se deben diseñar casos de prueba para los valores a y b y para los valores justo por debajo y justo por encima de ambos.

Si la condición de entrada especifica un número de valores, se deben desarrollar casos de prueba que ejerciten los valores máximo y mínimo además de los valores justo encima y debajo de aquéllos.

Análisis de valores límite

... Directrices:

Aplicar las directrices anteriores a las condiciones de salida. Componer casos de prueba que produzcan salidas en sus valores máximo y mínimo.

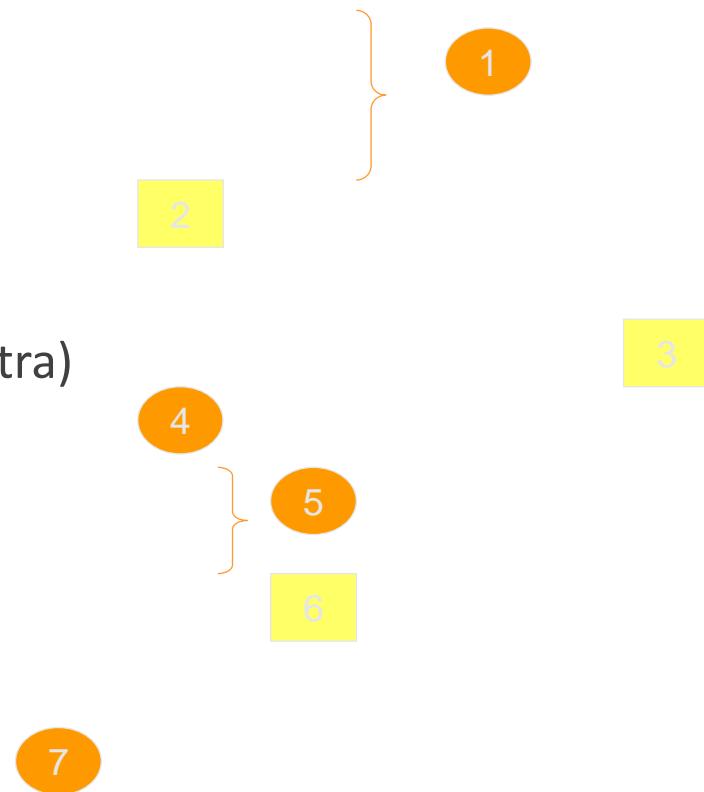
Si las estructuras de datos definidas internamente tienen límites prefijados (p.e., un array de 10 entradas), se debe asegurar diseñar un caso de prueba que ejercite la estructura de datos en sus límites.

Ejemplo Camino Básico: Código Fuente

```
int contar_letra(char cadena[10], char letra)
{
    int contador, n, lon;
    n=0; contador=0;
    lon = strlen (cadena);
    if (lon > 0) {
        do {
            if (cadena[contador] == letra) n++;
            contador++;
            lon--;
        } while (lon > 0);
    }
    return n;
}
```

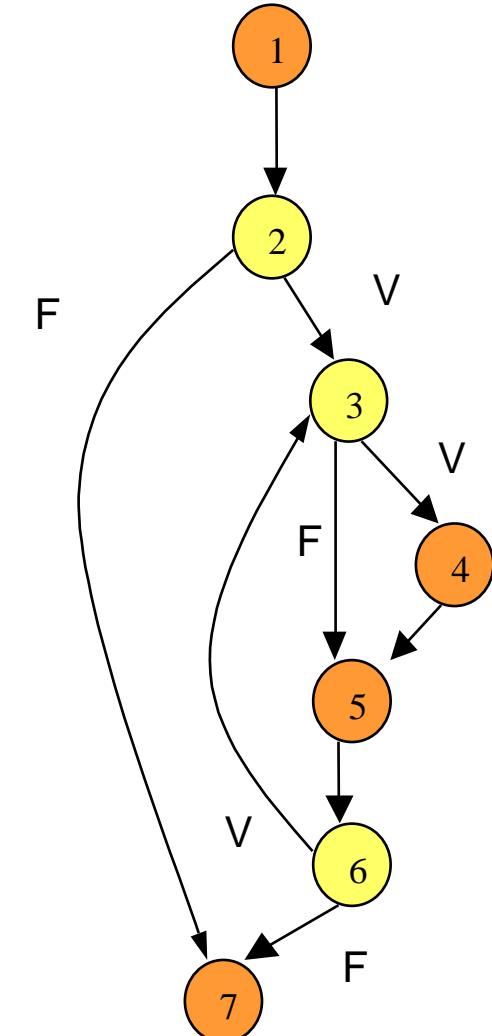
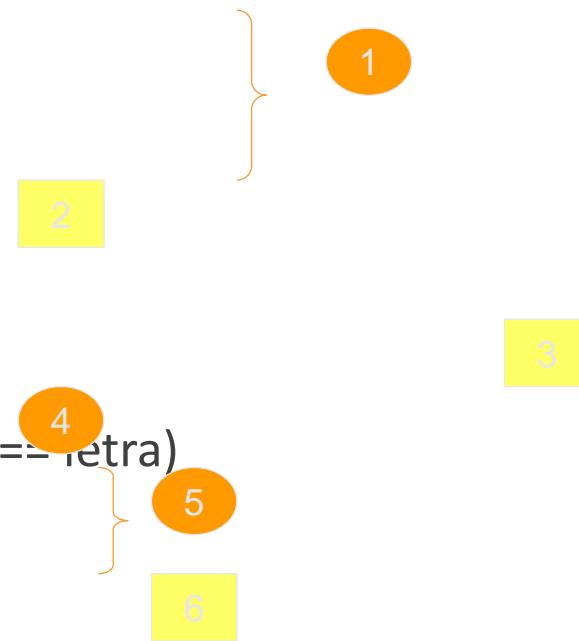
Ejemplo Camino Básico: Código Fuente Etiquetado

```
int contar_letra(char cadena[10], char letra)
{
    int contador, n, lon;
    n=0; contador=0;
    lon = strlen (cadena);
    if (lon > 0) {
        do {
            if (cadena[contador] == letra)
                n++;
            contador++;
            lon--;
        } while (lon > 0);
    }
    return n;
}
```



Ejemplo Camino Básico: Grafo de Flujo

```
int contar_letra(char cadena[10], char letra)
{
    int contador, n, lon;
    n=0; contador=0;
    lon = strlen (cadena);
    if (lon > 0) {
        do {
            if (cadena[contador] == letra)
                n++;
            contador++;
            lon--;
        } while (lon > 0);
    }
    return n;
}
```



Ejemplo Camino Básico: Caminos Independientes

$$V(G) = 4;$$

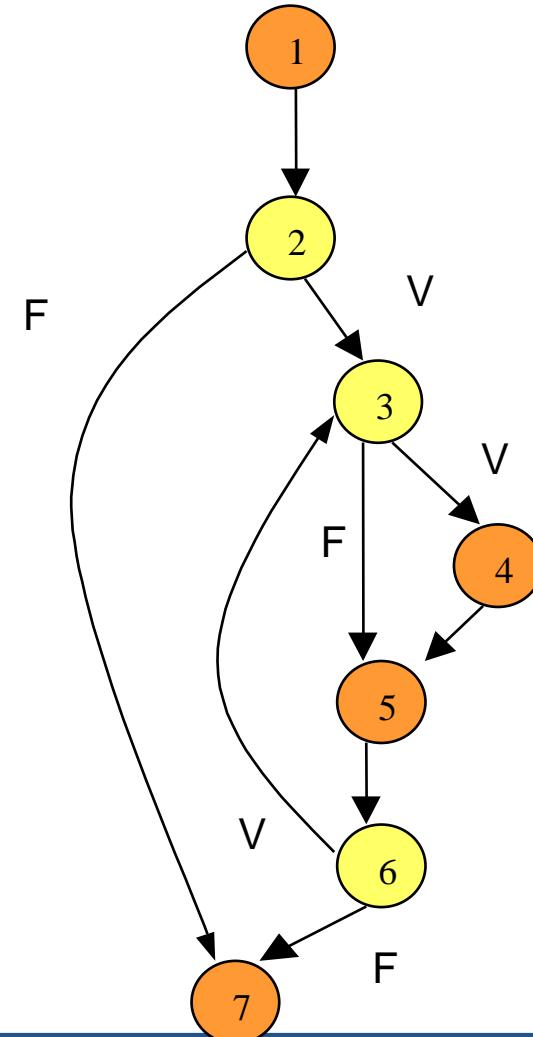
Nodos=7; Aristas=9;

Nodos Predicado=3;

Regiones = 4

Conjunto de caminos independientes:

1. 1-2-7
2. 1-2-3-4-5-6-7
3. 1-2-3-5-6-7
4. 1-2-3-4-5-6-3-5-6-7 (No es el único)



Ejemplo Camino Básico: Casos de Prueba

1.	1-2-7	cadena = ""	letra = 'a'	n = 0;
2.	1-2-3-4-5-6-7	cadena = "a"	letra = 'a'	n = 1;
3.	1-2-3-5-6-7	cadena = "b"	letra = 'a'	n = 0;
4.	1-2-3-4-5-6-3-5-6-7	cadena = "ab"	letra = 'a'	n = 1;

Otros tipos de prueba

- Recorridos (“walkthroughs”).
- Pruebas de robustez (“robustness testing”)
- Pruebas de aguante (“stress”)
- Prestaciones (“performance testing”)
- Conformidad u Homologación (“conformance testing”)
- Interoperabilidad (“interoperability tesing”)
- Regresión (“regression testing”)
- Prueba de comparación

Herramientas automáticas de prueba

Analizadores estáticos. Estos sistemas de análisis de programas soportan pruebas de las sentencias que consideran más débiles dentro de un programa.

Auditores de código. Son filtros con el propósito de verificar que el código fuente cumple determinados criterios de calidad (dependerá fuertemente del lenguaje en cuestión).

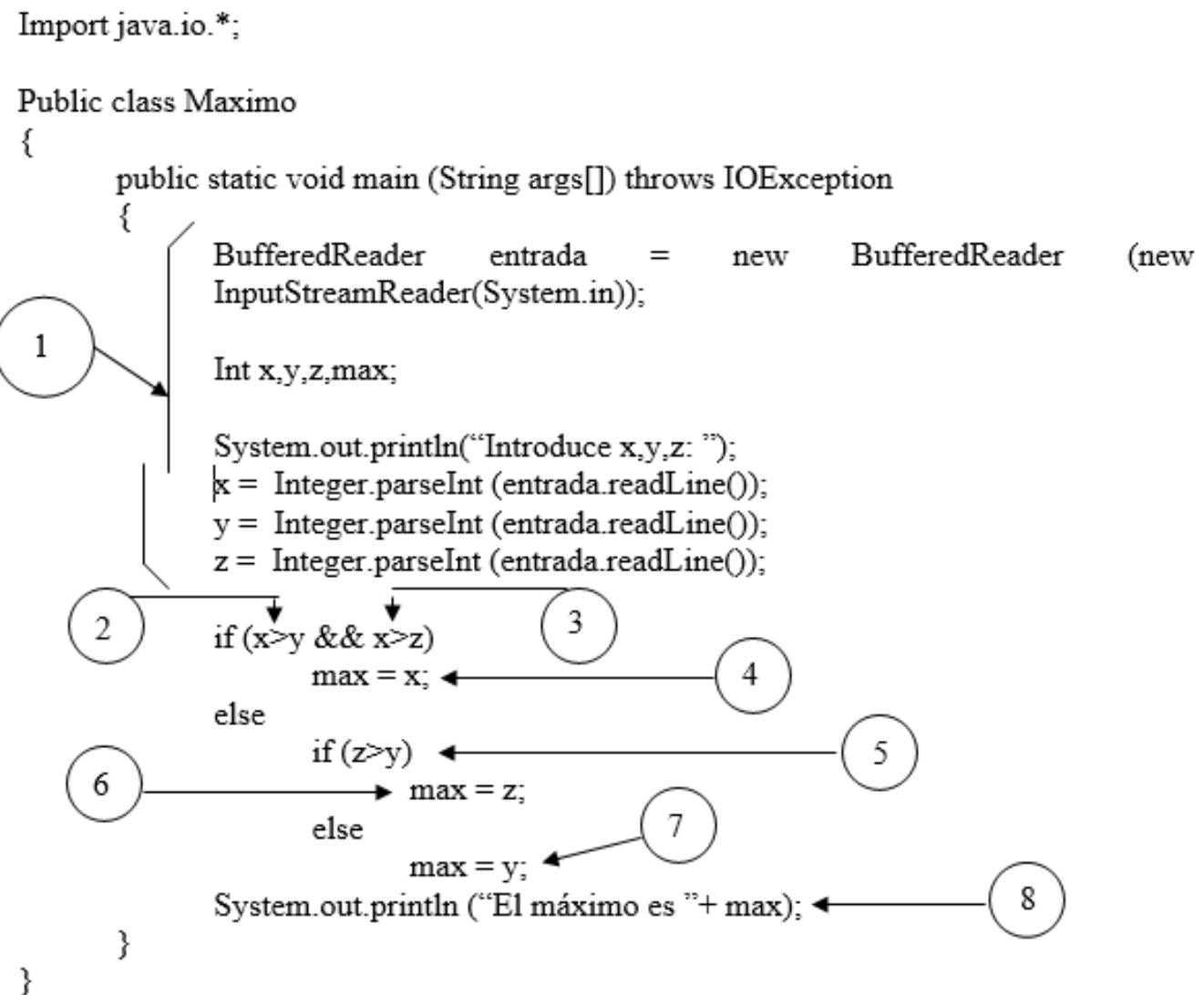
Generadores de archivos de prueba. Estos programas confeccionan automáticamente ficheros con datos que servirán de entrada a las aplicaciones.

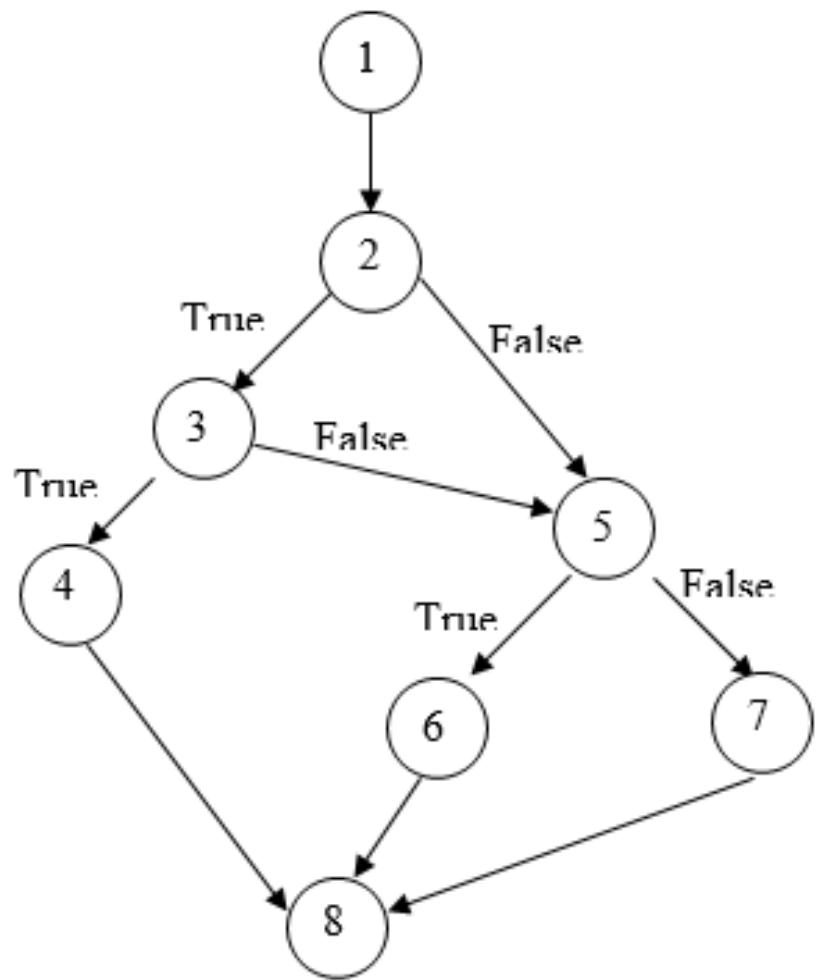
Generadores de datos de prueba. Confeccionan datos de entrada determinados que conlleven un comportamiento concreto del software.

Controladores de prueba. Confeccionan y alimentan los datos de entrada y simulan el comportamiento de otros módulos para restringir el alcance de la prueba.

Preguntas ?


```
Import java.io.*;  
  
Public class Maximo  
{  
    public static void main (String args[]) throws IOException  
    {  
        BufferedReader entrada = new BufferedReader (new  
InputStreamReader(System.in));  
  
        Int x,y,z,max;  
  
        System.out.println("Introduce x,y,z: ");  
        x = Integer.parseInt (entrada.readLine());  
        y = Integer.parseInt (entrada.readLine());  
        z = Integer.parseInt (entrada.readLine());  
  
        if (x>y && x>z)  
            max = x;  
        else  
            if (z>y)  
                max = z;  
            else  
                max = y;  
        System.out.println ("El máximo es "+ max);  
    }  
}
```





Calculamos la complejidad ciclomática de McCabe :

$$V(G) = a - n + 2 = 10 - 8 + 2 = 4$$

$$V(G) = r = 4$$

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

Por lo tanto tendremos cuatro caminos independientes, que mirando el grafo de flujo deducimos serán los siguientes:

- Camino 1 **➔** 1 - 2 - 3 - 4 - 8
- Camino 2 **➔** 1 - 2 - 3 - 5 - 6 - 8
- Camino 3 **➔** 1 - 2 - 5 - 6 - 8
- Camino 4 **➔** 1 - 2 - 5 - 7 - 8

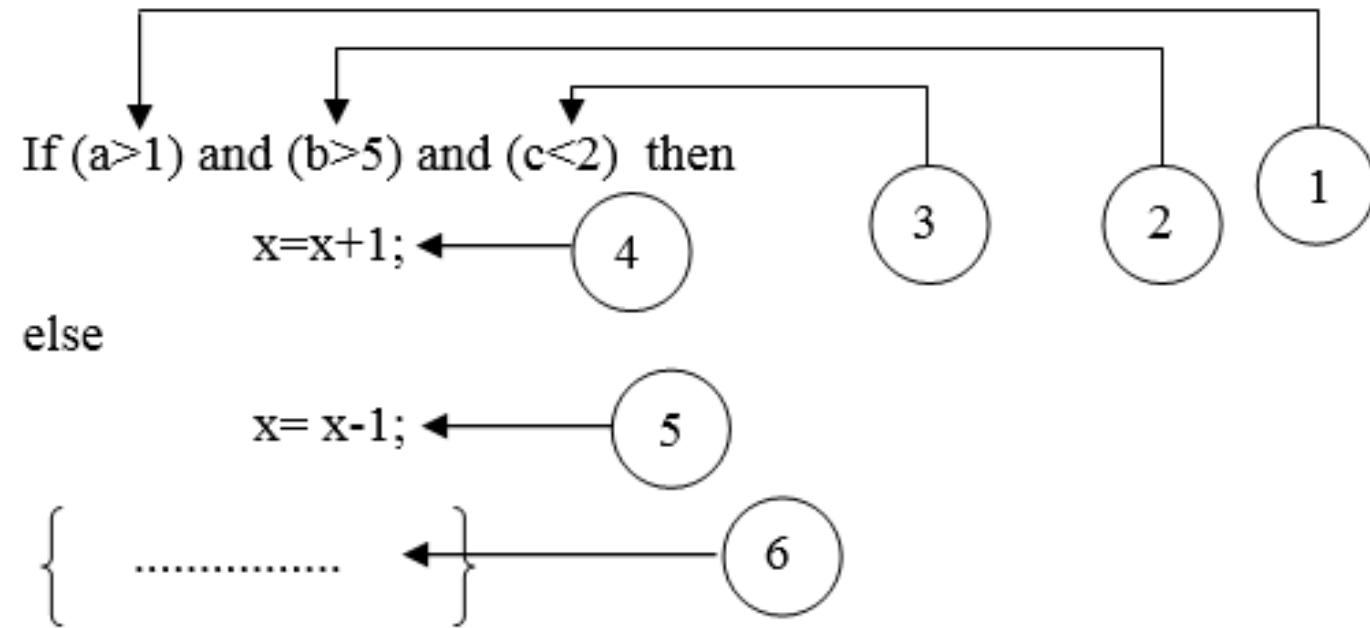
Camino	Características	Caso de Prueba		
		x	y	z
Camino 1	$x > y$, $x > z$	10	3	3
Camino 2	$y < x < z$	5	2	10
Camino 3	$x < y < z$	2	5	8
Camino 4	$x < y$, $z < y$	5	10	5

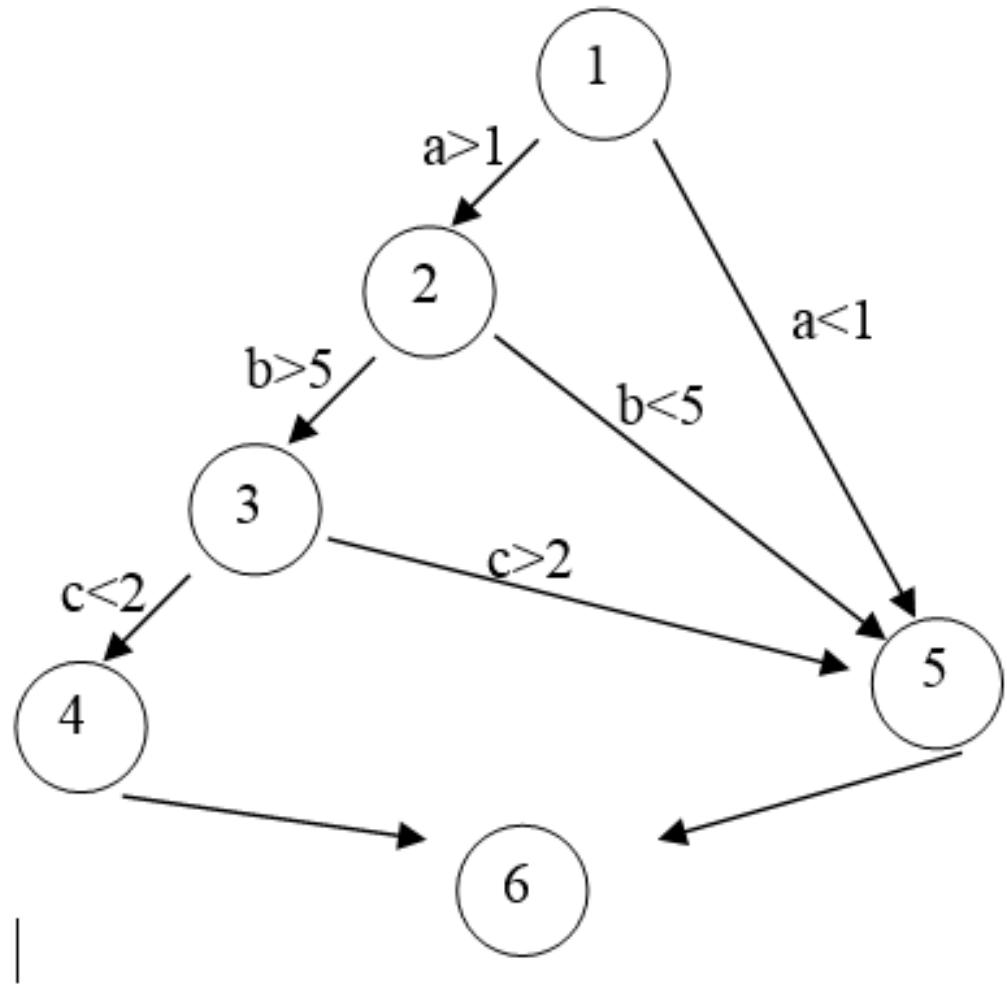
If ($a > 1$) and ($b > 5$) and ($c < 2$) then

$x = x + 1;$

else

$x = x - 1;$





$V(G) = 8 - 6 + 2 = 4$ caminos independientes

Camino 1 → 1 – 5 – 6

Camino 2 → 1 – 2 – 5 – 6

Camino 3 → 1 – 2 – 3 – 5 – 6

Camino 4 → 1 – 2 – 3 – 4 – 6

Con el caso de prueba {a=0, b=11, c=1} ejecutamos el primer camino independiente.

Con el caso de prueba {a=4, b=4, c=4} ejecutamos el segundo camino independiente.

Con el caso de prueba {a=2, b=6, c=0} ejecutamos el cuarto camino independiente.

Se desean realizar pruebas de la caja negra sobre un programa utilizado por una empresa de transporte para calcular la tarifa de cada billete según el trayecto, la antelación en la que se obtiene el billete y la edad del pasajero. Dicha empresa sólo opera viajes entre Santander, Madrid y Barcelona.

Como datos de entrada toma:

- **CiudadOrigen** que es un campo que puede tomar los valores “SNT”, “MAD” y “BCN”.
- **CiudadDestino** que puede tomar los mismos valores “SNT”, “MAD” y “BCN”.
- **Fecha** es un campo del tipo fecha que indica el día en el que se pretende realizar el viaje.
- **Edad** es un campo numérico positivo de 3 cifras (incluyendo el 000).

La tarifa obtenida además de estar en función del trayecto realizado, ofrece los siguientes descuentos por antelación y edad del pasajero. Los descuentos no son acumulables y siempre se aplicará el de mayor valor.

- **15%** de descuento sacando el billete con antelación superior a 1 semana y **25%** con antelación superior a 1 mes.
- **30%** a los pasajeros con edad inferior a 25 años y **40%** a los pasajeros con edad superior a 65 años.

Se pide:

- 1.1.** Realizar una tabla con las clases de equivalencia indicando las clases válidas y no válidas para cada variable de entrada.
- 1.2.** Obtener casos de prueba de dicha tabla, indicando las clases de equivalencia que cubriría cada caso (numerar previamente las clases).

Un programa toma como entrada un fichero cuyo formato de registro es el siguiente:

- **Número-empleado** es un campo de números enteros positivos de 3 dígitos (excluido el 000).
- **Nombre-empleado** es un campo alfanumérico de 10 caracteres.
- **Meses-Trabajo** es un campo que indica el número de meses que lleva trabajando el empleado; es un entero positivo (incluye el 000) de 3 dígitos.
- **Directivo** es un campo de un solo carácter que puede ser «+» para indicar que el empleado es un directivo y «-» para indicar que no lo es.

Numero-empleado	Nombre-empleado	Meses-Trabajo	Directivo
123	JOSE MIGUEL	000	-



UNMSM

Universidad Nacional Mayor de San Marcos
Universidad del Perú. Decana de América.



DISEÑO DE SOFTWARE

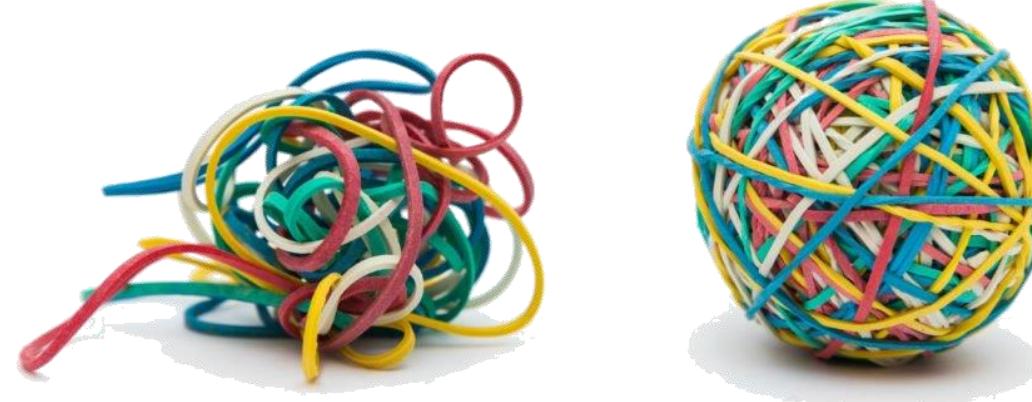
Refactorización - Antipatrones

Agenda

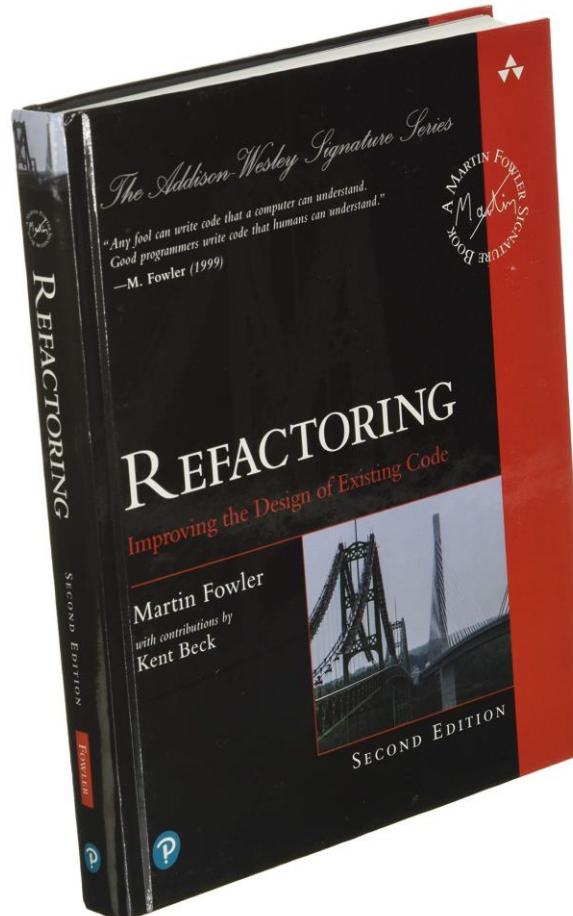
1. Refactoring
2. Anti-patrones

Refactorización

Refactoring es modificar el comportamiento interno sin modificar su **comportamiento externo**.



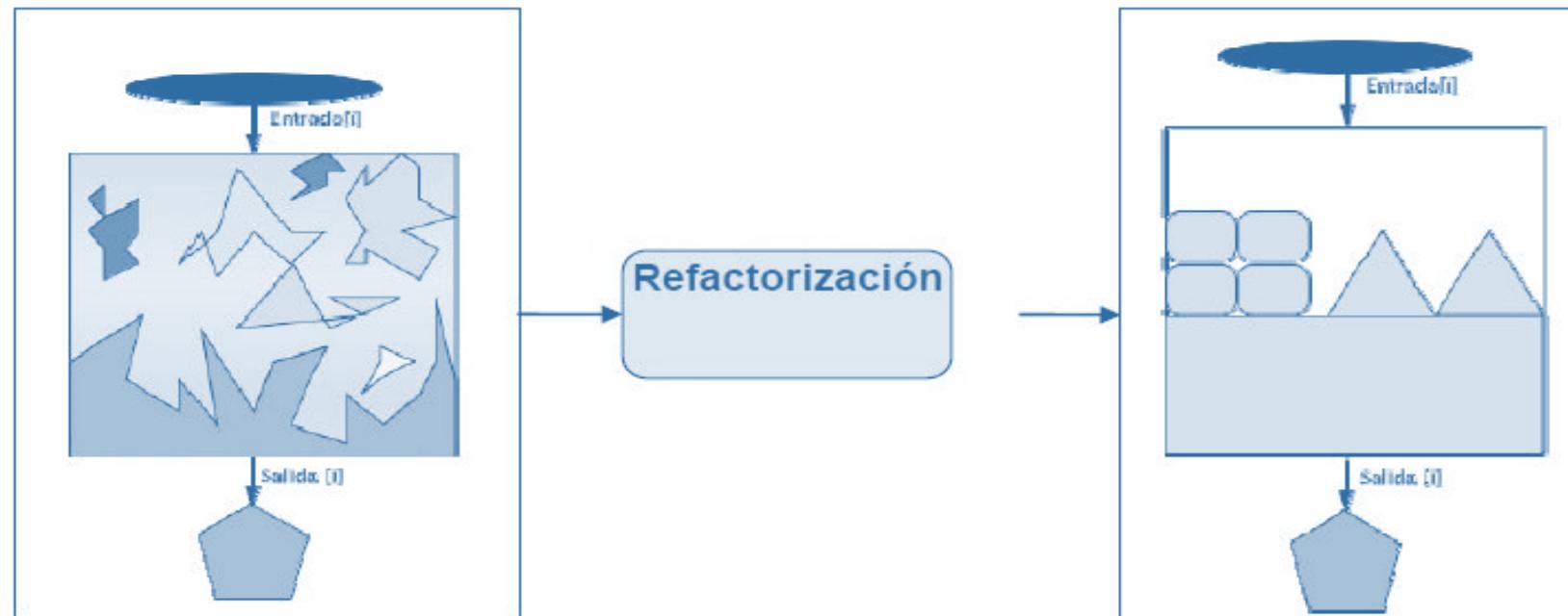
Refactorización



El término se creó como analogía con la factorización de números y polinomios.

El libro de Martin Fowler Refactoring es la referencia clásica (1999).

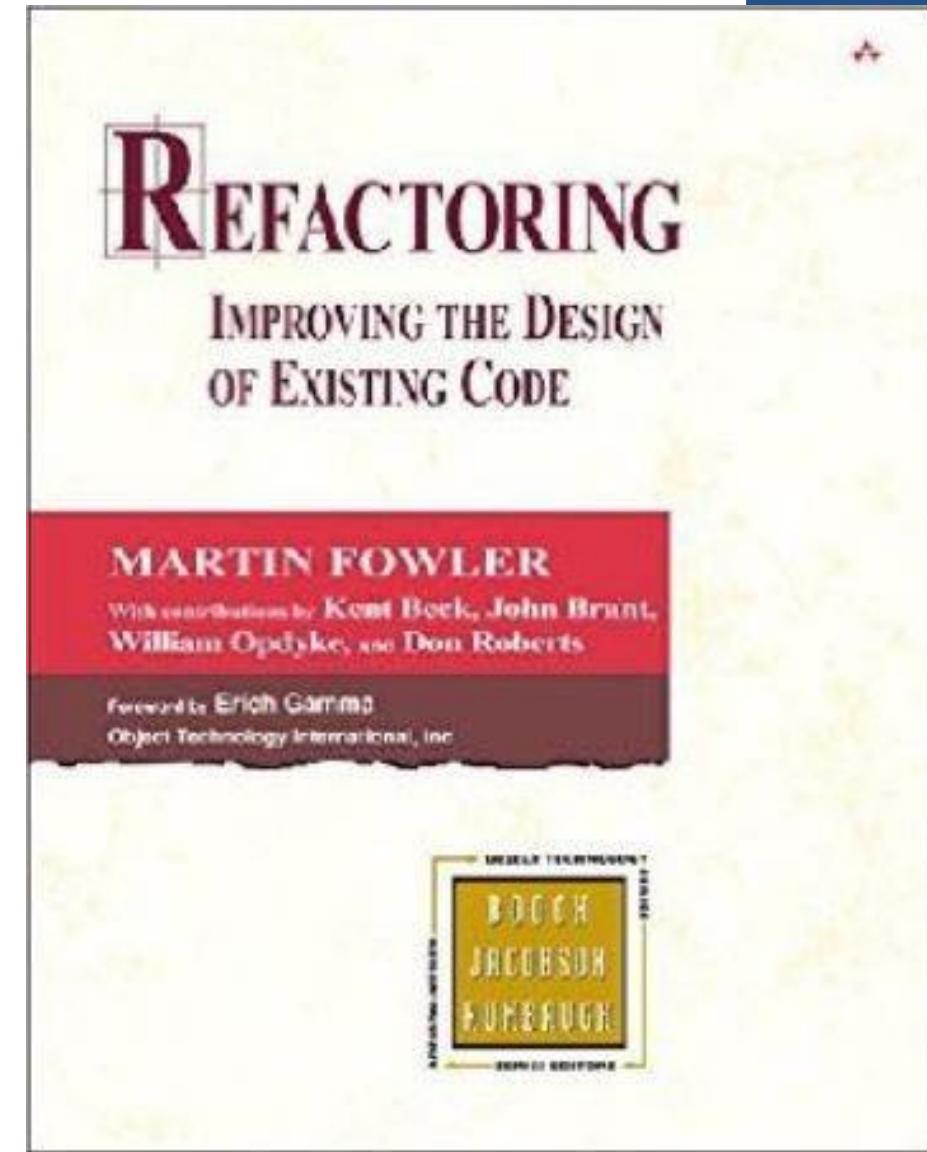
Refactoring



Refactorización: antes y después

Refactoring

- Pequeños cambios en el software que cambian su estructura interna sin modificar su comportamiento externo –
Martin Fowler



Por qué refactorizar?

- Para mejorar su diseño
- Para hacerlo más entendible
- Para encontrar errores
- Para programar más rápido

Cuando se debe refactorizar?

Cuando se está escribiendo un nuevo código

- Para hacerlo más entendible
- Para simplificar la implementación de las nuevas funcionalidades

Cuando se corrige un error

Cuando se revisa el código

Proceso de refactorizar

- Previamente se debe considerar contar con un conjunto **casos de prueba** que sean automáticos, auto verificables e Independientes.

Pasos:

1. Ejecutar las pruebas antes de cualquier cambio
2. Analizar los cambios a realizar
3. Aplicación del cambio
4. Volver a ejecutar las pruebas

Beneficios de refactoring

- facilita la comprensión del código fuente.
- permite detectar errores
- permite programar mas rápido
- mejora del diseño de nuestro software.
- Incremento de facilidad de lectura y comprensión del código fuente - auto-documentable

Desventajas de refactorizar

- Cambio en base de datos
- Cambio en Interfaces

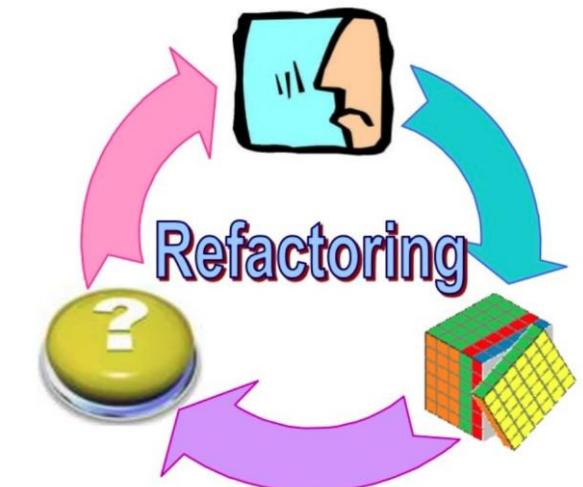
Síntomas de la necesidad de refactorizar

- 1. Duplicated code**
- 2. Long method**
- 3. Large class**
- 4. Long parameter list**
- 5. Divergent change**
- 6. Shotgun surgery**
- 7. Feature envy**
- 8. Data class**
- 9. Refused bequest**



Momentos para refactorizar

- Regla de los tres strikes
- Al momento de agregar funcionalidad
- Al momento de resolver una falla
- Al momento de realizar una revisión de código



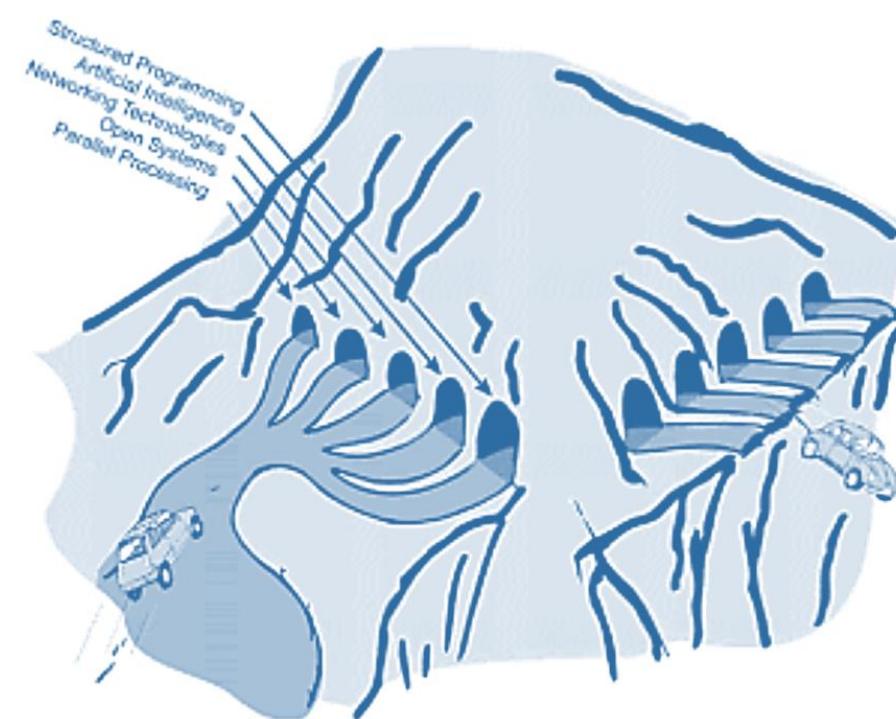
Momentos para no refactorizar

- Código simplemente no funciona
- Esfuerzo necesario demasiado grande
- Próximo a una entrega



Antipatrones (Anti-patterns)

“Un antipatrón es una forma literaria que describe una solución recurrente que genera consecuencias negativas” (**Brown et al, 1998**)

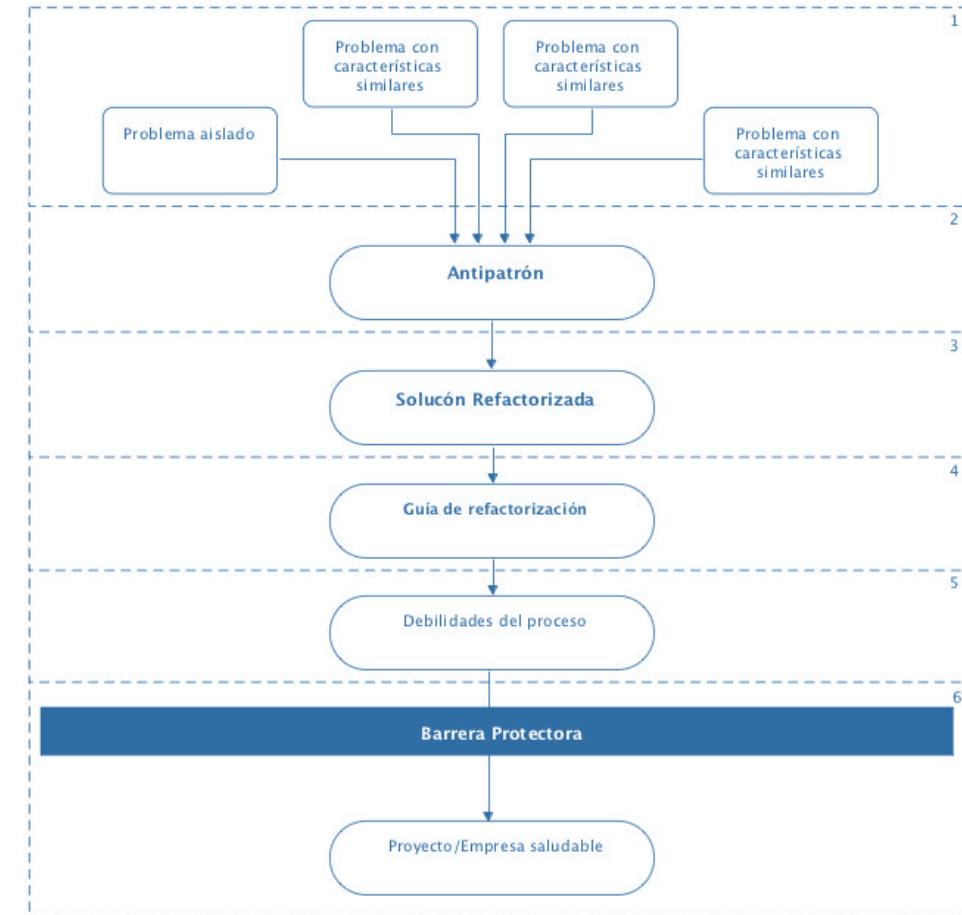


Relación con patrones de diseño y refactorizaciones

- Ambos proveen un vocabulario común
- Ambos documentan conocimiento
- Los patrones de diseño documentan **soluciones exitosas**, los antipatrones documentan **soluciones problemáticas**
- Los antipatrones son el lado oscuro de los patrones de diseño
 - No se evalúa que tan aplicables

Proceso para el uso de antipatrones

1. **Encontrar** el problema
2. **Establecer** un patrón de fallas
3. **Refactorizar** el código
4. **Publicar** la solución
5. **Identificar** debilidades, o posibles problemas del proceso.
6. **Corregir** el proceso

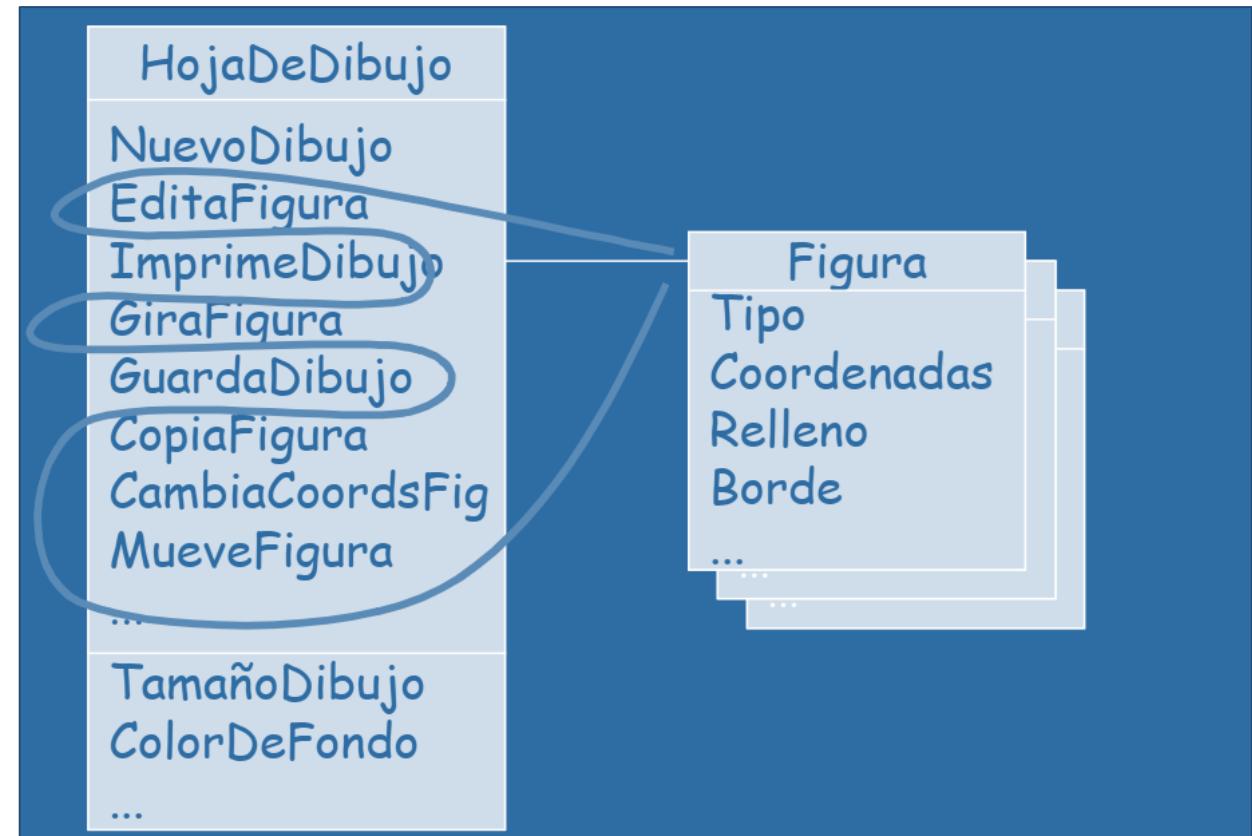


Categorías

Desarrollo de Software	Arquitectura	Gestión
<ul style="list-style-type: none">• The Blob• Lava Flow• Functional Decomposition• Poltergeists• Golden Hammer• Spaghetti Code• Cut and Paste Programming <p><i>Miní antipatterns</i></p> <ul style="list-style-type: none">• Continuous Obsolescence• Ambiguous Viewpoint• Boat Anchor• Dead End• Input Kludge• Walking through a Minefield• Mushroom Management	<ul style="list-style-type: none">• Stovepipe Enterprise• Vendor Lock-in• Architecture by Implication• Design by Committee• Reinvent the Wheel <p><i>Miní antipatterns</i></p> <ul style="list-style-type: none">• Autogenerated Stovepipe• Jumble• Cover your Assets• Wolf Ticket• Warm Bodies• Swiss Army Knife• The Grand Old Duke of York	<ul style="list-style-type: none">• Analysis Paralysis• Death by Planning• Corncob• Irrational Management• Project Missmanagement <p><i>Miní antipatterns</i></p> <ul style="list-style-type: none">• Blowhard Jamboree• Viewgraph Engineering• Fear of Success• Intellectual Violence• Smoke and Mirrors• Throw it over the wall• Fire Drill• The Feud• E-Mail is Dangerous

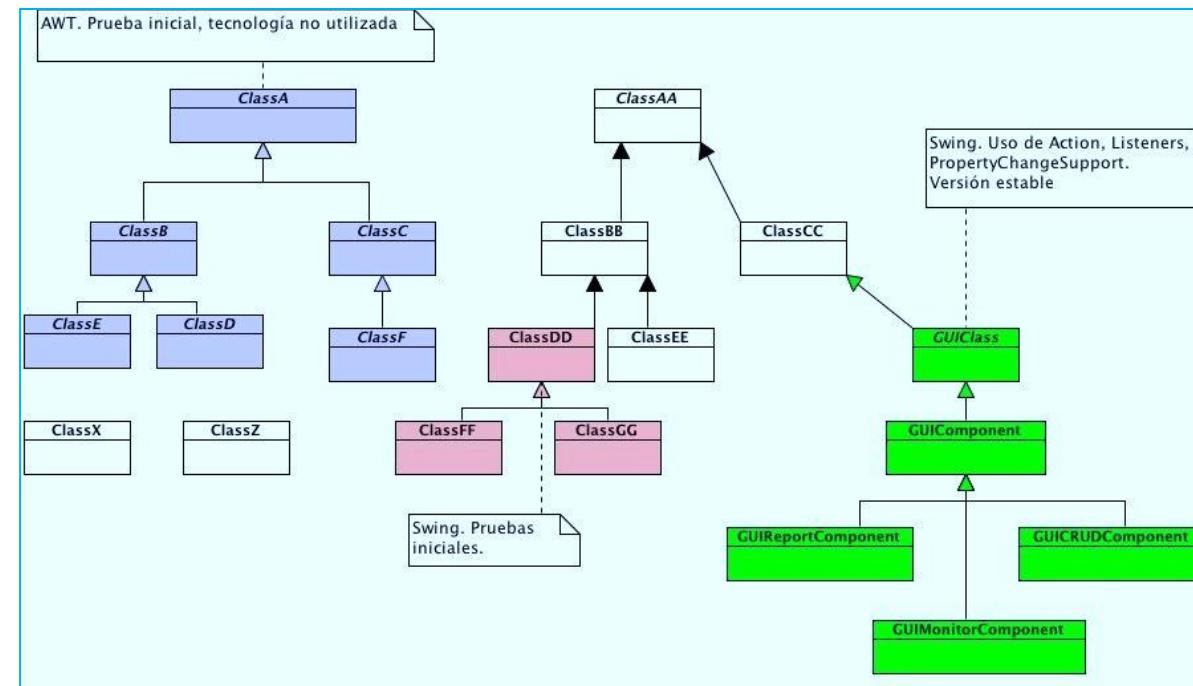
Catálogo de antipatrones de desarrollo

1. **The Blob:** God Class o Winnebago es una **clase**, o componente, que **conoce o hace demasiado**



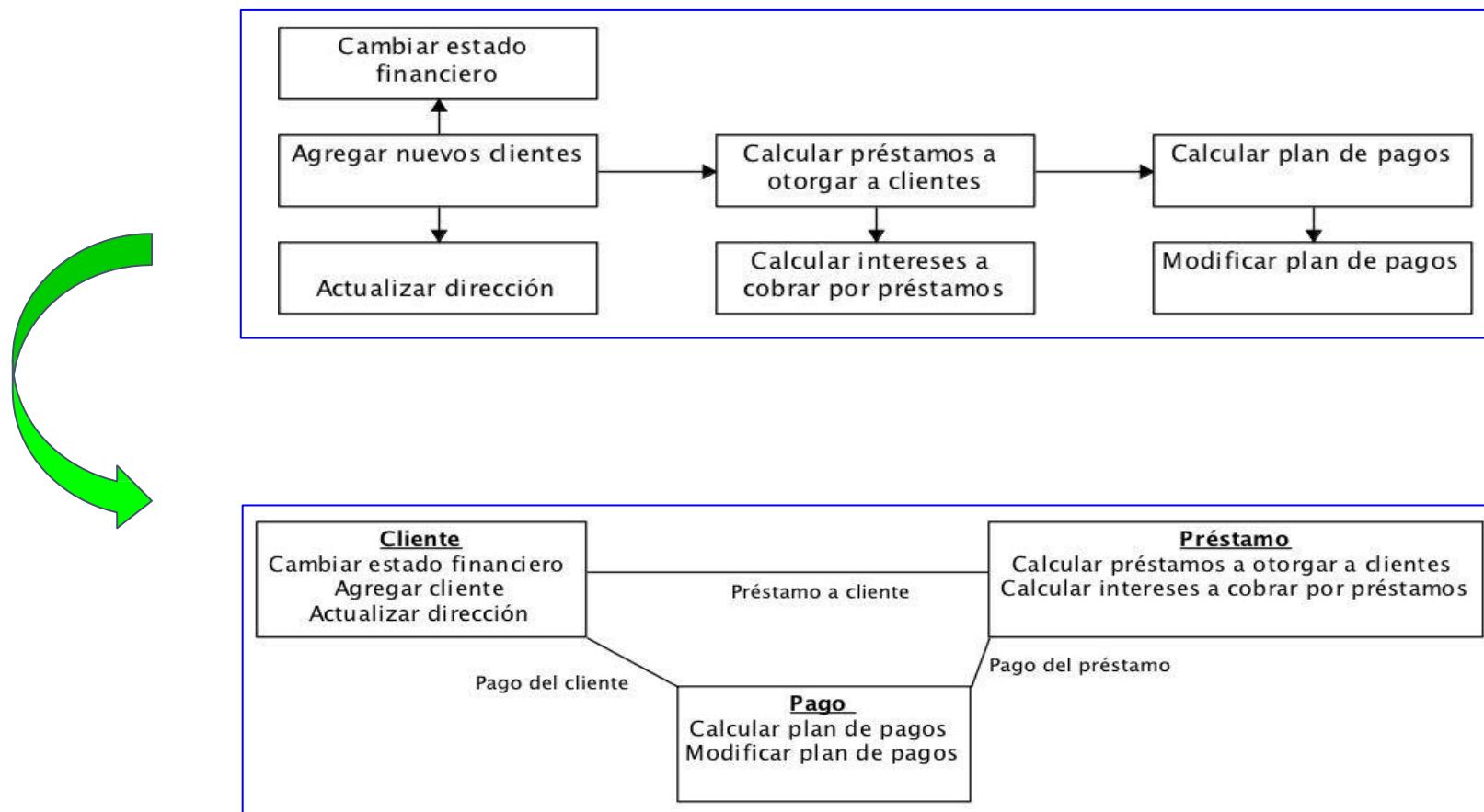
antipatrones de desarrollo

2. Lava Flow: Dead Code aparece principalmente en aquellos sistemas que comenzaron como investigación o pruebas de concepto y luego llegaron a producción.



antipatrones de desarrollo

3. Functional Decomposition: Traducen cada subrutina como una clase.



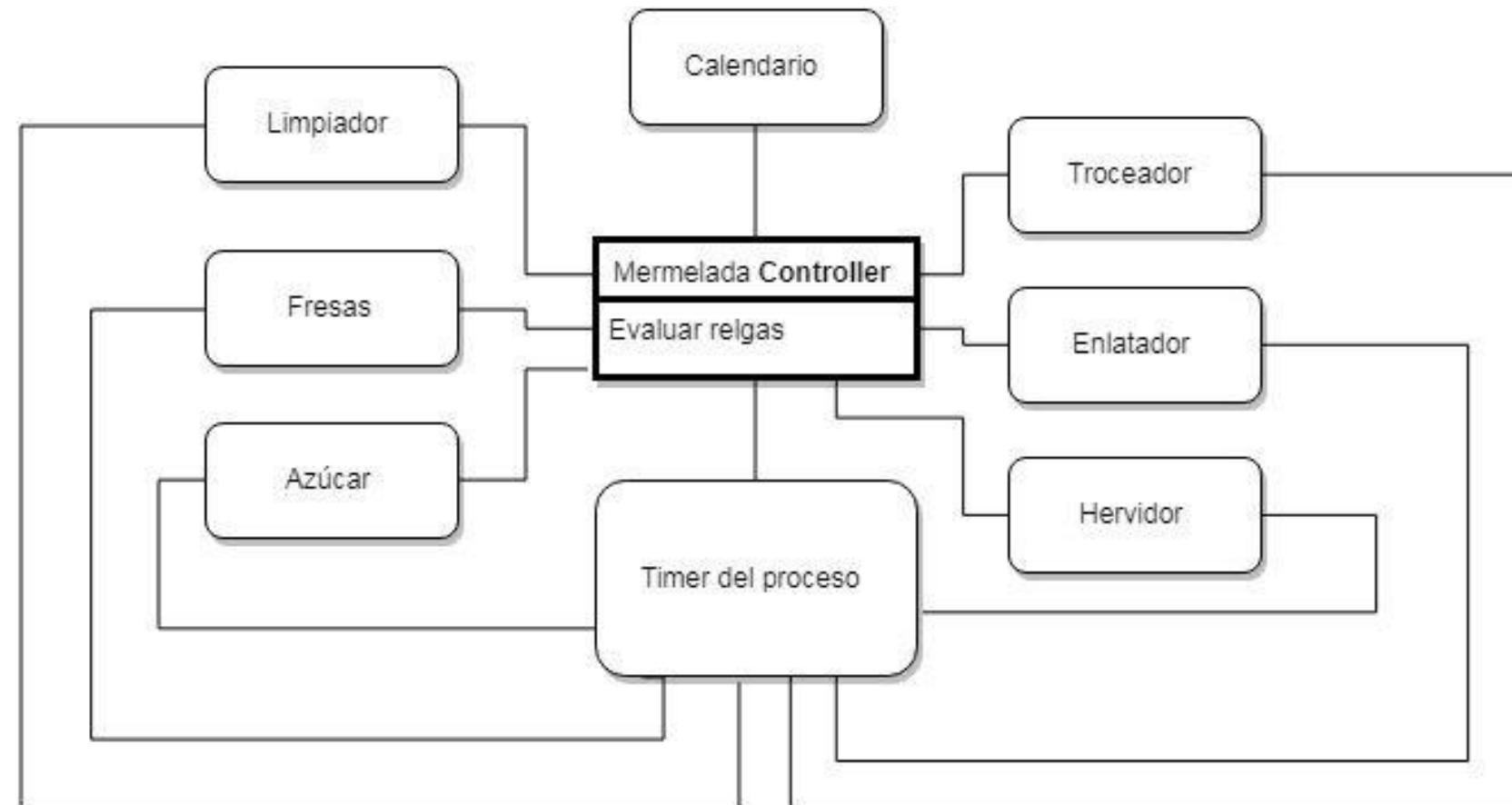
antipatrones de desarrollo

3. Poltergeists: Gipsy, o Proliferation of Classes, o Big Dolt Controller Class. Las clases fantasmas tienen pocas responsabilidades y un ciclo de vida breve. “**Aparecen**” solamente para iniciar algún método.

Son de relativa facilidad de encuentro ya que sus nombres suelen llevar el sufijo “**controller**” o “**manager**”.



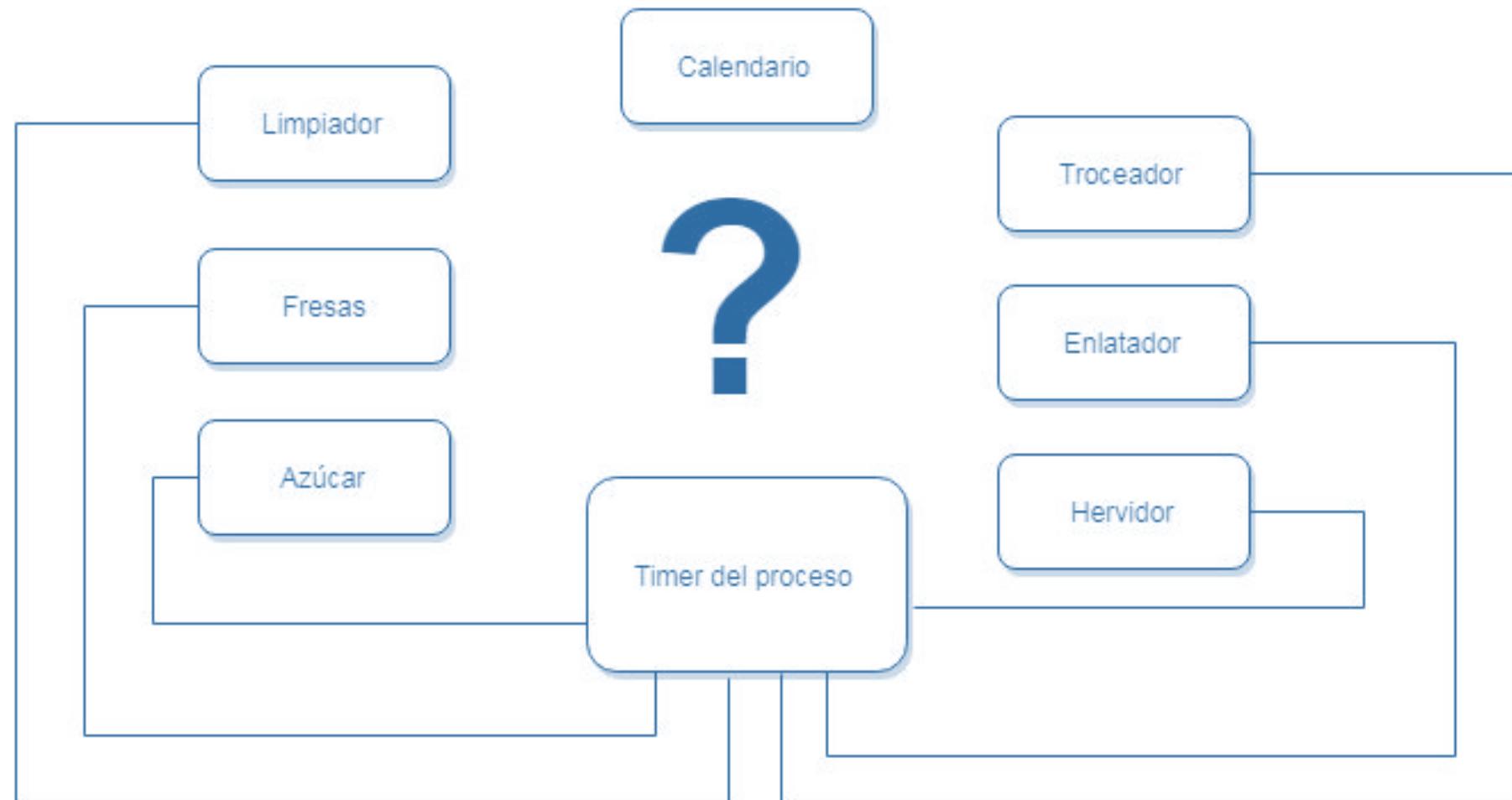
antipatrones de desarrollo



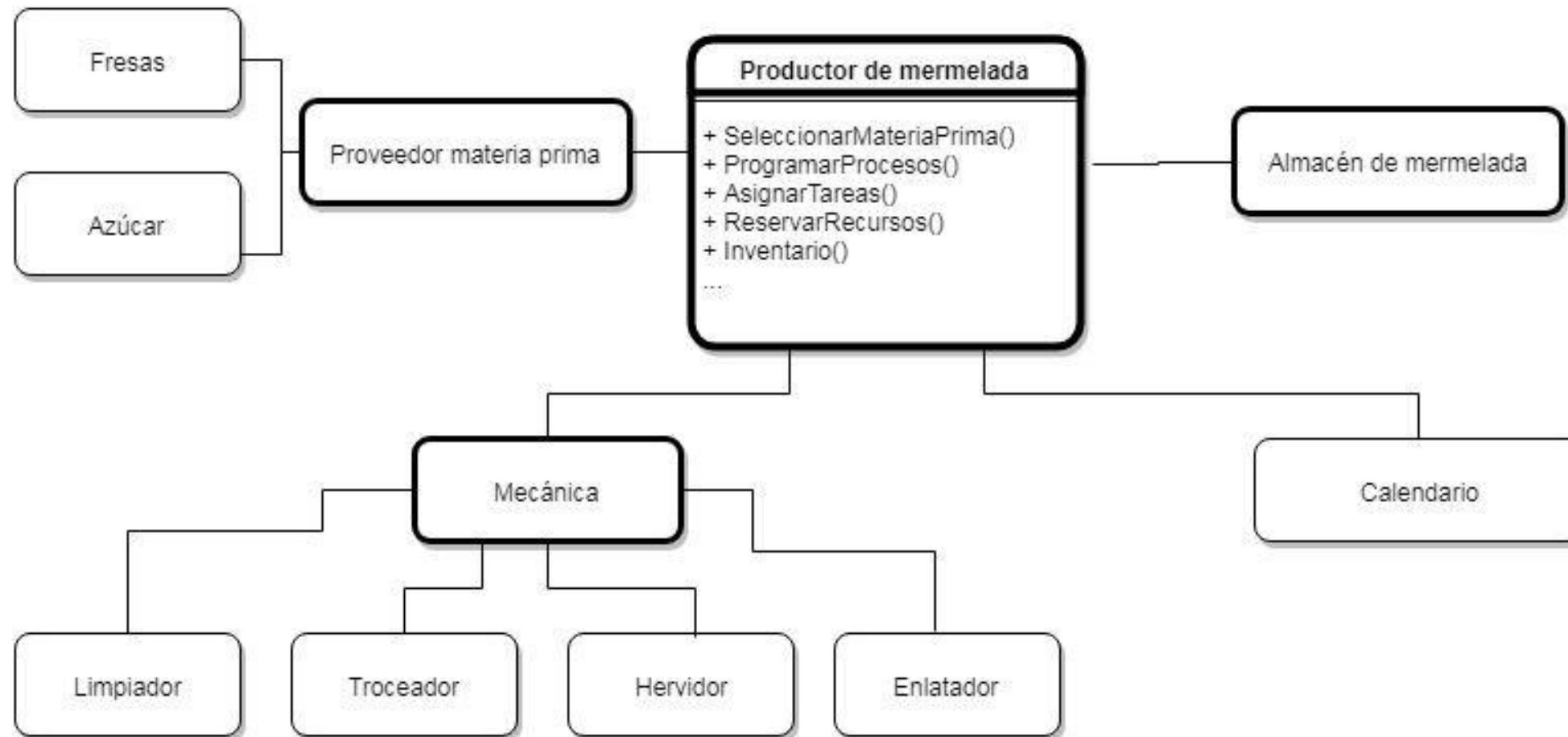
MermeladaController

- No tiene estado
- Sus instancias son temporales y aparecerán solamente para invocar a otras clases
- Todas sus asociaciones son transitorias
- Añade rutas de acceso a otras clases completamente redundantes (y innecesarias).

antipatrones de desarrollo



antipatrones de desarrollo



antipatrones de desarrollo

4. Copy-And-Paste Programming: más fácil modificar código preexistente que programar desde el comienzo.

```
abstract class Game {  
    /* Hook methods. Concrete implementation may differ in each subclass */  
    protected int playersCount;  
    abstract void initializeGame();  
    abstract void makePlay(int player);  
    abstract boolean endOfGame();  
    abstract void printWinner();  
  
    /* A template method : */  
    public final void playOneGame(int playersCount) {  
        this.playersCount = playersCount;  
        initializeGame();  
        int j = 0;  
        while (!endOfGame()) {  
            makePlay(j);  
            j = (j + 1) % playersCount;  
        }  
        printWinner();  
    }  
}
```

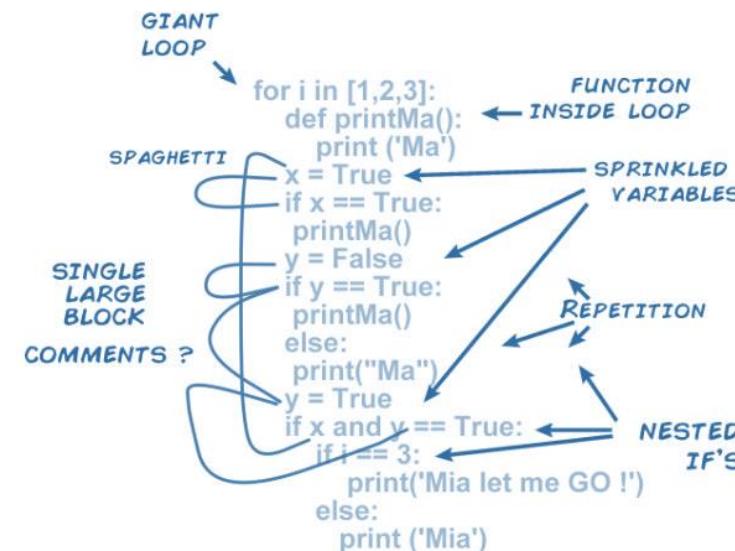
```
//Now we can extend this class in order  
//to implement actual games:  
  
class Monopoly extends Game {  
  
    /* Implementation of necessary concrete methods */  
    void initializeGame() {  
        // Initialize players  
        // Initialize money  
    }  
    void makePlay(int player) {  
        // Process one turn of player  
    }  
    boolean endofGame() {  
        // Return true if game is over  
        // according to Monopoly rules  
    }  
    void printWinner() {  
        // Display who won  
    }  
    /* Specific declarations for the Monopoly game. */  
    // ...  
}
```

antipatrones de desarrollo

5. **Spaghetti Code:** Sistema con poca estructura donde los cambios y **futuras extensiones** se tornan **difíciles** por haber perdido **claridad en el código**, incluso para el autor del mismo.



Un-structured code
is a liability



antipatrones de desarrollo

6. Golden Hammer: Old Yeller, o Head in the Sand. Un **martillo de oro** es cualquier herramienta, tecnología o paradigma que, según sus partidarios, es capaz de **resolver** diversos tipos de problemas, incluso aquellos para los cuales **no fue concebido**

- El lenguaje XML

Gracias.



DISEÑO DE SOFTWARE

Pruebas Unitarias

Sesión S13

© Nehil Muñoz C.

Agenda

- Pruebas Unitarias
- Prueba simulada
- TDD

Pruebas Unitarias

- Es un proceso de prueba donde se estudia de manera aislada un componente de SW, como por ejemplo un clase de Java
- Se centran en las pruebas de los programas / módulos generados por los desarrolladores.
 - Generalmente, son los propios desarrolladores los que prueban su código.
 - Diferentes técnicas: caja blanca y caja negra
 - Con caja blanca: medición de la cobertura, ramas, etc.



Pruebas Unitarias

Beneficios

- Permite arreglar los errores detectados **sin cambiar la funcionalidad** del componente probado
- Contribuye al proceso de **integración de componentes** del Sistema de SW al permitir asegurar el funcionamiento correcto de los componentes de **manera individual**

Limitaciones

- No permite identificar errores de **integración ni errores de desempeño** del Sistema de SW
- Son **imprácticas** para probar todos los casos posibles para los componentes que se prueban
- Su compatibilidad con los diferentes modelos de desarrollo de SW varía, siendo más compatible con el Desarrollo dirigido por pruebas.

Pruebas Unitarias

Para hacer pruebas unitarias se usan **frameworks** en entornos de pruebas automatizados

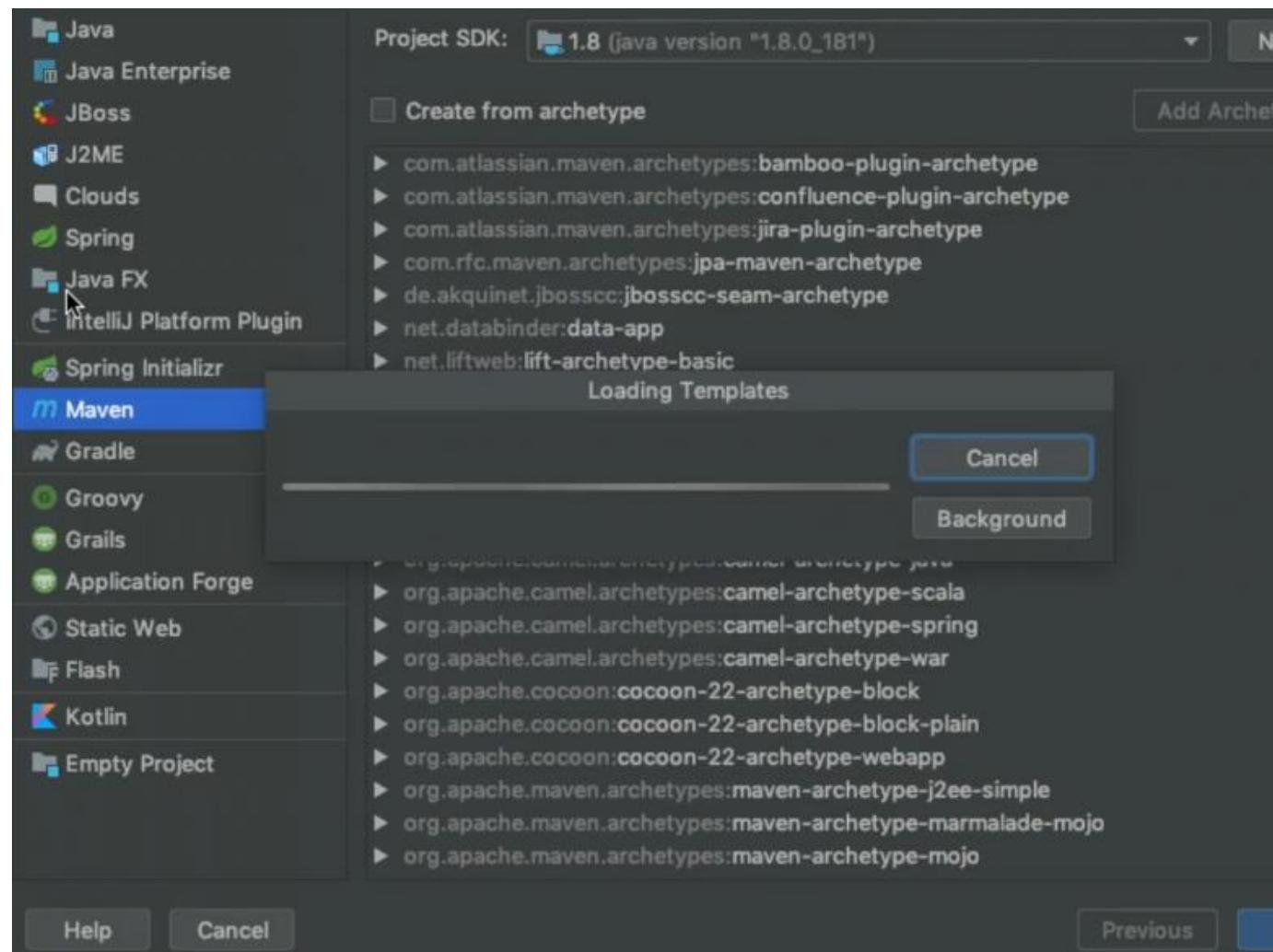
- Un framework es un conjunto de clases relacionadas que proveen un conjunto de funcionalidades para realizar una tarea específica
- En el caso de la realización de pruebas unitarias **xUnit** es el framework más usado para hacer pruebas unitarias automatizadas

- **JUnit:** xUnit para Java
- **VbUnit:** xUnit para Objetos Visual Basic
- **Cunit:** xUnit para lenguaje C
- **CPPUnit:** xUnit para lenguaje C++
- **csUnit, MbUnit, NUnit:** xUnit para lenguajes Microsoft .NET
- **DBUnit:** xUnit para proyectos con Bases de Datos
- **Dunit:** xUnit para Borland Delphi 4 y posteriores
- **PHPUnit:** xUnit para PHP
- **PyUnit:** xUnit para Python

Pruebas Unitarias

The image shows a screenshot of the IntelliJ IDEA website on the left and the IntelliJ IDEA interface on the right. The website features a large 'IntelliJ IDEA' logo with a stylized 'IJ' icon, followed by the text 'Capable and Ergonomic IDE for JVM'. It includes two buttons: 'DOWNLOAD' and 'TAKE A TOUR'. The IntelliJ IDEA interface on the right shows the splash screen with the 'IJ' logo and the text 'IntelliJ IDEA Version 2018.3.2'. Below this, there is a menu bar with options: '+ Create New Project' (highlighted with a mouse cursor), 'Import Project', 'Open', and 'Check out from Version Control'. At the bottom right of the interface, there are links for 'Configure' and 'Get Help'.

Pruebas Unitarias



MAVEN

- **Maven** es una herramienta de software para la gestión y construcción de proyectos **Java**
- Tiene un modelo de configuración de construcción simple, basado en un formato **XML**
- Maven utiliza un ***Project Object Model (POM)*** para describir el proyecto de software a construir, sus dependencias de otros módulos y componentes externos, y el orden de construcción de los elementos

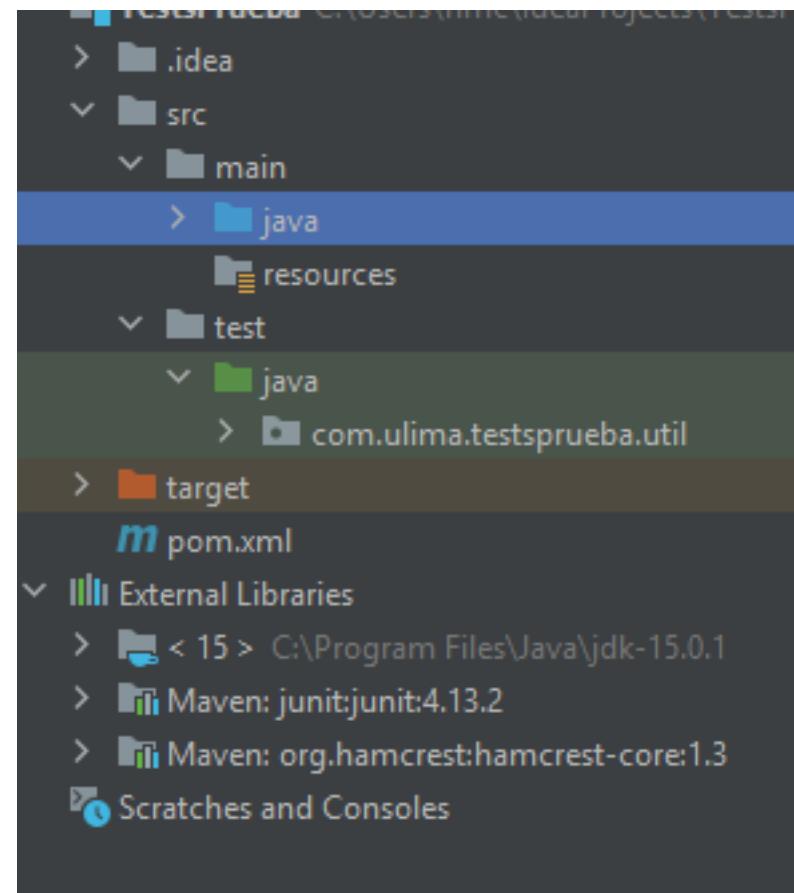
Repositorio oficial de proyectos maven <https://mvnrepository.com/>



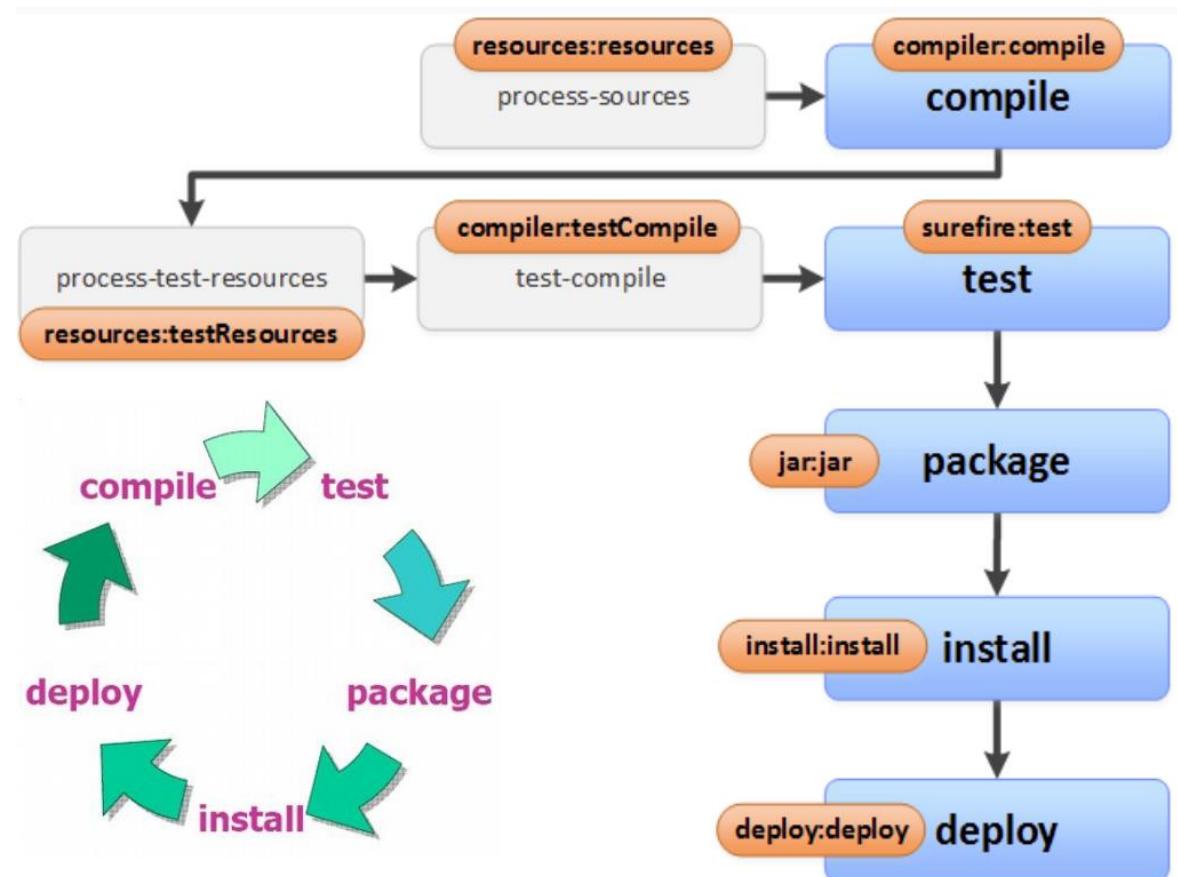
MAVEN

La estructura :

- **src/main/java** : contiene las clases java.
- **src/main/resources** : contiene XMLs, .properties, etc.
- **src/test/java** : contiene las clases java usadas en los tests.
- **src/test/resources** : contiene XMLs, .properties, etc. usados en los tests.



Ciclo de vida del proyecto Maven



Estructura de un proyecto- Maven

groupId: Aquí se pone el nombre de la empresa u organización, todos los proyectos con ese groupId pertenecen a una sola empresa.

artifactId: Es el nombre del proyecto.

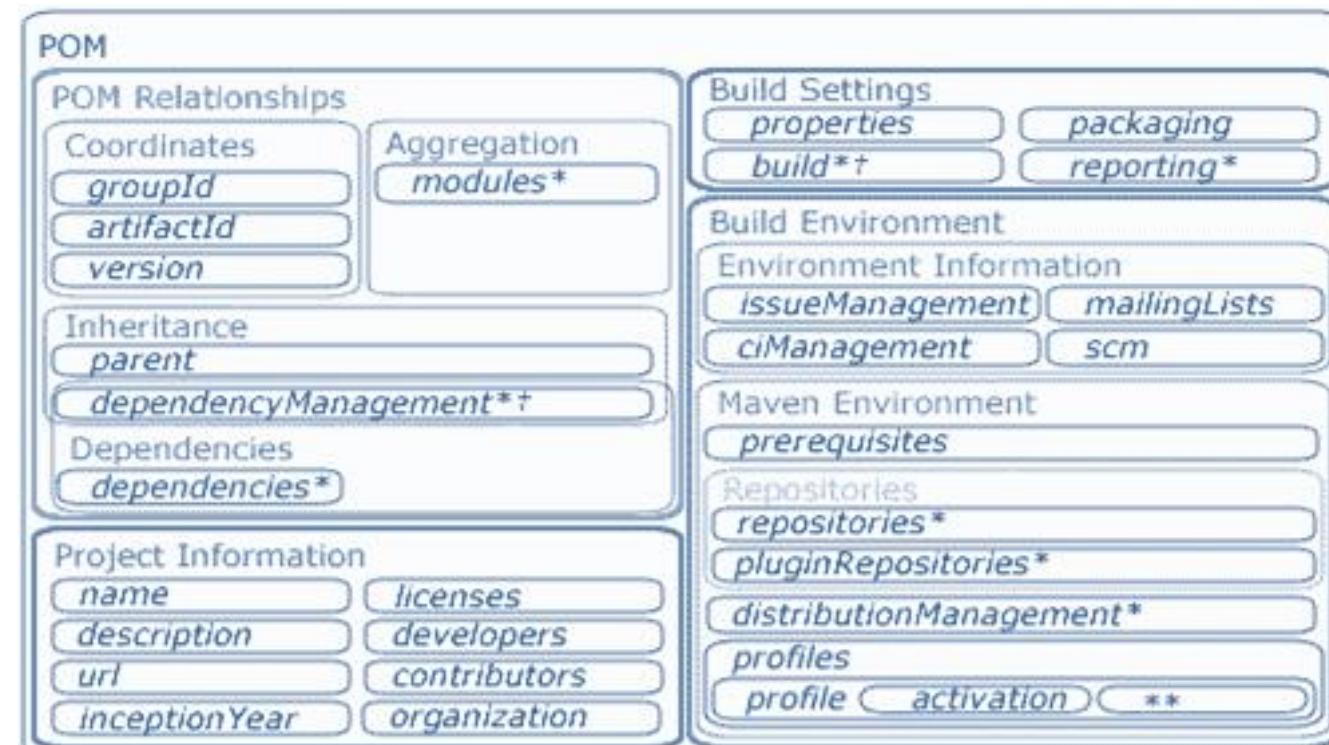
version: Número de versión del proyecto.

package: Paquete base donde irá el código fuente

Opción	Ejemplo
groupId	com.empres
artifactId	proyecto
version	1.3
package	com.empres.paquete

Maven

- Esquema de la estructura de un fichero pom.xml



```
<?XML VERSION="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.
<modelVersion>4.0.0</modelVersion>

<groupId>com.ulima.e</groupId>
<artifactId>TestsPrueba</artifactId>
<version>1.0-SNAPSHOT</version>

<properties>
    <maven.compiler.target>1.8</maven.compiler.target>
    <maven.compiler.source>1.8</maven.compiler.source>
</properties>

</project>
```

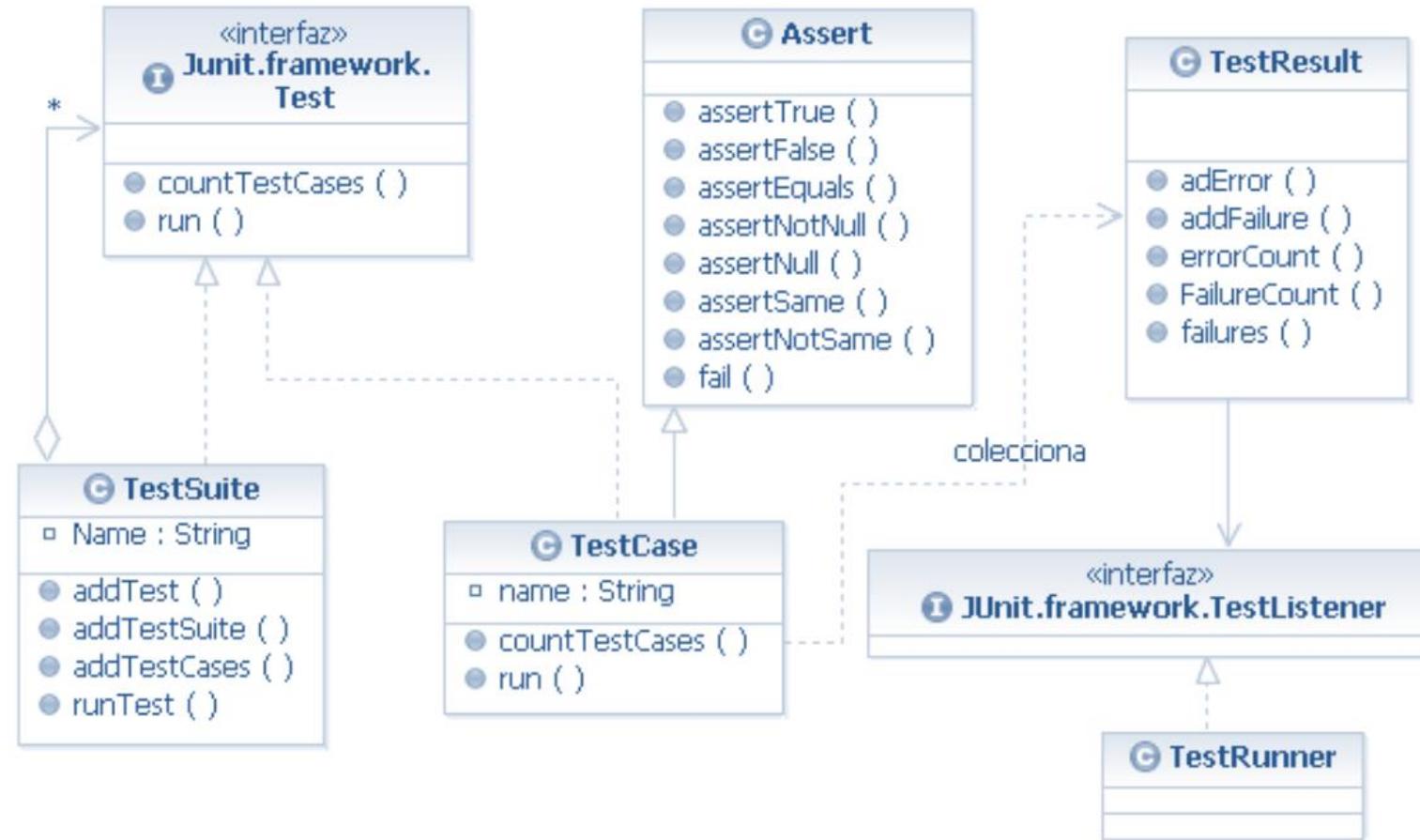
Las dependencias que necesitamos añadir al pom.xml son:

```
1<dependency>
2  <groupId>junit</groupId>
3  <artifactId>junit</artifactId>
4  <version>4.8.2</version>
5  <scope>test</scope>
6</dependency>
7<dependency>
8  <groupId>org.springframework</groupId>
9  <artifactId>spring-test</artifactId>
10 <version>3.2.4.RELEASE</version>
11 <scope>test</scope>
12</dependency>
13<dependency>
14  <groupId>org.mockito</groupId>
15  <artifactId>mockito-core</artifactId>
16  <version>1.8.5</version>
17  <scope>test</scope>
18</dependency>
```



JUnit es un paquete Java para automatizar las pruebas de clases Java.
<http://junit.org/>

Junit: Clases principales del Paquete



Junit - @Test

```
import static org.junit.Assert.*;  
import org.junit.Test;  
  
public class Ejemplo {  
  
    @Test  
    public void test() {  
        // prueba  
    }  
}
```

Los casos de prueba son métodos que

- no devuelven nada: void
- están etiquetados como @Test
- no tienen argumentos ()
- contienen aserciones

Junit - Assert

Las aserciones **son condiciones lógicas** que se usan para **verificar si los resultados generados por el código corresponden con los resultados esperados.**

assertEquals (X esperado, Xreal)	compara un resultado esperado con el resultado obtenido, determinando que la prueba pasa si son iguales, y que la prueba falla si son diferentes. Usa el método equals().
assertSame(X esperado, X real)	Ídem; pero usa == para determinar si es el objeto esperado.
assertFalse (boolean resultado)	verifica que el resultado es FALSE
assertTrue (boolean resultado)	verifica que el resultado es TRUE
assertNull (Object resultado)	verifica que el resultado es "null"
assertNotNull (Object resultado)	verifica que el resultado no es "null"
fail	sirve para detectar que estamos en un sitio del programa donde NO deberíamos estar

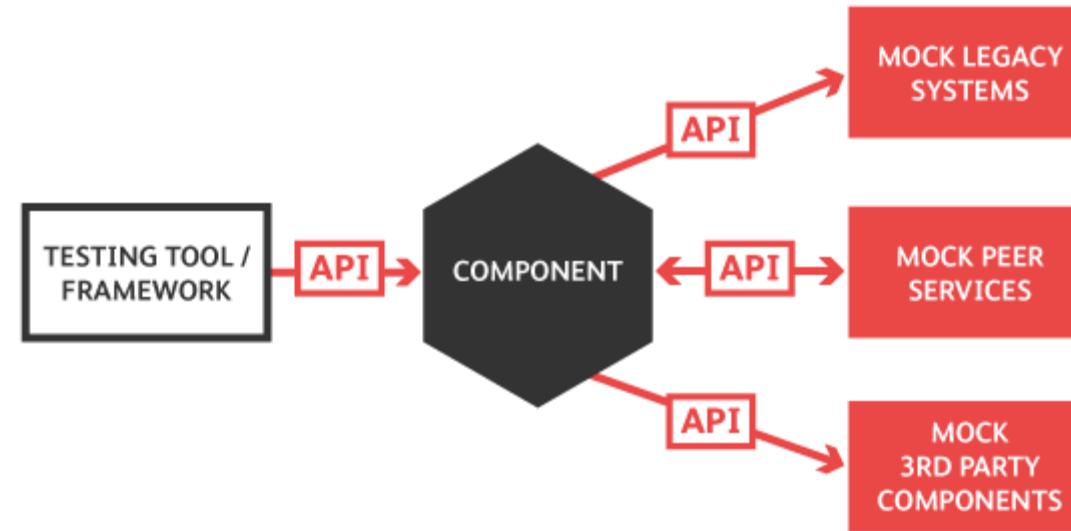
<https://junit.org/junit4/javadoc/latest/org/junit/Assert.html>

Junit

Anotación	Comportamiento
@Before	El método se ejecutará antes de cada prueba (antes de ejecutar cada uno de los métodos marcados con @Test). Será útil para inicializar los datos de entrada y de salida esperada que se vayan a utilizar en las pruebas.
@After	Se ejecuta después de cada <i>test</i> . Nos servirá para liberar recursos que se hubiesen inicializado en el método marcado con @Before.
@BeforeClass	Se ejecuta una sola vez antes de ejecutar todos los <i>tests</i> de la clase. Se utilizarán para crear estructuras de datos y componentes que vayan a ser necesarios para todas las pruebas. Los métodos marcados con esta anotación deben ser estáticos.
@AfterClass	Se ejecuta una única vez después de todos los <i>tests</i> de la clase. Nos servirá para liberar los recursos inicializados en el método marcado con @BeforeClass, y al igual que este último, sólo se puede aplicar a métodos estáticos.

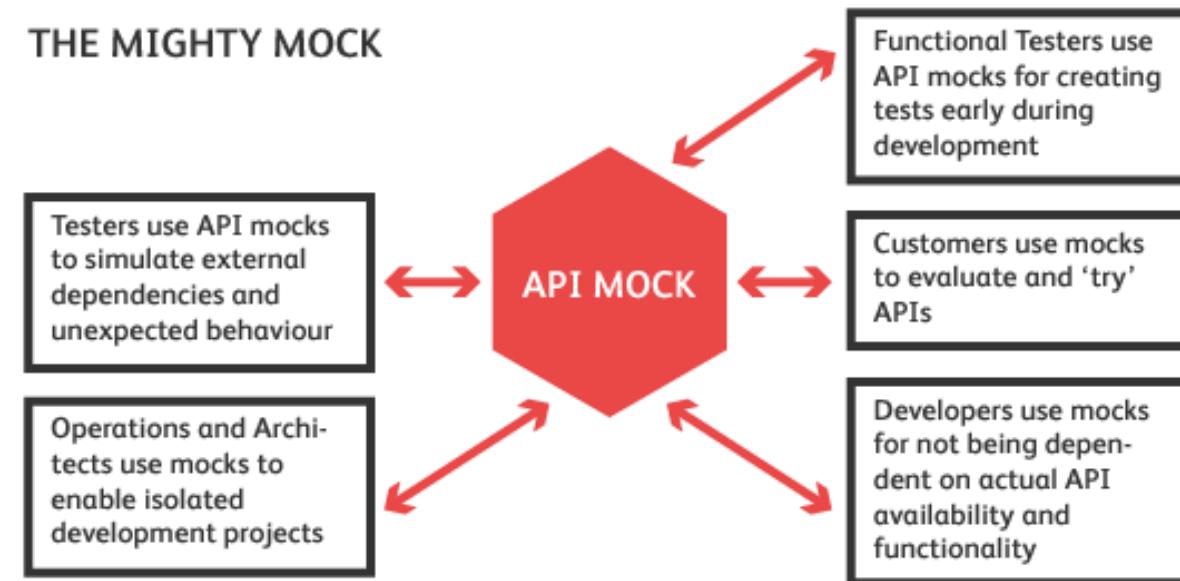
Prueba Simulada

La prueba simulada es un enfoque de la prueba unitaria que le permite hacer afirmaciones sobre cómo el código bajo prueba está interactuando con otros módulos del sistema.

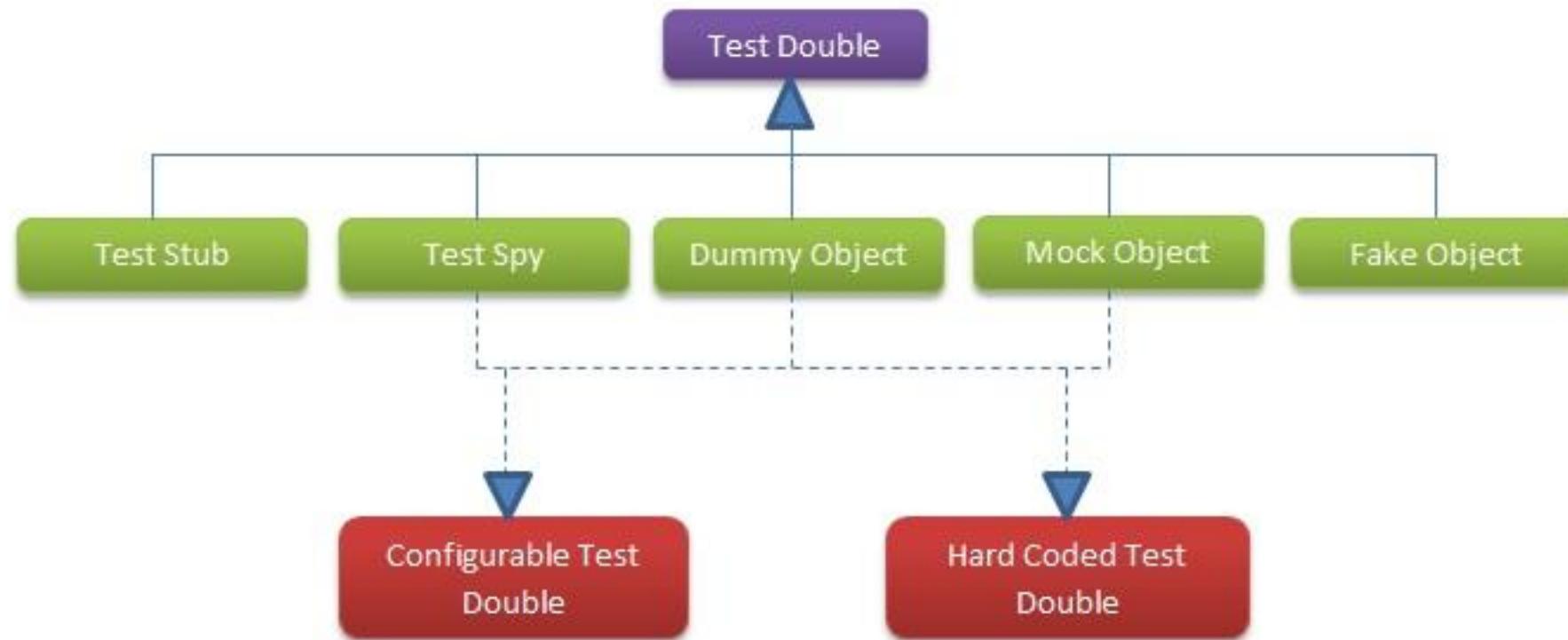


Utilidad de los Mocks

Son generalmente útil durante las pruebas unitarias para que las dependencias externas ya no sean una restricción para la unidad bajo prueba. A menudo, esas dependencias pueden estar en desarrollo.

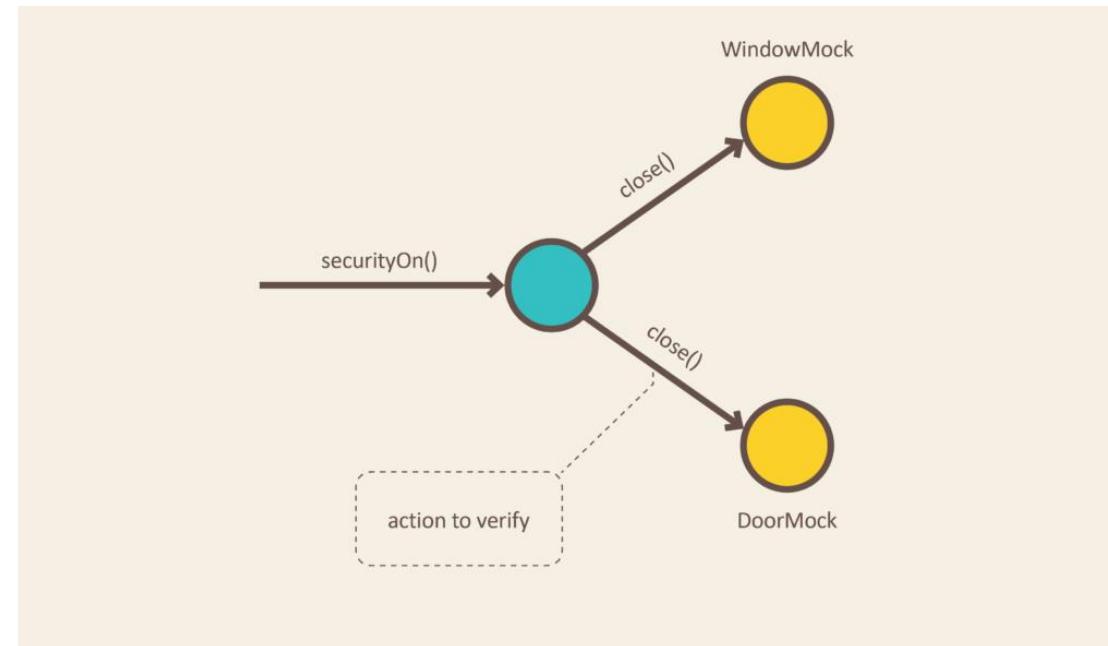


Dobles de Test

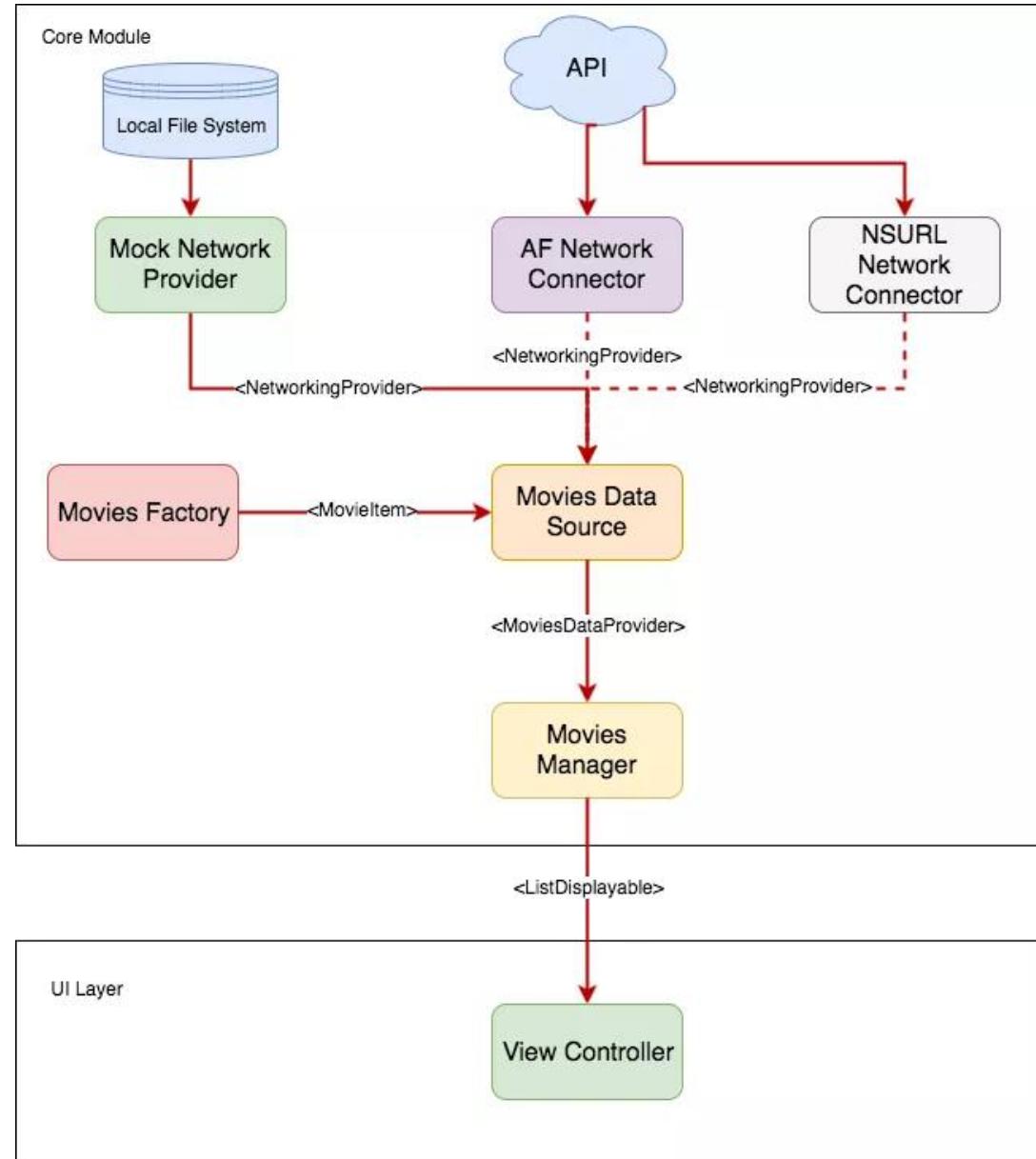


Mocks

Son objetos **preprogramados** con expectativas que conforman la **especificación de lo que se espera** que reciban las llamadas.



Ejemplo:



Mocks - Beneficios

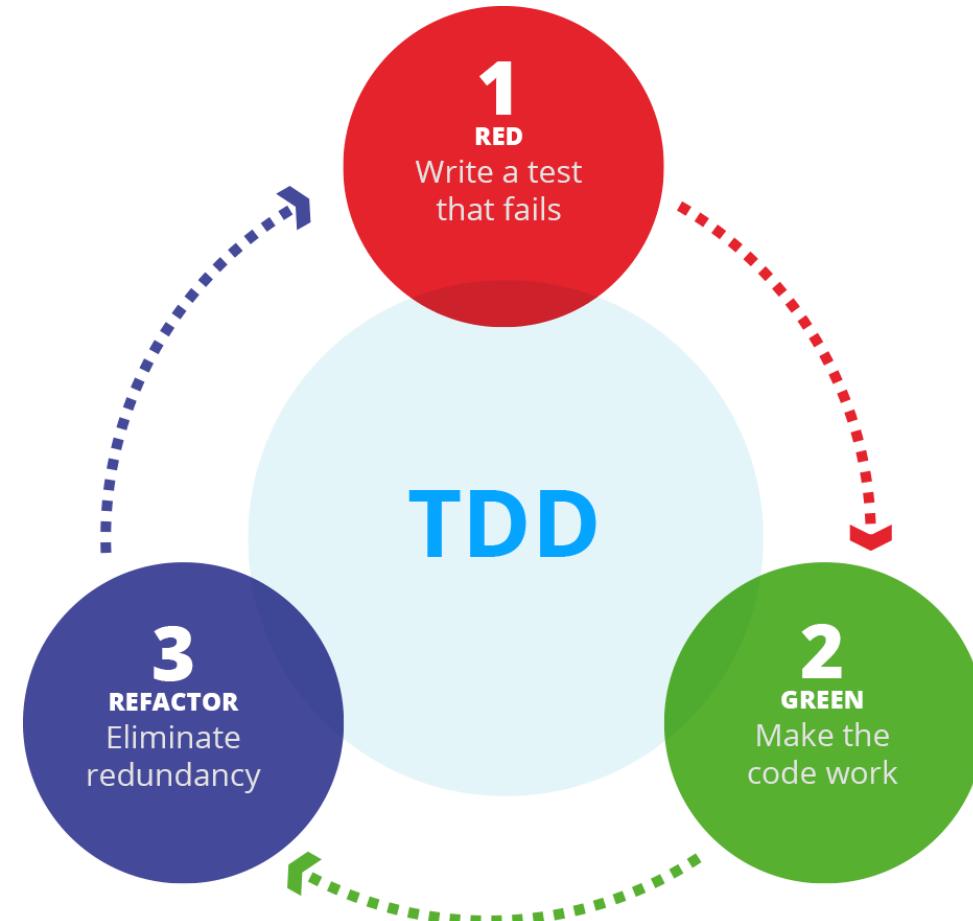
- Estos seudo objetos los podremos utilizar para simular el comportamiento de ciertas unidades de código para así poder hacer pruebas unitaria más flexibles.
- Beneficios
 - Mejores y más rápidos tests
 - Nos permite integrar distintos componentes sin necesidad de tenerlos terminados.



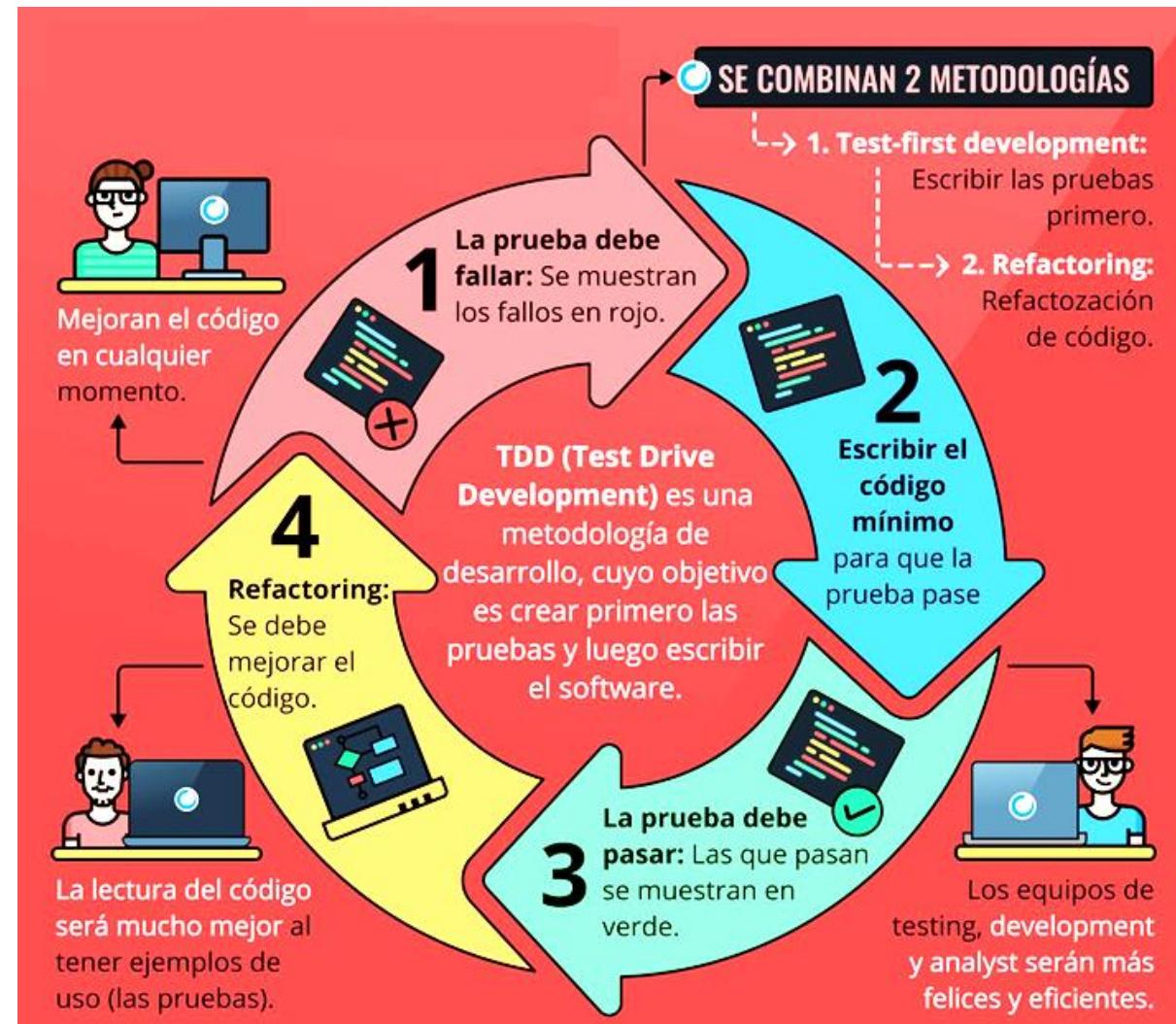
Proceso para usar Mock Objects

1. Crear los *Mock Objects* necesarios
2. Definir el comportamiento esperado de los *Mock Objects*, estableciendo las expectativas
3. Crear la instancia de la clase que va a ser probada de manera que use las instancias de los *Mock Objects* en vez de los objetos colaboradores originales
4. Ejecutar el método que se vaya a probar realizando la prueba correspondiente
5. Decir a cada *Mock Object* involucrado en la prueba que verifique si se cumplieron las expectativas

Desarrollo Dirigido por Pruebas



T.D.D.





Referencias bibliográficas

Pressman, R. Ingeniería del software: un enfoque práctico. McGraw-Hill, 1998.



DISEÑO DE SOFTWARE

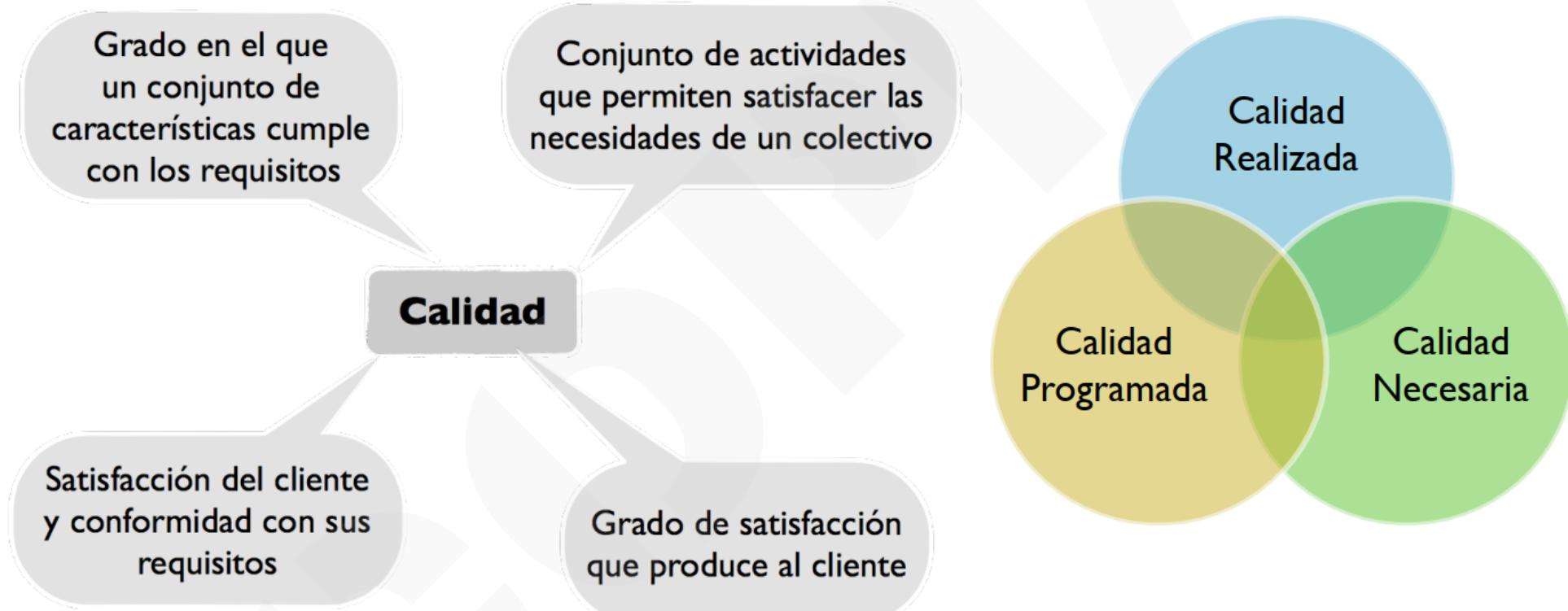
Métricas de Software - PCUS

Sesión S14

Agenda

1. Introducción
2. Métricas de Software
3. Estimación por Puntos de Casos de Uso
4. Ejemplo

Calidad del Software



Calidad del Software

Los requisitos del Software son la base de las medidas de **calidad**. La falta de concordancia con los requisitos es una falta de calidad.

Unos estándares específicos definen un conjunto de criterios de desarrollo que guían la manera en que se hace la ingeniería del Software. Si no se siguen los criterios , habrá seguramente poca calidad.

Existe un conjunto de requisitos implícitos que ha menudo no se nombran. Si el software cumple con sus requisitos explícitos pero falla en los implícitos , la calidad del software no será fiable.

Propósito

Las métricas de Software ayudan a una organización en dos frentes:

- Evaluación de la calidad del producto
- Evaluación de la calidad del proceso para producir productos de software

Conceptos

- **Medida**

Proporciona una indicación cuantitativa de la cantidad, dimensiones o tamaño de algunos atributos de un producto.

- **Medición**

Acto de determinar una medida.

- **Métrica**

Es una medida del grado en que un sistema, componente o proceso posee un atributo dado.



Conceptos

Entidades

Software

Programa

Programa

Software

Atributos

Calidad

tamaño

Complejidad

Confiabilidad

Mediciones

?

?

?

?

Métricas de Software

Las métricas del Software comprenden un amplio rango de actividades diversas, estas son algunas:

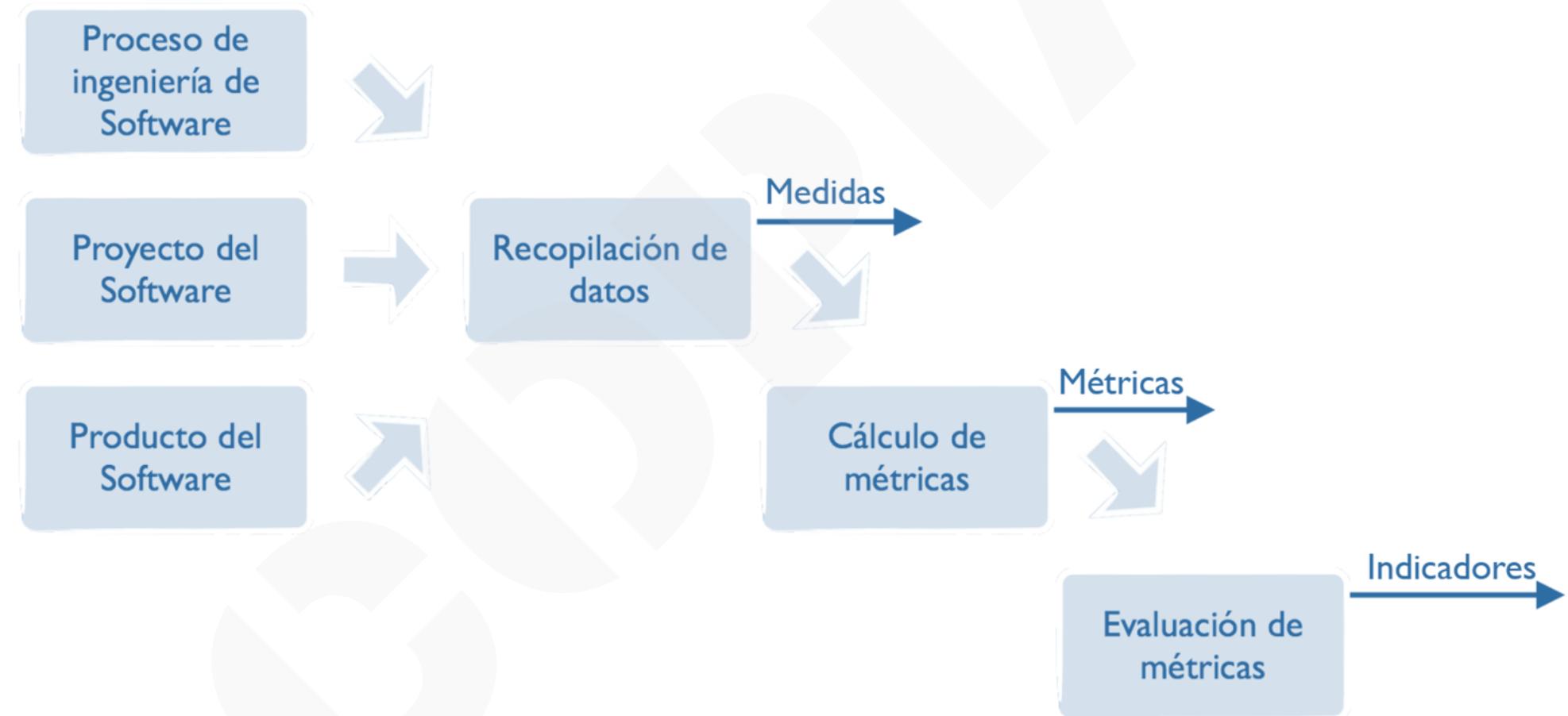
- Aseguramiento y control de calidad
- Modelos de fiabilidad
- Modelos y evaluación de ejecución
- Modelos y medidas de productividad

Aplicación continua de mediciones en el proceso de desarrollo del software y sus productos, para suministrar información relevante a tiempo

mejorar tanto el procesos como los productos

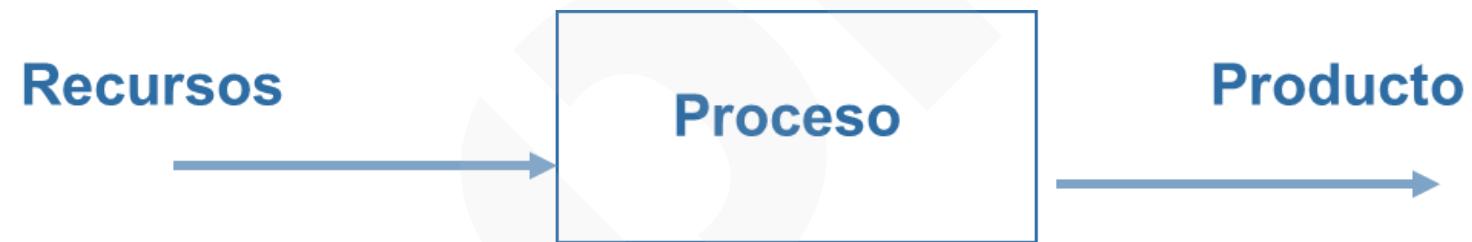


Proceso de recopilación de métricas de Software



Alcance de las métricas de Software

Métricas para entender, controlar y mejorar



Clasificación de las métricas de Software

Según los criterios:

de complejidad	Métricas que definen la medición de la complejidad: volumen, tamaño, anidaciones, y configuración.
de calidad	Métricas que definen la calidad del software: exactitud, estructuración o modularidad, pruebas, mantenimiento.
de competencia	Métricas que intentan valorar o medir las actividades de productividad de los programadores con respecto a su certeza, rapidez, eficiencia y competencia
de desempeño	Métricas que miden la conducta de módulos y sistemas de un software, bajo la supervisión del SO o hardware.
estilizadas	Métricas de experimentación y de preferencia: estilo de código, convenciones, limitaciones, etc.

Según el contexto en que se apliquen:

Métricas de proceso
Métricas de proyecto
Métricas de producto

Métricas de Calidad

- Principal objetivo de los ingenieros de software es producir sistemas, aplicaciones o productos de alta calidad.
- Para las evaluaciones que se quieran obtener es necesario la utilización de medidas técnicas, que evalúan la calidad de manera objetivo.



Modelos de Métricas de Calidad

Modelo de MCCALL (1977)

Factor	Criterio
Correctitud	Rastreabilidad
	Compleitud
	Consistencia
Confiabilidad	Consistencia
	Exactitud
	Tolerancia a fallas
Eficiencia	Eficiencia de ejecución
	Eficiencia de almacenamiento
Integridad	Control de acceso
	Auditoría de acceso
Usabilidad	Operabilidad
	Entrenamiento
	Comunicación
Interoperabilidad	Modularidad
	Similitud de comunicación
	Similitud de datos.

Criterios asociados a los factores de calidad

Factor	Criterio
Mantenibilidad	Simplicidad
	Concreción
Capacidad de Prueba	Simplicidad
	Instrumentación
Flexibilidad	Auto-descriptividad
	Modularidad
Portabilidad	Auto-descriptividad
	Independencia del sistema
Reusabilidad	Independencia de máquina
	Auto-descriptividad
Flexibilidad	Generalidad
	Modularidad
Portabilidad	Independencia del sistema
	Independencia de máquina

Modelo de FURPS (1987)

Factor	Criterio
Rendimiento	Características y capacidades del programa
	Generalidad de las funciones
Capacidad de Soporte	Seguridad del sistema
	Factores humanos
Extensibilidad	Factores estéticos
	Consistencia de la interfaz
Adaptabilidad	Documentación
	Frecuencia y severidad de las fallas
Capacidad de pruebas	Exactitud de las salidas
	Tiempo medio de fallos
Capacidad de configuración	Capacidad de recuperación ante fallas
	Capacidad de predicción

Factor	Criterio
Correctitud	Funcionalidad
	Confiabilidad
Internas	Mantenibilidad
	Eficiencia
	Confiabilidad
Contextuales	Mantenibilidad
	Reusabilidad
	Portabilidad
	Confiabilidad
Descriptivas	Mantenibilidad
	Reusabilidad
	Portabilidad
	Usabilidad

Modelo de DROMEY (1996)

Modelos de Métricas de Calidad

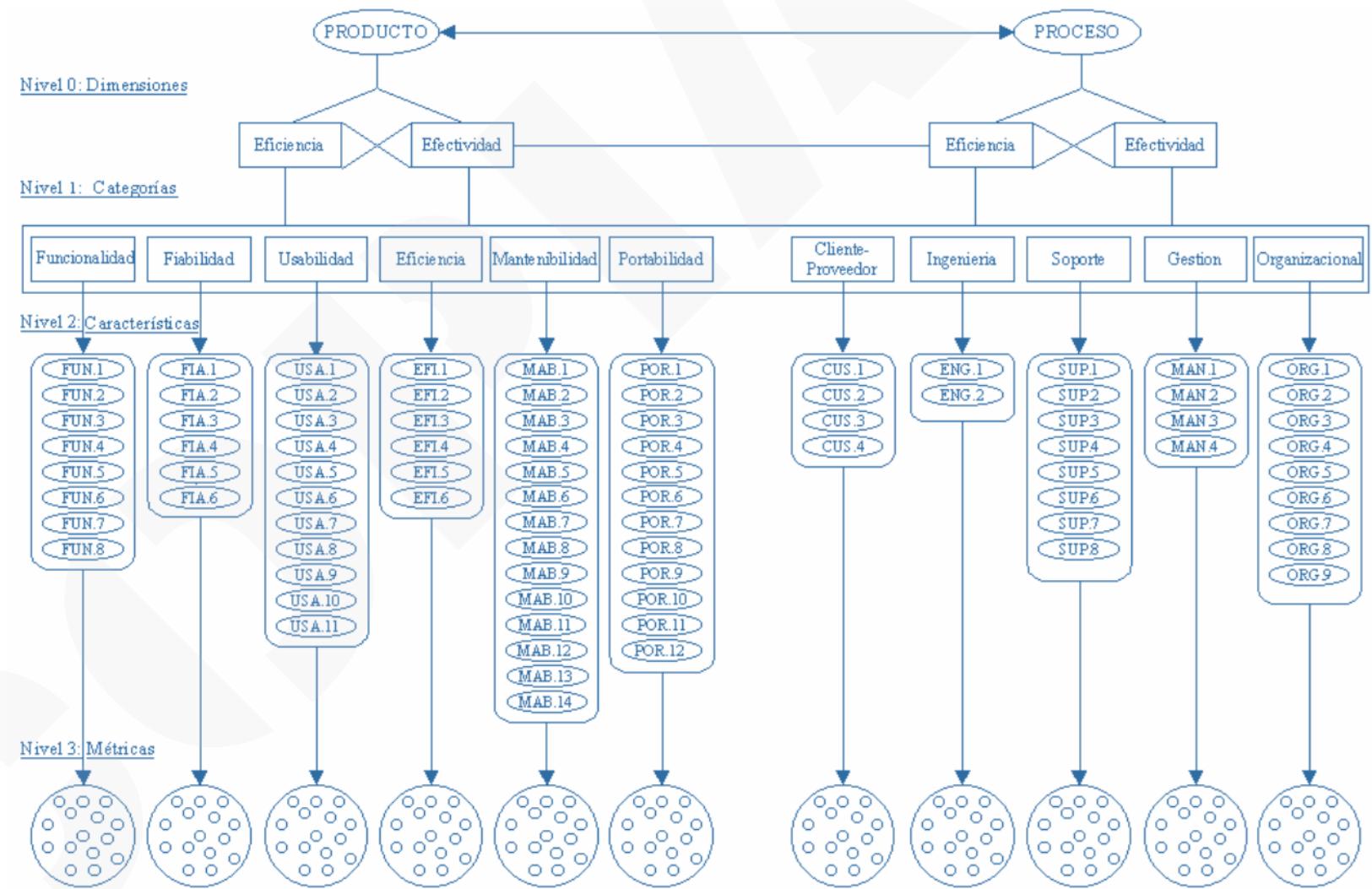
Normas ISO 9000
ISO/IEC 9126



Modelos de Métricas de Calidad

MOSCA (Modelo Sistémico de Calidad)

Consta de 4 niveles: dimensiones, categorías, características y las métricas. En base de tres ramas: el producto, el proceso y la humana. Contiene un total de 715 métricas.



Modelos de Métricas de Calidad

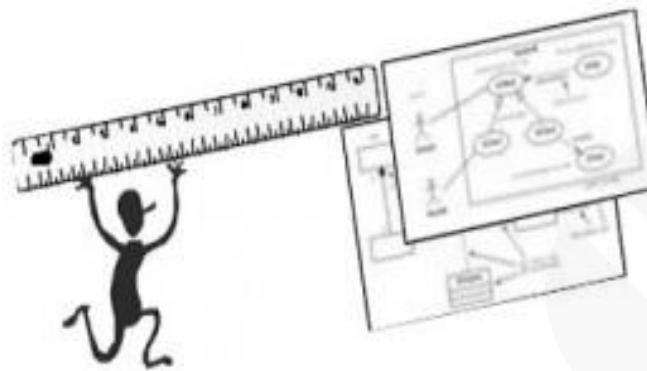
MOSCA (Modelo Sistémico de Calidad)

Ejemplo de agrupación de métricas

Categoría	Características	
	Aspectos Contextuales del Producto	Aspectos Internos del Producto
Funcionalidad (FUN) Total de métricas: 46	FUN 1. Ajuste a los propósitos (16) FUN 2. Precisión (10) FUN 3. Interoperabilidad (7) FUN 4. Seguridad (2)	FUN 5. Correctitud (8) FUN 6. Estructurado (1) FUN 7. Encapsulado (1) FUN 8. Especificado (1)
	Sub-total de métricas: 35	Sub-total de métricas: 11
Fiabilidad (FIA) Total de métricas: 32	FIA 1. Madurez (17) FIA 2. Tolerancia a fallas (1) FIA 3. Recuperación (4)	FIA 4. Correctitud (8) FIA 5. Estructurado (1) FIA 6. Encapsulado (1)
	Sub-total de métricas: 22	Sub-total de métricas: 10
Usabilidad (USA) Total de métricas: 38	USA 1. Facilidad de comprensión (5) USA 2. Capacidad de Aprendizaje (9) USA 3. Interfaz Gráfica (5) USA 4. Operabilidad (13) USA 5. Conformidad con los estándares	USA 6. Completo (1) USA 7. Consistente (1) USA 8. Efectivo (1) USA 9. Especificado (1) USA 10. Documentado (1) USA 11. Auto-descriptivo (1)
	Sub-total de métricas: 32	Sub-total de métricas: 6

Estimación

El proceso de gestión de proyectos de software comienza con un conjunto de actividades que se denominan ***planificación del proyecto***. La primera de estas actividades es la ***estimación*** . Estimar, o *cuantificar*, software no es una tarea fácil.



Antecedentes

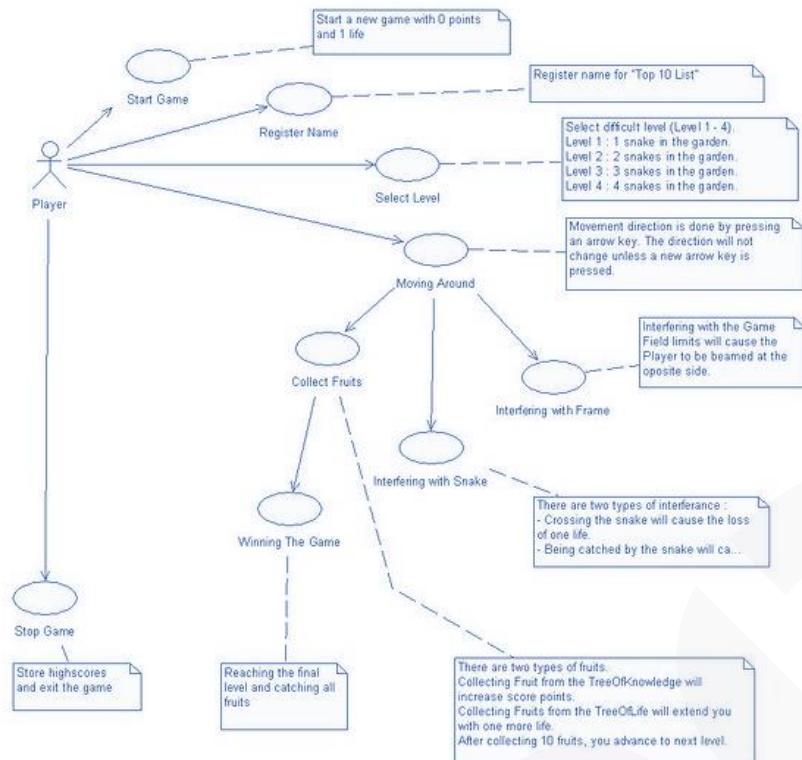
- Las metodologías de desarrollo de sistemas han evolucionado
- Por un lado, se conocen algunas técnicas como COCOMO, Puntos de Función y otras
- Cambio en las metodologías para el desarrollo de software

Estimación Puntos de Casos de Uso



- El tiempo para completar un proyecto se ve afectado por lo siguiente:
 1. Número de pasos para completar el caso de uso
 2. Número y complejidad de los actores
 3. Los requerimientos técnicos del caso de uso, como: concurrencia, seguridad o rendimiento
 4. Varios factores ambientales tales como: experiencia y conocimientos del equipo de desarrollo
- Un método de estimación que tome en cuenta tales factores en el ciclo de vida de un proyecto será muy útil para calcular: tiempo, costo y asignación de recursos

Puntos de Casos de Uso



El método de Puntos Casos de Uso (Use Case Points) fue desarrollado en 1993 por **Gustav Kamer**, bajo la supervisión de **Ivar Jacobson** (creador de los casos de uso y gran promovedor del desarrollo de UML y el Proceso Unificado).

Su principal ventaja es la rápida adaptación a empresas que ya estén utilizando la técnica de Casos de Uso.

- Estima el esfuerzo (en horas-hombre) de un proyecto de desarrollo de software a partir de los casos de uso.

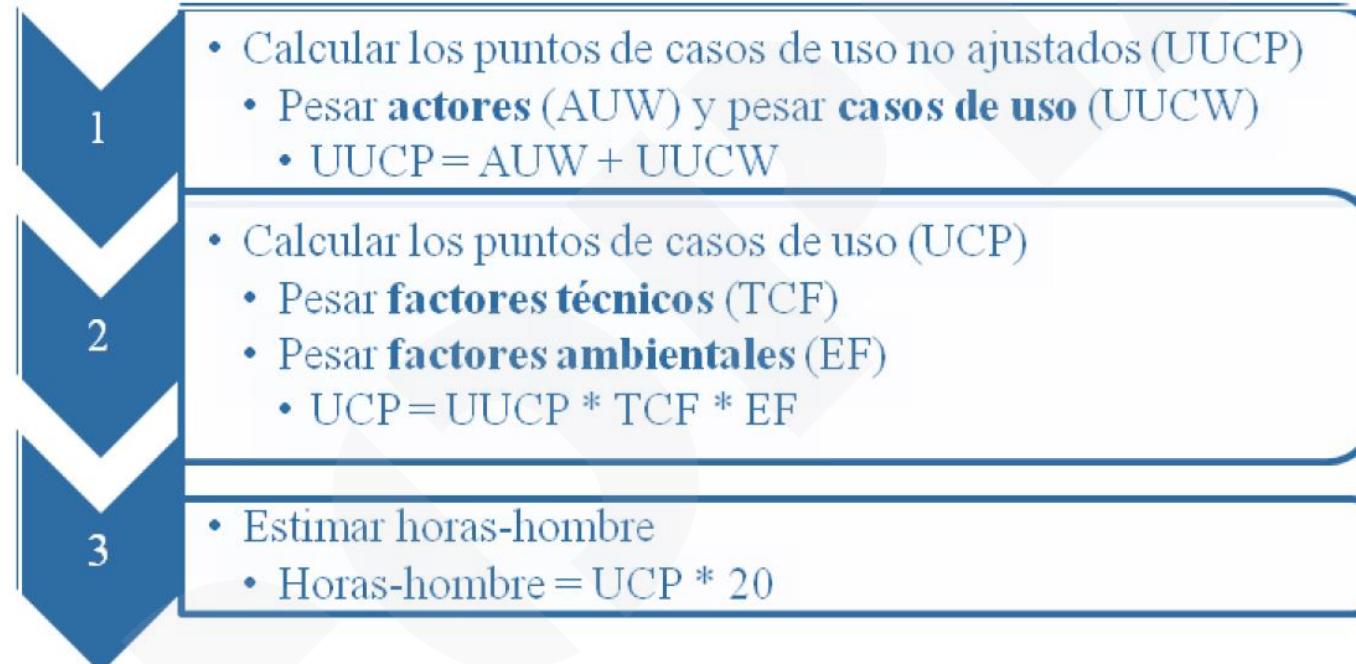
Ventajas – Desventajas de los PCU

Tabla 1. VENTAJAS Y DESVENTAJAS DE LA TÉCNICA DE PUNTOS DE CASOS DE USO

Ventajas	Desventajas
Trabaja bien con diferentes tipos de software	No existe un estándar para escribir casos de uso lo que dificulta la aplicación del método.
Muestra buen rendimiento en proyectos pequeños, medianos y grandes.	Las herramientas en esta área son caras y se enfocan en la evaluación del proyecto

Vale la pena aclarar que un caso de uso por sí solo no permite efectuar una estimación de esfuerzos ni de tiempos, solamente son una herramienta para el análisis. La idea central es estimar el tamaño (cuantificar) del software a partir de los requerimientos de los casos de uso.

Pasos de la técnica de puntos de casos de uso



Ecuación UCP (Use Case Points)

$$\mathbf{UCP=UUCP*TCF*ECF}$$

Donde:

- UUCP = Puntos de Caso de Uso sin ajustar (Unadjusted Use Case Points)
- TCF = Factor de complejidad técnica (Technical Complexity Factor)
- ECF = Factor de Complejidad del Medio Ambiente (Environment Complexity Factor)
- PF = Factor de productividad (Productivity Factor)

NOTA: Cuando la productividad se incluye como un coeficiente que expresa el tiempo, entonces la ecuación puede utilizarse para estimar el número de horas-hombre necesarias para completar un proyecto.

Ecuación UCP (Use Case Points)



Hay que hacer algunos cálculos ...

UUCP (Puntos de Caso de Uso sin ajustar)

Da una idea un poco más precisa de la dificultad de los casos de uso e interfaces

$$\mathbf{UUCP} = \mathbf{UAW} + \mathbf{UUCW}$$

Se calculan con base en:

- Pesos de los Casos de Uso sin Ajustar (**UUCW**): Basado en el número total de actividades (o pasos) contenidos en todos los escenarios del caso de uso.
- Pesos de los Actores sin Ajustar (**UAW**) : Basado en la combinación de la complejidad de todos los actores en todos los casos de uso.

UUCW - Peso de Actores

Tabla 2. PESO DE LOS ACTORES

Tipo de actor	Descripción	Factor
Simple	Otro sistema con una API definida	1
Medio	Otro sistema interactuando con algún protocolo (TCP) o una persona interactuando a través de una interfaz en modo texto	2
Complejo	Una persona interactuando a través de una interfaz gráfica de usuario	3

UUCW - Peso de CUS

Tabla 3. PESO DE LOS CASOS DE USO

Tipo de caso de uso	Descripción	Factor
Simple	3 transacciones o menos	5
Medio	4 a 7 transacciones	10
Complejo	Más de 7 transacciones	15

Simple : *Interfaz de usuario simple.* Toca solo una única entidad de la Base de datos. Su escenario de éxito tiene 3 pasos o menos. Su implementación involucra menos de 5 clases

Medio : *Más diseño de interfaz.* Toca 2 o más entidades de base de datos. Entre 4 y 7 pasos. Su implementación implica entre cinco y 10 clases

Complejo: *Interfaz de usuario compleja o procesamiento.* Toca 3 o más entidades de la Base de datos. Más de 7 pasos. Su implementación implica más de 10 clases.

UUCW

Luego, se debe completar este cuadro:

Tipo de Caso de Uso	Descripción	Peso (factor)	Número de Casos de Uso	Resultado
Simple	Transacciones= 3 ó menos Clases= Menos de 5	5	<input type="text"/>	<input type="text"/>
Medio	Transacciones= 4 a 7 Clases= 5 a 10	10	<input type="text"/>	<input type="text"/>
Complejo	Transacciones= Más de 7 Clases= Más de 10 clases	15	<input type="text"/>	<input type="text"/>
Total UUCW				<input type="text"/>

Resultado = Factor x número de CUs

Total es la sumatoria de Resultados

UAW

Consiste en la evaluación de la complejidad de los actores con los que tendrá que interactuar el sistema

Tabla 2. PESO DE LOS ACTORES		
Tipo de actor	Descripción	Factor
Simple	Otro sistema con una API definida	1
Medio	Otro sistema interactuando con algún protocolo (TCP) o una persona interactuando a través de una interfaz en modo texto	2
Complejo	Una persona interactuando a través de una interfaz gráfica de usuario	3

Luego, se debe completar este cuadro:

Tipo de Actor	Descripción	Peso (factor)	Número de Actores	Resultado
Simple	Otro sistema que interactúa con el sistema a desarrollar mediante una interfaz de programación (API).	1	<input type="text"/>	<input type="text"/>
Medio	Otro sistema interactuando a través de un protocolo (ej. TCP/IP) o una persona interactuando a través de una interfaz en modo texto.	2	<input type="text"/>	<input type="text"/>
Complejo	Una persona que interactúa con el sistema mediante una interfaz gráfica (GUI).	3	<input type="text"/>	<input type="text"/>
Total UAW			<input type="text"/>	

Resultado = Factor x número de actores

Total es la sumatoria de Resultados

TCF (Factor de Complejidad Técnica)

Son **13 puntos** que evalúan la complejidad de los módulos del sistema que se desarrolla. Cada uno tiene un peso predefinido y un factor subjetivo percibido por el equipo de desarrollo.

Tabla 4. FACTORES TÉCNICOS

Factor	Descripción	Peso
T1	Sistema distribuido	2
T2	Objetivos de performance o tiempo de respuesta	1
T3	Eficiencia del usuario final	1
T4	Procesamiento interno complejo	1
T5	El código debe ser reutilizable	1
T6	Facilidad de instalación	0.5
T7	Facilidad de uso	0.5
T8	Portabilidad	2
T9	Facilidad de cambio	1
T10	Concurrencia	1
T11	Objetivos especiales de seguridad	1
T12	Acceso directo a terceras partes	1
T13	Facilidades especiales de entrenamiento a usuarios	1

Tabla 5. ESCALAS DE ESTIMACION TCF

Descripción	Valor
Irrelevante	De 0 a 2
Medio	De 3 a 4
Esencial	5

Cada uno de estos factores se deben evaluar según la siguiente escala SUBJETIVA:

Descripción	Valor
Irrelevante	0 a 2
Medio	3 a 4
Esencial	5

TCF (Factor de Complejidad Técnica)

Luego, se debe completar este cuadro:

Factor técnico	Descripción	Peso	Impacto percibido	Factor calculado
T1	Sistema distribuido	2	<input type="text"/>	<input type="text"/>
T2	Rendimiento o tiempo de respuesta	1	<input type="text"/>	<input type="text"/>
T3	Eficiencia del usuario final	1	<input type="text"/>	<input type="text"/>
T4	Procesamiento interno complejo	1	<input type="text"/>	<input type="text"/>
T5	El código debe ser reutilizable	1	<input type="text"/>	<input type="text"/>
T6	Facilidad de instalación	0.5	<input type="text"/>	<input type="text"/>
T7	Facilidad de uso	0.5	<input type="text"/>	<input type="text"/>
T8	Portabilidad	2	<input type="text"/>	<input type="text"/>
T9	Facilidad de cambio	1	<input type="text"/>	<input type="text"/>
T10	Concurrencia	1	<input type="text"/>	<input type="text"/>
T11	Características especiales de seguridad	1	<input type="text"/>	<input type="text"/>
T12	Provee acceso directo a terceras partes	1	<input type="text"/>	<input type="text"/>
T13	Se requiere facilidades especiales de entrenamiento a usuario	1	<input type="text"/>	<input type="text"/>
Factor Total Técnico				<input type="text"/>

Resultado = Peso x impacto

$$TCF = 0,6 + (0,01 \cdot \sum_{i=1}^{i=13} R_i)$$

Total es la sumatoria
de Resultados

$$\mathbf{TCF = 0.6 + (0.01 * Factor Total Técnico)}$$

ECF (Factor de Complejidad Ambiental)

Establece la experiencia del equipo de desarrollo: están relacionados con las habilidades y experiencia del grupo de personas involucradas con el desarrollo del proyecto.

Tabla 6. FACTORES AMBIENTALES

Factor	Descripción	Peso
E1	Familiaridad con el modelo del proyecto utilizado	1.5
E2	Experiencia en la aplicación	0.5
E3	Experiencia en orientación a objetos	1
E4	Capacidad del analista líder	0.5
E5	Motivación	1
E6	Estabilidad en los requerimientos	2
E7	Personal de medio tiempo	-1
E8	Dificultad en el lenguaje de programación	-1

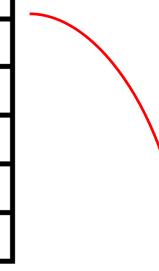


Tabla 7. ESCALAS DE ESTIMACION EF

Descripción	Valor
Sin experiencia, sin motivación, estabilidad	De 0 a 2
Promedio	3
Amplia experiencia, motivación, estabilidad	De 3 a 5

Cada uno de estos factores se deben evaluar de **0 a 5** según la siguiente escala SUBJETIVA:

- Un valor de 1 significa que el factor tiene un fuerte impacto negativo para el proyecto,
- 3 es medio
- 5 significa que tiene un fuerte impacto positivo

ECF (Factor de Complejidad Ambiental)

Factor Ambien-tal	Descripción	Peso	Impacto percibi-do	Factor calculado
E1	Familiaridad con el modelo de proyecto utilizado Familiaridad con UML	1.5	<input type="text"/>	<input type="text"/>
E2	Personal tiempo parcial	-1	<input type="text"/>	<input type="text"/>
E3	Capacidad del analista líder	0.5	<input type="text"/>	<input type="text"/>
E4	Experiencia en la aplicación	0.5	<input type="text"/>	<input type="text"/>
E5	Experiencia en orientación a objetos	1	<input type="text"/>	<input type="text"/>
E6	Motivación	1	<input type="text"/>	<input type="text"/>
E7	Dificultad del lenguaje de programación	-1	<input type="text"/>	<input type="text"/>
E8	Estabilidad de los requerimientos	2	<input type="text"/>	<input type="text"/>
11/04/2011				Factor Ambiental Total <input type="text"/>

Resultado = Peso x impacto

$$EF = 1,4 - (0,03 \cdot \sum_{i=1}^{i=8} R_i)$$

Total es la sumatoria de Resultados

$$\mathbf{ECF = 1.4 + (-0.03 * Factor Total Ambiental)}$$

PF (Factor de Productividad)

- El PF es la proporción de horas hombre de desarrollo necesario por caso de uso. Estadísticas de proyectos pasados proporcionan los datos para estimar la PF inicial.
- Por ejemplo, si un proyecto pasado con un UCP de 120 tomó 2,200 horas para completarse, divide 2,200 por 120 para obtener un PF de 18 horas hombre por punto de caso de uso.

PF (Factor de Productividad)

- El método original de Karner propone usar un factor de ajuste de 20 personas-hora por cada Punto Caso de Uso (UCP).

Horas-hombre sistema de procesamiento de órdenes = $57.77 * 20 = 1155.52$

~ 29 semanas a 40 horas por semana, para una persona

Tomando un equipo pequeño de 6 personas trabajando full-time

~ 5 semanas de esfuerzo

- Barnerjee propone un rango entre 15 y 30 horas

PF (Factor de Productividad)

Factor Ambiental	Descripción del factor	Peso
E1	Familiaridad con el modelo de proyecto	1.5
E2	Experiencia en la aplicación	0.5
E3	Experiencia en orientación a objetos	1
E4	Capacidad del analista líder	0.5
E5	Motivación	1
E6	Estabilidad de los requerimientos	2
E7	Personal part-time	-1
E8	Dificultad del lenguaje de programación	-1

PF (Factor de Productividad)

- Zcheider y Winters sugiere un refinamiento de los factores de entorno (EF) y seguir el procedimiento que se presenta a continuación:
 - Contar cuántos factores ambientales desde E1 a E6 son inferiores a 3 (valor de nivel promedio) y cuántos de los factores ambientales E7 y E8 son superiores a 3
 - Entonces se usan
 - **20** horas-hombre por UCP **si el valor es ≤ 2**
 - **28** horas-hombre por UCP **si el valor es ≤ 4**
 - **36** horas-hombre por UCP **si el valor es ≥ 5**
 - En este caso considerar modificar el proyecto ya que es muy riesgoso

Esfuerzo = UCP * Factor de Productividad

PF (Factor de Productividad)

Es inferior a 3

Factor Ambiental	Peso	ELevel	ELevel * Peso	Justificación
E1	1.5	1	1.5	La mayoría del equipo no familiarizado
E2	0.5	3	1.5	La mayoría del equipo son programadores
E3	1	3	3	Programadores OO
E4	0.5	5	2.5	El líder es idóneo
E5	1	5	5	El equipo está entusiasmado
E6	2	5	10	No se esperan cambios
E7	-1	0	0	Sin empleados a tiempo parcial
E8	-1	3	-3	Se programará con Java

Tenemos un EF inferior al promedio => 20 horas-hombre por UCP

Estimación del proyecto

Estimar Horas-hombre Sistema Procesamiento de órdenes
Para un **UCP= 57.77**

Horas-hombre sistema de procesamiento de órdenes = $57.77 * 20 = 1155.52$

Actividad	Porcentaje
Análisis	10%
Diseño	20%
Programación	40%
Pruebas	15%
Sobrecarga (otras actividades)	15%

Actividad	Porcentaje	Horas-hombre
Análisis	10%	288.88
Diseño	20%	577.76
Programación	40%	1155.52
Pruebas	15%	433.32
Sobrecarga (otras actividades)	15%	433.32
TOTAL Esfuerzo	100%	2888.8

~ 72 semanas a 40 horas por semana, para una persona
Tomando un equipo de 6 personas trabajando full-time:

~ 12 semanas de esfuerzo

Horas Estimadas

Total horas estimadas = UCP*PF

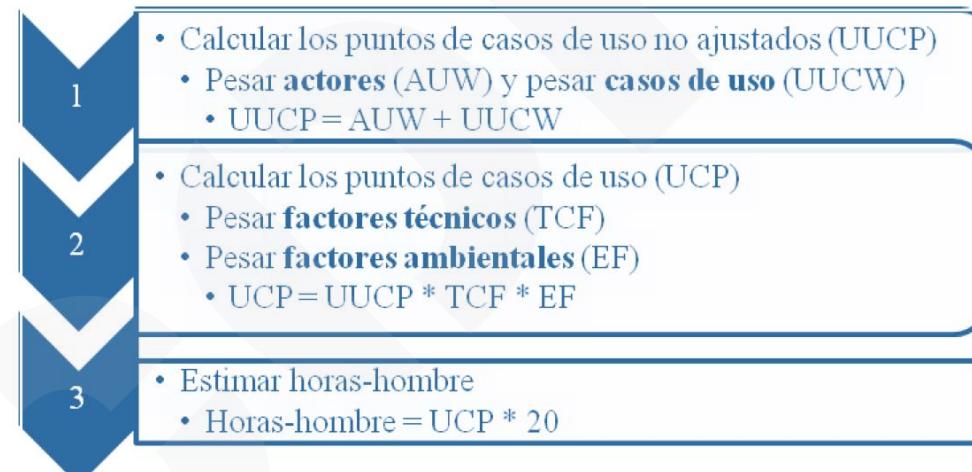
Si no hay datos históricos que hayan sido recabados se toman en cuenta dos posibilidades:

1. Establecer una base para cálculos de los UCP de proyectos completados anteriormente.
2. Utilice un valor entre **15 y 30** dependiendo de la experiencia y logros pasados del equipo de desarrollo. Si se trata de un nuevo equipo, use un valor de **20** para el primer proyecto.

Horas Estimadas

Siguiente paso de esta técnica nos indica que debemos estimar las horas hombre mediante la fórmula:

$$\text{HH} = \text{UCP} * 20$$



Horas Estimadas

El autor de la técnica sugiere usar 20 horas hombre por UCP. Por ejemplo, para un sistema de 60 UCP * 20 hrs/hombre nos da un total de 1200 hrs/hombre. Lo que equivale a 30 semanas (40 hrs por semana para una persona), de esta forma, un equipo de 5 personas desarrollarían el sistema en 6 semanas. En resumen:

Tabla 8. RESUMEN EJEMPLO

Actividad	Porcentaje de tiempo	Horas-Hombre
Análisis	10%	300
Diseño	20%	600
Programación	40%	1200
Pruebas	15%	450
Sobrecarga	15%	450
TOTAL	100%	3000

Ejemplo

Para demostrar los pasos en los que se basa la técnica, realizaremos un caso práctico para el caso de uso “**retirar dinero de cajero automático**”. El proceso se explica con el fin de lograr una mayor comprensión de la técnica y los pasos asociados.

- **Actor:** cliente.
- **Propósito:** realizar un retiro de una cuenta desde un cajero automático.
- **Visión general:** un cliente llega a un cajero automático, introduce la tarjeta, se identifica y solicita realizar un retiro de dinero. El cajero le da el dinero solicitado tras comprobar que la operación puede realizarse. El cliente toma el dinero y se va.

Ejemplo

Curso típico de eventos :

1. Este caso de uso empieza cuando un cliente introduce su tarjeta en el cajero.
2. El sistema pide la clave de identificación.
3. El cliente introduce la clave.
4. El sistema presenta las opciones disponibles.
5. El cliente selecciona la operación de Retiro.
6. El sistema pide la cantidad a retirar.
7. El cliente introduce la cantidad requerida.
8. El sistema procesa la petición y, eventualmente, da el dinero solicitado.
9. El sistema devuelve la tarjeta y genera un recibo.
10. El cliente recoge el dinero, el recibo y la tarjeta. Procede a retirarse

Ejemplo

calcular los puntos de casos de uso no ajustados (UUCP):

PESO ACTORES	
Actores	Tipo
Cliente	Complejo
UAW	

PESO CASOS DE USO	
Caso de uso	Tipo
Retirar dinero cajero	Promedio
UUCW	

$$\text{UUCP} = \text{UAW} + \text{UUCW} = 3 + 10 = 13$$

Ejemplo

calcularemos los puntos de caso de uso ajustados (UCP)

FACTORES TÉCNICOS EJEMPLO				
Factor	Descripción	Peso	Nivel	Pes*Niv
T1	Sistema distribuido	2	3	6
T2	Performance o tiempos de respuesta	1	5	5
T3	Eficiencia del usuario final	1	1	1
T4	Procesamiento interno complejo	1	1	1
T5	Código reutilizable	1	0	0
T6	Facilidad de instalación	0.5	1	0.5
T7	Facilidad de uso	0.5	2	1
T8	Portabilidad	2	0	0
T9	Facilidad de cambio	1	1	1
T10	Concurrencia	1	5	5
T11	Seguridad	1	5	5
T12	Acceso directo a terceras partes	1	0	0
T13	Facilidades especiales	1	0	0
TFactor = \sum (nivel * peso)				25.5

El peso de los factores técnicos será:

$$\text{TCF} = 0.6 + (0.01 * \text{TFactor}) = 0.6 + (0.01 * 25.5) = \mathbf{0.855}$$

Ejemplo:

FACTORES AMBIENTALES EJEMPLO				
Factor	Descripción	Peso	Nivel	Pes*Niv
E1	Familiaridad con el modelo	1.5	2	3
E2	Experiencia en la aplicación	0.5	3	1.5
E3	Experiencia en orientación a objetos	1	5	5
E4	Capacidad del analista líder	0.5	5	2.5
E5	Motivación	1	5	5
E6	Estabilidad en los requerimientos	2	5	10
E7	Personal de medio tiempo	-1	5	-5
E8	Dificultad en el lenguaje de programación	-1	0	0
EFactor = $\sum(\text{Peso} * \text{Nivel})$				22

El peso de los factores ambientales lo calculamos de acuerdo a la fórmula propuesta:

$$EF = 1.4 + (-0.03 * EFactor) = 1.4 + (-0.03 * 22) = \mathbf{0.74}$$

Ejemplo :

Los resultados finales los mostramos en la siguiente tabla:

HORAS-HOMBRE TOTALES EJEMPLO		
Actividad	Porcentaje	Horas Hombre
Análisis	10%	41.12
Diseño	20%	82.25
Programación	40%	164.50
Pruebas	15%	61.68
Sobrecarga	15%	61.68
Total esfuerzo	100%	411.25

- Analizando la tabla, podemos ver que el proceso total de desarrollo del caso de uso de este ejemplo son **411.25 hrs**, el equivalente a **10.28 semanas** para una sola persona (trabajando tiempo completo a 40 hrs por semana).
- Para un equipo de desarrollo de **5 personas** con trabajo de tiempo completo, demoraría **2.05 semanas** en crearse un sistema con dichas características.
- Los costos de producción de sistema se calcular en función del número de horas trabajadas, multiplicando el costo deseado por hora y el total de ellas. Por ejemplo, si el costo por hora es de \$300, el costo total del sistema seria:
Total = 411.25 hrs * 300 = \$123,376.5
- Y el plazo de entrega seria de 2 semanas, con equipo de trabajo de 5 personas de tiempo completo.

Los tiempos estimados por etapa (análisis, diseño, programación, etc.), así como los costos por hora, son criterio del equipo de desarrollo o de la empresa encargada de crear el sistema y dependen en gran medida de la experiencia de ellos.

Ejemplo :

Los puntos de casos de uso (UCP) para el caso de uso “retirar dinero de cajero automático” son:

$$\text{UCP} = \text{UUCP} * \text{TCF} * \text{EF} = 13 * 0.855 * 0.74 = 8.225$$

- Tomando en consideración la propuesta del creador de la técnica descrita en este trabajo, asignamos **20 hrs-hombre por punto de casos de uso**,
- dando un total de **164.50 Hrs-Hombre**,
- es decir, 4.11 semanas para una sola persona trabajando tiempo completo.