Ingeniería de Software I
Diseño de Componentes

Jocelyn Simmonds

Departamento de Ciencias de la Computación

Hacia el diseño

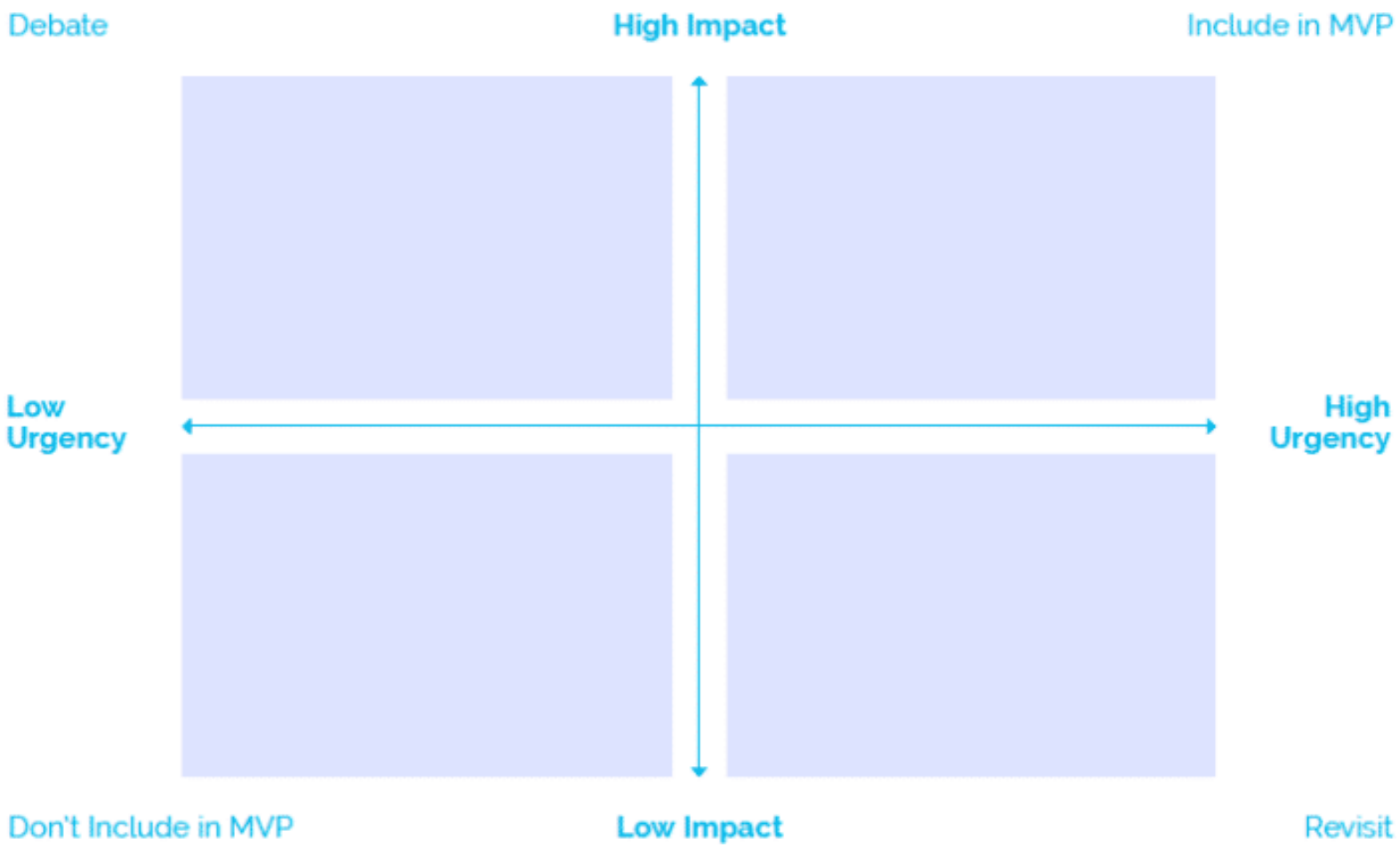
- Análisis: el objetivo es entender
  - Eventos y procesos del negocio
  - Actividades del sistema y requisitos de procesamientos
  - Objetivos de almacenamiento de información
- Diseño: el objetivo es definir, organizar y estructurar
  - Componentes del sistema solución final que servirá como modelo para la construcción
- Diseño y Análisis son iterativos

Minimum Viable Product (MVP)

Hoy se habla mucho de MVPs. Un MVP:

- incluye las funcionalidades necesarias para resolver un problema fundamental de un grupo de usuarios
- puede ser lanzado rápidamente al mercado, para obtener feedback de los usuarios
- permite testear/corregir supuestos que hemos hecho acerca de requisitos, procesos de negocio, etc.
  - ej: nuestra solución depende fuertemente del wifi . . . pero nuestros usuarios tienen acceso limitado a wifi/datos

Planificando un MVP



Colocar cada requisito, casos de uso, . . . en la caja correspondiente

Diseño

- Diseño: proceso creativo de derivar una solución a partir de un problema
  - Literatura tradicional: "convertir" problema en solución
  - Pero eso niega aspecto realmente creativo
- Una característica:
  - En general, no hay una solución
  - En general, no hay una solución "mejor"
  - Evaluación y elección depende del cliente
- Punto de partida
  - Entregables generados en el Análisis

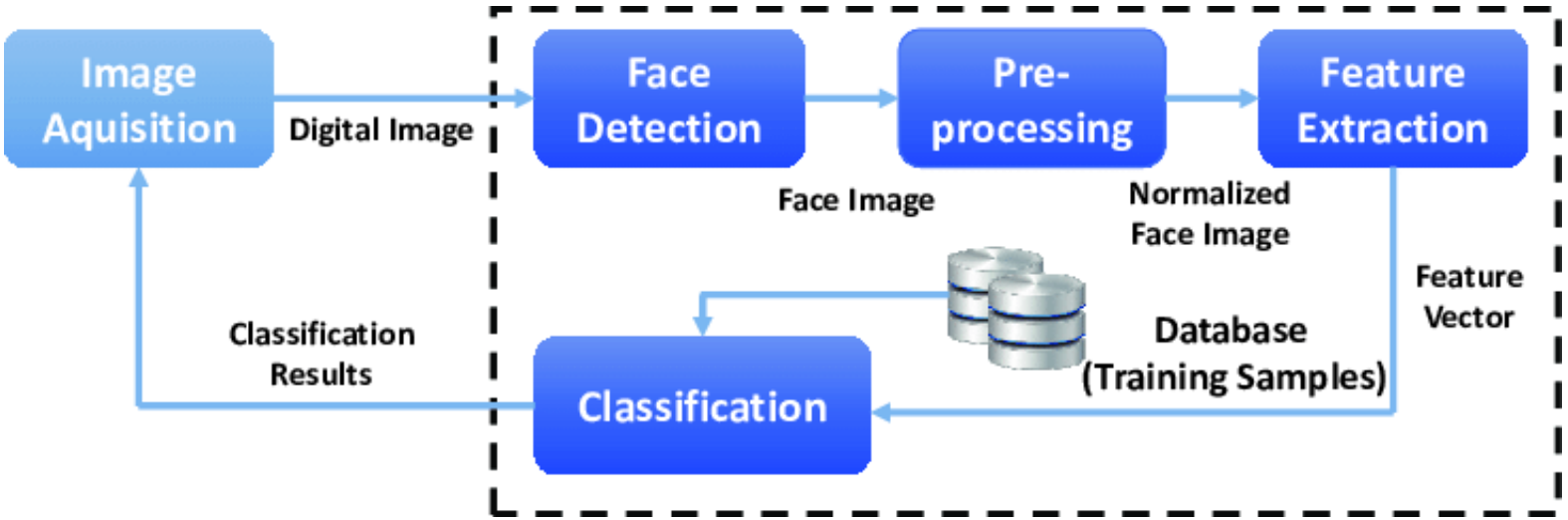
Tipos de diseño

1. Diseño arquitectural (de alto nivel)
   - Explicar al cliente/usuario que tipo de SW se construirá
   - Componentes de alto nivel (reuso de componentes, SaaS), arquitectura de aplicación (capas, SOA, ...), mecanismos para cumplir NFRs (replicación, protocolos de seguridad, cloud, ...)
2. Diseño detallado (de bajo nivel)
   - Explicar a los desarrolladores cómo construir el SW (clases, patrones GOF, ...)
3. Ambos diseños explican el mismo sistema ...
   - ... pero de diferente manera, con audiencias diferentes

Hoy hablaremos de un nivel intermedio: componentes ...

Pensando en cómo "organizar" una aplicación

Diagrama informal:



Problema con este tipo de diagrama: no hay notación clara . . . qué representan las cajas, flechas, estilos de lineas, etc.?

Principios de diseño

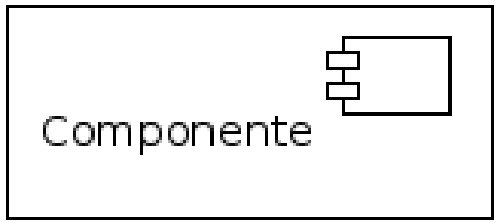
- Acoplamiento
  - Medida cualitativa de cuán relacionadas están las clases dentro de un diagrama de clases de diseño
- Cohesión
  - Medida cualitativa de la consistencia de funciones dentro de una sola clase en un diagrama de clases de diseño

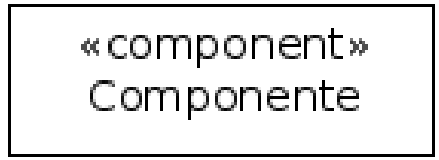
Objetivo

Alta cohesion, bajo acoplamiento

Notación

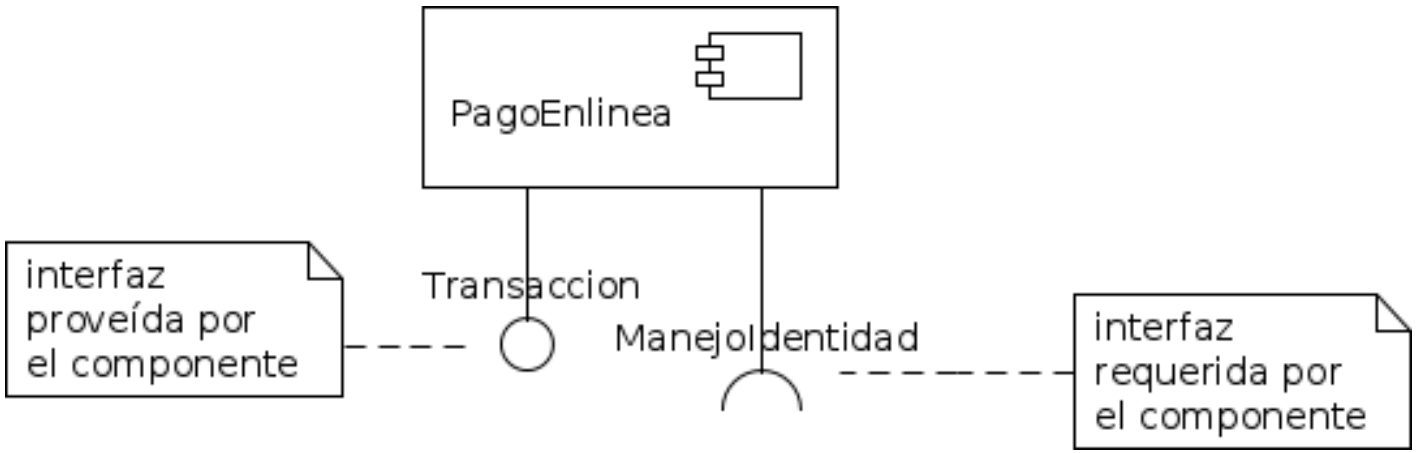
Un componente es una parte de un sistema, como un archivo (clase) compilado, una parte del código de la aplicación, una librería compartida, un EJB, etc.





Notación

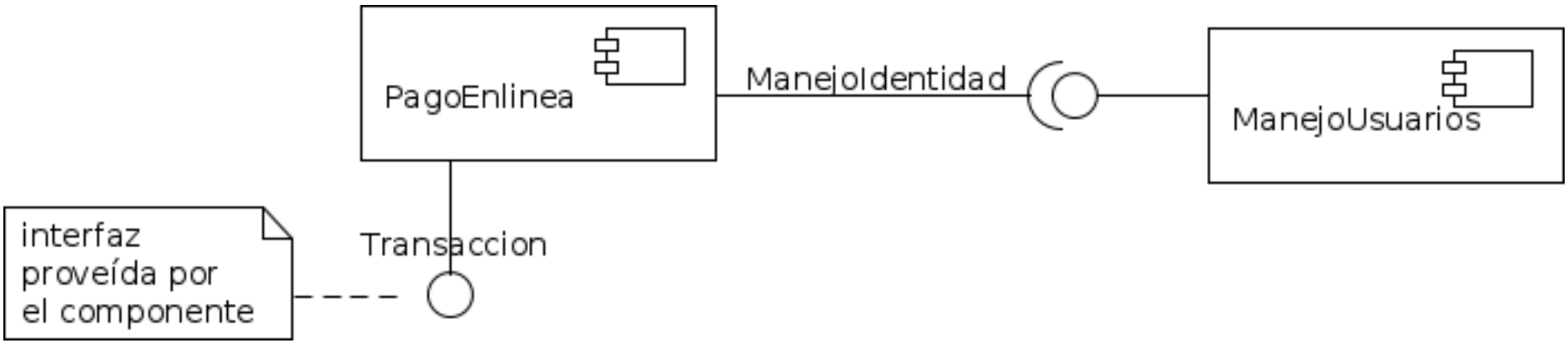
Los componentes proveen interfaces a otros componentes, y además pueden requerir ciertas interfaces para cumplir sus tareas.



- interfaz proveída: conjunto de atributos y métodos públicos implementados en alguna de las clases de la componente
- interfaz requerida: conjunto de atributos y métodos públicos que el componente requiere para funcionar

Juntando componentes

Pueden conectar componentes para formar subsistemas.

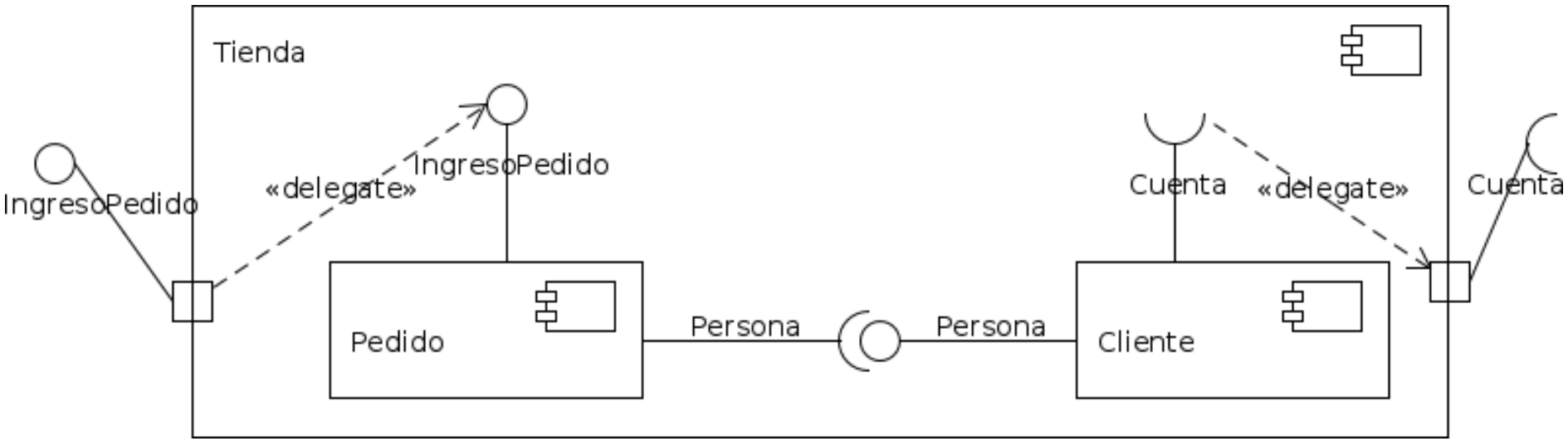


Juntando componentes

Pueden mostrar dependencias entre componentes y sus interfaces:

La "cajita" en el ComponenteA es un puerto. Esto significa que el ComponenteA en sí mismo no proporciona las interfaces necesarias (sean proveídas o requeridas), y que delega la implementación de estas a una clase interna o subcomponente.

Modelando puertos



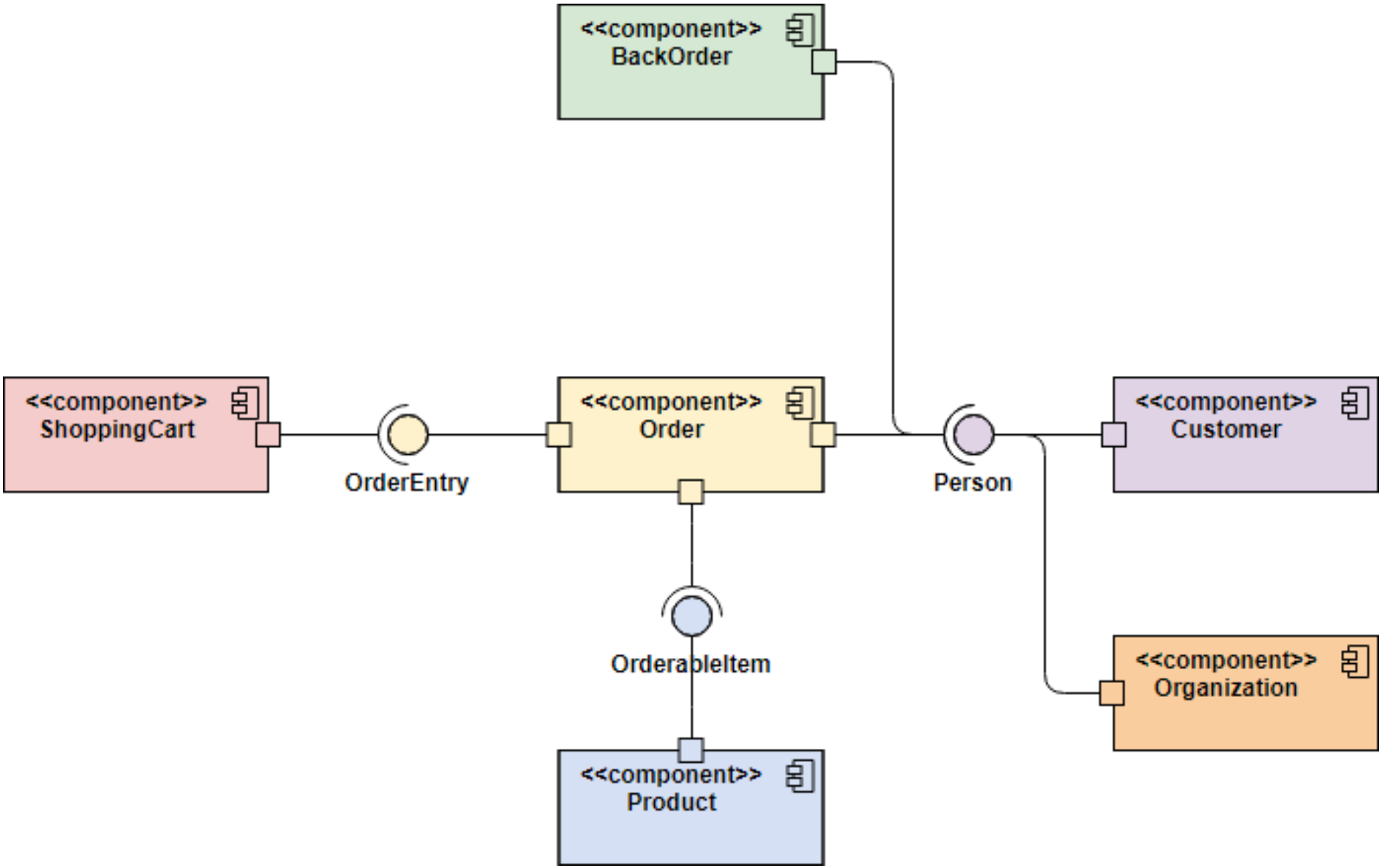
En este ejemplo, el componente Tienda:

- ofrece la interfaz IngresoPedido, la que es proveída por el subcomponente Pedido, y
- requiere una interfaz Cuenta, la que es requerida por el subcomponente Cliente.

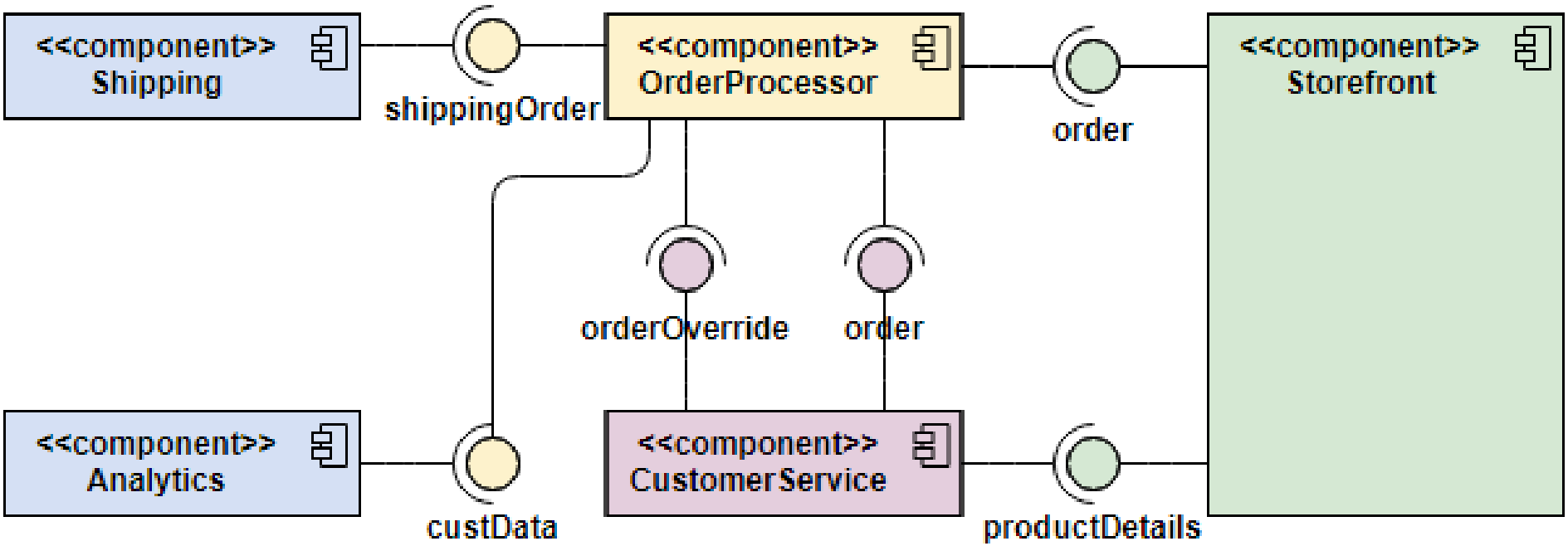
Guía para diseñar componentes

- Mantener componentes con alta cohesión
- Separa tipos distintos de clases en distintos componentes
  - Clases de interfaz de usuario
  - Clases técnicas (middleware, seguridad, persistencia)
- Dejar clases altamente acopladas en un mismo componente
- Identificar componentes de dominio
- Separar por la forma de colaboración
  - Clientes vs servidores
- Minimizar el flujo de mensajes entre componentes

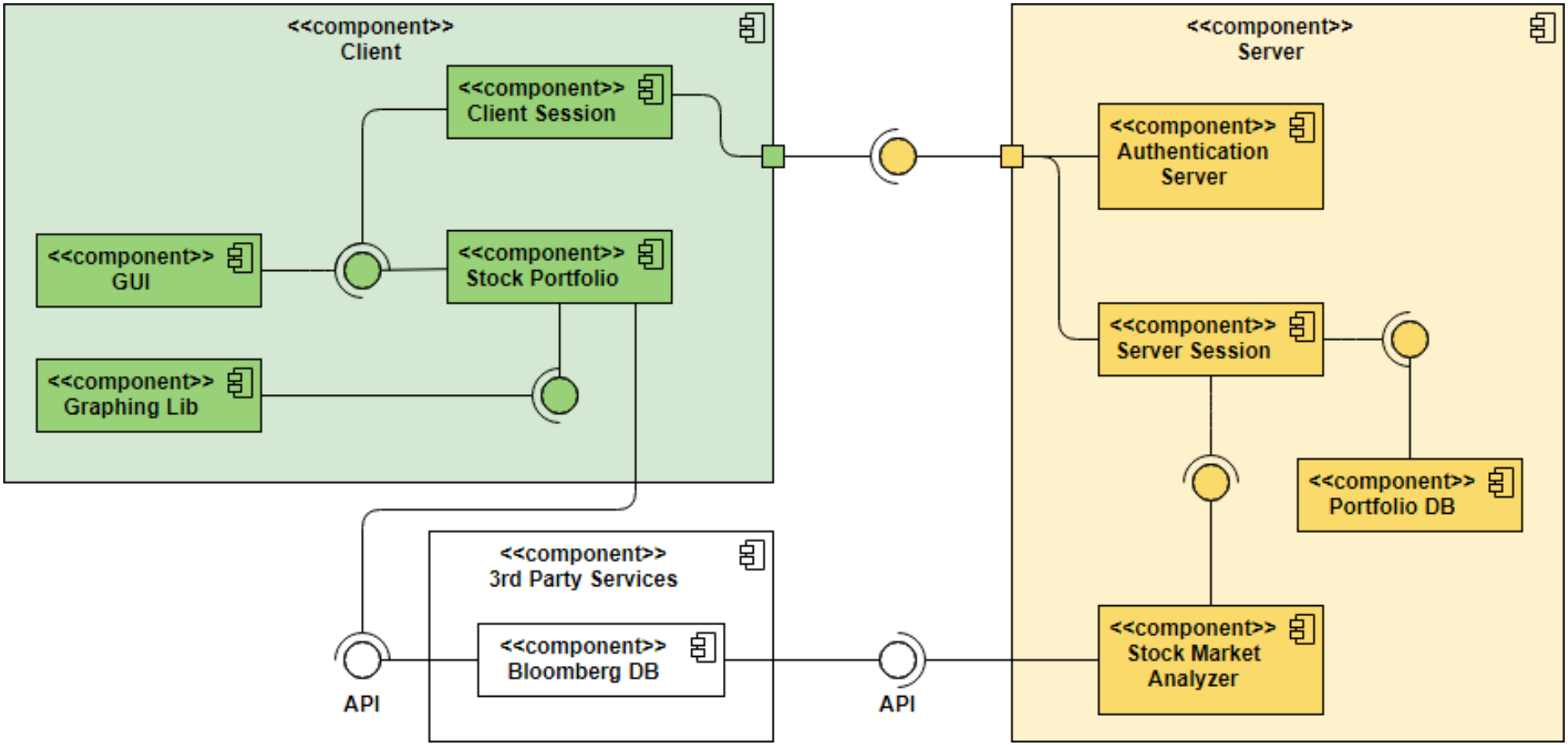
Ejemplos


<<component>>
BackOrder
<<component>>
ShoppingCart
OrderEntry
<<component>>
Order
Person
<<component>>
Customer
OrderableItem
<<component>>
Organization
<<component>>
Product

Ejemplos


<<component>>
Shipping
shippingOrder
<<component>>
OrderProcessor
order
<<component>>
Storefront
orderOverride
order
<<component>>
Analytics
custData
<<component>>
CustomerService
productDetails

Ejemplos


<<component>>
Client
<<component>>
Client Session
<<component>>
GUI
<<component>>
Stock Portfolio
<<component>>
Graphing Lib
<<component>>
3rd Party Services
<<component>>
Bloomberg DB
API
API
<<component>>
Server
<<component>>
Authentication Server
<<component>>
Server Session
<<component>>
Portfolio DB
<<component>>
Stock Market Analyzer

Principios del Diseño de Componentes

Dependencias acíclicas: No permitir ciclos en el grafo de dependencias entre componentes. Por ejemplo, prohibir dependencias como A $\dashrightarrow$ B $\dashrightarrow$ C $\dashrightarrow$ A porque introduce un ciclo.

Cerradura común: (common closure) Las clases agrupadas en un componente deben ser cohesivas, "cerradas" juntas contra el mismo tipo de cambios. Un cambio que afecta a una clase dentro de un componente no debe afectar a clases fuera de ese componente.

Reuso común: Las clases en un componente se reutilizan juntas. Si vuelve a utilizar una clase en un componente, entonces se reutilizan todas. Este principio también esta enfocado en mejorar la cohesión del sistema.

Principios del Diseño de Componentes

Inversión de dependencias: Las abstracciones no deben depender de los detalles de implementación, los detalles deben depender de las abstracciones

Abierto-Cerrado: Los elementos del sistema deben estar abiertas para la extensión pero cerrado para la modificación. O sea, una vez completada la implementación de un sistema, una clase solo debe ser modificada para corregir un error – features nuevos deben ser agregados con la creación de nuevas clases.

Equivalencia Release-Reuso: Componentes no debe ser reutilizadas por partes – las clases que se "liberan" (release) juntas deben ser reutilizadas juntas.

Principios del Diseño de Componentes

Abstracciones estables: Un componente debe ser tan abstracto como es estable. Un componente debe ser suficientemente abstracto de modo que se pueda extender sin afectar su estabilidad.

Dependencias estables: Si el componente A depende de componente B, entonces B deberia ser más estable (por ejemplo, menos probable que cambie) que A.

Diagrama de Paquetes

- Diagrama UML de alto nivel que asocia clases a grupos relacionados
- Identifica principales componentes de un sistema y dependencias
  - Dependencias implican impacto por cambios
- Determina particiones del programa por cada capa
- Puede dividir el sistema en subsistemas y anidar paquetes

Diagrama de Paquetes

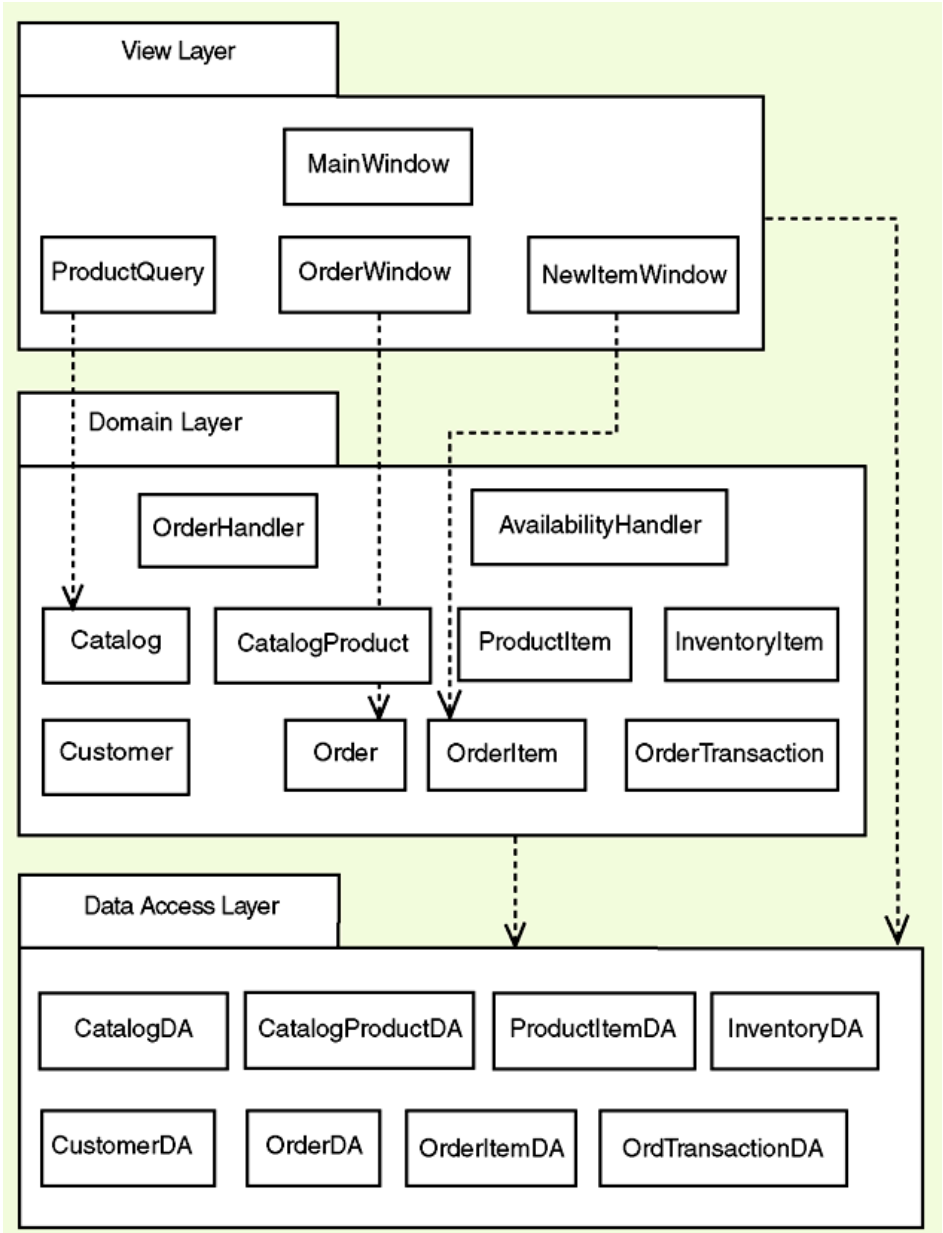


Diagrama de Paquetes