IN77J – Orientación al Objeto para el e-business

6. Diseño

Temario

- 6. Diseño
  - Descomposición
  - Realización de Casos de Uso
  - Taller
  - Patrones de Diseño

Descomposición

- Una de las principales técnicas para abordar el diseño de sistemas grandes (programming in the large) es la descomposición
- Existe una antigua regla basada en la descomposición: dividir para reinar
- La descomposición ayuda porque es mucho más simple abordar varios problemas pequeños, independientes entre sí, que un problema grande

Descomposición

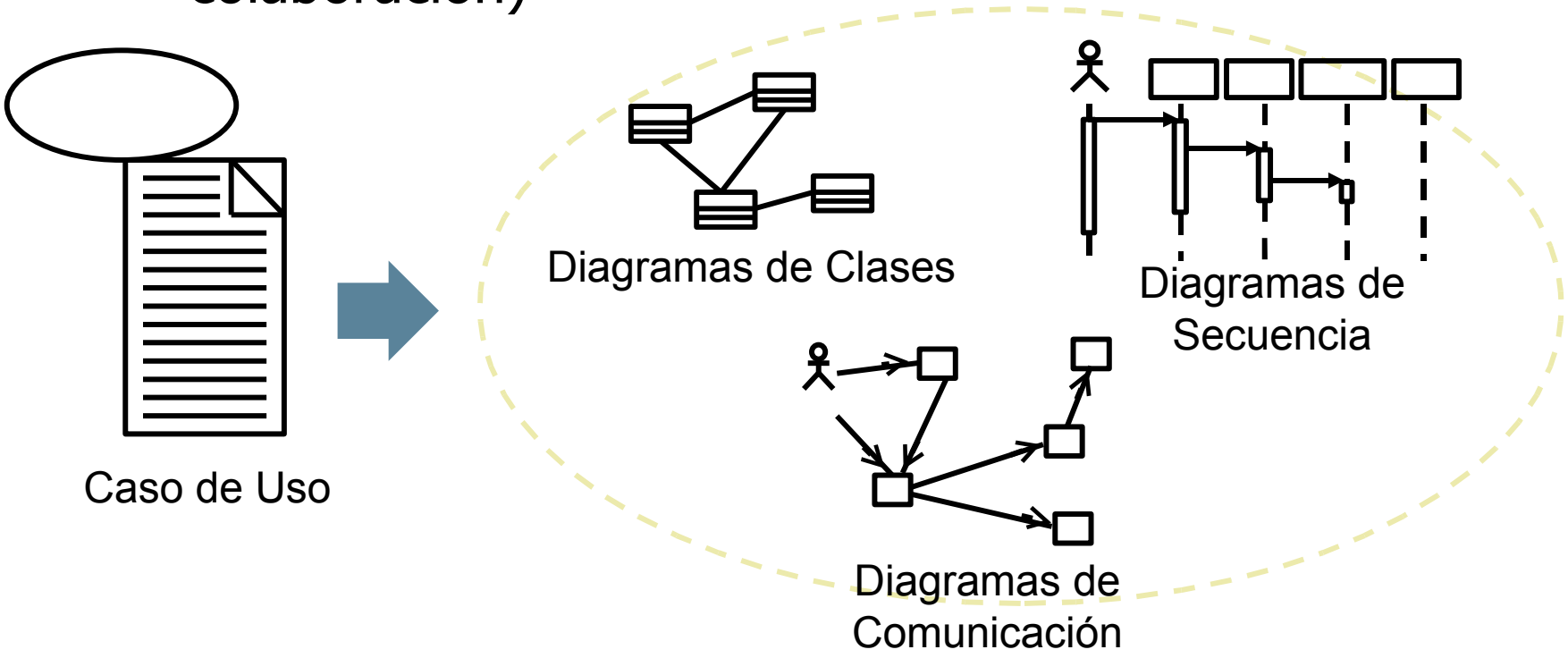
- En programación procedural, es común utilizar el esquema top-down, dividiendo el sistema en rutinas, éstas en subrutinas, y así sucesivamente
- En orientación a objetos, el sistema se descompone en clases (que a su vez se agrupan en paquetes), y la funcionalidad se encuentra dentro de ellas
- Un problema central, por lo tanto, es la identificación de las clases

Modularidad

- Se habla de modularidad para indicar que un sistema ha sido bien descompuesto
- Según Edward Yourdon, una buena descomposición es aquella en que los módulos presentan alta cohesión y bajo acoplamiento
- Beneficios de las técnicas de orientación a objetos:
  - El concepto de clase incrementa la cohesión
  - Se reduce el acoplamiento:
    - ocultamiento de información
    - dependencia de interfaces

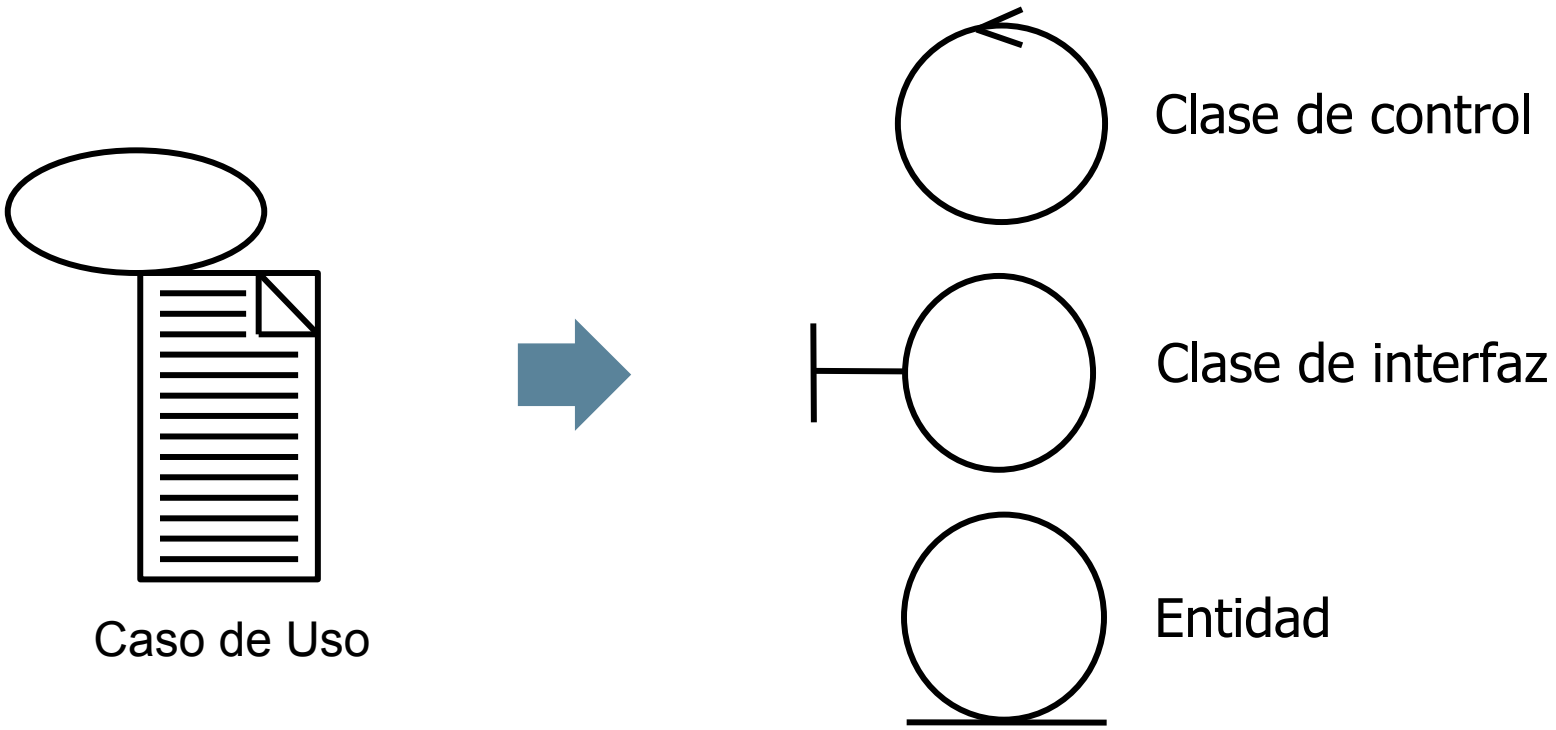
Realización de Casos de Uso

- RUP define el concepto de realización de casos de uso, que corresponde al diseño del caso de uso, utilizando diagramas de clases, de secuencia, y de comunicación (antiguamente llamados de colaboración)



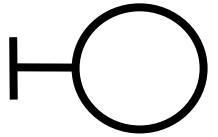
Realización de Casos de Uso

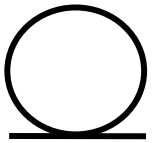
- A partir del comportamiento definido en el caso de uso, es necesario identificar las clases (de control, de interfaz, entidades)

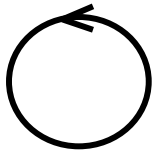


Tipos de Clases

- Las clases de interfaz (boundary) comunican al sistema con elementos externos (un usuario, otro sistema, un dispositivo, etc.)
- Las entidades son las abstracciones clave del sistema, a menudo son persistentes
- Las clases de control coordinan el comportamiento definido en los casos de uso

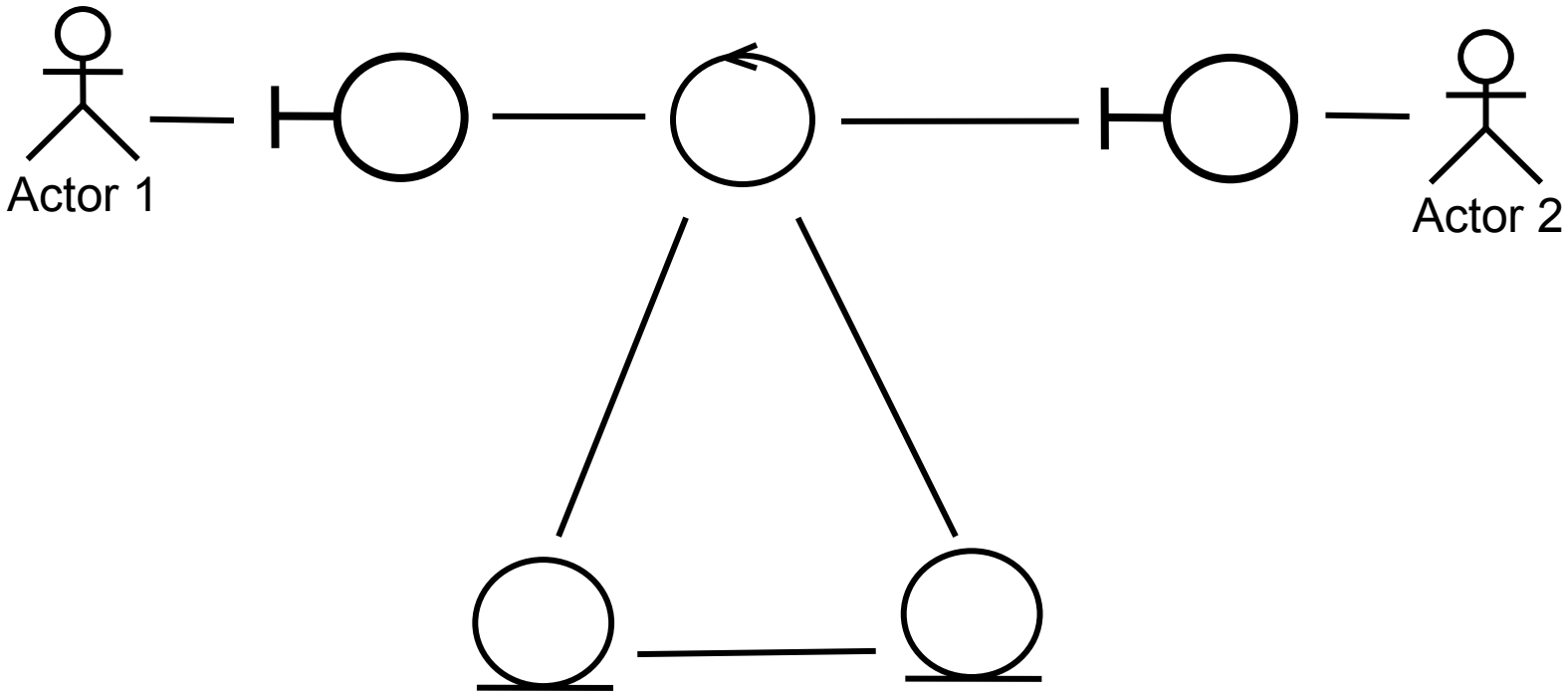






Identificación de Clases

- Descomposición inicial en clases de análisis sugerida por RUP:
  - Una clase de control por caso de uso
  - Una clase de interfaz por cada par {caso de uso, actor}
  - Entidades que se requieran



Identificación de Entidades

- Las entidades son las abstracciones clave del caso de uso
- Una recomendación tradicional para la identificación de las entidades es leer la documentación (especificación, caso de uso, etc.)
  - Algunos sustantivos podrían corresponder a entidades o actores
  - Los adjetivos podrían corresponder a atributos
  - Los verbos podrían corresponder a métodos

Identificación de Responsabilidades

- Una tarea de análisis/diseño es identificar las responsabilidades del caso de uso, y asignarlas a las clases
- En este proceso, es posible que aparezcan clases no contempladas inicialmente
- Las responsabilidades deben describirse con diagramas de interacción cuando su complejidad lo amerite
  - Diagramas de secuencia: más cómodos para visualizar la secuencia de los mensajes en el tiempo
  - Diagramas de comunicación: además del flujo de mensajes, muestran las relaciones entre los objetos, y son más cómodos para las sesiones de trabajo grupal

Modelo de Clases

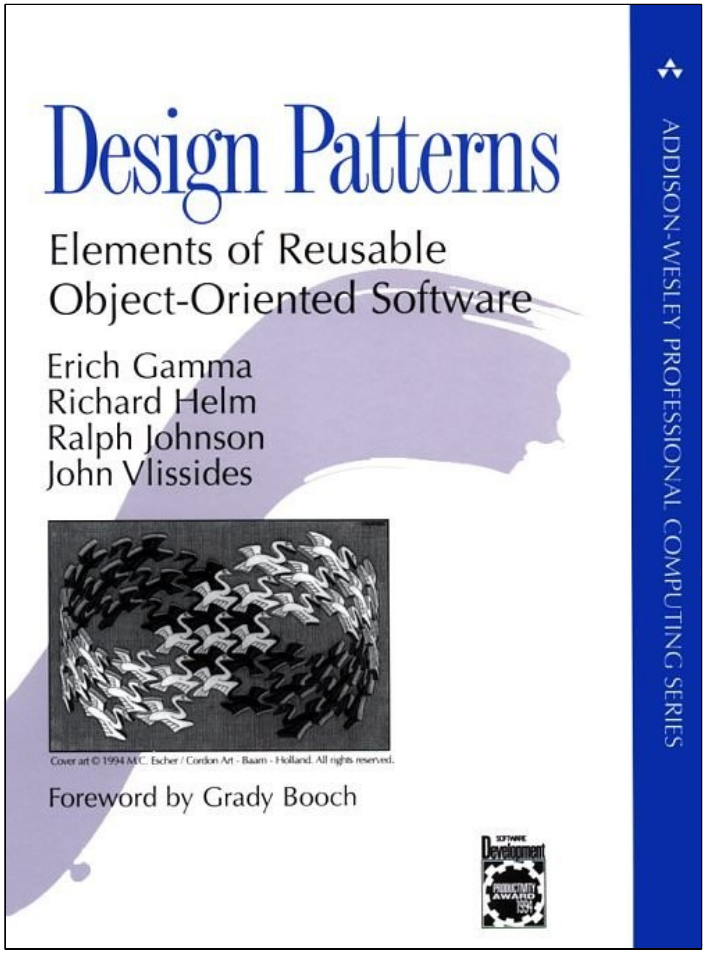
- El modelo de clases para el caso de uso se describe con un diagrama de clases, que contiene las clases definidas, sus atributos, sus métodos, y sus relaciones

Patrones de Diseño

- Un **patrón de diseño** es una solución a un problema tipo, que se presenta en diferentes situaciones
- Aproximación práctica al Diseño
- Forma de organizar el conocimiento adquirido en Diseño, con vistas a la reutilización

La Fuente

- Design Patterns (94):
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides



Patrones de Diseño (Gamma)

Creación	Estructura	Comportamiento
Abstract Factory	Adapter	Chain of Responsibility
Builder	Bridge	Command
Factory Method	Composite	Interpreter
Prototype	Decorator	Iterator
Singleton	Façade	Mediator
	Flyweight	Memento
	Proxy	Observer
		State
		Strategy
		Template Method
		Visitor

Elementos de un Patrón

- Un nombre, de una o dos palabras
- El problema describe cuándo el patrón es aplicable
- La solución describe los elementos que forman el diseño, sus relaciones, responsabilidades y colaboraciones
- Las consecuencias son los resultados y trade-offs de la aplicación del patrón

Patrones de Creación

Patrón	Objetivo
Abstract Factory	Proveer una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas.
Builder	Separar la construcción de un objeto complejo de su representación, de modo que el mismo proceso de construcción pueda crear diferentes representaciones.
Factory Method	Definir una interfaz para crear un objeto, permitiendo que subclases decidan qué clase instanciar.
Prototype	Especificar los tipos de objetos a crear utilizando una instancia como prototipo, y crear objetos copiando esta instancia.
Singleton	Asegurar que una clase tiene sólo una instancia, y proveer un punto global de acceso a ella.

Patrones de Estructura

Patrón	Objetivo
Adapter	Convertir la interfaz de una clase en otra interfaz que el cliente espera.
Bridge	Desacoplar una abstracción de su implementación, de modo que ambas puedan variar independientemente.
Composite	Componer objetos en estructuras de tipo árbol para representar jerarquías todo-partes.
Decorator	Agregar responsabilidades adicionales a un objeto de manera dinámica.
Façade	Proveer una interfaz unificada a un conjunto de interfaces en un subsistema.
Flyweight	Compartir objetos de modo de soportar un número grande de objetos cliente de manera eficiente.
Proxy	Proveer un "surrogate" o "placeholder" que controle el acceso a un objeto, de modo de instanciar el objeto requerido sólo cuando se vaya a utilizar.

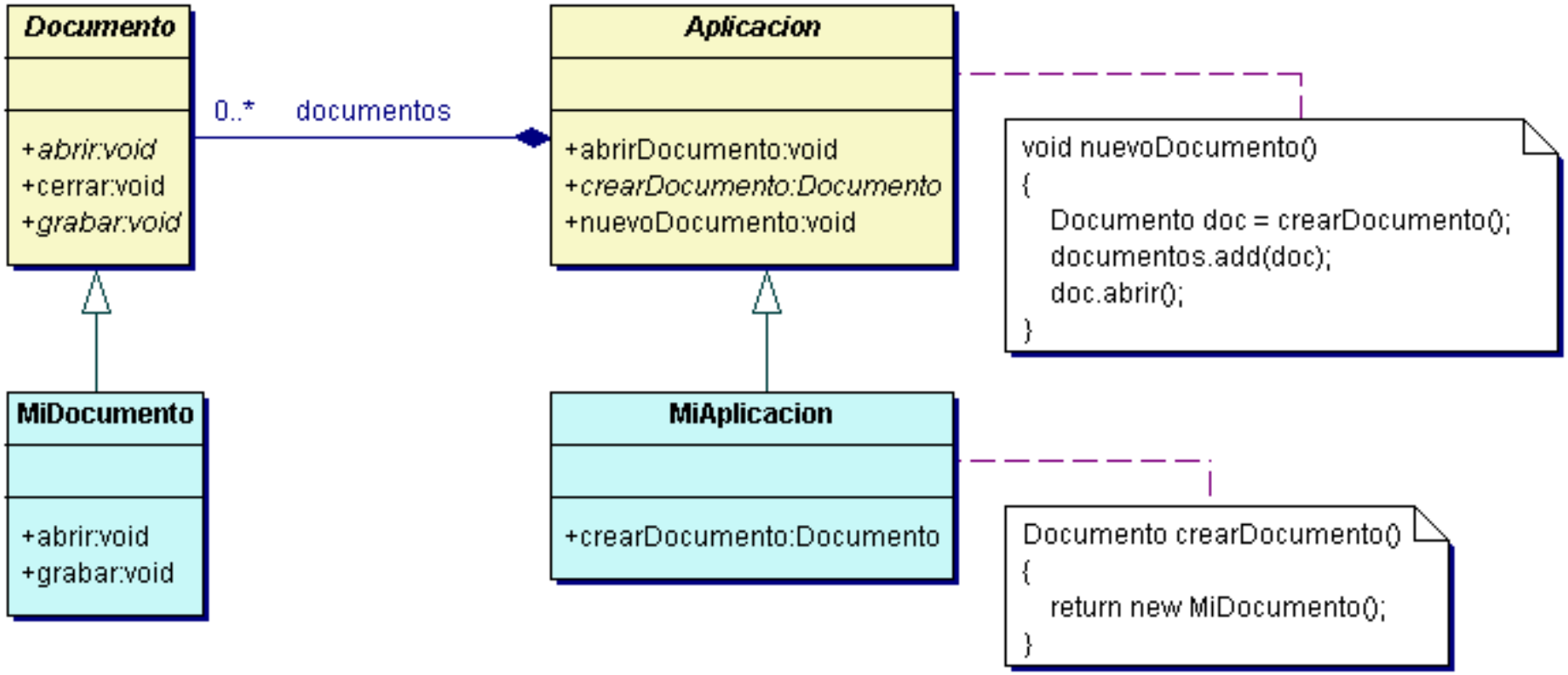
Patrones de Comportamiento

Patrón	Objetivo
Chain of Responsibility	Evitar el acoplamiento entre el remitente de un requerimiento y su receptor, dando a varios objetos la posibilidad de atender el requerimiento.
Command	Encapsular un requerimiento como un objeto.
Interpreter	Dado un lenguaje, definir la representación de su gramática en conjunto con un intérprete que use la representación para interpretar sentencias en el lenguaje.
Iterator	Proveer un mecanismo de acceso secuencial a los elementos de un objeto agregado.
Mediator	Definir un objeto que encapsula la manera en que un conjunto de objetos interactúa.
Memento	Sin violar la encapsulación, capturar y externalizar el estado interno de un objeto, de modo de poder restablecer el estado posteriormente.
Observer	Definir una dependencia uno a muchos entre objetos, de modo que cuando uno cambie de estado, los objetos dependientes sean notificados y actualizados automáticamente.
State	Permitir a un objeto modificar su comportamiento cuando su estado interno cambie. El objeto se verá como si hubiese cambiado su clase.
Strategy	Definir una familia de algoritmos, y hacerlos intercambiables.
Template Method	Definir el esqueleto de un algoritmo, difiriendo algunos pasos a subclases.
Visitor	Representar una operación a realizar sobre los elementos de una estructura de objetos.

Factory Method

- Objetivo
  - Definir una interfaz para crear un objeto, permitiendo que subclases decidan qué clase instanciar
- Aplicabilidad
  - Aplicable cuando una clase no puede anticipar la clase de objetos que debe crear
  - Aplicable cuando una clase desea que sus subclases especifiquen los objetos que crea

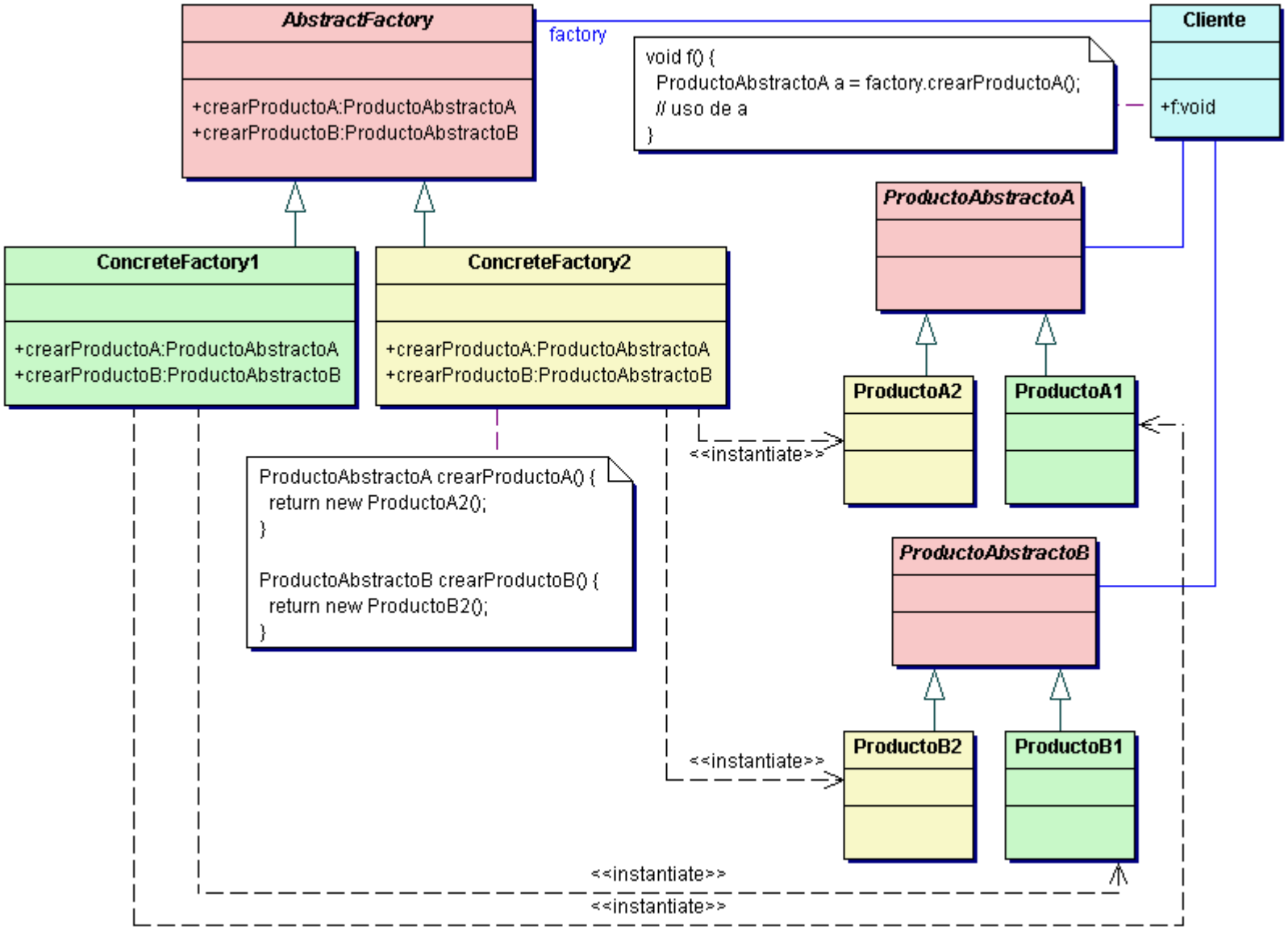
Factory Method



Abstract Factory

- Objetivo
  - Proveer una interfaz para crear familias de objetos relacionados sin especificar sus clases concretas
- Aplicabilidad
  - Aplicable en sistemas que deben ser independientes de cómo sus objetos son creados
  - Aplicable en sistemas en que una familia de objetos relacionados ha sido diseñada para ser usada en conjunto

Abstract Factory



Singleton

- Objetivo
  - Asegurar que existe sólo una instancia de una clase, y proveer acceso a ella
- Aplicabilidad
  - Aplicable cuando sólo debe haber una instancia de una clase, y ella debe ser accesible por parte de clientes
  - Aplicable cuando la única instancia debe ser extensible mediante herencia

Singleton

MiSingleton.java

```
public class MiSingleton {
    // constructor privado impide instanciación
    // (en ocasiones protegido, para permitir extensión)
    private MiSingleton() {}

    // el singleton
    private static MiSingleton instancia = null;

    // el punto de acceso
    public static MiSingleton get() {
        if (instancia == null) {
            instancia = new MiSingleton();
        }
        return instancia;
    }

    // método de negocio
    public void f() {
        …
    }
}
```

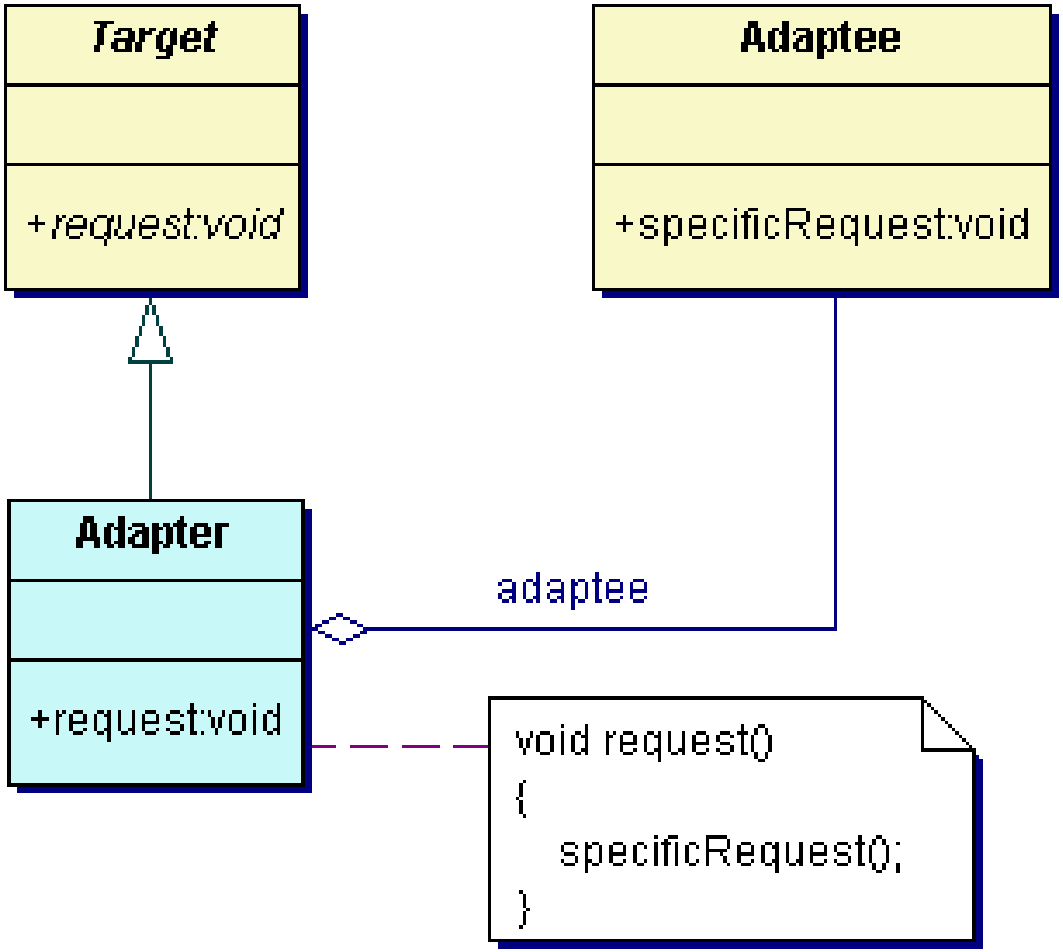
MiCliente.java

```
public class MiCliente {
    public void miMetodo() {
        …
        MiSingleton.get().f();
        …
    }
}
```

Adapter

- Objetivo
  - Convertir la interfaz de una clase en otra interfaz que el cliente espera
- Aplicabilidad
  - Aplicable cuando se desea utilizar una clase que existe, pero su interfaz no calza con la que se requiere

Adapter

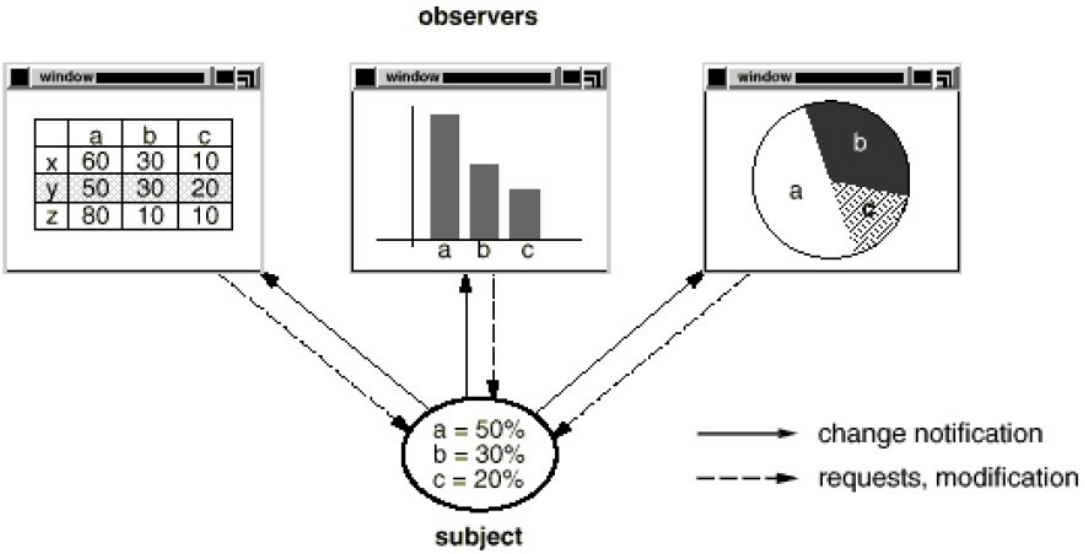

Target
+request:void
Adaptee
+specificRequest:void
Adapter
+request:void
adaptee
void request()
{
specificRequest();
}

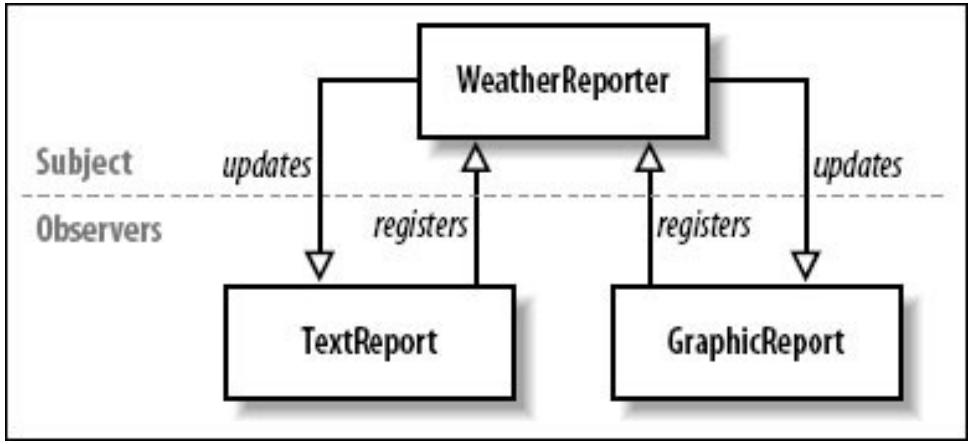
Observer

- Objetivo
  - Definir una dependencia “uno a muchos” entre objetos, de modo que cuando se modifique el estado de un objeto los objetos dependientes sean notificados y se actualicen automáticamente
- Aplicabilidad
  - Aplicable cuando se desea que un objeto notifique sus cambios a otros objetos, de manera desacoplada

Observer

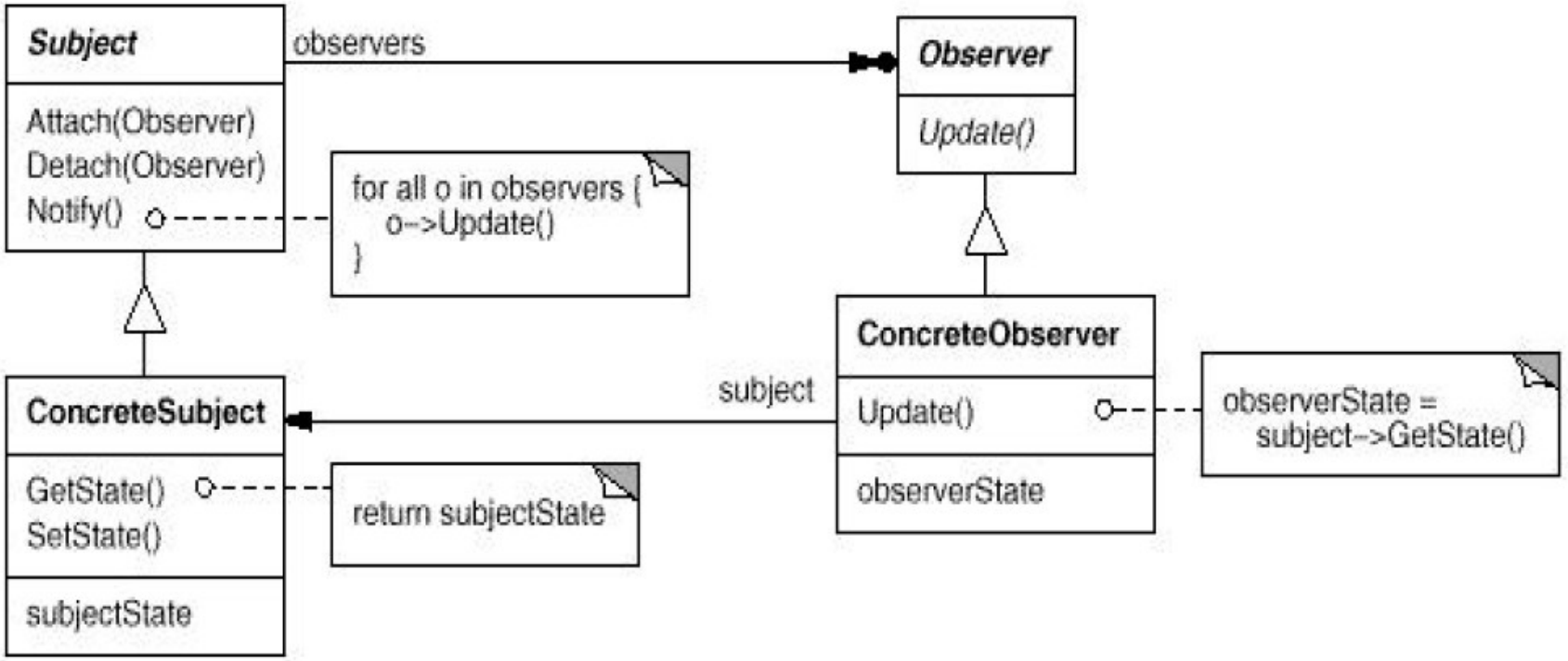
- Ejemplos





Observer

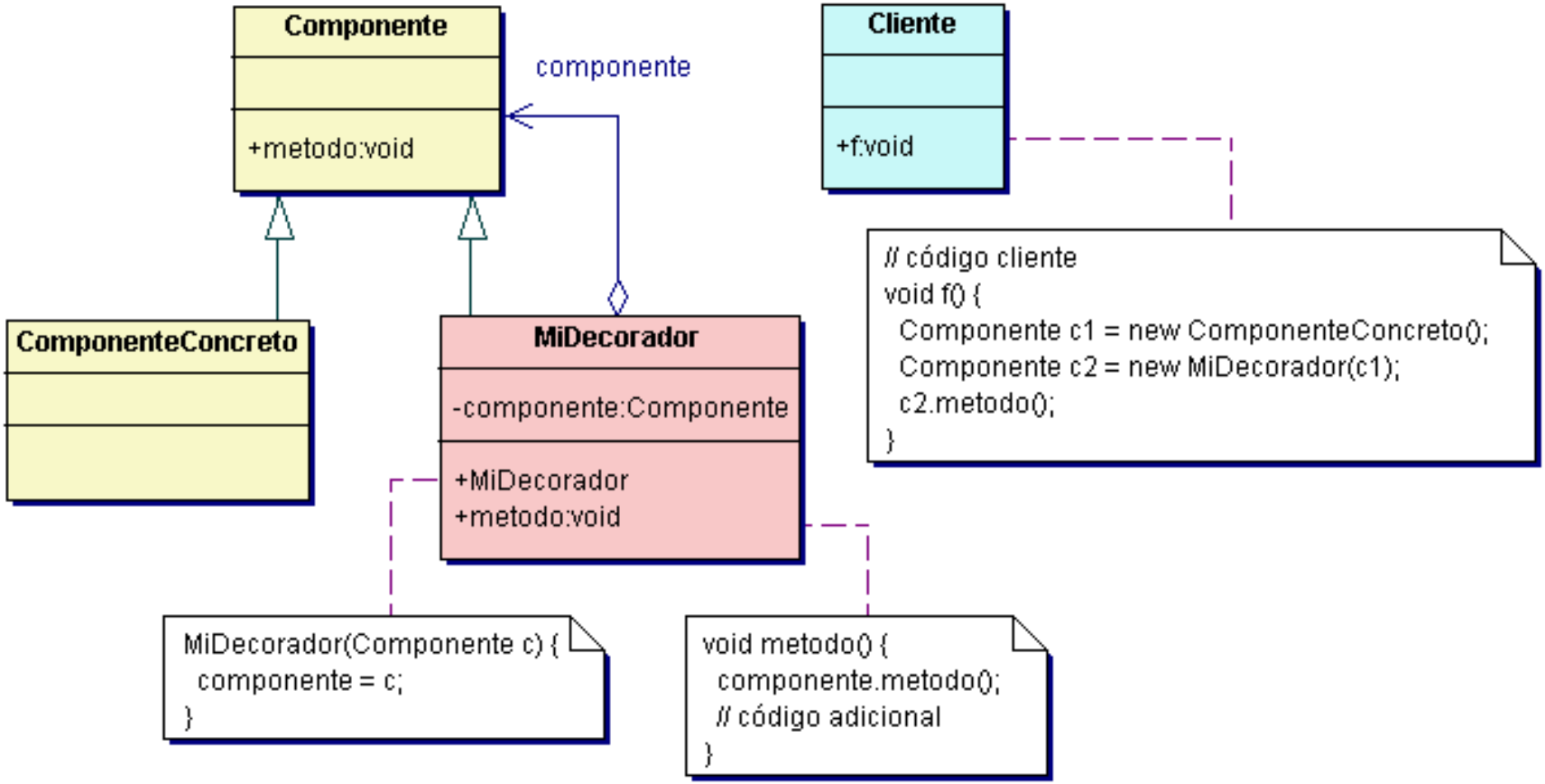
- Estructura



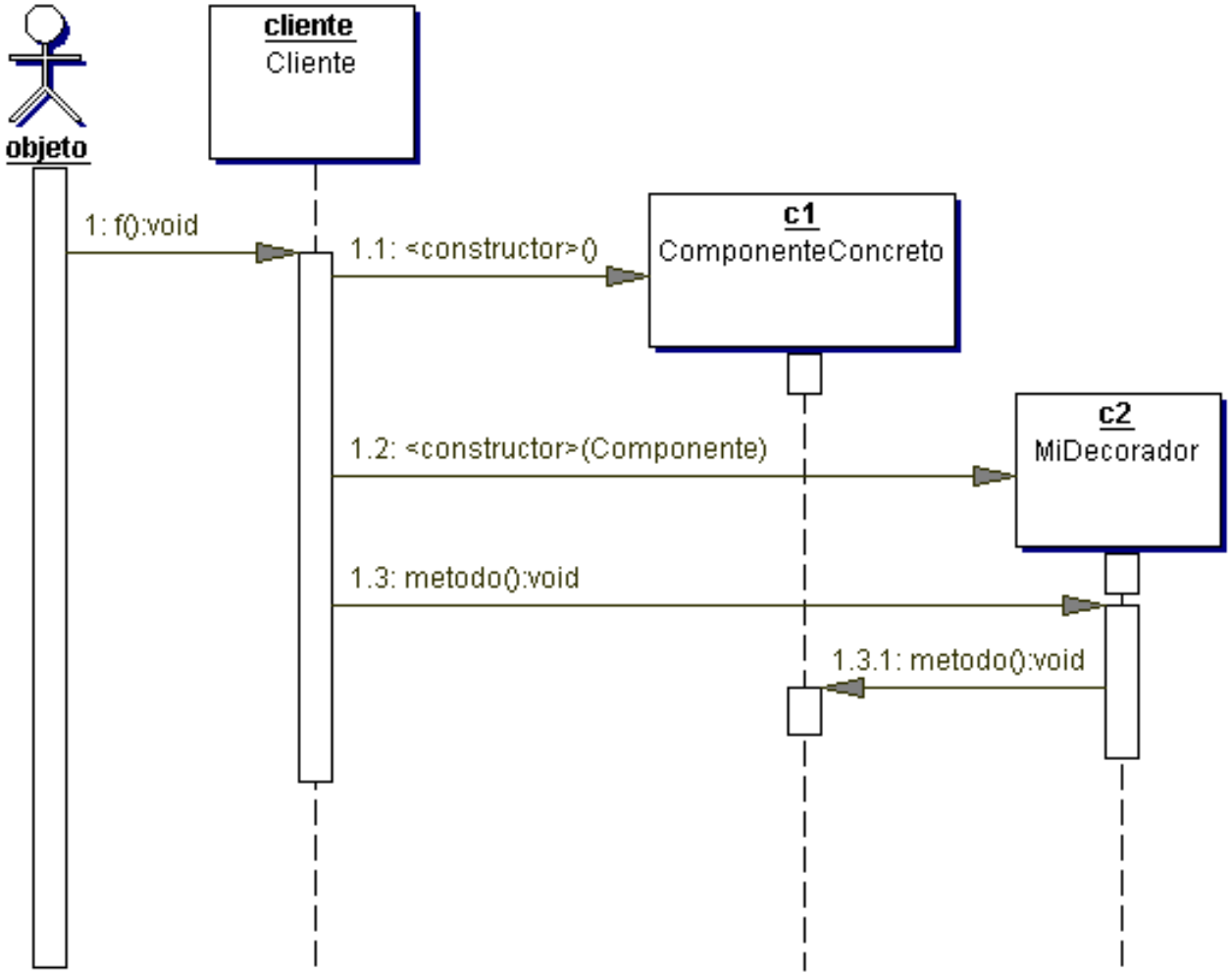
Decorator

- Objetivo
  - Agregar responsabilidades adicionales a un objeto de manera dinámica; constituye una alternativa flexible a extender la funcionalidad mediante herencia
- Aplicabilidad
  - Util cuando se desea agregar responsabilidades a objetos individuales de manera dinámica
  - Util cuando extensión mediante herencia resulta poco práctico, porque se tiene un número grande de extensiones independientes que generarían muchas subclases para soportar todas las combinaciones

Decorator


Componente
+metodo:void
componente
Cliente
+f:void
// código cliente
void f() {
Componente c1 = new ComponenteConcreto();
Componente c2 = new MiDecorador(c1);
c2.metodo();
}
ComponenteConcreto
MiDecorador
-componente:Componente
+MiDecorador
+metodo:void
MiDecorador(Componente c) {
componente = c;
}
void metodo() {
componente.metodo();
// código adicional
}

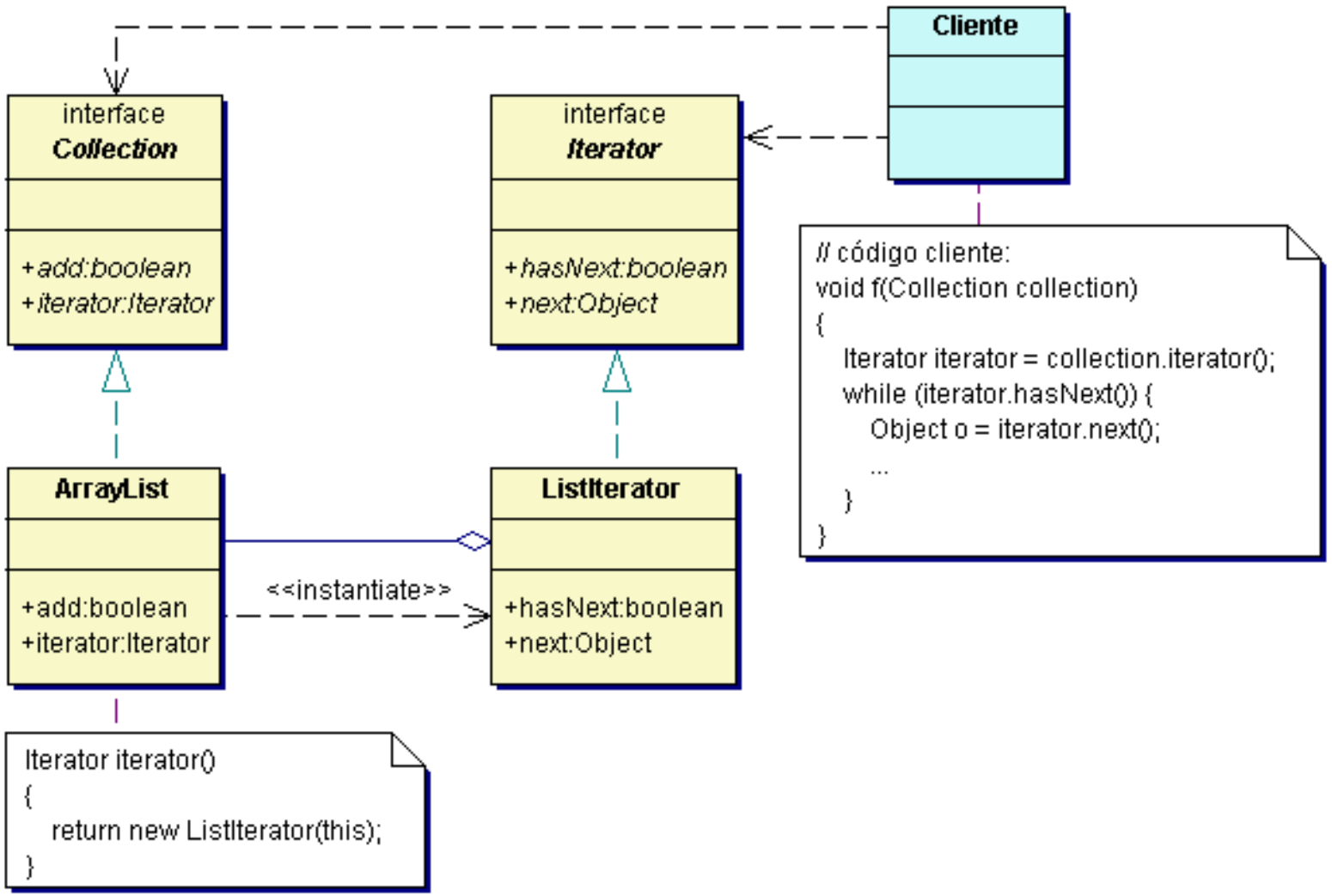
Decorator


objeto
cliente
Cliente
c1
ComponenteConcreto
c2
MiDecorador
1: f():void
1.1: <constructor>()
1.2: <constructor>(Componente)
1.3: metodo():void
1.3.1: metodo():void

Iterator

- Objetivo
  - Proveer un mecanismo para acceder a los elementos de un objeto agregado, de forma secuencial
- Aplicabilidad
  - Permite acceder a los elementos contenidos por el objeto agregado sin exponer su representación interna
  - Soporta múltiples recorridos de objetos agregados
  - Permite proveer una interfaz uniforme para recorrer diferentes estructuras agregadas

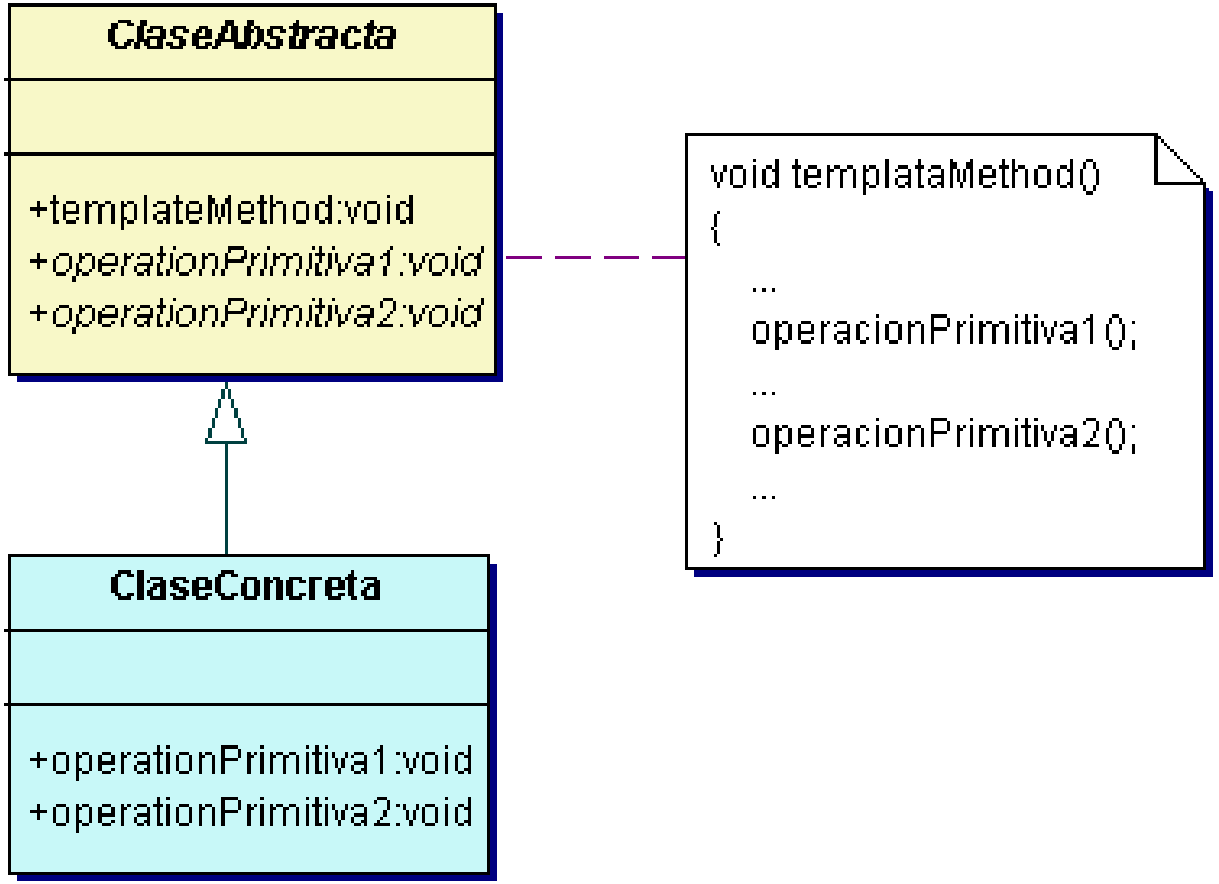
Iterator


Cliente
interface
Collection
+add:boolean
+iterator:Iterator
interface
Iterator
+hasNext:boolean
+next:Object
// código cliente:
void f(Collection collection)
{
Iterator iterator = collection.iterator();
while (iterator.hasNext()) {
Object o = iterator.next();
...
}
}
ArrayList
+add:boolean
+iterator:Iterator
<<instantiate>>
ListIterator
+hasNext:boolean
+next:Object
Iterator iterator()
{
return new ListIterator(this);
}

Template Method

- Objetivo
  - Definir el esqueleto de un algoritmo en una operación, difiriendo algunos pasos a subclases
- Aplicabilidad
  - Implementa la parte invariante de un algoritmo una vez, dejando a subclases la tarea de implementar el comportamiento que varía

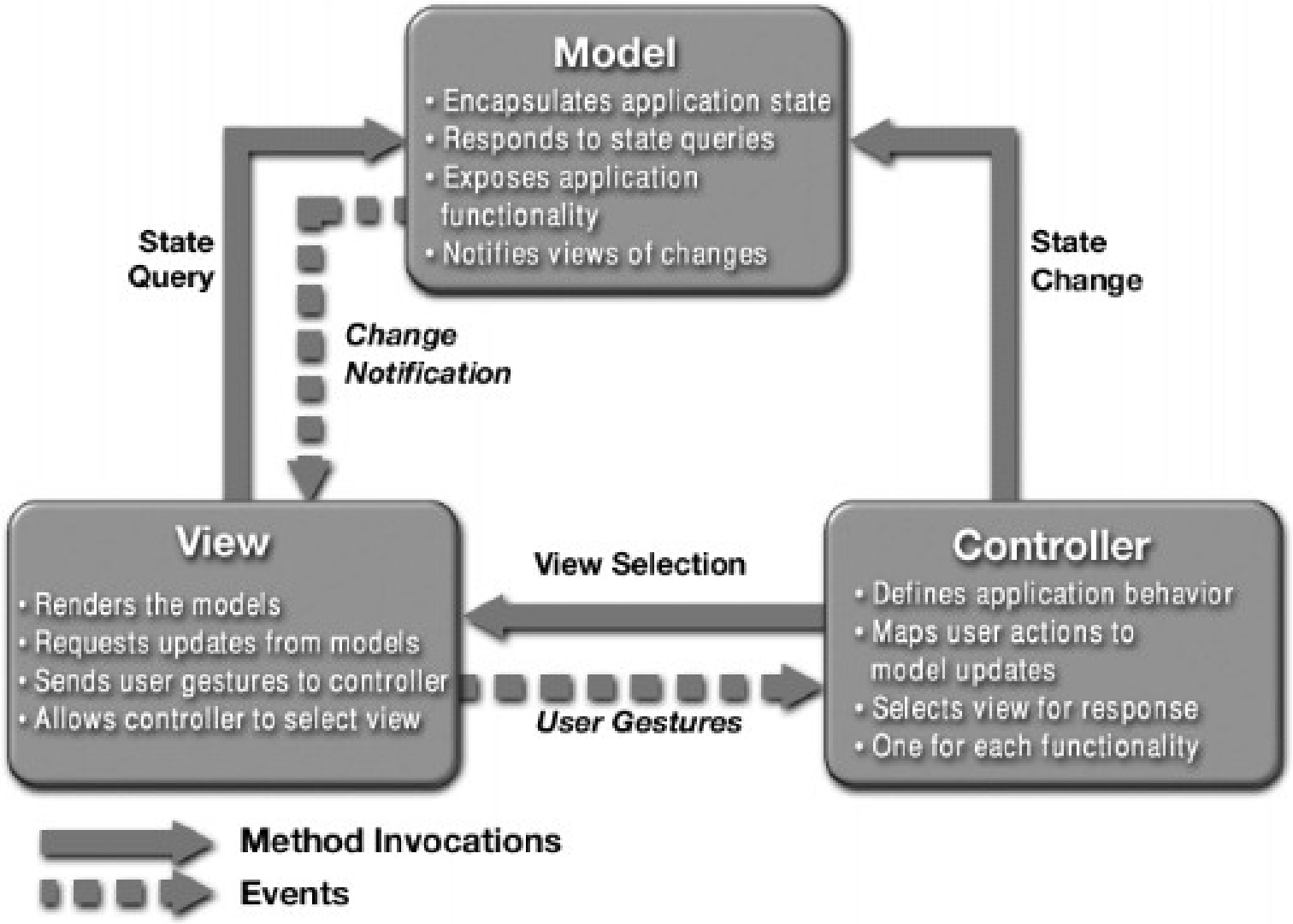
Template Method



Model-View-Controller

- El patrón MVC, descrito en el libro Design Patterns (E. Gamma y otros), propone un diseño basado en los siguientes componentes:
  - Model
    - Maneja información, y notifica a los observadores cuando ésta cambia
    - Representa una abstracción de un proceso o sistema del mundo real
  - View
    - Encargado de realizar la presentación en un dispositivo gráfico
  - Controller
    - Medio por el cual el usuario interactúa con la aplicación
    - Acepta requerimientos del usuario, e instruye al model y al view para que realicen las acciones que corresponda

Model-View-Controller



Patrones de Diseño para J2EE

- "Core J2EE Patterns – Best Practices and Design Strategies", Deepak Alur, John Crupi y Dan Malks, Sun Microsystems
- "EJB Design Patterns – Advanced Patterns, Processes, and Idioms", Floyd Marinescu (director de la comunidad J2EE TheServerSide.com), 2002

Los Patrones de Sun

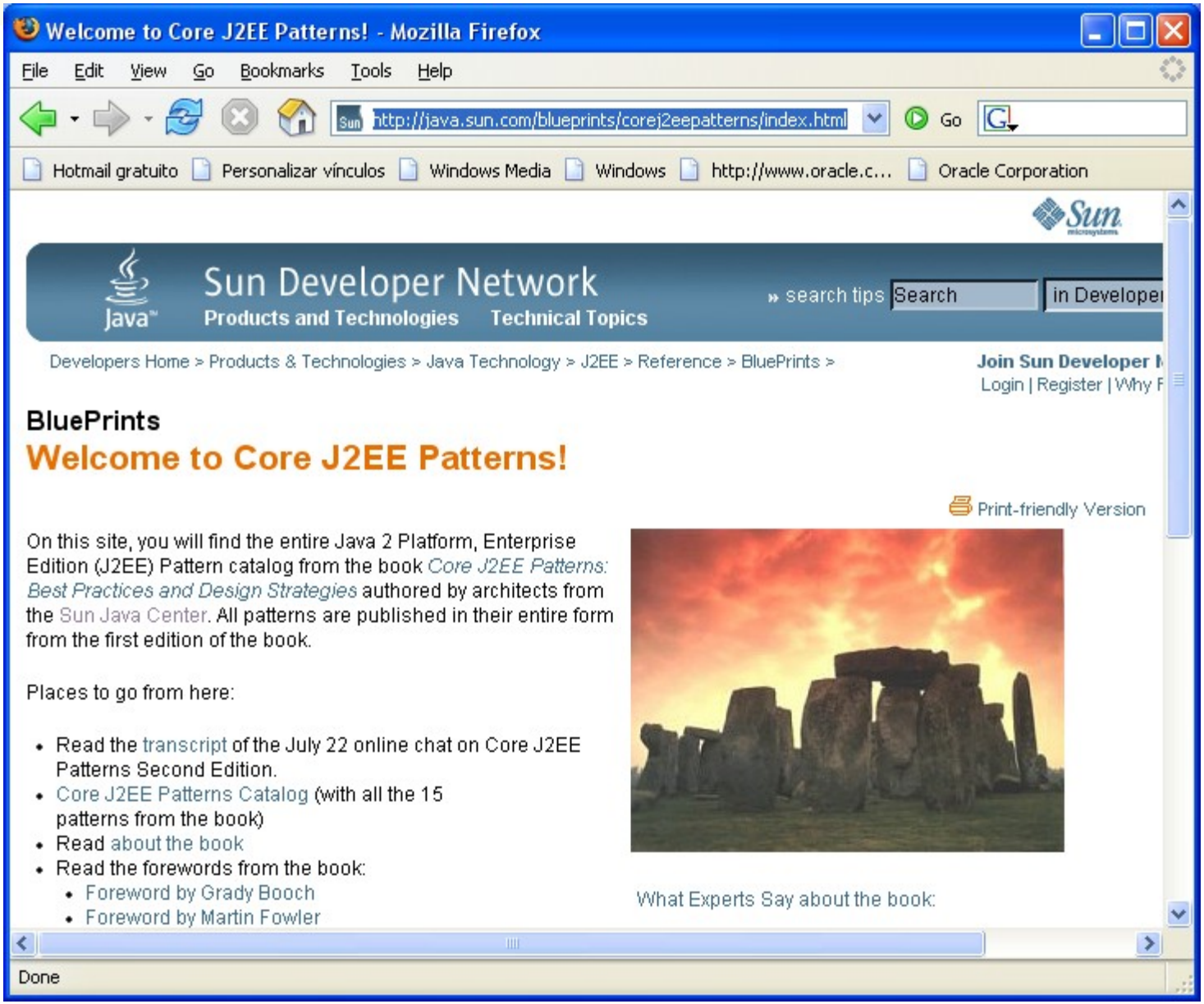
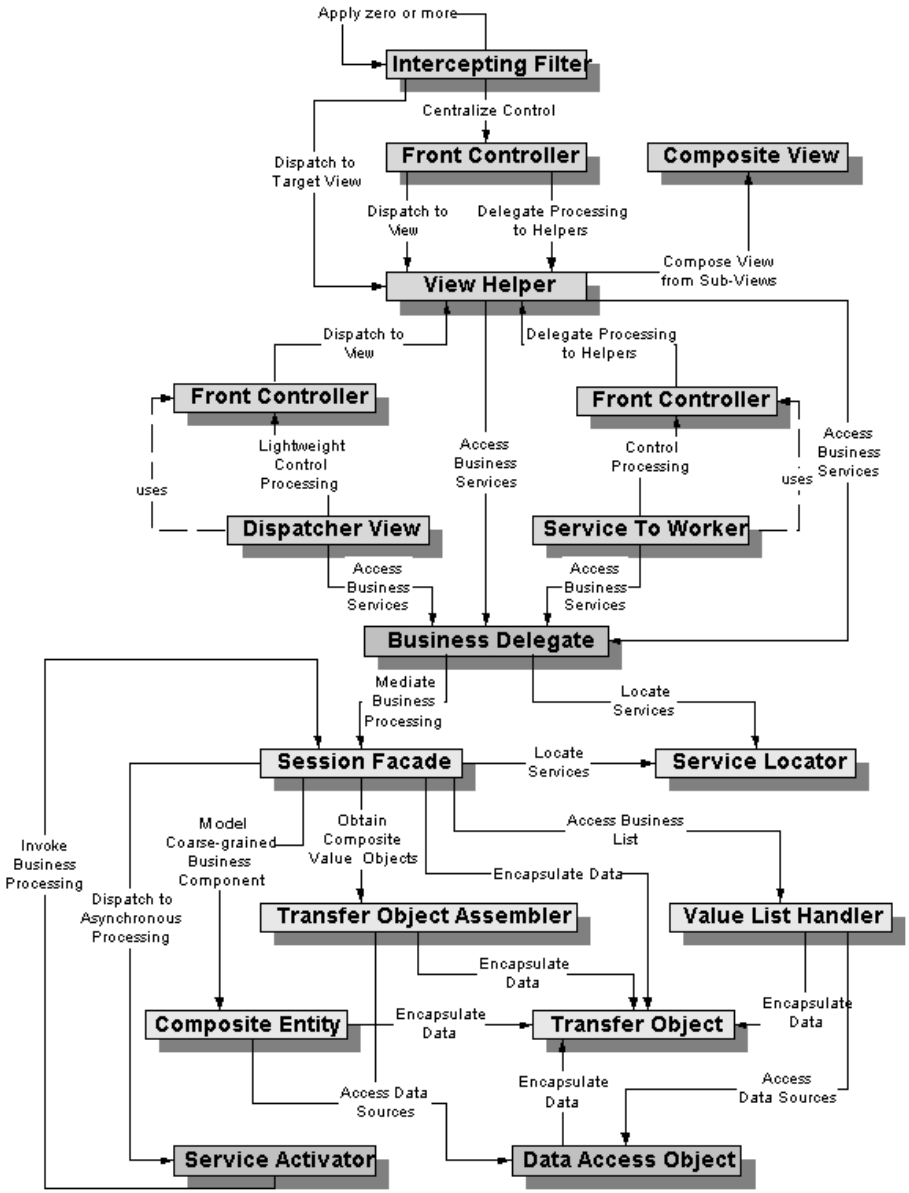


Diagrama de Patrones J2EE (Sun)



Patrones de Martin Fowler

- http://www.martinfowler.com/eaaCatalog

Categoría	Patrones
Domain Logic	Transaction Script, Domain Model, Table Module, Service Layer
Data Source Architectural	Table Data Gateway, Row Data Gateway, Active Record, Data Mapper
Object-Relational Behavioral	Unit of Work, Identity Map, Lazy Load
Object-Relational Structural	Identity Field, Foreign Key Mapping, Association Table Mapping, Dependent Mapping, Embedded Value, Serialized LOB, Single Table Inheritance, Class Table Inheritance, Concrete Table Inheritance, Inheritance Mappers
Object-Relational Metadata Mapping	Metadata Mapping, Query Object, Repository
Web Presentation	Model View Controller, Page Controller, Front Controller, Template View, Transform View, Two-Step View, Application Controller
Distribution	Remote Facade, Data Transfer Object
Offline Concurrency	Optimistic Offline Lock, Pessimistic Offline Lock, Coarse Grained Lock, Implicit Lock
Session State	Client Session State, Server Session State, Database Session State
Base	Gateway, Mapper, Layer Supertype, Separated Interface, Registry, Value Object, Money, Special Case, Plugin, Service Stub, Record Set