Università degli Studi di Napoli "Federico II"

Ingegneria del Software

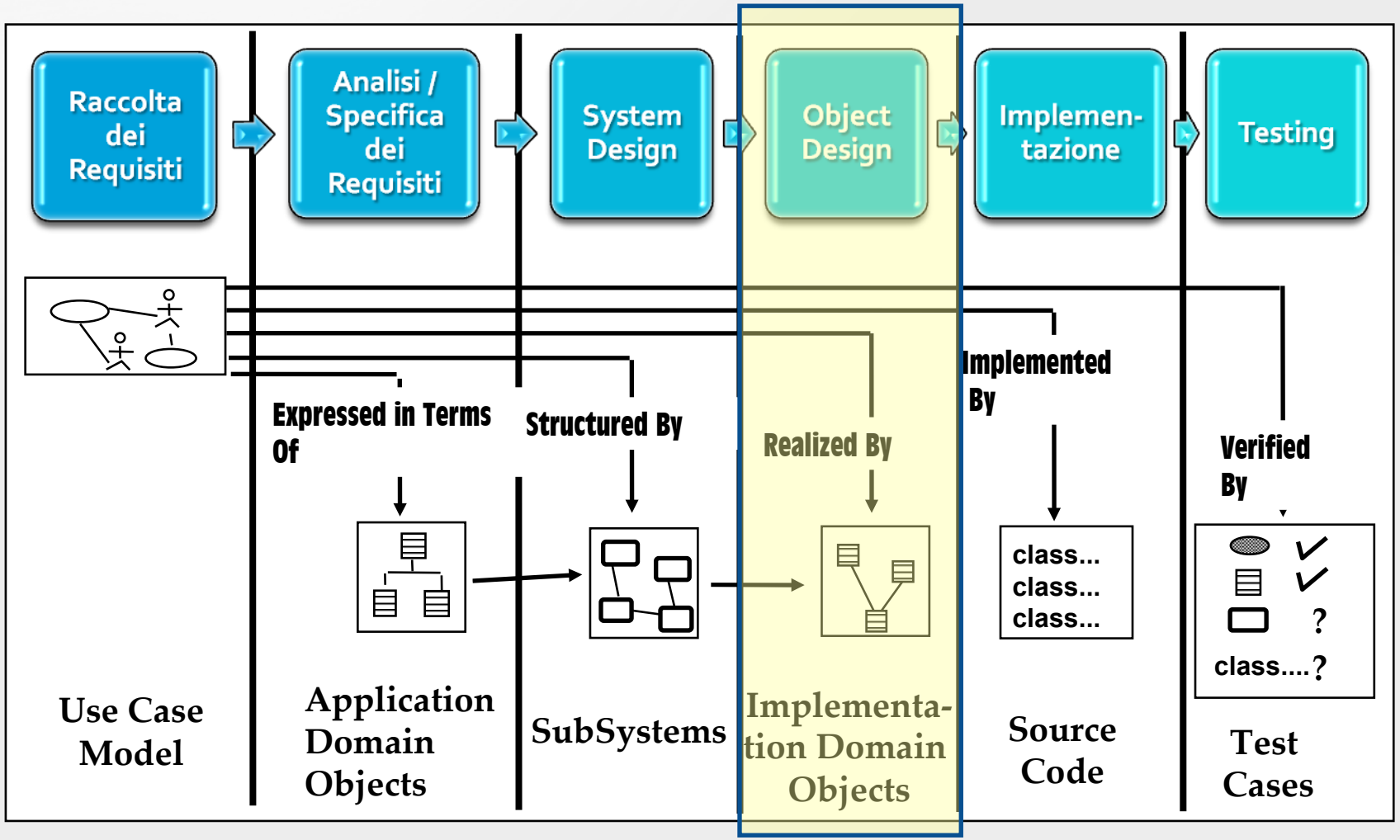
a.a. 2012/13

Object Design e Pattern Design

Obiettivi della lezione

- Comprendere le attività dell'Object Design
  - Attività dell'object design
  - Principi dell'object design
  - Design patterns

Software Lifecycle Activities



Object Design

L'Object Design è la fase nel ciclo di vita del software in cui si definiscono le scelte finali prima dell'implementazione

- **Riuso**:
  - Componenti Off-the-shelf identificate durante la fase di system design
  - Librerie di classi per strutture di dati di base e servizi
  - Design patterns per la soluzione di problemi comuni

- **Spesso può essere richiesta una attività di adattamento delle componenti oggetto di riuso.**

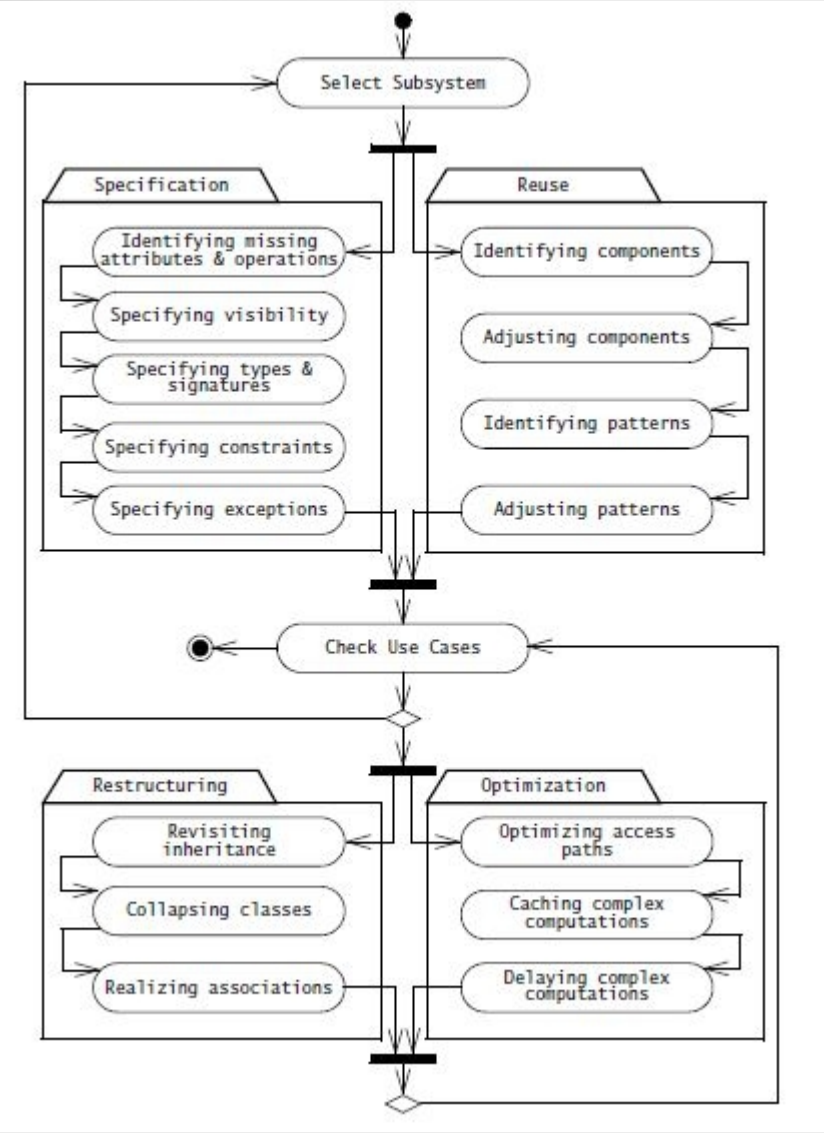
- **Specifica delle interfacce:**
  - I servizi dei sottosistemi identificati nella fase di system design sono identificati in termini di interfacce: **Operazioni, Argomenti, Segnature, Eccezioni (API dei sottosistemi)**
  - Oggetti ed operazioni aggiuntive per il trasferimento di dati tra i sottosistemi.

Object Design (2)

- **Ristrutturazione**:
  - Attività di ristrutturazione che perseguono le seguenti finalità di progettazione: riuso, manutenibilità, leggibilità, comprensibilità del modello del sistema
  - I **principi dell'object design** costituiscono una guida per la fase di ristrutturazione

- **Ottimizzazione:**
  - La fase di ottimizzazione è guidata dai requisiti di performance del modello del sistema
  - Scelta di algoritmi che rispondano a requisti di memoria/performance.
  - Ottimizzazioni nella velocità di recupero dell'informazione persistente.
  - Apertura della stratificazione dell'architettura per fini di prestazione.

Attività dell'Object Design



Principi di buon o-o-design

http://www.oodesign.com/design-principles.html

Lineee guida per evitare cattiva progettazione.

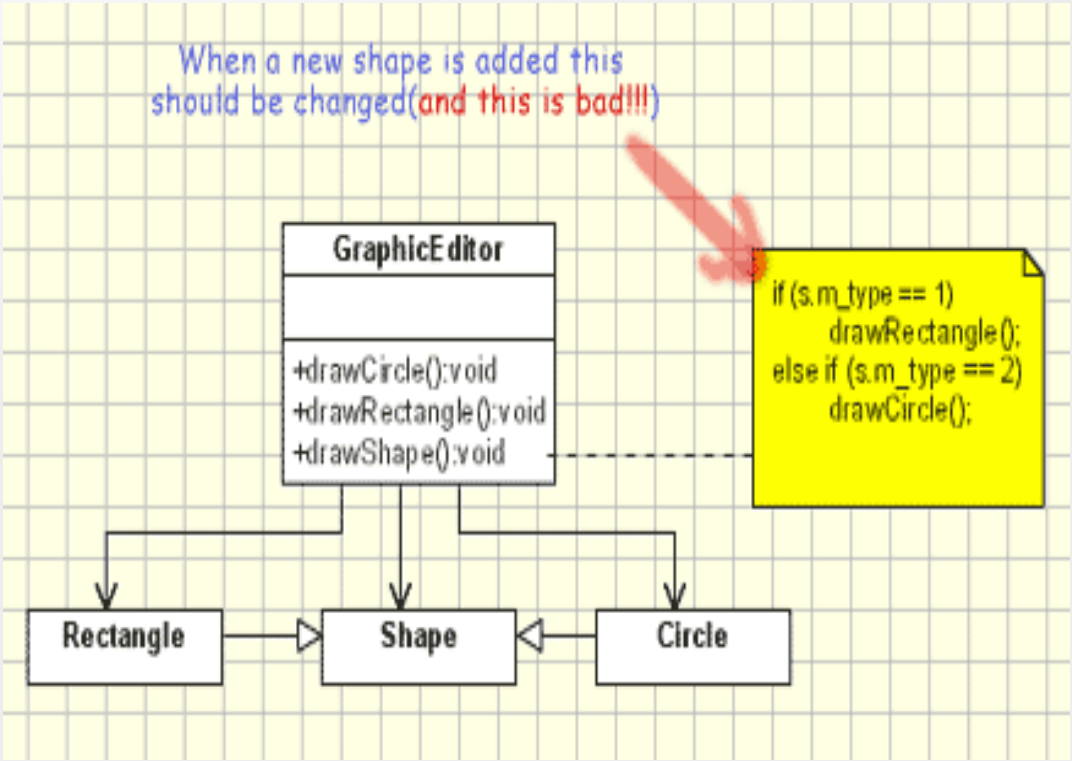
- Principio di Apertura-Chiusura (Open-Close)
- Principio di inversione delle dipendenze (Dependency Inversion)
- Principio di segregazione delle interfacce (Interface Segregation)
- Principio di Singola Responsabilità
- Principio di Sostituzione di Liskov
- Anche …. Legge di Demetra

Principio Open Close

Entità software come classe, moduli e funzioni dovrebbero essere **aperte** per le estensioni ma **chiuse** per le modifiche.

- Tener conto delle frequenti necessità di cambiamento in fase di sviluppo e manutenzione.
- Cambiamenti intervengono quando si aggiunge una nuova funzionalità.
- Limitare la necessità di cambiamenti alle unità già sviluppate
- Aggiungere le nuove funzionalità come nuove classi lasciando il codice esistente invariato

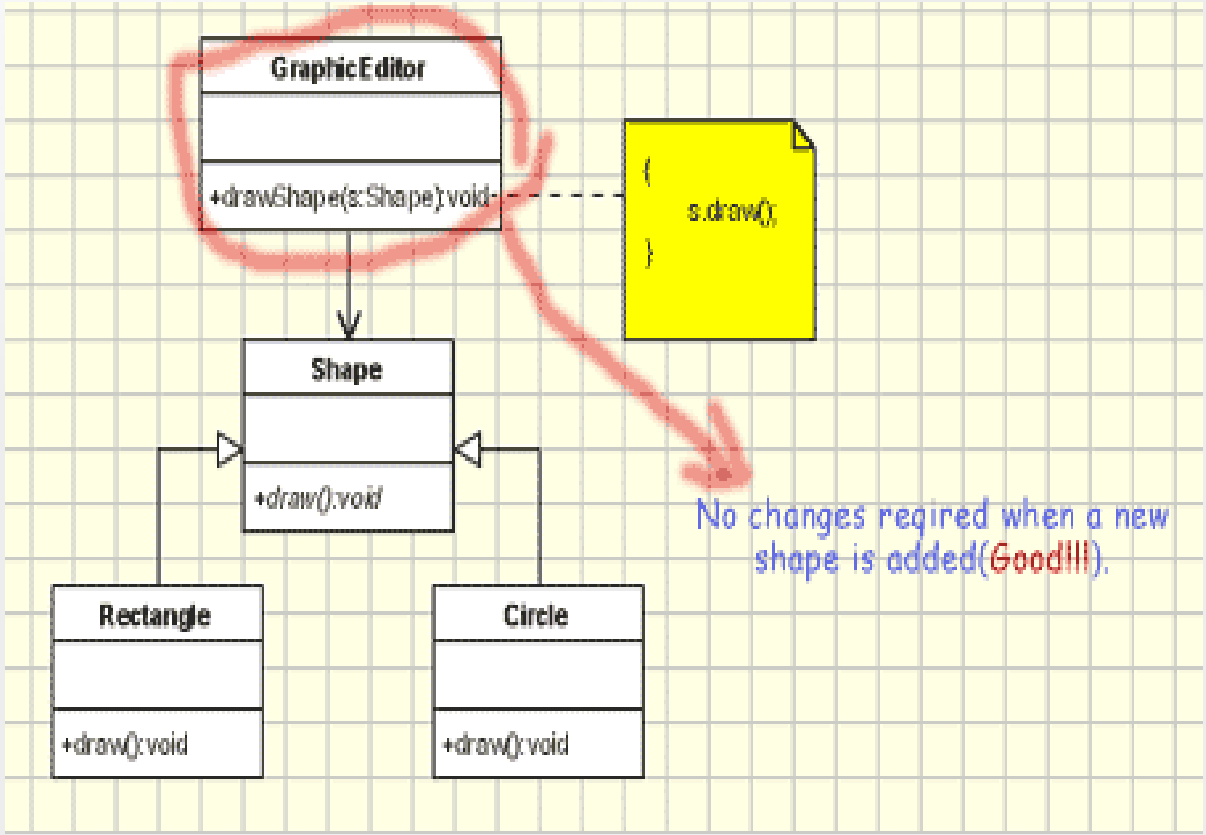
Principio Open Close- Esempio (1)



Principio Open Close- Esempio (1)

```
class GraphicEditor {
    public void drawShape(Shape s) {
        if (s.m_type==1)
            drawRectangle(s);
        else if (s.m_type==2)
            drawCircle(s);}
    public void drawCircle(Circle r) {....}
    public void drawRectangle(Rectangle r) {....}       }

class Shape {
    int m_type;    }

class Rectangle extends Shape {
    Rectangle() {
        super.m_type=1;    }       }

class Circle extends Shape {
    Circle() {
        super.m_type=2;    }      }
```

Principio Open Close- Esempio (2)



Principio Open Close- Esempio (2)

```
class GraphicEditor {
    public void drawShape(Shape s) {
        s.draw();
    }
}

class Shape {
    abstract void draw();
}

class Rectangle extends Shape  {
    public void draw() {
        // draw the rectangle
    }
}
```

Principio Inversione dipendenza

Moduli di alto livello non dovrebbero dipendere da moduli di bassi livello.

Le astrazioni non dovrebbero dipendere da dettagli. I dettafgli dovrebbero dipendere dalle astrazioni

- Classi alto livello: incapsulano logiche complesse
- Classi basso livello: implementano operazione di base
- Classi di alto livello non dovrebbero usare direttamente classi di basso livello
- Isolare le classi di alto livello dai cambiamenti delle classi di basso livello.

- **Classi alto livello → Livello astrazione → Classi di basso livello**

Inversione dipendenza: Esempio (1)

```
class Worker {
public void work() {          // ....working     }
}

class Manager {
Worker m_worker;

public void setWorker(Worker w) {m_worker=w;   }

public void manage() { m_worker.work();   }
}

class SuperWorker      {
public void work() {          //.... working much more      }
}
```

Inversione dipendenza: Esempio (2)

```
interface IWorker {    public void work();        }

class Worker implements IWorker{
public void work() {    // ....working        }
}

class SuperWorker  implements IWorker{
public void work() {    //.... working much more    }
}

class Manager {
IWorker m_worker;

public void setWorker(IWorker w) {       m_worker=w;      }

public void manage() {              m_worker.work();             }
}
```

Principio Interface Segregation

L'utilizzatore (client) non dovrebbe essere forzato a dipendere da interfacce che non utilizza.

- Una interfaccia potrebbe astrarre un sistema costituito da più moduli (fat interface)
- Un client non dovrebbe essere forzato a implementare parti di un interfaccia che non intende utilizzare (interessato solo a prte dei moduli)
- Invece di una fat interface potrebbe essere preferibile una collezione di interfacce a servizio dei sottomoduli.

Interface Segregation – Esempio (1)

```
interface IWorker {
public void work();
public void eat();
}

class Worker implements IWorker{
public void work() {  // ....working }
public void eat() {     // ..... eating in launch break    }
}
class SuperWorker implements IWorker{
public void work() {  //.... working much more          }
public void eat() {     //.... eating in launch break       }
}

class Manager {
IWorker worker;
public void setWorker(IWorker w) {    worker=w;    }
public void manage() {  worker.work();        }
}
```

Interface Segregation – Esempio (2)

interface IWorker extends IFeedable, **IWorkable** { }

```
interface IWorkable {
public void work();
}

interface IFeedable{
public void eat();
}

class Worker implements IWorkable, IFeedable{
public void work() {    // ....working        }
public void eat() {       //.... eating in launch break }
}

class Robot implements IWorkable{
public void work() {    // ....working        }
}
```

Interface Segregation – Esempio (2)

```
class SuperWorker implements IWorkable, IFeedable{
public void work() {  //.... working much more        }

public void eat() {     //.... eating in launch break      }
}

class Manager {
Workable worker;

public void setWorker(Workable w) {  worker=w;   }

public void manage() {     worker.work();    }
}
```

Principio di Singola Responsabilità

Una classe dovrebbe avere una sola possibile causa di cambiamento.

- Se una classe non ha una unica responsabilità la necessità di cambiamento di una funzionalità può avere effetti sulle rimanenti.
- Il principio suggerisce che se vi sono due possibili cause di cambiamento (rispetto alle funzionalità) la classe dovrebbe essere divisa in due classi.

Singola Responsabilità: Esempio (1)

```
interface IEmail {
    public void setSender(String sender);
    public void setReceiver(String receiver);
    public void setContent(String content);
}

class Email implements IEmail {
    public void setSender(String sender) {// set sender; }
    public void setReceiver(String receiver) {// set receiver; }
    public void setContent(String content) {// set content; }
}
```

Due possibili motivi di cambiamento

Aggiungere un nuovo protocollo

Aggiungere nuovi tipi e formati di contenuto

Singola Responsabilità: Esempio (2)

```
interface IEmail {
    public void setSender(String sender);
    public void setReceiver(String receiver);
    public void setContent(IContent content);
}

interface IContent {
    public String getAsString(); // used for serialization
}

class Email implements IEmail {
    public void setSender(String sender) {// set sender; }
    public void setReceiver(String receiver) {// set receiver; }
    public void setContent(IContent content) {// set content; }
}
```

Principio di Sostituzione di Liskov

Deve essere rispettata la sostitutività dei tipi derivati rispetto ai tipi base.

- Tipo derivato: ottenuto da un tipo base per specializzazione.
- Essere certi che quando si costruisce un tipo derivato da un tipo di base il tipo di base solo estende le funzionalità del tipo base senza rimpiazzarle.
- Se un modulo di programma sta usando una classe base, allora il riferimento alla classe di base può essere rimpiazzato con una classe derivata senza effetti sul modulo.

Sostituzione di Liskov – Esempio (1)

```
class Rectangle
{
    protected int m_width;
    protected int m_height;

    public void setWidth(int width){  m_width = width;      }
    public void setHeight(int height){      m_height = height;     }
    public int getWidth(){    return m_width;         }
    public int getHeight(){    return m_height;        }
    public int getArea(){ return m_width * m_height;           }
}
class Square extends Rectangle
{
public void setWidth(int width){ m_width = width; m_height = width; }
public void setHeight(int height){ m_width = height; m_height = height; }
}
```

Sostituzione di Liskov – Esempio (1)

```
class LspTest
{
    private static Rectangle getNewRectangle()
    {
        // it can be an object returned by some factory ...
        return new Square();
    }
    public static void main (String args[])
    {
        Rectangle r = LspTest.getNewRectangle();
            r.setWidth(5);
        r.setHeight(10);
        // user knows that r it's a rectangle.
        // It assumes that he's able to set the width and height as for the
            base class
        System.out.println(r.getArea());
        // now he's surprised to see that the area is 100 instead of 50.
    }
}
```

Riuso

Riuso di codice

- Evita la duplicazione di lavoro già svolto
- Il presupposto è che il codice sia di buona qualità: pulito, elegante, stabile, comprensibile, documentato....

Riuso di elementi di progettazione

- Evita la soluzione di problemi per cui esista una buona soluzione standard
- Il presupposto è che si conoscano e comprendano dei modelli di progettazione ricorrenti (**Design Patterns**)
- Strumenti per il riuso o-o: **Ereditarietà, Delega**

Ereditarietà

Nella fase di analisi dei requisiti è usata nei class diagram per classificare concetti/oggetti in tassonomie.

Nella fase di obiect design l'erditarietà è usata con lo scopo di

- Ridurre la ridondanza
- Aumentare le possibilità di estendibilità
- Aumentare le possibilità di riuso

Due forme di ereditarietà:

- **Implementation Inheritance** (Ereditarietà implementativa)

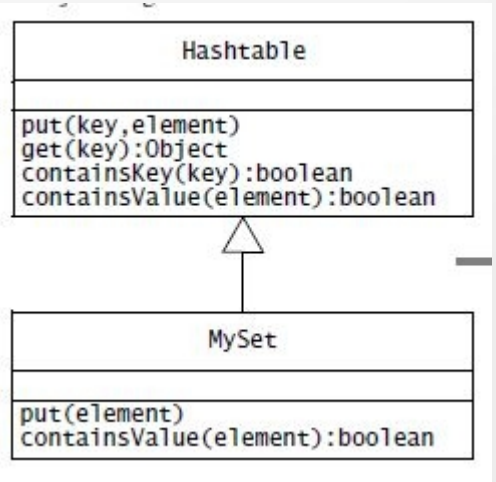
uso dell'ereditarietà per soli fini di riuso del codice

- **Specification Inheritance** (Ereditarietà di specifica)

uso dell'ereditarietà per classificazione (subtyping) ed estendibilità

Implementation Inheritance: Esempio

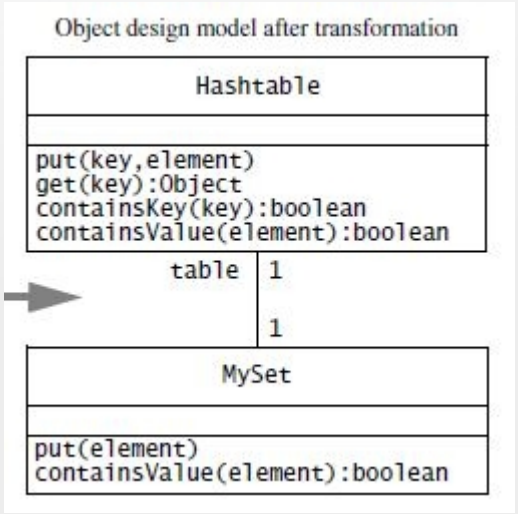
Ereditarietà tra classi concettualmente scorrelate con sola finalità di riuso.




```
/* Implementation of MySet using
inheritance */
class MySet extends Hashtable {
    /* Constructor omitted */
    MySet() {
    }

    void put(Object element) {
        if (!containsKey(element)){
            put(element, this);
        }
    }
    boolean containsValue(Object
            element){
        return containsKey(element);
    }
    /* Other methods omitted */
}
```

Delega vs Implementation Inheritance

La delega è preferibile alla implementatio inheritance se si hanno finalità di riuso.




```
/* Implementation of MySet using
delegation */
class MySet {
    private Hashtable table;
    MySet() {
        table = Hashtable();
    }
    void put(Object element) {
        if (!containsValue(element)){
            table.put(element,this);
        }
    }
    boolean containsValue(Object
            element) {
        return
        (table.containsKey(element));
    }
    /* Other methods omitted */
}
```

Architetture e Design Patterns

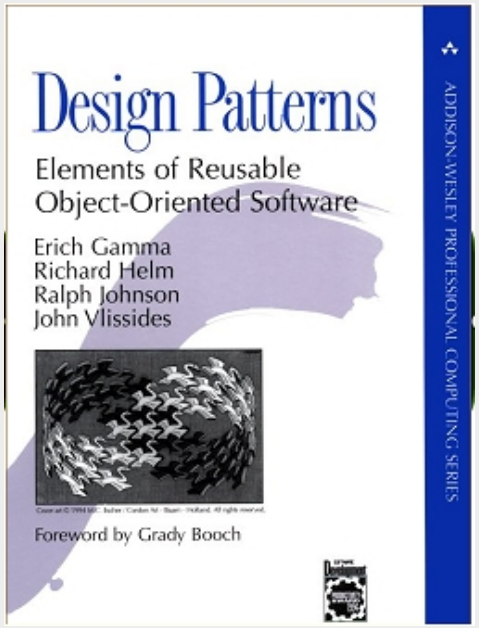
- Le Architetture definiscono la struttura di un sistema ad alto livello (system design)
  - Definizione della struttura di moduli
  - Grazie alle soluzioni architetturali standard, si ha la disponibilità di una vasta scelta di soluzioni largamente sperimentate per una variegata casistica di problemi
- Situazione analoga al livello più basso dello object design
- Ci sono modi "standard" di combinare classi tra loro per svolgere funzionalità e gestire situazioni ricorrenti: Design Patterns

Motivationi

- L'OO usa strutture di progettazione ricorrenti che promuovono
  - Astrazione
  - Flessibilità
  - Modularità
  - Eleganza
  - Riuso
  - Estendibilità

Che cos'è un Design Pattern?

- Un design pattern
  - Una soluzione sperimentata a un problema ricorrente di progettazione
  - Astrazione di una struttura ricorrente in progettazione
  - Riguarda classi e/o oggetti
    - Dipendenze
    - Strutture
    - Interazioni
    - Convenzioni
  - Sommario di esperienza di progettazione



Storia dei Design Patterns

- Christopher Alexander
  - A Pattern Language (1977)
  - A Timeless Way of Building (1979)
- "Gang of four"
    - Erich Gamma
    - Richard Helm
    - Ralph Johnson
    - John Vlissides
  - Il libro: "*Design Patterns: Elements of Reusable Object-Oriented Software*" (1995)

Gli elementi di un Design Pattern?

- Un design pattern ha 4 basic elementi di base:

1. **Name**
2. **Problem**
3. **Solution**
4. **Consequences and trade-offs of application**

E' indipendente dal linguaggio e dall'implementazione
E' una “micro-architettura”
Non è meccanicamente applicabile
La soluzione deve essere rifrasata nei termini del problema concreto.

Catalogo di Design Pattern

- Catalogo GoF (“the Gang of Four”)
  - “Design Patterns: Elements of Reusable Object-Oriented Software,” Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995
- Catalogo POSA
  - Pattern-Oriented Software Architecture, Buschmann, et al.; Wiley, 1996
- …

Classificatione del catalogo GoF

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Singleton

(Creational)

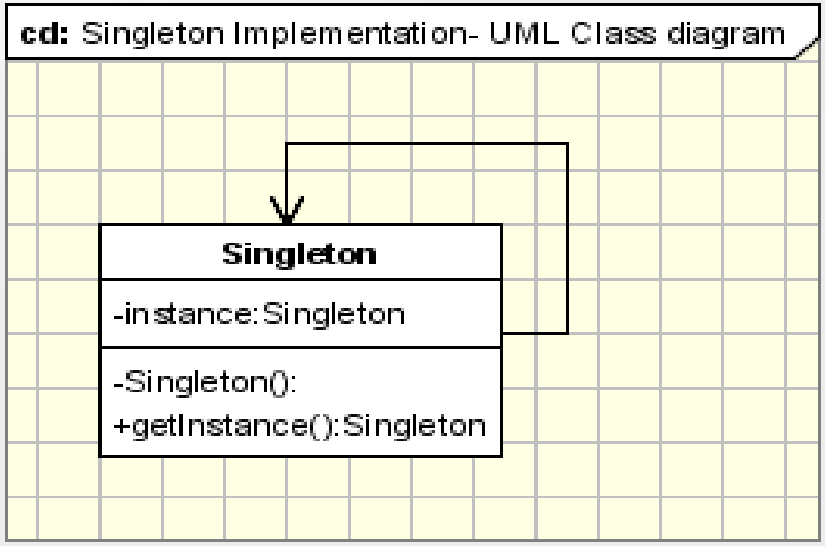
Singleton (Creational)

- **Scopo**
  - Assicura che istanziato un solo esemplare per una classe
  - Fornisce un punto di accesso centralizzato per una sruttura.
- **Applicability**

  Ci deve essere una sola classe per una istanza e deve essere accessibile da un preciso e noto punto di accesso.
  - Quando l'unica istanza dovrebbe poter essere estesa per specializzazione e il client dovrebbe essere capace di usarla senza modificare il codice.

Singleton

- Struttura



Singleton - Esempio

```
class Singleton
{
    private static Singleton instance;
    private Singleton()
    {   ...
    }

    public static synchronized Singleton getInstance()
    {
        if (instance == null)
            instance = new Singleton();

        return instance;
    }
    ...
    public void doSomething()
    {   ...
    }
}
```

Singleton esempio

- Permette di frnire un punto di accesso globale usando

```
Singleton.getInstance().doSomething();
```

Esempi di applicazioni:

- Global logging access point per le classi di una operazione
- Classi che provvedono i set di configurazioni per un'applicazione
- Factory per la generazione di oggetti
- Gestione dell'accesso di risorse condivise.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

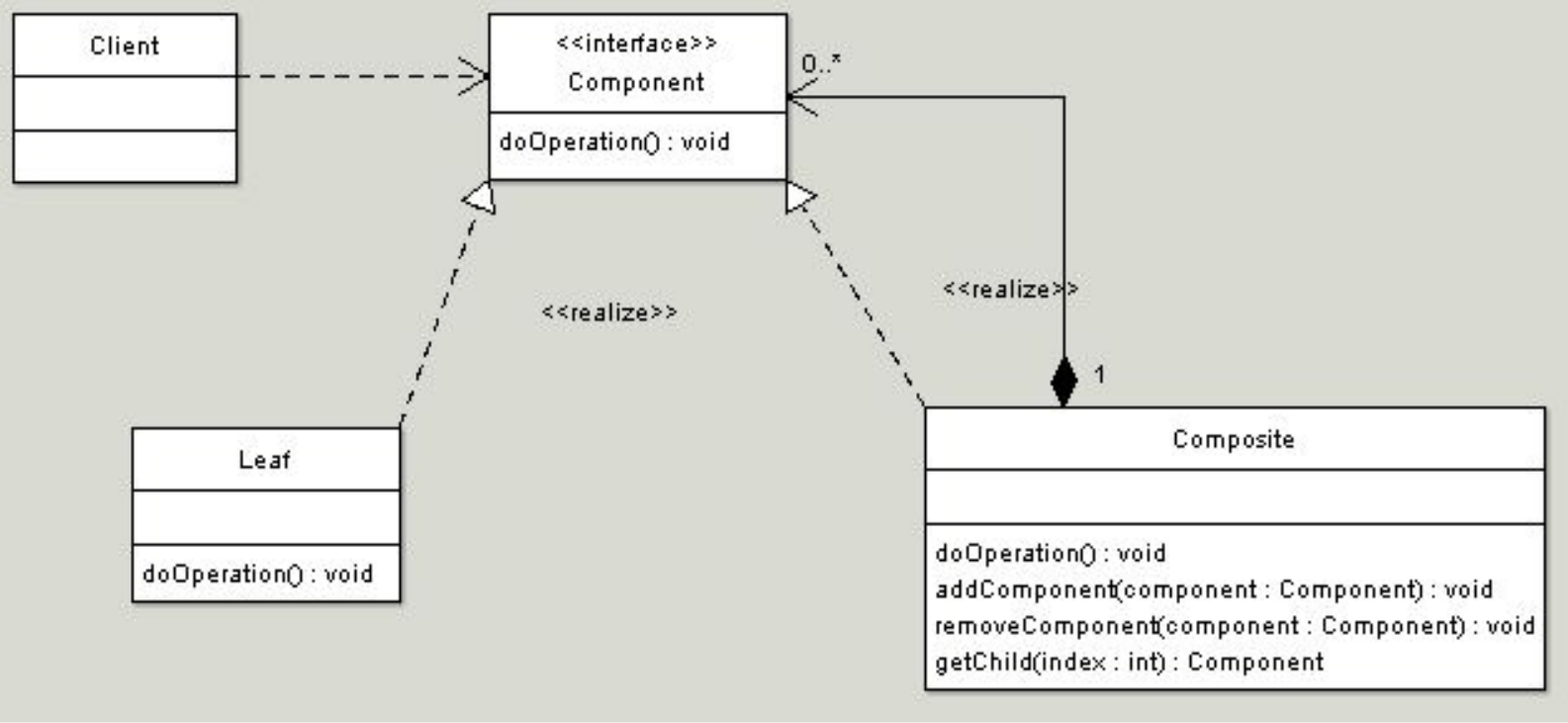
Composite

(Structural)

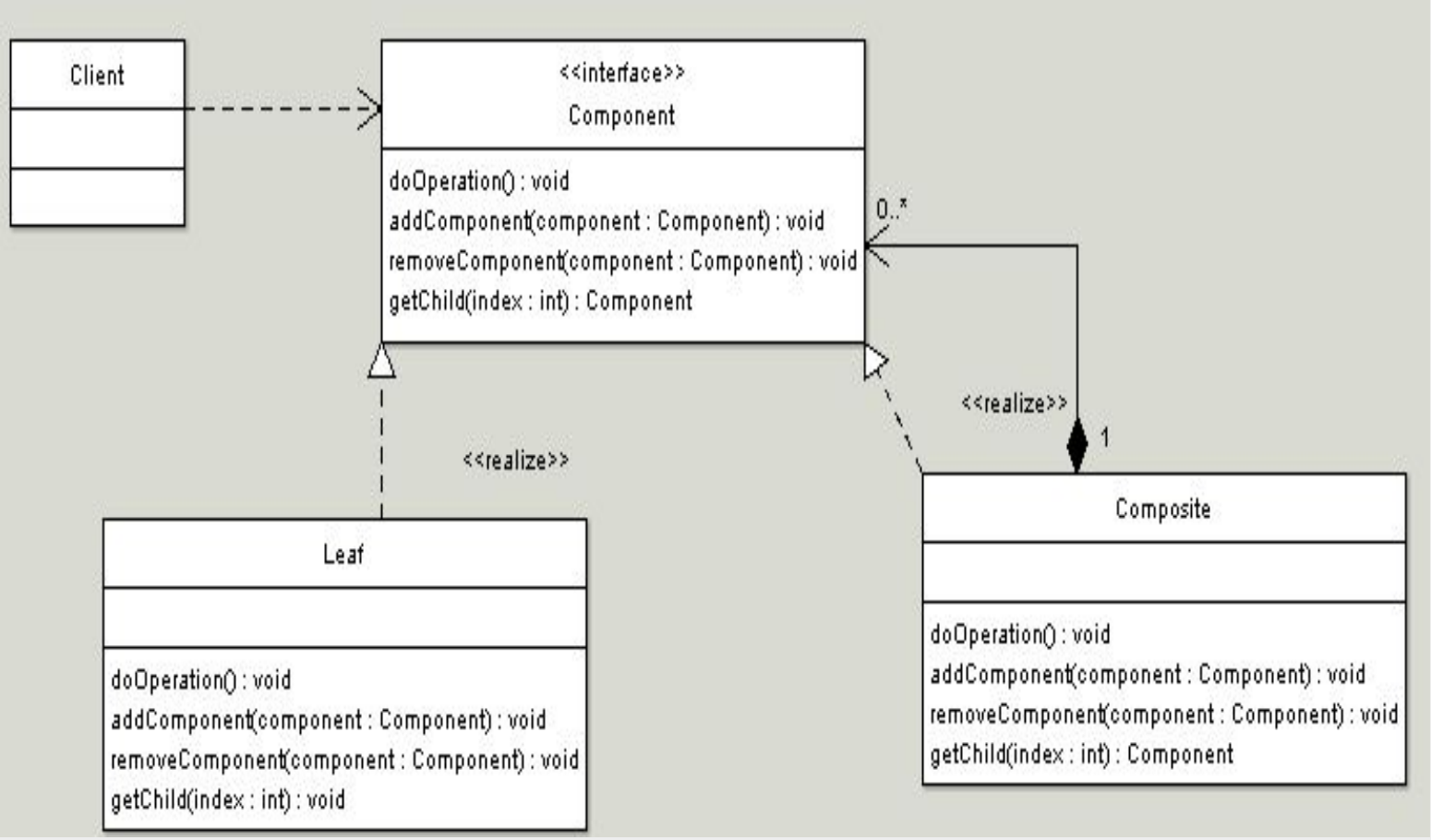
Composite (Structural)

- **Intent**
  - Comporre oggetti in strutture ad albero per rappresentare gerarchie parti-tutto
  - Permettere al client di di tratatre uniformemente oggetti individuali e composizione di oggetti
- **Applicability**
  - Composizione ricorsiva parti-tutto di oggetti,
  - Gestione uniforme di oggetti individuali e composizioni

Composite: Implementazione

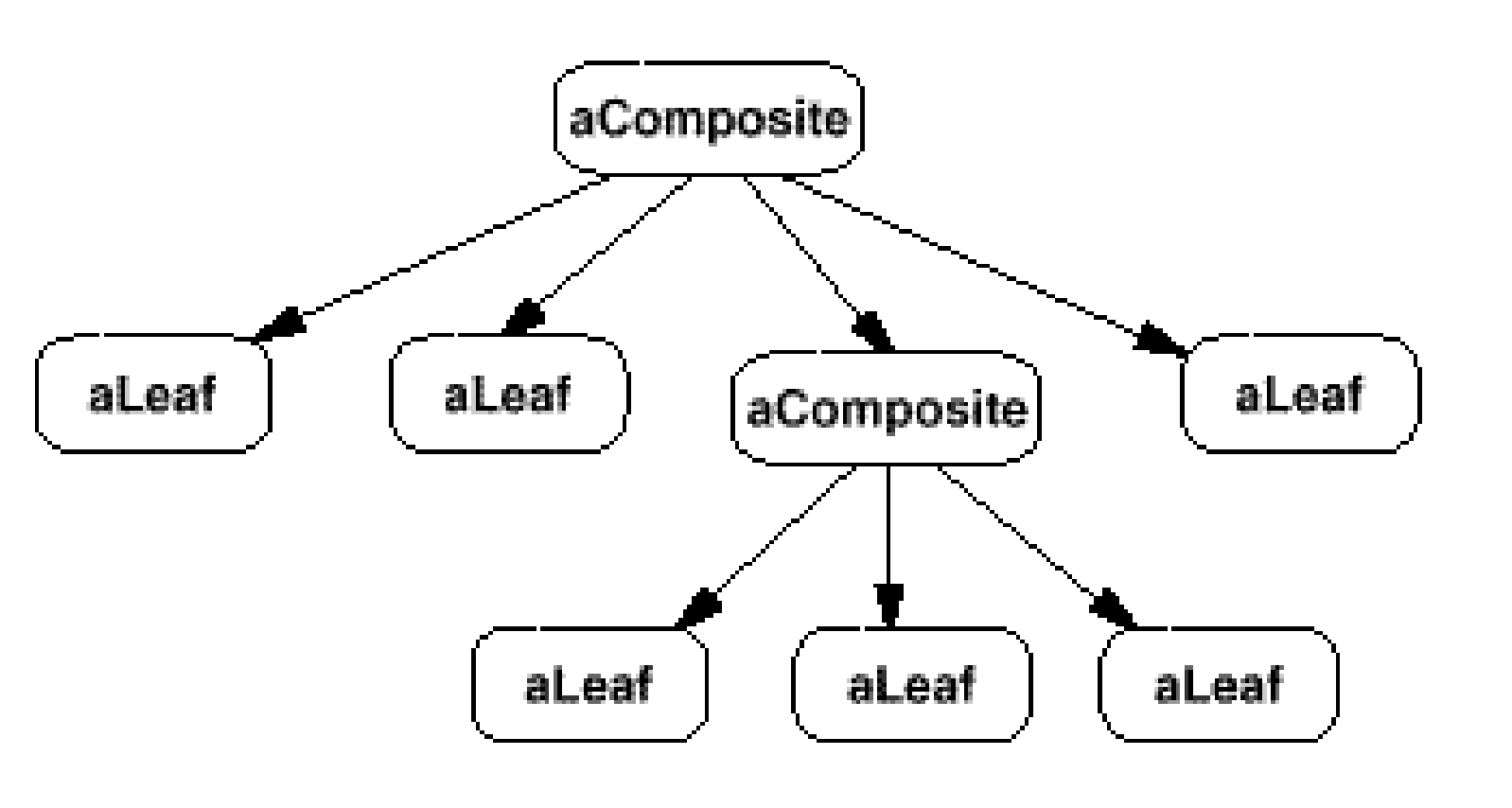


Composite: Implementazione alternativa



Composite (Cont'd)

- Object Stucture:



Composite Implementation (1)

```
/** "Component" */
interface Graphic { //Prints the graphic.
public void print(); }

/** "Composite" */
class CompositeGraphic implements Graphic {
//Collection of child graphics.
private List<Graphic> mChildGraphics = new ArrayList<Graphic>();

 //Prints the graphic.
public void print() {
for (Graphic graphic : mChildGraphics)
   { graphic.print(); }
}
```

Composite Implementation (2)

```
 //Adds the graphic to the composition.
public void add(Graphic graphic)
{ mChildGraphics.add(graphic); }

//Removes the graphic from the composition.
public void remove(Graphic graphic)
{ mChildGraphics.remove(graphic); } }

/** "Leaf" */
class Ellipse implements Graphic {
    //Prints the graphic.
    public void print() {
       System.out.println("Ellipse");
    }
}
```

Composite

- **Consequences**
  - Uniformità: trattamento uniforme indipendente dalla complessità
  - Estendibilità: aggiunta trasparente al funzionamento di nuove componenti
  - Overhead: possono essere richiesti un grande numero di oggetti
- **Implementation**
  - Le Component conoscono i genitori?
  - Interfaccia uniforme per foglie e composti (due soluzioni proposte)?
  - Non allocare spazio per i figli nelle Component class di base
  - Responsibilitàper la cancellazione dei figli.

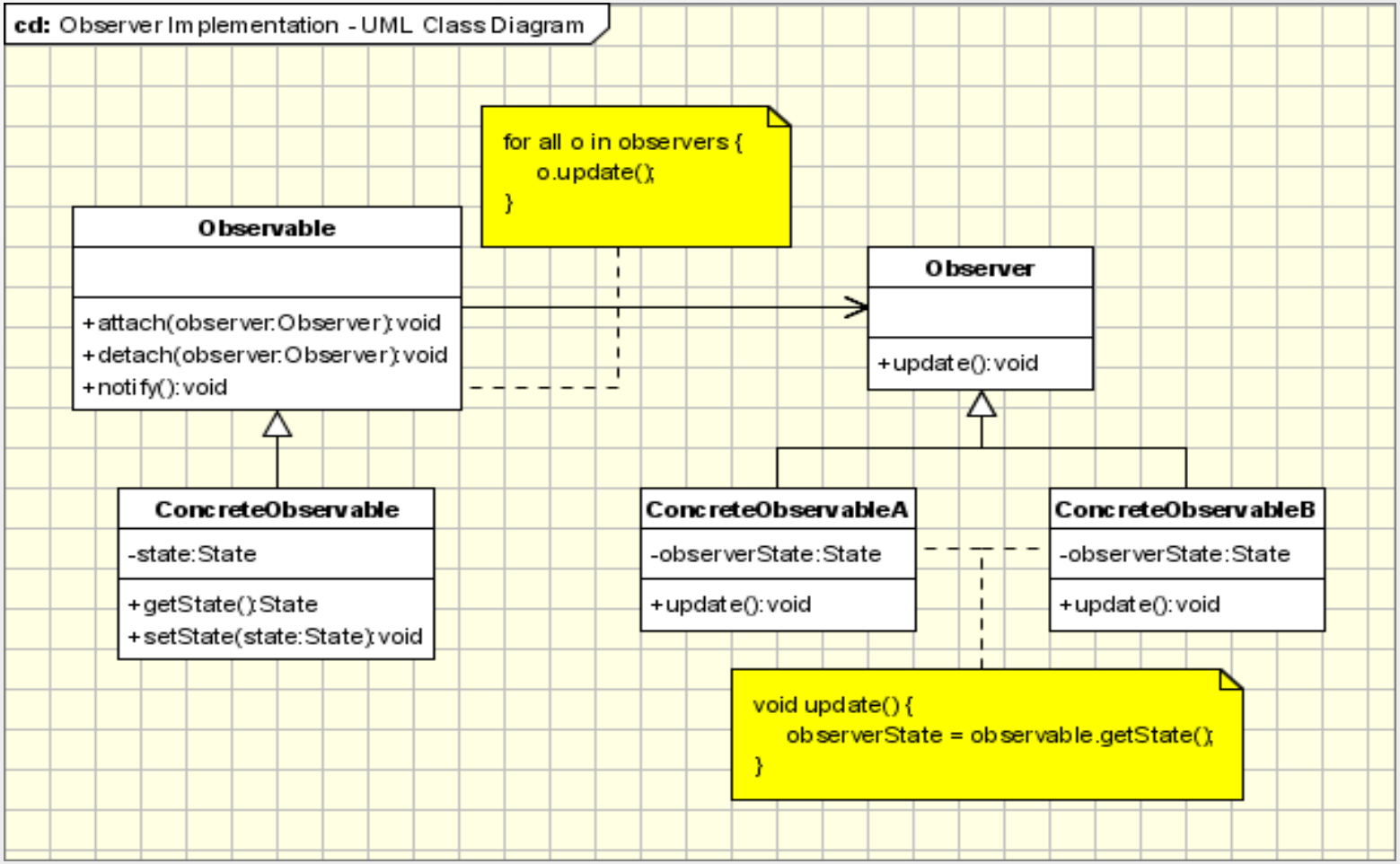
Observer

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Observer

- **Intent**
  - Definire una dipendenza uno-a-molti tra gli oggetti
  - Quando un oggetto cambia stato i sui dipendenti vengono notificati ed aggiornati automaticamente
- **Applicability**
  - Il cambiamento di stato di un oggetto si deve riflettere nel cambiamento di stati in molti oggetti mantenedo basso accoppiamento
  - Previsione di aumento degli osservatori con richiesta di cambiamento minimo
  - Raffrontabile al modello architetturale MVC
  - Event management

Observer (Behavioural)



Observer

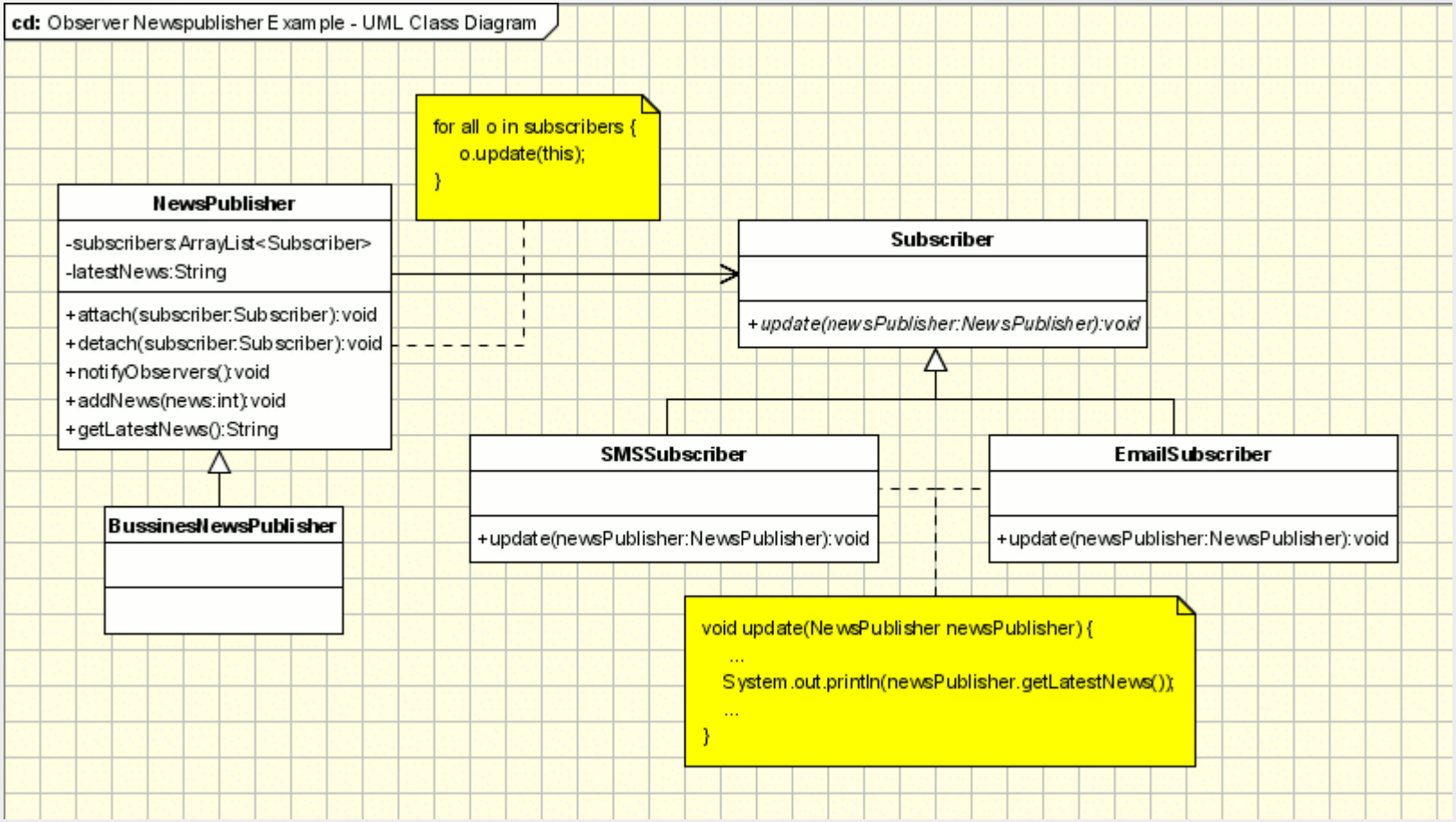
Observable - interface o abstract class con operazione di attaching e de-attaching di osservatori al client.

ConcreteObservable - concrete Observable class. Tiene lo stato degli oggetti It maintain the state of the object e a fronte di cambiamenti notifica gli osservatori noti..

Observer - interface o abstract class che definisce le operazioni per le notifiche.

ConcreteObserverA, ConcreteObserver2 – Observer concreti

Esempio: NewsAgency



Observer - Observable Interface

```
// ISubject --> interface for the subject

public interface ISubject {


// Registers an observer to the subject's notification list

void RegisterObserver(IObserver observer);


// Removes a registered observer from the subject's notification list

void UnregisterObserver(IObserver observer);


// Notifies the observers in the notification list of any change that
occurred in the subject

void NotifyObservers();

}
```

Observer - Observer Interface

```
// IObserver --> interface for the observer
public interface IObserver {

// called by the subject to update the observer of any change
// The method parameters can be modified to fit certain criteria
 void Update();
}
```

Observer - ObservableImpl (1)

```
// Subject --> class that implements the ISubject interface....

using System.Collections;

public class Subject : ISubject {

// use array list implementation for collection of observers private ArrayList observers;

// decoy item to use as counter

private int counter;

// constructor public Subject() {

observers = new ArrayList();

counter = 0; }

public void RegisterObserver(IObserver observer)

{ // if list does not contain observer, add

if(!observers.Contains(observer))

        { observers.Add(observer); }

}
```

Observer - Observable Impl (2)

```
public void UnregisterObserver(IObserver observer) {
// if observer is in the list, remove
if(observers.Contains(observer))
        { observers.Remove(observer); } }
public void NotifyObservers() {
foreach(IObserver observer in observers)
        { observer.Update(); } }
// the subject will notify only when the counter value is divisible by 5
public void Operate() {
for(counter = 0; counter < 25; counter++)
{ if(counter % 5 == 0)

        { NotifyObservers(); } } } }
```

Observer - Observer Implementation

```
// Observer --> Implements the IObserver
public class Observer : IObserver {
// this will count the times the subject changed
// evidenced by the number of times it notifies this observer
private int counter;
public Observer() { counter = 0; }
// counter is incremented with every notification
public void Update() {
    counter += 1; }
// a getter for counter
public int Counter {
 get { return counter; }
} }
```

Observer

- Consequences
  - Estendibilità: aggiunta trasparente al funzionamento di nuovi osservabili e di nuovi osservatori
  - Complessa gestione della comunicazione in caso di dipendenze molti-a -molti tra gli oggetti
- Implementation
  - Chi e come sollecita gli update: problemi con cambi frequenti di stato dell'osservato.
  - Lo stato dell'osservabile deve essere mantenuto consistente quando l'operazione di notifica viene sollecitata (fino alla sua conclusione)

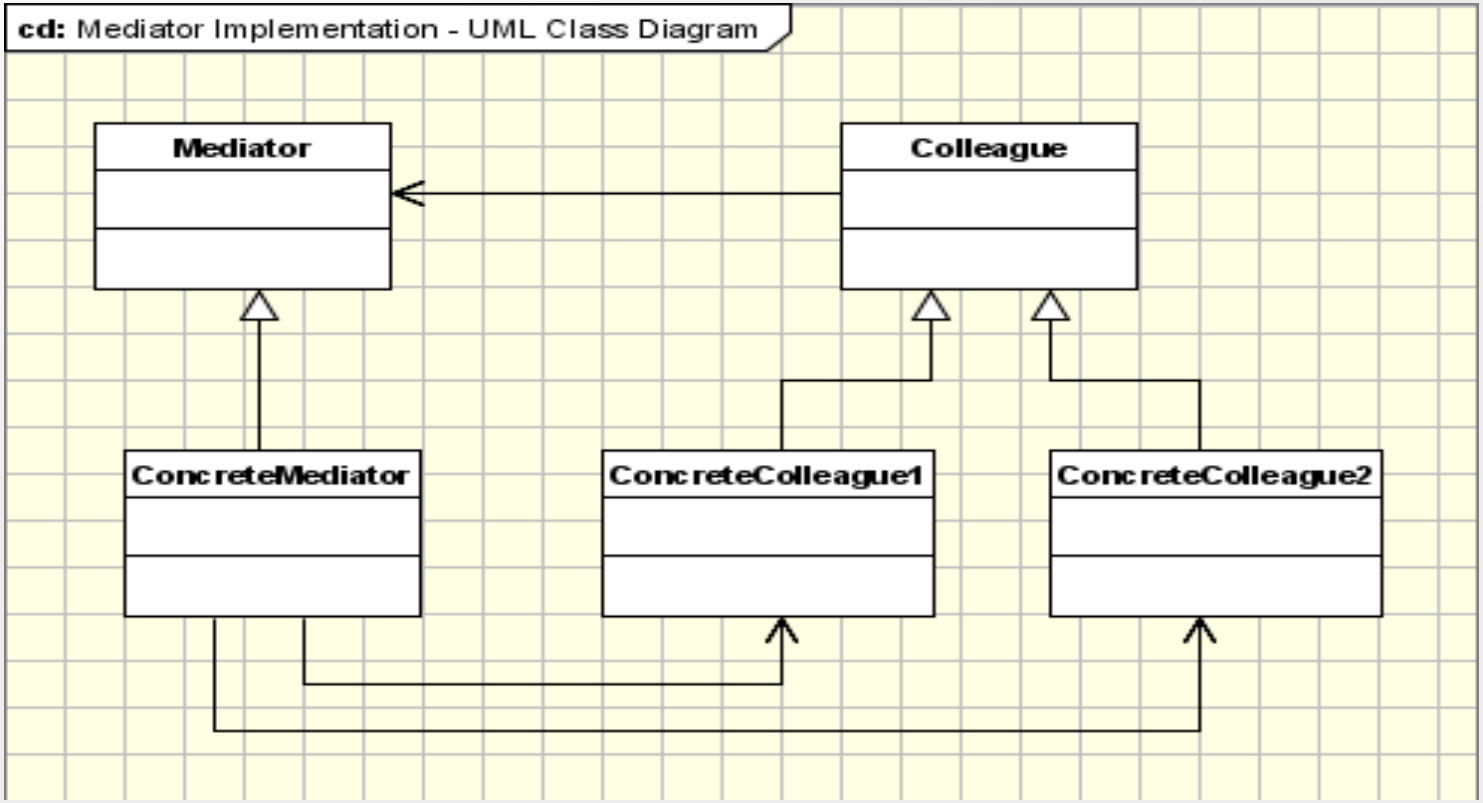
Mediator

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Mediator (Behavioural)

- **Intent**
  - Definire un oggetto che incapsula l'interazione complessa di numerosi oggetti
  - Accoppiamento lasco tra oggetti che interagiscono (non fanno riferimento diretto uno all'altro)
- **Applicability**
  - Quando un insieme di oggetti comunica in mod ben definito ma complesso
  - Il riuso di un oggettto è reso difficile perché fa riferimento diretto e comunica con molti altri.

Mediator: Implementazione



Mediator: Implementazione

- **Mediator**: definisce una interfaccia per comunicare con oggetti Collegue

- **ConcreteMediator**: conosce le classi collegue e mantiene i riferimenti agli oggetti collegue

- **Collegue classes:** tiene riferimento agli oggetti mediator;

comunicano con il mediatore invece di comunicare con i colleghi

Esempio: Chat application

- **Mediator (Chat)**: definisce una interfaccia per comunicare con i partecipanti
- **ConcreteMediator (ChatImpl)**: implementa le operazioni, cioè gestire le interazioni con gli oggetti

Una implementazione per ogni tipologia di chatroom.

Un oggetto di tipo ChatImpl per ogni chatroom

Un oggetto di tipo ChatImpl diffonde un messaggio a tutti i partecipanti alla chatroom

- **Collegue (Participant):** interfaccia per i partecipanti;

comunicano con il mediatore invece di comunicare con i colleghi

- **ConcreteCollegue (HumanParticipant, Bot)** : implementa i partecipanti.. Ciascun partecipante tiene solo una referenza al mediatore (non ad altri partecipanti). Ha la referenza agli oggetti corrispondenti alle chatroom a cui partecipa.

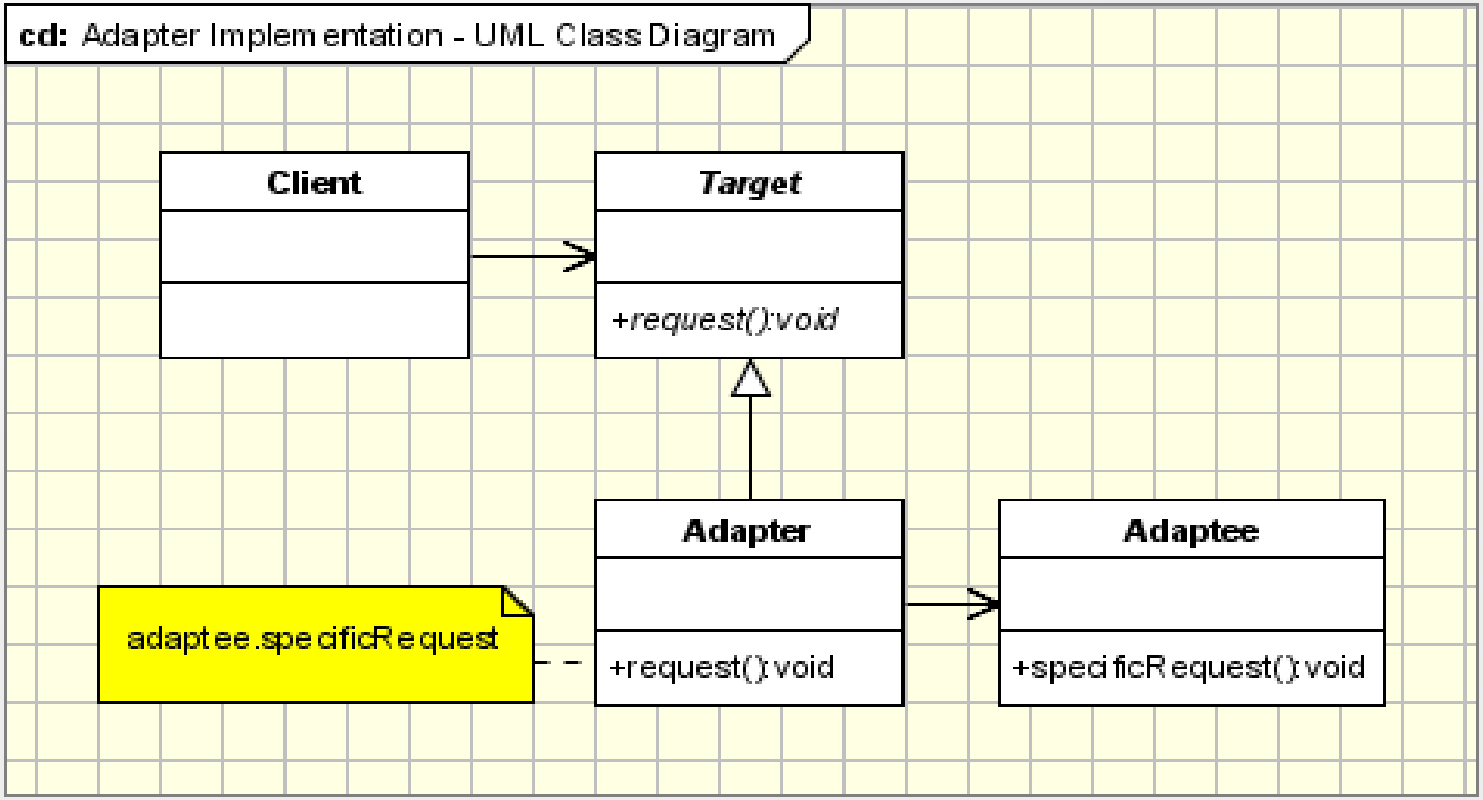
Adapter

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Adapter (Structural)

- **Intent**
  - Convertire l'interfaccia di una classe in un altra interfaccia conforme alle attese del client
  - Permettere di interoperare a classi le cui interfacce sono incompatibili.
- **Applicability**
  - Quando si implememenTa una classe che si basa su una interfaccia generale e le classi implementate non implementano l'interfaccia
  - Realizzazione di Wrappers per importare librerie e frameworks di parti terze.

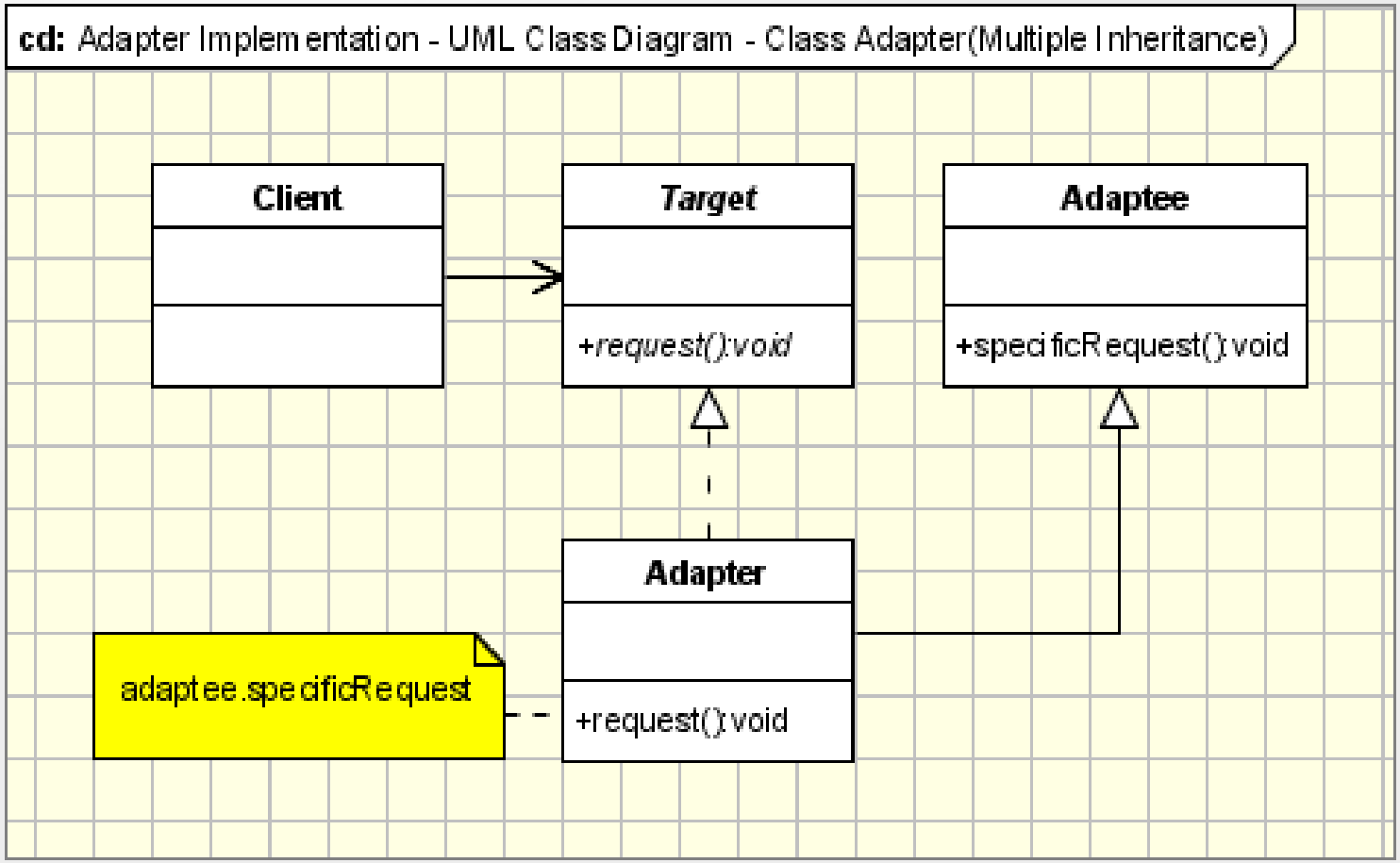
Adapter – Basato su delega



Adapter

- **Target** – definisce l'interfaccia di dominio usata dal Client.
- **Adapter** – adatta l'inerfaccia Adaptee all'interfaccia Target.
- **Adaptee** – definisce una interfaccia esistente che deve essere adattata.
- **Client –** interagisce con gli oggetti in conformità all'interfaccia Target

Adapter – Basato su eredità multipla



Bridge

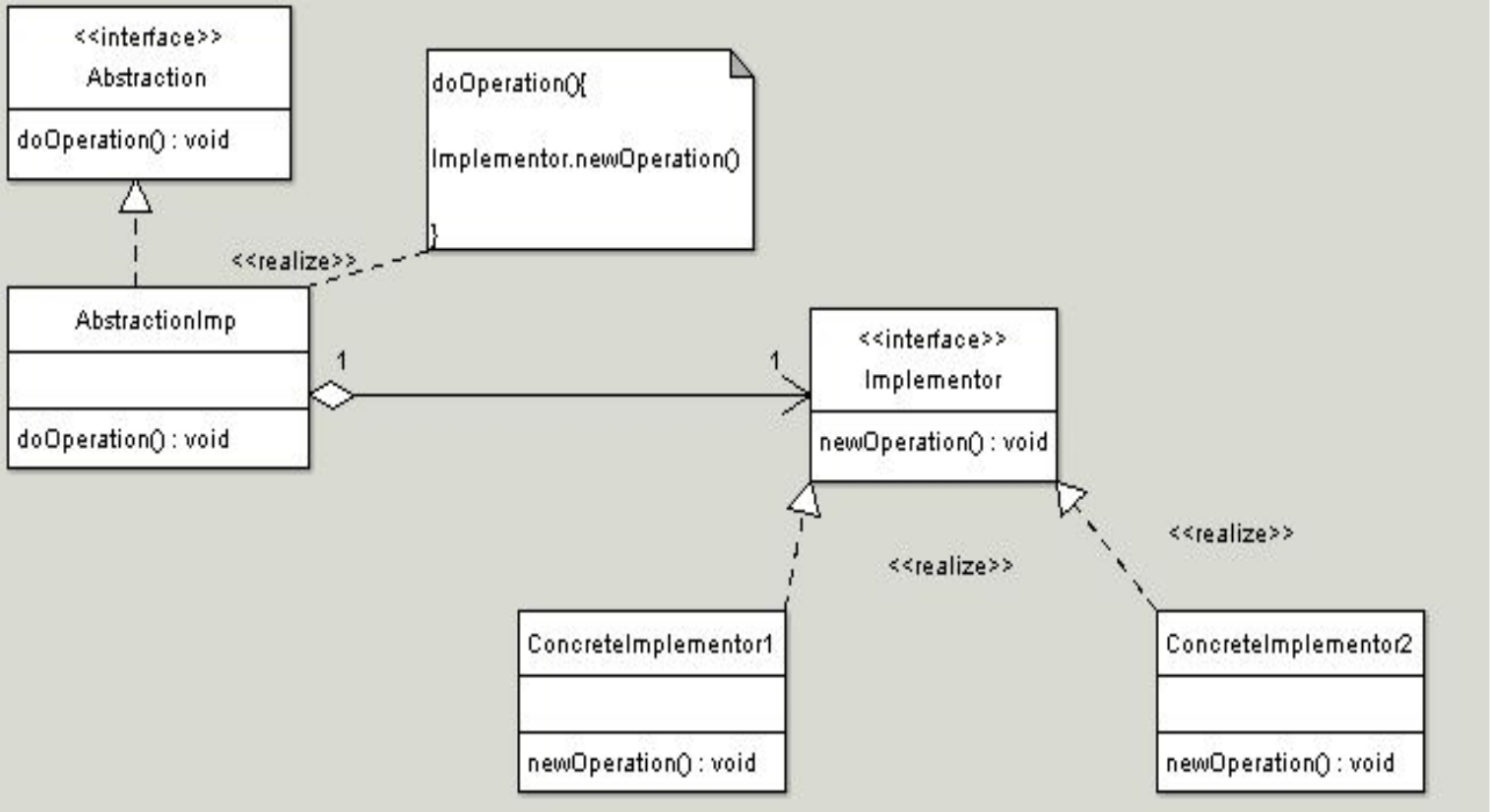
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Bridge (Structural)

- **Intent**
  - Disaccoppiare astrazione dall'implementazione in modo che possano variare indipendentemene.

- **Applicability**
  - Quando c'e' bisogno di evitare un legame permanente tra una astrazione e l'implementazione.
  - Quando si vuole scegliere dinamicamente l'implementazione di una interfaccia

Bridge: Implementazione



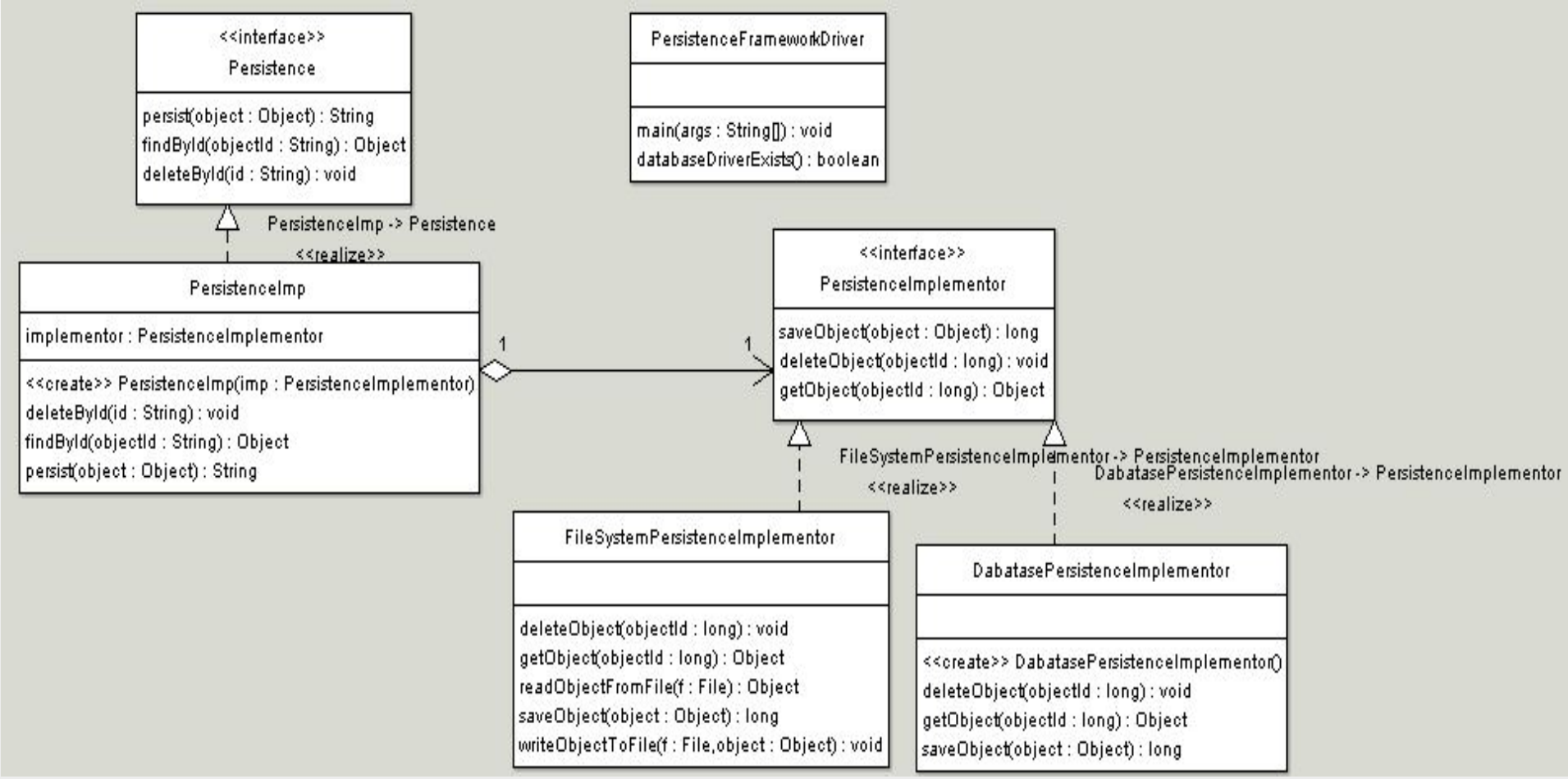
Bridge

- **Abstraction**: interfaccia astratta
- **AbstractionImp**: implementa una astrazione delegando a una interfaccia di tipo Implementor
- **Implementor**: interfaccia delle classi Implementor;

non deve essere in corrispondenza precisa con l'interfaccia astratta

- **Concrete Implementor**: implementa l'interfaccia Implementor.

Esempio: Object Persistence API



Persistence Interface

```
public interface Persistence {
/**
        * @param object
        * @return returns objectID
        */
       public String persist(Object object);
       /**
        * @param objectId
        * @return persisted Object
        */
       public Object findById(String objectId);
       /**
        * @param id
        */
       public void deleteById(String id);
}
```

PersistenceImp

```
public class PersistenceImp implements Persistence {

    private PersistenceImplementor implementor = null;

    public PersistenceImp(PersistenceImplementor imp) {

        this.implementor = imp;      }

    @Override
    public void deleteById(String id) {

        implementor.deleteObject(Long.parseLong(id));  }

    @Override
    public Object findById(String objectId) {

        return implementor.getObject(Long.parseLong(objectId));    }

    @Override
    public String persist(Object object) {

        return Long.toString(implementor.saveObject(object));}       }
```

Implementor Interface

```
/**
 * Implementor Interface
 */
public interface PersistenceImplementor {

        public long saveObject(Object object);

        public void deleteObject(long objectId);

        public Object getObject(long objectId);


}
```

File System Concrete Implementor

```
public class FileSystemPersistenceImplementor implements PersistenceImplementor{

    @Override
    public void deleteObject(long objectId) {

        File f = new File("/persistence/"+Long.toString(objectId));
        f.delete();
        return;       }

    @Override
    public Object getObject(long objectId) {

        File f = new File("/persistence/"+Long.toString(objectId));
        return readObjectFromFile(f);     }

    private Object readObjectFromFile(File f) {

        // open file  and load object  return the object
        return null;}
```

File System Concrete Implementor

```
@Override
public long saveObject(Object object) {

        long fileId = System.currentTimeMillis();

// open file
        File f = new File("/persistence/"+Long.toString(fileId));

// write file to Streanm
        writeObjectToFile(f,object);
        return fileId;
}

private void writeObjectToFile(File f, Object object) {

        // serialize object and write it to file

}

}
```

Database Concrete Implementor

```
public class DabatasePersistenceImplementor implements PersistenceImplementor{

public DabatasePersistenceImplementor() {
// load database driver        }

@Override
public void deleteObject(long objectId) {

        // open database connection  remove record }

@Override
public Object getObject(long objectId) {

        // open database connection read records create object from record
        return null; }

@Override
public long saveObject(Object object) {

        // open database connection create records for fields inside the object
        Return 0; }
}
```

Framework Driver

```
public class PersistenceFrameworkDriver {

public static void main(String[] args) {
        // this program needs a persistence framework
        // at runtime  an implementor is chosen between file system implementation and
        //database implememtor , depending on existence of databse drivers

        PersistenceImplementor implementor = null;

        if(databaseDriverExists()){
                implementor = new DabatasePersistenceImplementor();
                }else{
                implementor = new FileSystemPersistenceImplementor();
        }

        Persistence persistenceAPI = new PersistenceImp(implementor);

        Object o = persistenceAPI.findById("12343755");
        // do changes to the object  then persist
        persistenceAPI.persist(o);
```

Framework Driver

```
            // can also change implementor
            persistenceAPI = new PersistenceImp(new DabatasePersistenceImplementor());

            persistenceAPI.deleteById("2323");    }

    private static boolean databaseDriverExists() { return false; }
}
public class DabatasePersistenceImplementor implements PersistenceImplementor{

    public DabatasePersistenceImplementor() {
    // load database driver        }

    @Override
    public void deleteObject(long objectId) {

        // open database connection  remove record }

    @Override
    public Object getObject(long objectId) {

        // open database connection read records create object from record
        return null; }
```

Factory

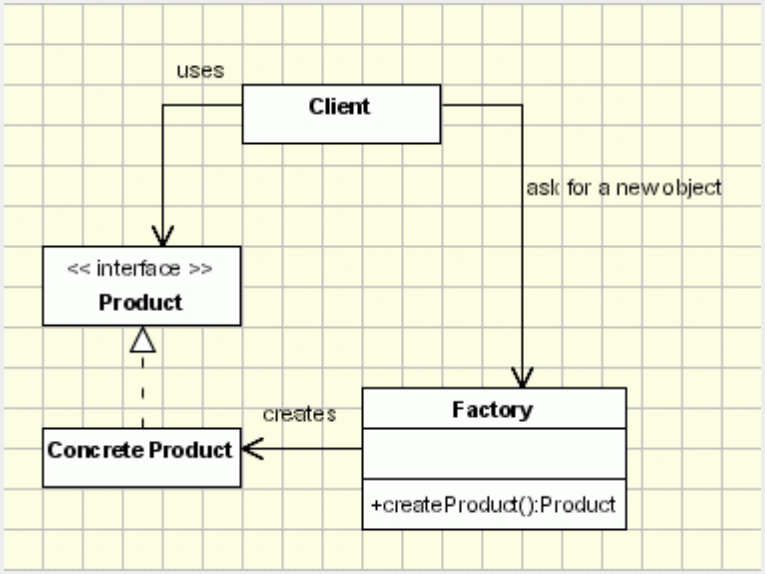
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Factory Method (Creational)

- **Intent**
  - Creare oggetti senza esporre la logica di istanziazione
  - Permettere di riferirsi al nuovo oggetto creato mediante una interfaccia comune.
- **Applicability**
  - Uno dei pattern più usati.

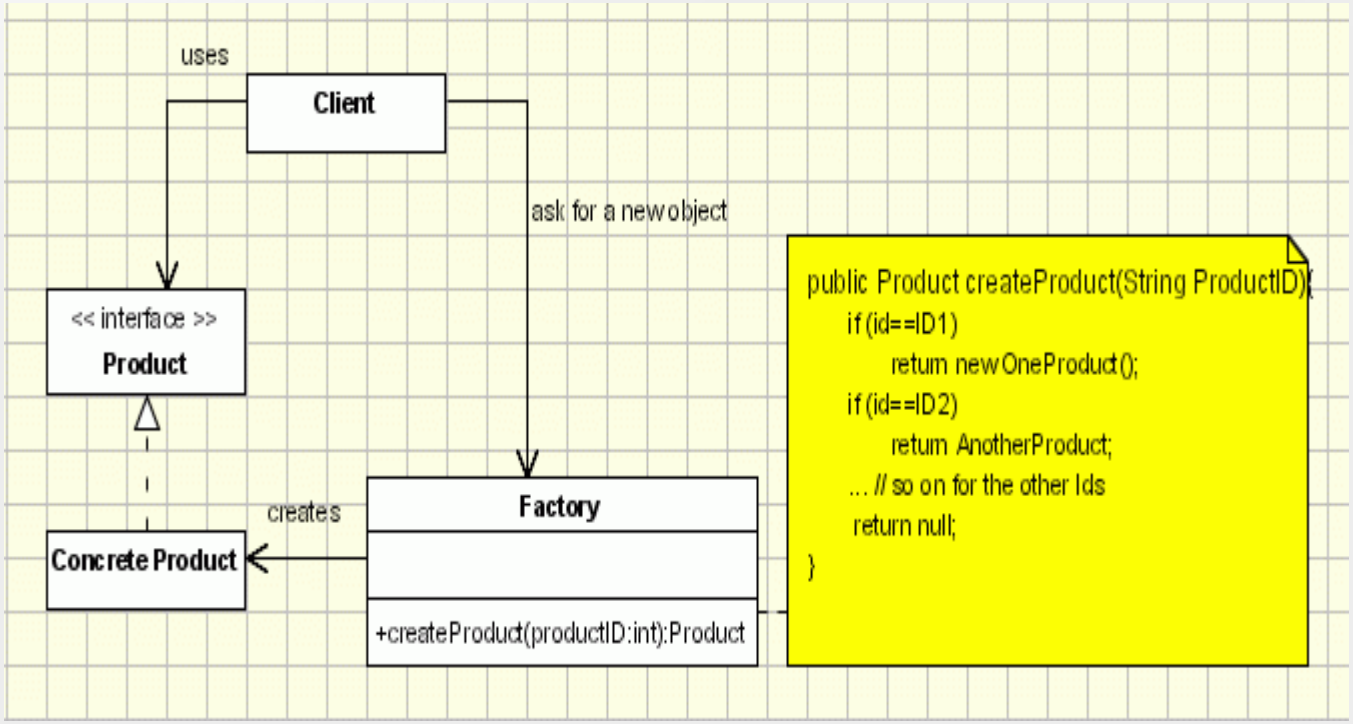
Factory Method (Creational)

- Il Client chiede la creazione alla factory la creazione di un oggetto di tipo specificato
- La factory crea l'oggetto e lo restituisce al Client con cast a un tipo prodotto astratto
- Il Client usa il prodotto con cast a prodotto astratto.



Factory Method (Implementazione 1)

- Implementazione base (viola il principio Open-Close)
- L'aggiunta di nuovi prodotti richiede la riscrittura della Factory.



Factory Method (Implementazione 1)

```
public class ProductFactory{
      public Product createProduct(String ProductID){
            if (id==ID1)
                  return new OneProduct();
            if (id==ID2) return
                  return new AnotherProduct();
            ... // so on for the other Ids

       return null; //if the id doesn't have any of the expected values
   }
   ...
}
```

Implementazione 2: registrazione e uso della riflessione.

Un nuovo prodotto concreto viene registrato alla sua creazione e prima dell'uso presso la factory associando un productId al tipo.

```
class ProductFactory
{
    private HashMap m_RegisteredProducts = new HashMap();
    public void registerProduct (String productID, Class productClass)
    {
        m_RegisteredProducts.put(productID, productClass);
    }
    public Product createProduct(String productID)
    {
        Class productClass = (Class)m_RegisteredProducts.get(productID);
        return (Product)productClass.newInstance();
    }
}
```

Implementazione 2: registrazione.

Registrazione esterna alla classe

```
public static void main(String args[]){
        Factory.instance().registerProduct("ID1", OneProduct.class);
    }
```

Registrazione interna alla classe.

```
class OneProduct extends Product
{
    static {
        Factory.instance().registerProduct("ID1",OneProduct.class);
    }
    ... }
```

Implementazione 2: registrazione senza uso della riflessione.

Un nuovo prodotto concreto viene registrato alla sua creazione e prima dell'uso presso la factory associando un productId ad una istanza del tipo che viene usata poi per le duplicazioni.

```
abstract class Product
{
        public abstract Product createProduct();
        ...     }
class OneProduct extends Product
{   ...   static
        {
                ProductFactory.instance().registerProduct("ID1", new OneProduct());
        }
        public OneProduct createProduct()
        {       return new OneProduct();            }
        ...
}
```

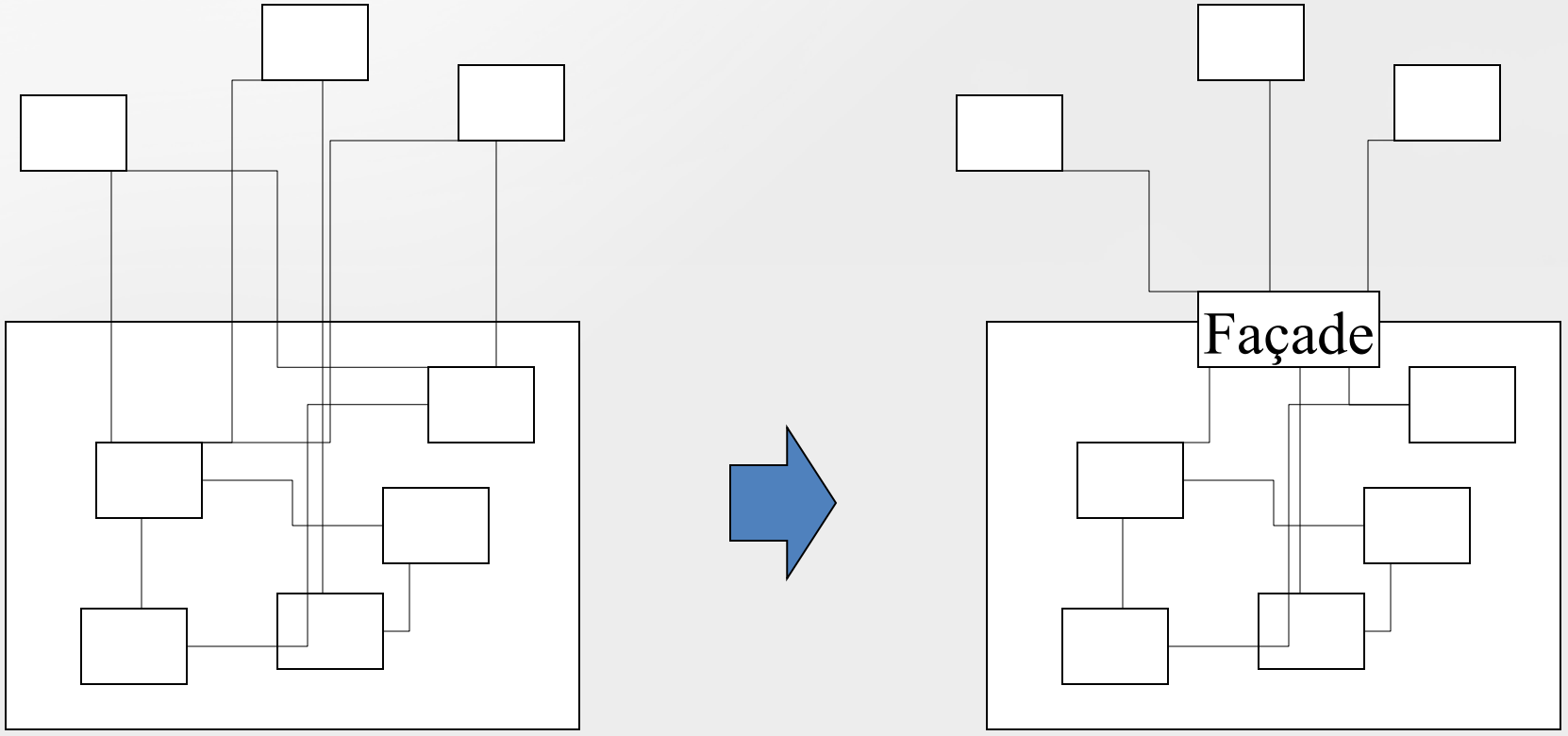
Implementazione 2: registrazione senza uso della riflessione.

```
class ProductFactory
{
    private HashMap m_RegisteredProducts = new HashMap();

    public void registerProduct (String productID, Product p)
    { m_RegisteredProducts.put(productID, p); }

    public Product createProduct(String productID)
    {
        Class productClass = (Class)m_RegisteredProducts.get(productID);
        return ((Product)m_RegisteredProducts.get(productID)).createProduct();
    }
    }
}
```

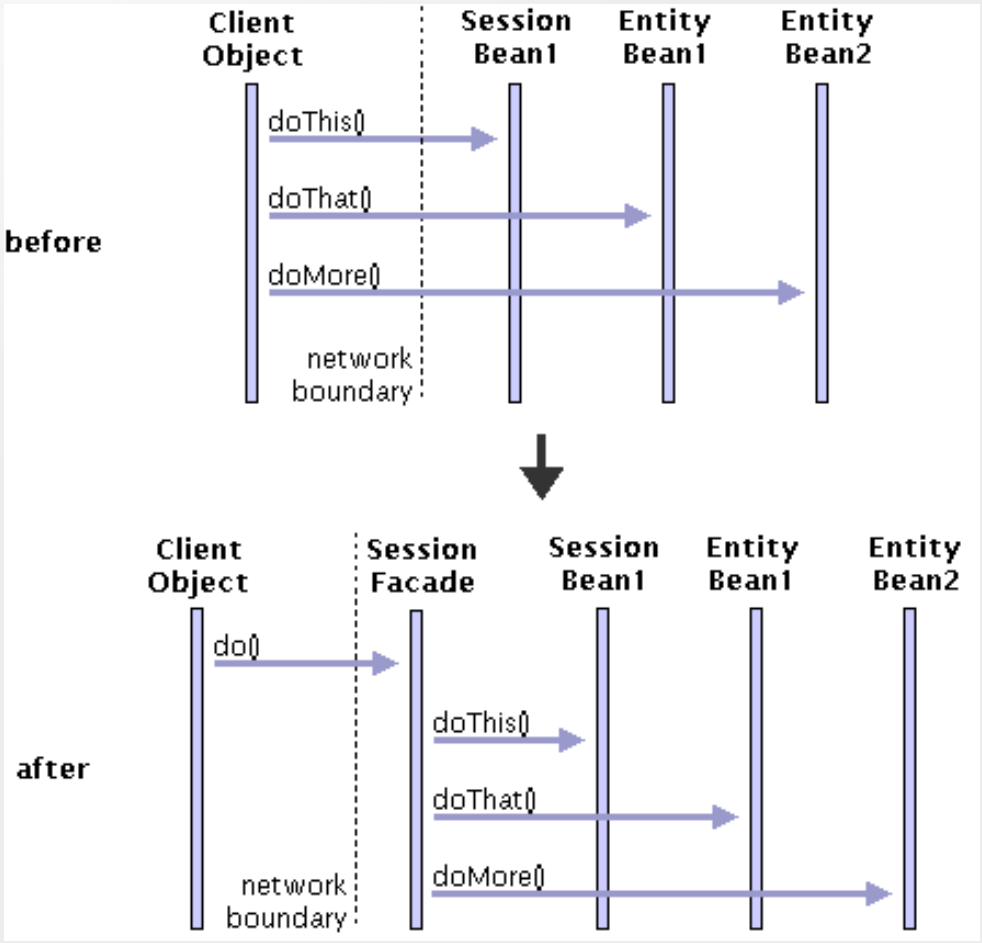
Facade

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

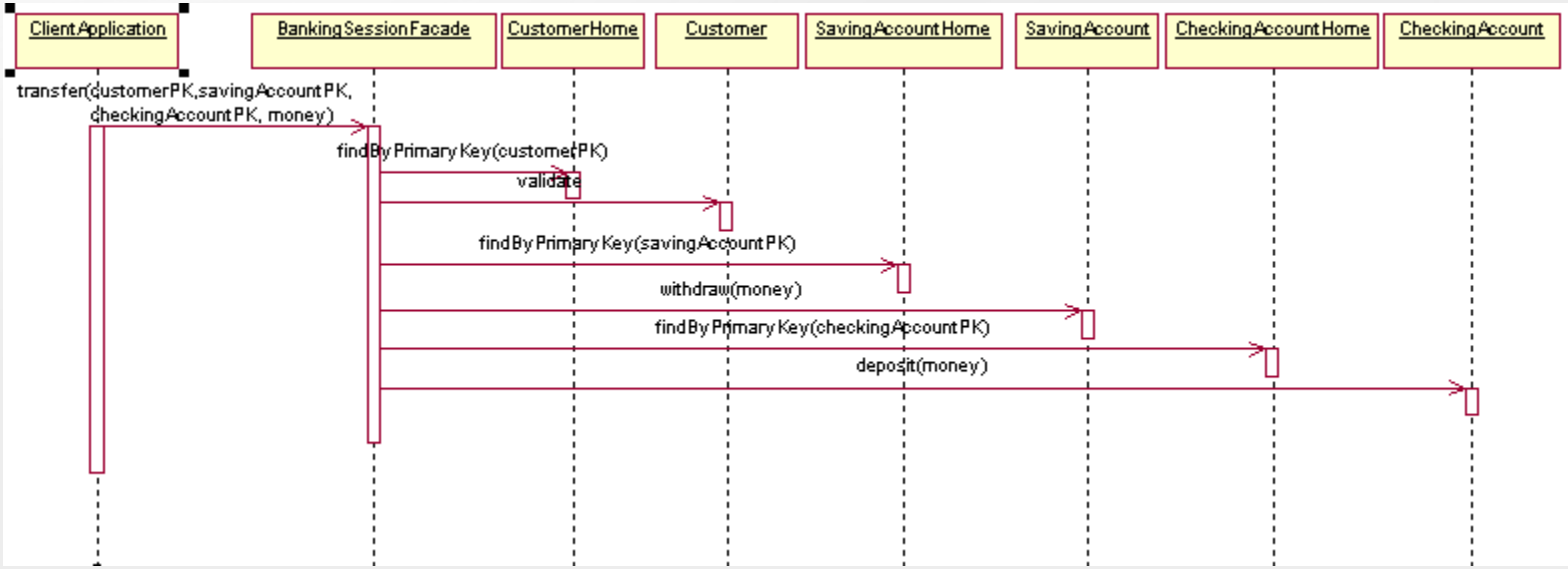
Façade



Façade


Client Object
Session Bean1
Entity Bean1
Entity Bean2
before
doThis()
doThat()
doMore()
network boundary
Client Object
Session Facade
Session Bean1
Entity Bean1
Entity Bean2
after
do()
doThis()
doThat()
doMore()
network boundary

Façade - Example



Strategy

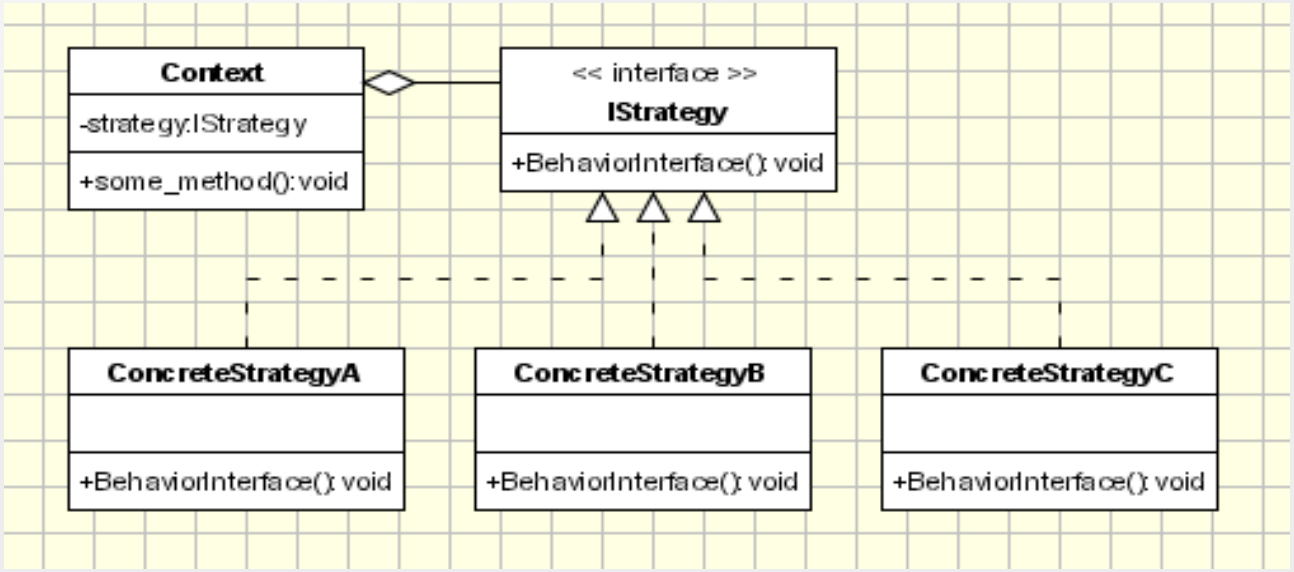
		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Strategy Pattern

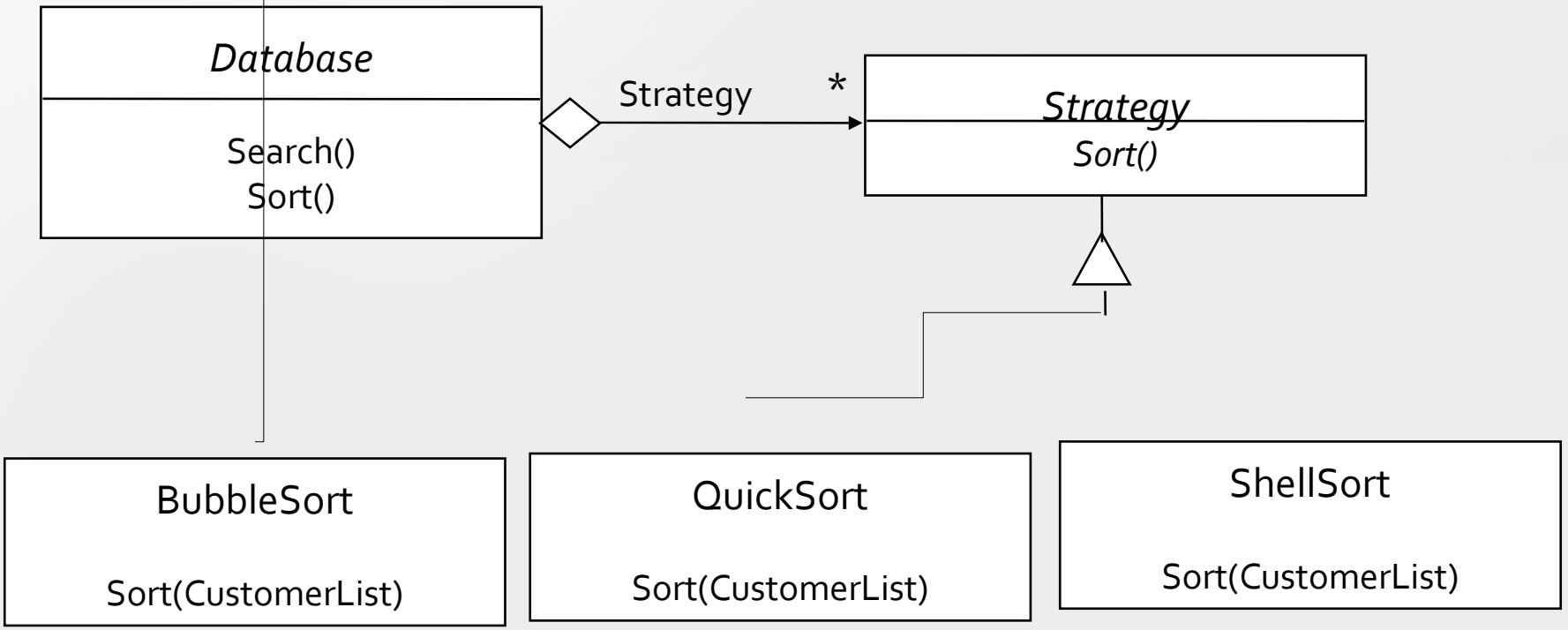
- Sono disponibili molti algoritmi diversi per risolvere lo stesso problema
- Esempi:
  - Frazione uno stream di testo in lineee
  - Parsing un insieme di token in un albero sintattico astratto
  - Ordinamento di una lista di entità
- Diversi algoritmi potrebbero essere adatti in circosatnze diverse
  - Rapid prototyping vs consegna del prodotto finito
- Non si vogliono supportatre tuttin gli algoritmi se non necessario
- Se serve un nuovo algoritmo deve essere aggiunto senza alterare l'applicativo.

Strategy Pattern

- Policy decide quale Strategy sia migliore dato il Context



Applicazone dello Strategy Pattern in una applicazione Database



Applicabilità dello Strategy Pattern

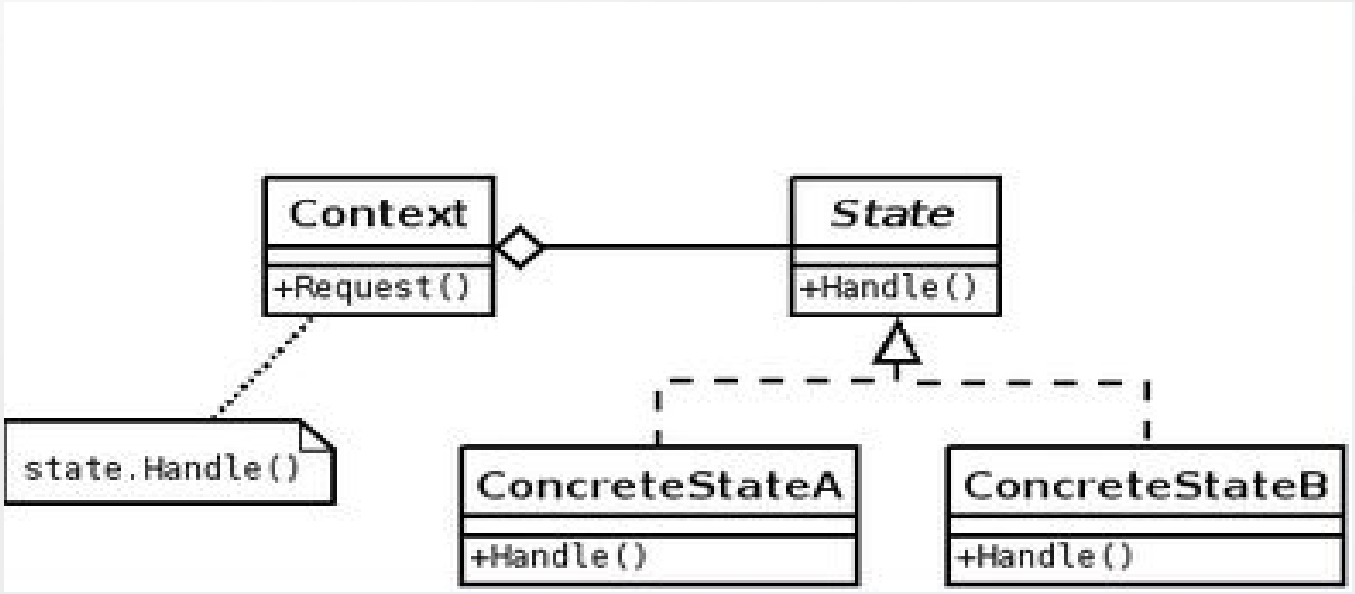
- Molte classi differiscono solo rispetto al loro comportamento. Lo Strategy permette di configurare una singola classe con uno dei molti possibili comportamenti.
- Molte varianti dello stesso algoritmo possono essere usate e scelte run-time in funzione delle necessità.
  - Le varianti dell'algoritmo possono essere strutturate in una gerarchia di classi e la specializzazione di interesse può essere scelta run-time

State Pattern

- Permette ad un oggetto di cambiare il suo comportamento al cambiare del suo stato interno. L'oggetto si comporterà come avesse cambiato la sua classe
- Applicabilità:
  - Esistono metodi con grandi blocchi di codice per scelte condizionali annidate il cui esito dipende dallo stato dell'oggetto.
  - Lo stato è di solito rappresentato da una o più costanti.
  - Spesso molti metodi contengono la stessa struttura condizionale.
  - Il patetrn inserisce ogni ramo condizionale in una classe separata.

State Pattern

- Struttura del pattern



State Pattern

Context:

- definisce un'interfaccia usabile dai client
- Contiene un'istanza di una sottoclasse di State ConcreteState

State

- Definisce un'interfaccia che incapsula il comportamento associato a uno stato particolare

Concrete State

- Ogni sottoclasse implementa un comportamento associato ad uno stato di Context.

Esempio senza uso di pattern

Produzione cottura e consegna della pizza, flussi di controllo dipendenti dallo stato.

-

```
public class Pizza {

 public final static int COOKED = 0;
 public final static int BAKED = 1;
 public final static int DELIVERED = 2;

 private String name;

 int state = COOKED;

 public String getName() {
  return name;
 }

 public void setName(String name) {
  this.name = name;
 }

 public int getState() {
  return state;
 }

 public void setState(int state) {
  this.state = state;
 }
```

```
 public void bake() throws Exception {

  if(state == COOKED) {
   System.out.print("Baking the pizza...");
   state = BAKED;
  }
  else if(state == BAKED) {
   throw new Exception("Can't bake a pizza already baked");
  }
  else if(state == DELIVERED) {
   throw new Exception("Can't bake a pizza already delivered"
);
  }
 }

 public void deliver() throws Exception {

  if(state == COOKED) {
   throw new Exception("Can't deliver a pizza not baked yet")
;
  }
  else if(state == BAKED) {
   System.out.print("Delivering the pizza...");
   state = DELIVERED;
  }
  else if(state == DELIVERED) {
   throw new Exception("Can't deliver a pizza already deliver
ed");
  }
 }
}
```

Esempio con uso di state pattern

- PizzaState.class

```
public interface PizzaState {

 void bake() throws Exception;

 void deliver() throws Exception;
}
```

CookedPizzaState.class

```
public class CookedPizzaState implements PizzaState {

 private Pizza pizza;

 public CookedPizzaState(Pizza pizza) {
  this.pizza = pizza;
 }

 public void bake() throws Exception {
  System.out.print("Baking the pizza...");
  pizza.setState(pizza.getBakedState());
 }

 public void deliver() throws Exception {
  throw new Exception("Can't deliver a pizza not baked yet");
 }

}
```

Esempio con uso di state pattern (2)

Pizza.class

```
public class Pizza {

 PizzaState cookedState;
 PizzaState bakedState;
 PizzaState deliveredState;

 private String name;

 //State initialization
 private PizzaState state = cookedState;

 public Pizza() {
  cookedState = new CookedPizzaState(this);
  bakedState = new BakedPizzaState(this);
  deliveredState = new DeliveredPizzaState(this);
 }

 public String getName() {
  return name;
 }

 public void setName(String name) {
  this.name = name;
 }

 public PizzaState getState() {
  return state;
 }

 public void setState(PizzaState state) {
  this.state = state;
 }

 public void bake() throws Exception {
  this.state.bake();
 }

 public void deliver() throws Exception {
  this.state.deliver();
 }

 public PizzaState getCookedState() {
  return createdState;
 }

 public PizzaState getBakedState() {
  return bakedState;
 }

 public PizzaState getDeliveredState() {
  return deliveredState;
 }
}
```

Summary

- Structural Patterns
  - Focus: How objects are composed to form larger structures
  - Problems solved:
    - Realize new functionality from old functionality,
    - Provide flexibility and extensibility
- Behavioral Patterns
  - Focus: Algorithms and the assignment of responsibilities to objects
  - Problem solved:
    - Too tight coupling to a particular algorithm
- Creational Patterns
  - Focus: Creation of complex objects
  - Problems solved:
    - Hide how complex objects are created and put together

(Design) Pattern References

- The Timeless Way of Building, Alexander; Oxford, 1979; ISBN 0-19-502402-8
- A Pattern Language, Alexander; Oxford, 1977; ISBN 0-19-501-919-9
- Design Patterns, Gamma, et al.; Addison-Wesley, 1995; ISBN 0-201-63361-2; CD version ISBN 0-201-63498-8
- Pattern-Oriented Software Architecture, Buschmann, et al.; Wiley, 1996; ISBN 0-471-95869-7
- Analysis Patterns, Fowler; Addison-Wesley, 1996; ISBN 0-201-89542-0
- Smalltalk Best Practice Patterns, Beck; Prentice Hall, 1997; ISBN 0-13-476904-X
- The Design Patterns Smalltalk Companion, Alpert, et al.; Addison-Wesley, 1998; ISBN 0-201-18462-1
- AntiPatterns, Brown, et al.; Wiley, 1998; ISBN 0-471-19713-0