



Università degli Studi di Napoli "Federico II"

Ingegneria del Software

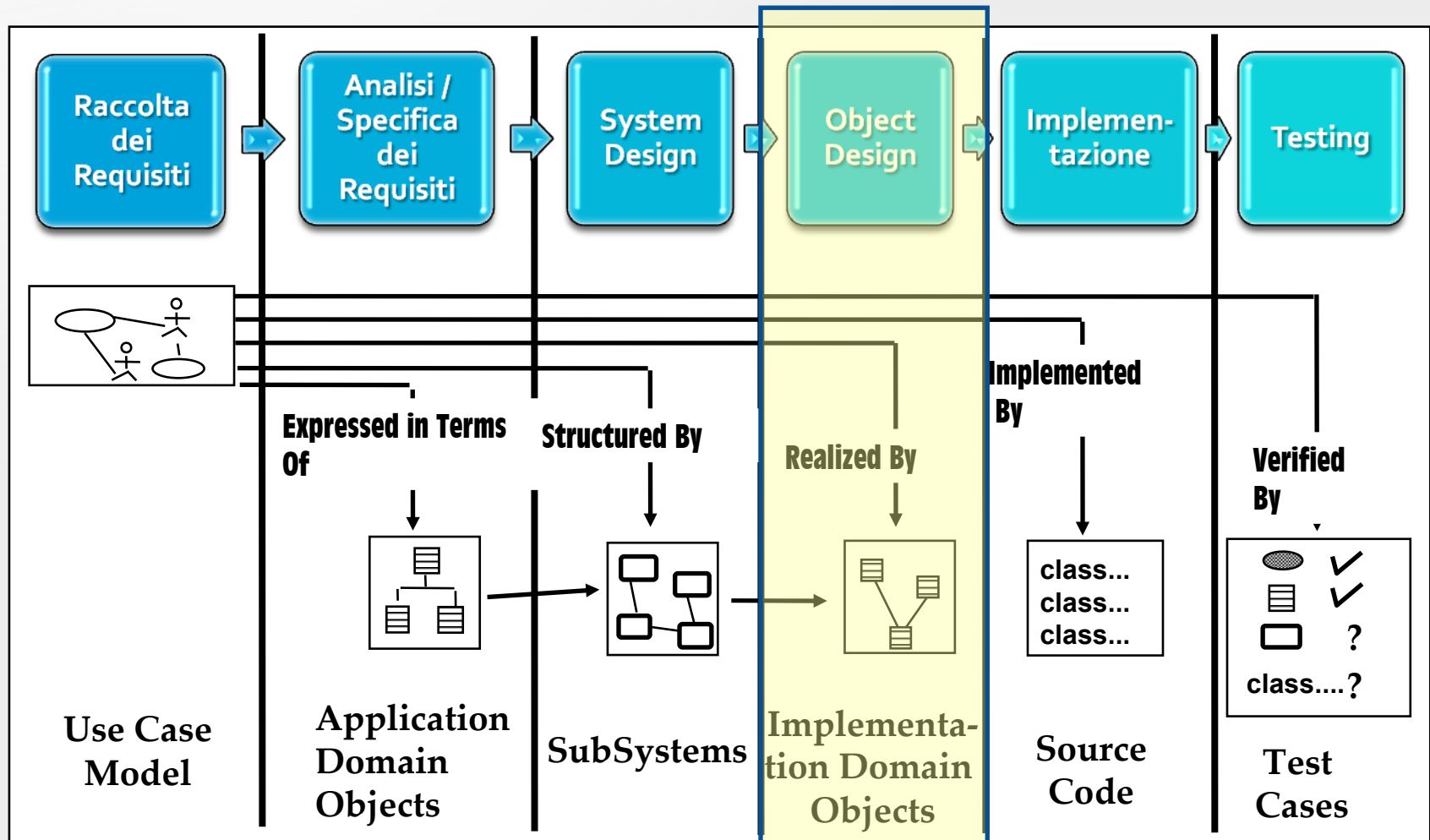
a.a. 2013/14

Design Pattern

Obiettivi della lezione

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

Software Lifecycle Activities



Architetture e Design Patterns

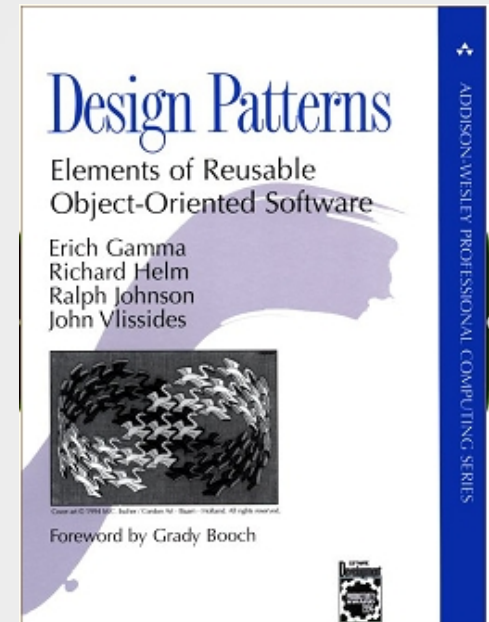
- Le Architetture definiscono la struttura di un sistema ad alto livello (system design)
 - Definizione della struttura di moduli
 - Grazie alle soluzioni architetture standard, si ha la disponibilità di una vasta scelta di soluzioni largamente sperimentate per una variegata casistica di problemi
- Situazione 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

Motivazioni

- L'OO usa strutture di progettazione ricorrenti che promuovono
 - Astrazione
 - Flessibilità
 - Modularità
 - Eleganza
 - Riutilizzo
 - 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
- ...

Classificazione del catalogo GoF

		<i>Purpose</i>		
		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

		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	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**

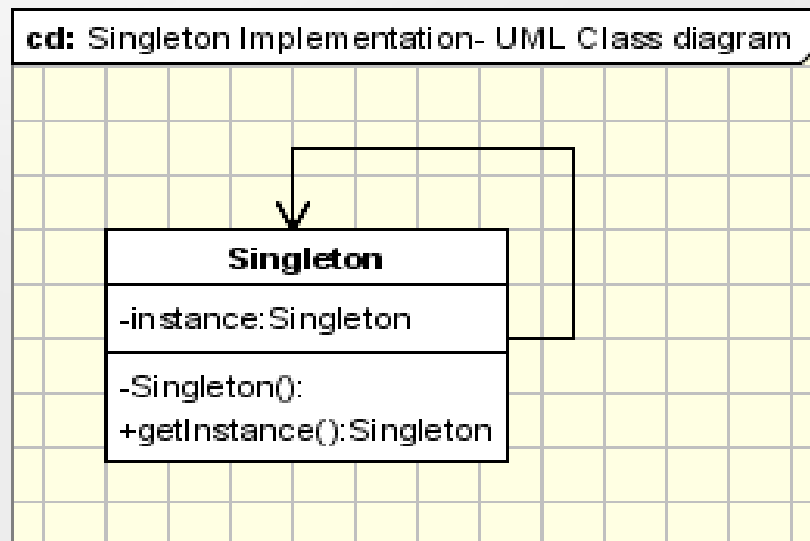
- Assicura che istanziato un solo esemplare per una classe
- Fornisce un punto di accesso centralizzato per una struttura.

- **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 fornire 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.

		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	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)

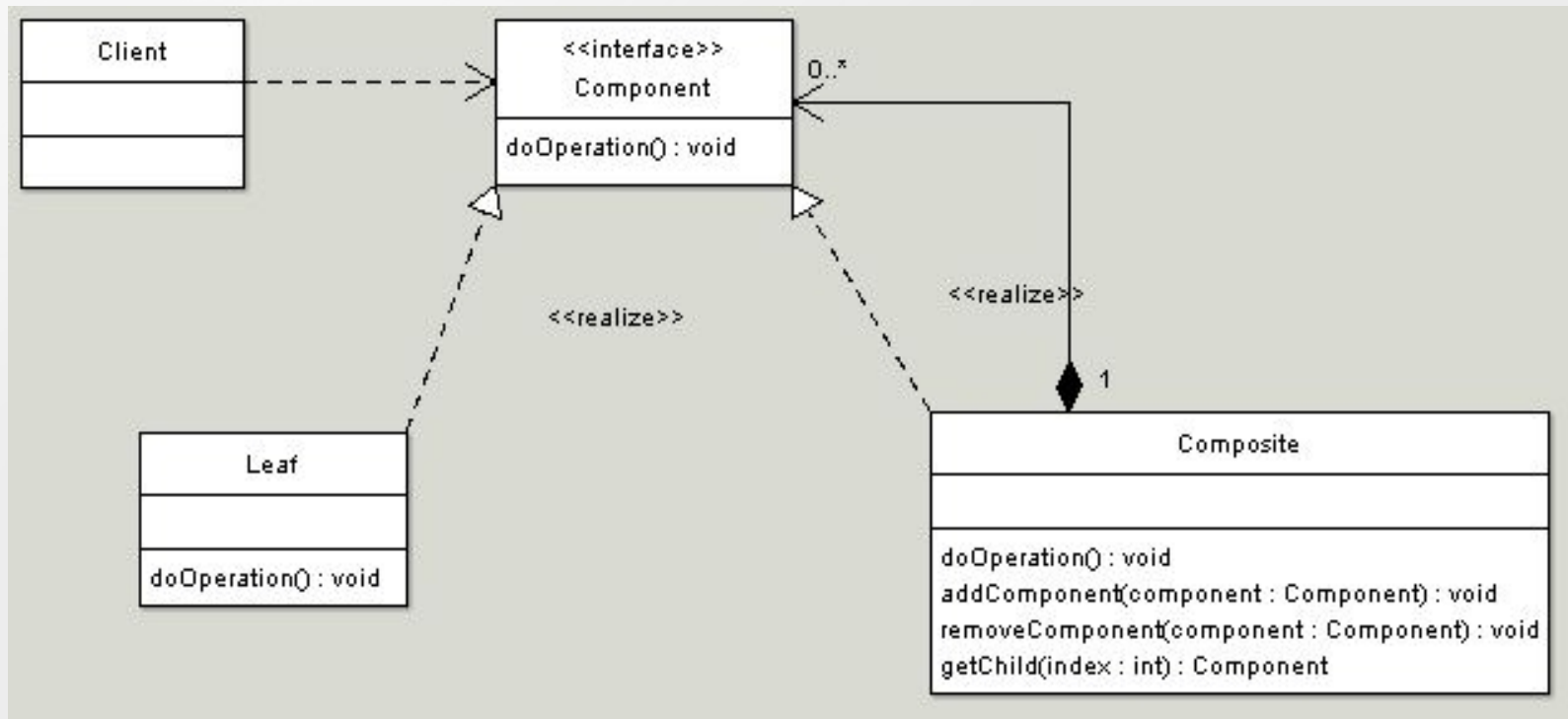
- **Intent**

- Comporre oggetti in strutture ad albero per rappresentare gerarchie parti-tutto
- Permettere al client di di trattare uniformemente oggetti individuali e composizione di oggetti

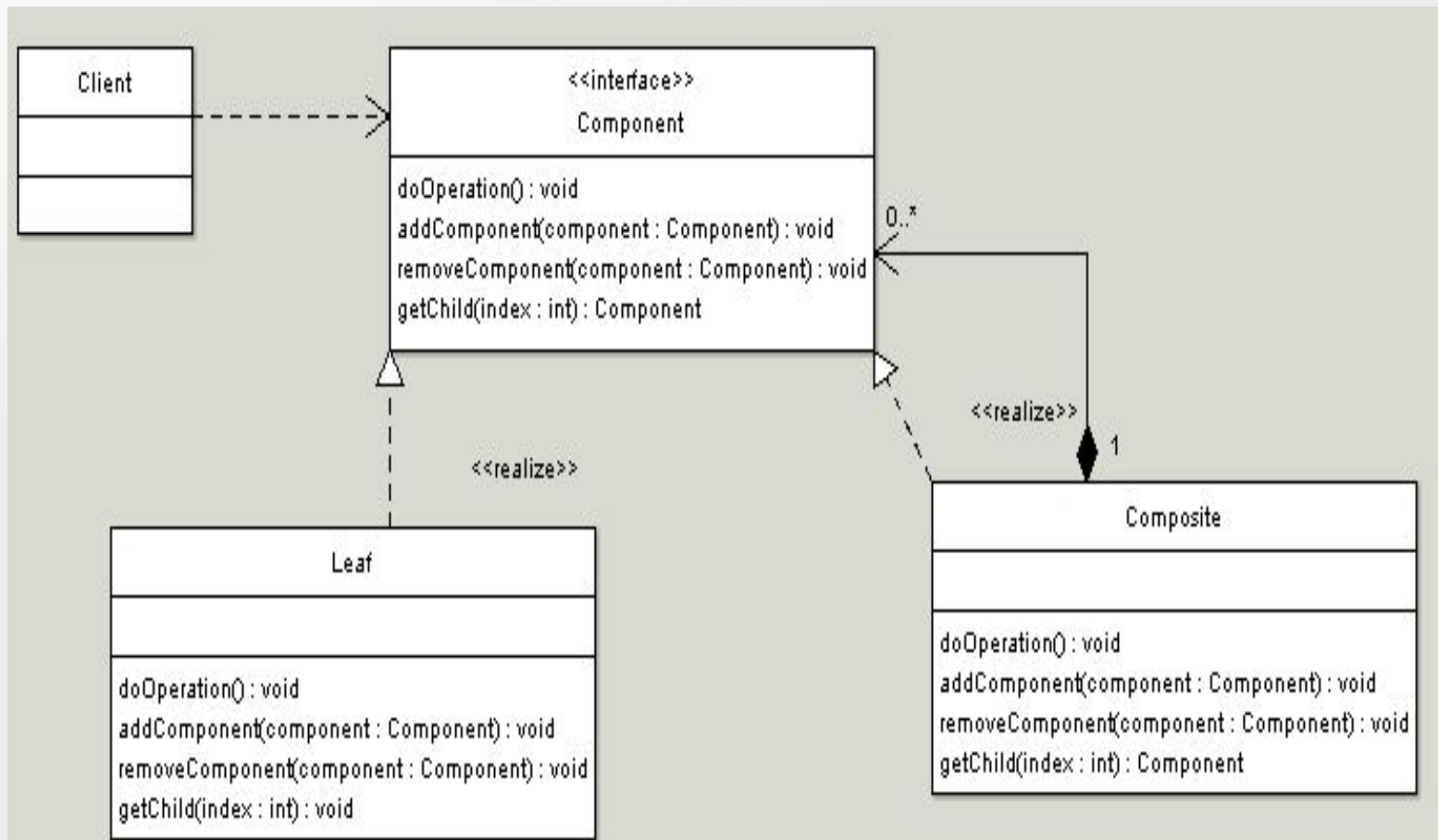
- **Applicability**

- Composizione ricorsiva parti-tutto di oggetti,
- Gestione uniforme di oggetti individuali e composizioni

Composite: Implementazione

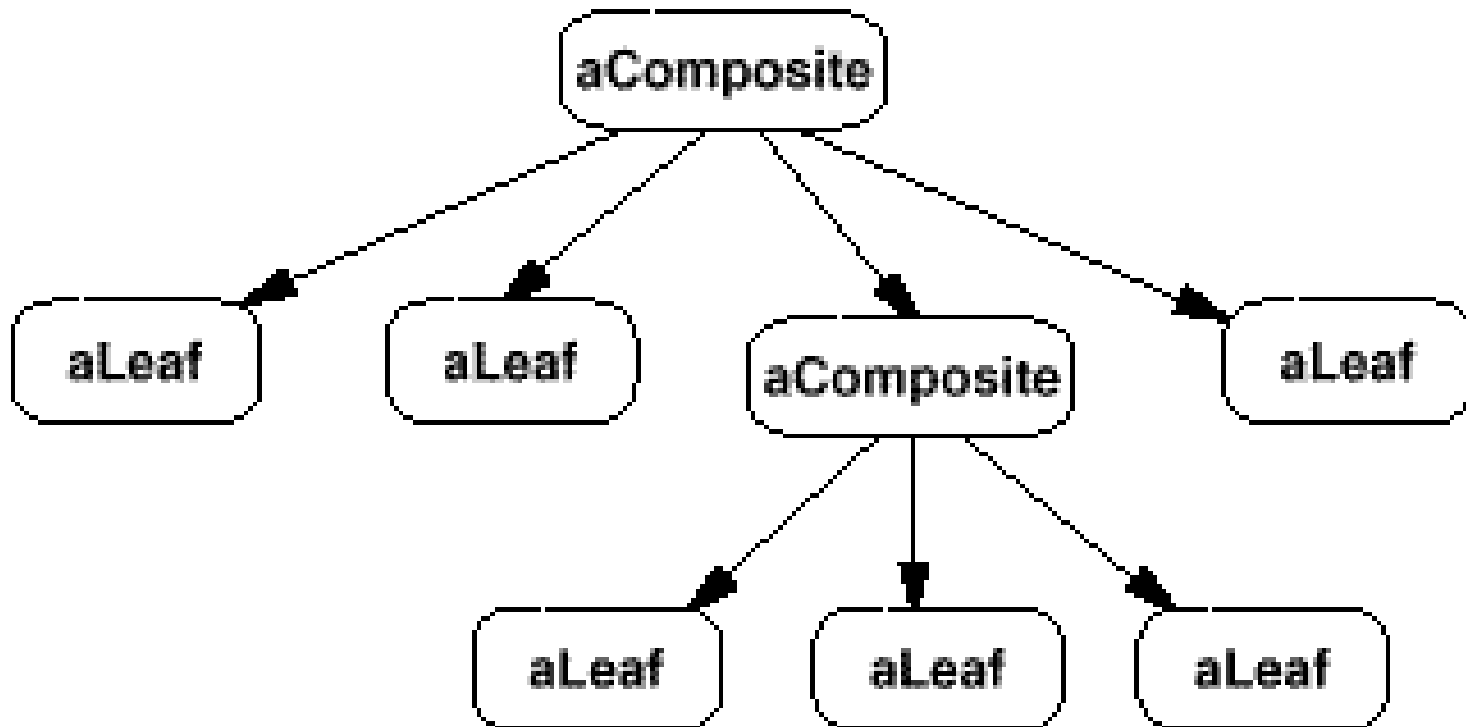


Composite: Implementazione alternativa



Composite (Cont'd)

- Object Structure:



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
- Responsabilità per la cancellazione dei figli.

		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	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

Observer

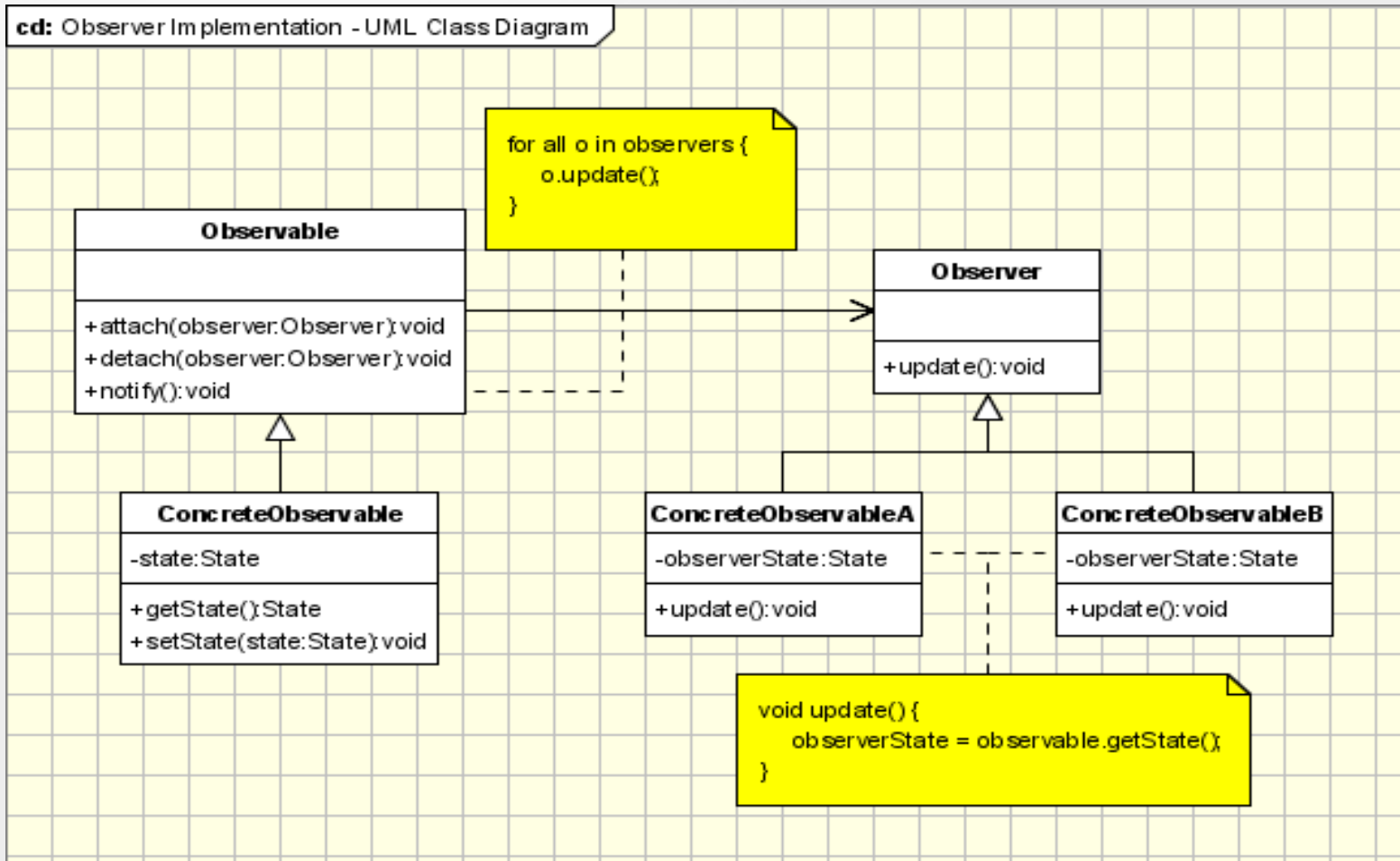
- **Intent**

- Definire una dipendenza uno-a-molti tra gli oggetti
- Quando un oggetto cambia stato i suoi dipendenti vengono notificati ed aggiornati automaticamente

- **Applicability**

- Il cambiamento di stato di un oggetto si deve riflettere nel cambiamento di stati in molti oggetti mantenendo 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.

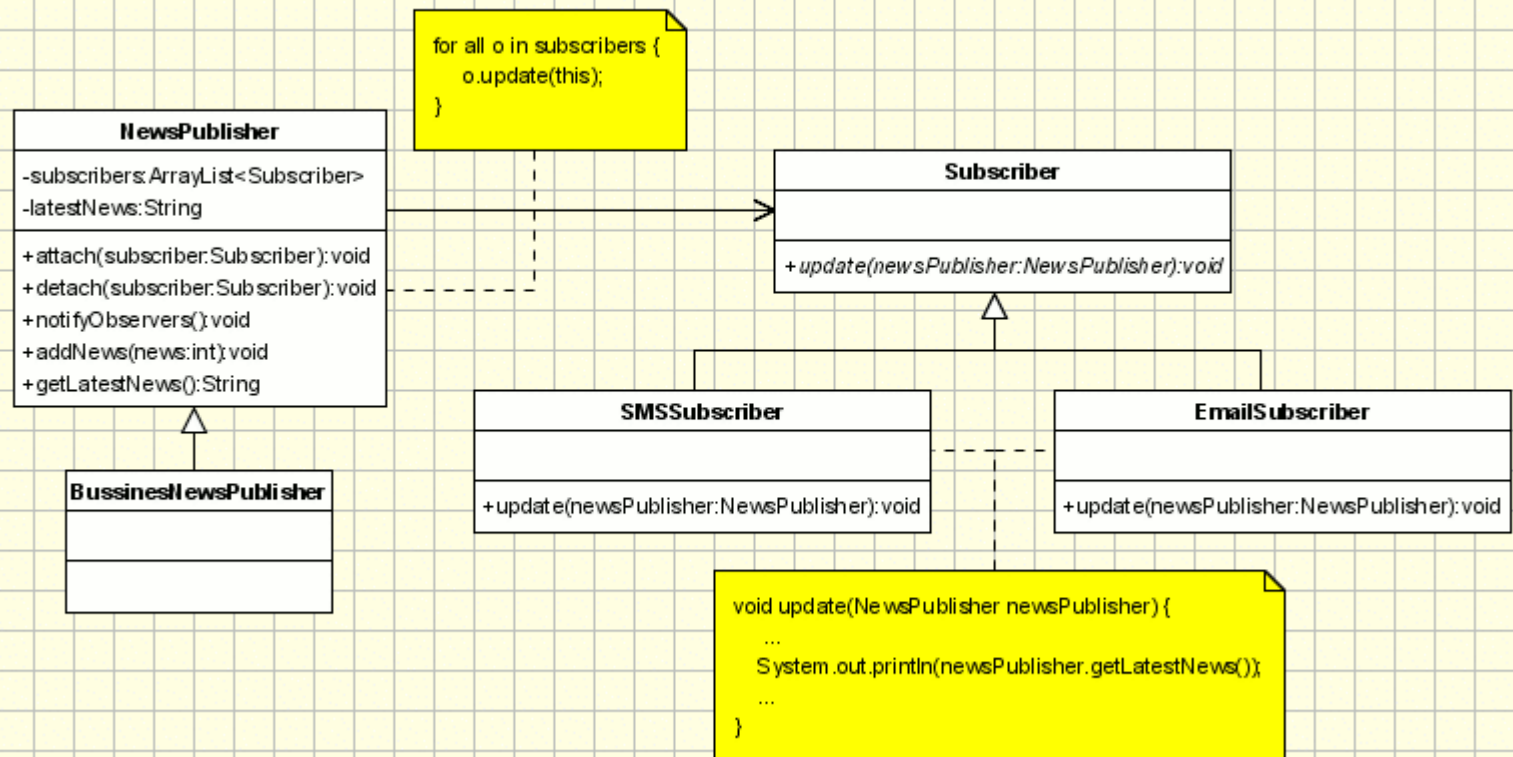
ConcreteObservable - concrete Observable class. Tiene lo stato degli oggetti It maintain the state of the object e a fronte di cambiamenti notifica gli osservatori noti..

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

ConcreteObserverA, ConcreteObserver2 – Observer concreti

Esempio: NewsAgency

cd: Observer NewsPublisher Example - UML Class Diagram



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 multi-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)

		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	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

Mediator (Behavioural)

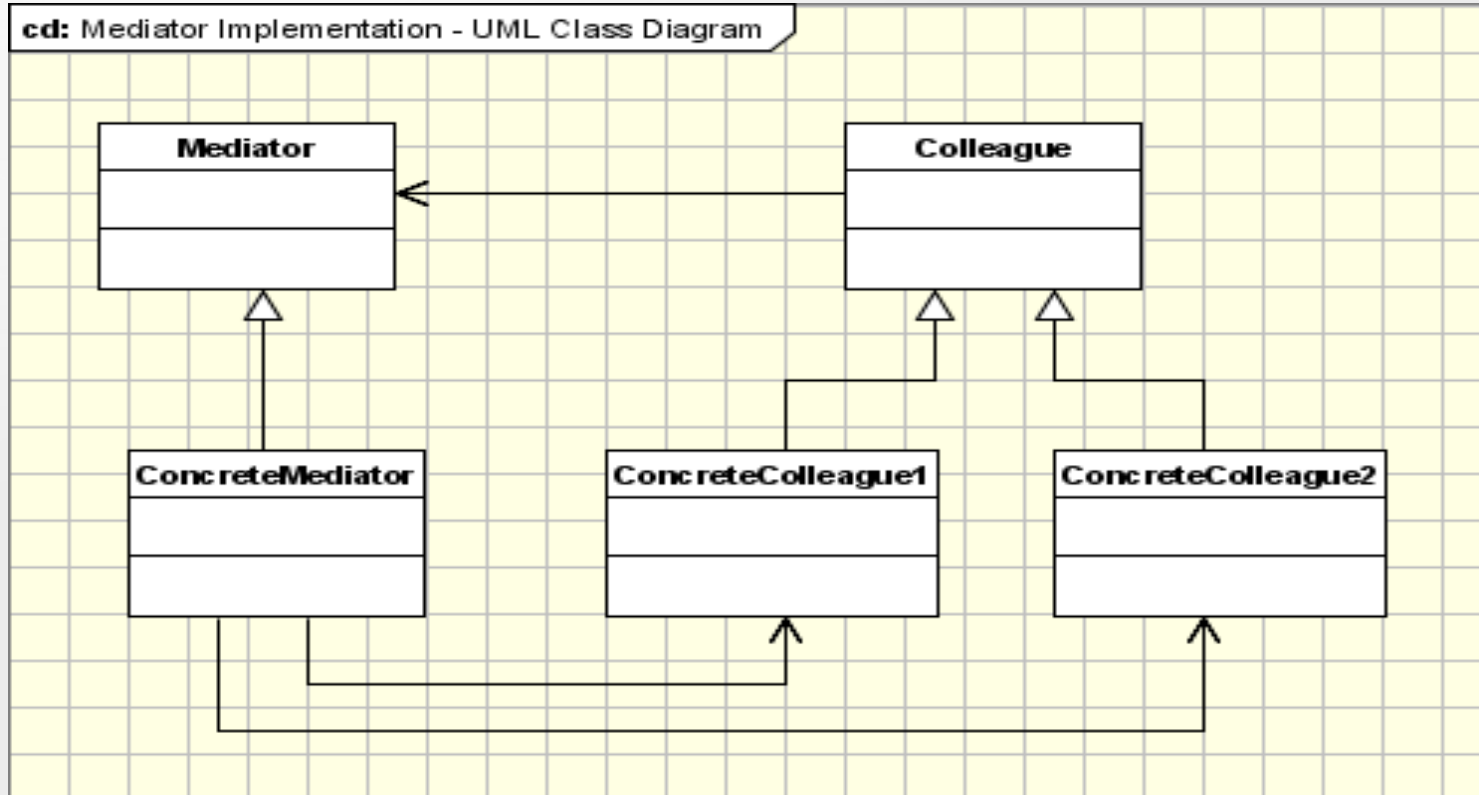
- **Intent**

- 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 modo ben definito ma complesso
- Il riutilizzo di un oggetto è reso difficile perché fa riferimento diretto e comunica con molti altri.

Mediator: Implementazione



Mediator: Implementazione

- **Mediator:** definisce una interfaccia per comunicare con oggetti Colleague
- **ConcreteMediator:** conosce le classi colleague e mantiene i riferimenti agli oggetti colleague
- **Colleague 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

- **Colleague (Participant):** interfaccia per i partecipanti; comunicano con il mediatore invece di comunicare con i colleghi
- **ConcreteColleague (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.

		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	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

Adapter (Structural)

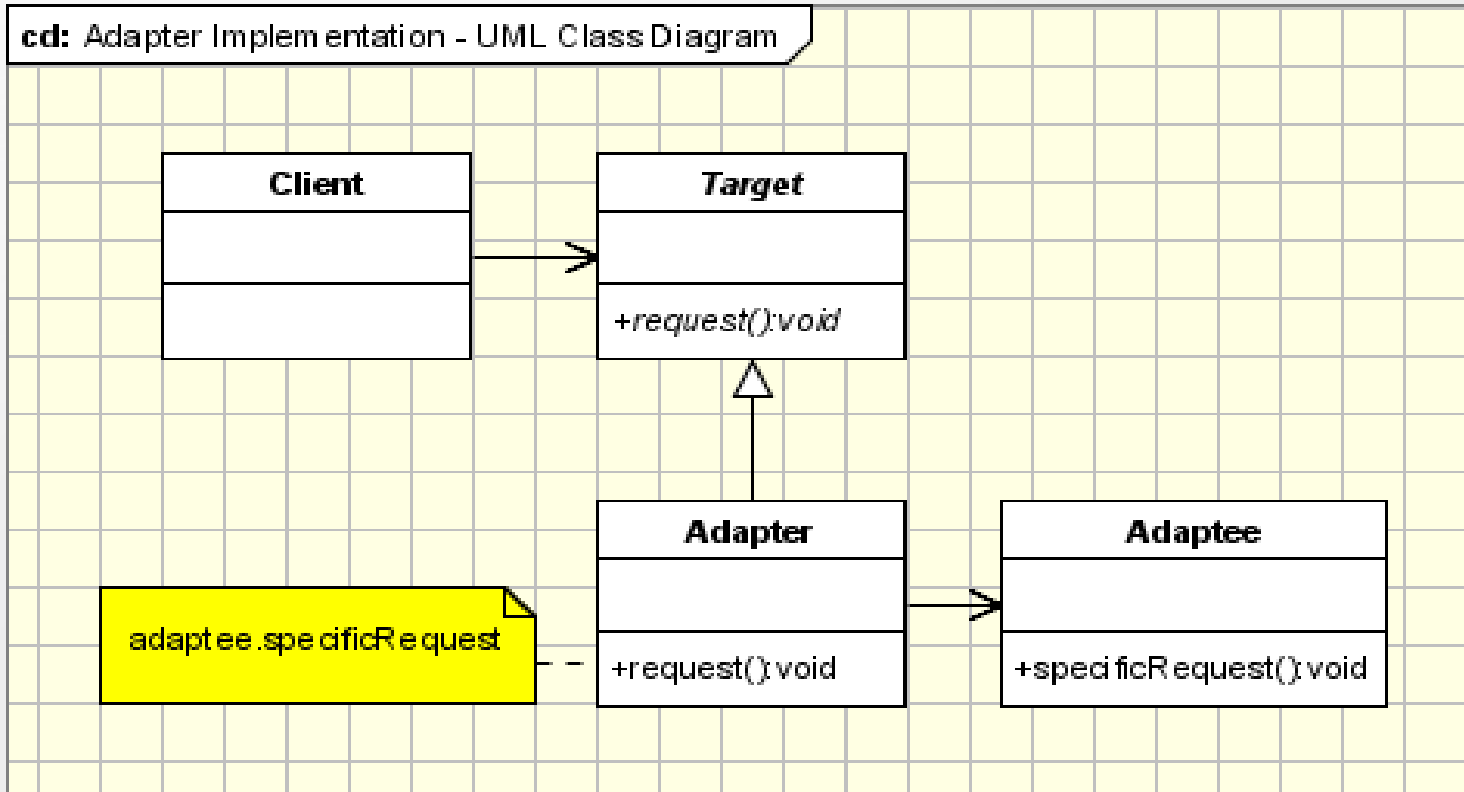
- **Intent**

- 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 implementa 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.

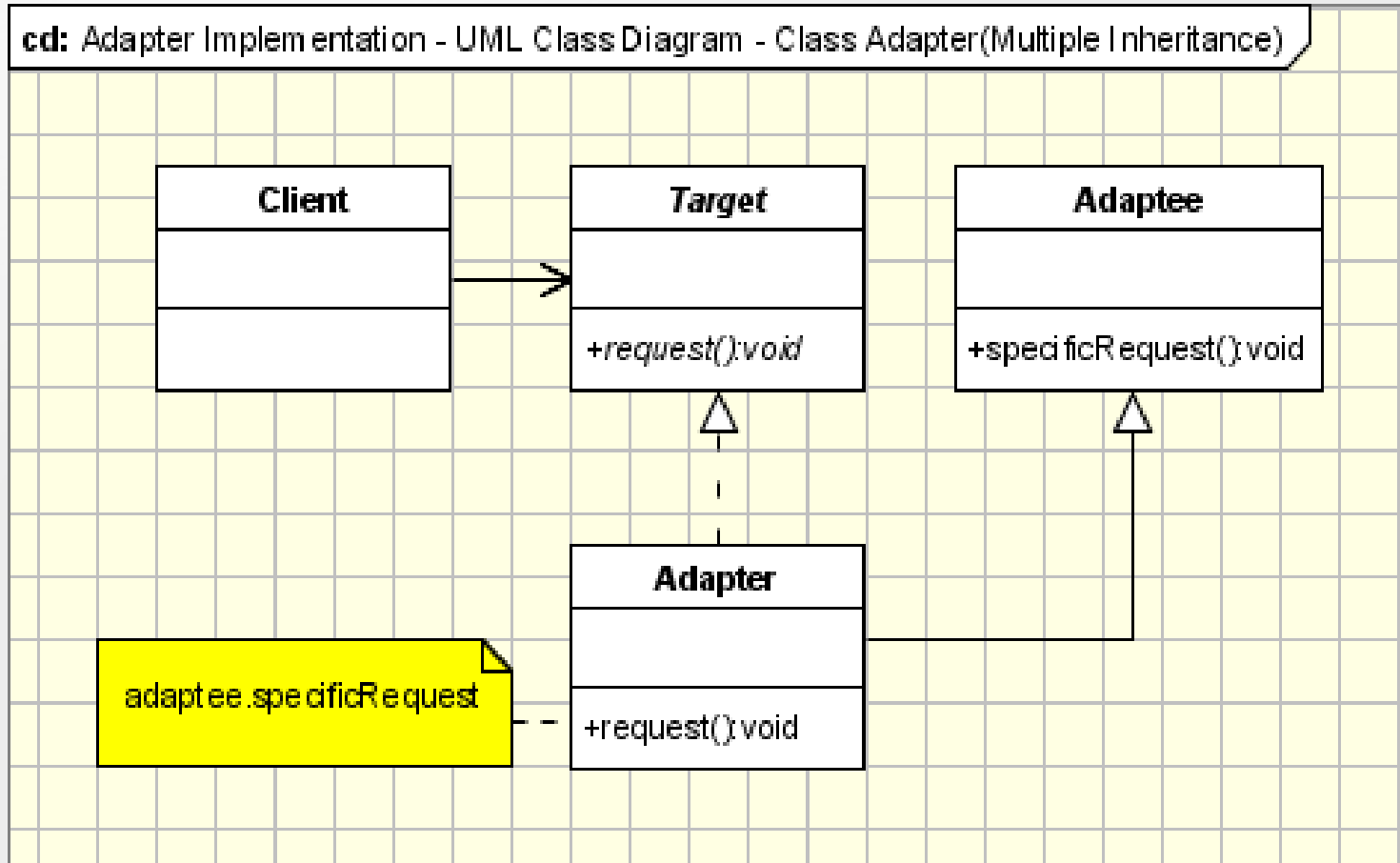
Adapter – Basato su delega



Adapter

- **Target** – definisce l'interfaccia di dominio usata dal Client.
- **Adapter** – adatta l'interfaccia 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



		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	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

Bridge (Structural)

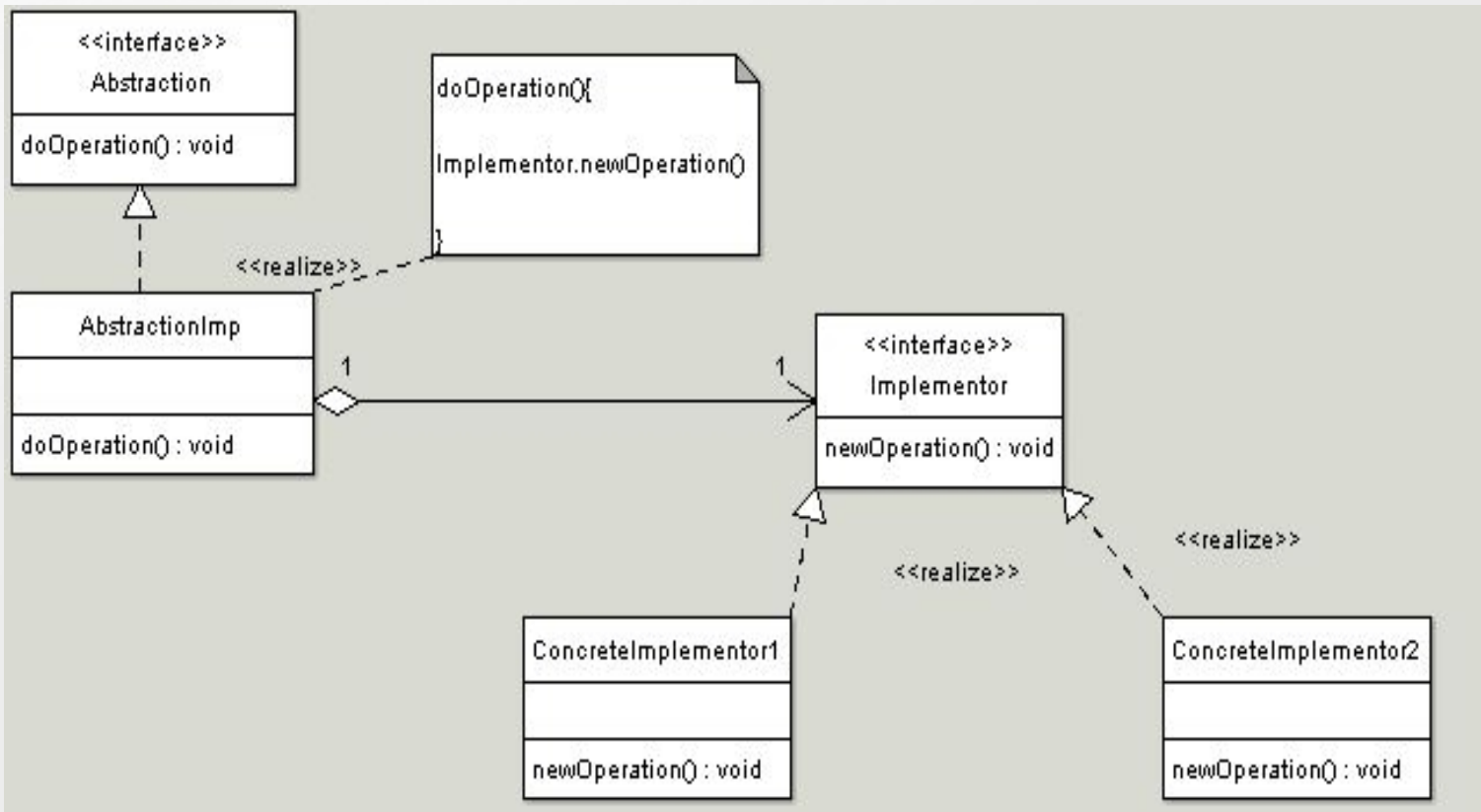
- **Intent**

- Disaccoppiare astrazione dall'implementazione in modo che possano variare indipendentemente.

- **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

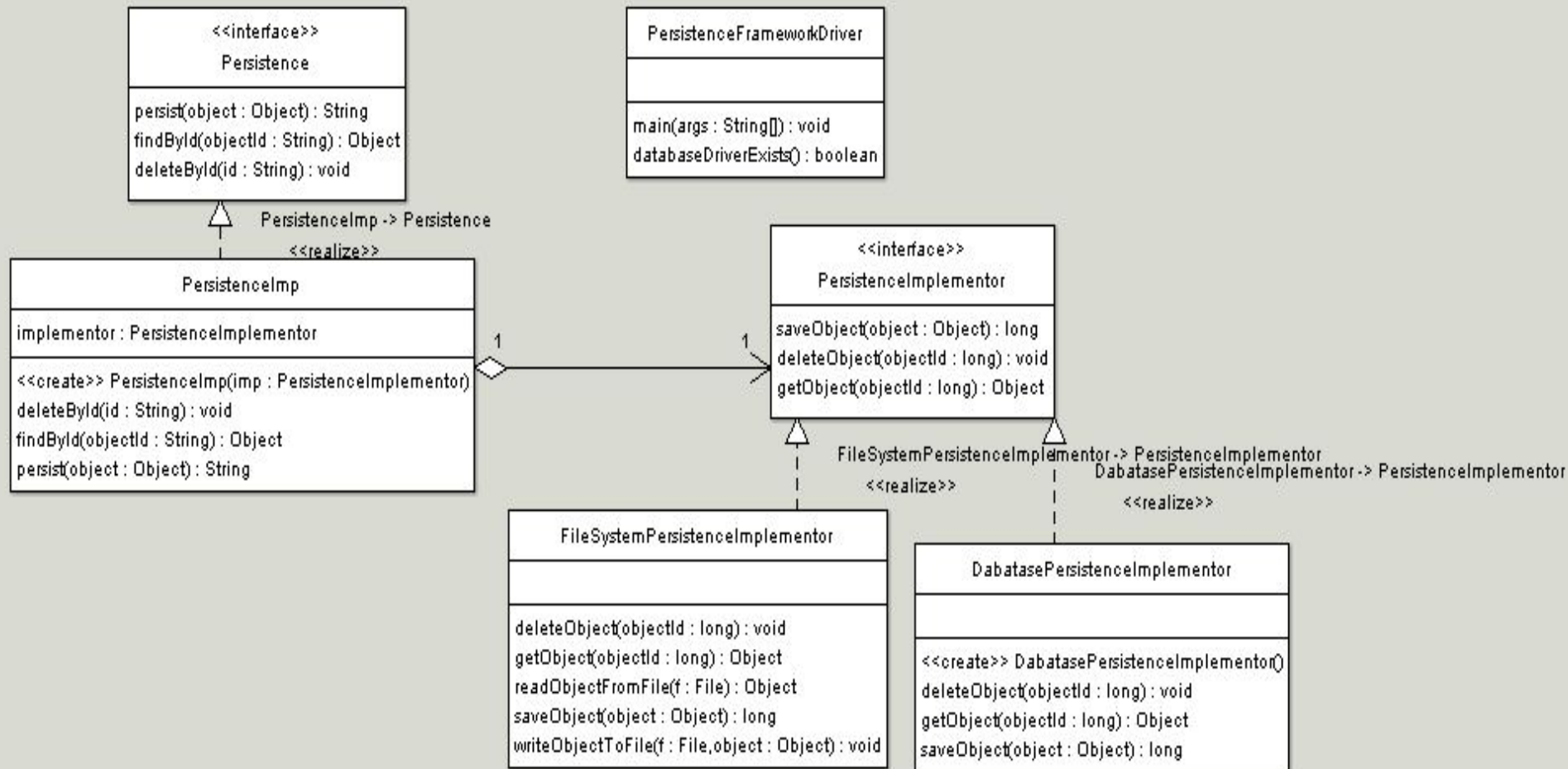
Bridge: Implementazione



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 Stream
    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 implementor , depending on existence of database drivers

        PersistenceImplementor implementor = null;

        if(databaseDriverExists()){
            implementor = new DatabasePersistenceImplementor();
        }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; }
}
```

		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	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

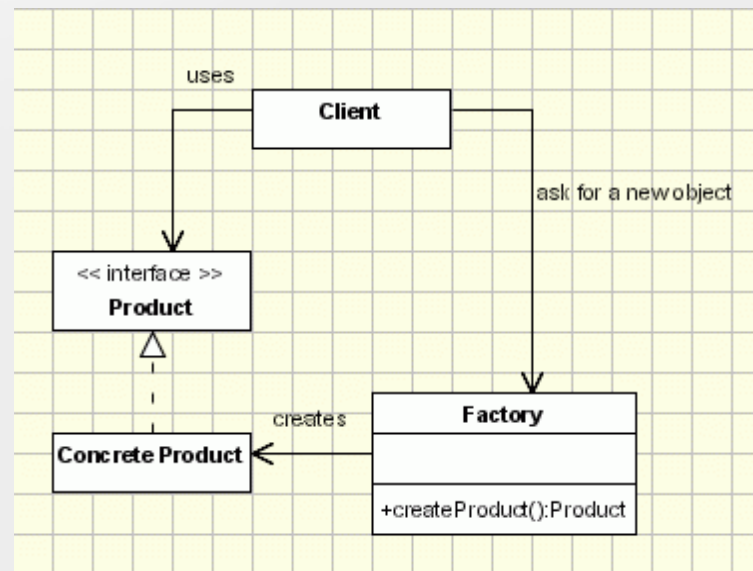
Factory

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.

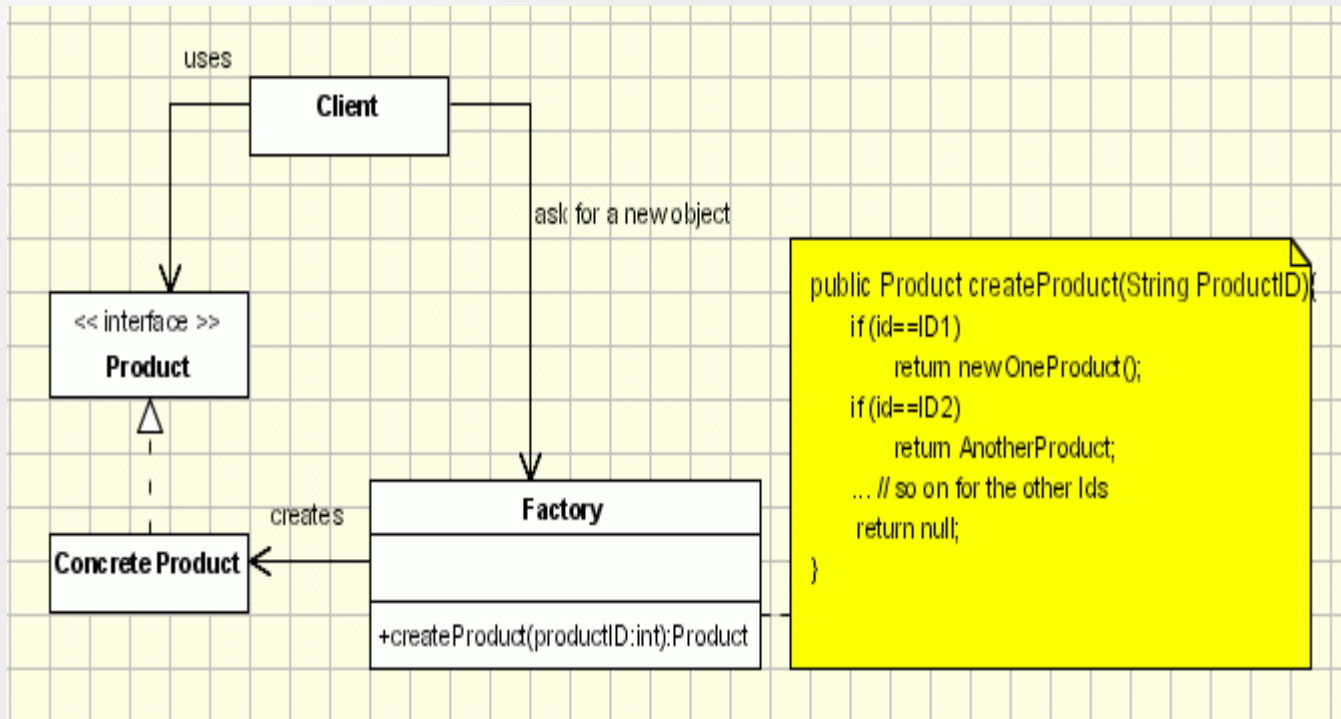
Factory Method (Creational)

- 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();
```

```
    }
```

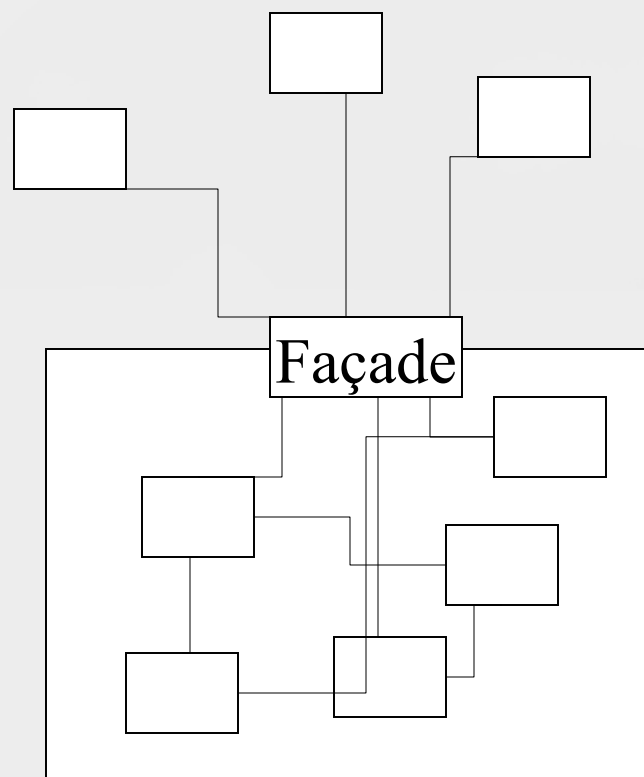
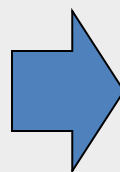
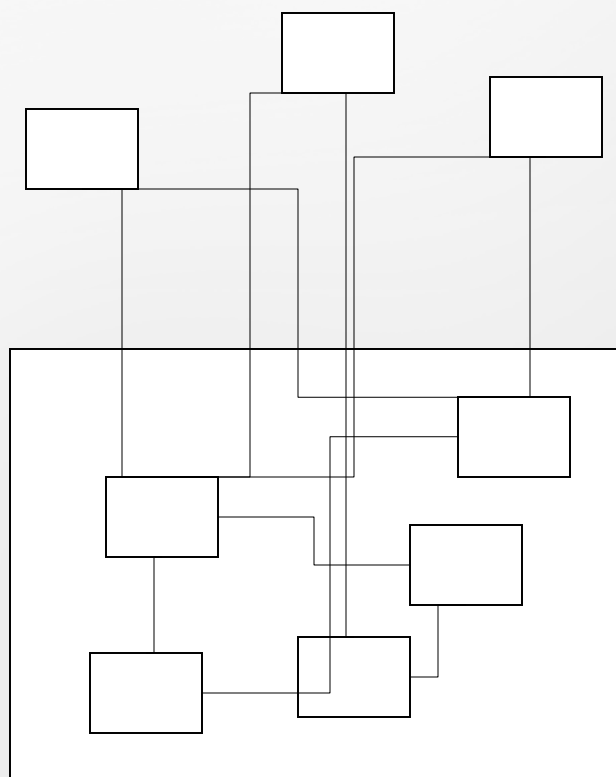
```
    }
```

```
}
```

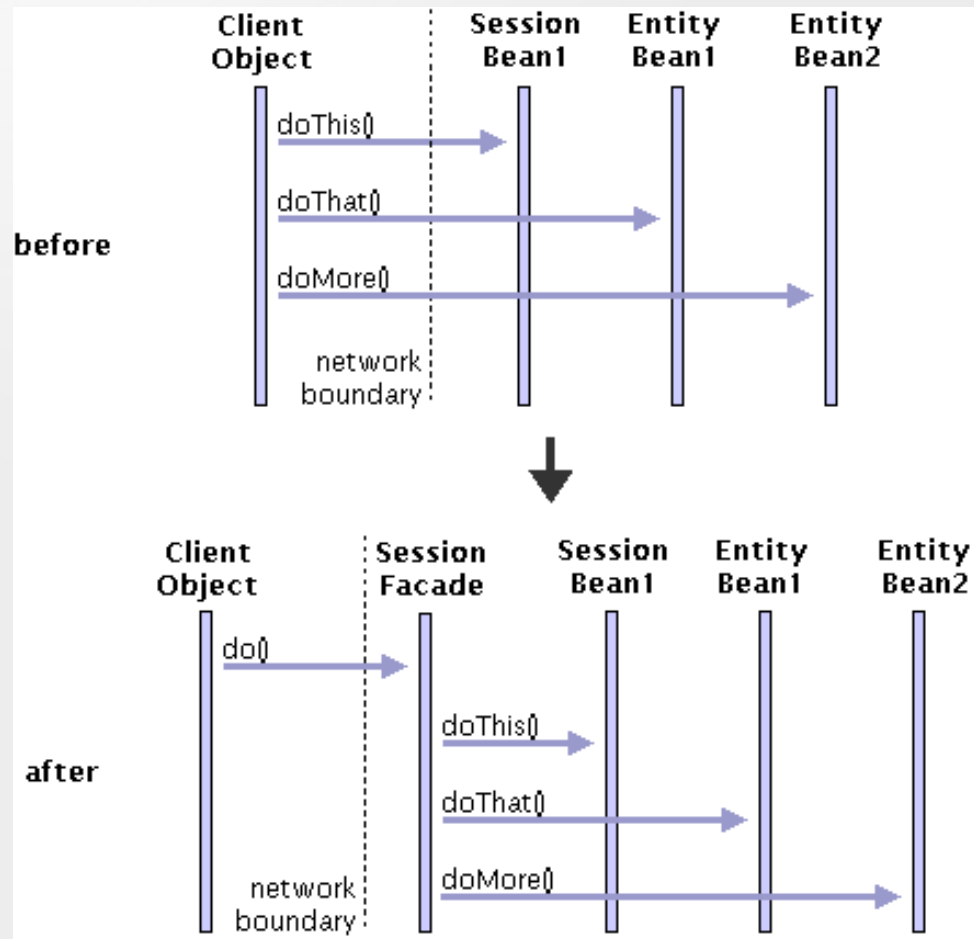
		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	Class	Factory Method	Adapter (class)	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Flyweight Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento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

Facade

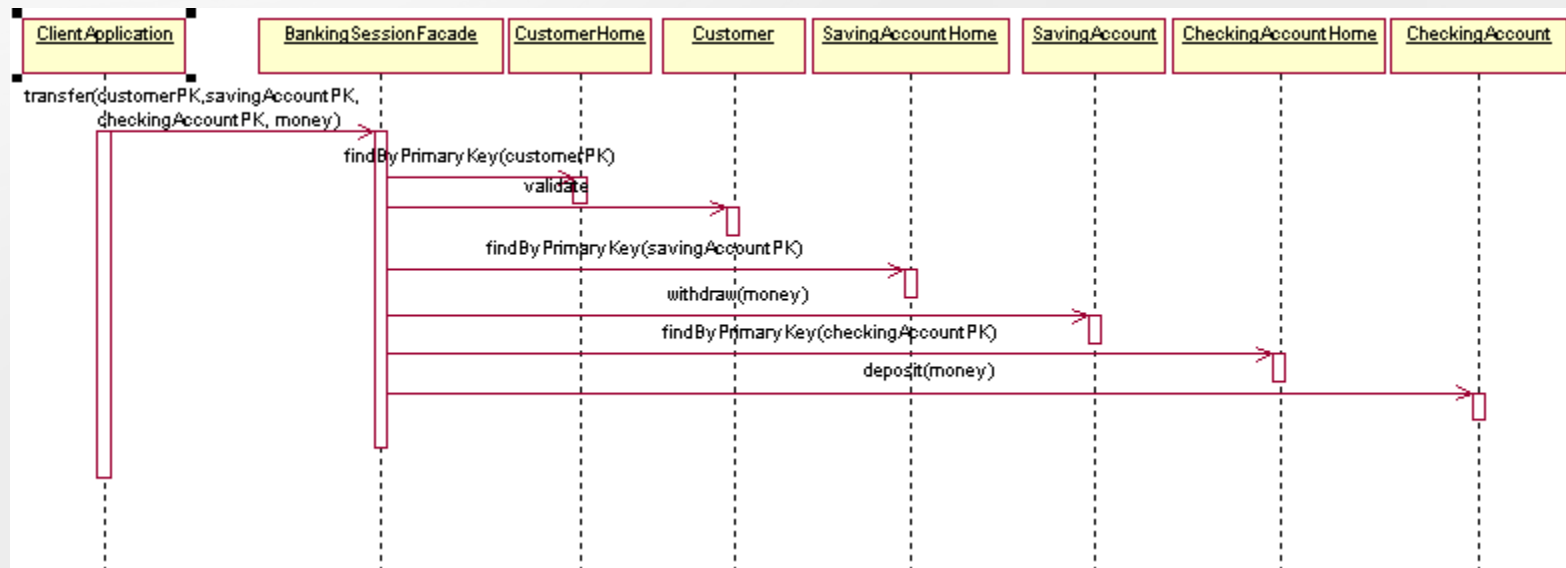
Façade



Façade



Façade - Example



		<i>Purpose</i>		
		Creational	Structural	Behavioral
<i>Scope</i>	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

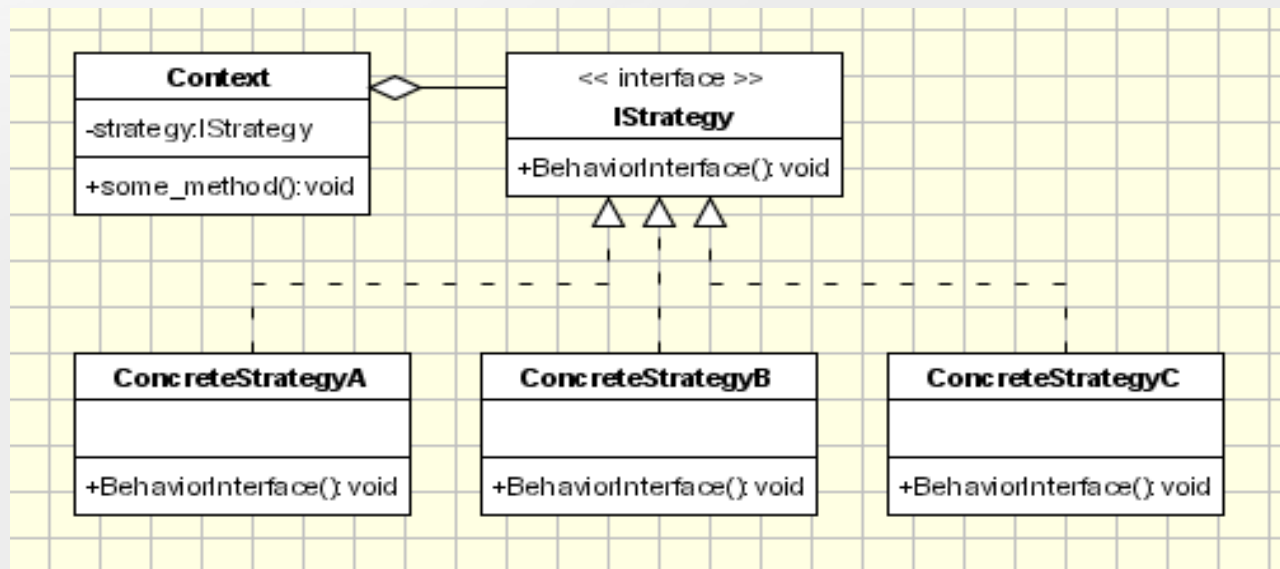
Strategy

Strategy Pattern

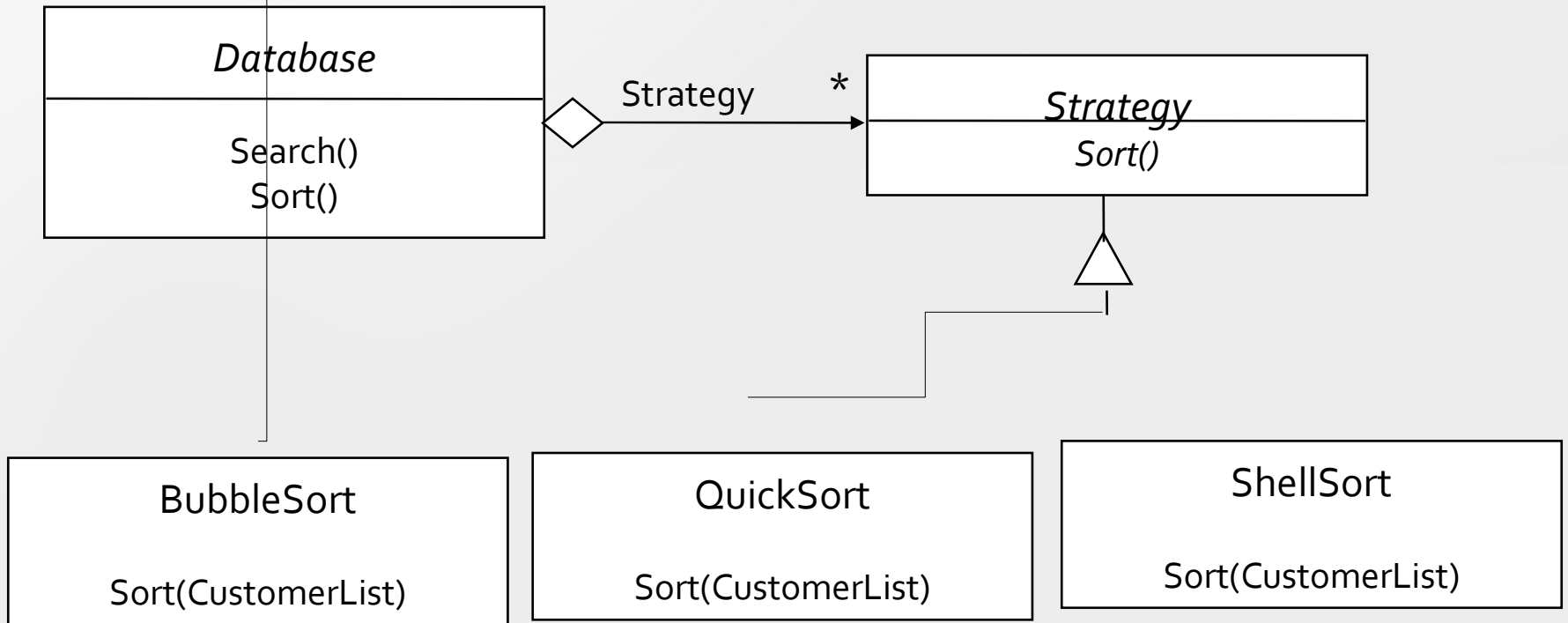
- Sono disponibili molti algoritmi diversi per risolvere lo stesso problema
- Esempi:
 - Frazione uno stream di testo in linee
 - Parsing un insieme di token in un albero sintattico astratto
 - Ordinamento di una lista di entità
- Diversi algoritmi potrebbero essere adatti in circostanze diverse
 - Rapid prototyping vs consegna del prodotto finito
- Non si vogliono supportare tutti 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



Applicazione dello Strategy Pattern in una applicazione Database



Applicabilità dello Strategy Pattern

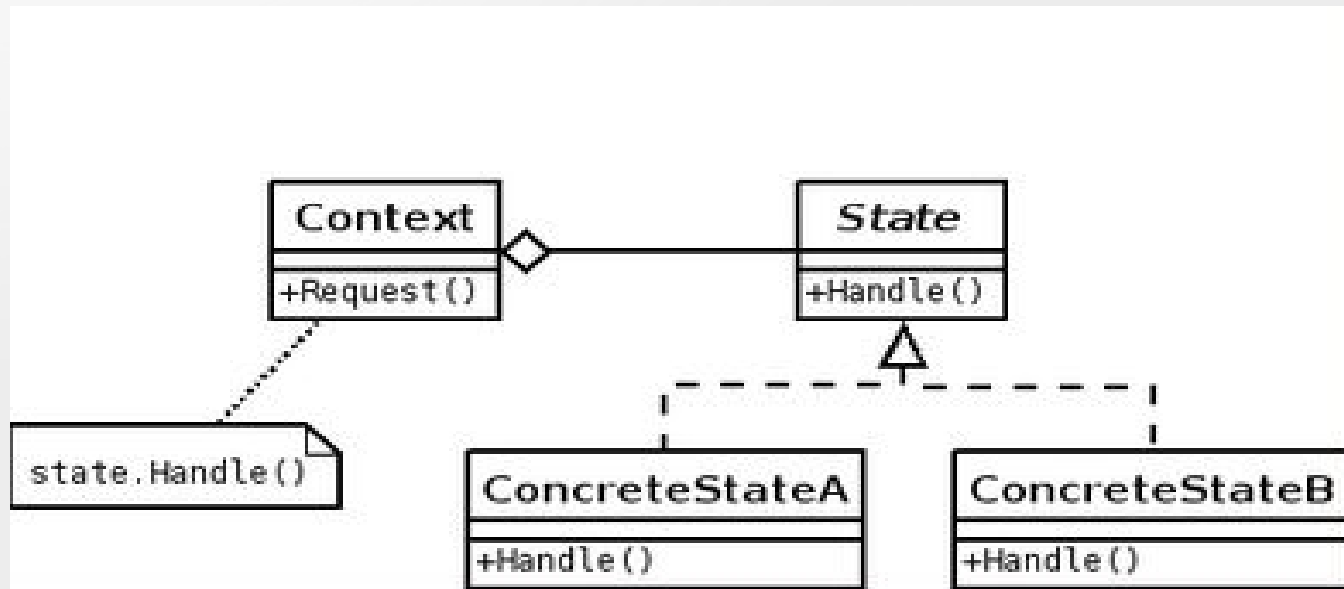
- Molte classi differiscono solo rispetto al loro comportamento. Lo Strategy permette di configurare una singola classe con uno dei molti possibili comportamenti.
- Molte varianti dello stesso algoritmo possono essere usate e scelte run-time in funzione delle necessità.
 - Le varianti dell'algoritmo 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 pattern 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;  
    }  
}
```

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