

Scuola Politecnica e delle Scienze di Base

Corso di *Laurea Magistrale* in
Ingegneria Informatica



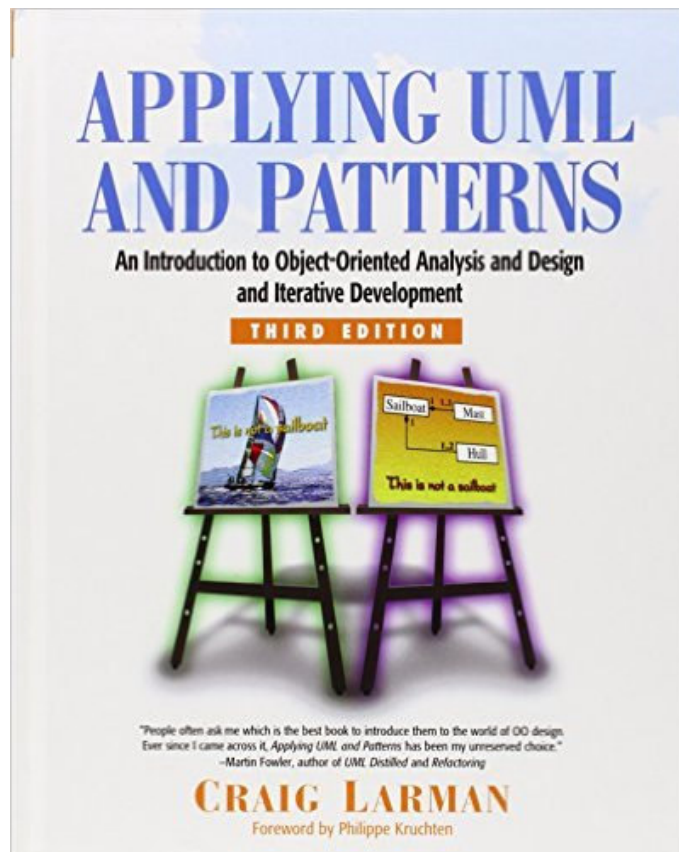
Progettazione guidata dai Patterns: Pattern GRASP e Patterns GoF

Outline

- **GRASP** (General Responsibility Assignment Software **Patterns**).
- **GoF** (Gang of Four).

GRASP

- Riferimenti



GRASP: progettazione di oggetti con responsabilità

- In ogni iterazione di Elaborazione vengono svolte anche attività di progettazione e codifica
- La progettazione ad oggetti verrà svolta a partire dagli elaborati disponibili (Casi d'uso di dettaglio, SSD, Contratti delle operazioni) e richiederà ulteriore modellazione (diagrammi delle classi, di interazione, package diagrams, UI)
- La progettazione guidata dalle responsabilità (RDD) ci guiderà in questa fase di modellazione

Responsibility Driven Design

- RDD è un modo di progettare ad oggetti facendosi guidare dalle responsabilità.
- Gli oggetti possono avere responsabilità di *fare*:
 - Es. fare esso stesso qualcosa, oppure chiedere ad altri oggetti di fare qualcosa, controllare attività di altri oggetti
- oppure Responsabilità di *conoscere*:
 - Conoscere i propri dati incapsulati, oggetti correlati, le cose che può derivare o calcolare
- Es. una Sale è responsabile di creare oggetti SaleLineItem, oppure di conoscere il suo totale

Pattern GRASP

- Sono 9 patterns che definiscono linee guida per l'assegnazione di responsabilità a collaborating objects.
 - Creator
 - Information Expert
 - Low Coupling
 - Controller
 - High Cohesion
 - Indirection
 - Polymorphism
 - Protected Variations
 - Pure Fabrication

Le responsabilità

- La responsabilità può essere:
 - realizzata da un unico oggetto,
 - realizzata da un gruppo di oggetti che operano in collaborazione.
- GRASP ci aiuta a decidere quale responsabilità deve essere assegnata a quale oggetto/classe.
 - Identifica gli oggetti e le responsabilità dal dominio del problema e identifica anche come gli oggetti interagiscono tra loro.
 - Definisce un progetto per questi oggetti
 - Identifica le classi che implementano queste responsabilità

Creator

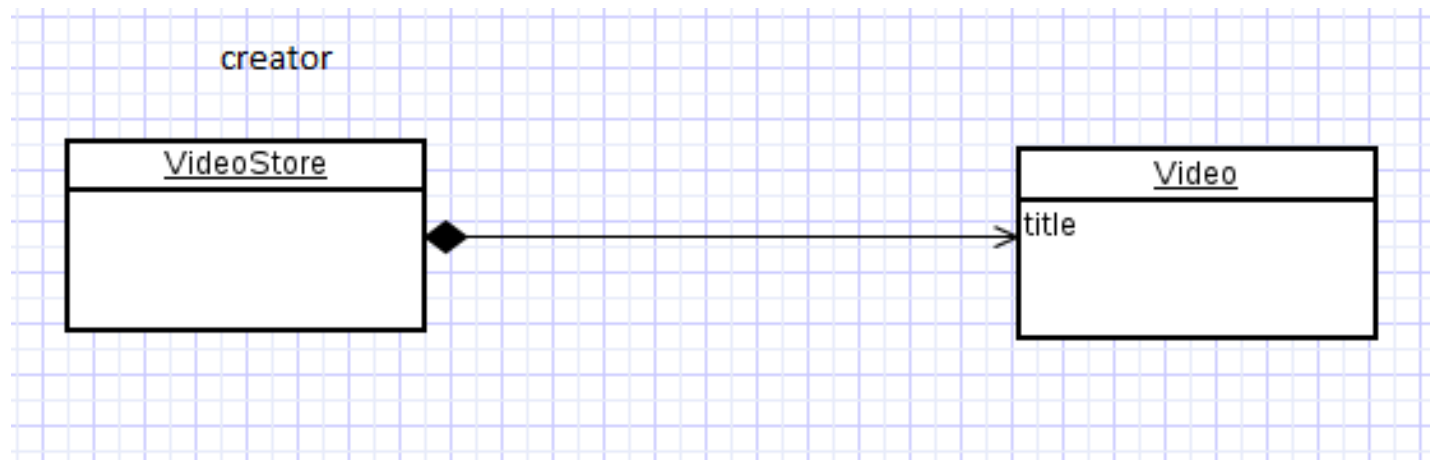


- **Problema:** Chi crea un oggetto? Chi deve creare una nuova istanza di una classe?
- **Soluzione:** assegna alla class B la responsabilità di creare oggetti di A se almeno una delle seguenti condizioni è vera:
 - B contiene o aggrega una composizione di oggetti di tipo A
 - B registra A
 - B utilizza strettamente/ esclusivamente A
 - B possiede i dati per l'inizializzazione di A

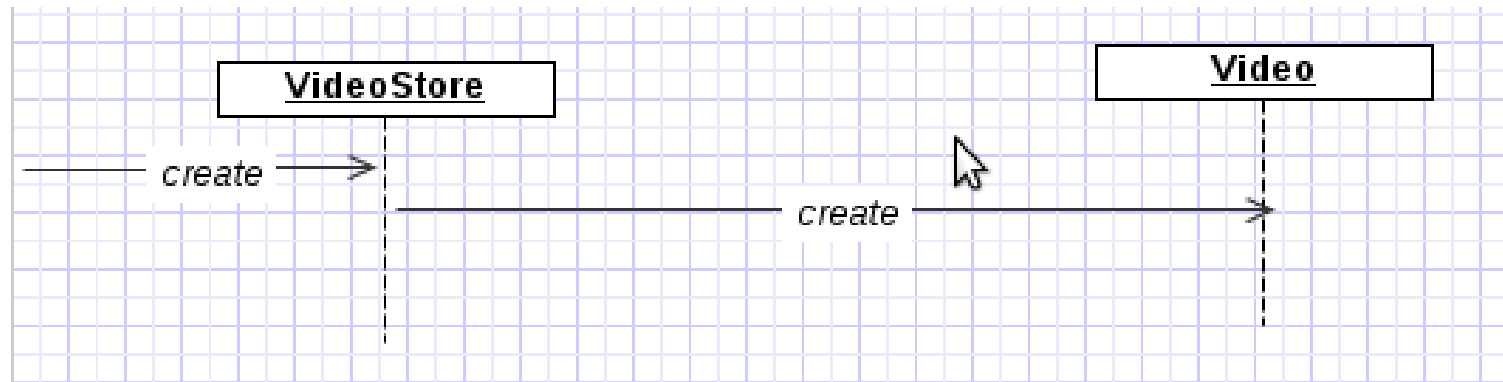
Esempio di Creator

- Si consideri un VideoStore e i Video che esso contiene
- VideoStore ha una associazione di aggregazione con Video.
 - VideoStore è il Container
 - Video è il Contained
- Dobbiamo instanziare oggetti Video nella VideoStore class

UML Class Diagram



UML Sequence Diagram



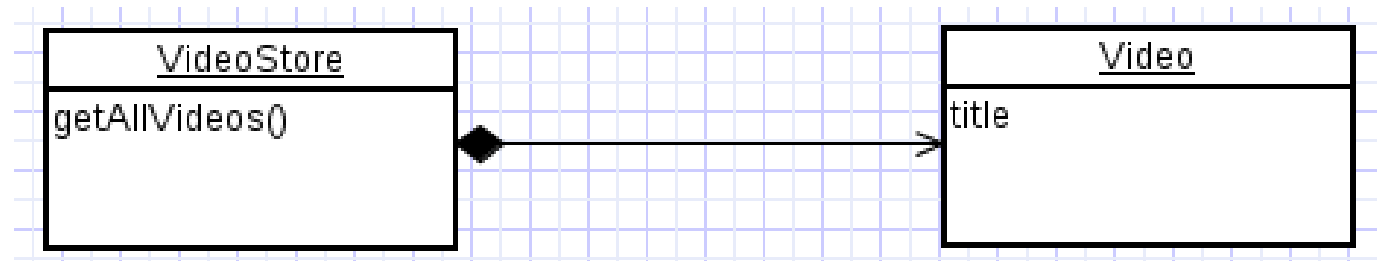
Information Expert

- Dato un oggetto **o** quali responsabilità possono essere assegnate ad esso?
- Il pattern ci consiglia di assegnare ad **o** le responsabilità per cui **o** possiede le informazioni necessarie a soddisfarle.
- Gli oggetti devono avere le informazioni necessarie per soddisfare le operazioni, oppure, in alcuni casi essi possono collaborare con altri oggetti per soddisfare tali responsabilità.

Esempio

- Dobbiamo prelevare tutti i video del VideoStore.
- Poiché VideoStore conosce tutti i video, questa responsabilità può essere assegnata alla classe VideoStore.
- VideoStore è **l'information expert**.

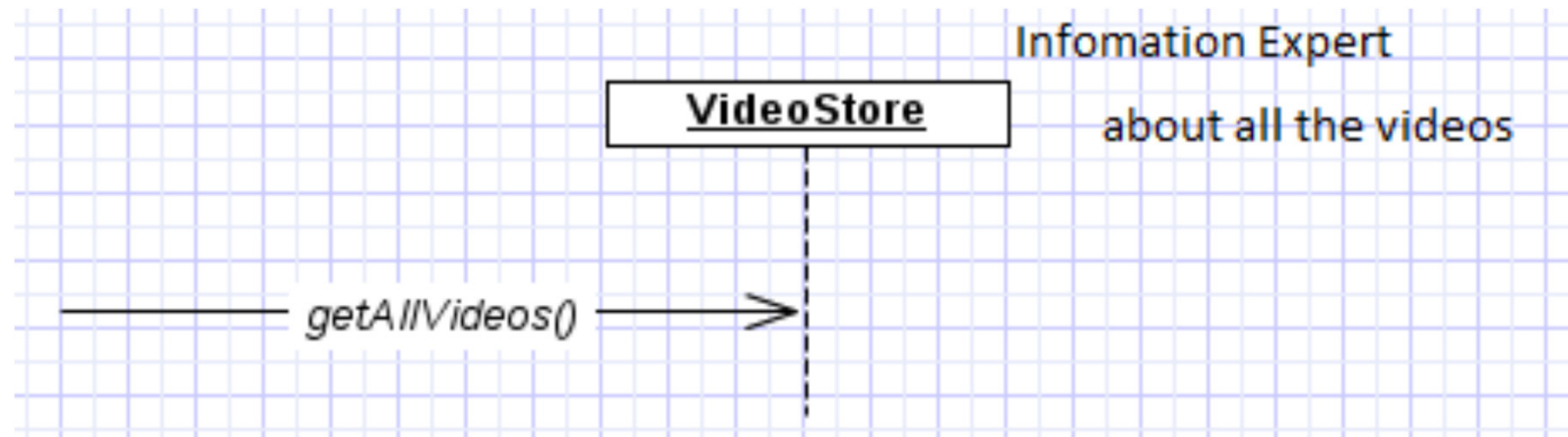
UML Class Diagram



Analogamente VideoStore potrà avere la responsabilità di ricercare un particolare Video, a partire dal suo Titolo.

Altri esempi: Biblioteca, aggregato di Libri, avrà la responsabilità di elencare tutti i libri o di Cercare un dato Libro

UML Sequence Diagram



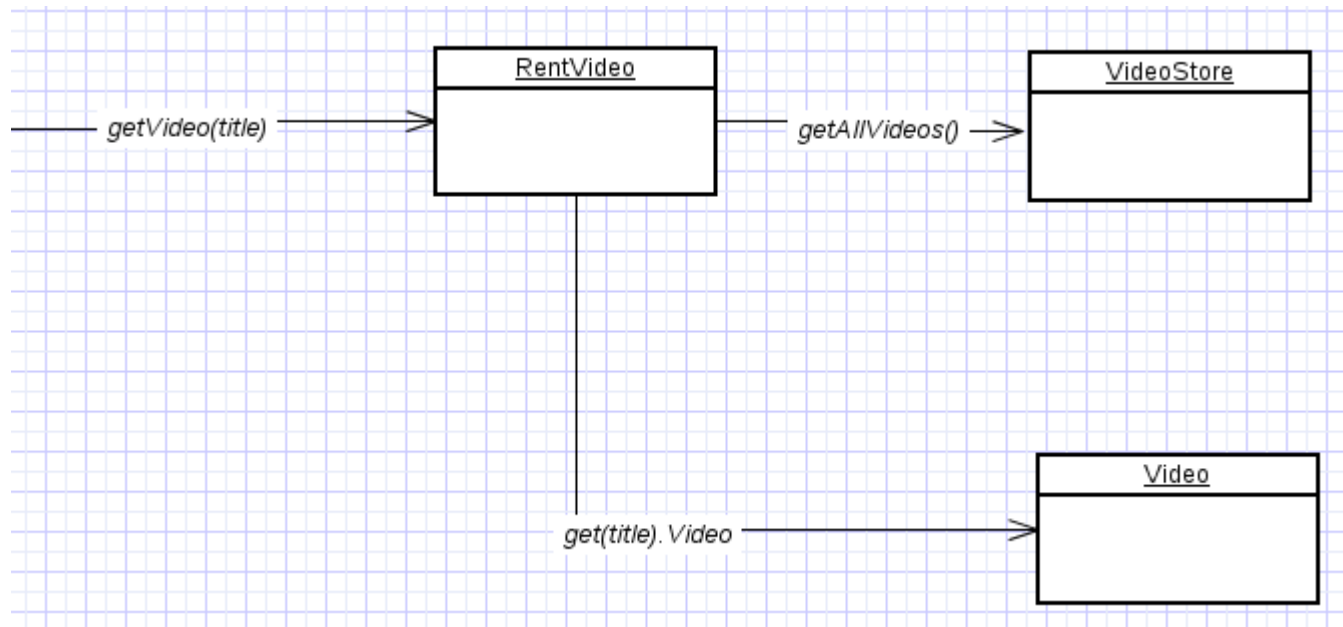
Low Coupling

- **Problema:** Come ridurre l'impatto dei cambiamenti in un sistema?
- **Coupling:** Se un oggetto B dipende da A, ogni modifica ad A si ripercuote su B
- Bisogna evitare di creare troppe dipendenze, per ridurre l'impatto delle variazioni degli oggetti indipendenti sugli oggetti dipendenti.
- **Soluzione:** assegna le responsabilità in modo che l'accoppiamento (non necessario) rimanga basso.

Low coupling

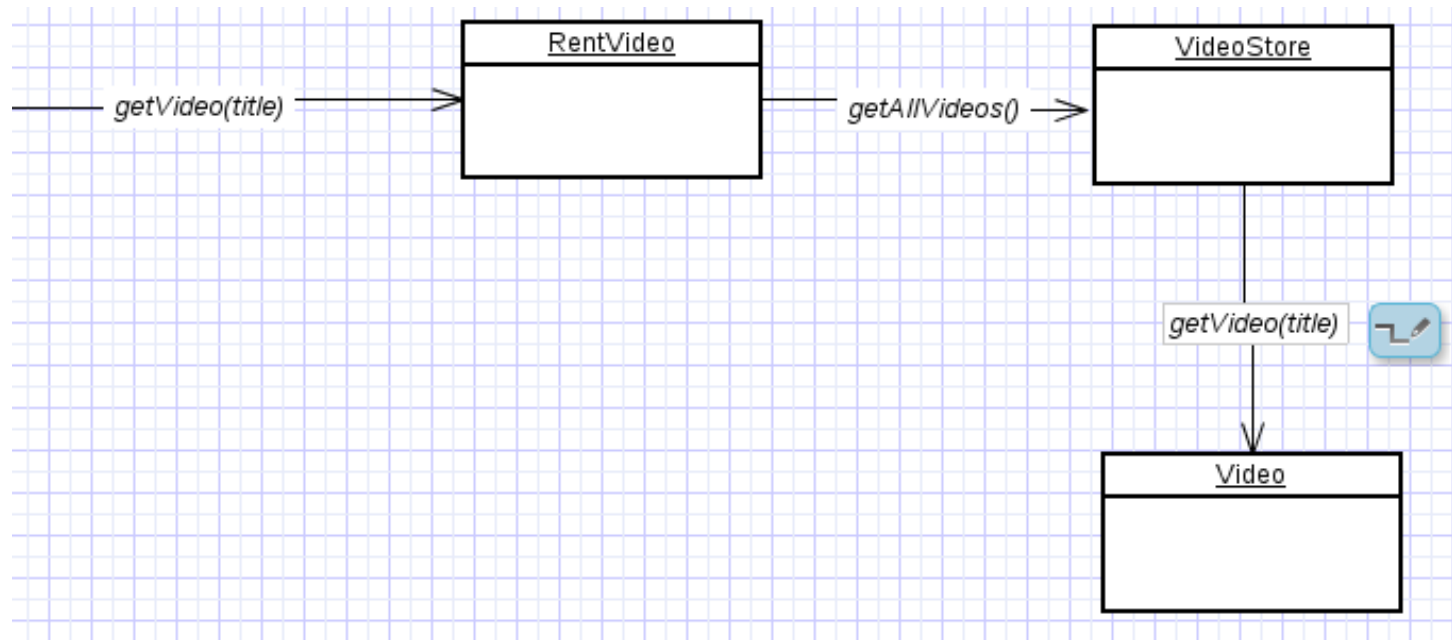
- Due elementi sono accoppiati, coupled, se:
 - Un elemento ha una aggregazione/composizione/associazione con l'altro elemento.
 - Un elemento implementa/estende l'altro elemento.
 - Es. in una gerarchia di ereditarietà, una classe figlia dipende (è accoppiata) dalla classe padre
 - Un elemento usa/ richiama le operazioni dell'altro

Esempio di **poor (cattivo)** coupling



- L'operazione `getVideo(title)` deve ottenere l'oggetto `Video` avente titolo= title
- La classe `RentVideo` ha conoscenza sia del `VideoStore` sia della classe `Video` → `RentVideo` è dipendente da entrambe le classi.

Esempio di low coupling





- VideoStore e Video sono accoppiate. RentVideo e VideoStore sono accoppiate. → è garantito il low coupling
- Il pattern Information Expert sostiene il Low Coupling

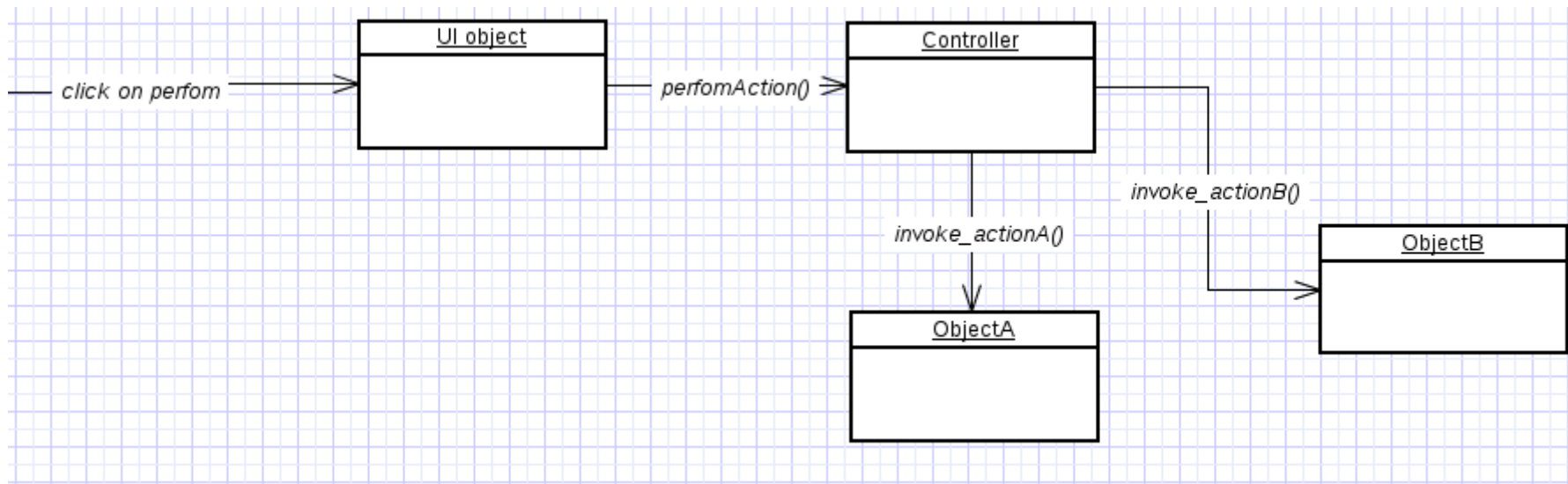
Controller

- Le architetture a strati prevedono separazione fra UI e logica applicativa: gli oggetti della UI non devono possedere logica applicativa, ma quando arriva una richiesta sulla UI, essi devono delegare un oggetto del dominio
- Quando una richiesta arriva da un oggetto del livello UI, il Controller pattern ci aiuta a determinare quale sia il primo oggetto che riceverà tale messaggio.
 - Questo oggetto è definito **controller object**.
 - Riceve una richiesta dal livello UI e poi controlla/coordina altri oggetti del livello di dominio per soddisfare la richiesta.
 - Delega il lavoro ad altre classi e coordina l'esecuzione dell'intera attività.

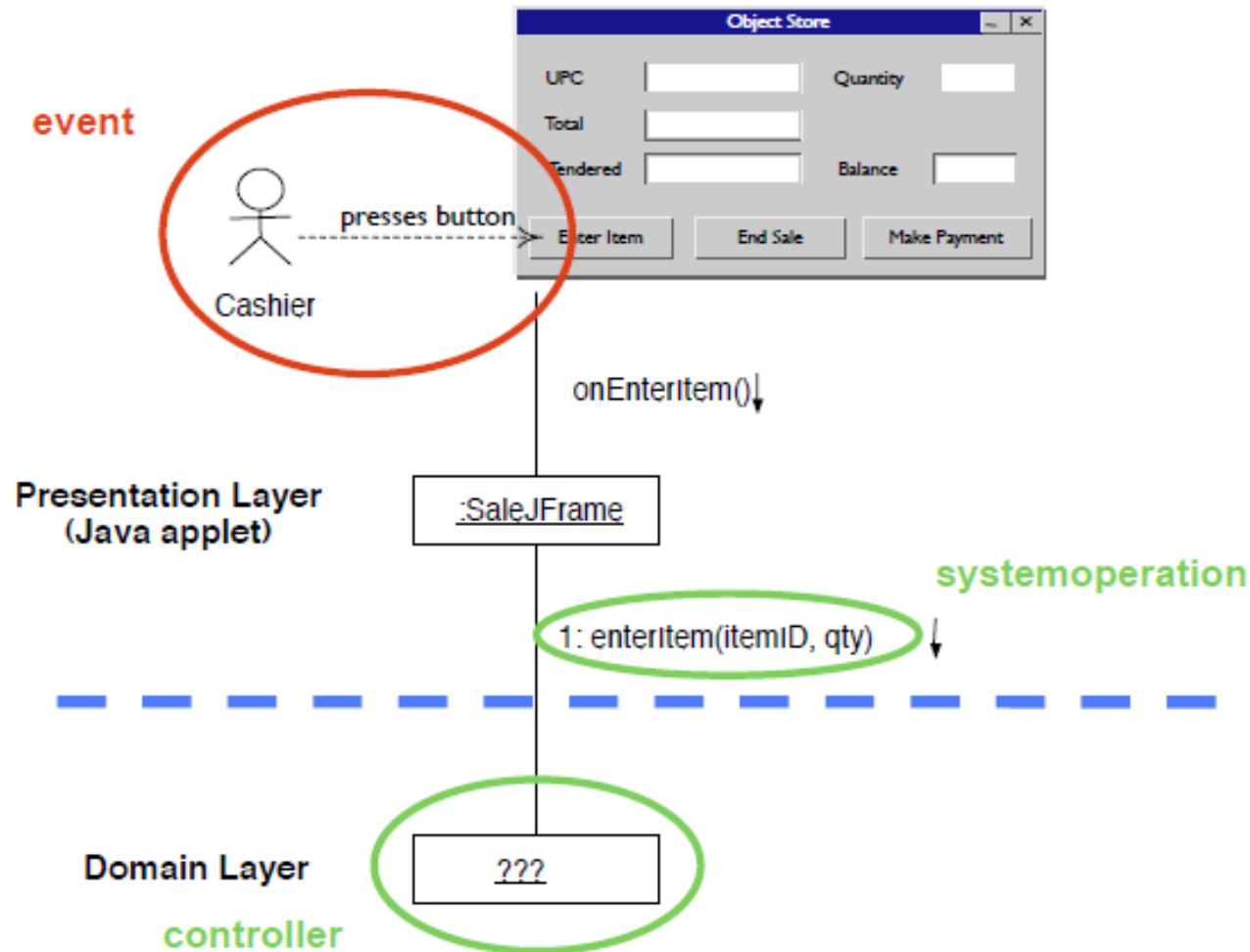
Controller

- L'oggetto Controller riceve e controlla la gestione di una richiesta dalla UI.
- Due possibili opzioni per il Controller:
 - Può essere un oggetto che rappresenta l'intero sistema (**facade controller**) 
 - Es. a root object, il device sul quale il software è in esecuzione, o un subsystem.
 - Può essere un oggetto che rappresenta un caso d'uso, gestisce una sequenza di operazioni (**controller di caso d'uso o session controller**) 
 - artificial object (vedi Pure Fabrication pattern) che gestisce tutti gli eventi di un caso d'uso o una sessione.

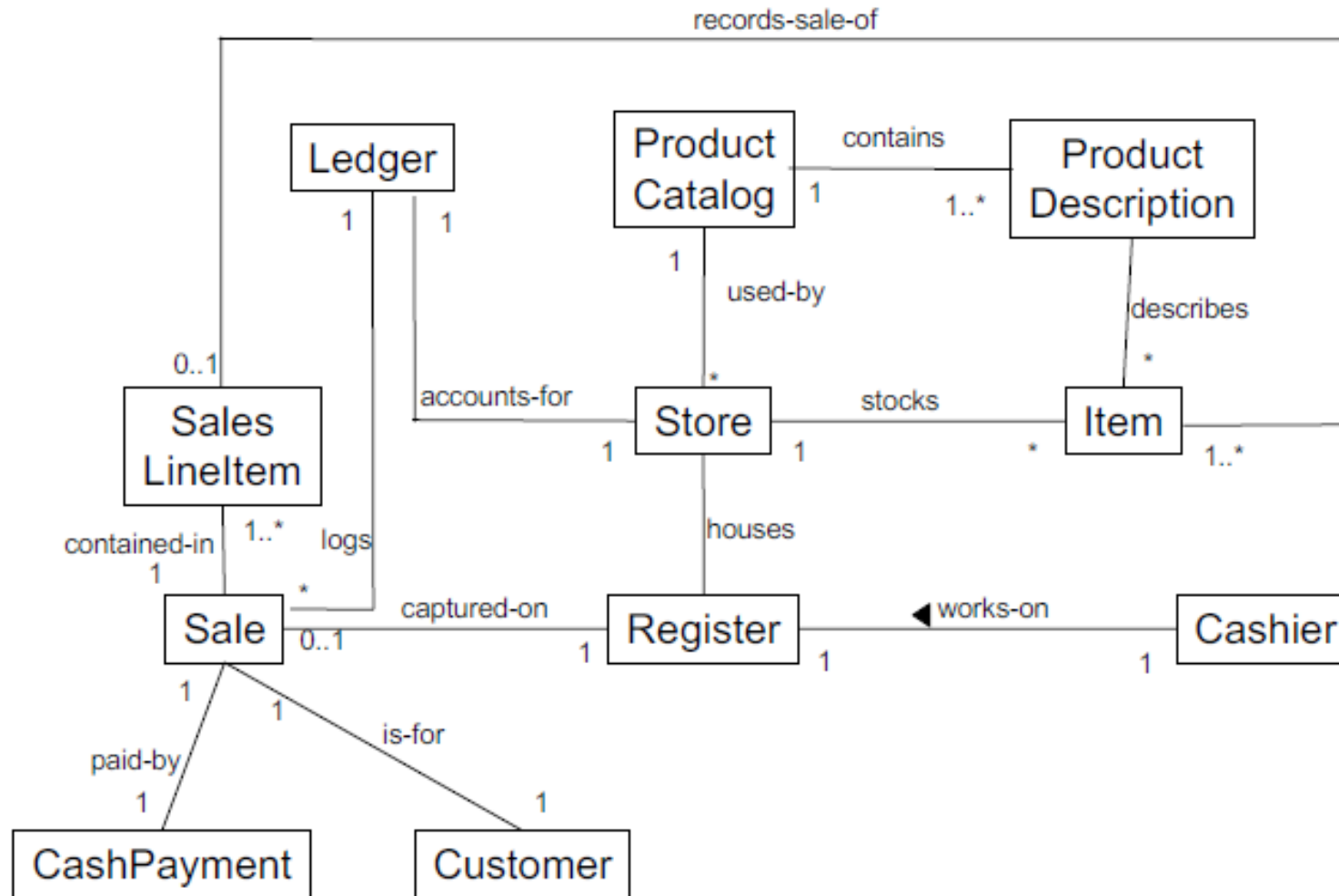
Esempio



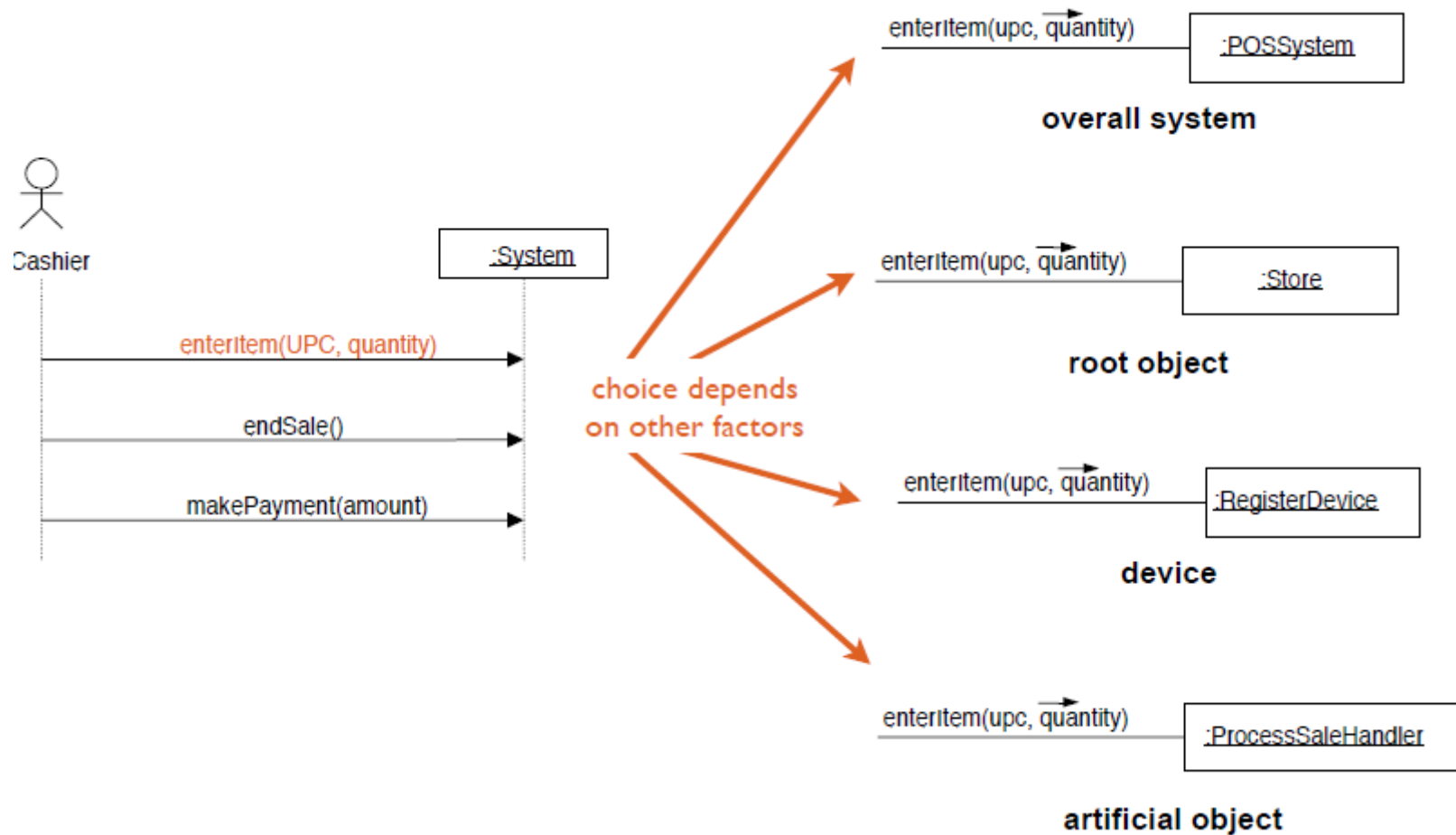
Esempio





Dato il Dominio: Nextgen POS (da Larman)



Chi processa il System event?



Confronto fra Controller

- La soluzione col Facade Controller prevede un solo controller per tutte le operazioni richieste al dominio 
 - Va bene se non sono richieste troppe operazioni
- La soluzione con Controller di Caso d'uso prevede un Controller diverso per ogni caso d'uso 
 - In questo caso il controller è artificiale , non corrisponde a nessun elemento del dominio



Bloated Controller

- Una classe Controller è definita bloated (gonfio), se:
 - La classe è sovraccaricata con troppe responsabilità.
 - Soluzione: aggiungere più controller.
 - La classe esegue molto lavoro per rispondere alla richiesta dalla UI, senza delegare ad altre classi.
 - Soluzione: la classe deve delegare compiti ad altre classi.
 - La classe ha molti attributi e conserva informazioni che andrebbero distribuite ad altri oggetti, oppure duplica informazioni trovate altrove

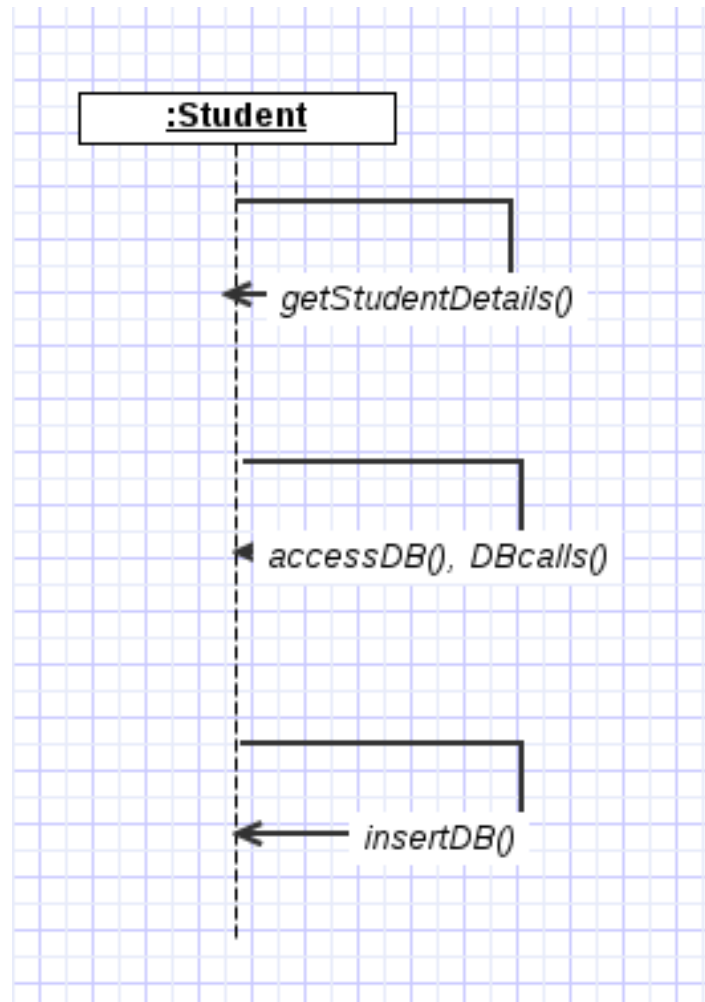
High Cohesion

- Problema: come mantenere gli oggetti focalizzati, comprensibili e gestibili, e sostenere il Low Coupling?
- La coesione è una misura di quanto fortemente siano correlate le responsabilità di un elemento
- Una classe con bassa coesione fa molte cose non correlate fra loro ed è:
 - Difficile da comprendere, da modificare, da riusare
 - Crea molti accoppiamenti con altre classi

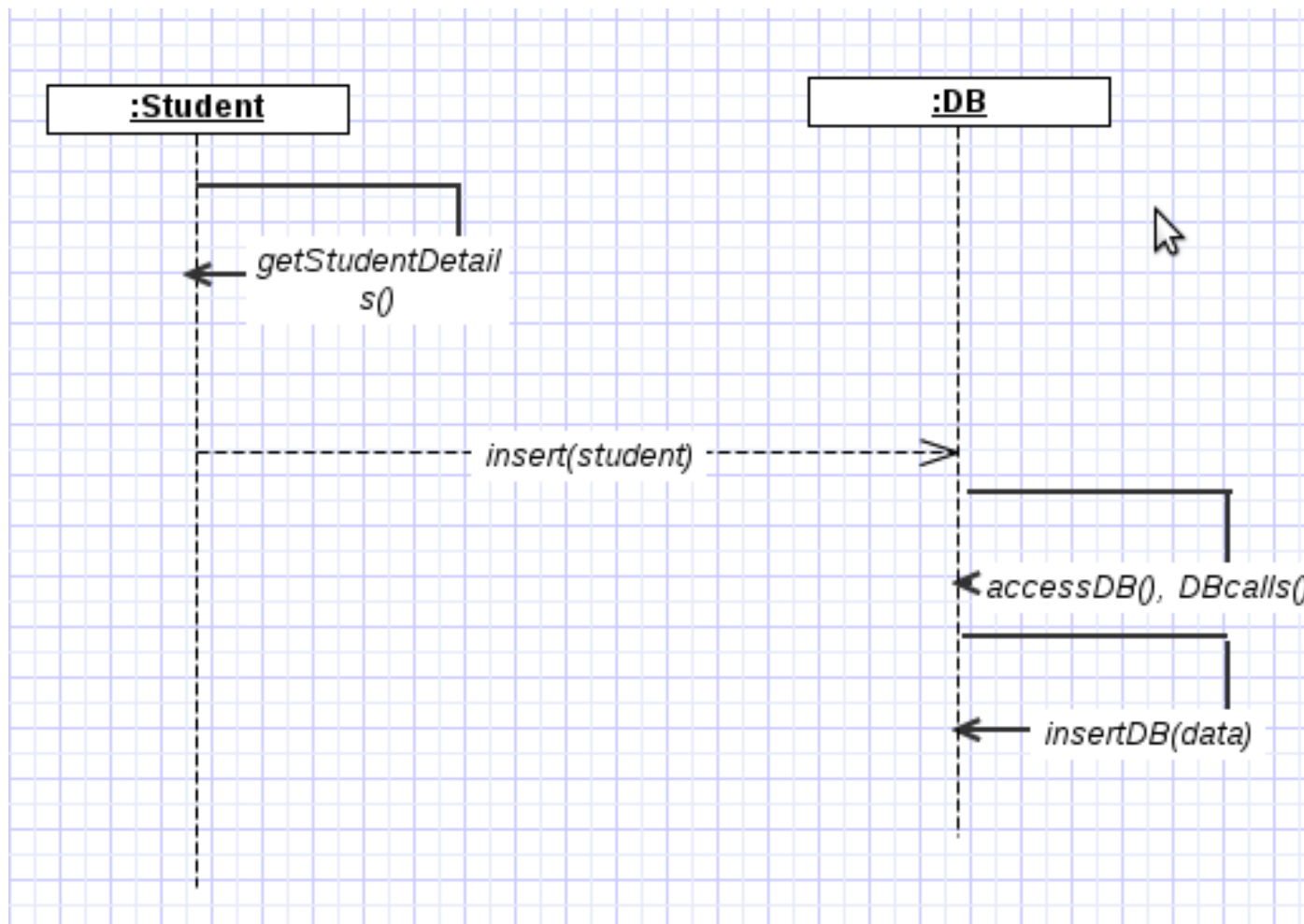
High Cohesion

- Consiglia di delegare responsabilità correlate ad una singola unità.
- Permette di definire chiaramente lo scopo dei diversi elementi.
- Benefici:
 - Facilmente comprensibile e manutenibile.
 - Code Reuse
 - Low Coupling


Esempio di Low Cohesion



Esempio di High Cohesion



Diversi tipi di Coesione

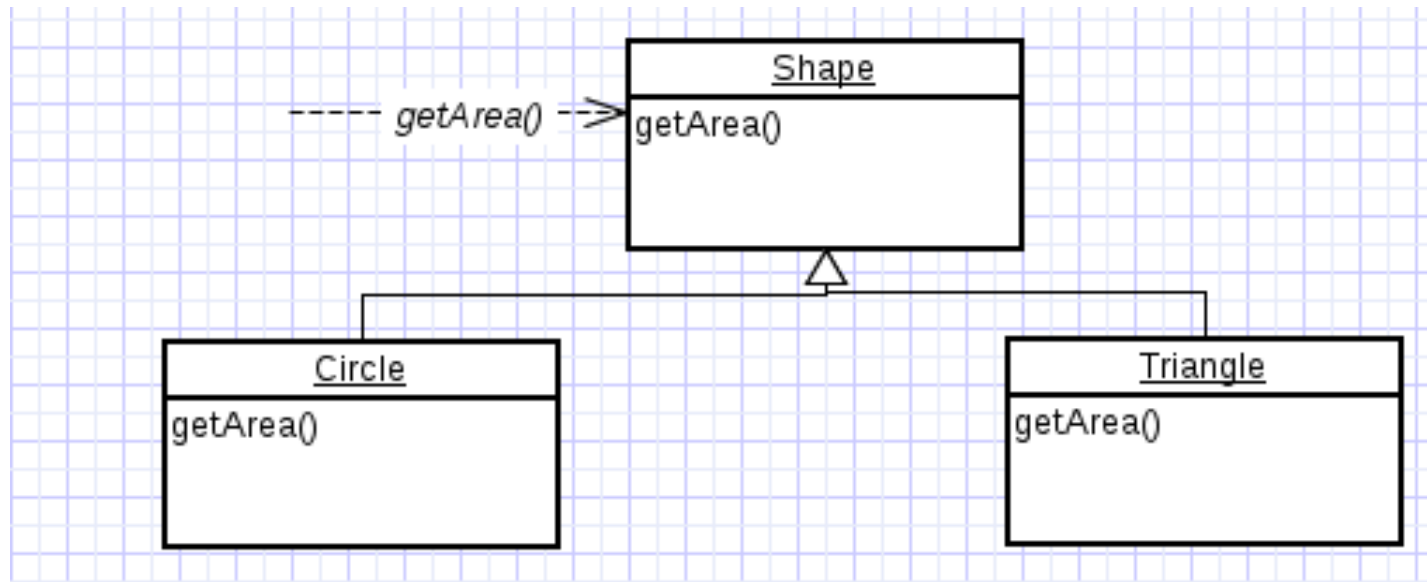
- Coesione dei dati: una classe implementa un tipo di dati (buona coesione) 
- Coesione funzionale: gli elementi di una classe svolgono una sola funzione (buona o molto buona)
- Coesione temporale: gli elementi sono raggruppati perché usati circa nello stesso momento (è la coesione degli oggetti controller-buona in questo caso, ma non sempre)
- Coesione casuale: es. raggruppa tutti i metodi avente il nome che inizia con “D”...(pessima)

Polymorphism

- Come gestire elementi correlati ma diversi in base al tipo di elemento?
- Il polimorfismo ci guida nel decidere quale oggetto è responsabile per gestire tali elementi variabili.
- Benefici: è semplice gestire nuove variazioni.

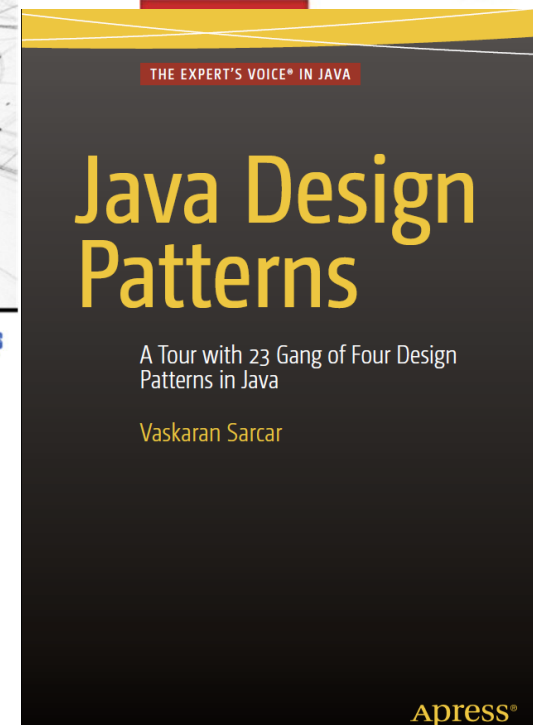
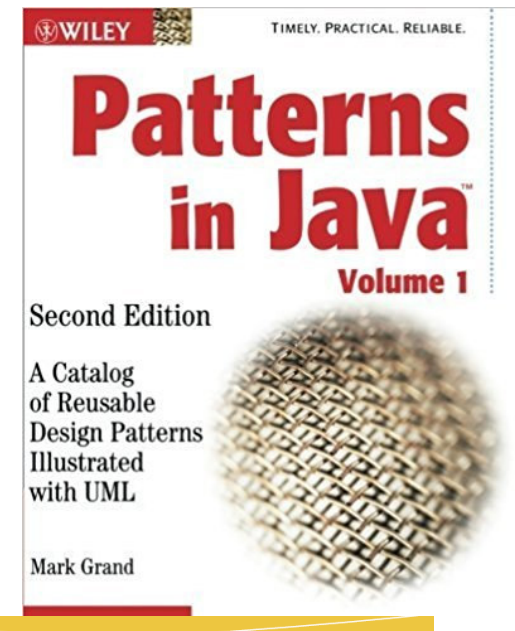
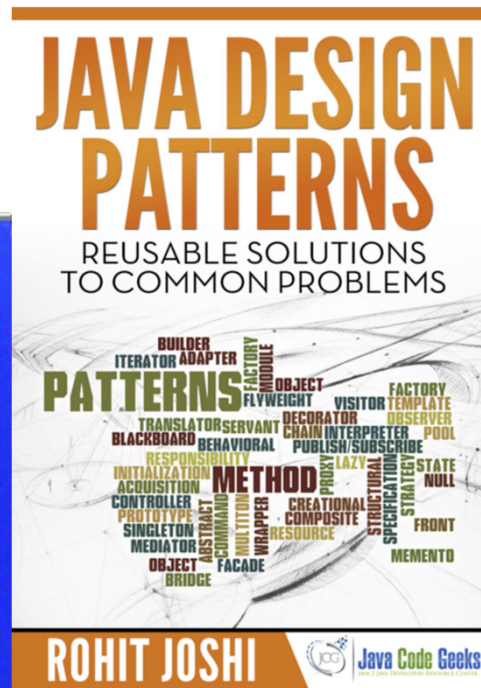
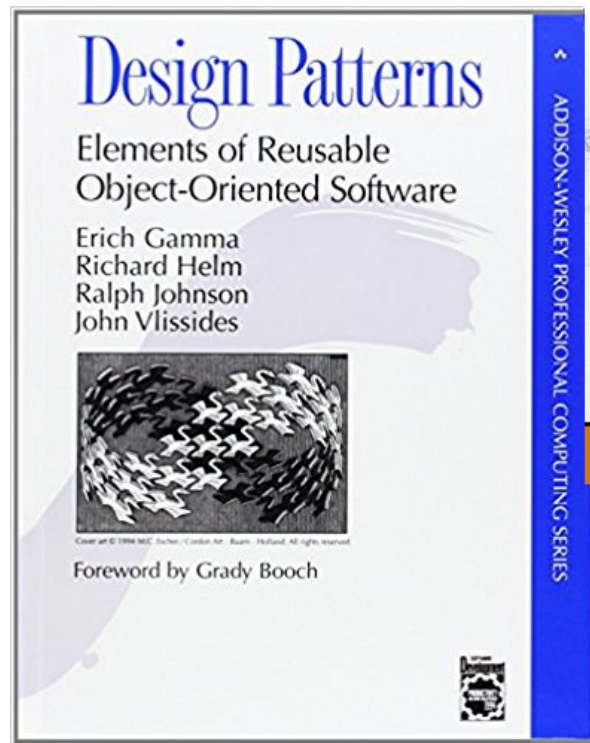
Esempio di Polimorfismo

- Il metodo `getArea()` varia col tipo di Shape.
 - Si assegna questa responsabilità alle sottoclassi.
- Inviando un messaggio all'oggetto Shape, una call effettiva verrà inviata al corrispondente sotto-oggetto (Circle oppure Triangle)



GoF

- Riferimenti



The Sacred Elements of the Faith

Creazionali
the holy
origins

Strutturali
the holy
structures

Comportamentali

the holy
behaviors

107 FM Factory Method								139 A Adapter
117 PT Prototype	127 S Singleton					223 CR Chain of Responsibility	163 CP Composite	175 D Decorator
87 AF Abstract Factory	325 TM Template Method	233 CD Command	273 MD Mediator	293 O Observer	243 IN Interpreter	207 PX Proxy	185 FA Façade	
97 BU Builder	315 SR Strategy	283 MM Memento	305 ST State	257 IT Iterator	331 V Visitor	195 FL Flyweight	151 BR Bridge	

Creational Pattern

- I Pattern creazionali astraggono il processo di istanziazione degli oggetti.
- Aiutano a rendere il sistema indipendente da come i loro oggetti sono creati, composti e rappresentati.
- Un class creational pattern utilizza l'ereditarietà per variare la classe che è istanziata mentre un object creational pattern delegherà l'istanziamento ad un altro oggetto.

Quando usare i pattern creazionali

- Diventano importanti in quanto i sistemi si evolvono per dipendere maggiormente dalla composizione degli oggetti che dalla loro ereditarietà
- L'accento si sposta dalla pura codifica di un insieme fisso di comportamenti verso la definizione di un insieme più piccolo di comportamenti fondamentali che possono essere composti per implementare quelli più complessi.
- Creare oggetti con comportamenti particolari richiede più di una semplice istanziazione di una classe.

Aspetti comuni ai DP creazionali

- Ci sono due temi ricorrenti in questi modelli:
 1. Tutti incapsulano la conoscenza di quali classi concrete il sistema utilizza.
 2. Nascondono come le istanze di tali classi sono create e messe insieme.


Elenco dei Pattern Creazionali

Nome	Descrizione
Abstract Factory	Fornisce un'interfaccia per creare famiglie di oggetti in relazione senza specificare le loro classi concrete
Factory Method	Definisce un'interfaccia per creare un oggetto ma lascia decidere alle sottoclassi quale classe istanziare
Singleton	Assicura che la classe abbia una sola istanza e fornisce un metodo di accesso globale a tale istanza
Builder	Separa l'algoritmo di costruzione di una struttura di oggetti dalla creazione dei singoli oggetti.
Prototype	Specifica il tipo di oggetto da creare usando un'istanza prototipo e crea nuovi oggetti copiando questo prototipo

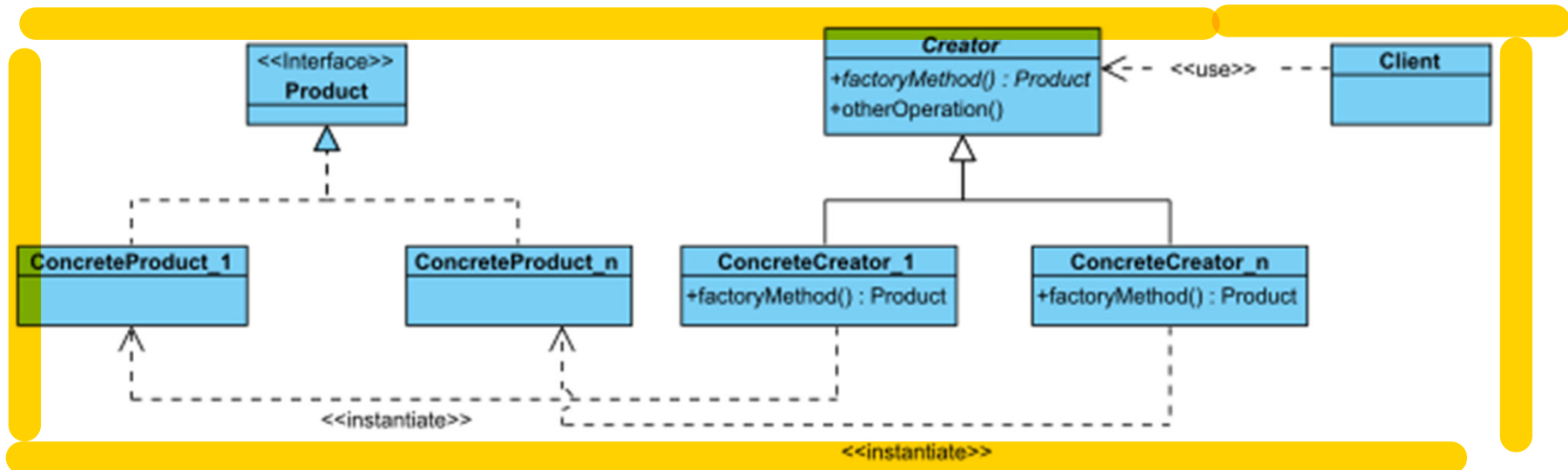
Factory Method Design Pattern

- Recently, a product company has shifted the way they used to take orders from their clients.
- The company is now looking to use an application to take orders from them.
- They receive orders, errors in orders, feedback for the previous order, and responses to the order in an XML format.
- The company has asked you to develop an application to parse the XML and display the results.
- **The main challenge for you is to parse an XML and display its content to the user.**
- **There are different XML formats depending on the different types of messages the company receives from its clients.** Like, for example, an order type XML has different sets of xml tags as compared to the response or error XML. But **the core job is the same**; that is, **to display to the user the message being carried in these XMLs.**
- **Although the core job is the same, the object that would be used varies according to the kind of XML the application gets from the user.**
- So, an application object may only know that it needs to access a class from within the class hierarchy (hierarchy of different parsers), but does not know exactly which class from among the set of subclasses of the parent class is to be selected.
- **In this case, it is better to provide a factory, i.e. a factory to create parsers, and at runtime a parser gets instantiated to do the job, according to the kind of XML the application receives from the user.**

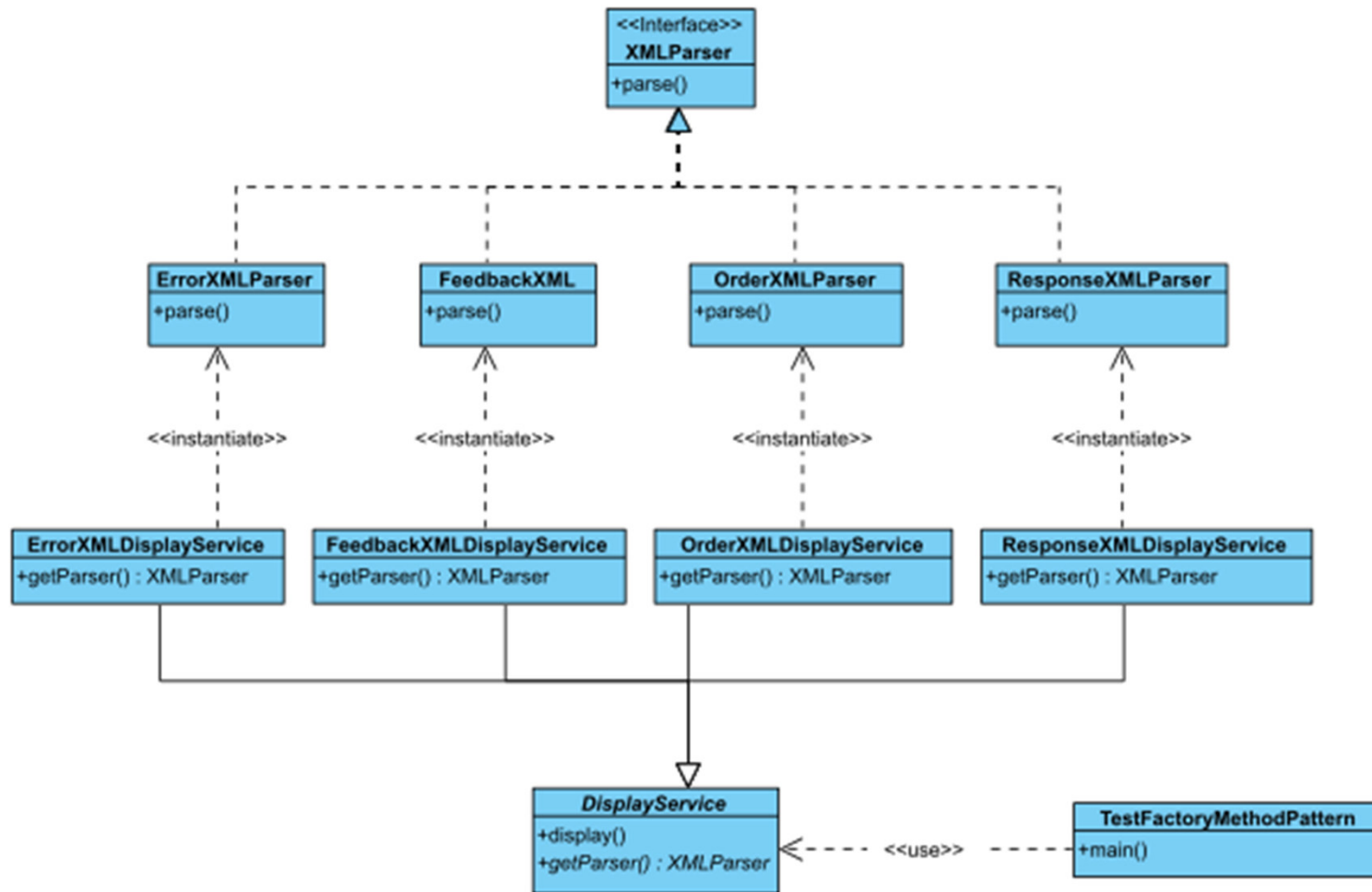
Soluzione con Factory Method Design Pattern

- Il Factory Method Pattern è adatto a questa situazione, in quanto definisce un'interfaccia per la creazione di un oggetto, ma consente alle sottoclassi di decidere quale classe instanziare. 
- Factory Method permette a una classe di delegare l'istanziamento alle sottoclassi.

Struttura del Pattern



Applicazione del Pattern all'esempio



Esempio di Codice 1/3

```
public interface XMLParser {  
    public String parse();  
}
```

```
public class ResponseXMLParser implements XMLParser{  
    @Override  
    public String parse() {  
        System.out.println("Parsing response XML...");  
        return "Response XML Message";  
    }  
}
```


Esempio di Codice 2/3

```
public abstract class DisplayService {  
    public void display(){  
        XMLParser parser = getParser();  
        String msg = parser.parse();  
        System.out.println(msg);  
    }  
    protected abstract XMLParser getParser();  
}
```

```
public class ResponseXMLDisplayService extends DisplayService{  
    @Override  
    public XMLParser getParser() {  
        return new ResponseXMLParser();  
    }  
}
```

Esempio di Codice 3/3

```
public class TestFactoryMethodPattern {  
    public static void main(String[] args) {  
        DisplayService service = new FeedbackXMLDisplayService();  
        service.display();  
        service = new ErrorXMLDisplayService();  
        service.display();  
        service = new OrderXMLDisplayService();  
        service.display();  
        service = new ResponseXMLDisplayService();  
        service.display();  
    }  
}
```



Istanziamo il parser, analizziamo il file XML e mostriamo i risultati a video

Singleton Pattern

- È consentita / necessario avere una sola istanza di una classe, ovvero un **Singleton**.
 - Es. quando si crea il contesto di un'applicazione, un driver per la connessione input/output con una console, un pool di thread, etc.
 - Più oggetti di tale classe causerebbero inconsistenza nel programma.
- Tale pattern assicura che ci sia una solo istanza della classe, e fornisce un punto di accesso **globale**.

Progettazione e implementazione

<<Singleton>> SingletonLazy
-sc : SingletonLazy
-SingletonLazy() +getInstance() : SingletonLazy

Dichiarazione

```
public class SingletonLazy {  
    private static SingletonLazy sc = null;  
    private SingletonLazy() {}  
    public static SingletonLazy getInstance() {  
        if (sc == null) {  
            sc = new SingletonLazy();  
        }  
        return sc;  
    }  
}
```

Utilizzo

```
SingletonLazy sl = SingletonLazy.getInstance();
```

Problemi in concorrenza?

- Sembra che tutto funzioni alla perfezione, vero?
- NO! Questo codice fallisce in uno scenario multi-threaded.
- Ad esempio:
 - Ci sono due thread t1 e t2 in concorrenza sul singleton.
 - t1 chiama per primo il metodo `getInstance()`.
 - Verifica che la variabile statica `sc` sia null e poi viene interrotto per qualche motivo.
 - A questo punto, t2 chiama il metodo `getInstance()`, supera il controllo e istanzia l'oggetto singleton.
 - Quando t1 si sveglia anch'esso crea l'oggetto singleton.
 - Ci saranno due oggetti della classe singleton.

Una possibile soluzione



```
public class SingletonLazyMultithreaded {  
    private static SingletonLazyMultithreaded sc = null;  
    private SingletonLazyMultithreaded() {}  
  
    public static synchronized SingletonLazyMultithreaded  
    getInstance() {  
        if(sc==null) {  
            sc = new SingletonLazyMultithreaded();  
        }  
        return sc;  
    }  
}
```

In questo modo, forziamo ogni thread ad attendere il suo turno prima che possa entrare nel metodo.

Quindi, due thread non eseguiranno mai contemporaneamente il metodo.

CONTRO: riduce le performance del Sistema.

MA: Ma se la chiamata al metodo `getInstance()` non sta causando un sostanziale sovraccarico per la tua applicazione, dimentica....

Double-Checked locking

La keyword indica alla JVM che una variabile può essere modificata in modo "asincrona" da diversi thread. La JVM ricarica e salva in memoria il valore della variabile ad ogni accesso alla variabile.

```
public class SingletonLazyDoubleCheck {  
    private volatile static SingletonLazyDoubleCheck sc = null;  
    private SingletonLazyDoubleCheck() {}  
    public static SingletonLazyDoubleCheck getInstance() {  
        if (sc==null) {  
            synchronized (SingletonLazyDoubleCheck.class) {  
                if (sc==null) {  
                    sc = new  
SingletonLazyDoubleCheck();  
                }  
            }  
        }  
        return sc;  
    }  
}
```

Riduce l'uso della sincronizzazione.

Controlliamo per prima cosa se esiste già un'altra istanza, e se no, sincronizziamo.

In questo modo, sincronizziamo solo la prima volta che l'oggetto viene creato.

Pattern Strutturali

- I pattern strutturali riguardano la composizione di classi e oggetti per formare strutture più grandi e/o complessi.
- Usano l'ereditarietà per comporre interfacce o implementazioni di queste.

Elenco dei Pattern Strutturali

- 1. Adapter**
2. Bridge
- 3. Composite**
4. Decorator
- 5. Façade**
6. Flyweight
- 7. Proxy**

Adapter Pattern

- Si può utilizzare quando ad esempio:
 - Data una classe la sua interfaccia non corrisponde a quella richiesta dalla classe client.
 - Si desidera creare una classe riusabile che coopera con classi non correlate o impreviste, cioè classi che non necessariamente dispongono di interfacce compatibili.
- È anche detto **Wrapper**.

Un'illustrazione di Adapter Pattern

Abbiamo questo



E questo?



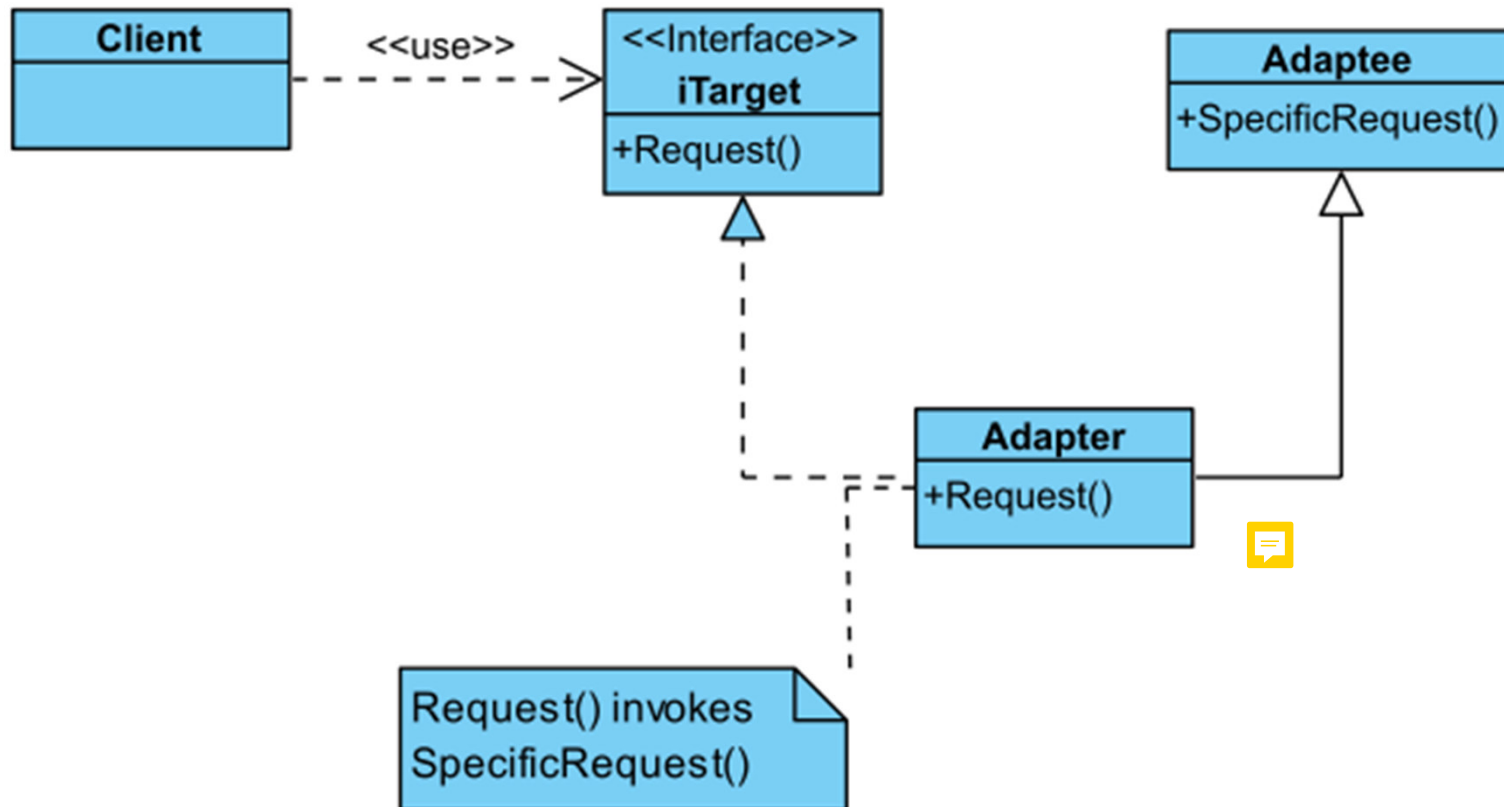
Usiamo l'adattatore



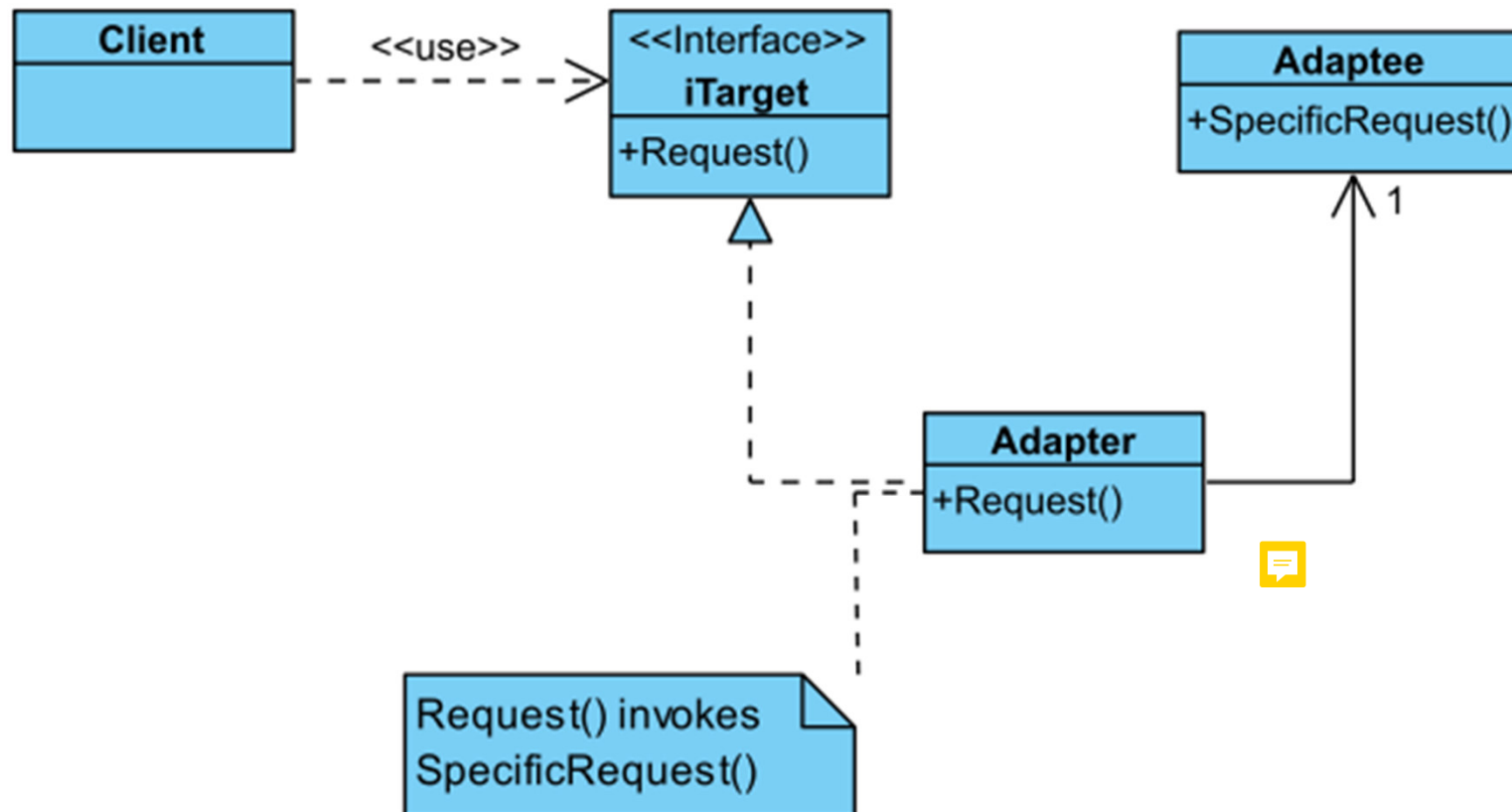
Un esempio pratico di utilizzo dell'Adapter Pattern

1. A software developer, Max, has worked on an e-commerce website. The website allows users to shop and pay online.
2. The site is integrated with a 3rd party payment gateway, through which users can pay their bills using their credit card. Everything was going well, until his manager called him for a change in the project.
3. **The manager told him that they are planning to change the payment gateway vendor, and he has to implement that in the code.**
4. The problem that arises here is that the site is attached to the Xpay payment gateway which takes an Xpay type of object.
5. The new vendor, PayD, only allows the PayD type of objects to allow the process.
6. Max doesn't want to change the whole set of 100 of classes which have reference to an object of type XPay.
7. This also raises the risk on the project, which is already running on the production. Neither he can change the 3rd party tool of the payment gateway.
8. The problem has occurred due to the incompatible interfaces between the two different parts of the code.
9. In order to get the process work, **Max needs to find a way to make the code compatible with the vendor's provided API.**

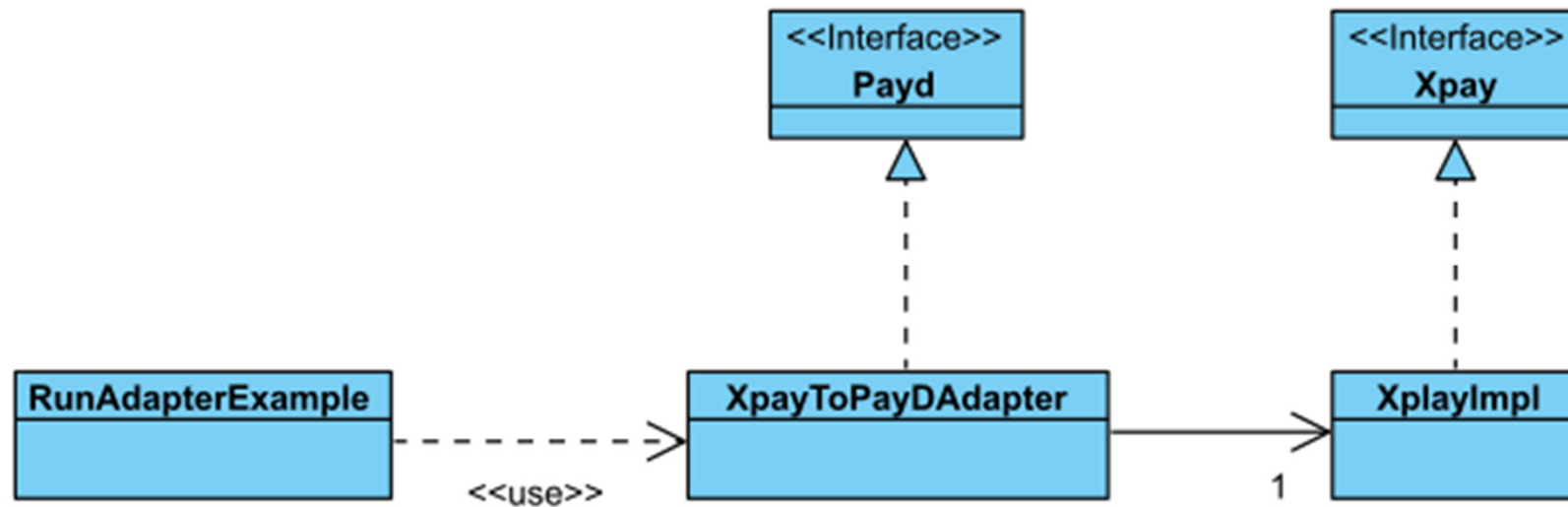
Struttura del Pattern



Struttura alternativa



Soluzione del Problema



Non sono stati aggiunti i metodi e le operazioni per leggibilità

Esempio di Codice

```
public interface Xpay {  
    public String getCreditCardNo();  
    public String getCustomerName();  
    public String getCardExpMonth();  
    public String getCardExpYear();  
    public Short getCardCVVNo();  
    public Double getAmount();  
  
    public void setCreditCardNo(String creditCardNo);  
    public void setCustomerName(String customerName);  
    public void setCardExpMonth(String cardExpMonth);  
    public void setCardExpYear(String cardExpYear);  
    public void setCardCVVNo(Short cardCVVNo);  
    public void setAmount(Double amount);  
}
```

```
public interface PayD {  
    public String getCustCardNo();  
    public String getCardOwnerName();  
    public String getCardExpMonthDate();  
    public Integer getCVVNo();  
  
    public Double getTotalAmount();  
    public void setCustCardNo(String custCardNo);  
    public void setCardOwnerName(String cardOwnerName);  
    public void setCardExpMonthDate(String cardExpMonthDate);  
    public void setCVVNo(Integer cVVNo);  
    public void setTotalAmount(Double totalAmount);  
}
```

Esempio di codice – l'Adapter

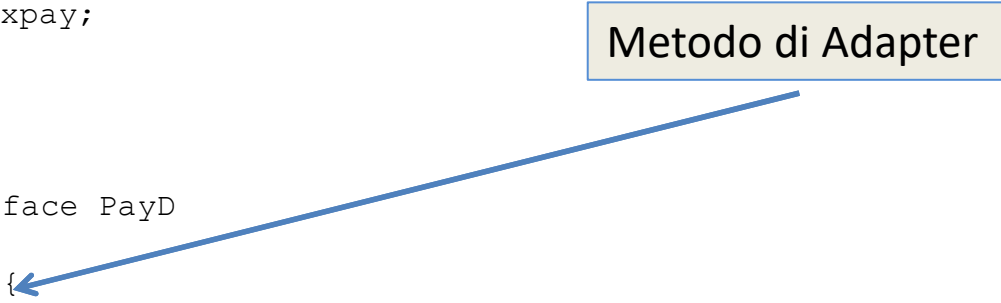
```
public class XpayToPayDAdapter implements PayD {
    private String custCardNo;
    private String cardOwnerName;
    private String cardExpMonthDate;
    private Integer cVVNo;
    private Double totalAmount;
    private final Xpay xpay;

    public XpayToPayDAdapter(Xpay xpay) {
        this.xpay = xpay;
        setProp();
    }

    @Override
    //all methods of interface PayD

    private void setProp() {
        setCardOwnerName(this.xpay.getCustomerName());
        setCustCardNo(this.xpay.getCreditCardNo());

        setCardExpMonthDate(this.xpay.getCardExpMonth()+"/"+this.xpay.getCardExpYear());
        setCVVNo(this.xpay.getCardCVVNo().intValue());
        setTotalAmount(this.xpay.getAmount());
    }
}
```



Metodo di Adapter

Usare l'Adapter

```
public class RunAdapterExample {
    public static void main(String[] args) {
        // Object for Xpay
        Xpay xpay = new XpayImpl();
        xpay.setCreditCardNo("4789565874102365");
        xpay.setCustomerName("Max Warner");
        xpay.setCardExpMonth("09");
        xpay.setCardExpYear("25");
        xpay.setCardCVVNo((short)235);
        xpay.setAmount(2565.23);
        PayD payD = new XpayToPayDAdapter(xpay);
        testPayD(payD);
    }

    private static void testPayD(PayD payD) {
        System.out.println(payD.getCardOwnerName());
        System.out.println(payD.getCustCardNo());
        System.out.println(payD.getCardExpMonthDate());
        System.out.println(payD.getCVVNo());
        System.out.println(payD.getTotalAmount());
    }
}
```

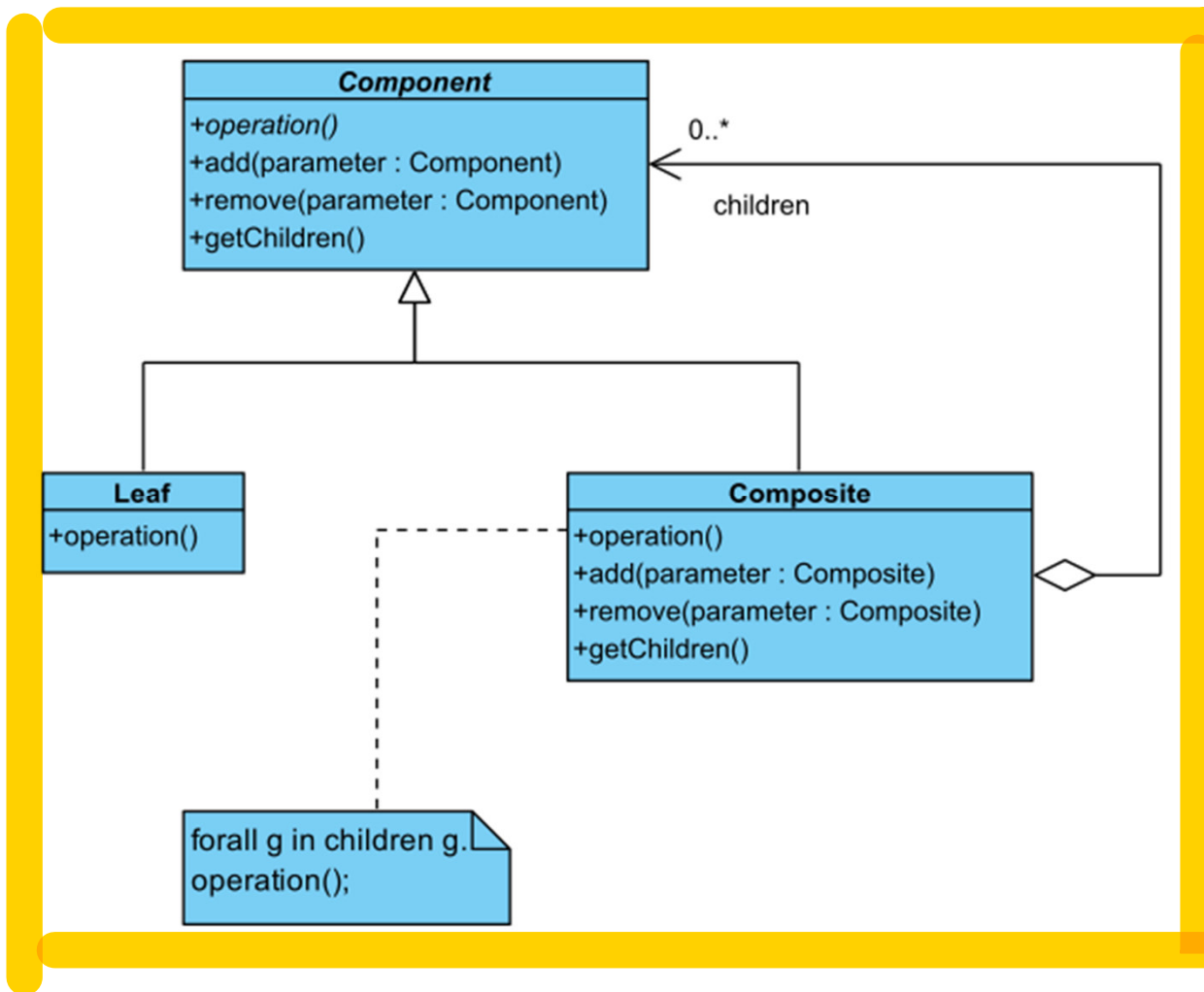

Composite Pattern

- Descrive come costruire una gerarchia di classi costituita da classi per due tipi di oggetti: primitivi e compositi.
- Gli oggetti compositi consentono di comporre oggetti primitivi e altri oggetti compositi in strutture arbitrariamente complesse.
- Consente di comporre gli oggetti in una struttura ad albero per rappresentare una intera gerarchia.
- È possibile creare un albero di oggetti che è fatto di parti diverse, ma che può essere trattato come un unico insieme.

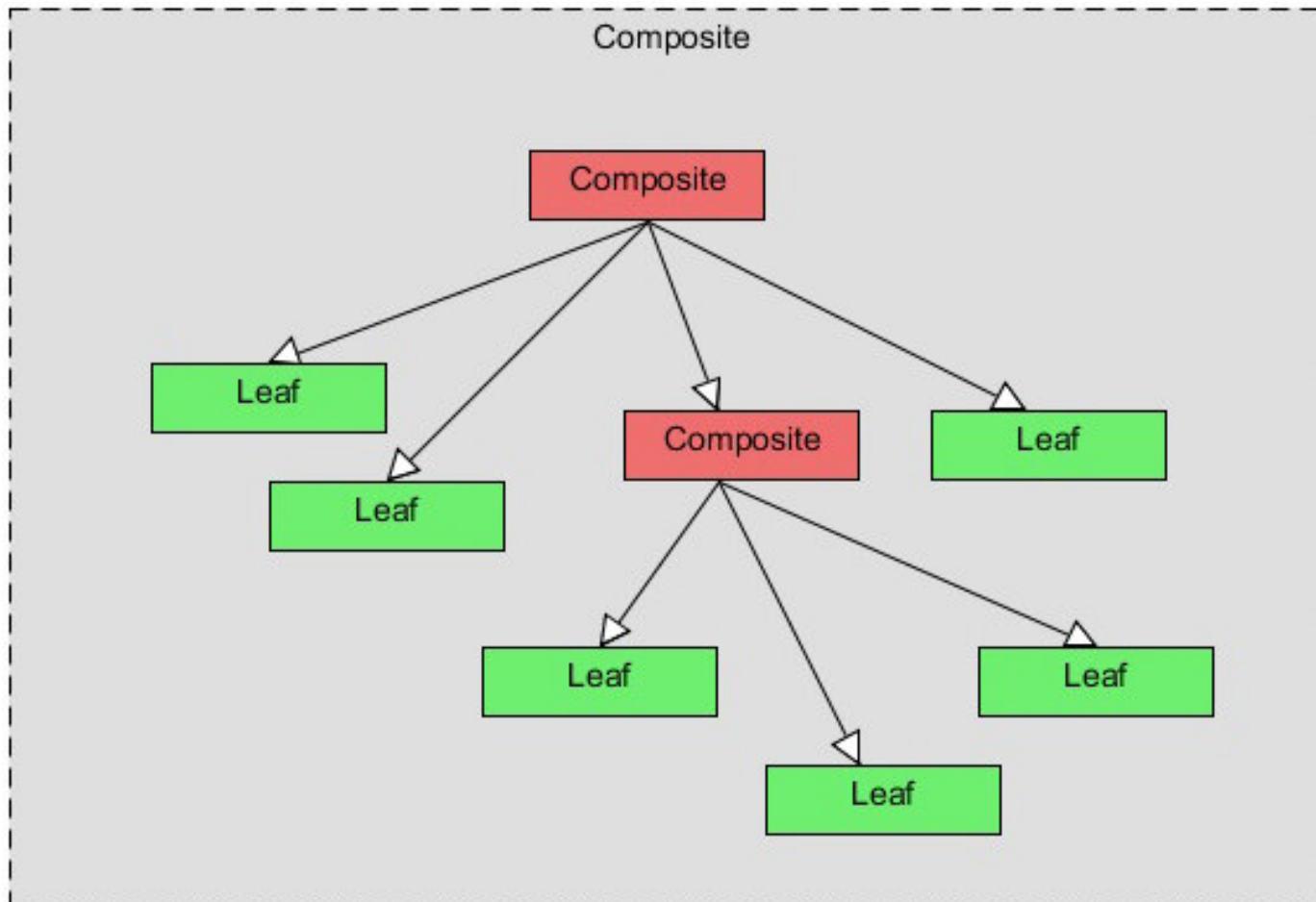
Esempi pratici di utilizzo del pattern

- File directory system
- html representation in java
- XML parser
- Graphical User Interface (GUI)
- Etc..

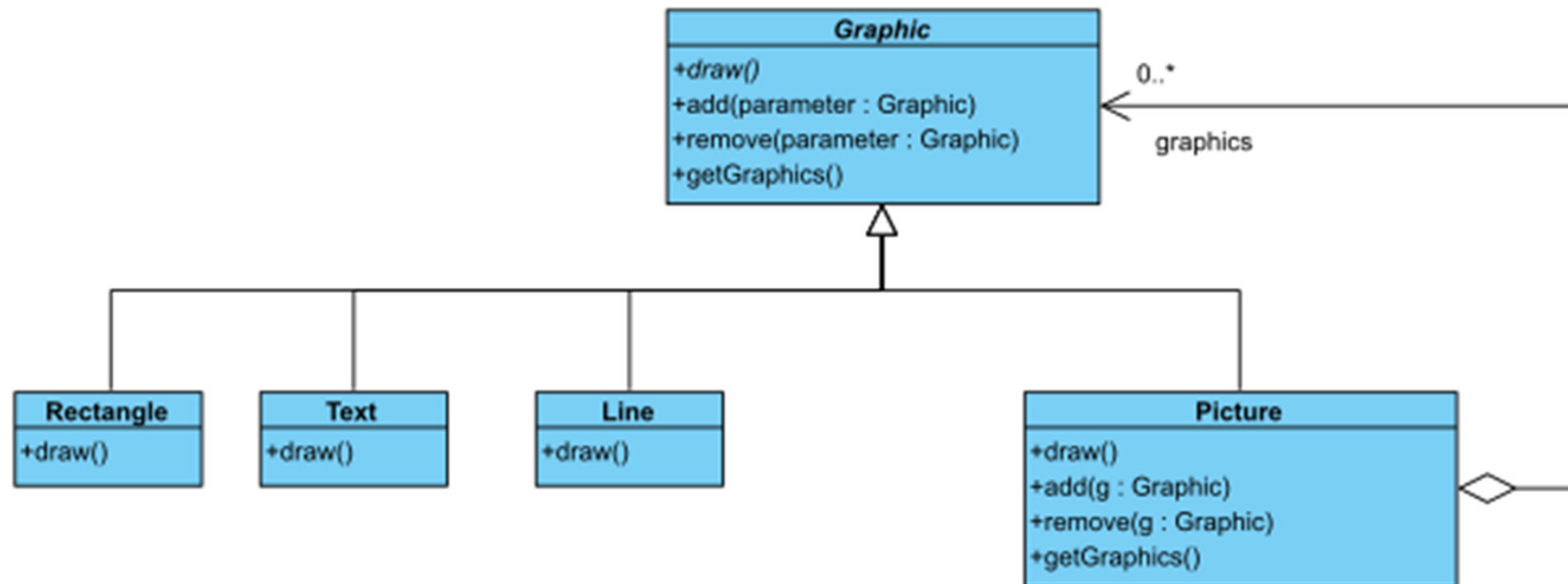
Struttura del Pattern




A runtime



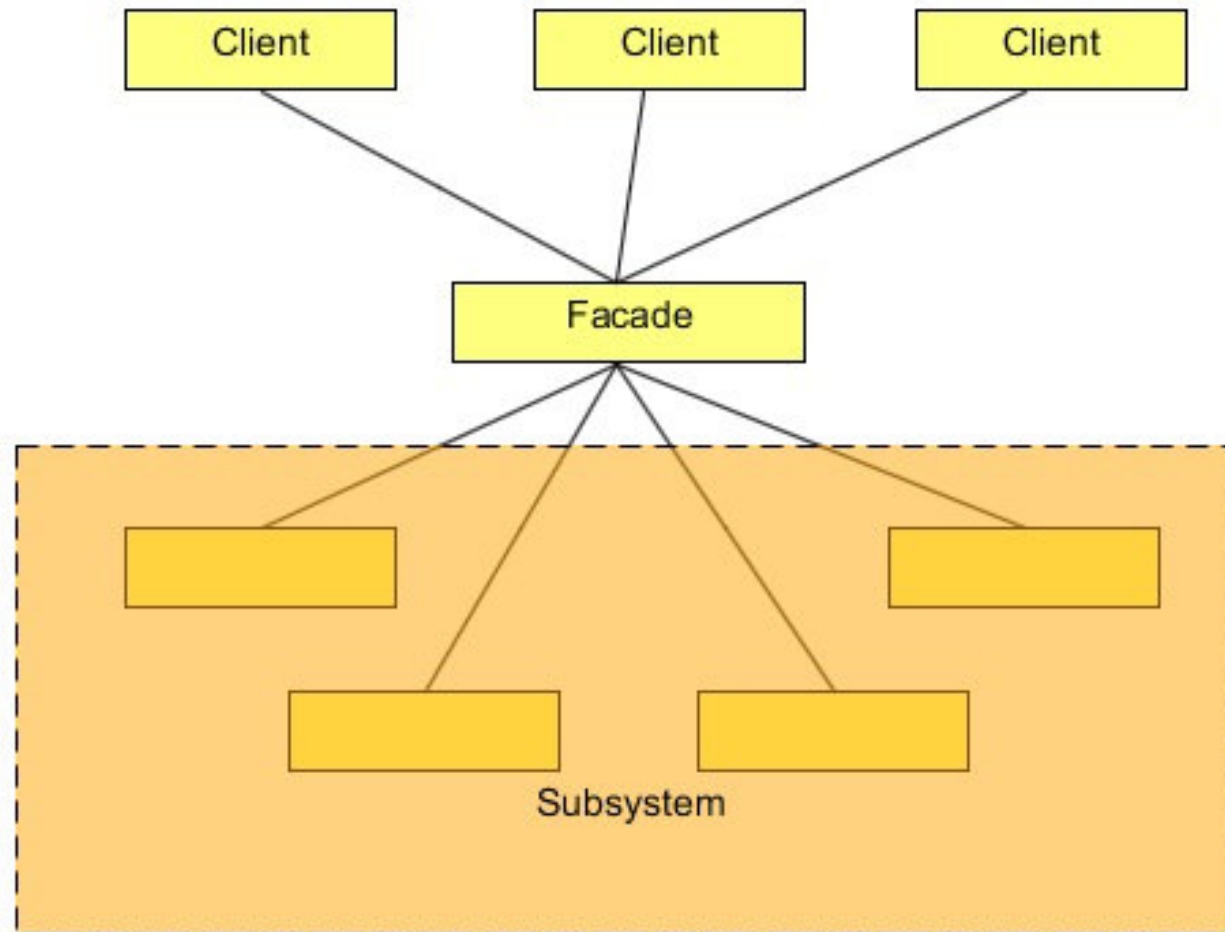
Esempio nel caso di GUI



Façade Pattern

- Fornisce un'interfaccia unificata verso un insieme di interfacce di un sottosistema.
- Definisce un'interfaccia di **alto livello** che rende il sistema più semplice da utilizzare.
- Unifica le interfacce complesse di più **basso livello** fornendo un unico punto di accesso.
- Non solo semplifica le interfacce ma disaccoppia i client dal sottosistema. 

Esempio



Un Façade così come un Adapter può wrappare più classi ma:

- Il Façade è utilizzato per semplificare l'uso di un'interfaccia complessa.
- L'Adapter è utilizzato per convertire le interfacce.

Un esempio pratico di utilizzo del Façade Pattern

1. Your company has launched a product in the market, named **Schedule Server**. It is a kind of server in itself, and it is **used to manage jobs**.
2. The jobs could be any kind of jobs like sending a list of **emails, sms, reading or writing files** from a destination, or **just simply transferring files from a source to the destination**.
3. The product is used by the developers to manage such kind of jobs and able to concentrate more towards their business goal.
4. The server executes each job at their specified time and also manages all underline issues like concurrency issue and security by itself. As a developer, one just need to code only the relevant business requirements and a good amount of API calls is provided to schedule a job according to their needs.
5. **Everything was going fine, until the clients started complaining about starting and stopping the process of the server.**
6. **They said, although the server is working great, the initializing and the shutting down processes are very complex and they want an easy way to do that.**
7. **The server has exposed a complex interface to the clients which looks a bit hectic to them.**
8. **We need to provide an easy way to start and stop the server.**
9. A complex interface to the client is already considered as a fault in the design of the current system. But fortunately or unfortunately, we cannot start the designing and the coding from scratch.
10. We need a way to resolve this problem and make the interface easy to access.

Avvio del Server

```
ScheduleServer scheduleServer = new ScheduleServer();

scheduleServer.startBooting();
scheduleServer.readSystemConfigFile();
scheduleServer.init();
scheduleServer.initializeContext();
scheduleServer.initializeListeners();
scheduleServer.createSystemObjects();

System.out.println("Start working.....");
System.out.println("After work done.....");
```

Per far partire il server, il client deve creare un oggetto della classe `ScheduleServer` e successivamente deve chiamare un insieme di metodi per avviare ed inizializzare il server.

Arresto del Server

```
scheduleServer.releaseProcesses();  
scheduleServer.destroy();  
  
scheduleServer.destroySystemObjects();  
scheduleServer.destroyListeners();  
scheduleServer.destroyContext();  
scheduleServer.shutdown();
```

Soluzione adottando il Façade

```
public class ScheduleServerFacade {  
    private final ScheduleServer scheduleServer;  
    public ScheduleServerFacade(ScheduleServer scheduleServer){  
        this.scheduleServer = scheduleServer;  
    }
```

```
    public void startServer(){  
        scheduleServer.startBooting();  
        scheduleServer.readSystemConfigFile();  
        scheduleServer.init();  
        scheduleServer.initializeContext();  
        scheduleServer.initializeListeners();  
        scheduleServer.createSystemObjects();  
    }
```

```
    public void stopServer(){  
        scheduleServer.releaseProcesses();  
        scheduleServer.destory();  
        scheduleServer.destorySystemObjects();  
        scheduleServer.destoryListeners();  
        scheduleServer.destoryContext();  
        scheduleServer.shutdown();  
    }
```

```
}
```

ScheduleServerFacade
è la classe façade , che
wrappa un oggetto
ScheduleServer, questa:

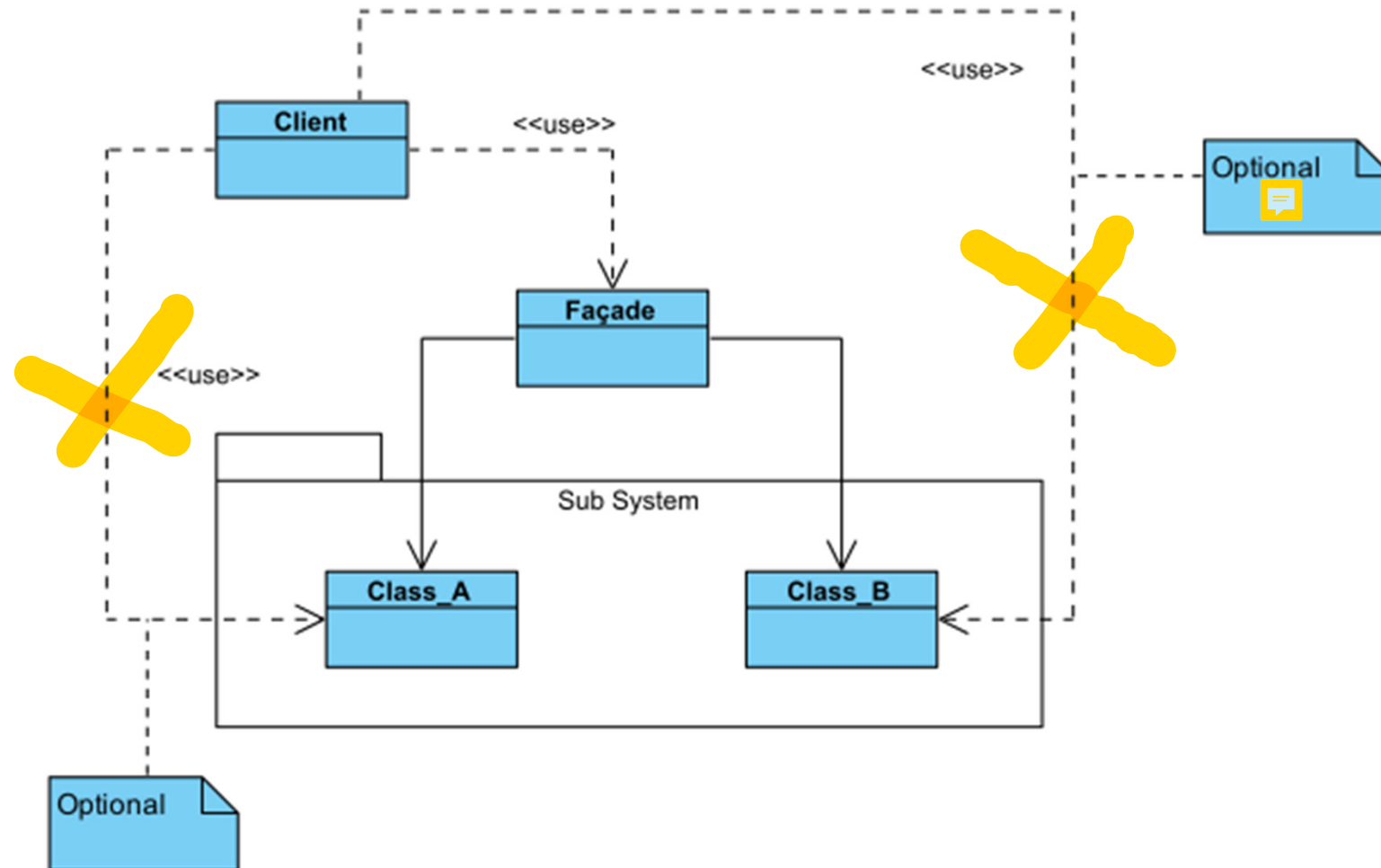
1. Istanza il server mediante
il suo costruttore
2. Fornisce due semplici
metodi per avviare ed
arrestare il server.

Utilizzo della classe Façade da parte del client


```
public class TestFacade {  
    public static void main(String[] args) {  
        ScheduleServer scheduleServer = new ScheduleServer();  
  
        ScheduleServerFacade facadeServer = new  
            ScheduleServerFacade(scheduleServer);  
  
        facadeServer.startServer();  
        System.out.println("Start working.....");  
        System.out.println("After work done.....");  
        facadeServer.stopServer();  
    }  
}
```

Il client continua ad accedere alla classe `ScheduleServer`, ma lo fa attraverso il Façade


Struttura del pattern



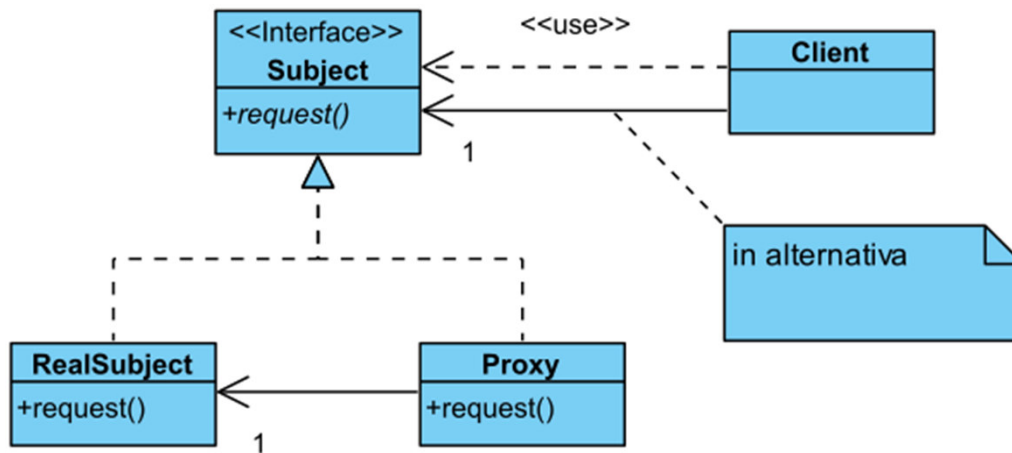
Proxy Pattern

- È utilizzato per creare un oggetto che controlla l'accesso ad un altro oggetto che sia:
 - Remoto. 
 - Costoso da creare.
 - Sottoposto a specifici diritti di protezione.
- In questo pattern il **client** non interagisce direttamente con l'oggetto originale.
 - Delega le sue chiamate all'oggetto proxy che a sua volta richiama i metodi dell'oggetto originale.

Varianti del Proxy Pattern

- Il pattern presenta quattro varianti che possono essere utilizzati in base al problema da risolvere:
 - **Remote Proxy:** fornisce una rappresentazione locale per un oggetto che si trova in differente address space
 - Oggetto Remoto (es. in esecuzione su una differente JVM).
 - **Virtual Proxy:** ritarda la creazione e/o l'inizializzazione dell'oggetto poiché richiede ingenti risorse (es: caricamento immagini). 
 - **Smart Proxy:** fornisce una ottimizzazione dell'oggetto (es: caricamento in memoria dell'oggetto)
 - **Protection Proxy:** fornisce controlli sugli accessi all'oggetto originale.
 - Utile quando gli oggetti dovrebbero avere diversi diritti di accesso.
 - Es: richiesta username/password per l'accesso

Struttura del pattern



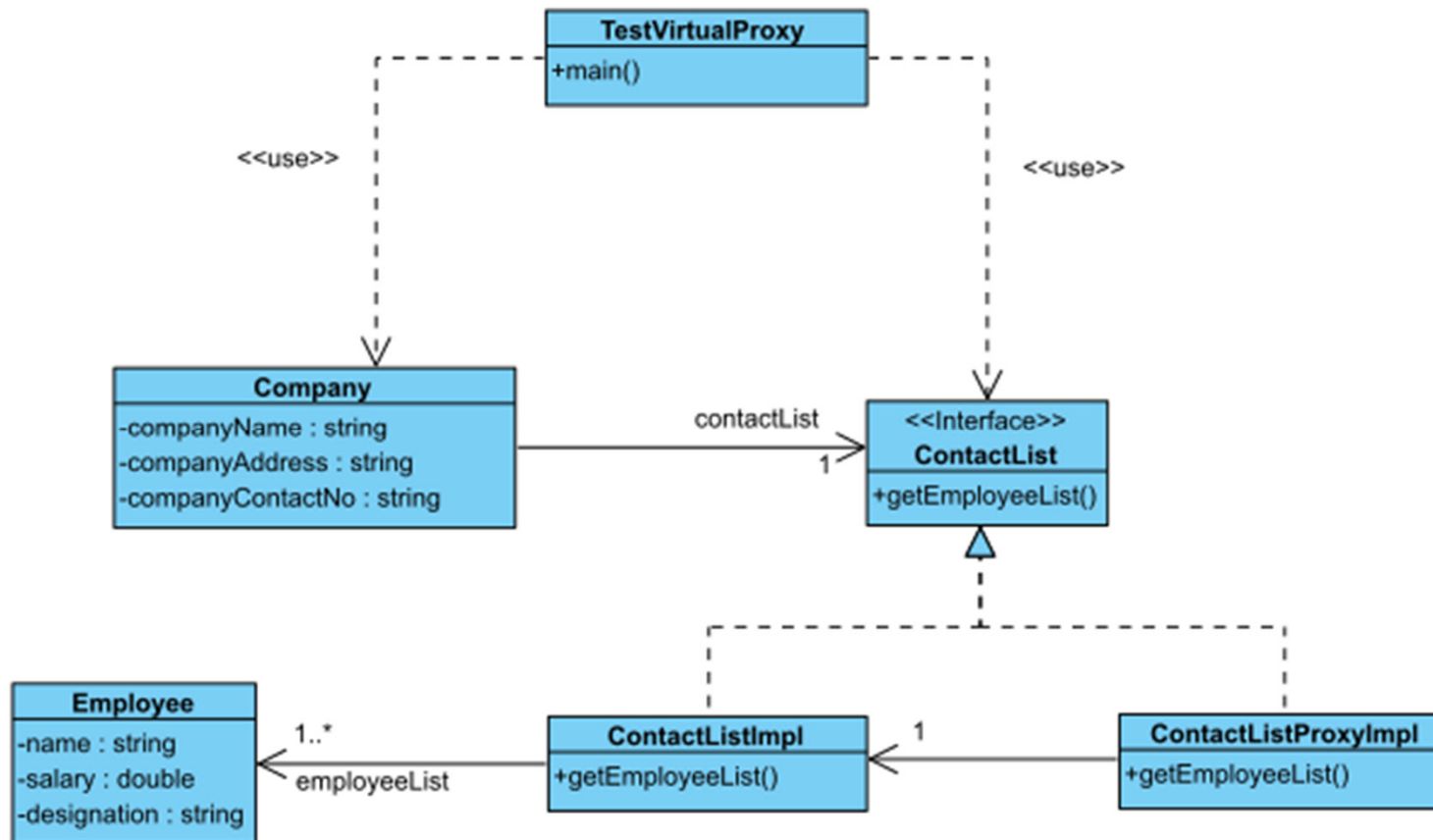
Elementi del Pattern:

1. Proxy: mantiene un riferimento che consente al Proxy di accedere al real subject.
2. Subject: Definisce una interfaccia comune per RealSubject e Proxy. In questo modo un Proxy può essere utilizzato ovunque ci si aspetti di utilizzare un RealSubject.
3. RealSubject: definisce l'oggetto reale che il proxy rappresenta.

Esempi pratici del pattern

- Vedremo l'utilizzo di **Remote Proxy** con Java RMI.
- **Virtual Proxy:**
 - Suppose there is a `Company` object in your application and this object contains a list of employees of the company in a `ContactList` object.
 - There could be thousands of employees in a company.
 - Loading the `Company` object from the database along with the list of all its employees in the `ContactList` object could be very time consuming.
 - In some cases you don't even require the list of the employees, but you are forced to wait until the company and its list of employees loaded into the memory.
 - One way to save time and memory is to avoid loading of the employee objects until required, and this is done using the Virtual Proxy.
 - This technique is also known as **Lazy Loading** where you are fetching the data only when it is required.

Soluzione del problema



Utilizzo del pattern per Lazy Load

```
public class TestVirtualProxy {
    public static void main(String[] args) {
        ContactList contactList = new ContactListProxyImpl();
        Company company = new Company("ABC Company", "India", "+91-011-28458965", contactList);
        System.out.println("Company Name:
"+company.getCompanyName());
        System.out.println("Company Address:
"+company.getCompanyAddress());
        System.out.println("Company Contact No.:
"+company.getCompanyContactNo());
        System.out.println("Requesting for contact list");
        contactList = company.getContactList();
        List<Employee>empList = contactList.getEmployeeList();
        for(Employee emp : empList){
            System.out.println(emp);
        }
    }
}
```

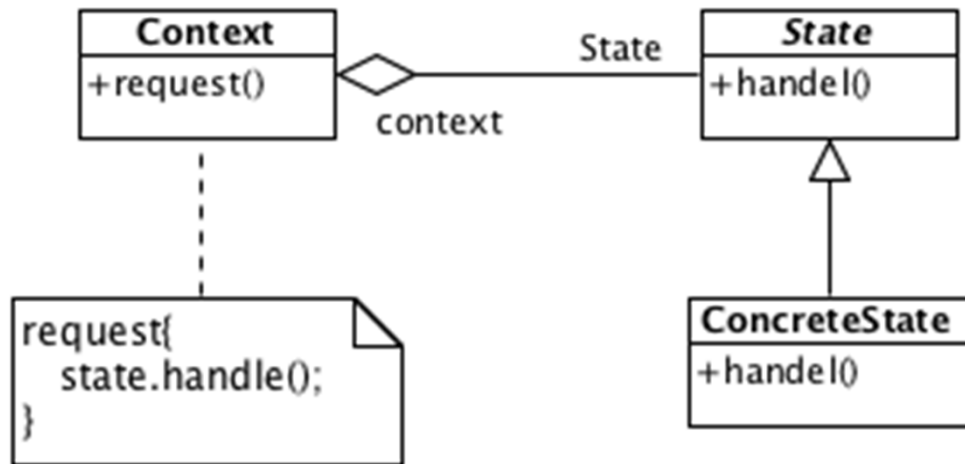
State Pattern

- Il comportamento di una classe dipende dallo stato della classe stessa.
- Pattern state consente ad un oggetto modificare il suo comportamento quando il suo stato interno cambia.

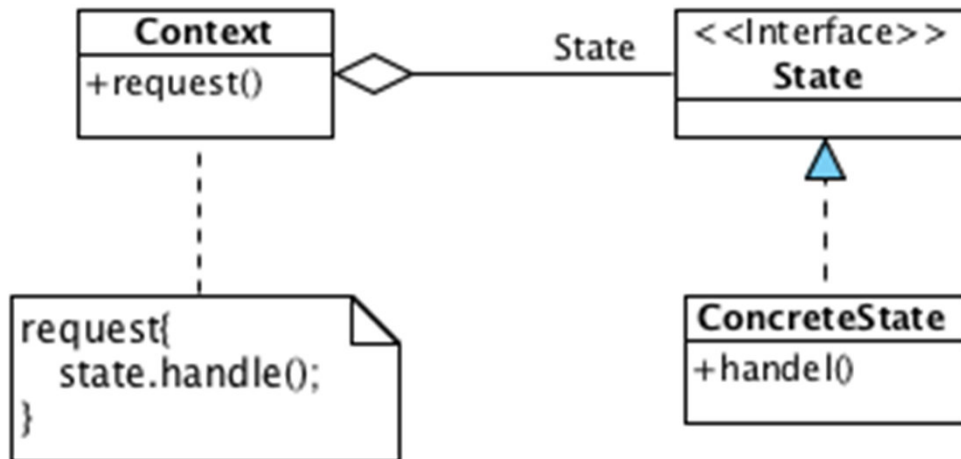
Descrizione del Pattern State

- Il pattern suggerisce incapsulare, all'interno di una classe, il modo particolare in cui le operazioni di un oggetto (**Context**) vengono svolte quando lo si trova in quello stato.
- Ogni classe (**ConcreteState**) rappresenta un singolo stato possibile del Context e implementa una interfaccia comune (**State**) contenente le operazioni che il Context delega allo stato.
- L'oggetto Context deve tenere un riferimento al ConcreteState che rappresenta lo stato corrente.

Modello del Pattern



Modello con classe Astratta

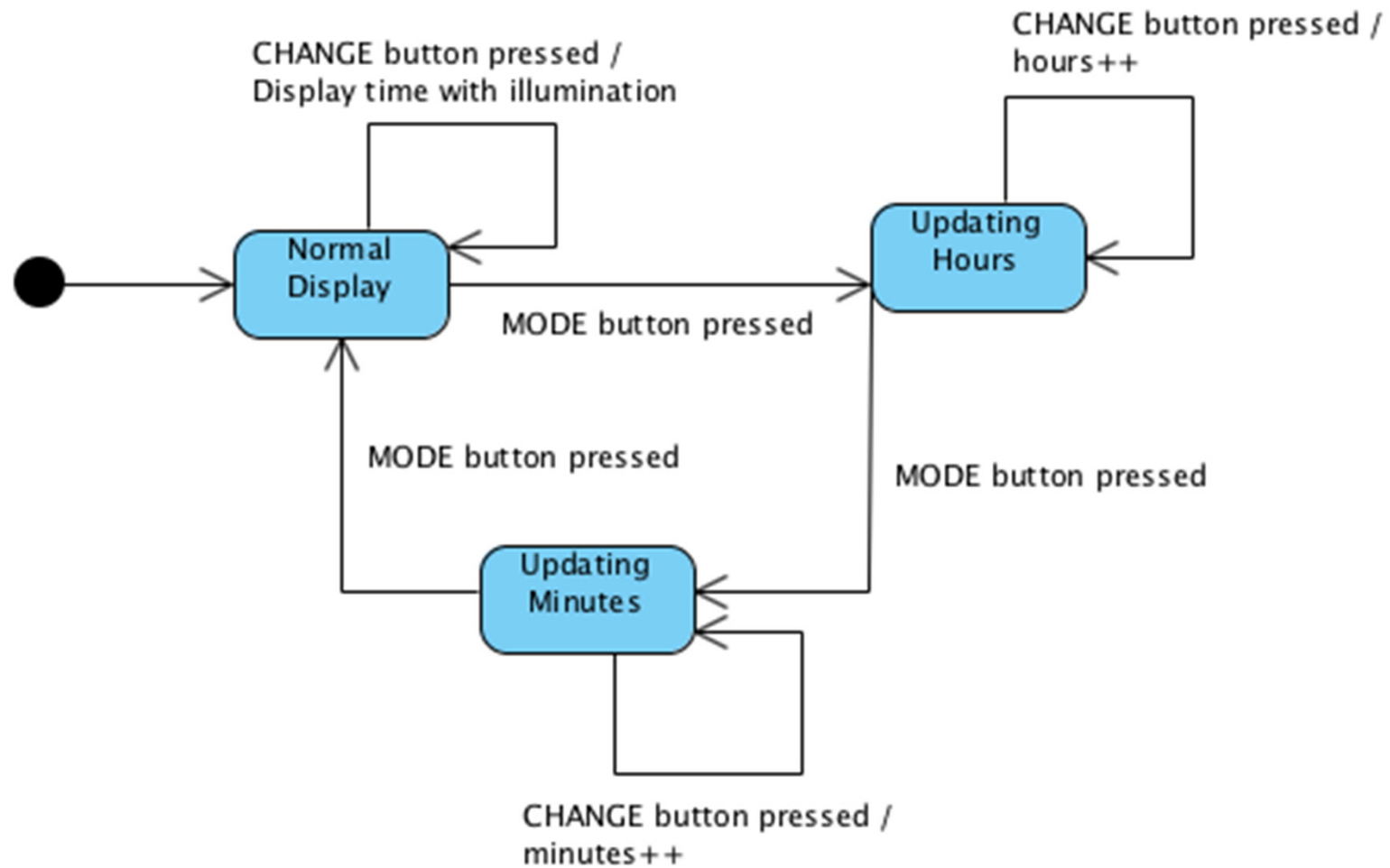


Modello con Interfaccia

Esempio

- Si pensi ad un orologio che possiede due pulsanti: MODE e CHANGE.
- Il primo pulsante serve per settare il modo di operazione di tre modi possibili: “visualizzazione normale”, “modifica delle ore” o “modifica dei minuti”.
- Il secondo pulsante, invece, serve per accendere la luce del display, se è in modalità di visualizzazione normale, oppure per incrementare in una unità le ore o i minuti, se è in modalità di modifica di ore o di minuti.

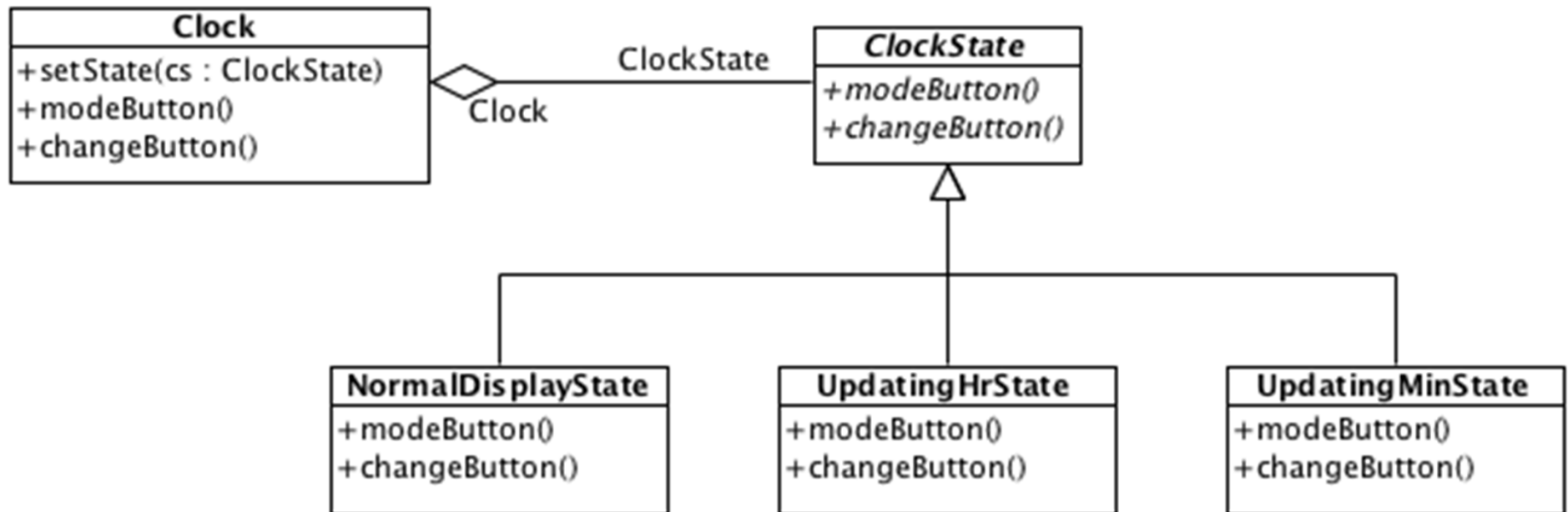
UML State Machine



Problemi di Implementazione

- Un approccio semplicistico condurrebbe all'implementazione del codice di ogni operazione come una serie di decisioni
 - Difficile manutenzione, la creazione di nuovi stati comporta la modifica di tutte le operazioni.
 - Non si tiene una visione dello stato, in modo di capire come agisce l'oggetto.

Applicazione del Pattern



ClockState

- La classe astratta ClockState (State) specifica l'interfaccia che ogni ConcreteState deve implementare.
- Offre due metodi modeButton e changeButton.
- Le operazioni da eseguire se viene premuto il tasto MODE o il tasto CHANGE dell'orologio.
- Queste operazioni hanno comportamenti diversi secondo lo stato in cui ritrova l'orologio.
- La classe ClockState gestisce anche un riferimento all'oggetto Clock a cui appartiene, in modo che iparticolari stati possano accedere alle sue proprietà

```
public abstract class ClockState {  
    protected Clock clock ;  
    public ClockState(Clock clock) {  
        this.clock = clock;  
    }  
    public abstract void modeButton();  
    public abstract void changeButton();  
}
```

NormalDisplayState

- Il ConcreteState NormalDisplayState estende ClockState.
- Il suo costruttore richiama il costruttore della superclasse per la gestione del riferimento al rispettivo oggetto Clock.
- Il metodo modeButton semplicemente cambia lo stato dell'orologio da "visualizzazione normale" a "aggiornamento delle ore" (creando una istanza di UpdatingHrState e associandola allo stato corrente dell'orologio).
- Il metodo changeButton accende la luce del display per visualizzare l'ora corrente (si ipotizzi che la luce si spenga automaticamente)

```
public class NormalDisplayState extends
ClockState {

    public NormalDisplayState(Clock
clock) {
        super( clock );
        System.out.println( "** Clock is
in          normal display.");
    }

    public void modeButton() {
        clock.setState( new
UpdatingHrState( clock ) );
    }

    public void changeButton() {
        System.out.print( "LIGHT ON: " );
        clock.showTime();
    }
}
```

Clock

- La classe Clock rappresenta il Context dello stato.
- Lo stato corrente di ogni istanza di Clock viene gestito con un riferimento verso un oggettoConcreteState (variabile clockState), tramite l'interfaccia ClockState.
- Al momento della creazione, ogni clock viene settato alle ore 12:00 e nello stato di visualizzazione normale.

```
public class Clock {  
    private ClockState clockState;  
    public int hr, min;  
    public Clock() {  
        clockState = new NormalDisplayState(  
            this );  
    }  
    public void setState( ClockState cs ) {  
        clockState = cs;  
    }  
    public void modeButton() {  
        clockState.modeButton();  
    }  
    public void changeButton() {  
        clockState.changeButton();  
    }  
    public void showTime() {  
        System.out.println( "Current time is Hr  
        : " + hr + " Min: "  
        + min );  
    }  
}
```

Il pattern State in esecuzione

```
public class StateExample {  
    public static void main ( String arg[] ) {  
        Clock theClock = new Clock();  
        theClock.changeButton();  
        theClock.modeButton();  
        theClock.changeButton();  
        theClock.changeButton();  
        theClock.modeButton();  
        theClock.changeButton();  
        theClock.changeButton();  
        theClock.changeButton();  
        theClock.changeButton();  
        theClock.modeButton();  
    }  
}
```

Observer 1/3

- Il pattern “Observer” assegna all’oggetto monitorato (Subject) il ruolo di registrare ai suoi interni un riferimento agli altri oggetti che devono essere avvisati (ConcreteObservers) degli eventi del Subject, e notificarli tramite l’invocazione a un loro metodo, presente nella interfaccia che devono implementare (Observer)
- Questo pattern è stato proposto originalmente dai GoF per la manutenzione replicata dello stato del ConcreteSubject nei ConcreteObserver
 - motivo per il quale sono previsti una copia dello stato del primo nei secondi, e la esistenza di un riferimento del ConcreteSubject nel ConcreteObserver.

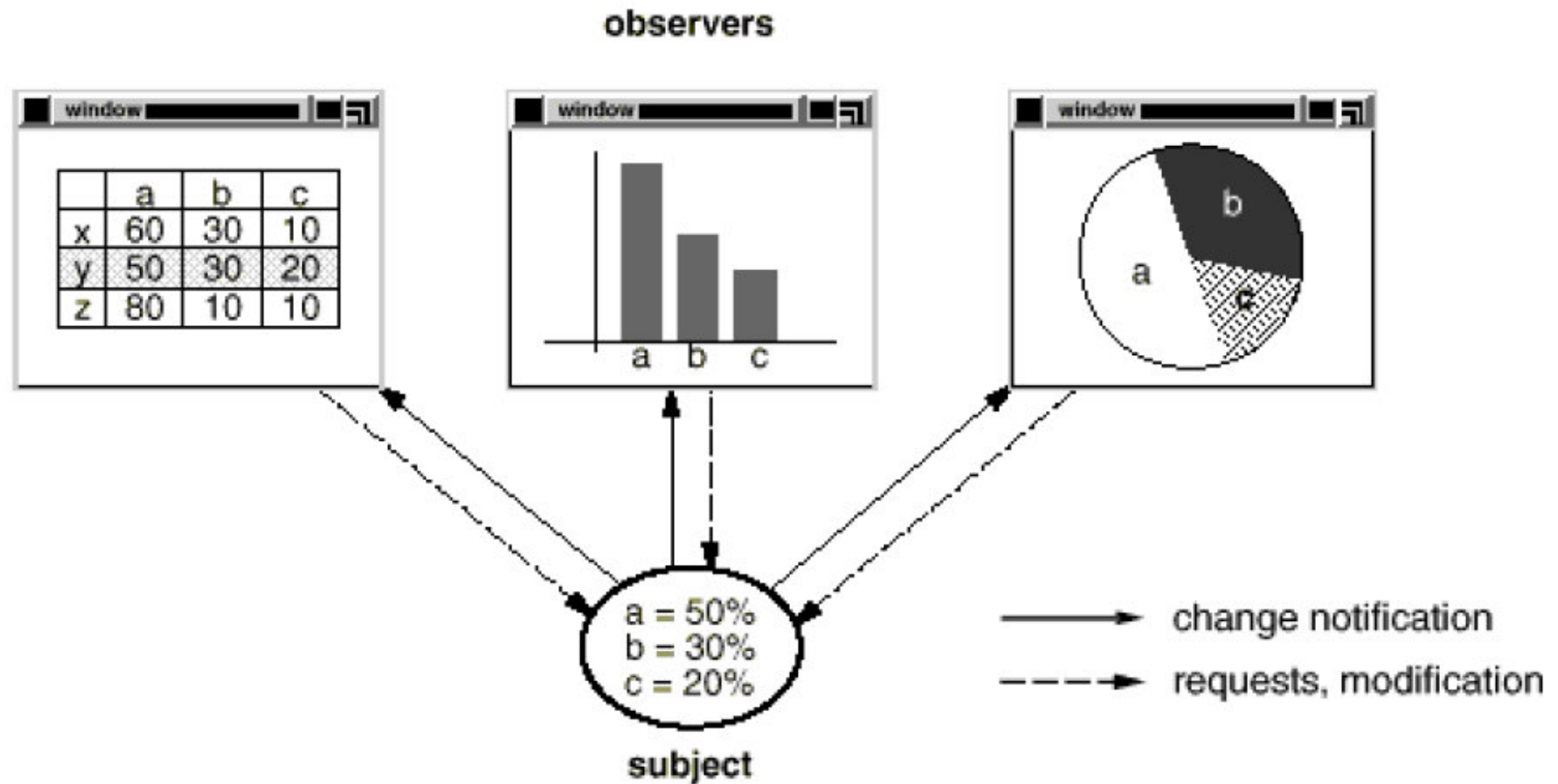
Observer 2/3

- Si deve tenere in conto che questo modello può servire anche a comunicare eventi, in situazioni nelle quali non sia di interesse gestire una copia dello stato del Subject.
- Si noti che non è sempre necessario che ogni ConcreteObserver abbia un riferimento al Subject di interesse, oppure, che i riferimenti siano verso un unico Subject.

Observer 3/3

- Un'altra versione del pattern Observer, esteso con una gestione più completa degli eventi, è implementato in ambienti (framework) per lo sviluppo di GUI o più in generale EBS.
 - In questi modelli è consentita la registrazione del Observer presso il Subject, indicando additionally il tipo di evento per il quale l'Observer deve essere notificato, e la funzione del Observer da invocare.
- Il tipo interazione tra Subject e Obsers è anche noto come **publish-subscribe**.
 - Il subject è il subscriber delle notifiche. Invia queste notifiche senza conoscere chi sono gli osservatori.
 - Un certo numero di osservatori possono sottoscrivere per ricevere le notifiche.

Esempio



Una Struttura del Pattern

