

Ingegneria del Software – Parte II

Pattern della Gang of Four (cont.)



UNIVERSITÀ
DEGLI STUDI
DI BRESCIA

Prof. Alessandro Saetti

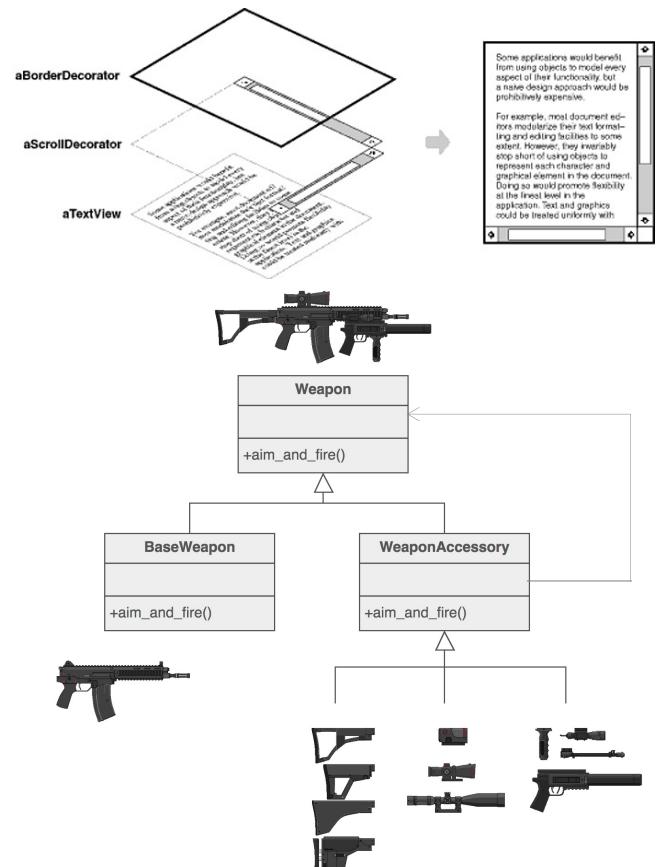
Design pattern



- **Decorator**
- Repository
- Proxy
- Convert Exception
- Abstract Factory
- Template & Factory method
- State

Il problema di estendere le funzionalità

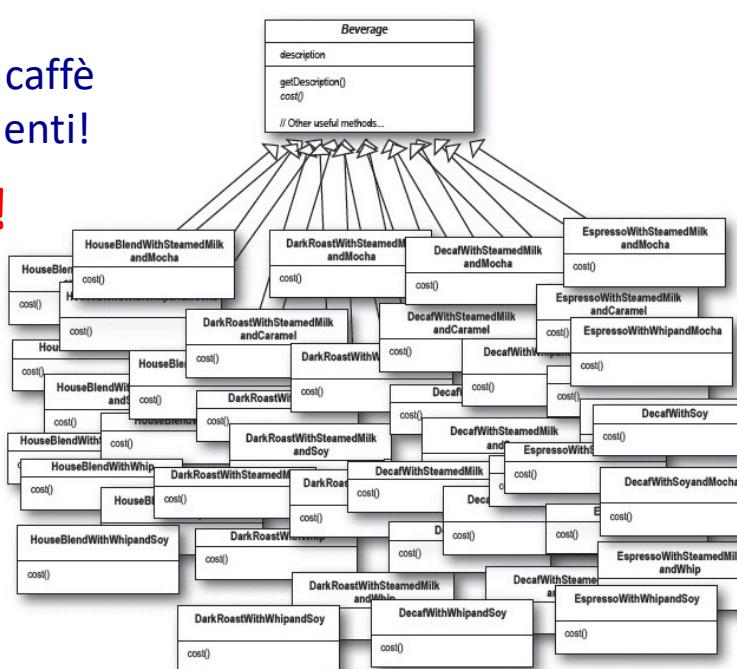
- Vogliamo combinare le decorazioni di una interfaccia grafica (bordi, scrollbar, buttoni, ecc.)
- L'aggiunta di accessori a pistola d'assalto la rendono più precisa, silenziosa e devastante
- Vorrei estendere le funzionalità di base di un oggetto?



Se si usasse l'ereditarietà per estendere il comportamento...

Con 4 tipi di caffè e 4 supplementi!

64 classi!



La gerarchia è statica e ci sarebbe un'esplosione di classi!



Se si usasse l'ereditarietà per estendere il comportamento...

- Beverage è specializzato per i 4 tipi di caffé
- cost() di Beverage calcola il costo dei supplementi
- Le sottoclassi sovrascrivono cost() ma invocano cost della super classe per calcolare il costo del supplemento
- Se il prezzo cambia o sono aggiunti supplementi il costo di Beverage cambia
- Si viola open-closed principle! 🤦

<<abstract>>
Beverage
description: String
milk: boolean
soy: boolean
mocha: boolean
whip: boolean
getDescription()
cost()
hasMilk()
setMilk()
hasSoy()
setSoy()
hasMocha()
setMocha()
hasWhip()
setWhip()

Il pattern Decorator

Problema

Come aggiungere o rimuovere dinamicamente responsabilità a un singolo oggetto?

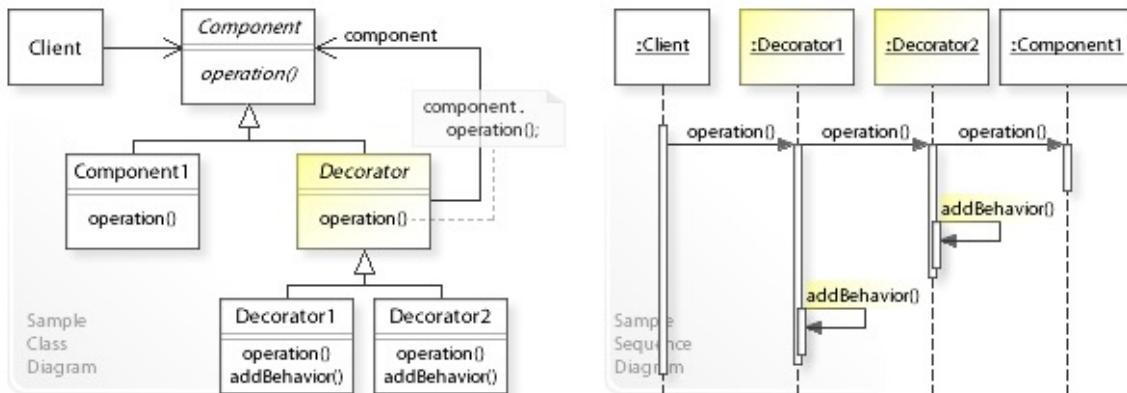
Soluzione

Definisci degli oggetti decoratori che implementano l'interfaccia dell'oggetto esteso (decorato) inoltrando le richieste a esso ed eseguendo funzionalità aggiuntive

Altri usi

Per la realizzazione di interfacce utente

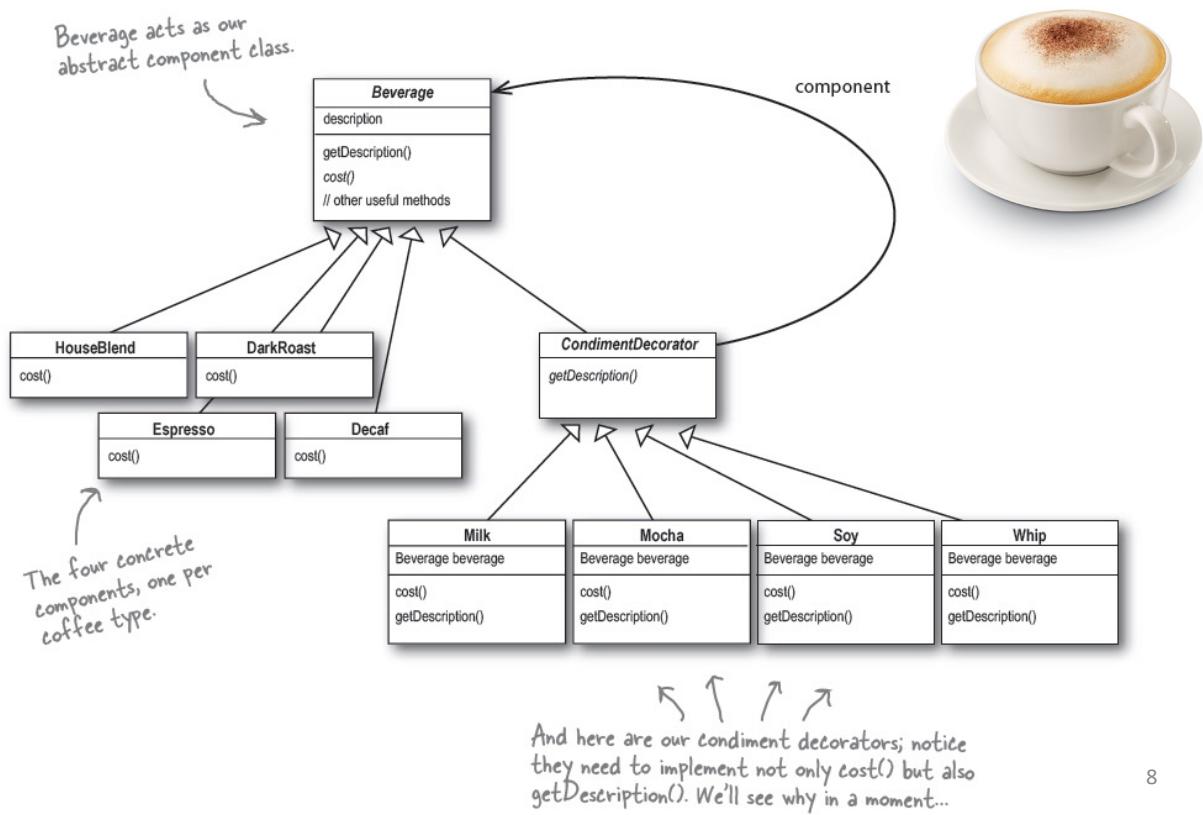
Il pattern Decorator in UML



- **Component:** definisce l'interfaccia per gli oggetti decorati
- **Component1:** l'oggetto a cui si aggiungono responsabilità
- **Decorator:** mantiene un riferimento a un Component e implementa l'interfaccia Component
- **Decorator1,2:** aggiunge responsabilità al Componente

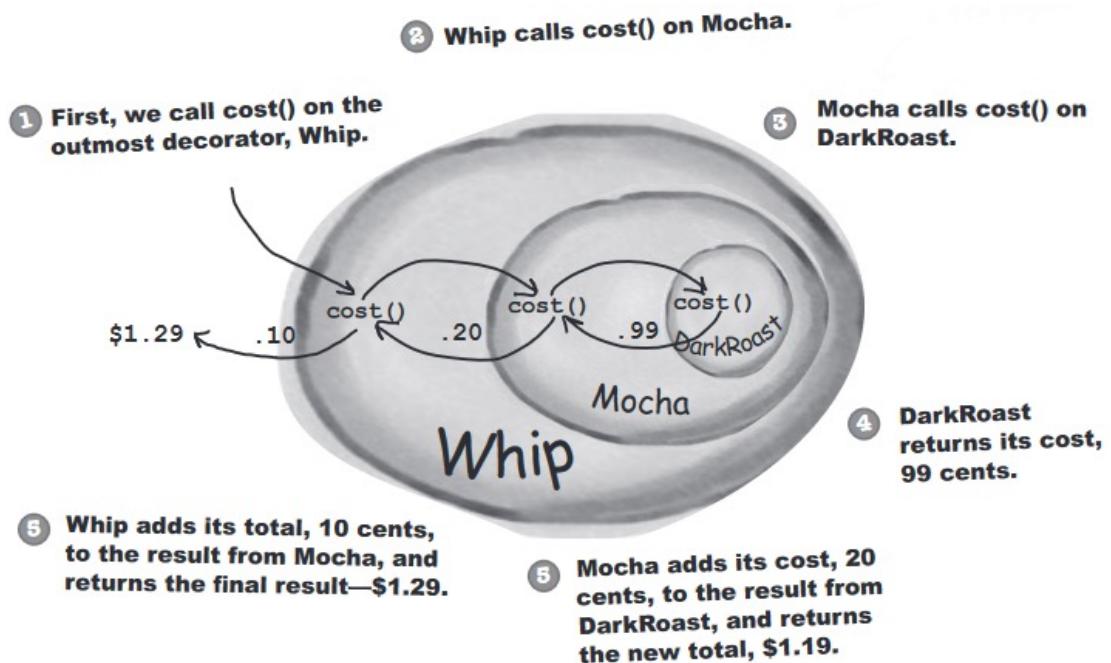
7

Esempio di Decorator



8

Esempio di Decorator (cont.)



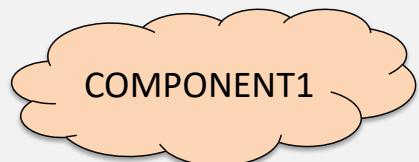
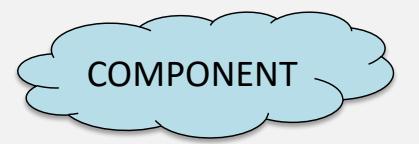
9

Conseguenze

PRO	CONTRO
<ul style="list-style-type: none">• Maggiore flessibilità rispetto alla ereditarietà statica• Evita di definire classi troppo complesse• Evita di definire troppe classi• Permette di definire nuovi decoratori indipendentemente dalle classi decorate	Porta a molti piccoli oggetti tra loro molto simili. Questo rende il sistema difficile da capire e correggere

Il pattern Decorator in codice JAVA

```
public interface Coffee {  
    public double getCost();  
}  
  
public class SimpleCoffee implements Coffee {  
    @Override public double getCost() {  
        return 1;  
    }  
}  
  
public abstract class CoffeeDecorator implements Coffee {  
    protected final Coffee decoratedCoffee;  
  
    public CoffeeDecorator(Coffee c) {  
        this.decoratedCoffee = c;  
    }  
  
    public double getCost() {  
        return decoratedCoffee.getCost();  
    }  
}
```



11

```
class WithMilk extends CoffeeDecorator {  
    public WithMilk(Coffee c) {  
        super(c);  
    }  
  
    public double getCost() {  
        return super.getCost() + 0.5;  
    }  
}  
  
public class Main {  
    public static void printInfo(Coffee c) {  
        System.out.println("Cost: " + c.getCost());  
    }  
  
    public static void main(String[] args) {  
        Coffee c = new SimpleCoffee();  
        printInfo(c);  
        c = new WithMilk(c);  
        printInfo(c);  
    }  
}
```



12

Pattern correlati

Adapter fornisce a un oggetto una nuova interfaccia senza modificare le responsabilità	Decorator modifica le responsabilità di un oggetto senza modificare l'interfaccia
Composite gestisce l'aggregazione di oggetti	Decorator può essere visto come un oggetto composito degenere costituito da un singolo componente
Strategy consente di cambiare gli “organi interni” di un oggetto	Decorator consente di cambiare la “pelle” di un oggetto

13

Design pattern



- Decorator
- **Repository**
- Proxy
- Convert Exception
- Abstract Factory
- Template & Factory method
- State

14

Il problema di gestire dati persistenti

- Pattern Repository (Evans 2004) -

Problema

Come gestire dati persistenti?



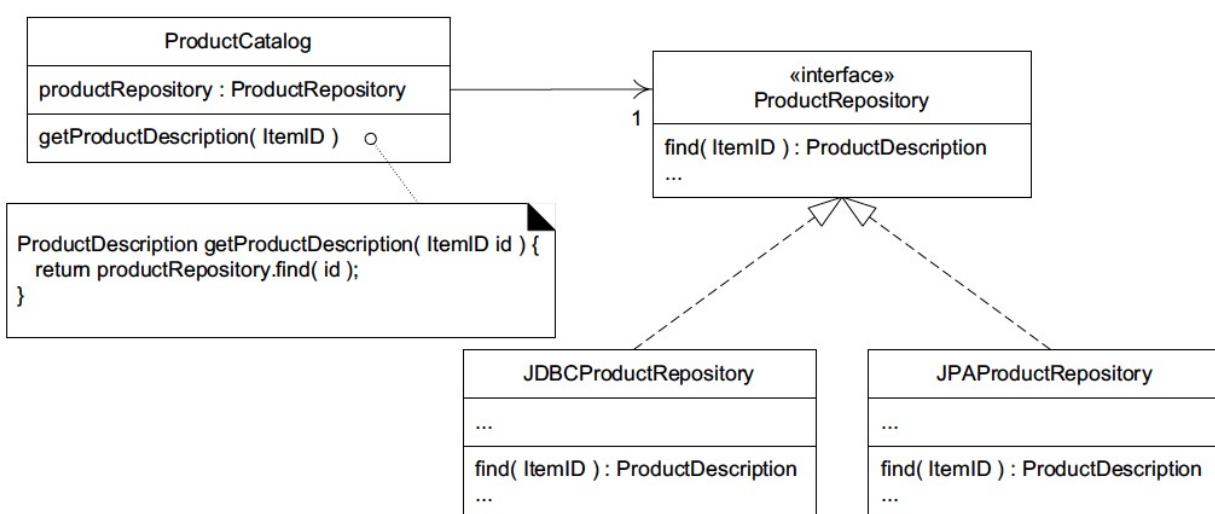
Soluzione

Per ogni oggetto persistente si utilizzi un oggetto che fornisca l'illusione di una collezione di oggetti di quel particolare tipo

L'accesso avviene mediante un'interfaccia che definisce operazioni di inserimento, cancellazione...

15

Esempio di Repository



Il pattern Repository può essere usato anche per oggetti persistenti aggregati, ad esempio la Sale



16

Design pattern

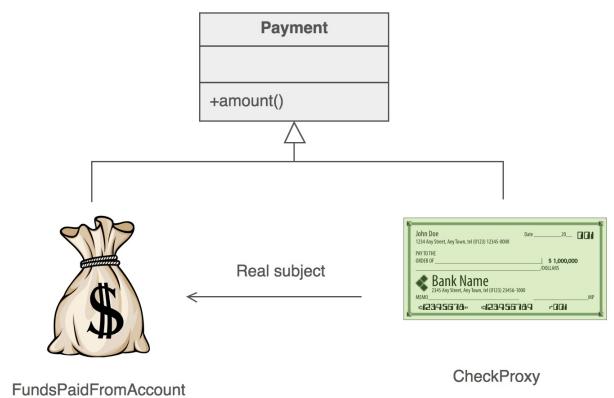


- Decorator
- Repository
- **Proxy**
- Convert Exception
- Abstract Factory
- Template & Factory method
- State

17

Il problema di fornire un surrogato di un oggetto

- Caricare tutte le foto di un album fotografico assieme comporta costi significativi!
- Le vendite tramite POS devono continuare anche se i servizi remoti non sono disponibili
- Un assegno è un surrogato per i soldi su un conto corrente
- Come fornire un surrogato di un oggetto?



Il pattern Proxy

Problema

L'accesso diretto a un oggetto soggetto non è desiderato o possibile. Che cosa fare?

Soluzione

Aggiungere un livello di indirezione con un oggetto surrogato, che implementa la stessa interfaccia del soggetto ed è responsabile dell'accesso allo stesso

Usi tipici

Proxy remoto, virtuale, di protezione, intelligente

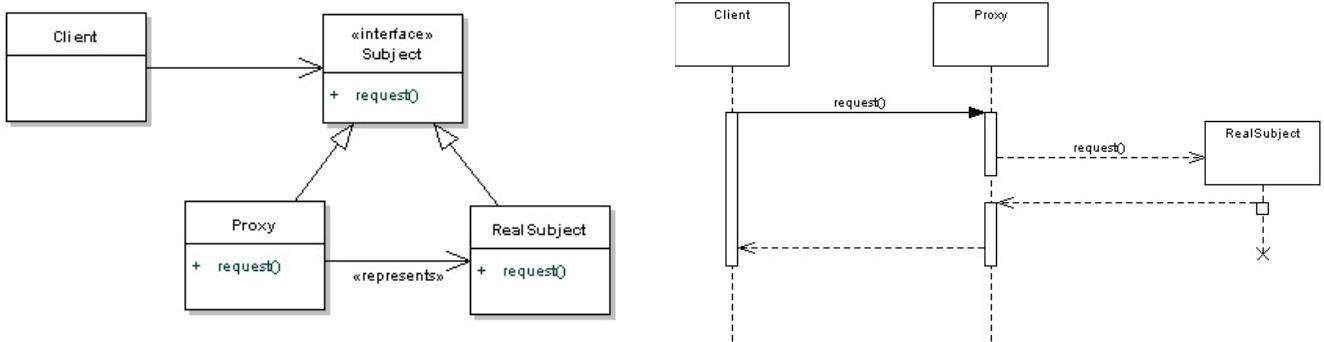
19

Applicabilità

- **Proxy remoto:** fornisce un rappresentante locale per un oggetto esterno al sistema
- **Proxy virtuale:** gestisce la creazione su richiesta di oggetti costosi
- **Proxy di protezione:** controlla l'accesso al soggetto
- **Riferimento intelligente:** consente l'esecuzione di attività aggiuntive quando si accede al soggetto

20

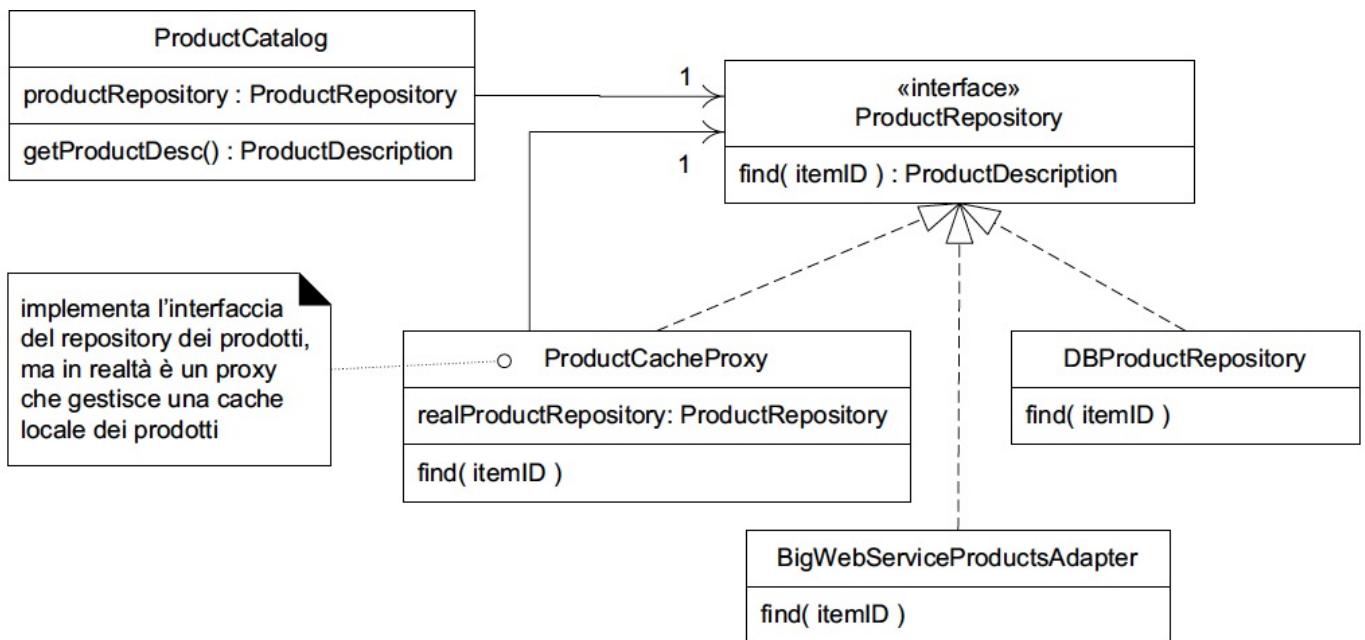
Il pattern Proxy in UML



- **Proxy:** ha la stessa interfaccia di **RealSubject**, mantiene un riferimento a **RealSubject** e controlla il suo accesso
- **Subject:** definisce l'interfaccia comune
- **RealSubject:** è l'oggetto soggetto

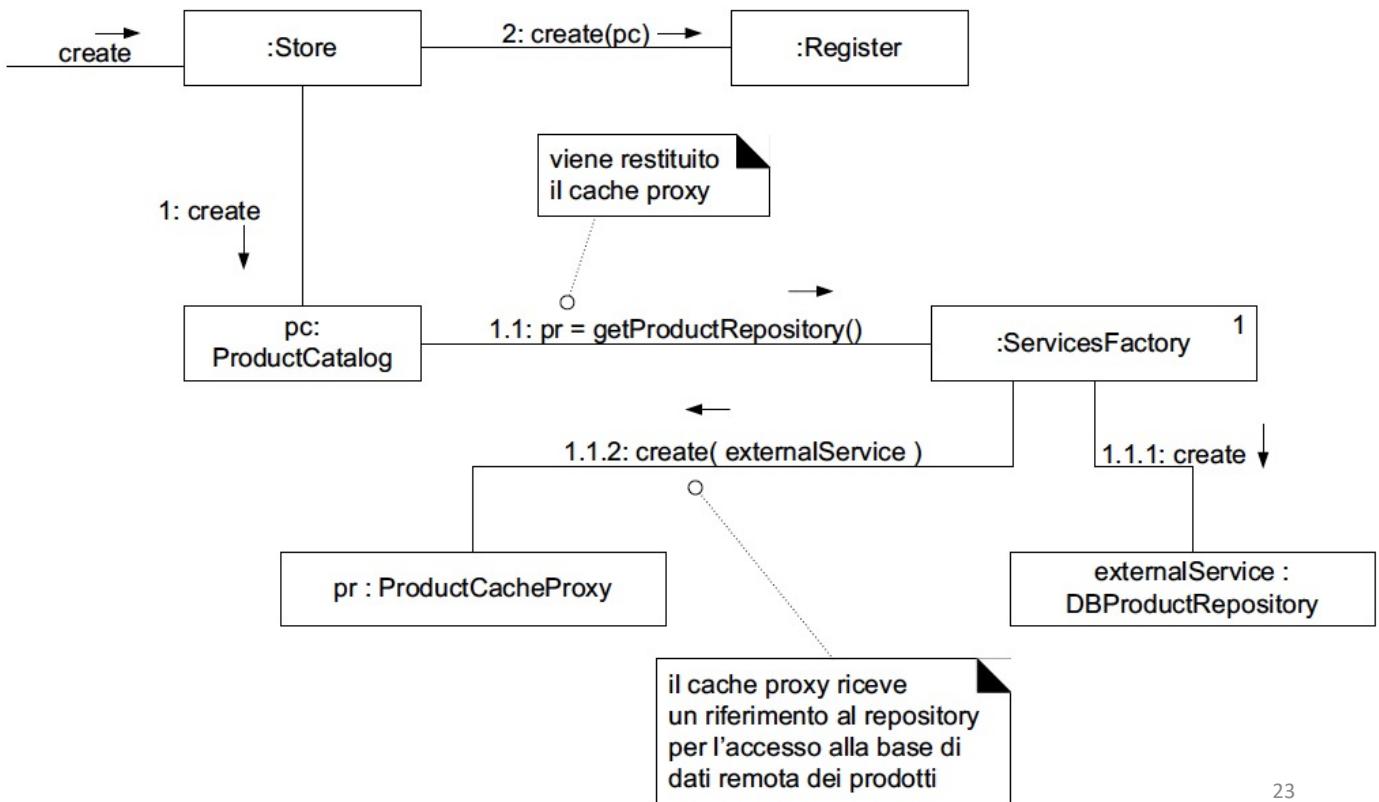
21

Esempio di Proxy



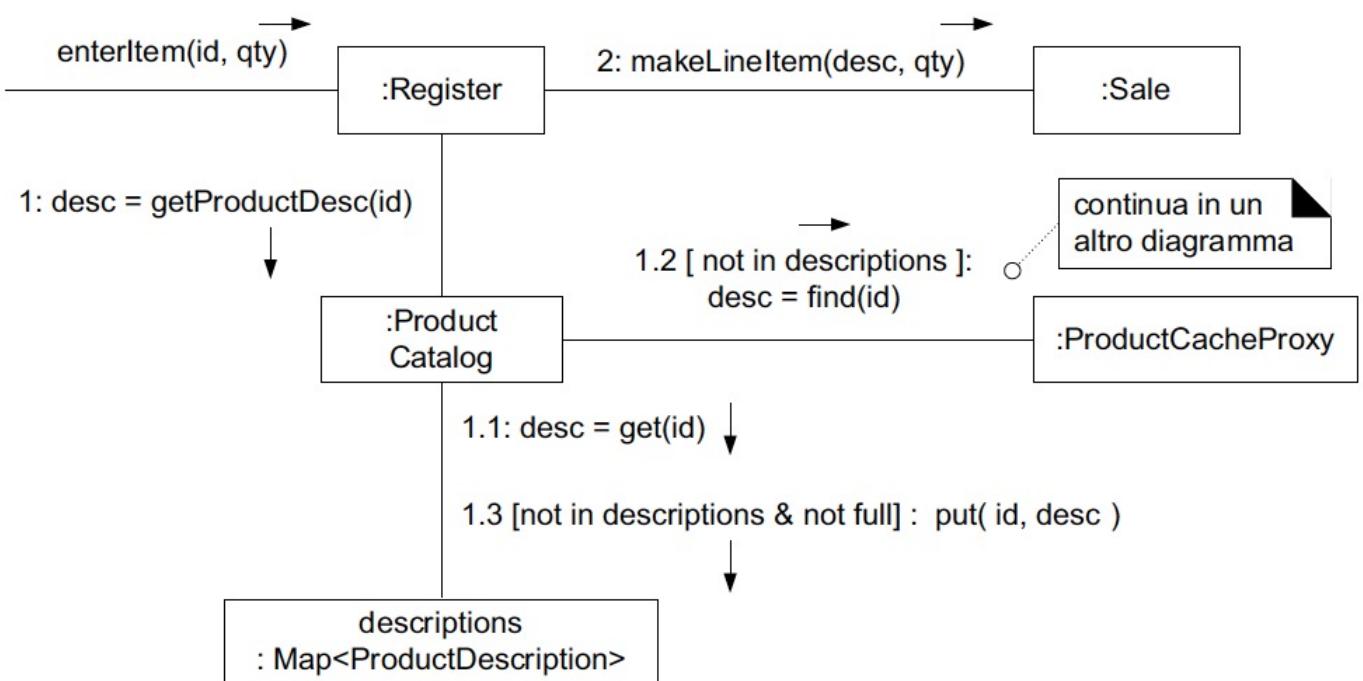
22

Esempio di creazione di Proxy



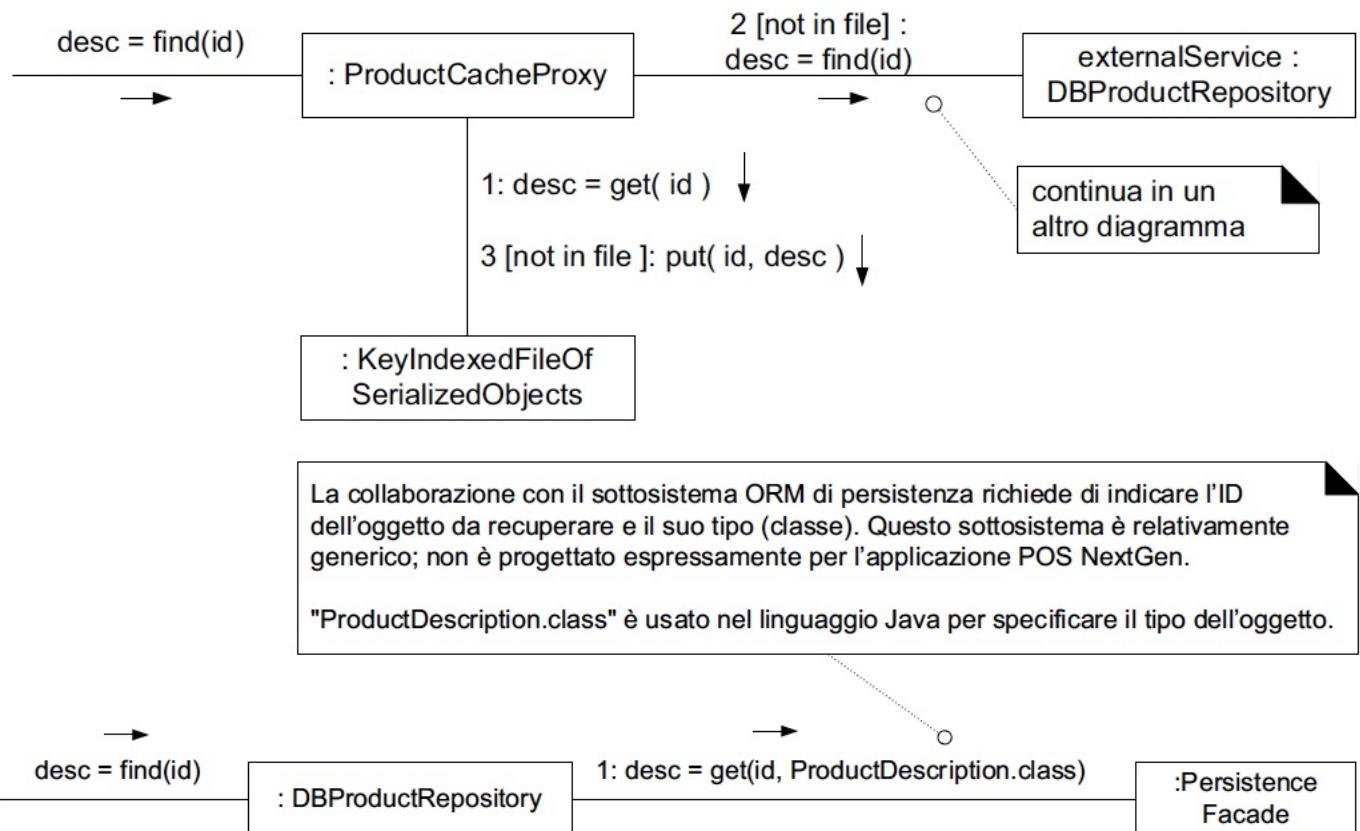
23

Esempio di Proxy (enterItem)



24

Esempio di Proxy (enterItem)

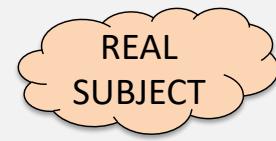
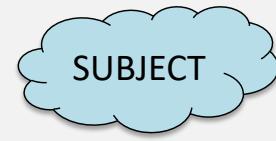


Conseguenze

PRO	CONTRO
<ul style="list-style-type: none"> Introduce un livello di indirezione nell'accesso a un oggetto da usare a seconda del bisogno Permette copy-on-write: posticipa la copia di un oggetto a quando questo è realmente modificato 	Nessuno!

Il pattern Proxy in codice JAVA

```
public interface Image {  
    public void displayImage();  
}  
  
public class ReallImage implements Image {  
    private String filename = null;  
    public ReallImage(final String filename) {  
        this.filename = filename;  
        loadImageFromDisk();  
    }  
  
    private void loadImageFromDisk() {  
        System.out.println("Loading " + filename);  
    }  
  
    public void displayImage() {  
        System.out.println("Displaying " + filename);  
    }  
}
```



27

```
public class ProxyImage implements Image {  
    private ReallImage image = null;  
    private String filename = null;  
  
    public ProxyImage(final String filename) {  
        this.filename = filename;  
    }  
  
    public void displayImage() {  
        if (image == null) {  
            image = new ReallImage(filename);  
        }  
        image.displayImage();  
    }  
}  
  
public class ProxyExample {  
    public static void main(final String[] arguments) {  
        final Image image1 = new ProxyImage("HiRes_10MB_Photo1");  
        final Image image2 = new ProxyImage("HiRes_10MB_Photo2");  
  
        image1.displayImage(); // loading necessary  
        image1.displayImage(); // loading unnecessary  
        image2.displayImage(); // loading necessary  
        image2.displayImage(); // loading unnecessary  
        image1.displayImage(); // loading unnecessary  
    }  
}
```



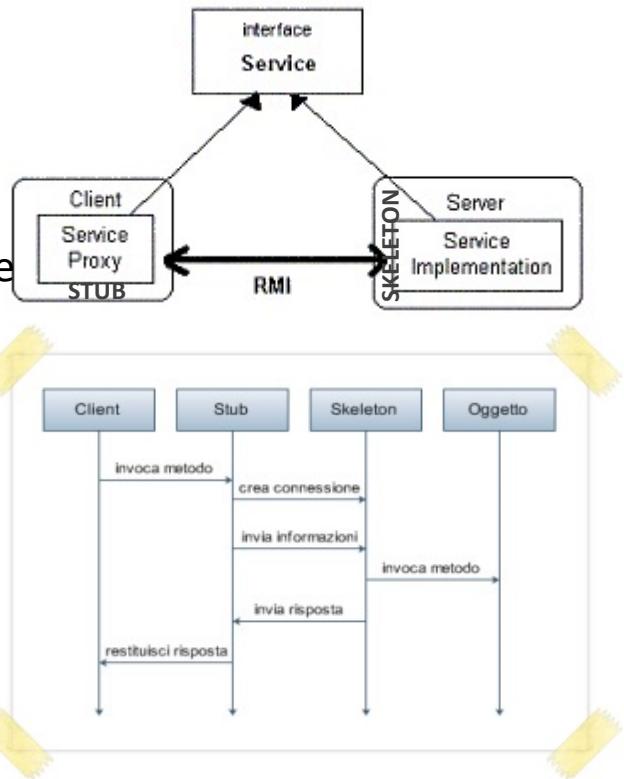
Remote method invocation (RMI)

- È un modo nativo che Java offre per utilizzare oggetti remoti
- Consente di invocare un metodo di un oggetto remoto come se tale oggetto fosse locale

➤ Favorisce omogeneità!



- Il Client conosce soltanto l'interfaccia del servizio
- L'implementazione del servizio è disponibile sul Server
- **Lo stub è un proxy del servizio!**



29

Pattern correlati

Adapter fornisce a un oggetto una nuova interfaccia

Proxy fornisce la stessa interfaccia dell'oggetto rappresentato

Decorator aggiunge una o più funzionalità a un oggetto e può essere usato ricorsivamente

Proxy controlla l'accesso alle funzionalità di un singolo oggetto

30

Design pattern



- Decorator
- Repository
- Proxy
- **Convert Exception**
- Abstract Factory
- Template & Factory method
- State

31

Il problema della gestione del fallimento

- **Fallimento:** mancata erogazione di un servizio provocata da un errore
- POS NextGen non riesce a fornire il servizio di informazioni sui prodotti
- Un approccio alla gestione del fallimento è quella di sollevare *eccezioni*
- Ad es. si solleva un'eccezione, `java.sql.SQLException`, all'interno del sottosistema di persistenza del POS
- **Va sollevata una SQLException per tutto il percorso fino allo strato di presentazione?**



32

Il pattern Convert Exception

Problema

Come gestire le eccezioni?

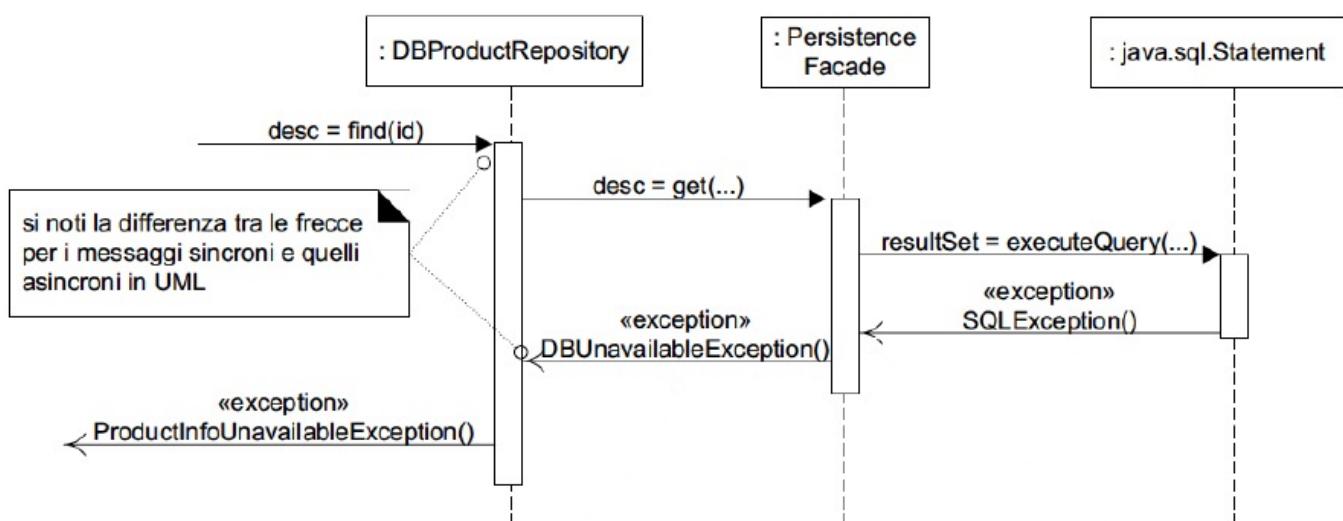
Soluzione

Si converte una eccezione di livello basso in una eccezione significativa a livello di sottosistema

L'eccezione di livello più alto avvolge l'eccezione di livello basso, aggiungendo informazioni per rendere l'eccezione significativa nel contesto superiore

33

Esempio di Conversion Exception



1. Il sottosistema di persistenza cattura una SQLException
2. Solleva una DBUnavailableException che avvolge la SQLException
3. Il DBProductRepository cattura la DBUnavailableException
4. Solleva ProductInfoUnavailableException che avvolge DBUnavailableException

34

Il problema della gestione degli errori

Errore: una manifestazione del guasto nel sistema in esecuzione

Problema

Come gestire gli errori?

Soluzione (Error dialog/logger)

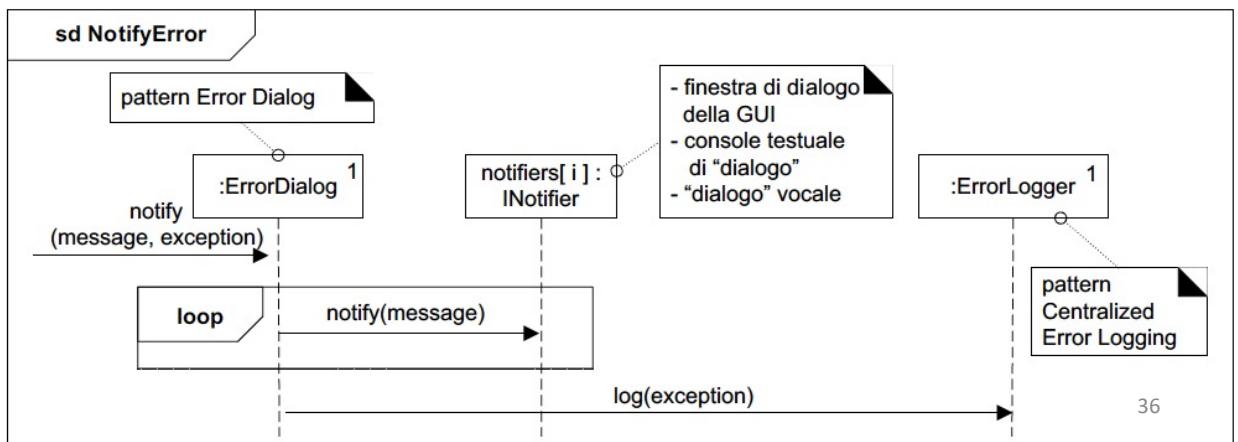
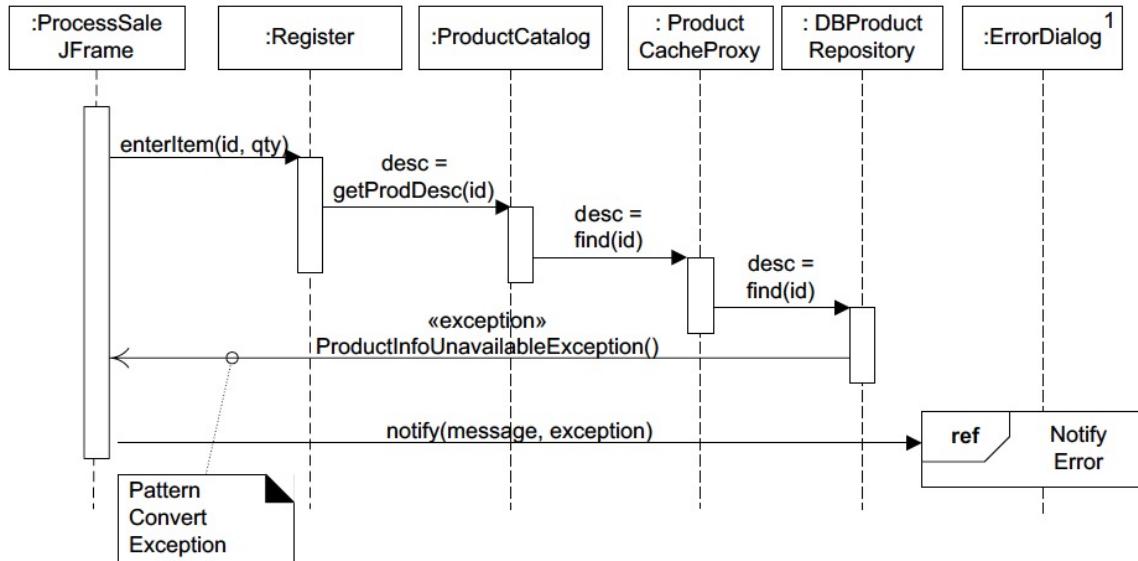


Si utilizzi un oggetto standard, non appartenente alla UI e accessibile attraverso un Singleton, per comunicare gli errori agli utenti

Si utilizzi un altro oggetto standard per il logging degli errori, a cui si accede attraverso un Singleton, e si riportino a esso tutte le eccezioni

35

Esempio di gestione eccezioni



36

Conseguenze

PRO	CONTRO
<ul style="list-style-type: none">• Stile coerente di report degli errori• Controllo centralizzato della notifica degli errori• Minore consumo di prestazioni; se viene usata una risorsa costosa è facile da inserire nella cache• E' facile realizzare protected variations rispetto alle modifiche del meccanismo di output	Nessuno!

37

Design pattern



- Decorator
- Repository
- Proxy
- Convert Exception
- **Abstract Factory**
- Template & Factory method
- State

38

Il problema di creare una famiglia di oggetti correlati

- Presse meccaniche possono essere configurate per lo stampaggio di parti della carrozzeria di un auto
- Un editor grafico può dovere supportare differenti standard look-and-feel per le interfacce grafiche
- L'applicazione POS deve caricare i driver opportuni per potere funzionare con hardware differenti
- Come creare una famiglia di oggetti correlati?



Il pattern Abstract Factory

Problema

Come creare famiglie di classi correlate che implementano un'interfaccia comune?

Soluzione

Definire un'interfaccia factory (la factory astratta)

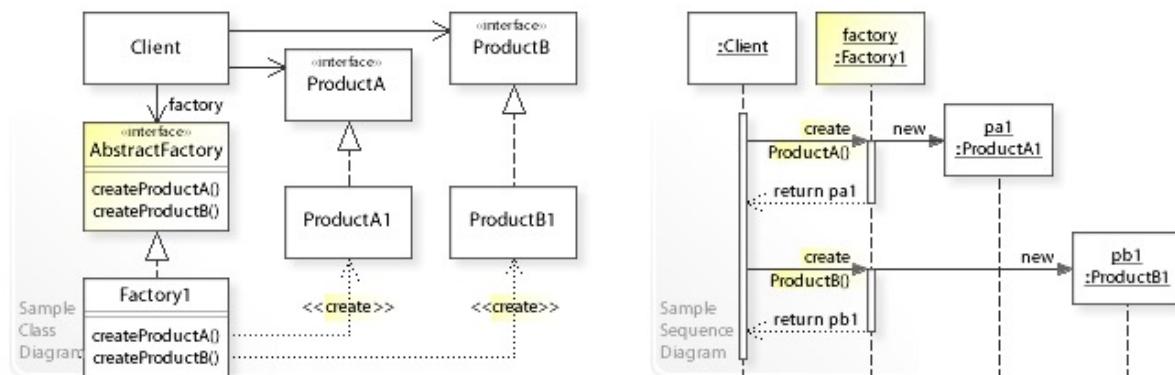
Definire una classe factory concreta per ciascuna famiglia di elementi da creare.

Altri usi

Per rendere un sistema indipendente dalla modalità di presentazione

Per famiglie di oggetti correlati da usare assieme

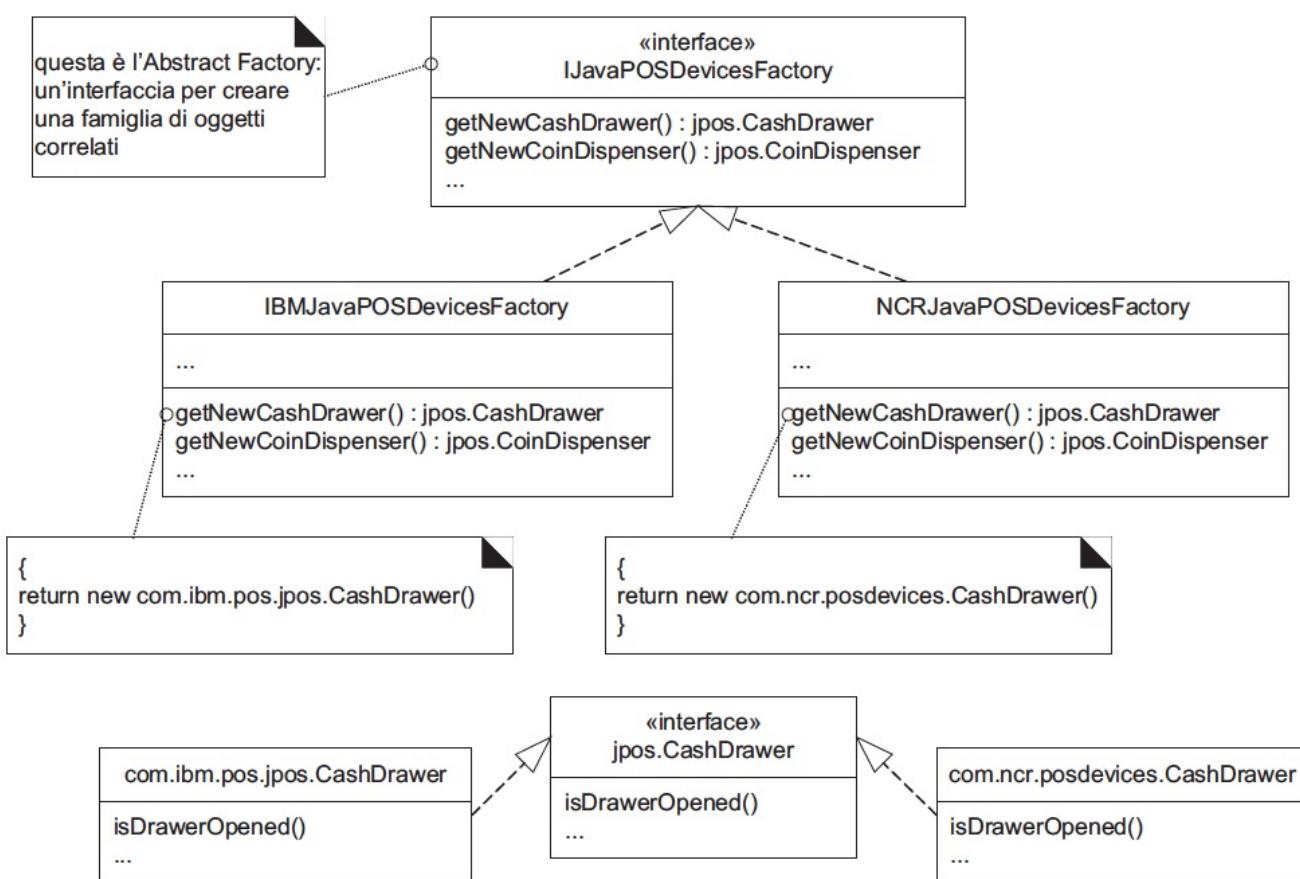
Abstract Factory in UML



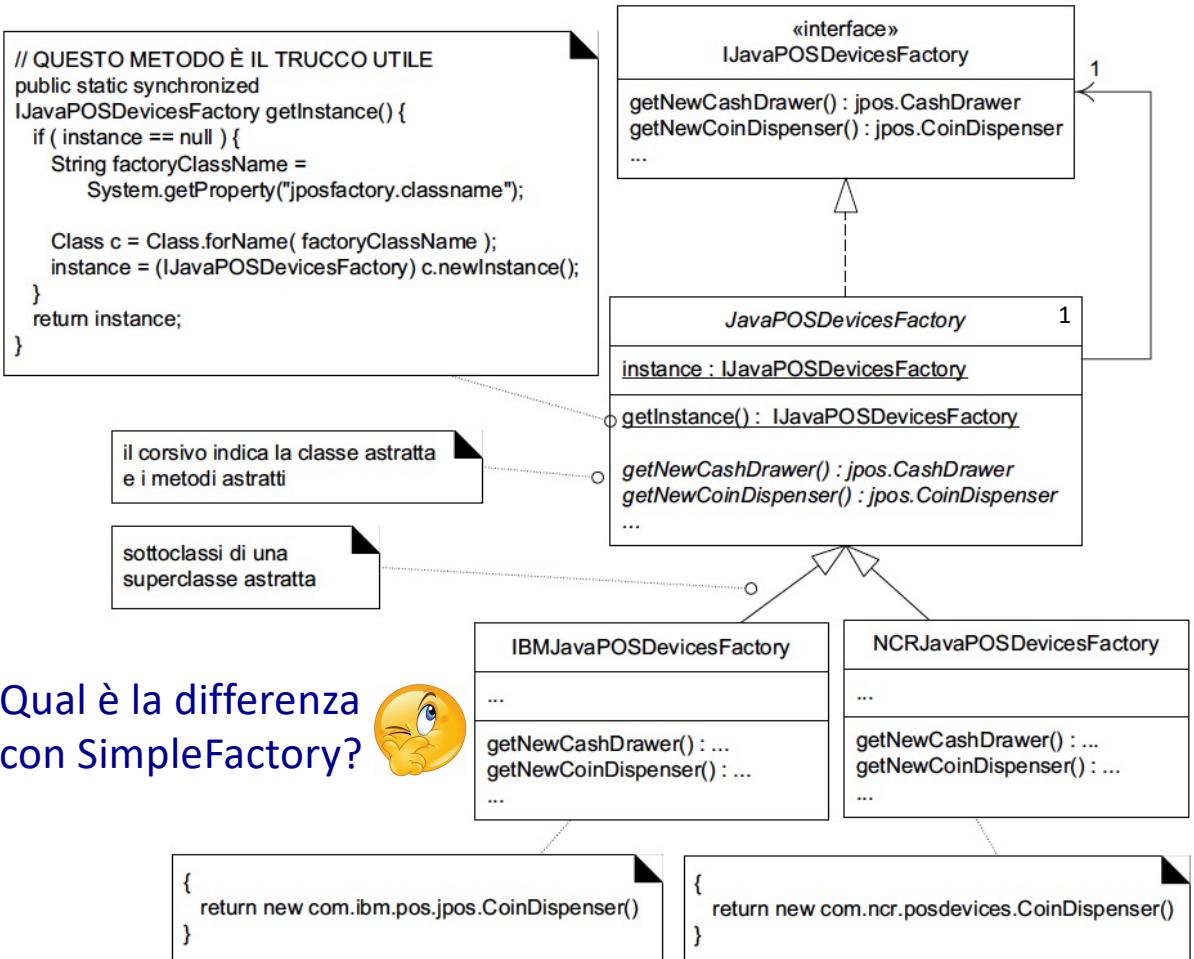
- **AbstractFactory:** è l'interfaccia per creare oggetti prodotto
- **ProductA, ProductB:** sono interfacce per certe tipologie di prodotti
- **Factory1:** crea gli oggetti prodotto concreti
- **ProductA1, ProductB1:** oggetti prodotto creati da Factory1
- **Client:** utilizza soltanto le interfacce del Pattern

41

Esempio di Abstract Factory



Varianti di Abstract Factory



Qual è la differenza con SimpleFactory?



43

Conseguenze

PRO	CONTRO
<ul style="list-style-type: none"> Rende i client indipendenti dalle classi effettivamente utilizzati Consente di cambiare in modo semplice la famiglia di prodotti utilizzata Promuove la coerenza nell'uso dei prodotti 	Estendere la factory per supportare nuove tipologie di prodotti può essere difficile

44

Abstract Factory in codice JAVA

```
public interface IGUIFactory {  
    public IButton createButton();  
}  
  
public class WinFactory implements IGUIFactory {  
    @Override public IButton createButton() {  
        return new WinButton();  
    }  
}  
  
public class OSXFactory implements IGUIFactory {  
    @Override public IButton createButton() {  
        return new OSXButton();  
    }  
}  
  
public interface IButton {  
    void paint();  
}
```



45

```
public class WinButton implements IButton {  
    @Override public void paint() {  
        System.out.println("WinButton");  
    }  
}  
  
public class OSXButton implements IButton {  
    @Override public void paint() {  
        System.out.println("OSXButton");  
    }  
}  
  
public class Main {  
    public static void main(void) {  
        IGUIFactory factory = new WinFactory();  
        // IGUIFactory factory = new OSXFactory();  
        final IButton button = factory.createButton();  
        button.paint();  
    }  
}
```



46

Design pattern

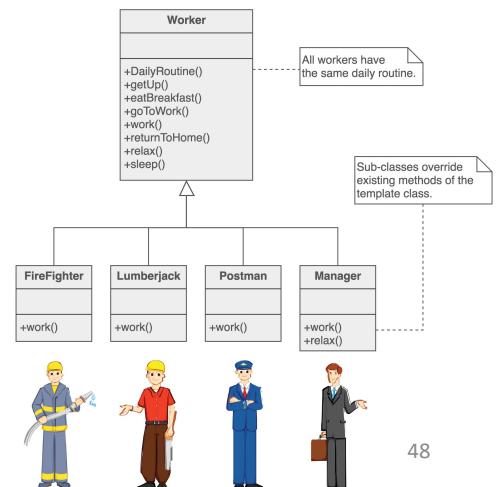


- Decorator
- Repository
- Proxy
- Convert Exception
- Abstract Factory
- **Template & Factory method**
- State

47

Il problema di configurare un modello

- Case a schiera sono costruite a partire da un numero limitato di piante e diverse varianti ciascuna
- Un compilatore va configurato per iPhone o Android
- La routine di ogni lavoratore è costituita da attività uguali per tutte e alcune attività differenti
- **Come definire un algoritmo con parti varianti?**



Il pattern Template

Problema

Come definire un algoritmo con parti varianti di modo che le parti invarianti siano scritte una sola volta e restino effettivamente inalterate?

Soluzione

Definire operazioni astratte primitive in sottoclassi per le parti varianti dell'algoritmo

Definire un metodo modello (final) per le parti invarianti dell'algoritmo che invoca le operazioni primitive

Noto anche come *inversione del controllo*

49

Il principio di Hollywood

Don't call us, we'll call you

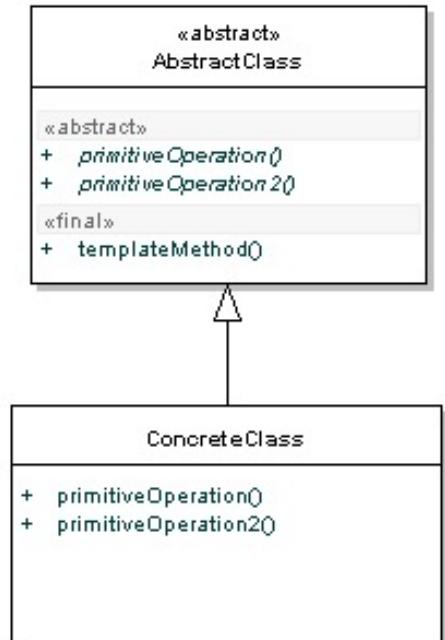


Usato dai plugin per estendere le funzionalità di un programma!

50

Il pattern Template in UML

- **AbstractClass:** implementa un metodo template (final) e dei metodi “hook”; contiene:
 - Metodi template
 - Metodi astratti (primitive)
 - Metodi gancio (hook)
 - Metodi concreti
 - Factory method
- **ConcreteClass:** implementa le operazioni primitive del template



51

Conseguenze

PRO	CONTRO
<ul style="list-style-type: none">• Dà alla classe padre il controllo di cosa possa essere esteso dalla classe figlia• Porta a fattor comune comportamenti analoghi di più classi• Evita la duplicazione di codice	Nessuno!

...se mi accorgo che ho due classi quasi identiche che lavorano con la stessa logica



52

```

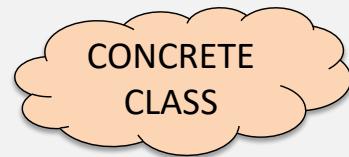
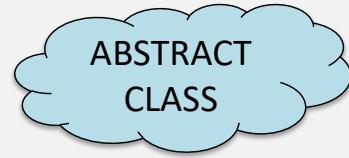
public abstract class CrossCompiler {
    public final void crossCompile() {
        collectSource();
        compileToTarget();
    }
    protected void collectSource() { };
    protected void compileToTarget() { };
}

public class IPhoneCompiler extends CrossCompiler {
    protected void collectSource() {
        //anything specific to this class
    }
    protected void compileToTarget() {
        //iphone specific compilation
    }
}

public class Client {
    public static void main(String[] args) {
        CrossCompiler iphone = new IPhoneCompiler();
        iphone.crossCompile();
    }
}

```

Esempio di Template



53

Pattern correlati

Strategy usa la delega per potere variare *l'intero algoritmo*

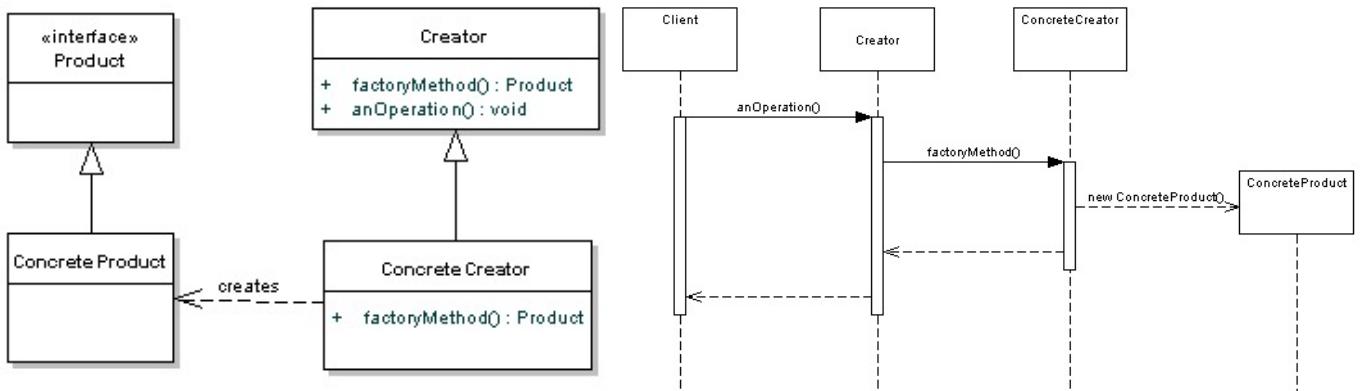
Template usa l'ereditarietà per potere variare *alcune parti* di un algoritmo

Il pattern Factory method può essere usato assieme al pattern Template... [come fare se il Client conosce solo a run-time la classe concreta da creare?](#)



54

Il pattern Factory Method in UML



- **Creator**: definisce il metodo che restituisce un oggetto di tipo **Product**
- **ConcreteCreator**: sovrascrive/estende il metodo **factory** in modo da creare uno specifico **ConcreteProduct**
- **Product**: definisce l'interfaccia degli oggetti creati dal metodo **Factory**
- **ConcreteProduct**: implementa l'interfaccia di **Product**

55

Factory Method in codice JAVA

```

public abstract class MazeGame {
    private final List<Room> rooms = new ArrayList<>();
    public MazeGame() {
        Room room1 = makeRoom();
        Room room2 = makeRoom();
        room1.connect(room2);
        rooms.add(room1);
        rooms.add(room2);
    }
    abstract protected Room makeRoom();
}
  
```



Factory method come
metodo di gancio

```

public class MagicMazeGame extends MazeGame {
    @Override protected Room makeRoom() {
        return new MagicRoom();
    }
}
  
```



56

Design pattern

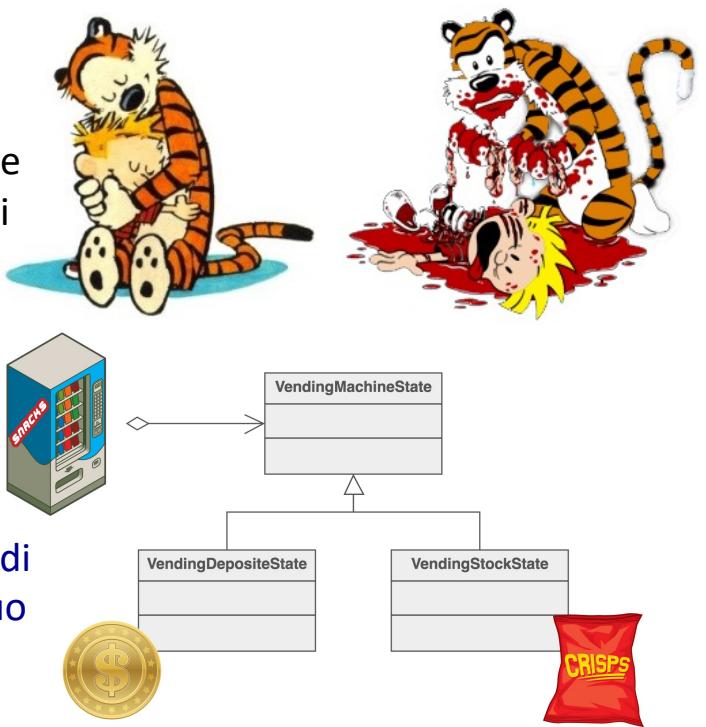


- Decorator
- Repository
- Proxy
- Convert Exception
- Abstract Factory
- Template & Factory method
- **State**

57

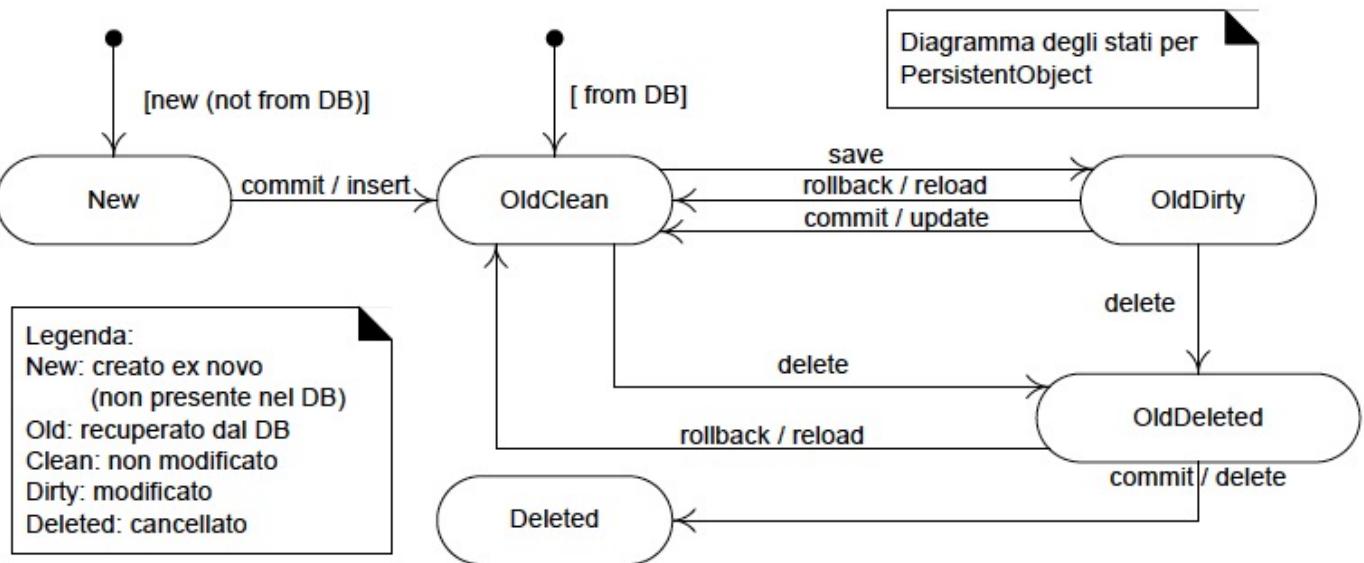
Il problema di cambiare la propria reazione a seconda dello stato

- Una tigre reagisce in maniera differente se affamata
- Schiacciare il pulsante play/pause del telecomando origina reazioni differenti
- Un distributore automatico si comporta in modo diverso a seconda della disponibilità di resto, prodotti, ecc.
- Come fare se il comportamento di un oggetto è condizionato dal suo stato interno?



58

Es. la classe degli oggetti persistenti



Supponiamo che...

gli oggetti persistenti possono essere inseriti, eliminati o modificati
per aggiornare la base dati è necessario eseguire un'operazione commit

59

Una possibile soluzione... ma...

```
public void commit() {  
    ...  
    switch ( state ) {  
        case OLD_DIRTY:  
            // ...  
            break;  
        case OLD_CLEAN:  
            // ...  
            break;  
        ...  
    }  
    ...  
}  
  
public void rollback() {  
    ...  
    switch ( state ) {  
        case OLD_DIRTY:  
            // ...  
            break;  
        case OLD_CLEAN:  
            // ...  
            break;  
        ...  
    }  
    ...  
}
```

Questo viola open-closed principle!



60

Il pattern State

Problema

Il comportamento di un oggetto dipende dal suo stato.
C'è un'alternativa alla logica condizionale?

Soluzione

Creare una classe stato per ogni stato, che implementa un'interfaccia comune

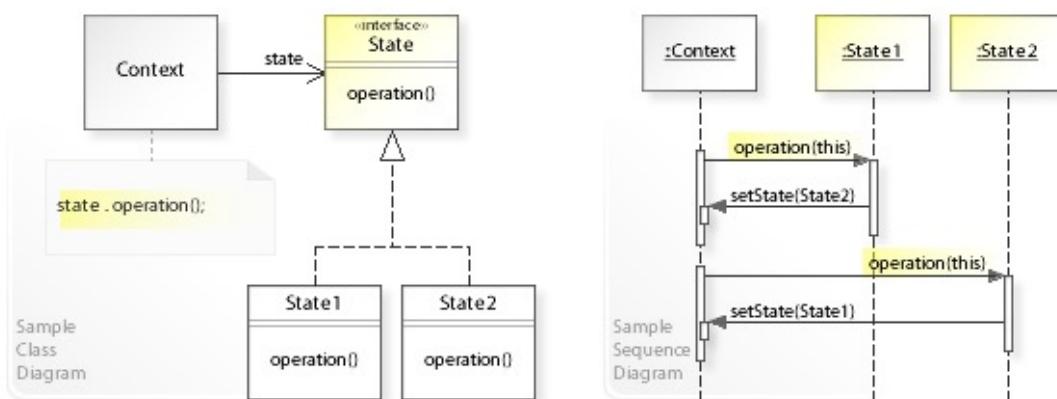
Delegare le operazioni che dipendono dallo stato all'oggetto che rappresenta il proprio stato corrente

Altri usi

Per eliminare blocchi di codice di scelte condizionali

61

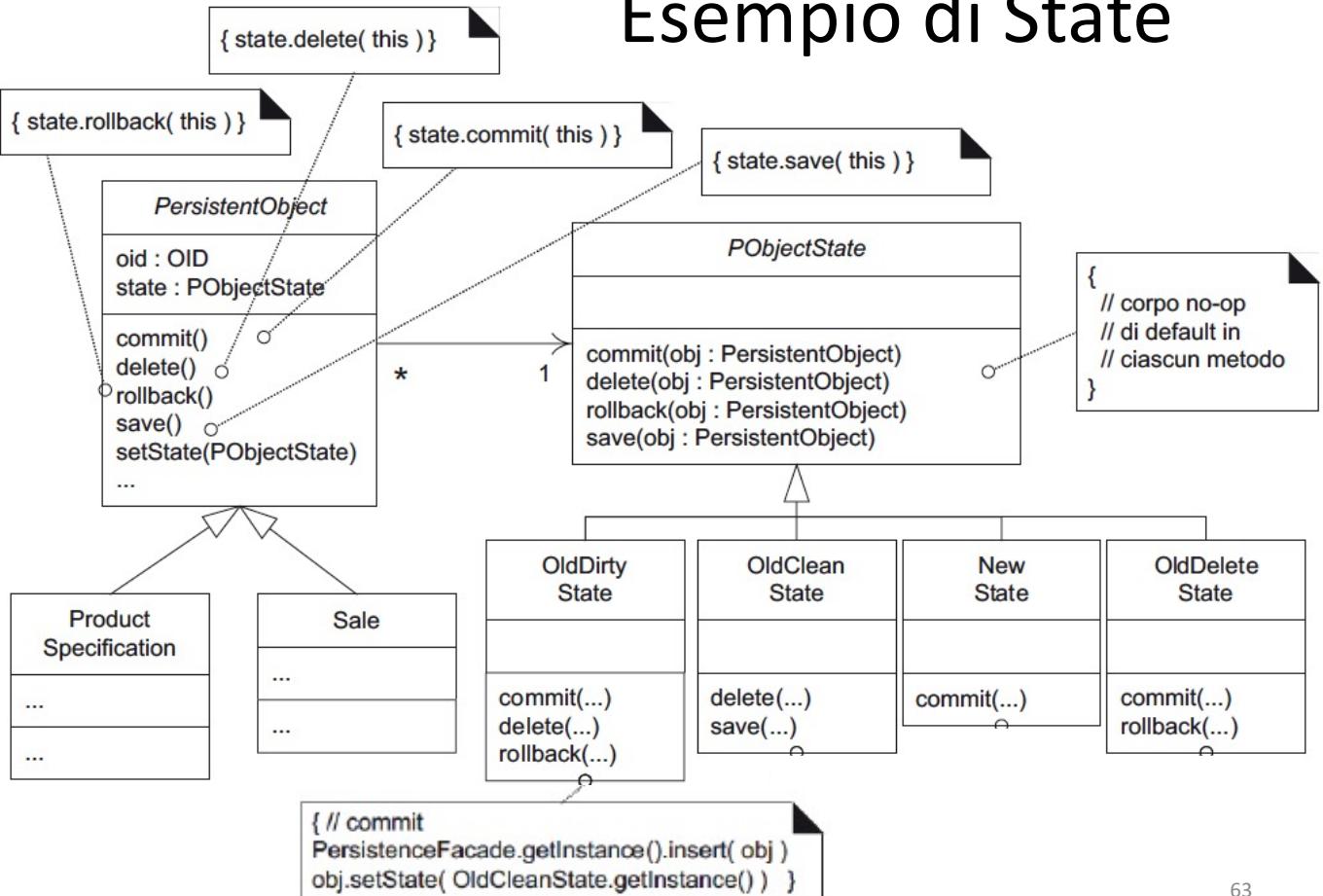
Il pattern State in UML



- **Context:** contiene un'istanza dello stato corrente
- **State:** definisce un'interfaccia per lo stato
- **State1,2:** implementa il comportamento associato a un particolare stato

62

Esempio di State



63

Conseguenze

PRO	CONTRO
<ul style="list-style-type: none"> Localizza il comportamento specifico di uno stato senza l'uso di codice switch Rende le transizioni di stato esplicite Permette la condivisione degli oggetti State Fornisce protected variation rispetto all'aggiunta di stati 	Aumenta il numero di classi

64

Il pattern State in codice JAVA

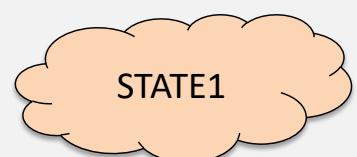
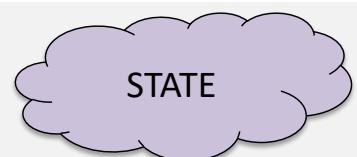
```
public class MP3PlayerContext {  
    private State state;  
  
    private MP3PlayerContext(State state) {  
        this.state = state;  
    }  
  
    public void play() {  
        state.pressPlay(this);  
    }  
  
    public void setState(State state) {  
        this.state = state;  
    }  
  
    public State getState() {  
        return state;  
    }  
}
```



65

Il pattern State in codice JAVA (cont.)

```
private interface State {  
    public void pressPlay(MP3PlayerContext context);  
}  
  
public class StandbyState implements State {  
    public void pressPlay(MP3PlayerContext context) {  
        // do actions for pausing  
        context.setState(new PlayingState());  
    }  
}  
  
public class PlayingState implements State {  
    public void pressPlay(MP3PlayerContext context) {  
        // do actions for playing  
        context.setState(new StandbyState());  
    }  
}
```



66

State vs Strategy

STATE	STRATEGY
Context passa (solitamente) a State un riferimento a sé stesso	Context passa (meno frequentemente) a Strategy un riferimento a sé stesso
A oggetti State è permesso rimpiazzare loro stessi	A oggetti Strategy non è permesso rimpiazzarsi a vicenda
Oggetti State possono essere creati dallo stesso Context	Oggetti Strategy sono passati al Context come parametri
Un oggetto State implementa ogni metodo di Context che dipende dallo stato	Un oggetto Strategy gestisce un singolo specifico compito

67

Domande?



68