



DESIGN PATTERN COMPORTAMENTALI


INGEGNERIA DEL SOFTWARE - DESIGN PATTERN

Università degli Studi di Perugia
Dipartimento di Matematica e Informatica
Corso di Laurea in Informatica



INTRODUZIONE



Relazioni tra		Campo di applicazione		
		Creational (5)	Structural (7)	Behavioral (11)
	Class	Factory method	Adapter (Class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter(Object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor	
Architetturali				
Model view controller				

INTRODUZIONE



- Scopo dei *design pattern* comportamentali
 - In che modo un **oggetto** svolge la sua **funzione**?
 - In che modo diversi oggetti **collaborano** tra loro?

CHAIN OF RESPONSIBILITY



○Scopo

- Separare** il mittente di una richiesta dal destinatario, in modo da consentire a più di un oggetto di gestire la richiesta. Gli oggetti destinatari vengono messi in catena e la richiesta viene trasmessa dentro questa catena fino a trovare un oggetto che la gestisca.



CHAIN OF RESPONSIBILITY



- Motivazione

Non si conosce a priori quale oggetto è in grado di **gestire** una determinata richiesta (perché effettivamente è sconosciuto staticamente o perché l'insieme degli oggetti in grado di gestire tali richieste cambia dinamicamente a runtime).

CHAIN OF RESPONSIBILITY



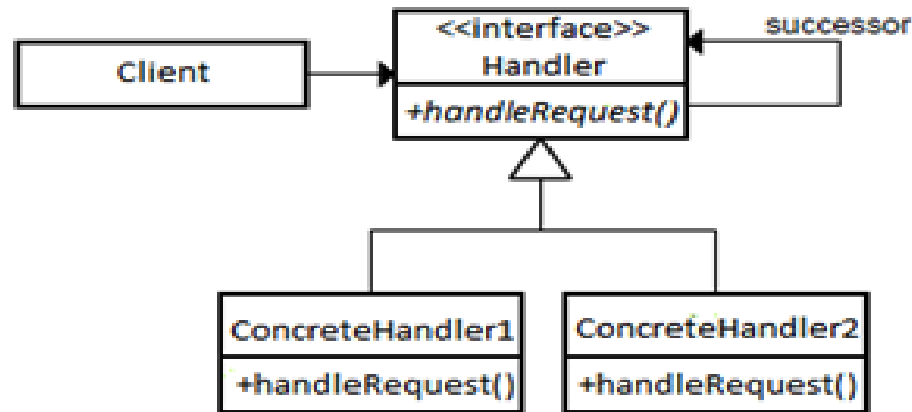
○ Applicabilità

- Uno o più oggetti possono gestire una richiesta ma gli handler non sono conosciuti a priori.
- Inviare una richiesta ad uno di più oggetti senza specificare esplicitamente il destinatario.
- L'insieme degli oggetti che può gestire una richiesta è specificato dinamicamente

CHAIN OF RESPONSIBILITY



○ Struttura



```
public void handleRequest(Handler  
next){
```

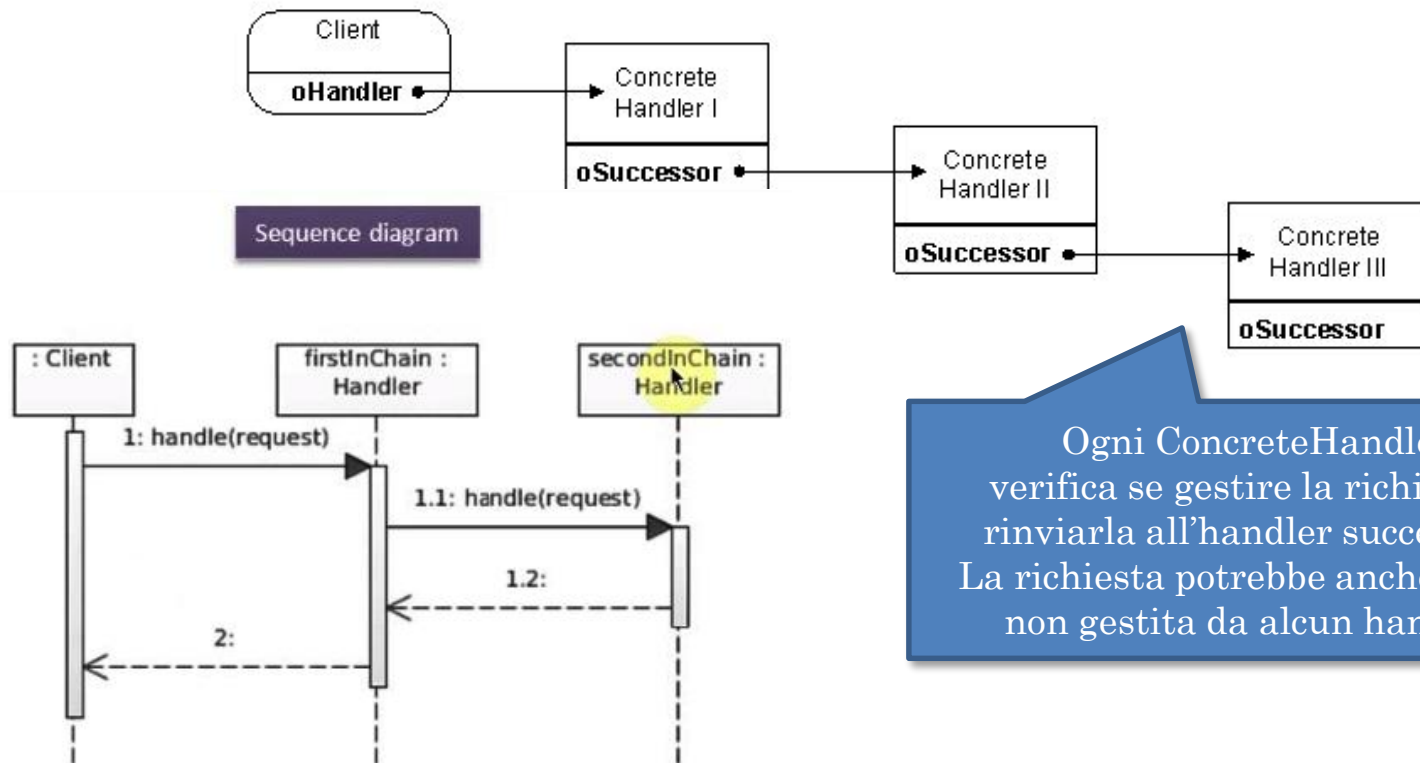
```
    if( ..... )  
        // gestire la  
        richiesta  
    else if(next != null)  
        //la richiesta viene  
        inviata al  
        handler  
        //successivo
```

```
        next.handleRequest();
```

CHAIN OF RESPONSIBILITY



○ Struttura

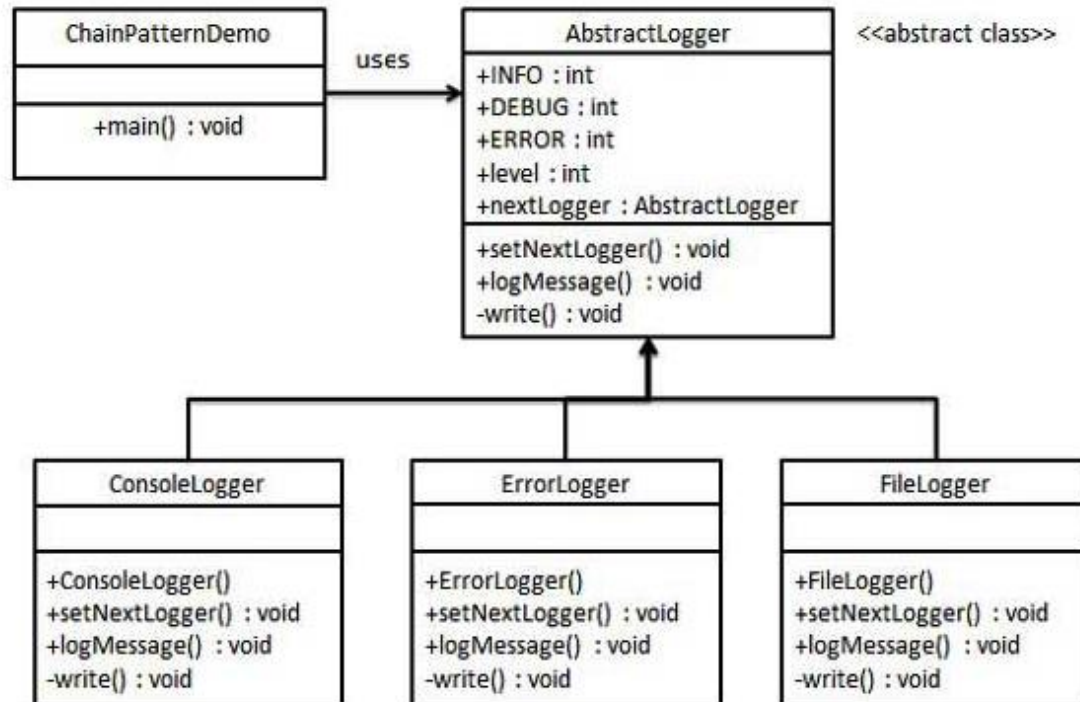


Ogni ConcreteHandler verifica se gestire la richiesta o rinviarla all'handler successivo. La richiesta potrebbe anche essere non gestita da alcun handler.

CHAIN OF RESPONSIBILITY



○ Esempio:



CHAIN OF RESPONSIBILITY



```
public abstract class AbstractLogger {
    public static int INFO = 1;
    public static int DEBUG = 2;
    public static int ERROR = 3;

    protected int level;

    //next element in chain or responsibility
    protected AbstractLogger nextLogger;

    public void setNextLogger(AbstractLogger nextLogger){
        this.nextLogger = nextLogger;
    }

    public void logMessage(int level, String message){
        if(this.level <= level){
            write(message);
        }
        if(nextLogger !=null){
            nextLogger.logMessage(level, message);
        }
    }

    abstract protected void write(String message);
}
```

CHAIN OF RESPONSIBILITY



```
public class ConsoleLogger extends AbstractLogger {  
  
    public ConsoleLogger(int level){  
        this.level = level;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("Standard Console::Logger: " + message);  
    }  
}
```

```
public class ErrorLogger extends AbstractLogger {  
  
    public ErrorLogger(int level){  
        this.level = level;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("Error Console::Logger: " + message);  
    }  
}
```

```
public class FileLogger extends AbstractLogger {  
  
    public FileLogger(int level){  
        this.level = level;  
    }  
  
    @Override  
    protected void write(String message) {  
        System.out.println("File::Logger: " + message);  
    }  
}
```

CHAIN OF RESPONSIBILITY



```
public class ChainPatternDemo {  
  
    private static AbstractLogger getChainOfLoggers(){  
  
        AbstractLogger errorLogger = new ErrorLogger(AbstractLogger.ERROR);  
        AbstractLogger fileLogger = new FileLogger(AbstractLogger.DEBUG);  
        AbstractLogger consoleLogger = new ConsoleLogger(AbstractLogger.INFO);  
  
        errorLogger.setNextLogger(fileLogger);  
        fileLogger.setNextLogger(consoleLogger);  
  
        return errorLogger;  
    }  
  
    public static void main(String[] args) {  
        AbstractLogger loggerChain = getChainOfLoggers();  
  
        loggerChain.logMessage(AbstractLogger.INFO,  
            "This is an information.");  
  
        loggerChain.logMessage(AbstractLogger.DEBUG,  
            "This is an debug level information.");  
  
        loggerChain.logMessage(AbstractLogger.ERROR,  
            "This is an error information.");  
    }  
}
```

Si creano tutti gli handler

Gli handler vengono collegati in una chain

CHAIN OF RESPONSIBILITY



- Usato in Java 1.0 AWT per la propagazione degli eventi:

```
public boolean action(Event event, Object obj) {  
    if (event.target == test_button)  
        doTestButtonAction();  
    else if (event.target == exit_button)  
        doExitButtonAction();  
    else  
        return super.action(event,obj);  
    return true; // Return true to indicate the event has been  
                // handled and should not be propagated further.  
}
```

CHAIN OF RESPONSIBILITY



- In Java 1.1 il modello AWT di eventi è stato modificato usando il pattern **Observer**: ogni oggetto che vuole catturare un evento si registra come listener di tale evento ad un componente. Quando un evento viene scatenato, questo è posizionato sul componente il quale dispaccia l'evento a tutti gli ascoltatori che si sono registrati. Se il componente non ha ascoltatori, l'evento viene scaricato.

CHAIN OF RESPONSIBILITY



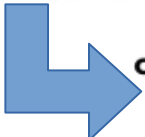
```
import java.awt.*;
import java.awt.event.*;
class finestra {
    public static void main(String[] args) {
        Frame f = new Frame("titolo");
        f.setBounds(20,20,200,150);
        f.addWindowListener(new ascoltatore());
        f.setVisible(true);
    }
}
```

Il componente Frame si registra
all'ascoltatore

```
f.addWindowListener(new
ascoltatore);
```

che cattura e implementa i due eventi:

- windowClosed(...)
- windowClosing(...)



```
class ascoltatore implements WindowListener {
    public void windowClosing(WindowEvent e) {
        e.getWindow().dispose();
    }
    public void windowClosed(WindowEvent e) {
        System.exit(0);
    }
    public void windowOpened(WindowEvent e) {}
    public void windowIconified(WindowEvent e) {}
    public void windowDeiconified(WindowEvent e) {}
    public void windowActivated(WindowEvent e) {}
    public void windowDeactivated(WindowEvent e) {}
}
```

COMMAND



◦ Scopo

- Incapsulare una richiesta in un oggetto, cosicché i *client* sia indipendenti dalle richieste



◦ Motivazione

- Necessità di gestire richieste di cui non si conoscono i particolari
 - *Toolkit* associano ai propri elementi, richieste da eseguire
- Una classe astratta, *Command*, definisce l'interfaccia per eseguire la richiesta
 - La richiesta è un semplice oggetto

COMMAND



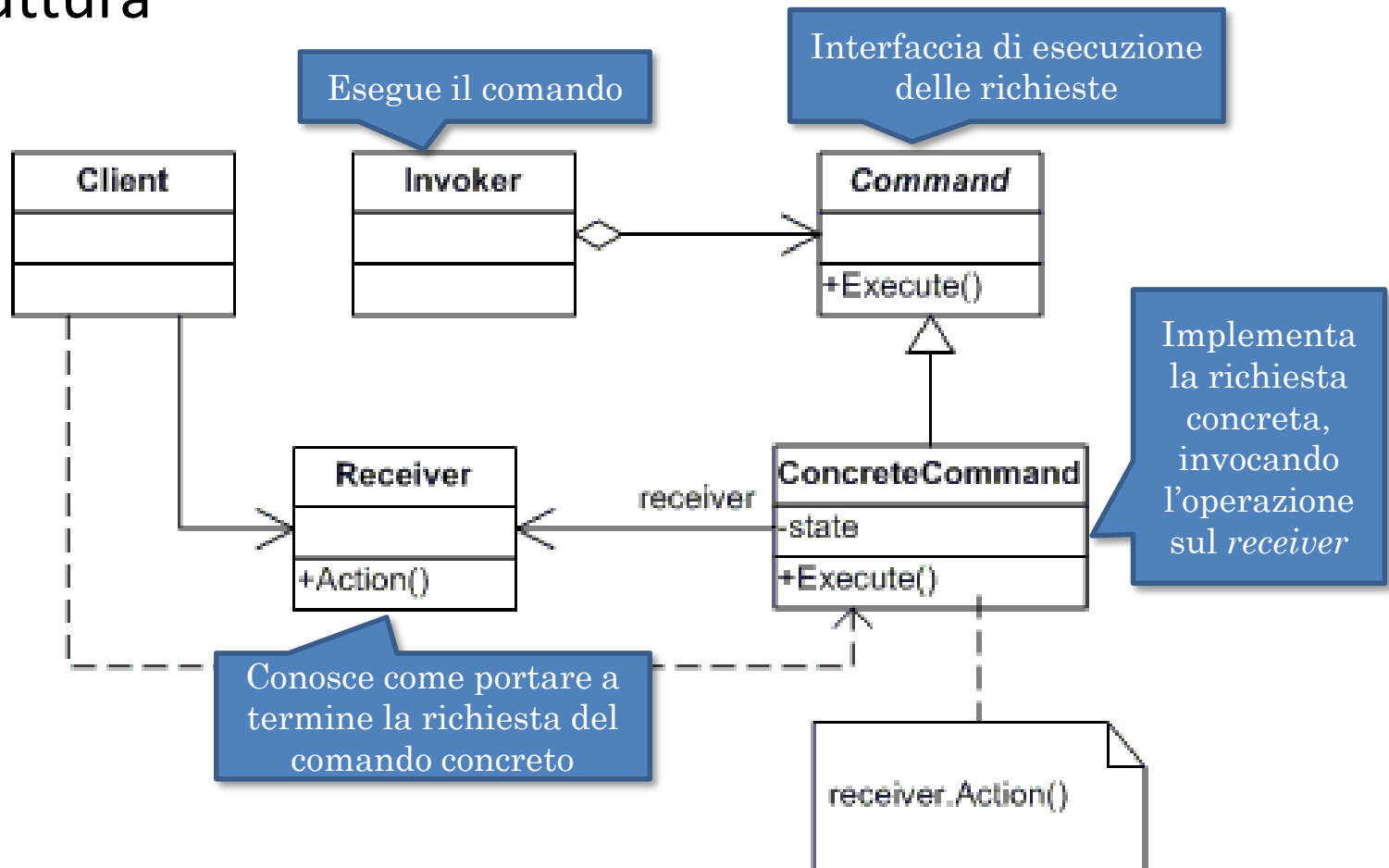
○ Applicabilità

- Parametrizzazione di oggetti sull'azione da eseguire
 - *Callback function*
- Specificare, accodare ed eseguire richieste molteplici volte
- Supporto ad operazione di Undo e Redo
- Supporto a transazione
 - Un comando equivale ad una operazione atomica

COMMAND



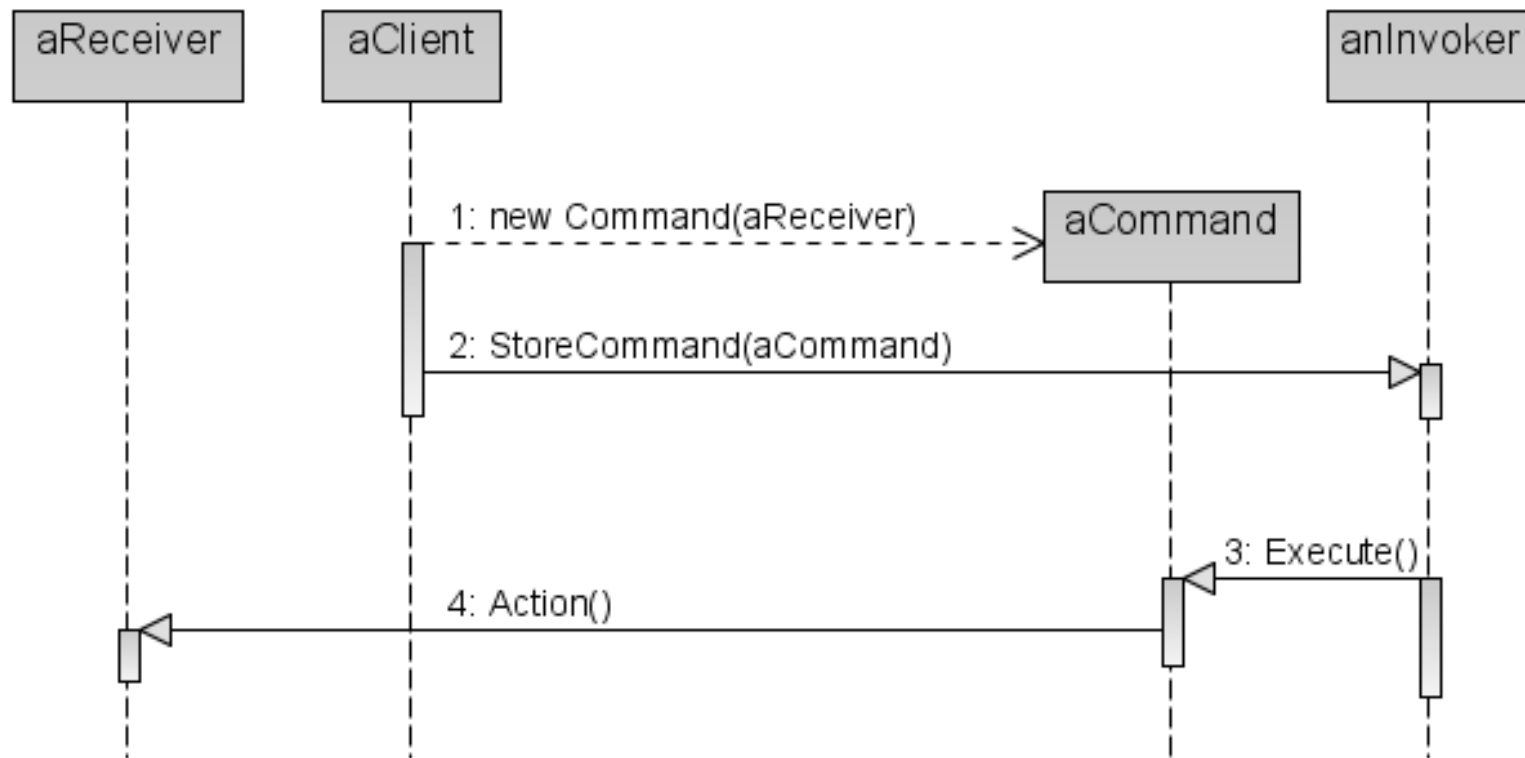
○ Struttura



COMMAND



○ Struttura



COMMAND



○ Conseguenze

- Accoppiamento “lasco” tra oggetto invocante e quello che porta a termine l’operazione
- I *command* possono essere estesi
- I comandi possono essere composti e innestati
- È facile aggiungere nuovi comandi
 - Le classi esistenti non devono essere modificate

COMMAND



○ Esempio

Esempio

Una classe Account modella conti correnti. Le funzionalità che si vogliono realizzare sono:

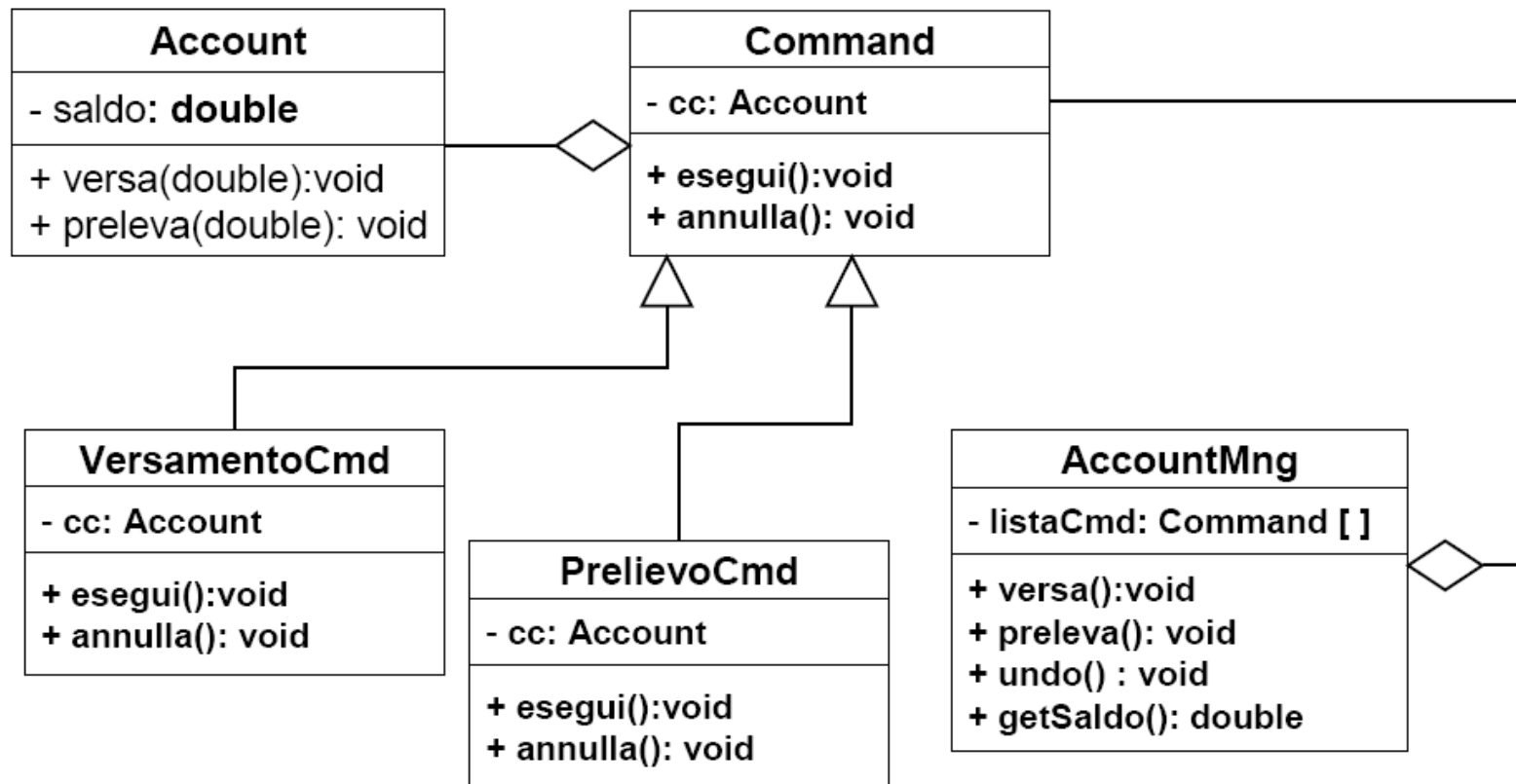
- Prelievo
- Versamento
- Undo

Questa operazione consente di annullare una delle precedenti, ma con il vincolo che l'annullamento deve avvenire con ordine cronologico inverso.

COMMAND



○ Esempio



COMMAND



○ Esempio

- Scala: *first order function*

```
object Invoker {  
  private var history: Seq[() => Unit] = Seq.empty  
  
  def invoke(command: => Unit) { // by-name parameter  
    command  
    history := command _  
  }  
}
```

Parametro *by-name*

```
Invoker.invoke(println("foo"))
```

```
Invoker.invoke {  
  println("bar 1")  
  println("bar 2")  
}
```

È possibile sostituire il *command* con oggetti funzione: maggior concisione, ma minor configurabilità

COMMAND



○ Esempio

- Javascript: utilizzo **oggetti funzione** e *apply*

```
(function(){  
  var CarManager = {  
    // request information  
    requestInfo: function( model, id ) { /* ... */ },  
    // purchase the car  
    buyVehicle: function( model, id ) { /* ... */ },  
    // arrange a viewing  
    arrangeViewing: function( model, id ){ /* ... */ }  
  };  
})();
```

Rende uniforme l'API,
utilizzando il metodo
apply

```
CarManager.execute = function ( name ) {  
  return CarManager[name] && CarManager[name].apply( CarManager,  
    [].slice.call(arguments, 1) );  
};
```

Trasforma l'oggetto
arguments in un array

```
CarManager.execute( "buyVehicle", "Ford Escort", "453543" );
```


COMMAND



○ Implementazione

- Quanto deve essere **intelligente** un comando?
 - Semplice *binding* fra il *receiver* e l'azione da eseguire
 - Comandi agnostici, autoconsistenti
- Supporto all'*undo* e *redo*
 - Attenti allo **stato del sistema** da mantenere (*receiver*, argomenti, valori originali del sistema ...)
 - *History list*
- Accumulo di errori durante l'esecuzione di più comandi successivi
- Utilizzo dei *template* C++ o dei Generics Java

ITERATOR



○ Scopo

- Fornisce l'**accesso sequenziale** agli elementi di un aggregato
 - Senza esporre l'implementazione dell'aggregato

○ Motivazione

- “Per scorrere non è necessario conoscere”
 - Devono essere disponibili **diverse politiche** di **attraversamento**
- Iterator *pattern* sposta la responsabilità di attraversamento in un oggetto iteratore
 - Tiene traccia dell'elemento corrente

ITERATOR



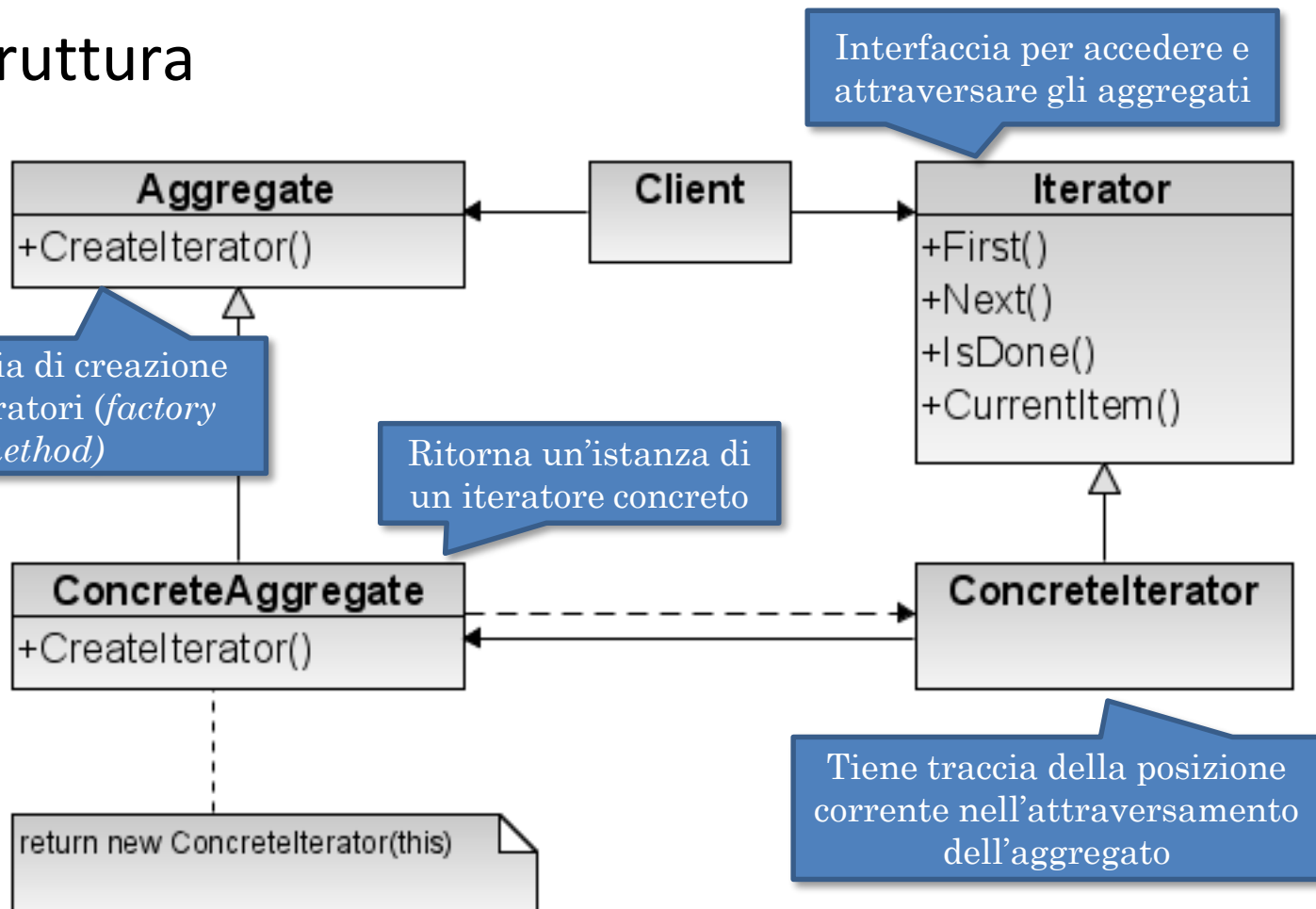
○ Applicabilità

- Accedere il contenuto di un aggregato **senza esporre** la **rappresentazione interna**
- Supportare diverse politiche di attraversamento
- Fornire un'**interfaccia unica** di attraversamento su diversi aggregati
 - *Polymorphic iteration*

ITERATOR



○ Struttura



ITERATOR



○ Conseguenze

- Supporto a **variazioni** nelle politiche di **attraversamento** di un aggregato
- **Semplificazione** dell'interfaccia dell'**aggregato**
- Attraversamento contemporaneo di più iteratori sul medesimo aggregato

ITERATOR



○ Esempio

Esempio

Vediamo alcuni esempi di implementazione del pattern nella libreria J2SE di Java

ITERATOR



○ Esempio

java.sql.ResultSet

```
// preparo ed eseguo una query con JDBC
String sql = "select * from utenti where user = ?";
PreparedStatement pst = connection.prepareStatement(sql);
pst.setString(1,x);
ResultSet rs = pst.executeQuery();

// ciclo i risultati con un generico iteratore
while(rs.next()) {
    Utente utente = new Utente();
    utente.setUser(rs.getString("user"));
    utente.setPassword(rs.getString("password"));
    // ...
}
```

java.util.Iterator

```
// creo un aggregatore concreto
List<Employee> lista = new ArrayList<Employee>();
lista.add(new Employee(...));
lista.add(new Employee(...));

// ciclo tramite un generico iteratore
Iterator iterator = lista.iterator();
while(iterator.hasNext()) {
    Employee e = iterator.next();
    System.out.print(e.getNome() + " guadagna ");
    System.out.println(e.getSalario());
}
```

ITERATOR



○ Implementazione

- Chi **controlla** l'iterazione?
 - *External (active) iterator*: il client controlla l'iterazione
 - *Internal (passive) iterator*: l'iteratore controlla l'iterazione
- Chi **definisce** l'algoritmo di attraversamento?
 - Aggregato: iteratore viene definito "cursore"
 - Il *client* invoca *Next* sull'aggregato, fornendo il cursore
 - Iteratore: viene violata l'*encapsulation* dell'aggregato
 - Miglior **riuso** degli algoritmi di attraversamento
- Iteratori **robusti**
 - Assicurarsi che l'inserimento e la cancellazione di elementi dall'aggregato non creino interferenze

ITERATOR



○ Implementazione

- Operazioni aggiuntive
- *Polymorphic iterator*
 - Utilizzo del Proxy Pattern per deallocazione dell'iteratore
- **Accoppiamento stretto** tra iteratore e aggregato
 - C++, dichiarare *friend* l'iteratore
- Null Iterator
 - Iteratore degenero che implementa `IsDone` con il ritorno di `true`
 - Utile per scorrere strutture ricorsive

OBSERVER



○ Scopo

- Definisce una **dipendenza “1..n”** fra oggetti, riflettendo la **modifica** di un oggetto sui dipendenti

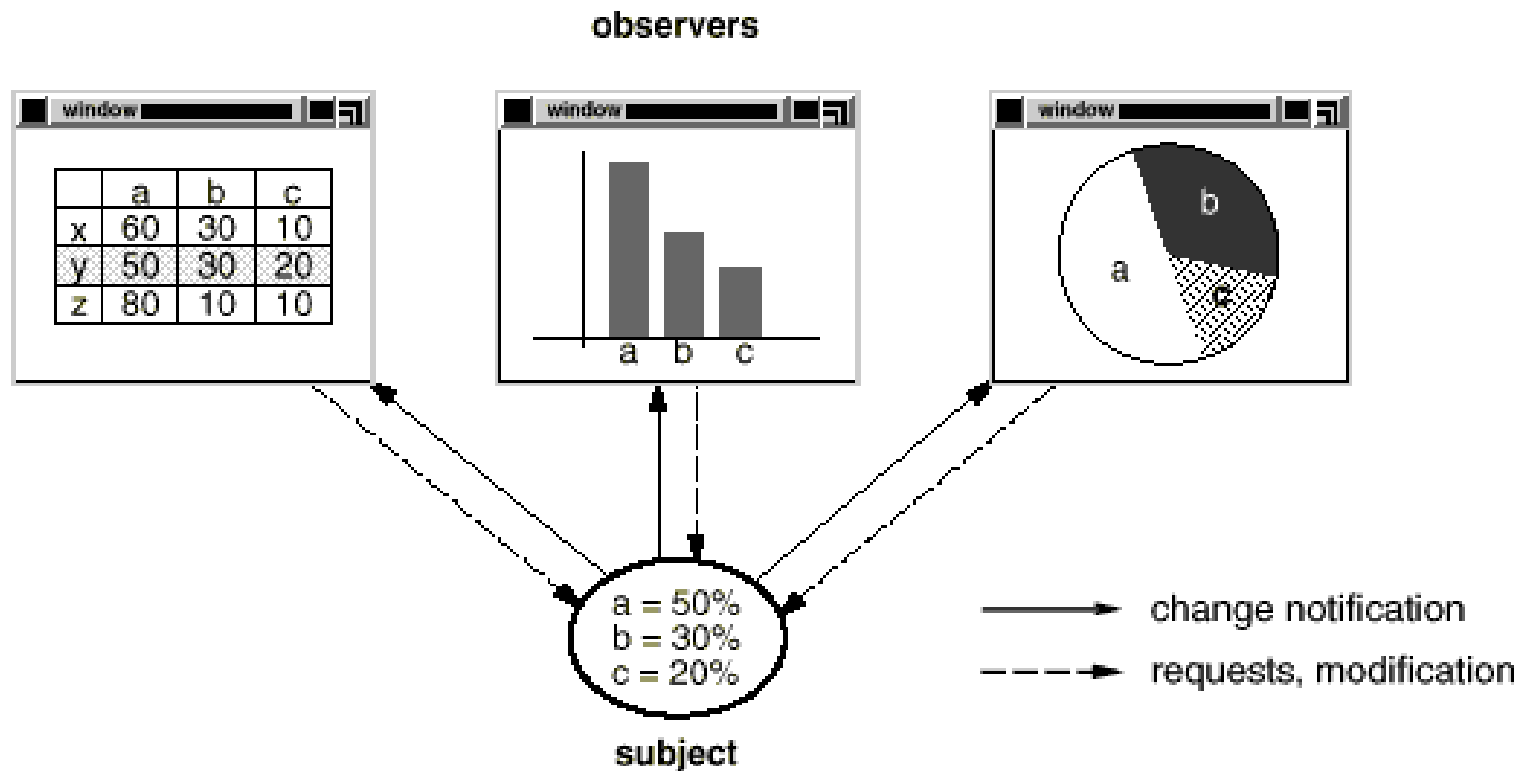
○ Motivazione

- Mantenere la **consistenza** fra oggetti
 - Modello e viste ad esso collegate
- Observer pattern definisce come **implementare** la relazione di dipendenza
 - *Subject*: effettua le notifiche
 - *Observer*: si aggiorna in risposta ad una notifica
- *“Publish - Subscribe”*

OBSERVER



○ Motivazione





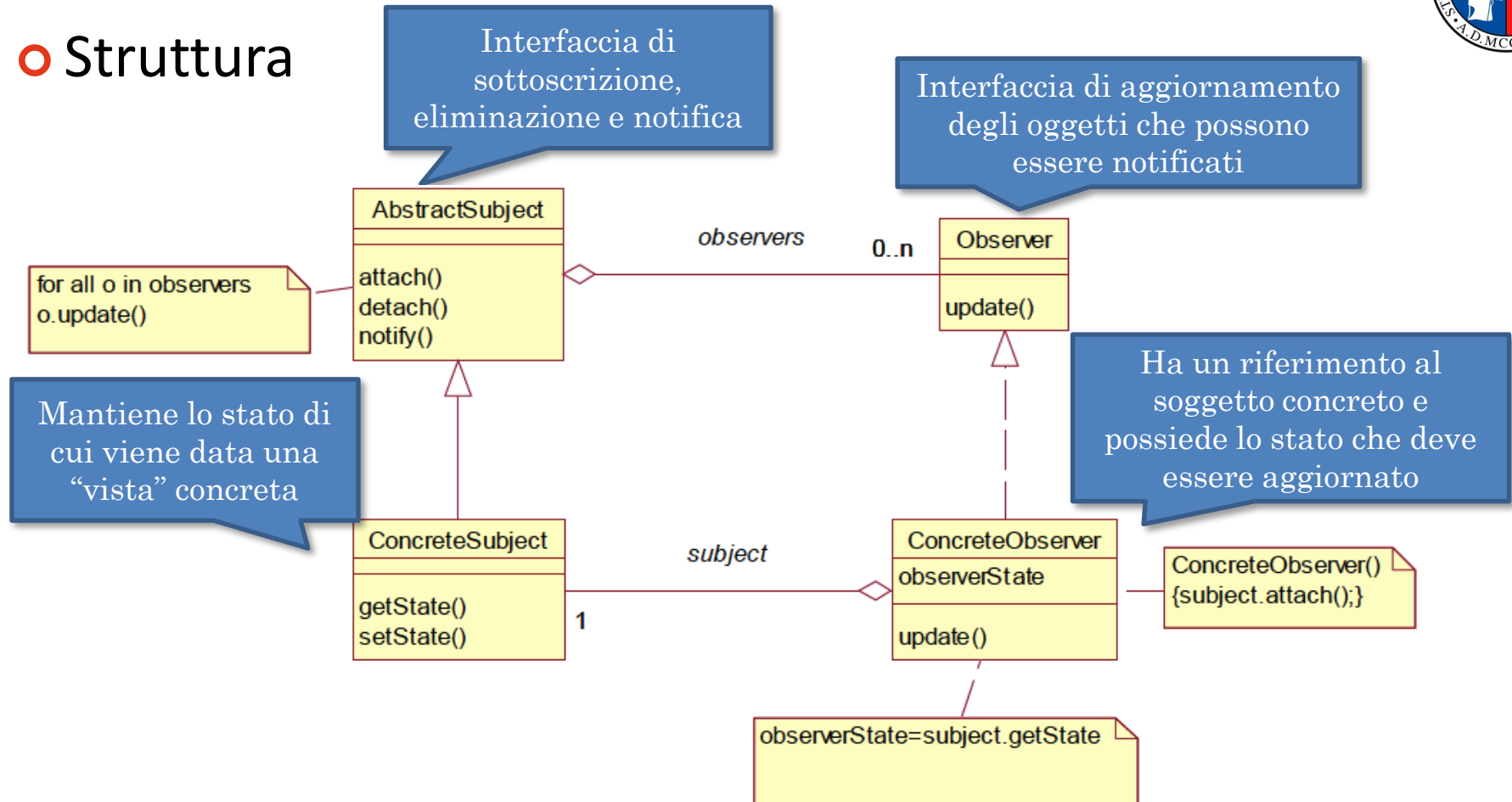
○ Applicabilità

- Associare più “**viste**” differenti ad una astrazione
 - Aumento del grado di **riuso** dei singoli tipi
- Il **cambiamento** di un oggetto richiede il cambiamento di altri oggetti
 - Non si conosce quanti oggetti devono cambiare
- **Notificare** oggetti senza fare assunzioni su quali siano questi oggetti
 - Evita l'accoppiamento “forte”

OBSERVER



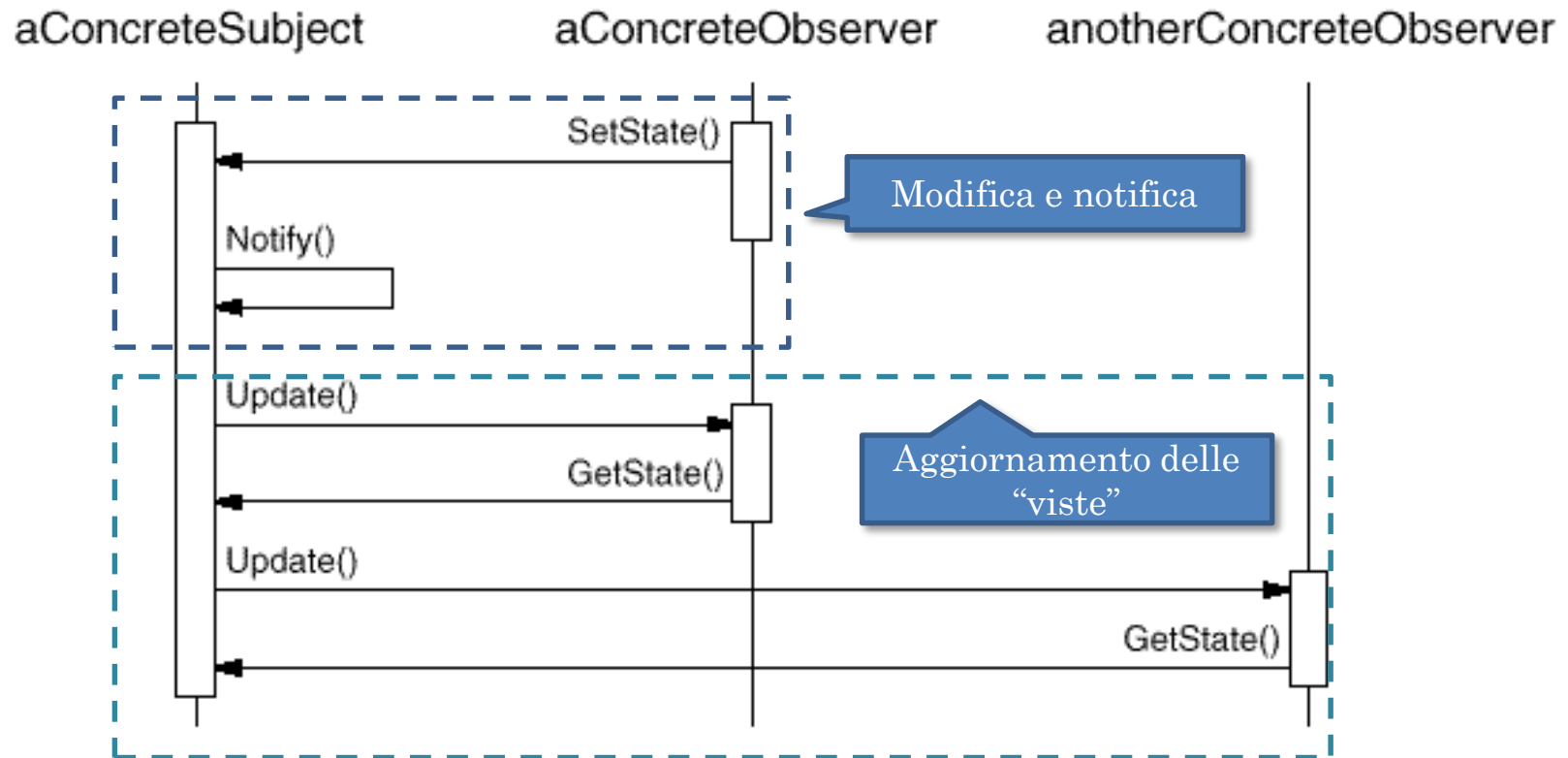
○ Struttura



OBSERVER



○ Struttura





○ Conseguenze

- Accoppiamento “astratto” tra soggetti e osservatori
 - I soggetti non conoscono il tipo concreto degli osservatori
- Comunicazione *broadcast*
 - Libertà di aggiungere osservatori dinamicamente
- Aggiornamenti non voluti
 - Un operazione “innocua” sul soggetto può provocare una cascata “pesante” di aggiornamenti
 - Gli osservatori non sanno cosa è cambiato nel soggetto ...

OBSERVER



○ Esempio

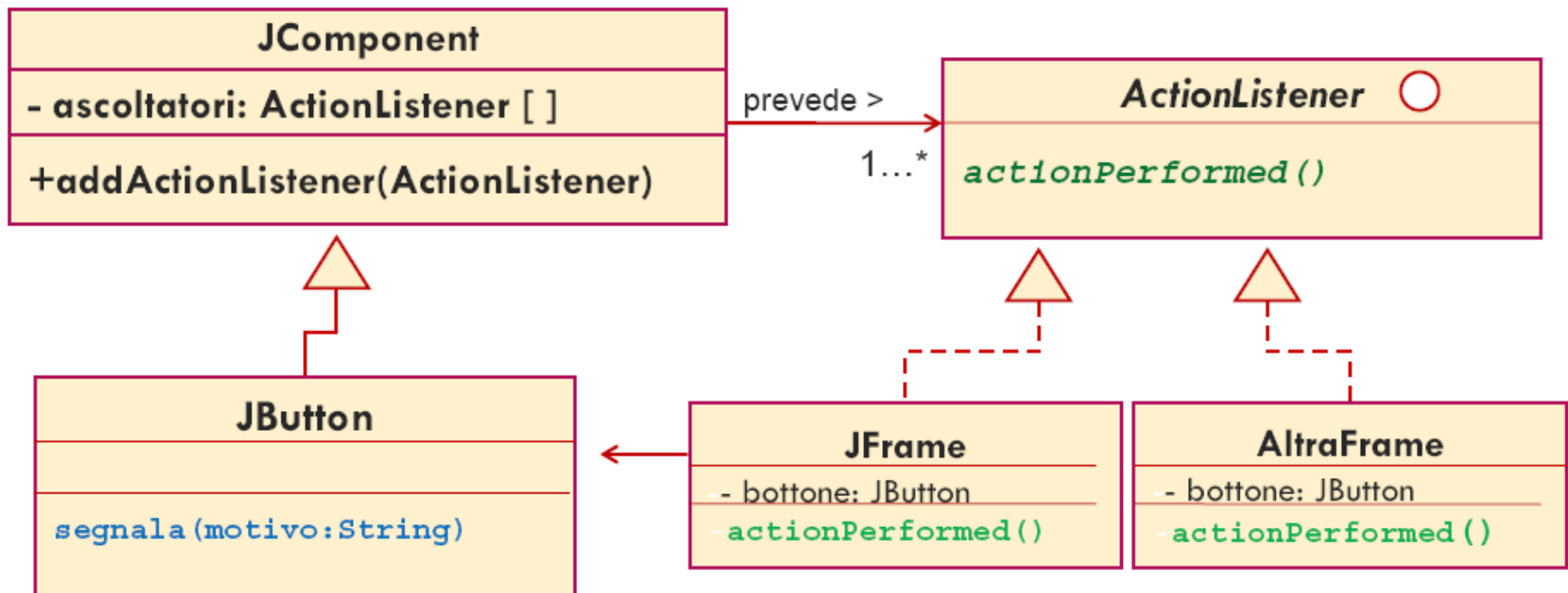
Esempio

Modifica di una o più aree di finestre in risposta alla pressione di un pulsante (Java Swing)

OBSERVER



○ Esempio



- Il costruttore della classe `JFrame` possiede l'istruzione `bottone.addActionListener(this)`
- L'utente clicca sul pulsante e il metodo `segnala` viene invocato
- Il metodo `segnala` invoca il metodo `actionPerformed` su tutti gli oggetti presenti nel vettore "ascoltatori"

OBSERVER



○ Implementazione

- Utilizzo di sistemi di *lookup* per gli osservatori
 - Nessun spreco di memoria nel soggetto
- Osservare più di un soggetto alla volta
 - Estendere l'interfaccia di aggiornamento con il soggetto che ha notificato
- Chi deve attivare l'aggiornamento delle “viste”?
 - Il soggetto, dopo ogni cambiamento di stato
 - Il *client*, a termine del processo di interazione con il soggetto
- Evitare puntatori “pendenti” (*dangling*)
- Notificare solo in stati consistenti
 - Utilizzo del *Template Method pattern*

OBSERVER



○ Implementazione

- Evitare protocolli di aggiornamento con assunzioni
 - *Push model*: il soggetto conosce i suoi osservatori
 - *Pull model*: il soggetto invia solo la notifica
- Notifica delle modifiche sullo stato del soggetto
 - Gli osservatori si registrano su un particolare evento

```
void Subject::Attach(Observer*, Aspect& interest)
```

```
void Observer::Update(Subject*, Aspect& interest)
```

- Unificare le interfacce di soggetto e osservatore
 - Linguaggi che non consentono l'ereditarietà multipla
 - Smalltalk, ad esempio ...

STRATEGY



○ Scopo

- Definisce una **famiglia** di **algoritmi**, rendendoli interscambiabili
 - Indipendenti dal *client*



○ Motivazione

- Esistono differenti algoritmi (strategie) che non possono essere inserite direttamente nel *client*
 - I *client* rischiano di divenire troppo **complessi**
 - Differenti strategie sono appropriate in casi differenti
 - È difficile aggiungere nuovi algoritmi e modificare gli esistenti

STRATEGY



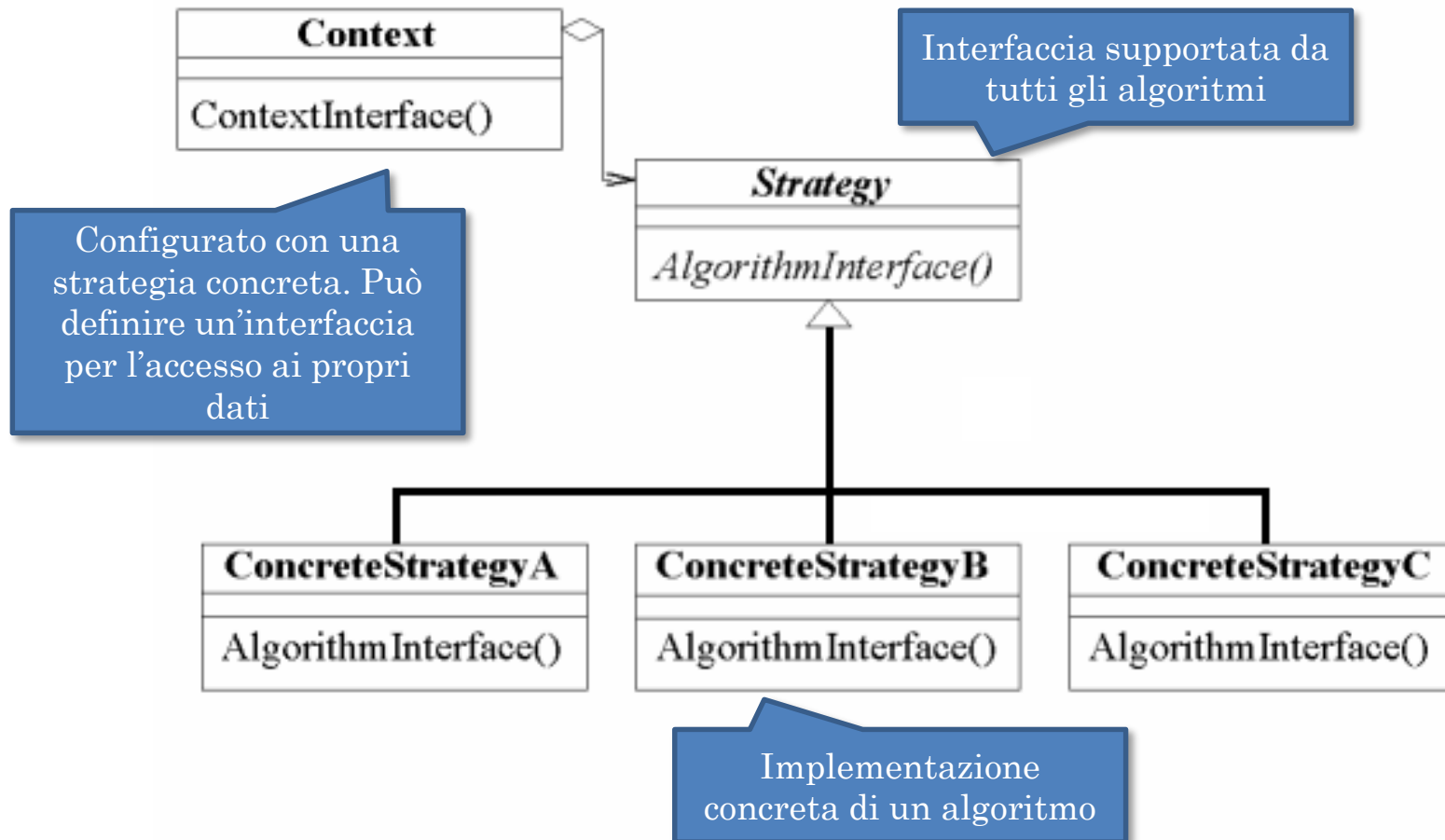
○ Applicabilità

- Diverse classi **differiscono** solo per il loro **comportamento**
- Si necessita di differenti **varianti** dello stesso algoritmo
- Un algoritmo utilizza dati di cui i *client* non devono occuparsi
- Una classe definisce differenti **comportamenti**, tradotti in un serie di *statement* condizionali

STRATEGY



○ Struttura



STRATEGY



○ Conseguenze

- Definizione di famiglie di algoritmi per il **riuso** del contesto
- Alternativa all'ereditarietà dei *client*
 - **Evita** di effettuare **subclassing** direttamente dei contesti
- Eliminazione degli *statement* condizionali

```
void Composition::Repair() {  
    switch (_breakingStrategy) {  
        case SimpleStrategy:  
            ComposeWithSimpleCompositor();  
            break;  
        case TeXStrategy:  
            ComposeWithTeXCompositor();  
            break;  
        // ...  
    }  
}
```

```
void Composition::Repair() {  
    _compositor->Compose();  
    // merge results with existing  
    // composition, if necessary  
}
```

STRATEGY



○ Conseguenze

- Differenti implementazioni dello stesso comportamento
- I *client* a volte devono conoscere dettagli implementativi
 - ... per poter selezionare il corretto algoritmo ...
- Comunicazione tra contesto e algoritmo
 - Alcuni algoritmi non utilizzano tutti gli *input*
- **Incremento** del **numero** di **oggetti** nell'applicazione



○ Esempio

Esempio

Si vuole realizzare una classe `MyArray` per disporre di tutte le funzioni utili per lavorare con vettori di numeri. Si prevedono 2 funzioni di stampa:

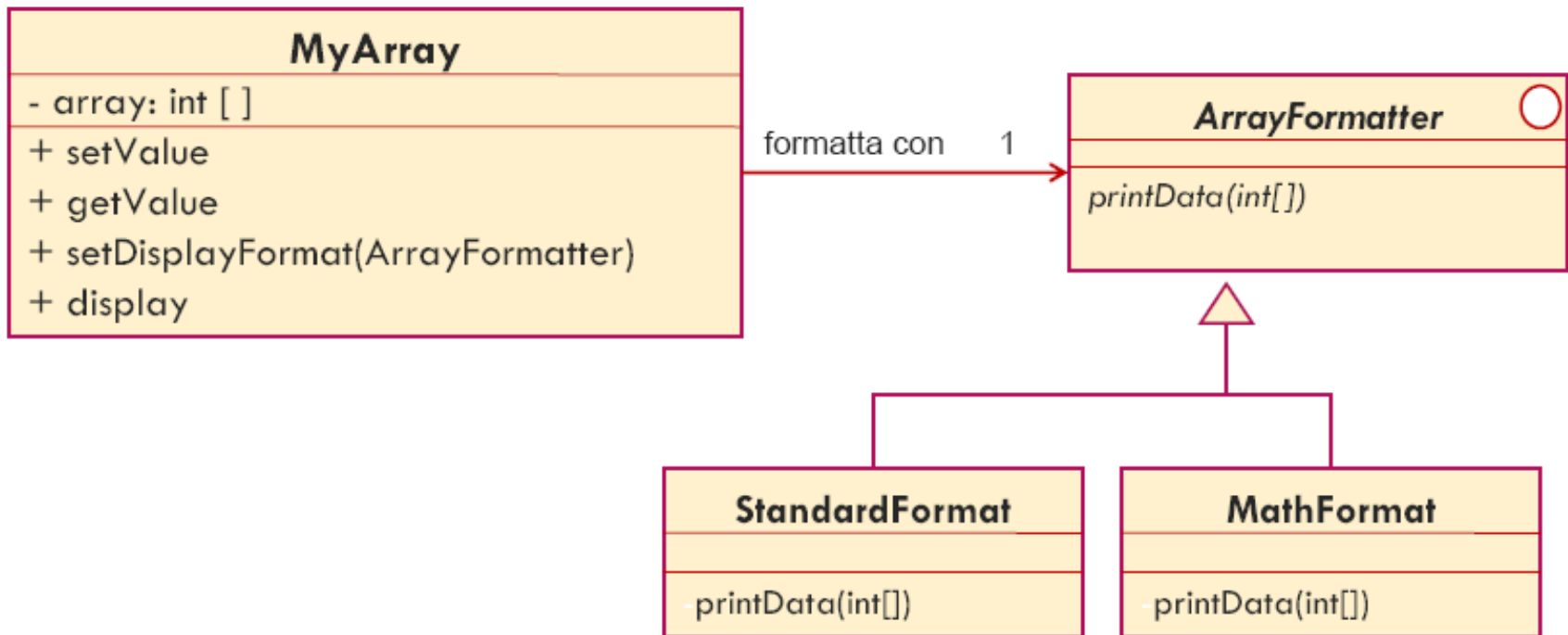
- Formato matematico $\{ 67, -9, 0, 4, \dots \}$
- Formato standard `Arr[0] = 67 Arr[1] = -9 Arr[2] = 0 ...`

Questi formati potrebbero, in futuro, essere sostituiti o incrementati

STRATEGY



○ Esempio



STRATEGY



○ Esempio

- Scala: *first-class functions*

```
type Strategy = (Int, Int) => Int
```

Definizione di un
tipo **funzione**

```
class Context(computer: Strategy) {  
  def use(a: Int, b: Int) { computer(a, b) }  
}
```

```
val add: Strategy = _ + _  
val multiply: Strategy = _ * _
```

Implementazioni
possibili *Strategy*

```
new Context(multiply).use(2, 3)
```

- Le funzioni sono **tipi**
 - Possono essere assegnate a variabili
 - `_` è una *wildcard* ed equivale ad un parametro differente per ogni occorrenza

STRATEGY



○ Implementazione

- Definire le interfacce di strategie e contesti
 - Fornisce singolarmente i dati alle strategie
 - Fornire l'intero contesto alle strategie
 - Inserire un puntamento al contesto nelle strategie
- Implementazione strategie
 - C++: *Template*, Java: *Generics*
 - Solo se l'algoritmo può essere determinato a *compile time* e non può variare dinamicamente
- Utilizzo strategia opzionali
 - Definisce una strategia di *default*

TEMPLATE METHOD



○ Scopo

- Definisce lo **scheletro** di un **algoritmo**, lasciando l'**implementazione** di alcuni passi alle **sottoclassi**
 - Nessuna modifica all'algoritmo originale

○ Motivazione

- Definire un algoritmo in termini di operazioni astratte
 - Viene fissato solo l'ordine delle operazioni
- Le sottoclassi forniscono il comportamento concreto

TEMPLATE METHOD



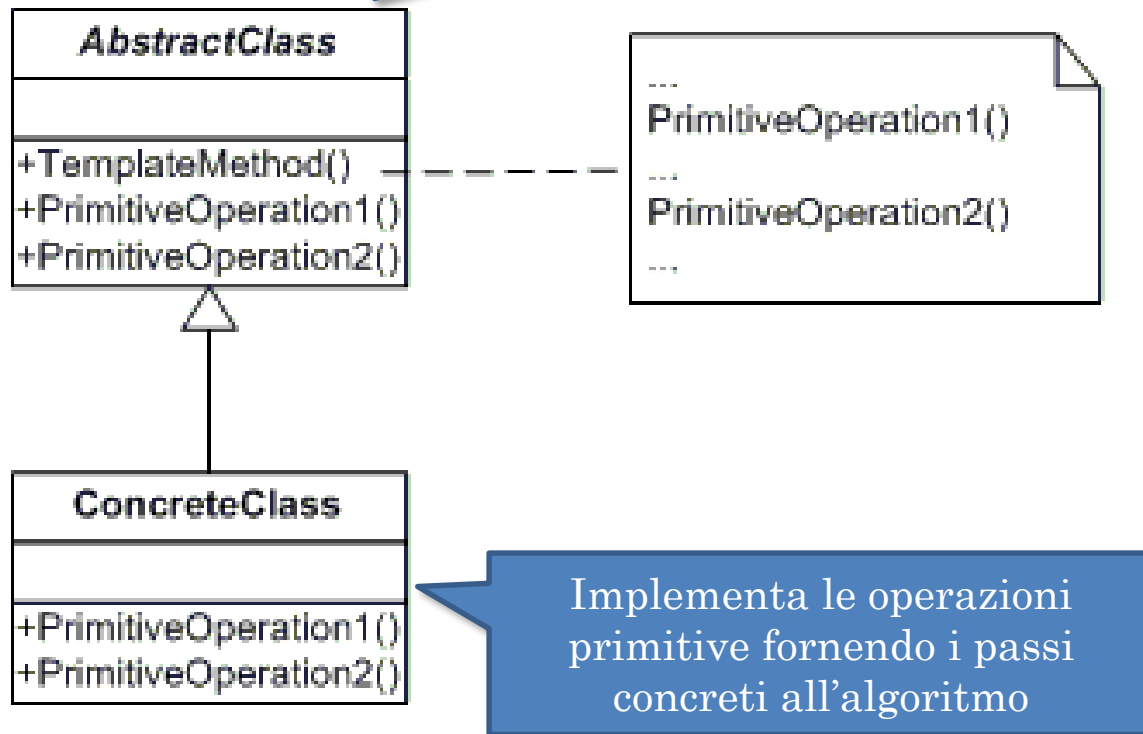
○ Applicabilità

- Implementare le **parti invarianti** di un algoritmo una volta sola
- Evitare la **duplicazione** del codice
 - Principio “*refactoring to generalize*”
- **Controllare** le possibili **estensioni** di una classe
 - Fornire sia operazioni astratte sia operazioni *hook* (*wrapper*)

TEMPLATE METHOD



○ Struttura



TEMPLATE METHOD



○ Conseguenze

- Tecnica per il **riuso** del codice
 - Fattorizzazione delle responsabilità
- “*The Hollywood principle*”
- Tipi di operazioni possibili
 - Operazioni concrete della classe astratta
 - Operazioni primitive (astratte)
 - Operazioni *hook*
 - Forniscono operazioni che di *default* non fanno nulla, ma rappresentano punti di estensione
- Documentare bene quali sono operazioni primitive e quali *hook*



TEMPLATE METHOD



○ Esempio

Esempio

Si vuole realizzare un set di funzioni per effettuare operazioni sugli array. Si prevedono 2 funzioni aritmetiche:

- Somma di tutti gli elementi
- Prodotto di tutti gli elementi

TEMPLATE METHOD



○ Esempio

- Soluzione *naïve*

```
public int somma(int[] array) {  
    int somma = 0;  
    for (int i = 0; i < array.length; i++) {  
        somma += array[i];  
    }  
    return somma;  
}
```

```
public int prodotto(int[] array){  
    int prodotto= 1;  
    for (int i = 0; i < array.length; i++) {  
        prodotto *= array[i];  
    }  
    return prodotto;  
}
```

TEMPLATE METHOD



○ Esempio

- Soluzione con *Template Method pattern*

```
public abstract class Calcolatore {  
  
    public final int calcola(int[] array){  
        int value = valoreIniziale();  
        for (int i = 0; i < array.length; i++) {  
            value = esegui(value, array[i]);  
        }  
        return value;  
    }  
  
    protected abstract int valoreIniziale();  
    protected abstract int esegui(int currentValue, int element);  
}
```

```
public class CalcolatoreSomma {  
  
    protected int esegui(int currentValue, int element) {  
        return currentValue + element;  
    }  
  
    protected int valoreIniziale() {  
        return 0;  
    }  
}
```

TEMPLATE METHOD



○ Esempio

- Scala: **idioma**, utilizzo *high order function*

```
def doForAll[A, B](l: List[A], f: A => B): List[B] = l match {  
  case x :: xs => f(x) :: doForAll(xs, f)  
  case Nil => Nil  
}
```

```
// Already in Scala specification  
List(1, 2, 3, 4).map {x => x * 2}
```

- Utilizzo metodi *map*, *forall*, *flatMap*, ...
- **Monads**
- ...

TEMPLATE METHOD



○ Esempio

- Javascript: utilizzo *delegation*
 - Invocazione di un metodo è propagata ai livelli superiori dell'albero dell'ereditarietà

```
function AbsProperty() {  
    this.build = function() {  
        var result = this.doSomething();  
        return "The decoration I did: " + result;  
    };  
}  
OpenButton.prototype = new AbsProperty();  
function OpenButton () {  
    this.doSomething = function() { return "open button"; };  
}  
SeeButton.prototype = new AbsProperty();  
function SeeButton () {  
    this.doSomething = function() { return "see button"; };  
}  
var button = new SeeButton(); button.build();
```

Ricerca nel contesto
del metodo

Risale l'albero dei
prototipi

TEMPLATE METHOD



○ Implementazione

- Le operazioni **primitive** dovrebbero essere membri **protetti**
- Il *template method* non dovrebbe essere ridefinito
 - Java: dichiarazione `final`
- **Minimizzare** il numero di operazioni **primitive**
 - ... resta poco nel *template method* ...
- Definire una *naming convention* per i nomi delle operazioni di cui effettuare *override*

RIFERIMENTI



- Design Patterns, Elements of Reusable Object Oriented Software, GoF, 1995, Addison-Wesley
- Design Patterns
http://sourcemaking.com/design_patterns
- Java DP
<http://www.javacamp.org/designPattern/>
- Deprecating the Observer Pattern
<http://lampwww.epfl.ch/~imaier/pub/DeprecatingObserverTR2010.pdf>
- Ruminations of a Programmer
<http://debasishg.blogspot.it/2009/01/subsuming-template-method-pattern.html>