CORSO I.F.T.S.

"TECNICHE PER LA PROGETTAZIONE E LA GESTIONE DI DATABASE"

Matricola 2014LA0033

DISPENSE DIDATTICHE
MODULO DI "PROGETTAZIONE SOFTWARE"

Dott. Imad Zaza

Lezione del 23/07/2014

Diagramma di sequenza

- è utilizzato per definire la logica di uno scenario (specifica sequenza di eventi) di un caso d'uso (in analisi e poi ad un maggior livello di dettaglio in disegno)
- è uno dei principali input per l'implementazione dello scenario
- mostra gli oggetti coinvolti specificando la sequenza temporale dei messaggi che gli oggetti si scambiano
- è un **diagramma di interazione**: evidenzia come un caso d'uso è realizzato tramite la collaborazione di un insieme di oggetti

Alcune definizioni

- Un diagramma di sequenza è un diagramma che descrive interazioni tra oggetti che collaborano per svolgere un compito
- Gli oggetti collaborano scambiandosi messaggi
- Lo scambio di un messaggio in programmazione ad oggetti equivale all'invocazione di un metodo

Sequence diagram: componenti

- **Condizione:** è associata ad un messaggio: solo se è soddisfatta il messaggio viene generato

[ha disponib]

- **Iterazione:** indica che un messaggio viene inoltrato più volte ad oggetti diversi di uno stesso gruppo
- **Ritorno:** indica il valore restituito all'oggetto chiamante e non un nuovo messaggio; può essere omesso

- **Distruzione:** deallocazione di un oggetto come metodo interno o tramite messaggio da un altro oggetto

Sequence diagram: messaggi

- In generale un **messaggio** rappresenta il trasferimento del controllo da un oggetto ad un altro
- Se l'oggetto che invia il messaggio rimane in attesa che l'oggetto ricevente ritorni, si ha un messaggio **sincrono**
- Se invece l'oggetto che invia il messaggio prosegue la propria elaborazione in parallelo all'oggetto chiamato, siamo in presenza di un messaggio **asincrono**.

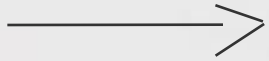
Scambio Messaggi Sincroni (1/2)

- collega freccia di invocazione e freccia di ritorne a chiamato. La freccia è etichettata col nome del metodo invocato, e opzionalmente i suoi parametri e il suo valore di ritorno
- Il chiamante attende la terminazione del metodo del chiamato prima di proseguire
- Il life-time (durata, vita) di un metodo è rappresentato da un rettangolino che collega freccia di invocazione e freccia di ritorno

Scambio Messaggi Sincroni (2/2)

- Life-time corrisponde ad avere un record di attivazione di quel metodo sullo stack di attivazione
- Il ritorno è rappresentato con una freccia tratteggiata
- Il ritorno è sempre opzionale. Se si omette, la fine del metodo è decretata dalla fine del life-time

Scambio Messaggi Asincroni



- Si usano per descrivere interazioni concorrenti
- Si disegna con una freccia aperta

  da chiamante a chiamato. La freccia è etichettata col nome del metodo invocato, e opzionalmente i suoi parametri e il suo valore di ritorno
- Il chiamante non attende la terminazione del metodo del chiamato, ma prosegue subito dopo l'invocazione
- Il ritorno non segue quasi mai la chiamata

Esecuzione condizionale di un messaggio

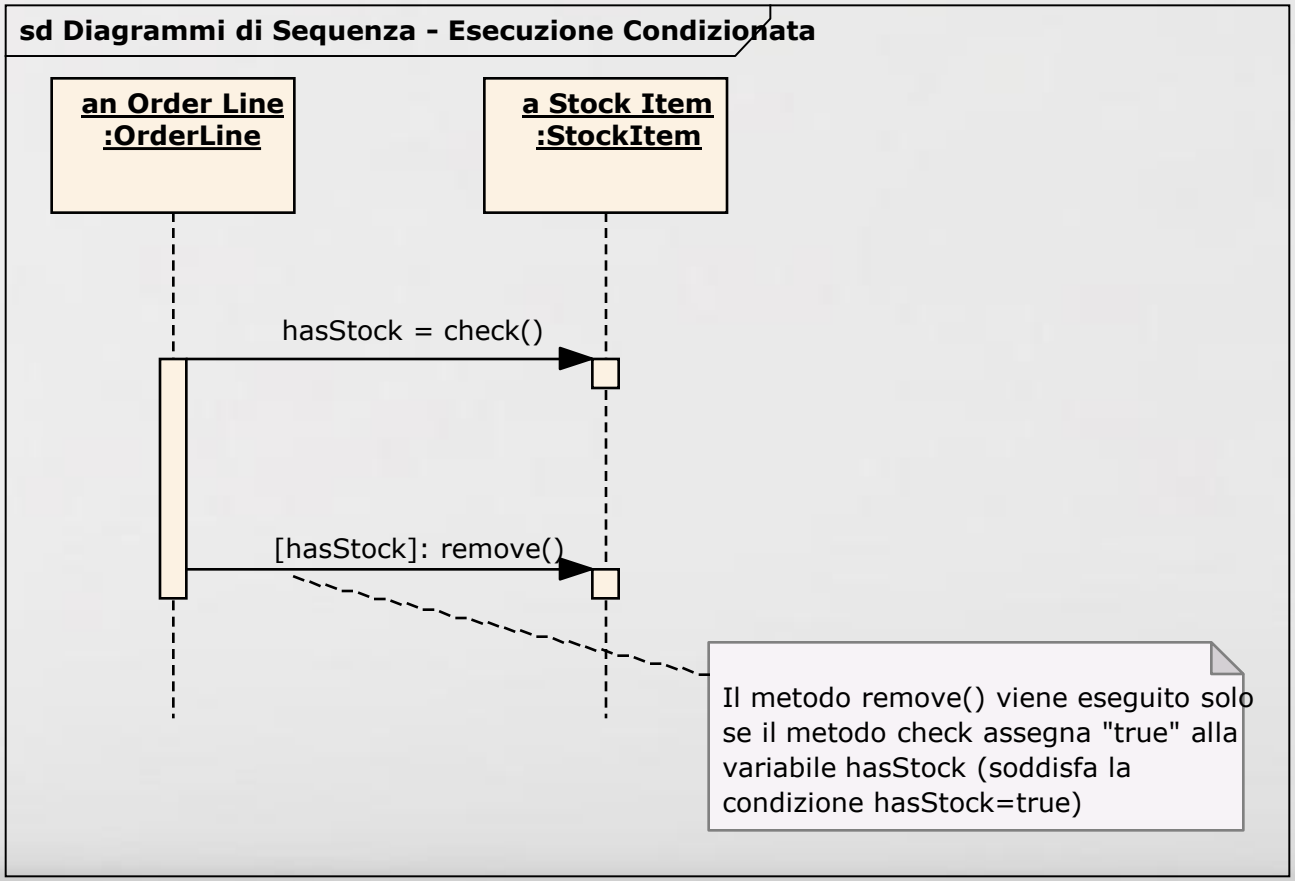
- L'esecuzione di un metodo può essere assoggettata ad una condizione. Il metodo viene invocato solo se la condizione risulta verificata a run-time

- Si disegna aggiungendo la condizione, racchiusa tra parentesi quadre, che definisce quando viene eseguito il metodo

- Sintassi:

[cond] : nomeMetodo()

Esecuzione condizionale di un messaggio - esempio



Iterazione di un messaggio

- Rappresenta l'esecuzione ciclica di messaggi
- Si disegna aggiungendo un * (asterisco) prima del metodo su cui si vuole iterare
- Si può aggiungere la condizione che definisce l'iterazione
- La condizione si rappresenta tra parentesi quadre. Sintassi completa:

[cond] : * nomeMetodo()

Iterazione di un blocco di messaggi

- Rappresenta l'esecuzione ciclica di più messaggi
- Si disegna raggruppando con un blocco (riquadro, box) i messaggi (metodi) su cui si vuole iterare
- Si può aggiungere la condizione che definisce l'iterazione sull'angolo in alto a sinistra del blocco
- La condizione si rappresenta al solito tra parentesi quadre

“Auto-Chiamata” (Self-Call)

- Descrive un oggetto che invoca un suo metodo (chiamante e chiamato coincidono)
- Si rappresenta con una “freccia circolare” che rimane all’interno del life time di uno stesso metodo

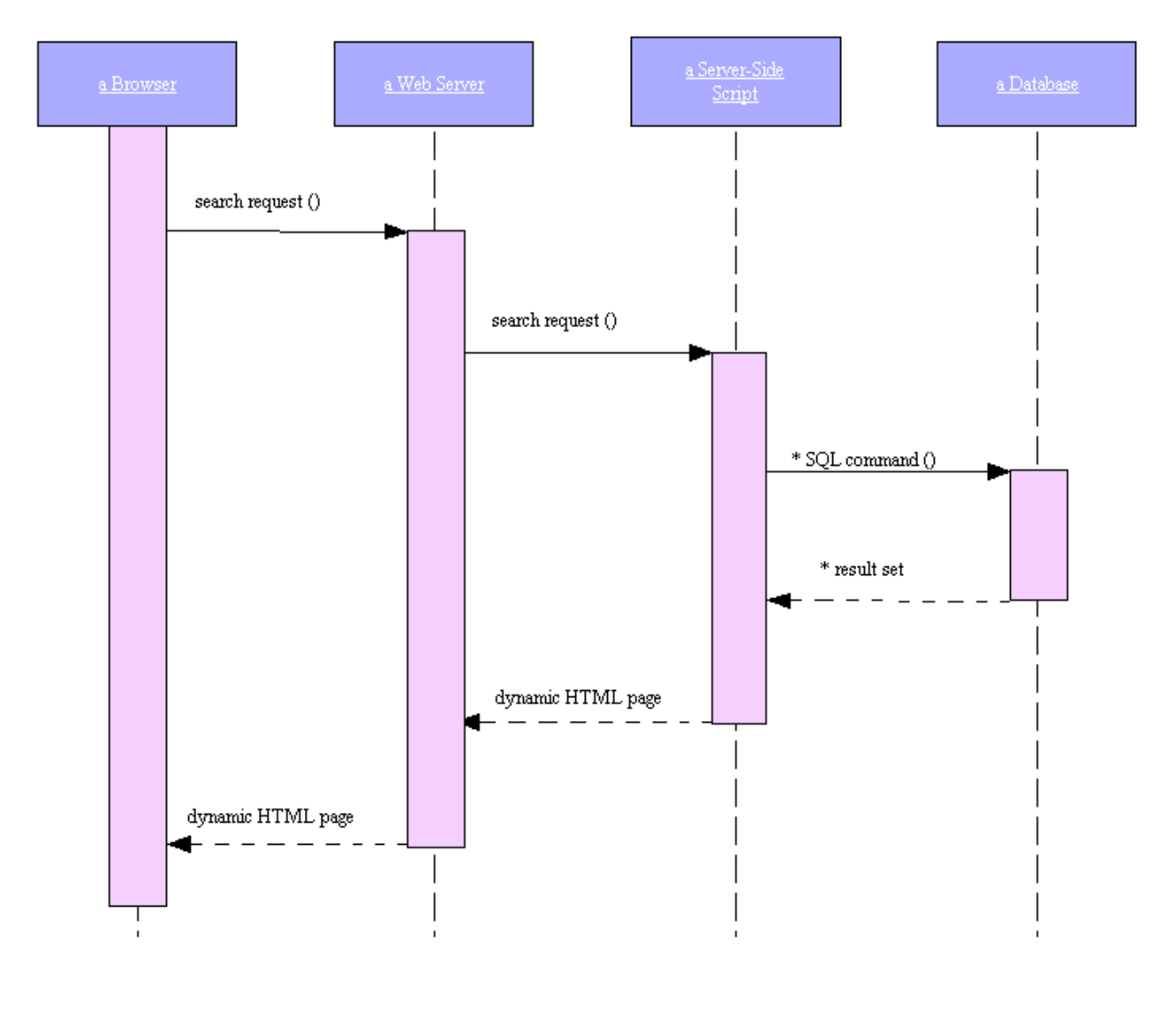
Costruzione di un oggetto

- Rappresenta la costruzione di un nuovo oggetto non presente nel sistema fino a quel momento
- Messaggio etichettato new, create,…
- L'oggetto viene collocato nell'asse temporale in corrispondenza dell'invocazione nel metodo new (o create…)

Eliminazione di un oggetto

- Rappresenta la distruzione di un oggetto presente nel sistema fino a quel momento
- Si rappresenta con una X posta in corrispondenza della life-line dell'oggetto
- Da quel momento in poi non è legale invocare alcun metodo dell'oggetto distrutto

Esempio Applicazione Web

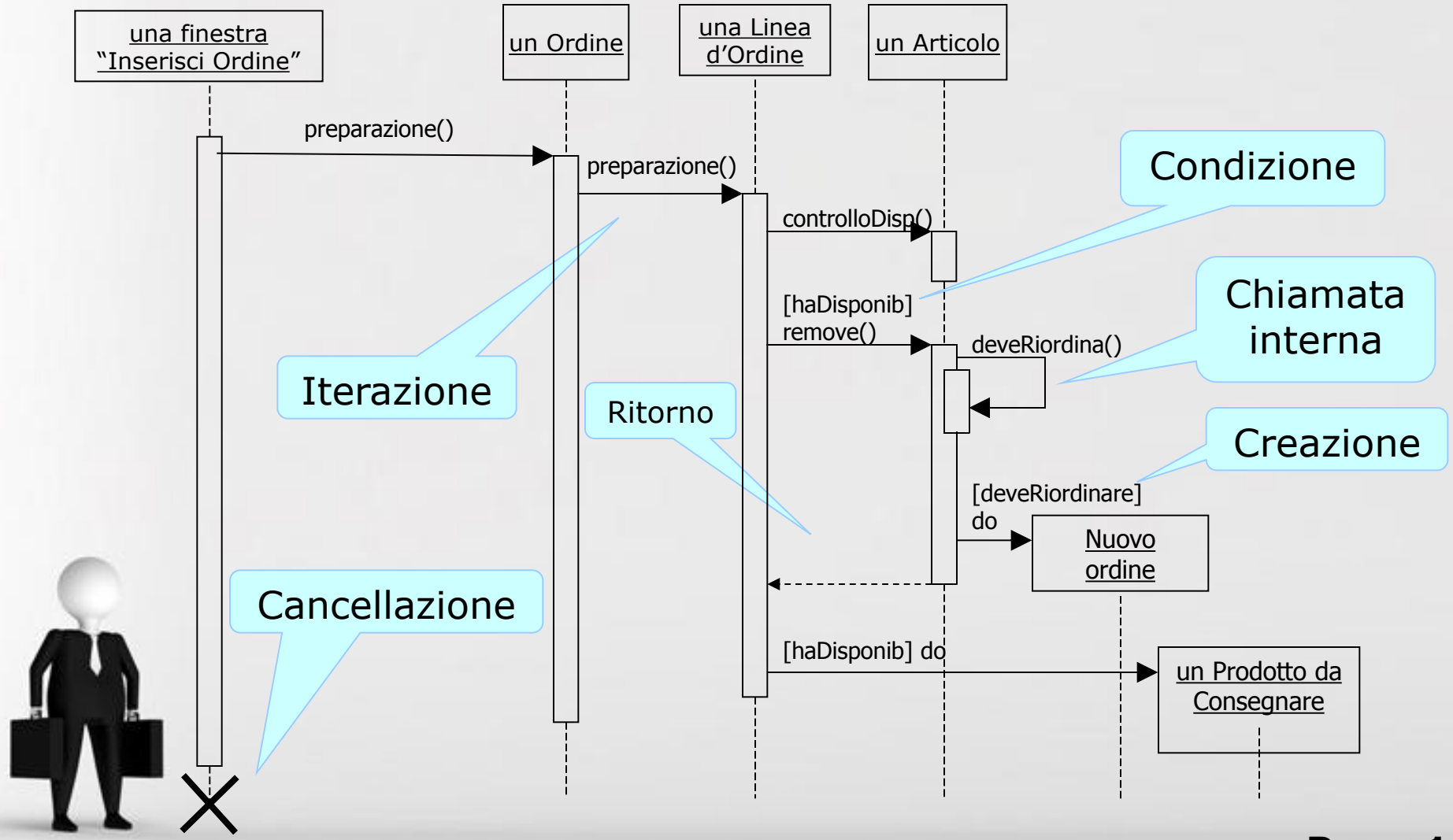


Esercizio: ordine prodotto

Supponiamo di dover illustrare il seguente caso:

- La finestra Inserisci Ordine manda un "messaggio di preparazione" ad un Ordine
- L'Ordine invia messaggi di preparazione a tutte le Linee d'Ordine contenute nell'Ordine
- Ciascuna Linea controlla la disponibilità del proprio Articolo:
  - se è presente lo rimuove dal magazzino e crea un prodotto da consegnare
  - se la disponibilità del prodotto è scesa al di sotto di una certa soglia l'oggetto Articolo genera una richiesta di un nuovo ordine

Esercizio: soluzione



Esercizio

- Costruiamo un diagramma di sequenza per il seguente use case
  - Una finestra di tipo Order Entry invia il messaggio “prepare” ad un Ordine (Order)
  - L’ordine invia il messaggio “prepare” ad ogni sua linea (Order Line)
  - Ogni linea verifica gli elementi in stock (Stock Item)
  - Se il controllo ha esito positivo, la linea rimuove l’appropriata quantità di elementi in stock e crea un’unità di delivery (DeliveryItem)
  - Se gli elementi in stock rimanenti scendono al di sotto di una soglia di riordino, viene richiesto un riordino (ReorderItem)

Alcuni suggerimenti finali

- Assicurarsi che i metodi rappresentati nel diagramma siano gli stessi definiti nelle corrispondenti classi (con lo stesso numero e lo stesso tipo di parametri)
- Documentare ogni assunzione nella dinamica con note o condizioni (ad es. il ritorno di un determinato valore al termine di un metodo, il verificarsi di una condizione all'uscita da un loop, ecc.)
- Mettere un titolo per ogni diagramma (ad es. "sd Diagrammi di Sequenza – Eliminazione di un Oggetto")

Alcuni suggerimenti finali

- Scegliere nomi espressivi per le condizioni e per i valori di ritorno
- Non inserire troppi dettagli in un unico diagramma (flussi condizionati, condizioni, logica di controllo)
- Non bisogna rappresentare tutto quello che si rappresenta nel codice …
- Se il diagramma è complesso, scomporlo in più diagrammi semplici (ad es. uno per il ramo if, un altro per il ramo else, ecc.)

Collaboration diagram

- Evidenzia la collaborazione fra gli oggetti, ma lo svolgimento temporale complessivo risulta meno chiaro rispetto ai sequence diagrams.
- I componenti del diagramma sono sempre **oggetti** e **messaggi** ma questa volta non è modellato esplicitamente in maniera grafica lo scorrere del tempo. Gli oggetti sono disposti liberamente, e in modo da rendere chiare le interazioni più importanti
- Ogni oggetto ha il formato

  NomeOggetto: NomeClasse

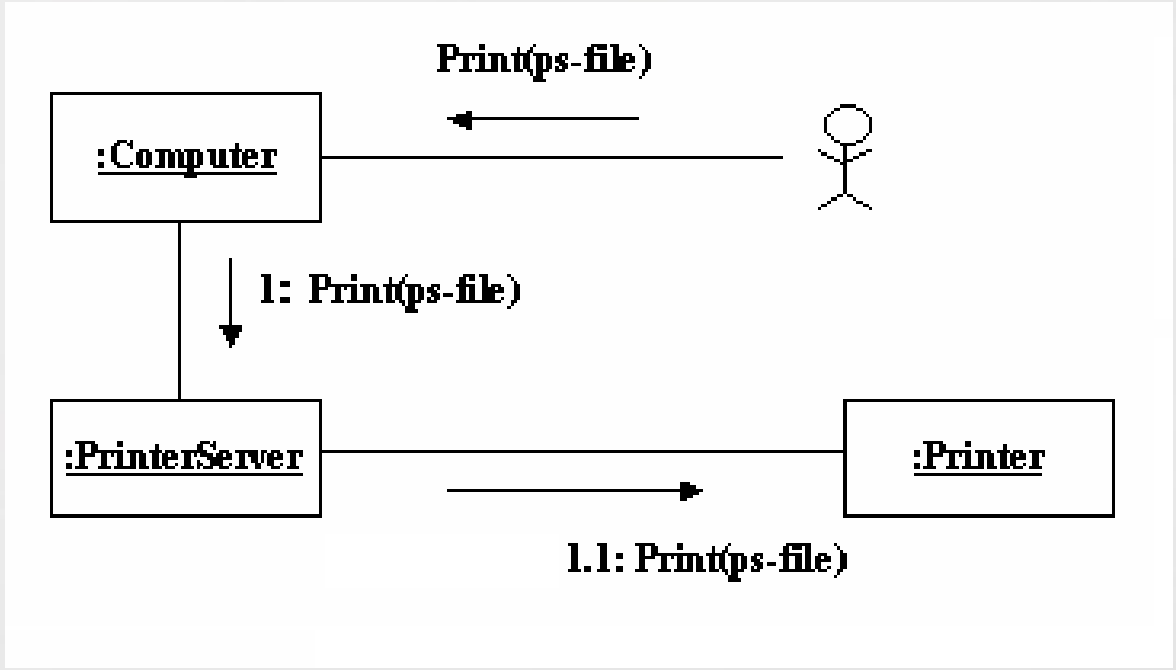
  ma è possibile indicare solo l'uno o l'altro

Collaboration diagram

- A differenza del sequence diagram, i messaggi sono numerati, in modo da evidenziare esattamente la sequenza

- I messaggi sono accompagnati da frecce per capire quale oggetto invia il messaggio e quale lo riceve

Collaboration diagram: esempio

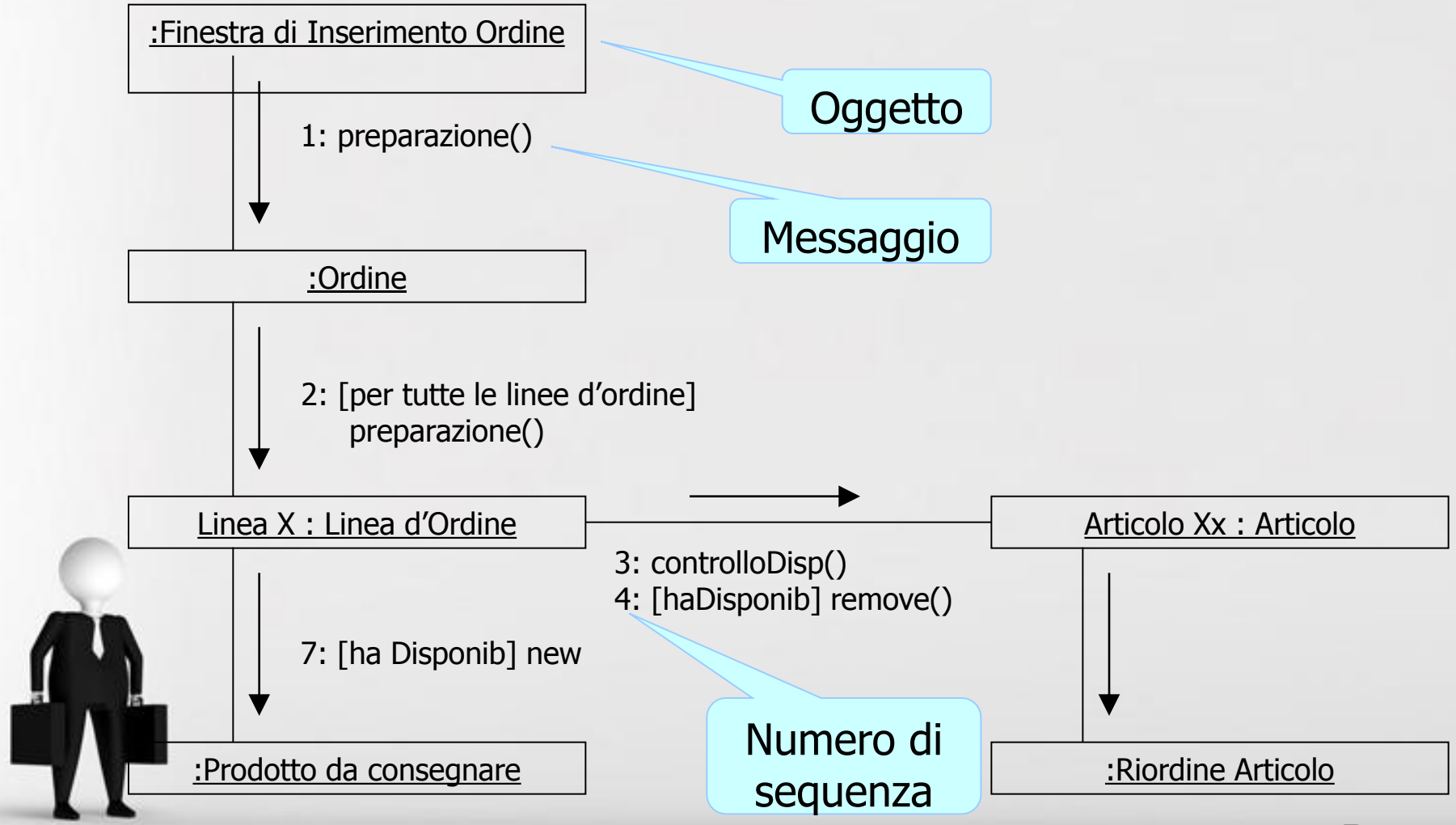


Esercizio: ordine prodotto

Supponiamo di dover illustrare il seguente caso:

- La finestra Inserisci Ordine manda un "messaggio di preparazione" ad un Ordine
- L'Ordine invia messaggi di preparazione a tutte le Linee d'Ordine contenute nell'Ordine
- Ciascuna Linea controlla la disponibilità del proprio Articolo:
  - se è presente lo rimuove dal magazzino e crea un prodotto da consegnare
  - se la disponibilità del prodotto è scesa al di sotto di una certa soglia l'oggetto Articolo genera una richiesta di un nuovo ordine

Esercizio: soluzione

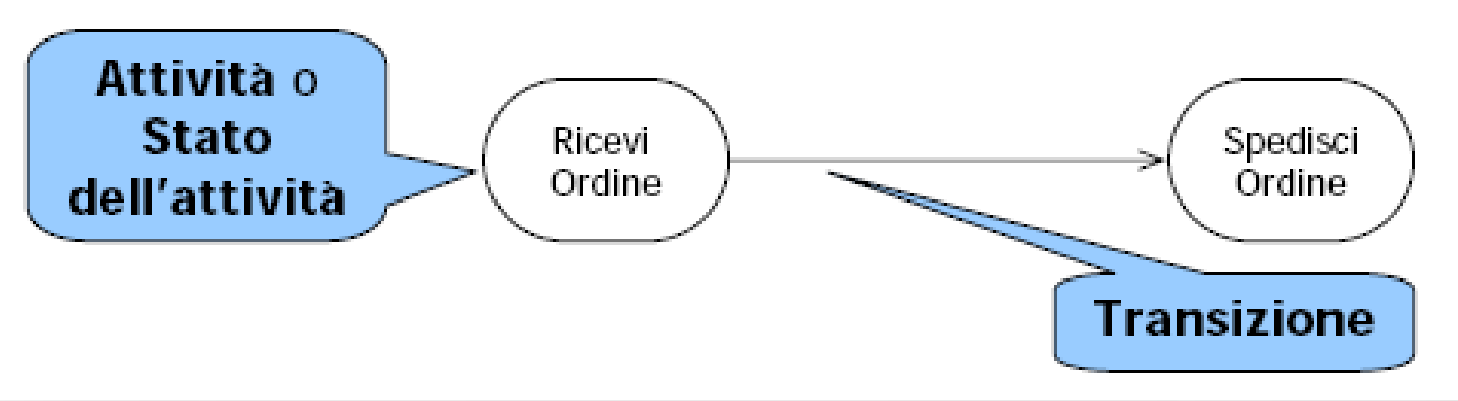


Activity Diagram

- Servono per la modellazione di workflow e per descrivere processi con una forte componente di computazione parallela
- Il componente principale è **l'Attività**: si può definire in maniera informale come il "fare qualcosa", cioè un processo del mondo reale (es. compilare un ordine) o l'esecuzione di una procedura software (es. metodo di una classe)
- Il diagramma consiste in una sequenza di attività e supporta l'esecuzione di cicli,l'esecuzione parallela e quella condizionale

Activity Diagram

- Oltre alle Attività le altre componenti sono le **Transizioni**
- Le transizioni non hanno condizioni e sono innescate semplicemente dal termine dell'attività precedente

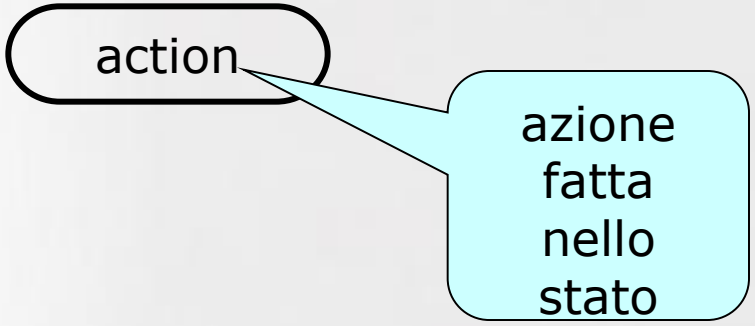


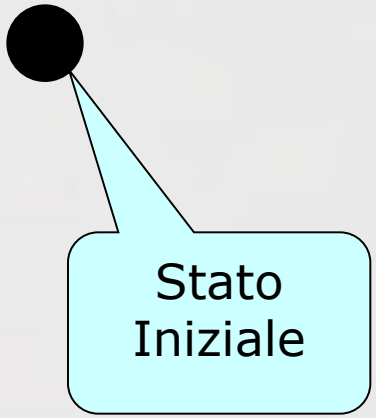
Activity Diagram

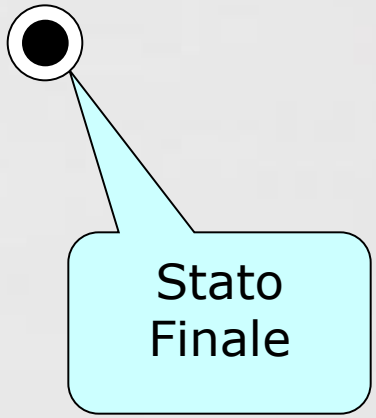
- Un Activity Diagram può essere associato:
  - A una classe
  - All'implementazione di un'operazione
  - Ad uno Use Case

Activity Diagram: Componenti

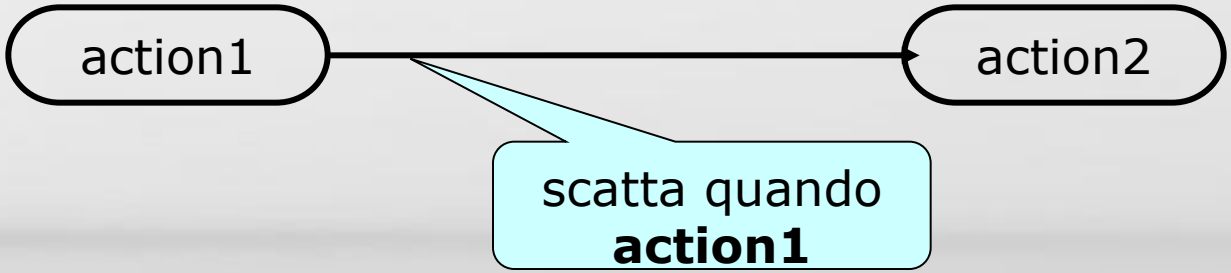
- Action state





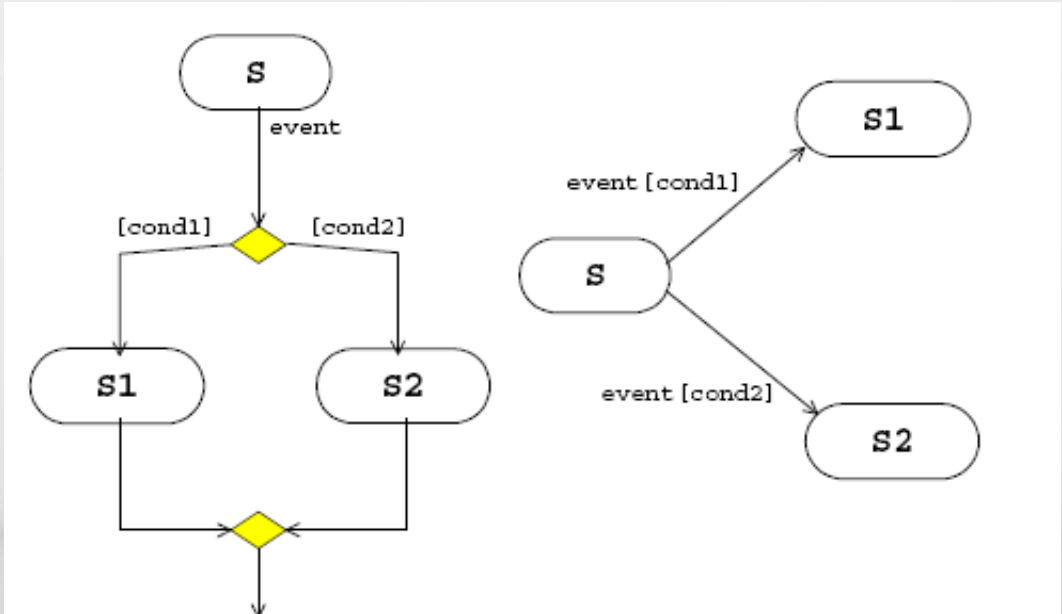


- Transizioni scatenate da completion events



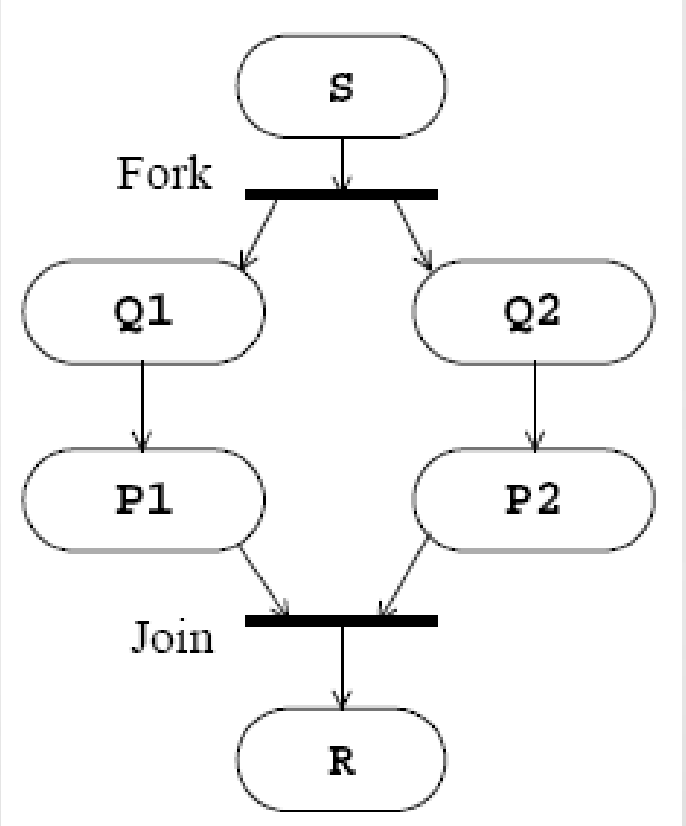
Activity Diagram: Branch & Merge

- **Branch**: diramazione con una transizione entrante e più di una uscente, con condizioni mutuamente esclusive
  - Non è possibile che valgano entrambe (comportamento ambiguo) o nessuna comportamento bloccato)
  - Se le condizioni sono più di due deve comunque esserne verificata ogni volta una e una soltanto
- **Merge**: giunzione con più transizioni entranti ed una sola uscente, termina il blocco aperto dal Branch

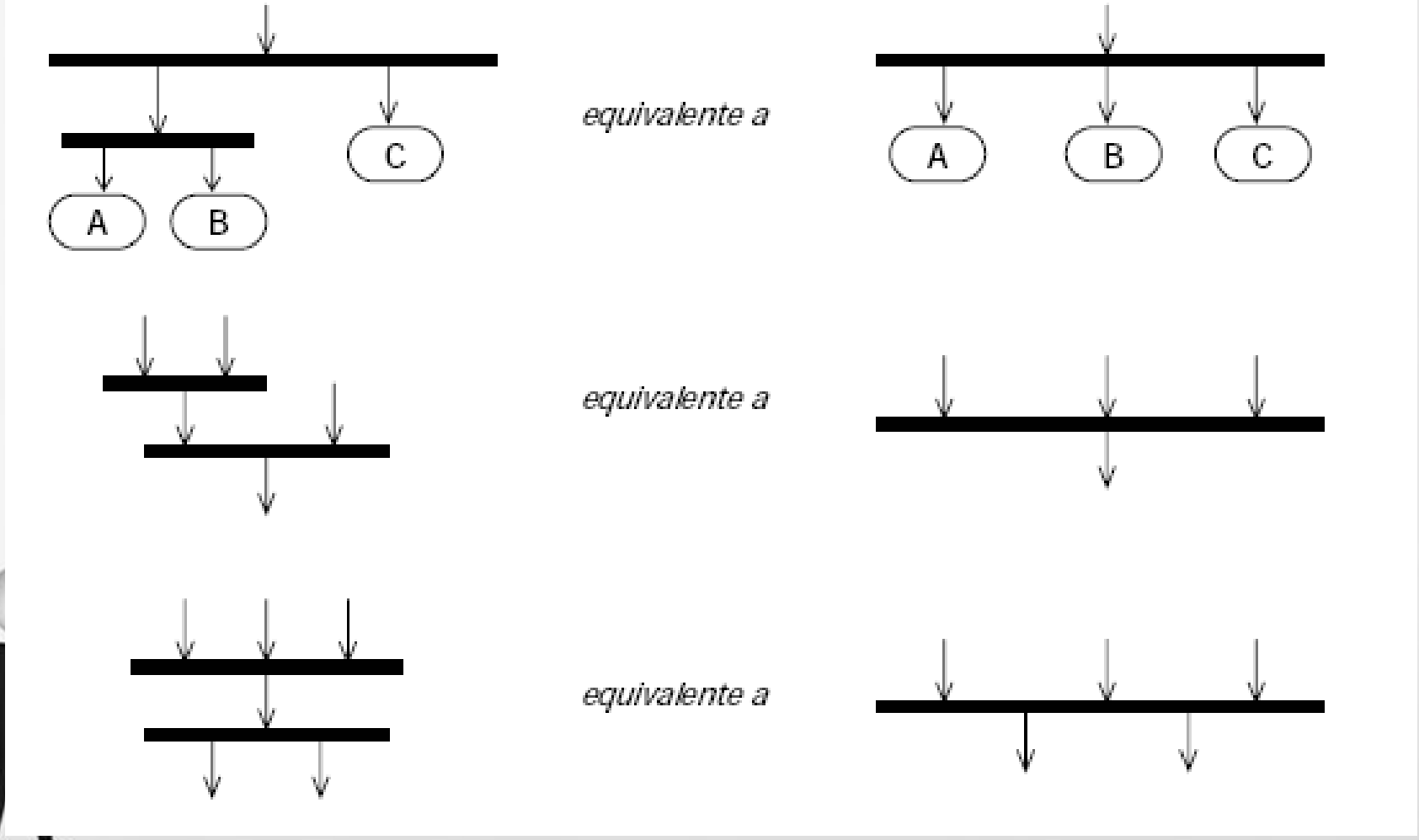


Activity Diagram: Fork e Join

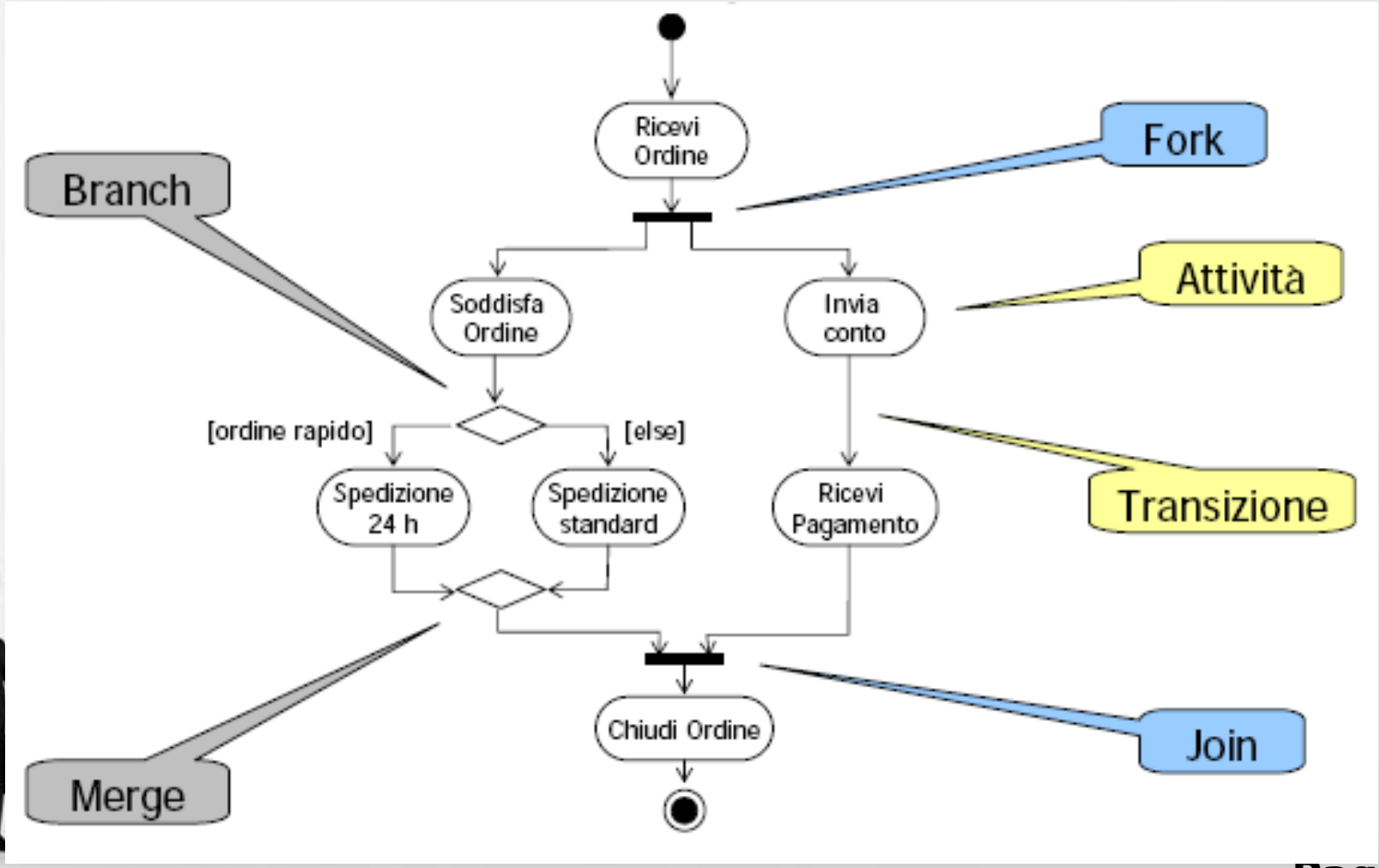
- **Fork**: divisione con una transizione entrante e più di una uscente che si eseguono in parallelo
- **Join**: unione con più transizioni entranti ed una sola uscente che può scattare solo dopo che sono terminate le attività degli stati corrispondenti alle transizioni entranti
- **Fork e Join si devono corrispondere!**



Combinazione di Fork e Join



Activity diagram: Esempio

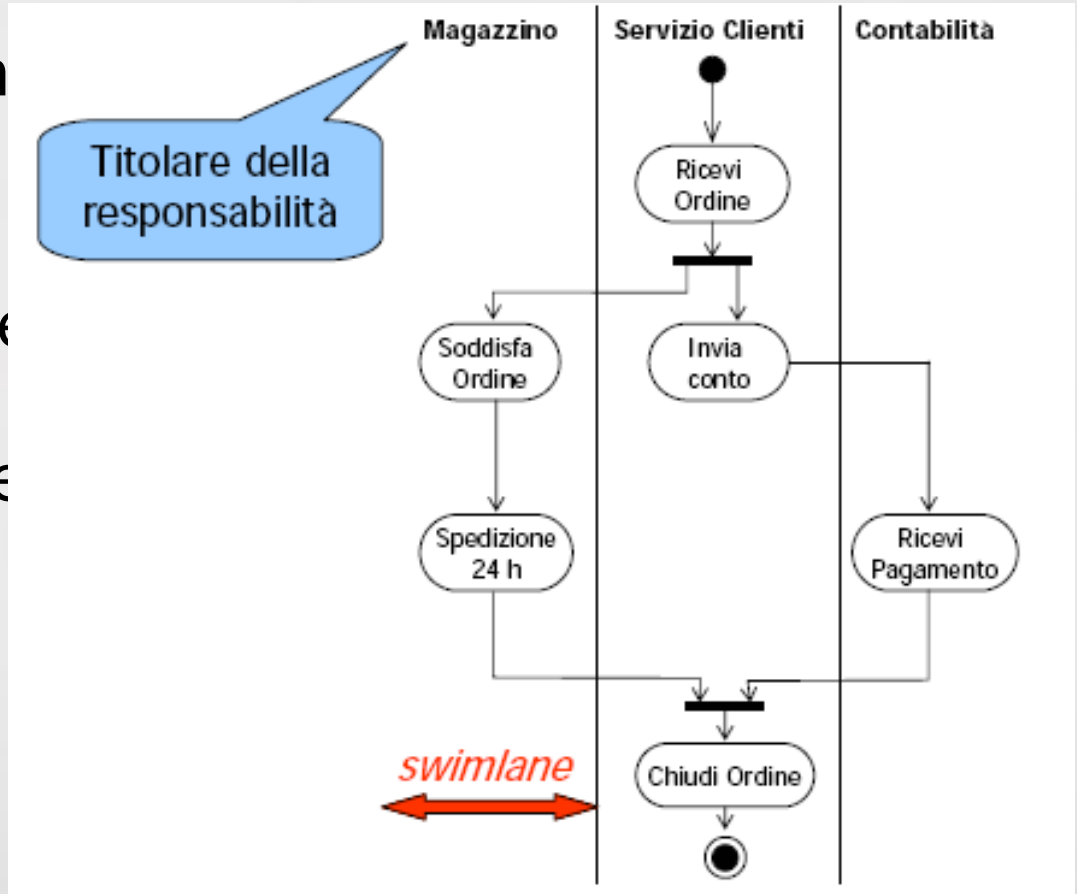


Swimlanes

- I diagrammi delle attività documentano bene ciò che accade,ma non chi fa che cosa
- Un modo di risolvere il problema, indicando le responsabilità per le attività, è l'uso delle **swimlanes**
- Le swimlanes sono corsie verticali separate da linee continue, in cui ogni corsia è coperta dalla responsabilità di una particolare classe (o sottosistema)

Swimlanes

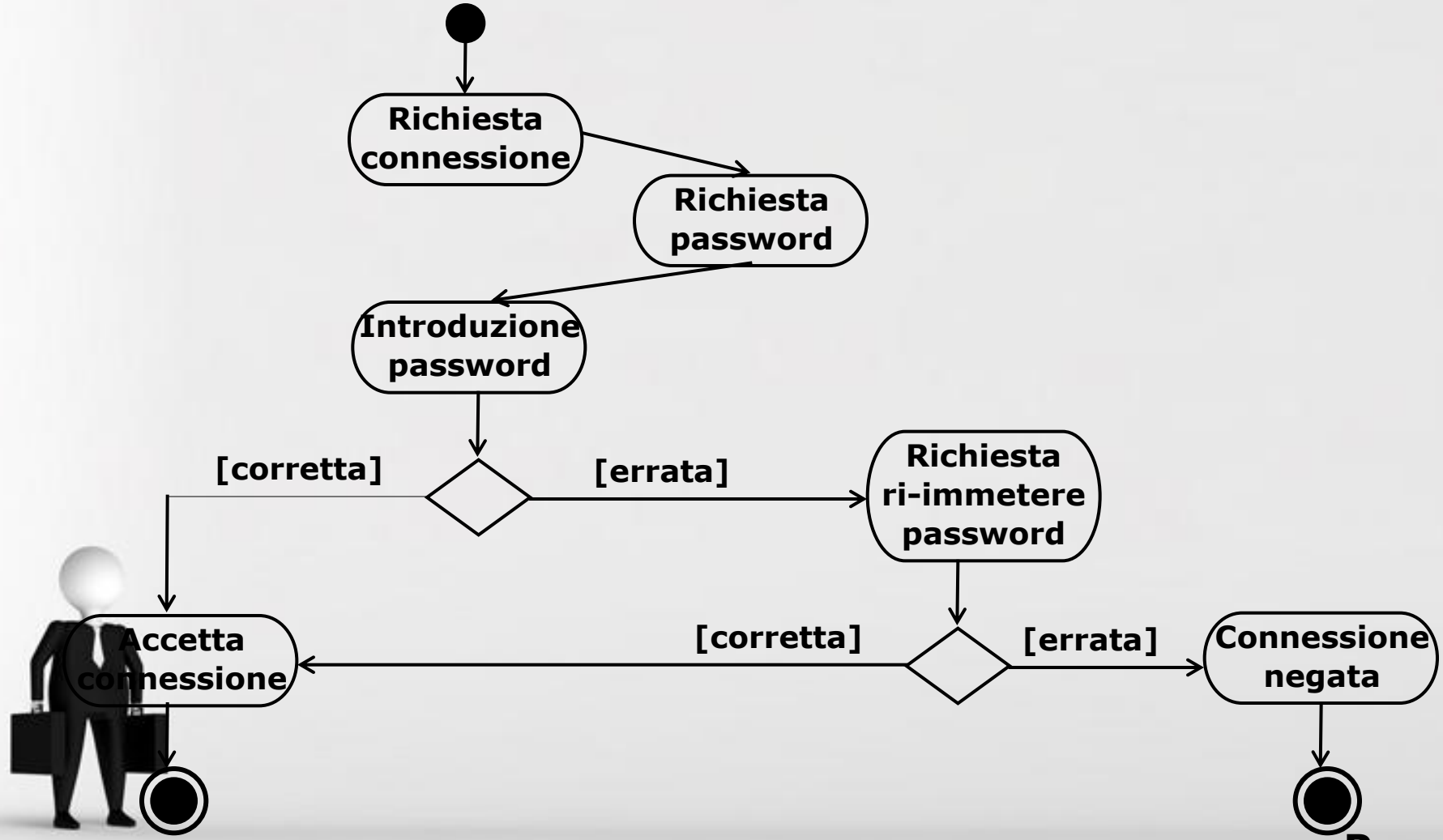
- Ogni attività deve essere contenuta in una singola swimlane
- Le transizioni invece possono attraversare le linee verticali di demarcazione



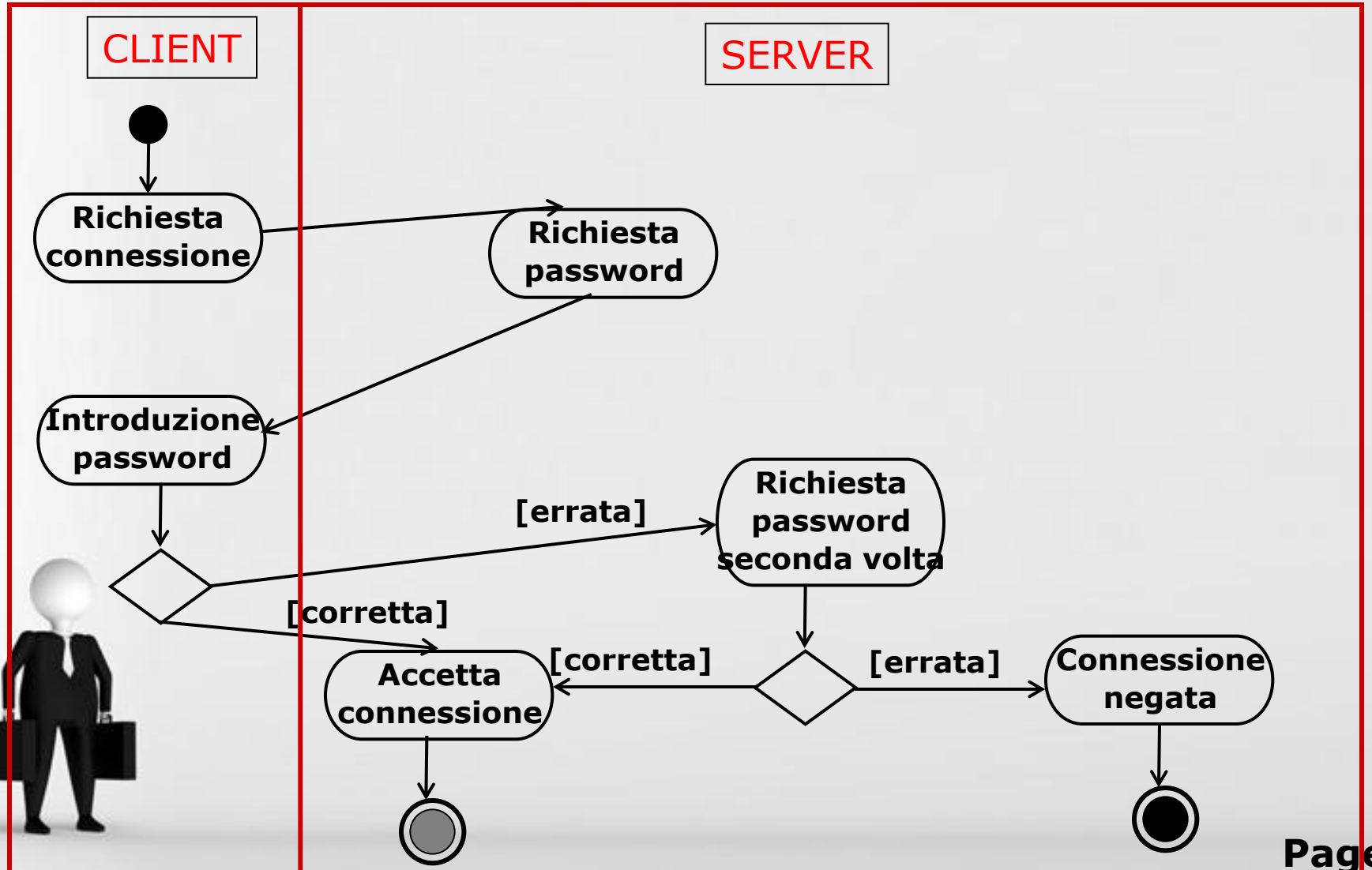
Esercizio

- Sistema di autenticazione ad un servizio:
  - Richiesta password
  - Verifica password

Esercizio: soluzione(1)



Esercizio: soluzione(2)



Design Patterns

Perchè

I problemi incontrati nello sviluppare grossi progetti software sono spesso ricorrenti e prevedibili.

- I design pattern sono schemi utilizzabili nel progetto di un sistema.
- Permettono quindi di non inventare da capo soluzioni ai problemi già risolti, ma di utilizzare dei "mattoni" di provata efficacia.
- Inoltre, un bravo progettista sa riconoscerli nella documentazione o direttamente nel codice, e utilizzarli per comprendere in fretta i programmi scritti da altri. Quindi:
  - forniscono un vocabolario comune che facilita la comunicazione tra progettisti;
  - svantaggio potenziale: possono rendere la struttura del codice più complessa del necessario. Di volta in volta bisogna decidere se adottare semplici soluzioni ad hoc o riutilizzare pattern noti;
  - pericolo di "overdesign".

Vantaggi

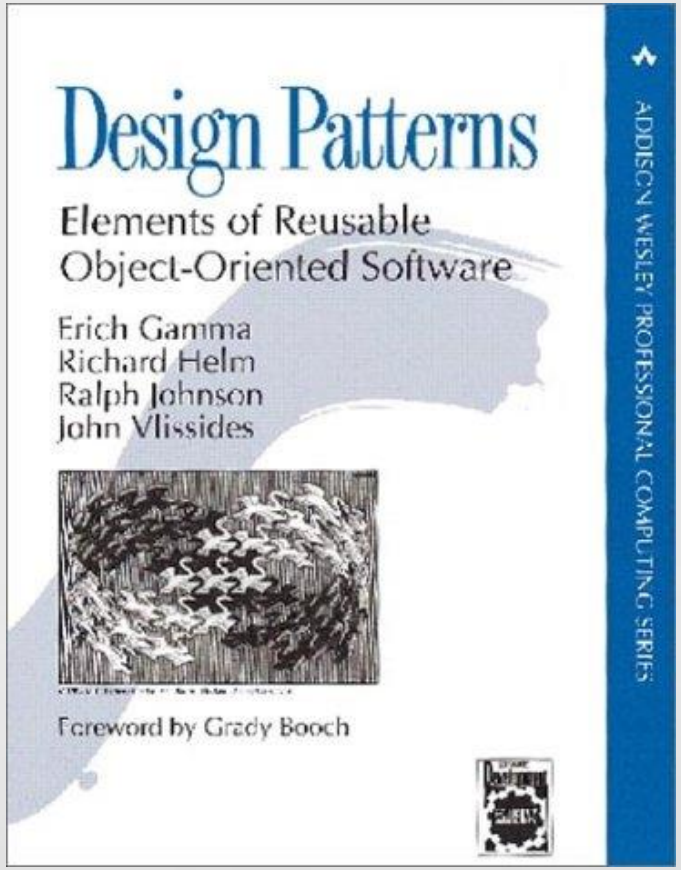
- Notevole aumento della capacità di produrre software riutilizzabile;
- Si danno allo sviluppatore strumenti utili per la modellazione di nuovi sistemi;
- Si aumenta la documentazione e la chiarezza;
- Si aumenta la velocità di sviluppo;
- Si aumenta la robustezza del software;
- Si aumenta la flessibilità e l'eleganza del software.

Pattern

- Il termine "pattern" fu introdotto dall’architetto austriaco Christopher
- Alexander negli anni ’70 (per la pianificazione di costruzioni in ambienti urbani).
- Nel 1987 Cunningham e Beck adattarono l’idea di Alexander per guidare programmatori inesperti in Smalltalk.
- Erich Gamma, tesi di dottorato, 1988-1991.
- Dal 1990 al 1992 la famosa Gang of Four (Gamma, Helm, Johnson,
- Vlissides) incominciò la stesura di un catalogo di pattern.
- Nel 1995 la Gang of Four pubblicò "Design Patterns - Elements of Reusable Object-Oriented Software" con la descrizione di 23 pattern.

Gang of Four

- E' il libro da cui è nato tutto, l'unica Bibbia
- Le figure sono riprese da qui



AntiPattern

La definizione di antipattern è stata coniata dalla "Gang of Four" per indicare i tipici problemi che incorrono i programmatori nella scrittura del codice. Vi è un elenco esteso di "trappole" in cui cade il programmatore, eccone alcune:

- Coltellino svizzero quando il programmatore prevede un numero elevato di funzionalità, la maggior parte inutile.
- Fede cieca quando il programmatore presume la correttezza di un codice e non prevede alcun controllo.
- Azione a distanza quando elementi del programma che interagiscono sono posti a distanza non controllando gli effetti della modifica di una parte sull'altra.
- Reinventare la ruota quando il programmatore rinuncia ad adattare un modulo esistente riscrivendolo da capo e inserendo, presumibilmente, errori.
- Spaghetti code quando si un uso eccessivo di costrutti per il controllo del flusso rendendo praticamente illeggibile un codice e propenso agli errori.
- Secondo gli autori, l'uso dei pattern contribuisce nell'evitare queste trappole.

Definizione

Un pattern è l'astrazione di un problema che si verifica nel nostro dominio, rappresentandone la soluzione in modo che sia possibile riutilizzarla per numerosi altri contesti (Christopher Alexander).

Descrizione di classi ed oggetti comunicanti adatti a risolvere un problema progettuale generale in un contesto particolare. (Gamma, Helm, Johnson, Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley).

Struttura di un pattern

Nel libro dei GoF ogni pattern è descritto nel seguente modo:

- **Descrizione**: una breve descrizione dell'obiettivo del pattern.
- **Esempio**: si presenta un problema la cui soluzione si ottiene tramite l'applicazione del pattern.
- **Descrizione della soluzione offerta dal pattern**: si descrive testualmente l'architettura del pattern e come questa si applica al problema.
- **Struttura del pattern**: diagramma di classi in UML della struttura generica del pattern.
- **Applicazione del pattern**: offre un diagramma UML delle classi del problema, presenta l'abbinamento delle classi del problema con le classi che descrivono la struttura concettuale del pattern, descrive l'implementazione del codice Java, presenta e commenta gli output dell'esecuzione.
- **Osservazioni sull'implementazione in Java**: presenta gli aspetti particolari che riguardano l'implementazione del pattern in Java.

Proprietà

I pattern:

- Costituiscono un vocabolario comune per i progettisti;
- Sono una notazione abbreviata per comunicare efficacemente principi complessi;
- Aiutano a documentare l'architettura software;
- Catturano parti critiche di un sistema in forma compatta;
- Mostrano più di una soluzione;
- Descrivono astrazioni software;
- NON costituiscono una soluzione precisa di problemi progettuali;
- NON risolvono tutti i problemi progettuali;
- NON si applicano solo alla progettazione OO, ma anche ad altri domini.

Nome

Per identificare un pattern si utilizza un nome.

- Il nome del pattern è molto utile per descrivere il problema, la sua soluzione ed il suo uso;
- Esso è composto da una o due parole;
- Bisogna cercare di omogeneizzare i vocabolari personali di tutti i colleghi.

Problema

Un pattern si applica per risolvere un problema che si presenta nella fase di modellazione.

- Descrive quando applicare un pattern definendo il contesto ed il dominio di appartenenza;
- In generale include la lista di condizioni che devono essere valide per poter giustificare l'uso di un determinato pattern.

Soluzione

Il pattern coinvolge componenti con particolari vincoli.

- Descrive gli elementi che verranno usati durante la modellazione;
- Descrive le relazioni e le responsabilità degli elementi;
- E’ importante capire che la soluzione non rappresenta una specifica implementazione o caso d’uso ma un modello che si applica a differenti situazioni.

Conseguenze

L’applicazione di un pattern mostra vantaggi e svantaggi.

- Raccoglie l’elenco dei tempi e dei risultati;
- E’ importante quando si devono prendere decisioni di modellazione;
- Descrive varie metriche, i costi ed i tempi in relazione ai benefici che il pattern introdurrebbe.

Scelta

Esistono numerosi pattern più o meno adatti al problema da modellare.

- Esistono numerosi cataloghi di pattern;
- Solitamente sono descritti attraverso una notazione comune "Design Language";
- E’ importante reperire il pattern adeguato per il proprio specifico dominio;
- Considerare come un pattern risolve il problema: ogni pattern affronta il probelma con una soluzione originale;
- Considerare il suo intento: l’obiettivo del programma deve essere lo stesso del pattern;
- Studiare le interazioni tra pattern: i pattern generalmente sono composizionali;
- Considerare come deve variare il progetto: diversi approcci con diversi pattern.

In java !?

I pattern sono utilizzati abbondantemente dalle classi standard di Java.

Iteraror, Observer e molti altri sono stati introdotti già dalle prime versioni.

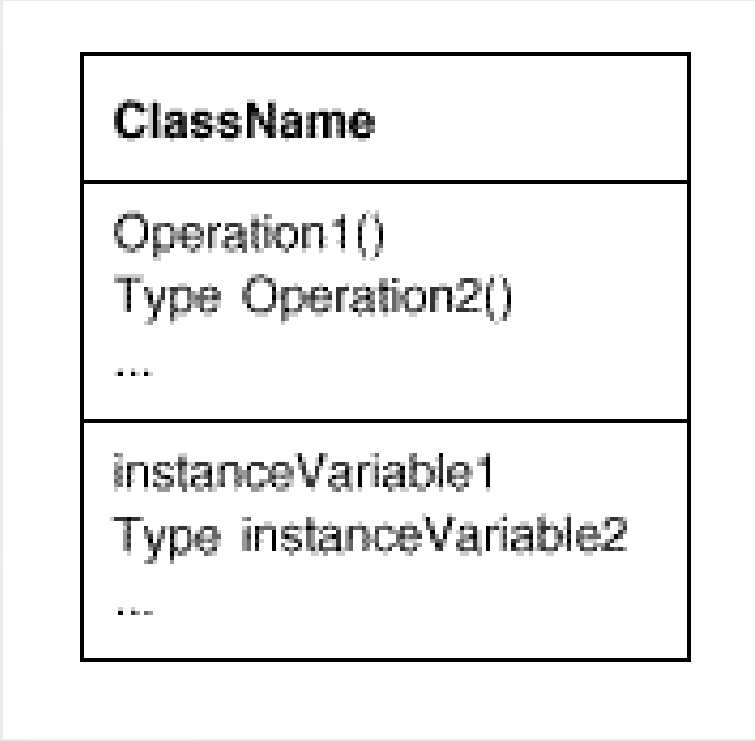
I pattern calzano perfettamente e traggono i maggiori benefici dal polimorfismo e dall'ereditarietà dei linguaggi di programmazione orientati agli oggetti.

Notazione: Object Modeling Technique

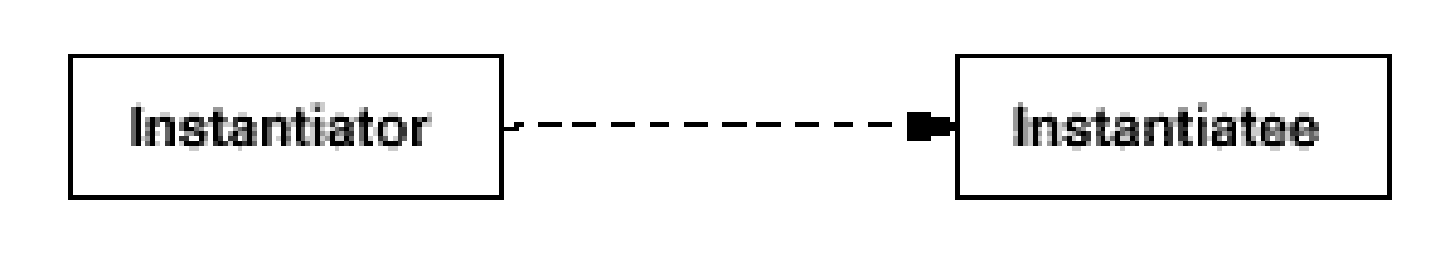
In pratica UML:

- Diagramma delle classi
- Diagramma degli oggetti
- Diagrammi di iterazione

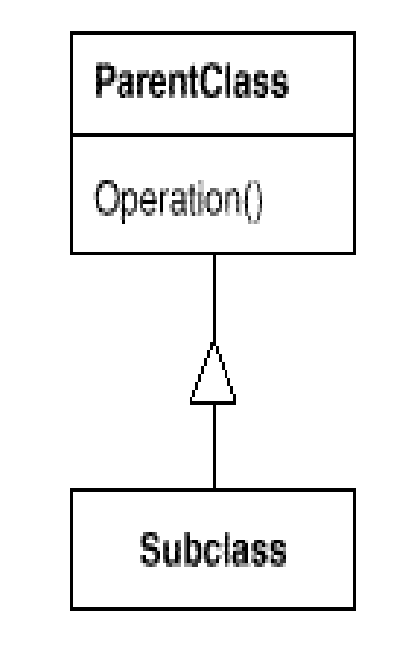
Classi

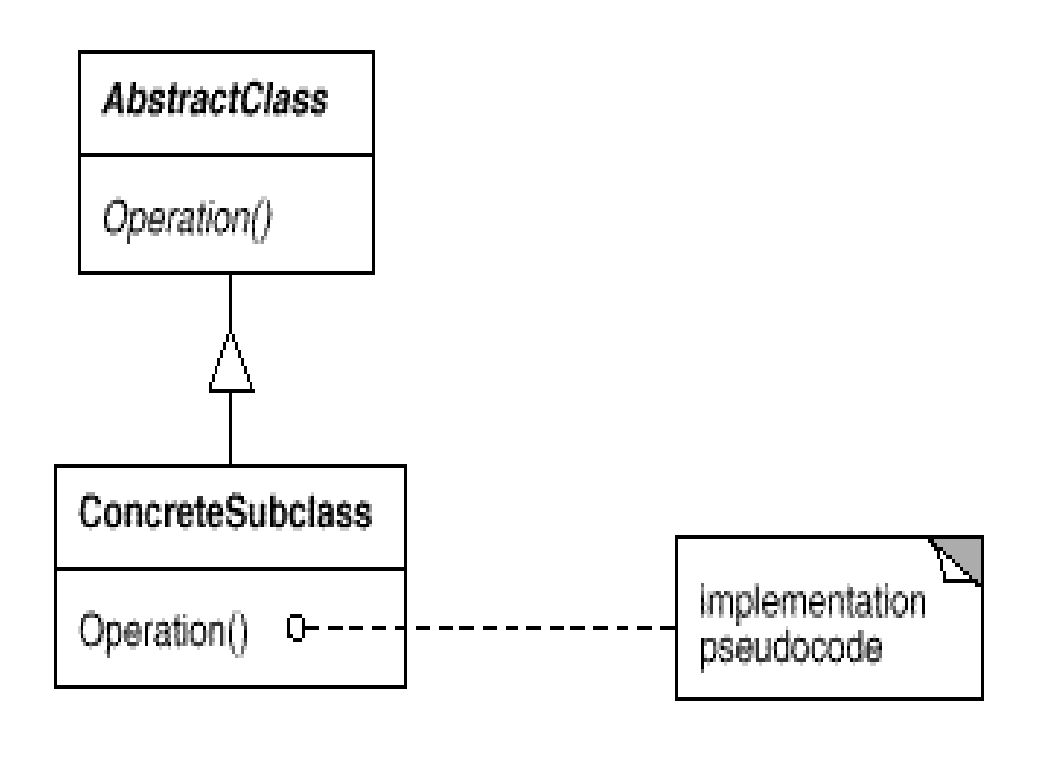

ClassName
Operation1()
Type Operation2()
...
instanceVariable1
Type instanceVariable2
...

Istanzazione oggetti



Gerarchie e classi astratte





Pseudo codice e contenitori

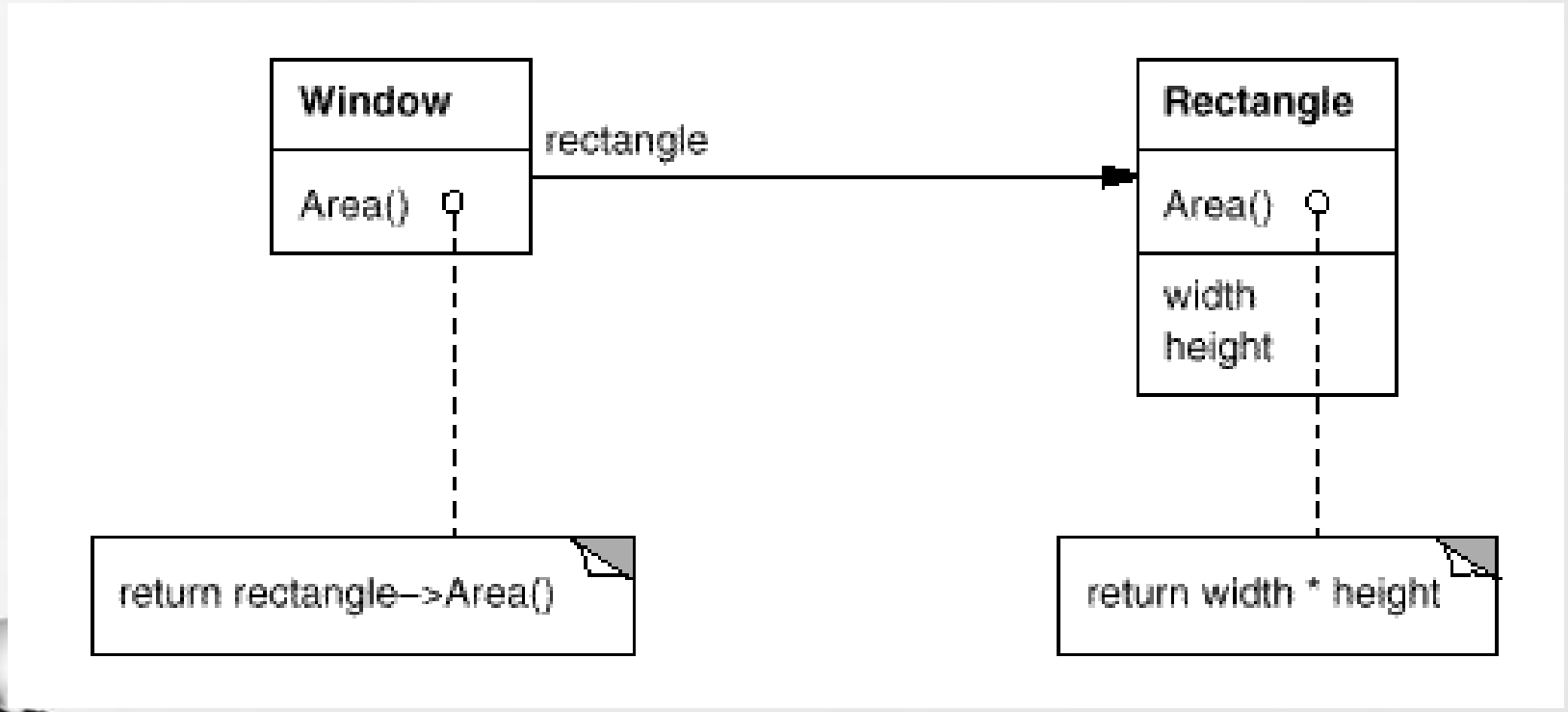


Diagramma degli oggetti

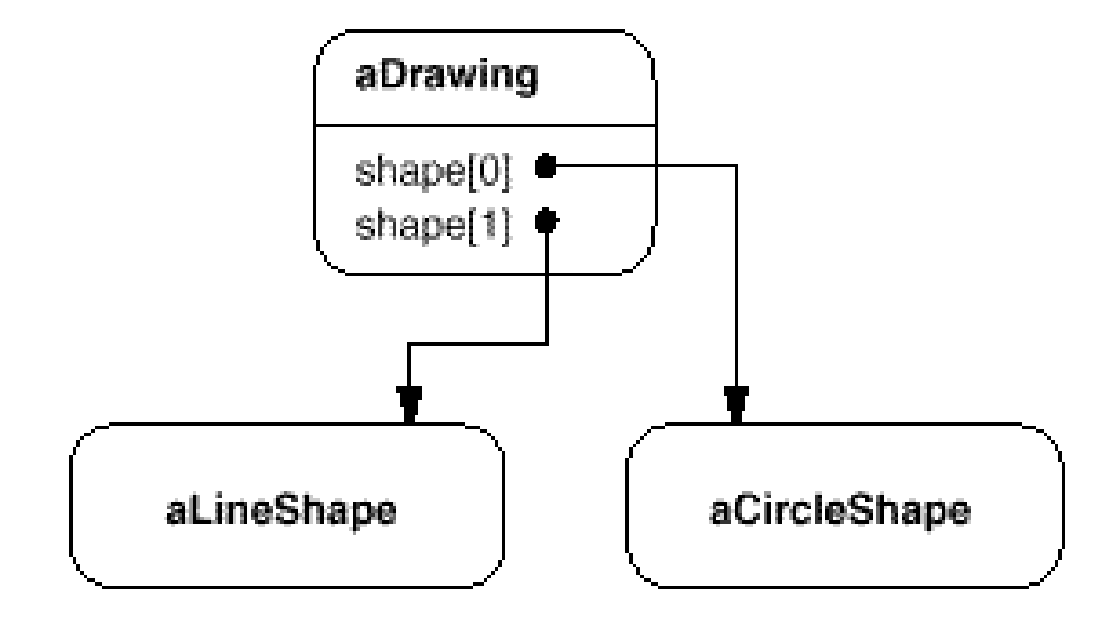
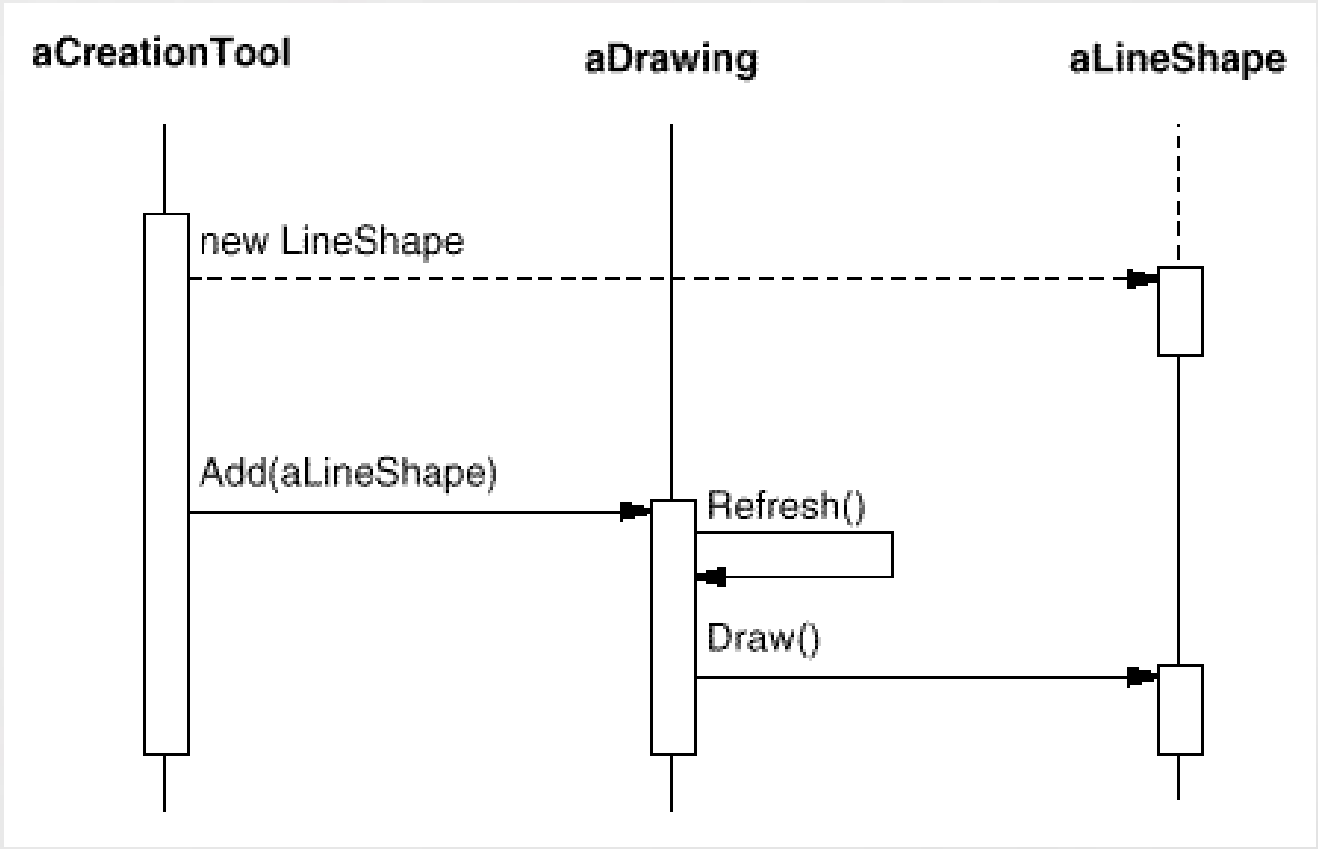


Diagramma di iterazione



Principi sottostanti

Basarsi su classi astratte per nascondere le differenze fra le sottoclassi dei client

Classe dell'oggetto vs. Tipo del oggetto

- Le classi definiscono come un oggetto è implementato
- I tipi definiscono l'interfaccia del oggetto
- Programmare verso un interfaccia non verso un implementazione

Principi sottostanti

- Black-box vs. white-box
  - Il riuso mediante black-box si basa sui riferimenti degli oggetti, generalmente attraverso l'istanziane delle variabili
    - Preferito in contesto di information hiding, flessibilità al runtime, eliminazione delle dipedneze implementatitive
    - L'efficienza run-time ne soffre Un elevato numero di istanze e overhead di comunicazione

- Favorire la composizione anzichè l'eriditarietà

Principi sottostanti

- Delegare
  - Tecnica potente quando combinata con black box reuse
    - Permettere la delgazione di differenti istanza a run-time, fintantochè le istanze rispondo a messaggi simili
  - Però:
    - Quaclhe volta il codice è difficle da leggere
      - Efficienza perchè uso Black box

Tassonimia

I design patterns possono essere raggruppati secondo il pricipale contesto di applicazione in:

- Pattern di creazione delegati alla gestione della costruzione di oggetti;
- Pattern di struttura delegati alla rappresentazione di oggetti;
- Pattern di comportamento delegati al comportamento dinamico degli oggetti.

Tassonomia

Creazionali	Pattern di creazione di nuove istanze
Strutturali	Pattern di strutturazione dati/comunicazione
Comportamentali	Gestione del comportamento negli algoritmi

Creazionali

- **Abstract Factory** - Crea oggetti appartenenti a famiglie di classi senza specificare le classi concrete.
- **Builder** - Separa il processo di creazione di un oggetto dalla rappresentazione definitiva.
- **Factory Method** - Crea oggetti derivanti da diversi tipi di classi.
- **Prototype** - Creazione di oggetti a partire da altri oggetti.
- **Singleton** - Una classe di cui può esistere solo un singolo oggetto.

Strutturali

- **Adapter** - Realizza interfacce per differenti classi.
- **Bridge** - Separa l'interfaccia di un oggetto dalla sua implementazione.
- **Composite** - Oggetti composti da oggetti con la stessa struttura.
- **Decorator** - Aggiunge dinamicamente funzionalità ad un oggetto.
- **Facade** - Una singola classe che rappresenta un intero sottosistema.
- **Flyweight** - Una rappresentazione fine di istanze efficentemente condivise.
- **Proxy** - Un oggetto rappresenta un altro oggetto.

Comportamentali

- **Chain of Responsibility** - Un modo di passare richieste tra una catena di ggetti.
- **Command** - Incapsula richieste di comandi ad un oggetto.
- **Interpreter** - Un modo di includere elementi di linguaggio in un programma.
- **Iterator** - Scansione degli elementi di una collezione.
- **Mediator** - Definisce un modo semplificato di comuncazione tra classi.
- **Memento** - Congela e ripristina lo stato interno di una classe.
- **Observer** - Un modo di notificare cambiamenti ad un insieme di classi.
- **State** - Modifica del comportamento degli oggetti al cambiamento dello stato.
- **Strategy** - Incapsulamento di algoritmi nelle classi.
- **Template Method** - Rimanda l'esatto passo di elaborazione di un algoritmo ad una sottoclasse.
- **Visitor** - Aggiunge una nuova operazione senza cambiare la classe.