Università degli Studi di Verona
Dipartimento di Informatica
A.A. 2018-2019

APPUNTI DI “INGEGNERIA DEL SOFTWARE”

Creato da *Davide Zampieri*

DESIGN PATTERN

Singleton: usato per assicurare che una classe abbia una sola istanza; le classi Singleton vengono progettate con costruttori privati; un metodo statico restituisce il riferimento all'unica istanza esistente.

Observer: quando un oggetto (osservato) cambia stato, tutti quelli che ne dipendono (osservatori) vengono automaticamente notificati del fatto ed aggiornati di conseguenza (cioè si sincronizzano con lo stato dell'osservato).

Template: definisce la struttura di un algoritmo all'interno di un metodo, delegando alcuni passi dell'algoritmo alle sottoclassi; tali sottoclassi ridefiniscono alcuni passi dell'algoritmo senza dover implementare di nuovo la struttura dell'algoritmo stesso.

Decorator: consente di aggiungere nuove funzionalità ad oggetti già esistenti; una classe decoratore (wrapper) avvolge l'oggetto originale passandolo al suo costruttore.

Facade: un oggetto (facciata) permette, attraverso un'interfaccia più semplice, l'accesso a sottosistemi che espongono interfacce complesse e molto diverse tra loro; nasconde (maschera) la complessità del sistema.

Factory: indirizza il problema della creazione di oggetti senza specificarne l'esatta classe; fornisce un'interfaccia per creare degli oggetti, ma lascia che le sottoclassi decidano quale oggetto istanziare.

Abstract Factory: fornisce un'interfaccia per creare famiglie di oggetti connessi o dipendenti tra loro (factory di factory), in modo che non ci sia necessità da parte dei client di specificare i nomi delle classi concrete all'interno del proprio codice; in questo modo si permette che un sistema sia indipendente dall'implementazione degli oggetti concreti e che il client, attraverso l'interfaccia, utilizzi diverse famiglie di prodotti.

Iterator: usato per scorrere sequenzialmente gli elementi di una collezione senza dover conoscere la sua rappresentazione sottostante.

Proxy: una classe funziona come interfaccia per un'altra; viene creata un'istanza di un oggetto complesso e molteplici oggetti proxy, in modo che ogni operazione svolta sui proxy venga trasmessa all'oggetto originale.

USE CASE

I *diagrammi dei casi d'uso* individuano chi ha a che fare con il sistema (attore) e che cosa l'attore può fare (caso d'uso). I *requisiti funzionali* specificano cosa deve essere fatto. I *casi d'uso* modellano i requisiti funzionali perché specificano cosa ci si aspetta da un sistema ma nascondono come il sistema lo implementa.

Include: è una dipendenza tra casi d'uso; il caso incluso fa parte del comportamento di quello che lo include; l'inclusione non è opzionale ed avviene in ogni istanza del caso d'uso; la corretta esecuzione del caso d'uso che include dipende da quella del caso d'uso incluso; usato per riutilizzare parti comuni a più casi d'uso.

Extend: è una dipendenza tra casi d'uso (è importante il verso della freccia); il caso d'uso che estende specifica un incremento di comportamento a quello esteso; si tratta di un comportamento supplementare ed opzionale.

ACTIVITY DIAGRAM

I *diagrammi delle attività* modellano un comportamento (che riguarda una o più entità) come un insieme di azioni organizzate secondo un flusso. L'attività può essere relativa ad un qualsiasi *oggetto* e quindi si possono utilizzare i diagrammi delle attività per: modellare il flusso di un caso d'uso, modellare il funzionamento di un'operazione di classe o di un componente.

SEQUENCE DIAGRAM

I *diagrammi di sequenza* modellano le interazioni (scambio di messaggi nel tempo) tra varie entità di un sistema. Il loro scopo è mostrare come un certo *comportamento* (un caso d'uso o un'operazione di classe) viene realizzato dalla collaborazione delle *entità* in gioco (attori o istanze di classe).

PROGRAMMAZIONE AD OGGETTI

La programmazione ad oggetti prevede di raggruppare in una zona circoscritta del codice sorgente (chiamata *classe*), la dichiarazione delle strutture dati e delle procedure che operano su di esse. Le classi, quindi, costituiscono dei modelli astratti, che a tempo di esecuzione vengono invocate per istanziare o creare *oggetti* software relativi alla classe invocata. Questi ultimi sono dotati di *attributi* (dati) e *metodi* (procedure) secondo quanto definito/dichiarato dalle rispettive classi.

Un linguaggio di programmazione è definito ad oggetti quando permette di implementare tre meccanismi usando la sintassi nativa del linguaggio:

- *Incapsulamento*, che consiste nella separazione della cosiddetta interfaccia di una classe dalla corrispondente implementazione, in modo che i client di un oggetto di quella classe possano utilizzare la prima, ma non la seconda; in sostanza è un meccanismo atto a limitare l'accesso diretto agli elementi dell'oggetto (information hiding).
- *Ereditarietà*, che permette essenzialmente di definire delle classi a partire da altre già definite, realizzando una gerarchia di classi; una classe derivata attraverso l'ereditarietà (sottoclasse) mantiene i metodi e gli attributi della classe da cui deriva (superclasse) ed inoltre può definire i propri metodi o attributi, e ridefinire il codice di alcuni dei metodi ereditati tramite un meccanismo chiamato overriding.
- *Polimorfismo*, che permette di scrivere un client che può servirsi di oggetti di classi diverse, ma dotati di una stessa interfaccia comune; a tempo di esecuzione, quel client attiverà comportamenti diversi senza conoscere a priori il tipo specifico dell'oggetto che gli viene passato; in sostanza lo stesso codice eseguibile può essere utilizzato con istanze di classi diverse, aventi una superclasse comune.