Ingegneria Del Software

Stefano Perrini e Alessio Iodice

September 12, 2018

Contents

Chapter 1

Disclaier

Questa dispensa, non serve a sostituire i libri di testo né si pone come scorciatoia in vista della preparazione dell'esame di Ingegneria del Software tenuto dal Prof. Sergio Di Martino che cito testualmente:
"è un ottimo canovaccio per sapere cosa approfondire, ma solo con questa non si passa l'esame!"

Sergio Di Martino

Chapter 2

Requirements Engineering

2.1 Requirements types

I requisiti di un sistema sono la descrizione dei servizi che il sistema deve fornire e dei suoi vincoli. Esistono due tipologie di requisiti:

1. Requisiti dell'utente: descrivono, nel linguaggio naturale, quali servizi il sistema dovrebbe fornire e i vincoli del sistema.

2. Requisiti di sistema: sono descrizioni più dettagliate delle funzioni, servizi e dei vincoli del sistema software. Il documento dei requisiti dovrebbe definire cosa deve essere implementato.

I vari tipi di lettori dei documenti sono esempi di **stakeholder** del sistema.
Gli stakeholder includono chiunque sia influenzato dal sistema e chiunque abbia un interesse verso di esso, ad esempio : utenti finali, manager, persone esterne, etc.

I requisiti dei sistemi sono divisi in:

1. Requisiti funzionali: sono definizioni di servizi che il sistema deve fornire; indicano come il sistema dovrebbe reagire a particolari input e come dovrebbe comportarsi in particolari situazioni.

2. Requisiti non funzionali: sono requisiti che non riguardano direttamente specifici servizi forniti dal sistema. Possono riferirsi a proprietà del sistema, come l'affidabilità, i tempi di risposta e l'uso della memoria. Si applicano al sistema completo.

3. Requisiti di dominio: riflettono caratteristiche generali del dominio dell'applicazione. Se questi requisiti non sono soddisfatti, potrebbe essere impossibile far funzionare il sistema correttamente.

2.2 Raccolta, specifica e validazione dei requisiti

2.3 Question & Answers

1. Cosa si intende con il termine "Stakeholders"? In quale/i fase/i del ciclo di vita è rilevante?

   Per Stakeholders si intendono le parti coinvolte durante la raccolta di requisiti. Includono chiunque sia influenzato dal sistema e chiunque abbia un interesse verso di esso (utenti finali, manager, persone esterne).

Chapter 3

Software processes

3.1 Software process models

3.1.1 Waterfall model

In questo modello il processo di sviluppo del software è costituito da un certo numero di stadi. I principali stadi sono quelli fondamentali di un software:

1. Analisi e definizione dei requisiti.
2. Progettazione del sistema e del software.
3. Implementazione e test delle unità.
4. Integrazione e test del sistema.
5. Operatività e manutenzione.

In teoria, il risultato di ogni stadio è costituito da uno o più documenti approvati (*signed off*). In realtà, il processo software richiede una sequenza di feedback da uno stadio all'altro: se emergono nuove informazioni, i documenti devono essere modificati. Perciò, sia i clienti che gli sviluppatori potrebbero congelare prematuramente la specifica del software, ciò potrebbe portare a sistemi mal strutturati.
Il modello a cascata è appropriato per alcuni tipi di sistemi, come i sistemi integrati e i sistemi critici, al contrario, non è appropriato quando i requisiti del software cambiano rapidamente.

3.1.2 Incremental development

Lo sviluppo incrementale si basa sull'idea di sviluppare un'implementazione iniziale, esporla agli utenti e perfezionarla attraverso molte versioni, finché non si ottiene il sistema richiesto. In questo modello, le attività di specifica, sviluppo e convalida sono intrecciate tramite feedback.

Lo sviluppo incrementale è una parte fondamentale dei metodi di sviluppo agile, ed offre vantaggi rispetto al modello a cascata:

1. Il costo di implementazione delle modifiche è ridotto.
2. E' più facile ottenere feedback dal cliente.
3. E' possibile consegnare in anticipo al cliente una versione utilizzabile del software.

D'altra parte, l'approccio incrementale ha due problemi:

1. Il processo non è visibile all'esterno.
2. Il sistema tende a degradarsi all'aggiunta di nuovi incrementi.

3.2 Software changes

3.2.1 System prototyping

Una versione del sistema o una sua parte vengono sviluppate rapidamente per verificare i requisiti del cliente e la fattibilità. Questo è un metodo di anticipazione dei cambiamenti, in quanto consente agli utenti di provare il sistema prima della consegna, riducendo le proposte di modifica dopo la consegna.

3.2.2 Incremental delivery

Gli incrementi del sistema vengono consegnati al cliente per essere provati. Questo è un metodo di tolleranza ai cambiamenti, in quanto evita l'approvazione prematura dei requisiti per l'intero sistema e consente alle modifiche di essere incluse in successivi incrementi.

3.3 Agile methods

Caratteristiche comuni dei metodi agili:

1. I processi di specifica, progettazione e implementazione sono intrecciati. La documentazione dei progetti è minimizzata.
2. Il sistema viene sviluppato in una serie di incrementi. Gli utenti finali e gli stakeholder sono coinvolti nella specifica e nella valutazione di ogni incremento.
3. Molti strumenti che supportano il processo di sviluppo: test automatici, produzione automatica delle interfacce utente, etc.

I metodi agili sono indicati per sviluppare applicazioni nelle quali i requisiti del sistema cambiano rapidamente durante il processo di sviluppo. Consentono una consegna rapida del software e riducono la burocrazia nei processi di sviluppo.

3.3.1 SCRUM

Il metodo agile Scrum offre un framework per organizzare progetti agili e fornire una visibilità esterna su ciò che sta accadendo.
Il punto di partenza del processo Scrum, o ciclo degli sprint, è il product backlog (la lista degli elementi, come le caratteristiche del prodotto, i requisiti, etc), di cui si dovrà occupare il team di Scrum.
Il product owner ha la responsabilità di garantire che il livello di dettagli nella specifica sia appropriato al lavoro da svolgere, e, all'inizio di ogni ciclo, stabilisce le priorità del product backlog.
Ogni ciclo di sprint ha una durata prestabilita, compresa solitamente tra 2 e 4 settimane. Gli elementi che non possono essere completati vengono restituiti al product backlog.
Tutti i membri del team vengono coinvolti nella scelta degli elementi con priorità più alta; poi valutano il tempo richiesto per completarli.
Durante lo sprint, il team si riunisce ogni giorno per esaminare l'avanzamento del lavoro: eventuali problemi vengono condivisi. Alla fine di ogni sprint, si tiene una riunione di verifica per migliorare il processo e fornire il nuovo input per lo sprint successivo.
Lo ScrumMaster, da non confondere con il project manager, riferisce sull'avanzamento del lavoro al senior manager e prende parte alla pianificazione a lungo termine e al budget del progetto.
Punti di forza del metodo Scrum:

1. Il prodotto è suddiviso in parti gestibili e compresibili.
2. I requisiti instabili non fanno ritardare il processo.
3. L'intero team ha una visione su tutto.
4. I clienti ricevono in tempo gli incrementi e hanno un feedback su come funziona il prodotto.
5. C'è fiducia tra clienti e sviluppatori.

Chapter 4

UML

4.1 Class Diagram

4.1.1 Class Diagram to JAVA: One To Many

La relazione "uno a molti" tra due entità prevede che la Classe A in relazione 1 a Molti con la classe B (quindi 1 elemento di A ha più elementi della classe B, un elemento della classe B ha un riferimento nella classe A) possegga una "Collezione" di elementi che puntano alla classe in relazione.

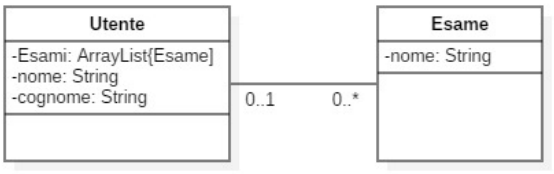

Figure 4.1: Esempio di Class Diagram di una classe Utente che può collezionare Esami

Listing 4.1: OneToMany

```
1 public class Utente {
2   private final ArrayList<Esame> esami = new ArrayList();
3   private final String nome;
4   private final String cognome;
5
6   public static addEsame(Esame e){
7     esami.add(e);
8   }
9 }
10 public class Esame {
11   private nome;
12 }
```

4.1.2 Class Diagram to JAVA: Composizione

La composizione è una relazione molto forte che collega la classe principale alle singole parti (classi) componenti (la distruzione della classe principale comporta la distruzione delle singole classi componenti). Nel costruttore della classe principale vengono istanziati i componenti che non avranno più ragione di esistere qualora l'oggetto "principale" dovesse essere eliminato. Di seguito un classico esempio

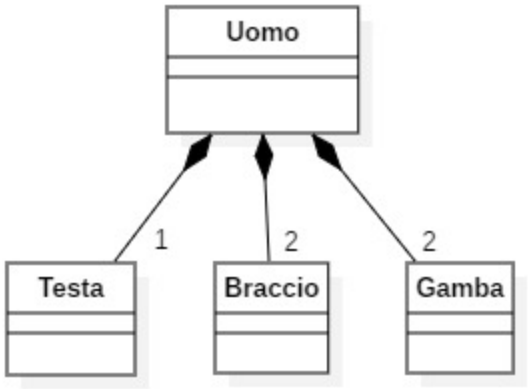

Figure 4.2: Esempio di Class Diagram di una classe Uomo e sottoclassi interne in "composizione"

Listing 4.2: Composizione

```
14 public class Uomo{
15   private Gamba[] gambe;
16   private Braccio[] braccia;
17   private Testa t;
18
19   public Uomo(){
20     this.t = new Test();
21     this.gambe = new Gamba[2];
22     this.braccio = new Braccio[2];
23     this.gambe[0] = new Gamba();
24     this.gambe[1] = new Gamba();
25     this.braccio[0] = new Braccio();
26     this.braccio[1] = new Braccio();
27   }
28
29   /**
30   * Classe Interna
31   *
32   */
33   public class Testa{
```

```
34      public Testa(){
35        // Classe Interna
36      }
37    }
38
39    /**
40    * Classe Interna
41    *
42    */
43    public class Gamba{
44      public Gamba(){
45        // Classe Interna
46      }
47    }
48
49    /**
50    * Classe Interna
51    *
52    */
53    public class Braccio{
54      public Braccio(){
55        // Classe Interna
56      }
57    }
58  }
```

4.1.3 Class Diagram to JAVA: Aggregazione

L'associazione è una relazione che collega due classi considerate indipendenti (ad esempio: preside scuola) **Aggregazione** e **composizione** sono le relazioni del tipo "part of" che legano entità non indipendenti.

Figure 4.3: Esempio di Class Diagram di una classe Uomo e sottoclassi interne in "composizione"

Listing 4.3: Aggregazione

```
59 public class Scuola{
60   private ArrayList<Studente> studenti;
61
62   public Scuola(){
63     this.studenti = new ArrayList<Studente>();
64   }
65
66   public void setStudente(Studente s){
67     this.studenti.add(s);
68   }
69
70 }
71
72 public class Studente{
73   public Studente(){
74
75   }
76 }
```

4.2 Sequence Diagram

4.3 Activity Diagram

4.4 State Charts Diagram

4.5 Question & Answers

1. Descrivere i simbolismi grafici degli statecharts diagrams di UML.

   Entry Point è rappresentato da un cerchio pieno End Point è rappresentato da un cerchio pieno iscritto in un cerchio vuoto. Una entità è rappresentata da un rettangolo con bordi arrotondati definito da un nome ed al suo interno è possibile inserire informazioni sulle operazioni che può svolgere o da trigger che lo possono attivare.

Chapter 5

Design & Architectural Pattern

Per utilizzare pattern all'interno di un progetto è necessario riconoscere se il problema di progettazione che si sta affrontando può essere associato ad un pattern da poter applicare. Esempio

1. Comunicare a molti oggetti che lo stato di alcuni altri oggetti è cambiato (Observer pattern)
2. Ordinare le interfacce ad un numero di oggetti correlati che sono stati spesso sviluppati incrementalmente (Facade Pattern)
3. Fornire un metodo standard di accesso a collezioni di elementi, a prescindere dal tipo di collezione implementata (Factory Pattern)
4. Permettere di estendere funzionalità di una classe esistente a run-time (Decorator Pattern)

5.1 Layers architectural patterns

Descrizione
Lo schema di architettura a strati è un modo per ottenere la separazione e l'indipendenza all'interno di un progetto. Organizza il sistema in vari strati, con funzionalità associate a ciascuno strato. Uno strato fornisce i servizi allo strato sopra di esso.
Si usa per costruire nuove funzioni per un sistema esistente; quando lo sviluppo è distribuito fra più team; quando c'è una richiesta di protezione su più livelli.
Vantaggi
I principali vantaggi di questa architettura sono portabilità e modificabilità.
Consente la sostituzione di interi strati, se l'interfaccia non viene modificata. Inoltre, se si modificano le interfacce di uno strato o vengono aggiunte nuove funzionalità, solo lo strato adiacente ne è influenzato.
Svantaggi
Spesso è difficile ottenere una netta separazione fra gli strati, e uno strato di alto livello potrebbe aver bisogno di interagire con strati di livelli inferiori.

Esempio di architettura a strati a 3 livelli:

- Livello 1: gestione dei dati (database).
- Livello 2: logica aziendale, funzionalità dell'applicazione (processamento dati).
- Livello 3: funzioni per l'interfaccia utente.

5.2 MVC Pattern

5.2.1 MVC: Model View Controller

E' un pattern architetturale per definire uno schema generale di riferiemento per la progettazione e strutturazione di applicazioni di tipo interattivo (attraverso una GUI).
Consente di separare e disaccoppiare il modello dei dati (model) e la logica applicativa (controller) dalle modalità di visualizzazione e interazione con l'utente (view).
L'applicazione deve separare i componenti software che implementano il modello delle informazioni, dai componenti che implementano la logica di presentazione e la logica di controllo.
L'uso di MVC permette di separare i ruoli dei componenti software al fine di ottenere

- Estendibilità: semplicità di progetto e decentralizzazione dell'architettura
- Riusabilità: possibilità di estrarre e riutilizzare componenti
- Interoperabilità: interazione tra moduli con ruoli differenti e possiblità di creare gerarchie tra i moduli

Il tutto favorisce la strutturazione del software e la relativa comprensibilità dei ruoli al fine di agevolare la manutenibilità del software.

Consideriamo l'esempio di un editor di pagine HTML

- Il **modello** è rappresentato dai caratteri che compongono la pagina: **classe HtmlPage**

- Possiamo immaginare diverse **viste** dei dati, una WYSIWYG **(Classe HtmlRenderedView)** ed una grezza WYSINWYG **(Classe HtmlRawView)**
- **Conctroller**, l'observer che riceve gli eventi dall'utente e produce modifiche sul modello **(classe InsertListener)**

La figura mostra il flusso di messaggi tipico del paradigma MVC.
otiamo che il **controller**, ricevendo un evento dalla **GUI**, come l'inserimento di un carattere da parte dell'utente, effettua una modifica del **Model**.

- Quindi, il controller non dialoga direttamente con le viste
- Nato per le interfacce grafiche, il paradigma MVC è stato poi applicato in altri domini, come le applicazioni complesse basate sul web.
- Molti framework per la realizzazione di applicazioni web si ispirano a MVC, ma lo hanno reinterpretato ed adattato alle loro esigenze

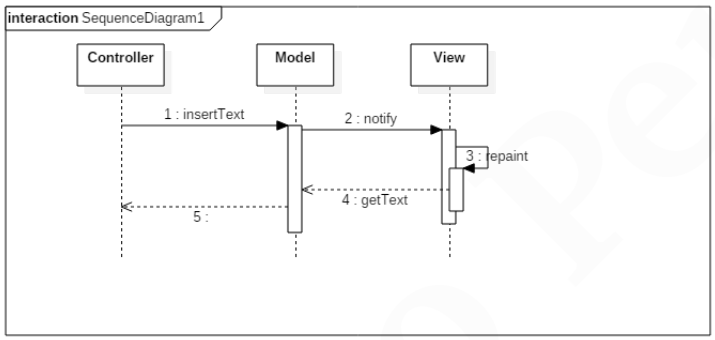

Figure 5.1: MVCExample: Editor di Testo

5.3 DAO Pattern

5.3.1 DAO: Data Access Object

E' una interfaccia usata per accedere ai dati in persistenza. Serve a separare l'interfaccia cliente di una risorsa dati, dal suo meccanismo di accesso ai dati.
Il problema di accedere direttamente ai dati è che la sorgente dei dati può cambiare e che questo comporterebbe

- Dover riscrivere parte dell'applicazione oppure introdurre delle logiche condizionali per gestire le differenze
- Creare un livello intermedio tra la logica di controllo e l'accesso ai dati

Tutto è riferito al DAO Pattern che consiste nei seguenti punti

- **Data Access Object Interface**: Queste interfacce definisce lo standard delle operazioni da eseguire sui modelli (oggetti).
- **Data Access Object concrete class**: Questa classe implementa l'interfaccia su menzionata. La classe è responsabile di prendere dati dalla persistenza (database, file etc).
- **Model Object or Value Object**: Questo oggetto è un semplice model contenente i metodi getter e setter per salvare o recuperare dati attraverso l'interfaccia DAO

Secondo il pattern, la classe Utente, si limiterà a contenere metodi accessori e modificatori per tutti gli attributi.

```
77 public class User (){
78   private int id;
79   private String nome;
80
81   public String getNome(){
82     return this.nome;
83   }
84
85   public void setNome(String nome){
86     this.nome=nome;
87   }
88   // getter e setter di id...
89 }
```

A questa classe si affianca un'interfaccia che contiene i metodi dedicati alla persistenza

```
90 public interface UserDAO (){
91   void createUser(User u){}
92   User getUserById(int id){}
93 }
```

In conclusione, per ogni meccanismo di persistenza concreto si ha una classe che implementa UserDAO.

```
94 public class MySQLDAO implements UserDAO{
95
96   public void createUser(User u){
97     PreparedStatement st=null;
98     String sql = "insert into users values (?)";
99     try {
100      st = DbConnection.getConnection().prepareCall(sql);
101      st.setString(1, u.getNome());
102      rs=st.execute();
103    } catch (SQLException ex) {
104      System.err.println("MySQLDAO createUser exception "+ex.getMessage());
105    }
106  }
107
108  public User getUserById(int id){
109    // String sql="select nome from users where id = ?"
110    // ...
111  }
112 }
```

In conclusione il pattern DAO si può presentare nella forma standard dei design pattern.

Contesto:

1. C'è bisogno di manipolare degli oggetti (entità del dominio) e assicurare la persistenza dei dati tra un'esecuzione e l'altra
2. Si vuole supportare diversi tipi di memoria persistente

Soluzione:

1. Creare una classe Entity che rappresenta l'oggetto del dominio con metodi accessiri e modificatori
2. Creare un'interfaccia EntityDAO con i metodi necessari a leggere e scrivere oggetti di tipo Entity su un supporto persistente
3. Per ogni meccanismo di persistenza supportato, creare una classe che implementa EntityDAO e realizza i suoi metodi sulla base di quel meccanismo

5.4 Observer Pattern

Descrizione

Si tratta di un pattern comportamentale basato su oggetti che viene utilizzato quando si vuole realizzare una dipendenza uno-a-molti in cui il cambiamento di stato di un soggetto venga notificato a tutti i soggetti che si sono mostrati interessati. Un esempio molto semplice è rappresentato dalle newsletters in cui gli utenti interessati a degli argomenti inseriscono il loro indirizzo email ed a fronte di novità inerenti gli argomenti, riceveranno una email di notifica. In questo modo viene applicata una gestione ad eventi, cioè al verificarsi di una notizia i soggetti interessati verranno informati tramite email. In questo modo l'interessato evita di fare polling, cioè evita di fare continue richieste al soggetto osservato per sapere se è avvenuto o meno un cambiamento ma al contrario verrà notificato in push dal soggetto osservato nel caso in cui dovesse interviene una modifica.
Questo pattern viene impegato in molte librerie, nei toolkit delle GUI e nel pattern architetturale MVC. Nel pattern MVC abbiamo la presenza di 3 soggetti: il Model, la View ed il Controller. Questi soggetti svolgono compiti diversi e tra di loro è presente una separazione di responsabilità c.d. "separation of concern". Ma c'è da dire che tra di loro esiste un forte legame in merito al cambiamento di stato.
In particolare il **Controller** è interessato ai cambiamenti di stato della **View**, mentre la **View** è interessata ai cambiamenti di stato del **Model**. Questo comporta che nel caso in cui dovessero avvenire dei cambiamenti il Model notifica alla View mentre la View notifica al Controller. Quindi il pattern Observer trova applicazione 2 volte nell'MVC su coppie di soggetti diversi (Model-View e View-Controller).
La View svolge un ruolo doppio poichè si trova ad essere *osservata* dal Controller e nello stesso tempo ad essere *osservatore* nei confronti del Model. A differenza del Model e del Controller che invece giocano un ruolo singolo, infatti il Model è *osservato* dalla View mentre il Controller è un *osservatore* della View.
Il ruolo di *osservatore* è il ruolo svolto da colui che si mostra interessato ai cambiamenti di stato, Observer. Il ruolo di osservato è il ruolo svolto da colui che viene monitorato, c.d. Subject o Observable.

Partecipanti e Struttura

Questo pattern è composto dai seguenti partecipanti:

1. **Subject**: espone l'interfaccia che consente agli osservatori di iscriversi e cancellarsi; mantiene una reference a tutti gli osservatori iscritti.

2. **Observer**: espone l'interfaccia che consente di aggiornare gli osservatori in caso di cambio di stato del soggetto osservato.

3. **ConcreteSubject**: mantiene lo stato del soggetto osservato e notifica gli osservatori in caso di un cambio di stato.

4. **ConcreteObserver**: implementa l'interfaccia dell'Observer definendo il comportamento in caso di cambio di stato del soggetto osservato

Conseguenze

Tale pattern presenta i seguenti vantaggi/svantaggi:

- **Astratto accoppiamento tra Subject e Observer:** il Subject sa che una lista di Observer sono interessati al suo stato ma non conosce le classi concrete degli Observer, pertanto non vi è un accoppiamento forte tra di loro.
- **Notifica diffusa:** il Subject deve notificare a tutti gli Observer il proprio cambio di stato, gli Observer sono responsabili di aggiungersi e rimuoversi dalla lista.

Contesto

- Un oggetto (soggetto) genera eventi
- Uno o più oggetti (osservatori) vogliono essere informati del verificarsi di tali eventi

Soluzione

- Definire un'interfaccia, chiamata Observer e dotata di un metodo notify, che sarà implementata dagli osservatori
- Il soggetto ha un metodo (chiamato attach) per registrare un osservatore
- Il soggetto gestisce l'elenco dei suoi osservatori registrati
- Quando si verifica un evento, il soggetto informa tutti gli osservatori registrati, invocando il loro metodo notify

La figura rappresenta il diagramma delle classi tipico del pattern **Observer**

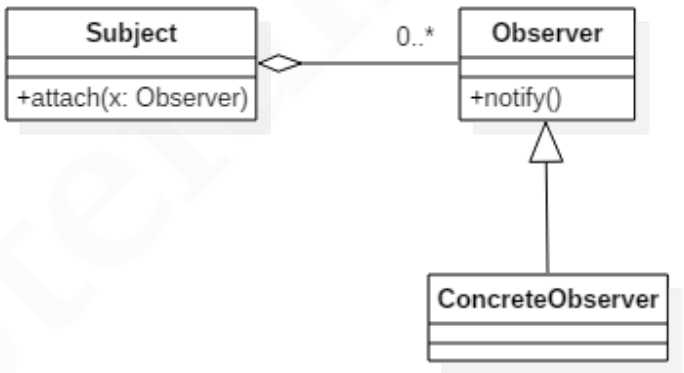

Figure 5.2: Diagramma UML del pattern Observer

Sono presenti:

- Classe Subject (soggetto osservato)
- Interfaccia Observer (osservatore)

Da notare la relazione di aggregazione **oneToMany** tra **Subject** e **Observer**

- tale relazione indica che ogni oggetto di tipo **Subject** conserva un insieme di riferimenti ad oggetti di tipo Observer.
- Ogni SUbject tiene traccia degli osservatori che si sono registrati chiamando "attach()"

5.5 Factory Pattern

Questo pattern si applica nel caso in cui diverse classi, chiamate *produttori*, che creano degli oggetti, chiamati *prodotti*, che sono di tipo diverso tra loro, ma sostanzialmente equivalente per chi li utilizza. Java offre una varietà di classi che rappresentano collezioni, come insiemi, liste etc. Per esaminare il contenuto di una collezione, si usa un oggetto chiamato iteratore, che viene creato dalla collezione. Un iteratore per un insieme è internamente diverso da un *iteratore* per una lista, però i due iteratori sono equivalenti per chi li utilizza. nel senso che entrambi permettono di esaminare la loro rispettiva collezione nello stesso modo.
Il pattern Factory Method suggerisce come mettere in relazione produttori e prodotti, in modo da mettere in evidenza le caratteristiche comuni dei produttori (tutti creano un prodotto) e quelle dei prodotti (tutti si usano nello stesso modo).
Il nome del pattern si riferisce al metodo con cui un produttore crea un prodotto.
Contesto:

1. Un tipo (*produttore*) crea oggetti di un altro tipo (prodotto)
2. Le sottoclassi del tipo produttore devono creare prodotti di tipi diversi
3. I clienti non hanno bisogno di sapere il tipo esatto dei prodotti

Soluzione:

1. Definire un tipo per un produttore generico
2. Definire un tipo per un prodotto generico
3. Nel tipo produttore generico, definire un metodo (detto metodo fabbrica), che restituisce un prodotto generico.
4. Ogni sottoclasse concreta del produttore generico realizza il metodo fabbrica in modo che restituisca un prodotto richiesto.

Tornando all'esempio degli iteratori, si fa nuovamente riferimento a come si esamina il contenuto di una lista:

```
113 LinkedList<Integer> list = new LinkedList<>();
114 list.add(5);
115 Iterable<Integer> iterableList = list;
116 Iterator<Integer> iterableList = iterator = iterableList.iterator();
117
118 while (iterator.hasNext())
119   System.out.println(iterator.next());
```

La LinkedList, come per tutte le collezioni, implementa Iterable, che nell'ambito del pattern rappresenta un produttore generico. L'interfaccia Iterable contiene il metodo "iterator", il cui tipo di ritorno è l'interfaccia "iterator", che nel pattern, rappresenta il prodotto generico. Le collezioni concrete, come LInkedList, producono iteratori concreti, ma gli utenti non hanno bisogno di conoscere il tipo esatto di questi iteratori; gli basta sapere che rispettano il contratto sabilito per gli iteratori dall'interfaccia Iterator.
Può essere utile considerare un caso che assomiglia al pattern Factory Method, ma che in realtà non gli corrisponde. Ad esempio, si ipotizzi che il metodo toString della classe Object. Non c'è un interfaccia che rappresenti il produttore generico, però la classe Object potrebbe farne le veci. Allo stesso modo, la classe String potrebbe fungere da prodotto generico, ma d'altra parte la classe String è final, cioè non estendibile. Quindi manca uno dei presupposti principali del pattern, ovvero "Le sottoclassi del tipo produttore devono creare prodotti di tipi diversi".

5.6 Singleton Pattern

Questo pattern serve ad assicurarsi che una classe abbia un'unica istanza. Si vuole cioè impedire agli utenti di una classe di creare più di una istanza di quella classe. In aggiunta si vuole affidare questo controllo al compilatore piuttosto che lanciare un'eccezione se vengono create più istanze.
Ci sono due soluzioni per affrontare questo problema: una basata su una classe regolare e una'altra basata su una classe **enumerata**.
Nel primo caso, la classe deve avere:

- Deve avere costruttore privato
- L'unica istanza che si desidera rendere disponibile può essere presentata tramite una *costante di classe*, cioè un attributo pubblico, final, statico.
- Deve avere un metodo statico per accedere all'unica istanza dell'oggetto

```
120 public class MySingleton  {
121   private static MySingleton  instance;
122
123   private MySingleton (){}
124
125   public static MySingleton  getInstance(){
126     if (instance==null){
127       instance = new MySingleton ();
128     }
129
130     return instance
131   }
132 }
```

Questa soluzione non è "thread-safe" in quanto se due thread diversi invocano getInstance contemporaneamente, potrebbe capitare che vengano istanziate due oggetti Singleton1, violando il principio fondamentale del pattern. Si tratta di una race condition.
Per risolvere questo problema basta dichiarare synchronized il metodo getInstance

```
134 public class MySingleton  {
135 private static MySingleton  instance;
136
137   private MySingleton (){}
138
139   public static synchronized  MySingleton  getInstance(){ // synchronized
140     if (instance==null){
141       instance = new MySingleton ();
142     }
143
144   return instance
145   }
146 }
```

Il secondo tipo di soluzione si basa su una **classe enumerata** che, per definizione non può essere instanziata dall'esterno.

```
147 enum MySingletonE {
148   INSTANCE;
149
150   private MySingletonE(){}
151
152 }
```

I client accedono all'istanza con la sintassi $MySingletonE.INSTANCE$, il che suggerisce che l'istanza viene creata al caricamento della classe.

5.7 Iterator Pattern

Descrizione
Iterator è un pattern comportamentale basato su oggetti e viene utilizzato quando, data una collezione di oggetti, si vuole accedere ai suoi elementi senza dover esporre la sua struttura. L'obiettivo di questo pattern è disaccoppiare l'utilizzatore e l'implementatore dell'aggregazione dei dati, tramite un oggetto intermedio che esponga sempre gli stesssi metodi indipendentemente dall'aggregato di dati. E' costituito di 3 soggetti:

1. Utilizzatore dei dati.
2. Iteratore che intermedia i dati.
3. Aggregatore che detiene i dati secondo una propria logica.

Partecipanti e Struttura
Questo pattern è composto dai seguenti partecipanti:

1. Iterator: colui che espone i metodi di accesso alla struttura dati.
2. ConcreteIterator: implementa l'Iteratore e tiene il puntatore alla struttura dati.
3. Aggregator: definisce l'interfaccia per creare un oggetto di tipo Iteratore.
4. ConcreteAggregator: implementa l'interfaccia di creazione di un oggetto Iteratore.

Conseguenze
Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Unica interfaccia di accesso: l'accesso ai dati avviene tramite l'Iterator che espone un'unica interfaccia e nasconde le diverse implementazioni degli Aggregator
2. Diversi iteratori di accesso: l'Aggregator può essere attraversato tramite diversi Iterator in cui ogni Iterator nasconde un algoritmo diverso

5.8 Composite Pattern

Descrizione

Si tratta di un pattern strutturale basato su oggetti che viene utilizzato quando si ha la necessità di realizzare una gerarchia di oggetti in cui l'oggetto contenitore può detenere oggetti elementari e/o oggetti contenitori. L'obiettivo è di permettere al Client che deve navigare la gerarchia, di comportarsi sempre nello stesso modo sia verso gli oggetti elementari e sia verso gli oggetti contenitori.

Partecipanti e Struttura

Questo pattern è composto dai seguenti partecipanti:

- Client: colui che effettua l'invocazione all'operazione di interesse
- Component: definisce l'interfaccia degli oggetti della composizione.
- Leaf: rappresenta l'oggetto foglia della composizione. Non ha figli. Definisce il comportamento "primitivo" dell'oggetto della composizione
- Composite: definisce il comportamento degli oggetti usati come contenitori ed detiene il riferimento ai componenti "figli".

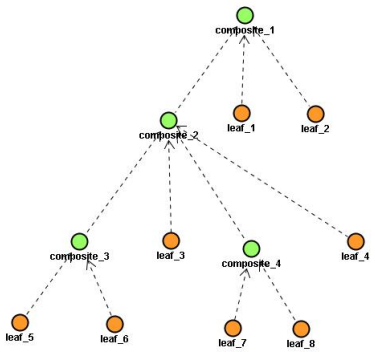

Figure 5.3: Nella programmazione java abbiamo esempi pratici di questa gerachia quando utilizziamo i packages che sono delle cartelle che contengono classi o packages annidati

Conseguenze

Tale pattern presenta i seguenti vantaggi/svantaggi:

1. Definisce la gerarchia: Gli oggetti della gerarchia possono essere composti da oggetti semplici e/o da oggetti contenitori che a loro volta sono composti ricorsivamente da altri oggetti semplici e/o da oggetti contenitori .
2. Semplifica il client: il Client tratta gli oggetti semplici e gli oggetti contenitori nello stesso modo. Questo semplifica il suo lavoro il quale astrae dalla specifica implementazione.
3. Semplifica la modifica dell'albero gerarchico: l'alberatura è facilmente modificabile aggiungendo/rimuovendo foglie e contenitori.

5.9 Question & Answers

1. Quali sono i vantaggi/svantaggi dell'applicazione del Design Pattern Factory rispetto alle caratteristiche di Coesione e Accoppiamento?

   L'utilizzo del design pattern Factory riduce l'accoppiamento, permettendo di demandare ad una classe specifica il ruolo di produzione di oggetti (prodotto) senza rendere note al client le rispettive implementazioni. Incentrando le responsabilità di produzione di un determinato tipo di prodotto ad una sola classe aumenta di conseguenza la coesione.

2. L'utilizzo del Design Pattern Factory ha impatti sulle attività di testing? Motivare la risposta.

   Assolutamente si, l'utilizzo del Design pattern factory agevola la creazione di unità di testing, permette di migliorare la leggibilità del testing stesso ed unificare il metodo di creazione degli oggetti da testare.

3. Descrivere le differenze e/o similitudini tra un'architettura a tre livelli e un'architettura MVC. Quando è preferibile una piuttosto che un'altra?

   Partendo dal presupposto che non ha senso implementare MVC se non si implementa anche Observable, spesso si finisce per confondere queste due strutture architetturali solo perchè dividono i ruoli in 3 elementi.
   MVC è un pattern ideato per facilitare la codifica della UI facilitare la manutenibilità ed il testing. Quando è implementato il pattern MVC una grande porzione dell'UI può essere testata. Il 3 Tier Architecture è un pattern usato per ragioni diverse. Serve giusto a separare l'intera applicazione in supergruppi (groups): UI, Busigness Logic, Data Storage. E' riferito quindi alla struttura di tutta una applicazione, mentre MVC è riferito principalmente alla parte grafica della architettura a 3 livelli.
   Ci sono molte intersezioni tra le due soluzioni (per questo spesso vengono confuse):
   I modelli "Business Model" e "Domain Models" sono comuni alle due soluzioni, ma il "View Models" è specifico di MVC.
   Views, Controllers sono specifici di MVC
   Data Access, Services sono specifici di 3-tier
   Per MVC, l'implementazione del DAO è quasi obbligatoria se non si vuole snaturare il pattern, visto che l'accesso ai dati non è riservato a nessuno dei 3 componenti di MVC.

4. Descrivere il design pattern DAO, fornendo anche il class Diagram e il Sequence Diagram di un'operazione di Retrieve.

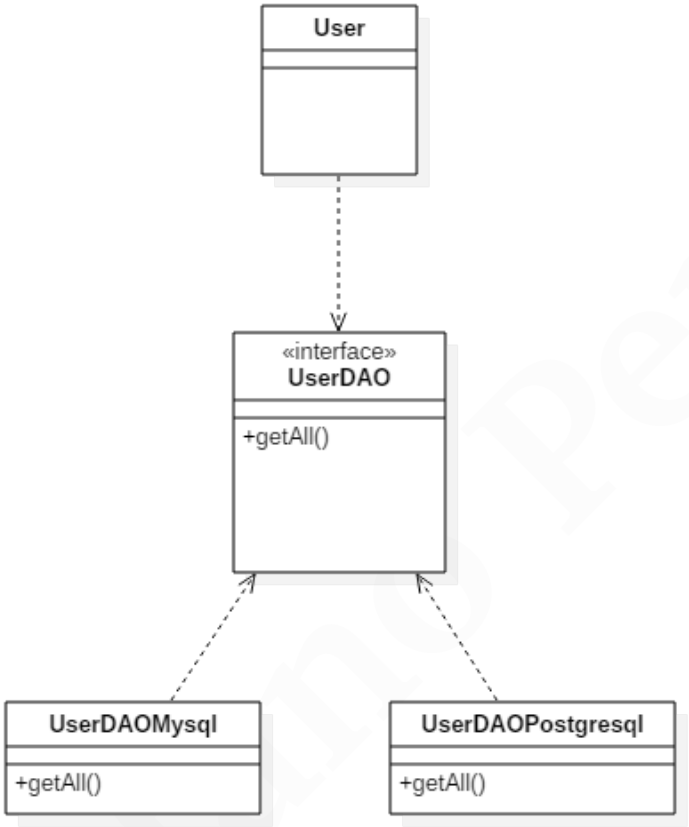

Figure 5.4: Esempio di Class Diagram di una classe Utente che implementa una interfaccia DAO

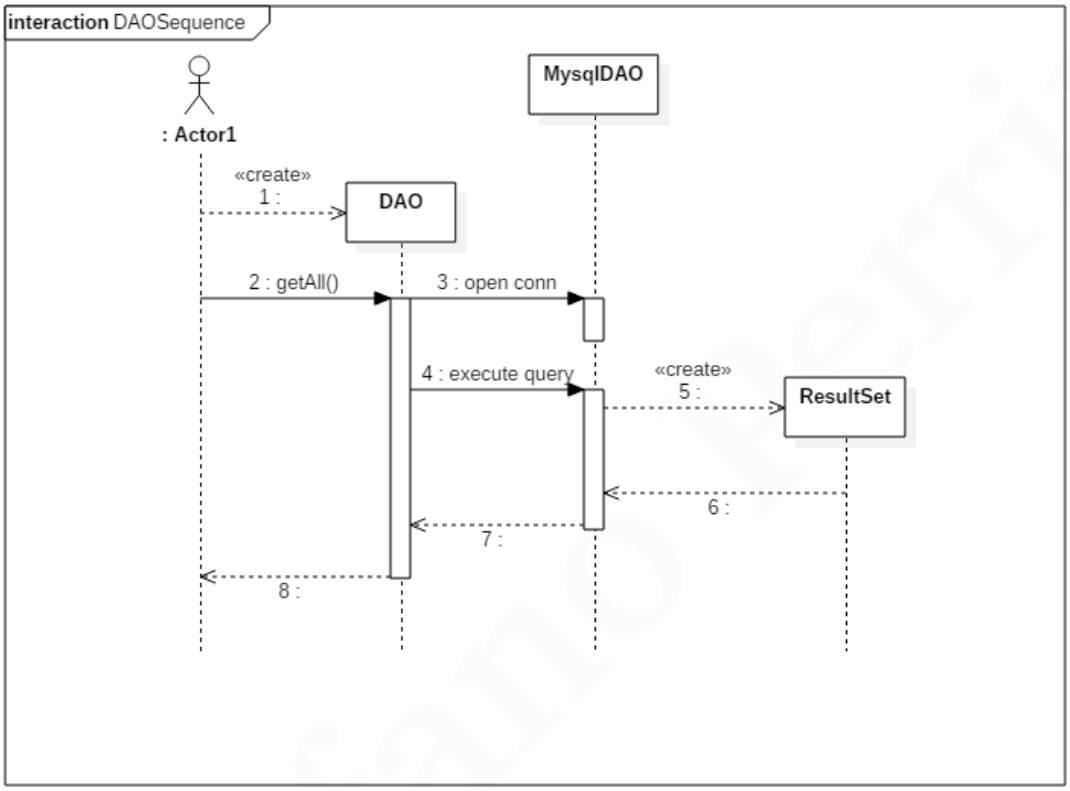

Figure 5.5: Esempio di Class Diagram di una classe Utente che implementa una interfaccia DAO

Chapter 6

Cloud Computing

6.1 IaaS, PaaS, Saas

Per Cloud Computing si intende la fornitura di servizi, applicazioni, attraverso la rete, sfruttando un "cloud" in qualità di server da un provider esterno.
Il "cloud" è implementato in in vari modi; attraverso l'uso di sistemi distribuiti o di virtualizzazione al fine di rendere accessibili le risorse da questi sistemi. In sintesi è una forma di **terziarizzazione tecnologica avanzata**. Grazie ai sistemi "cloud" l'onere di mantenere l'infrastruttura è demandato completamente ad un fornitore di servizi (provider).
Potenza di calcolo, spazio di archiviazione, banda disponibile, sono parametri che possono essere contrattualizzati con il provider al fine di personalizzare l'offerta in base alle esigenze.

6.1.1 SaaS: Software as a Service

Il **SaaS** è un modello di distribuzione del software (gestionali, crm etc) in cui un produtore sviluppa, opera e gestisce un'applicazione web, mettendola a disposizione dei propri clienti via internet. In questo modo un'azienda può accedere alle varie applicazioni che ha comprato tramite un'interfaccia web.
Con il **SaaS** l'azienda che fruisce del servizio non controlla l'infrasttuttura che supporta il software che è totalmente di competenza del provier.

6.1.2 PaaS: Platform as a Service

Attraverso un sistema PaaS, viene resa disponibile una piattaforma adatta allo sviluppo di applicazioni in cloud.
La piattaforma comprende linguaggi di programmazione, librerie, servizi e strumenti dedicati, interamente sviluppati dal provider.
Anche in questo caso tutte le responsabilità di infrasttuttura sono demandati al provider.

Evoluzioni della PaaS sono **iPaaS** (integration Platform as a Service) e il **dPaaS** (Data Platform as a Service).

- **iPaaS** consente alle aziende clienti di sviluppare, eseguire, gestire i processi di integrazione applicativa senza doversi occupare di installare o gestire alcun tipo di hardware o di middleware.
- con **dPaaS** sarà il provider a sviluppare direttamente le soluzioni per la gestione dei dati e la creazioni di applicazioni su misura per il cliente

6.1.3 DaaS: Desktop as a Service

Quando si parla di DaaS, si parla di servizi attraverso i quali un client potrà operare direttamente su un server sfruttando servizi di Desktop remoto. Questi possono essere gestiti in vari modi dal provider in base alla piattaforma a cui bisogna accedere e al tipo di licenza. Servizi di desktop remoto possono essere forniti come accesso ad una unico sistema operativo o a sistemi virtualizzati. Bisogna fare una distinzione più dettagliata quando si parla di desktop fornito come servizio esterno (in cloud).

- La **VDI** (Virtual Desktop Infrastructure) è un approccio in house, dove i sistemi di virtualizzazione sono resi disponibili all'interno dell'azienda da un server locale.
- Attraverso il **DaaS** (Desktop as a Service) invece è un provider esterno all'azienda a fornire accesso ai servizi virtualizzati.

6.1.4 IaaS: Infrastructure as a Service

Lo **IaaS** prevede un outsourcing evoluto di tutte le risorse ICT. Attraverso il "cloud" è possibile virtualizzare qualsiasi tipo di apparato, anche hardware, come switch, firewall, router, sistemi di storage etc. Con IaaS un'azienda decide di esternalizzare completamente le risorse, gestite a livello di infrasttruttura da un provider. Sicurezza, manutenzione, aggiornamenti: tutto viene affidato alla responsabilità del provider.

6.2 Question & Answers

1. Descrivere brevemente il concetto di architetture basate su Cloud Computing spiegando le differenze tra IaaS, PaaS e SaaS

   L'uso di tecnologie legate all'ambito cloud prevedono la possiblità di demandare all'esterno dell'azienda determinati servizi al fine di deresponsabilizzare l'azienda e affidarsi ad una infrastuttura dedicata. Dagli acronimi Infrastructure as Service, Platform as Service, Service as Service è facile intuire le differenze: la prima è una struttura di servizio in cloud che addirittura permette di esternalizzare apparati hardware (router, switch, storage etc) al fine di ottenere una più robusta struttura di supporto; la seconda, permette di ottenere servizi offerti in remoto e globalmente ad una azienda atti a produrre un ambiente di sviluppo comune; le risorse che possono essere messe a disposizione sono tools di sviluppo, ambienti grafici o di sviluppo etc. Infine il più comune Software as a Service che prevede la possibilità di sfruttare il web come strumento di interfaccia per la gestione della propria azienda. Il caso più comune sono i software di tipo gestionale, CRM etc che permettono di poter accedere a determinate informazioni aziendali ovunque nel mondo.

Chapter 7

Software di Versioning

7.1 SVN, GitHub, BitBucket, etc

Esistono vari strumenti **software** per il versionamento di progetti. **S**ub**v**ersio**n** è sviluppato da Apache ed è liberamente installabile. E' possibile scaricarlo da https://subversion.apache.org/ ed è facilmente gestibile attraverso il tool Tortoise SVN, scaricabile da https://tortoisesvn.net/, oppure è possibile interagire con un repository svn utilizzando gli strumenti integrati nei più moderni IDE, come Netbeans https://netbeans.org/. Ci sono molti altri sistemi di controllo versione alcuni dei quali mirano a soddisfare gli stessi obiettivi di Subversion, tra questi Git https://git-scm.com/, creato da Linus Torvalds per gestire il versionamento del kernel linux, e noto per anche grazie a GitHub, che è una piattaforma on line utilizzabile gratuitamente.

7.2 Question & Answers

Quali sono le principali operazioni offerte dai sistemi per il controllo delle versioni?

1. **CheckOut**: *Attraverso questo comando si stabilisce la relazione tra un progetto ed il sofware di versionamento; inizia il versionamento di una directory, contenente un progetto.*
2. **Commit**: *Permette di inviare al repository di versionamento i files recentemente modificati (commit parziale o totale)*
3. **Update**: *Permette di recuperare dal repository l'ultima release (oppure una specifica selezionata) del progetto; è importante lanciare questo comando appena si inizia a lavorare su un progetto, specialmente se condiviso con altri sviluppatori, per essere sicuri di essere allineati alla release corrente.*
4. **Diff**: *Permette di confrontare la versione di uno o più files tra quella locale e quella presente sul repository.*

5. **Ignore**: *Fondamentale se si lavora con software che generano cache o per evitare di gestire il versionamento di una direcotry specifica.*

Chapter 8

OCL

8.1 Espressioni e vincoli sui modelli

Object Constraint Language (OCL) è un linguaggio di specifica formale che permette di descrivere espressioni e vincoli su modelli object oriented.

- Una espressione è una specifica o riferimento ad un valore.
- Un vincolo è una restrizione su uno o più valori o parti di un modello orientatto agli oggetti.

OCL è un linguaggio standard ("non pittura" Cit.) che fa parte di Unified Modeling Language (UML). Le espressioni possono essere usate in diversi diagrammi UML:

- per specificare un valore iniziale di un attributo o associazione
- per specificare valori derivati di un attributo o associazione
- per specificare il corpo di una operazione
- per indicare una istanza in un diagramma dinamico
- per indicare una condizione in un diagramma dinamico
- per indicare il valore attuale di un parametro in un diagramma dinamico

Vi sono quattro tipi principali di vincoli:

1. una **invariante** è un vincolo che stabilisce una condizione che deve essere sempre valida per tutte le istanze di una classe o interfaccia. Una invariante è descritta usando una espressione logica che valutata è sempre vera.
2. una **precondizione** per una operazione è una restrizione che deve essere vera prima dell'esecuzione dell'operazione.
3. una **postcondizione** per una operazione è una restrizione che deve essere vera subito dopo l'esecuzione dell'operazione.
4. una **guardia** è un vincolo che deve essere vero in una fase della vita del sistema. Tipi di guardie sono init: e derive.

8.1.1 Il contesto di una espressione OCL

La definizione del contesto in una espressione OCL specifica la parte del modello per cui agisce l'espressione OCL. Generalmente è una classe, una interfaccia, un tipo di dato, un componente. Molte volte la parte interessata del modello è una operazione o un attributo, raramente una istanza. Il contesto indica sempre l'elemento sintattico del modello generalmente definito in UML. Questo elemento è chiamto contesto dell'espressione. Successivamente è importante il tipo di contesto dell'espressione.

L'espressione viene valutata sempre sugli oggetti (istanze) per cui spesso viene indicato esplicitamente l'istanze mediante la parola chiave self (l'equivalente di this in Java).
Esempio Se si vuol modellare gli studenti che frequentano corsi, un possibile modello è quello in figura.
***Invariante** 1* Per ogni istanza della classe Corso l'attributo capienza deve essere sempre minore o uguale a studenti.
***Invariante** 2* Per ogni istanza della classe Corso il numero delle istanze dell'associazione frequentatati deve essere uguale a studenti.
***Precondizione** 3* L'operazione di iscrizione(c:Corso) in Studente deve tenere conto della capienza del corso c ovvero deve essere inferiore di almeno una unità alla capienza.
***Postcondizione** 4* Dopo una operazione di iscrizione(c:Corso) l'attributo studenti è incrementato di uno e viene creata una nuova istanza dell'associazione frequenta.

```
154 -- Invariante 1
155 contex Corso inv: capienza<=frequentanti.size()
156 -- Invariante 2
157 contex Corso inv: studenti=frequentanti.size()
158 -- Pre e post condizione 3 e 4
159 context Studente::iscrizione(c:Corso)
160 pre: c.capienza > c.studenti
161 post: c.studenti=c.studenti@pre+1
162 and c.frequentanti=c.frequentanti@pre+1
```

8.2 Question & Answers

1. Specificare in al più mezza pagina cosa sono le invarianti, precondizioni e postcondizioni di OCL, fornendone un esempio. Leggere argomento in dettaglio

Chapter 9

Testing

9.1 Gerarchia nel software testing

1. Unit Testing
2. Integration Testing
3. System Testing
4. Acceptance Testing: beta testing, solitamente fatto da utenti esterni allo sviluppo.

9.2 Verification & Validation

Il processo di verifica e validazione di un software serve ad aumentarne l'affidabilità.
La verifica può essere **statica**, se effettuata su carta, o **dinamica**, se effettuata attraverso l'esecuzione del software.
La fase di verifica e validazione serve ad accertare che il software rispetti i requisiti e che li rispetti nella maniera dovuta.
Le operazioni di **verifica** servono ad assicurarsi:

- il software rispetti le specifiche
- che sia stato implementato tutto ciò che è stato descritto nel documento dei requisiti
- che tutte le funzionalità siano state implementate correttamente

Ma la sola verifica non basta!
Validazione (o convalida):

- il software rispetta ciò che voleva il cliente?
- i requisiti modellano ciò che il cliente realmente voleva?

Questa fase è molto delicata in quanto, dopo tutto il processo si può ottenere un software perfettamente funzionante, senza errori, ma del tutto inutile in quanto non rispecchia quanto era stato chiesto all'inizio.

La **verifica** può essere **statica**, se effettuata su carta, o **dinamica**, se effettuata attraverso l'esecuzione del software

9.3 Verifica Statica e Dinamica

Le tecniche di verifica possono essere divise in due macro categorie:

1. Verifica Statica: il software NON viene eseguito
2. Verifica Dinamica: il software va in esecuzione

9.3.1 Verifica Statica

La Verifica Statica è basata su tecniche di analisi statica del software senza ricorso alla esecuzione del codice.
Analisi statica: è un processo di valutazione di un sistema o di un suo componente basato sulla sua forma, struttura, contenuto, documentazione.
Tecniche: review, ispezione, verifica formale, esecuzione simbolica, etc...

La Review

Esisono due tipi di review:

1. WalkThrough: Lo sviluppatore presenta informalmente le API, il codice, la documentazione associata delle componenti al team di review.
2. Inspection. Simile al walkthrough, ma la presentazione delle unità è formale.

La **ispection** prevede che lo sviluppatore non possa presentare gli artefatti, demandando il compito al team di revisione, responsabile del controllo delle interfacce e del codice dei requisiti. In questo tipo revisione viene controllata l'efficienza degli algoritmi con le richieste non funzionali. Lo sviluppatore interviene solo se si richiedono chiarimenti.
Durante la fase di revizione l'obiettivo è trovare il maggior numero possibile di difetti.
I metodi di ispezione sono molto efficaci: la maggior parte dei "fault" possono essere individuati in questa fase.

9.3.2 Verifica Dinamica

E' fondata sull'esecuzione del codice.

È convinzione diffusa che la sola applicazione dell'analisi statica basti ad evidenziare e debellare i problemi di sicurezza più comuni.
Tuttavia, dato l'elevato numero di falsi positivi segnalati quando si utilizza l'analisi statica, capire se si è rilevato un errore reale è un processo che richiede tempo, e le modifiche messe in atto potrebbero proprio aver nascosto l'errore sotto un falso-negativo (un codice che compila, funziona per un indeterminato numero di dati in input, ma in realtà presenta dei *fault*).

- Analisi Dinamica: il processo di valutaizone di un sistema software o di un suo componente basato sulla osservazione del suo comportamento in esecuzione.
- Testing: Approccio strutturato alla selezione di casi di test e dati associati.
- Debugging, per il quale non c'è uniformità in letteratura, ma in generale riguarda l'individuazione e l'eliminazione di difetti.

Le differenze tra **Debugging** e **Testing** sono che, mentre per il **debugging** si formulano ipotesi osservando il comportamento del programma per poi verificarle per localizzare gli errori. Il **Testing** invece consiste nel trovare le differenze tra il comportamento atteso, specificato attraverso il modello del sistema e il comportamento osservato dal sistema implementato.
Il debugging è una attività di ricerca che viene effettuata di solito in conseguenza di un test che ha avuto successo.
Anche in questo caso l'**obiettivo** è trovare il maggior numero di difetti possibile. Il testing dovrebbe essere realizzato da persone che non sono state coinvolte nell'attività di sviluppo del sistema.

9.4 Equivalence Class Testing

L'equivalence class testing nasce dalla necessità di dover testare un largo numero di elementi (valori), ma l'impossibilità di testartli tutti per motivi di tempo.
Raggruppare gli elementi di test, suddividendoli per classi di equivalenza, dove tutti gli elementi in ogni classe si suppone abbiano lo stesso comportamento.
Teoricamente, è necesario che corrisponda un caso di test per ogni classe di equivalenza.
Esempio 1:
E' necessario gestire un valore all'interno di un range specifico, come la valutazione di un esame universitario, oppure la gestione dei mesi dell'anno. I valori da 1 a 12 sono accettati, quelli inferiori o superiori, no. Come classe di equivalenza, quindi, tutto l'intervallo $[1-12]$ sarà un'unica classe di equivalenza, mentre i valori nell'isieme: $]-\infty,0] \cup [13,+\infty[$ possono essere considerati una unica classe di equivalenza, dei valori "non validi" o due classi di equivalenza distinte, per maggiore precisione.
Pericoli dell'equivalence class testing: Non è detto che se due valori si suppone possano rientrare nella stessa classe di equivalenza, allora i test generati dall'utilizzo di tutti i valori in quella classe genererà lo stesso risultato.
EC testing è un ottimo strumento di test, ma non è "fool proof" e bisogna usarlo con attenzione.
In molti casi è necessario testare ogni singolo valore e anche di più per ridurre al minimo le possibilità che il software, e un metodo in particolare, abbia difetti.

9.4.1 Boundary Value Testing

Il "BV Testing" si basa sulla scelta di testare i valori ai margini di ogni classe di equivalenza definita nell'EC Testing. La maggior parte dei problemi nasce proprio nell'utilizzo di questi valori "limite", all'interno delle EC prese in considerazione. Esempio:

EC	Values	BV
EC1	$[1-12]$	$1, 12$
EC2	$]-\infty, 0]$	$intmin, 0$
EC3	$]13, +\infty]$	$13, intmax$

Questo genere di test è facilmente applicabile qualora si debbano verificare interi, e costringe ad effettuare almeno 2 valori per ogni classe di equivalenza, ma è molto più complesso se non si lavora con valori "integer" (immagina il caso in cui si debbano testare stringhe contenenti numeri di serie di un device).

9.4.2 Hidden Boundary Value Testing

BV of a class son spesso basati sulle specifiche di come un sistema dovrebbe funzionare. Questo vale per molti sistemi, ma per alcuni sistemi ci sono valori limite non esplicitati nella documentazione.
Per esempio, quanti caratteri possono essere scritti in una text area di un browser prima che questo si blocchi su un dato sistema (non si parla del limite dei dati che possono essere contenuti sulla struttura dati che dovrà, eventualmente, salvare quel dato).

9.4.3 SECT

Criterio forte (Strong Equivalence Class Testing – SECT). Quando si parla di Unit testing, una volta analizzate le classi di equivalenza per la copertura di test di un metodo, se si vuole garantire una SECT completa è necessario eseguire 1 test per ogni classe di equivalenza per ogni parametro formale del metodo da testare.

Esempio:

```
163 /**
164 * Find user, by Name or Lastname from persistence
165 * @param Name: String
166 * @param Lastname: String
167 * @return integer with id of the user
168 */
169 public int findUser(String Name, String Lastname){...}
```

Trovandoci in un caso di Black Box Testing, non abbiamo modo di sapere il codice del metodo, ma attraverso la firma (e magari anche quel pizzico di javadoc che ci permette di capire cosa dovrà fare quel metodo). Il metodo prende in input 2 Stringhe e possiamo quindi immaginare le seguenti classi di equivalenza

Equivalence Class	Description
EC1	name is null
EC2	name is an empty String
EC3	name is a String not matching any users's name
EC4	name is a String matching at least one users's name

Equivalence Class	Description
EC5	lastname is null
EC6	lastname is an empty String
EC7	lastname is a String not matching any users's lastname
EC8	lastname is a String matching at least one users's lastname

Un totale di 4 classi di equivalenza per ogni parametro. Per ottenere una SECT è necessario testare le combinazioni ottenute dal prodotto cartesiano tra l'ordine delle classi di equivalenza evidenziate $4x4$.

Otterremo quindi 16 TestCases derivanti dal prodotto cartesiano.

TestCase	Equivalence Class	Equivalence Class
TC1	EC1	EC5
TC2	EC1	EC6
TC3	EC1	EC7
TC4	EC1	EC8
TC5	EC2	EC5
TC6	EC2	EC6
TC7	EC2	EC7
TC8	EC2	EC8
TC9	EC3	EC5
TC10	EC3	EC6
TC11	EC3	EC7
TC12	EC3	EC8
TC13	EC4	EC5
TC14	EC4	EC6
TC15	EC4	EC7
TC16	EC4	EC8

Completando questi 16 test si otterrebbe un **S**trong **E**quivalence **C**lass **T**est (che ad ogni modo, in alcuni casi, non basta ad essere considerata completa, ma lo vedremo in seguito).

9.4.4 WECT

E' facile immaginare che testare un metodo con tanti parametri formali in input ci costringerebbe a scrivere un innumerevole quantitativo di casi di test. Prese in esame le stesse classi di equivalenza scritte per la SECT, nel caso in cui quel metodo avesse preso in input anche "indirizzo, città, codice fiscale", ci saremmo trovati con $4x4x4x4x4 = 1024$ TestCases... Ci sono poi alcuni casi in cui un singolo test può richiedere parecchie ore per essere portato a termine e in alcune situazioni, eseguire una SECT richiede troppo tempo.
La Weak Equivaalence Class Testing (WECT) richiede che per ogni classe di equivalenza ci debba essere un Test Case che usa un valore nominale da quella classe di equivalenza. Se possibile il valore nominale o il valore medio della classe (possibile per le classi superiormente o inferiormente limitate e ordinate)... (mo arriva Cutolo, ndr.).
E' necessario, per una WECT, che tutte le classi di equivalenza siano eseguite almeno una volta.
Abbiamo detto che per $Stringname$ avremmo avuto 4 classi di equivalenza e altrettante per $Stringlastname$
Sia A il dominio delle classi di equivalenza di $Stringname$ e B l'insieme delle classi di equivalenza di $Stringlastname$ una WECT richiede un massimo numero di: $max(|A|,|B|)$ Mentre per la SECT avevamo visto che il numero dei test cases da analizzare sarebbe stato $|A|x|B|$

Volendo fare una WECT del metodo su descritto saranno sufficienti appena 4 casi di test.

TestCase	Equivalence Class	Equivalence Class
TC1	EC1	EC5
TC2	EC2	EC6
TC3	EC3	EC7
TC4	EC4	EC8

9.4.5 Considerazioni sull'equivalence class testing

- In ambienti safety critical bisogna estendere la SECT e includere sia tutti i valori validi che quelli non validi.
- ECT è appropiato quando i dati in input sono definiti su un range e in un insieme di valori discreti.
- SECT Assume che le variabili siano indipendenti e che le dipendenze possano generare errori nei test cases.

9.5 WhiteBox & BlackBox

Il blackBox testing è basato sulla funzionalità del sistema e la struttura del programma. Le tecniche usate per il BlackBox testing sono

- Equivalence partitioning
- Analisi dei valori limite (boundary value analysis)
- Error guessing (tecniche di testing basate sull'esperienza dei testers che permettono di analizzare situazioni in cui il software potrebbe avere problemi.)

Il tester può non essere un tecnico, serve ad identificare problemi funzionali e le specifiche.
Il WhiteBox testing utilizza invece

- Basis path testing
- Flow Graph Notation:
- Control Structure Testing
- Loop Testing

Lo scopo è individuazione dei percorsi di base al fine di individuare il numero massimo di casi di test per una totale copertura del codice.
Il tester deve essere uno sviluppatore e deve essere in grado di identificare problemi logici legati alla codifica.

Prendiamo un metodo tipo:

```
170 public int getIdByUsername(String username){...}
```

In Black Box, osservando la signature del metodo è possibile individuare le classi di equivalenza affinchè il metodo sia testato con una coperatura SECT, ma non ci è possibile sapere se il metodo sarà testato completamente (tutti i branches). Solo osservando il codice sorgente del metodo, ed eventualmente implementando un diagramma di flusso, sarà possibile accertarsi che tutto il metodo sarà testato completamente e stabilire la corretta strategia per lo UNIT Testing.

```
171   public int getIdByUsername(String username){
172     int id = 0;
173     ResultSet rs = null;
174     PreparedStatement ps = null;
175     String sql = "SELECT id FROM users WHERE username = ?";
176
177     if (username==null) // Durante il jUnit testing bisogna testare anche le
            eccezioni
178       throw new IllegalArgumentException("Stringa nulla");
179
180     if (username.equals("Shady"){
181       return 1    // Per assurdo, sappiamo che l'utente shady, essendo
              amministratore, ha sicuramente id =1
182                 // e vogliamo risparmiarci la connessione al db ottenendo
                      quindi il risultato
183                 // a tempo costante...
184                 // Questa cosa in black box non avremmo mai potuto intuirla e
                      non avremmo mai potuto
185                 // eseguire un test per verificare se questa parte del metodo
                      presenta bugs
186     } else {
187       ps=DbConnection.getConnection().prepareStatement(sql);
188       rs=ps.executeQuery();
189       rs.next();
190       id=rs.getInt("id");
191       return id;
192     }
193   }
```

9.6 STUB

Eseguire test case su un componente (o una compbinazione di componenti).
Test driver e test stub sono usati per sostituire le parti mancanti del sistema.

- Un test **stub** è una implementazione parziale di componenti da cui la componente testata dipende (componenti che sono chiamate dalla componente da testare).
- Un test driver è un blocco di codice che inizia e chiama la componente da testare.

$TestDriver + TestStub = Scaffolding$

9.6.1 Driver e Stub

Driver:

- Deve simulare l'ambiente chiamante
- Deve occuparsi dell'inizializzazione dell'ambiente non locale del modulo in esame

Stub(Modulo Fittizio):

- Ha la stessa interfaccia del modulo simulato (tipicamente ne è una sottoclasse), ma è privo di implementazioni significative.
- Detti anche Mock Objects

Un test stub deve fornire la stessa API del metodo della componente simulata e ritornare un valore il cui tipo è conforme con il tipo del valore di ritorno specifico nella signature. Se l'interfaccia di una componente cambia, anche il corrispondente test driver e test stub devono cambiare. L'implementazione di un test stub non è una cosa semplice. Non è sufficiente scrivere un test stub che semplicemente stampa un messaggio attestante che il test stub è in esecuzione.
La componente chiamata deve fare un qualche lavoro. Il tests stub non può restituire sempre lo stesso valore.

9.7 Scaffolding

La necessità di moduli di stub e moduli driver dipende dalla posizione del modulo nell'architettura del sistema. Creare l'ambiente per l'esecuzione dei test richiede

- Lo scaffolding è estremamente importante per il test di unità di integrazione
- Può richiedere un grosso sforzo programmativo

- Uno scaffolding buono è un passo importante per test di regressione efficiente
- La generazione di scaffolding può essere parzialmente automatizzato a partire dalle specifiche.
- Esistono pacchetti software per supportare la generazione di scaffolding (jUnit, phpUnit, cUnit)

9.8 Unit testing

Un unit test è una parte di codice, scritta da uno sviluppatore per eseguire una specifica funzionalità e testarla "asserendo" di ottenere un certo comportamento.
La percentuale di codice testato da un "unit test" è chiamato "test coverage".
L'obiettivo di un unit test è testare una piccola porzione di codice come un metodo o una classe. Le dipendenze esterne dovrebbero essere rimosse dall'unit test e sostituite/implementate nella realizzazione del codice di test, utilizzando un test framework.
L'unit testing non è adatto a testare complesse interfacce grafiche o interazione tra componenti... per questo esiste l'integration testing.

9.9 JUNIT

JUnit è un framework orientato alla creazione di unità di test. Nella fattispecie, jUnit è un framework completo di test per Java.

Di seguito una struttura di esempio per il test di un metodo

```
194 import org.junit.*;
195
196 public class HelloClass {
197
198   @Rule
199   public ExpectedException thrown = ExpectedException.none(); // Necessario
          se un metodo da testare lancia eccezioni
200
201   @BeforeClass
202   public static void setUpClass() {
203   }
204
205   @AfterClass
206   public static void tearDownClass() {
207   }
208
209   @Before
210   public void setUp() {
211     // Questo metodo viene eseguito prima dei test.
212     // Serve a preparare l'ambiente di test
213   }
214
215   @After
216   public void tearDown() {
217     // Questo metodo viene eseguito alla fine dei test
218     // Serve a riportare in dietro lo stato degli eventi
219     // al fine di eliminare i side-effects derivati dall'esecuzione del
          metodo di test
220   }
221
222   @Test
223   public void testReturnHelloWorld(){
224     String expResult="Hello Word";    // L'oracolo: il risultato che ci
          aspettiamo di ottenere
225     String result = null;
226     result=returnHelloWorld();      // Si invoca il metodo
227     assertEqual(result, expResult);     // Verifichiamo che il risultato sia
           uguale a quanto ci aspettavamo
228   }
229
```

```
230   @Test
231   public void testReturnHelloWord(String s){
232     thrown.expect(IllegalArgumentException.class);            //Tipo di
          eccezione attesa, lanciata dal metodo
233     thrown.expectMessage("la stringa in input non deve essere null");   //
          messaggio di errore atteso dall'eccezione lanciata
234   }
235 }
```

9.10 Integration Testing

Un integration test mira a testare il comportamento di un componente o l'integrazione tra un set di componenti.
Attraverso l'integration test si controlla che l'intero sistema funzioni come previsto, riducendo quindi la necessità di test manuali.
Questo tipo di test permette di tradurre situazioni in casi di test. Il test dovrebbe simulare un comportamento aspettato dall'interazione tra metodi nell'applicazione.

9.11 System Testing

Il system test rientra nella categoria del **blackbox testing** in quanto è coinvolto nel modo in cui un software è usato dall'esterno dalla prospettiva di un utente.

Il system testing si basa su una serie di test differenti con lo scopo di testare un intero sistema. Il software testing si basa su due categorie

- Black Box Testing
- White Box Testing

9.12 Question & Answers

1. Descrivere la differenza tra verifica statica e dinamica. Porre particolare enfasi sul corretto inquadramento di entrambe all'interno del ciclo di vita del software.

   La verifica statica prevede l'analisi del software senza far ricorso all'esecuzione. E' un processo di valutazione del sistema solitamente affidabile ad un team esterno che si occuperà di rivedere e quindi validare documentazione, requisiti e analizzare il codice. Questo tipo di analisi va fatto a seguito della realizzazione di una componente del software.
   La verifica dinamica prevede invece l'analisi del codice attraverso la sua esecuzione e la realizzazione di sistemi di testing per verficare robustezza e funzionalità di classi e metodi.

2. Commentare la seguente affermazione:"Se si ha accesso al codice sorgente, è meglio effettuare Whitebox. Il blackBox è usato principalmente se non si ha accesso al sorgente". E' condivisibile? Giustificare la risposta.

   Sicuramente il blackBox è l'unico metodo per testare una unità se impossibilitati ad accedere al codice, ma questo è altresì importante per analizzare i casi di test attraverso i metodi di Equivalence Testing. In definitiva, se si ha accesso al codice è sempre buona norma eseguire prima un blackbox testing e successivamente il whitebox, specialmente in ambienti safety critical.

3. Descrivere differenze e similitudini (se ve ne sono) tra WhiteBox e Ispection.

   La verifica in WhiteBox fa parte della verifica dinamica se completata attraverso unit testing; questa fase è preposta alla ricerca di tutti i branch possibili atti ad ottenere una full coverage nel test di un metodo, ma non cerca "fault", permette di trovarli attraverso il testing.
   L'ispection, facendo parte della verifica statica, prevede la rircerca dei "fault" in fase di analisi. Questi possono essere successivamente "fixati" attraverso la modifica del codice da parte del team di sviluppo.

4. Descrivere la differenza tra SECT e testing dei valore limite, mostrando un esempio.

   SECT sta per "Strong equivalence class testing" e prevede la copertura di tutti i casi analizzati in blackbox testing una volta evidenziate la classi di equivalenza. Il testing dei valore limite è sicuramente previsto all'interno di una SECT, ma non garantisce una verifica di tutti i casi che possono essere evidenziati dalle EC.

5. Descrivere come si può testare con jUnit il corretto lancio di eccezioni in un programma Java.

   Per testare una eccezione con jUnit è necessario definire nelle @Rule la regola ExpectedException e all'interno del metodo, per lanciarla, usare il metodo thrown.expect(nomeEccezione.class) e quindi thrown.expectMessage("messaggio atteso, inviato dall'eccezione"); Vededere codice esempio

6. Descrivere la differnza tra WECT e SECT fornendo un esempio

WECT è un tipo di Equivalence Class Testing che prevede un limitato numero di casi di test rispetto ad una SECT. Un metodo che ha troppi parametri formali da testare potrebbe richiedere tempi biblici per l'esecuzione di una SECT, per questo può essere analizzata una WECT riducendo i casi di test in modo da testare tutti i branches possibili, senza dover coprire tutte le classi analizzate nella SECT.

7. Come si colloca JUnit rispetto a BlackBox Testing, WhiteBox Testing, Unit Testing, Integration Testing, System Testing, motivando la risposta.

   E' uno strumento indispensabile! Si potrebbero scrivere metodi di testa anche senza l'uso del framework jUnit, ma non avremmo nessuna garanzia di robustezza che ci garantisce il framework e avremmo bisogno di scrivere altri metodi per testare i metodi scritti per il testing.. una spirale infernale, direi!

Chapter 10

System Design

10.1 Legge di demetra

Nella pura OOP, la Legge di Demetra non è un suggerimento, è la legge della terra. Questo tuffo nella legge copre usi comuni e difetti nell'interpretazione.

Per tutte le classi C, e tutti i metodi M appartenenti a C, tutti gli oggetti a cui i metodi M mandano messaggi devono essere

- Gli oggetti argomento di M, incluso l'oggetto stesso, oppure
- Le variabili di istanza della classe C

Gli oggetti creati da M, o da funzioni o metodi chiamati da M, e gli oggetti nelle variabili globali sono considerati argomenti di M.

Tradotto in java:

- Quando si parla dell'oggetto stesso, in java si usa "this".
- Gli oggetti argomento di M
- Instance of C
- Oggetto creato da M o funzioni o metodi chiamati da M
- Gli attributi statici della classe

Questo sta ad intendere che la Legge proibisce di "inviare un messaggio" ad un qualsiasi altro oggetto esistente che è mantenuto in una variabile di istanza di altre classi, a meno che esso non sia mantenuto dalla nostra classe o passata a noi come parametro.

In conclusione, la legge di demtra è un ben definito insieme di regole per lo sviluppo OOP. Seguire queste regole alla lettera è una buona pratica per procedere ad un corretto sviluppo di un software orientato ad oggetti.

Inoltre, produce segnali chiari se il codice si discosta dal percorso orientato agli oggetti, quindi è uno strumento prezioso per mantenere i nostri progetti e lo stile del codice sulla giusta direzione.

Listing 10.1: Esempio1 dalla traccia 29-01-2016

```
238
239 public class Foo {
240
241   class D {
242     public void doSomethingElse(){};
243   }
244   class Bar {
245     public C getC(){ return null; };
246   }
247
248   public class C{
249     public void doIt(){};
250   }
251
252   public void example(Bar b) {
253     C c = b.getC();  // Prima violazione
254     c.doIt();
255     b.getC().doIt(); // Seconda violazione
256     D d = new D();
257     d.doSomethingElse();
258   }
259
260 }
```