

Ingegneria del software

Appunti di Luciano Imbimbo

September 2022

Contents

1 Introduzione al corso	4
1.1 Writing a program	5
1.2 Building a System	9
1.2.1 Size and Complexity	9
1.2.2 Problemi decisionali	10
1.2.3 Technical issues and Methodology	10
1.2.4 Nontechnical considerations	12
1.2.5 Other considerations	12
1.3 Software Engineering	13
1.3.1 Software Failures	13
1.3.2 Definition of Software Engineering	15
1.3.3 Software Process	15
2 Specifica dei Requisiti	18
2.1 Requirements Engineering	18
2.1.1 Elicitation	20
2.1.2 Analysis	21
2.1.3 Definition,Prototyping, and Reviews	27
2.1.4 Specification and Agreement	28
2.2 UML - Unified Modeling Language	29
2.3 UML - Use Case Diagram	30
3 Progettazione	40
3.1 Design: Architecture and Methodology	40
3.1.1 SOLID principles	41
3.1.2 Design Components	49
3.1.3 Software Architecture	57
3.1.4 Meta-Architectural Knowledge	68
3.2 Design Characteristics and Metrics	79
3.3 Design Pattern	90
3.3.1 Strategy Pattern	92
3.3.2 Observer Pattern	97
3.3.3 Decorator Pattern	102
3.3.4 Factory Pattern	107
3.3.5 Singleton Pattern	117
3.3.6 Command Pattern	121
3.3.7 Adapter Pattern	125
3.3.8 Facade Pattern	128
3.3.9 Template Pattern	131
3.3.10 Iterator Pattern	135
3.3.11 Composite Pattern	144
3.3.12 CompositIterator Pattern	148
3.3.13 State Pattern	151
3.3.14 Proxy Pattern	156

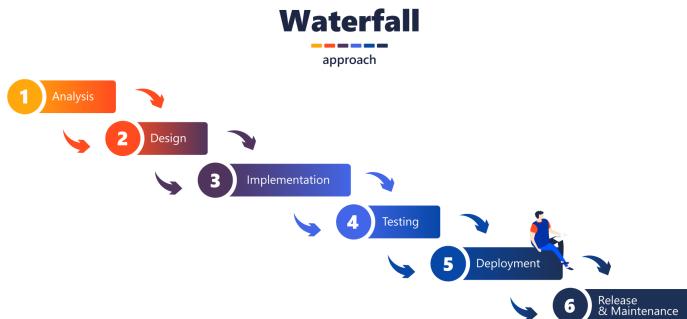
3.3.15	Model-view-controller(MVC)	163
3.3.16	Bridge Pattern	164
3.3.17	Builder Pattern	166
3.3.18	Chain of Responsibility	168
3.3.19	Visitors Pattern	172
3.3.20	Pattern Together	176
3.4	UML - Class Diagram	177
3.4.1	Class Diagram Advanced	185
3.5	UML - Activity Diagram	191
3.5.1	Nodi speciali	199
3.6	UML - Sequence Diagrams	203
3.7	UML - State Machine Diagram	215
4	Implementazione	221
5	Testing	230
5.1	Black-box Testing	235
5.2	White-Box Testing	243
5.3	Automated Unit Testing	250
5.4	Inspection	250
5.5	JUnit	252
6	Distribuzione e Manutenzione	255
6.1	Configuration Management	255
6.2	Integration and Builds	256
6.3	Software Support	256
6.4	Maintenance	259
6.5	VCS - Git	259
6.6	Docker	263

1 Introduzione al corso

Perché studiare Ingegneria del software? Questa potrebbe essere una prima spontanea e lecita domanda da porsi; la risposta è che il corso di ingegneria del corso serve per imparare metodologie e tecniche per costruire in modo struttura del software. Infatti il processo di realizzazione del software può essere scomposto in 4/5 fasi:

1. **Analisi dei Requisiti, Analisi tempi di scadenza** e obiettivi prefissati dal cliente. Gli errori più impattanti provengono da una scorsa analisi dei requisiti. Per progettare e studiare questa fase utilizzeremo i diagrammi UML.
2. **Progettazione e Design.** In questa fase definiremo classi ed architettura in funzione della leggibilità del codice. A supporto di questa fase utilizzeremo UML, JPA, Hibernate.
3. **Implementazione e testing.** Passiamo dal design all'implementazione del codice
4. **Distribuzione e manutenzione**

Questa sequenza di fasi veniva seguita rigorosamente fino agli anni '70 nel cosiddetto modello Waterfall, caduto poi in disuso post anni '70 per via della sua inefficienza(gli errori in fase di analisi ricadono a cascata sulle fasi successive) e degli elevati costi di mantenimento.



Oggi queste fasi non seguono ordine rigoroso ma vengono utilizzati modelli più flessibili ed efficienti.

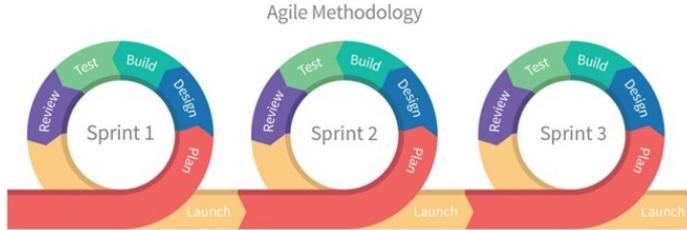


Figure 1: Modello Agile

1.1 Writing a program

Vediamo come queste fasi si articolano con un piccolo e semplicissimo problema.

Data una collezioni di stringhe, salvate su un file, dobbiamo ordinarle in ordine alfabetico e scriverle su un altro file.

Anche un piccolo problema come questo, a primo impatto, può destare dei dubbi su come implementare in codice questo problema. Per procedere dobbiamo analizzare il problema definendo requisiti e vincoli:

- **Program Requirements** che definiscono cosa il programma deve fare. Ogni requisito viene definito e accettato dal cliente che ne valuta il costo. I requisiti si dividono in 2 macrocategorie:
 - **Functional Requirements**, cioè quello che il programma fa che nel nostro esempio possono essere il formato dell'input e dell'output, il tipo di ordinamento, la gestione dei caratteri speciali nell'ordinamento e gestione di casi speciali ed errori(file vuoto).
 - **Nonfunctional Requirements** che definiscono il modo in cui i requisiti funzionali devono essere raggiunti; nel nostro esempio ma anche in generale sono requisiti non funzionali particolari richieste sulle performance, vincoli di efficienza a scapito delle leggibilità per applicativi real-time e future estensioni del software (per nuove funzioni e bug fixes durante il ciclo di durata del software).
- **Design Constraints**, o vincoli progettuali, che definiscono il modo in cui il software deve progettato ed implementato (molte volte considerati requisiti non funzionali). Sono vincoli progettuali il tipo di interfaccia grafica da usare(CLI or GUI), la piattaforma su cui sviluppare (OS, IDE, Architettura), il linguaggio di programmazione che impatta notevolmente sulle performance e sui tipi di algoritmo da usare, scadenze e sicurezza.

Una volta analizzato e capito il problema dobbiamo, basandoci sui requisiti, organizzare le funzionalità principali anche ricorrendo all'uso di diagrammi (UML),

decidere gli algoritmi e potenziali miglioramenti per realizzare le funzionalità principali e ragionare sui vincoli di progetto e requisiti non funzionali.

Un ruolo importante nella pianificazione del progetto lo ricopre la stima dei tempi(tempo ideale e tempo di calendario) e delle risorse(effort). Per stimare i tempi e le risorse da allocare dobbiamo usare il criterio di *dividi et impera*, letteralmente divide e conquista, che si basa sul dividere il problema generale in sottoproblemi più piccoli e meno complessi. Vediamo un esempio con il problema dell'ordinamento:

- Dobbiamo leggere le stringhe dal file (Read)
- Dobbiamo scrivere le stringhe su file (Write)
- Ordinare la collection in ordine alfabetico(Sort)
 1. Trova l'elemento più grande (findIdxBiggest)
 2. Mettilo alla fine dell'array (swap)
 3. Ripeti lo stesso procedimento per gli altri elementi
- Sviluppo interfaccia grafica (GUI)
- Test programma

Una volta diviso il problema in sottoperazioni dobbiamo per ognuna di esse stimare il tempo ideale di sviluppo e il tempo effettivo di realizzazione(tempo di calendario che considera anche i fattori esterni come impegni, tempi lavorativi, imprevisti).

	Ideal time	Calendar time
findIdxBiggest	5Min	22/03
Swap	3Min	22/03
Sort	15Min	22/03
Read	5Min	22/03
Write	5Min	22/03
GUI	1h	23/03
Testing	3h	24/03
Total	4h 33Min	25/03

Si procede traformando il design realizzato in codice. Per un'implementazione corretta dobbiamo implementare gli algoritmi generali nello specifico linguaggio di programmazione, aver definito convenzioni (camelCase, kebabCase) dei nomi e la lingua dell'applicativo, avere una ampia conoscenza delle librerie standard (non contengono errori) del linguaggio per risparmiare tempo e risorse ed eseguire una Review del codice. Vediamo un'implementazione del nostro problema.

```

import java.io.*;
import java.util.*;

public class StringSorter{
    ArrayList<String> lines;

    public void readFromStream(Reader r) throws
        IOException{
        BufferedReader br = new BufferedReader(r);
        lines = new ArrayList<String>();

        while(true){
            String input = br.readLine();
            if(input == null){
                break;
            }
            lines.add(input)
        }
    }

    static void swap (List<String> l, int i1, int i2){
        String tmp = l.get(i1);
        l.set(i1,l.get(i2));
        l.set(i2,tmp);
    }

    static int findIdxBiggest(List<String> l, int from,
        int to){
        String biggest = l.get(0);
        int idxBiggest = from;

        for(int i = from + 1 ; i <= to; i++){
            if(biggest.compareTo (l.get(i)) < 0){
                biggest = l.get(i);
                idxBiggest = i;
            }
        }

        return idxBiggest;
    }

    public void sort(){
        for(int i = lines.size() - 1 ; i > 0; i--){
            int big = findIdxBiggest(lines, 0, i)
            swap(lines, i, big);
        }
    }
}

```

```

        }
    }

    // il tutto poteva essere sostituito con una
    // funzione della libreria standard
    void sort(){
        java.util.Collections.sort(lines);
    }
}

public static void main(String[] args){
    Reader in = new FileReader("filename");
    wWriter out = new FileWriter("filename");

    StringSorter ss = new StringSorter();
    ss.readFromStream(in);
    ss.sort();
    ss.writeToStream(out);

    in.close();
    out.close();
}

```

Durante lo sviluppo del programma e subito dopo il completamento bisogna effettuare dei test per verificare il funzionamento del programma. Per effettuare un buon test bisogna ragionare su tutti i possibili input e per far ciò possiamo eseguire 2 tipi di test:

1. Acceptance test, effettuato dal cliente che verifica il funzionamento richiesto
2. Unit test, test eseguito per ogni funzione(unità) dal programmatore.

Inoltre il programmatore può utilizzare principalmente 2 tipi di test per verificare il funzionamento del programma:

- **Black box test**, test senza avere la conoscenza del codice e quindi simula l'utente
- **White test**, test di controllo del codice fatto a basso livello che comporta una ampia conoscenza del linguaggio di programmazione.

Il core del codice difficilmente viene modificato poiché più vecchio e quindi già sottoposto ad una ampia gestione degli errori; quindi conviene memorizzare i test effettuati per future espansioni del codice(nuove feature) che possono comprenderne il core del codice (test di rigressione). In caso di mancanza di risorse e di tempo durante la fase di test dobbiamo sempre testare i casi limite poiché quelli comuni vengono "controllati" dagli utenti del servizio.

1.2 Building a System

Nello sviluppare un sistema informatico ed applicativo passiamo da poche componenti di un semplice programma a centinaia se non migliaia di componenti che aumentano notevolmente la complessità e la gestione del sistema stesso. È questo il principale motivo per cui dobbiamo utilizzare la Software Engineering come una vera e propria disciplina. Gli obiettivi di questo capitolo sono:

- Definire dimensione e complessità di un sistema
- Descrivere i problemi tecnici e decisionali
- Dimostrare le preoccupazioni nello sviluppo e nel supporto attività di un grande software applicativo
- Descrivere l'impegno coordinativo necessario al processo, al prodotto e agli utenti

1.2.1 Size and Complexity

I problemi complessi sono formati da molteplici livelli sia in ampiezza che in profondità:

- **Breadth issue**

- Principali funzionalità
- Interfacce con sistemi esterni
- Utenti simultanei
- Tipi di dati e di strutture dati

- **Depth Issue**

- Collegamenti (Condivisione e Controllo del flusso di dati)
- Relazioni (Gerarchia, Cicli, Ricorsione)

I moduli del software devono risultare coesi, cioè devono fare poche ed indispensabili azioni e devono essere poco accoppiati gli uni con gli altri(debole dipendenza).

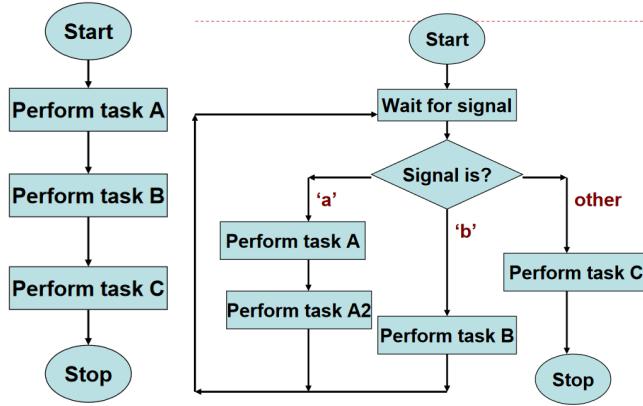


Figure 2: Un relativo incremento del numero di task e di decisioni aumentano notevolmente la complessità

1.2.2 Problemi decisionali

Il problema di base nella gestione di un sistema formato da molteplici componenti è quello di definire come gestire il codice, le componenti e le relazioni. La soluzione più comune ed intuitiva è la **Modularization** basata sul concetto di "divide et conquer"(dividi e conquista); la modularizzazione semplifica il problema complesso spezzandolo in piccoli sottoproblemi. Per decomporre e spezzare il problema complesso in più parti dobbiamo eseguire le seguenti attività:

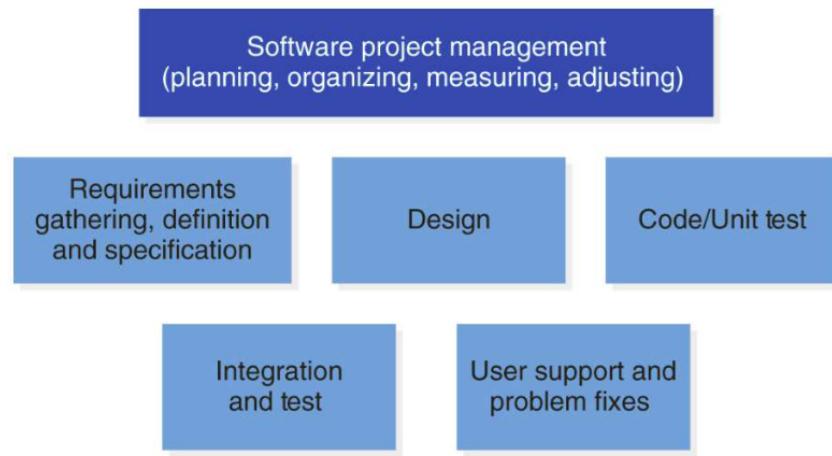
1. **Decomposition**, decomposizione in funzioni.
2. **Modularization**, sviluppo in moduli.
3. **Separation**, separazione e indipendenza dei moduli.

1.2.3 Technical issues and Methodology

La scelta di uno specifico linguaggio di programmazione può portare a notevoli problemi nel gestire e sviluppare applicazioni grandi e complesse. Bisogna cercare di selezionare un linguaggio di programmazione e un ambiente di sviluppo (tool, database, network, code version control, middleware) comune a tutti gli sviluppatori. Anche in team molto piccoli (1-3 persone) c'è bisogno di capire il problema ed i requisiti, definire una soluzione ed implementarla e testare l'applicazione; ma, a differenza di grandi team, non c'è bisogno di molto comunicazione tra gli utenti, non c'è bisogno di passaggio di informazioni, c'è la necessità di documentare il lavoro e non c'è bisogno di una grande organizzazione del lavoro. Quando ci troviamo in situazioni di sviluppo complesso e di notevoli dimensioni c'è la necessità di un processo di sviluppo software in grado di guidare e coordinare il gruppo di attori coinvolti(utenti, sviluppatori, ...). Il processo di sviluppo permette di definire:

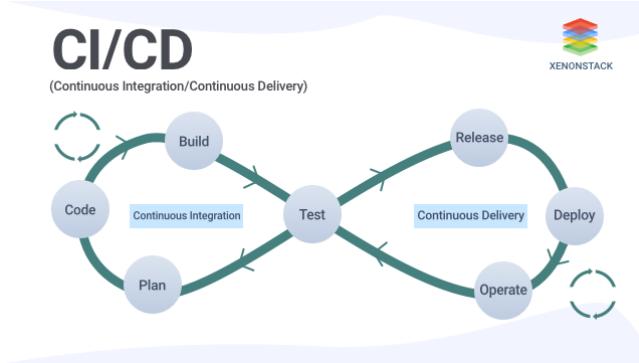
- le attività e la loro sequenza
- input e output di ogni attività
- le pre-condizioni e le post-condizioni per ogni attività

Per assicurare la buon riuscita del processo di sviluppo bisogna definire una sintassi ed un metodo comune nell'individuazioni di attività.



Queste sono le piú comuni attività eseguite durante il processo di sviluppo e di supporto software. Notiamo come le attività sembrano essere indipendenti l'una dall'altra ma se molte persone sono coinvolte nel processo abbiamo bisogno di definire l'esatta sequenza di attività e le condizioni iniziali di ogni attività. Per esempio un modello molto utilizzato é quello di **Incremental Development and Continuos integration** in cui le attività principali(compresa la release) vengono svolte per ogni funzione/requisito necessario al funzionamento dell'applicativo.

Un altro modello molto utilizzato é il **Continuous integration and Continuous deployment** in cui avviene la rapida implementazione di funzionalità complete tramite l'integrazione continua delle funzionalità completate nel prodotto e il rapido rilascio di tali funzionalità agli utenti.



1.2.4 Nontechnical considerations

La maggior parte dei progetti falliscono per una errata stima dell'effort e per non avere utilizzato una affidabile programmazione delle fasi del progetto. In particolar modo nei sistemi di grandi dimensioni, l'acquisizione e comprensione dei requisiti può essere travolgente. C'è bisogno quindi di stimare l'effort e di definire una schedule/programmazione. Per una buona stima dell'effort bisogna avere un'ottima conoscenza di tutte le features funzionali del progetto e della produttività dei team di sviluppo impegnati nel progetto. Nei progetti di grandi dimensioni, però, le features possono essere centinaia e quindi la probabilità di avere a disposizione le ipotesi iniziali è molto bassa; in questi tipi di progetti il risultato della stima è, nella maggior parte, una ipotesi ottimistica dell'effort lontana dall'accuratezza. Invece per definire una buona schedule bisogna conoscere le skills delle persone coinvolte nello sviluppo e le tasks da sviluppare così da poter assegnare le task alle persone più adatte.

Tra i vari team di sviluppo e tra tutte le persone coinvolte nel progetto avviene una comunicazione ed uno scambio di informazioni che se non gestito rallenta notevolmente lo sviluppo; la comunicazione "aumenta" all'aumentare del numero di partecipanti al progetto e quindi è chiaro che in grandi team la gestione della comunicazione gioca un ruolo fondamentale nel successo del progetto.

1.2.5 Other considerations

- Il software prima di essere rilasciato viene valutato (esente da errori) e validato (il prodotto soddisfa le richieste del cliente)
- Il costo degli errori e dei difetti durante la fase di acquisizione dei requisiti è terribilmente elevato. Le attività relative al completamento di una specifica dei requisiti sono difficili e hanno un impatto significativo su tutto il downstream delle attività e sul prodotto finale
- L'analisi dei requisiti deve essere documentata così da essere consultabile durante le fasi dello sviluppo; inoltre durante questa fase potrebbe essere necessario un team di analisti dei requisiti (specializzati nel dominio

individuato es.Tasse, Finanza, ...).

- L'interazione o l'accoppiamento dei componenti verticali (componenti specifici) con i servizi comuni (moduli orizzontali a tutte le funzionalità) è un problema chiave di progettazione
- Se ci sono molte unità di programmazione dobbiamo definire degli standard (convezioni per la definizione dei moduli, per la gestione dei dati, ...)
- Dopo che le unità funzionali sono state testate e integrate in componenti, i componenti devono essere testati insieme; ciò garantisce che l'intero sistema funzioni nel suo complesso
- L'utente deve inoltre essere istruito all'uso del sistema e questo richiede la preparazione della documentazione e del personale a supporto degli utenti nei dettagli del software e nella raccolta di segnalazioni sui bug.
- Almeno due gruppi di personale di supporto con competenze distinte sono necessarie per supportare un progetto complesso:
 1. Un gruppo per rispondere e gestire l'utilizzo del sistema e per dare semplici soluzioni ai problemi
 2. Un gruppo per risolvere problemi specifici e implementare miglioramenti futuri
- La coordinazione e la gestione dell'effort migliora la qualità del software e aumenta la produttività degli sviluppatori di software

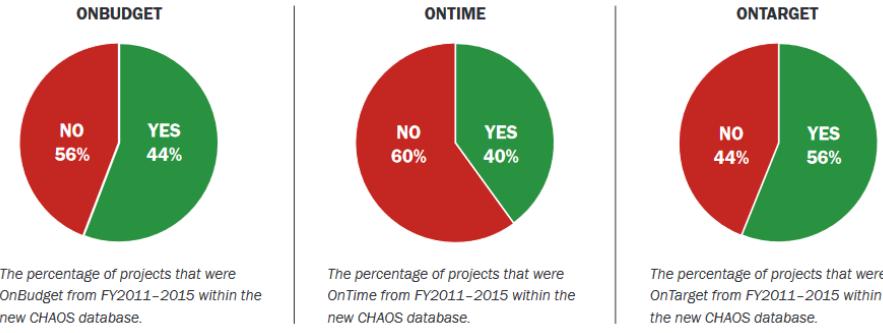
1.3 Software Engineering

In questa sezione discuteremo ed analizzeremo le principali cause che portano al fallimento di un progetto, definiremo il concetto di software engineering e introdurremo i concetti fondamentali di un modello di sviluppo.

1.3.1 Software Failures

Con l'aumentare delle dimensioni e della complessità dei progetti software, aumenta anche il numero dei progetti falliti. L'ingegneria del software cerca di definire una certa disciplina della "programmazione" utilizzando un modello "study-understand-improve" e porta lo sviluppo software ad non essere solo programmazione/coding.

Standish Group nel 1995 investigò su un campione di 300 progetti software, riportando che solo il 16% di questi progetti erano stati completati in tempo e rispettando il budget mentre il restante 84% erano tutti falliti (Chaos Report). Nuovamente nel 2009 (+14 anni) Standish Group analizzò un altro campione di progetti software riportando che lo sviluppo software era migliorato con il 32% di progetti completati in tempo e rispettando il budget ma comunque la



maggior parte di essi (64%) continuava a fallire. I principali fattori di fallimento dei progetti software sono:

- Mancanza di input da parte dell'utente/cliente
- Erronea analisi dei requisiti
- Cambio dei requisiti
- Mancanza di coinvolgimento del cliente
- Mancanza di risorse e di supporto esecutivo
- Manca di un modello di sviluppo
- Manca di un planning

Il coinvolgimento e le esigenze degli utenti sono fattori chiave di successo perché senza capire cosa deve essere sviluppato, c'è solo una piccola possibilità di successo per qualsiasi progetto. Il fallimento del software include molti tipi di problemi (Capers Jones' studies):

- Errori generati dal codice (38,33%)
- Errori durante la fase di analisi dei requisiti che si propagano sulla fase di design e di coding (12,5% + 24,17%). Questi errori sono quelli che generalmente incidono notevolmente sul budget e sul tempo.
- Errori di coordinazione durante la fase di sviluppo (bad management); per ridurre i rischi molte aziende delegano la produzione di software a terzi.

In generale per sviluppare bene rispettando tempo e budget richiede i 3 seguenti fattori:

1. Attenzione focalizzata sull'ambiente di sviluppo software (persone/strumenti/gestione/ecc.)

2. Processo di sviluppo “disciplinato” (Metodologia).
3. Uso metodico di metriche per misurare costi, tempi e obiettivi prestazionali (introdurre delle metriche significa anche imparare ad utilizzarle ed ottimizzarle)

1.3.2 Definition of Software Engineering

Il termine è stato introdotto per la prima volta in una conferenza NATO del 1968. Infatti durante gli anni '60 il campo informatico stava iniziando ad affrontare una crisi del software; il notevole sviluppo hardware aveva aumentato la potenza di calcolo e la capacità di storage dei calcolatori permettendo lo sviluppo di grandi e complesse applicazioni software che richiedevano però un numero molto grande di programmatore. Il software non riusciva a stare al passo dello sviluppo hardware e la maggior parte dei progetti falliva. Serviva una metodologia di programmazione e serviva definire una certa disciplina di sviluppo software (**Software Engineer**). Vediamo ora alcune delle molteplici definizioni di Software Engineer:

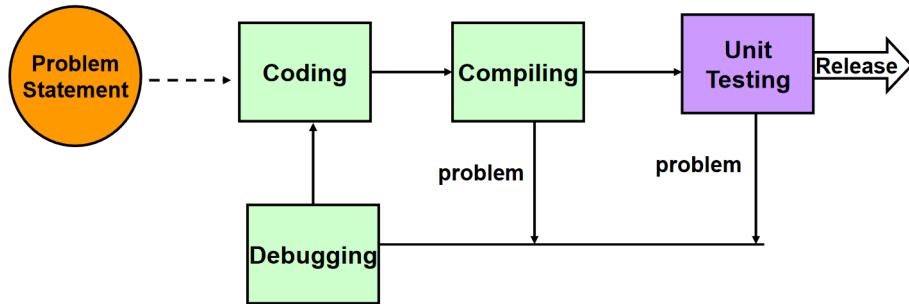
- *Software Engineer is a multiperson construction of multiversion software (Parnas 2001)*
- *An engineering discipline whose focus on the cost-effective development of high quality software system (Sommerville 2004)* (spiega anche che il software è astratto ed intangibile)
- *The establishment and use of sound engineering principles (methods) in order to obtain economically software that is reliable and works on real machines (Bauer)*
- *form of engineering that applies the principles of computer science and mathematics to achieving cost-effective solutions to software problems (Software Engineering Institute, Carnegie Mellon University)*
- *application of a systematic, disciplined, quantifiable approach to the i) development of, ii) operation of, and iii) maintenance of software” (IEEE std 610-1990)*
- *Software Engineering is about creating high-quality software in a systematic, controlled and efficient manner. (Curriculum Guidelines)*

1.3.3 Software Process

Il processo di sviluppo e supporto del software richiede molti compiti distinti che devono essere svolti da persone diverse in una determinata sequenza. I modelli di sviluppo software servono per fornire una guida per il coordinamento sistematico e controllare i compiti da svolgere per raggiungere il fine del prodotto e gli obiettivi del progetto; questi modelli definiscono l'insieme delle attività che devono essere eseguite, l'input e l'output di ciascuna attività, le precondizioni e le postcondizioni per ogni task e la sequenza e il flusso di questi compiti.

1. "Simplest" Process Model

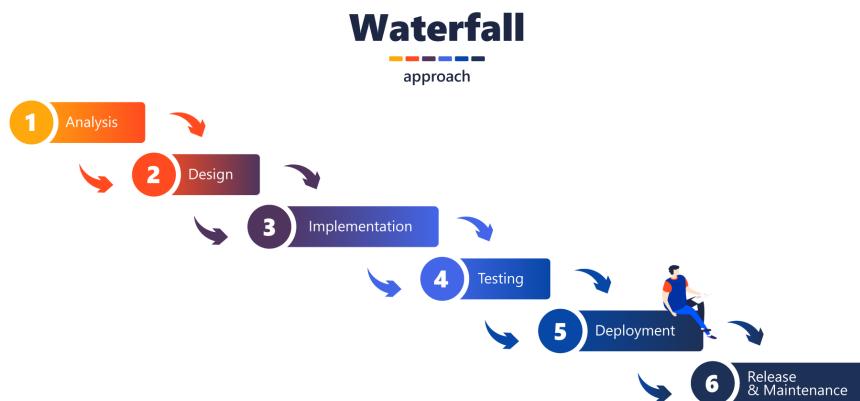
Il modello più semplice di tutti si chiama "code and fix" (utilizzato di solito da singoli programmatori).



Man mano che i progetti diventavano più grandi e complessi sussiste la necessità di chiarire e stabilizzare i requisiti, di testare più funzionalità e di progettare con maggiore attenzione.

2. Waterfall Model

Le attività si verificano in sequenza una dopo l'altra con l'output di un'attività che diventa l'input dell'attività successiva (Royce 1970).

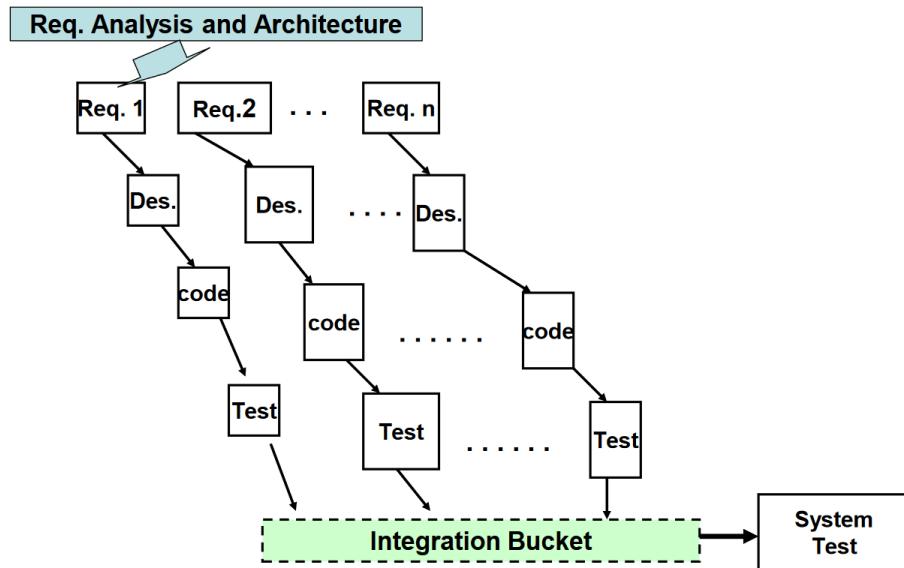


Questo modello porta i seguenti aspetti positivi: i requisiti devono essere specificati nella prima fase, le 4 attività principali devono essere complete prima che il sistema possa essere impacchettato/release (requisiti,

design, codice e test), l'output di ogni fase viene immesso nella fase successiva in sequenza, il progetto può essere monitorato e si sposta in sequenza attraverso compiti identificabili (È anche chiamato document-driven approach per la grande quantità di documenti generati in ogni fase). Il waterfall model è caduto in disuso per via di una bassa interazione con il cliente (solo nella fase di analisi requisiti) e per il ruolo centrale e fondamentale che ricopriva l'analisi dei requisiti (errori cadono a cascata).

3. Incremental Model

I grandi progetti sono molto più facili se suddivisi in più piccoli componenti che possono essere sviluppati in modo incrementale e iterativo.



Attua una sorta di contenimento del rischio: se un componente ha un problema, gli altri componenti possono essere sviluppati in modo indipendente; inoltre consente di sviluppare in primo luogo i componenti principali con le funzionalità aggiuntive che possono essere rilasciate in seguito.

2 Specifica dei Requisiti

2.1 Requirements Engineering

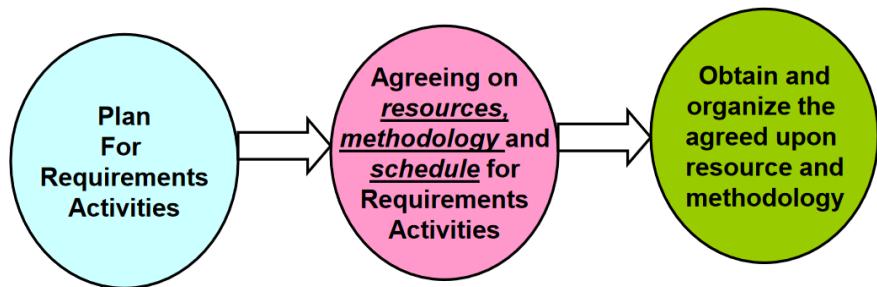
Come già ampiamente visto in precedenza, uno dei motivi principali del fallimento dei progetti software è l'incompletezza delle specifiche dei requisiti ed allo stesso tempo una delle ragioni più importanti per il successo di un progetto può essere attribuito alla chiarezza nella dichiarazione dei requisiti.

Per effettuare una buona analisi dei requisiti dobbiamo per prima definire e chiarire il concetto stesso di requisito. I requisiti sono un insieme di dichiarazioni che descrivono le esigenze e i desideri dell'utente che devono essere pienamente compresi dagli ingegneri del software che sviluppano il sistema software. I requisiti devono essere basati su quali operazioni deve eseguire il sistema software(*what*) piú che sul come realizzare l'applicativo (*how*); infatti i requisiti devono essere indipendenti dalla tecnologia da utilizzare e dall'implementazione specifica.

La branca dell'ingegneria del software che si occupa dell'analisi dei requisiti si chiama **Requirement Engineering** e consiste in un insieme di attività legate allo sviluppo e all'accordo sull'insieme finale delle specifiche dei requisiti. Le principali attività sono (non tutte sono necessarie):

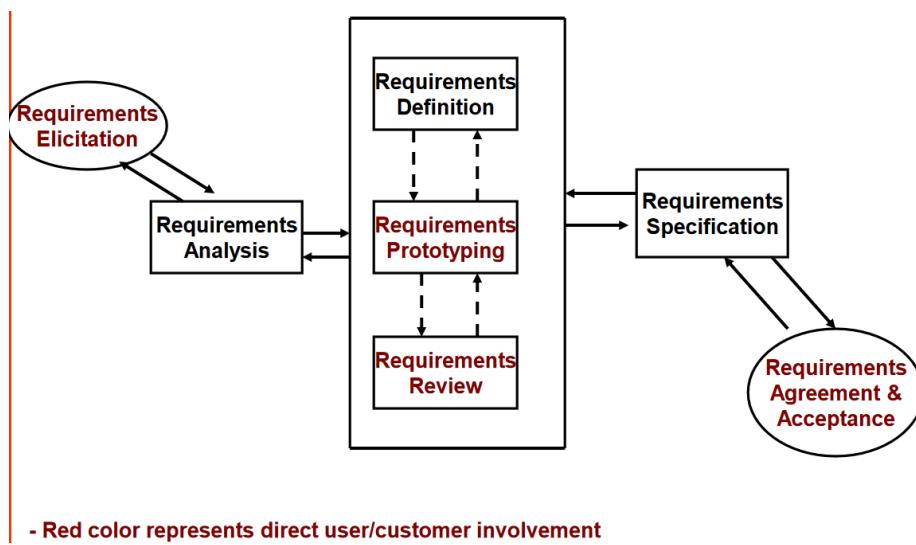
- Elicitation, fase di raccolta dei requisiti
- Documentazione e definizione dei requisiti
- Specifica dei requisiti
- Prototyping
- Analisi
- Controllo e validazione
- Accordo finale

Come tutte le fasi della progettazione anche l'analisi dei requisiti deve essere pianificata impostando priorità sulle risorse, la metodologia e lo schedule dei vari requisiti. Durante questa fase dobbiamo pianificare anche il modo in cui clienti ed utenti dovranno essere coinvolti visto che i requisiti rappresentano i loro desideri; inoltre anche il management dovrà essere coinvolto per avere una chiara situazione sulle risorse da allocare e sui costi. Infine il piano deve essere rivisto e approvato da tutte le parti coinvolte.



Dopo aver concordato il piano, le risorse - dagli analisti esperti agli strumenti di prototipazione necessari, devono essere acquisite. Un buon analista dei requisiti deve possedere molteplici talenti, quali capacità di comunicazione (ottima ascoltatore ed interprete), competenze specifiche del settore e competenze tecniche e quindi le persone coinvolte devono essere adeguatamente formate sul processo e sugli strumenti che verranno impiegati.

Una volta completata la preparazione per analisi dei requisiti, comincia l'effettivo sviluppo dei requisiti che prevede diverse fasi (nella metodologia agile non si rispettano tutte queste fasi perché c'è una costante interazione con gli utenti che assicurano che i requisiti siano interpretati correttamente).



Qualsiasi modifica o richiesta di cambiamento deve essere controllata e gestita attraverso un processo per evitare il famigerato **project scope-creeping problem**, cioè che il progetto cresce lentamente di dimensione senza che nessuno se ne accorga (aumento esponenziale dei costi). Infatti l'errore di non dedicare tempo

e fatica alla raccolta e alla comprensione dei requisiti può essere costoso (no requisiti documentati su cui basare i test, no requisiti concordati per controllare l'ampliamento dell'ambito ed il flusso del progetto, no requisiti documentati su cui basare le attività di formazione e assistenza al cliente, difficoltà nel gestire i tempi).

Vediamo ora nel dettaglio tutte le fasi dell'analisi dei requisiti.

2.1.1 Elicitation

In questa fase avviene la raccolta della maggior parte, se non la totalità, dei requisiti. Gli analisti dei requisiti devono disporre di un insieme di domande organizzate da porre agli utenti in diversi modi:

- Verbal, raccomandato perché il contatto personale e diretto con gli utenti spesso innesca buone domande di follow-up e consentirà agli utenti di ampliare i loro input.
- Written (preformatted form)
- Online form (a distanza)

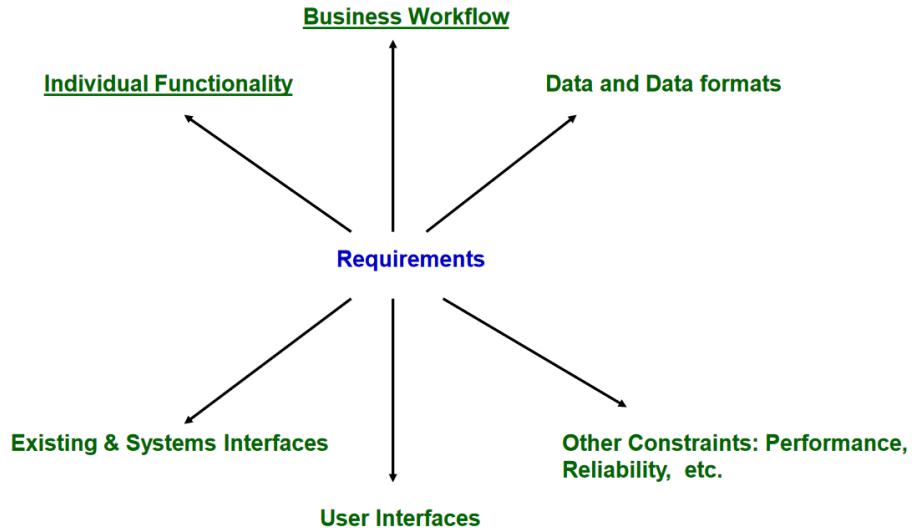
Esistono due livelli di elicitazione dei requisiti:

- **High level**, ad alto livello per comprendere le motivazioni di business e la giustificazione per lo sviluppo del software (generalità di carattere organizzativo ed economico). La categoria di informazioni che contribuiscono a questo profilo business di alto livello comprende i seguenti elementi:

- Opportunità/necessità che stabiliscono i limiti e l'ambito del progetto software.
- Giustificazione
- Ambito di applicazione
- Vincoli principali che stabiliscono limiti di budget e di tempo
- Principale funzionalità che il nuovo software dovrà fornire (aspettative del cliente)
- Fattore di successo
- Caratteristiche dell'utente per lo sviluppo della UI

L'analista dei requisiti dovrebbe trasformare questi requisiti di alto livello in obiettivi di alto livello infatti il progetto è spesso considerato un successo se gli obiettivi di business di alto livello vengono raggiunti.

- **Low level**, a basso livello per raccogliere i dettagli delle esigenze e dei desideri degli utenti (requisiti specifici, dettagli informatici)



Le funzionalità individuali (specifiche del dominio del progetto) sono di solito il punto di partenza naturale dell'elicitazione dei requisiti per poterle posizionare nel flusso aziendale. Poi è necessario raccogliere le informazioni relative ai dati e ai formati dei dati; come minimo, è necessario conoscere almeno i dati di ingresso e di uscita dell'applicazione. Bisogna inoltre definire come gestire le interfacce utente e quindi come vengono presentati gli input e gli output di un software (tipicamente catturati da prototipazione dell'interfaccia) e gli eventuali messaggi di errore. Infine i requisiti di affidabilità, prestazioni, sicurezza, adattabilità, trasportabilità e manutenibilità (di norma requisiti non funzionali).

2.1.2 Analysis

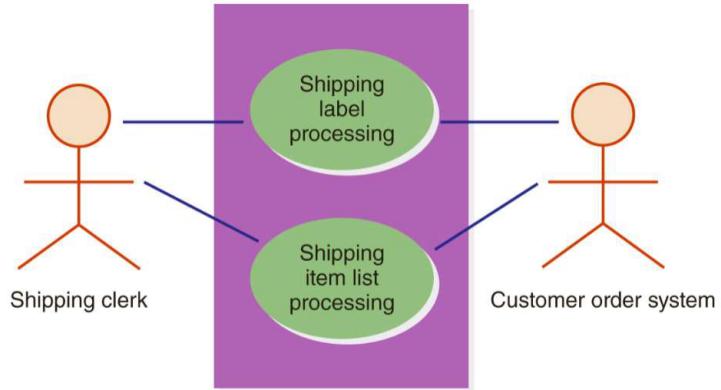
Anche dopo che i requisiti sono stati elicitati e raccolti, sono ancora solo un insieme di dati non organizzati. L'analisi dei requisiti consiste in due compiti principali:

1. Categorizzazione o raggruppamento dei requisiti
2. Definizione delle priorità dei requisiti

Un raggruppamento naturale può seguire le sei dimensioni dei requisiti ma non esiste una regola generale perché la definizione del raggruppamento può seguire diverse regole (anche regole specifiche del dominio del progetto).

Requirement Area	Prefix	Requirement Statement Numbering
Individual functionality	IF	IF-1.1, IF-1.2, IF-1.3, IF-2.1
Business flow	BF	BF-1, BF-2
Data and data format	DF	DF-1.1, DF-1.2, DF-2.1
User interface	UI	UI-1.1, UI-1.2, UI-2.1
Interface to systems	IS	IS-1
Further constraints	FC	FC-1

L'analisi dei requisiti può essere fatta, in particolar modo in progetti OO, attraverso l'uso dello use case diagram che è fondamentalmente una rappresentazione delle seguenti informazioni sui requisiti: funzionalità di base, precondizioni per le funzionalità , flusso di eventi(scenario) per la funzionalità, postcondizioni per la funzionalità, condizioni di errore e flusso alternativo. Attraverso lo use case diagram definiamo anche i contorni del sistema con la delimitazione di ciò che è incluso ed escluso dal sistema.



Vengono usati anche usati diversi patterns come il **FURPS+**, in cui ogni requisito deve esprimere almeno una delle seguenti caratteristiche:

- Funzionalità: le principali caratteristiche del prodotto
- Usabilità: la facilità con cui l'utente può imparare a operare, preparare gli input e interpretare gli output di un sistema.
- Affidabilità: la capacità di un sistema di svolgere le funzioni richieste in determinate condizioni per un determinato periodo di tempo
- Prestazioni: attributi quantificabili come tempo di risposta, precisione, ...

- Supportabilità: facilità di modifica del sistema dopo l'implementazione (adattabilità, manutenibilità, portabilità, ecc...).
- +, Requisiti di progettazione, Requisiti di implementazione, Requisiti di interfaccia, Requisiti fisici

Un altro modello molto utilizzato è il **VORD**(Viewpoint-oriented requirements definition) in cui per ogni funzionalità dobbiamo individuare gli stakeholder, cioè le parti che interagiscono con la funzionalità elicitata. In particolar modo questa metodologia si divide nei seguenti punti:

1. Identificare gli stakeholder e i vari punti di vista (requisiti)
2. Documentare i vari punti di vista (punto di vista dei dati, dell'UI, della sicurezza, ecc.)
3. Mappare i punti di vista sul sistema e sui servizi che il sistema deve fornire

Una volta raggruppati i requisiti è necessario stabilire un ordine di priorità in modo che quelli a più alta priorità vengano sviluppati per primi. La priorità può essere stabilita in base a numerosi criteri: le attuali richieste dei clienti, concorrenza e condizioni attuali del mercato, esigenze future dei clienti, vantaggio di vendita immediato, problemi critici del prodotto esistente. Inoltre la priorità viene discussa anche con il cliente, con il management ed con un esperto del dominio del progetto.

Requirement Number	Brief Requirement Description	Requirement Source	Requirement Priority*	Requirement Status
1	One-page query must respond in less than 1 second	A major account marketing representative	Priority 1	Accepted for this release
2	Help text must be field sensitive	Large account users	Priority 2	Postponed for next release

*Priority may be 1, 2, 3, or 4, with 1 being the highest.

Un approccio metodico è l'Analytical Hierarchical Process(AHP) che introduce un maggior rigore nel processo di definizione delle priorità dei requisiti. Ogni requisito viene confrontato con ciascuno degli altri requisiti a coppie ed a questa relazione viene assegnato un "valore di intensità"(il valore di intensità viene discusso). Il requisito con i valori complessivi di intensità più alti sarà essenzialmente il requisito con la priorità più alta; il requisito con il successivo valore di intensità relativa più alto sarà il requisito successivo con la priorità più alta e così via.

	Req 1	Req 2	Req 3
Req 1	1	2	3
Req 2	1/2	1	2
Req 3	1/3	1/2	1
↓	1.83	3.5	6.0
	Req 1	Req 2	Req 3
Req 1	.55	.57	.5
Req 2	.27	.29	.33
Req 3	.18	.14	.17

(x,y) → increasing value
of x w.r.t. y (from 1 to 9)

Requirements Prioritization

$$\text{Req 1} = 1.62 / 3 = 54\%$$

$$\text{Req 2} = .89 / 3 = 30\%$$

$$\text{Req 3} = .49 / 3 = 16\%$$

In realtà vengono fatte tante tabelle in base a quante parti sono interessate nel progetto (sviluppatori, clienti e management) ed attraverso una media si raggruppano tutti i valori di intesità in una tabella finale che permetterà di stabilire la priorità dei requisiti.

Requirement ID	Magnus	Elmira	Faegheh	Niclas	Farhan	Sean	Sarah	Total
FR1	9	7	5	6	6	8	6	47
FR2	3	6	4	3	4	5	3	28
FR3	4	6	8	6	7	7	6	44
FR4	6	5	3	6	7	6	6	39
FR5	6	9	3	5	7	6	5	41
FR6	8	10	10	9	9	10	10	66
FR7	10	8	10	10	8	10	8	64
FR8	10	10	10	8	8	10	8	64
FR9	6	8	5	8	9	7	8	51
FR10	3	6	7	5	6	8	4	39
FR11	3	4	3	6	5	5	5	31

Figure 3: Valutazione dei requisiti da parte dei developer (scala 1-10)

10 most important requirements

Requirement ID	Title	Requirement Type
FR6	Mobile application - Search	Function
FR7	Mobile application - Search result in a map view	Function
FR8	Mobile application - Search result in a list view	Function
FR24	Restaurant owner manages information	Function
FR27	Administrator verifies restaurant owner	Function
QR6	System response time	Quality
QR7	System Availability	Quality
QR9	System Reliability	Quality
QR12	Communication Security	Quality
QR22	Internet Connection	Quality

Value

Value	FR6	FR7	FR8	FR24	FR27	QR6	QR7	QR9	QR12	QR22
FR6	1	5	7	7	1/3	1/5	1/3	1/3	5	7
FR7	1/5	1	3	5	6	1/5	1/3	1/3	1/3	5
FR8	1/7	1/3	1	4	5	1/6	1/4	1/4	1/5	3
FR24	1/7	1/5	1/4	1	1/3	1/5	1/5	1/3	5	4
FR27	3	1/6	1/5	3	1	1/9	1/5	1/5	1/7	2
QR6	5	5	6	5	9	1	3	3	2	8

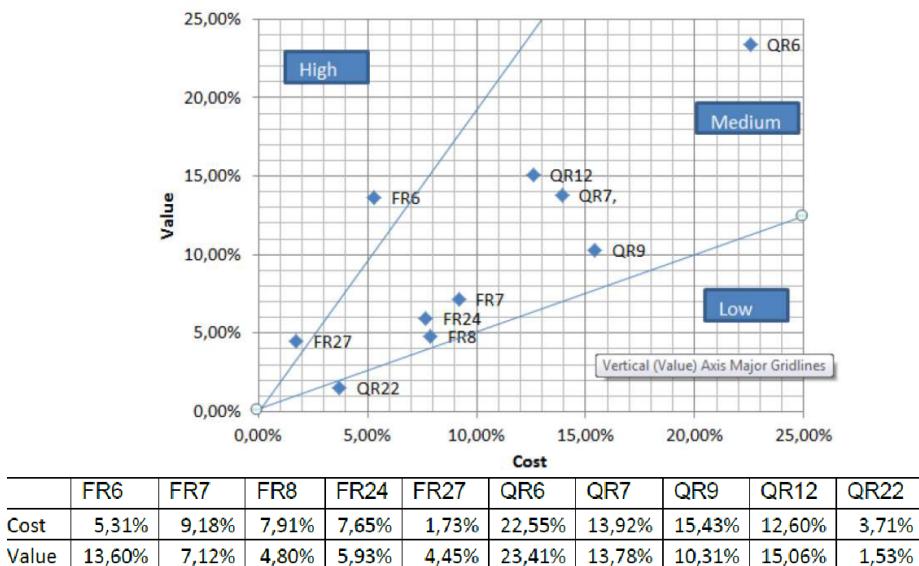
Figure 4: Valutazione dei 10 requisiti in base al valore portato

Cost	COST	FR6	FR7	FR8	FR24	FR27	QR6	QR7	QR9	QR12	QR22
	FR6	1	1/5	1/2	3	5	1/7	1/3	1/5	1/3	7
	FR7	5	1	1/5	7	3	1/5	1/3	1/5	3	5
	FR8	2	5	1	3	5	1/9	1/5	1/6	1/5	7
	FR24	1/3	1/7	1/3	1	3	1/3	2	1/5	3	7
	FR27	1/5	1/3	1/5	1/3	1	1/7	1/6	1/7	1/6	2
	QR6	7	5	9	3	7	1	3	2	3	9
	QR7	3	3	5	1/2	6	1/3	1	2	3	7
	QR9	5	5	6	5	7	1/2	1/2	1	3	5

----- Francesco Guerra -----

Figure 5: Valutazione dei 10 requisiti in base al costo da sostenere

Alla fine normalizziamo i valori e mappiamo i valori su un piano cartesiano (non é sempre necessario) per stabilire la prioritá.



Questo non é l'unico di procedere, infatti esistono anche approcci alternativi basati sul VORD, sul FURPS+, ecc.

Requirement ID		Priority by Stakeholder			
		User	Restaurant owner	Administrator	Customer
FR1	Magnus	1	-2	-2	2
	Elmira	1	-1	-2	2
	Niclas	1	-2	-2	1
	Faegheh	1	-2	-2	1
	Sarah	1	-1	-1	1
	Average	1	-1,6	-1,8	1,4
Sum					-1
Rank of Req.		13			
FR2	Magnus	1	-2	-2	1
	Elmira	1	-1	-2	1
	Niclas	0	-2	-2	1
	Faegheh	-1	-2	-2	0
	Sarah	0	-2	-2	0
	Average	0,2	-1,8	-2	0,6
Sum					-3
Rank of Req.		34			

Una volta definiti requisiti ed aver stabilito un ordine tra loro risulta molto utile definire una tracciabilità dei requisiti, cioè la capacità di risalire ai requisiti dopo lo sviluppo e di verificare che tutti i requisiti siano stati sviluppati, testati, confezionati e consegnati. Kotonya e Sommerville (1998) hanno elencato quattro tipi di tracciabilità:

- Tracciabilità a ritroso (Backward from): Collega il requisito alla fonte del documento o alla persona che l'ha creato.
- In avanti rispetto alla tracciabilità (Forward from): Collega il requisito alla progettazione e all'implementazione.
- All'indietro rispetto alla tracciabilità (Backward to): Collega la progettazione e l'implementazione ai requisiti.
- In avanti verso la tracciabilità(Forward to): Collega i documenti precedenti ai requisiti

2.1.3 Definition, Prototyping, and Reviews

La definizione , la prototipazione e la revisione dei requisiti sono rappresentate come tre attività separate ma nella pratica spesso si sovrappongono. La definizione dei requisiti comporta la formalizzazione dei requisiti; una delle notazioni più semplici per la definizione dei requisiti in termini di linguaggio naturale è la notazione input-process-output. Le notazioni più diffuse a supporto del processo sono UML e in particolare use case, activity diagram e il DFD (Data Flow Diagram).

Requirement Number	Input	Process	Output
12: Customer order	<ul style="list-style-type: none"> • Items by type and quantity • Submit request 	<ul style="list-style-type: none"> • Accept the items and respective quantities 	<ul style="list-style-type: none"> • Display acceptance message • Ask for confirmation message

Come parte della raccolta e dell'analisi dei requisiti, l'interfaccia utente è un componente fondamentale e di conseguenza è richiesta una prototipazione. Sono 2 gli aspetti principali dell'interfaccia utente che devono essere analizzati:

- Aspetto e visualizzazione visiva
- Interazione con le persone e flusso

Oggi, l'interfaccia utente viene prototipata con codice eseguibile dalla macchina durante la raccolta dei requisiti e l'analisi ed a volte questi prototipi vengono conservati e trasformati nel codice finale. Infine abbiamo la revisione dei requisiti con gli utenti e i clienti che è una parte essenziale dell'analisi dei requisiti e della prototipazione; infatti un singolo errore di un requisito spesso si espande in molteplici errori di progettazione, ognuno dei quali può a sua volta diventare fonte di diversi errori di programmazione. In genere è condotta in modo informale e frequentemente ma può essere eseguita anche con una metodologia formale (IBM-1970s) dove dopo le revisioni, ogni modifica e correzione deve essere apportata alle definizioni dei requisiti.

2.1.4 Specification and Agreement

Una volta che i requisiti sono stati analizzati e revisionati, è prudente inserirli in un documento di specifica dei requisiti. Esistono tantissimi standard per la specifica dei requisiti, uno dei più popolari e famosi è l'**ISO/IEC/IEEE 29148**. Questo documento specifica i processi richiesti per l'implementazione delle attività di ingegneria che danno origine ai requisiti per i sistemi e i prodotti software, fornisce linee guida per l'applicazione dei requisiti e dei processi legati ai requisiti, specifica gli elementi informativi richiesti prodotti attraverso l'implementazione dei processi relativi ai requisiti, specifica i contenuti richiesti degli elementi informativi richiesti, fornisce le linee guida per il formato degli elementi informativi richiesti. È un documento molto complesso formato in linea generale da 4 parti:

- Business requirement specification (BRS), cioè la specifica dei requisiti legati alla parte economica e al management.
- Stakeholder requirements specification (StRS), requisiti delle parti interessate.
- System requirements specification (SyRS), requisiti legate al sistema da sviluppare.
- Software requirements specification (SRS), requisiti legate al software.

2.2 UML - Unified Modeling Language

Nell'ambito della tecnologia dell'informazione, siamo interessati ai sistemi software ed ai modelli che descrivono questi sistemi software. Prima di continuare dobbiamo dare le definizioni di modello e sistema:

- Un sistema è un insieme integrato di componenti che sono in relazione tra loro e si influenzano reciprocamente in modo tale da poter essere percepiti come un'unità unica.
- Un modello è un oggetto che permette di descrivere i sistemi in modo efficiente ed elegante.

La maggior parte dei sistemi software è basata sull'astrazione di un processo reale, cioè è basata sulla generalizzazione dell'oggetto reale nascondendo alcune caratteristiche uniche. Attraverso l'astrazione di uno oggetto reale definiamo un modello che raggruppa le caratteristiche comuni di una determinata famiglia di sistemi. Il modello ha 3 caratteristiche (Herbert Stachowiak 1973):

1. Mappatura: un modello è sempre un'immagine (mappatura) di qualcosa, una rappresentazione di oggetti originali naturali o artificiali che possono essere essi stessi modelli.
2. Riduzione: un modello non cattura tutti gli attributi dell'oggetto originale, ma solo quelli che sembrano rilevanti per il modellatore o l'utente del modello.
3. Pragmatismo: pragmatismo significa orientamento all'utilità. Un modello viene assegnato ad un oggetto reale in base alle seguenti domande: Per chi? Perché? Per cosa?

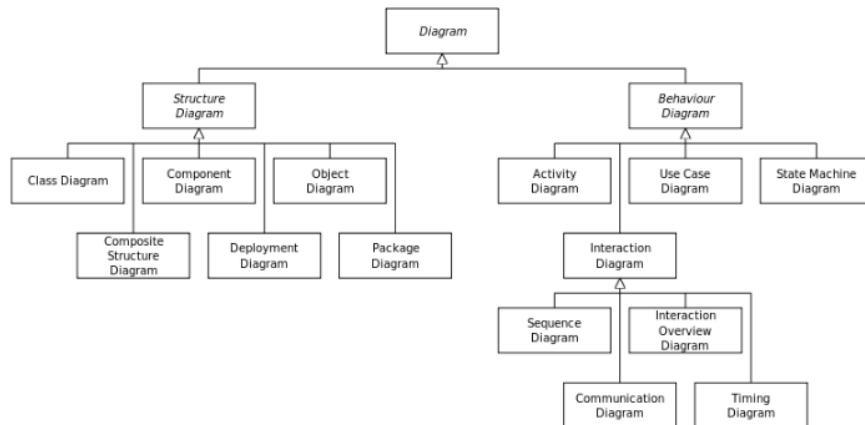
La qualità di un modello viene valutata secondo le seguenti metriche (Bran Selic):

- Astrazione: un modello è una rappresentazione ridotta del sistema che rappresenta. I dettagli che sono irrilevanti in un contesto specifico vengono nascosti o rimossi per facilitare la comprensione dell'insieme.
- Comprensibilità: è importante presentare gli elementi nel modo più intuitivo possibile, per esempio, in una notazione grafica.
- Accuratezza: un modello deve evidenziare le proprietà rilevanti del sistema reale e quindi riflettere il più possibile la realtà
- Predittività: un modello deve consentire la previsione di proprietà interessanti ma non ovvie del sistema modellato.
- Costo-efficacia: nel lungo periodo, deve essere più conveniente creare il modello che creare il sistema da modellare. (minimizzazione dei costi)

The Unified Modeling Language (UML) è una famiglia di notazioni grafiche che aiutano a descrivere e progettare sistemi software, in particolare sistemi costruiti secondo lo stile orientato agli oggetti (OO). L'UML è uno standard relativamente aperto, controllato dall'Object Management Group (OMG); ha avuto un notevole sviluppo tra il 2000 ed il 2010 ma adesso viene utilizzato per lo più come standard accademico. L'UML può essere utilizzato in diversi modi:

- Come **Sketch** per aiutare a comunicare alcuni aspetti di un sistema
- Come sviluppo di **progetti**
- Come **linguaggio di programmazione**: gli sviluppatori disegnano diagrammi UML che vengono compilati in codice eseguibile.

Esistono tantissimi tipi di diagrammi UML tra i quali studieremo il Class diagram, l'Activity diagram, lo Use Case diagram ed il Sequence diagram.



UML è un linguaggio piuttosto preciso, ci si potrebbe aspettare delle regole prescrittive, cioè delle regole che esprimono prescrizioni (norme, comandi, direttive, consigli, ammonizioni, giudizi di valore), aventi la funzione di indirizzare il comportamento degli sviluppatori (come dovrebbe essere la realtà). Ma l'UML ha regole descrittive, cioè rappresenta una particolare realtà di cui però vengo nascosti alcuni dettagli (come è la realtà).

Infine non esiste una definizione formale di come l'UML si adatti a un particolare linguaggio di programmazione.

2.3 UML - Use Case Diagram

I diagrammi Use Case sono utilizzati per catturare i requisiti funzionali di un sistema descrivendo le interazioni tipiche tra gli utenti di un sistema e il sistema

stesso, fornendo una narrazione di come il sistema viene usato. Il contenuto di un diagramma use case esprime le aspettative del cliente ha nei confronti del sistema da sviluppare e documenta i requisiti che il sistema deve soddisfare; se i casi d'uso vengono dimenticati o specificati in modo non corretto, le conseguenze possono essere estremamente gravi portando ad un aumento dei costi di sviluppo e manutenzione e gli utenti ad essere insoddisfatti. Il processo di scrittura di un caso d'uso comprende le seguenti fasi:

1. Identificazione degli attori

Gli attori sono entità esterne che interagiscono con il sistema che però non rappresentano un utente specifico, ma i ruoli che gli utenti adottano. Se un utente ha adottato il rispettivo ruolo, è autorizzato a eseguire i casi d'uso associati a questo ruolo.

2. Identificazione dei casi d'uso

Un casi d'uso descrivono le funzionalità attese dal sistema da sviluppare e vengono determinati raccogliendo i desideri dei clienti e analizzando i problemi specificati in linguaggio naturale quando questi sono la base per l'analisi dei requisiti. Il diagramma Use case però non copre la struttura interna e l'implementazione effettiva di un caso d'uso (non c'è una associazione 1:1 tra implementazione e use case diagram).

3. Identificazione delle associazioni

Un attore è collegato ai casi d'uso tramite associazioni che esprimono come l'attore comunica con il sistema e come utilizza una determinata funzionalità. Ogni attore deve comunicare con almeno un caso d'uso e di conseguenza un caso d'uso deve essere in relazione con almeno un attore. (associazione binaria, se non viene specificata alcuna molteplicità all'estremità dell'associazione dell'attore, viene assunto 1 come valore predefinito.)

4. Identificazione degli scenari

Uno scenario è una descrizione narrativa di ciò che le persone fanno e sperimentano mentre utilizzano i sistemi e le applicazioni. In particolar modo è una descrizione concreta, mirata e informale di una caratteristica del sistema dal punto di vista di un singolo attore.

- Scenari as-is: descrivono la situazione attuale.
- Scenari visionari: descrivono un sistema futuro
- Scenari di valutazione: descrivono i compiti dell'utente rispetto ai quali il sistema deve essere valutato.
- Scenari di formazione: tutorial per introdurre nuovi utenti ai sistemi.

Lo use case diagram non esiste solo in forma grafica, infatti esiste anche una descrizione in forma scritta (forma tabellare) dei vari casi d'uso.

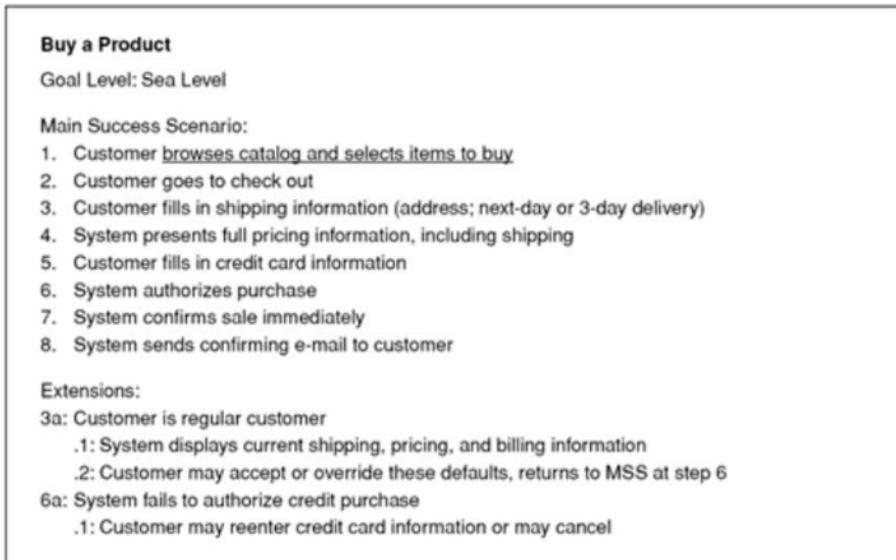
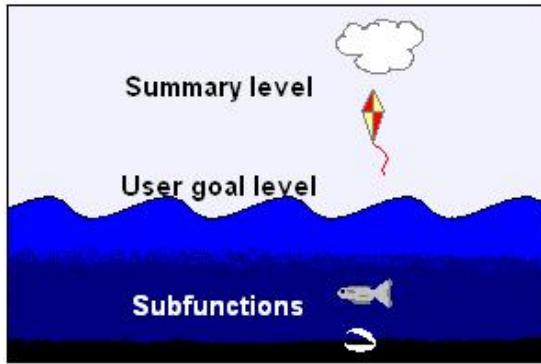


Figure 6: N. Inoltre

Attraverso la descrizione non grafica notiamo che ogni caso d'uso ha un attore principale, cioè l'attore con l'obiettivo che il caso d'uso sta cercando di soddisfare e ogni step è un elemento dell'interazione tra un attore e il sistema. (lo step mostra l'intento dell'attore, non la meccanica di ciò che fa). Inoltre un passo complicato in un caso d'uso può essere a sua volta un altro caso d'uso e per indicare ciò utilizziamo dei link (browses catalog and selects items to buy). In questo tipo di descrizione vengono anche definiti dei casi particolari(*extensions*) che nominano una condizione che comporta interazioni con l'attore principali diverse dal quotidiano. Infine notiamo che viene settato un determinato livello come obiettivo: in questo caso viene posto come Goal level il sea level; questa terminologia è stata introdotta da Alistair Cockburn nel suo libro "Writing effective use cases". L'idea principale è che i casi d'uso principali sono al "livello del mare" e rappresentano tipicamente un'interazione discreta tra un attore principale e il sistema(obiettivi concreti di utente aziendale). Ciò che è sopra il mare è il livello alto (quindi la nuvola e l'aquilone). Ciò che è sotto il mare, sono i dettagli che si trovano sotto la superficie e che non sono di primario interesse per l'utente aziendale (ad esempio perché sono dettagli che appaiono banali o troppo tecnici).



È possibile aggiungere altre informazioni comuni ad un determinato caso d'uso (bisogna essere scettici nell'aggiungere elementi; infatti è meglio fare troppo poco che troppo.)

- Una pre-condizione descrive ciò che il sistema deve garantire; questo è utile per indicare ai programmatore quali condizioni devono e quali non devono controllare nel loro codice.
- Una garanzia(garantuee) che descrive ciò che il sistema assicurerà alla fine del caso d'uso.
- Un trigger specifica l'evento che fa iniziare il caso d'uso.

Ora passiamo a parlare della descrizione grafica dello use case diagram. Il modo migliore per pensare ad un use case diagram è che si tratta di un grafico dell'insieme dei casi d'uso che mostra gli attori, i casi d'uso e le relazioni tra essi.

	Actor
	Use-case
	Relationship
	System Boundary

Figure 7: Elementi di un use case diagram

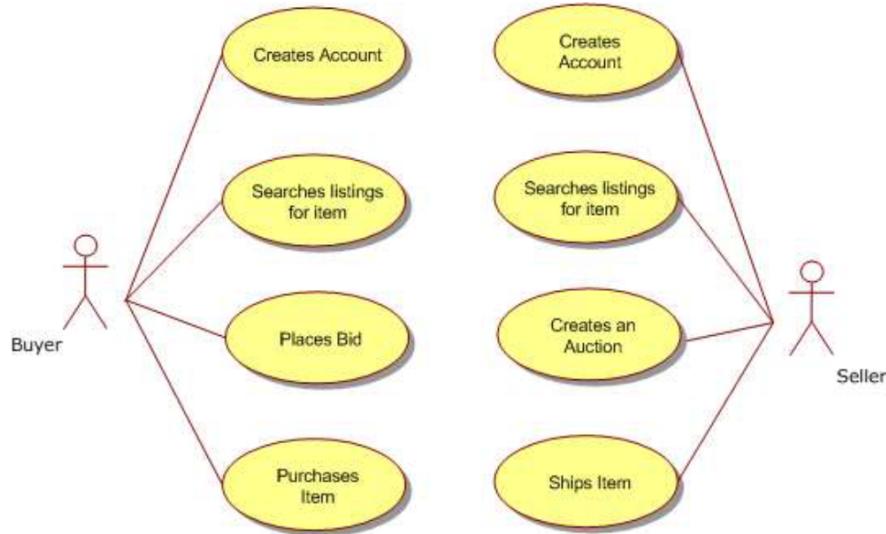


Figure 8: Esempio completo di use case diagram

Inoltre l'UML include relazioni tra i vari casi d'uso:

- **Ereditarietà**

Gli attori spesso hanno proprietà comuni e quindi alcuni casi d'uso possono essere utilizzati da diversi attori. Per esprimere questo concetto, gli attori possono essere rappresentati in una relazione di ereditarietà (generalizzazione) tra loro. Quando un attore Y (sotto-attore) eredita da un attore X (super-attore), Y è coinvolto in tutti i casi d'uso in cui X è coinvolto. In termini semplici, la generalizzazione esprime una relazione "is a".

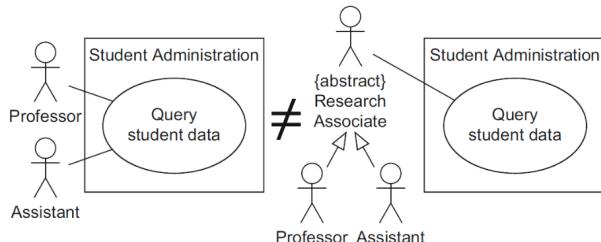


Figure 9: Nel primo caso professori ed assistenti interrogano insieme i dati degli studenti; nel secondo caso abbiamo aggregato i due attori nel super-attore Research che interroga i dati degli studenti. Se non esiste un'istanza di un attore, questo attore può essere etichettato con la parola chiave `abstract`, oppure l'etichetta deve essere scritta in corsivo. Notiamo che l'ereditarietà viene espressa con una freccia con la punta vuota.

- **Associazione Included**

Se un caso d'uso A include un caso d'uso B, il comportamento di B è integrato nel comportamento di A (da non confondere con i prerequisiti dei casi d'uso). L'uso di "include" è analogo alla chiamata di una subroutine in un linguaggio di programmazione procedurale. (A viene definito caso d'uso base e B caso d'uso incluso)

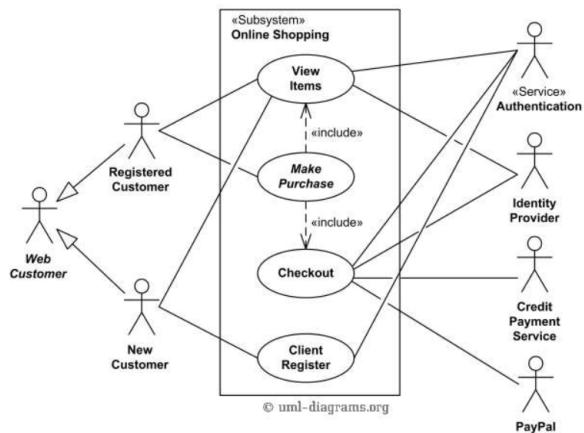


Figure 10: La clausola include viene indicata con una freccia tratteggiata e con label «include» che va dal caso d'uso base al caso d'uso incluso

- **Associazione Extended**

Se un caso d'uso B è in una relazione «extend» con un caso d'uso A, allora A può usare il comportamento di B ma non è obbligato (indica un caso particolare di A). Anche in questo caso, A è indicato come caso d'uso base e B come caso d'uso esteso. Entrambi i casi d'uso possono essere eseguiti anche indipendentemente l'uno dall'altro e un caso d'uso può fungere da caso d'uso esteso più volte (può essere esteso da più casi d'uso).

Inoltre una condizione che deve essere soddisfatta affinché il caso d'uso base inserisca il comportamento del caso d'uso che si estende, può essere specificata per ogni relazione «extend»; la condizione è specificata, tra parentesi graffe, in una nota collegata alla relazione "extend" corrispondente ed è indicata dalla parola chiave precedente Condition seguita da due punti. All'interno della condizione possono essere definiti anche dei punti di estensione, cioè punti in cui il comportamento dei casi d'uso estesi deve essere inserito nel caso d'uso di base.

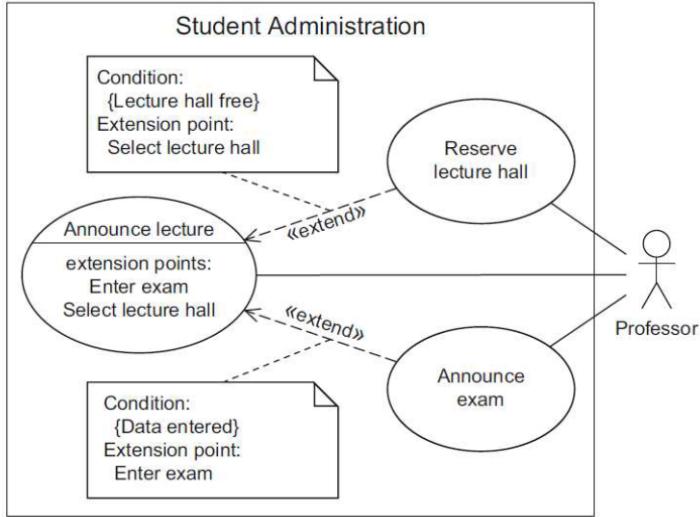
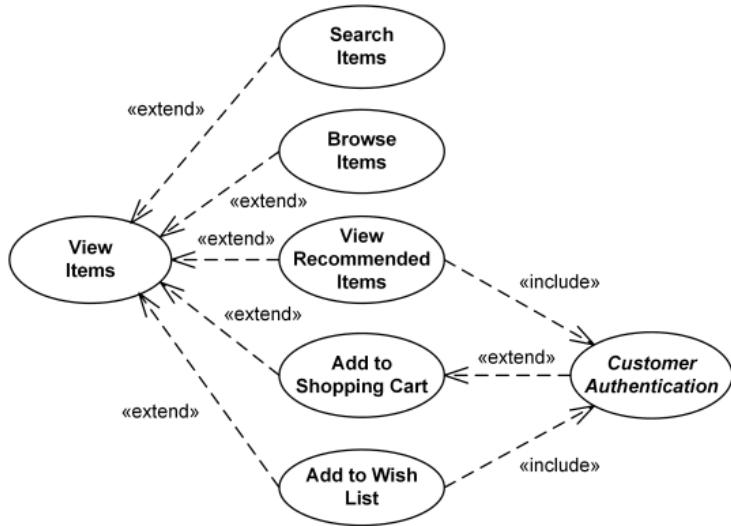


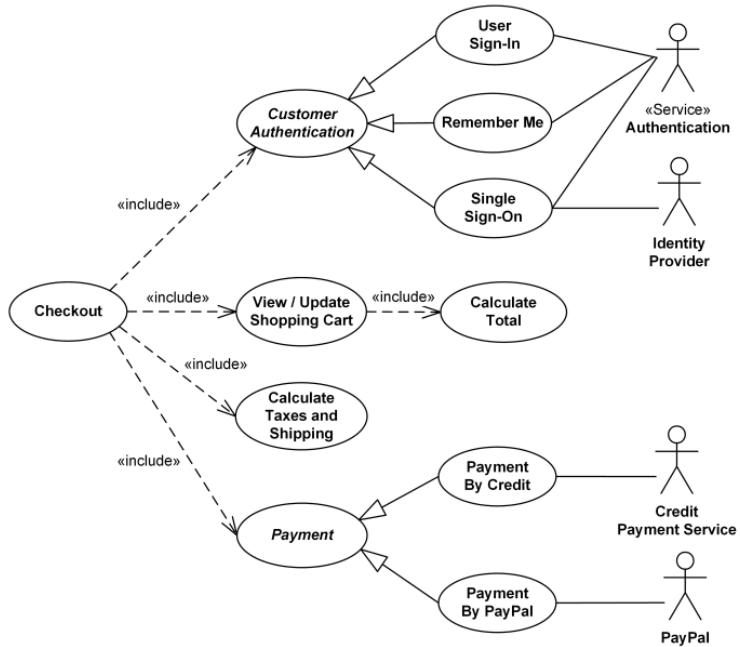
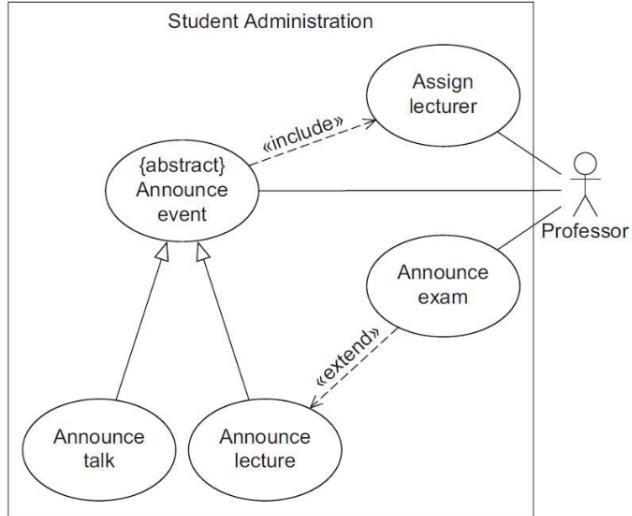
Figure 11: La clausola «extend» viene indicato con una freccia tratteggiata con label «extend» che va dal caso d'uso esteso al caso d'uso base. Un'estensione è una relazione tra l'estensione ed il caso esteso.

Vediamo un esempio completo di use case diagram



La generalizzazione è possibile anche tra i casi d'uso. Le proprietà comuni e i comportamenti comuni di diversi casi d'uso possono essere raggruppati in un caso d'uso padre. Se un caso d'uso A generalizza un caso d'uso B, quest'ultimo

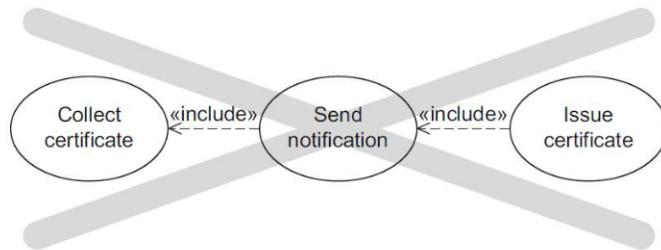
eredita il comportamento di A, che B può estendere o sovrascrivere. Inoltre, B eredita anche tutte le relazioni da A.



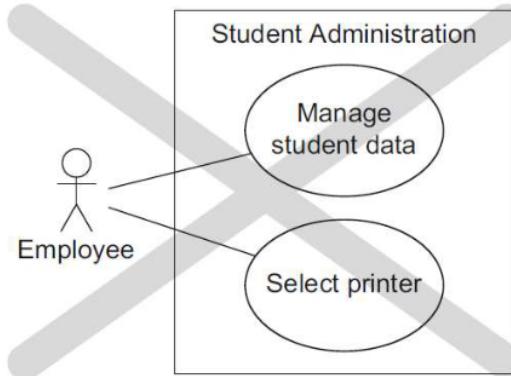
Per garantire che i diagrammi dei casi d'uso di grandi dimensioni rimangano chiari, è importante scegliere nomi concisi per i casi d'uso; quando ciò non è

possibile, è necessario descrivere i casi d'uso attraverso un nome, una breve descrizione, le precondizioni e le postcondizioni, situazione d'errore, triggere, ecc.. Come già detto, i casi d'uso sono uno strumento prezioso per aiutare a comprendere i requisiti funzionali di un sistema e inoltre rappresentano una visione esterna del sistema. Ma un grande pericolo dei casi d'uso è quello di renderli troppo complicati e di cadere in comuni errori di progettazione:

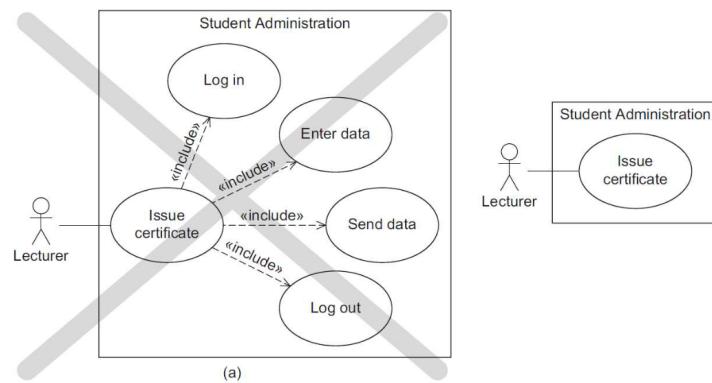
- Errori nel modellare il processo descritto dal linguaggio naturale: nella figura sottostante notiamo come attraverso una catena di include si cerca di descrivere una sequenza di attività che però non può essere descritta per definizione dallo use case diagram ma solo dall'activity diagram (Non esiste un ordine tra i casi d'uso, lo use case diagram non descrive una sequenza)



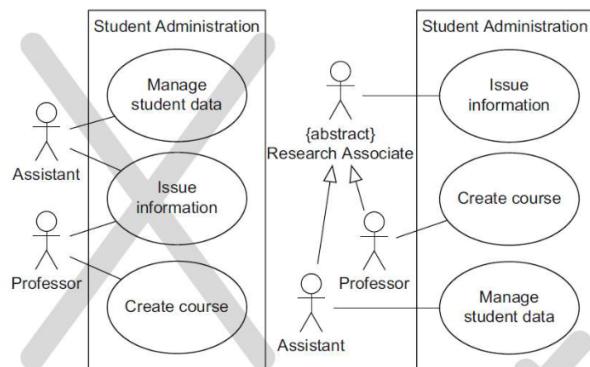
- Mescolare i livelli di astrazione



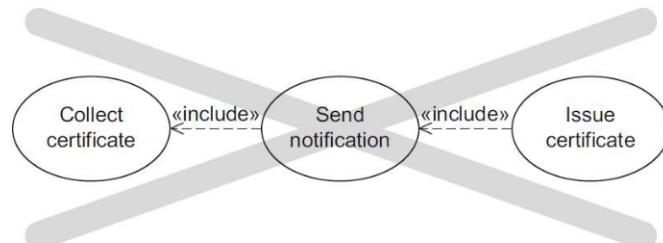
- Decomposizione funzionale (dobbiamo sempre semplificare e nascondere i dettagli)



- Associazioni errate (le associazioni sono solo tra attori ed use case, non esistono associazioni attori-attori o use case-use case)



- Modellare casi d'uso rindondanti



3 Progettazione

3.1 Design: Architecture and Methodology

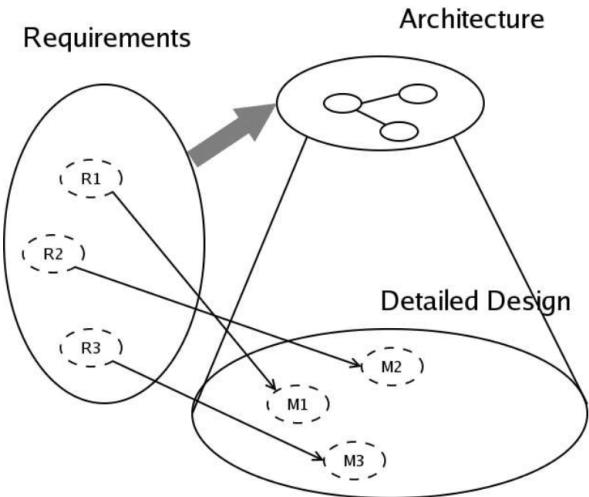
Qual è l'obiettivo di una buona progettazione del software? Sappiamo che se il software viene utilizzato inevitabilmente sarà modificato nel tempo; quindi un'applicazione è progettata in modo corretto se è facile gestirne il cambiamento(nuove feature, espansione del dominio,ecc.). L'obiettivo dell'architettura del software è quello di ridurre al minimo le risorse umane necessarie per costruire e mantenere il sistema e la misura della qualità della progettazione sarà semplicemente la misura dello sforzo necessario per soddisfare le nuove esigenze del cliente (effort basso = buon design; effort alto = bad design). Infatti gli sviluppatori di software hanno la responsabilità di garantire che comportamento e struttura del software rimangano elevati:

- Il comportamento, quindi le specifiche funzionali devono continuare a funzionare anche a seguito di cambiamenti
- L'architettura (struttura) deve essere progettata in modo che sia facilmente modificabile, altrimenti si rischia di aumentare notevolmente i costi di sviluppo

Quale tra il comportamento e l'architettura fornisce il valore maggiore? Per i responsabili aziendali e gli sviluppatori, è più importante che il sistema software funzioni ma un programma che funziona perfettamente ma che è impossibile da modificare, diventerà rapidamente inutile. Quindi bisogna cercare sempre un equilibrio tra comportamento e architettura.

Ora cominciamo a definire il design ed ad introdurre metodi per progettare un buon design. Una volta compresi i requisiti di un progetto, inizia la trasformazione dei requisiti in design (fase di progettazione). Ma che cosa si intende con progettazione di un software? La progettazione del software si occupa di come il software deve essere strutturato, cioè quali sono i componenti e come sono collegati tra loro. In particolar modo, per sistemi grandi questa fase è divisa in 2 parti:

- Fase di Architectural design, panoramica di alto livello (sono elencati i componenti principali, le proprietà esterne ai componenti e le relazioni tra i componenti). In questa fase i requisiti funzionali e non funzionali guidano la progettazione dell'architettura.
- Fase di Detailed design, i componenti vengono decomposti ad un livello di dettaglio maggiore



L'ideale è che il progetto sia creato e documentato fino al più basso livello di dettaglio possibile cosicché i programmatore possano tradurre il progetto facilmente in codice vero e proprio. Per documentare e specificare un design possiamo utilizzare una notazione grafica, in particolar modo per la progettazione orientata agli oggetti lo standard de facto è l'UML. Nella progettazione dobbiamo sempre mantenere alta la flessibilità/agilità dell'architettura, cioè la capacità di rispondere rapidamente ad un ambiente in costante e continua evoluzione (nuovi competitor, nuove tecnologie).

3.1.1 SOLID principles

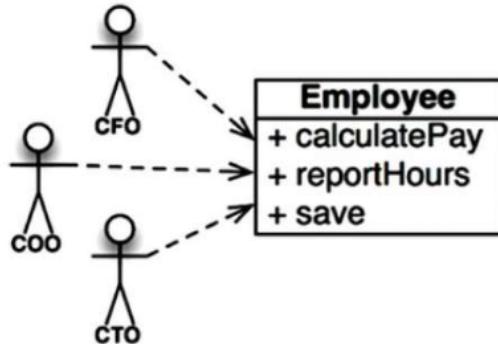
SOLID è un acronimo mnemonico per cinque principi di progettazione che hanno lo scopo di rendere i progetti orientati agli oggetti più comprensibili, flessibili e manutenibili. I principi sono un sottoinsieme di molti principi promossi dall'ingegnere informatico e istruttore americano Robert C. Martin.

- **S →The Single Responsibility Principle (SRP)**

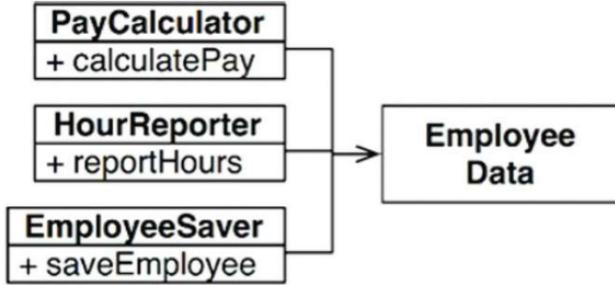
Una classe dovrebbe avere una, e una sola, ragione di cambiare. In particolar modo una classe deve avere singole responsabilità e quindi svolgere singole funzionalità. Che cos'è una responsabilità? È "un motivo per cambiare" e se si riesce a pensare a più di un motivo per cambiare una classe, allora quella classe ha più di una responsabilità. L'SRP è uno dei principi più semplici e uno dei più difficili da realizzare (poco utilizzato anche per l'enorme mole di classi che genera).

Ma perché è importante separare le responsabilità in classi distinte? Se una classe ha più di una responsabilità, allora le responsabilità diventano accoppiate e quindi le modifiche ad una responsabilità possono compromettere o inibire la capacità della classe di soddisfare le altre. La **coesione** è il termine usato per misurare quanto una classe o un modulo

supporta un singolo scopo o una singola responsabilità; classi che tendono ad avere un'elevata coesione sono più manutenibili rispetto alle classi che si assumono più responsabilità ed hanno una bassa coesione. Per chiarire questo concetto vediamo un esempio



Questa classe viola l'SRP perché questi tre metodi sono responsabili di tre attori diversi; infatti il metodo calculatePay() è specificato dal reparto contabilità(CFO), il metodo reportHours() è specificato e utilizzato dal reparto risorse umane(COO) ed il metodo save() è specificato dagli amministratori del database(CTO). Supponiamo che la funzione calculatePay() e la funzione reportHours() condividano un algoritmo comune per il calcolo delle ore non straordinarie e quindi gli sviluppatori, che non vogliono duplicare il codice, inseriscono l'algoritmo nella funzione regularHours(). Supponiamo ora che il team del CFO decida che il modo in cui vengono calcolate le ore non straordinarie deve essere modificato. Al contrario, il team HR del COO non vuole questa particolare modifica. Uno sviluppatore viene incaricato di apportare la modifica e vede la funzione regularHours() richiamata dal metodo calculatePay(), apporta la modifica richiesta e la testa con attenzione ma non si accorge che la funzione è richiamata anche dalla funzione reportHours(). Il team del CFO verifica che la nuova funzione funzioni come desiderato e il sistema viene distribuito. Naturalmente, il team del COO non sa che questo sta accadendo ed il personale delle risorse umane continua a utilizzare i rapporti generati dalla funzione reportHours(), che però ora contengono numeri errati. Per risolvere questo problema possiamo definire tre classi separate PayCalculator, HourReporter ed EmployeeSaver che contengono solo il codice sorgente necessario alla loro particolare funzione. In questo modo si evita qualsiasi duplicazione accidentale e gli sviluppatori hanno tre classi distinte da istanziare e monitorare.



Un'altra soluzione, molto comune è quella di utilizzare il Facade pattern(facciata), in cui definiamo una classe di facciata che mostri allo sviluppatore le varie funzioni di calcolo senza fornire dettagli sull'implementazione.



```

public class PayCalculator{
    EmployeeData ed;

    public void calculatePay(ed){
        ...
    }
}

public class EmployeeFacade{
    PayCalculator pc;
    HourReporter hr;
    EmployeeSaver es;

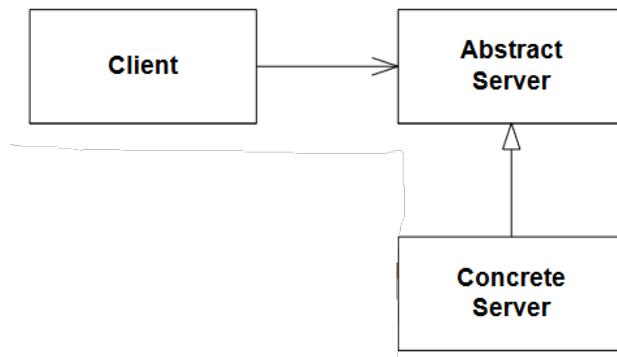
    public void calculatePay(){
        pc.calculatePay();
    }
}

```

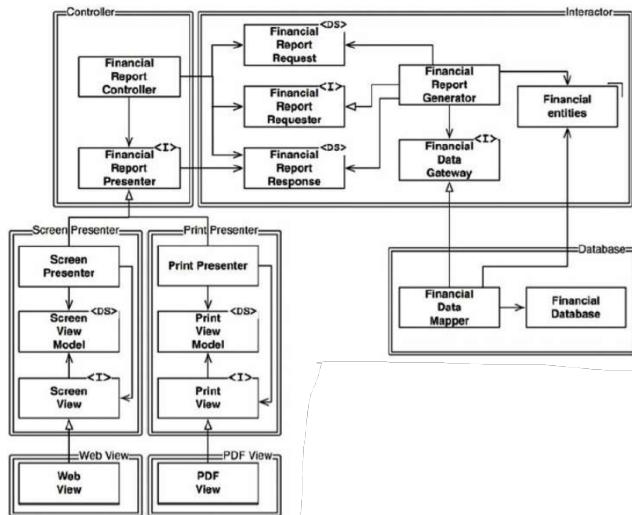
Alcuni sviluppatori preferiscono mantenere le regole di business più importanti vicine ai dati. Questo può essere fatto mantenendo il metodo più importante nella classe originale classe Employee e poi utilizzando tale classe come facciata per le funzioni meno importanti.

- O →**The Open Closed Principle**

Dovreste essere in grado di estendere il comportamento di una classe, senza modificarla. Quando una singola modifica ad un programma provoca una cascata di modifiche a moduli dipendenti, il programma diventa fragile, rigido, imprevedibile e non riutilizzabile. Questo principio ci dice che si dovrebbero progettare moduli che non cambiano mai. Quando i requisiti cambiano, si estende il comportamento di questi moduli aggiungendo nuovo codice, non modificando il vecchio codice che già funziona. Le classi/moduli(file sorgente) dovrebbero essere **Open For Extension**, cioè il comportamento del modulo può essere esteso e **Closed for Modification**, cioè non possiamo modificare il comportamento dei metodi già definiti e funzionanti. Il metodo più utilizzato per estendere il comportamento di un modulo è quello di utilizzare l'astrazione; con l'utilizzo delle classi astratte/interfacce possiamo bloccare il nome di metodi già esistenti e estendere le funzionalità con l'aggiunta di nuovi metodi.



Ovviamente nessun programma può essere chiuso al 100% e quindi la chiusura deve essere strategica, cioè il progettista deve scegliere i tipi di cambiamenti rispetto ai quali chiudere il proprio progetto; ciò richiede una certa dose di preveggenza derivante dall'esperienza. Vediamo un esempio per chiarire il concetto: Abbiamo un sistema che visualizza un riepilogo finanziario su una pagina web e gli stakeholder chiedono che le stesse informazioni possano essere stampate su una stampante in bianco e nero (con cambi nella visualizzazione dei dati). Per progettare in modo efficiente questo sistema dovremmo dividere la generazione del report in due responsabilità distinte: calcolo dei dati riportati e presentazione di tali dati. Dopo aver effettuato questa separazione, dobbiamo organizzare le dipendenze del codice per assicurare che le modifiche a una di queste responsabilità non causino cambiamenti nell'altra. La struttura da generare sarà simile alla seguente:



Tutte le frecce rappresentano le dipendenze dal codice sorgente. Una freccia che punta dalla classe A alla classe B significa che il codice sorgente della classe A menziona il nome della classe B, ma la classe B non menziona nulla della classe A (FinancialReportController conosce FinancialReportPresenter ma non vale il viceversa). Inoltre ogni doppia linea è attraversata in una direzione, ciò significa che tutte le relazioni tra i componenti sono unidirezionali; queste frecce puntano verso i componenti che si vogliono proteggere dal cambiamento. Si noti come questo crea una gerarchia di protezione basata sulla nozione di "livello", in questo modo i componenti di livello superiore nella gerarchia sono protetti dalle modifiche apportate ai componenti di livello inferiore. Il principio di Open Closed è la motivazione principale di molte delle convenzioni che sono state proposte nell'OOP:

- Information hiding
- Rendere private tutte le variabili interne alla classe
- Nessuna variabile globale
- L'identificazione del tipo a tempo di esecuzione è pericolosa.

• L →The Liskov Substitution Principle

Formalmente il principio definisce che gli oggetti di una superclasse devono essere sostituibili con oggetti delle sue sottoclassi senza interrompere l'applicazione. Ciò richiede che gli oggetti delle sottoclassi si comportino allo stesso modo degli oggetti della superclasse (non bisogna portare il contenuto della superclasse nella sottoclasse). Per esempio un metodo sovrascritto di una sottoclasse deve accettare gli stessi valori dei parametri di input come il metodo della superclasse o il valore di ritorno di un metodo della sottoclasse deve essere conforme alle regole del valore di ritorno del

metodo della superclasse. Per chiarire questo principio riporto questo esempio preso da Stack Overflow

```
public class Bird{  
    public void fly(){}
}  
public class Duck extends Bird{}  
public class Ostrich extends Bird{}
```

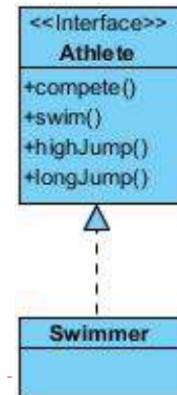
L'anatra può volare perché è un uccello; lo struzzo è un uccello, ma non può volare; la classe struzzo è un sottotipo della classe uccello, ma non dovrebbe essere in grado di usare il metodo fly, il che significa che stiamo violando il principio LSP.

```
public class Bird{}  
public class FlyingBirds extends Bird{  
    public void fly(){}
}  
public class Duck extends FlyingBirds{}  
public class Ostrich extends Bird{}
```

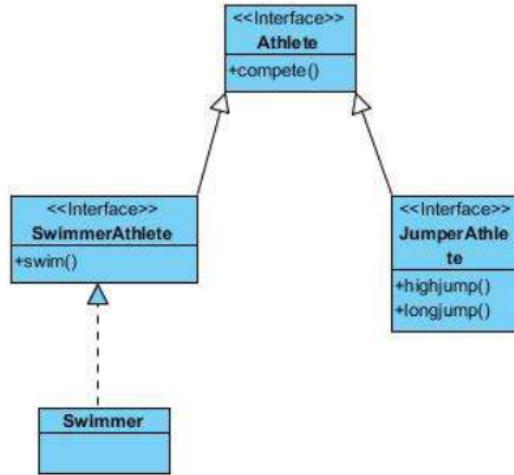
Questo ci porta a una conclusione molto importante, cioè che un modello, visto isolatamente, non può essere validato in modo significativo perché deve essere valutato solo dai clienti, cioè quando viene utilizzato.

- I →The Interface Segregation Principle

I client non devono essere costretti a dipendere da interfacce che non utilizzano. L'obiettivo del principio è quello di ridurre gli effetti collaterali e la frequenza delle modifiche necessarie, suddividendo il software in parti multiple e indipendenti. In pratica un oggetto deve avere il minimo possibile di funzioni necessarie (solo metodi correlati). Vediamo subito un esempio.



Il nuotatore deve implementare tutti i metodi anche se alcuni di essi non hanno senso per un nuotatore. Invece di avere un'unica grande interfaccia, dovremmo implementare interfacce più piccole e specifiche.

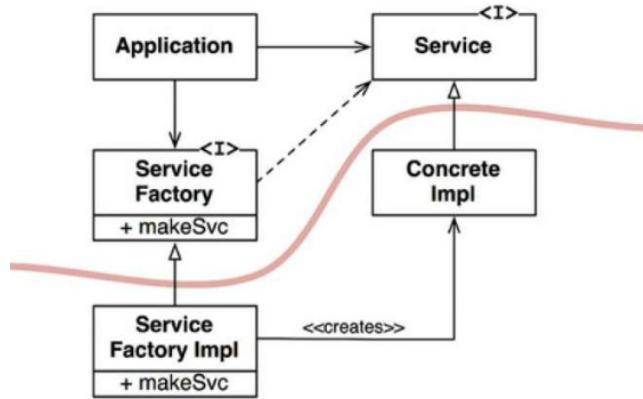


- D → **The Dependency Inversion Principle**

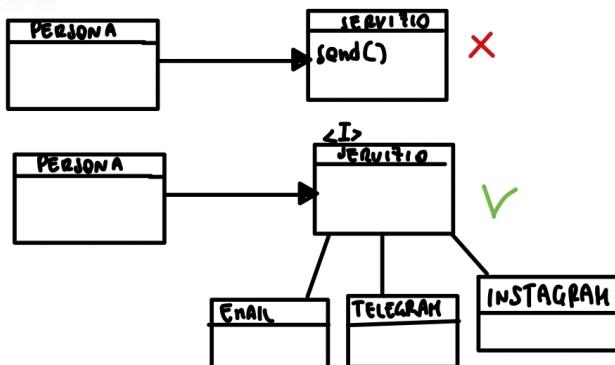
I moduli di alto livello, che forniscono una logica complessa, dovrebbero essere facilmente riutilizzabili e non influenzati dalle modifiche apportate ai moduli di basso livello, che forniscono funzionalità di utilità (entrambi devono dipendere dall'astrazione). Ogni modifica ad un'interfaccia astratta corrisponde ad una modifica alle sue implementazioni concrete ma, al contrario, le modifiche alle implementazioni concrete non devono richiedere modifiche alle interfacce che implementano. Questa implicazione si riduce a un insieme di pratiche di codifica molto specifiche:

- Non fare riferimento a classi concrete volatili
- Non derivare da classi concrete volatili; nei linguaggi tipizzati staticamente, l'ereditarietà è la più forte e la più rigida di tutte le relazioni del codice sorgente; di conseguenza, deve essere usata con grande attenzione (crea una notevole dipendenza).
- Non sovrascrivere le funzioni concrete(implementate). Quando si sovrascrivono tali funzioni, non si eliminano le dipendenze; per gestire queste dipendenze, si dovrebbe rendere astratta la funzione e creare più implementazioni.

Per rispettare queste regole, la creazione di oggetti concreti volatili richiede una gestione particolare; per gestire questa dipendenza indesiderata, si può utilizzare un Abstract Factory.



La linea curva è un confine architettonico che separa l'astratto dal concreto. Tutte le dipendenze del codice sorgente che attraversano questa linea curva puntano nella stessa direzione, verso il lato astratto (così da isolare il cambiamento).



Il concetto generale che sta alla base della dependency injection si chiama Inversion of Control. Una classe non dovrebbe configurare le sue dipendenze staticamente, ma dovrebbe essere configurata dall'esterno. Idealmente le classi Java dovrebbero essere il più possibile indipendenti da altre classi Java. Questo aumenta la possibilità di riutilizzare le classi e di poterle testare in modo indipendente da altre classi.

```

public class MyClass {
    private Logger logger;
    public MyClass(Logger logger) {
        this.logger = logger;
        // write an info log message
        logger.info("This is a log message.")
    }
}

```

```
    }  
}
```

In questo caso la classe MyClass non conosce nulla sulla classe Logger; infatti MyClass eseguirà delle operazioni in base al tipo di Logger passato (non è MyClass che decide che operazioni eseguire ma il Logger → Inversion of Control). Ci sono almeno tre modi in cui un oggetto può ricevere un riferimento a un modulo esterno:

1. constructor injection, le dipendenze sono fornite tramite un costruttore di classe.
2. setter injection, il client espone un metodo setter che l'iniettore usa per iniettare la dipendenza.
3. interface injection, la dipendenza fornisce un metodo iniettore che inietterà la dipendenza in qualsiasi client gli venga passato; i client devono implementare un'interfaccia con un metodo setter che accetti la dipendenza.

3.1.2 Design Components

Un insieme di classi formano i moduli/componenti che sono le unità di distribuzione, cioè le entità più piccole che possono essere distribuite come parte di un sistema. Come possiamo organizzare questi componenti? I componenti possono essere collegati tra loro in un singolo eseguibile, aggregati in un singolo archivio,(file .war) o distribuiti come plugin separati caricati dinamicamente, come file .jar o .dll o .exe.

Quali classi appartengono a quali componenti? Tre principi di coesione dei componenti possono supportare la vostra scelta:

1. REP: The Reuse/Release Equivalence Principle

È un principio che sembra ovvio. Chi vuole riutilizzare i componenti del software non può farlo a meno che questi componenti non siano tracciati attraverso un processo di rilascio e siano dotati di numeri di rilascio (numero di versione); senza numeri di rilascio, non c'è modo di garantire che i componenti siano compatibili tra loro e non c'è modo di distinguere le vecchie versioni del software dalle nuove. Inoltre bisogna raggruppare le classi nello stesso componente in modo da dare un senso sia allo sviluppatore e sia agli utenti("dare un senso" è una regola molto vaga perché dipende da persona a persona).

2. CCP: The Common Closure Principle

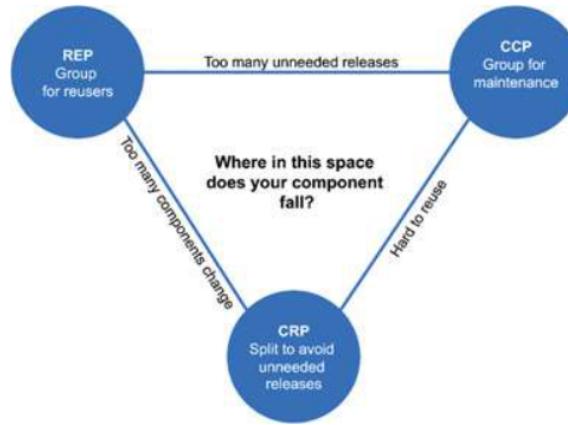
Riunire in componenti le classi che cambiano per le stesse ragioni e negli stessi momenti. Questo è il Principio di Single Responsibility riadattato per i componenti, cioè una classe non dovrebbe contenere molteplici motivi per cambiare, quindi il CCP dice che un componente non dovrebbe avere più motivi per cambiare. Questo principio è strettamente associato

al principio di Open-Closed (OCP). Poiché la chiusura al 100% non è raggiungibile, deve essere strategica, cioè progettiamo le nostre classi in modo che siano chiuse ai più comuni tipi di cambiamenti che ci aspettiamo o che abbiamo sperimentato.

3. CRP: The Common Reuse Principle

Non costringere gli utenti di un componente a dipendere da cose di cui non hanno bisogno. Il CRP afferma che le classi ed i moduli che tendono ad essere riutilizzati insieme devono appartenere allo stesso componente. Inoltre ci dice anche quali classi non tenere insieme in un componente; quando un componente ne usa un altro, si crea una dipendenza tra i componenti (anche se usa una sola classe) e quindi a causa della dipendenza, quando il componente usato viene modificato, è probabile che il componente che lo utilizza debba essere modificato (di conseguenza ricompilato, riconvalidato e distribuito nuovamente).

REP e CCP sono principi inclusivi, cioè entrambi tendono a rendere i componenti più grandi; invece il CRP è un principio esclusivo, che spinge i componenti a essere più piccoli. È la tensione tra questi principi che i bravi architetti cercano di risolvere ma bilanciare queste forze non è banale.

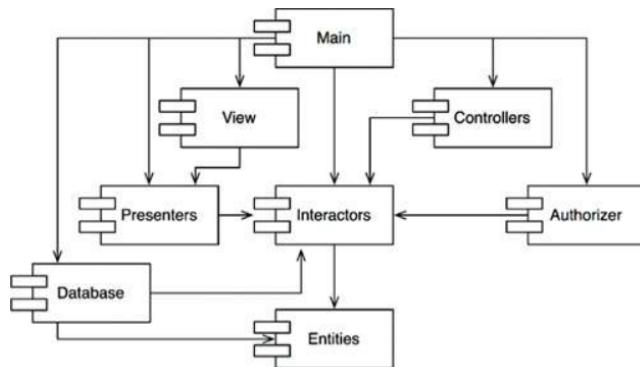


Come possiamo gestire le relazioni tra i componenti? Si possono introdurre tre principi:

1. ADP: Acyclic Dependencies Principle

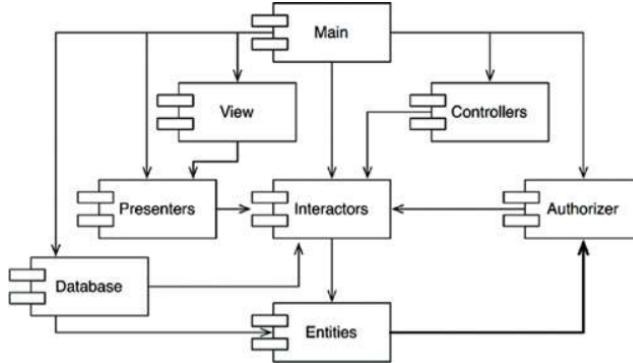
Non consente cicli nel grafico delle dipendenze dei componenti. In ambienti di sviluppo in cui molti sviluppatori modificano gli stessi file sorgenti, capita che qualcuno modifichi qualcosa da cui dipendete e il vostro lavoro improvvisamente non funziona più. La soluzione più semplice a questo problema è la "weekly build", comune nei progetti di medie dimensioni, dove tutti gli sviluppatori si ignorano per i primi quattro giorni della settimana, lavorando tutti su copie private del codice e non si preoccupano

di integrare il loro lavoro su base collettiva. Alla fine delle settimana (venerdì) integrano tutte le loro modifiche e costruiscono il sistema. Questo approccio consente agli sviluppatori di lavorare in isolamento per quattro giorni su cinque ma lo svantaggio, ovviamente, è la grande mole di lavoro durante l'integrazione del venerdì (Non adatto a problemi di grande dimensione). La soluzione a questo problema è la suddivisione dell'ambiente di sviluppo in componenti rilasciabili, cioè le componenti diventano unità di lavoro che possono essere di competenza di un singolo sviluppatore o di un team di sviluppatori. Quando gli sviluppatori riescono a far funzionare un componente, lo rilasciano per essere utilizzato dagli altri sviluppatori assegnandoli un numero di release e spostandolo in una directory a disposizione degli altri team e continuano a modificare il loro componente nelle loro aree private. Quando vengono rese disponibili le nuove versioni di un componente, gli altri team possono decidere se adottare immediatamente la nuova versione o continuare ad usare la vecchia. In questo modo nessuna squadra è in balia delle altre ma per far sì che funzioni con successo è necessario gestire la struttura delle dipendenze dei componenti ed evitare la creazione di cicli. Vediamo un esempio



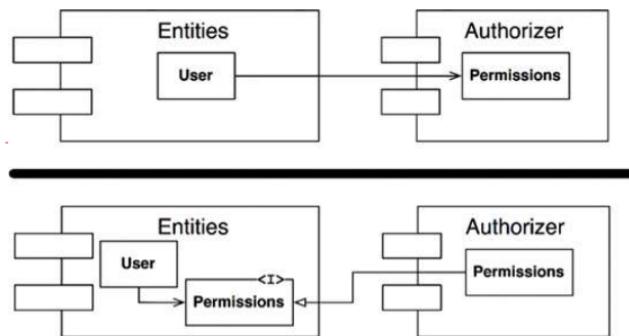
Come notiamo questa struttura non ha cicli è infatti un grafo aciclico diretto(DAG). Quando il team responsabile dei Presenters rilascia una nuova versione del componente sia View e che Main saranno interessati dalla modifica e gli sviluppatori che stanno lavorando a questi componenti dovranno decidere quando integrare il loro lavoro con la nuova versione di Presenters. Quando è il momento di rilasciare l'intero sistema, il processo procede dal basso verso l'alto, prima viene compilato, testato e rilasciato il componente Entità, poi si fa lo stesso per Database e Interattori, e così via. Questo processo è molto chiaro e facile da gestire.

Vediamo ora un esempio con un ciclo per capirne l'effetto e vedere le possibili soluzioni.

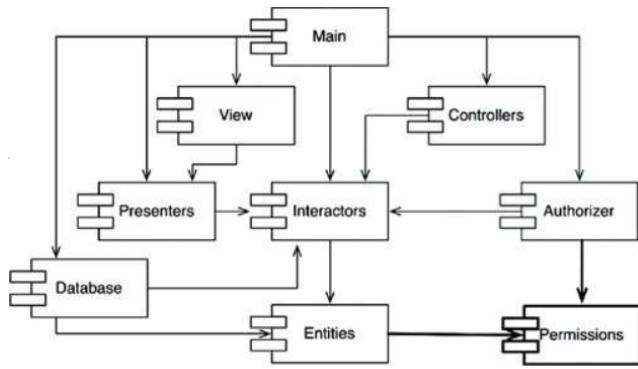


Questo ciclo crea alcuni problemi immediati. Per esempio, gli sviluppatori che lavorano sul componente Database sanno che il componente deve essere compatibile con Entities ma il componente Database deve essere compatibile anche con Authorizer, con Authorizer che dipende da Interactors. Questo rende Database molto più difficile da rilasciare. Inoltre quando vogliamo testare il componente Entities, dobbiamo costruire e integrare sia Authorizer che Interactors. La soluzione principale è quella di rompere il ciclo e per ciò possiamo utilizzare 2 metodi:

- (a) Applicare il principio di Dependency Inversion (DIP). Potremmo creare un'interfaccia che abbia i metodi di cui User ha bisogno ed inserire tale interfaccia in Entità ed ereditarla in Authorizer. In questo modo si inverte la dipendenza tra Entità e Authorizer, interrompendo così il ciclo.



- (b) Creare un nuovo componente da cui dipendono sia Entità che Authorizer



2. SDP: Stable Dependencies Principle

La maggior parte dei progetti non sono completamente statici, infatti alcuni di questi componenti sono progettati per essere volatili, cioè ci aspettiamo che cambino. Inoltre non tutti i pacchetti del nostro software sono uguali, un piccolo cambiamento ad un pacchetto può generare dei cambiamenti a cascata su altri pacchetti. Il SPD ci dice che ogni pacchetto deve dipendere da un pacchetto più stabile di lui, cioè meno soggetto a cambiamenti. Cosa si intende per "stabilità"? La stabilità è legata alla quantità di lavoro necessaria per apportare un cambiamento.

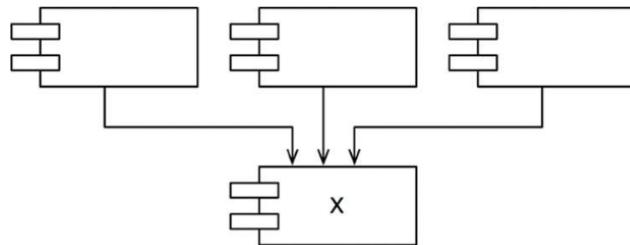


Figure 12: X è un componente stabile

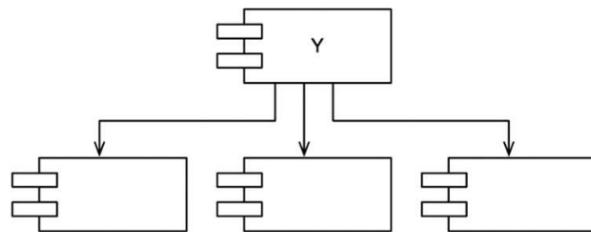


Figure 13: Y è un componente instabile

Come possiamo misurare la stabilità di un componente? Un modo è contare il numero di dipendenze che entrano ed escono dal componente, infatti questi conteggi ci permettono di calcolare la stabilità posizionale del componente. Definiamo le seguenti metriche:

- Fan-in: dipendenze in entrata.
- Fan-out: Dipendenze in uscita.
- Instability: $I = \text{Fan-out} / (\text{Fan-in} + \text{Fan-out})$. Questa metrica ha un intervallo $[0,1]$ con $I = 0$ che indica una componente stabile al massimo e $I = 1$ che indica un componente instabile al massimo.

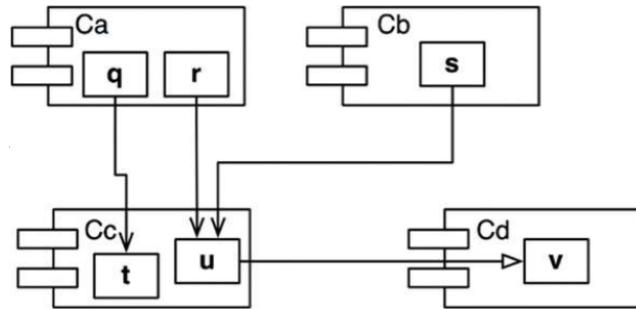
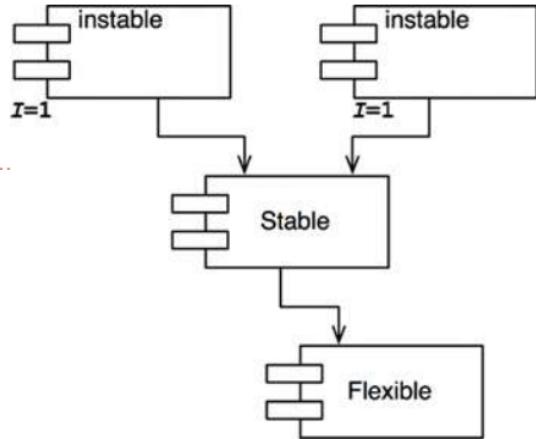


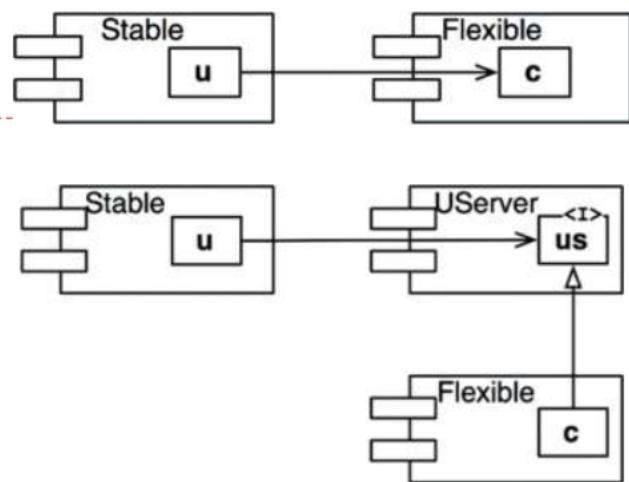
Figure 14: Vogliamo calcolare la stabilità del componente C_c : Fan-in = 3, Fan-out = 1 e $I = 1/4$

Quando la metrica I è uguale a 1 nessun altro componente dipende da questo componente ($\text{Fan-in} = 0$), e questo componente dipende da altri componenti ($\text{Fan-out} > 0$). Questa situazione è quanto di più instabile possa esistere per un componente: è irresponsabile e dipendente. Al contrario, quando la metrica I è uguale a 0, significa che altri componenti dipendono da questo componente ($\text{Fan-in} > 0$), ma non dipende da altri componenti ($\text{Fan-out} = 0$). Un tale componente è responsabile e indipendente ed è il più stabile possibile. L'SDP dice che la metrica di instabilità di un componente deve essere più grande della metrica di instabilità dei componenti da cui dipende. Cioè, le metriche devono diminuire nella direzione della dipendenza.

Se tutti i componenti di un sistema fossero massimamente stabili, il sistema sarebbe immutabile (situazione non auspicabile). Quindi dobbiamo progettare la nostra struttura di componenti in modo che alcuni siano instabili e altri stabili, così da permettere il cambiamento. Vediamo subito un esempio



Flexible è un componente che abbiamo progettato per essere facile da modificare (instabile). Questo viola l'SDP perché la metrica I per Stable è molto più piccola della metrica I di Flexible. Di conseguenza, Flexible non sarà più facile da modificare ed una modifica a Flexible ci costringerà ad occuparci di Stable e di tutte le sue dipendenze. Possiamo risolvere questo problema utilizzando il DIP. Creiamo una classe di interfaccia chiamata US e la inseriamo in un componente chiamato UServer che deve dichiarare tutti i metodi che U deve utilizzare. Questo interrompe la dipendenza di Stable da Flexible e costringe entrambi i componenti a dipendere da UServer ($I = 0$). Tutte le dipendenze ora fluiscano nella direzione della diminuzione di I



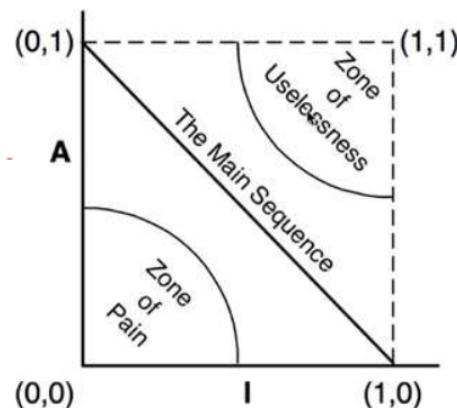
3. SAP: Stable Abstraction Principle

Un componente deve essere tanto astratto quanto stabile. Alcune parti del sistema non dovrebbero cambiare molto spesso (architettura di alto livello); pertanto, queste parti dovrebbero essere collocate in componenti stabili ($I = 0$). Invece le componenti instabili ($I = 1$) dovrebbero contenere solo il software volatile.

Il principio stabilisce una relazione tra stabilità e astrattezza. Un componente stabile deve essere anche astratto, in modo che la sua stabilità non ne impedisca l'estensione. Un componente instabile dovrebbe essere concreto, poiché la sua instabilità consente di modificare facilmente il codice concreto al suo interno. Come possiamo misurare l'astrazione di un componente? Possiamo introdurre le seguenti metriche:

- N_c : Il numero di classi del componente.
- N_a : Il numero di classi astratte e di interfacce presenti nel componente
- A : Astrattezza. $A = N_a \div N_c$. La metrica A varia da 0 a 1. Un valore di 0 implica che il componente non ha classi astratte. Un valore di 1 implica che il componente non contiene altro che classi astratte.

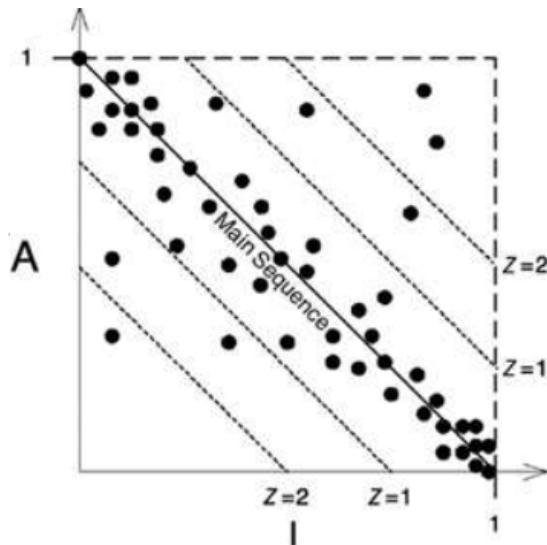
Vediamo la relazione che c'è tra stabilità ed astrazione



Nella Zone of Pain un componente è altamente stabile e concreto ma non può essere esteso perché non è astratto, ed è molto difficile da cambiare a causa della sua stabilità. Nella Zona of Uselessness un componente è astratto al massimo, ma non ha dipendenze, è quindi inutile. Sembra chiaro che i nostri componenti più volatili dovrebbero essere tenuti il più possibile lontani da entrambe le zone di esclusione. The Main Sequence è la massima distanza da ciascuna zona ed è la posizione più auspicabile per un componente.

$|A+I-1|$ (con intervallo $[0, 1]$) in cui il valore di 0 indica che il componente si trova direttamente nella Sequenza principale ed un valore di 1 indica che il componente è il più lontano possibile dalla Sequenza Principale. Data questa metrica, un progetto può essere analizzato per la sua conformità complessiva alla Sequenza Principale.

È inoltre possibile effettuare l'analisi statistica di un design, calcolando la media e la varianza di tutte le metriche D per i componenti di un progetto. Ci si aspetta che un progetto conforme abbia una media e una varianza prossime allo zero. La varianza può essere utilizzata per stabilire "limiti di controllo" in modo da identificare i componenti che sono "eccezionali" rispetto a tutti gli altri



3.1.3 Software Architecture

L'architettura software di un programma è la struttura o le strutture del sistema. Comprende gli elementi software, le proprietà visibili all'esterno di tali elementi e le relazioni tra di essi. Una vista è una rappresentazione della struttura di un sistema; Kruchten (1995) ha proposto di avere quattro punti di vista architettonici:

- Vista logica: Rappresenta la decomposizione orientata agli oggetti di un sistema, cioè le classi e le relazioni tra di esse.
- Vista di processo: Rappresenta i componenti di run-time (processi) e il modo in cui comunicano tra loro.
- Vista di decomposizione del sottosistema: Rappresenta i moduli e i sottosistemi, uniti con relazioni di esportazione e importazione.
- Vista dell'architettura fisica: Rappresenta la mappatura del software all'hardware.

Invece Bass, Clements e Kazman (2003) classificano i punti di vista in tre categorie:

- Viste di modulo: Rappresentano la gerarchia di moduli e sottomoduli, come i moduli dipendono l'uno dall'altro e la gerarchia di ereditarietà delle classi.
- Viste in tempo di esecuzione (viste dei componenti e dei connettori): la struttura del programma in esecuzione.
- Viste di allocazione: la mappatura dei moduli software ad altri sistemi

Un punto importante è che i diversi punti di vista sono utili a diversi stakeholder. L'architettura di un software è la forma data a quel sistema da chi lo costruisce (gli architetti del software sono di solito i migliori programmatore, guidano tutto il team di sviluppo); lo scopo di tale forma è quello di facilitare lo sviluppo, la distribuzione, il funzionamento e la manutenzione del software.

- **Sviluppo**

Un software difficile da sviluppare non avrà probabilmente una vita lunga e sana.

- **Distribuzione**

Per essere efficace, un software deve essere distribuibile. Più alto è il costo di distribuzione, meno utile è il sistema.

- **Funzionamento**

L'impatto dell'architettura sul funzionamento del sistema tende a essere meno drammatico infatti quasi tutte le difficoltà operative possono essere risolte con l'aggiunta di hardware (senza incidere sull'architettura del software). L'architettura del sistema rende evidente agli sviluppatori il funzionamento del sistema.

- **Mantenimento e Manutenzione**

L'incessante sfilata di nuove funzionalità e la scia di difetti e correzioni consumano grandi quantità di risorse umane. Il costo principale della manutenzione è rappresentato dalla ricerca(Spelunking = è il costo di scavare nel software, cercando di determinare il luogo e la strategia migliori per aggiungere una nuova funzionalità o riparare un difetto) e dal rischio. L'architettura dovrebbe ridurre questi costi.

L'obiettivo principale dell'architettura del software non è quello di far funzionare il sistema correttamente: è quello di supportare il ciclo di vita del sistema. Infatti una buona architettura rende il sistema facile da capire, facile da sviluppare, facile da mantenere e facile da distribuire. L'obiettivo finale è minimizzare il costo del ciclo di vita del sistema e massimizzare la produttività dei programmatore. La struttura mantiene il software "soft" e per far ciò bisogna lasciare aperte il maggior numero possibile di opzioni (i dettagli che non contano), per tutto il tempo possibile.

Tutti i sistemi software possono essere scomposti in due elementi principali: la policy e i dettagli.

- L'elemento della policy racchiude tutte le regole e le procedure aziendali
- I dettagli sono gli elementi necessari per consentire agli esseri umani, agli altri sistemi e ai programmati di comunicare con la policy, ma che non influiscono sul comportamento del software (Database, Server, Framework).

L'obiettivo dell'architetto è creare una forma per il sistema che riconosca la policy come l'elemento più essenziale del sistema e che renda i dettagli irrilevanti per tale policy. Ad esempio, non è necessario scegliere un sistema di database nei primi giorni di sviluppo. Infatti, se lo sviluppatore è attento, la politica di alto livello non si preoccupa se il database è relazionale, distribuito, noSQL o semplicemente un file piatto. Se si può sviluppare la policy di alto livello senza impegnarsi sui dettagli che la circondano, si possono ritardare e rimandare le decisioni su quei dettagli per molto tempo.

Una buona architettura deve supportare:

- I casi d'uso del sistema. L'intento del sistema deve essere visibile a livello architettonico.
- Il funzionamento del sistema. Se il sistema deve gestire 100.000 clienti al secondo, l'architettura deve supportare quel tipo di throughput e di tempo di risposta per ogni caso d'uso che lo richiede.
- Lo sviluppo del sistema. Un sistema sviluppato da un'organizzazione con molti team deve avere un'architettura che faciliti le azioni indipendenti da parte di questi team, in modo che non interferiscano l'uno con l'altro durante lo sviluppo.
- La distribuzione del sistema. L'obiettivo è "l'impiego immediato" (rilasciare il software funzionante al più presto)

Una buona architettura bilancia tutte queste preoccupazioni con una struttura dei componenti che li soddisfa tutti. In realtà raggiungere questo equilibrio è piuttosto difficile, il problema è che il più delle volte non sappiamo quali siano tutti i casi d'uso, né conosciamo i vincoli operativi, la struttura del team o i requisiti di distribuzione.

Il sistema deve essere suddiviso in livelli orizzontali disaccoppiati: l'interfaccia utente, regole di business specifiche dell'applicazione, regole di business indipendenti dall'applicazione e il database (dettagli tecnici). Mentre dividiamo il sistema in strati orizzontali, dividiamo anche il sistema in sottili casi d'uso verticali che attraversano questi strati (se si disaccoppiano gli elementi del sistema che cambiano per motivi diversi, si può continuare ad aggiungere nuovi casi d'uso senza interferire con quelli vecchi). Inoltre se i casi d'uso sono separati, quelli che devono essere eseguiti ad alta velocità sono probabilmente già separati da quelli che devono essere eseguiti a bassa velocità; questo disaccoppiamento aiuta nelle operazioni. Per sfruttare il vantaggio operativo, il disaccoppiamento deve avere

una modalità appropriata (devono essere servizi indipendenti, che comunicano su una rete di qualche tipo). Per suddividere il sistema in livelli indipendenti bisogna garantire:

- INDEPENDENT DEVELOP-ABILITY: Quando i componenti sono fortemente disaccoppiati, l'interferenza tra team viene attenuata. Se i casi d'uso sono disaccoppiati l'uno dall'altro, allora un team che si concentra sul caso d'uso di addOrder non interferisce con un team che si concentra sul caso d'uso deleteOrder.
- INDEPENDENT DEPLOYABILITY: L'aggiunta di un nuovo caso d'uso può essere semplice come l'aggiunta di alcuni nuovi file jar o servizi al sistema, lasciando il resto inalterato.

Sia durante la divisione in livelli verticali che durante la divisione in livelli orizzontali dobbiamo stare attenti alla duplicazione di use case e di strutture dati. Vediamo ora qualche modalità di disaccoppiamento di livelli e casi d'uso:

- **Source level**

Possiamo controllare le dipendenze tra i moduli del codice sorgente in modo che le modifiche ad un modulo non costringano a modificare o ricompilare gli altri.

- **Deployment level**

Possiamo controllare le dipendenze tra le unità distribuibili, come i file jar, le DLL o le librerie condivise, in modo tale che le modifiche al codice sorgente di un modulo non costringano gli altri a essere ricostruiti e distribuiti nuovamente.

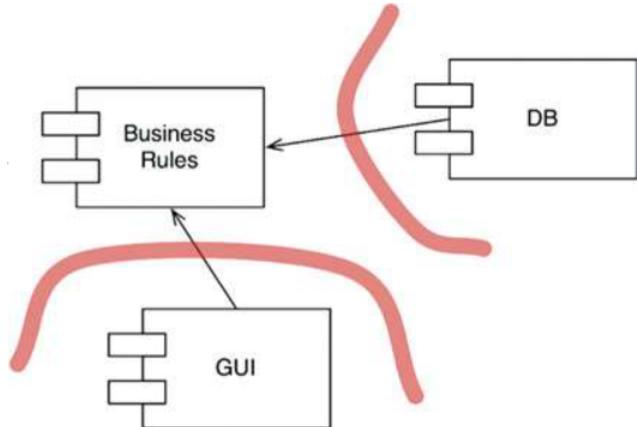
- **Service level**

Un servizio è un processo, in genere avviato dalla riga di comando o attraverso una chiamata di sistema equivalente. In questo disaccoppiamento riduciamo le dipendenze al livello delle strutture dati e comunichiamo esclusivamente tramite pacchetti di rete, in modo che ogni unità di esecuzione sia completamente indipendente dalle modifiche alla sorgente e ai binari di altre (ad esempio, servizi o microservizi).

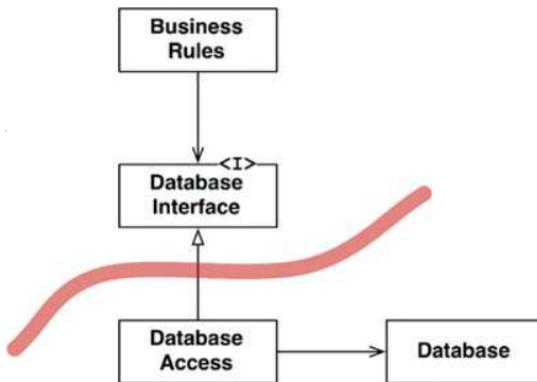
È difficile sapere quale sia la modalità migliore durante le prime fasi di un progetto perché man mano che il progetto matura, la modalità ottimale può cambiare. Una soluzione è quella di disaccoppiare semplicemente a livello di servizio per impostazione predefinita (il disaccoppiamento a livello di servizio è costoso, sia in termini di tempo di sviluppo che di risorse di sistema ma da grandi vantaggi in termini di flessibilità). Una buona architettura consentirà ad un sistema di nascere come un monolite, distribuito in un unico file, ma poi di crescere in un insieme di unità distribuibili indipendentemente, fino ad arrivare a servizi indipendenti e/o microservizi.

Possiamo definire l'architettura del software come l'arte di tracciare confini, nel separare i vari elementi software così da renderli indipendenti. Si tracciano

linee tra le cose che contano e quelle che non contano: l'interfaccia grafica non è importante per le regole di business, il database non è importante per l'interfaccia grafica e il database non è importante per le regole aziendali.

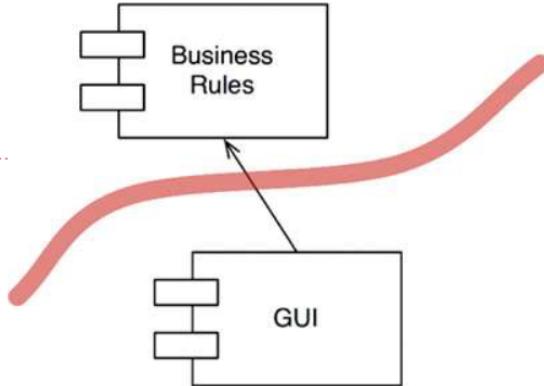


Analizziamo i casi singolarmente. Il database è uno strumento che le regole aziendali possono utilizzare indirettamente. Infatti le regole aziendali non hanno bisogno di conoscere lo schema, il linguaggio di interrogazione o qualsiasi altro dettaglio del database; devono solo sapere che esiste un insieme di funzioni che possono essere usate per recuperare o salvare i dati. Questo ci permette di mettere il database dietro a un'interfaccia. Con questa architettura, le BusinessRules possono utilizzare qualsiasi tipo di database.



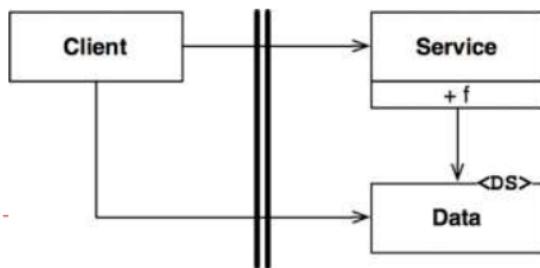
Sviluppatori e clienti pensano che la GUI sia il sistema e non si rendono conto di un principio di fondamentale importanza: l'IO è irrilevante. Il modello non ha bisogno dell'interfaccia, infatti eseguirebbe tranquillamente i suoi compiti,

modellando tutti gli eventi del gioco, senza che il gioco venga mai visualizzato sullo schermo. I componenti GUI e BusinessRules sono separati da una linea di confine.



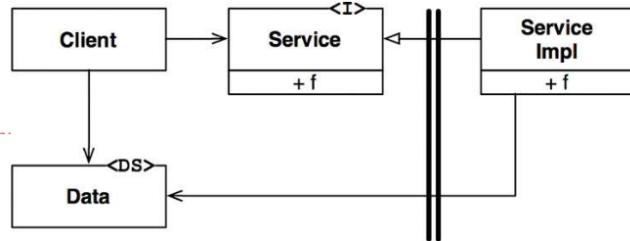
La storia della tecnologia per lo sviluppo del software è la storia di come creare plugin per stabilire un'architettura di sistema scalabile e manutenibile. Le regole di base del business sono tenute separate e indipendenti da quei componenti opzionali o che possono essere implementati in molte forme diverse. Poiché l'interfaccia utente in questo progetto è considerata un plugin, abbiamo reso possibile l'inserimento di diversi tipi di interfacce utente.

Il più semplice e comune dei confini architettonici è semplicemente una segregazione disciplinata di funzioni e dati all'interno di un singolo processore e di un singolo spazio di indirizzi. Dal punto di vista del deployment, si tratta di un singolo file eseguibile, il cosiddetto **monolite**. Il più semplice attraversamento di confini possibile(una funzione su un lato del confine che richiama una funzione dall'altro lato e che passa alcuni dati) è una chiamata di funzione da un client di basso livello a un servizio di livello superiore.



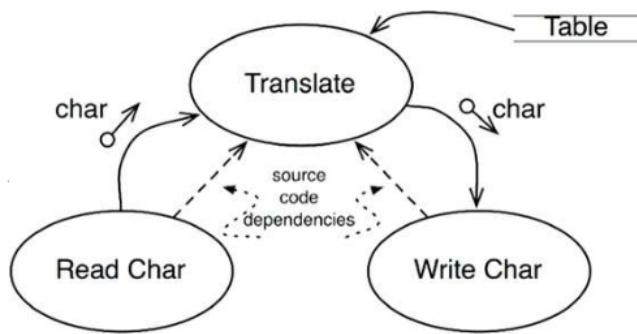
Il flusso di controllo attraversa il confine da sinistra a destra: il client chiama la funzione `f()` sul servizio e passa un'istanza di Data (il marcatore `<DS>` indica semplicemente una struttura di dati) come argomento di una funzione o

con qualche altro mezzo più elaborato. Quando un client ha bisogno di invocare un servizio, il polimorfismo dinamico viene utilizzato per invertire la dipendenza rispetto al flusso di controllo (da sinistra a destra) e il client chiama la funzione f() del ServiceImpl attraverso l'interfaccia Service interfaccia.



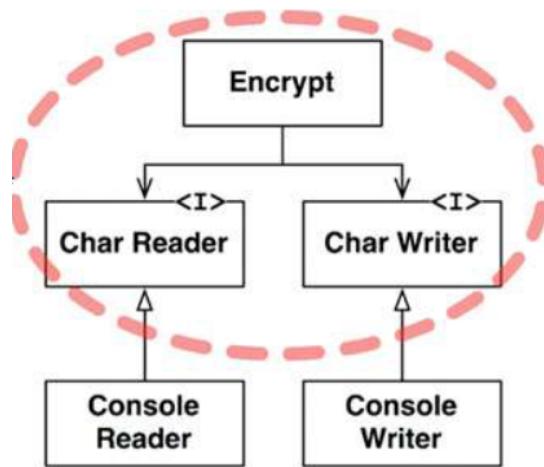
La rappresentazione fisica più semplice di un confine architettonale è una libreria collegata dinamicamente, come una DLL .Net, un file jar Java o una libreria condivisa UNIX. L'atto di distribuzione è semplicemente la raccolta di queste unità distribuibili in una forma conveniente, come un file WAR. Come per i monoliti, le comunicazioni tra i confini dei componenti di distribuzione sono solo chiamate di funzione.

Un programma informatico è una descrizione della politica con cui gli input vengono trasformati in output. Nella maggior parte dei sistemi, questa politica può essere suddivisa in dichiarazioni di politica più piccole che descrivono come devono essere calcolate particolari regole aziendali, come devono essere formattati determinati report e come devono essere convalidati i dati di input. Il progetto dell'architettura deve separare e raggruppare le politiche in base ai modi in cui cambiano (queste componenti raggruppati formano un grafo aciclico diretto).

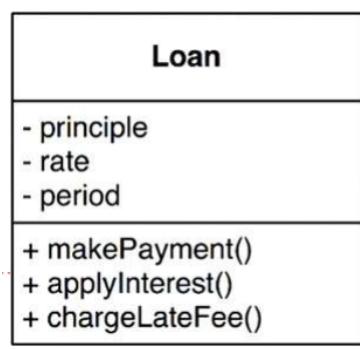


Una definizione rigorosa di "livello" è "la distanza tra gli ingressi e le uscite". Il diagramma del flusso di dati illustra un semplice programma di crittografia che legge i caratteri da un dispositivo di ingresso, traduce i caratteri utilizzando una tabella, li traduce utilizzando una tabella e scrive i caratteri tradotti su

un dispositivo di uscita. Il componente Translate è il componente di livello più alto in questo sistema perché è il componente più lontano dagli ingressi e dalle uscite. Ma l'architettura sopra descritta non è corretta perché la funzione di alto livello encrypt dipende dalle funzioni di basso livello readChar e writeChar. Un'architettura migliore è mostrata nel diagramma delle classi che disaccoppia la politica di crittografia di alto livello dalle politiche di input/output di livello inferiore. Mantenere queste politiche separate, con tutte le dipendenze del codice sorgente che puntano in direzione del livello superiore, riduce l'impatto delle modifiche.



Vediamo ora cosa sono nello specifico le regole di business. Le regole di business sono regole o procedure che fanno guadagnare o risparmiare denaro all'azienda. In particolar modo possiamo definire le Critical Business Rules, le busines rules fondamentali per l'azienda stessa e esisterebbero anche se non ci fosse un sistema informatico. Queste business rules possono richiedere dati di business critici con cui lavorare; le regole e i dati critici sono inestricabilmente legati quindi sono un buon candidato per un oggetto. Chiameremo questo tipo di oggetto Entità; in particolar modo un'Entità è un oggetto all'interno del nostro sistema informatico che racchiude un piccolo insieme di regole aziendali critiche che operano sui dati aziendali critici. Creando questo tipo di classe, si raccoglie il software che implementa un concetto critico per l'azienda e lo si separa da ogni altro aspetto.(si legano i dati aziendali critici e le regole aziendali critiche in un singolo modulo software separato).

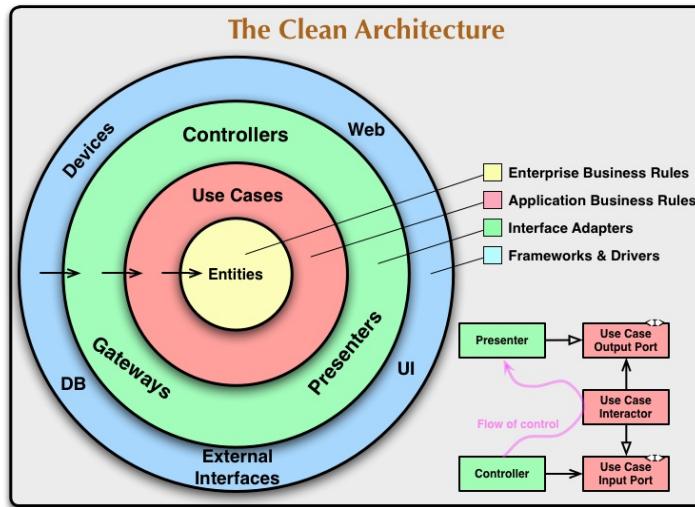


Non tutte le regole aziendali sono Entità infatti alcune regole di business definiscono il modo in cui opera un sistema automatico. Quest'ultime vengono chiamate use case: un caso d'uso è una descrizione del modo in cui viene utilizzato un sistema automatizzato che specifica l'input che l'utente deve fornire, l'output che deve essere restituito all'utente e le fasi di elaborazione coinvolte nella produzione di tale output. I casi d'uso contengono le regole che specificano come e quando vengono invocate le regole di business critiche all'interno delle entità. Sono regole specifiche dell'applicazione che regolano l'interazione tra utenti ed entità:

- Le entità non conoscono i casi d'uso che le controllano.
- I concetti di alto livello, come le entità, non conoscono i concetti di livello inferiore, come i casi d'uso.
- I casi d'uso di livello inferiore conoscono le Entità di livello superiore.

Le architetture software sono strutture che supportano i casi d'uso del sistema e in particolar modo le buone architetture sono incentrate sui casi d'uso, in modo che gli architetti possano descrivere in modo sicuro le strutture che supportano i casi d'uso senza impegnarsi in framework, strumenti e ambienti. Qualsiasi architettura deve essere indipendente dai framework/librerie esterne, indipendente dalla UI, indipendente dal database, indipendente da fattori esterni e deve essere testabile, cioè le regole di business possono essere testate senza l'interfaccia utente, il database, server web o qualsiasi altro elemento esterno.

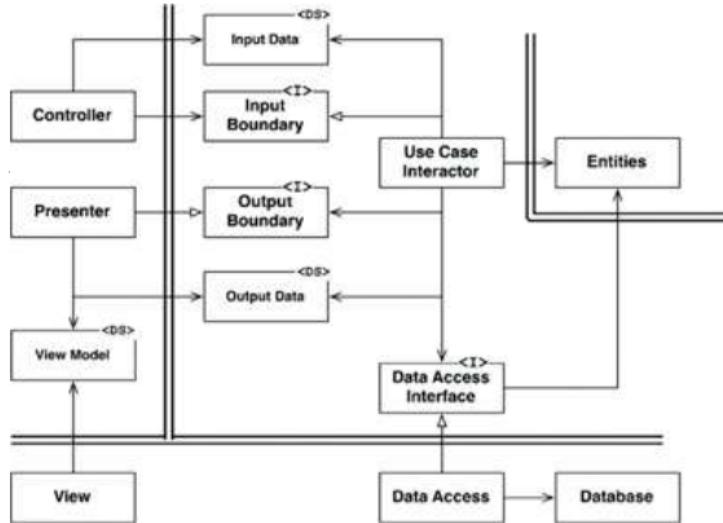
La **Clean Architecture** è una filosofia di progettazione del software che separa gli elementi di un progetto in livelli ad anello.



I cerchi concentrici rappresentano diverse aree del software: più si va avanti, più il software diventa di livello superiore. I cerchi esterni sono i meccanismi, invece i cerchi interni sono le politiche. La regola fondamentale che fa funzionare questa architettura è la regola delle dipendenze, cioè le dipendenze del codice sorgente devono puntare solo verso l'interno, verso politiche di livello superiore (dove il livello di astrazione aumenta).

- Le entità encapsulano le regole di business critiche a livello aziendale e possono essere utilizzate da molte applicazioni diverse nell'azienda.
- Il software nel livello dei casi d'uso contiene regole di business specifiche per l'applicazione.
- Il livello degli adattatori di interfaccia converte i dati dal formato utilizzato dai casi d'uso e dalle entità al formato utilizzato dall'agenzia esterna, ad esempio il database o il web. Il livello include qualsiasi altro adattatore necessario per convertire i dati da forma esterna, come ad esempio un servizio esterno, alla forma interna utilizzata dai casi d'uso e dalle entità.
- Lo strato più esterno è composto da framework e strumenti, come il database e il framework web (non scriviamo molto codice in questo livello).

In basso a destra del diagramma di architettura è mostrato come i controlleri e i presentatori comunicano con i casi d'uso del livello successivo. Il flusso di controllo inizia nel controllore, passa attraverso il caso d'uso e finisce poi nel presentatore. La stessa tecnica viene utilizzata per attraversare tutti i confini delle architetture (in genere i dati che attraversano i confini sono costituiti da strutture di dati semplici). Vediamo ora un tipico scenario per una applicazione web-based Java che fa uso di un database.



Il server Web raccoglie i dati di input dall'utente e li trasmette al Controllore in alto a sinistra. Il controllore impacchetta i dati in un oggetto e li passa attraverso l'InputBoundary all'UseCaseInteractor. L'UseCaseInteractor interpreta i dati e li usa per controllare le entità. Inoltre utilizziamo l'interfaccia DataAccessInterface per portare in memoria dal database i dati utilizzati dalle entità. Al termine, l'UseCaseInteractor raccoglie i dati dalle entità e costruisce l'OutputData. L'OutputData viene quindi passato al Presentatore attraverso l'interfaccia OutputBoundary. Il Presentatore riconfeziona gli OutputData in forma visualizzabile come ViewModel, un altro oggetto. La Vista sposta i dati dal ViewModel alla pagina HTML.

All'interno di questa struttura utilizziamo anche **l'Humble Object** Pattern che è un design pattern utile per aiutare i tester delle unità a separare comportamenti difficili da testare da quelli facili da testare. L'idea è molto semplice: dividere i comportamenti in due moduli o classi.

1. Uno è humble; contiene tutti i comportamenti difficili da testare, ridotti alla loro essenza.
2. L'altro modulo contiene tutti i comportamenti testabili

Utilizzando lo schema Humble Object, possiamo separare questi due tipi di comportamenti in due classi diverse, chiamate Presenter e View. Le GUI sono difficili da testare perché è molto difficile scrivere test che vedano lo schermo e verifichino che gli elementi appropriati siano visualizzati e quindi la View è l'oggetto umile difficile da testare e il Presenter è l'oggetto testabile. Il Main è l'ultimo dettaglio, la politica di livello più basso. È il punto di ingresso iniziale del sistema e nulla, a parte il sistema operativo, dipende da esso. Inoltre è in questo componente principale che le dipendenze devono essere iniettate da un framework di Dependency Injection.

3.1.4 Meta-Architectural Knowledge

Possiamo definire Meta-Architectural Knowledge come le caratteristiche comuni alle architetture software. La comunità dell'architettura del software ha codificato queste conoscenze in: modelli architettonici, tattiche architettoniche e architetture di riferimento. La conoscenza meta-architettonica ha due scopi principali:

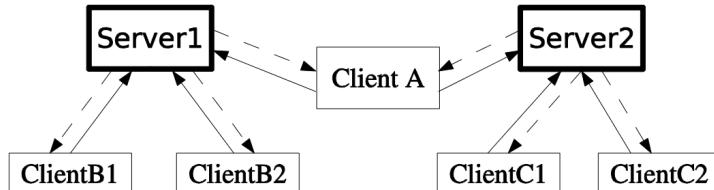
1. Può essere usata come punto di partenza per l'architettura di un particolare sistema, risparmiando un po' di lavoro e fornendo una guida per l'architettura finale.
2. È un meccanismo di comunicazione efficace per fornire un'idea rapida della struttura di alto livello di un sistema.

Partiamo dalla descrizione dei modelli architettonici:

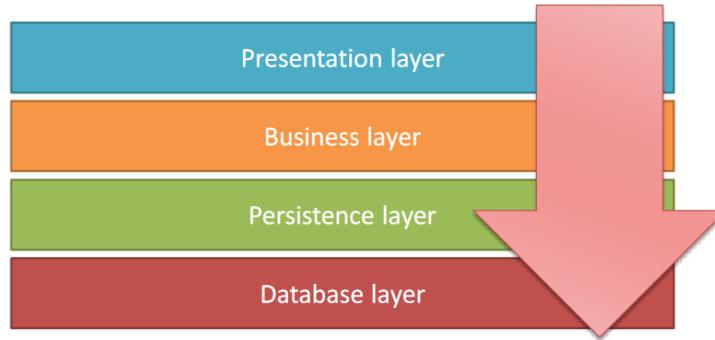
1. Monolithic

• Layered Architecture

Basata sulla meccanica di **Client-Server**, nella quale il processo client è il richiedente del servizio o dell'informazione e il processo del server accetta la richiesta del client, esegue l'elaborazione, raccoglie le informazioni richieste e genera la soluzione per il cliente (dopo l'invio del servizio il server è pronto ad accettare altre richiesta). L'architettura client-server è fortemente influenzata dalle modifiche hardware e dai costi di mantenimento.

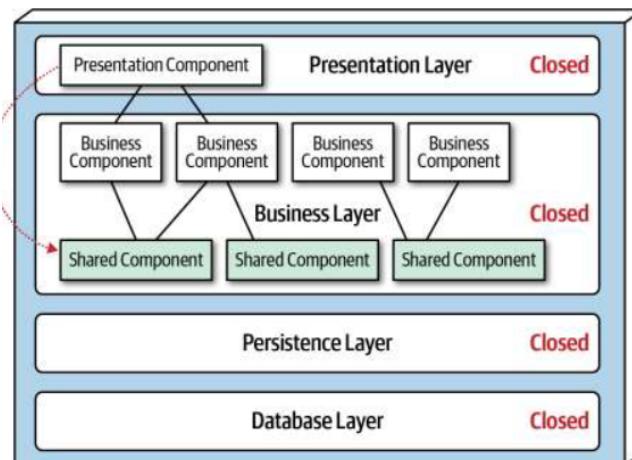


I componenti sono raggruppati in strati logici orizzontali (livelli o tier) e comunicano solo con gli altri componenti degli strati immediatamente sopra e sotto il proprio livello. A volte presenta un problema di prestazioni in termini di numero di livelli che un messaggio può dover attraversare prima di essere elaborato. La maggior parte delle architetture a strati è composta da quattro livelli standard (non ci sono restrizioni specifiche in termini di numero e tipi di strati che devono esistere)

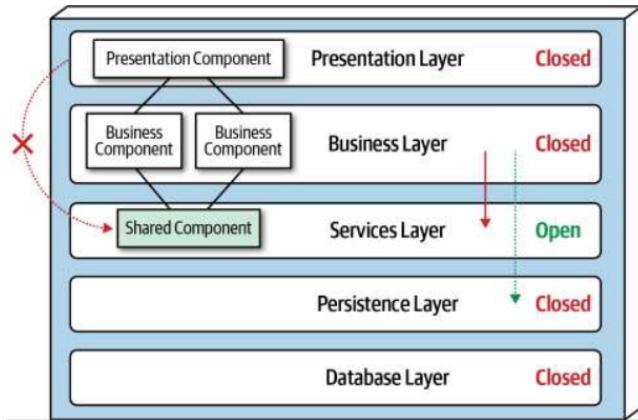


Ogni livello è indipendente e non ha quindi alcuna conoscenza del funzionamento interno degli altri livelli dell'architettura; questo significa che le modifiche apportate a un livello non hanno alcun impatto o effetto sui componenti di altri livelli.

In alcuni casi è opportuno che alcuni livelli siano aperti. Ad esempio, supponiamo che all'interno del livello business ci siano oggetti condivisi con funzionalità comuni per i componenti di business. Esiste una decisione di architettura che stabilisce che il livello di presentazione non può utilizzare questi oggetti di business condivisi. Questo scenario è difficile da governare e controllare, perché architettonicamente il livello di presentazione deve accedere al livello di business e agli oggetti condivisi di tale livello.



Un modo per imporre architettonicamente questa restrizione è quello di aggiungere all'architettura un nuovo livello di servizi che contenga tutti gli oggetti di business condivisi.



Questo problema dell'architettura a livelli viene chiamato *architecture sinkhole anti-pattern* ed uno degli approcci per risolverlo è quello di rendere aperti tutti gli strati dell'architettura, rendendosi conto, ovviamente, che il compromesso è una maggiore difficoltà nel gestire i cambiamenti all'interno dell'architettura.

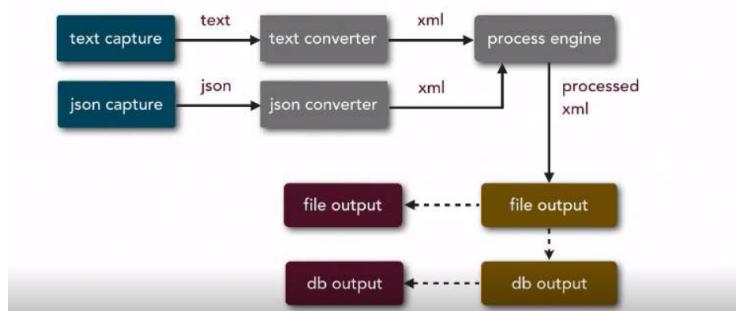
- **Pipe & Filter Architecture**

È un sistema di flusso di dati per semplici sistemi di elaborazione dati. Vengono definiti due tipi di componenti:

- (a) Pipes (flusso unidirezionale di informazioni: messaggi -testuali o binari)
- (b) Filtri (autonomi, indipendenti da altri filtri e senza stato. Progettati per eseguire un singolo compito specifico) che possono essere produttori, trasformatori, operatori o consumatori.

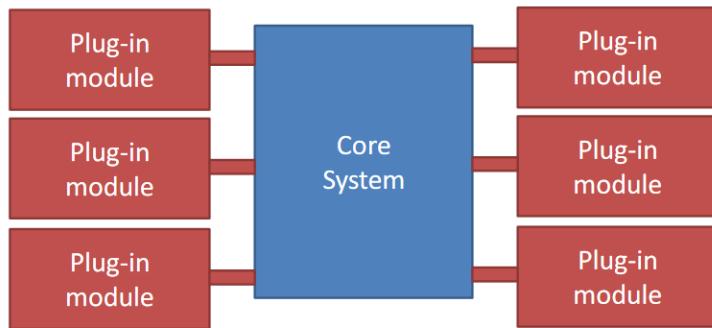
I problemi che richiedono l'elaborazione di file batch sembrano adattarsi a questa architettura (payroll, compilers).

example: capture data in multiple formats, process the data, and send to multiple outputs



I principali vantaggi di questa architettura sono che i filtri sono adatti ad essere riutilizzati in diverse parti del sistema (facili da mantenere e migliorare), possono essere implementati e testati in modo indipendente, possono essere eseguiti in parallelo. Ma i filtri devono essere pianificati in funzione dell'integrazione, non è adatta ai sistemi interattivi e l'efficienza è ridotta a causa della computazione necessaria per comporre e analizzare i dati tra i diversi componenti.

- **Microkernel Architecture** aka Plug-in architecture pattern

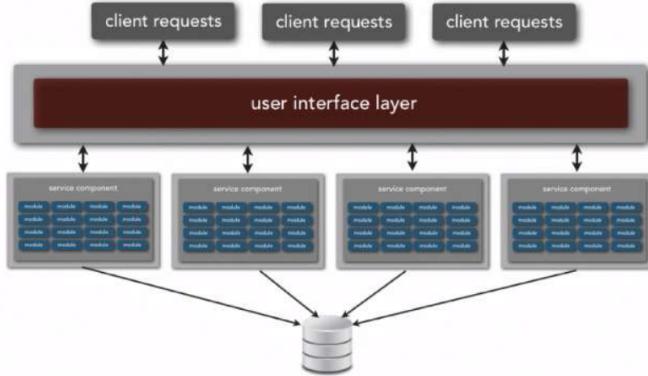


Formata da due componenti: lo Small core system che implementa le funzionalità minime per il funzionamento del sistema / regole e logiche aziendali generali e i Plug-in (moduli indipendenti autonomi) che aggiungono funzionalità. All'interno del sistema centrale esiste un registro che permette al sistema centrale di conoscere i plug-in disponibili e di sapere come raggiungerli (contiene informazioni su ogni modulo plug-in). I contratti tra i componenti plug-in e il sistema centrale sono solitamente standard ed includono il comportamento, i dati di ingresso e i dati di uscita restituiti dal componente plug-in. Questa architettura è molto flessibile ed è quindi in grado di rispondere velocemente ai cambiamenti.

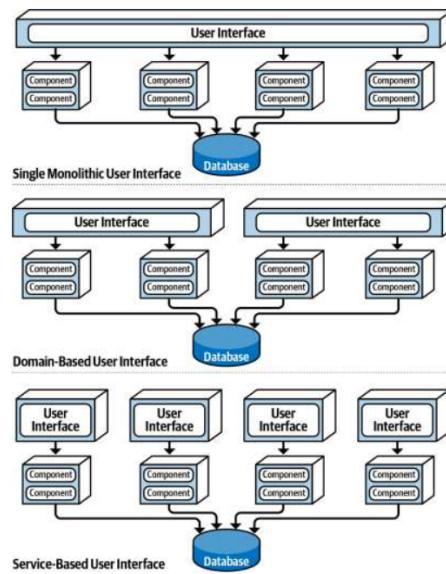
2. Distributed (più unità di distribuzione collegate tramite protocolli di protocolli di accesso remoto). Sono più potenti in termini di prestazioni, scalabilità e disponibilità rispetto agli stili di architettura monolitici ma soffrono di alcuni problemi: la rete non è sempre affidabile e non è sicura, il tempo di latenza deve essere considerato, la larghezza di banda non è infinita e ci sono costi di trasporto.

- **Service Based Architecture**

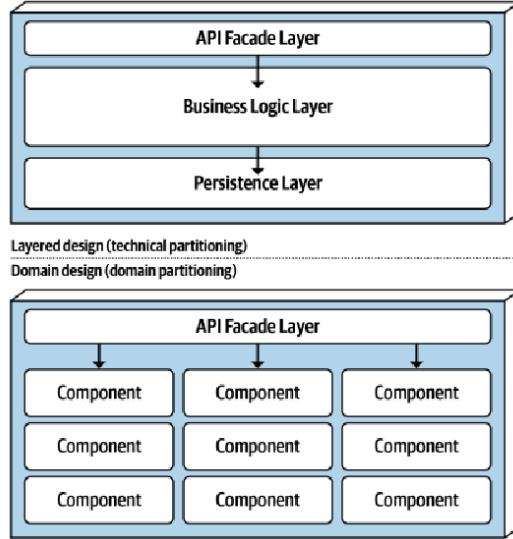
Struttura macro stratificata distribuita composta da un'interfaccia utente distribuita separatamente, servizi remoti a grana grossa distribuiti separatamente e un database monolitico.



I servizi sono tipicamente "porzioni di un'applicazione" indipendenti e distribuite separatamente (di solito chiamati servizi di dominio, esiste una singola istanza del servizio). Sono accessibili a distanza da un'interfaccia utente tramite un protocollo di accesso remoto (REST). Questa architettura usa un database centralizzato condiviso a cui tutti i servizi fanno riferimento; in alternativa si può suddividere un singolo database monolitico in database separati, fino ad arrivare ad un database per ogni servizio di dominio. In questi casi è importante assicurarsi che i dati in ciascun database non siano necessari ad altri servizi di dominio. Di questa architettura esistono tantissime varianti:

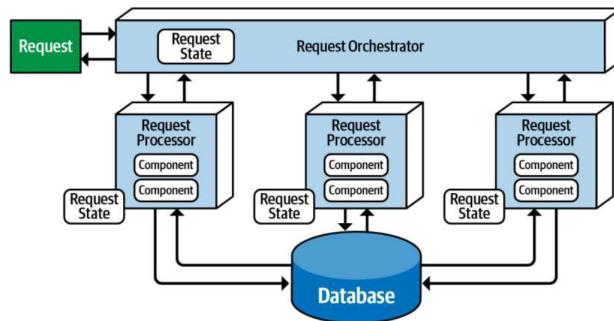


Ogni servizio è costruito utilizzando una architettura a livelli.



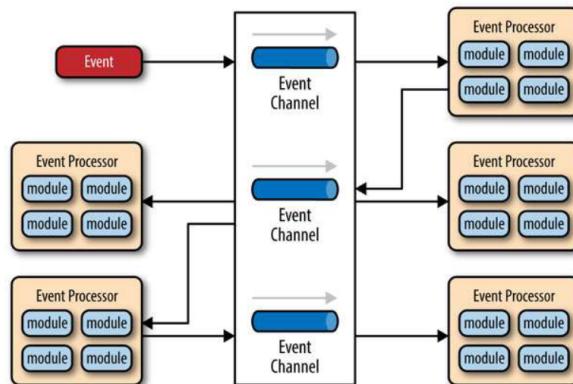
- **Event-driven Architecture**

Lo stile di architettura event-driven è un popolare stile di architettura asincrona distribuita che produce applicazioni altamente scalabili e ad alte prestazioni. È costituito da componenti di elaborazione degli eventi disaccoppiati che ricevono ed elaborano gli eventi in modo asincrono. La maggior parte delle applicazioni segue un modello basato sulle richieste: le richieste fatte al sistema vengono inviate a un orchestratore di richieste (un'interfaccia utente, implementata attraverso un livello API o un service bus aziendale); il ruolo dell'orchestratore è quello di indirizzare in modo deterministico e sincrono la richiesta ai vari processori di richiesta, i quali gestiscono la richiesta, recuperando o aggiornando le informazioni in un database.



L'architettura guidata dagli eventi (EDA) può essere basata su 2 determinate topologie:

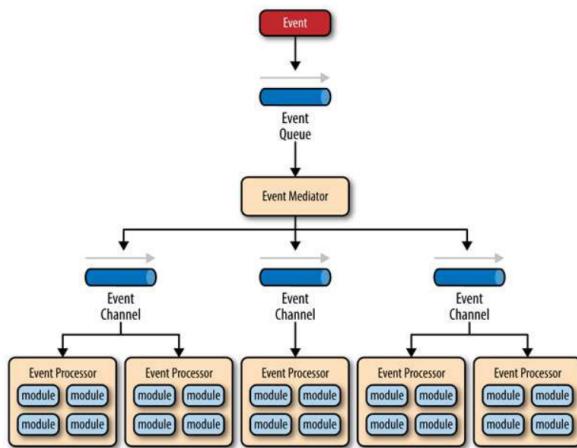
- **Broker topology** è utilizzato quando si richiede un alto grado di reattività e di controllo dinamico sull'elaborazione di un evento. In questa topologia non c'è un mediatore di eventi centralizzato e il flusso dei messaggi è distribuito tra i componenti del processore di eventi in modo concatenato. Esistono quattro tipi principali di componenti architettonici:
 - (a) Un evento iniziatore avvia l'intero flusso di eventi
 - (b) Il broker di eventi riceve in un event channel l'evento iniziale
 - (c) Un processore di eventi accetta l'evento iniziatore dal broker di eventi e inizia l'elaborazione dell'evento. Esegue un'attività specifica associata all'elaborazione di quell'evento in modo sincrono e comunica ciò che ha fatto al resto del sistema creando un evento di elaborazione.
 - (d) Questo evento di elaborazione viene inviato in modo asincrono al broker di eventi per un'ulteriore elaborazione. Questo processo continua fino a quando nessuno è interessato a ciò che ha fatto l'ultimo processore di eventi.



Con questa topologia abbiamo una maggiore reattività, scalabilità e prestazione; tuttavia non c'è controllo sul flusso di lavoro complessivo associato all'evento iniziatore e la gestione degli errori è una grande sfida con la topologia del broker (se si verifica un guasto, nessuno nel sistema è a conoscenza dell'incidente).

- **Mediator topology** è comunemente usato quando si richiede il controllo sul flusso di lavoro di un processo di eventi. Al centro di questa topologia c'è un mediatore di eventi, che gestisce e controlla il flusso di lavoro per l'avvio di eventi che richiedono

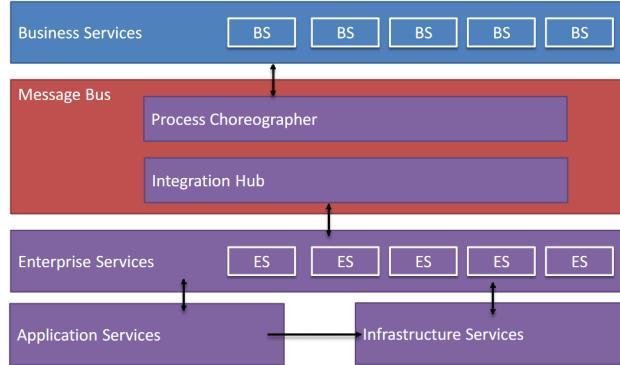
il coordinamento di più processori di eventi. L'evento iniziatore viene inviato a una coda di eventi iniziatori, che viene accettata dal mediatore di eventi. Il mediatore di eventi conosce solo le fasi di elaborazione dell'evento e quindi genera gli eventi di elaborazione corrispondenti, che vengono inviati a canali di eventi dedicati (di solito code) con comunicazioni punto a punto. I processori di eventi ascoltano quindi i canali di eventi dedicati, elaborano l'evento e di solito rispondono al mediatore che hanno completato il loro lavoro.



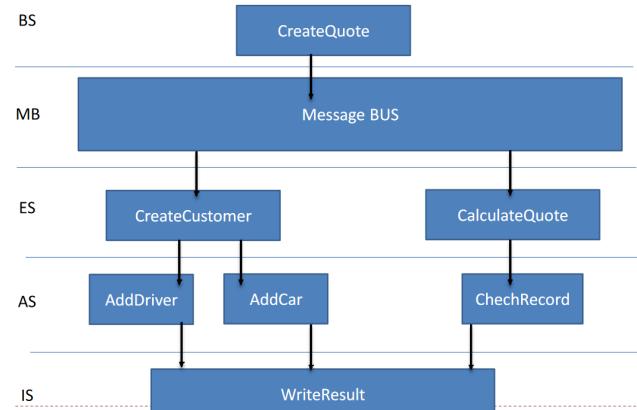
Nella maggior parte delle implementazioni, ci sono più mediatori, di solito associati a un particolare dominio o gruppo di eventi (questo aumenta il throughput e le prestazioni complessive). Il mediatore di eventi può essere implementato in vari modi, a seconda della natura e della complessità degli eventi che deve elaborare. In questa tipologia il componente mediatore ha conoscenza e controllo del flusso di lavoro e quindi può mantenere lo stato degli eventi e gestire la gestione degli errori (recuperare gli eventi) ma le performance sono minori rispetto alla tipologia broker e possono crearsi dei notevoli ritardi nelle code di eventi.

- **Service Oriented**

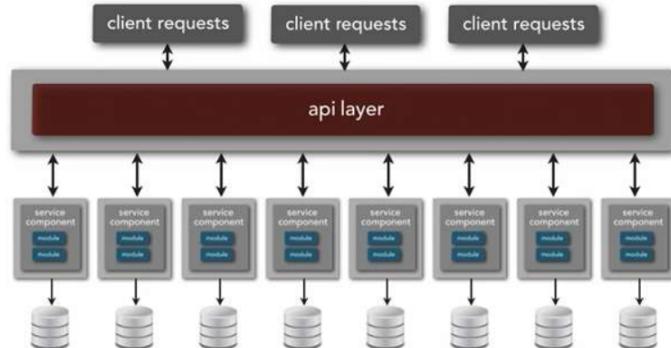
La struttura dell'architettura Service Oriented è definita nel seguente modo.



I Business Service sono i servizi richiesti dagli utenti e che forniscono il punto di ingresso; non contengono codice, ma solo input e output, e talvolta informazioni sullo schema. Gli Enterprise Service sono applicazioni concrete e che possono essere riutilizzati in altre applicazioni. Gli Infrastructure Services implementano funzionalità non aziendali e gli Application Services sono delle applicazioni specifiche. Questa architettura è tipicamente legata ad un singolo database relazionale piuttosto che a un database per servizio come nelle architetture a microservizi. Il motore di orchestrazione definisce la relazione tra i servizi aziendali e di business, il modo in cui si integrano tra loro; funge anche da hub di integrazione, consentendo agli architetti di integrare il codice personalizzato con i pacchetti e il software legacy.

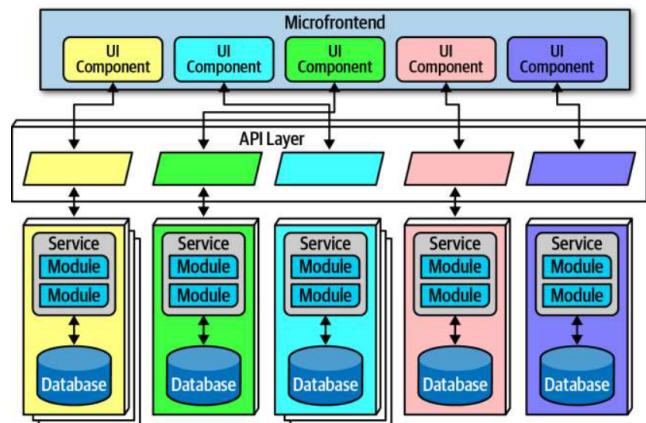
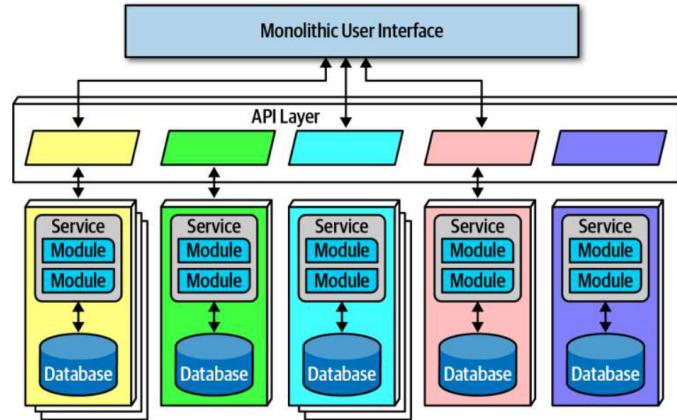


- **Microservices** Nell'architettura a Microservizi ogni componente viene distribuito come unità separata (all'interno di una pipeline efficace e snella).

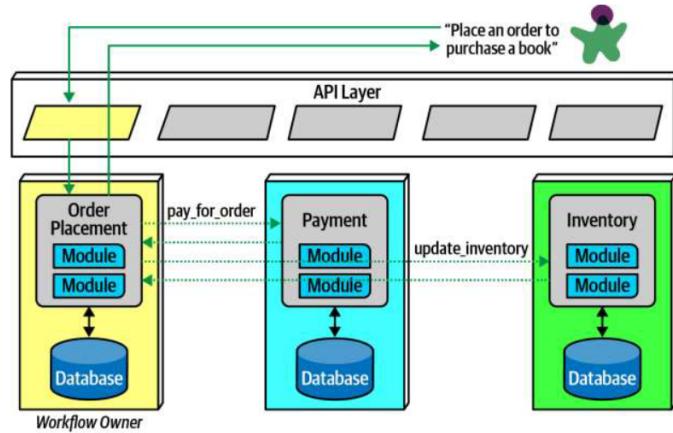


Le prestazioni sono spesso l'effetto collaterale negativo della natura distribuita dei microservizi. Infatti le chiamate di rete richiedono molto più tempo rispetto alle chiamate di metodo e la verifica della sicurezza in ogni endpoint aggiunge ulteriore tempo di elaborazione, richiedendo agli architetti di riflettere attentamente sulle implicazioni della granularità durante la progettazione del sistema. Gli architetti faticano a trovare la giusta granularità per i servizi nei microservizi e spesso commettono l'errore di rendere i loro servizi troppo piccoli, il che li obbliga a costruire una comunicazione tra i servizi per svolgere un lavoro utile. Lo scopo dei confini dei servizi nei microservizi è quello di catturare un dominio o un flusso di lavoro; infatti ogni servizio deve modellare un dominio o un flusso di lavoro, includendo tutto ciò che è necessario per l'applicazione, comprese classi, altri sottocomponenti e schemi di database. Per delimitare i confini dei vari servizi gli architetti del software possono seguire questi 4 principi:

- Purpose: il confine più ovvio si basa sull'ispirazione dello stile architettonico, il dominio.
- Transactions: spesso le entità che devono cooperare in una transazione possono essere accorpate
- Choreography: L'architetto può prendere in considerazione la possibilità di raggruppare dei servizi in un servizio più ampio per evitare troppa comunicazione.
- Data Isolation: i microservizi cercano di evitare tutti i tipi di accoppiamento, compresi gli schemi condivisi e i database utilizzati come punti di integrazione.



In questa architettura non esiste un coordinatore centrale e quindi utilizza lo stesso stile di comunicazione di un broker EDA. Poiché le architetture a microservizi non includono un mediatore globale come altre architetture orientate ai servizi, se un architetto ha bisogno di coordinarsi tra più servizi, può creare il suo mediatore localizzato, come mostrato in figura:



Questo pattern è chiamato front controller, dove un singolo servizio diventa un mediatore per alcuni problemi.

Uno dei grandi vantaggi presunti della suddivisione di un sistema in servizi è che i servizi sono fortemente disaccoppiati tra loro; infatti ogni servizio viene eseguito in un processo diverso, o addirittura in un processore diverso. Purtroppo i servizi sono fortemente accoppiati dai dati che condividono (il cambiamento di un dato può provocare una modifica all'intero servizio). Un altro dei presunti vantaggi dei servizi è che possono essere posseduti e gestiti da un team dedicato; questo team può essere responsabile della scrittura, della manutenzione e del funzionamento del servizio come parte di una strategia di cooperazione.

3.2 Design Characteristics and Metrics

Non esiste una definizione univoca di buon design perché un buon progetto è caratterizzato da diversi attributi. I due requisiti più importanti per un buon design sono la coerenza grafica (UI con aspetto logico rispetto ai requisiti) e la completezza del design rispetto ai requisiti. Prima di introdurre le metriche per valutare un buon design dobbiamo definire i principi e le motivazioni di tali metrich (IEEE Standard 1061, 1998):

- *The purpose of software metrics is to make assessments throughout the software life cycle as to confirm whether the software quality requirements are being met*
- *The use of software metrics reduces subjectivity in the assessment and control of software quality* (con l'introduzione delle metriche aumenta l'oggettività)
- *However, the use of software metrics does not eliminate the need for human judgment in software evaluations*

Una metrica software rappresenta una misura quantitativa del grado in cui il sistema, un componente o un processo software possiede una proprietà o caratteristica definita. I risultati sono valori misurati e l'interpretazione di tali valori ha un forte carattere empirico, cioè dobbiamo sta al valutatore dargli un significato. In genere possiamo dividere le metriche in 3 grandi famiglie:

1. Metriche di prodotto: caratterizzano un prodotto intermedio o finale del processo di sviluppo del software (dimensione del codice, la complessità del codice e la soddisfazione del cliente)
2. Metriche di processo : caratterizzano i metodi, le tecniche e gli strumenti impiegati nello sviluppo, implementazione e manutenzione del sistema software (l'efficacia nella rimozione dei difetti durante lo sviluppo o il tempo di risposta dei processi di correzione dei bug).
3. Metriche di progetto: caratterizzano il progetto in termini di tempo, budget e gestione (numero di sviluppatori di software, l'assetto del personale nel ciclo di vita del software, i costi, le tempistiche e la gestione del progetto).

Le metriche di prodotto, le quali misurano la qualità del software, si dividono in misure statiche e misure dinamiche.

L'**analisi statica del codice sorgente** riflette gli aspetti della qualità della codifica, della complessità della codifica e della modularizzazione.

- La metrica delle linee di codice (**LOC**) è un insieme di metriche del software che misura quantitativamente le dimensioni del codice sorgente contando le "linee di codice sorgente". Ciò che si considera una "riga di codice sorgente" dipende dal tipo specifico di metrica LOC (misurato anche in 1000 unità, abbreviato con KLOC). Sono utilizzate per determinare l'ordine di grandezza dell'implementazione di un sistema informatico complesso, per stimare e prevedere gli sforzi e per stimare i costi e gli sforzi generali. Il rapporto tra linee di commento e linee di codice sorgente (CLOC/LOC) riflette il grado di documentazione del codice sorgente. Esistono diverse varianti della metrica LOC:

1. LOC o linee fisiche: una linea corrisponde a una riga nel testo del codice sorgente
2. SLOC: linee di codice del codice sorgente, senza commenti e righe vuote
3. NCLOC: linee di codice non commentate; esclude i commenti, le intestazioni e i più di pagina come le dichiarazioni di inclusione, le parentesi graffe dei corpi delle funzioni, ecc.

```
public static int factorial(int n){  
    // n= number of interest  
    int result = 1;  
    for(int i = 2; i <= n; i++){
```

```

        result *= i;
    }
    return result;
}

//LOC = 8
//SLOC = 6
//NCLOC = 5

```

Questa metrica però porta con sé notevoli svantaggi: il LOC può essere correlato con l'impegno ma non con la funzionalità, non considera la complessità se viene utilizzato come misura dell'attività, gli sviluppatori possono facilmente compromettere tale misura con "stili di codifica opportunistici" ed infine per sua natura, dipende dal linguaggio di programmazione è quindi risulta difficile confrontare la LOC di unità di codice sorgente scritte in diversi linguaggi di programmazione.

- La metrica della **duplicazione del codice** valuta il grado di ripetizione e ridondanza utilizzando tool di ricerca di cloni basata sul testo (corrispondenze sintattiche) e di ricerca di cloni basata su token (robusta contro la riformattazione e modifiche alla documentazione). La duplicazione e la copia del codice può essere categorizzata in 4 categorie:
 1. Tipo-1: frammenti di codice identici, tranne che per variazioni di spazi bianchi, layout e commenti.
 2. Tipo-2: Frammenti di codice sintatticamente identici, eccetto che per le variazioni in identificatori, letterali, tipi, spazi bianchi, layout e commenti.
 3. Tipo 3: Frammenti copiati con ulteriori modifiche, come ad esempio dichiarazioni cambiate, aggiunte o rimosse, oltre a variazioni in identificatori, letterali, tipi, spazi bianchi, layout e commenti.
 4. Tipo-4: Due o più frammenti di codice che eseguono lo stesso calcolo ma sono implementati da varianti sintattiche diverse.
- La metrica del livello di annidamento (**NL**) è una metrica del software che misura quantitativamente la complessità del codice sorgente misurando la profondità di racchiudere le strutture di controllo una dentro l'altra (livello profondità di cicli innestati). I codici sorgente applicano il nesting in diverse forme, tra cui chiamate annidate, livelli annidati di parentesi nelle espressioni aritmetiche, blocchi annidati di codice sorgente, classi annidate e funzioni annidate. Esistono tante metriche per il livello di annidamento ma la più importante è la **NL/NLMAX** che misura il livello di annidamento più alto di una data unità di codice sorgente.

```

public String truncateAInFirst2Positions(String str)
{
    String first2Chars = str.substring(0, 2);

```

```

        String stringMinusFirst2Chars = str.substring(2);
        return first2Chars.replaceAll("A", "") +
               stringMinusFirst2Chars;
    }
    // funzione con un unico flusso di controllo lineare,
    // cioe' senza decisioni o ripetizioni.
    // NLMAX = 1

    public String truncateAInFirst2Positions(String str)
    {
        if (str.length() <= 2)
            return str.replaceAll("A", "");
        String first2Chars = str.substring(0, 2);
        String stringMinusFirst2Chars = str.substring(2);
        return first2Chars.replaceAll("A", "") +
               stringMinusFirst2Chars;
    }
    // funzione con un if (con una condizione) con
    // singola decisione
    // NLMAX=2

    public boolean
        areFirstAndLastTwoCharactersTheSame(String str) {
        if (str.length() <= 1)
            return false;
        if (str.length() == 2)
            return true;
        String first2Chars = str.substring(0, 2);
        String last2Chars = str.substring(str.length() -
                                         2);
        return first2Chars.equals(last2Chars);
    }
    // il controllo di flusso contiene due if con una
    // condizione ciascuno
    // NLMAX = 2

    public void printStar() {
        for (int i = 0; i<10; i++)
            for (int j=0; j<10; j++)
                if ((i+j) % 2)
                    System.out.println("*");
                else
                    System.out.println(" ");
    }
    // il controllo di flusso contiene 2 loop innestati e
    // un if

```

```
// NLMAX = 4
```

Valori elevati di NL indicano in genere che è necessario spendere maggiori sforzi per la comprensione dell'unità di codice rispetto alla comprensione del programma. L'osservazione di fondo è che il codice profondamente anidato diventa estremamente difficile da comprendere da un punto di vista cognitivo (il livello di nidificazione identifica quelle unità o parti di codice sorgente che potrebbero essere difficili da capire, correggere o mantenere). Inoltre le misure metriche di NL possono essere utilizzate per identificare le parti di codice sorgente che dovrebbero essere riprogettate o rifattorizzate usando modularizzazione, astrazione o strategie di encapsulamento. Normalmente preferiamo un NL baso per diminuire la complessità ed aumentare la flessibilità e la leggibilità.

- **Halstead Complexity Metric** sviluppata da Maurice Halstead negli anni '70 per analizzare la complessità del codice sorgente dei programmi. Si concentra sulla complessità lessicografica del codice sorgente e sugli operatori ed operandi definiti dalla loro relazione reciproca: un operatore esegue un'azione e un operando partecipa a tale azione. La metrica utilizza quattro unità metriche per l'analisi di un programma sorgente: n1 = numero di operatori distinti, n2 = numero di operandi distinti, N1 = somma di tutte le occorrenze di n1 e N2 = somma di tutte le occorrenze di n2. Inoltre ci sono altre misure importanti: vocabolario del programma o token unici n = n1 + n2, la lunghezza del programma N = N1 + N2, Volume V = N * (Log2 n), difficoltà = (n1/2) * (N2/n2), sforzo E = V * DT e tempo necessario per programmare = E/18 sec.

```
voidsort( int*a, intn ) {
inti, j, t;
if( n < 2 ) return;
for( i=0 ; i < n-1; i++ ) {
    for( j=i+1 ; j < n ; j++ ) {
        if( a[i] > a[j] ) {
            t= a[i];
            a[i] = a[j];
            a[j] = t;
        }
    }
}
Operators      N1 = 50 n1 = 17
Operands       N2 = 30 n2 = 8
```

Volume: $80 \log_2(24) \approx 392$

Difficulty: $D = (n1 / 2) * (N2 / n2) = 17/2 * 30/8 \approx 36$

Effort: $E = V * D$

Time to understand/implement (sec): T

= $E/18$

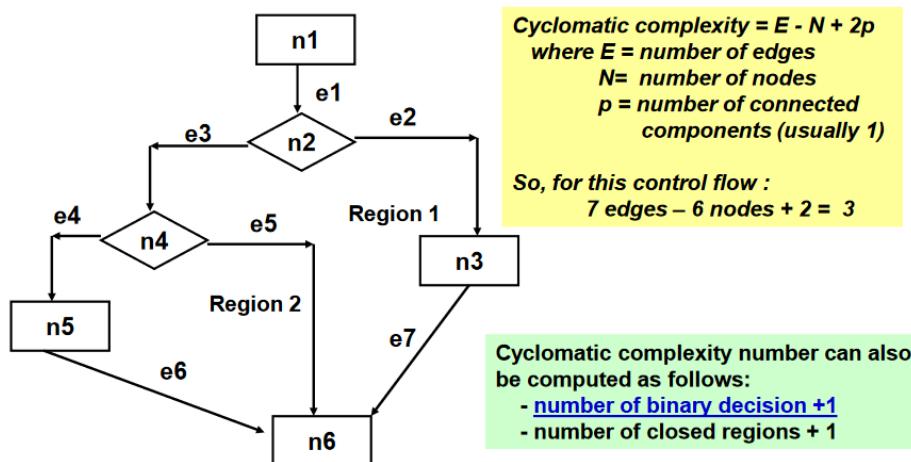
793 sec ≈ 13 min

Bugs delivered: $E2/3/3000$

For C/C++: known to underapproximate

- **McCabe's Cyclomatic Complexity** misura di complessità ciclomatica per il software nata dall'osservazione di T.J. McCabe. La complessità ciclomatica di un programma o di un progetto dettagliato viene calcolata a partire da una rappresentazione del diagramma del flusso di controllo del programma. È molto utile in fase di test per individuare il numero di test necessari.

Complessità ciclomatica = $E - N + 2p$, dove E = numero di archi del grafo, N = numero di nodi del grafo e p = numero di componenti connesse. Può essere anche calcolata nel seguente modo: complessità ciclomatica = numero di decisioni binarie + 1 o complessità ciclomatica = numero di regioni chiuse + 1.



La complessità ciclomatica misura la complessità strutturale del progetto all'interno del programma. Viene applicata all'analisi dei rischi del progetto e del codice e alla pianificazione dei test per valutare il numero di casi di test necessari per verificare ogni punto di decisione. Più alto è il numero di complessità ciclomatica, maggiore è il rischio e la necessità di test.

- **Module dependencies** che descrive il grado di dipendenza del software. Una dipendenza denota una relazione formale tra due moduli: Il modulo A dipende dal modulo B se A chiama B o A eredita da B o A utilizza una variabile di tipo B.
- **Henry-Kafura Information Flow metric** misura il flusso intermodulare. È una misura delle interazioni tra moduli o tra un modulo e il suo ambiente. Fan-in: Numero di informazioni che entrano in un modulo del

programma (calcolato come un conteggio dei moduli che chiamano mod-A); Fan-out: numero di flussi di informazioni in uscita da un modulo di programma (il numero di moduli di programma chiamati da mod-A). La complessità è definita come segue $C_p = (fan-in \times fan-out)^2$

Module	Mod-A	Mod-B	Mod-C	Mod-D
Fan-in	3	4	2	2
Fan-out	1	2	3	2

$$C_p \text{ for Mod-A} = (3 \times 1)^2 = 9$$

$$C_p \text{ for Mod-B} = (4 \times 2)^2 = 64$$

$$C_p \text{ for Mod-C} = (2 \times 3)^2 = 36$$

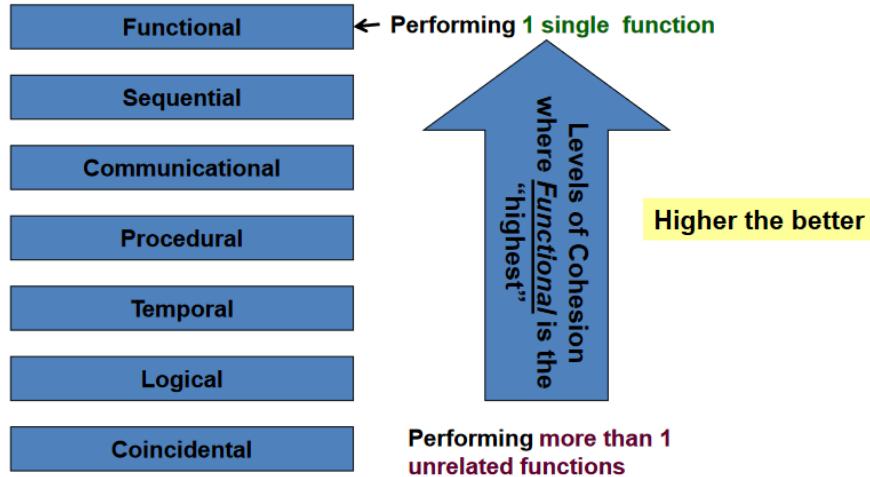
$$C_p \text{ for Mod-D} = (2 \times 2)^2 = 16$$

La complessità strutturale totale di Henry-Kafura per tutti e quattro i moduli del programma sarebbe la somma di questi moduli, che è 125

- Card e Glass hanno utilizzato lo stesso concetto di "fan-in" e "fan-out" per descrivere la complessità del progetto, la complessità strutturale del modulo x è $Sx = (fan-out)^2$

- Le caratteristiche di un buon design sono facile da capire, facile da modificare, facile da riutilizzare, facile da testare, facile da integrare e facile da codificare. Il filo conduttore di tutte queste proprietà "facili da" è la nozione di semplicità. Secondo Yourdon e Constantine (1979), la nozione di semplicità può essere misurata in base a due caratteristiche la coesione e l'accoppiamento.

L'**coesione** di un modulo si riferisce alle dipendenze esterne da altri moduli, cioè il grado di dipendenza inter-modulare (simile alla metrica di Halstead and McCabe). Invece la **coesione** di un modulo si riferisce alle dipendenze interne allo stesso modulo, cioè il grado di dipendenza intramodulare (simile alla metrica di Henry-Kafura fan-in and fan-out information flow). In generale, ci sforziamo di ottenere una forte coesione e un accoppiamento lasco nella progettazione. Per mantenere il design semplice, dobbiamo assicurarci che ogni unità del progetto si concentri su un unico scopo (coesione)



A livello di coesione coincidente, l'unità di progetto o di codice sta eseguendo compiti multipli non correlati; questo livello più basso di coesione di solito non si verifica in una progettazione iniziale. A livello di coesione logica il progetto esegue una serie di compiti simili (sembra avere un senso, in quanto gli elementi sono correlati). A livello di coesione temporale gli elementi sono correlati dal tempo. La coesione procedurale coinvolge azioni che sono correlate in modo procedurale. La coesione comunicativa è un livello in cui il progetto è legato alla sequenza delle attività. Infine la coesione sequenziale e funzionale sono i livelli in cui l'unità di progettazione svolge un'attività principale o raggiunge un obiettivo.

Bieman e Ott (1994) hanno introdotto diverse misure quantitative di coesione a livello di programma, basate su fette di programma e di dati. Consideriamo un programma sorgente: un token di dati è qualsiasi variabile o costante, una fetta all'interno del programma o della procedura è costituita da tutte le istruzioni che possono influenzare il valore di una specifica variabile di interesse, una slice di dati è costituita da tutti i token di dati in una slice che influenzano il valore di una variabile di interesse, i token collanti sono i token di dati che si trovano in più di una slice di dati, i token supercolla sono i token di dati che si trovano in ogni slice di dati. È chiaro che i token collanti e i token supercollanti sono quelli che attraversano le fette e offrono la forza vincolante o la forza di coesione: coesione funzionale debole = Numero di token colla / Numero totale di token dati e coesione funzionale forte = Numero di token supercolla / Numero totale di token di dati.

Finding the maximum
and the minimum values
procedure:

```
MinMax ( z, n )
Integer end, min, max, i ;
end = n ;
max = z[0] ;
min = z[0] ;
For ( i = 0, i = < end , i++ ) {
    if z[ i ] > max then max = z[ i ];
    if z[ i ] < min then min = z[ i ];
}
return max, min;
```

Data Tokens:	Slice max:	Slice min:	Glue Tokens:	Super Glue:
z1	z1	z1	z1	z1
n1	n1	n1	n1	n1
end1	end1	end1	end1	end1
min1	max1	min1	i1	i1
max1	i1	i1	end2	end2
i1	end2	end2	n2	n2
end2	n2	n2	i2	i2
n2	max2	min2	03	03
max2	z2	z3	i3	i3
z2	01	02	end3	end3
01	i2	i2	i4 (11)	i4 (11)
min2	03	03		
z3	i3	i3		
02	end3	end3		
i2	i4	i4		
03	z4	z6		
i3	i5	i7		
end3	max3	min3		
i4	max4	min4		
z4	z5	z7		
i5	i6	i8		
max3	max5	min5		
max4				
z5				
i6				
z6				
i7				
min3				
min4				
z7				
i8				
max5				
min5 (33)				
	(22)	(22)		

A Pseudo-Code Example
of Functional Cohesion
Measure

47

For the example of finding min and max, the glue tokens are the same as the super glue tokens.

- ▶ Super glue tokens = 11
- ▶ Glue tokens = 11

The data slice for min and data slice for max turns out to be the same number, 22

The total number of data tokens is 33

The cohesion metrics for the example of min-max are:

weak functional cohesion = $11 / 33 = 1/3$

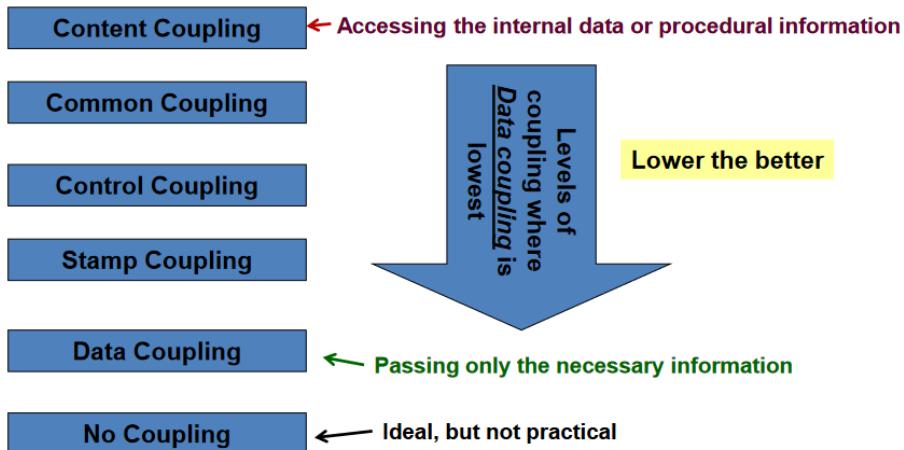
strong functional cohesion = $11 / 33 = 1/3$

If we had only computed one function (e.g. max), then :

weak functional cohesion = $22 / 22 = 1$

strong functional cohesion = $22 / 22 = 1$

L'accoppiamento invece è un attributo che riguarda il grado di interazione e interdipendenza tra due unità software. Le classi strettamente accoppiate sono difficili da riutilizzare isolatamente ed una classe che dipende fortemente da altre classi sarebbe molto difficile da capire da sola.



A livello di accoppiamento di contenuti, due unità accedono ai dati interni o alle informazioni procedurali dell'altra. Due unità software sono a livello di accoppiamento comune se entrambe fanno riferimento alla stessa variabile globale; l'accoppiamento di controllo è il livello in cui un'unità software passa un'informazione di controllo e influenza esplicitamente la logica di un'altra unità software. Nell'accoppiamento di timbro, un'unità software passa un gruppo di dati a un'altra unità software, invece l'accoppiamento dei dati è il livello in cui solo i dati necessari vengono passati tra le unità software. Infine c'è il livello di no accoppiamento, livello ideale ma non sempre praticabile (senza accoppiamento c'è una sola classe che fa tutto e quindi c'è un alto livello di coesione). Per misurare il livello di accoppiamento si usa la metrica di **Fenton and Melton** nella quale si assegna a ciascun livello di accoppiamento un numero intero da 1 a 5, si valutano le unità software x e y identificando il più alto livello di accoppiamento tra le coppie x, y e assegnarlo come i, si identificano tutte le relazioni di accoppiamento tra x e y per assegnarle ad n ed infine si definisce l'accoppiamento a coppie come $C(x, y) = i + [n/(n + 1)]$.

- **Object-Oriented Design Metrics** misurano come le classi sono interrelate tra loro e come le classi interagiscono tra loro. Le metriche CK misurano la complessità del progetto in relazione all'impatto sugli attributi esterni di qualità esterna, come la manutenibilità e la riusabilità,

convalidate teoricamente ed empiricamente. WMC (Metodi per classe) è la somma ponderata di tutti i metodi di una classe. DIT (profondità dell'albero di ereditarietà) è la lunghezza massima dell'ereditarietà da una data classe alla sua classe "radice". Il NOC (numero di figli) di una classe è il numero di figli, o le sue immediate sottoclassi. CBO (accoppiamento tra classi di oggetti) rappresenta il numero di classi di oggetti a cui è accoppiata. RFC (risposta per una classe) è l'insieme dei metodi che saranno coinvolti in una risposta ad un messaggio ad un oggetto di questa classe. LCOM (mancanza di coesione nei metodi) misura la correlazione tra i metodi e le variabili di istanza locali di una classe.

Lack of Cohesion in Methods

- $M = \{M_1, M_2, M_3, \dots, M_m\}$ Set of methods of a class
- $A = \{A_1, A_2, A_3, \dots, A_a\}$ Set of attributes of a class
- $\mu(A_k)$ results the number of methods in M accessing A_k

$$LCOM^* = \frac{\left(\frac{1}{a} \sum_{k=1}^a \mu(A_k)\right) - m}{1 - m}$$

- Assumption that $\mu(A_k) \geq 1$
- Properties:
 - $LCOM^* = 0$, desired situation (...all methods accessing every attribute)
 - $LCOM^* = 1$, undesired situation (...only one method per attribute)

L'analisi dinamica a tempo di esecuzione si riferiscono e riflettono aspetti del comportamento del sistema (Copertura del codice, Bug per linea di codice, Utilizzo delle funzioni).

Nonostante l'uso di queste metriche è difficile trovare "intervalli di normalità" generali per i valori misurati delle singole metriche di qualità del software perché questi intervalli variano da progetto a progetto.

3.3 Design Pattern

I Design Pattern sono metodi riutilizzabili per risolvere problemi di progettazione comuni all'interno di un determinato contesto. In altre parole offrono una soluzione progettuale generale a problemi ricorrenti. Tali modelli sono formalizzati come best practice che lo sviluppatore può utilizzare per risolvere problemi comuni durante la progettazione dell'applicazione. I modelli non forniscono codice, ma soluzioni generali a problemi di progettazione e devono essere implementati nella propria applicazione specifica. *Design Patterns: Elements of Reusable Object-Oriented Software* del 1991 by GoF(Gang of Four) è uno libri capisaldo dei Design Patterns. Vediamo qualche definizione formale di design patterns:

- *describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice,* C. Alexander
- *the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts,* D.Riehle
- *both a thing and the instructions for making the thing,* J.Coplien

Il nome del pattern è un'etichetta che possiamo usare per descrivere un **problema** di design le sue **soluzioni** e le **conseguenze**:

1. Il problema descrive quando applicare il modello.
2. La soluzione descrive gli elementi che compongono il progetto, le loro relazioni, le responsabilità e le collaborazioni.
3. Le conseguenze sono i risultati e i compromessi dell'applicazione del modello.

La GoF, nel loro libro, ha dato uno schema formale per la descrizione di un pattern, basata sui seguenti punti:

- Pattern Name and Classification
- Intento, Che cosa fa il design pattern? Qual è la sua logica e il suo intento? Quale particolare problema di progettazione affronta?
- Also Known As, altri nomi noti del modello.
- Motivazione, uno scenario che illustra un problema di progettazione e il modo in cui le strutture di classi e oggetti del pattern risolvono il problema.
- Applicabilità
- Struttura, una rappresentazione grafica delle classi del modello.

- Partecipanti, le classi e/o gli oggetti che partecipano al modello di progettazione e le loro responsabilità
- Collaborazioni, come i partecipanti collaborano per svolgere le loro responsabilità
- Conseguenze, quali sono i compromessi e i risultati dell'utilizzo del modello?
- Implementazione, quali sono le insidie, i suggerimenti o le tecniche di cui dovete essere consapevoli quando implementate il modello? Ci sono problemi specifici di linguaggio di programmazione?
- Codice di esempio, frammenti di codice che illustrano come si potrebbe implementare il modello
- Usi comuni in sistemi reali
- Pattern correlati

I principali Design Patterns sono organizzati nel seguente modo.

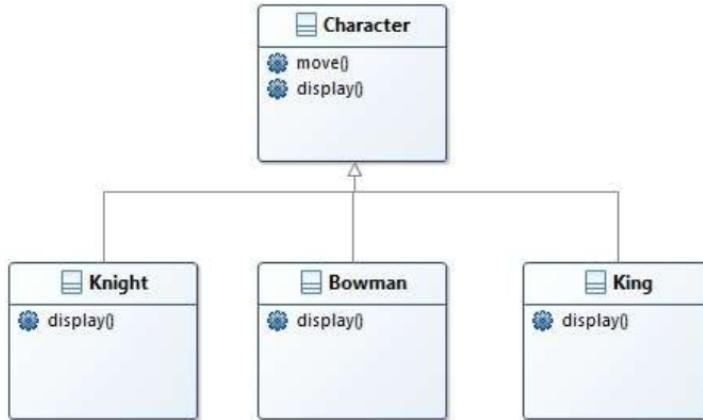
Scope		Purpose		
		Creational	Structural	Behavioral
		Class	Object	
		Factory Method	Adapter	Interpreter Template Method
		Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

La principale differenza esiste tra Class Pattern e Object Pattern:

- I **Class Pattern** descrivono il modo in cui le relazioni tra le classi sono definite tramite l'ereditarietà. Le relazioni nei modelli di classe sono stabilite in fase di compilazione.
- Gli **Object Pattern** descrivono le relazioni tra gli oggetti e sono definiti principalmente tramite composizione. Le relazioni nei modelli a oggetti sono tipicamente create in fase di esecuzione e sono più dinamiche e flessibili.

3.3.1 Strategy Pattern

Come per esplicitato in precedenza, per descrivere un Design Pattern dobbiamo partire dalla motivazione. Per capire quest'ultima analizziamo questo problema: la nostra azienda produce giochi di ruolo; il progettista del sistema ha adottato tecniche standard di modellazione OO per rappresentare i vari tipi di personaggi.



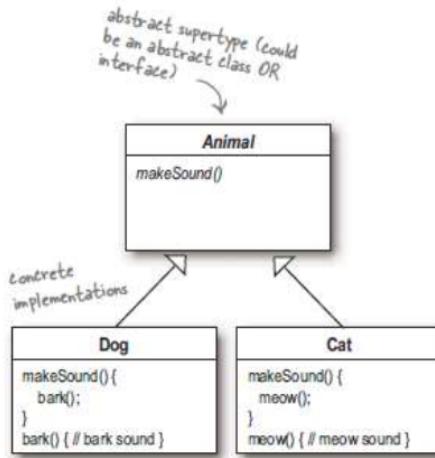
Dobbiamo introdurre il metodo fight nei personaggi (nell'oggetto Character) per permettere il combattimento. Ma non tutti personaggi combattono, per esempio i maghi lanciano incantesimi e gli unicorni non fanno nulla. Potremmo risolvere il problema facendo l'override dei metodi nelle classi specifiche che non hanno bisogno di "combattere". Ma cosa succede quando introduco una nuova classe che rappresenta i Chierici? e se devo aggiungere una classe per i Troll? Come è possibile implementare diversi "tipi di combattimento" e diversi tipi di "lancio di incantesimi"? E se dovessimo cambiare il "metodo di combattimento" in tutte le 50 classi che lo implementano? Che cosa succede se ci si rende conto che ogni 4-6 mesi si devono aggiungere nuove classi e modificare quelle presenti nel sistema?

In questo caso ci viene in aiuto lo Strategy Pattern basato seguenti principi:

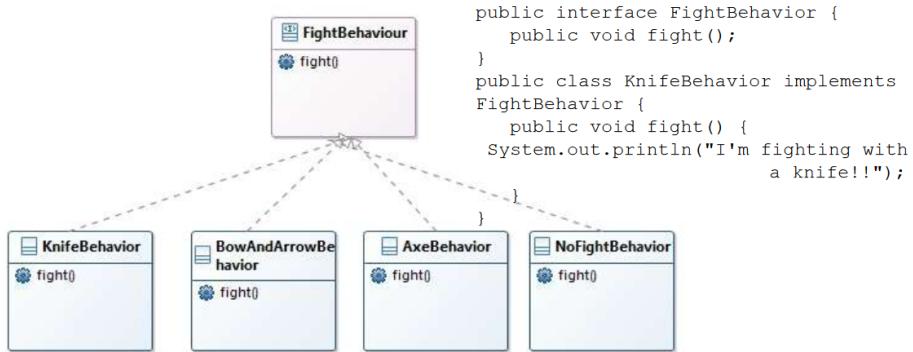
- *Identificate gli aspetti della vostra applicazione che variano e separateli da quelli che rimangono invariati; in particolar modo prendete le parti che variano ed "incapsulatele".* In questo modo separo la parte del codice che cambia dalla parte del codice che non cambia nel tempo (o cambia poco).
- *Programma un'interfaccia, non un'implementazione.*

```
//implementation
Dog d = new Dog();
d.bark();
//interface
Animal animal = new Dog();
```

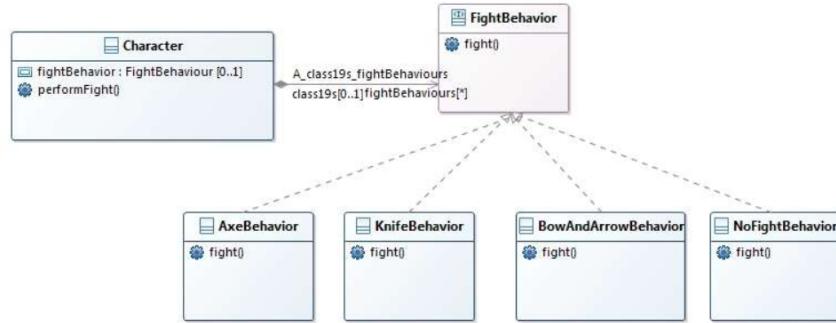
```
animal.makeSound();
```



Quindi introduciamo un'interfaccia che descrive in modo generale il combattimento (FightBehavior) e delle classi che implementando l'interfaccia FightBehavior descrivono un particolare tipo di combattimento (Knife, Axe, BowAndArrow).



Il metodo `fight()` implementa il compito specifico per il comportamento specifico. Con questo design, altri tipi di oggetti possono riutilizzare i comportamenti, perché non sono più nascosti nelle classi dei personaggi. Inoltre possiamo aggiungere nuovi comportamenti senza modificare nessuna delle nostre classi di comportamento esistenti o una qualsiasi delle classi di personaggi che usano i comportamenti di lotta.



Nel definire il Character non ci importa dell'implementazione specifica di un determinato comportamento perché il tutto è delegato all'interfaccia

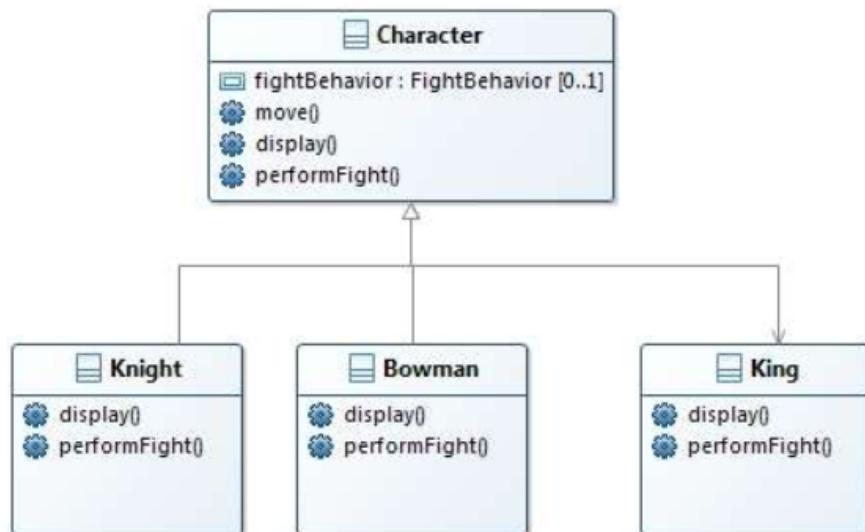
```

public class Character{
    FightBehavior fightBehavior;

    public void performFight() {
        fightBehavior.fight();
    }
}

```

Per implementare uno specifico personaggio procediamo nel seguente modo (estendendo la classe Character)



```

public class King extends Character {
    public King() {
        fightBehavior = new KnifeBehavior();
    }
    public void display() {
        System.out.println("I'm a king");
    }
}

public class Simulator{
    public static void main(String[] args) {
        Character lion= new King();
        lion.performFight();
    }
}

```

In questo modo abbiamo reso il tutto più flessibile ma notiamo che un personaggio può performare un solo tipo di combattimento perché il comportamento viene inizializzato staticamente nel costruttore. Possiamo aumentare la flessibilità di questo codice, permettendo di cambiare il comportamento del personaggio real time, nel seguente modo:

```

public abstract class Character{
    FightBehavior fightBehavior;

    //Add setFightBehavior
    public void setFightBehavior(FightBehavior fb){
        fightBehavior= fb;
    }
}

public class King extends Character {
    public King() {
        fightBehavior = new KnifeBehavior();
    }
}

public class Simulator{
    public static void main(String[] args) {
        Character lion= new King();
        lion.performFight();
        lion.setFightBehavior(new AxeBehavior());
        lion.performFight()
    }
}

```

Invece di pensare ai comportamenti dei personaggi come ad un insieme di piccoli comportamenti, inizieremo a pensarli come una famiglia di algoritmi.

Nel simulatore, gli algoritmi rappresentano le azioni di un personaggio (diversi modi di combattere), ma potremmo usare le stesse tecniche per altri scenari (movimento, ricarica, ecc.). Quando si mettono insieme due classi, come abbiamo fatto, si usa la composizione (relazione HAS-A invece di IS-A). Infatti un altro principio base dello Strategy Pattern è *Favorire la composizione rispetto all'ereditarietà*.

Continuiamo ora la nostra descrizione formale dello Strategy Pattern:

- **Intento**

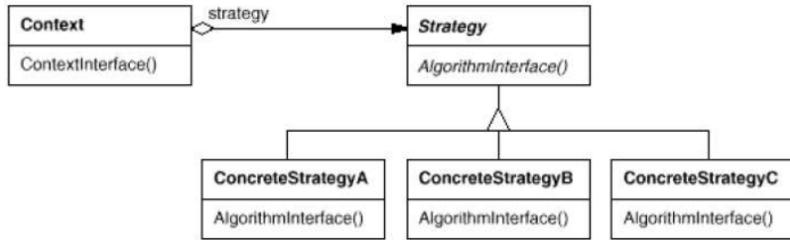
Definire una famiglia di algoritmi, incapsulare ciascuno di essi e renderli intercambiabili. La strategia consente all'algoritmo di variare in modo indipendente dai client che lo utilizzano.

- Also Known As Policy

- **Applicabilità**

Quando molte classi correlate differiscono solo per il loro comportamento. Lo Strategy Pattern fornisce un modo per configurare una classe con uno dei tanti comportamenti.

- **Struttura**



- **Partecipanti**

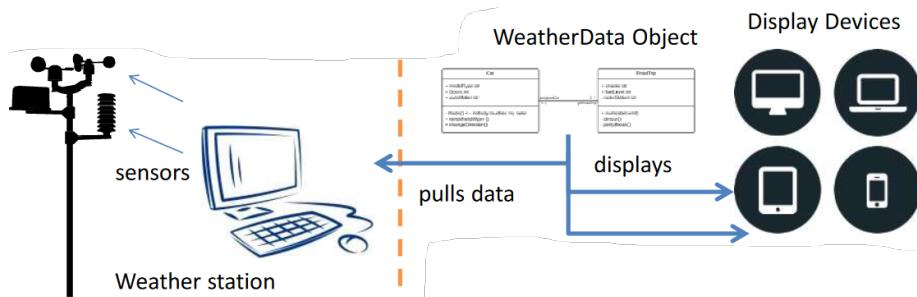
Strategy dichiara un'interfaccia comune a tutti gli algoritmi supportati; **Concrete Strategy** implementa l'algoritmo utilizzando l'interfaccia **Strategy**; **Context** configurato con un oggetto **Concrete Strategy** e mantiene un riferimento ad un oggetto **Strategy**.

- **Conseguenze**

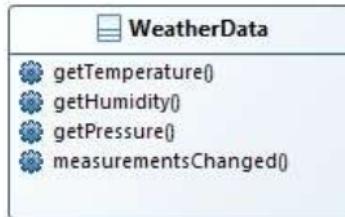
- Crea una famiglia di algoritmi correlati.
- È un'alternativa alle sottoclassi.
- Elimina le dichiarazioni condizionali.
- Costi computazionali elevati per la comunicazione tra **Strategy** e **Context**.
- Aumenta il numero di oggetti.

3.3.2 Observer Pattern

Il pattern Observer è uno dei pattern più utilizzati nello sviluppo di applicazioni Java. Si tratta di un pattern che permette agli oggetti di essere informati quando accade qualcosa di cui potrebbero preoccuparsi. Per capire la motivazione di questo pattern introduciamo l'esempio della Wheather Station, in cui gli stessi dati devono essere rappresentati in modi diversi su dispositivi diversi.



L'oggetto WeatherData parla con la stazione meteo e viene utilizzato per aggiornare tre dispositivi di visualizzazione (condizioni attuali, statistiche meteo e previsioni). Il metodo measurementsChanged() viene richiamato ogni volta che sono disponibili nuovi dati di misurazione meteorologica e quindi abbiamo bisogno di implementare tre elementi di visualizzazione che utilizzano i dati meteo e rendere possibile aggiungere nuove visualizzazioni.

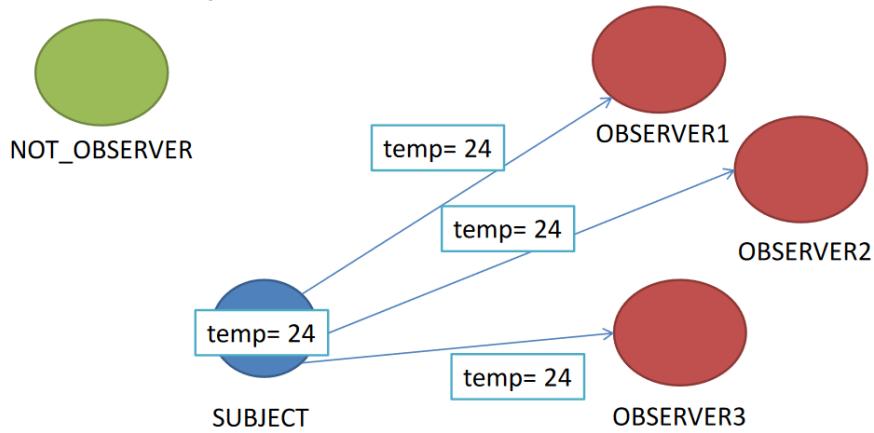


```
public class WeatherData {
    // instance variable declarations
    public void measurementsChanged() {
        float temp = getTemperature();
        float humidity = getHumidity();
        float pressure = getPressure();

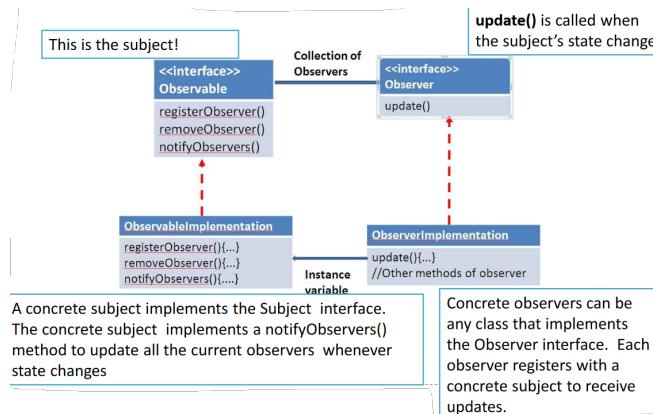
        Display1.update(temp, humidity, pressure);
        Display2.update(temp, humidity, pressure);
        Display3.update(temp, humidity, pressure);
    }
}
```

```
// other WeatherData methods here
}
```

Ma in questo modo stiamo codificando l'implementazione concreta e non c'è modo di cambiarla senza cambiare l'applicazione software. Dobbiamo trovare un modo per incapsulare i display e per far ciò introduciamo il meccanismo di publisher-subscriber. Pensate a un giornale d'informazione: i giornali sono online e pubblicano articoli, l'utente si abbona ad un determinato giornale e quando esce una nuova edizione gli viene consegnata; si disdice l'abbonamento quando non si vogliono più i giornali, e questi smettono di essere consegnati (le persone possono abbonarsi/disabbonarsi dinamicamente).



L'Observer Pattern definisce una dipendenza uno-a-molti tra gli oggetti, in modo che quando un oggetto cambia stato, tutti i suoi suoi dipendenti vengono notificati e aggiornati automaticamente. Il Subject fornisce i dati a tutti gli osservatori che hanno deciso di ricevere dati.



Il principio principale di questo pattern è quello di cercare di progettare degli oggetti che interagiscono tra loro ma poco accoppiati. Infatti l'Observer Pattern fornisce un design di oggetti in cui i soggetti e gli osservatori sono accoppiati in modo lasco; l'unica cosa che l'oggetto conosce di un osservatore è che implementa una certa interfaccia (l'interfaccia Observer). Possiamo aggiungere, sostituire e rimuovere nuovi osservatori in qualsiasi momento, non è mai necessario modificare il soggetto per aggiungere nuovi tipi di osservatori e possiamo riutilizzare soggetti o osservatori indipendentemente l'uno dall'altro. Infine le modifiche apportate ad un soggetto o ad un osservatore non influiscono sull'altro. Vediamo come applicare questo pattern sul problema della Wheater Station.



```

//Subject
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void notifyObservers();
}
//Observer
public interface Observer {
    public void update(float temp, float humidity, float pressure);
}
public interface DisplayElement {
    public void display();
}
  
```

La Weather Station sarà un'implementazione concreta dell'interfaccia Subject e avrà al suo interno una collezioni di Observer (display).

```

public class WeatherData implements Subject {
    private ArrayList<Observer> observers;
    private float temperature;
    private float humidity;
    private float pressure;

    public WeatherData() {
        observers = new ArrayList<Observer>();
    }
    public void registerObserver(Observer o) {
  
```

```

        observers.add(o);
    }
    public void removeObserver(Observer o) {
        int i = observers.indexOf(o);
        if (i >= 0) {
            observers.remove(i);
        }
    }
    public void notifyObservers() {
        for (Observer observer : observers)
        {
            observer.update(temperature, humidity,
                           pressure);
        }
    }
    public void measurementsChanged() {
        notifyObservers();
    }
    public void setMeasurements(float temperature, float
                                humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        this.pressure = pressure;
        measurementsChanged();
    }
    // other WeatherData methods here
}

```

Ogni display/Ogni tipo di visualizzazione dei dati sarà un Observer.

```

public class Display1 implements Observer,
    DisplayElement {
    private float temperature;
    private float humidity;
    private Subject weatherData;
    public Display1(Subject weatherData) {
        this.weatherData = weatherData;
        weatherData.registerObserver(this);
    }
    public void update(float temperature, float
                      humidity, float pressure) {
        this.temperature = temperature;
        this.humidity = humidity;
        display();
    }
    public void display() {
        System.out.println("Current conditions: " +

```

```

        temperature + "F degrees and " + humidity +
        "% humidity");
    }
}

public class WeatherStation {
    public static void main(String[] args) {
        WeatherData weatherData = new WeatherData();
        Display1 currentDisplay = new
            Display1(weatherData);

        weatherData.setMeasurements(80, 65, 30.4f);
        weatherData.setMeasurements(82, 70, 29.2f);
        weatherData.setMeasurements(78, 90, 29.2f);
    }
}

```

Come vediamo l'oggetto Weather Data è indipendente da Display1. Il trasferimento dei dati nell'Observer Pattern è basato sul meccanismo di **push/pull**:

- **Push**, l'osservatore ottiene i dati esatti che desidera. La responsabilità è tutta del subject, gli observer devono solo assicurarsi che abbiano inserito il codice richiesto nei loro metodi di aggiornamento. Il vantaggio principale è l'ulteriore riduzione dell'accoppiamento tra l'osservatore e il soggetto ma si ha meno flessibilità (in caso di 1000 osservatori con la maggior parte di essi che richiedono diversi tipi di dati si potrebbe avere molto confusione nel codice). Dovrebbe essere utilizzato quando ci sono al massimo 2-3 tipi diversi di osservatori (tipi diversi significa che l'osservatore richiede dati diversi) o tutti gli osservatori richiedono lo stesso tipo di dati.
- **Pull**, l'osservatore ottiene i dati encapsulati in qualche oggetto (per lo più Observable) e deve estrarre da esso i dati richiesti. È responsabilità del subject notificare a tutti gli osservatori che qualcosa è stato modificato (condivide l'intero oggetto che ha subito le modifiche). Saranno poi gli Observer ad estrarre i dettagli richiesti. In questo modo abbiamo maggiore flessibilità, infatti ogni osservatore può decidere autonomamente cosa estrarre, senza dipendere dall'invio delle informazioni corrette ma abbiamo un maggiore grado di accoppiamento (gli observer devono conoscere alcune cose del soggetto per poter interrogare le giuste informazioni). Dovrebbe essere utilizzato quando ci sono al massimo 2-3 tipi diversi di osservatori

Di seguito la descrizione formale dell'Observer Pattern:

- **Intento**

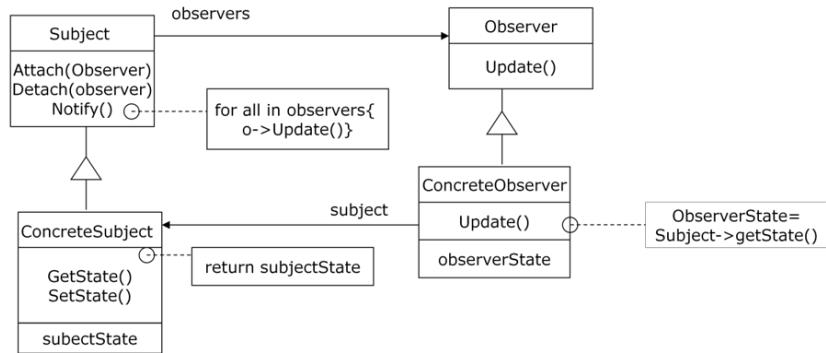
Definire una dipendenza uno-a-molti tra gli oggetti, in modo che quando un oggetto cambia stato, tutti i suoi dipendenti vengono notificati e aggiornati automaticamente.

- Also Known As Dependents, Publish-Subscribe

• Applicabilità

Quando un'astrazione ha due aspetti, uno dipendente dall'altro. Quando la modifica di un oggetto richiede la modifica di altri e non si sa quanti oggetti devono essere modificati. Quando un oggetto deve essere in grado di notificare altri oggetti senza fare ipotesi su chi siano questi oggetti.

• Struttura



• Partecipanti

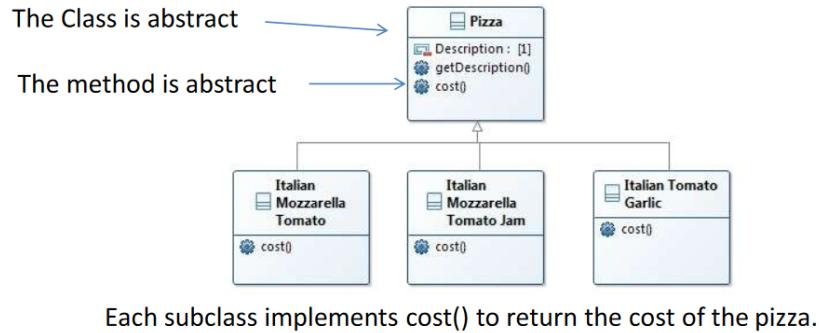
Subject fornisce un'interfaccia per attaccare e staccare gli oggetti **Observer**. **Observer** definisce un'interfaccia di aggiornamento per gli oggetti che devono essere notificati delle modifiche. **ConcreteSubject** memorizza lo stato di interesse per gli oggetti **ConcreteObserver** e invia una notifica ai suoi **Observer** quando il suo stato cambia. **ConcreteObserver** implementa l'aggiornamento dell'**Observer**.

• Conseguenze

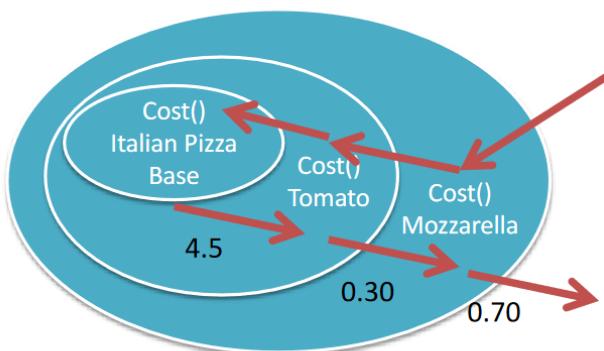
- Accoppiamento astratto tra Subject e Observer.
- Supporto per la comunicazione broadcast.
- Update inaspettati.

3.3.3 Decorator Pattern

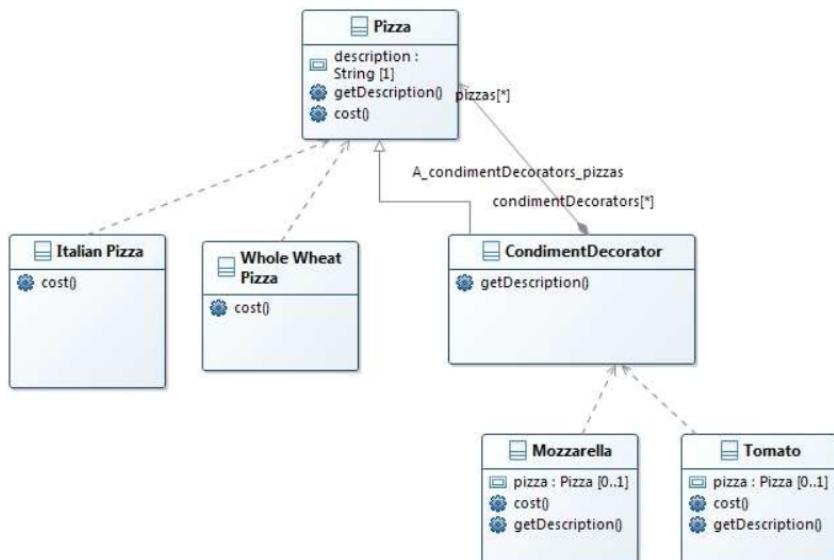
Con le tecniche di decorazione, sarete in grado di assegnare agli oggetti nuove responsabilità senza apportare modifiche al codice delle classi sottostanti. Per capire la motivazione vediamo un semplicissimo esempio: Dovete modellare le "pizze" vendute da un'azienda; la pizza è formata da una base (italiana, teglia, crosta farcita, integrale, ...) e da un condimento (mozzarella, salame piccante, funghi, olive, ananas). I clienti possono combinare basi e condimenti come preferiscono. Il prezzo finale della pizza dipende dalla base e dai condimenti selezionati. Una prima implementazione potrebbe essere la seguente.



Ma cosa succede se abbiamo una grande quantità di combinazioni? Cosa succede quando il prezzo della mozzarella aumenta? E se aggiungiamo un nuovo condimento? Dobbiamo creare una sottoclasse per ogni tipo di base ma in questo modo le modifiche ai prezzi dei condimenti ci costringeranno a modificare il codice esistente ed i nuovi condimenti ci costringeranno ad aggiungere nuovi metodi e a modificare il metodo del costo nella superclasse. Inoltre alcune sottoclassi erediteranno funzioni inutili (la sottoclasse delle pizze dolci erediterà comunque metodi come hasMozzarella()). Il Decorator Pattern si basa sull'Open-Closed Principle: le classi dovrebbero essere aperte per l'ampliamento, ma chiuse per la modifica. Si parte da una base e la si "decora" a tempo di esecuzione con dei condimenti.



Il pattern Decorator attribuisce responsabilità aggiuntive ad un oggetto in modo dinamico. I decoratori forniscono un'alternativa flessibile alla sottoclasse per estendere le proprie funzionalità.



```
public abstract class Pizza{
    String description = "Unknown Pizza";
    public String getDescription() {
        return description;
    }
    public abstract double cost();
}
//Decorator
public abstract class CondimentDecorator extends Pizza{
    Pizza pizza;
    public abstract String getDescription();
}

//base
public class Italian extends Pizza{
    public Italian() {
        description = "Italian base";
    }
    public double cost() {
        return 4.50;
    }
}
```

```

public class WholeWheat extends Pizza{
    public WholeWheat () {
        description = " WholeWheat base";
    }
    public double cost() {
        return 5.30;
    }
}

//Condimenti
public class Tomato extends CondimentDecorator {
    Pizza pizza;
    public Tomato(Pizza pizza) {
        this.pizza = pizza;
    }
    public String getDescription() {
        return pizza.getDescription() + ", Tomato";
    }
    public double cost() {
        return pizza.cost() + .20;
    }
}

public class Mozzarella extends CondimentDecorator {
    Pizza pizza;
    public Mozzarella (Pizza pizza) {
        this.pizza = pizza;
    }
    public String getDescription() {
        return pizza.getDescription() + ", Mozzarella";
    }
    public double cost() {
        return pizza.cost() + .70;
    }
}

```

Infine aggiungiamo il main.

```

public class PizzaTest{
    public static void main(String args[]) {
        Pizza pizza = new Italian();
        System.out.println(pizza.getDescription() + " euro" +pizza.cost());
        Pizza pizza2 = new Italian();
        pizza2 = new Mozzarella(pizza2);
        pizza2 = new Mozzarella(pizza2);
        pizza2 = new Tomato(pizza2);
    }
}

```

```

        System.out.println(pizza2.getDescription() + " "
                           + "euro" + pizza2.cost());
    }
}

```

Di seguito la descrizione formale del Decorator Pattern:

- **Intento**

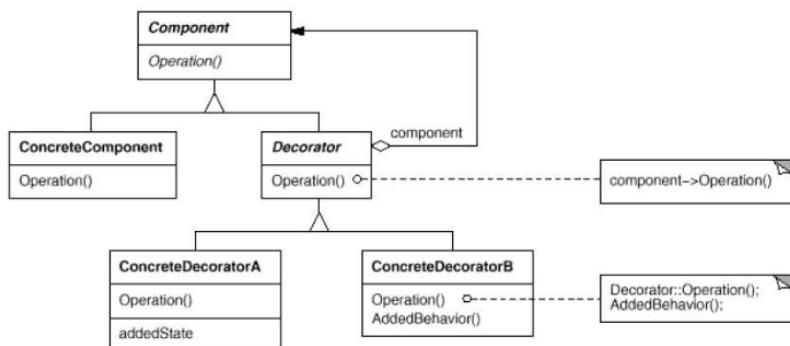
Collegare responsabilità aggiuntive a un oggetto in modo dinamico. I decoratori forniscono un'alternativa flessibile alla sottoclasse per estendere le funzionalità.

- Also Known As Wrapper

- **Applicabilità**

Per aggiungere responsabilità a singoli oggetti in modo dinamico e trasparente, ossia senza influenzare gli altri oggetti e quando l'estensione tramite sottoclassi non è praticabile.

- **Struttura**



- **Partecipanti**

Component definisce l'interfaccia per gli oggetti a cui possono essere aggiunte responsabilità. **ConcreteComponent** definisce dinamicamente un oggetto a cui possono essere aggiunte responsabilità. **Decorator** mantiene un riferimento ad un oggetto Component e definisce un'interfaccia conforme all'interfaccia di Component. **ConcreteDecorator** aggiunge responsabilità al componente.

- **Conseguenze**

- Maggiore flessibilità rispetto all'ereditarietà statica
- Evita le classi cariche di caratteristiche in alto nella gerarchia
- Definisce tanti piccoli oggetti.

3.3.4 Factory Pattern

Quando si usa new, si istanzia sicuramente una classe concreta, quindi si tratta di un'implementazione, non di un'interfaccia. Quando arriverà il momento di apportare modifiche o estensioni, si dovrà riaprire il codice ed esaminare cosa deve essere aggiunto (o eliminato) (non rispettando il principio di Open-Close) e spesso questo tipo di codice finisce in diverse parti dell'applicazione, rendendo più difficile la manutenzione e gli aggiornamenti più difficili e a rischio di errori. Tecnicamente non c'è nulla di sbagliato nel nuovo, infatti è una parte fondamentale di Java; il problema è gestire il cambiamento e il modo in cui il cambiamento influisce sul nostro programma. E quindi cosa si può fare? Seguiamo i principi di design, in particolar modo "identificare gli aspetti che variano e separarli da ciò che rimane invariato". Supponiamo, per esempio, di avere una pizzeria.

```
Pizza orderPizza() {
    Pizza pizza = new Pizza();
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

Visto che non si diventa ricchi con un solo tipo di pizza, vogliamo estendere l'oggetto pizza con tanti tipi di pizza.

```
Pizza orderPizza(String type) {
    Pizza pizza;
    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("greek")){
        pizza = new GreekPizza();
    } else if (type.equals("pepperoni") {
        pizza = new PepperoniPizza();
    }
    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();
    return pizza;
}
```

In base al tipo di pizza, la classe concreta corretta viene istanziata (ogni tipo di pizza deve implementare l'interfaccia). Ma se la pizzeria cambia la sua offerta di pizze, dobbiamo entrare in questo codice e modificarlo (anche notevoli volte). Per risolvere questo problema tutto quello che dobbiamo fare è prendere il codice di creazione della pizza e spostarlo in un altro oggetto che si occuperà solo di gestire i dettagli della creazione dell'oggetto. Una volta che ab-

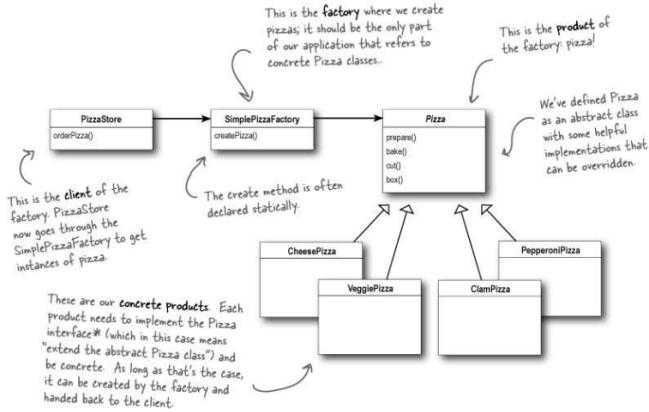
biamo una SimplePizzaFactory, il nostro metodo orderPizza() diventa un client di quell'oggetto.

```
public class SimplePizzaFactory {  
    public Pizza createPizza(String type) {  
        Pizza pizza = null;  
        if (type.equals("cheese")) {  
            pizza = new CheesePizza();  
        } else if (type.equals("peperoni")) {  
            pizza = new PeperoniPizza();  
        } else if (type.equals("clam")) {  
            pizza = new ClamPizza();  
        } else if (type.equals("veggie")) {  
            pizza = new VeggiePizza();  
        }  
        return pizza;  
    }  
}
```

Incapsulando la creazione della pizza in un'unica classe, ora abbiamo un solo posto dove apportare modifiche quando l'implementazione cambia. Definire un semplice factory come metodo statico è una tecnica comune che viene chiamata **static factory**. In questo caso non è possibile creare una sottoclasse e modificare il comportamento del metodo di creazione ma essendo static si può chiamare senza instanziare l'oggetto factory.

```
public class PizzaStore {  
    SimplePizzaFactory factory;  
    public PizzaStore(SimplePizzaFactory factory) {  
        this.factory = factory;  
    }  
    public Pizza orderPizza(String type) {  
        Pizza pizza;  
        pizza = factory.createPizza(type);  
        pizza.prepare();  
        pizza.bake();  
        pizza.cut();  
        pizza.box();  
        return pizza;  
    }  
    // other methods here  
}
```

Il Simple Factory non è in realtà un designm a è più che altro un idioma di programmazione (quindi da non confondere con il "Factory Pattern").



Adesso si vuole estendere il codice per implementarlo in diversi punti vendita che realizzano diversi tipi di pizza (in diverse regioni, si producono diversi tipi di pizze). Grazie lo static factory questo è facile da gestire perché possiamo creare diverse fabbriche, per gestire ogni regione/città.

```

NaplesPizzaFactory naFactory = new NaplesPizzaFactory();
PizzaStore naStore = new PizzaStore(naFactory);
naStore.orderPizza("Veggie");

MilanPizzaFactory milanFactory = new MilanPizzaFactory();
PizzaStore milanStore = new PizzaStore(milanFactory);
milanStore.orderPizza("Veggie");

```

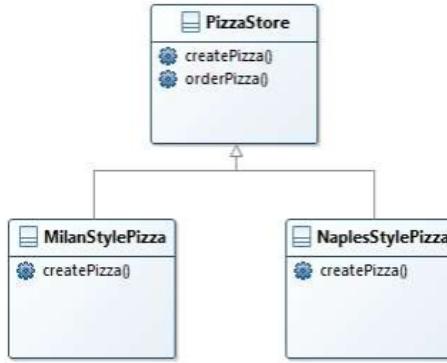
Il pizza store sarà quindi gestito nel seguente modo.

```

public abstract class PizzaStore {
    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }

    abstract Pizza createPizza(String type);
}

```



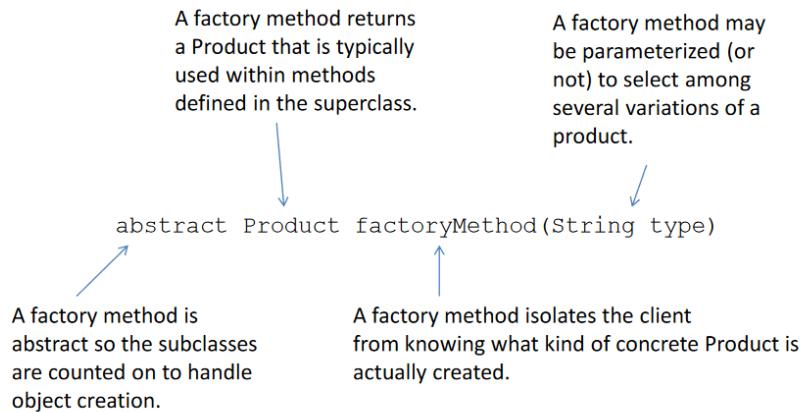
Ogni sottoclasse fornisce un'implementazione del metodo `createPizza()` sovrascrivendo il metodo astratto `createPizza()` di `PizzaStore`, mentre tutte le sottoclassi utilizzano il metodo `orderPizza()` definito in `PizzaStore`. Il metodo `orderPizza()` però non ha idea di quale sottoclasse stia effettivamente eseguendo il codice e preparando le pizze (è disaccoppiato). Quando `orderPizza()` chiama `createPizza()`, una delle vostre sottoclassi verrà chiamata in azione, la quale deciderà il tipo di pizza in base alla scelta della pizzeria da cui si ordina `MilanStylePizza` o `NapoliStylePizza`. Per fare ciò per ogni tipo di Pizza creiamo lo stile Napoli attraverso il **Factory Method**.

```

public class NaplesStylePizzaStore extends PizzaStore {
    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NaplesStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NaplesStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NaplesStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NaplesStylePepperoniPizza();
        } else return null;
    }
}

```

Ora il metodo `orderPizza()` della superclasse non ha la minima idea di quale Pizza stiamo creando. Un metodo factory gestisce la creazione di un oggetto e lo incapsula in una sottoclasse. Questo disaccoppia il codice del client nella superclasse dal codice di creazione dell'oggetto nella sottoclasse.



Con il Factory Method il procedimento per ordinare una pizza è il seguente:

1. Creare un'istanza del PizzaStore desiderato.

```
PizzaStore naplesPizzaStore = new NaplesPizzaStore();
```

2. È possibile richiamare il metodo orderPizza() e passare il tipo di pizza che si desidera

```
naplesPizzaStore.orderPizza("cheese");
```

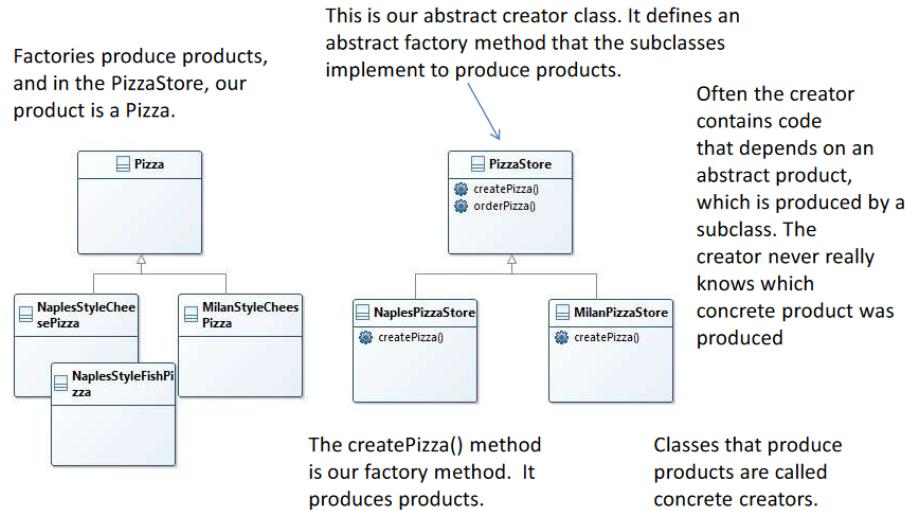
3. Esiste un metodo specifico per ogni sottoclasse di PizzaStore. La Pizza viene restituita al metodo orderPizza().

```
Pizza = createPizza("formaggio");
```

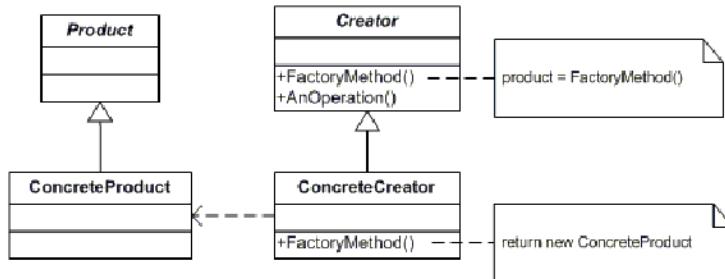
4. Il metodo orderPizza() non ha idea di che tipo di pizza sia stata creata. ma sa che si tratta di una pizza e la prepara, la inforna, la taglia e la inscatola. la inscatola.

```
pizza.prepare();
pizza.bake();
pizza.cut();
pizza.box();
```

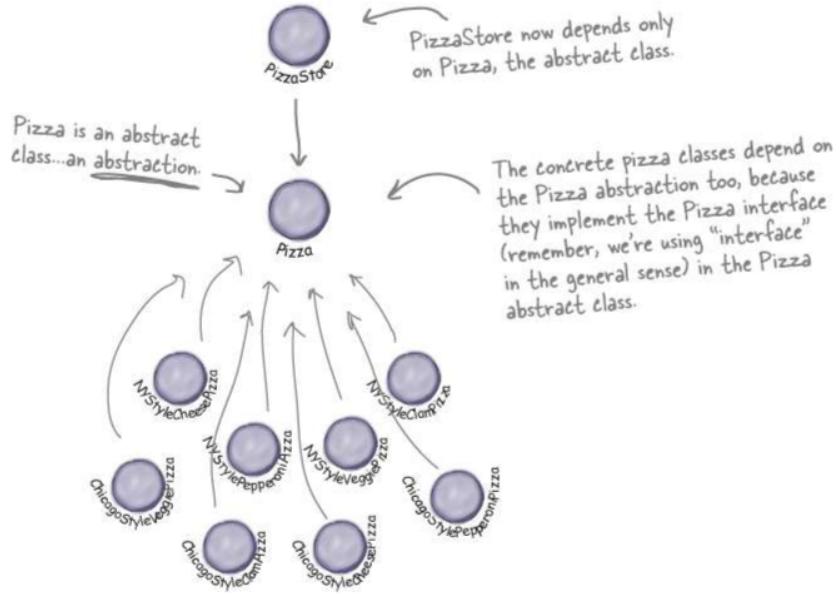
La rappresentazione finale del pattern è la seguente.



Lo schema del metodo Factory definisce un'interfaccia per la creazione di un oggetto ma lascia che siano le sottoclassi a decidere quale classe istanziare. Con "decidere" non intendiamo che il pattern permetta alle sottoclassi stesse di decidere in fase di esecuzione, ma intendiamo che la classe creatrice è scritta senza conoscenza dei prodotti effettivi che saranno creati, che è deciso dalla scelta della sottoclasse che viene utilizzata.



Attraverso il factory pattern rispettiamo anche il principio di Dependecy Inversion che ci dice che i nostri componenti di alto livello non devono dipendere dai nostri componenti di basso livello; piuttosto, entrambi dovrebbero dipendere dalle astrazioni.



Per evitare progetti OO che violano il Principio di Inversione di Dipendenza
dobbiamo rispettare le seguenti regole:

- Nessuna variabile dovrebbe contenere un riferimento a una classe concreta.
- Nessuna classe dovrebbe derivare da una classe concreta.
- Nessun metodo dovrebbe sovrascrivere un metodo implementato di una qualsiasi delle sue classi base.
- Nessun metodo deve sovrascrivere un metodo implementato di una qualsiasi delle sue classi base(questo é possibile farlo utilizzando lo Strategy Pattern)

Torna all'esempio della pizzeria dobbiamo garantire la coerenza degli ingredienti.
Ogni famiglia (Store) è composta da un tipo di pasta, di salsa e di formaggio e per far ciò potremmo costruire un factory per creare gli ingredienti.

```
public interface PizzaIngredientFactory {
    public Dough createDough();
    public Sauce createSauce();
    public Cheese createCheese();
    public Veggies[] createVeggies();
    public Pepperoni createPepperoni();
    public Clams createClam();
```

```
// Lots of new classes here, one per ingredient
}
```

Costruiamo un Factory per ogni regione; si creerà una sottoclasse di PizzaIngredientFactory che implementa ogni metodo di creazione (queste classi possono essere condivise tra le regioni, se necessario). Poi dobbiamo ancora collegare tutto questo, inserendo le nostre nuove factory di ingredienti nel vecchio codice di PizzaStore.

```
public class NaplesPizzaIngredientFactory implements
    PizzaIngredientFactory {
    public Dough createDough() {
        return new ThinCrustDough();
    }
    public Sauce createSauce() {
        return new MarinaraSauce();
    }
    public Cheese createCheese() {
        return new MozzarellaCheese();
    }
    public Veggies[] createVeggies() {
        Veggies veggies[] = { new Garlic(), new Onion(),
            new Mushroom(), new
            RedPepper() };
        return veggies;
    }
    public Pepperoni createPepperoni() {
        return new SlicedPepperoni();
    }
    public Clams createClam() {
        return new FreshClams();
    }
}
```

Ora abbiamo reso il metodo prepare() nella classe pizza astratto. È qui che raccolgeremo gli gli ingredienti necessari per la pizza, che naturalmente provengono dalla fabbrica degli ingredienti.

```
public class CheesePizza extends Pizza {
    PizzaIngredientFactory ingredientFactory;
    public CheesePizza(PizzaIngredientFactory
        ingredientFactory) {
        this.ingredientFactory = ingredientFactory;
    }
    void prepare() {
        System.out.println("Preparing " + name);
        dough = ingredientFactory.createDough();
        sauce = ingredientFactory.createSauce();
```

```

        cheese = ingredientFactory.createCheese();
    }
}

public class NaplesPizzaStore extends PizzaStore {
    protected Pizza createPizza(String item) {
        Pizza pizza = null;
        PizzaIngredientFactory ingredientFactory =
            new NaplesPizzaIngredientFactory();
        if (item.equals("cheese")) {
            pizza = new CheesePizza(ingredientFactory);
            pizza.setName("Naples Style Cheese Pizza");
        } else if (item.equals("clam")) {
            pizza = new ClamPizza(ingredientFactory);
            pizza.setName("Naples Style Clam Pizza");
        } else if (item.equals(".")) {
            ...
        }
        return pizza;
    }
}

```

Abbiamo fornito un mezzo per creare una famiglia di ingredienti per le pizze introducendo un nuovo tipo di factory, chiamato **Abstract Factory**. Un Abstract Factory fornisce un'interfaccia per creare una famiglia di prodotti; scrivendo codice che utilizza questa interfaccia, disaccoppiamo il nostro codice dalla fabbrica che crea i prodotti. Questo ci permette di implementare una varietà di fabbriche che producono prodotti destinati a contesti diversi, come regioni diverse, sistemi operativi diversi o a look and feel diversi. Adesso con l'Abstract Factory il procedimento per ordinare una pizza è leggermente cambiato:

1. Per prima cosa abbiamo bisogno di un PizzaStore Napoli

```
PizzaStore naplesPizzaStore = new NaplesPizzaStore();
```

2. Ora che abbiamo un negozio, possiamo prendere un ordine

```
naplesPizzaStore.orderPizza("cheese");
```

3. Il metodo orderPizza() richiama prima il metodo createPizza():

```
Pizza pizza = createPizza("cheese");
```

4. Quando viene richiamato il metodo createPizza(), entra in gioco la nostra fabbrica di ingredienti

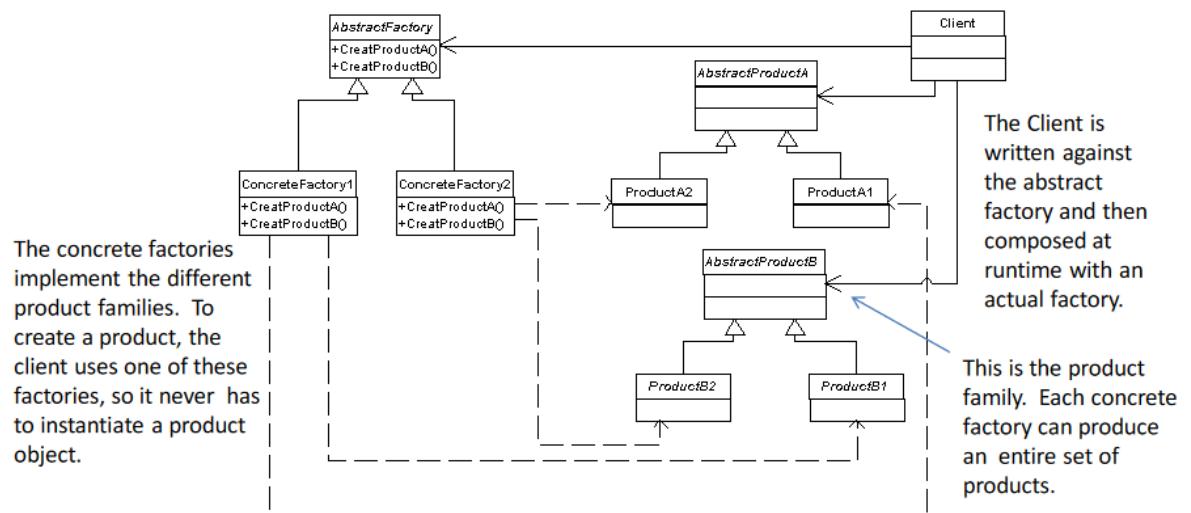
```
Pizza pizza = new
    CheesePizza(naplesIngredientFactory);
```

5. Poi dobbiamo preparare la pizza. Una volta richiamato il metodo `prepare()`, viene chiesto alla fabbrica di preparare gli ingredienti

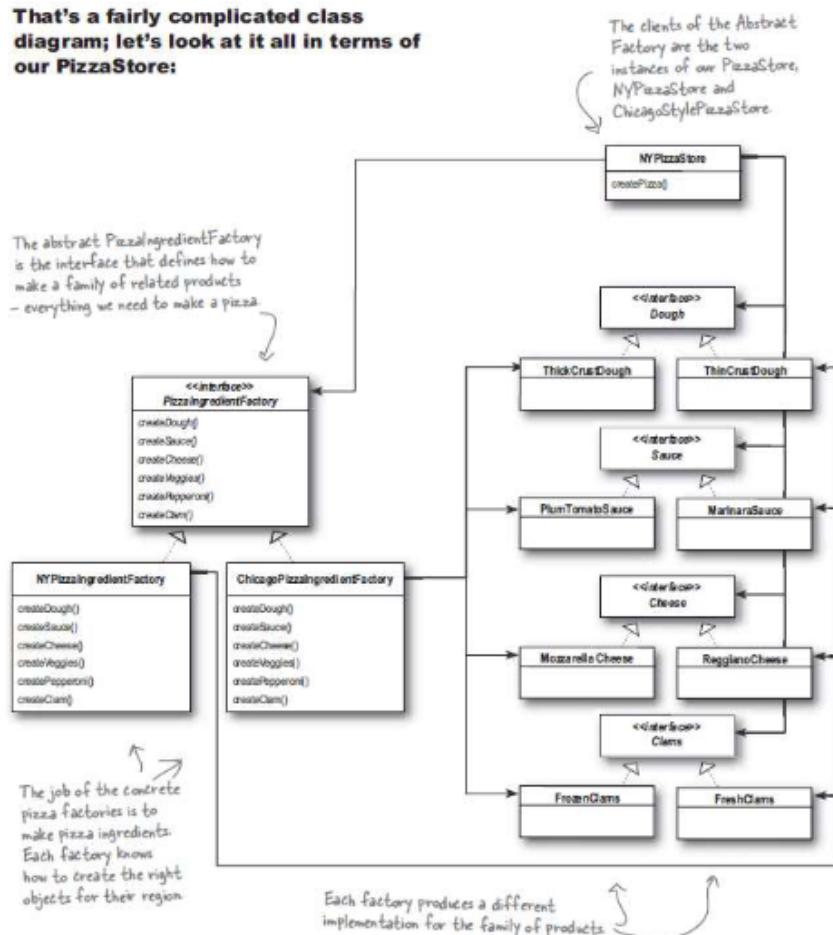
```
void prepare() {
    dough = factory.createDough();
    sauce = factory.createSauce();
    cheese = factory.createCheese();
}
```

6. Infine, abbiamo la pizza preparata in mano e il metodo `orderPizza()` inforna, taglia e inscatola la pizza.

Il pattern Abstract Factory fornisce un'interfaccia per la creazione di famiglie di oggetti correlati o dipendenti senza specificare le loro classi concrete.



That's a fairly complicated class diagram; let's look at it all in terms of our PizzaStore:



La differenza principale tra i due Factory Pattern è che con il modello Abstract Factory una classe delega la responsabilità dell'istanziazione dell'oggetto a un altro oggetto tramite la composizione, mentre il pattern Factory Method utilizza l'ereditarietà e si affida a una sottoclasse per gestire l'istanziazione dell'oggetto desiderato.

3.3.5 Singleton Pattern

Il modello Singleton assicura che una classe abbia una sola istanza e fornisce un punto di accesso globale ad essa. Crea oggetti unici per i quali esiste una sola istanza. Nonostante la sua semplicità dal punto di vista del design della classe, la sua implementazione può presentare problemi critici. Come possiamo rendere una classe istanziabile una sola volta? Potremmo provare a mettere il

costrutture di una classe pubblica privato.

```
public MyClass {  
    private MyClass() {}  
}
```

Ma in questo modo non possiamo instanziare la classe perché non possiamo accedere al metodo costruttore.

Proviamo invece ad usare una variabile statica per istanziare una classe.

```
public MyClass {  
    private MyClass() {}  
    public static MyClass getInstance() {  
        return new MyClass();  
    }  
}  
  
MyClass.getInstance();
```

In questo modo però potremmo comunque istanziare più oggetti di una singola classe. Come posso modificare il codice in modo che venga istanziato un solo oggetto? Possiamo usare il **Singleton Pattern**.

```
public class Singleton {  
    private static Singleton uniqueInstance;  
    // other useful instance variables here  
    private Singleton() {}  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
    // other useful methods here  
}
```

Vediamo subito un esempio per chiarire il concetto: dobbiamo gestire un controllore per il calderone utilizzato nella produzione di Parmigiano Reggiano.

```
public class Cauldron{  
    private boolean empty;  
    private boolean boiled;  
    public Cauldron() {  
        empty = true;  
        boiled = false;  
    }  
    public void fill() {  
        if (isEmpty()) {  
            empty = false;
```

```

        boiled = false;
        // fill the cauldron
        with a milk mixture
    }
}
public void removed() {
    if (!isEmpty() && isBoiled()) {
        // remove the boiled milk
        empty = true;
    }
}
public void boil() {
    if (!isEmpty() && !isBoiled()) {
        // bring the contents to a boil
        boiled = true;
    }
}
public boolean isEmpty() {
    return empty;
}
public boolean isBoiled() {
    return boiled;
}
}

```

Vogliamo permettere la creazione di un solo calderone e quindi utilizziamo il Singleton Pattern.

```

public class Cauldron{
    private boolean empty;
    private boolean boiled;
    private static Cauldron uniqueInstance;
    private Cauldron() {
        empty = true;
        boiled = false;
    }

    public static Cauldron getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Cauldron();
        }
        return uniqueInstance;
    }
}

```

```

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
        }
    }
    // rest of Cauldron code...
}

```

Uno dei problemi critici del Singleton Pattern è quello di gestire più thread alla volta. Infatti in caso di thread multipli, possiamo avere due istanziazioni della stessa classe (una istanza per ogni thread). Per risolvere questo problema possiamo aggiungere la parola chiave synchronized a getInstance(), forzando ogni thread ad aspettare il suo turno prima di poter entrare nel metodo (la sincronizzazione di un metodo però può diminuire le prestazioni di un fattore 100)

```

public class Singleton {
    private static Singleton uniqueInstance;
    // other useful instance variables here
    private Singleton() {}

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
    // other useful methods here
}

```

In caso di criticità possiamo creare un oggetto statico all'interno della classe Singleton e restituire all'utente l'oggetto interno con il metodo getInstance().

```

public class Singleton {
    private static Singleton uniqueInstance = new
        Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return uniqueInstance;
    }
}

```

La JVM garantisce che l'istanza venga creata prima che qualsiasi thread acceda alla variabile statica uniqueInstance. Un altro modo per ridurre l'uso di sincronizzazione è quello di fare "double-checked locking".

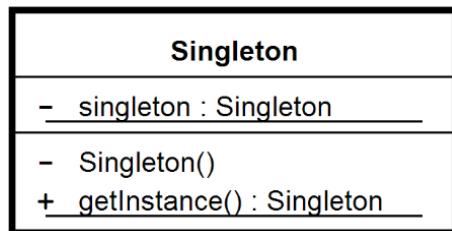
```
public class Singleton {
```

```

private volatile static Singleton uniqueInstance;
private Singleton() {}
public static Singleton getInstance() {
    if (uniqueInstance == null) {
        synchronized (Singleton.class) {
            if (uniqueInstance == null) {
                uniqueInstance = new Singleton();
            }
        }
    }
    return uniqueInstance;
}
}

```

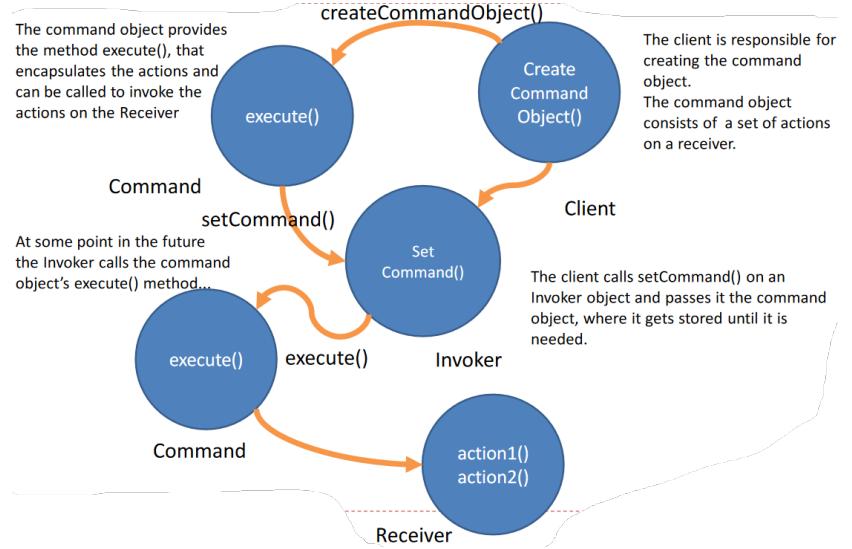
La keyword volatile assicura che più thread gestiscano correttamente la variabile uniqueInstance quando viene inizializzata all'istanza singleton. Un'altra implementazione del Singleton molto efficiente è quella basata su Enum che però non tratteremo in questo corso (<https://dzone.com/articles/java-singletons-using-enum>)



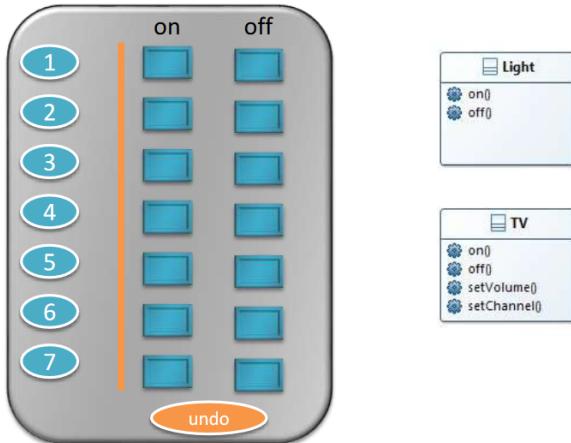
Lo schema Singleton assicura che ci sia al massimo un'istanza di una classe nell'applicazione ed un punto di accesso globale a tale istanza. L'implementazione di Java più usata dello schema Singleton fa uso di un costruttore privato, un metodo statico combinato con una variabile statica. Ma bisogna sempre esaminare i nostri vincoli di prestazioni e risorse e scegliere con attenzione un'implementazione Singleton appropriata per le applicazioni multithread.

3.3.6 Command Pattern

Il **Command Pattern** ci permette di incapsulare l'invocazione di metodi per cristallizzare pezzi di calcolo, in modo che l'oggetto che invoca il calcolo non abbia il bisogno di preoccuparsi di come fare le cose.



Vediamo subito un esempio per capirne la motivazione e l'utilizzo: È necessario implementare il codice per la gestione di un telecomando universale per la gestione di diversi dispositivi in casa (ad esempio, controllo di luci e TV, ...).



Implementiamo l'interfaccia Command

```

public interface Command {
    public void execute();
}

```

Implementare un comando per accendere una luce. Per ogni dispositivo e per funzione da abilitare deve essere creato un comando.

```

public class LightOnCommand implements Command {

```

```

        Light light;
    public LightOnCommand(Light light) {
        this.light = light;
    }
    public void execute() {
        light.on();
    }
}

public class SimpleRemoteControl {
    Command slot;
    public SimpleRemoteControl() {}
    public void setCommand(Command command) {
        slot = command;
    }
    public void buttonWasPressed() {
        slot.execute();
    }
}

public class RemoteControlTest { // client
    public static void main(String[] args) {
        SimpleRemoteControl remote = new
            SimpleRemoteControl(); //invoker
        Light light = new Light();
        LightOnCommand lightOn = new
            LightOnCommand(light);
        remote.setCommand(lightOn);
        remote.buttonWasPressed();
    }
}

```

Con la stessa logica possiamo implementare il telecomando universale per la gestione di più accessori.

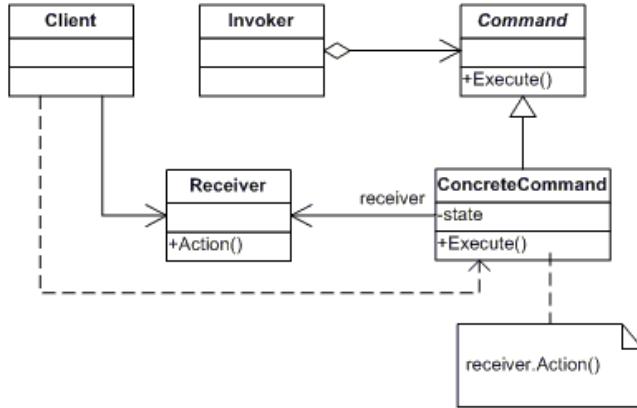
Possiamo anche introdurre un NoCommand object utile quando non si ha un oggetto significativo da restituire, ma si vuole rimuovere la responsabilità di gestire i null dal client.

```

public class NoCommand implements Command {
    public void execute() { }
}

```

La struttura UML del Command Pattern è riportata nella figura sottostante.



Il client (caricatore remoto) crea degli oggetti Command, i quali vengono caricati negli slot del RemoteControl. Il telecomando (invoker) gestisce una serie di oggetti command, uno per ogni pulsante; quando viene premuto un pulsante, viene richiamato il metodo ButtonWasPushed(), che richiama il metodo execute() sul comando. Tutti i comandi del telecomando implementano l'interfaccia Command, che consiste in un solo metodo: execute(). Utilizzando l'interfaccia Command, implementiamo ogni azione che può essere invocata premendo un pulsante sul telecomando con un oggetto Command. L'oggetto Command contiene un riferimento a un oggetto e implementa un metodo execute che richiama uno o più metodi di quell'oggetto. Infine le classi Vendor sono utilizzate per eseguire il lavoro di controllo dell'automazione domestica.

Con questo pattern possiamo anche definire un macro comando, cioè un comando che esegue uno o più comandi.

```

public class MacroCommand implements Command {
    Command[] commands;
    public MacroCommand(Command[] commands) {
        this.commands = commands;
    }
    public void execute() {
        for (int i = 0; i < commands.length; i++) {
            commands[i].execute();
        }
    }
}

Command[] allOn = { lightOn, stereoOn, tvOn, hottubOn};
MacroCommand allOnMacro = new MacroCommand(allOn);
  
```

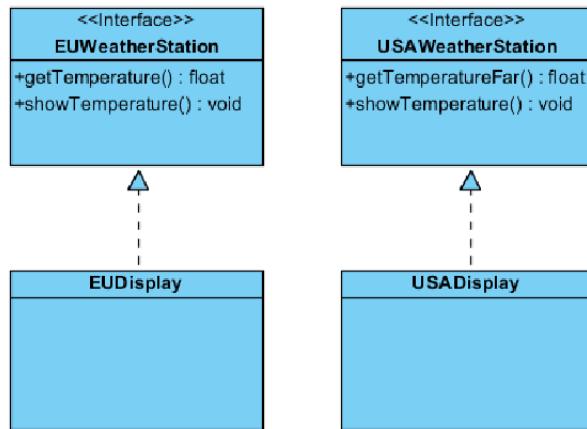
I comandi ci danno un modo per impacchettare un pezzo di calcolo (un ricevitore e un insieme di azioni) e passarlo come un oggetto ed il calcolo può essere invocato dopo che un'applicazione client ha creato l'oggetto comando. Applicazioni utili per il Command Pattern possono essere uno scheduler, un pool di

thread e delle code di lavoro.

Immaginate una coda di lavoro: da un lato si aggiungono comandi alla coda, dall'altro un gruppo di thread, i quali eseguono il seguente script: rimuovono un comando dalla coda, chiamano il suo metodo execute(), aspettano che la chiamata finisca, quindi scartano l'oggetto comando e ne recuperano uno nuovo. In questo modo le classi della coda di lavoro sono disaccoppiate dagli oggetti che eseguono i calcoli che si limitano a recuperare i comandi e a chiamare execute(). La semantica di alcune applicazioni richiede che vengano registrate tutte le azioni ed essere in grado di riprendersi dopo un arresto anomalo, richiamando quelle azioni. Il Command Pattern può supportare questa semantica con l'aggiunta di due metodi: store() e load(). Quando si eseguono i comandi, se ne memorizza la cronologia su disco cosiché quando si verifica un arresto anomalo, si ricaricano gli oggetti comando e si invocano i loro metodi execute() in batch e in ordine. Utilizzando il logging, possiamo salvare tutte le operazioni dall'ultimo punto di controllo e, in caso di guasto del sistema, applicare tali operazioni al nostro checkpoint.

3.3.7 Adapter Pattern

Possiamo adattare un progetto che prevede un'interfaccia ad una classe che implementa un'interfaccia diversa. Consideriamo un sistema software esistente: è necessario lavorare con la libreria di classi di un nuovo fornitore ma il nuovo fornitore ha progettato le proprie interfacce in maniera in modo diverso rispetto al precedente fornitore; l'adattatore agisce come intermediario, ricevendo le richieste dal client e convertendole in richieste che abbiano senso per le classi del fornitore. Vediamo subito un esempio:



L'idea è quella di rendere l'EUDisplay in grado di mostrare la temperatura espressa da una stazione di misura in Fahrenheit.

```

public interface EUWeatherStation{
    public float getTemperature();
    public void showTemperature();
}

public interface USAWeatherStation{
    public float getTemperatureFar();
    public void showTemperature();
}

public class EUDisplay implements EUWeatherStation{
    public float getTemperature() {
        System.out.println("Temperature in Celsius");
        return 22.7; //example
    }
    public void showTemperature() {
        System.out.println("This is the temperature in
                           Celsius");
    }
}

public class USADisplay implements UsaWeatherStation{
    public float getTemperatureFar(){
        System.out.println("Temperature in Fahrenheit");
        return 76; //example
    }
    public void showTemperature() {
        System.out.println("This is the temperature in
                           Fahrenheit");
    }
}

```

Dobbiamo quindi definire una classe Adapter che implementa EUWheaterStation che però ha al suo interno un riferimento ad un oggetto USADisplay che usa i gradi Fahrenheit

```

public class EUUSADisplayAdapter implements
EUWeatherStation {
    USADisplay newStation;
    public EUUSADisplayAdapter(USADisplay newStation) {
        this.newStation = newStation;
    }
    public float getTemperature() {
        return ((newStation.getTemperatureFar()-32)*5/9);
    }
    public void showTemperature() {

```

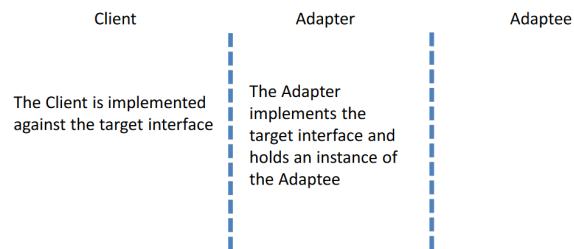
```

        System.out.println("This is the temperature " +
            (newStation.getTemperatureFar()-32)*5/9); }
    }

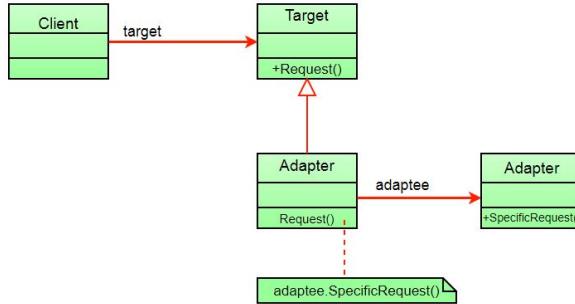
public class TestDrive {
    public static void main(String[] args) {
        EUDisplay euDisplay= new EUDisplay();
        USADisplay usaDisplay= new USADisplay();
        EUWeatherStation wStation= new
            EUUSADisplayAdapter(usaDisplay);
        test(wStation);
    }

    static void test(EUWeatherStation euW) {
        euW.getTemperature();
        euW.showTemperature();
    }
}

```



Il client effettua una richiesta all'adattatore utilizzando l'interfaccia di destinazione utilizzando l'interfaccia target. L'adattatore traduce la richiesta in una o più chiamate all' adaptee utilizzando l'interfaccia adaptee. Il client riceve i risultati della chiamata e non sa mai che c'è un adattatore che sta facendo la traduzione. Lo schema Adapter converte l'interfaccia di una classe in un'altra interfaccia che i clienti si aspettano; consente alle classi di lavorare insieme che altrimenti non potrebbero lavorare insieme a causa di interfacce incompatibili.



In verità esistono due tipi di adattatori: gli adattatori di oggetti e gli adattatori di classi. L'adattatore di classi utilizza l'ereditarietà multipla per adattare un'interfaccia a un'altra, implementazione impossibile in Java!

3.3.8 Facade Pattern

Una facciata è un oggetto che fornisce un'interfaccia semplificata a un corpo di codice più grande, come una libreria di classi. Supponiamo di dover gestire un home theater: se vogliamo guardare un film, dobbiamo abbassare le luci, abbassare lo schermo, accendere il proiettore, mettere il proiettore in modalità wide-screen, accendere l'amplificatore audio, impostare l'amplificatore sull'ingresso DVD, impostare l'amplificatore sul suono surround, impostare il volume dell'amplificatore su medio, accendere il lettore DVD, avviare la riproduzione del lettore DVD ed impostare l'ingresso del proiettore su DVD.

```

lights.dim(10);
screen.down();
projector.on();
projector.setInput(dvd);
projector.wideScreenMode();
amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);
dvd.on();
dvd.play(movie);

// involves 5 different classes!

```

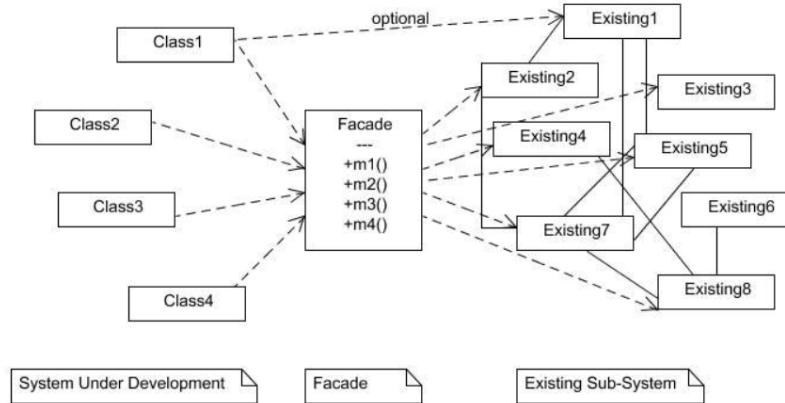
Con il pattern Facade è possibile prendere un sottosistema complesso e renderlo più facile da usare, implementando una classe Facade che fornisca un'interfaccia più ragionevole. Possiamo creare una nuova classe HomeTheaterFacade, che espone alcuni semplici metodi (ad esempio watchMovie()). Il client ora chiama i metodi della Facade dell'home theater, non quelli del sottosistema e quindi per guardare un film basta chiamare un metodo, watchMovie(), che comunica con

le altre classi. La Facade lascia comunque il sottosistema accessibile per essere utilizzato direttamente.

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    public HomeTheaterFacade(Amplifier amp, Tuner
        tuner, DvdPlayer dvd,
        CdPlayer cd, Projector projector, Screen screen,
        TheaterLights lights) {
        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
    }
    // other methods here
}

public void watchMovie(movie){
    lights.dim(10);
    screen.down();
    projector.on();
    projector.setInput(dvd);
    projector.wideScreenMode();
    amp.on();
    amp.setDvd(dvd);
    amp.setSurroundSound();
    amp.setVolume(5);
    dvd.on();
    dvd.play(movie);
}
```

Il pattern Facade quindi fornisce un'interfaccia unificata a un insieme di interfacce di un sottosistema. La facciata definisce un'interfaccia di livello superiore che rende il sottosistema più facile da usare ("avvolge" un insieme di oggetti per semplificare).



Grazie al Facade Pattern possiamo rispettare **The Principle of Knowledge** che ci dice che "bisogna parlare solo con i vostri amici piú stretti". Per qualsiasi oggetto, bisogna fare attenzione al numero di classi con cui interagisce e al anche al modo in cui interagisce con tali classi. Questo principio ci impedisce di creare progetti con un gran numero di classi accoppiate tra loro, in modo che i cambiamenti in una parte del sistema si ripercuotano a cascata su altre parti. Il principio ci dice che dobbiamo invocare solo i metodi che appartengono a: all'oggetto stesso, agli oggetti passati come parametro al metodo, qualsiasi oggetto creato o istanziato dal metodo, qualsiasi componente dell'oggetto. Vediamo come applicarlo.

```
public float getTemp() {
    return station.getTemperature();
}
```

Senza il principio si ottiene l'oggetto termometro dalla stazione e poi chiamiamo il metodo `getTemperature()` da soli.

```
public float getTemp() {
    Termometro termometro = station.getThermometer();
    restituisce thermometer.getTemperature();
}
```

Con il principio aggiungiamo un metodo alla classe `Station` che fa la richiesta al termometro per noi (questo riduce il numero di classi da cui dipendiamo).

```

public class Car {
    Engine engine;
    // other instance variables

    public Car() {
        // initialize engine, etc.
    }

    public void start(Key key) {
        Doors doors = new Doors();
        boolean authorized = key.turns();
        if (authorized) {
            engine.start();
            updateDashboardDisplay();
            doors.lock();
        }
    }

    public void updateDashboardDisplay() {
        // update display
    }
}

```

Here's a component of this class. We can call its methods.

Here we're creating a new object; its methods are legal.

You can call a method on an object passed as a parameter.

You can call a method on a component of the object.

You can call a local method within the object.

You can call a method on an object you create or instantiate

3.3.9 Template Pattern

Consideriamo due classi di supporto alla preparazione di tè e caffè.

```

public class Tea {
    void prepareRecipe() {
        boilWater();
        steepTeaBag();
        pourInCup();
        addLemon();
    }

    public void boilWater() {
        System.out.println("Boiling ");
    }

    public void steepTeaBag() {
        System.out.println("Steeping the tea");
    }

    public void addLemon() {
        System.out.println("Adding Lemon");
    }

    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
}

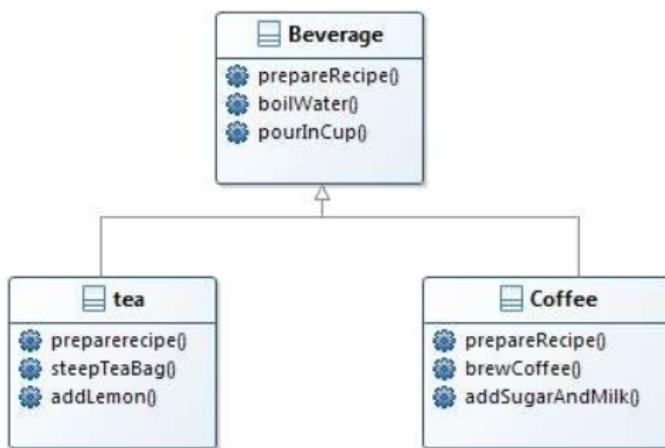
```

```

public class Coffee {
    void prepareRecipe() {
        boilWater();
        brewCoffeeGrinds();
        pourInCup();
        addSugarAndMilk();
    }
    public void boilWater() {
        System.out.println("Boiling");
    }
    public void brewCoffeeGrinds() {
        System.out.println("Dripping Coffee through
                           filter");
    }
    public void pourInCup() {
        System.out.println("Pouring into cup");
    }
    public void addSugarAndMilk() {
        System.out.println("Adding Sugar and Milk");
    }
}

```

Come possiamo notare il codice è duplicato (prepareRecipe(), boilWater()). Come possiamo progettare le classi per ridurre le duplicazioni? Dobbiamo usare classi astratte ed interfacce per astrarre il processo.



Il metodo prepareRecipe() è diverso in ogni sottoclasse, quindi è definito come astratto. Ogni sottoclasse sovrascrive il metodo prepareRecipe() e implementa la propria ricetta. I metodi boilWater() e pourInCup() sono condivisi

da entrambe le sottoclassi, quindi sono definiti nella superclasse. È possibile un'astrazione migliore?

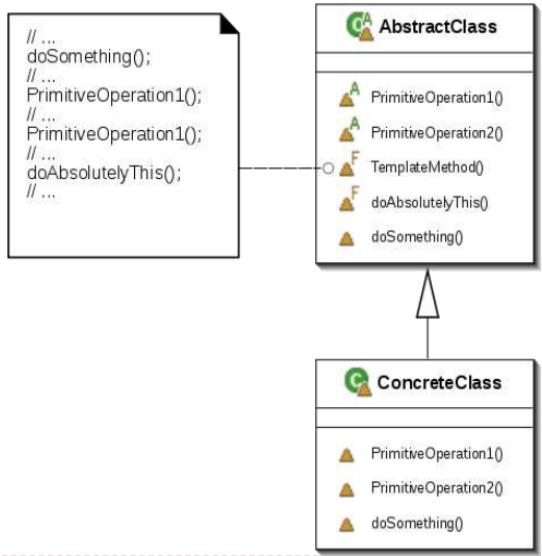
I metodi che differiscono nelle sottoclassi possono essere unificati e definiti come astratti nella superclasse

- addLemon(), addMilkAndSugar() –> addCondiments()
- stepteaBag(), brewCoffee() –> brew()

Il metodo prepareRecipe() è lo stesso in entrambe le classi e quindi può essere reso final nella superclasse.

```
public abstract class CaffeineBeverage {  
    void final prepareRecipe() {  
        boilWater();  
        brew();  
        pourInCup();  
        addCondiments();  
    }  
    abstract void brew();  
    abstract void addCondiments();  
    void boilWater() {  
        // implementation  
    }  
    void pourInCup() {  
        // implementation  
    }  
}
```

Lo schema Template Method definisce lo scheletro di un algoritmo in un metodo, rinviano alcuni passaggi alle sottoclassi. Permette alle sottoclassi di ridefinire alcuni passaggi di un algoritmo senza modificare la struttura dell'algoritmo.



Possiamo anche avere metodi concreti dichiarati nella classe astratte che non fanno nulla o hanno un'implementazione di default chiamati "hooks". Le sottoclassi sono libere di sovrascrivere questi metodi, ma non obbligate a farlo.

```

abstract class AbstractClass {
    final void templateMethod() {
        primitiveOperation1();
        primitiveOperation2();
        concreteOperation();
        hook();
    }
    abstract void primitiveOperation1();
    abstract void primitiveOperation2();
    final void concreteOperation() {
        // implementation here
    }
    void hook() {}
}

public abstract class BeverageWithHook {
    final void prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        if (wantsCondiments()) {
            addCondiments();
        }
    }
}

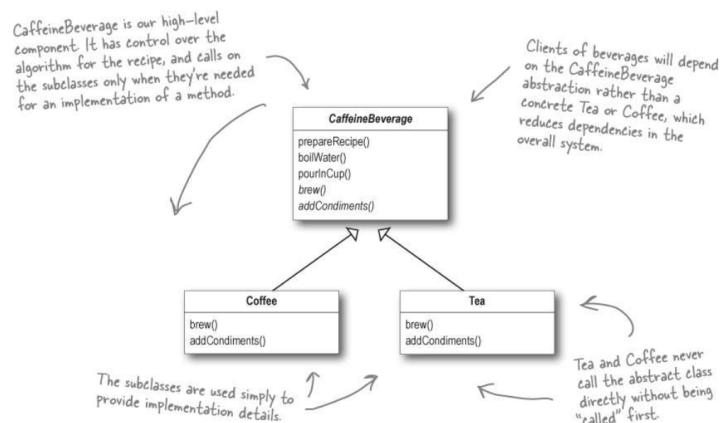
```

```

abstract void brew();
abstract void addCondiments();
void boilWater() {
    System.out.println("Boiling water");
}
void pourInCup() {
    System.out.println("Pouring into cup");
}
boolean wantsCondiments() { //hook
    return true;
}
}

```

Il Template Pattern permette di rispettare **The Hollywood Principle** che ci dice che "Non chiamateci, vi chiameremo noi". Il Principio di Hollywood ci offre un modo per prevenire il "dependency rot"; permettiamo ai componenti di basso livello di agganciarsi (hook) a un sistema, ma i componenti di alto livello determinano quando sono necessari e come.



3.3.10 Iterator Pattern

Consideriamo il menu di un ristorante, il quale ha molte voci, ognuna delle quali ha un nome, una descrizione e un prezzo.

```

public class MenuItem {
    String name;
    String description;
    boolean vegetarian;
    double price;
    public MenuItem(String name, String description,
        boolean vegetarian, double price){

```

```

        this.name = name;
        this.description = description;
        this.vegetarian = vegetarian;
        this.price = price;
    }
    public String getName() {
        return name;
    }
    public String getDescription() {
        return description;
    }
    public double getPrice() {
        return price;
    }
    public boolean isVegetarian() {
        return vegetarian;
    }
}

// Menu1 con ArrayList
public class Menu1 {
    ArrayList<MenuItem> menuItems;
    ...
}

// Menu2 con Array
public class Menu2 {
    static final int MAX_ITEMS = 6;
    int numberOfItems = 0;
    MenuItem[] menuItems;
    ...
}

```

Qual è il problema di avere due diverse rappresentazioni del menu? Proviamo a implementare un client che utilizzi i due menu.

```

public void printMenu(){
    // il metodo sembra lo stesso, ma le chiamate
    // restituiscono diversi tipi
    Menu1 menu1= new Menu1();
    ArrayList<MenuItem> menu1Items =
        menu1.getMenuItems();

    Menu2 menu2 = new Menu2();
    MenuItem[] menu2Items = menu2.getMenuItems();
}

```

L'implementazione di ogni altro metodo sarà una variazione di questo tema. I principali problemi in questa implementazione è che stiamo codificando le implementazioni concrete di Menu1 e Menu2 e non un'interfaccia, se decidiamo di passare a un altro tipo di menu che implementa l'elenco delle voci di menu con una tabella HashTable, dobbiamo modificare il codice in Manager, il Manager deve sapere come ogni menu rappresenta la sua collezione interna di voci di menu e abbiamo una duplicazione del codice. Potremmo provare ad usare il for..each per ogni menù così da nascondere la complessità delle diversi tipi di iterazione ma questo non risolverebbe il vero problema, cioè le due diverse implementazioni dei menu.

Sarebbe bello poter implementare la stessa interfaccia per i menu.



L'iteratore consente di nascondere l'implementazione interna della struttura dati, basandosi su un'interfaccia chiamata Iterato con due metodi:

- Il metodo hasNext() ci dice se ci sono altri elementi da iterare.
- Il metodo next() restituisce l'oggetto successivo nell'aggregato.

In questo modo possiamo implementare gli iteratori per qualsiasi tipo di collezione di oggetti (array, liste, hashmap, ...).

```

public interface Iterator {
    boolean hasNext();
    Object next();
}

public class Menu2Iterator implements Iterator {
    MenuItem[] items;
    int position = 0;

    public Menu2Iterator(MenuItem[] items) {
        this.items = items;
    }

    public MenuItem next() {
        MenuItem menuItem = items[position];
        position = position + 1;
        return menuItem;
    }
}

```

```

        }
    public boolean hasNext() {
        if (position >= items.length || items[position]
            == null) {
            return false;
        } else {
            return true;
        }
    }
}

```

Modifichiamo il Menu2

```

public class Menu2 {
    static final int MAX_ITEMS = 6;
    int numberofItems = 0;
    MenuItem[] menuItems;
    // constructor here
    // addItem here

    // Ritorniamo un'interfaccia Iterator
    // Il client non ha bisogno di sapere come vengono
    // mantenuti i menuItem nel menu
    // Deve solo usare gli iteratori per scorrere le
    // voci del menu.
    public Iterator createIterator() {
        return new Menu2Iterator(menuItems);
    }
    // other menu methods here
}

```

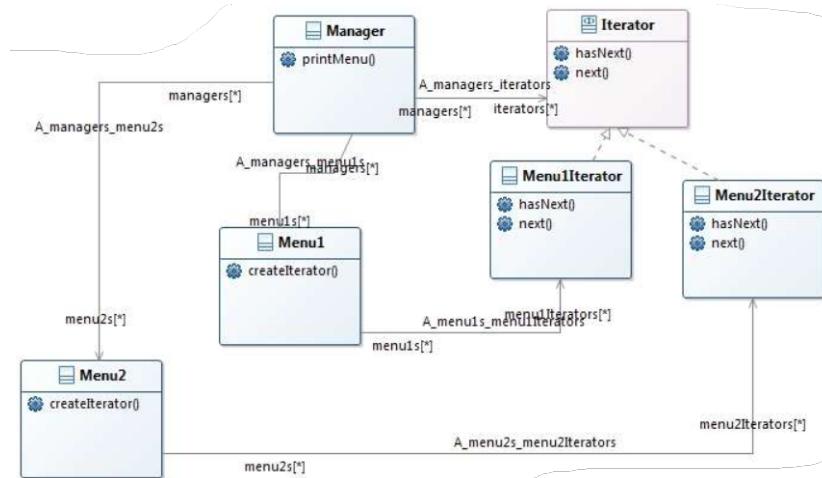
Per visualizzare i menu dovremmo modificare la funzione printMenu nel seguente modo

```

public void printMenu() {
    Iterator menu1Iterator = menu1.createIterator();
    Iterator menu2Iterator = menu2.createIterator();
    printMenu(menu1Iterator);
    printMenu(menu2Iterator);
}
private void printMenu(Iterator iterator) {
    while (iterator.hasNext()) {
        MenuItem menuItem = iterator.next();
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " -- ");
        System.out.println(menuItem.getDescription());
    }
}

```

}



Le implementazioni dei menu sono ora incapsulate. `printMenu()` non ha più idea di come i Menù mantengano la loro collezione di voci di menu. Tutto ciò di cui abbiamo bisogno è un ciclo che gestisce in modo polimorfico qualsiasi collezione di voci implementando l'iteratore.

Proviamo ad aggiungere un metodo aggiuntivo che consente di rimuovere l'ultimo elemento. Non è necessario fornire la funzionalità di rimozione ma solo fornire il metodo, perché fa parte dell'interfaccia dell'iteratore. Non abbiamo mai avuto bisogno di implementare il nostro iteratore per `ArrayList` ma avremo ancora bisogno della nostra implementazione per `Menu2` perché si basa su un `Array` (che non supportano `iterator()`).

```

public interface Menu {
    public Iterator<MenuItem> createIterator();
}

import java.util.Iterator;
public class Menu2Iterator implements Iterator {
    MenuItem[] list;
    int position = 0;
    public Menu2Iterator (MenuItem[] list) {this.list =
        list;}
    public MenuItem next() {
        //implementation here}
    public boolean hasNext() {
        //implementation here}
    public void remove() {
        if (position <= 0) {
            throw new IllegalStateException

```

```

        ("You can't remove an item until you've
         done at least one next()");
    }
    if (list[position-1] != null) {
        for (int i = position-1; i <
            (list.length-1); i++) {
            list[i] = list[i+1];
        }
        list[list.length-1] = null;
    }
}

public class Menu2 implements Menu{
    static final int MAX_ITEMS = 6;
    int numberofItems = 0;
    MenuItem[] menuItems;

    // constructor here
    // addItem here
    public Iterator createIterator() {
        return new Menu2Iterator(menuItems);
    }
}

import java.util.Iterator

// Semplicemente chiamiamo il metodo iterator() senza
// creare il nostro iteratore
public class Menu1{
    public Iterator<MenuItem> createIterator() {
        return menuItems.iterator();
    }
}

import java.util.Iterator;
public class Manager {
    Menu menu1;
    Menu menu2;
    public Manager(Menu menu1, Menu menu2) {
        this.menu1 = menu1;
        this.menu2 = menu2;
    }
    public void printMenu() {
        Iterator<MenuItem> menu1Iterator =
            menu1.createIterator();

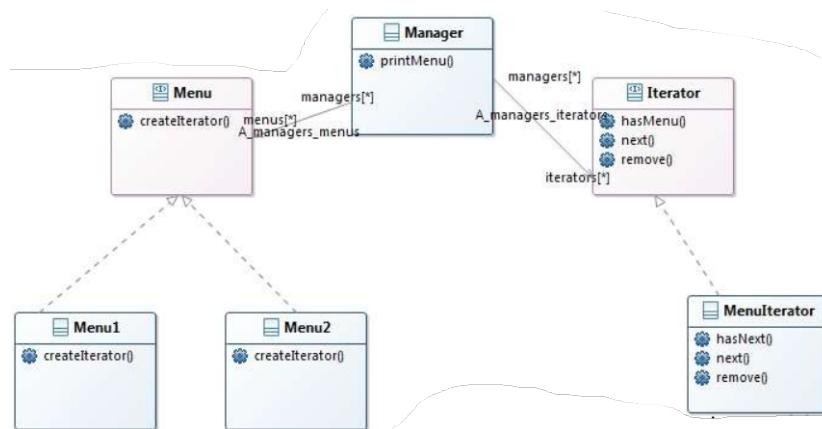
```

```

Iterator<MenuItem> menu2Iterator =
    menu2.createIterator();
printMenu(menu1Iterator);
printMenu(menu2Iterator);
}
private void printMenu(Iterator iterator) {
    while (iterator.hasNext()) {
        MenuItem menuItem =
            (MenuItem) iterator.next();
        System.out.print(menuItem.getName() + ", ");
        System.out.print(menuItem.getPrice() + " --");
        System.out.println(menuItem.getDescription());
    }
}

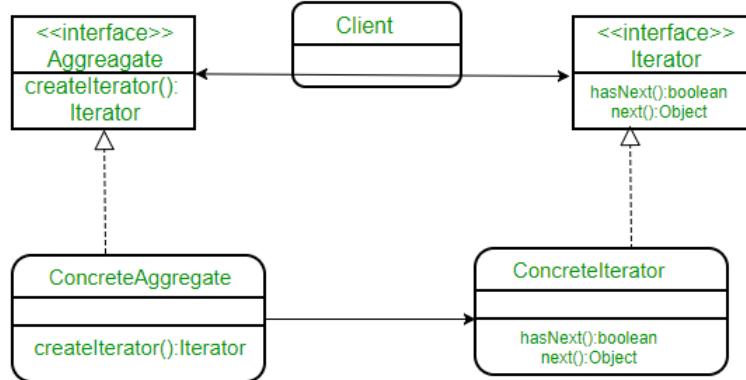
```

Cosa è cambiato in questa versione? Il gestore è disaccoppiato dall'implementazione dei menu e quindi possiamo usare un iteratore senza dover sapere come viene implementato l'elenco di elementi.



Entrambi i menu ora implementano l'interfaccia Menu e ciò significa che devono implementare il nuovo metodo `createIterator()`. `Menu1` (e 2) restituiscono un `Menulerator` dal metodo `createIterator()` perché questo è il tipo di iteratore necessario per iterare sulla matrice di voci di menu.

Il **pattern Iterator** fornisce un modo per accedere agli elementi di un oggetto aggregato in modo sequenziale, senza esporre la sua rappresentazione/implementazione sottostante.



L'interfaccia `Iterator` fornisce un insieme di metodi per l'attraversamento degli elementi di un collezione. Il `ConcreteAggregate` ha un insieme di oggetti e implementa il metodo che restituisce un iteratore per la sua collezione. Ogni `ConcreteAggregate` è responsabile dell'istanziazione di un Iteratore concreto che possa iterare sulla sua collezione di oggetti. È possibile implementare un iteratore che possa andare sia indietro che in avanti? Bisogna aggiungere due metodi, uno per arrivare all'elemento precedente e uno per dire quando si è all'inizio della collezione di elementi. Chi definisce l'ordine dell'iterazione in un insieme come `Hashtable`, che sono intrinsecamente non ordinati? Gli iteratori non implicano alcun ordine; le collezioni sottostanti possono essere non ordinate come in una `hashtable` e possono anche contenere duplicati. Cosa significa che possiamo scrivere "codice polimorfico" usando un iteratore? Significa che stiamo creando del codice che può iterare su qualsiasi collezione, a patto che supporti gli iteratori. Ogni volta che aggiungiamo un nuovo menu, dobbiamo aprire l'implementazione di Manager e aggiungere altro codice (violazione del principio di Open-Close).

```

public class Menu3 implements Menu {
    HashMap<String, MenuItem> menuItems = new
        HashMap<String, MenuItem>();
    public Menu3() {
        // constructor code here
    }
    public void addItem(String name, String
        description, boolean vegetarian, double price){
        MenuItem menuItem = new MenuItem(name,
            description, vegetarian, price);
        menuItems.put(menuItem.getName(), menuItem);
    }
    public Map<String, MenuItem> getItems() {
        return menuItems;
    }
}

```

```

        public Iterator<MenuItem> createIterator() {
            return menuItems.values().iterator();
        }
    }
    public class Manager{
        Menu menu1;
        Menu menu2;
        Menu menu3;
        public Manager(Menu menu1, Menu menu2, Menu menu3) {
            this.menu1 = menu1;
            this.menu2 = menu2;
            this.menu3 = menu3;
        }
        public void printMenu() {
            Iterator<MenuItem> menu1 =
                menu1.createIterator();
            Iterator<MenuItem> menu2= menu2.createIterator();
            Iterator<MenuItem> menu3 = menu3
                .createIterator();
            printMenu(menu1);
            printMenu(menu2);
            printMenu(menu3);
        }
    }
}

```

La soluzione è impacchettare i menu in un ArrayList e poi ottenere il suo iteratore per scorrere ogni menu.

```

public class Manager{
    ArrayList<Menu> menus;
    public Manager(ArrayList<Menu> menus) {
        this.menus = menus;
    }
    public void printMenu() {
        Iterator<Menu> menuIterator = menus.iterator();
        while(menuIterator.hasNext()) {
            Menu menu = menuIterator.next();
            printMenu(menu.createIterator());
        }
    }

    void printMenu(Iterator<MenuItem> iterator) {
        while (iterator.hasNext()) {
            MenuItem menuItem = iterator.next();
            System.out.print(menuItem.getName() + ", ");
            System.out.print(menuItem.getPrice() + " --");
        }
    }
}

```

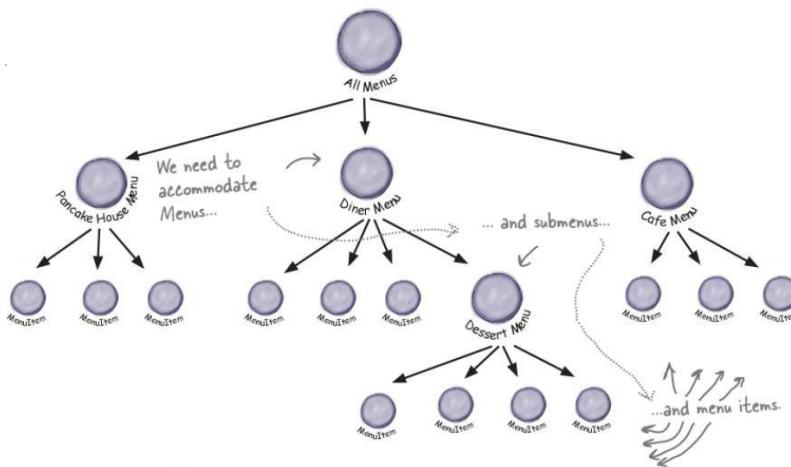
```

        System.out.println(menuItem.getDescription());
    }
}
}

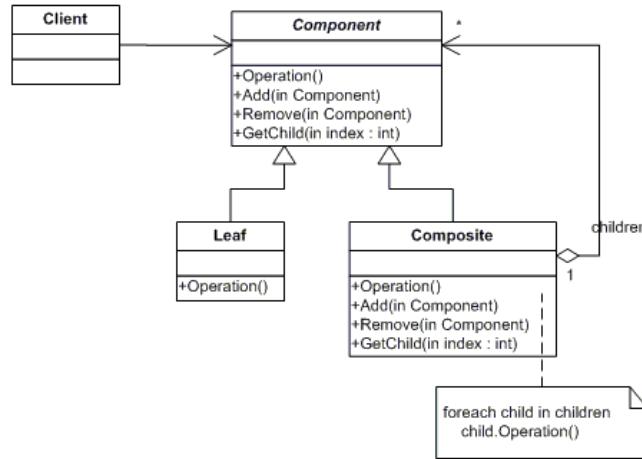
```

3.3.11 Composite Pattern

Se, riprendendo l'esempio dei Menu precedente, volessimo gestire anche dei sottomenù raggiungeremmo un livello di complessità tale per cui, se non rielaborassimo il progetto ora, non avremo mai un progetto che possa gestire acquisizioni o sottomenu. Poiché è necessario rappresentare i menu, i sottomenu annidati e le voci di menu possiamo naturalmente inserirli in una struttura ad albero.

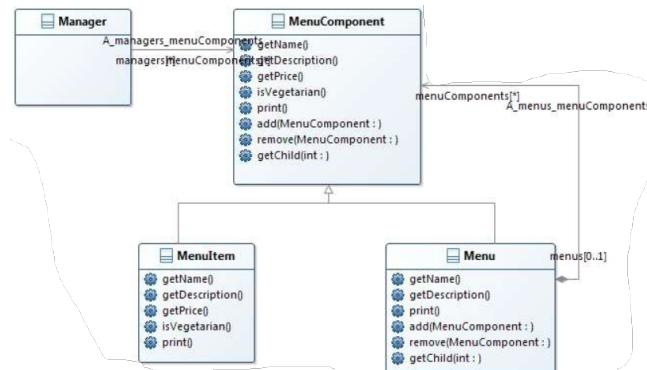


Il **pattern Composite** consente di comporre gli oggetti in strutture ad albero per rappresentare gerarchie di parti e interi; consente di trattare i singoli oggetti e le composizioni di oggetti in modo uniforme. Il Composite Pattern ci permette di costruire strutture di oggetti in forma di alberi che contengono sia composizioni di oggetti che singoli oggetti come nodi così da poter applicare le stesse operazioni sia alle composizioni che ai singoli oggetti.



Il Component definisce un'interfaccia per tutti gli oggetti della composizione: sia il composito e i nodi foglia e deve implementare un comportamento predefinito per add(), rimuovere(), getChild() e le sue operazioni. Una Leaf definisce il comportamento per gli elementi della composizione e lo fa implementando le operazioni che il Composite supporta. Il Composite implementa le operazioni di Leaf (alcune di queste potrebbero non avere senso in un Composite, quindi in caso potrebbe essere generata un'eccezione). Il client utilizza l'interfaccia Component per manipolare gli oggetti della composizione. La particolarità di questa struttura è che è ricorsiva.

Applichiamo questo pattern all'esempio del menu.



Il Manager utilizza l'interfaccia MenuComponent per accedere sia ai menu che ai MenuItem. MenuComponent rappresenta l'interfaccia per entrambi i MenuItem e Menu. MenuItem e Menu sovrascrivono alcuni metodi ed in genere l'implementazione predefinita è UnsupportedOperationException, in quanto alcuni dei metodi hanno senso solo per MenuItem ed altri solo per Menu.

```
public abstract class MenuComponent {
```

```

        public void add(MenuComponent menuComponent) {
            throw new UnsupportedOperationException();
        }
        public void remove(MenuComponent menuComponent) {
            throw new UnsupportedOperationException();
        }
        public MenuComponent getChild(int i) {
            throw new UnsupportedOperationException();
        }
        public String getName() {
            throw new UnsupportedOperationException();
        }
        public String getDescription() {
            throw new UnsupportedOperationException();
        }
        public double getPrice() {
            throw new UnsupportedOperationException();
        }
        public boolean isVegetarian() {
            throw new UnsupportedOperationException();
        }
        public void print() {
            throw new UnsupportedOperationException();
        }
    }

    public class MenuItem extends MenuComponent {
        String name;
        String description;
        boolean vegetarian;
        double price;
        public MenuItem(String name, String description,
                       boolean vegetarian, double price){ this.name
            = name;
            this.description = description;
            this.vegetarian = vegetarian;
            this.price = price;
        }
        public String getName() {
            return name;
        }
        public String getDescription() {
            return description;
        }
        public void print() {
            System.out.print(" " + getName());
        }
    }
}

```

```

        if (isVegetarian()) {
            System.out.print("(v)");
        }
        System.out.println(", " + getPrice());
        System.out.println(" -- " + getDescription());
    }
}

public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new
        ArrayList<MenuComponent>();
    String name;
    String description;
    public Menu(String name, String description) {
        this.name = name;
        this.description = description;
    }
    public void add(MenuComponent menuComponent) {
        menuComponents.add(menuComponent); }
    public void remove(MenuComponent menuComponent) {
        menuComponents.remove(menuComponent);
    }
    public MenuComponent getChild(Integer i) {
        return menuComponents.get(i);
    }
    public String getName() {
        return name;
    }
    public String getDescription() {
        return description;
    }

    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("-----");
        Iterator<MenuComponent> iterator =
            menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                iterator.next();
            menuComponent.print();
        }
    }
}

```

```

public class Manager{
    MenuComponent allMenus;
    public Manager(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }
    public void printMenu() {
        allMenus.print();
    }
}

```

Lo schema Composite prende il principio di progettazione della responsabilità singola e lo scambia con la trasparenza. Infatti perdiamo un po' di sicurezza, perché un cliente potrebbe tentare di fare qualcosa di inappropriato o di insensato su un elemento (ad esempio, tentare di aggiungere un menu a un elemento del menu).

3.3.12 CompositIterator Pattern

Possiamo anche integrare i due pattern, vediamo come procedere sull'esempio del Menu.

```

public class Menu extends MenuComponent {
    ArrayList<MenuComponent> menuComponents = new
        ArrayList<MenuComponent>();
    Iterator<MenuComponent> iterator = null;
    // other code here doesn't change
    public Iterator<MenuComponent> createIterator() {
        if (iterator == null) {
            iterator = new
                CompositeIterator(menuComponents.iterator());
        }
        return iterator;
    }
}

```

Aggiungiamo un metodo createIterator() nell'astratto MenuComponent

```

import java.util.*;
public class CompositeIterator implements Iterator {
    Stack<Iterator<MenuComponent>> stack = new
        Stack<Iterator<MenuComponent>>();
    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }
    public MenuComponent next() {
        if (hasNext()) {
            Iterator<MenuComponent> iterator =
                stack.peek();

```

```

        MenuComponent component = iterator.next();
        stack.push(component.createIterator());
        return component;
    } else {
        return null;
    }
}
public boolean hasNext() {
    if (stack.empty()) {
        return false;
    } else {
        Iterator<MenuComponent> iterator =
            stack.peek();
        /* Quindi lanciamo l'iteratore di quel
           componente sullo stack. Se il componente
           e' un menu, si iterera' su tutti i suoi
           elementi. Se il componente e' un MenuItem,
           otterremo
           l'iteratore NullIterator e non verra'
           eseguita alcuna iterazione.*/
        if (iterator == null || !iterator.hasNext())
        {
            stack.pop();
            return hasNext();
        } else {
            return true;
        }
    }
}

```

MenuItem non ha nulla su cui iterare e quindi, come gestiamo l'implementazione del suo metodo createIterator()?

- Ritornare null ma poi avremmo bisogno di un codice nel client per vedere se è stato restituito null o meno.

```

public class MenuItem extends MenuComponent {
    // other code here doesn't change
    public Iterator<MenuComponent> createIterator() {
        return null;
    }
}

```

- Restituire un iteratore che restituisce sempre FALSE quando viene richiamato il metodo hasNext().

```

public class MenuItem extends MenuComponent {

```

```

// other code here doesn't change
public Iterator<MenuComponent> createIterator() {
    return new NullIterator();
}

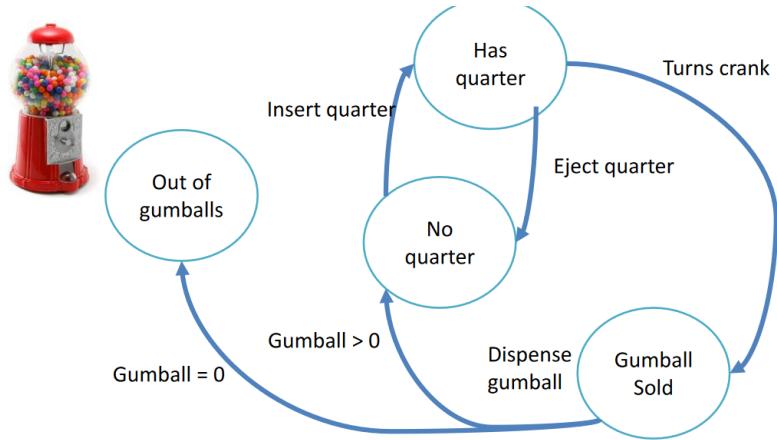
import java.util.Iterator;
public class NullIterator implements
    Iterator<MenuComponent> {
    public Object next() {
        return null;
    }
    public boolean hasNext() {
        return false;
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}

public class Manager{
    MenuComponent allMenus;
    public Manager(MenuComponent allMenus) {
        this.allMenus = allMenus;
    }
    public void printMenu() {
        allMenus.print();
    }
    // Iterate through every element of the composite.
    public void printVegetarianMenu() {
        Iterator<MenuComponent> iterator =
            allMenus.createIterator();
        System.out.println("\nVEGETARIAN MENU\n----");
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                iterator.next();
            try {
                if (menuComponent.isVegetarian()) {
                    menuComponent.print();
                }
            } catch (UnsupportedOperationException e) {}
        }
    }
}

```

3.3.13 State Pattern

State Pattern aiuta gli oggetti a controllare il loro comportamento cambiando il loro stato interno.

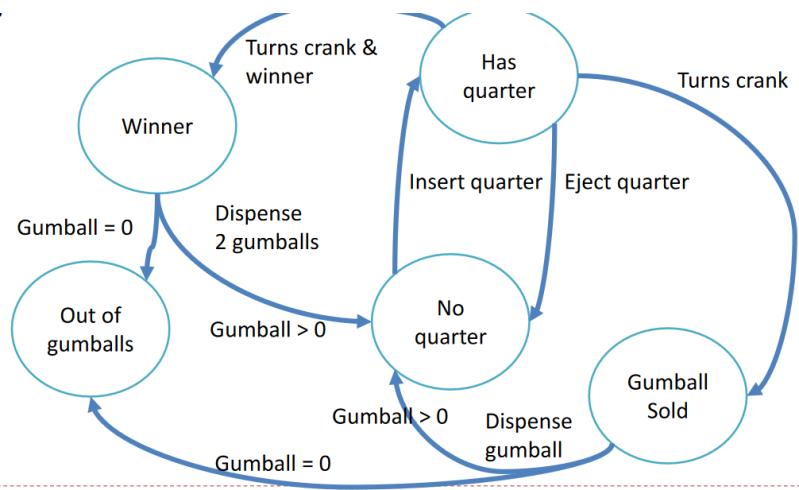


Creiamo una classe che funge da macchina a stati e che per ogni azione (inserire il quarto, espellere il quarto...), esiste un metodo che utilizza dichiarazioni condizionali per determinare il comportamento appropriato.

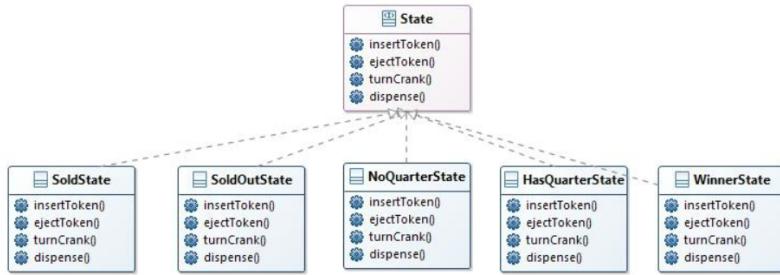
```
public class GumballMachine {
    // stati della state machine
    final static int SOLD_OUT = 0;
    final static int NO_QUARTER = 1;
    final static int HAS_QUARTER = 2;
    final static int SOLD = 3;
    int state = SOLD_OUT;
    int count = 0;
    public GumballMachine(int count) {
        this.count = count;
        if (count > 0) {
            state = NO_QUARTER;
        }
    }
    public void insertQuarter() {
        if (state == HAS_QUARTER) {
            System.out.println("You can't insert another
                quarter");
        } else if (state == NO_QUARTER) {
            state = HAS_QUARTER;
            System.out.println("You inserted a quarter");
        } else if (state == SOLD_OUT) {
    
```

```
        System.out.println("The machine is sold  
            out");  
    } else if (state == SOLD) {  
        System.out.println("We're already giving you  
            a gumball");  
    }  
}
```

Aggiungiamo la seguente feature: Il 10% delle volte, quando la manovella viene girata, il cliente ottiene due palline di gomma invece di una.



Il codice precedentemente implementato non è semplice da estendere perché si dovrebbe aggiungere un nuovo stato WINNER (vincitore), aggiungere una nuova condizione in ogni singolo metodo per gestire lo stato WINNER e turn-Crank() diventerà particolarmente complicato perché si dovrà aggiungere del codice per verificare se si ha una vittoria e poi passare allo stato WINNER o allo stato SOLD (no rispetta l'Open Closed Principle). Una soluzione potrebbe essere quella di rielaborare il codice esistente, incapsulando gli oggetti di stato nelle proprie classi e delegando allo stato corrente quando si verifica un'azione. Per far ciò dobbiamo definire un'interfaccia di Stato che contenga un metodo per ogni azione della Gumball Machine, implementare una classe State per ogni stato della macchina e delegare alla classe State di fare il lavoro al posto nostro.



```

// Interfaccia Stato
public interface State{
    void insertToken();
    void ejectToken();
    void turnCrank();
    void dispense();
}

// Stati
public class NoQuarterState implements State {
    GumballMachine gumballMachine;
    public NoQuarterState(GumballMachine gumballMachine)
    {
        this.gumballMachine = gumballMachine;
    }
    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }
    public void ejectQuarter() {
        System.out.println("You haven't inserted a
                           quarter");
    }
    public void turnCrank() {
        System.out.println("You turned, but there's no
                           quarter");
    }
    public void dispense() {
        System.out.println("You need to pay first");
    }
}
public class HasQuarterState implements State {
    GumballMachine gumballMachine;

```

```

public HasQuarterState(GumballMachine
    gumballMachine) {
    this.gumballMachine = gumballMachine;
}
public void insertQuarter() {
    System.out.println("You can't insert another
        quarter");
}
public void ejectQuarter() {
    System.out.println("Quarter returned");
    gumballMachine.setState(gumballMachine.getNoQuarterState());
}
public void turnCrank() {
    System.out.println("You turned...");
    gumballMachine.setState(gumballMachine.getSoldState());
}
public void dispense() {
    System.out.println("No gumball dispensed");
}
}

// Gumball Machine
public class GumballMachine {
    State soldOutState;
    State noQuarterState;
    State hasQuarterState;
    State soldState;
    State state;
    int count = 0;
    public GumballMachine(int numberGumballs) {
        soldOutState = new SoldOutState(this);
        noQuarterState = new NoQuarterState(this);
        hasQuarterState = new HasQuarterState(this);
        soldState = new SoldState(this);
        this.count = numberGumballs;
        if (numberGumballs > 0) {
            state = noQuarterState;
        } else {
            state = soldOutState;
        }
    }
    public void insertQuarter() {
        state.insertQuarter();
    }
    public void ejectQuarter() {
        state.ejectQuarter();
    }
}

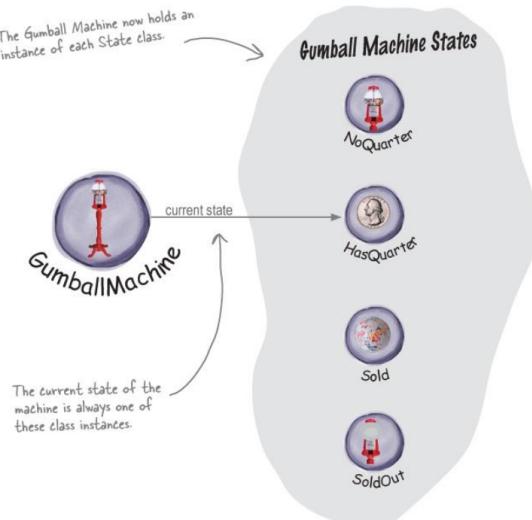
```

```

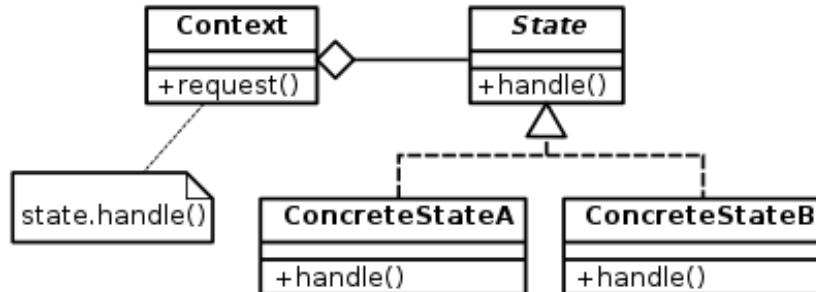
    }
    public void turnCrank() {
        state.turnCrank();
        state.dispense();
    }
    void setState(State state) {
        this.state = state;
    }
    void releaseBall() {
        System.out.println("A gumball comes rolling out
                           the slot...");
        if (count != 0) {
            count = count - 1;
        }
    }
    // More methods here including
    getters for each State...
}

```

L'implementazione della Gumball Machine è strutturalmente diversa dalla prima versione, ma dal punto di vista funzionale è esattamente la stessa con il vantaggio che il comportamento di ogni stato localizzato nella propria classe.



Lo State Pattern consente ad un oggetto di modificare il suo comportamento quando il suo stato interno cambia (l'oggetto sembrerà cambiare classe perché cambia con lo stato interno).



Il Context è una classe che può avere un numero di stati interni (nel nostro esempio, la GumballMachine). L'interfaccia State definisce un'interfaccia comune per tutti gli stati concreti. Gli stati concreti gestiscono le richieste del contesto, in particolar modo ogni stato fornisce la propria implementazione per una richiesta. In questo modo, quando il contesto cambia stato, cambia anche il suo comportamento.

I diagrammi delle classi dello State Pattern e dello Strategy Pattern sono uguali, ma i due schemi differiscono nell'intento. Con lo State abbiamo un insieme di comportamenti encapsulati in oggetti di stato ; invece con lo Strategy il cliente di solito specifica l'oggetto strategia con cui il contesto è composto. Pensate allo Strategy come a un'alternativa flessibile alla sottoclassificazione ed invece pensate allo State come a un'alternativa all'inserimento di molti condizioni nel contesto.

Nella GumballMachine, gli stati decidono quale debba essere lo stato successivo. Gli Stati concreti decidono sempre quale sarà lo stato successivo? Come linea guida generale, quando le transizioni di stato sono fisse, sono appropriate per essere inserite nel Context; quando invece le transizioni sono più dinamiche vengono tipicamente collocate nelle classi di stato. Perché abbiamo bisogno del WinnerState? Non potremmo semplicemente far sì che il SoldState dispensi due palline di gomma? SoldState e WinnerState sono quasi identici, tranne per il fatto che WinnerState dispensa due palline di gomma invece di una. Si potrebbe certamente inserire il codice per distribuire due palline di gomma nel SoldState. Il rovescio della medaglia è che ora ci sono DUE stati rappresentati in una classe State lo stato in cui si vince e lo stato in cui non si vince. Quindi si sacrifica la chiarezza della classe State per ridurre la duplicazione del codice e rispettare il principio One class, One Responsibility.

3.3.14 Proxy Pattern

Il Proxy Pattern permette di controllare e gestire l'accesso ad un determinato oggetto. Implementiamo un codice per il monitoraggio del distributore di gomme da masticare.

```

public class GumballMachine {
    // other instance variables
    String location;
    public GumballMachine(String location, int count) {
    }
}
  
```

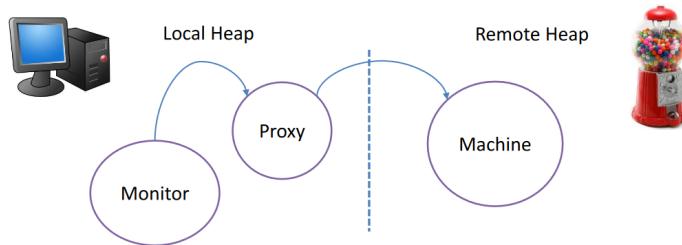
```

        // other constructor code here
        this.location = location;
    }
    public String getLocation() {
        return location;
    }
    // other methods here
}

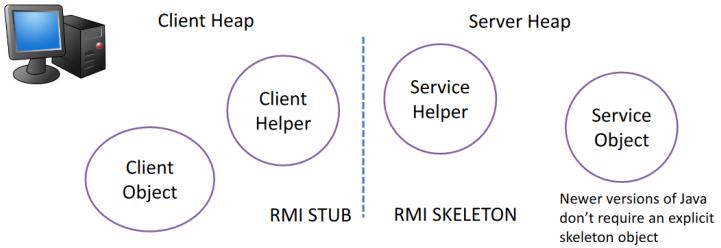
public class GumballMonitor {
    GumballMachine machine;
    public GumballMonitor(GumballMachine machine) {
        this.machine = machine;
    }
    public void report() {
        System.out.println("Gumball Machine: " +
            machine.getLocation());
        System.out.println("Current inventory: " +
            machine.getCount() + " gumballs");
        System.out.println("Current state: " +
            machine.getState());
    }
}

```

Nell'implementazione attuale diamo a GumballMonitor un riferimento a una macchina e lui ci fornisce un rapporto. Il problema è che il monitor viene eseguito nella stessa JVM della macchina gumball, cosa succederebbe se lasciassimo la nostra classe GumballMonitor così com'è, ma gli dessimo un oggetto remoto ? Il proxy remoto agisce come rappresentante locale (un oggetto su cui si possono chiamare metodi locali e che vengono inoltrati all'oggetto remoto) di un oggetto remoto, cioè di un oggetto che vive nell'heap di un'altra macchina virtuale Java.



Progettiamo un sistema che ci permetta di chiamare un oggetto locale che inoltre ogni richiesta ad un oggetto remoto.



Il client chiama un metodo sul client helper, come se il client helper fosse il servizio vero e proprio. Il client helper non è in realtà il servizio remoto: contatta il server, trasferisce le informazioni sul metodo ed attende un ritorno dal server. Sul lato server, l'helper del servizio riceve la richiesta (attraverso una connessione Socket), scompatta le informazioni sulla chiamata e poi invoca il vero metodo sul vero oggetto del servizio. Il service helper ottiene il valore di ritorno dal servizio, lo impacchetta e lo rispedisce indietro (tramite una connessione Socket) all'helper del client. Il client helper spacchetta le informazioni e restituisce il valore all'oggetto client. Per sviluppare questa struttura in Java possiamo usare la classe RMI che ci permette costruire gli oggetti client e service helper, creando un oggetto client helper con gli stessi metodi del servizio remoto. Vediamo come implementare il nostro servizio remoto:

1. Creare un'interfaccia remota

```
import java.rmi.*;
// Remote e' un'interfaccia 'marcatore': non ha
// metodi
// in questo modo il client invoca metodi su
// qualcosa che implementa l'interfaccia remota
public interface MyRemote extends Remote {
    public String sayHello() throws RemoteException;
}
```

2. Creare un'implementazione remota

```
import java.rmi.*;
import java.rmi.server.*;
// Implementare l'interfaccia remota
// Estendere UnicastRemoteObject per funzionare come
// oggetto di servizio remoto,
public class MyRemoteImpl extends
    UnicastRemoteObject implements MyRemote {
    private static final long serialVersionUID = 1L;
    public String sayHello() {
        return "Server says, 'Hey'";
    }
    public MyRemoteImpl() throws RemoteException { }
```

```

        public static void main (String[] args) {
            try {
                MyRemote service = new MyRemoteImpl();
                //Registrare il servizio con il registro
                //RMI per renderlo disponibili ai
                //client remoti
                Naming.rebind("RemoteHello", service);
            } catch(Exception ex) {
                ex.printStackTrace();
            }
        }
    }
}

```

3. Eseguire rmiregistry da terminale

```
%rmiregistry
```

4. Far partire il servizio

```
%java MyRemoteImpl
```

Il client deve ottenere l'oggetto stub (il nostro proxy), poiché è la cosa su cui chiamaerà i metodi; qui che entra in gioco il registro RMI

```

MyRemote service = (MyRemote)
naming.lookup("rmi://127.0.0.1/RemoteHello");

```

Il client esegue una ricerca nel registro RMI; il registro RMI restituisce l'oggetto stub (come valore di ritorno del metodo lookup) e RMI deserializza automaticamente lo stub. Ora il client invoca un metodo sullo stub, come se lo stub fosse il servizio vero e proprio.

```

import java.rmi.*;
public class MyRemoteClient {
    public static void main (String[] args) {
        new MyRemoteClient().go();
    }
    public void go() {
        try {
            MyRemote service = (MyRemote)
            Naming.lookup("rmi://127.0.0.1/RemoteHello");
            String s = service.sayHello();
            System.out.println(s);
        } catch(Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

Adesso faremo in modo che la GumballMachine diventi un servizio: creare un'interfaccia remota per la GumballMachine per fornire un insieme di metodi che possono essere richiamati in remoto, assicurarsi che tutti i tipi di ritorno dell'interfaccia siano serializzabili, implementare l'interfaccia in una classe concreta.

```
import java.rmi.*;
// Interfaccia Remota
public interface GumballMachineRemote extends Remote {
    public int getCount() throws RemoteException;
    public String getLocation() throws RemoteException;
    public State getState() throws RemoteException;
}

import java.io.*;
// Serializzazione
public interface State extends Serializable {
    public void insertQuarter();
    public void ejectQuarter();
    public void turnCrank();
    public void dispense();
}
public class NoQuarterState implements State {
    private static final long serialVersionUID=2L;
    // The transient keyword tells the JVM not to
    // serialize this field.
    transient GumballMachine gumballMachine;
    public NoQuarterState(GumballMachine gumballMachine)
    {
        this.gumballMachine = gumballMachine;
    }
    public void insertQuarter() {
        System.out.println("You inserted a quarter");
        gumballMachine.setState(gumballMachine.getHasQuarterState());
    }
    public void ejectQuarter() {
        System.out.println("You haven't inserted a
            quarter");
    }
    public void turnCrank() {
        System.out.println("You turned, but there's no
            quarter");
    }
    public void dispense() {
        System.out.println("You need to pay first");
    }
}
```

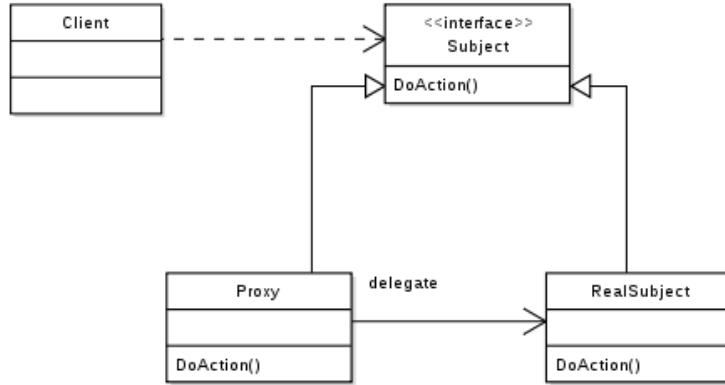
```

import java.rmi.*;
import java.rmi.server.*;
public class GumballMachine extends UnicastRemoteObject
    implements GumballMachineRemote{
    private static final long serialVersionUID = 2L;
    // other instance variables here
    public GumballMachine(String location, int
        numberGumballs) throws RemoteException {
        // code here
    }
    public int getCount() {
        return count;
    }
    public State getState() {
        return state;
    }
    public String getLocation() {
        return location;
    }
    // other methods here
}

import java.rmi.*;
public class GumballMonitor {
    GumballMachineRemote machine;
    public GumballMonitor(GumballMachineRemote machine) {
        this.machine = machine;
    }
    public void report() {
        try {
            System.out.println("Gumball Machine: " +
                machine.getLocation());
            System.out.println("Current inventory: " +
                machine.getCount() + " gumballs");
            System.out.println("Current state: " +
                machine.getState());
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }
}

```

Lo schema Proxy fornisce un surrogato o un segnaposto per un altro oggetto per controllarne l'accesso.



Sia il Proxy che il RealSubject implementano l'interfaccia soggetto: questo permette a qualsiasi client di trattare il proxy proprio come il RealSubject. Il RealSubject è solitamente l'oggetto che svolge la maggior parte del lavoro reale. Il Proxy mantiene un riferimento all'oggetto soggetto, in modo da poter inoltrare le richieste al Subject quando necessario.

Ci sono molte varianti dello schema Proxy e le varianti in genere ruotano intorno al modo in cui il proxy "controlla l'accesso".

- **Remote Proxy**

Il proxy agisce come rappresentante locale di un oggetto che risiede in un'altra JVM. Una chiamata di metodo sul proxy risulta come una chiamata all'oggetto remoto, il cui risultato viene restituito al proxy e quindi al client.

- **Virtual Proxy**

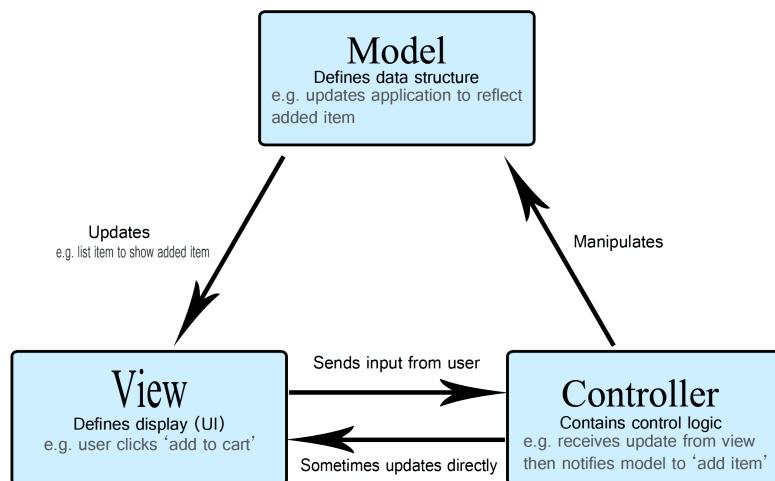
Il Proxy virtuale agisce come rappresentante di un oggetto che può essere costoso da creare. Il Virtual Proxy spesso rinvia la creazione dell'oggetto fino a quando non è necessario ed agisce anche come surrogato dell'oggetto prima e durante la creazione. Vediamo un esempio per chiarire il concetto: Vogliamo scrivere un'applicazione che visualizzi le copertine dei compact disc. Potremmo creare un menu con i titoli dei CD e poi recuperare le immagini da un servizio online come Amazon.com. Se si utilizza Swing, si potrebbe creare un'icona e chiederle di caricare l'immagine dalla rete. L'unico problema è che il recupero della copertina del CD potrebbe richiedere un po' di tempo, quindi l'applicazione dovrebbe visualizzare qualcosa nell'attesa. Una volta caricata l'immagine, il messaggio dovrebbe scomparire e si dovrebbe vedere l'immagine. Il proxy virtuale può sostituirsi all'icona e gestire il caricamento in background e, prima che l'immagine sia completamente recuperata dalla rete, visualizzare il messaggio "Caricamento della copertina del CD, attendere...". Una volta caricata l'immagine, il proxy delega la visualizzazione all'icona.

- **Protection proxy**

Controlla l'accesso a una risorsa in base ai diritti di accesso (può essere implementato con il package `java.lang.reflect`).

3.3.15 Model-view-controller(MVC)

Model-view-controller è un pattern per organizzare programmi GUI che devono visualizzare diverse viste dei dati. L'idea principale è di separare i dati dalla visualizzazione. Il componente centrale di MVC, il modello, cattura il comportamento dell'applicazione in termini di dominio del problema, indipendentemente dall'interfaccia utente; gestisce direttamente i dati, la logica e le regole dell'applicazione. Una vista può essere una qualsiasi rappresentazione di informazioni, come un grafico o un diagramma. La terza parte, il controllore, accetta gli input e li converte in comandi per il modello o la vista.

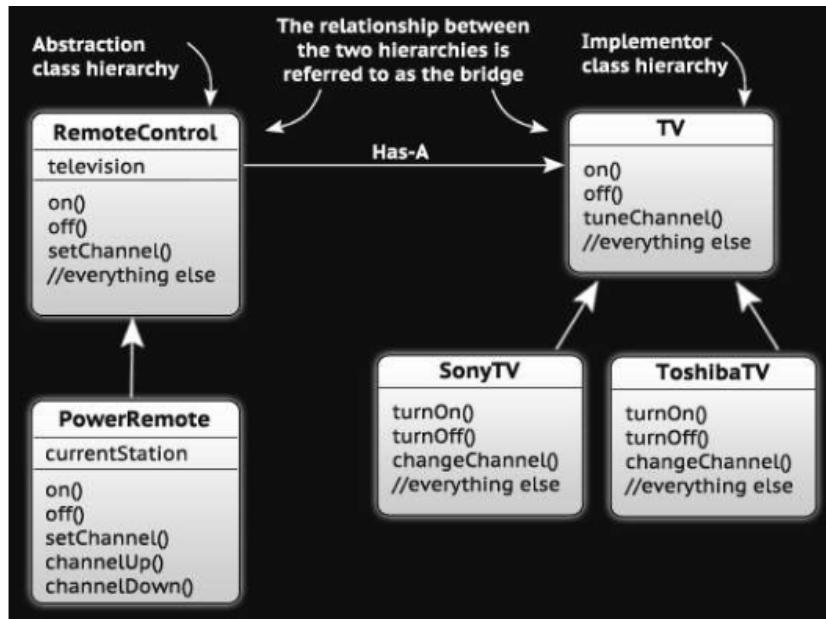


Il design MVC definisce le interazioni tra i componenti; infatti il controllore può inviare comandi al modello per aggiornarne lo stato ma può anche inviare comandi alla sua vista associata per modificare la presentazione del modello da parte della vista. Questo pattern viene molto utilizzato per le web application. I principali vantaggi di questo pattern è che riduce la complessità del sistema e migliora la manutenibilità (la separazione dei problemi minimizza l'impatto dei cambiamenti). Tuttavia il meccanismo di propagazione delle modifiche limita le prestazioni, la mancanza di struttura all'interno di ogni singolo componente comporta un rischio di accoppiamento elevato e infine l'intero sistema si basa

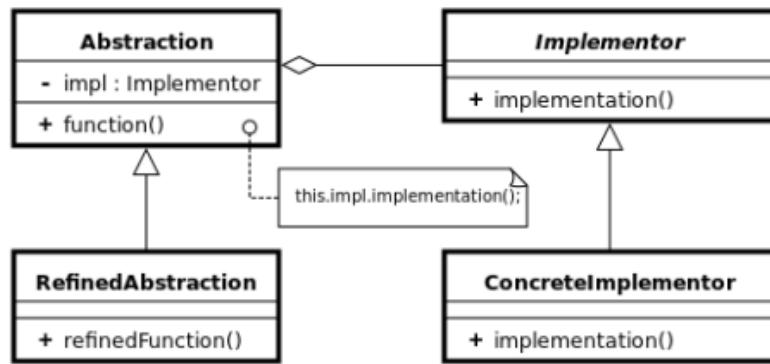
sul componente Model, il che significa che le modifiche ad esso possono rompere l'intero sistema.

3.3.16 Bridge Pattern

Utilizzate il Bridge Pattern per variare non solo le implementazioni, ma anche le astrazioni. Ad esempio, state scrivendo il codice per un nuovo telecomando ergonomico e facile da usare per i televisori. Il telecomando si basa sulla stessa astrazione, ma ci saranno molte implementazioni, una per ogni modello di TV. I telecomandi cambieranno e i televisori cambieranno. Avete già astrazione dell'interfaccia utente, in modo da poter variare l'implementazione molti televisori che i clienti possiederanno ma dovrete anche variare l'astrazione, perché questa cambierà nel tempo, man mano che il telecomando verrà migliorato. È quindi necessario un progetto OO che consenta di variare l'implementazione e l'astrazione. Il Bridge Pattern consente di variare l'implementazione e l'astrazione, collocando le due cose in gerarchie di classi separate.



Tutti i metodi dell'astrazione sono implementati in termini di implementazione e le sottoclassi concrete sono implementate in termini di astrazione, non di implementazione.



```

//Implementor
public interface TV{
    public void on();
    public void off();
    public void tuneChannel(int channel);
}

//Concrete Implementor
public class Sony implements TV{
    public void on(){//Sony specific on}
    public void off(){//Sony specific off}
    public void tuneChannel(int channel);{//Sony
        specific tuneChannel}
}

//Concrete Implementor
public class Philips implements TV{
    public void on(){//Philips specific on}
    public void off(){//Philips specific off}
    public void tuneChannel(int channel);{//Philips
        specific tuneChannel}
}

//Abstraction
public abstract class RemoteControl{
    private TV implementor;
    public void on(){
        implementor.on();
    }
    public void off(){
        implementor.off();
    }
    public void setChannel(int channel){
        implementor.tuneChannel(channel);
    }
}
  
```

```

        }
    }
//Refined abstraction
public class ConcreteRemote extends RemoteControl{
    private int currentChannel;
    public void nextChannel(){
        currentChannel++;
        setChannel(currentChannel);
    }
    public void prevChannel(){
        currentChannel--;
        setChannel(currentChannel);
    }
}

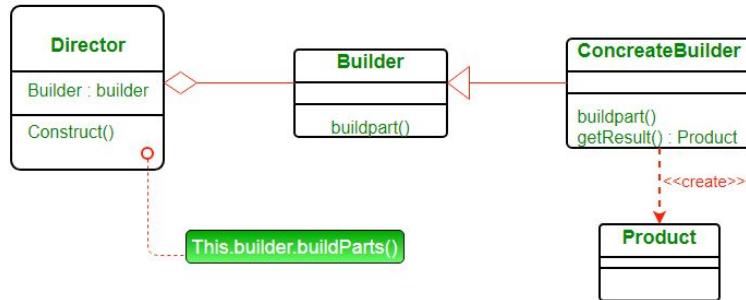
```

Il Bridge Pattern, il quale può essere visto come una estensione dello Strategy Pattern, consente di disaccoppiare un'implementazione, in modo che non sia vincolata in modo permanente ad un'interfaccia, di estendere l'astrazione e l'implementazione in modo indipendente e di modificare le classi di astrazione concrete senza influire sul client. Questo pattern risulta molto utile nei sistemi grafici e a finestre che devono essere eseguiti su più piattaforme e ogni volta che è necessario variare un'interfaccia e un'implementazione in modi diversi. Il più grande svantaggio del Bridge Pattern è che la complessità aumenta notevolmente.

3.3.17 Builder Pattern

Utilizzate il Builder Pattern per encapsulare la costruzione di un prodotto e consentirne la costruzione in fasi. Un esempio è rappresentato dagli happy meal di un fast food. L'happy è tipicamente composto da un hamburger, patatine fritte, coca cola e un giocattolo. Non importa che si scelgano hamburger/bevande differenti, la costruzione del pasto per bambini segue lo stesso processo. Quindi, è necessaria una struttura di dati flessibile che possa rappresentare gli happy meal e tutte le loro variazioni.

UML diagram of Builder Design pattern



```

// Builder
public abstract class MealBuilder {
    protected Meal meal = new Meal();
    public abstract void buildDrink();
    public abstract void buildMain();
    public abstract void buildDessert();
    public abstract Meal getMeal();
}
// Concrete Builder
public class KidsMealBuilder extends MealBuilder {
    public void buildDrink() { // add drinks to the meal}
    public void buildMain() { // add main part of the
        meal}
    public void buildDessert() { // add dessert part to
        the meal}
    public Meal getMeal() {return meal;}
}
public class AdultMealBuilder extends MealBuilder {
    public void buildDrink(){ // add drinks to the meal}
    public void buildMain(){ // add main part of the meal}
    public void buildDessert(){ // add dessert part to
        the meal}
    public Meal getMeal(){return meal;}
}

// Director
public class MealDirector {
    public Meal createMeal(MealBuilder builder) {
        builder.buildDrink();
        builder.buildMain();
        builder.buildDessert();
    }
}
  
```

```

        return builder.getMeal();
    }
}

// Integration with overall application
public class Main {
    public static void main(String[] args) {
        MealDirector director = new MealDirector();
        MealBuilder builder = null;
        if (isKid) {
            builder = new KidsMealBuilder();
        }
        else{
            builder = new AdultMealBuilder();
        }
        Meal meal = director.createMeal(mealBuilder);
    }
}

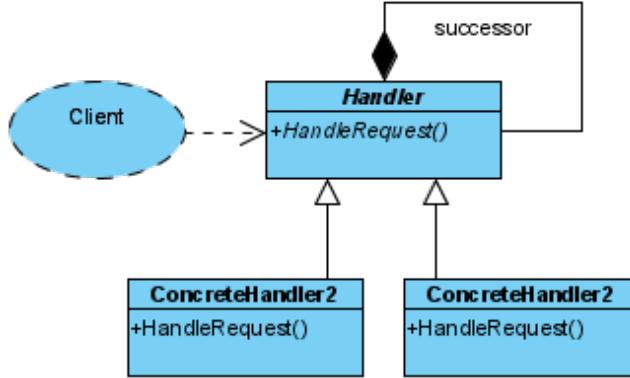
```

Nell'Iterator Pattern, abbiamo incapsulato l'iterazione in un oggetto separato per nascondere al client la rappresentazione interna dell'insieme. È la stessa stessa idea: incapsuliamo la creazione dell'happy meal in un oggetto (chiamato Builder) e facciamo in modo che il nostro cliente chieda al costruttore di costruire la struttura dell'happy meal per lui. Il Builder Pattern quindi consente di incapsulare il modo in cui viene costruito un oggetto complesso, di costruire oggetti in un processo multistep e variabile (al contrario del Factory in un solo passaggio), di nascondere la rappresentazione interna del prodotto al cliente e di poter scambiare facilmente le implementazioni del prodotto perché il cliente vede solo un'interfaccia astratta. Il principale svantaggio è che la costruzione di oggetti richiede una maggiore conoscenza del dominio da parte del cliente rispetto all'uso del Factory Pattern.

3.3.18 Chain of Responsibility

Usare lo schema della catena di responsabilità quando si vuole dare a più di un oggetto la possibilità di gestire una richiesta (applica funzioni diverse allo stesso oggetto). La struttura del pattern è piuttosto semplice, le componenti principali sono 2:

- Handler, che rappresenta la classe astratta che offre il metodo HandleRequest che sarà il metodo utilizzato dalle componenti per inoltrare richieste all'oggetto contenuto;
- ConcreteHandler, che rappresenta l'effettiva implementazione della gestione degli eventi per un oggetto.



Ad esempio, vi sono differenti ruoli ognuno con un limite massimo per un acquisto e un successore. Ogni volta che una persona (che ha un determinato ruolo) riceve un ordine di acquisto che sorpassa il proprio limite, passa la richiesta al successore nella catena di comando.

```

//Handler
abstract class PurchasePower {
    protected static final double BASE = 500;
    protected PurchasePower successor;

    abstract protected double getAllowable();
    abstract protected String getRole();

    public void setSuccessor(PurchasePower successor) {
        this.successor = successor;
    }

    public void processRequest(PurchaseRequest request){
        if (request.getAmount() < this.getAllowable()) {
            System.out.println(this.getRole() + " will
                approve $" + request.getAmount());
        } else if (successor != null) {
            // se non e' nostra responsabilita' passiamo al
            // nostro successore di grado superiore
            successor.processRequest(request);
        }
    }
}

```

Quattro implementazioni della classe astratta con quattro ruoli: Manager, Director, Vice President, President .

```

//Manager
class ManagerPPower extends PurchasePower {

```

```

        protected double getAllowable(){
            return BASE*10;
        }

        protected String getRole(){
            return "Manager";
        }
    }

//Direttore
class DirectorPPower extends PurchasePower {

    protected double getAllowable(){
        return BASE*20;
    }

    protected String getRole(){
        return "Director";
    }
}

//Vicepresidente
class VicePresidentPPower extends PurchasePower {

    protected double getAllowable(){
        return BASE*40;
    }

    protected String getRole(){
        return "Vice President";
    }
}

//Presidente
class PresidentPPower extends PurchasePower {

    protected double getAllowable(){
        return BASE*60;
    }

    protected String getRole(){
        return "President";
    }
}

```

La classe PurchaseRequest contiene i dati di una richiesta d'acquisto.

```
class PurchaseRequest {  
    private double amount;  
    private String purpose;  
  
    public PurchaseRequest(double amount, String  
                           purpose) {  
        this.amount = amount;  
        this.purpose = purpose;  
    }  
  
    public double getAmount() {  
        return amount;  
    }  
    public void setAmount(double amt) {  
        amount = amt;  
    }  
  
    public String getPurpose() {  
        return purpose;  
    }  
    public void setPurpose(String reason) {  
        purpose = reason;  
    }  
}  
  
class CheckAuthority {  
    public static void main(String[] args) {  
        ManagerPPower manager = new ManagerPPower();  
        DirectorPPower director = new DirectorPPower();  
        VicePresidentPPower vp = new  
            VicePresidentPPower();  
        PresidentPPower president = new  
            PresidentPPower();  
  
        //creiamo la catena di responsabilita'  
        manager.setSuccessor(director);  
        director.setSuccessor(vp);  
        vp.setSuccessor(president);  
  
        // Press Ctrl+C to end.  
        try {  
            while (true) {  
                System.out.println("Enter the amount to  
                                  check who should approve your  
                                  expenditure.");
```

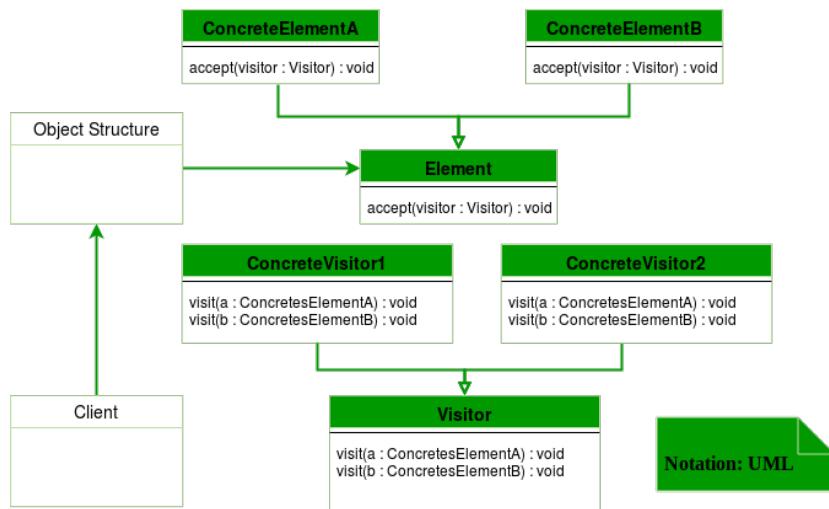
```
        System.out.print(">");  
        double d = Double.parseDouble(new  
            BufferedReader(new  
                InputStreamReader(System.in)).readLine());  
        manager.processRequest(new  
            PurchaseRequest(d, "General"));  
    }  
} catch(Exception e) {  
    System.exit(1);  
}  
}
```

Il Chain of Responsibility permette di disaccoppiare il mittente della richiesta e i suoi destinatari e viene comunemente usato nei sistemi Windows per gestire eventi come i clic del mouse e gli eventi della tastiera. Il principale svantaggio è che l'esecuzione della richiesta non è garantita: può cadere alla fine della catena se nessun oggetto la gestisce.

3.3.19 Visitors Pattern

Si usa quando si deve eseguire un'operazione su un gruppo di oggetti simili. Possiamo spostare la logica operativa dagli oggetti ad un'altra classe. È composto principalmente da due parti:

- un metodo Visit() implementato dal visitatore e chiamato per ogni elemento della struttura dati
 - classi visitabili che forniscono metodi Accept() che accettano un visitatore



La classe Client è un consumatore delle classi del design pattern visitor. Ha accesso agli oggetti della struttura dati e può istruirli ad accettare un Visitatore per eseguire l'elaborazione appropriata. Il Visitor è un'interfaccia o una classe astratta utilizzata per dichiarare le operazioni di visita per tutti i tipi di classi visitabili. ConcreteVisitor: per ogni tipo di visitatore devono essere implementati tutti i metodi di visita dichiarati nel visitatore astratto. Ogni visitatore sarà responsabile di operazioni diverse. Visitable è un'interfaccia che dichiara l'operazione di accettazione. È il punto di ingresso che consente a un oggetto di essere "visitato" dall'oggetto visitatore. Le classi ConcreteVisitable implementano l'interfaccia o la classe Visitable e definiscono l'operazione accept. L'oggetto visitatore viene passato a questo oggetto utilizzando l'operazione accept.

```

interface ItemElement{
    //ogni elemento della collezione implementa accept
    public int accept(ShoppingCartVisitor visitor);
}

class Book implements ItemElement{
    private int price;
    private String isbnNumber;

    public Book(int cost, String isbn)
    {
        this.price=cost;
        this.isbnNumber=isbn;
    }

    public int getPrice()
    {
        return price;
    }

    public String getIsbnNumber()
    {
        return isbnNumber;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        return visitor.visit(this);
    }
}

```

```

class Fruit implements ItemElement{
    private int pricePerKg;
    private int weight;
    private String name;

    public Fruit(int priceKg, int wt, String nm)
    {
        this.pricePerKg=priceKg;
        this.weight=wt;
        this.name = nm;
    }

    public int getPricePerKg()
    {
        return pricePerKg;
    }

    public int getWeight()
    {
        return weight;
    }

    public String getName()
    {
        return this.name;
    }

    @Override
    public int accept(ShoppingCartVisitor visitor)
    {
        // punto di accesso all'oggetto
        return visitor.visit(this);
    }
}

interface ShoppingCartVisitor
{
    int visit(Book book);
    int visit(Fruit fruit);
}

class ShoppingCartVisitorImpl implements
    ShoppingCartVisitor
{

```

```

@Override
public int visit(Book book)
{
    int cost=0;
    //apply 5$ discount if book price is
    //greater than 50
    if(book.getPrice() > 50)
    {
        cost = book.getPrice()-5;
    }
    else
        cost = book.getPrice();

    System.out.println("Book
        ISBN::"+book.getIsbnNumber() + " cost
        =" +cost);
    return cost;
}

@Override
public int visit(Fruit fruit)
{
    int cost =
        fruit.getPricePerKg()*fruit.getWeight();
    System.out.println(fruit.getName() + "
        cost = "+cost);
    return cost;
}

class ShoppingCartClient
{

    public static void main(String[] args)
    {
        ItemElement[] items = new
            ItemElement[]{new Book(20, "1234"),
        new Book(100, "5678"), new Fruit(10, 2,
            "Banana"), new Fruit(5, 5, "Apple")};

        int total = calculatePrice(items);
        System.out.println("Total Cost =
            "+total);
    }
}

```

```

private static int calculatePrice(ItemElement[]
    items)
{
    ShoppingCartVisitor visitor = new
        ShoppingCartVisitorImpl();
    int sum=0;
    for(ItemElement item : items)
    {
        sum = sum + item.accept(visitor);
    }
    return sum;
}

```

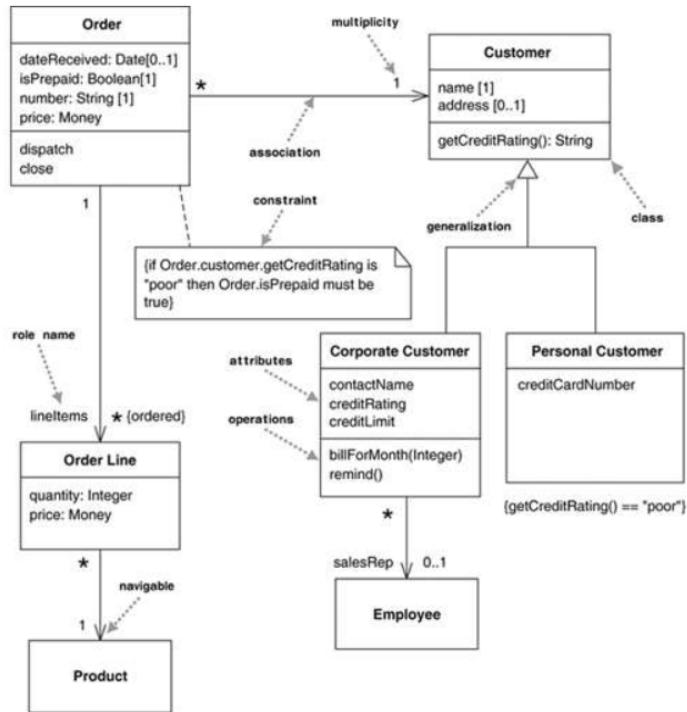
Il Visitor Pattern consente di aggiungere operazioni a una struttura composita senza modificare la struttura stessa ma poiché è coinvolta la funzione di attraversamento, le modifiche alla struttura composita sono più difficili.

3.3.20 Pattern Together

I Design Pattern possono essere utilizzati insieme, combinati all'interno di una stessa soluzione progettuale, per risolvere problemi "complessi".

3.4 UML - Class Diagram

Un class diagram è una notazione grafica che descrive i tipi di oggetti presenti nel sistema e le relazioni statiche che esistono tra loro. Mostrano anche le proprietà e le operazioni di una classe ed i vincoli che si applicano alle relazioni.



Le proprietà rappresentano le caratteristiche strutturali di una classe e appaiono in due notazioni distinte:

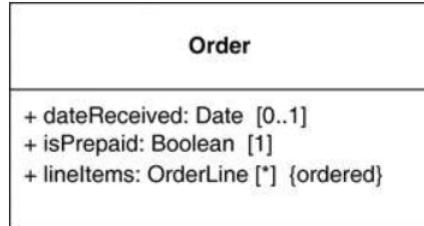
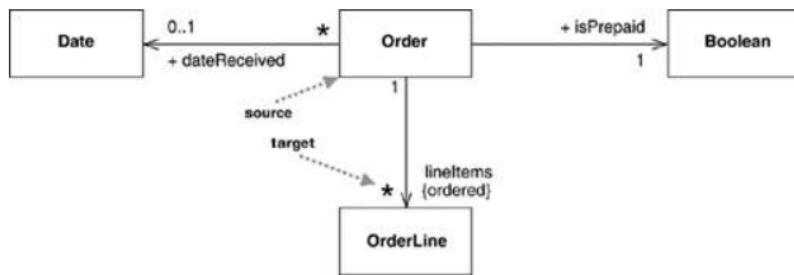
- **Attributi**

Describe una proprietà come una riga di testo. La forma completa di un attributo è visibility name: type multiplicity = default {property - string} (name: String [1] = "Untitled" {readOnly}) ma solo il nome è necessario. Il marcitore di visibilità indica se l'attributo è public (+), private (-), protected (# - gli elementi delle classi sono accessibili ai metodi che fanno parte della classe e anche ai metodi dichiarati su qualsiasi classe che eredita dalla classe) o package (~) (se il marcitore non viene indicato o rappresenta il default(+)) o il significato è talmente banale che viene omesso). Di solito è meglio evitare gli attributi pubblici, ma ci sono sempre delle eccezioni alla regola, per esempio quando l'attributo è una costante che può essere usata da diverse classi. Il nome corrisponde al nome di un campo in un linguaggio di programmazione. Il tipo indica una restrizione sul tipo

di oggetto che può essere inserito nell'attributo. Il valore predefinito è il valore di un oggetto appena creato e la stringa property-string consente di indicare proprietà aggiuntive per l'attributo.

- **Associazione**

È una linea continua tra due classi, diretta dalla classe di origine alla classe di destinazione. Il nome della proprietà si trova all'estremità dell'associazione, insieme alla sua molteplicità (la molteplicità si ha solo sull'estremità della classe d'origine)



N.B. In Java non possiamo implementare la molteplicità delle classi.

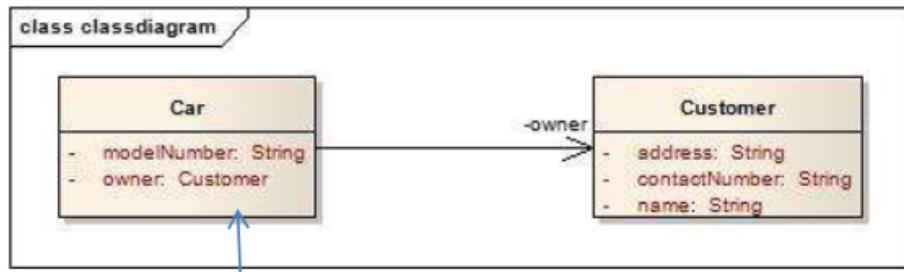
La regola generale nell'uso del class diagram è che usiamo gli attributi per le piccole cose e le associazioni per classi più significative. La molteplicità di una proprietà è un'indicazione di quanti oggetti possono riempire la proprietà e sono definiti con un limite inferiore e un limite superiore

- 1..1 → 1 (obbligatoria)
- 0..1 (facoltativa)
- * (tanti)

Per impostazione predefinita, gli elementi di una molteplicità multivariata formano un insieme e quindi non hanno un ordine preciso. Se l'ordine degli ordini ha un significato, è necessario aggiungere *{ordered}* alla fine dell'associazione

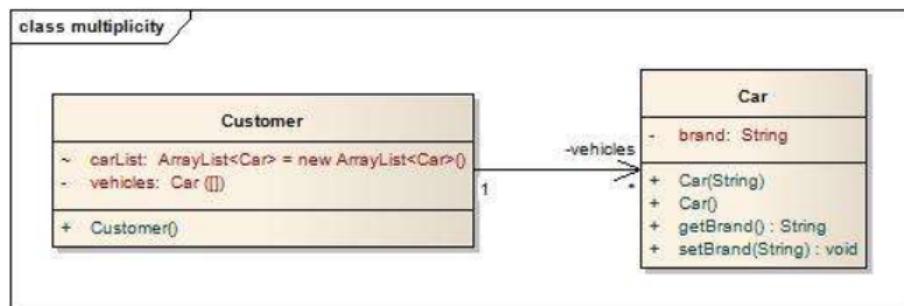
e se si vogliono consentire i duplicati, aggiungere `{nouunique}`. La molteplicità predefinita di un attributo è [1].

Come per qualsiasi altra cosa nell'UML, non c'è un solo modo di interpretare le proprietà nel codice. L'uso di campi privati è un'interpretazione del diagramma molto incentrata sull'implementazione del diagramma o un'interpretazione più orientata all'interfaccia potrebbe invece concentrarsi sui metodi getter o setter. Se un attributo è multivalente, i dati in questione sono una collezione che potrebbe essere una lista o un set (se unordered).



```

public class Customer {
    private String name;
    private String address;
    private String contactNumber;
}
public class Car {
    private String modelNumber;
    private Customer owner;
}
  
```



```

public class Car {
    private String brand;
  
```

```

public Car(String brands){
    this.brand = brands;
}
public Car() {
}
public String getBrand() {
    return brand;
}
public void setBrand(String brand) {
    this.brand = brand;
}
}
public class Customer {
    private Car[] vehicles;
    Protected ArrayList<Car> carList = new
    ArrayList<Car>();
    public Customer(){
    }
}

```

Un'associazione bidirezionale è una coppia di proprietà collegate tra loro come inversi.

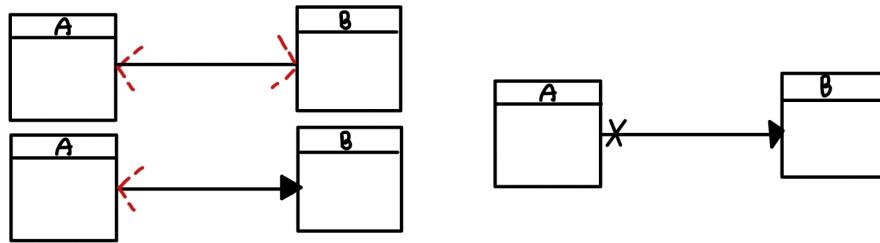


In alternativa all'etichettatura di un'associazione in base a una proprietà, molte persone, amano etichettare l'associazione utilizzando una frase verbale.



Implementare un'associazione bidirezionale in un linguaggio di programmazione è spesso un po' complicato, perché bisogna essere sicuri che entrambe le proprietà siano mantenute sincronizzate.

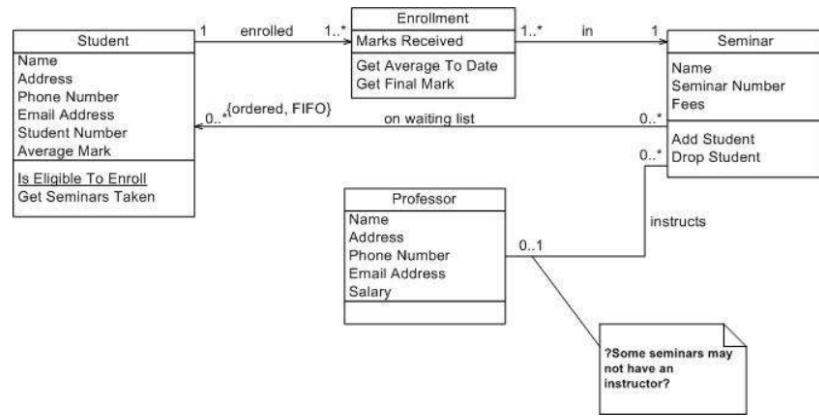
N.B. Usare associazioni senza l'utilizzo di frecce può generare ambiguità perché il default è ancora vivo. Per bloccare la navigabilità dovremmo utilizzare la notazione a destra nell'immagine sottostante; tuttavia questa notazione è poco utilizzata perché appesantisce il diagramma. Nella maggior parte dei casi se mettiamo solo una freccia è perché ammettiamo la navigabilità solo in quella determinata direzione.



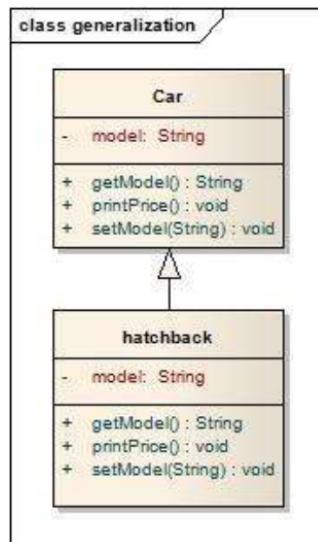
All'interno delle classi oltre alle proprietà dobbiamo definire le **operazioni** che sono le azioni che una classe sa eseguire. Nella maggior parte dei casi corrispondono ovviamente ai metodi di una classe (getter e setter non vengono mostrati). La sintassi UML completa è: visibility name (parameter-list) : return-type {property-string}:

- Il marcatore di visibilità può essere pubblico (+) o privato (-);
- Il nome è una stringa.
- L'elenco dei parametri è l'elenco dei parametri dell'operazione.
- Il tipo di ritorno è il tipo del valore restituito, se esiste.
- La stringa property indica i valori delle proprietà che si applicano all'operazione data (ad esempio, abstract).
- I parametri dell'elenco dei parametri sono annotati in modo simile agli attributi. La forma è: direzione nome: tipo = valore predefinito

Nonostante metodo e operazione sono due termini che vengono comunemente interscambiati hanno due significati diversi infatti un'operazione è qualcosa che viene invocata su un oggetto (la dichiarazione della procedura) mentre un metodo è il corpo di una procedura.



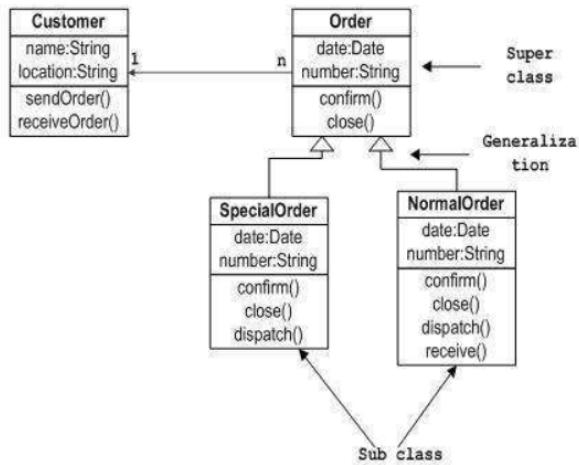
Essendo il class diagram molto utilizzato nella progettazione di software ad oggetti è possibile utilizzare la generalizzazione e ereditarietà. La sottoclassse eredita tutte le caratteristiche della superclasse e può sovrascrivere qualsiasi metodo della superclasse. Un principio importante per utilizzare efficacemente l'ereditarietà è la sostituibilità (dovrei essere in grado di sostituire la sottoclasse in qualsiasi codice che richiede la superclasse e tutto dovrebbe funzionare bene).



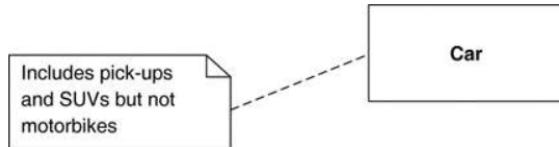
```

public class Car {
    private String model;
    public void printPrice() {
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}
public class hatchback extends Car {
    private String model;
    public void printPrice() {
        System.out.println("Hatchback Price");
    }
    public String getModel() {
        return model;
    }
    public void setModel(String model) {
        this.model = model;
    }
}

```



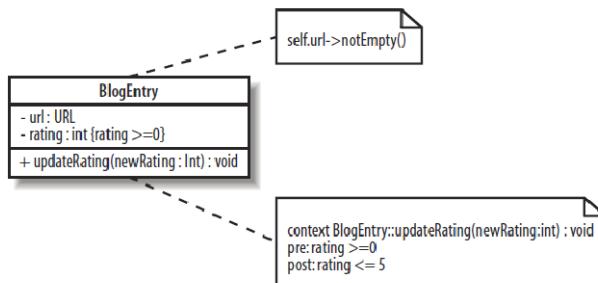
All'interno del class diagram possiamo anche inserire delle note che fungono da commento e possono stare da sole o essere collegate con una linea tratteggiata agli elementi che commentano.



Molte volte tra due classi vengono a formarsi delle dipendenze cioè che il cambiamento di una classe può causare il cambiamento anche dell'altra classe. Esistono diverse tipi di dipendenza:

- «call», L'origine chiama un'operazione nella destinazione.
- «create», L'origine crea istanze della destinazione.
- «use», La sorgente richiede la destinazione per la sua implementazione

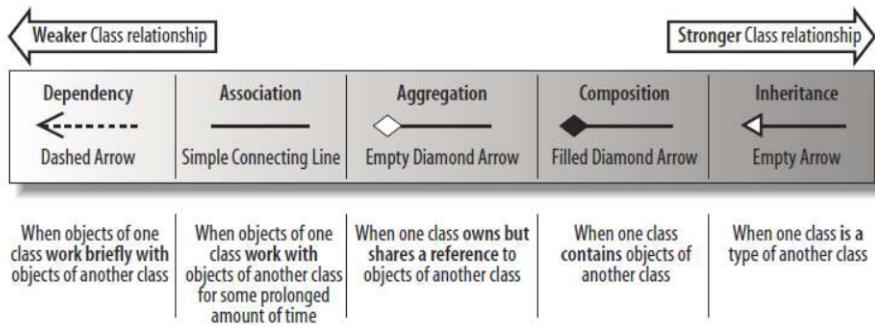
Nel disegnare un class diagram stiamo indicando e definendo dei vincoli. I costrutti di base dell'associazione, dell'attributo e della generalizzazione fanno molto per specificare i vincoli importanti, ma non possono indicare tutti i vincoli; quest'ultimi però devono essere catturati; il diagramma delle classi è un buon luogo per farlo. L'UML consente di utilizzare qualsiasi cosa per descrivere i vincoli, l'unica regola è quella di inserirli all'interno di parentesi graffe () o di utilizzare Object Constraint Language (OCL) (linguaggio molto formale e quindi poco utilizzato).



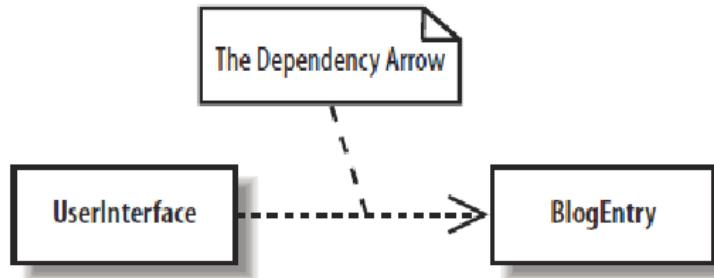
I diagrammi delle classi sono la spina dorsale dell'UML ma il problema principale è che sono così ricchi che possono essere usati in modo eccessivo. Dobbiamo cercare di utilizzare solo le notazioni semplici ed essenziali: classi, associazioni, attributi, generalizzazione e vincoli. Un altro pericolo con i diagrammi delle classi è che ci si possa concentrare esclusivamente sulla struttura e ignorare il comportamento. Pertanto, quando si disegnano diagrammi di classe per capire il software, fatelo sempre insieme a con qualche forma di tecnica comportamentale.

3.4.1 Class Diagram Advanced

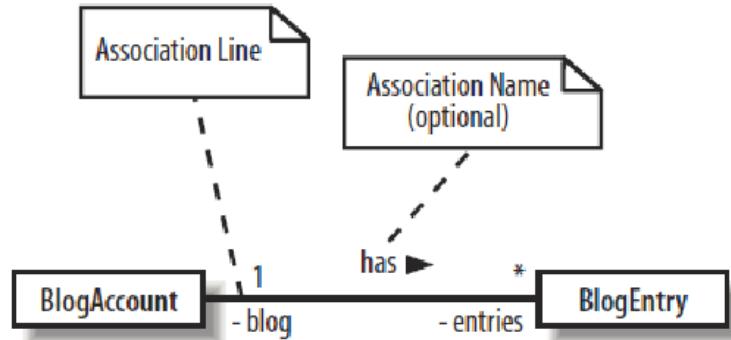
Le classi non vivono nel vuoto ma lavorano insieme utilizzando diversi tipi di relazioni.



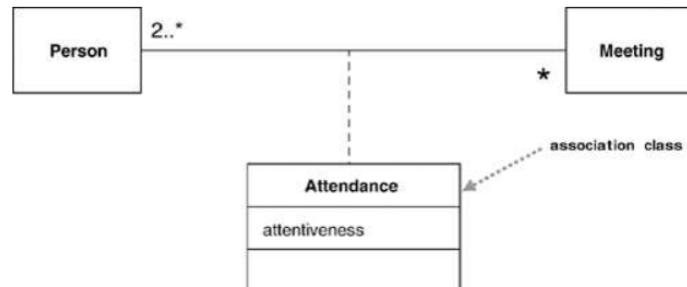
Una **dipendenza** implica solo che gli oggetti di una classe possono lavorare insieme. È considerata la più debole relazione diretta che può esistere tra due classi. La relazione di dipendenza viene spesso utilizzata quando si ha una classe con una serie di funzioni di utilità generale, come nel caso delle classi di matematica (java.math).



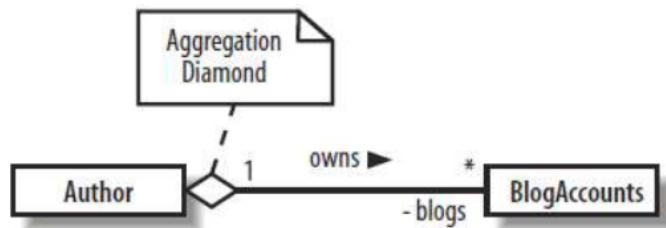
L'**associazione** implica che una classe contiene effettivamente un riferimento a uno o più oggetti dell'altra classe sotto forma di attributo. La navigabilità è spesso applicata a una relazione di associazione per descrivere quale classe contiene l'attributo che supporta la relazione.



Possiamo anche utilizzare delle classi di associazione che consentono di aggiungere attributi, operazioni e altre caratteristiche alle associazioni. Infatti la classe di associazione aggiunge un ulteriore vincolo, in quanto può esistere solo un'istanza della classe di associazione tra due oggetti partecipanti. Sono particolarmente utili nei casi complessi in cui si vuole mostrare che una classe è correlata a due classi perché queste due classi hanno una relazione tra loro.

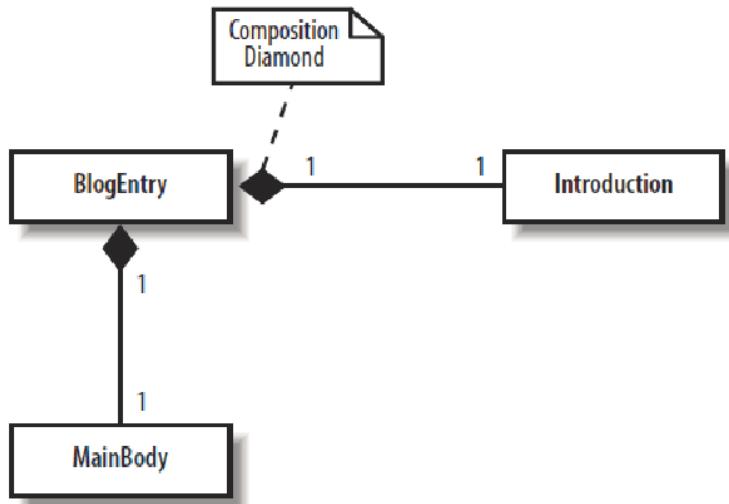


L'aggregazione è la relazione part-of (membro di). L'aggregazione è in realtà solo una versione più forte dell'associazione e indica che una classe possiede effettivamente, ma può condividere, oggetti di un'altra classe. Il problema è capire quale sia la differenza tra aggregazione e associazione visto che non c'è praticamente nessuna differenza nell'implementazione. È un tipo di relazione in cui il figlio è indipendente dal genitore.



Il rapporto tra un autore e i suoi blog è molto più forte di una semplice associazione. Un autore è il proprietario dei suoi blog e, anche se può condividerli con altri autori, alla fine i suoi blog sono suoi e se decide di rimuovere uno dei suoi blog, può farlo.

La composizione è una parte dell'aggregazione e rappresenta la relazione intero-parte. Rappresenta la dipendenza tra un composito (genitore) e le sue parti (figli), il che significa che se il composito viene eliminato, anche le sue parti verranno eliminate. Esiste tra oggetti simili.

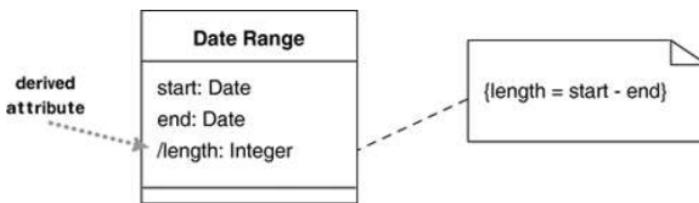


Le sezioni dell'introduzione e del corpo principale di un blog sono in realtà parti del blog stesso e di solito non vengono condivise con altre parti del sistema. Se l'articolo del blog viene cancellato, vengono cancellate anche le parti corrispondenti. Questo è esattamente l'obiettivo della composizione: si modellano le parti interne che compongono una classe.

La generalizzazione è la forma più forte di relazione tra classi, perché crea un accoppiamento stretto tra le classi. Le classi genitore descrivono un tipo più generale, che viene poi specializzato nelle classi figlio che eredita e riutilizza tutti

gli attributi e i metodi che il genitore contiene e che siano pubblici o protetti. La generalizzazione offre quindi un ottimo modo per esprimere che una classe è un tipo di un'altra classe. Tuttavia, dal momento che una classe figlia può vedere la maggior parte degli interni della sua genitrice, essa diventa strettamente accoppiata all'implementazione del suo genitore. Uno dei principi di una buona progettazione orientata agli oggetti è evitare di accoppiare strettamente le classi e per questo motivo, è buona norma utilizzare la generalizzazione solo quando una classe è davvero un tipo più specializzato di un'altra classe e non solo per comodità.

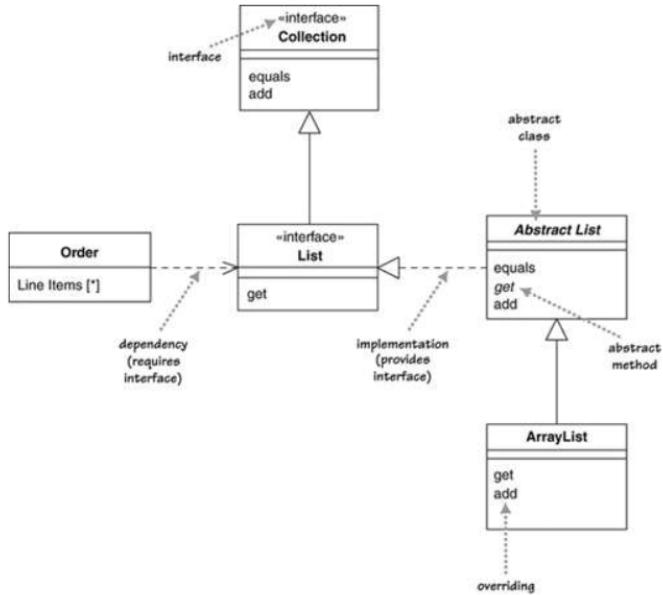
All'interno di una classe possiamo inserire anche delle **proprietà derivate**, calcolate sulla base del valore di altre proprietà interne alla classe.



Visto che il class diagram è utilizzato principalmente per la progettazione OO possiamo definire, come in tutti i linguaggi ad oggetti, classi astratte e interfacce. Le classi che non possono essere istanziate sono modellate come **classi astratte** (solo le loro sottoclassi possono essere istanziate). Il modo più comune per indicare una classe o un'operazione astratta è quello di mettere in corsivo il nome (in alternativa si può usare l'etichetta: abstract). Inoltre si possono rendere astratte anche le proprietà (un'operazione astratta non offre di per sé alcuna implementazione e richiede un'implementazione nelle sottoclassi concrete).

Un'interfaccia non ha un'implementazione né istanze dirette e rappresenta un contratto. Le classi che stipulano questo contratto, cioè le classi che implementano l'interfaccia, si obbligano a fornire il comportamento specificato dall'interfaccia. Le classi hanno due tipi di relazioni con le interfacce: fornire e richiedere.

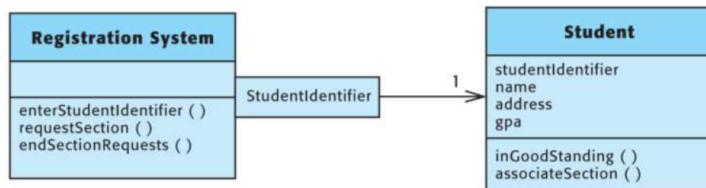
- Una classe fornisce un'interfaccia se è sostituibile con l'interfaccia stessa.
- Una classe richiede un'interfaccia se ha bisogno di un'istanza di quell'interfaccia per poter funzionare.



Il linguaggio UML cerca spesso di ridurre il numero di simboli e di utilizzare keyword. Le parole chiave sono di solito mostrate come testo tra le parentesi graffe. Vediamo ora due keyword molto utilizzate:

- `readOnly`, per contrassegnare una proprietà che può essere letta solo dai client e che non può essere aggiornata.
- `frozen`, non possono cambiare durante la vita di un oggetto (spesso chiamate immutabili); è possibile applicare la parola chiave anche ad una classe per indicare che tutte le proprietà di tutte le istanze sono congelate.

All'interno del class diagram possiamo inserire anche un'associazione qualificata cioè l'equivalente UML di un concetto di programmazione noto come array associativi, mappe, hash e dizionari.



In questo caso possiamo accedere alle informazioni sullo studente attraverso l'attributo `StudentIdentifier` della classe `RegistrationSystem`.

Infine possiamo inserire delle **data classes** e delle **enumeration**:

- Il tipo di dati è visualizzato allo stesso modo di una classe, con la differenza che il nome del tipo di dati è annotato con la keyword «datatype».(anche i tipi di dato definiti da utente)

«primitive» Float	«datatype» Date
round(): void	day month

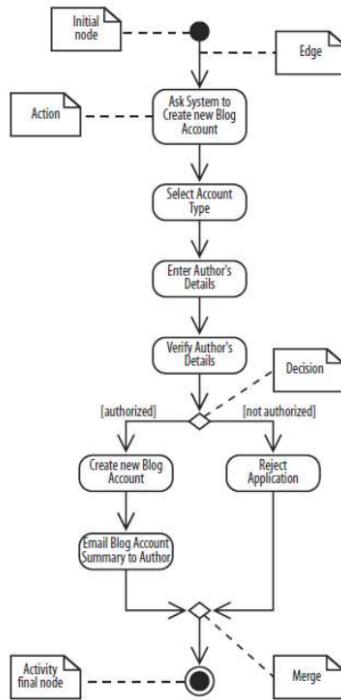
- Le enumerazioni sono tipi di dati in cui valori sono definiti in un elenco. Le enumerazioni sono utilizzate per mostrare un insieme fisso di valori che non hanno proprietà diverse dal loro valore simbolico e sono indicate come classe con la parola chiave «enumerazione».

«enumeration» Color	«enumeration» AcademicDegree
red white blue	bachelor master phd

3.5 UML - Activity Diagram

I diagrammi di attività consentono di specificare come il sistema raggiungerà i suoi obiettivi e sono particolarmente utili nel modellare i processi aziendali. Partendo da un determinato caso d'uso possiamo definire tutta la sequenza delle attività e graficare l'Activity Diagram. Vediamo subito un esempio: il caso d'uso è creare un nuovo account blog e per far ciò dobbiamo eseguire le seguenti azioni

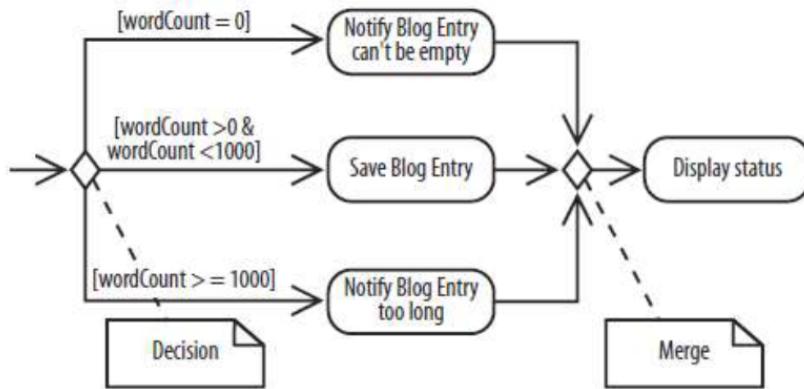
1. L'amministratore chiede al sistema di creare un nuovo account blog.
2. L'amministratore seleziona un tipo di account.
3. L'amministratore inserisce i dati dell'autore.
4. I dati dell'autore vengono verificati utilizzando il database delle credenziali dell'autore.
5. Il nuovo account del blog viene creato.
6. Un riepilogo dei dettagli del nuovo account blog viene inviato via e-mail all'autore.



All'interno del diagramma sono state introdotte anche le seguenti condizioni grazie all'utilizzo di un nodo di decisione:

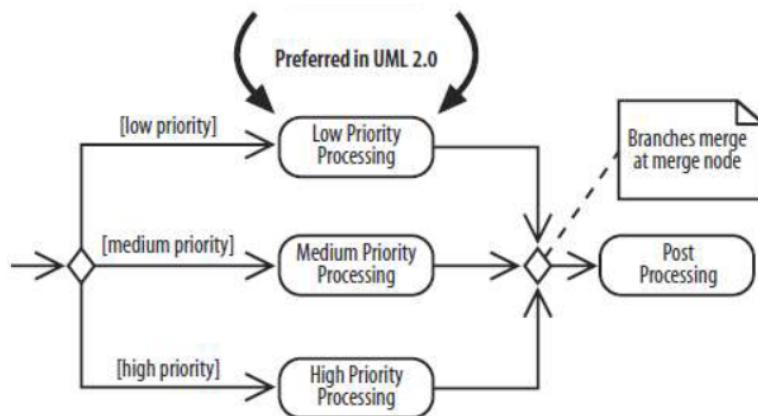
- Il database delle credenziali dell'autore non verifica i dati dell'autore.
- La richiesta di un nuovo account blog dell'autore viene respinta.

Le decisioni vengono utilizzate quando si desidera eseguire una sequenza diversa di azioni seconda di una determinata condizione. Ogni bordo ramificato contiene una condizione di guardia scritta tra parentesi che determina quale bordo viene preso dopo un nodo di decisione. I flussi ramificati si uniscono in un nodo di unione, che segna la fine del comportamento condizionale iniziato nel nodo di decisione.

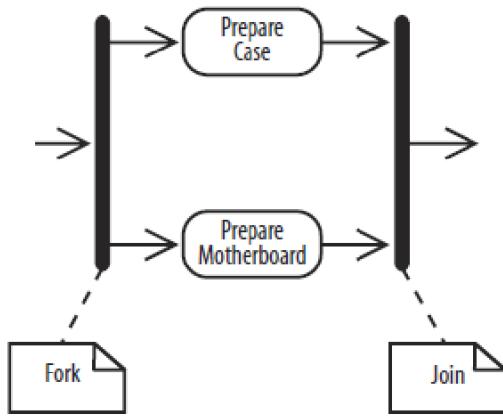


Bisogna stare molto attenti nel definire le condizioni di guardia perché nel caso mancasse qualche particolare condizione l'activity rimarrebbe bloccato al nodo di decisione.

Grazie all'activity diagram e al nodo di decisione possiamo anche svolgere più task in parallelo.

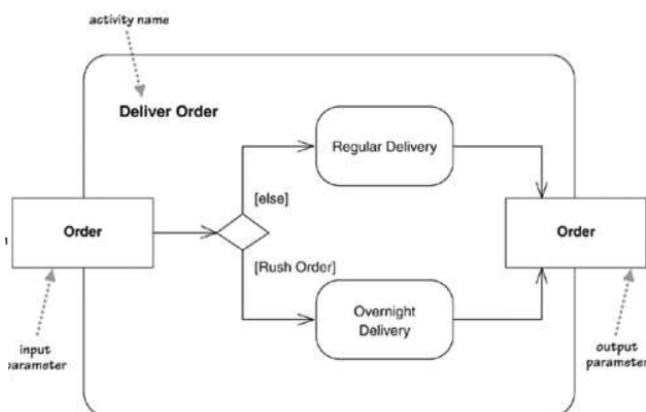


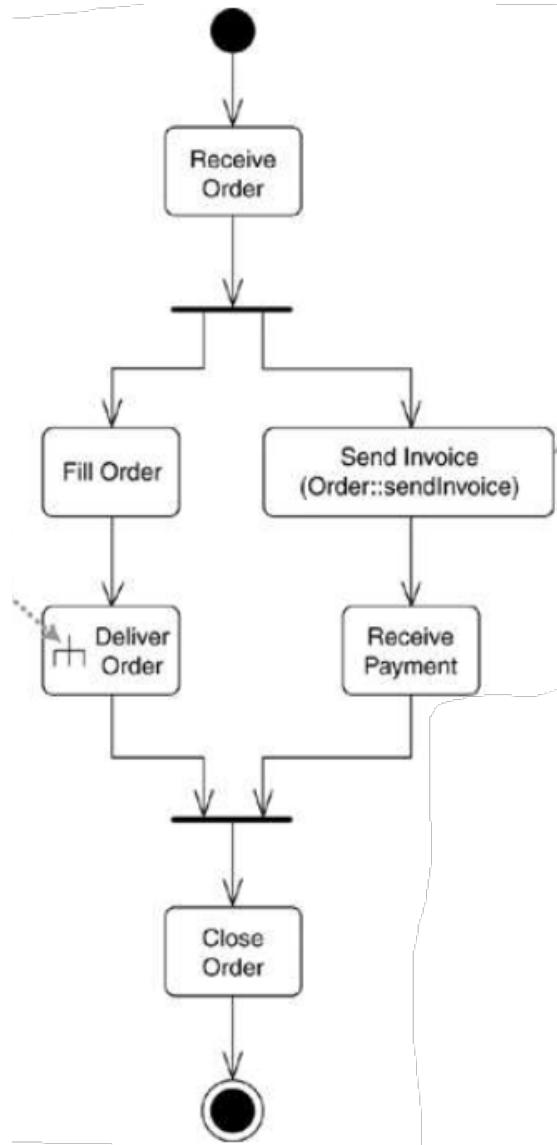
Quando le azioni avvengono in parallelo, non significa necessariamente che finiranno nello stesso momento, anzi è molto probabile che un compito finisca prima dell'altro. Tuttavia, possiamo introdurre un join che impedisce al flusso di proseguire oltre la giunzione fino a quando tutti i flussi in entrata sono stati completati.



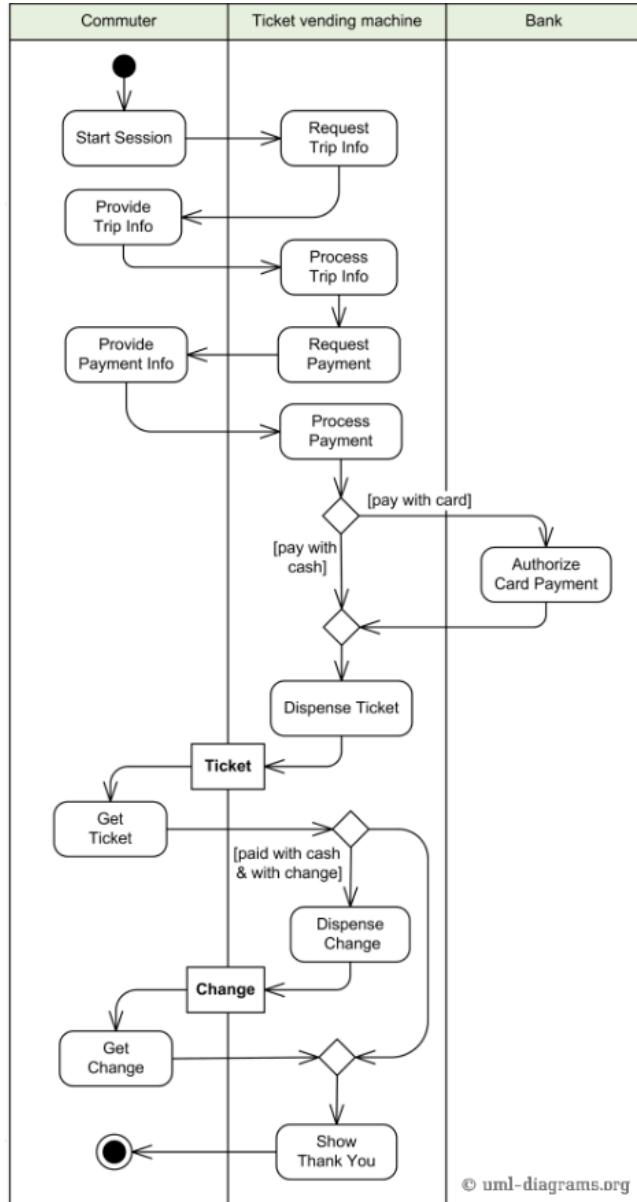
Le azioni possono essere implementate in 2 modi:

1. Scomposte in sottoattività (si mostra una sottoattività usando il simbolo `rake`).
2. Come metodi sulle classi con la sintassi `nome-classe::metodo-nome` (si può anche scrivere un frammento di codice nel simbolo dell'azione)



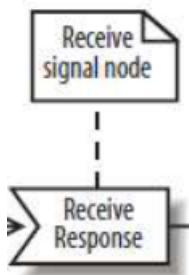


I diagrammi di attività indicano cosa succede, ma non dicono chi fa cosa e quindi non indicano quale classe è responsabile di ogni azione. Questo non è necessariamente un problema; spesso ha senso concentrarsi su ciò che viene fatto piuttosto che su chi fa quali parti. Se si vuole mostrare chi fa cosa, si può dividere un diagramma delle attività in partizioni, che mostrano quali azioni una classe o un'unità organizzativa esegue.

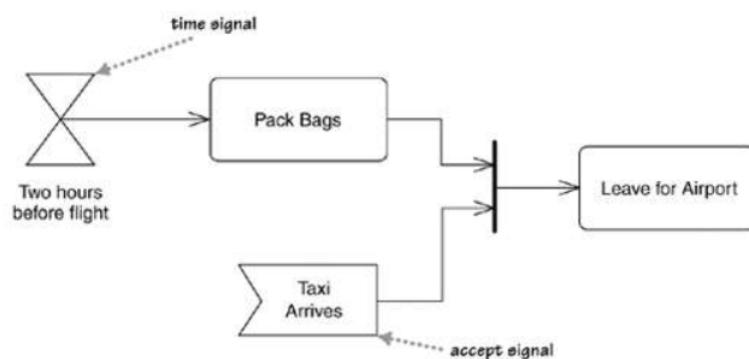
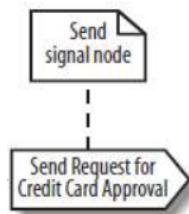


Nei diagrammi di attività, possiamo inserire anche dei segnali, i quali rappresentano le interazioni con i partecipanti esterni. Esistono tanti tipi di segnali ma noi possiamo categorizzare in 2 principali categorie:

1. Un segnale di ricezione ha l'effetto di svegliare un'azione nel diagramma delle attività.

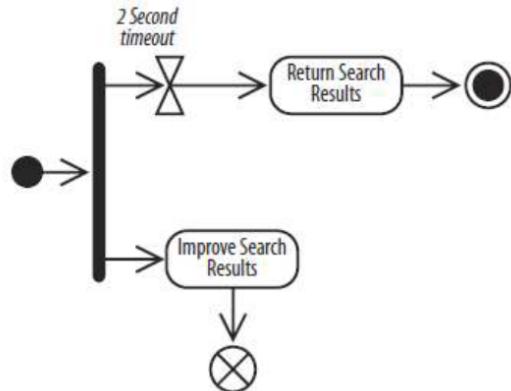


2. Invece i segnali di invio sono segnali inviati a un partecipante esterno (la risposta non viene modellata nel diagramma delle attività)

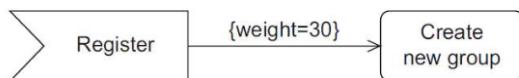


Un segnale temporale si verifica a causa del trascorrere del tempo. Tali segnali possono indicare la fine di un mese in un periodo finanziario o ogni microsecondo in un controllore in tempo reale.

Dalla versione 2.0 dell'UML un nodo finale di flusso termina il proprio percorso, non l'intera attività.



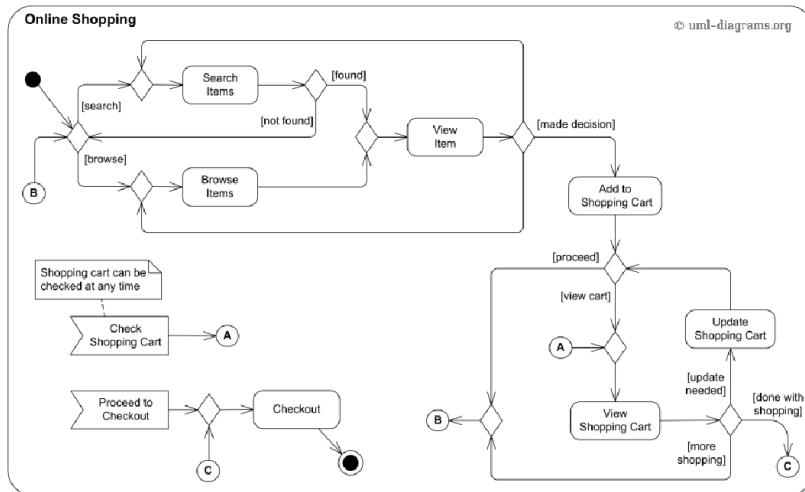
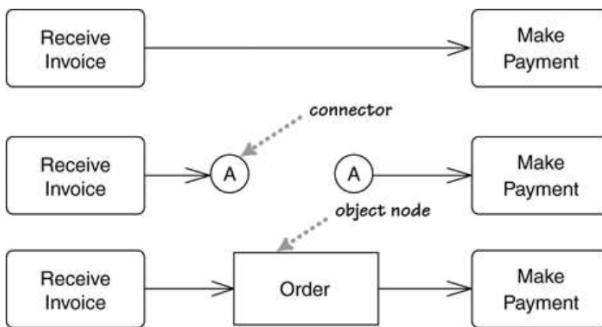
Per integrare gli aspetti del comportamento dinamico nel diagramma, abbiamo bisogno di una semantica dell'esecuzione, cioè dobbiamo specificare esattamente come viene eseguito un diagramma di attività. Il concetto di token è la base per la semantica di esecuzione del diagramma di attività. Un token è un meccanismo di coordinazione virtuale (non è un componente fisico del diagramma) che descrive l'esecuzione in modo esatto. Se un'azione riceve un token, l'azione è attiva e può essere eseguita; una volta terminata l'azione, essa passa il token attraverso il bordo ad un nodo successivo che attiverà la propria azione (il passaggio di un token può essere impedito da una guardia con valore false). Quando un'azione viene eseguita, di solito viene consumato un token di ciascuno dei bordi in entrata. In alternativa, si può assegnare un peso a un bordo per consentire un certo numero di token da consumare su quel bordo con una singola esecuzione (il peso di un bordo è specificato tra parentesi graffe con la parola chiave peso). Il peso è sempre un numero intero maggiore o uguale a zero: se il peso è zero, all o *, significa che tutti i gettoni presenti vengono consumati, invece se non viene specificato alcun peso, si assume 1 come valore predefinito.



Il nodo iniziale crea un token, che passa all'azione successiva, che esegue e passa il token alla successiva. Ma cosa succede al token con una fork? Arriva un token alla biforcazione, la quale produce un token su ogni suo flusso in uscita. Al contrario, in un join, all'arrivo di ogni token in entrata, non succede nulla finché tutti i token non appaiono al join; a quel punto un token viene passato sul flusso di uscita. È possibile visualizzare i token con le monete o i contatori che si muovono attraverso il diagramma che nella maggior parte dei casi facilitano la visualizzazione delle cose.

UML 2 utilizza i termini flusso e bordo come sinonimi per descrivere le con-

nessioni tra due azioni; il tipo più semplice di bordo è la semplice freccia tra due azioni. Se si hanno difficoltà a tracciare linee, si possono usare i connettori, che evitano semplicemente di dover disegnare una linea per tutta la distanza. I bordi più semplici passano un token che non ha altro significato se non quello di controllare il flusso. Tuttavia, è possibile far passare anche oggetti lungo gli spigoli; gli oggetti svolgono allora il ruolo di token, oltre a trasportare dati.

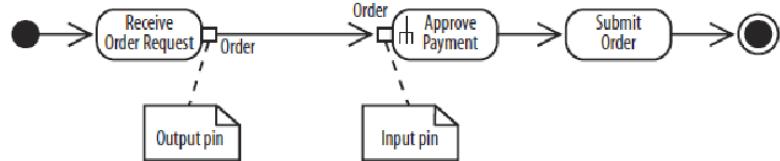


Man mano che si aggiungono dettagli al diagramma delle attività, il diagramma potrebbe diventare troppo grande oppure la stessa sequenza di azioni può verificarsi più di una volta. In questo caso, si può migliorare la leggibilità fornendo i dettagli di un'azione in un diagramma separato, consentendo al diagramma di livello superiore di essere meno ingombrante.

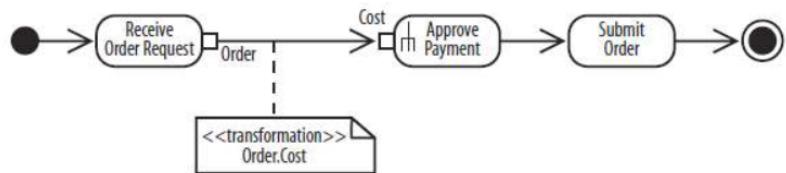
Le azioni possono anche avere dei parametri. Non é però necessario mostrare le informazioni sui parametri, ma se lo si desidera, si possono mostrare attraverso i pin:

- Un pin di ingresso significa che l'oggetto specificato è in ingresso a un'azione.

- Un pin di uscita significa che l'oggetto specificato è in uscita da un'azione.



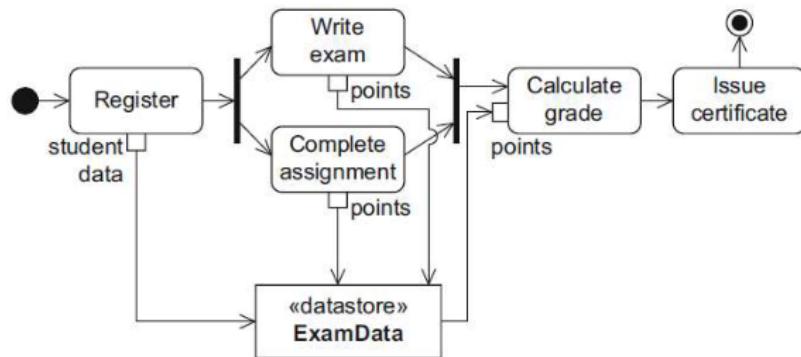
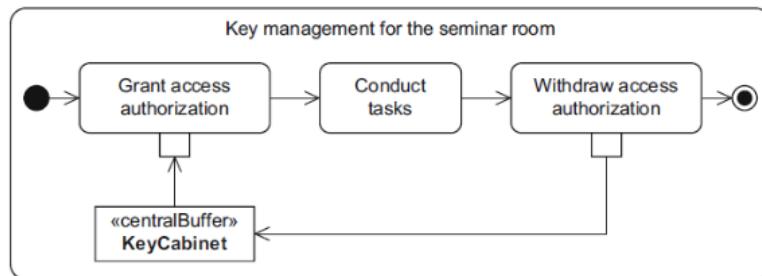
Quando si disegna un diagramma delle attività in modo rigoroso, è necessario assicurarsi che i parametri di uscita di un'azione corrispondano ai parametri di ingresso di un'altra azione. Se non corrispondono, si può indicare una trasformazione per passare da uno all'altro (deve essere un'espressione priva di effetti collaterali).



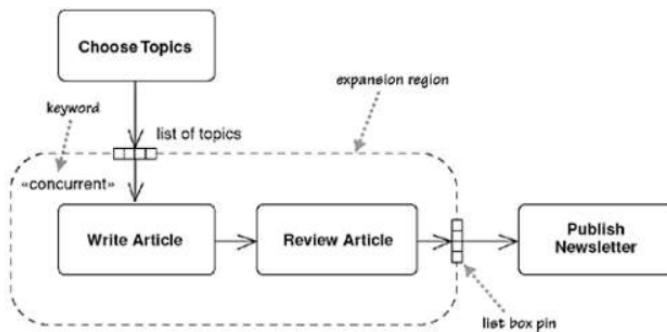
3.5.1 Nodi speciali

Il **buffer centrale** è un nodo oggetto speciale che gestisce il flusso di dati tra più sorgenti e più riceventi. Accetta i token di dati in arrivo dai nodi oggetto e li passa ad altri nodi oggetto ma a differenza dei pin e dei parametri delle attività, il buffer centrale non è vincolato ad azioni o attività. Quando un token di dati viene letto dal buffer centrale, viene cancellato e non può essere consumato di nuovo.

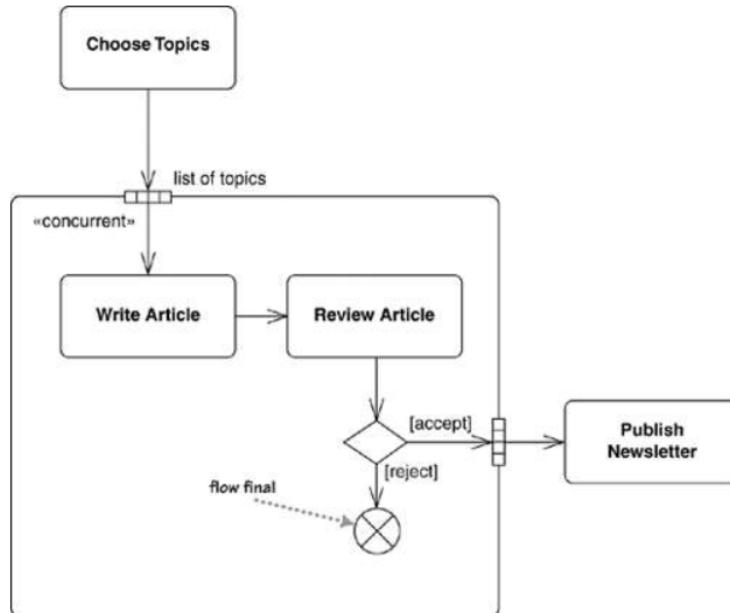
Esiste anche un altro nodo speciale il **data store**, nel quale tutti i token di dati che confluiscono sono salvati in maniera permanentemente, infatti vengono copiati prima di lasciare nuovamente l'archivio dati. Come una specie di database è possibile definire interrogazioni relative al contenuto dell'archivio dati sui bordi in uscita dall'archivio dati per estrarre dei dati.



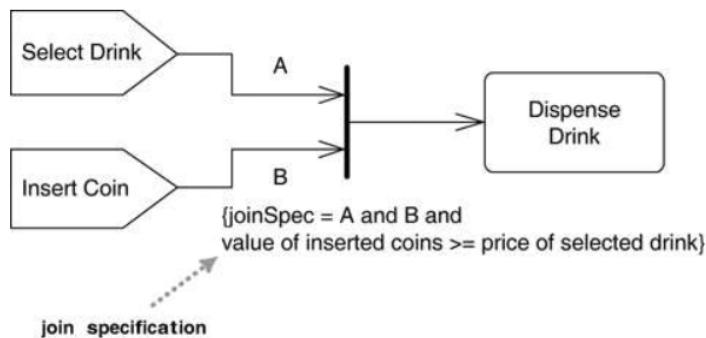
In situazioni in cui l'uscita di un'azione innesta più invocazioni di un'altra azione è utile una regione di espansione, cioè una regione che contrassegna un'area del diagramma delle attività in cui le azioni si verificano una volta per ogni elemento di una collezione (una sorta di ciclo).



Per fermare una regione di espansione iterativa prima della sua naturale conclusione possiamo inserire un nodo finale di flusso (una sorta di break).

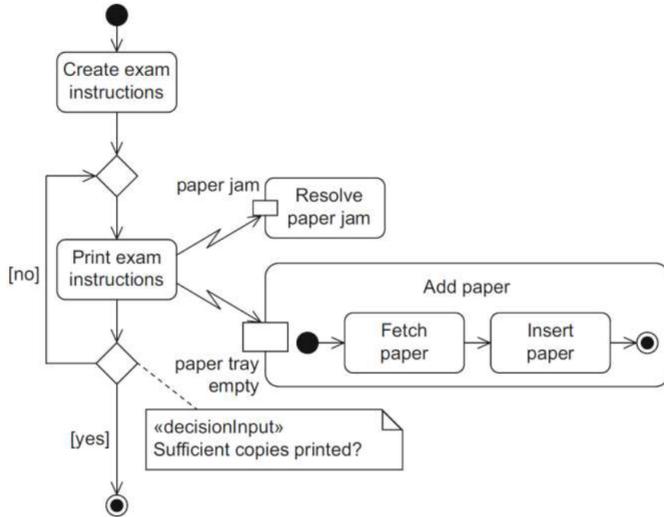


In alcuni casi, in particolare quando si dispone di un flusso con più token è utile avere una regola più complessa. Una specifica di join è un'espressione booleana collegata a una join cosicché ogni volta che un token arriva al join, viene valutata la specifica di join e se è vera, viene emesso un token di uscita

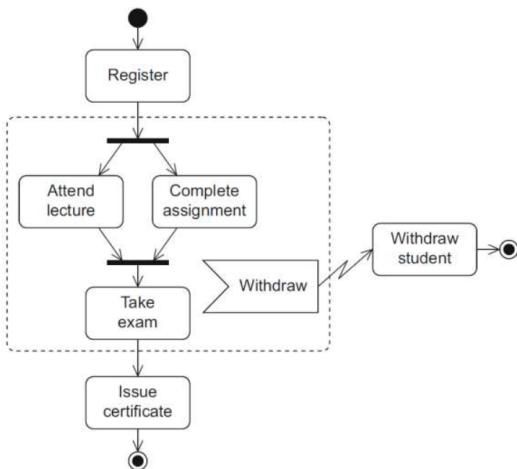


All'interno dell'activity diagram possiamo anche avere una gestione degli errori. Normalmente se si verifica un errore durante un'azione, l'esecuzione viene

interrotta e tutti i token vengono eliminati ma possiamo inserire un gestore di eccezioni che viene attivato quando si verifica un'eccezione. I gestori di eccezioni consentono di definire il modo in cui il sistema reagisce in una specifica situazione di errore (si specifica un gestore di eccezioni per un tipo specifico di errore).

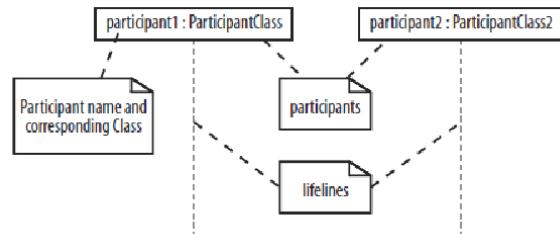


La regione di attività interrompibile offre un altro modo per gestire le eccezioni: è possibile definire un gruppo di azioni la cui esecuzione deve essere terminata (tutti i token vengono cancellati) immediatamente se si verifica un evento specifico. L'area di attività interrompibile è rappresentata da un rettangolo tratteggiato con angoli arrotondati che racchiude le azioni interessate; l'esecuzione di questi eventi viene monitorata per il verificarsi di un evento specifico, ad esempio un errore.

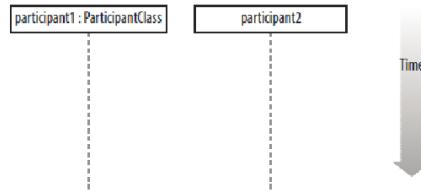


3.6 UML - Sequence Diagrams

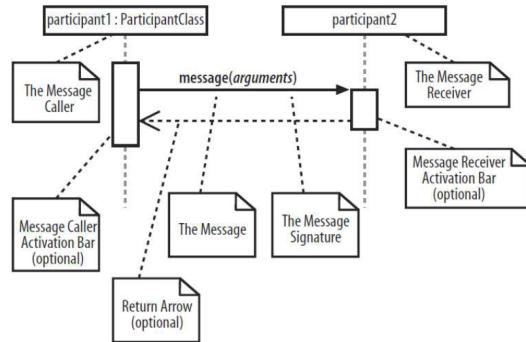
I diagrammi di sequenza sono un membro importante del gruppo conosciuto come diagrammi di interazione, i quali modellano le interazioni importanti tra le parti che compongono il sistema. Un diagramma di sequenza è costituito da un insieme di partecipanti che sono solitamente software oggetti nel senso tradizionale della programmazione orientata agli oggetti (nell'UML 2.0 hanno un senso più generale). La linea di vita di un partecipante afferma che la parte esiste in quel punto della sequenza. I partecipanti a un diagramma di sequenza possono essere denominati in diversi modi, la forma più utilizzata è la seguente: name [selector] : class_name



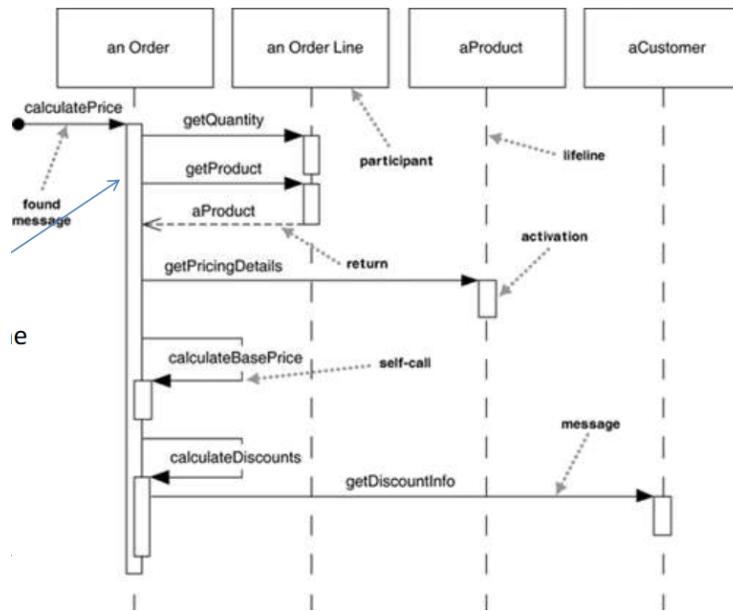
Il tempo in un diagramma di sequenza è una questione di ordine, non di durata infatti la quantità di spazio verticale occupata dall'interazione non ha nulla a che fare con la durata dell'interazione.



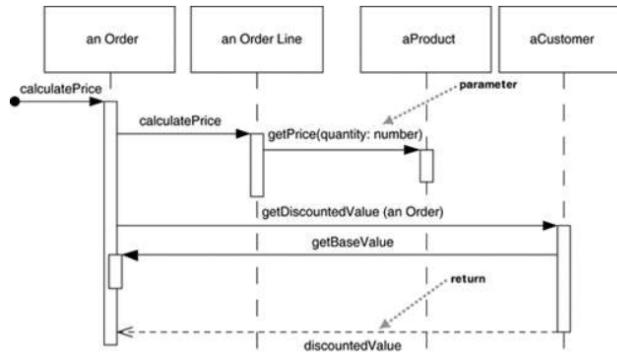
La parte più piccola di un'interazione è un evento. Tra i vari eventi possono essere scambiati dei messaggi (segnali) specificati con una freccia dal partecipante che manda il messaggio al partecipante che deve ricevere il messaggio.



Vediamo subito un esempio completo

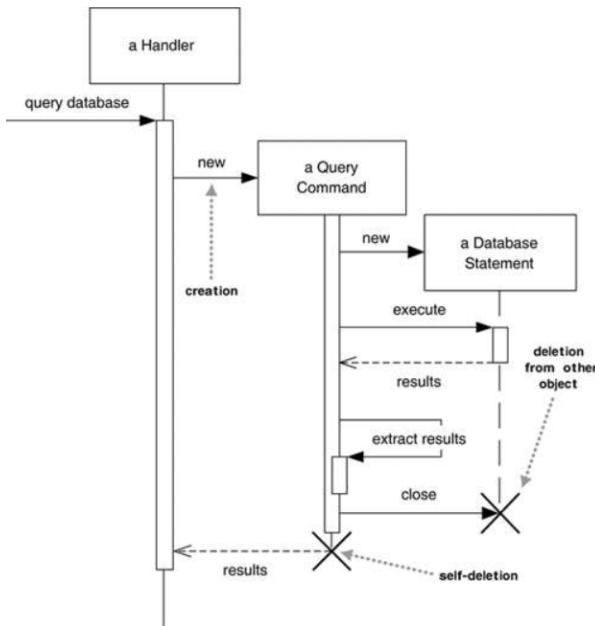


Il diagramma non mostra tutto molto bene. La sequenza di messaggi **getQuantity**, **getProduct**, **getPricingDetails** e **calculateBasePrice** deve essere eseguita per ogni riga d'ordine, mentre **calculateDiscounts** viene invocato una sola volta. Notiamo che ogni linea di vita ha una barra di attivazione che mostra quando il partecipante è attivo nell'interazione(opzionale). Le frecce di ritorno sono opzionali. Il primo messaggio non ha un partecipante che l'ha inviato, poiché proviene da una fonte indeterminata e viene chiamato found message.



Questi tipi di diagrammi non mostrano i dettagli degli algoritmi, ma rendono chiare le chiamate tra i partecipanti. Infatti il diagramma utilizza il controllo distribuito, cioè l'elaborazione è suddivisa tra molti partecipanti, ognuno dei quali esegue un frammento dell'algoritmo.

Per creare un partecipante, si disegna la freccia del messaggio direttamente nella casella del partecipante (il nome del messaggio è facoltativo). Invece l'eliminazione di un partecipante è indicata da una grande X. Una freccia di messaggio che va verso la X indica che un partecipante ne elimina esplicitamente un altro; una X alla fine di una linea di vita indica l'eliminazione di un partecipante. In un ambiente con garbage-collector, non si cancellano direttamente gli oggetti, ma vale comunque la pena di usare la X per indicare quando un oggetto non è più necessario ed è pronto per essere raccolto.

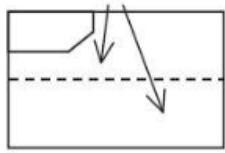


All'interno del sequence diagram possiamo specificare anche degli operatori definiti come **frammenti combinati**. In questo modo è possibile descrivere un certo numero di possibili percorsi di esecuzione in modo compatto e preciso.

Operator



Operands

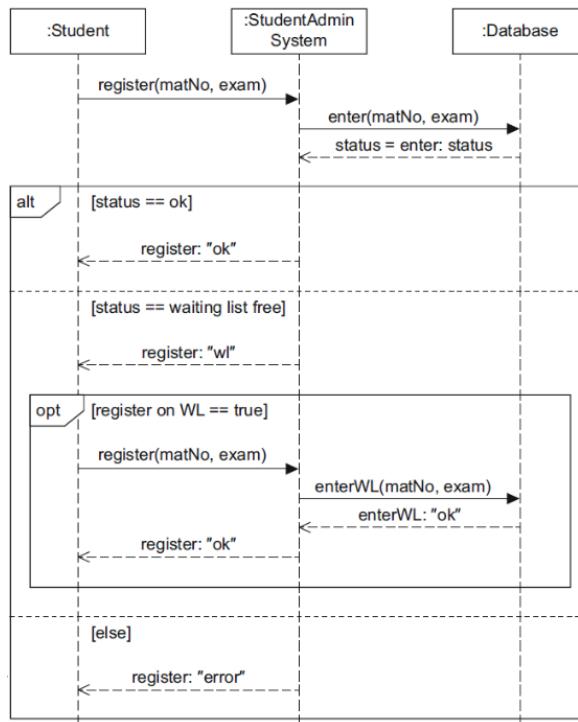
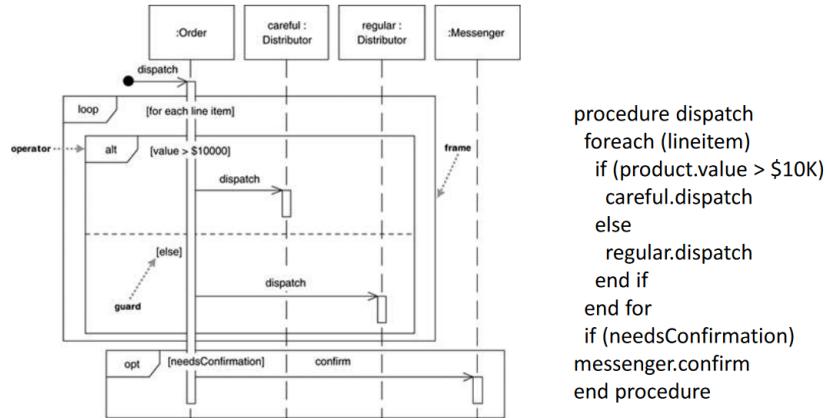


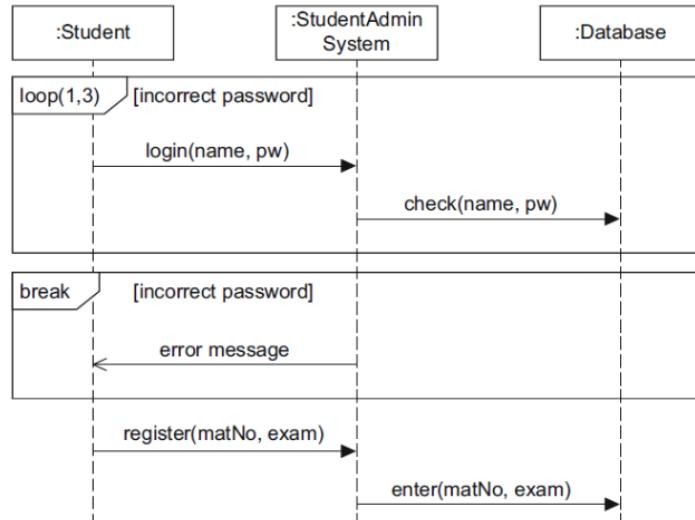
Esistono 12 tipi diversi di operatori, suddivisi in tre gruppi:

	Operator	Purpose
Branches and loops	alt	Alternative interaction
	opt	Optional interaction
	loop	Iterative interaction
	break	Exception interaction
Concurrency and order	seq	Weak order
	strict	Strict order
	par	Concurrent interaction
	critical	Atomic interaction
Filters and assertions	ignore	Irrelevant interaction parts
	consider	Relevant interaction parts
	assert	Asserted interaction
	neg	Invalid interaction

È possibile utilizzare un frammento alt con almeno due operandi per rappresentare sequenze alternative. Infatti alt fragment rappresenta un percorso alternativo nell'esecuzione, che corrisponde approssimativamente ai più casi nei linguaggi di programmazione, ad esempio l'istruzione switch in Java. Ogni

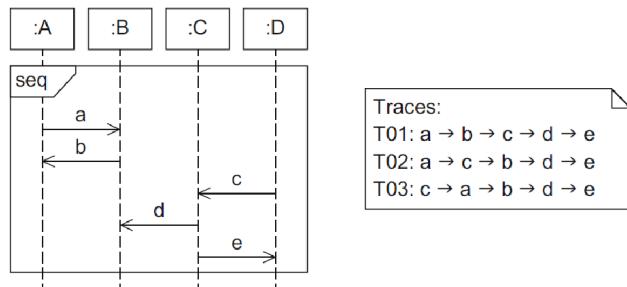
operando ha una guardia, cioè un'espressione booleana racchiusa tra parentesi quadre. Una guardia speciale [else] viene valutata come vera se non è soddisfatta nessun'altra condizione.



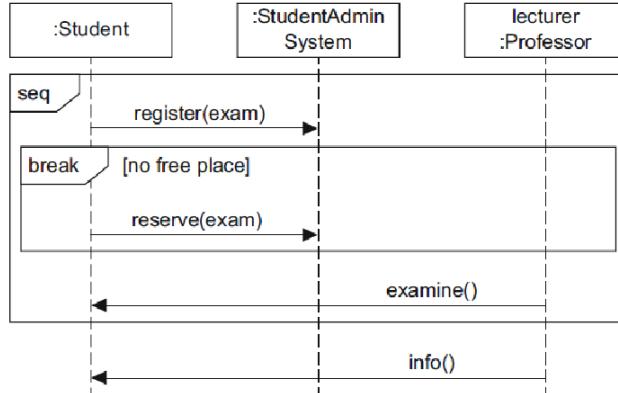


La disposizione degli eventi sull'asse verticale rappresenta l'ordine cronologico di questi eventi, a condizione che ci sia uno scambio di messaggi tra i partner dell'interazione coinvolti. I frammenti combinati descritti qui di seguito consentono di controllare esplicitamente l'ordine degli eventi.

- Il frammento seq rappresenta l'ordine predefinito. Ha almeno un operando ed esprime una sequenzialità debole, specificata dallo standard UML:
 1. L'ordine degli eventi all'interno di ciascuno degli operandi viene mantenuto nel risultato.
 2. Gli eventi su linee di vita diverse provenienti da operandi diversi possono arrivare in qualsiasi ordine.
 3. Gli eventi di operandi diversi sulla stessa linea di vita sono ordinati in modo tale che un evento del primo operando venga prima di quello del secondo operando

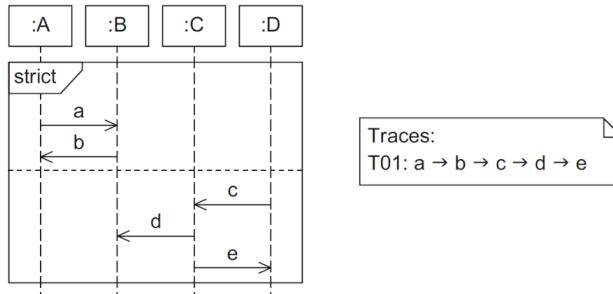


Possiamo usare il frammento seq per raggruppare i messaggi con un frammento break

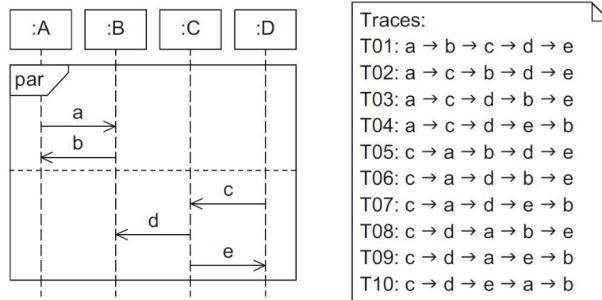


Se non c'è posto, lo studente prenota per il giorno successivo. Tuttavia se c'è posto l'istruzione reserve viene saltata.

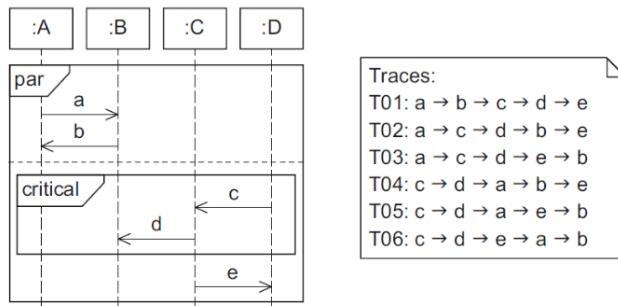
- Il frammento strict descrive un'interazione sequenziale con un ordine rigoroso. Anche se non c'è scambio di messaggi tra i partner dell'interazione, i messaggi di un operando che si trova più in alto sull'asse verticale vengono sempre scambiati prima dei messaggi di un operando che si trova più in basso sull'asse verticale.



- Il frammento par consente di ignorare l'ordine cronologico tra i messaggi dei diversi operandi. Tuttavia, questo costrutto non induce un vero parallelismo, cioè non richiede l'elaborazione simultanea degli operandi. L'operatore par esprime in realtà la concomitanza, cioè l'ordine degli eventi che si trovano nei diversi operandi è irrilevante.

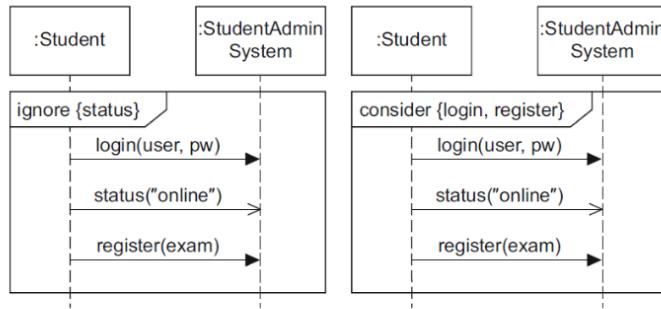


- Per assicurarsi che alcune parti di un'interazione non vengano interrotte da eventi da eventi inattesi, si può usare il frammento critical, il quale segna un'area atomica nell'interazione.

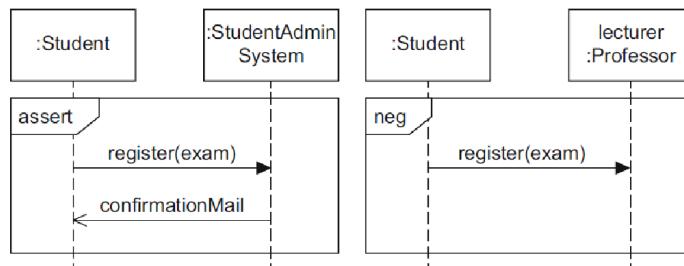


Il gruppo **"filtr e asserzioni"** definisce quali messaggi possono essere presenti ma non sono rilevanti per la descrizione del sistema, quali messaggi devono verificarsi e quali messaggi non devono essere presenti.

- I messaggi non rilevanti sono indicati dal frammento ignore, che esprime che questi messaggi possono essere verificarsi in fase di esecuzione, ma non hanno alcun significato per le funzioni rappresentate nel modello.
- Il frammento consider specifica i messaggi che sono considerati di particolare importanza per l'interazione in esame.

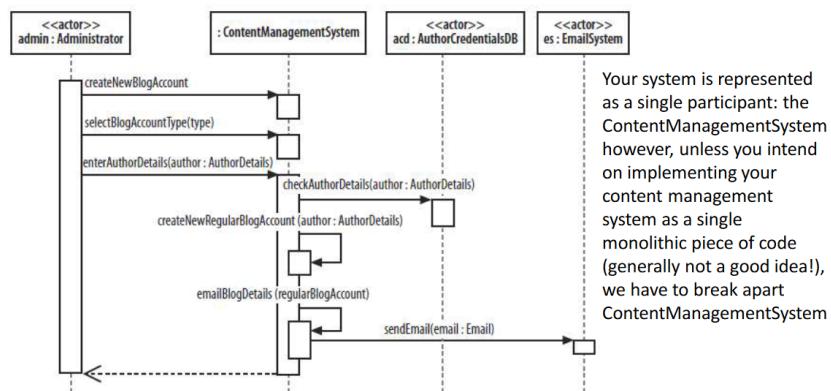
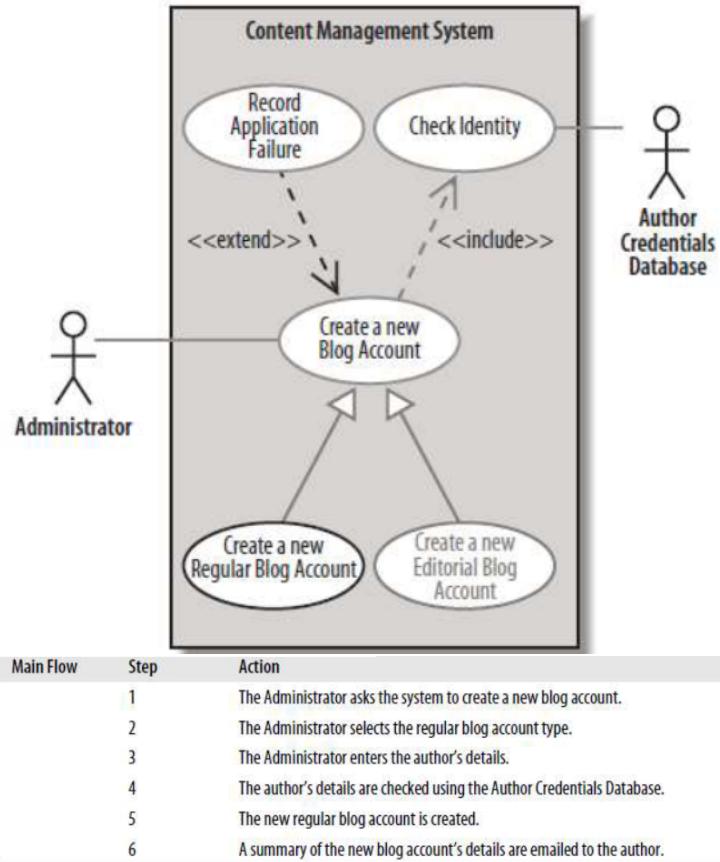


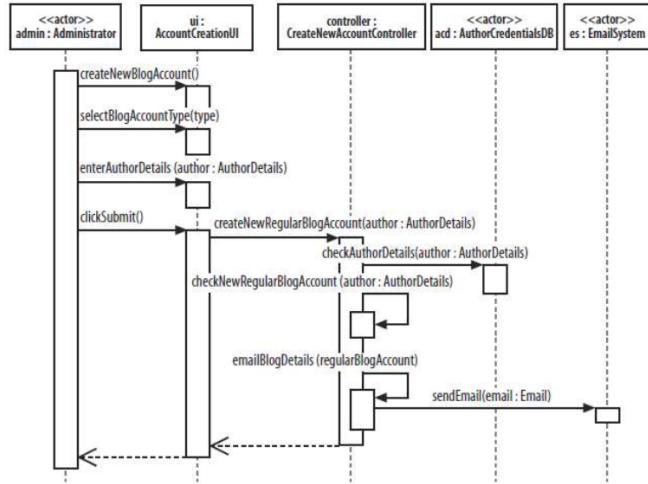
- Il frammento assert identifica alcune tracce modellate come obbligatorie.
- Con il frammento neg si modella un'interazione non valida ,cioé si descrivono situazioni che non devono verificarsi.



Infine le punte di freccia piene indicano un messaggio sincrono(se un chiamante invia un messaggio sincrono, deve attendere che il messaggio sia terminato) mentre quelle a bastoncino indicano un messaggio asincrono (se un chiamante invia un messaggio asincrono, può continuare a elaborare e non deve attendere la risposta).

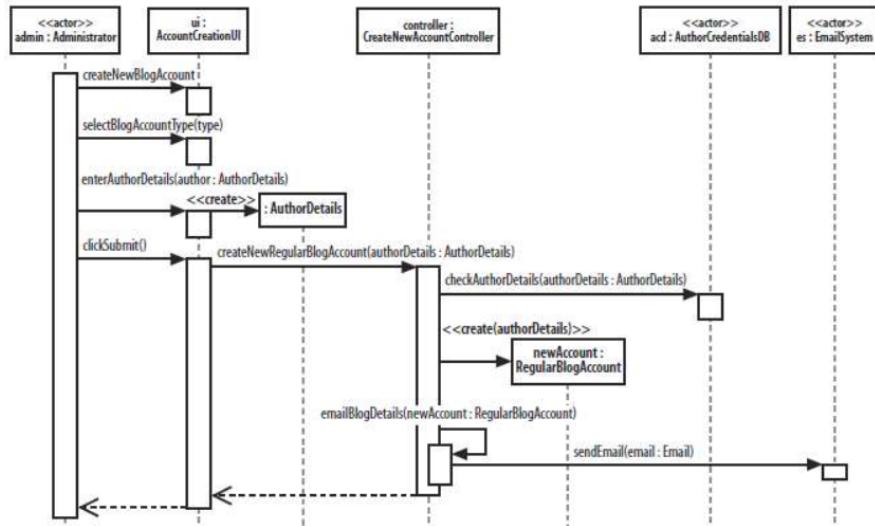
Quando usare il sequence diagram? È consigliabile utilizzare i diagrammi di sequenza quando si vuole esaminare il comportamento di diversi oggetti all'interno di un singolo caso d'uso; in effetti sono utili per mostrare le collaborazioni tra gli oggetti ma non sono così bravi a definire con precisione il comportamento (in questo caso usare use case o state machine diagram). Vediamo un esempio completo partendo da uno use case.

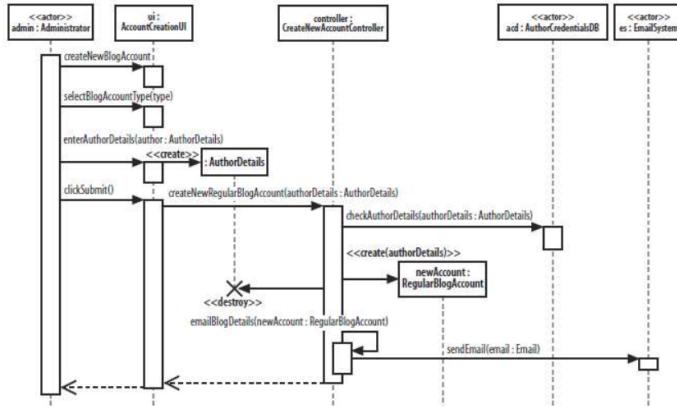




Something critical is missing from the sequence diagram.

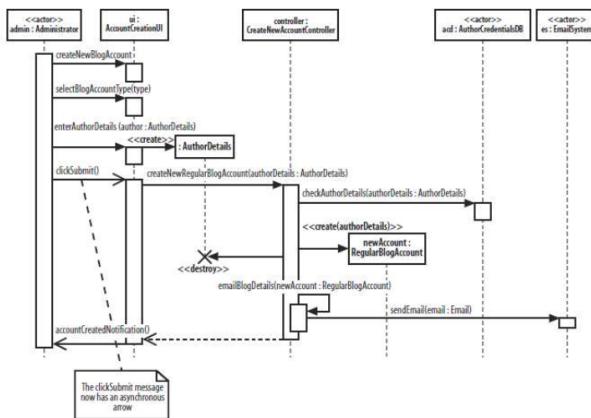
The title of the use case in which the sequence diagram is operating is Create a new Regular Blog Account, but where is the actual *creation* of the blog account?





When the Administrator clicks on the submit button the system freezes, until the new blog account has been created

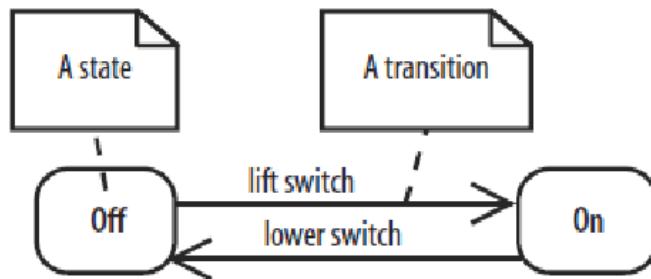
the `authorDetails:AuthorDetails` participant is no longer required once the `newAccount:RegularBlogAccount` has been created.



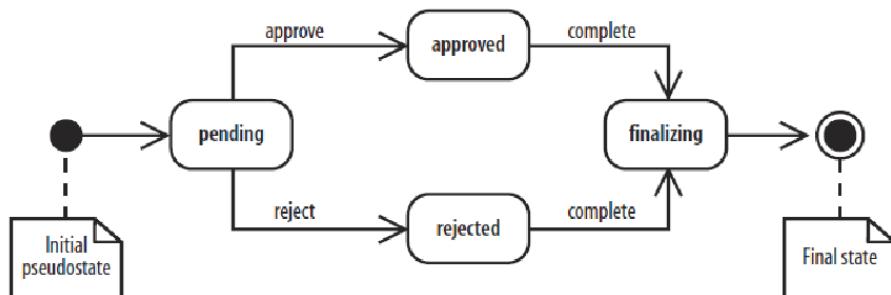
Converting the `clickSubmit()` from a synchronous to an asynchronous message means that the sequence diagram now shows that when the new regular blog account information is submitted, the user interface will not lock and wait for the new account to be created. Instead, the user interface allows the Administrator actor to continue working with the system.

3.7 UML - State Machine Diagram

I diagrammi di attività sono utili per descrivere il comportamento, ma c'è un pezzo mancante. A volte lo stato di un oggetto o di un sistema è un fattore importante per il suo comportamento; in queste situazioni, è utile modellare gli stati di un oggetto e gli eventi che causano i cambiamenti di stato: questo è ciò che i diagrammi delle macchine a stati fanno. Gli elementi fondamentali di un diagramma di stato sono gli stati e le transizioni tra gli stati. Uno stato è attivo quando vi si accede attraverso una transizione, mentre diventa inattivo quando esce attraverso una transizione. L'evento che provoca il cambiamento di stato, o trigger, viene scritto lungo la freccia di transizione.



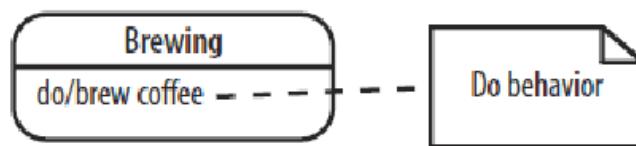
I diagrammi di stato hanno di solito uno pseudostato iniziale e uno stato finale, che segnano i punti di inizio e fine della macchina a stati.



Uno **stato** è una condizione dell'essere in un certo momento e viene rappresentato come un rettangolo arrotondato con il nome dello stato al centro. Uno stato può essere di 2 tipi:

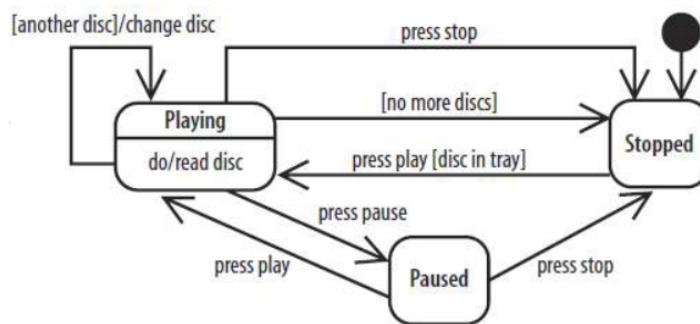
1. Può essere passivo

- Può essere attivo. In questi stati si può scrivere il comportamento all'interno dello stato con la sintassi do/behavior (comportamento che avviene finché lo stato è attivo).



Gli Stati possono reagire agli eventi senza transizione, utilizzando attività interne (si mette l'evento, la guardia e l'attività all'interno della scatola dello stato stesso). Esistono due attività interne speciali: l'attività di ingresso, che viene eseguita ogni volta che si entra in uno stato e l'attività di uscita, che viene eseguita ogni volta che si lascia lo stato. Si possono avere anche stati in cui l'oggetto sta svolgendo un lavoro (do-activities). La differenza tra i due è che le attività regolari avvengono "istantaneamente" e non possono essere interrotte da eventi regolari, mentre le attività di lavoro possono richiedere un tempo finito e possono essere interrotte.

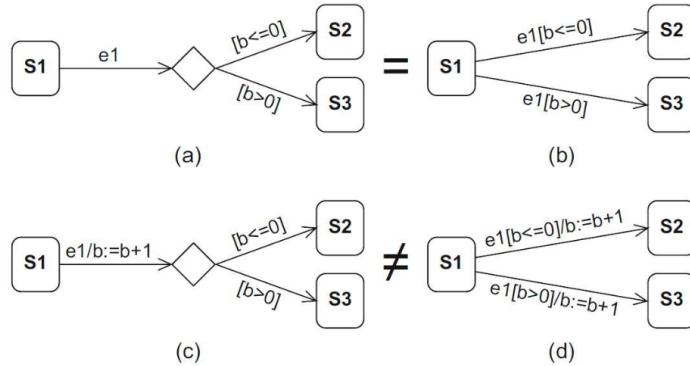
La **transizione** indica il passaggio da uno stato all'altro. Ogni transizione ha un'etichetta composta da tre parti: trigger-signature [guard] /activity (tutte e tre opzionali): La firma di attivazione è solitamente un singolo evento che innesca un potenziale cambiamento di stato, la guardia, se presente, è una condizione booleana che deve essere vera perché la transizione venga eseguita e l'activity è un'attività non interrompibile che viene eseguita durante la transizione.



Lo stato finale indica che la macchina a stati è completata e implica la cancellazione dell'oggetto controllore. Con oggetto indichiamo lo stato dell'oggetto per indicare la combinazione di tutti i dati nei campi dell'oggetto.

Il **nodo decisionale** è rappresentato nel diagramma con un diamante. Può essere utilizzato per modellare transizioni alternative, valutando le protezioni e

selezionando così il bordo in uscita da utilizzare. Per evitare che il sistema rimanga "bloccato" nel nodo decisionale, è necessario assicurarsi che le protezioni coprano tutte le situazioni possibili.

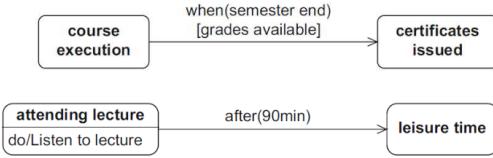


Nello stato machine diagram è possibile introdurre il concetto di parallelizzazione con il nodo di parallelizzazione ed il nodo di sincronizzazione entrambi rappresentati da una barra nera.

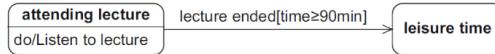
Vediamo ora una panoramica sui vari tipi di eventi; i tipi più importanti di eventi definiti in UML sono il signal event, call event, time event, change event, any receive event e completion event.

- Il signal event è utilizzato per la comunicazione asincrona quindi il mittente invia un segnale al destinatario senza attendere la risposta.
- Call events sono chiamate di operazioni. Il nome dell'evento corrisponde al nome di un'operazione comprensiva di parametri (register(exam)).
- Time events consentono transizioni di stato basate sul tempo (after(5 seconds)).
- È possibile utilizzare un evento di modifica (change event) per monitorare in modo permanente se una condizione diventa vera. Un evento di modifica è costituito da un'espressione booleana e dalla parola chiave preceduta dal when (when(x > y)).
- È possibile utilizzare un evento any receive per specificare un tipo di transizione "else"
- Un evento di completamento (completion event) ha luogo quando tutto ciò che deve essere fatto nello stato corrente è stato completato.

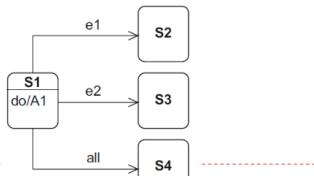
Change events



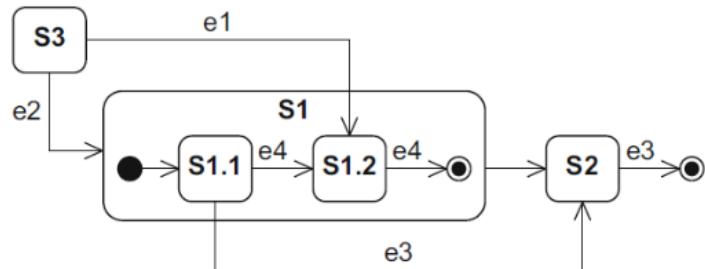
- ▶ Change event modeled with guard



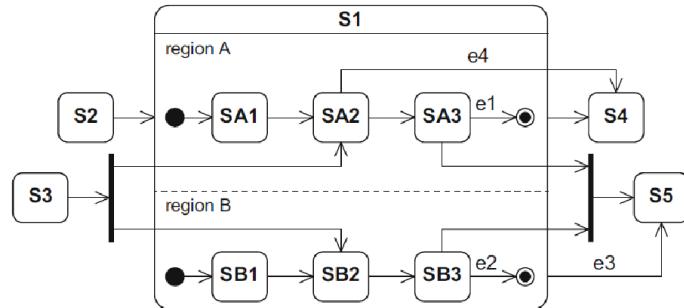
- ▶ Any receive event



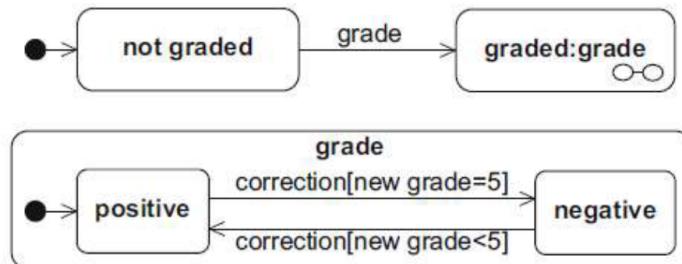
Un **composite state**, definito anche stato complesso o stato annidato, è uno stato che contiene più stati e pseudostati (gli stati contenuti all'interno di uno stato composito sono chiamati substati).



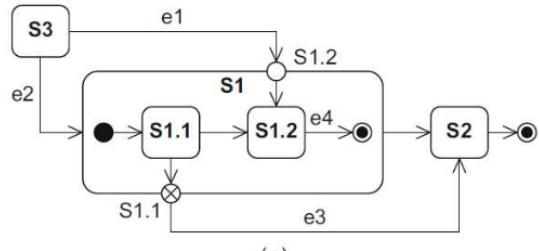
Se uno stato composito è attivo, solo uno dei suoi substati è attivo in qualsiasi momento. Se si desidera ottenere stati concomitanti, uno stato composito può essere suddiviso in due o più regioni, in cui uno stato di ciascuna regione è sempre attivo in qualsiasi momento. Questo tipo di stato composito è chiamato stato ortogonale.



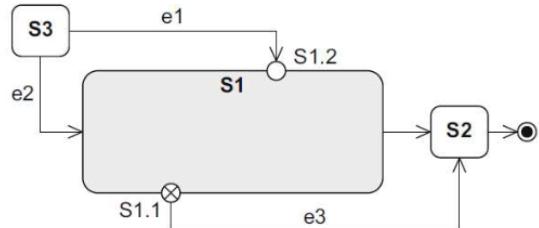
Se più diagrammi di macchine a stati condividono parti con lo stesso comportamento, non è pratico modellare lo stesso comportamento più volte. È possibile riutilizzare parti di diagrammi di macchine a stati in altri diagrammi di macchine a stati.



Se si entra o si esce da uno stato composito attraverso uno stato diverso da quello iniziale e finale, è possibile modellarlo utilizzando punti di ingresso e di uscita. Un punto di ingresso è modellato da un piccolo cerchio sul confine dello stato composito e ha un nome che descrive il punto di ingresso (il punto di ingresso ha una transizione verso lo stato in cui deve iniziare l'esecuzione). Invece un punto di uscita è indicato al confine dello stato composito da un piccolo cerchio contenente una X e ha un nome che descrive il punto di uscita. I punti di uscita ed entrate effettuano una sorta di "incapsulamento".

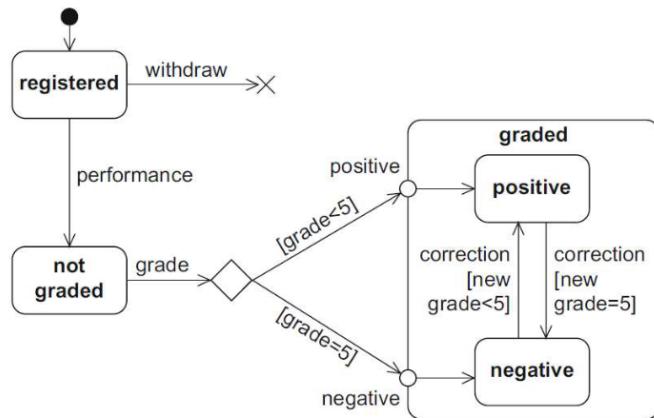


(a)



(b)

Infine vediamo un esempio completo.

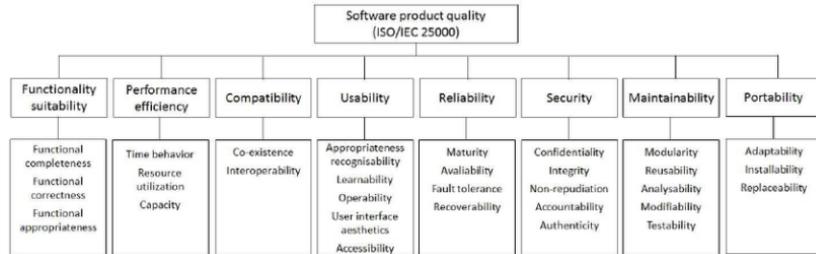


I diagrammi di stato sono utili per descrivere il comportamento di un oggetto nei diversi casi d'uso ma bisogna cercare uno state diagramma per ogni classe del sistema (spreco di energie). Inoltre come si può notare i diagrammi di stato non sono molto adatti a descrivere un comportamento che coinvolge oggetti che collaborano tra loro.

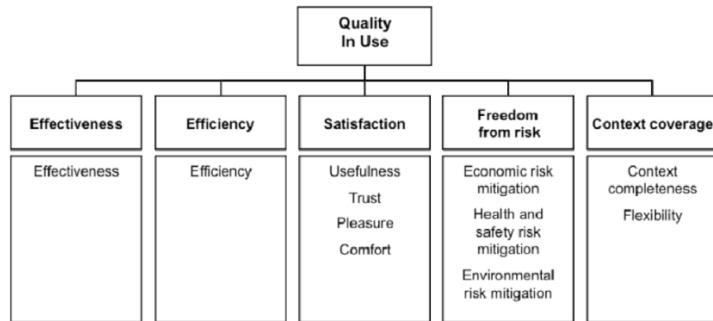
4 Implementazione

L'obiettivo dei progetti di ingegneria del software è produrre un programma funzionante. L'implementazione è l'atto di trasformare il progetto dettagliato di un programma in vero proprio codice di un determinato linguaggio di programmazione. Solitamente nei progetti di piccole dimensioni, la progettazione dettagliata viene di solito lasciata come parte dell'implementazione ma nei progetti più grandi, o quando i programmatore sono inesperti, la progettazione dettagliata è affidata ad un'altra persona. Lo standard ISO/IEC 25010:2011 definisce due modelli:

- Modello di qualità del prodotto software: Questo modello è composto da otto caratteristiche suddivise in sottocaratteristiche. Gli attributi di qualità derivati dal modello possono essere misurati internamente o esternamente.



- Modello di qualità in uso: Questo modello è composto da cinque caratteristiche divise in sottocaratteristiche. Gli attributi vengono misurati quando il software viene testato in un contesto d'uso realistico. Introducendo la nozione di contesto d'uso, gli attributi dipendono dal software ma anche dall'utente, dalla piattaforma e dall'ambiente.



Le caratteristiche principali di una buona implementazione sono le seguenti:

- Leggibilità: Il codice può essere facilmente letto e compreso da altri programmati.
- Manutenibilità: Il codice può essere facilmente modificato e mantenuto.
- Prestazioni: A parità di altre condizioni, l'implementazione deve produrre codice che funzioni il più velocemente possibile.
- Tracciabilità: Tutti gli elementi del codice devono corrispondere a un elemento del progetto. Il codice può essere ricondotto al progetto (e il progetto ai requisiti).
- Correttezza: L'implementazione deve fare quello per cui è stata pensata (come definito nei requisiti e nel progetto dettagliato).
- Completezza: Tutti i requisiti del sistema sono soddisfatti.

Di solito il primo istinto dei programmati è quello di concentrarsi sulla correttezza, le prestazioni e non dare importanza alla leggibilità e manutenibilità ma la manutenibilità è importante quanto la correttezza, a leggibilità aiuta la manutenibilità, la leggibilità e la manutenibilità aiutano a raggiungere la correttezza e le ottimizzazioni eccessive delle prestazioni possono ridurre la leggibilità e la manutenibilità. Le cattive prestazioni possono essere il sintomo di un cattivo design o di una cattiva codifica, con un alto grado di leggibilità e manutenibilità si avrà anche un miglioramento delle prestazioni. Per ottimizzare le prestazioni senza ridurre leggibilità e manutenibilità si può ricorrere ad un profiler, uno strumento che esegue un programma e calcola il tempo speso in ogni parte; questo vi aiuterà a trovare i colli di bottiglia delle prestazioni e i moduli che devono essere ottimizzati.

Quasi tutte le organizzazioni che sviluppano software hanno delle linee guida che specificano gli stili di denominazione, l'indentazione e l'impostazione dei commenti. Nei progetti di software (di grandi dimensioni), di solito esistono delle convenzioni di programmazione. Vediamo ora le principali concezioni.

Scegliere i **nomi** per classi, metodi, variabili e altre entità è principalmente una questione semantica che riguarda la scelta di nomi validi secondo delle regole comuni (coerenza). Favorire metodi di piccole dimensioni perché le funzioni o i metodi di grandi dimensioni sono statisticamente più inclini all'errore di quelli più piccoli. Standard per specificare il nome dei file, quali file generare per ogni modulo e come individuare un determinato file da un modulo. da un modulo è molto vantaggioso. Raccomandazione di vietare alcune caratteristiche del linguaggio di programmazione che si sono rivelate, per l'organizzazione, fonte di errori (ereditarietà multipla). I **commenti** possono aiutare o danneggiare in modo significativo la leggibilità e la manutenibilità, infatti possono distrarre dal codice vero e proprio e rendere il programma più difficile da leggere. Inoltre i commenti possono diventare obsoleti con la modifica del codice (cambia il codice ma non cambia il commento). Esistono 6 differenti tipi di commenti:

1. Repeat of the code: These kinds of comments should be avoided

```
// increment i by one  
++i;
```

2. Spiegazione del codice: i programmatore spiegano il codice in linguaggio umano. Se il codice è così complesso, in genere deve essere riscritto.
3. Marcatori nel codice per indicare elementi incompleti, miglioramenti, informazioni simili e pertenere traccia delle modifiche (meglio utilizzare un software di gestione delle versioni).
4. Sintesi del codice: commenti che riassumono ciò che fa il codice, piuttosto che ripeterlo semplicemente. Sono utili per comprendere il codice, ma devono essere aggiornati.
5. Descrizione dell'intento del codice: descrivono ciò che il codice dovrebbe fare piuttosto che quello che fa.
6. Riferimenti esterni: Sono commenti che collegano il codice ad entità esterne, di solito libri o altri programmi.

È necessario riconoscere il compromesso che i commenti comportano infatti i commenti possono aiutare a chiarire il codice e a metterlo in relazione con altre fonti ma la loro manutenzione richiedono un certo impegno.

Un'altra pratica molto utilizzata è quella di fare uso del **debugging**, cioè l'atto di individuare e correggere gli errori nel codice. Il debug è un processo altamente iterativo: si crea un'ipotesi sulla causa degli errori, si scrivono i casi di test per dimostrare o confutare l'ipotesi e in caso di errore si modifica il codice per cercare di risolvere il problema. Possiamo identificare quattro fasi nel processo di debug:

- Stabilizzazione: riproduzione dell'errore su una configurazione, per individuare le condizioni che hanno portato all'errore costruendo un caso di test minimo
- Localizzazione: individua le sezioni del codice che hanno portato all'errore
- Correzione
- Verifica: assicura la correzione dell'errore.

Gli strumenti possono aiutare nel processo di debug sono i comparatori di codice sorgente, i controller estesi, i debugger interattivi le librerie appositamente costruite e i profilatori. Una tecnica molto utile è l'uso delle asserzioni, legata ai concetti di precondizioni e postcondizioni, le quali possono dimostrare che un codice viene eseguito correttamente. È buona norma rendere esplicite le precondizioni mediante l'uso di asserzioni (la maggior parte dei linguaggi di programmazione moderni dispone di strutture specifiche per le asserzioni). L'asserzione ha due forme:

1. assert Espressione1, nella quale Espressione1 è un'espressione booleana. Quando il sistema esegue l'asserzione, valuta l'espressione1 e se questa è falso lancia un AssertionError senza alcun messaggio di dettaglio.
2. assert Espressione1 : Espressione2, dove Espressione1 è un'espressione booleana ed Espressione2 è un'espressione che ha un valore.

Ci sono molte situazioni in cui è bene usare le asserzioni, tra cui:

- Invarianti interni

```
if (i % 3 == 0){
    ...
} else if(i % 3 == 1){
    ...
} else{
    assert i % 3 == 2 : 1;
    ...
}
```

L'asserzione può fallire se i è negativo, poiché l'operatore `%` non è un vero e proprio operatore di modulo, ma calcola il resto che può essere negativo.

- Invarianti del flusso di controllo

```
void foo(){
    for(...){
        if(...){
            return;
        }
        assert false; // Execution should never reach
                      this point
    }
}
```

Inserire un'asserzione in qualsiasi punto che si presume non possa essere raggiunto

- Precondizioni, postcondizioni e invarianti di classe: asserzioni solitamente utilizzate nei loop per capirne il comportamento.

La programmazione difensiva è una pratica in cui gli sviluppatori prevedono errori nel loro codice, partendo dal presupposto che i bug esistono e quindi dovremmo trovare modi per identificare o contrastare più facilmente questi problemi.

- Mettere il letterale di stringa per primo

```
// Bad
if (variable.equals("literal")) {...}
```

```

// Good
if ("literal".equals(variable)) {...}

// per prevenire NullPointerException

• Non fidatevi di quel -1

//Bad
if (string.indexOf(character) != -1) {...}

//Good
if (string.indexOf(character) >= 0) {...}

• Verifica per null e lunghezza

//Bad
if (array.length > 0)

//Good
if (array != null && array.length > 0) {...}

// Ogni volta che si dispone di un insieme, di un
// array controllare che sia presente e che non sia
// vuoto

```

Tutti i vostri programmi possono essere migliorati. Una tecnica molto utilizzata, introdotta da Martin Fowler nel 1999, è il refactoring, attività che migliora lo stile del codice senza alterarne il comportamento. Secondo Fowler (1999), il refactoring è "un cambiamento apportato alla struttura interna di un software per renderlo più facile da comprendere e più economico da modificare senza cambiare il suo comportamento osservabile". Per far ciò possiamo estrarre dei metodi, introdurre algoritmo di sostituzione, spostare gli algoritmi tra le classi ed estrarre delle classi.

Vediamo ora di descrivere con precisione la manutenibilità e le tecniche che ci permettono di mantenere un certo grado di manutenibilità durante lo sviluppo del progetto. Esistono 4 tipi di manutenzione del software:

1. Manutenzione correttiva: I bug vengono scoperti e devono essere corretti
2. Manutenzione adattativa: Il sistema deve essere adattato ai cambiamenti dell'ambiente in cui opera.
3. Manutenzione perfettiva
4. Manutenzione preventiva: Vengono identificati i modi per aumentare la qualità o evitare che si verifichino bug futuri

I principi che guidano la manutenzione sono i seguenti:

- La manutenibilità trae i maggiori benefici dal rispetto di semplici linee guida.
- La manutenibilità non è un ripensamento, ma deve essere affrontata fin dall'inizio di un progetto di sviluppo.
- Alcune violazioni sono peggiori di altre.

Vediamo ora le linee guida per mantenere un alto grado di manutenibilità.

- **Write short units of code**

Limitare la lunghezza delle unità di codice a 15 righe di codice. Questo migliora la manutenibilità perché le unità piccole sono facili da analizzare/-capire, da testare e da riutilizzare. Per far ciò possiamo utilizzare la tecnica di estrazione dei metodi. Un'obiezione a questa pratica potrebbe essere la seguente "Scrivere unità brevi significa avere più unità, e quindi più chiamate di metodo. Questo non sarà mai performante"; dobbiamo ricordarci però di non sacrificare la manutenibilità per ottimizzare le prestazioni, a meno che solidi test sulle prestazioni non abbiano dimostrato che si ha effettivamente un problema di prestazioni.

```

public Board createBoard(Square[][] grid) {
    assert grid != null;
    Board board = new Board(grid);

    int width = board.getWidth();
    int height = board.getHeight();
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            Square square = grid[x][y];
            for (Direction dir : Direction.values()) {
                int dirX = (width + x + dir.getDeltaX()) % width;
                int dirY = (height + y + dir.getDeltaY()) % height;
                Square neighbour = grid[dirX][dirY];
                square.link(neighbour, dir);
            }
        }
    }
    return board;
}

private void setLink(Square square, Direction dir, int x, int y, int width,
                     int height, Square[][] grid) {
    int dirX = (width + x + dir.getDeltaX()) % width;
    int dirY = (height + y + dir.getDeltaY()) % height;
    Square neighbour = grid[dirX][dirY];
    square.link(neighbour, dir);
}

```

18 lines of code

Method with 7 parameters

Extract Method

Lines to extract and refactor

- **Write simple units of code**

Limitare il numero di punti di diramazione per unità a 4. Questo migliora la manutenibilità perché mantenere il numero di punti di diramazione basso rende le unità più facili da modificare e da testare. Dobbiamo limitare il numero di punti di diramazione (in Java le seguenti istruzioni e operatori che contano come punti di diramazione sono if, case, ?, and, or,

While, For, catch) e per far ciò possiamo rimpiazzare le condizioni con il pattern Polymorphism.

```
public List<Color> getFlagColors(Nationality nationality) {
    List<Color> result;
    switch (nationality) {
        case DUTCH:
            result = Arrays.asList(Color.RED, Color.WHITE, Color.BLUE);
            break;
        case GERMAN:
            result = Arrays.asList(Color.BLACK, Color.RED, Color.YELLOW);
            break;
        case BELGIAN:
            result = Arrays.asList(Color.BLACK, Color.YELLOW, Color.RED);
            break;
        case FRENCH:
            result = Arrays.asList(Color.BLUE, Color.WHITE, Color.RED);
            break;
        case ITALIAN:
            result = Arrays.asList(Color.GREEN, Color.WHITE, Color.RED);
            break;
        case UNCLASSIFIED:
        default:
            result = Arrays.asList(Color.GRAY);
            break;
    }
    return result;
}

public interface Flag{
    List<Color> getColors();
}

public class DutchFlag implements Flag{
    public List<Color> getColors(){
        return Arrays.asList(Color.RED, Color.WHITE,
                            Color.BLUE);
    }
}

public class ItalianFlag implements Flag{
    public List<Color> getColors(){
        return Arrays.asList(Color.GREEN,
                            Color.WHITE, Color.RED);
    }
}
```

Inoltre possiamo rimpiazzare le condizioni innestate con delle clausole di guardia.

```
double getPayAmount() {  
    double result;  
    if (_isDead) result = deadAmount();  
    else {  
        if (_isSeparated) result = separatedAmount();  
        else {  
            if (_isRetired) result = retiredAmount();  
            else result = normalPayAmount();  
        };  
    }  
    return result;  
};
```

Isolate all special checks and edge cases into separate clauses and place them before the main checks. Ideally, you should have a "flat" list of conditionals, one after the other.

```
double getPayAmount() {  
    if (_isDead) return deadAmount();  
    if (_isSeparated) return separatedAmount();  
    if (_isRetired) return retiredAmount();  
    return normalPayAmount();  
};
```

- **Write code once**

Non copiare il codice: questo migliora la manutenibilità perché quando il codice viene copiato, i bug devono essere corretti in più punti, il che è inefficiente e soggetto a errori. Il problema fondamentale della duplicazione è che non si sa se esiste un'altra copia del codice che si sta analizzando, quante copie esistono e dove si trovano; questo ne rende difficile anche la modifica. La tecnica di rifattorizzazione Extract Method è il cavallo di battaglia che risolve molti problemi di duplicazione. Inoltre possiamo anche utilizzare la tecnica di rifattorizzazione Extract Superclass che estrae un frammento di linee di codice non solo per un metodo, ma anche per una nuova classe che è la superclasse della classe originale.

- **Keep unit interface small**

Limitare il numero di parametri per unità a un massimo di 4. Questo migliora la manutenibilità perché mantenere il numero di parametri basso rende le unità più facili da capire e da riutilizzare. In caso di molti parametri possiamo passare un oggetto che accoppiava caratteristiche e parametri comuni.

- **Separate concerns in modules**

Evitare moduli di grandi dimensioni per ottenere un accoppiamento lasco tra di essi. Questo si ottiene assegnando le responsabilità a moduli separati e nascondendo i dettagli dell'implementazione dietro le interfacce.

- **Couple architecture components loosely**

Ottenere un accoppiamento lasco tra i componenti di top level. Questo migliora la manutenibilità, perché i componenti indipendenti facilitano la manutenzione isolata.

- **Keep architecture components balanced**

Bilanciare il numero e la dimensione relativa dei componenti di primo livello nel codice. A tal fine, organizzare il codice sorgente in modo che il numero di componenti sia vicino a 9 (cioè tra 6 e 12) e che i componenti siano di dimensioni approssimativamente uguali. Questo migliora la manutenibilità perché i componenti bilanciati facilitano l'individuazione del codice e consentono una manutenzione isolata.

- **Keep your codebase small**

Mantenete la vostra base di codice il più piccola possibile (una base di codice è una raccolta di codice sorgente che viene memorizzata in un repository, la quale può essere compilata e distribuita in modo indipendente e che viene mantenuta da un solo team). A questo scopo, evitate la crescita della base di codice e riducete attivamente le dimensioni del sistema. Questo migliora la manutenibilità, perché avere un prodotto, un progetto e un team di dimensioni ridotte è un fattore di successo.

- **Automate development pipeline and tests**

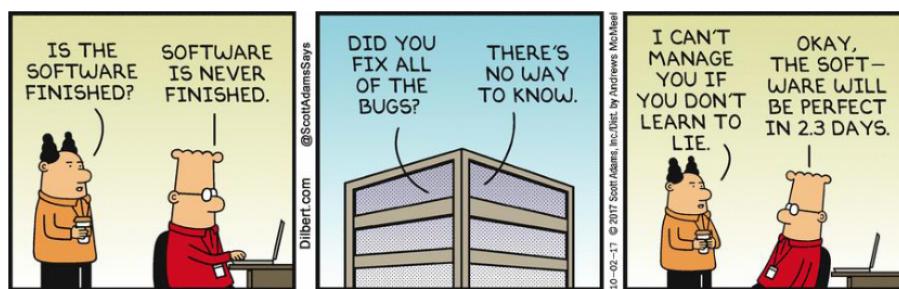
Automatizzare i test per la propria base di codice. A tal fine, è necessario scrivere test automatizzati utilizzando un framework di test. Questo migliora la manutenibilità, perché i test automatizzati rendono lo sviluppo prevedibile e meno rischioso.

- **Write clean code**

Scrivi codice pulito. Per farlo, non lasciate tracce di codice dopo il lavoro di sviluppo (non lasciare commenti errati e non lasciare codice nei commenti.). Questo migliora la manutenibilità, perché il codice pulito è manutenibile.

5 Testing

Uno degli obiettivi principali dello sviluppo del software è produrre software di alta qualità e per raggiungere questo obiettivo, è necessario eseguire dei test. La necessità di test è il motivo principale per il quale il software "nuovo" è di minor qualità rispetto al software "vecchio" che sarà stato testato innumerevoli volte durante il suo sviluppo e manutenzione (i sistemi critici utilizzano sempre il software vecchio stabile e funzionante).



Il modo migliore per ottenere la qualità in un prodotto è metterla al primo posto. Infatti se si segue un processo ben definito e adeguato all'azienda e al progetto, è probabile che il prodotto finale sarà di alta qualità e quindi il testing degli artefatti, sia lungo il percorso che alla fine, diventa un processo continuo e centrale nello sviluppo. Per garantire un'alta qualità possiamo utilizzare 2 diverse tecniche:

- Assicurazione della qualità

Attività volte a misurare e migliorare la qualità di un prodotto, compreso l'intero processo, la formazione e la preparazione del team.

- Controllo qualità

Attività volte a verificare la qualità del prodotto, a rilevare difetti e a garantire che i difetti vengano risolti prima del rilascio.

Le principali tecniche per il rilevamento degli errori nei programmi e nei documenti intermedi sono le seguenti:

- Test, il processo di progettazione dei casi di test, l'esecuzione del programma in un ambiente controllato e la verifica della correttezza dell'output.
- Ispezioni e revisioni, applicate a un programma o ad artefatti software intermedi.
- Metodi formali, tecniche matematiche utilizzate per "dimostrare" la correttezza di un programma.

- Analisi statica, il processo di analisi della struttura statica di un programma o di un prodotto software intermedio.

Ma come possiamo valutare la qualità di progetto software? La qualità di un prodotto può essere definita in due modi: è conforme alle specifiche o se rispetta l'obiettivo prefissato. Ma può succedere che un prodotto può essere perfettamente conforme alle sue specifiche, ma non serve a nulla; per questo dividiamo la valutazione della qualità in 2 atti differenti:

- Verifica: l'atto di controllare che un prodotto software sia conforme ai requisiti e alle specifiche.
- Convalida: l'atto di controllare che un prodotto software finito soddisfi i requisiti e le specifiche degli utenti (fatto alla fine del progetto).

Verification	Validation
Verification addresses the concern: "Are you building it right?"	Validation addresses the concern: "Are you building the right thing?"
Ensures that the software system meets all the functionality.	Ensures that the functionalities meet the intended behavior.
Verification takes place first and includes the checking for documentation, code, etc.	Validation occurs after verification and mainly involves the checking of the overall product.
Done by developers.	Done by testers.
It has static activities, as it includes collecting reviews, walkthroughs, and inspections to verify a software.	It has dynamic activities, as it includes executing the software against the requirements.
It is an objective process and no subjective decision should be needed to verify a software.	It is a subjective process and involves subjective decisions on how well a software works.

Per categorizzare i vari problemi del software, che possono emergere in fase di testing, usiamo i seguenti vocaboli:

- Fault/Defect: Condizione che può causare un guasto al sistema. È causato da un errore commesso da un ingegnere del software. Un guasto è anche chiamato "bug" e si manifesta solo in alcune situazioni operative.
- Failure/Problem: l'incapacità di un sistema di eseguire una funzione secondo alle sue specifiche. È il risultato di un difetto del sistema.
- Error: Un errore commesso da un ingegnere del software o da un programmatore.

Solo in alcuni casi un difetto genera un failure. Ad esempio, un requisito difficile da capire è un fault e diventa failure solo se porta al fraintendimento di un requisito, che a sua volta provoca un errore nella progettazione e nel codice che si manifesta sotto forma di software failure. Nella valutazione degli errori dobbiamo distinguere tra la gravità di un errore, che misura l'impatto o le conseguenze che può avere e la priorità, cioè la misura della sua importanza.

Ora vediamo di definire il **Testing**. Secondo lo standard ANSI/IEEE 1059, il test può essere definito come un processo di analisi di un elemento del software per individuare le differenze tra le condizioni esistenti e quelle richieste (cioè difetti/errori/bug) e per valutare le caratteristiche del software. In particolar modo il test del software consiste nella verifica dinamica del comportamento di un programma basata su una serie finita di casi di test opportunamente selezionati da un dominio di esecuzioni solitamente infinito. Il testing è un'attività complessa che coinvolge molte attività tra le quali: determinazione obiettivo del test o obiettivo della qualità, metodologia e tecniche di test da utilizzare, risorse da assegnare, strumenti da utilizzare, tabella di marcia. I test hanno solitamente due scopi principali:

- Individuare i difetti del software in modo da poterli correggere o attenuare.
- Fornire una valutazione generale della qualità

Ma i test non possono dimostrare che un prodotto funziona al 100%. Infatti puoi trovare difetti e dimostrare che il prodotto funziona per i casi che sono stati testati, senza garantire nulla per gli altri casi che non sono stati testati.

Chi esegue i test? In primo luogo i programmatori, i quali creano dei casi di test e li eseguono per convincersi che il programma funziona (test unitario). In secondo luogo i tester che sono operatori tecnici che devono scrivere casi di test e assicurarne l'esecuzione. Infine ci sono gli utenti, i quali contribuiscono a rilevare i problemi di usabilità ed ad esporre il software ad un'ampia gamma di input in condizioni reali (gli utenti sono i tester di maggiore importanza). Se gli utenti appartengono all'organizzazione che sta sviluppando il progetto, si parla di Alpha test, mentre se gli utenti non appartengono all'organizzazione si parla di Beta test.

Quando iniziare i test? L'avvio tempestivo dei test riduce i costi e i tempi di rielaborazione e di produzione di un software privo di errori da consegnare al cliente. I test possono essere avviati sin dalla fase di raccolta dei requisiti e continuare fino all'implementazione del software.

Cosa viene testato?

- **Test funzionali**

Sulla base delle specifiche del software da testare.

- Unit testing, comporta la verifica delle singole unità di funzionalità, come procedura, metodo o classe. Questo tipo di test viene eseguito dai rispettivi sviluppatori sulle singole unità di codice sorgente assegnate. I principali limiti di questi test sono che non è possibile

individuare ogni singolo bug di un'applicazione e non è possibile valutare ogni percorso di esecuzione in ogni applicazione software.

- Test di funzionalità/integrazione, si tratta di determinare se le unità quando vengono messe insieme, funzionano come un'unità funzionale. Può essere effettuato in due modi: Test di integrazione dal basso verso l'alto, nel quale si parte dal test delle unità, seguito da test di combinazioni di unità di livello progressivamente più alto, chiamate moduli o build e test di integrazione top-down, i moduli di livello più alto vengono testati per primi e progressivamente i moduli di livello inferiore.
- Test del sistema, si tratta di testare la funzionalità integrata nel sistema.
- Regression Testing, viene eseguita per verificare che un bug risolto non abbia compreso un'altra funzionalità o violazione delle regole aziendali. L'intento del test di regressione è quello di garantire che una modifica non provochi un'altra anomalia nell'applicazione.
- Acceptance Testing, valuterà se l'applicazione soddisfa le specifiche previste e se soddisfa i requisiti del cliente. Eseguendo i test di accettazione su un'applicazione, il team di collaudo dedurrà come l'applicazione si comporterà in produzione.
- Alpha Testing, prima fase del testing che verrà eseguito da un team. I test unitari, i test di integrazione e i test di sistema, se combinati insieme, sono conosciuti come alpha testing. Controlla eventuali errori di ortografia e collegamenti interrotti.
- Beta Testing, viene eseguito dopo che il test alfa è stato eseguito con successo. Il test beta è noto anche come test pre-rilascio e le varie versioni beta di prova del software sono idealmente distribuite a un vasto pubblico sul Web, in parte per dare al programma un test "del mondo reale" e in parte per fornire un'anteprima.

• **Test non funzionali**

Il test non funzionale prevede il collaudo di un software a partire da requisiti non funzionali, ma importanti, come le prestazioni, la sicurezza, l'interfaccia utente.

- Test delle prestazioni, viene utilizzato soprattutto per identificare eventuali colli di bottiglia o problemi di prestazioni in un software. È considerato uno dei tipi di test importanti e obbligatori in termini dei seguenti aspetti: velocità, capacità, stabilità e scalabilità.
- Test di carico, si tratta di un processo di verifica del comportamento di un software applicando un carico massimo in termini di accesso e manipolazione da parte del software di grandi dati in ingresso.
- Stress test, comprende la verifica del comportamento di un software in condizioni anomale. Lo scopo dello stress test è quello di testare

il software applicando il carico al sistema e assumendo le risorse utilizzate dal software per identificare il punto di rottura.

- Test di usabilità, vengono utilizzati per identificare eventuali errori e miglioramenti nel software osservando gli utenti durante l'uso e il funzionamento. L'usabilità può essere definita in termini di cinque fattori, vale a dire: efficienza d'uso, capacità di apprendimento, capacità di memorizzazione, errori/sicurezza e soddisfazione.
- Test di sicurezza, prevede la verifica di un software per identificare eventuali difetti e lacune dal punto di vista della sicurezza e delle vulnerabilità (integrità, autenticazione, autorizzazione, dati del software sono sicuri, controllo e convalida degli input).
- Test di portabilità, ha lo scopo di garantire la riutilizzabilità del software e la possibilità di spostarlo da una piattaforma all'altra.
- Test di conformità, si tratta di testare il prodotto software rispetto a un insieme di standard o politiche. I casi di test sono solitamente generati dagli standard e dalle politiche a cui il prodotto deve essere conforme.
- Test di configurazione. Alcuni prodotti software possono consentire diverse configurazioni. Ad esempio, un prodotto software può funzionare con diversi database o reti diverse.
- Test dell'interfaccia utente, test che si concentrano solo sull'interfaccia utente

Come si generano e si scelgono i casi di test?

- Intuizione: non c'è nessuna guida su come generare i casi, ci affidiamo esclusivamente all'intuizione.
- Specifica: test basati esclusivamente sulle specifiche, senza guardare il codice. La tecnica di testare senza conoscere il funzionamento interno dell'applicazione si chiama è chiamata test black-box (o test comportamentali).

Advantages	Disadvantages
Well suited and efficient for large code segments.	Limited coverage, since only a selected number of test scenarios is actually performed.
Code access is not required.	Inefficient testing, due to the fact that the tester only has limited knowledge about an application.
Clearly separates user's perspective from the developer's perspective through visibly defined roles.	Blind coverage, since the tester cannot target specific code segments or error-prone areas.
Large numbers of moderately skilled testers can test the application with no knowledge of implementation, programming language, or operating systems.	The test cases are difficult to design.

- Codice: Le tecniche basate sulla conoscenza del codice reale sono chiamate white-box testing. I test white-box sono l'indagine dettagliata della logica interna e della struttura del codice (chiamati anche glass testing o open-box testing). Per eseguire i test white-box su un'applicazione, il tester deve conoscere il funzionamento interno del codice.

Advantages	Disadvantages
As the tester has knowledge of the source code, it becomes very easy to find out which type of data can help in testing the application effectively.	Due to the fact that a skilled tester is needed to perform white-box testing, the costs are increased.
It helps in optimizing the code.	Sometimes it is impossible to look into every nook and corner to find out hidden errors that may create problems, as many paths will go untested.
Extra lines of code can be removed which can bring in hidden defects.	It is difficult to maintain white-box testing, as it requires specialized tools like code analyzers and debugging tools.
Due to the tester's knowledge about the code, maximum coverage is attained during test scenario writing	

- Caso di test esistente: Si riferisce a una tecnica chiamata test di regressione, che esegue alcuni (o tutti) i casi di test disponibili per una versione precedente del sistema su una nuova versione.
- Guasti: le 2 principali tecniche sono l'indovinello, in cui i casi di test vengono progettati nel tentativo di individuare i guasti più plausibili, di solito basandosi sulla storia dei guasti scoperti in progetti simili e l'analisi degli errori, che identifica attraverso revisioni e ispezioni le aree dei requisiti e della progettazione che sembrano contenere continuamente difetti.

5.1 Black-box Testing

Ricopre particolare importanza il test black box **Equivalence Class Partitioning**, vediamo come funziona. Il partizionamento in classi di equivalenza è un metodo basato sulle specifiche e si basa sulla suddivisione dell'input in diverse classi che sono equivalenti ai fini della ricerca di errori. Le classi di equivalenza sono determinate esaminando la specifica dei requisiti e l'intuizione del tester, senza guardare l'implementazione.

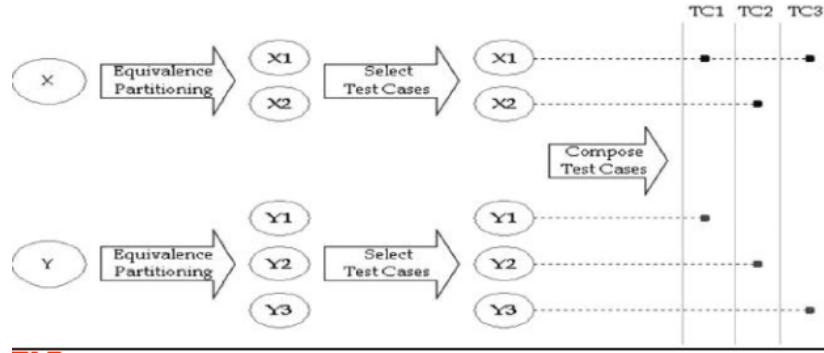
***Example: pick “larger” of
two integers and -----***

Class	Representative
First > Second	10,7
Second > First	8,12
First = second	36, 36

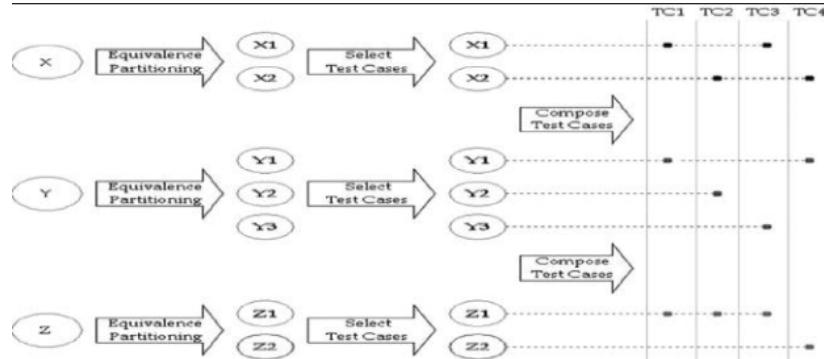
La suddivisione in classi di equivalenza è più utile quando i requisiti sono una serie di condizioni ed invece meno utile quando i requisiti sembrano richiedere dei cicli. Esistono classi di equivalenza valide che descrivono situazioni che il sistema dovrebbe gestire normalmente e classi di equivalenza non valide che descrivono situazioni non valide che il sistema dovrebbe rifiutare. Di solito si lavora con più insiemi di classi di equivalenza contemporaneamente. Per far ciò creiamo un insieme di test validi con un membro valido di ogni partizione di equivalenza, continuiamo questo processo fino a quando ogni classe valida è rappresentata in almeno un test valido. Successivamente, possiamo creare test non validi selezionando un membro non valido per una partizione di equivalenza e un membro valido per ogni altra partizione (non combinare più membri non validi in un singolo test perché un valore non valido potrebbe mascherare la gestione non corretta di un altro valore non valido); continuiamo il processo fino a quando ogni classe non valida è rappresentata in almeno un test non valido. Nel far ciò dobbiamo stare attenti ai seguenti errori:

- Le partizioni di equivalenza devono essere disgiunte (nessuno dei due sottoinsiemi può avere uno o più membri in comune).
- Nessuna delle partizioni di equivalenza può essere vuota.
- L'unione dei sottoinsiemi prodotti dal partizionamento per equivalenza deve essere uguale all'insieme originale che è stato partizionato per equivalenza.

Per chiarire il concetto vediamo una serie di esempi.



Si partiziona l'insieme X in due sottoinsiemi, X_1 e X_2 , si suddivide l'insieme Y in tre sottoinsiemi, Y_1 , Y_2 e Y_3 , si seleziona i valori dei test da ciascuno dei cinque sottoinsiemi, X_1 , X_2 , Y_1 , Y_2 e Y_3 . Comporremo quindi tre test, dal momento che possiamo combinare i valori dei sottoinsiemi X con i valori dei sottoinsiemi Y (supponendo che i valori siano indipendenti e tutti validi). In alcuni casi, i valori non sono validi. Per esempio, stiamo testando un'applicazione per la gestione dei progetti: X è il tipo di evento con cui abbiamo a che fare, un'attività (X_1) o una milestone (X_2), Y è la data di inizio dell'evento, che può essere nel passato (Y_1), oggi (Y_2) o nel futuro (Y_3), Z è la data di fine dell'evento, che può essere o alla data di inizio (Z_1) o prima della data di inizio (Z_2). In questo caso come ovvio che sia Z_2 non è valido.



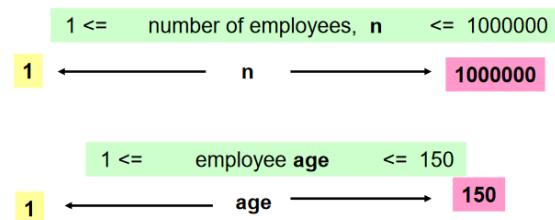
Vediamo un ulteriore esempio.

Apply for Product?	<input type="button" value="Select a product"/>	▼
	Home equity loan	
	Home equity line of credit	
	Reverse mortgage	
Existing Checking?	<input type="button" value="Select Yes or No"/>	▼
	<input type="button" value="If Yes input account number"/>	
	Yes	
	No	
Existing Savings?	<input type="button" value="Select Yes or No"/>	▼
	<input type="button" value="If Yes input account number"/>	
	Yes	
	No	

Questa schermata richiede tre informazioni: Il prodotto per il quale si fa richiesta, che è uno dei seguenti (Prestito di capitale per la casa, Linea di credito per la casa e Mutuo inverso), se il richiedente ha un conto corrente Globobank esistente, che può essere Sì o No, se si dispone di un conto di risparmio Globobank, sì o no. Se l'utente indica un conto Globobank esistente, deve inserire il numero di conto corrispondente; Questo numero viene convalidato nel database centrale della banca al momento dell'inserimento. Se l'utente non indica alcun conto, deve lasciare vuoto il campo del numero di conto corrispondente. Se i campi sono validi, compresi quelli relativi al numero di conto, la schermata verrà accettata altrimenti verrà visualizzato un messaggio di errore. Per il campo applicazione-prodotto, le partizioni di equivalenza sono: Prestito per l'acquisto di beni immobili / Linea di credito per l'acquisto di beni immobili / Mutuo ipotecario inverso. Le partizioni delle informazioni sul conto corrente esistente sono: Sì-Valido / Sì-Invalido / No-Vuoto / No-Non Vuoto. Le partizioni di informazioni sul conto di risparmio esistenti sono: Sì-Valido / Sì-Invalido / No-Vuoto / No-Non Vuoto. I test case generati saranno i seguenti:

Inputs	1	2	3	4	5	6	7
Product	HEL	LOC	RM	HEL	LOC	RM	HEL
Existing Checking?	Yes	No	No	Yes	No	No	No
Checking Account	Valid	Blank	Blank	Invalid	Nonblank	Blank	Blank
Existing Savings?	No	Yes	No	No	No	Yes	No
Savings Account	Blank	Valid	Blank	Blank	Blank	Invalid	Nonblank
Outputs							
Accept?	Yes	Yes	Yes	No	No	No	No
Error?	No	No	No	Yes	Yes	Yes	Yes

In questa tecnica molti errori vengono effettivamente commessi ai confini piuttosto che in condizioni normali. Un'altra tecnica che possiamo utilizzare è la **Boundary Value Analysis**, la quale utilizza le stesse classi dell'equivalenza, testando i confini piuttosto che un elemento della classe. Questa tecnica può generare un numero elevato di casi di test ma è possibile di solito ridurre i casi di test limitandosi a testare sopra e sotto i confini. Di norma vengono utilizzati solo due valori limite per classe ed il confine si trova tra il membro più grande di una classe di equivalenza e il membro più piccolo della classe di equivalenza superiore. Tuttavia, alcuni autori sostengono che i valori limite sono tre; si sceglie il valore stesso come valore limite intermedio, poi si aggiunge a quel valore la quantità più piccola possibile e si sottrae la più piccola quantità possibile a quel valore per generare gli altri due valori limite.



The “basic” boundary value testing for a value would include:

1. - at the “minimum” boundary
2. - immediately above minimum
3. - between minimum and maximum (nominal)
4. - immediately below maximum
5. - at the “maximum” boundary

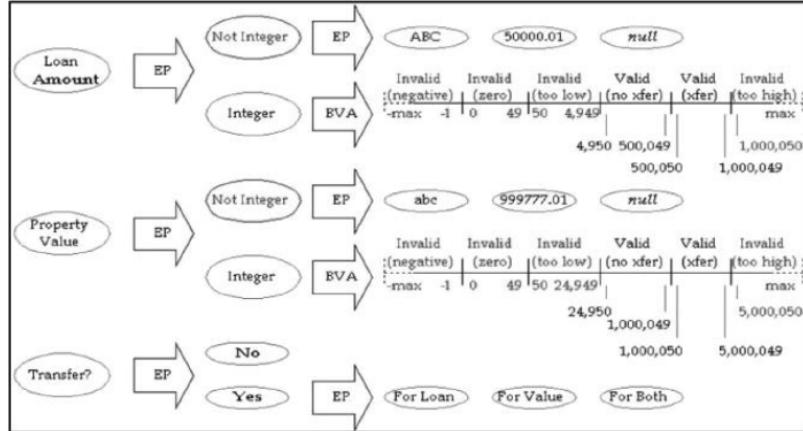
Vediamo come applicare questa tecnica al problema della schermata hellocare.

The screenshot shows a user interface for a loan application. It includes two input fields with associated validation messages:

- Loan amount?** Input field with placeholder "[Enter a loan amount]". Below it is a note: "Whole dollar amounts (no cents). System will round to nearest \$100."
- Property value?** Input field with placeholder "[Enter the property's value]". Below it is a note: "Whole dollar amounts (no cents). System will round to nearest \$100."

L'importo minimo del prestito è di 5.000 dollari e l'importo massimo è di \$1.000.000; il valore minimo della proprietà è di \$25.000 ed il valore massimo è \$5.000.000. Se i campi sono validi, allora la schermata sarà accettata. Se uno o entrambi i campi non sono validi, viene visualizzato un messaggio di errore. Inoltre indicare al centralinista di trasferire la chiamata ad un banchiere telefonico se la richiesta ha un importo di prestito superiore a 500.000 o un proprietà di

valore superiore a 1.000.000 di dollari; questi prestiti richiedono un'ulteriore approvazione da parte della direzione. Le classi di equivalenza saranno le seguenti:



I test case da generare sono riportati nella figura sottostante.

Inputs	1	2	3	4	5	6
Loan amount	4,950	500,050	500,049	1,000,049	ABC	50,000.01
Property value	24,950	1,000,049	1,000,050	5,000,049	100,000	200,000
Outputs						
Accept?	Y	Y	Y	Y	N	N
Transfer?	N	Y (loan)	Y (prop)	Y (both)	-	-

Inputs	7	8	9	10	11	12
Loan amount	null	100,000	200,000	300,000	-max	-1
Property value	300,000	abc	999,777.01	null	400,000	500,000
Outputs						
Accept?	N	N	N	N	N	N
Transfer?	-	-	-	-	-	-

Inputs	13	14	15	16	17	18
Loan amount	0	49	50	4,949	1,000,050	max
Property value	600,000	700,000	800,000	900,000	1,000,000	1,100,000
Outputs						
Accept?	N	N	N	N	N	N
Transfer?	-	-	-	-	-	-

Inputs	19	20	21	22	23	24
Loan amount	400,000	500,000	600,000	700,000	800,000	900,000
Property value	-max	-1	0	49	50	24,949
Outputs						
Accept?	N	N	N	N	N	N
Transfer?	-	-	-	-	-	-

Inputs	25	26	27	28	29	30
Loan amount	1,000,000	555,555				
Property value	5,000,050	max				
Outputs						
Accept?	N	N				
Transfer?	-	-				

Un'altra tecnica da utilizzare è la **Decision table**, che ci permette di testare il modo in cui varie combinazioni di condizioni interagiscono per produrre determinati risultati che si verificano e che non si verificano. La tabella dispone le combinazioni di condizioni in colonne, con le condizioni in alto e le azioni associate intraprese e non intraprese in basso. Vediamo subito un esempio: Form di validazione pagamento in cui dobbiamo inserire il tipo di carta di credito, il numero di carta, il codice di sicurezza della carta, mese di scadenza, anno di scadenza e nome del titolare della carta. Una volta che queste informazioni vengono inviate alla società di elaborazione delle carte di credito per la convalida, come possiamo verificarle? La persona indicata è in possesso della carta di credito inserita e le altre informazioni sono corrette? Il conto è ancora attivo o è stato cancellato? La persona ha raggiunto o superato il suo limite? La transazione proviene da un luogo normale o sospetto?

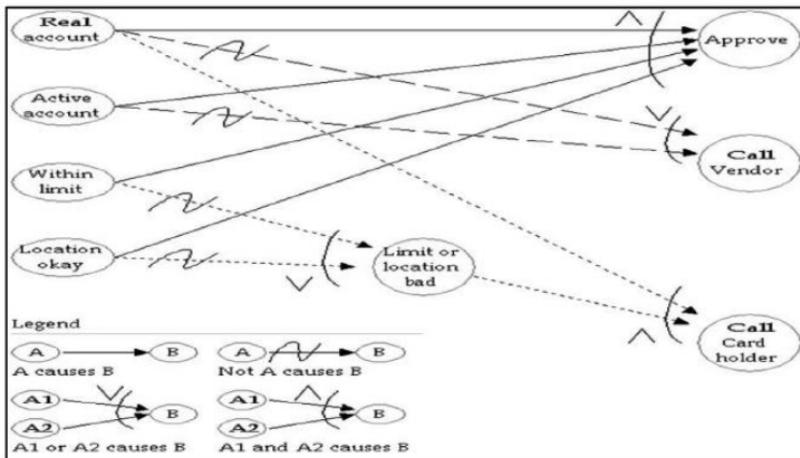
Conditions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Real account?	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N
Active account?	Y	Y	Y	Y	N	N	N	N	Y	Y	Y	Y	N	N	N	N
Within limit?	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N	Y	Y	N	N
Location okay?	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N	Y	N
Actions																
Approve?	Y	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
Call cardholder?	N	Y	Y	Y	N	Y	Y	Y	N	N	N	N	N	N	N	N
Call vendor?	N	N	N	N	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y

La tabella delle decisioni mostra come le condizioni interagiscono per determinare quale delle tre azioni seguenti si verificherà: Approvare la transazione? Chiamare il titolare della carta? Chiamare il fornitore?

È possibile comprimere la tabella delle decisioni, combinando le colonne, per ottenere una tabella delle decisioni più concisa.

Conditions	1	2	3	5	6	7	9
Real account?	Y	Y	Y	Y	Y	Y	N
Active account?	Y	Y	Y	N	N	N	-
Within limit?	Y	Y	N	Y	Y	N	-
Location okay?	Y	N	-	Y	N	-	-
Actions							
Approve?	Y	N	N	N	N	N	N
Call cardholder?	N	Y	Y	N	Y	Y	N
Call vendor?	N	N	N	Y	Y	Y	Y

Un'ulteriore tecnica utilizzata è il **grafico causa-effetto**, il quale è la rappresentazione grafica della stessa logica aziendale mostrata in una tabella decisionale.(possiamo sempre convertire l'una nell'altra).



Infine l'ultima tecnica per test black-box è il **test di casi d'uso**. È un modo per garantire che siano stati testati flussi di lavoro e scenari tipici ed eccezionali per il sistema, dal punto di vista dei vari attori che interagiscono direttamente con il sistema.

exceptions

#	Test Step	Expected Result
1	Place one item in cart	Item in cart
2	Click check out	Checkout screen
3	Input valid US address, valid payment using American Express, and valid shipping method information	Each screen displays correctly, and valid inputs are accepted
4	Verify order information	Shown as entered
5	Confirm order	Order in system
6	Repeat steps 1–5, but place two items in cart, pay with Visa, and ship internationally	As shown in 1–5
7	Repeat steps 1–5, but place the maximum number of items in cart and pay with MasterCard	As shown in 1–5
8	Repeat steps 1–5, but pay with Discover	As shown in 1–5

#	Test Step	Expected Result
1	Do not place any items in cart	Cart empty
2	Click check out	Error message
3	Place item in cart, click check out, enter invalid address, then invalid payment, then invalid shipping information	Error messages, can't proceed to next screen until resolved
4	Verify order information	Shown as entered
5	Confirm order	Order in system
6	Repeat steps 1–3, but stop activity and abandon transaction after placing item in cart	User logged out exactly ten minutes after last activity
7	Repeat steps 1–3, but stop activity and abandon transaction on each screen	As shown in 6
8	Repeat steps 1–4; do not confirm order	As shown in 6

5.2 White-Box Testing

I principali test che fanno parte di questa categoria sono i seguenti:

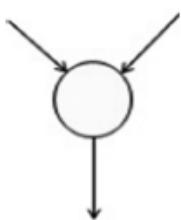
- Condition Testing
- Decision Condition Testing
- Modified Condition/Decision Testing (MC/DC)
- Multiple Condition Testing
- Path Testing
- API Testing

Per eseguire questa serie di test al **Control Flow Testing**, un approccio al testing basato su delle strutture grafiche in cui i casi di test sono progettati per eseguire specifiche sequenze di eventi. Il test del flusso di controllo viene effettuato attraverso i grafici del flusso di controllo, che forniscono un modo per astrarre un modulo di codice per capire meglio cosa fa. Per costruire il grafico facciamo ricorso a 3 principali componenti:

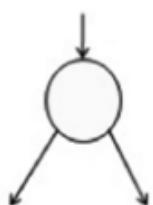
1. Il blocco del processo, graficamente costituito da un nodo (bolla o cerchio) con un percorso che conduce ad esso e un percorso che parte da esso.

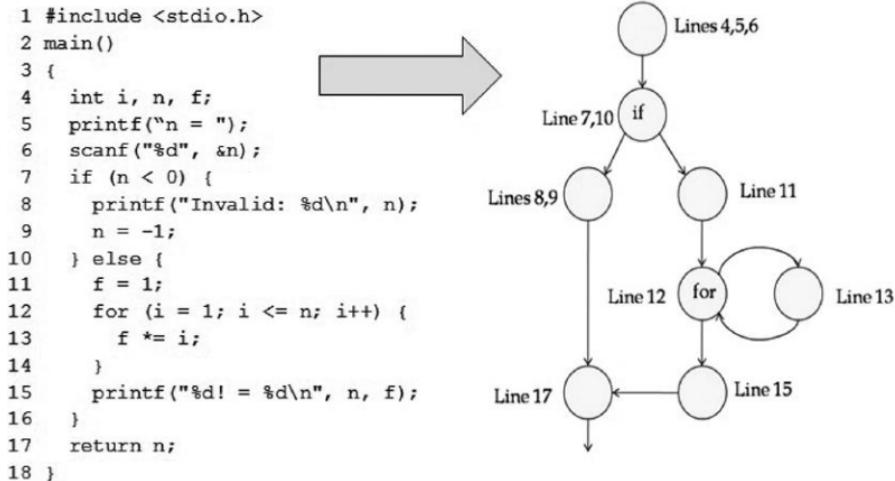


2. punto di giunzione



3. punto di decisione



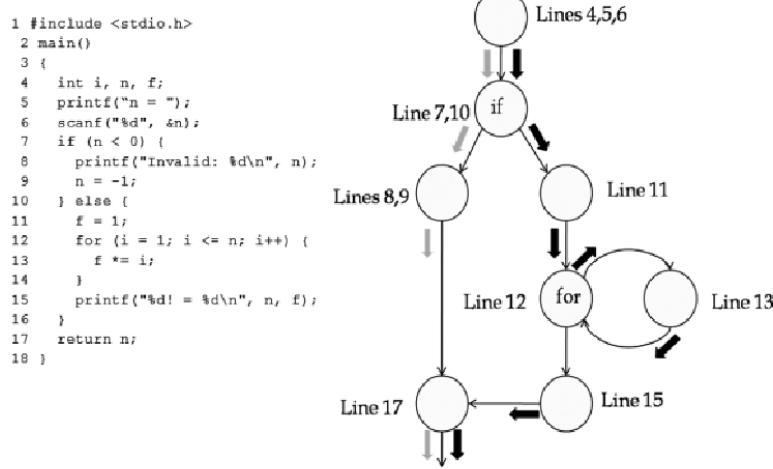


Quanto dobbiamo testare? Le risposte possibili vanno da nessun test a un test totale ed esaustivo che colpiscono ogni possibile percorso attraverso il software.

- **Statement Coverage**

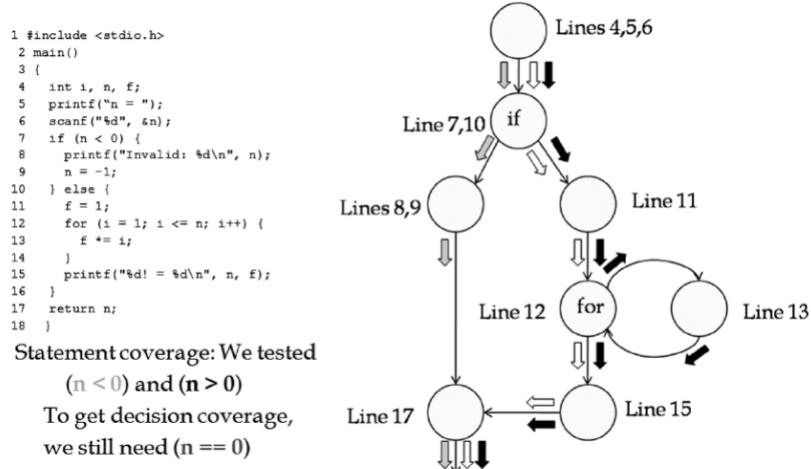
La copertura a livello di dichiarazione è considerata la meno efficace di tutte le tecniche di flusso di controllo (deve essere al 100%). Per raggiungere questo modesto livello di copertura, un tester deve proporre un numero sufficiente di casi di test per forzare l'esecuzione di ogni riga almeno una volta. IEEE nello standard di test unitario ANSI 87B (1987), ha dichiarato che la copertura delle dichiarazioni è il livello minimo di copertura che dovrebbe essere accettabile. Anche Boris Beizer, famoso autore ed ingegnere del software, ha dichiarato che "testing less than this for new software is unconscionable and should be criminalized". In particolar modo Beizer ha stilato una serie di regole per il testing tra le quali le più importanti sono:

1. Not testing a piece of code leaves a residue of bugs in the program in proportion to the size of the untested code and the probability of bugs.
2. The high-probability paths are always thoroughly tested if only to demonstrate that the system works properly. If you have to leave some code untested at the unit level, it is more rational to leave the normal, high-probability paths untested, because someone else is sure to exercise them during integration testing or system testing (meglio testare percorsi meno utilizzati).



• Decision/Branch Coverage

Questo livello di copertura esamina le decisioni stesse. Ogni decisione ha la possibilità di essere risolta come TRUE o FALSE (nell'istruzione switch ogni decisione atomica è un confronto tra due valori TRUE o FALSE). La copertura decisionale implica anche la copertura di statement.



• Loop Coverage

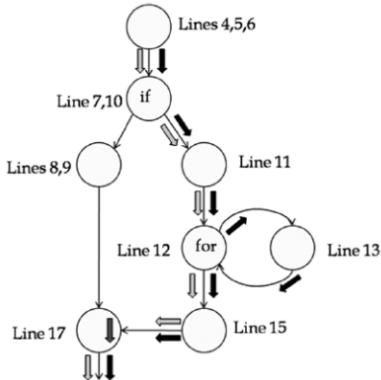
Se vogliamo testare completamente un ciclo, dobbiamo testarlo zero volte, una volta, due volte, tre volte, fino a n volte. Il minimo di base tende a essere due test, zero volte attraverso il ciclo e una volta attraverso il ciclo (per ottenere una copertura decisionale). Si suggerisce però di provare a testare il ciclo zero volte ed una sola volta e poi di testare il numero massimo di

volte in cui ci si aspetta che il ciclo venga eseguito (se si sa quante volte è probabile che sia).

```

1 #include <stdio.h>
2 main()
3 {
4     int i, n, f;
5     printf("n = ");
6     scanf("%d", &n);
7     if (n < 0) {
8         printf("Invalid: %d\n", n);
9         n = -1;
10    } else {
11        f = 1;
12        for (i = 1; i <= n; i++) {
13            f *= i;
14        }
15        printf("%d! = %d\n", n, f);
16    }
17    return n;
18 }
```

Loop 0 times by testing ($n=0$)
Loop 1 time by testing ($n = 1$)



• Condition Coverage

Una tecnica di progettazione di test white-box in cui i casi di test vengono progettati per eseguire i risultati delle condizioni. Il concetto di base è che, quando una decisione è presa da un'espressione complessa che valuta VERO o FALSO, ogni condizione atomica viene testata in entrambi i modi, VERO e FALSO. Cos'è una condizione atomica? una condizione atomica è una condizione che non può essere scomposta, cioè una condizione che non contiene due o più condizioni singole unite da un operatore logico (AND, OR). La copertura delle decisioni e delle condizioni sarà sempre esattamente la stessa quando tutte le decisioni sono prese da semplici condizioni atomiche($(if == 1)$).

<i>if (A && B) then</i>	
<i>{Do something}</i>	Test 1: $A == \text{FALSE}, B == \text{TRUE}$ resolves to FALSE
<i>else</i>	Test 2: $A == \text{TRUE}, B == \text{FALSE}$ resolves to FALSE
<i>{Do something else}</i>	

La copertura delle condizioni non è più forte della copertura delle decisioni, perché il 100 per cento di copertura delle condizioni non implica il 100% di copertura delle decisioni.

• Decision Condition Coverage

Questo livello di copertura non è altro che una combinazione della copertura di condizioni e della copertura di decisioni.. Dobbiamo ottenere una copertura a livello di condizione in cui ogni condizione atomica condizione

atomica viene testata in entrambi i modi, VERO e FALSO, e dobbiamo anche assicurarci di raggiungere la copertura della decisione, garantendo che il predicato complessivo sia testato in entrambi i modi, VERO e FALSO. Vediamo un esempio: (A AND B) A è impostato su FALSO e B su VERO, A è impostato su VERO e B su FALSO, A e B sono entrambi impostati su VERO, A e B sono entrambi impostati su FALSO.

- **Modified Condition/Decision Coverage**

Per ogni condizione atomica, esiste almeno un caso di test in cui l'intero predicato viene valutato FALSO solo perché quella specifica condizione atomica ha valutato FALSO. E viceversa per ogni condizione atomica, esiste almeno un caso di test in cui l'intero predicato è valutato come VERO solo perché quella specifica condizione atomica ha dato esito VERO. Quindi controlla la decisione nell'interezza e l'impatto di una delle condizioni atomiche. Molti riferimenti utilizzano il termine MC/DC come sinonimo di test esaustivo. test ma tuttavia, questo livello di test non raggiunge il livello di test esaustivo. Inoltre la copertura MC/DC può non essere utile a tutti i tester. In questa copertura possiamo definire il valore neutro per una condizione atomica come quello che non ha alcun effetto sulla valutazione del risultato finale dell'espressione. Esempio: A AND B, il valore neutro sarà TRUE; A OR B, il valore neutro sarà FALSO.

(A OR B) AND (C AND D)	T	F
A	T F T T	F F T T
B	F T T T	F F T T
C	T F T T	T T F T
D	T F T T	T T T F

(A OR B) AND (C AND D)	T	F
A	T F T T	F F T T
B	F T T T	F F T T
C	T F T T	T T F T
D	T F T T	T T T F

Questa tecnica ci sono 2 principali problemi:

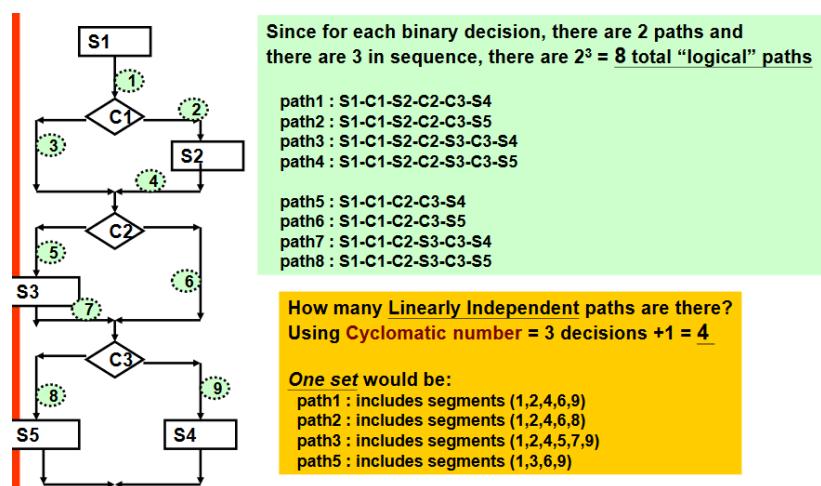
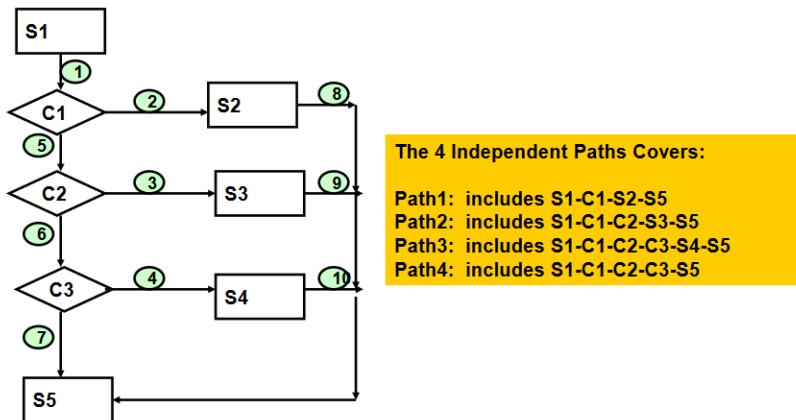
1. Short-Circuiting, le espressioni possono essere risolte senza valutare ogni sottoespressione, a seconda dell'operatore booleano utilizzato. In quest caso non stiamo valutando tutte le condizioni.
2. Coupling, questo accade quando ci sono più occorrenze di una condizione in un'espressione. Esempio: $A \text{OR}(\text{!}A \text{AND} B)$, A e $\text{!}A$ sono accoppiati.

- **Multiple Condition Coverage**

Il concetto è quello di testare ogni possibile combinazione di condizioni atomiche in una decisione. Si tratta di un test esaustivo

- **Path Testing via Flow Graphs**

Il primo passo di questa tecnica di copertura dei percorsi è la creazione di un grafico del flusso di controllo per il codice che vogliamo testare. Noi utilizzeremo la seguente regola: "scegliete il percorso di ingresso/uscita più semplice e funzionalmente sensato". Per far ciò verifichiamo i percorsi linearmente indipendenti, cioè i percorsi attraverso l'applicazione che introducono almeno un nuovo nodo che non è incluso in nessun altro percorso linearmente indipendente.



5.3 Automated Unit Testing

Vediamo ora le pratiche più efficaci e comuni utilizzate per automatizzare i test. I programmatore inesperti tendono a eseguire test unitari limitati, piuttosto che scrivere grandi pezzi di codice e testare solo le parti di alto livello. Un errore comune è scrivere i casi di test come parte della funzione principale e di scartarli dopo l'esecuzione. Dovremmo, invece, mantenere il patrimonio dei test e per far ciò una pratica migliore è quella di utilizzare uno strumento di unit testing automatizzato come JUnit per Java. La buona pratica sarebbe di scrivere i test unitari subito dopo aver scritto un pezzo di codice o ancora meglio scrivere i test unitari ancora prima di scrivere il codice.

When to Stop Testing? Una risposta semplice è quella di interrompere i test quando tutti i casi di test pianificati vengono eseguiti e tutti i problemi riscontrati sono stati risolti(nella realtà, spesso siamo costretti a rilasciare il prodotto software per motivi di tempo).

5.4 Inspection

Una tecnica efficace dal punto di vista dei costi per individuare gli errori è la revisione del codice o dei documenti intermedi da parte di un team di sviluppatori di software. Le ispezioni del software sono revisioni dettagliate del lavoro in corso e consistono solitamente nelle seguenti fasi:

1. Pianificazione: Vengono designati un team di ispezione e un moderatore e vengono distribuiti materiali ai membri del team alcuni giorni prima dell'evento.
2. Panoramica: Viene presentata una panoramica del prodotto di lavoro e delle aree ad esso correlate
3. Preparazione: Ogni ispettore deve studiare il prodotto di lavoro e i materiali correlati. Le liste di controllo possono essere utilizzate per individuare i problemi più comuni.
4. Esame: Si organizza una riunione in cui gli ispettori esaminano insieme il prodotto.
5. Rilavorazione: Dopo la riunione, l'autore corregge tutti gli eventuali difetti
6. Follow-up: le correzioni vengono controllate dal moderatore, oppure il lavoro corretto viene ispezionato di nuovo, a seconda del risultato dell'ispezione.

Le ispezioni hanno dimostrato di essere una tecnica efficace dal punto di vista dei costi per trovare i difetti.

► **Inspections**

- Partially Cost-effective
- Can be applied to intermediate artifacts
- Catches defects early
- Helps disseminate knowledge about project and best practices

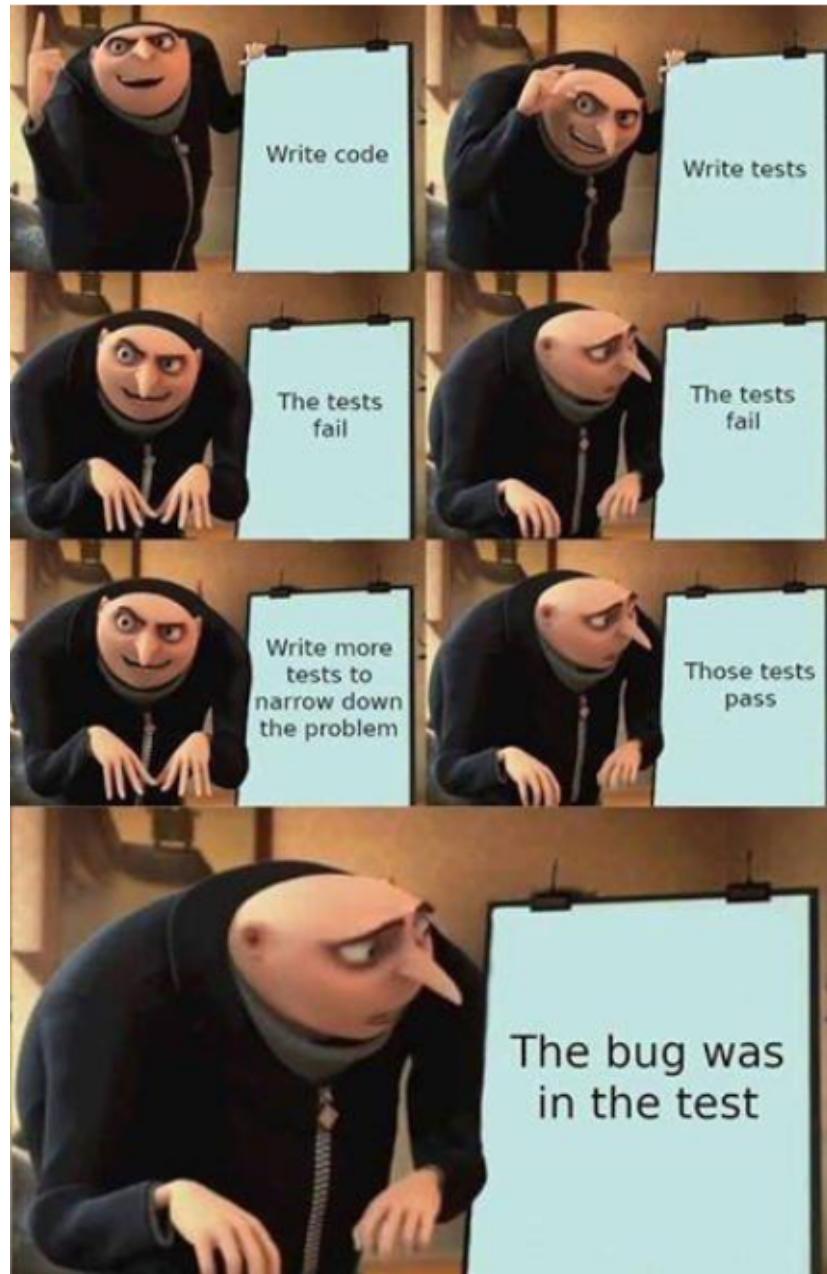
► **Testing**

- Finds errors cheaper, but correcting them is expensive
- Can only be applied to code
- Catches defects late (after implementation)
- Necessary to gauge quality

Per eseguire l'inspezione possiamo utilizzare le seguenti tecniche:

- Metodi formali, tecniche matematiche utilizzate per dimostrare che un programma funziona ma richiede una formazione matematica, non sono applicabili a tutti i programmi, solo verifica, non validazione e non applicabile a tutti gli aspetti del programma (UI).
- Analisi statiche, prevede l'esame delle strutture statiche di file eseguibili e non eseguibili con l'obiettivo di rilevare condizioni di errore. Un tool molto utilizzato (incorporato anche da Google) è FindBugs, un programma che utilizza l'analisi statica per cercare i bug nel codice Java. Questo programma fa un parsing del codice ed individua le zone in cui potrebbero esserci degli errori.

```
public String foundType(){  
    return this.foundType();  
}  
// funzione richiamata in modo recursivo va in loop  
  
s.toLowerCase();  
// non c'e' assegnamento
```



5.5 JUnit

JUnit è un framework semplice e open source per scrivere ed eseguire test ripetibili. JUnit permette di utilizzare asserzioni per testare i risultati attesi, test

fixtures per la condivisione di dati di test comuni e test runner per l'esecuzione dei test. Il progetto dovrebbe essere strutturato nel seguente modo:

- directory src/main -> per il codice
- directory src/test -> per i file di test

Vediamo subito un esempio

```
public class StringHelper {  
    //AABD --> BD, ABD --> BD CDAA --> CDAA  
    public String truncateAInFirst2Positions(String str)  
    {  
        if (str.length() <= 2)  
            return str.replaceAll("A", "");  
        String first2Chars = str.substring(0, 2);  
        String stringMinusFirst2Chars = str.substring(2);  
        return first2Chars.replaceAll("A", "")  
            + stringMinusFirst2Chars;  
    }  
    public boolean  
        areFirstAndLastTwoCharactersTheSame(String str) {  
        //AABD --> false, ABDAB --> true AA --> true  
        if (str.length() <= 1)  
            return false;  
        if (str.length() == 2)  
            return true;  
        String first2Chars = str.substring(0, 2);  
        String last2Chars = str.substring(str.length() -  
            2);  
        return first2Chars.equals(last2Chars);  
    }  
}  
  
package test;  
import static org.junit.Assert.*;  
import org.junit.Test;  
  
public class StringHelperTest {  
    @Test  
    public void testTruncateAInFirst2Positions() {  
        fail("Not yet implemented");  
    }  
    @Test  
    public void  
        testAreFirstAndLastTwoCharactersTheSame() {  
        fail("Not yet implemented");  
    }  
}
```

```
}
```

Per definizione ed per il modo in cui è stato implementato questo test fallirà (istruzione fail). Per testare le funzionalità possiamo usare la classe java **Assert** che fornisce un insieme di metodi di asserzione utili per la scrittura di test.

- void assertEquals(boolean/String/Int/... expected, assertEquals(boolean/String/Int/... actual) → verifica che due primitive/oggetti siano uguali.
- void assertTrue(boolean expected, boolean actual) → verifica che una condizione sia vera.
- void assertFalse(boolean condition) → verifica che una condizione sia falsa

Il metodo di test deve testare una sola condizione e il suo nome deve essere test + nome del metodo da testare.

Tramite le keyword Before e After possiamo decidere di far eseguire alcuni metodi (per preparare l'ambiente di test o per pulire eventuali tracce del test) prima e dopo il test.

```
public class BeforeAndAfterTest {  
    @Before  
    public void setup() {  
        System.out.println("Before the test");  
    }  
    @Test  
    public void test1() {  
        System.out.println("Test 1 executed");  
    }  
    @Test  
    public void test2() {  
        System.out.println("Test 2 executed");  
    }  
    @After  
    public void close_test() {  
        System.out.println("Close the test");  
    }  
}
```

Grazie a JUnit possiamo anche testare un'eccezione, cioè testare che si verifichi correttamente una determinata eccezione.

```
@Test(expected=NullPointerException.class)  
public void testArraySort_NullPointer() {  
    int[] numbers = null;  
    int[] expected = {1,4,5,8,12};  
    Arrays.sort(numbers);  
    assertEquals(expected, numbers);  
}
```

Infine possiamo testare le prestazioni di un metodo cosa da poter identificare le funzionalità critiche dal punto di vista prestazionale del nostro progetto software.

```
public class testPerformance {
    @Test(timeout=100)
    public void testArraySort_Performance() {
        int[] numbers = {1,5,4,8,12};
        for(int i=0;i<=100000000;i++){
            numbers[0] = i;
            Arrays.sort(numbers);
        }
    }
}
```

6 Distribuzione e Manutenzione

6.1 Configuration Management

La gestione della configurazione del software è il processo di gestione di tutti pezzi e parti di artefatti prodotti nell'ambito delle attività di sviluppo e supporto del software. Il processo di sviluppo del software determina i pezzi da gestire ed il processo scelto e gli output sviluppati giocano un ruolo importante nel determinare quali pezzi e quale livello di dettaglio dobbiamo gestire. Per esempio, un'organizzazione che adotta un approccio conservativo può decidere di gestire: il processo scelto e gli output sviluppati, specifiche dei requisiti, specifiche di progettazione, codice sorgente, codice eseguibile e casi di test. La gestione delle versioni degli artefatti è noiosa ma non molto complessa infatti complessità aumenta solo quando si devono controllare più artefatti e le relazioni tra gli artefatti. Invece la tracciabilità può essere un problema perché cade in cascata sulla manutenzione e l'aggiornamento futuro del software.

	Requirements Elements	Design Elements	Code Logic	UI Screens	DB tables	Initialization data	Test cases
Requirements Elements		X	X	X			X
Design Elements	X		X	X	X	X	
Code Logic	X	X		X	X	X	X
UI Screens	X	X	X			X	X
DB tables		X	X			X	
Initialization data		X	X	X	X		X
Test cases	X		X	X		X	

Figure 15: Relazioni tra artefatti e requisiti

La maggior parte dei tool di gestione del software (tracciabilità e manutenzione) utilizza un sistema di naming e numerazione progressiva (codice univoco per ogni artefatto).

Un tool molto famoso di gestione delle configurazione è **Github**.

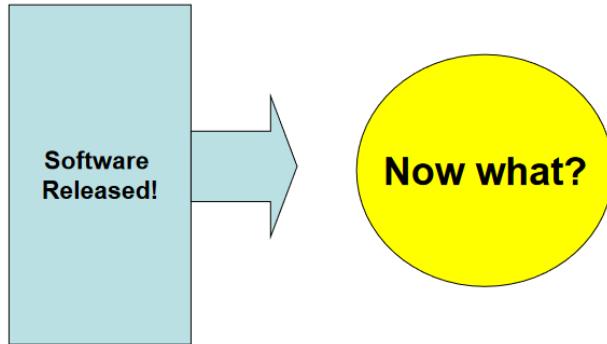
6.2 Integration and Builds

Il processo di compilazione è un insieme di attività associate all'integrazione e alla conversione dei file sorgenti in un insieme di file eseguibili destinati da uno specifico ambiente di esecuzione (hardware e sistema software).

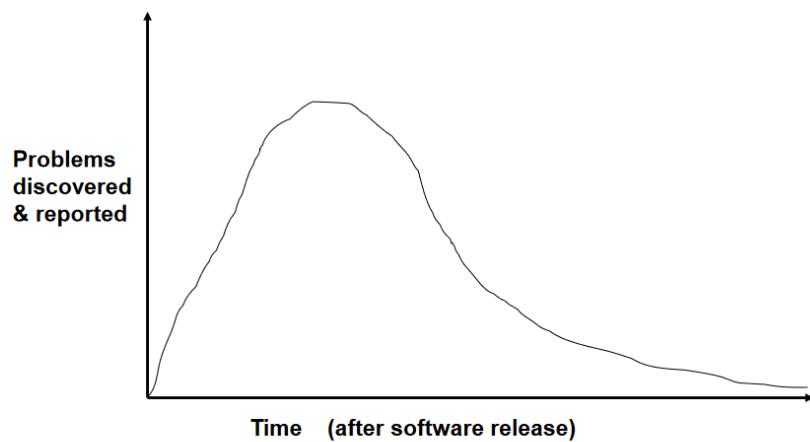
Un tool per la gestione della compilazione e integrazione è **Maven/Gradle**.

6.3 Software Support

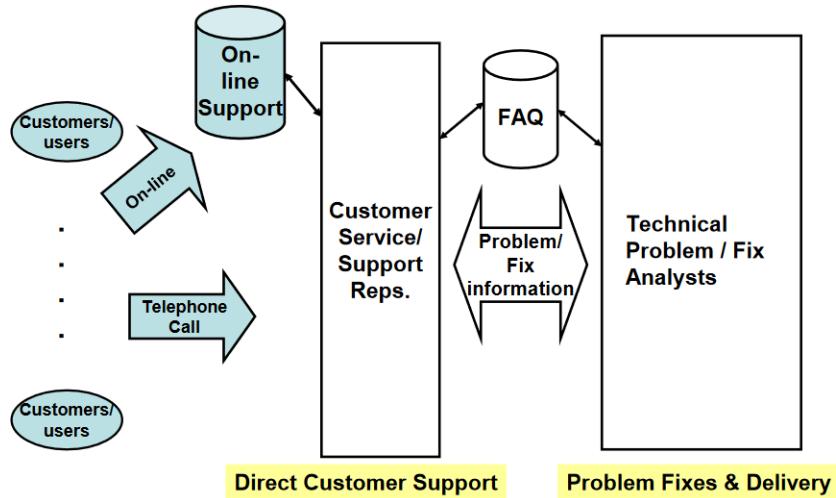
I grandi prodotti software sono sottoposti a un processo di sviluppo molto lungo e costoso. Processo di sviluppo molto lungo e costoso. Il ciclo di supporto e di manutenzione del prodotto dopo il rilascio è diverse volte più lungo del ciclo di sviluppo (il tempo di sviluppo è di 2/3 mesi invece il supporto software è di 40/60 mesi). Il successo del prodotto dipende dall'esperienza degli utenti reali.



Molti prodotti software di grandi dimensioni sono complessi e contengono difetti al momento del rilascio del prodotto ed è estremamente importante che vi sia una buona comprensione e una preparazione per l'assistenza ai clienti dopo il rilascio (molte volte a pagamento). Per garantire un efficace supporto dobbiamo essere in grado di prevedere il picco di utenza, il quale può avvenire al momento del rilascio ma anche dopo diversi mesi.



Il modello più utilizzato per il supporto software è quello rappresentato in figura.



Il rappresentante dell'assistenza segue una sequenza di attività:

- Identificare l'utente per assicurarsi che si tratti di un cliente qualificato (ad esempio, un cliente che ha pagato il servizio software).
- Ascoltare e registrare la descrizione del problema.
- Eseguire una scansione del database delle FAQ per verificare la presenza di problemi/soluzioni simili.
- Fornire la risposta se si tratta di un problema già segnalato e risolto
- Fornire una data di risoluzione prevista se si tratta di un problema segnalato in precedenza ma non risolto
- Se si tratta di un nuovo problema, fornire una soluzione se possibile. Se non è possibile trovare una soluzione, concordare con il cliente un livello di priorità per il problema e un tempo di risoluzione stimato per il cliente.
- Registrare il problema e inviare un rapporto al team di correzione tecnica.

Ai rappresentanti dell'assistenza viene anche chiesto di gestire un sito web online per l'assistenza clienti asincrona. Tutti i clienti esperti sanno che il modo per ricevere una considerazione rapida e veloce è quello di far valutare la priorità del problema il più possibile.

Priority Level	Problem Category	Fix Response Time
1	Severe functional problem with no work-around	As soon as possible
2	Severe functional problem but has work-around	1 – 2 weeks
3	Functional problem that has a work-around	3 – 4 weeks
4	Nice to have or to change	Next product release or earlier

Se il problema va oltre le capacità dell'addetto all'assistenza, si ricorre all'analista dei problemi tecnici.. Viene creata una richiesta di modifica formale (la descrizione del problema, la priorità del problema e altre informazioni correlate vengono registrate in un rapporto sul problema) in modo da generare una soluzione permanente. L'analista tecnico è di solito un ingegnere ben qualificato, esperto nella progettazione, nella codifica e nel collaudo.
Il ciclo di progettazione, codifica e collaudo di un problema di correzione non è diverso dalle attività di progettazione, codice e test del ciclo di sviluppo.

6.4 Maintenance

I rilasci di correzioni regolari e periodiche hanno un ciclo trimestrale o semestrale e contengono solo i moduli e il codice interessati dalle correzioni dei problemi. È importante ricordare che non tutti i clienti applicheranno (o installeranno) immediatamente i rilasci di correzioni e quindi potrebbero riscontrare un problema in un secondo momento e desiderare una particolare correzione del problema. L'applicazione retroattiva di tutte le release di correzione può essere molto dispendiosa in termini di tempo e frustrante quando il cliente vuole risolvere solo un problema o una singola patch. Per mantenere l'impegno del servizio di assistenza a un livello ragionevole, i clienti sono sempre incoraggiati ad applicare la release di correzione il prima possibile dopo che è stata resa disponibile. Di norma quando è disponibile una nuova release di correzione, viene incluso un documento informativo che contiene l'elenco dei problemi, le relative correzioni e un'insieme di consigli per i clienti che potrebbero trovarsi in una versione diversa. Inoltre per gestire correttamente la manutenzione un'organizzazione di assistenza clienti di grandi dimensioni deve avere un controllo dei cambiamenti delle modifiche. In particolar modo, deve controllare tutte le modifiche, sia che esse derivino da un problema del cliente, sia che si tratti di un piccolo miglioramento incrementale delle funzionalità.

6.5 VCS - Git

Il controllo delle versioni, noto anche come controllo delle fonti, è la pratica di tracciare e gestire le modifiche al codice software. I sistemi di controllo delle versioni sono strumenti software che aiutano i team software a gestire le modifiche al codice sorgente nel tempo. Questi tool consentono lo sviluppo

collaborativo, di aprire chi ha apportato quali modifiche e quando e di ripristinare qualsiasi modifica e tornare ad uno stato precedente. Esistono 3 principali tipi di **Version Control System**:

- **Local Version Control Systems**

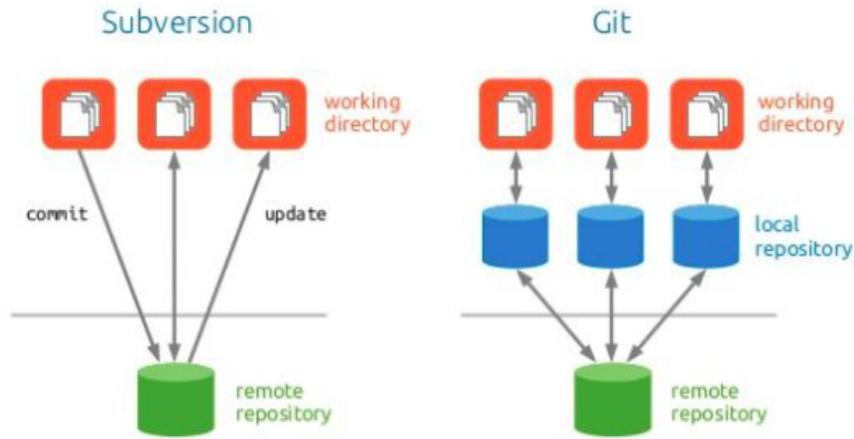
Un primo metodo banale di controllo della versione consiste nel copiare, durante lo sviluppo di un progetto, tutti i suoi file (e le relative modifiche nel tempo) in diversi file con data e ora. Questo approccio è molto comune perché è molto semplice, ma è anche incredibilmente incline agli errori; infatti è molto facile dimenticare quale sia la directory che contiene le modifiche di interesse e scrivere accidentalmente nel file sbagliato o copiare i file di destinazione. Un'estensione diretta di questo approccio è l'uso di un semplice database che conserva tutte le modifiche ai file sotto controllo di revisione. Tutti questi metodi però non forniscono alcuna soluzione all'esigenza di collaborazione.

- **Centralized Version Control Systems**

Un singolo server che contiene tutti i file versionati e un certo numero di client che prelevano i file da quella postazione centrale. I vantaggi principali sono che è facile da mantenere e c'è un singolo punto di guasto (crash del server).

- **Distributed Version Control Systems**

Con i sistemi di controllo di versione distribuiti, i clienti non si limitano a controllare l'ultima istantanea dei file dal server, ma eseguono il mirroring completo del repository, compresa la sua storia completa. In questo modo, tutti coloro che collaborano a un progetto possiedono una copia locale dell'intero progetto, cioè possiedono il proprio database locale con la propria storia completa. Con questo modello, se il server diventa indisponibile o muore, un qualsiasi dei repository client può inviare una copia della versione del progetto a qualsiasi altro client o tornare al server quando sarà disponibile. È sufficiente che un client contenga una copia corretta che può essere facilmente distribuita.



Git è l'esempio più noto di sistema di controllo di versione distribuito. Progetto open source sviluppato originariamente nel 2005 da Linus Torvalds per aiutare lo sviluppo del kernel di Linux (tool a linea di comando). In questo libro cercherò di definire i punti chiave di Git senza entrare nei dettagli:

- **Snapshot**

Uno Snapshot registra essenzialmente l'aspetto di tutti i file in un determinato momento ed è il modo in cui git tiene traccia della cronologia del codice. Decidete voi quando scattare un'istantanea e di quali file e c'è sempre la possibilità di tornare indietro per visitare qualsiasi snapshot.

- **Commit**

L'atto di creare un'istantanea (snapshot). Il messaggio di commit dovrebbe fornire alcune informazioni su come i file sono cambiati rispetto a quelli precedenti. Essenzialmente, un progetto è composto da un insieme di commit.

- **Clone**

L'atto di copiare un repository da un server remoto e che permette ai team di lavorare insieme.

- **Pull**

Il processo di scaricare da un repository remoto i commit che non esistono sulla vostra copia locale

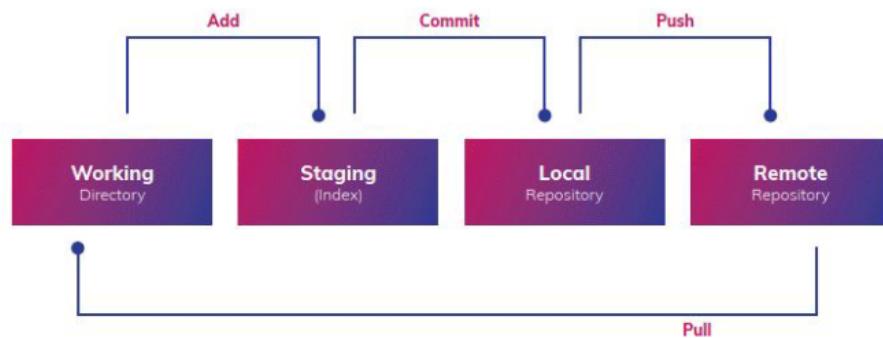
- **Push**

Il processo di aggiunta delle modifiche locali al repository remoto.

• Repository

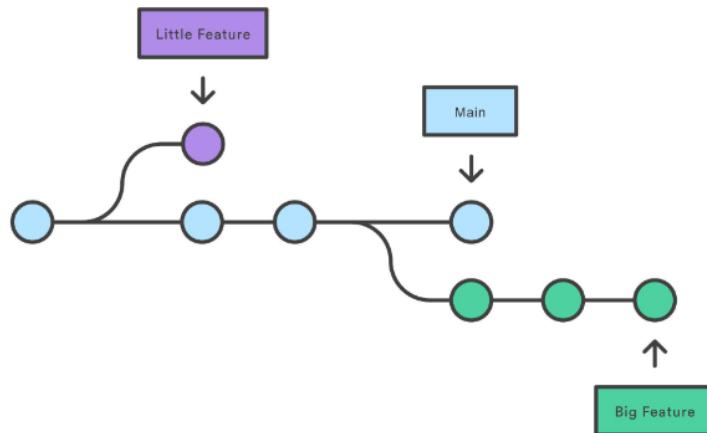
Git memorizza i file e la loro cronologia in una struttura di dati chiamata repository. Il flusso di lavoro di Git consiste in 4 fasi:

1. Introduzione di modifiche a un file nella directory di lavoro del progetto.
2. Aggiungere la modifica a un'area di staging (per esempio, usando git add).
3. Rendere la modifica "permanente", memorizzandola nel repository (ad esempio, usando git commit).
4. Spingere le modifiche locali su quelle remote per poter collaborare con gli altri.



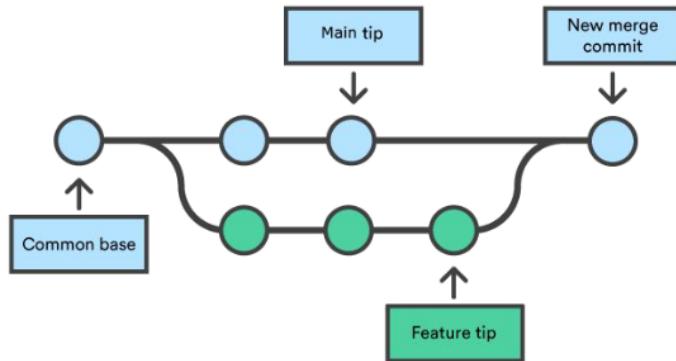
• Branch

È una linea di sviluppo indipendente. Tutti i commit in git risiedono in un ramo, ma possiamo avere molti rami all'interno del nostro repository (di norma il ramo principale di un progetto è chiamato Main Branch).



- **Merge**

Unisce più sequenze di commit (più rame) in un'unica cronologia unificata (operazione con estrema attenzione e cautela).

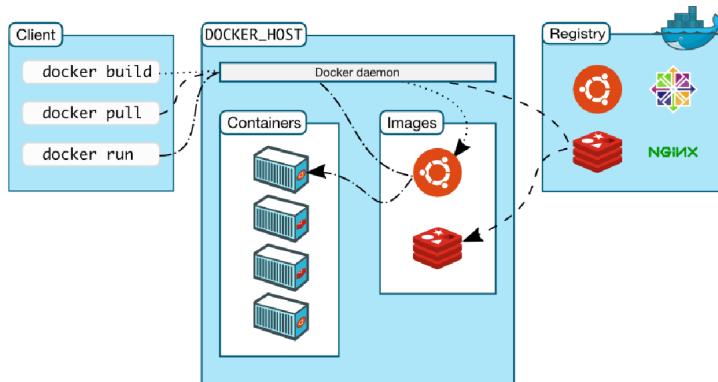


6.6 Docker

Wikipedia definisce Docker come un progetto open-source che automatizza la distribuzione di applicazioni software all'interno di container, fornendo un ulteriore livello di astrazione e automazione della virtualizzazione a livello di OS su Linux. Il vantaggio principale di Docker è che permette agli utenti di impacchettare un'applicazione con tutte le sue dipendenze in un'unità standardizzata per lo sviluppo del software. Lo standard del settore oggi è l'utilizzo di macchine virtuali (VM) per l'esecuzione di applicazioni software ma l'overhead computazionale speso per la virtualizzazione dell'hardware da utilizzare da parte di un sistema operativo è notevole ed qui che entra in gioco **Docker**. Infatti Docker offre la possibilità di pacchettizzare ed eseguire un'applicazione in un ambiente isolato chiamato contenitore. L'isolamento e la sicurezza consentono di eseguire molti container simultaneamente su un determinato host. Inoltre i container sono leggeri, non necessitano del carico aggiuntivo di un hypervisor (comunicatore tra kernel e VM), ma vengono eseguiti direttamente all'interno del kernel della macchina host. I contenitori offrono un meccanismo di impacchettamento logico in cui le applicazioni possono essere astratte dall'ambiente in cui vengono effettivamente eseguite. Questo disaccoppiamento permette alle applicazioni basate su container di essere distribuite facilmente e in modo coerente. Docker è molto utile in quelle situazioni in cui abbiamo bisogno di distribuzione rapida ed uniforme delle nostre applicazioni, distribuzione e scalabilità reattive ed esecuzione di un maggior numero di carichi di lavoro sullo stesso hardware. Vediamo ora un po' più nel dettaglio come è strutturato Docker, partendo dal **Docker Engine**. Docker Engine è un'applicazione client-server con i seguenti

componenti principali:

- Un server che è un tipo di programma a lunga durata chiamato processo daemon.
- Un'API REST che specifica le interfacce che i programmi possono usare per parlare con il demone ed istruirlo su cosa fare.
- Un client di interfaccia a riga di comando (CLI) (il comando docker).



Il demone Docker (dockerd) ascolta le richieste dell'API Docker e gestisce oggetti Docker come immagini, reti e volumi.

- Un'immagine è un modello di sola lettura con le istruzioni per creare un contenitore Docker. Per creare la propria immagine, si crea un file Docker con una sintassi che definisce i passaggi necessari per creare l'immagine ed eseguirla.
- Un contenitore è un'istanza eseguibile di un'immagine. È possibile creare, avviare, arrestare, spostare o cancellare un contenitore utilizzando l'API o la CLI di Docker. Per creare un container dobbiamo eseguire un'immagine creata tramite build da un dockerfile.
- I servizi consentono di scalare i container su più demoni Docker, che lavorano tutti insieme.

Il client Docker (docker) è il modo principale con cui molti utenti interagiscono con Docker, inviando comandi a dockerd, che li esegue, utilizzando l'API Docker. Un registro Docker memorizza le immagini Docker.