NoiStudiamo

APPUNTI

Ingegneria del Software

Programmazione a oggetti in Java, design pattern, UML, testing, metodi di sviluppo agili e Scrum

A CURA DI

Paolo Volpini

Novembre 2026

INDICE

Indice degli argomenti

Gli argomenti principali degli appunti e la pagina in cui iniziano.

1. Introduzione 3
2. Java: prima lezione 5
3. Introduzione alla Object Oriented Programming 9
4. Terza lezione 11
5. Utilizzo corretto della OOP 13
6. Design Patterns 18
7. Null Object 23
8. UML 26
9. Factory Method Pattern 36
10. Abstract Factory 43
11. Template, Strategy, State 49
12. Composite pattern 57
13. Observer pattern e programmazione reattiva 63
14. Lambda 69
15. Streams 73
16. Testing 77
17. Testing, 2 81
18. Modeling e design 89
19. Metodi di sviluppo 95
20. Metodi agili, refactoring 101
21. Scrum 107
22. Ultimi design pattern 116

00 - Introduzione

La realtà odierna dipende fortemente dall'utilizzo di software. Le infrastrutture nazionali e molti altri sistemi sono oggi controllati da sistemi basati su calcolatori e la maggior parte dei prodotti elettronici è controllata da software: basta, ad esempio, pensare ai sistemi di streaming e all'industria della musica. Risulta quindi essenziale l'**ingegneria del software** per il corretto sviluppo - e quindi funzionamento - di tali prodotti.

Definizione di ingegneria del software

> *L'**ingegneria del software** è una disciplina ingegneristica che si occupa di tutti gli aspetti della produzione di software, partendo dagli stadi iniziali di specifiche del sistema fino al mantenimento di quest'ultimo*.

All'interno di tale definizione ritroviamo due aspetti fondamentali:

1. *disciplina ingegneristica*: l'aspetto ingegneristico di tale disciplina risiede nell'applicazione delle teorie, dei metodi e degli strumenti in maniera appropriata in virtù del sistema da sviluppare e dei limiti presenti;
2. *ogni aspetto dello sviluppo software*: l'ingegneria del software non riguarda solamente i procedimenti tecnici dello sviluppo del software, ma anche attività quali la gestione del progetto software, lo sviluppo di metodi e teorie (nonché di strumenti nuovi) per supportare lo sviluppo del software finale.

La nozione di "ingegneria del software" venne proposta nel 1968 durante una conferenza per discutere quella che prese il nome di "crisi di software". Ci si rese conto che gli approcci individuali per lo sviluppo di programmi non potevano mai essere scalati per sistemi software grandi e complessi: risultavano infatti essere **inaffidabili**, **costosi** e **consegnati in ritardo**. Negli anni Settanta e Ottanta vennero sviluppate nuove tecniche di ingegneria del software, come la **programmazione strutturata**, l'**occultamento delle informazioni** e le tecniche di **programmazione orientata agli oggetti** con i suoi paradigmi.

Importanza e sfide dell'ingegneria del software

Sempre più persone e società fanno affidamento a sistemi software avanzati. Risulta necessario produrre software che sia **affidabile** e **attendibile** e anche in maniera più o meno **rapida**. Oltre a ciò si presentano altre tre sfide:

1. i requisiti del sistema da sviluppare possono **cambiare** inaspettatamente nel tempo;
2. il sistema software può essere un sistema **complesso**. Questo può essere affrontato utilizzando un certo grado di **astrazione** e la **modularità**;
3. **difetti software sono sempre inevitabili** (è impossibile sviluppare un software che sia infallibile).

Le tecniche di ingegneria del software permettono quindi di produrre software di **alta qualità** con **risorse temporali ed economiche limitate** mentre dei **cambiamenti** sono possibili.

Qualità del software

Per misurare la qualità del software sviluppato si utilizzano i seguenti criteri operativi:

1. **correttezza**: il sistema software è conforme alle specifiche richieste e raggiunge lo scopo. Ciò significa che vengono soddisfatti i seguenti punti:
   - **validazione funzionale**: il sistema software sviluppato fa quello che il cliente ha richiesto;
   - **correttezza tecnica**: il sistema software soddisfa le specifiche;
2. **efficienza, sicurezza, affidabilità** e **usabilità**.

Manutenibilità

Un sistema software è modificabile intrinsecamente in quanto non ha parti fisiche. Poiché tuttavia la realtà cambia, sarà necessario in futuro modificare il sistema software sviluppato per adattarlo alle nuove esigenze. La **manutenzione** del software è la più costosa, in quanto:

- le richieste di estensione del software di successo aumentano col crescere del successo;
- il sistema software "sopravvive" all'hardware per cui era stato sviluppato, generando una nuova esigenza per le nuove piattaforme che si sviluppano.

Riformulazione della definizione di ingegneria del software

In virtù delle osservazioni precedenti, possiamo riformulare parzialmente la definizione di ingegneria del software, aggiungendo che l'ingegneria del software soddisfa i bisogni del cliente applicando **processi quantificabili** e **sistematici** allo sviluppo e all'evoluzione dei prodotti o servizi software in virtù di **limiti** a livello di **risorse disponibili** o **requisiti imposti**.

Processo software

> *Un **processo software** è un metodo sistematico per raggiungere gli obiettivi di un progetto attraverso la definizione e l'organizzazione delle attività di sviluppo, la definizione dei ruoli e delle abilità necessarie per il compimento del progetto e l'impostazione dei criteri e dei tempi di fine per passare da un'attività alla successiva*.

In maniera molto semplice, un processo guida "chi farà cosa, quando e perché", ossia è un insieme di attività in relazione tra loro che porta alla produzione di un prodotto software. I processi sono spesso chiamati **metodi**, e classi di processi vengono chiamati **modelli**.
La definizione del processo dipende dai bisogni del cliente e dai limiti imposti. In virtù di tale processo si produrrà un prodotto che soddisferà i bisogni del cliente e che soddisfa i limiti imposti.

Ci sono molti processi software diversi tra loro, ma tutti **devono** includere le seguenti attività fondamentali:

1. **analisi e specificazioni software**: attività volta a definire lo scopo dello sviluppo software e i suoi limiti di sviluppo;
2. **sviluppo e design software**: produzione del codice del sistema software;
3. **validazione**: verifica del prodotto sviluppato in virtù dei bisogni del cliente;
4. **mantenimento ed evoluzione**: cambiamento del software in virtù delle nuove esigenze.

Ruolo di un ingegnere del software

Per un ingegnere del software non basta semplicemente essere un abile programmatore, ma è anche necessario:

- comprendere i requisiti e scrivere specifiche per il sistema da sviluppare;
- derivare modelli, motivando anche il perché del modello scelto;
- padroneggiare lo sviluppo di software;
- operare a diversi livelli di astrazione;
- essere capace di lavorare in team, quindi avere competenze nella gestione e nella comunicazione di squadra.

Risulta quindi essere una figura diversa rispetto al **computer scientist**:

- il **computer scientist** dimostra teoremi riguardante algoritmi, sviluppa linguaggi e si concentra sulla teoria. Ha inoltre risorse temporali anche infinite (non c'è una "scadenza" per lo studio di nuovi linguaggi e/o algoritmi);
- il **software engineer** ha risorse temporali limitate e deve sviluppare software che sia attendibile e affidabile, facendo fronte anche a possibili futuri cambiamenti del prodotto sviluppato.

01 - Java - Prima Lezione

Java è un linguaggio di programmazione strutturato che segue il paradigma della programmazione orientata ad oggetti. Java inoltre è **puramente ad oggetti**, pertanto ogni programma richiede la definizione di una classe.

Java non produce un eseguibile tramite un processo diretto di compilazione come in linguaggi quali C++, dove il pre-processore risolve le direttive, il compilatore traduce il sorgente in linguaggio assembly e un assembler produce uno o più file oggetto che tramite il linker vengono uniti in un unico eseguibile. Il sorgente di Java viene tradotto in un **bytecode** dal compilatore Java, che è un linguaggio assembler specifico per un processore virtuale, la **Java Virtual Machine** (**JVM**). Utilizzando la JVM si potrà poi tradurre il linguaggio nel linguaggio assembler per lo specifico calcolatore. Si tratta quindi di un processo di compilazione in un linguaggio bytecode che poi verrà tradotto in linguaggio macchina localmente. Questo permette di produrre un eseguibile che viene compilato da qualsiasi macchina a patto che la macchina sia dotata del processore virtuale JVM.
Ci sono 3 vantaggi principali in questo processo di compilazione:

- **portabilità**: il bytecode di Java permette al codice di essere compilato su qualsiasi piattaforma con una JVM compatibile;
- **sicurezza**: nel processo di compilazione si ha una verifica del bytecode che impedisce la compromissione del sistema da parte di un codice dannoso;
- **efficienza**: il bytecode viene effettivamente compilato in porzioni.

In Java inoltre esiste il **garbage collector**, che al termine dell'esecuzione del programma libera la memoria occupata dagli oggetti allocati durante l'esecuzione.

Processo di compilazione di un file Java

L'ambiente di esecuzione per i programmi Java è costituito da due blocchi principali:

- la **macchina virtuale Java** (in inglese **JVM, Java Virtual Machine**);
- le **API** Java.

La macchina virtuale è la base della piattaforma Java, mentre le API sono le librerie usate dal codice Java.
I file sorgenti Java vengono memorizzati come file `.java`. Il compilatore Java (`javac`) genera il bytecode, traducendo il codice sorgente in una rappresentazione intermedia indipendente dalla piattaforma, verificando che non ci siano errori sintattici. Una volta compilato, il bytecode verrà fornito alla JVM per l'esecuzione. Da notare che il bytecode presenta istruzioni utilizzando un linguaggio **astratto**, del tutto **indipendente dal sistema su cui verrà eseguita l'applicazione**.
Dopo la compilazione del bytecode, la JVM si occupa di **virtualizzare** l'hardware su cui viene eseguito il risultato della compilazione. La JVM è costituita da tre elementi:

1. il **class loader**;
2. il **bytecode verifier**;
3. l'**interprete**.

Il **class loader** carica le classi che formano il bytecode. Dopo che le classi vengono caricate, il **bytecode verifier** controlla che il bytecode sia valido, verificando che non ci siano operazioni che portano a stati illegali della JVM (ad esempio, operazioni che portano a **violazioni di memoria**, spesso verificando che non ci siano operazioni di aritmetica dei puntatori). Se il bytecode passa i controlli, allora può essere eseguito dall'interprete.
L'interprete può essere a sua volta di varie forme:

1. può essere un normale interprete, che esegue uno per uno ogni istruzione del bytecode;
2. può utilizzare un **JIT compiler**, ossia un **compilatore just-in-time**, che traduce il bytecode in codice macchina del calcolatore su cui viene eseguito il programma.

Esiste una soluzione mista, in cui la compilazione JIT avviene solo per le porzioni di codice più usate, mentre il resto viene interpretato.

Struttura del sorgente Java

Ogni programma Java deve contenere una sola classe. Ogni file Java deve avere un solo metodo `main`. Da notare che ogni software Java può contenere più `main` possibili, in quanto esso è definibile in ogni classe del software.
Il main viene dichiarato **sempre** come:

```
    public static void main(String[] args) {
    ...
    }
```

Il parametro `String[] args` indica i possibili argomenti che vengono passati nel momento dell'esecuzione del file. Il file sorgente viene memorizzato con estensione `.java`. Il sorgente viene compilato a riga di comando con:

```
    :~$ javac <nome_file>.java
```

Producendo un file con estensione `.class`. Il compilatore `javac` fa riferimento ad una variabile di sistema nota come `javaclasspath`, che indica dove sono localizzate le classi standard che possono essere importante nel codice. Possiamo aggiungere ulteriori `classpath` utilizzando l'opzione `-cp`:

```
    :~$ javac -cp <path> <nome_file>.java
```

Per eseguirlo basta utilizzare:

```
    :~$ java <nome_file>
```

Senza scrivere l'estensione.
Le librerie possono essere importate usando `import`, come segue:

```
import java.time.LocalDate
```

che importa da `time` il tipo `LocalDate`. Java organizza le sue classi in una struttura gerarchica organizzata in **folder**: questi folder prendono il nome di **package**. Lo snippet di codice soprastante quindi indica di utilizzare la classe `LocalDate` che si trova dentro il package `time` che si trova dentro il package `java`: poiché questi sono folder, è come se stessimo indicando un path al file `LocalDate.class`.
Per creare una classe all'interno di un nuovo package, è necessario specificare nel programma:

```
package <folder_name>
```

dove `<folder_name>` specifica la cartella in cui è definito il file. Per eseguire con `java`, dobbiamo andare nel folder principale ed eseguire:

```
    :~$ java <folder_name>.<nome_file>
```

Il folder principale nel nostro caso è la home stile UNIX `~/`. All'interno della home ci sarà un folder `<folder_name>`.
In Java inoltre **non sono presenti puntatori**.
Per stampare in java, si utilizza:

```
System.out.println("stringa esempio");
```

Per leggere invece input dall'utente, si utilizza la classe *Scanner*:

```
import java.util.Scanner

Scanner input = new Scanner(System.in);
```

Viene così istanziato un oggetto di tipo *Scanner* a cui viene passato lo stream da cui prendere l'input. Un programma esempio è il seguente:

```
import java.util.Scanner;

public class ScannerDemo {
    public static void main(String[] args) {
```

```
        Scanner input = new Scanner(System.in);

        System.out.println("Inserisci nome:" );
        String name = input.nextLine();

        System.out.println("Ciao " + name);
    }
}
```

Allocazione delle variabili in Java

Le variabili che non hanno un tipo primitivo sono tutte di tipo *oggetto*. Il nome della variabile in questo caso è un **riferimento all'oggetto** che punta all'oggetto allocato in memoria.
Di conseguenza il fatto che in Java non sono presenti puntatori fa riferimento alla impossibilità di utilizzare puntatori in maniera libera come in C++.
Ad esempio:

```
    String nome = "Paolo";
```

indica che `nome` è una variabile che è un riferimento ad un oggetto che punta all'area di memoria istanziata per l'oggetto.

Confronto tra oggetti

Come in C e C++, se vogliamo verificare che due oggetti sono uguali, ad esempio due stringhe, non possiamo semplicemente usare l'operando `==`, ma si confrontano i contenuti degli oggetti. In particolare, per due stringhe `s1, s2` si utilizza `s1.equals(s2)`.

Confronto con C++

Per fare un confronto con C++, vediamo adesso alcune differenze di sintassi:

C/C++	Java
`T a;` alloca una struct di tipo `T`	`T a;` alloca l'id `a` di tipo `T`
`a++` ; aumenta di uno `a` (può anche essere un puntatore)	`a++` ; illegale
`a->m();` accede al metodo `m()` del puntatore `a`	`a.m();` accede al metodo `m()` dell'oggetto `a`

Nella definizione delle classi in Java si pone sia l'astrazione che la sua implementazione. Questo è un obbligo che non è presente in C++: infatti si possono dichiarare i prototipi e poi definire le funzioni in un secondo momento.
I tipi ai lati di un assegnamento devono essere formalmente compatibili a tempo di compilazione. A tempo di esecuzione si verifica che i puntatori sono compatibili e che il tipo assegnato al puntatore sia compatibile al puntatore stesso.

Ereditarietà e interfacce

A differenza di C++, Java **non** supporta l'ereditarietà multipla. Questo vincolo permette di avere gerarchie lineari di classi. Risulta possibile "aggirare" questo vincolo utilizzando le **interfacce**.
Il problema dell'eredità multipla risiede nel fatto che concettualmente, se `A` e `B` sono due tipi distinti e rappresentabili come due insiemi disgiunti, dire che una classe `C` eredita sia da `A` sia da `B` è una contraddizione: sarebbe come definire un sottoinsieme comune ad `A` e a `B`. Definire un'interfaccia è simile a definire una classe astratta, ma ci sono delle differenze:

1. non sono presenti attributi non `final` nell'interfaccia;
2. è possibile implementare metodi all'interno dell'interfaccia, a differenza delle classi astratte, che non prevedono l'implementazione dei metodi dichiarati;

L'interfaccia, inoltre, **non** rappresenta un ADT, ma un **gruppo di funzionalità affini tra loro**, che è un concetto meno significativo di ciò che definisce un tipo di entità. Rappresenta quindi un modo d'uso, comune a più tipologie di entità anche non correlate tra loro. Ha senso ora concettualmente pensare che per un oggetto si possano implementare più interfacce.
Si può anche simulare l'ereditarietà multipla con il **pattern decorator** se esso è implementato correttamente.

Keyword `Static`

La keyword `static` permette di produrre un blocco di codice unico immediatamente disponibile per l'esecuzione. Si può quindi avere codice staticamente eseguibile **anche se non ci sono oggetti istanziati**.
Un metodo che viene dichiarato `static` diventa **unico** e **globale**, quindi non ci potranno essere altri metodi con lo stesso nome nell'applicazione. `static` pertanto permette di definire variabili e metodi globali.

Keyword `final`

La keyword `final` indica che la variabile non può essere riassegnata, quindi assume lo stesso significato di `const` in C++.

02 - Introduzione alla Object Oriented Programming

Caratteristiche della OOP

Ricordiamo le tre principali caratteristiche della **Object Oriented Programming**:

- **incapsulamento**, ossia la proprietà degli oggetti di **incorporare** al loro interno attributi e metodi;
- **ereditarietà**, ossia la capacità di costruire nuove classi a partire da classi già sviluppate;
- **polimorfismo**, ossia la possibilità per i metodi di assumere implementazioni diverse utilizzando tuttavia **lo stesso identificatore**.

A livello sintattico viene aggiunto solamente un solo costrutto, ossia il costrutto `class`.
La OOP risolve problemi quali la **namespace pollution** e la **non-rientranza**:

- la prima fa riferimento all'impossibilità di poter utilizzare un nome in quanto lo stesso simbolo è già utilizzato nello stesso namespace (o header file, ad esempio, nel caso di un linguaggio strutturato come C);
- la seconda fa riferimento all'impossibilità di invocare in maniera sicura una funzione prima che una sua precedente invocazione è stata completata (id est può essere eseguita in maniera concorrente). Questo accade ad esempio quando una funzione lavora su variabili che sono globali e quindi esterne alla funzione.

Ciò viene raggiunto tramite l'aggiunta di gradi di **astrazione**. Nel paradigma di programmazione strutturata o procedurale possiamo raggiungere ciò definendo degli **Abstract Data Type** (**ADT**). L'aggiunta di gradi di astrazione permette di poter **riutilizzare il codice**. Lo stesso principio vale nella programmazione a oggetti. La programmazione strutturata tuttavia non permette un livello sufficiente di astrazione:

1. non abbiamo alcuna garanzia di inizializzazione e di terminazione;
2. non si può generalizzare la gestione degli errori;
3. il compilatore (o interprete) non **forza** l'occultamento dei dati.

Ad esempio, nel linguaggio C possiamo accedere direttamente ai dati contenuti all'interno di una struct in maniera del tutto legale (ossia senza che il compilatore lo impedisca). Quindi è concreta la possibilità di scrivere codice che può generare errore con più facilità.
La soluzione a questi problemi risiede nell'utilizzo delle **classi**, un tipo di dato astratto che permette di incapsulare e nascondere informazioni. Ogni classe contiene un **costruttore** e un **distruttore** per inizializzare e de-allocare gli oggetti in maniera automatica (in C++ dipende se l'oggetto è allocato dinamicamente. La chiamata al distruttore tuttavia è automatica quando viene usato `delete`).

Gestione delle eccezioni

Un'**eccezione** è un nome che diamo ad una situazione di errore relativo ad un concetto. Non si tratta di un errore di sistema.
A livello implementativo un'eccezione non è altro che una **classe** che gestisce una problematica. Tale problematica è relativa ad un ADT e non sono **generiche**, ma si ripresentano nel contesto dell'ADT.
L'eccezione permette di definire l'errore in fase di progettazione e la gestione di questo nell'implementazione del client. Quindi permette di disaccoppiare l'identificazione dell'errore dalla sua gestione.
In C++ è possibile provocare un'eccezione con `throw`:

```
//...
void foo() {
    if ({condition}) {
        throw {exception}
    }
}
//...
```

In questo modo possiamo gestire i casi sia comuni sia particolari senza scrivere blocchi di codice che **prevedono** le eccezioni durante le operazioni:

```
try {
    //...
}
```

```
catch (Exception Ex) {
    // gestisci eccezione
}
```

Il sollevamento delle eccezioni in Java va specificato nella firma della funzione:

```
public void pop(T item) throws StackUnderflowException {
    if (this.is_empty()) throw StackUnderflowException(this);
    //...
}
```

03 - Terza lezione

Riusabilità tramite parametrizzazione

Risulta possibile riutilizzare un codice tramite la **parametrizzazione** delle variabili. Questo è possibile attraverso il meccanismo di **binding dinamico**, che permette a tempo di esecuzione di risolvere il tipo della variabile.
In C++ è possibile fare ciò con i template:

```
template <class T> class Stack {
    public:
        Stack(size_t size);
        ~Stack();
        void push(const T &item);
        void pop(T &item);
        void top();
        bool isEmpty();
        bool isFull();
    private:
        size_t top;
        size_t size;
        T *_stack_;
};
```

A dover specificare il tipo `T` del modulo software riutilizzabile è il programmatore. Questo permette al software di poter avere più significati all'interno dell'applicazione
In Java l'allocazione in memoria di variabili di tipo `T` viene eseguita con un cast a `T` di variabili di tipo `Object`, in quanto ogni variabile allocata con `new` è internamente un puntatore ad un'area di memoria, che sarà quindi correttamente allocata.

Glossario OOP

> *Un **ADT** è una specifica di proprietà, una regola a cui oggetti concreti potrebbero obbedire, e permette l'astrazione di molte entità del mondo reale.*
> *Un **oggetto** è un **modello** di un'unica entità della vita reale e un'istanza è la sua corrispondenza funzionante nel programma. Ogni istanza è unica e distinguibile dalle altre tramite la sua "identità".*
> *Un **modulo** è una collezione **unica** di programmi, usabile per realizzare un'istanza di un oggetto.*
> *Un'**istanza** è un programma allocato in memoria pronto per l'esecuzione. Sono anche chiamati **oggetti** in virtù del fatto che nel contesto del sistema software essi sono le rappresentazioni di questi.*
> *Una **classe** è un **artefatto sintattico** le cui semantiche possono essere un ADT, un oggetto, un modulo o una combinazione di queste. In Java, una classe è l'implementazione di un template per la creazione di istanze di oggetti.*
> *Un **attributo di classe** è un identificatore statico allocato a tempo di compilazione memorizzato in una classe e condiviso tra tutte le istanze. L'istanza dell'oggetto a cui è riferito - che dev'essere statico - viene anch'esso condiviso.*
> *Un **attributo di istanza** è un identificatore dichiarato in una classe ma allocato a tempo di esecuzione nelle istanze di classe come copie separate.*
> *Un **metodo di classe** è un metodo globalmente disponibile per l'esecuzione, e viene chiamato dalla classe che lo definisce.*
> *Un **metodo di istanza** è un metodo di un oggetto che viene chiamato solo dall'istanza della classe che lo definisce.*
> *Un **call dispatch** è l'azione che consiste nel determinare quale codice eseguire in base al messaggio inviato ad un identificatore. Può essere fatto staticamente (**early binding**) con tipi statici associati, o dinamicamente (**late binding**) a tempo di esecuzione.*
> ***Self / this** sono alias utilizzati per fare riferimento all'istanza dell'oggetto che chiama il metodo.*
> ***Super** è un qualificatore di dispatch, **non** un identificatore. Indica di cercare un metodo corrispondente dalla classe genitore dell'oggetto considerato.*

Con "messaggio inviato ad un identificatore" e in generale con "scambio di messaggi" si intende la possibilità da parte degli oggetti di chiamare i metodi pubblici di altri oggetti.
Il call dispatch allora permette di risolvere la richiesta di esecuzione al metodo giusto. Ciò accade a tempo di esecuzione nel contesto del **polimorfismo**.

Ereditarietà e polimorfismo

L'**ereditarietà** è il meccanismo utilizzato in fase di strutturazione del software che permette di derivare nuove classi a partire da quelle già definite realizzando una gerarchia di classi. L'ereditarietà è particolarmente potente, in quanto:

- permette di **riutilizzare il codice**, definendo quindi nuovi **sottotipi**;
- permette la scrittura di **codice generico**, tramite il meccanismo del **polimorfismo**.

Il **polimorfismo** è la proprietà della programmazione orientata agli oggetti in base alla quale oggetti di classi diverse possono rispondere in maniera diversa a una stessa invocazione. Supponiamo di avere due oggetti, uno di tipo `Cerchio` e l'altro di tipo `Quadrato`. Entrambe sono istanze di due classi che provengono dalla stessa classe `Forma`. Tramite il polimorfismo, il programma che scriviamo non deve obbligatoriamente conoscere il tipo di ogni forma, e può semplicemente chiamare la funzione `draw()` per entrambi i tipi: tramite il meccanismo di **late binding** il call dispatch permetterà di risolvere l'associazione codice-messaggio (intendendo "messaggio inviato all'identificatore" come spiegato precedentemente).

04 - Utilizzo corretto della OOP

Utilizzo corretto della OOP: esempio pratico

Requisiti

Consideriamo un componente piccolo che dati vari file contenenti valori numerici, con un valore per ciascuna riga del file:

1. leggere da ciascun file la lista di valori;
2. tenere solo i valori non duplicati;
3. calcolare la somma dei numeri letti dal file (non duplicati);
4. calcolare il massimo tra i numeri letti.

Quando il software sarà pronto, saranno questi requisiti che dovranno essere soddisfatti dal risultato della programmazione.

Sviluppo

Per prima cosa avremo bisogno di importare alcune librerie. In particolare, importeremo:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.ArrayList;
import java.util.List;
```

Importiamo:

- la classe `File`, che permette di manipolare file o in generale stream di input e output (?);
- la classe `FileReader`, che permette di fornire operazioni di lettura più potenti per file;
- la classe `IOException`, che permette di gestire le eccezioni di lettura e scrittura (ad esempio, quando un file su disco non esiste o non è pronto per la lettura/scrittura);
- la classe `LineNumberReader`, un altro reader per la lettura della riga del file;
- la classe `ArrayList` e la classe `List`.

L'idea di importare queste classi è quella di creare strutture più avanzate a partire da classi più semplici. A partire dalla classe `File`, si può istanziare la classe `FileReader`. A partire da `FileReader`, si può istanziare la classe `LineNumberReader`.
Abbiamo utilizzato il termine "classe", ma a dir la verità abbiamo utilizzato dei nomi, che potrebbero indicare sia classi sia interfacce. Infatti, `ArrayList` è una **classe**, ma `List` è una **interfaccia**.
Così come esiste il `FileReader`, esiste un `FileWriter`, istanziabile come segue:

```
    FileWriter F = new FileWriter("test.txt", true);
```

La variabile booleana indica le modalità di append o overwrite:

- se la booleana è `true`, siamo in modalità append;
- se la modalità è `false`, il testo verrà sovrascritto.

Collezione

L'astrazione principale per gli aggregati di oggetti è la **collezione**, un concetto che indica un aggregato generico di elementi che permette la navigazione di questi. In Java è stato introdotto il concetto di collection come astrazione di un generico aggregato. Nel nostro caso, tale collezione è una **lista**, un aggregato sequenziale di elementi. Gli elementi non sono obbligatoriamente ordinati in maniera sequenziale tra loro.
Nel nostro caso, la lista è un array di oggetti di tipo `T`. Nel momento in cui `ArrayList` è vuoto, i valori degli oggetti di tipo `T` sono nulli: gli elementi saranno aggiunti o tolti uno dopo l'altro. A livello astratto essa è una sequenza di elementi che può essere anche implementata con una lista concatenata.

Definizione della classe

Definiamo ora la classe `CalcolaImporti`:

```
public class CalcolaImporti{
    private final List<String> importi = new ArrayList<>();
```

La lista è una struttura dati sequenziale e omogenea generica. `ArrayList` permette di implementare la lista attraverso un array.
Dichiariamo ora la variabile `totale` e la funzione `calcola`:

```
    //...
    private float totale;
    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));

    }
```

`LineNumberReader` è una classe che è costruita sulla classe più semplice `FileReader`, a sua volta istanziata sulla base della classe più semplice `File`, che presenta due parametri: il nome del file e il prefisso, ossia il percorso per tale file. Nel caso in cui il file si trovi nella stessa directory, allora basta scrivere `.`.

Composizione e derivazione

Il `FileReader` è stato costruito sulla base della classe `File`. Più precisamente, possiamo dire che l'oggetto di tipo `FileReader` è stato costruito con una **composizione** con la classe `File`. Ossia, non abbiamo detto che `FileReader` eredita da `File`, ma abbiamo istanziato `File` e `FileReader` **ha** un `File`, ma **non è** un `File`. La composizione di conseguenza è un **legame dinamico**, mentre l'ereditarietà è un **legame statico** in quanto è un **legame di tipo** che definisce la struttura del software, immutabile durante l'esecuzione. Lo stesso può essere detto per il `LineNumberReader`: `LineNumberReader` aggiunge una funzionalità a `FileReader`, ossia la possibilità di calcolare il numero della riga letta.
La composizione permette di creare funzionalità più avanzate senza derivare caratteristiche da una classe base.

Ciclo di lettura

Continuiamo con:

```
    //...
    totale = 0;
    while (true) {
        final String riga = f.readLine();
        if (riga == null) break;
        importi.add(riga);
        totale += Float.parseFloat(riga);
    }
    f.close();
    return totale;
```

`add()` è un metodo dell'interfaccia `List`.
`parseFloat()` è un metodo della classe `Float` (quindi `static`).
`f.close()` è necessario per chiudere il file. Il bytecode verifier **non** notificherà che `f.close()` non è stato invocato.

Precisazione su `final`

Porre `final` su una classe indica che **da quella classe non si possono derivare altre classi**. Se un metodo è posto come `final`, una sottoclasse **non può eseguire un override del metodo**. Nel nostro contesto, `final String riga` indica semplicemente che `riga` è costante.

Precisazioni sulla tipologia dello sviluppo dell'esempio

Quello che abbiamo fatto per ora è una bozza del nostro software utilizzando un approccio **incrementale**, ossia aggiungendo di volta in volta funzionalità. Ora come ora manca:

1. l'implementazione del massimo;
2. l'eliminazione dei duplicati.

Codice finale

Aggiungiamo le implementazioni finali e si ottiene il codice:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.ArrayList;
import java.util.List;

public class CalcolaImporti{
    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;
    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        totale = massimo = 0;
        while(true) {
            final String riga = f.readLine();
            if (riga == null) break;
            if(!importi.contains(riga)) {
                importi.add(riga);
                float x = Float.parseFloat(riga);
                totale += x;
                if (massimo < x) massimo = x;
            }
        }
        f.close();
        return totale;
    }
}
```

L'algoritmo tuttavia **non è conforme ai principi Object-Oriented**. Infatti vengono eseguite nello stesso algoritmo eseguo:

- lettura del file;
- inserimento delle righe lette;
- calcolo della somma;
- calcolo del massimo;
- restituzione della somma.

Questo modo di procedere prende il nome di **anti-pattern**, perché è un approccio da evitare (o spaghetti code). Ci sono alcune indicazioni che il nostro codice non segue un corretto pattern:

- i metodi sono **lunghi**, senza parametri e usano variabili globali;
- viene ignorato il principio di **singola responsabilità**, ossia il metodo svolge più compiti assieme. Una funzionalità dev'essere la più piccola possibile;
- il flusso di esecuzione è determinato solamente dall'implementazione dell'oggetto. Ossia, l'ordine di esecuzione è **cablato** in un'implementazione monolitica dell'oggetto;
- ereditarietà e polimorfismo non possono essere utilizzate. Il metodo vecchio, infatti, dev'essere riutilizzato in blocco, o se dev'essere ridefinito, va ridefinito l'intero blocco;
- il nome stesso della classe tradisce una cattiva implementazione dell'ADT. Nel nostro caso infatti coincide con la funzionalità che abbiamo dentro. Non abbiamo definito un ADT, ma un algoritmo monolitico racchiuso dentro una classe.

Cerchiamo allora di dividere i compiti:

```
import java.io.File;
import java.io.FileReader;
import java.io.IOException;
import java.io.LineNumberReader;
import java.util.ArrayList;
import java.util.List;

public class CalcolaImporti{
```

```
    private final List<String> importi = new ArrayList<>();
    private float totale, massimo;
    public float calcola(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while(true) {
            riga = f.readLine();
            if (riga == null) break;
            if(!importi.contains(riga)) importi.add(riga);
        }
        f.close();

        totale = 0;
        for(int i = 0; i < importi.size(), i++) {
            totale += Float.parseFloat(importi.get(i));
        }

        massimo = Float.parseFloat(importi.get(0));
        for (int i = 1; i < importi.size(); i++) {
            if (massimo < Float.parseFloat(importi.get(i)))
                massimo = Float.parseFloat(importi.get(i));
        }
        return totale;
    }
}
```

In questo modo abbiamo separato i compiti. Possiamo adesso separare il calcolo di queste funzioni e definire un ADT `Pagamenti`:

```
public class Pagamenti {
    private List<String> importi = new ArrayList<>();
    private float totale, massimo;

    public void leggiFile(String c, String n) throws IOException {
        LineNumberReader f = new LineNumberReader(new FileReader(new File(c, n)));
        String riga;
        while(true) {
            riga = f.readLine();
            if (riga == null) break;
            inserisci(riga);
        }
        f.close();
    }

    public void inserisci(String riga) {
        if(!importi.contains(riga)) importi.add(riga);
    }

    public void calcolaSomma() {
        for (String v : importi)
            totale += Float.parseFloat(v);
    }

    public void calcolaMassimo() {
        for(String v : importi)
            if (massimo < Float.parseFloat(v))
                massimo = Float.parseFloat(v);
    }

    public void svuota() {
        importi = new ArrayList<>();
        totale = massimo = 0;
    }

    public float getMassimo() { return massimo; }
```

```
    public float getSomma() { return totale; }
}
```

La classe risulta essere più riutilizzabile, sebbene sia più costosa in quanto il calcolo della somma, del massimo e della lettura del file sono tutte chiamate a funzione separate.
La scrittura `for (String v : importi)` è un **enhanced** `for` che permette di navigare gli elementi presenti in un aggregato generico. Questo tipo di ciclo `for` è valido per ogni oggetto `Iterable` ed è pensato per risolvere il problema dell'iterazione sugli elementi.
Le metriche per misurare un software generalmente sono:

- numero di classi;
- numero di metodi;
- numero di righe di codice (LOC, Lines Of Code).

Tali metriche sono utili per verificare se sono stati applicati correttamente i principi della OOP.

Command and Query

In questo caso stiamo utilizzando un paradigma noto come paradigma **Command and Query**: ogni metodo modifica lo stato dell'oggetto e non ritorna nessun valore (command) oppure non modifica lo stato dell'oggetto ma restituisce un valore interrogando l'oggetto per vedere qual è il suo stato (query).

Convalida (Testing)

Per la convalida basta dichiarare alcune funzioni di testing, come `testLeggiFile()`, `testCalcolaMassimo()` e `testCalcolaSomma()`, e richiamarle nel metodo `main()`.
Importante è anche avere un metodo che porta il mio sistema in uno stato necessario per il corretto funzionamento del nostro programma.
Per fare testing è importante anche avere un **oracolo**, ossia un qualcosa che permetta di verificare che l'output sia corretto. Tale oracolo è derivabile manualmente dai requisiti.
Ci sono strumenti più avanzati per eseguire testing del software prodotto.

05 - Design Patterns

Principi di design

Nella fase di progettazione del nostro software, ogni entità software identifica un'entità concreta. Gli ADT che definiamo devono avere un senso in virtù della realtà che modelliamo.

Principio della singola responsabilità

L'identificazione di un'entità dev'essere tale da delineare un insieme ben focalizzato di oggetti. Il **principio di singola responsabilità** afferma quindi che:

> *Ogni metodo deve svolgere una singola attività in relazione al suo ADT, e ogni classe deve rappresentare un singolo concetto.*

Principio di programmazione riferita a interfacce

> *Bisogna programmare riferendosi ad **interfacce**, non ad **implementazioni**.*

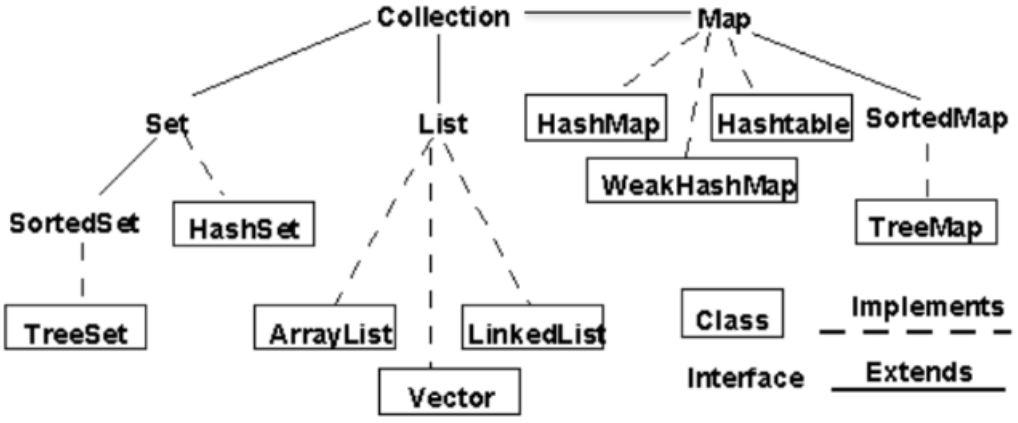
Ad esempio, supponendo nel nostro software di utilizzare un algoritmo implementato in una classe, che a sua volta implementa funzionalità tramite interfaccia o classe astratta, conviene far riferimento all'interfaccia o classe astratta. In questo modo se in un secondo momento si introduce un secondo modulo software che utilizza la stessa interfaccia, non vi è la necessità di riscrivere il modulo software del client.
A livello pratico, gli oggetti definiti nel modulo client i tipi associati agli identificatori sono i tipi delle **interfacce** o delle **classi astratte**.

Librerie di Java

Il principio esposto precedentemente è stato ampiamente utilizzato per i package java. Un esempio sono le **collection** di Java. La libreria Java introduce l'interfaccia `Collection`, un'interfaccia generica per **aggregati generici**. Ad esempio:

```
Collection mySet = new <type>;
```

identifica `mySet` come un oggetto che può essere di qualsiasi tipo di collection. Chiaramente `<type>` dev'essere una **implementazione** dell'interfaccia, come un `HashMap`, `List`, `Vector` e così via.
L'interfaccia `Collection` viene **estesa** con le definizioni delle interfacce `List`, `Set` e `Map`. Si crea così un albero le cui foglie sono le implementazioni delle interfacce:



Favorire la composizione all'ereditarietà

Importante è anche il seguente principio:

> *Bisogna sempre favorire la composizione all'ereditarietà.*

Abbiamo già visto che l'ereditarietà consiste nella definizione di un tipo `B` derivato da un tipo `A` con un legame di tipo, mentre la composizione prevede nella definizione del tipo `B` un oggetto di tipo `A`. Questo legame **non** è un legame di tipo, permettendo l'utilizzo delle funzionalità del tipo `A` senza che vi sia un legame concettuale. Questo legame è più dinamico, permettendo un approccio più flessibile: si può decidere quale oggetto è usato da un altro oggetto in maniera dinamica durante l'esecuzione, invece

di deciderlo in fase di design. Il comportamento quindi cambia durante l'esecuzione.
Tale approccio favorisce anche la **coesione** e l'abbassamento degli **accoppiamenti**.

Coesione

Nel momento della progettazione di una classe, possiamo facilmente individuare un metodo candidato ad essere **fuori posto** in virtù degli attributi che caratterizzano gli oggetti della classe da definire. I metodi devono lavorare sugli attributi della classe. Più i metodi lavorano sugli stessi attributi della classe, più sono legati tra loro.
Ad esempio, una classe `Calcolatrice` potrebbe permettere di memorizzare un valore salvato nella calcolatrice e calcolarne il quadrato. Il metodo `salva()` e il metodo `calcolaQuadrato()` sono legati tra loro.
La **coesione** esprime quanto i metodi della classe siano dipendenti tra loro (**coesi**). Le dipendenze derivano dall'usare gli stessi attributi.

Accoppiamento

Se la coesione è una misura interna relativa ai metodi della classe, l'accoppiamento è una misura definita in virtù delle relazioni tra un modulo software e altri moduli software (ad esempio, come si divide il sistema in sottosistemi). Vogliamo che il nostro software presenta **meno dipendenze possibili**.
Si può parlare di accoppiamento anche nello stesso modulo software, ed in tal caso si individuano le dipendenze tra le classi che compongono il nostro software. La composizione, in particolare, favorisce un accoppiamento **dinamico**.

Principio dei tipi parametrizzati

I tipi generici permettono la composizione dinamica all'interno del nostro software. Permette di ottenere maggiore flessibilità, creando composizioni a tempo di esecuzione.

Design Pattern

Un **design pattern** fa riferimento all'organizzazione del software che viene scelta per risolvere problemi che nascono dal modo in cui dobbiamo far interagire due classi tra di loro. Ogni pattern descrive un problema che si presenta nel nostro ambiente e che descrive la soluzione del problema in modo tale da riutilizzare la soluzione proposta per problemi dello stesso tipo.
Ogni design pattern è costituito da:

1. nome del pattern;
2. problema che descrive quando applicare il pattern;
3. la soluzione utilizzata che descrive gli elementi che costituiscono il design, le relazioni, le responsabilità e le collaborazioni tra gli elementi del design;
4. le conseguenze, che descrivono i trade-off nell'utilizzo del pattern.

Ogni pattern risolve un singolo problema. Nello stesso contesto è anche possibile usare più pattern insieme, aumentando i benefici.

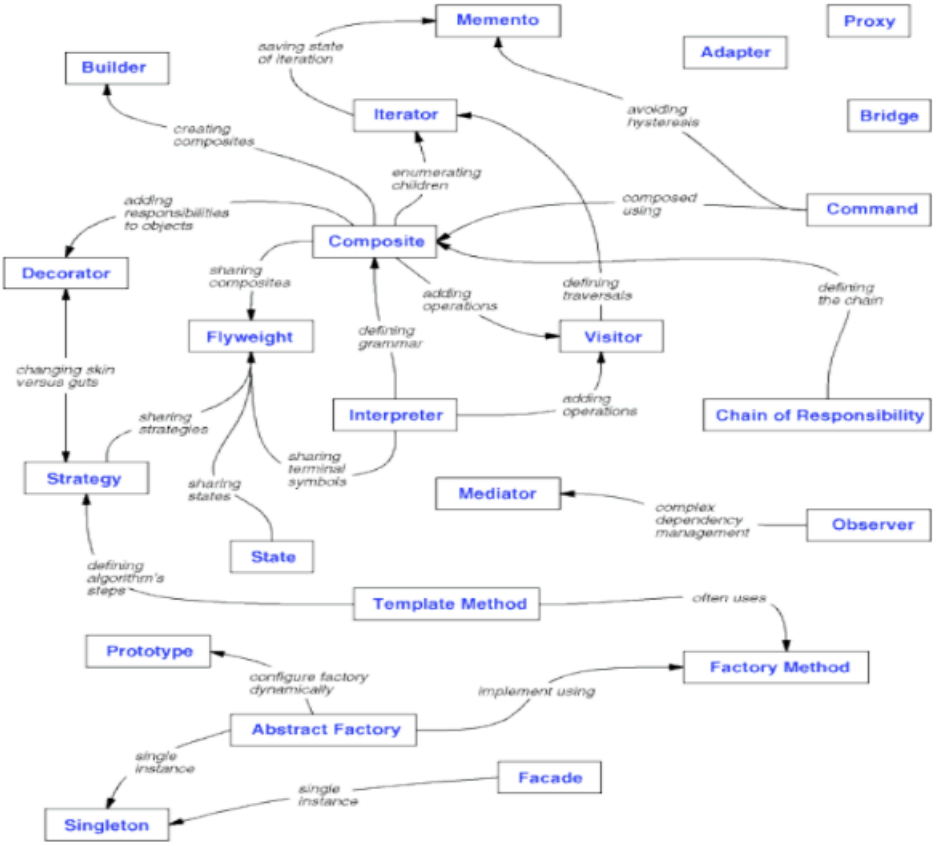
Catalogo dei pattern Gang of Four (GoF)

Il catalogo **GoF** è un catalogo che elenca in base all'ambito e allo scopo di un problema le soluzioni proposte. Più precisamente, è un insieme di 23 design patterns introdotti nel libro *Design Patterns: Elements of Reusable Object-Oriented Software*, scritto dagli autori Erich Gamma, Richard Helm, Ralph Johnson e Joh Vlissides (da cui appunto **Gang of Four**, che indica i quattro autori). I pattern sono classificati in tre categorie:

- **creational patterns**, pattern orientati al processo di creazione di oggetti fornendo maggiore flessibilità per il programma;
- **structural patterns**, riguardanti la composizione tra classi. Utilizzano l'ereditarietà per comporre interfacce e descrivono i modi per comporre gli oggetti e ottenere nuove funzionalità;
- **behavioral patterns**, riguardanti la comunicazione tra oggetti.

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory(87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

I pattern sono anche in stretta relazione tra loro, come mostra il grafo sottostante:



Pattern Iteratore

Il problema ricorrente è la navigazione di una struttura dati possibilmente eterogenea. Il problema non è immediatamente percepibile: se una struttura dati è omogenea, è molto semplice scorrere gli elementi.
Per poter navigare un aggregato (possibilmente eterogeneo), dobbiamo essere in grado di astrarre il tipo di elemento che stiamo navigando. Dobbiamo allora immaginare la navigazione come una **funzionalità astratta** e **separata** dall'aggregato.
Abbiamo allora un aggregato e un iteratore astratti (`Aggregate` e `Iterator`), e le classi `ConcreteAggregate` e `ConcreteIterator` che sono implementazioni delle definizioni astratte. All'interno dell'astrazione `Aggregate` abbiamo un metodo astratto `createIterator()` , mentre in `Iterator` avremo metodi quali `first()` , `next()` , `isDone()` , `currentItem()` . Ad esempio, `ConcreteAggregate` potrebbe essere uno stack, mentre il `ConcreteIterator` uno `StackIterator` . La navigazione dello stack diventa un'**entità separata dallo stack**. A livello implementativo lo `StackIterator` sarà realizzato con una **composizione** col tipo `Stack` . Notiamo tuttavia che queste implementazioni sono dipendenti tra loro, **ma non le loro astrazioni**: il nostro client farà riferimento all'astrazione, permettendo **riusabilità**. Un esempio pratico è il ciclo enhanced `for` , che utilizza implicitamente un iteratore.
Il vantaggio nell'utilizzo dell'iteratore inoltre è quello di permettere **attraversamenti multipli** di aggregati di oggetti. Ogni attraversamento infatti è un'istanza di `ConcreteIterator` , pertanto possiamo avere attraversamenti multipli utilizzando diversi

iteratori. Inoltre, permette di avere un'interfaccia uniforme anche per aggregati che sono diversi tra loro (stack, liste, ecc.).
Possiamo anche avere un'astrazione di un caso particolare, ossia il caso in cui abbiamo un **aggregato vuoto**. Tale astrazione è l'**iteratore nullo**, che semplicemente nel metodo `isDone()` restituisce automaticamente `true`.

Esempio pratico: Framework Collections

Il framework collections è un framework di Java che comprende un insieme di **iterabili** e i relativi iteratori. Si tratta di un'architettura non definita per rappresentare e manipolare collezioni in maniera indipendente dall'implementazione.
All'interno del framework si trova anche la definizione di `ArrayList` e `Vector`. Entrambe sono implementazioni dell'interfaccia `List`, ma differiscono per alcuni aspetti:

1. `ArrayList` non è sincronizzata, mentre `Vector` sì. Sincronizzato significa che in un ambiente multithread, gli altri thread aspettano che il thread attuale rilasci il lock sull'oggetto. Di conseguenza, `ArrayList` è più veloce di `Vector`;
2. `ArrayList` incrementa del 50% la dimensione dell'array quando vi è un overflow, mentre `Vector` la incrementa del 100%;
3. `ArrayList` utilizza l'iteratore per attraversare la struttura, ma il vettore può anche utilizzare `Enumeration` come interfaccia.

L'iteratore di `ArrayList` ci permette di scorrere gli elementi presenti all'interno della struttura astraendo la rappresentazione della collezione. Potremmo infatti avere i seguenti codici:

```
// senza iterator
void print(ArrayList list) {
    for(int k = 0; k < list.size(); k++) {
        System.out.println(list.get(k));
    }
}

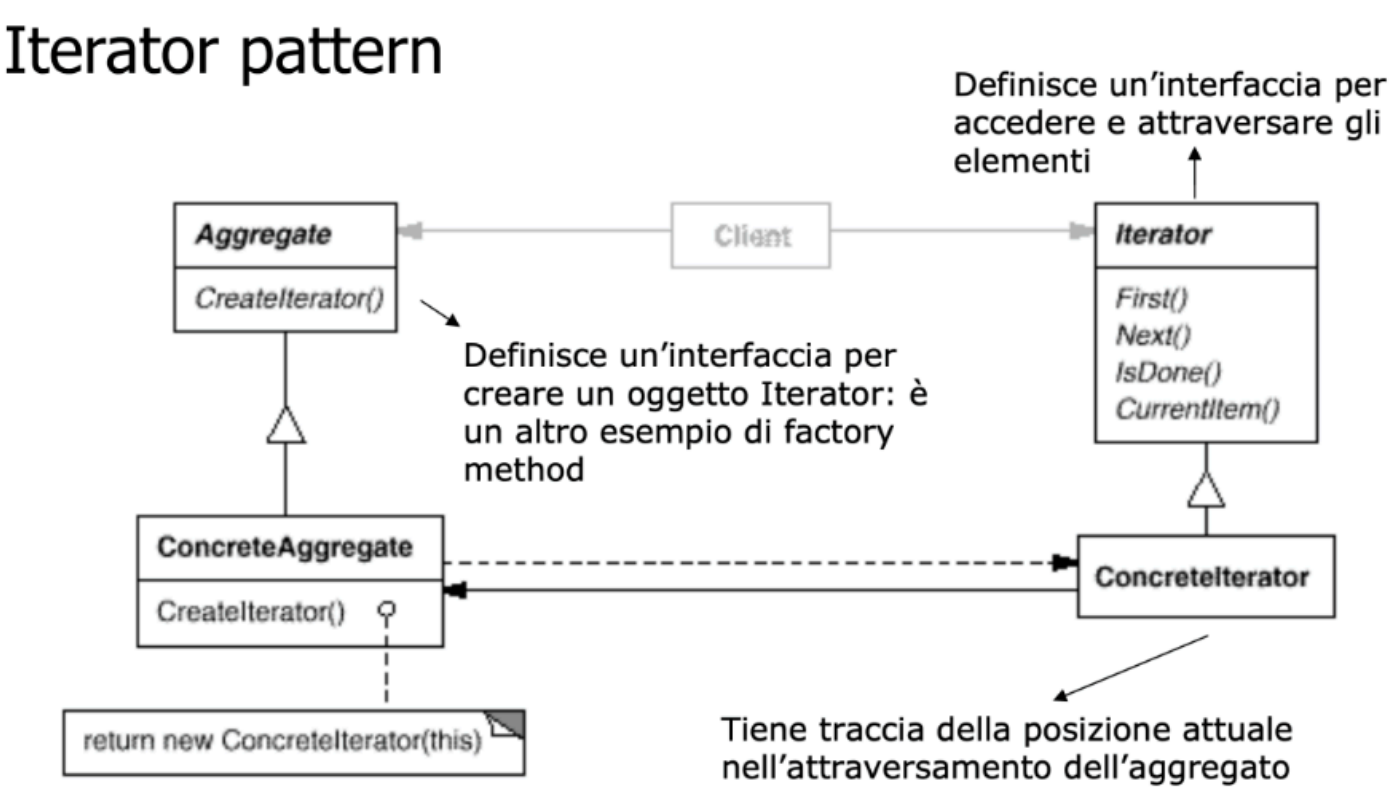
void print(LinkedList list) {
    Node current = list.first();
    while(current.hasNext()) {
        System.out.println(current);
        current = current.next();
    }
}

// con iterator
void print(Collection list) {
    Iterator items = list.iterator();
    while(items.hasNext()) {
        System.out.println(items.next());
    }
}
```

Chiaramente l'iteratore ci permette, **a prescindere dal tipo di collezione implementato**, di iterare sulla collezione.

Schema del pattern

Intento: Attraversare gli elementi di un aggregato senza esporre la sua struttura, nascondendo al client i dettagli della classe visitata.
Idea: togliere le funzionalità di accesso e attraversamento dall'aggregato e inserirle in un nuovo oggetto `Iterator`.
Risulta possibile eseguire molteplici scorrimenti in parallelo utilizzando più iteratori.
Lo schema del pattern è il seguente:



Astrazione stream

Ogni oggetto iterabile si può astrarre come `Stream`, un'interfaccia ispirata alla programmazione funzionale che incorpora al suo interno il concetto di aggregato astratto. L'aggregato può essere visto come stream per la sua navigazione.
L'interfaccia `Stream`, insieme all'interfaccia `Iterable`, presentano un metodo `forEach()` per iterare su tutti gli elementi presenti nella collection. Supponendo di avere:

```
List listOfStrings = new Vector<String>();
```

Possiamo iterare sugli elementi con **iteratore implicito** (enhanced `for`):

```
for(String s : listOfStrings) System.out.println(s);
```

oppure possiamo usare il metodo `forEach()` dell'interfaccia `Iterable`:

```
listOfStrings.forEach(System.out::println);
```

o ancora, `forEach()` dell'interfaccia `Stream`:

```
listOfStrings.stream().forEach((c) -> System.out.println(c));
```

Con gli stream abbiamo la possibilità di scrivere cicli sotto forma di singola espressione, non per rappresentare operazioni sequenziali, ma per rappresentare operazioni su aggregati. Possiamo così eseguire **operazioni parallele** sull'aggregato.

06 - Null Object

Supponiamo di voler implementare un albero binario di ricerca. Allora potremmo scrivere:

```
public class BinaryNode {
    int key;
    Node left = null;
    Node right = null;

    public boolean includes(int value) {
        if (value == key) return true;
        else if (value < key && left == null) return false;
        else if (value < key) return left.includes(value);
        else if (value > key && right == null) return false;
        else return right.includes(value);
    }
}
```

Il problema di questo codice, che risulta più difficilmente leggibile, risiede nel fatto che **controlliamo più volte il caso del valore nullo**. Tale problema può essere risolto utilizzando il pattern **Null Object**.
Il **Null Object pattern** consiste nell'implementazione di un **Null Object**. Ad esempio:

```
if (customer == null) plan = new BasicBillingPlan();
else plan = customer.getPlan();
```

Diventa:

```
plan = customer.getPlan();
```

dove in una classe `NullCustomer` si ha che la funzione `getPlan()` viene restituito `new BasicBillingPlan()`:

```
Class NullCustomer {
    getPlan() { return new BasicBillingPlan(); }
}
```

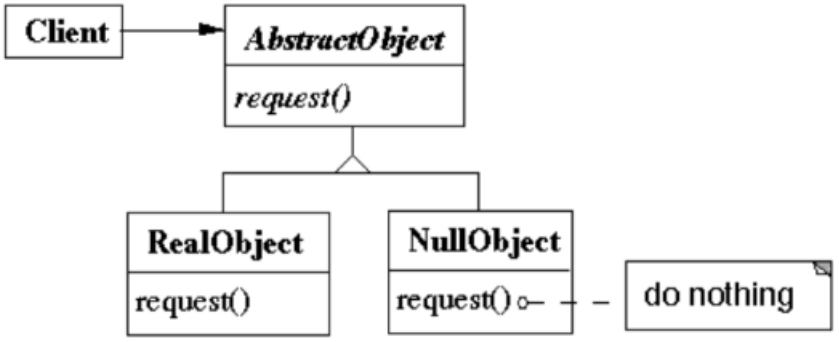
In questo modo la funzione `getPlan()` può essere invocata anche se `customer` è un oggetto nullo.
In Java gli identificatori senza assegnazione ricevono automaticamente valore nullo. Invocazioni di metodi a partire da identificatori nulli causano eccezioni. Di conseguenza, invece di lasciare che Java assegni `null` a identificatori in maniera automatica, assegniamo un'istanza di un `NullObject` che definiamo.
In questo modo il nostro codice diventa:

```
public class BinaryNode extends Node{
    int key;
    Node left = new NullNode();
    Node right = new NullNode();

    public boolean includes(int value) {
        if (value == key) return true;
        else if (value < key) return left.includes(value);
        else if (value > key) return right.includes(value);
    }
}

public class NullNode extends Node {
    public boolean includes(int value) { return false; }
}
```

Lo schema del pattern è il seguente:

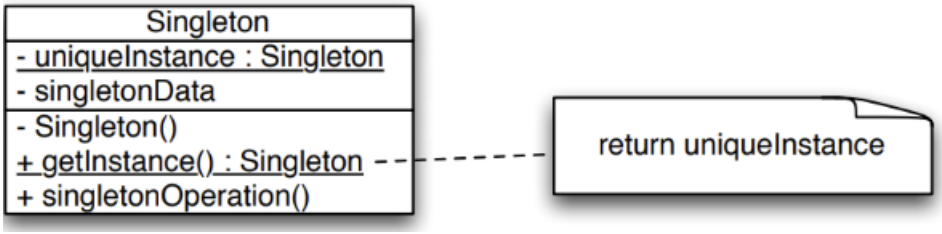


L'applicazione del pattern implica la definizione di una nuova versione della funzionalità `request()`, una versione "nulla". Oggetti nulli sono istanze di `NullObject`. Poiché abbiamo un oggetto e non un elemento puramente nullo, è necessario definire la funzione `request()` per stabilire il comportamento che deve avere l'oggetto quando l'oggetto non c'è. Questo permette anche di ridefinire il comportamento nel caso nullo semplicemente cambiando l'implementazione di `request()` nel `NullObject`, invece di modificare ogni riga di codice in cui abbiamo il caso `null`.
Ci sono tuttavia alcuni svantaggi nell'utilizzo di questo pattern:

- **forza uniformità**: clienti differenti potrebbero avere diverse idee sul "do nothing". Si potrebbero definire più `NullObject` in tal caso;
- **immutabilità**: oggetti `NullObject` non si possono trasformare in oggetti `RealObject`. Sarebbe utile avere un metodo che permette al `NullObject` di "trasformarsi" in un `RealObject`.
- **overhead**: l'aggiunta di `NullObject` implica occupazione in memoria. Tuttavia nel caso normale se il `NullObject` è di un solo tipo (quindi c'è solo un `NullObject`) e tale `NullObject` **non contiene stati**, non si ha la necessità di istanziare tanti oggetti nulli, bensì uno solo, e far si che ogni oggetto che dev'essere nullo punti a tale oggetto.

Singleton

Come visto precedentemente, si può avere la necessità di avere un oggetto unico condiviso dalle entità del sistema. Per questo motivo si utilizza il pattern **Singleton**.
Non si tratta di avere un oggetto che abbia ogni attributo e metodo statico (l'istanza d'altronde non avrebbe senso), ma di avere un'unica istanza all'interno del programma. Infatti avere una variabile globale rende un oggetto accessibile, ma non lo rende obbligatoriamente unico.
La soluzione risiede nel **vietare l'accesso al costruttore della classe**, rendendo il costruttore privato. Un metodo pubblico mi permette di ottenere l'istanza unica della classe:



Come si nota, `uniqueInstance` è un'istanza di `Singleton` statica. Il metodo `getInstance()` restituisce `uniqueInstance`.
Quando si restituisce `uniqueInstance`, bisogna assicurarsi che esso non sia nullo. Se esso non è nullo, semplicemente si restituisce il riferimento all'oggetto.
Nel caso del BST:

```
class BSTNullNode {
    static final BSTNullNode instance;
    private BSTNullNode() {}
    static BSTNullNode getInstance() {
        if (instance == null) instance = new BSTNullNode();
        return instance;
    }
}
```

I vantaggi del pattern Singleton sono molteplici:

- la classe Singleton ha controllo di come e quando i client accedono alla classe;
- si evitano variabili globali che tengono la sola istanza condivisa;
- si può controllare il numero di istanze nel programma. Da notare che la logica di `getInstance()` può essere modificata per avere anche più di una sola istanza. Questo permette di gestire un **pool di istanze**;
- è un meccanismo più potente di `static`, permettendo di gestire anche più istanze come visto precedentemente.

07- UML

Avere uno strumento per gestire la complessità software è importante. Ricordiamo che le tre sfide principali dell'ingegneria del software sono:

- gestire la complessità;
- gestire il cambiamento;
- gestire i difetti.

Il design Object Oriented ci permette di:

- **astrarre**, perché rappresentiamo simbolicamente le entità del sistema;
- **decomporre**, perché suddividiamo il sistema in sottosistemi, ad esempio in Java dividendo in package;
- **introdurre gerarchie**, tramite l'ereditarietà degli ADT.

Modellazione

> **Modellazione**
>
> La modellazione consiste nella costruzione di un'astrazione della realtà.

Le app di navigazione ad esempio forniscono una buona astrazione di dove andiamo. Si tratta quindi di semplificare correttamente la realtà (che di per sé è molto **complessa**).

> **Semplificazione corretta**
>
> Una semplificazione è **corretta** quando i dettagli ignorati sono irrilevanti e i dettagli rilevanti sono rappresentati in maniera fedele alla realtà da rappresentare.

Prima di fare un modello, bisogna avere in chiaro alcuni elementi, tra cui **quale aspetto della realtà bisogna rappresentare**. All'interno del nostro modello, rappresentiamo questi aspetti in maniera astratta, passando da entità reali R ad entità virtuali M. Bisogna inoltre tenere in considerazione le relazioni tra le entità che erano presenti nella realtà. Se esiste una relazione tra due oggetti reali R_1, R_2, denotata con f_R, allora devono esistere due entità virtuali M_1, M_2 in relazione tra loro, denotata con f_M.

Sistemi, modelli e viste

> **Modello**
>
> Con **modello** si intende una **descrizione astratta** di un sottoinsieme di un sistema.

> **Vista**
>
> Con **vista** si intende un sottoinsieme degli aspetti del modello. Essa rappresenta aspetti specifici di questo.

> **Notazione**
>
> Con **notazione** si intende un insieme di **regole grafiche o testuali** per rappresentare viste.

Consideriamo ad esempio una macchina. Una macchina piccola che riproduce in scala una macchina è un **modello**. Un gioco basato sulla guida di auto è un altro modello che rappresenta un differente aspetto della macchina.
Per costruire una macchina lego, che è un modello della macchina, abbiamo bisogno delle istruzioni. Tali istruzioni sono una **vista** del modello lego.

UML (Unified Modeling Language)

Il modello **UML** è una notazione per la rappresentazione di sistemi object oriented. Il linguaggio UML nacque con l'intento di unire le **migliori prassi** per la progettazione di sistemi object oriented, definendo uno standard industriale unificato. Si tratta quindi di un

modello **semi-grafico** e **semi-formale**, con una semantica precisa e un grande potere descrittivo, nonché con una collezione di diagrammi che permettono di descrivere i vari aspetti del modello realizzato. Esso consta di quattro viste, più una vista aggiuntiva che si sovrappone sulle altre, ossia la **vista dei casi utenti** (**use case view**). Le altre quattro viste sono:

- **logical view**, che descrive il **design** del sistema, in cui si esegue una sorta di fotografia statica delle entità che devono interagire nel funzionamento del sistema e delle loro relazioni con gli altri elementi. Le classi e le interfacce con le loro relazioni partecipano a questa vista, permettendo di analizzare le componenti dei moduli del sistema;
- **component view**, che descrive la struttura del design in termini di **modularità**, ossia come il sistema è diviso in moduli correlati tra loro. Permette quindi di dedurre l'architettura del sistema;
- **process view**, che permette di descrivere gli **aspetti dinamici** del sistema. I protagonisti sono gli **oggetti** e non più le classi, quindi gli elementi che a tempo di esecuzione sono presenti nel sistema. Si descrivono quindi le dinamiche del sistema;
- **deployment view**, che rappresenta come le componenti del sistema vanno mappate sulle parti fisiche che compongono l'ambiente in cui bisogna installare il software. Si descrivono quindi i requisiti delle componenti del sistema che deve supportare il software progettato.

Diagrammi UML

Un diagramma può essere di uno dei due tipi:

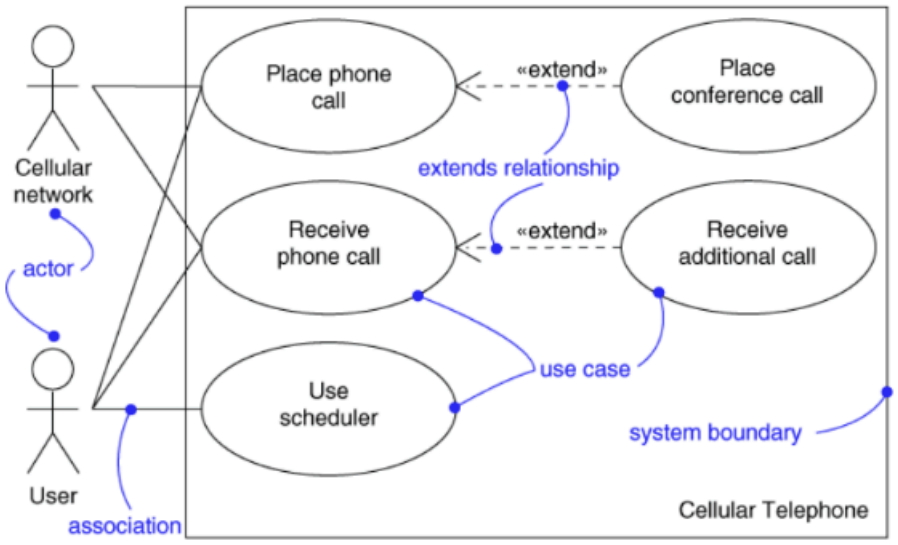
- **statico**, o **strutturale**. A questa classe appartengono i seguenti diagrammi:
  - diagrammi delle classi;
  - diagrammi delle componenti;
  - diagrammi di struttura composita;
  - diagrammi di deployment;
  - diagrammi degli oggetti;
  - diagrammi dei package;
- **dinamico**, o **comportamentale**. A questa classe appartengono i seguenti diagrammi:
  - diagrammi delle attività;
  - diagrammi dei casi d'uso;
  - diagrammi di macchine a stati;
  - diagrammi di interazione, che a loro volta si distinguono in:
    - diagrammi di sequenza;
    - diagrammi di comunicazione;
    - diagrammi di interazione generale;
    - diagrammi di temporizzazione, dove vengono specificate dinamiche temporali con indicazioni quantitative.

Tali diagrammi ci permettono di modellare il sistema in modi differenti:

- con la **modellazione funzionale** possiamo analizzare le funzioni del sistema. Utilizzeremo dei diagrammi dei casi d'uso;
- con la **modellazione a oggetti** analizziamo la struttura del sistema. Utilizzeremo dei diagrammi delle classi;
- con la **modellazione dinamica** si analizzano le dinamiche del sistema. Si utilizzano diagrammi di interazione.

Diagramma Use Case

Il diagramma **use case**, o diagramma dei **casi d'uso**, è costituito da attori che permettono di modellare entità esterne che comunicano col sistema, che possono essere utenti, sistemi esterni (sotto questo aspetto, il sistema esterno non riguarda la nostra implementazione del sistema, pertanto è astratto con un attore generico, quindi i dettagli di tale sistema **non sono di nostro interesse**) o ambienti fisici.
All'interno del diagramma troviamo una box con dei "pallini", che rappresentano **funzionalità**. La box rappresenta i **confini del sistema** che stiamo sviluppando. All'interno del sistema ci possono essere funzionalità che derivano da altre funzionalità del sistema: in tal caso la funzionalità che ne estende un'altra viene rappresentata con una freccia verso la funzionalità estesa.

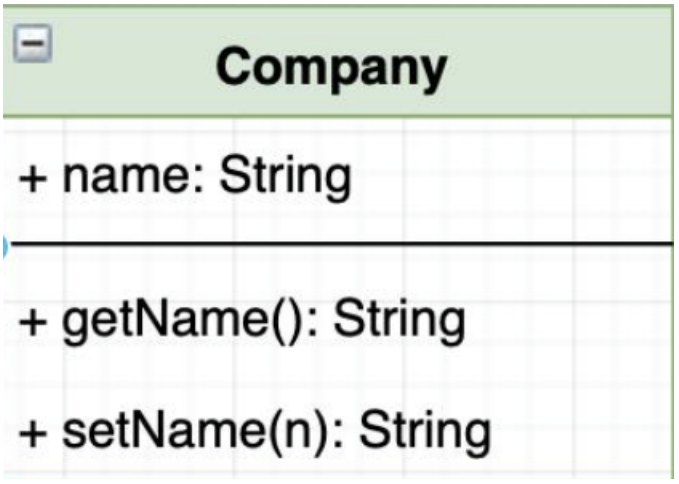


Class diagram

I protagonisti di questa classe sono le classi. I simboli UML delle classi sono dei rettangoli che rappresentano le **classi** in cui:

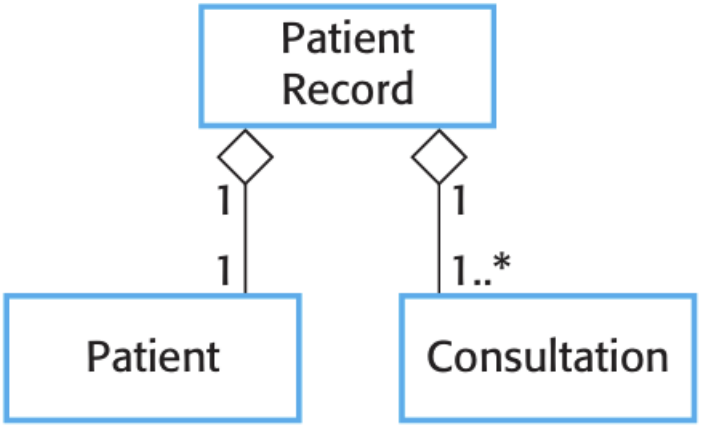
- gli attributi sono rappresentati nella parte superiore del rettangolo;
- i metodi sono rappresentati nella parte inferiore del rettangolo.

Una rappresentazione tipica di una classe è la seguente:

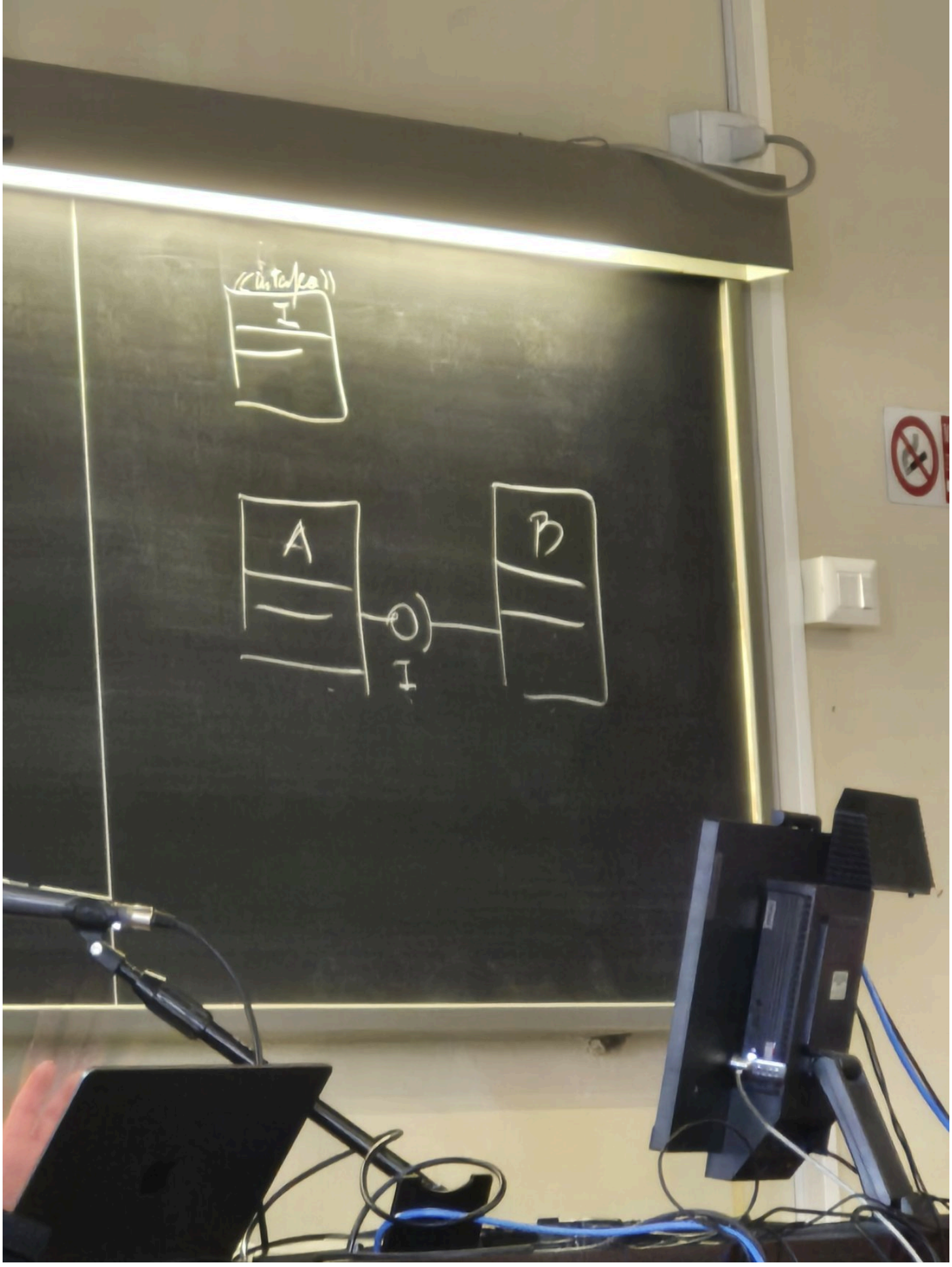


In questo caso il simbolo indica che l'attributo o metodo è pubblico. Si utilizza per indicare che l'attributo o metodo è privato.
Da precisare che le classi non rappresentano obbligatoriamente classi da implementare, ma generalmente **entità** all'interno del nostro sistema.
Vi sono alcuni tipi di relazioni tra le classi:

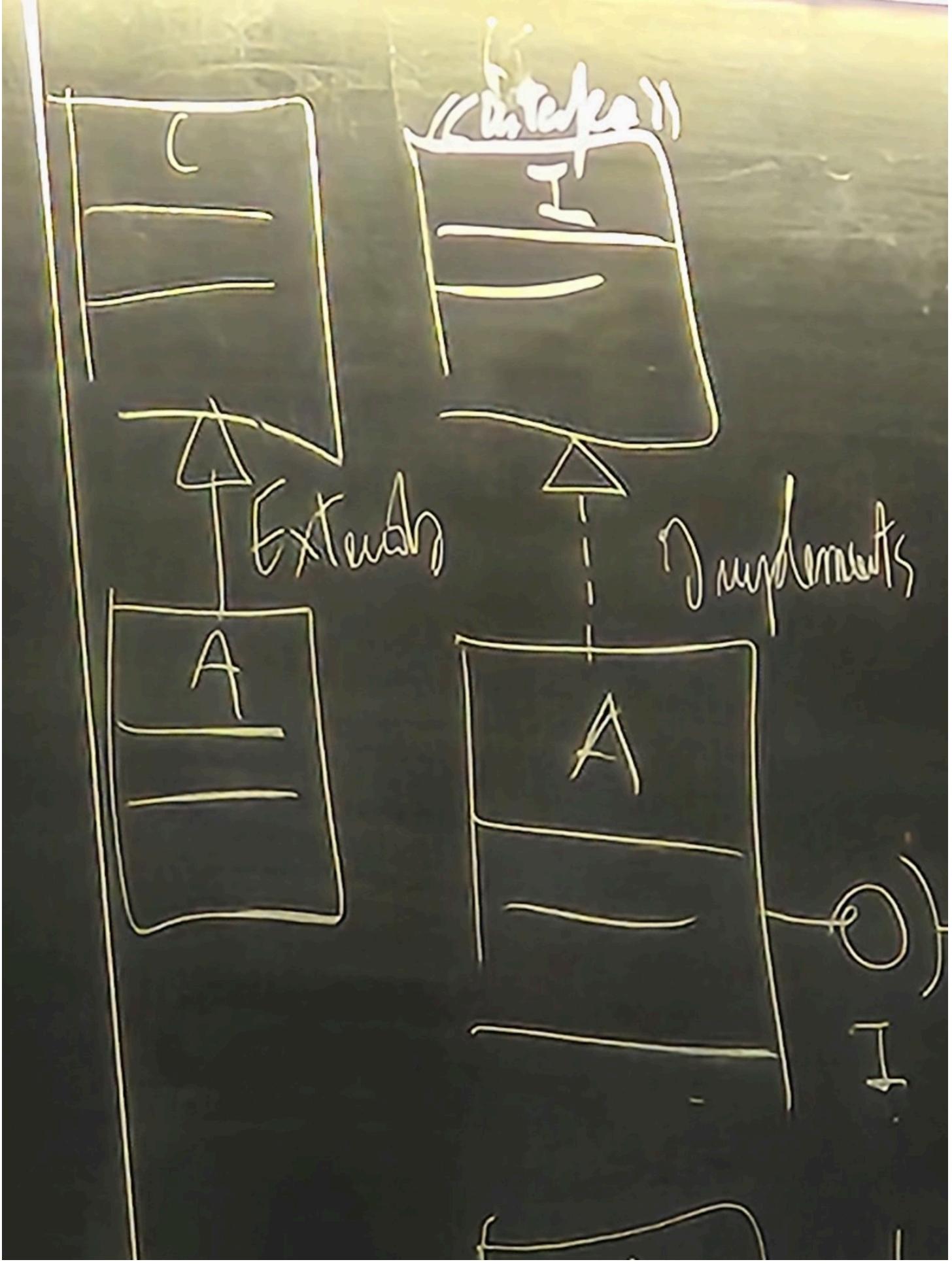
- **generalizzazioni**, ossia relazioni di derivazione. Indica che un'entità è una generalizzazione di un'altra. Sono rappresentate con frecce a punta: la freccia parte dall'entità sottoclasse punta verso l'entità superclasse;
- **associazione**, che indica una corrispondenza generica. Su un legame di tipo associativo si può specificare una cardinalità (o **molteplicità**). Ad esempio, ad un entità "Dipartimento" possono essere associate una o più entità "Dipendente". Per indicare un numero generico di entità in associazione, per la molteplicità utilizza la notazione `1` o direttamente . Tra le relazioni è possibile creare dei **vincoli**, ad esempio se ci sono più relazioni tra due o più entità coinvolte. Un vincolo può essere il vincolo **sottoinsieme**: un'associazione "manager", ad esempio, è sottoinsieme dell'associazione "membro", perché non vogliamo che un manager non sia membro del dipartimento;
- **dipendenza**, la forma di associazione più debole tra due entità. Significa semplicemente che un'entità dipende dall'esistenza di un'altra entità. Le dipendenze possono anche essere tra un metodo di un'entità e attributi di un'altra;
- **uses**, una relazione d'uso: un tipo usa un altro tipo.
- **aggregazione**, in cui un'entità è definita come un aggregato di altre entità. Si rappresentano con dei rombi vuoti. Ad esempio, un record per un paziente è composto da un paziente e da un numero indefinito di consultazioni:



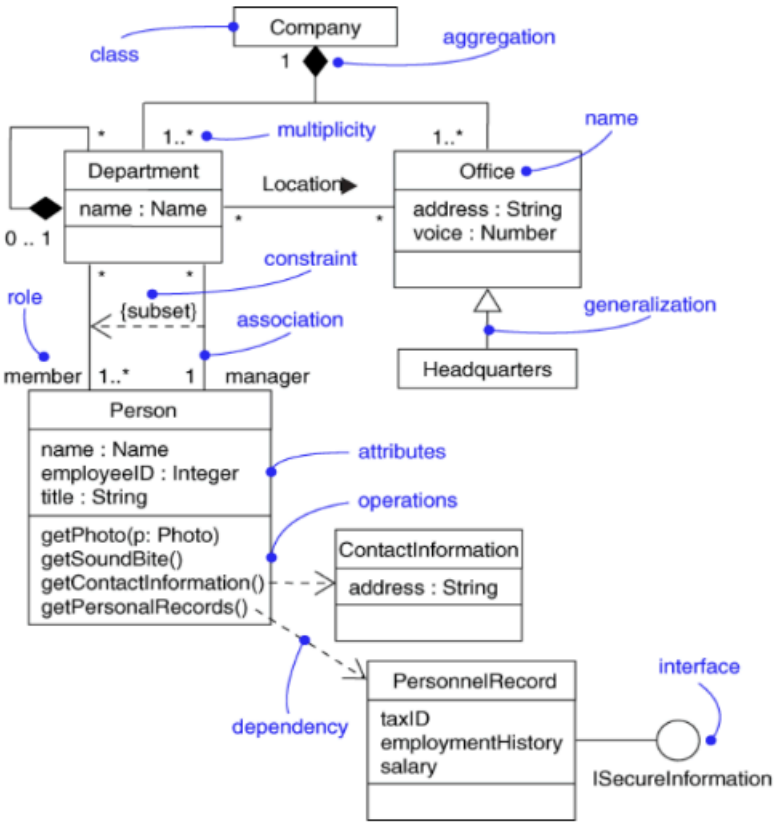
- **composizione**, situazione più particolare dell'aggregazione in cui una classe esiste solo dentro istanze di un'altra classe, ossia una **inner class**, quindi dichiarata all'interno di un'altra classe o tale che il suo ciclo di vita dipende dalla classe in cui esiste. Si rappresentano con dei rombi pieni. Esistono anche **auto-composizioni**. Ad esempio, ad una persona possono corrispondere uno o più fratelli, che sono comunque persone, o un dipartimento è diviso in più sotto-dipartimenti;
- **interfaccia**, rappresentato con una linea con un cerchio vuoto. Rappresenta un'interfaccia per un'entità. Generalmente per rappresentare un'interfaccia in UML si utilizza la stessa metodologia che si usa per rappresentare le classi, ponendo sopra la parola "interfaccia". Il "lollipop" che utilizziamo allora indica che l'interfaccia viene **implementata**, quindi l'entità mette a disposizione dei suoi client le funzionalità dell'interfaccia. Il motivo per cui utilizziamo tale simbologia è per esprimere la possibilità di utilizzare una classe con l'interfaccia di un'altra:



In questo esempio, B utilizza l'interfaccia di A. Per rappresentare l'implementazione dell'interfaccia, si può anche utilizzare una freccia tratteggiata verso l'interfaccia, a differenza della freccia non tratteggiata, che rappresenta estensione:



Un esempio di diagramma di classe è il seguente:



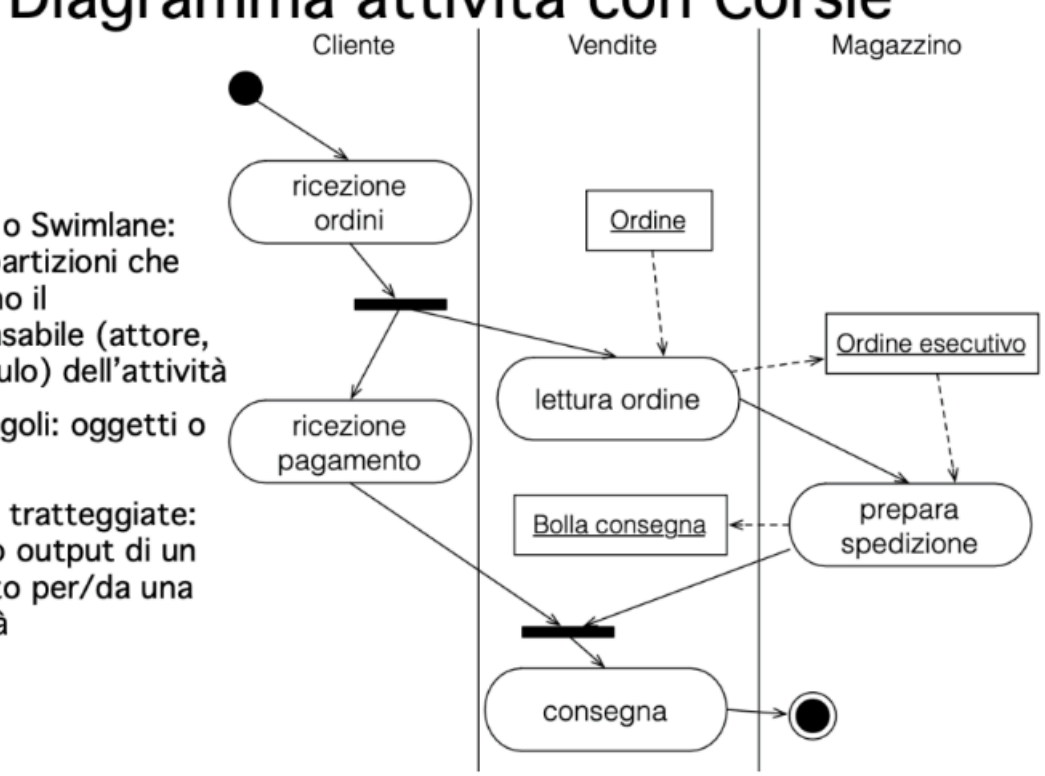
Nell'esempio vediamo `Company`, composizione (sebbene sia segnato come aggregazione, ma anche in aula abbiamo detto che è un errore) delle entità `Office` e `Department`. `Office` è una generalizzazione di `Headquarters`, sottoclasse di `Office`. Tra `Department` e `Office` l'associazione `Location` ha cardinalità molti a molti, indicata con .
`Department` è dotata di una **auto-composizione**, che indica il fatto che un dipartimento possa essere costituito da più dipartimenti. Sono anche presenti due associazioni con `Person`:

- `member`, con cardinalità `1` che indica che una persona è membro di uno o più dipartimenti;
- `manager`, con cardinalità `1` dal lato di `Person`, che indica che la persona è manager di un solo dipartimento. Da notare che `manager` ha un vincolo `subset` con `member`: una persona è manager di un dipartimento solo se è anche membro di questo.

I metodi `getContactInformation()` e `getPersonalRecords()` presentano dipendenze con le classi `ContactInformation` e `PersonnelRecord`. `PersonnelRecord` è l'implementazione dell'interfaccia `ISecureInformation`.

Diagramma delle attività

Supponendo di aver individuato una funzionalità del sistema attraverso uno use case, non è detto che si abbia chiarezza su come portare a termine tale attività. Un semplice metodo per modellare l'esecuzione della funzionalità è tramite il **diagramma delle attività**.
Il diagramma è caratterizzato da un pallino nero pieno per indicare l'inizio, un pallino nero con un cerchio esterno per indicare la fine e una serie di attività poste una dietro l'altra e concatenate con frecce. Se ci sono attività che devono essere eseguite in parallelo, si disegna una **barra di fork**, che divide la freccia successiva in una o più frecce. Nel momento in cui queste attività si devono sincronizzare tra loro, ossia ci dev'essere un **join**, si riuniscono in una nuova barra e la freccia diventa una singola freccia nuovamente.
Se invece non si ha un parallelismo ma un'**alternativa del percorso di esecuzione**, si utilizza un rombo vuoto. Anche in questo caso ci sarà un **merge**, sempre indicato con un rombo.
Un'ulteriore complicazione si osserva nel caso in cui si organizzano le attività del diagramma in **colonne**, dette **swim lane**, in cui si individua il **responsabile** di certe attività. Il responsabile è individuato col nome della colonna.
Il diagramma delle attività ci permette di raccontare una funzionalità a livello dell'intero sistema. Risulta essere un procedimento che è ideale come inizio di un processo top-down per lo sviluppo del sistema. Ci aiuta quindi a realizzare e progettare uno use-case.
Un altro aspetto importante riguarda la possibilità di **identificare istanze di oggetti** necessarie per lo svolgimento di una certa attività. Questi possono essere specificati con rettangoli con un nome dentro. Le istanze di oggetti, in particolare, non sono ancora con un tipo identificato.



Collaboration Diagram

Lo scopo del **diagramma di collaborazione** è quello di descrivere le interazioni tra oggetti. Il flusso dei messaggi è indicato da **frecce** accanto alle associazioni tra istanze che partecipano all'interazione.
Questo diagramma permette di identificare quale interazione viene svolta, quindi quale **metodo** o **messaggio** viene inviato ad un altro oggetto. Si identifica con un **numero sequenziale** anche un ordine temporale con cui avvengono i messaggi.
Il diagramma mette in evidenza quali oggetti sono coinvolti e quali interagiscono tra di loro. Ogni interazione tuttavia provocherà una serie di interazioni differenti tra gli elementi del sistema. Pertanto si avranno tanti diagrammi, ognuno dei quali definisce una serie di interazione tra i vari elementi del sistema.

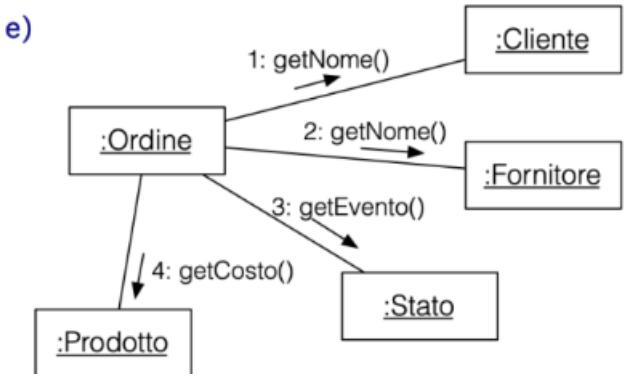
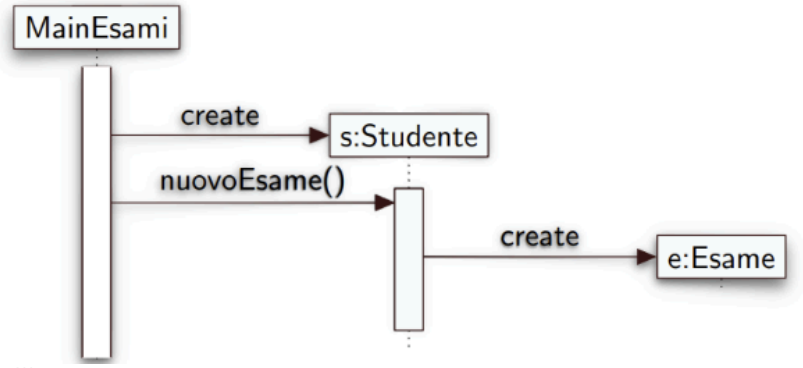


Diagramma di sequenza

In questo tipo di diagramma si mette in evidenza la sequenza temporale delle azioni. Questo tipo di diagrammi può essere di due modi:

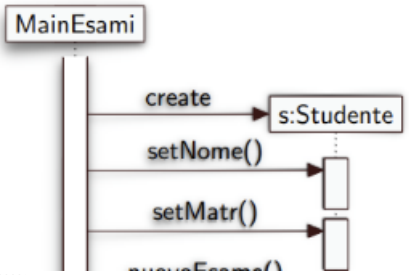
- il **diagramma a scaletta**, più comune. Un software a oggetti utilizza comunque una esecuzione sequenziale di chiamate e interazioni tra oggetti. In questo diagramma si rappresenta allora un omino, che indica che ad iniziare un'attività è qualcuno o qualcosa che prende iniziativa e fa scattare il primo step dell'interazione. Ci sono poi una serie di entità coinvolte, rappresentato da rettangoli. Il rettangolo viene presentato prima perché significa che si ha un'**istanza di oggetto**. Una linea tratteggiata individua una **timeline**; alla fine della timeline, una 'x' indica che l'**istanza viene distrutta**. Nella JVM se un oggetto rimane senza riferimenti, il garbage collector de-alloca automaticamente questi oggetti. Un rettangolo pieno presente nella timeline indica che l'istanza di oggetto è in esecuzione. Il diagramma a scaletta è ideale per indicare logiche di funzionamento **distribuite**, in cui ogni oggetto partecipa all'implementazione del software;

```java
public class MainEsami {
  public static void main(String[] args)
{
    Studente s = new Studente();
    s.nuovoEsame("Italiano", 8);
  }
}
```



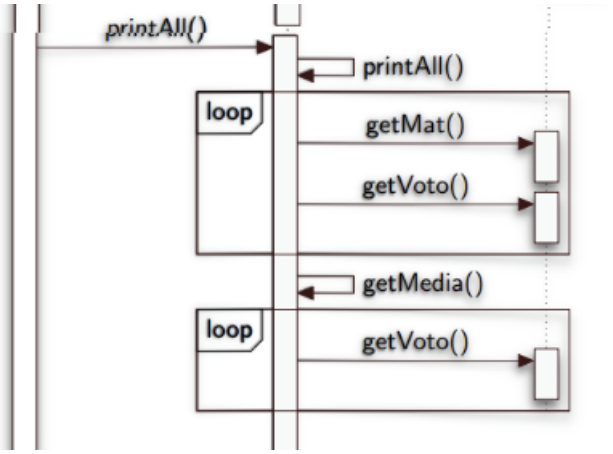
come si vede dall'immagine soprastante, allo studente `s` viene assegnata la creazione dell'esame, seguendo un sistema di chiamate a cascata (quindi chiamate distribuite tra gli oggetti del sistema);

- il **diagramma a fork**, ideale per indicare logiche **centralizzate**: un attore centrale si occupa di scambiare messaggi con le istanze di oggetto presenti nel sistema.



nella frazione mostrata, la classe `MainEsami` si occupa di inizializzare lo studente `s`.

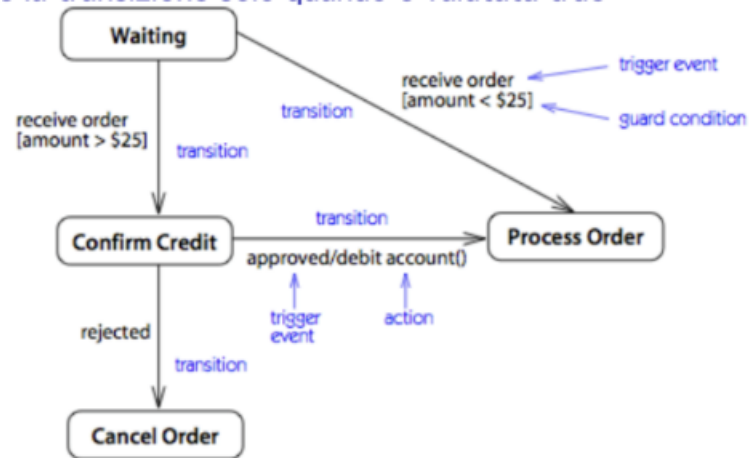
Può capitare che una certa interazione si ripeta più volte. Allora si racchiude uno scambio di messaggi in una box e si aggiunge un'etichetta **loop**:



Nella frazione soprastante, l'etichetta loop sottolinea che da un altro oggetto (a destra un oggetto di tipo `Esami`) bisogna recuperare le informazioni "matricola" e "voto". Come si deduce dal nome della procedura chiamate (`printAll()`), il processo viene ripetuto per ogni esame.

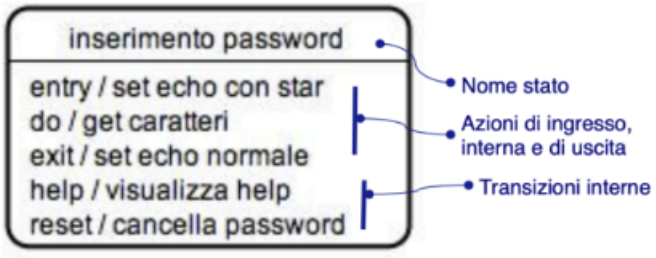
Diagramma degli stati

Il **diagramma degli stati** permette la rappresentazione del ciclo di vita delle entità del sistema. Supponiamo di avere una classe rappresentante un ADT con dei dati interni che rappresentano uno stato che cambia nel tempo. Vogliamo rappresentare il ciclo di vita dell'entità.
Un cerchio nero rappresenta lo stato iniziale dell'oggetto. Un altro cerchio nero con un contorno indica lo stato finale. Indica esplicitamente che si esce dallo stato. Frecce etichettate descrivono eventi che portano a cambiamenti di stato. Dei rettangoli con angoli arrotondati rappresentano gli stati dell'entità.

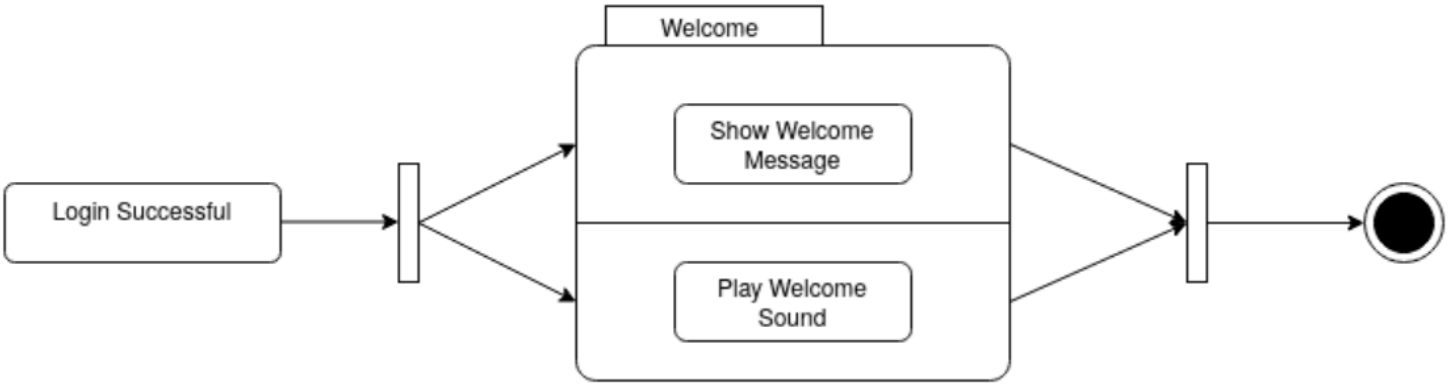


Le transizioni da uno stato ad un altro non devono solamente avere un nome, ma possono avere una **guardia**, ossia una **condizione** che se non viene verificata impedisce che la transizione avvenga. Possono anche avere un'**azione abbinata** che produce un certo risultato in virtù del cambiamento di stato. Si utilizza la notazione:

All'interno dello stato si può definire un insieme di azioni di ingresso e di uscita e azioni che caratterizzano quello stato.



Gli stati, inoltre, possono essere **composti e paralleli**. Questo dà la possibilità di descrivere uno stato come un sotto-diagramma degli stati, permettendo una modellazione più precisa dello stato di un oggetto. Possono esistere anche degli stati **concorrenti** e utilizzare quindi i meccanismi di fork e join:



In questo caso alla fine della transizione da entrambi gli stati presenti nello stato "Welcome" si giunge allo stato finale.

08 - Factory Method Pattern

Factory Method

All'interno dei nostri programmi dobbiamo utilizzare servizi forniti da altri agenti. Supponiamo di avere già l'implementazione di funzionalità, e lavoriamo sul lato client. Sappiamo esattamente quando è necessario istanziare un oggetto per utilizzare delle funzionalità di una classe (di cui sappiamo che è necessaria l'esistenza). Non vogliamo tuttavia istanziare esplicitamente la classe: in tal caso, infatti, dobbiamo sapere l'ADT di questo oggetto. Il problema quindi è il seguente: **sappiamo quando creare un oggetto di servizio, ma non sappiamo (o non vogliamo) stabilire quale classe istanziare**. Non vogliamo legare il codice client ad un ADT, in quanto tale legame è una dipendenza. Vogliamo rispettare il principio di programmazione secondo interfacce, in modo tale da rendere il programma client più riusabile, quindi **indipendente dal tipo di classe da utilizzare, pur sapendo di creare un'istanza di classe**.
Questo si differenzia dal **polimorfismo**. Un'istruzione è polimorfica quando lo stesso simbolo può indicare implementazioni diverse:

```
public class Studente extends BasicStud {
    private List<Esame> esami = new ArrayList<>();
    public void nuovoEsame(String m, int v) {
        Esami e = new Esame(m, v);
        esami.add(e);
    }
}

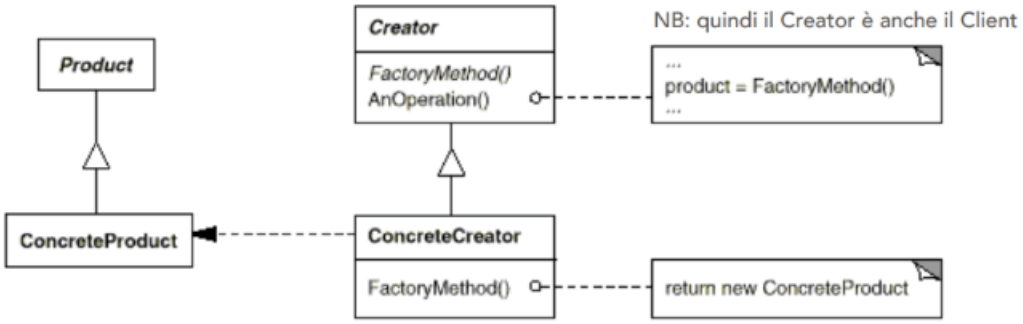
public class Sospeso extends BasicStud {
    public void nuovoEsame(String m, int v) {
        System.out.println("Non possibile");
    }
}

public class Client {
    BasicStud s = new Studente();
    public void registra() {
        s.nuovoEsame("Maths", 8);
    }
}
```

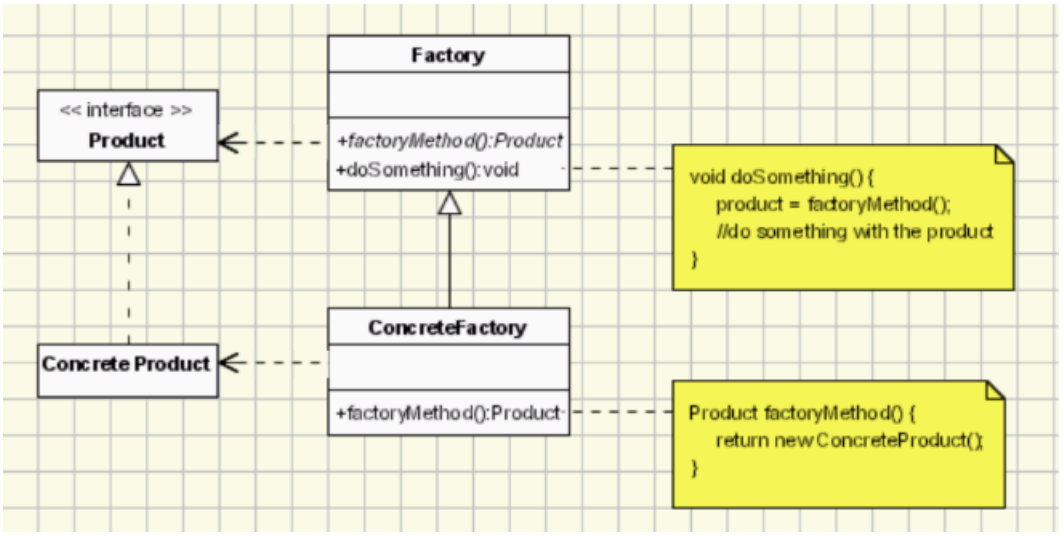
L'istruzione `registra()` che crea una dipendenza con la classe specifica `Studente` si trova nel client. La classe client complessivamente ha una dipendenza: infatti `nuovoEsame()` è polimorfico, ma dipende dalla classe `Studente` e non dalla classe `BasicStud`.

Soluzione

La soluzione che il pattern ci suggerisce consiste nello strutturare il programma in una maniera diversa.
Consideriamo una classe astratta `Creator`, che costituisce il client: si tratta del modulo software in cui istanzieremo un oggetto di servizio. L'oggetto a cui faremo riferimento è un `Product`, pensato sempre come classe astratta. Il software è concepito come un client astratto che istanzia un prodotto astratto.
In `Creator` il metodo `AnOperation()` fa riferimento alla presenza di un metodo che faccia uso dell'ADT astratto `Product`. Di tale ADT astratto dovrà esistere una implementazione; indichiamo tale implementazione come `ConcreteProduct`. Per far sì che il client resti indipendente da `ConcreteProduct`, utilizziamo un `FactoryMethod()`: tale metodo ha come scopo quella di creare l'oggetto di istanza, **nonché la scelta esplicita dell'istanza di oggetto da utilizzare**.
In questo modo è possibile implementare un algoritmo `AnOperation()` che fa utilizzo di un oggetto, e nel momento in cui bisogna istanziare un prodotto assegniamo la responsabilità di tale operazione a `FactoryMethod()`.



Il `ConcreteCreator` avrà all'interno dell'implementazione di `FactoryMethod()` la creazione di un oggetto di istanza specifico. Il programma `Creator` quindi lavora dall'interfaccia astratta e il software diventa indipendente dall'ADT da utilizzare. Se in un secondo momento si inventa un nuovo `ConcreteProduct`, basta creare una nuova classe `ConcreteCreator` in modo tale che `FactoryMethod()` restituisca questo nuovo `ConcreteProduct` e il modulo software client è riutilizzabile.



Implementazione esempio in Java

```
public interface Product {
    void request();
}

class CProduct implements Product {
    public void request() {
        System.out.println("CProduct");
    }
}

abstract class Creator {
    abstract public Product getProduct();

    public void AnOperation() {
        Product p = this.getProduct();
        p.request();
    }
}

public class CCreator extends Creator {
    @Override
    public Product getProduct() {
        return new CProduct();
    }
}

public class Client {
    public static void main(String[] args) {
        Creator myApp = new CCreator(); // incapsulamento
        myApp.AnOperation(); // usa il servizio del prodotto in maniera indiretta

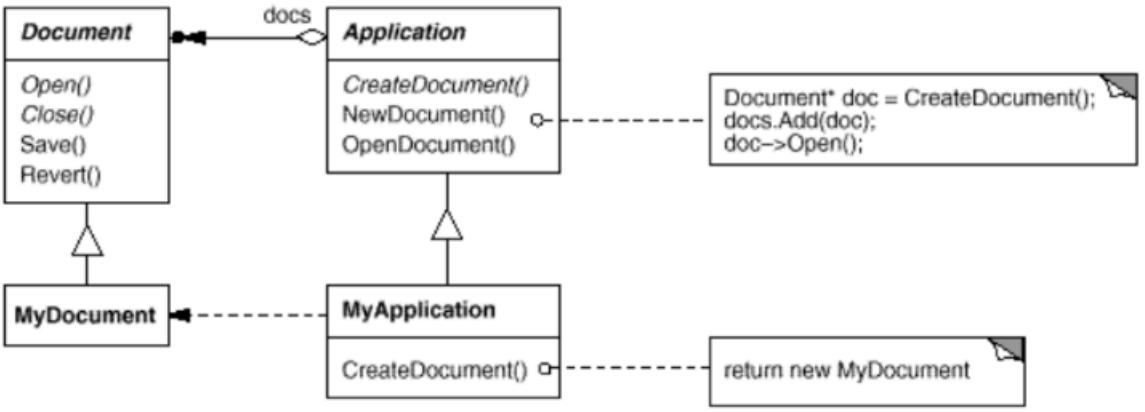
        Product myProduct = myApp.getProduct(); // incapsulamento
        myProduct.request(); // usa il servizio del prodotto in maniera diretta
```

```
    }
}
```

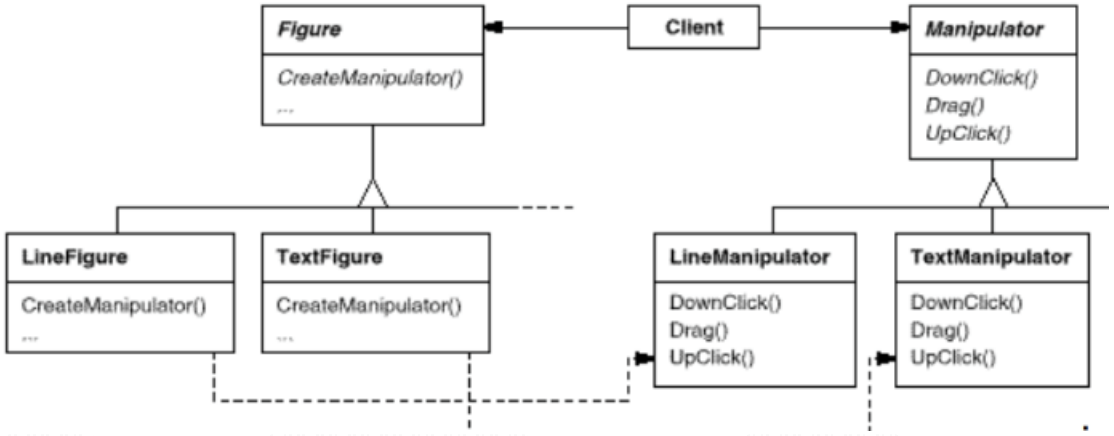
Con questo pattern abbiamo spostato la dipendenza dall'oggetto di servizio all'oggetto che ne incapsula la sua creazione. `CProduct` viene effettivamente utilizzato, ma in `Client` non si istanzia esplicitamente nessun `CProduct`. Questo crea quindi una **dipendenza indiretta** tra `Client` e `CProduct`: invece di istanziare direttamente l'oggetto `CProduct` nel `Client`, utilizziamo la classe `CCreator` come "tramite", rendendo il client **direttamente dipendente** a `CCreator`.

Utilità dello schema

Supponiamo di voler realizzare un'applicazione che tratti documenti, quindi un word processor. Vogliamo scrivere l'applicazione in modo tale che **supporti più tipologie di documenti**. I file quindi possono avere diversi formati, come file di testo, pdf e file word. Scriviamo allora una classe astratta `Application` che include i metodi `CreateDocument()`, `NewDocument()`, `OpenDocument()`. I metodi chiaramente saranno polimorfici in virtù del documento. La funzionalità di base è `CreateDocument()`. Possiamo strutturare quindi il software nel seguente modo:



Forniamo allora al nostro software un'indipendenza rispetto alla creazione di oggetti, creandoli senza fornire esplicitamente il tipo incapsulando l'istanziamento di oggetto.
Un'altra situazione in cui è utile il metodo è durante la creazione di **gerarchie di classi parallele**:



La classe astratta `Figure` ha un metodo `CreateManipulator()` che restituisce un `Manipulator`, ulteriore classe astratta che fornisce metodi per manipolare la figura in virtù dell'implementazione. Le classi `LineFigure`, `TextFigure` sono figure di tipo diverso, in cui verranno implementate le funzioni `CreateManipulator()`. Invochiamo sugli identificatori quindi `CreateManipulator()` per ottenere un'istanza corretta di `CreateManipulator()` utilizzando tuttavia le interfacce `Figure`, `Manipulator`: il client quindi funziona con tutti i tipi di figure e tutti i tipi di manipolatore. Questa situazione è praticamente molto simile alla situazione dell'**Iterator Pattern**: il `CreateIterator()` è un factory method.

Conseguenze del pattern

Il pattern permette di fornire indipendenza dall'ADT, fornendo possibilità di riutilizzare il software. Tuttavia, l'implementazione del `ConcreteCreator` dev'essere modificata per utilizzare altre funzionalità. Ne segue quindi che **il client è riutilizzabile, ma non le implementazioni dei creator**.
Poiché per ogni `ConreteProduct` va derivato un nuovo `Creator`, si ha una **proliferazione di classi**.

Creator come servente di client esterni

Il factory method può essere utilizzato per fornire un servizio di creazione di oggetti astratti:

```
public class Creator {
    public IStudente getStudente() {
        return new Studente();
    }
}

public class SoCreator extends Creator {
    public IStudente getStudente() {
        return new Sospeso();
    }
}

public class Client {
    public void registra(Creator sc) {
        if (sc == null) sc = new SoCreator();
        //...
        IStudente s = sc.getStudente();
        s.nuovoEsame("Maths", 8);
    }
}
```

In questo caso in `Client` non si utilizza comunque il tipo esplicito dello studente, mentre `Creator` **invece di essere un tipo astratto è implementato come classe istanziabile**. La classe astratta quindi è stata collassata in una classe implementabile, fornendo comunque possibilità polimorfiche e astraendo la necessità di esplicitare il tipo di studente per il metodo `nuovoEsame()`.
Se `Creator` viene passato come nell'esempio in maniera astratta il factory method presenta un enorme vantaggio.

Variante: Creator generico

Perché funzioni il pattern è necessario creare una sottoclasse della classe `Factory`. Vi sono molti modi per implementare il Factory Method. La variante più simile all'implementazione classica consiste nell'utilizzo di **generics**.
Una prima idea è quella di creare un software che funzioni con tanti tipi di prodotto: il prodotto che si utilizza è intercambiabile. Il software quindi è **parametrico** rispetto al prodotto.
Allora potremmo creare una classe `Creator` astratta come segue:

```
public interface Product {
    void request();
}

public class AProduct implements Product {
    void request() {
        System.out.println("Request sent to AProduct");
    }
}

public class BProduct implements Product {
    void request() {
        System.out.println("Request sent to BProduct");
    }
}

public class CProduct implements Product {
    void request() {
        System.out.println("Request sent to CProduct");
    }
}

abstract class Creator<P extends Product> {
    abstract public P getProduct();

    public void AnOperation() {
        P p = this.getProduct();
        p.request();
    }
}
```

```
public class ConcreteCreator extends Creator<AProduct> {
    @Override
    public ProductA getProduct() {
        return new ProductA;

    }

}
```

Il vantaggio nell'utilizzo delle generics risiede nel fatto che il `ConcreteCreator` può essere definito come un tipo in modo tale che la visibilità del tipo è visibile al compilatore. Il tipo di prodotto scelto è visibile al compilatore. Quindi se volessimo un `ConcreteCreator` per il prodotto `BProduct`, sarebbe necessario creare un nuovo `ConcreteCreator` che estende `Creator<BProduct>`.

Potremmo tuttavia pensare di evitare di scrivere un metodo `getProduct()` astratto ed eseguire direttamente l'operazione:

```
abstract class Creator<P extends Product> {
    public P getProduct() {
        return new P();
    }

}
```

Questa istruzione **non compila**, in quanto `P` non può essere risolto a tempo di compilazione.

Variante: factory method parametrico

Possiamo modificare il factory method e rendere esso parametrico. Se il factory method non dev'essere un metodo specifico ma incapsula una logica di scelta di prodotto, avere un unico `ConcreteCreator` con un `FactoryMethod()` parametrico può essere comodo. Tuttavia, avere un solo `Creator` può essere **limitante**.

```
public class StCreator {
    public static IStudente getStudente(boolean a) {
        if (a) return new Studente();
        return new Sospeso();
    }
}
```

Da questo momento in poi, tuttavia, il software può usare **un solo creator**. Se un domani volessimo creare un nuovo studente (assumendo in questo esempio vi sia solo `Sospeso` e `Studente`), è necessario modificare anche il client in quanto esso è dipendente dal `Creator`. Il numero di `Product` quindi **non è facilmente espandibile**.

Variante: factory method statico

Un'altra alternativa consiste nel rendere il factory method **statico**. All'interno della classe `ConcreteCreator` è possibile definire attributi statici che ne definiscono lo stato. In questo modo:

- non è necessario istanziare il `ConcreteCreator`;
- non bisogna dichiarare alcun `Creator` astratto;
- **non è possibile eseguire override del factory method**. Questo può essere limitante in quanto i metodi statici non possono essere ridefiniti.

A livello implementativo allora implementeremo `StCreator` con le seguenti modifiche:

```
public class StCreator {
    private static boolean a = true;
    public static IStudente getStudente() {
        if (a) return new Studente();
        return new Sospeso();
    }
}
```

Seguendo questa maniera le modifiche allo stato della classe `StCreator` vengono riflesse sui moduli e sugli oggetti che utilizzano `StCreator`.

In entrambi i casi (quindi sia factory method parametrico, sia factory method statico) la logica dell'istanziazione degli oggetti viene unificata in un unico metodo creazionale. Tale algoritmo creazionale viene condiviso in tutti i punti del software.

Variante: `Creator` con riflessione

Possiamo implementare un unico `Creator` utilizzando un parametro che permette di istanziare istanze del prodotto richiesto. La nostra classe `Creator` sarà la seguente:

```
class Creator {
    Class<?> cp;
    public Creator(Class<?> cp) {
        this.cp = cp;
    }

    public Product getProduct() {
        if(this.cp != null) try {
            return (Product) this.cp.getConstructor().newInstance();
        }
        catch (InstantiationException | IllegalAccessException | IllegalArgumentException |
InvocationTargetException | NoSuchMethodException | SecurityException e) {
            e.printStackTrace();
            return null;
        }
        return null;
    }
    //...

}

public class Client {
    public static void main(String[] args) {
        Creator myApp = new Creator(CProductA.class);

        Creator myOtherApp = new Creator(CProductB.class);
        //...
    }

}
```

Essenzialmente si controlla che tipo di classe `cp` identifica. Si estrae con `getConstructor()` il costruttore di `cp`, invocando il metodo per allocare un'istanza con `newInstance()`. Si tratta di un set di chiamate della libreria **Reflection API** di Java.

La Reflection API ispeziona a runtime un'istanza di una classe, deducendo il tipo di classe.

Risulta necessario ricordare che i generics sono simboli risolti a tempo di compilazione: si possono usare solamente a sinistra di un assegnamento. A tempo di esecuzione possiamo solamente verificare che il tipo a sinistra è compatibile con quello specificato a destra. Si tratta quindi di **etichette**, simili ai `#define` di C ma elaborate a tempo di compilazione; la reflection API invece controlla a tempo di esecuzione oggetti allocati in memoria.

Un approccio errato quindi che utilizza il Reflection API invece è il seguente:

```
public class Client {
    public static void main(String[] args) {
        Creator<?> myApp = new Creator<CProductB>(); // specificato a compile-time
        //...
    }

}
//...
public class Creator<P extends Product> {
    P obj;
    Class<?> cp = obj.getClass();
```

```
    //...
}
```

Il codice crea un'eccezione, perché a tempo di esecuzione `obj` è un **oggetto nullo**, quindi il metodo `getClass()` causa una `NullPointerException`.

Dependency Injection

Le implementazioni che abbiamo visto non utilizzano derivazioni, ma composizioni. Tramite un oggetto di un certo tipo si riesce ad incapsulare all'interno l'oggetto di servizio necessario.

Il meccanismo utilizzato prende il nome di **dependency injection**. Supponiamo una classe `C` utilizzi un servizio fornito da `S` (quindi `C` **non** dipende da `S`). Esistono varie implementazioni di `S`, ad esempio `S1`, `S2`: vogliamo che `C` non dipenda dalle implementazioni `S1`, `S2`. Scriviamo la classe `C` affinché essa non sia dipendente dalle implementazioni.

Facciamo allora in modo che `C` sia configurata con l'implementazione di `S` che si voglia utilizzare:

```
    C myC = new C1(new S1());
```

La dipendenza viene così creata **a tempo di esecuzione**, rendendo la composizione di oggetti dinamica.

09 - Abstract Factory

Abstract Factory

Supponiamo di sviluppare un'applicazione grafica per realizzare interfacce. Vogliamo una flessibilità per cambiare l'interfaccia grafica, ma anche far sì che cambiare da una modalità di interfaccia ad un'altra modifichi l'intera interfaccia.
Vogliamo quindi che sia rispettato un **vincolo**: cambiando versione dell'interfaccia, modifichiamo le versioni degli oggetti in blocco.

In questo caso abbiamo bisogno di un `Creator` che permetta la produzione di oggetti correlati; la factory dev'essere un **oggetto separato** dal client.

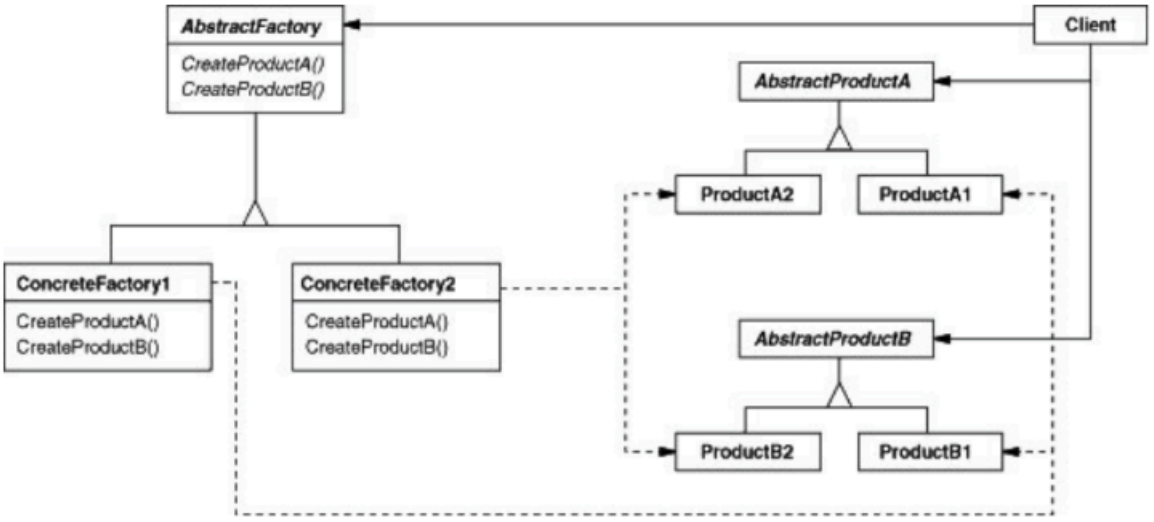
Ad esempio, supponiamo di avere un menu da dover riempire con determinate opzioni, come segue:

```
public void installDisneyMenu() {
    Menu disney = //crea un menu...
    disney.add("Disney world");
    disney.add("Donald duck");
    disney.addGrayBar();
    //...
}
```

Se volessimo modificare il menu è necessario modificare il codice, diventando meno flessibile. La soluzione è passare al client un oggetto `Factory`:

```
public void installDisneyMenu(WidgetFactory myFactory) {
    Menu disney = myFactory.createMenu();
    disney.addItem("Disney World");
    //...
}
```

Nel nostro codice `WidgetFactory` è una classe **astratta**. La abstract factory permette di definire una serie di metodi astratti che verranno implementati per la produzione di un set di oggetti.



Varianti di implementazione

Come nel factory method, abbiamo molteplici varianti di implementazione del pattern.

Variante classica

Si definisce una classe astratta `AbstractFactory` e si definiscono più classi derivate per le singole factory:

```
abstract class WidgetFactory {
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
}
```

```
class MacWidgetFactory extends WidgetFactory {
    public Window createWindow() { return new MacWindow(); }
    public Menu createMenu() { return new MacMenu(); }
    public Button createButton() { return new MacButton(); }

}
//...
```

In questo modo il client si libera dalle dipendenze dirette ma fa riferimento ad una abstract factory. Il legame della abstract factory con le sue sottoclassi è un legame di tipo statico.

Variante con composizione

Si evita un legame statico, utilizzando al posto della derivazione la **composizione**. Si utilizzano riferimenti a factory di prodotti **esterne**.
In questo caso nella classe astratta definiamo il comportamento dei metodi, utilizzando delle classi pre-esistenti:

```
abstract class WidgetFactory {
    WindowFactory wf;
    MenuFactory mf;
    ButtonFactory bf;

    public Window createWindow() { return (Window) wf.createFancyWindow(); }
    public Menu createMenu() { return (Menu) mf.createMenu(); }
    public Button createButton() { return (Button) bf.createMenu(); }
}

class MacWidgetFactory extends WidgetFactory {
    public MacWidgetFactory() {
        wf = new MacWindowFactory();
        mf = new MacMenuFactory();
        bf = new MacButtonFactory();
    }

}
```

Affinché funzioni basta collegare il codice con l'implementazione di `MacWindowFactory`, `MacMenuFactory` e `MacButtonFactory`, che si suppone siano implementazioni pre-esistenti e che implementano le funzioni di creazione di finestre, menu e pulsanti. Deleghiamo quindi alle sottoclassi l'inizializzazione delle factory da usare.

Variante con Reflection API

Basarsi sulla Reflection API permette di far riferimento ai nomi delle classi dei prodotti che vogliamo utilizzare nel momento di creazione delle factory da utilizzare. Si utilizza quindi un solo creator per tipo di oggetto e si parametrizza per la versione di oggetto.

È possibile modificare il codice con la seguente logica: invece di creare dei creator per il prodotto, utilizziamo i creator semplicemente per dire qual è la classe di prodotto da utilizzare:

```
public abstract class AbsFact {
    abstract Class<? extends Window> windowClass();
    abstract Class<? extends Menu> menuClass();
    abstract Class<? extends Button> buttonClass();

    public Button getButton() { return (Button) buttonClass().getCostructor().newInstance(); }
    ...
}

public class MacWidgetFactory extends AbsFact {
    Class<? extends Window> windowClass() { return MacWindow.class; }
    ...
}
```

Variante su prototipi

Esiste un'ulteriore versione basata sui **prototipi**. Analizzeremo questa variante quando parleremo del prototype pattern.

Vantaggi del pattern

Il motivo per cui utilizziamo il pattern risiede nella volontà di avere indipendenza dal tipo concreto di prodotti creati e usati. Vogliamo poter configurare il sistema con una tra famiglie di prodotti, imponendo agli utenti dei prodotti di usare **set di prodotti** e non singoli prodotti.

Lo svantaggio nell'utilizzo di tale approccio risiede nel fatto che in fase di design è necessario identificare correttamente il set di prodotti: una volta che esso è definito, **non è più possibile cambiarlo**. Non si possono creare nuovi set che abbiamo nuovi oggetti rispetto agli altri.

Combinazione con altri pattern

L'abstract factory pattern viene solitamente utilizzato:

- con il pattern Singleton per garantire che vi sia un'unica istanza di factory;
- con il pattern Prototype, per implementare la factory utilizzando i prototipi.

Prototype Pattern

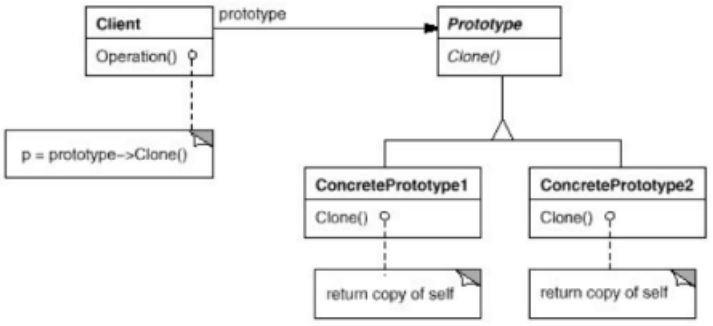
La domanda che ci poniamo è la seguente:

> *Come possiamo far funzionare un programma a run-time senza che a tempo di compilazione si conosca l'ADT dell'oggetto?*.

L'idea è la seguente: usare un'istanza di un oggetto originale come **stampo** per crearne altre. Si duplica così un'istanza base:

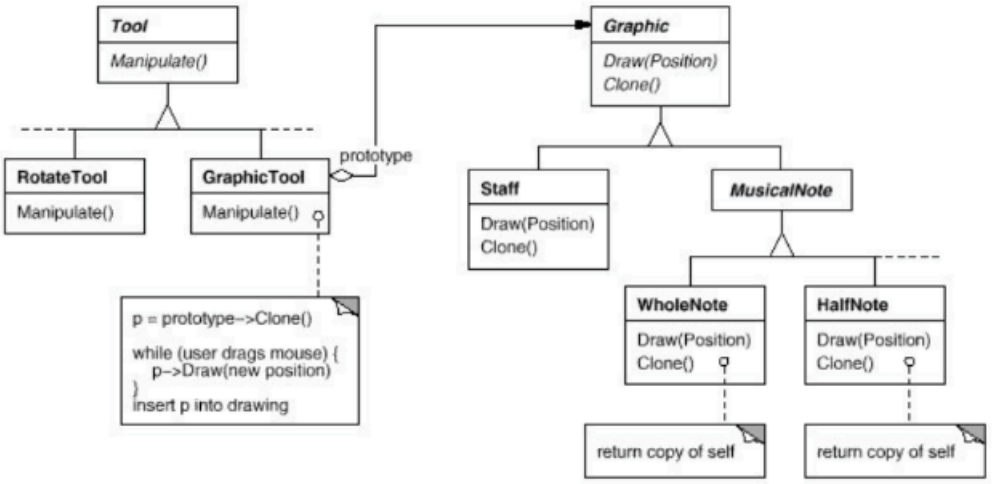
```
Product obj = prototype.clone()
```

Un **prototipo** deve avere un metodo `clone()` in modo tale che esso sia in grado di duplicarsi. Una volta duplicato il prototipo, le istanze che sono cloni possono essere configurate nei loro attributi per essere differenti l'una dall'altra.



Uso del pattern

Come negli altri pattern creazionali, vogliamo che il sistema sia indipendente dal prodotto concreto che utilizza, ma in più si ha la conoscenza della classe concreta **solo a runtime**, quindi è sconosciuta a tempo di design. Stabilire tale classe prima infatti stabilisce una dipendenza. In pratica, l'**oggetto a run-time funge da classe, affinché il client non dipenda da quella classe**.
Questo è utile in scenari in cui, ad esempio, il client conosce l'interfaccia dell'oggetto ma non conosce l'implementazione della classe; per conoscere quest'ultima, è necessario legare il client alla definizione della classe, creando una dipendenza.
Un altro vantaggio riguarda classi che hanno pochi stati possibili. Poiché abbiamo poche versioni di istanze dello stesso ADT, invece di definire più ADT, basta creare dei prototipi da clonare.
La clonazione può anche essere un processo **meno costoso**. Supponiamo ad esempio di avere la seguente situazione:



Se dovessimo disegnare delle note, ad esempio `WholeNote` o `HalfNote`, cambia solamente il glifo, ma non abbiamo bisogno di istanziare un oggetto per ogni glifo disegnato. Bastano solo due istanze da clonare quante volte voglio.
Vedere gli oggetti come prototipi permette che un oggetto configurato in modo diverso possa essere visto come una **classe**: si hanno prototipi dello stesso ADT ma diversi da loro in virtù del loro stato (i.e. configurazioni limitate degli attributi). Andiamo quindi oltre il concetto di ADT:

- l'ADT è un tipo astratto definito a tempo di compilazione;
- il prototipo è un "ADT" costruito a runtime, che non è prevedibile a tempo di compilazione e che non vogliamo sia specificato nella definizione software.

Il metodo `clone()` invocato su un oggetto può essere:

- una clonazione snella (**shallow clone**);
- una clonazione profonda (**deep clone**). L'implementazione di `clone()` in questo caso deve ricorsivamente clonare gli oggetti a cui è legato.

Clone in Java

In Java è implementata l'interfaccia `Cloneable`, interfaccia vuota che include il metodo `clone()`. Il metodo `clone()` crea una shallow copy, copiando valore per valore le variabili di istanza. Se i valori sono riferimenti ad altri oggetti, saranno allora gli stessi riferimenti di prima, quindi gli oggetti composti saranno condivisi. Per implementare una deep copy, bisogna eseguire un override del metodo `clone()`. Per poter invocare il metodo `clone()`, anche quello di default ereditato da `Object`, non è possibile invocarlo se prima non viene specificato che l'oggetto funzioni con il prototipo: la classe deve quindi implementare l'interfaccia `Cloneable`. Se non viene implementata l'interfaccia, si crea un'eccezione `CloneNotSupportedException`. Nel momento in cui utilizziamo il metodo `clone()` il nostro metodo deve avere la capacità di fare un raise dell'eccezione, quindi dobbiamo aggiungere `throws CloneNotSupportedException`.

Conseguenze del pattern

Per ipotesi gli oggetti con cui lavoriamo sono definiti a run-time. È più semplice quindi modificare gli oggetti senza modificare il client. Inoltre, è anche un modo per mantenere celata una struttura complessa all'interno.

Chiaramente il prototype pattern richiede l'implementazione del metodo `clone()`.

Prototype manager

Può essere comodo memorizzare i prototipi a cui i cloni fanno riferimento in una struttura, utilizzando ad esempio un simbolo per identificarli. In combinazione con il pattern prototype viene utilizzato il **prototype manger**.
Generalmente il manager viene implementato con una hash table, e con una funzione che preso in input una chiave restituisce il prototipo associato:

```
prototypeManager.get(key).clone()
```

Pattern prototype per la creazione di una abstract factory

```
class ProtoFactory {
    WindowPrototype window;
    MenuPrototype menu;
    ButtonPrototype button;

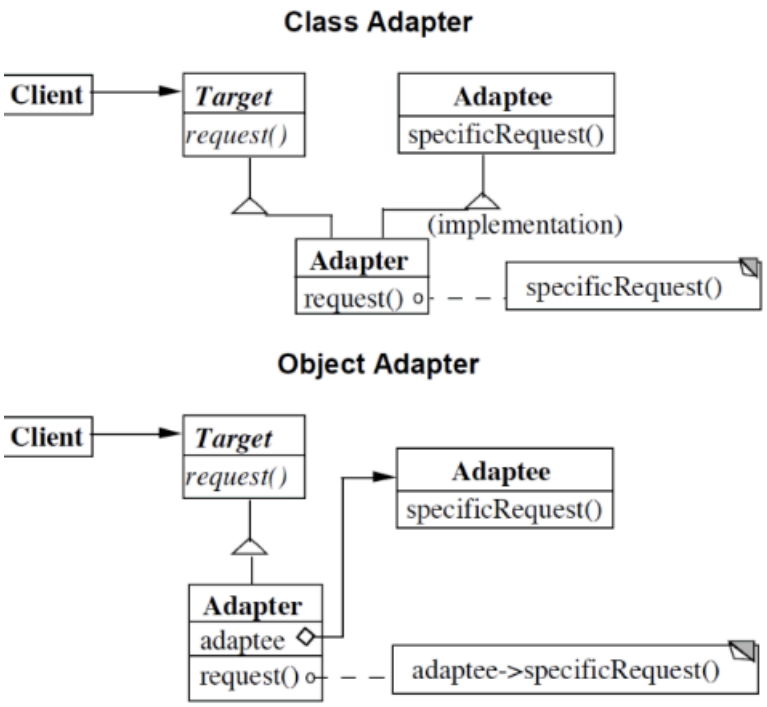
    public ProtoFactory(WindowPrototype wp, MenuPrototype mp, ButtonPrototye bp) {
        this.window = wp;
        this.menu = mp;
        this.button = bp;
    }

    public Window createWindow() { return (Window) window.clone(); }
    public Menu createMenu() { return (Menu) menu.clone(); }
    public Button createButton() { return (Button) button.clone(); }
}
```

Adapter Pattern

L'**adapter pattern** viene utilizzato per convertire l'interfaccia di una classe in un'altra interfaccia che i client attendono.
L'adapter esiste in due versioni:

- il **class adapter**, che consente l'override di parte dei metodi dell'adapter;
- l'object adapter, che adegua l'interfaccia di più classi a quella del client senza sottoclassare. Tuttavia, l'object adapter non sa quale sottoclasse specifica di Adaptee sta usando.



In foto:

- `Target` rappresenta l'interfaccia che il client si aspetta;
- `Adaptee` è un oggetto che necessita l'adattamento;
- `Adapter` converte la chiamata del client all'interfaccia della classe `Adaptee`.

Nella versione class adapter, `Adapter` è sia `Target` sia `Adaptee`. Nella versione object adapter, `Adapter` **compone** `Adaptee` dentro sé stessa.

Da notare che nel momento in cui abbiamo anche più versioni di `Adaptee` derivate da precedenti (quindi altri `Adaptee` che derivano l'originaria `Adaptee`), `Adapter` funge da convertitore anche per l'interfaccia di queste sottoclassi.

`Adapter` è una composizione di `Adaptee` in quanto `Adapter` esiste in virtù di `Adaptee`.

Funzionalità dell'Adapter

L'adapter funge da sostituto dell'oggetto che implementava `Target`, ma **non aggiunge funzionalità**. L'adapter fornisce funzionalità per un oggetto con un'interfaccia diversa, e teoricamente non deve aggiungere nuove funzionalità. Può tuttavia modificare il metodo chiamato occultandolo al client.

Facade Pattern

Supponiamo di avere diversi client che dipendono da uno stesso insieme di classi. Il numero di dipendenze può essere elevato; vogliamo che tale numero di dipendenze sia ridotto.
Possiamo dire che tale insieme di classi crea un **sottosistema** (e possibilmente appartengono ad uno stesso package). Possiamo allora inserire un'ulteriore classe, detta **classe facciata** (**Facade class**) che funge da interfaccia semplificata per i client. In questo modo i client non dipendono direttamente dall'insieme dei classi, ma semplicemente utilizzano la facade per eseguire le chiamate delle classi.

Il ruolo della classe facade non è banale: il facade non deve semplicemente raccogliere le interfacce, ma anche istanziare gli oggetti per il client. È quindi un **aggregatore** di classi del sottosistema.
Da notare che comunque il client è in grado di conoscere se esiste una classe nel sottosistema. L'unica cosa che abbiamo fatto è aggiungere un'interfaccia in modo tale da non avere una dipendenza specifica con le classi mascherate dalla facade.

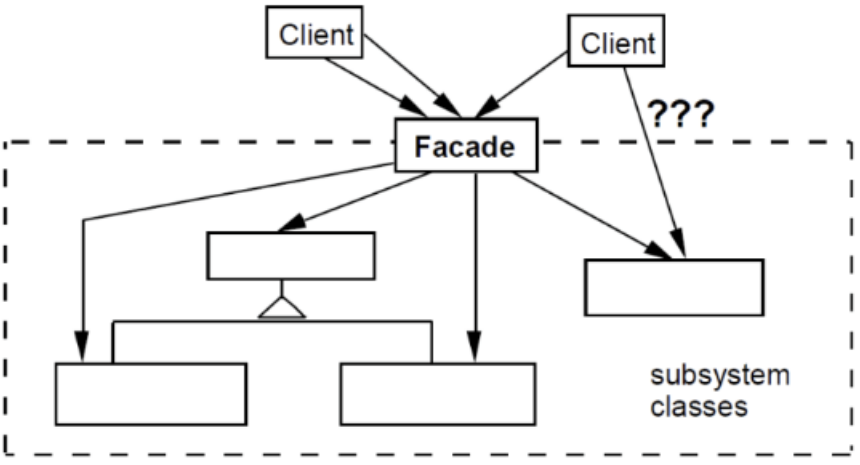
Funzionalità del facade

La classe facade **non deve aggiungere funzionalità** non presenti già nel sottosistema, in quanto è concettualmente sbagliato. Il facade non dev'essere un modulo software con una propria identità, ma deve solamente **semplificare l'accesso alle funzionalità delle classi che maschera**:

- può controllare condizioni per l'invocazione di una funzionalità;
- può verificare se il risultato di una certa elaborazione soddisfi requisiti;
- verificare che il client che invoca una funzione abbia diritto ad accedere ad alcuni metodi.

Il facade quindi può quindi rielaborare e comporre funzionalità, ma solo a fini di semplificare l'esecuzione e/o l'accesso ai metodi delle classi.

La classe facade, tuttavia, **non occulta la presenza delle classi che maschera**. Inoltre, ha la responsabilità, come anticipato precedentemente, di istanziare gli oggetti al posto del client. Ciò che fa il pattern è ridurre il numero di dipendenze all'interno del software.



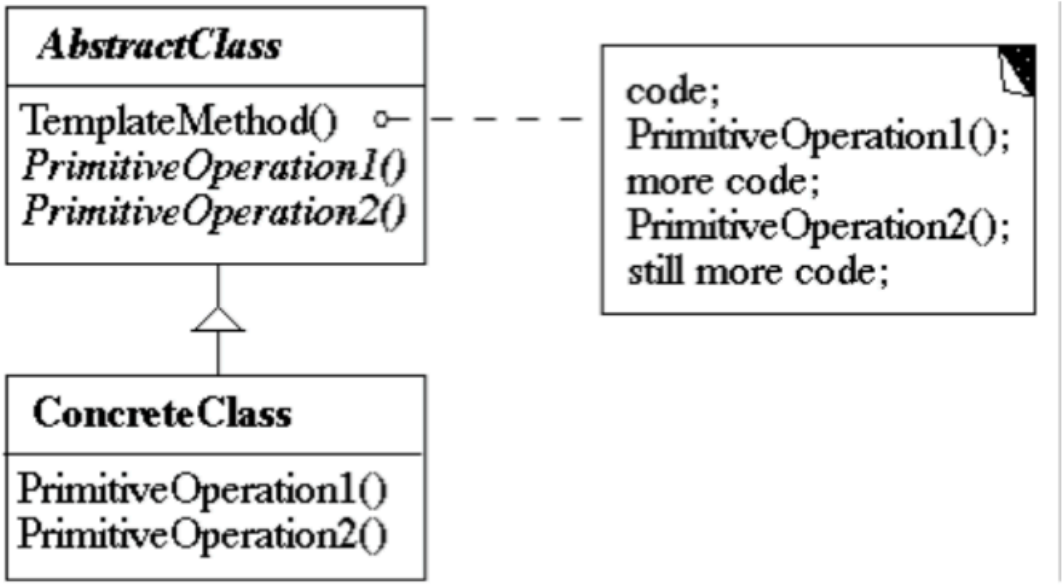
Il collegamento marcato con esplicita il fatto che, sebbene sia concettualmente **insensato**, è ancora possibile realizzare dipendenze di questo tipo.

10 - Template, Strategy, State

Template method

Sappiamo già che per ridefinire il comportamento di un metodo possiamo utilizzare il polimorfismo, ossia facendo un override del metodo. Tuttavia, questo approccio ha **granularità** relativa al metodo. Vogliamo stavolta però definire l'implementazione di un metodo, ma tralasciare il dettaglio specifico di uno o più passi interni all'implementazione del metodo nella sottoclasse. Definiamo quindi un ADT `AbstractClass` all'interno del quale è definita l'operazione `TemplateMethod()`. Dentro il `TemplateMethod()` sono definite delle operazioni simboliche `PrimitiveOperation()`. Queste operazioni primitive sono **astratte**. Le sottoclassi allora ereditano l'algoritmo ma solo nella sua struttura, ridefinendo le operazioni primitive.

Il metodo `TemplateMethod()` è un metodo ereditato dalla classe padre, ma richiamando all'interno della classe padre i metodi specificati nelle sottoclassi invertiamo la **struttura di controllo**: la classe padre esegue i metodi della sottoclasse.



\

Template method in Java

In Java il metodo template non dev'essere sostituibile dalle classi. Per far sì che un metodo ereditato non sia sostituibile in override, si dichiara un metodo `final`. Viceversa, i metodi che vogliamo che siano sostituibili nelle sottoclassi vengono dichiarati come `protected`.
Ricordiamo che ci sono quattro livelli di protezione in Java: `default`, `private`, `protected`, `public`. Il `protected` aggiunge visibilità del metodo alle sottoclassi anche se non appartengono allo stesso package.

```
public interface Algorithm {
    void partA();
    void partB();

    public final void algorithm() {
        partA();
        System.out.println("Invariable part");
        partB();
    }
}

public class MyAlg implements Algorithm {
    void partA() { System.out.println("My partA"); }
    void partB() { System.out.println("My partB"); }
}
```

Esempio: AWT

In Java esistono due gruppi di strumenti per le interfacce grafiche: AWT e la libreria swing. Nella libreria AWT il metodo `paint()` è un'operazione primitiva che è utilizzato nella costruzione del frame. `paint()` dev'essere ridefinito nelle nostre applicazioni.

Nella libreria swing invece sono presenti tre metodi `paint()`, più precisamente definiti in `javax.swing.JComponent` e invocati nel seguente ordine:

```
protected void paintComponent(Graphics g);
protected void paintBorder(Grapics g);
protected void paintChildren(Graphics g);
```

I metodi funzionano sempre seguendo la logica del template method e invertendo la struttura di controllo. Ridefiniamo tuttavia i tre metodi:

```
protected void paintComponent(Graphics g) {
    g.drawString("This is my custom Panel!", 10, 20);
    redSquare.paintSquare(g);
}
```

Inversione di controllo

Il meccanismo di inversione della struttura di controllo non è legato al template method ma è presente anche in altri pattern. Solitamente nel processo di **refactoring** è possibile introdurre dove può essere necessario il template method, individuando un possibile metodo in cui sono presenti tanti punti o blocchi condizionali multipli in cui una certa operazione può essere svolta in più modi. Si segue allora la logica:

1. si scrive l'intero codice in un grande metodo;
2. si divide in passi successivi usando commenti;
3. si incapsula ogni passo in un metodo separato;
4. si riscrive il template invocando i metodi estratti.

Si ripette poi successivamente il passo 1. fino a quando tutti i metodi primitivi hanno la stessa granularità; questi saranno dichiarati come `protected`. Il metodo template verrà poi definito come `final`.

Un caso d'uso del template method riguarda la **lazy initialization**, ossia l'"inizializzazione pigra". Una lazy initialization permette di ritardare l'inizializzazione dell'oggetto nel momento in cui è necessario ottenere l'istanza:

```
public class Foo {
    private Bar field;
    public Bar getField() {
        if (field == null) field = new Bar(10);
        return field;
    }
}
```

Per modificare il valore di default di inizializzazione di `field`, senza tuttavia **esporlo a qualsiasi valore** come parametro, possiamo applicare il template pattern sul metodo:

```
public class Foo {
    private Bar field;
    public Bar getField() {
        if (field == null) field = defaultField();
        return field;
    }
    protected Bar defaultField() {
        return new Bar(10);
    }
}
```

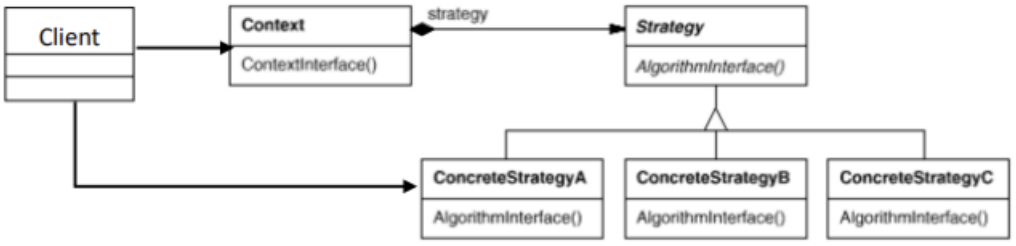
In questo modo eventuali sottoclassi di `Foo` possono ridefinire il comportamento di `defaultField()`, cambiando il valore di inizializzazione di `field`.

Vantaggi e svantaggi del pattern

L'applicazione del pattern permette di implementare le parti invarianti di un algoritmo una sola volta. Le parti uguali dell'algoritmo quindi possono essere scritte una sola volta, ridefinendo poi eventuali parti flessibili in virtù della sottoclasse. Abbiamo così la possibilità di **fattorizzare** la struttura dell'algoritmo, lasciando solo alcune parti interne modificabili.

Strategy

Anche in questo caso abbiamo un pattern comportamentale in cui abbiamo un algoritmo che può essere definito in modi differenti. Tuttavia, in questo caso introduciamo un `Context`, in cui è presente `ContextInterface()`. All'interno del `ContextInterface()` è presente un passo che può essere implementato in più modi, individuato con il metodo `AlgorithmInterface()`, presente all'interno di un oggetto `Strategy` esterno. `Strategy` è legato a `Context` con una composizione. Variamo allora una parte interna non perché sottoclassiamo `Context`, ma con una composizione di un oggetto `Strategy`.



Questo pattern non ha quindi un **class scope**, ma un **object scope**, in cui possiamo modificare l'istanza di `Strategy` presente in `Context`, e quindi ridefinire il comportamento in virtù di un oggetto che è intercambiabile. In questo caso abbiamo un algoritmo completo, ma all'interno di questo è presente un passo significativo che ne cambia il funzionamento.

Esempio: Java Layout Managers

All'interno del package AWT è presente un metodo per la gestione degli oggetti all'interno della finestra, in particolare per l'allineamento degli oggetti grafici. Esistono una serie di costanti statiche definite nella classe `FlowLayout`, che permettono di specificare un valore per indicare la variante di algoritmo da utilizzare:

```
class FlowExample extends Frame {
    public FlowExample(int width, int height) {
        setTitle("Flow Example");
        setSize(width, height);
        setLayout(new FlowLayout(FlowLayout.LEFT));
        //...
    }
}
```

Esempio 2: algoritmi di ordinamento

All'interno di `Collection` sono presenti algoritmi di ordinamento per l'aggregato. L'algoritmo di ordinamento necessita di un criterio di ordinamento. Possiamo allora dire che il confronto è una strategia da specificare in seguito. In questo caso non possiamo implementare questo con un template, quindi tramite sottoclassi legate a tempo di compilazione, ma utilizziamo la composizione.

Implementazione in java

Supponiamo di avere una situazione del tipo:

```
//...
switch(flag) {
    case A: doA(); break;
    case B: doB(); break;
    case C: doC(); break;
    //...
}
```

Invece di scrivere l'istruzione `switch`, possiamo allora definire un metodo `do()`, definendo un metodo astratto `do()` e delegando l'implementazione dell'applicazione della strategia ad una classe esterna. Quindi aggiungiamo **overhead** data la chiamata ad una

classe esterna, dovuto al fatto che contesto e strategia sono oggetti composti e distinti.
Potremmo effettivamente pensare di derivare semplicemente sottoclassi, come nel template method pattern. Tuttavia, questo potrebbe causare una **proliferazione eccessiva di classi**. Riconsideriamo l'esempio del package AWT. La classe `Panel` presenta circa 20 layout, e sono presenti circa 40 sottoclassi di `Component` che possono usarle. Tutte le possibili combinazioni tra i layout e le componenti sono circa 800: si dovrebbero derivare allora 800 classi. Chiaramente questo non è accettabile.

Interfaccia `Comparable`, `Comparator`

All'interno di Java il meccanismo strategy è utilizzato per l'interfaccia `Comparable`, che definisce un metodo `compareTo()` per restituire un'informazione su quale elemento è più grande di un altro. L'interfaccia `Comparable` è utilizzata per gli algoritmi di ordinamento, in quanto `compareTo()` viene utilizzato per definire quale oggetto è più grande di un altro. In questo caso notiamo che la `Strategy` è una **interfaccia**.

Da Java 8 sono state introdotte le **interfacce funzionali**: invece di avere oggetti che sono istanze di classi, possiamo anche istanziare metodi, definendo oggetti che sono in realtà riferimenti non a un'istanza di una classe, ma a un singolo metodo. Un esempio è l'interfaccia `Comparator`, diversa dall'interfaccia `Comparable`: la seconda si riferisce ad un attributo di un ADT, la prima invece è un'interfaccia funzionale che definisce un metodo `compare()` con due operandi risolvendo il problema del loro confronto. Le classi che definiscono solo il metodo di confronto, **che non sono legate staticamente ad ADT**, implementano quindi le interfacce funzionali. Inoltre, a differenza di `Comparable`, `Comparator` permette anche di definire confronti che non sono **naturali**, ossia confronti tra elementi di tipi diversi.
Il meccanismo quindi permette di definire oggetti che non sempre sono l'implementazione in memoria di un ADT, ma semplicemente l'implementazione di un metodo. Possiamo allora definire dei `Comparator` utilizzando **espressioni Lambda**:

```
Comparator<Player> byRanking = (Player player1, Player player2) -> Integer.compare(player1.getRanking(),
player2.getRanking())
```

Questa implementazione è alternativa quindi a `Comparable`.

`Comparator` è anche dotato di un metodo **statico** `Comparing(mr)`, che prende in input una lambda expression oppure un riferimento ad un metodo e restituisce in output l'implementazione corrispondente di `Comparator`:

```
Comparator<Player> <byRanking> = Comparator.comparing(Player::getRanking);
```

L'interfaccia `Comparable`, in conclusione, è un'ottima scelta per definire l'ordinamento naturale, quindi il modo principale per confrontare gli elementi. Ci sono diversi motivi per cui utilizziamo anche `Comparator`:

- risulta impossibile di modificare il codice sorgente della classe che implementa l'interfaccia `Comparable`;
- evitiamo di appesantire la nostra classe principale;
- definiamo molteplici strategie di confronto, cosa non possibile utilizzando solo l'interfaccia `Comparable`.

Esempio

Supponiamo di avere la classe:

```
public class Player {
    private int ranking;
    private String name;
    private int age;

    // ...
}
```

Vogliamo ordinare un `ArrayList` di `Player`:

```
public static void main(String[] args) {
    List<Player> footballTeam = new ArrayList<>();
    Player player1 = new Player(59, "John", 20);
    Player player2 = new Player(67, "Roger", 22);
    Player player3 = new Player(45, "Steven", 24);
    footballTeam.add(player1);
    footballTeam.add(player2);
```

```
    footballTeam.add(player3);

    System.out.println("Before Sorting : " + footballTeam);
    Collections.sort(footballTeam);
    System.out.println("After Sorting : " + footballTeam);
}
```

Chiaramente otteniamo l'errore a tempo di compilazione:

```
The method sort(List<T>) in the type Collections is not applicable for the arguments (ArrayList<Player>)
```

Pertanto modifichiamo la classe `Player`, implementando l'interfaccia `Comparable` e definendo come metodo di confronto tra giocatori il ranking:

```
public class Player implements Comparable<Player> {
    private int ranking;
    private String name;
    private int age;

    //...
    @Override
    public int compareTo(Player otherPlayer) {
        return Integer.compare(getRanking(), otherPlayer.getRanking());
    }

}
```

Adesso se eseguiamo il codice otteniamo:

```
Before Sorting : [John, Roger, Steven]
After Sorting : [Steven, John, Roger]
```

Supponiamo ora di voler ordinare la collezione in base all'età. Allora definiamo il comparatore:

```
public class PlayerAgeComparator implements Comparator<Player> {
    @Override
    public int compare(Player firstPlayer, Player secondPlayer) {
        return Integer.compare(firstPlayer.getAge(), secondPlayer.getAge());
    }
}
```

Istanziamo un oggetto di tipo `PlayerAgeComparator` e passiamo il comparatore al metodo `sort()`, come se stessimo passando un puntatore a funzione in C++ per il sorting di una struttura:

```
PlayerAgeComparator playerComparator = new PlayerAgeComparator();
Collections.sort(footballTeam, playerComparator);
```

E adesso vedremo:

```
Before Sorting : [John, Roger, Steven]
After Sorting : [Roger, John, Steven]
```

Alternativamente possiamo definire il `Comparator` nei seguenti modi:

```
Comparator byAge = (Player firstPlayer, Player secondPlayer) -> Integer.compare(firstPlayer.getAge(),
secondPlayer.getAge());
// oppure
Comparator<Player> byAge = Comparator.comparing(Player::getAge);
```

Applicabilità del pattern

Lo strategy pattern è particolarmente utile quando abbiamo versioni diverse di uno stesso algoritmo, permettendo anche la definizione di versioni future dell'algoritmo.

Un aspetto importante del pattern riguarda il contesto. `Context` invoca una funzionalità di `Strategy`, che è una classe esterna. `Strategy` tuttavia dovrebbe accedere ad informazioni che conosce il `Context`: ad esempio, nel caso dell'algoritmo di ordinamento `Strategy` deve conoscere gli elementi per confrontarli. Si può allora utilizzare il meccanismo di **inner class**, definendo la classe `Strategy` all'interno della classe `Context`.

Sia nel template sia nello strategy pattern, l'idea è sempre simile: abbiamo un algoritmo e vogliamo ridefinire il comportamento di alcune parti. Il client dei pattern ignorano quale variante è stata utilizzata, ma è il client che ha la possibilità di determinare quale variante utilizzare. Tuttavia:

- nel template pattern la variante è un ADT indicato nel contesto d'uso a tempo di compilazione. L'obiettivo è lasciare un passaggio aperto a varie possibili implementazioni, creando vari ADT;
- nello strategy pattern si definisce una famiglia di oggetti intercambiabili che implementano varianti interne dell'algoritmo, applicandole ad un contesto di alto livello e indicandole a tempo di esecuzione.

Varianti di implementazione

Possiamo definire in due modi come strategia e contesto interagiscono:

- il contesto può passare i dati alla strategia, come parametro;
- la strategia ha accesso privilegiato al contesto.

Possiamo implementare le strategie da passare come parametri ai contesti, come:

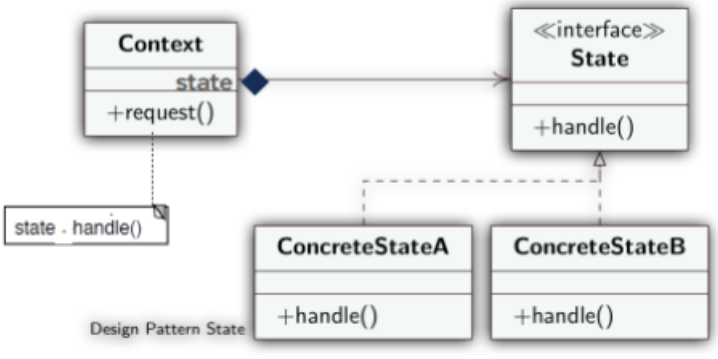
- istanze oggetti, tramite dependency injection se vogliamo cambiarle a runtime;
- usando le generics, a patto che il tipo sia riconosciuto a tempo di compilazione;
- usando le funzioni lambda che implementano i Comparator. In generale, Java Streams è un'applicazione più generale del pattern.

State pattern

La struttura dello state pattern sembra molto simile al pattern strategy. Tuttavia, qui il focus non è l'algoritmo. `Context` ha un metodo pubblico `request()` ed è dotato di uno **stato**. Per ipotesi, il `Context` è un oggetto con stato, quindi in questo caso ha senso fare un diagramma degli stati. Il metodo `request()` quindi non è sempre lo stesso: `request()` può usare gli attributi dell'ADT `Context`, quindi lo stato del `Context` per decidere come procedere.

`request()` rappresenta un algoritmo complesso il cui comportamento dipende dallo stato del contesto. Possiamo allora inserire ogni ramo condizionale in una classe separata `ConcreteState`, tutte implementazioni di un'interfaccia `State` dotata di un metodo `handle()`. La differenza quindi risiede nel fatto che non abbiamo un oggetto in cui abbiamo un algoritmo che presenta varianti: abbiamo una funzionalità interna, la cui implementazione dipende dallo stato interno che renderebbe complicata l'implementazione. Spostiamo allora la complessità alle classi `ConcreteState`.

`Context` allora opera come un **automa a stati finiti**. Il Client invoca richieste su `Context`, che a sua volta invoca `handle()` su `ConcreteState` per operare in virtù dello stato di `Context`.



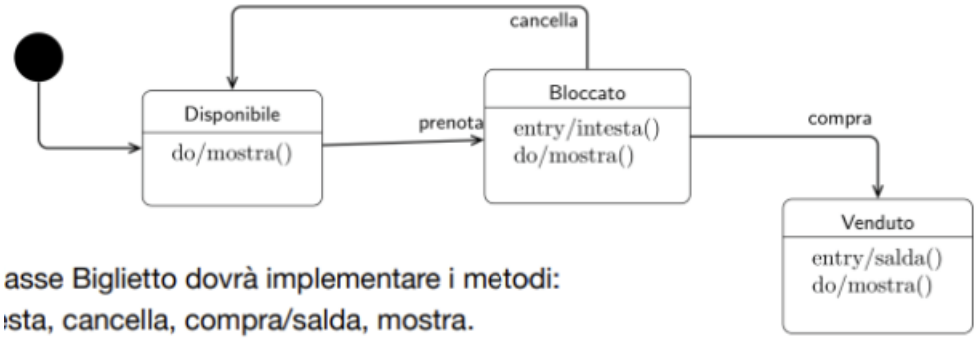
State vs Strategy

Anche lo state pattern ha **object scope**, ma cambiano frequenza e visibilità dei cambi. Per distinguere state pattern e strategy pattern, utilizziamo queste due informazioni:

- nello strategy applico al contesto **una sola strategia** per ogni ciclo di vita. Nello state, il contesto **muta più volte lo stato concreto** nello stesso ciclo di vita;
- nello strategy pattern tutte le strategie fanno la stessa cosa, ma in maniera diversa: per i client non fa alcuna differenza visibile se cambia il risultato della funzionalità invocata sul contesto. Nello state, gli stati concreti producono **risultati visibilmente differenti nel client**, vedendo il contesto reagire in virtù dello stato.

Esempio: biglietteria

Supponiamo di avere il seguente sistema:



Ogni operazione deve controllare in che stato si trova il biglietto, pertanto avremmo del codice del tipo:

```
public void prenota(String s) {
    switch(stato) {
        case DISP:
            //...
            break;
        case BLOC:
            //...
            break;
            //...
    }
}
```

Una tecnica di refactoring del codice è la **Replace conditional with Polimorphism**, ma in questo caso poiché le azioni dipendono dallo stato del biglietto, facciamo un refactoring utilizzando lo state pattern. Spostiamo la logica degli stati in nuove classi, utilizzando l'interfaccia `StatoBiglietto`. L'implementazione si semplifica in maniera enorme:

```
public class Biglietto {
    private String codice = "XYZ";
    private int prezzo = 100;

    private StatoBiglietto sb = new Disponibile();

    //...

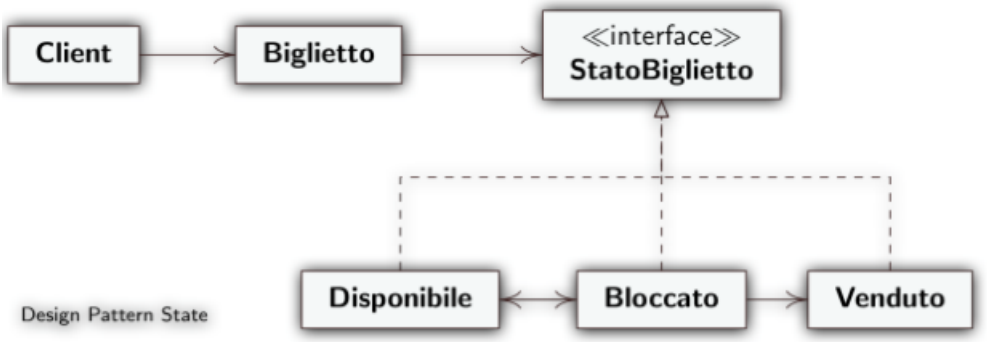
}

public class Disponibile implements StatoBiglietto {
    @Override public void mostra() {//...

    }
    @Override //..

}
```

L'automa non è più concentrato nel contesto, ma **distribuito negli stati**: ogni stato restituisce qual è il prossimo stato da utilizzare. Lo schema UML diventa quindi:



Implementare le transizioni

Implementare le transizioni tramite lo stato permette di dare molta flessibilità a run-time, rendendo sconosciuto al contesto la conoscenza degli stati.
Se invece il contesto è responsabile delle transazioni dello stato, gli stati invece sono più semplici, cablando la logica dell'automa nel contesto. Il vantaggio inoltre risiede nella possibilità di leggere la logica dell'automa nel contesto di utilizzo. Inoltre, poiché gli stati sono semplici, questi possono anche essere utilizzati in più automi.

Se gli stati sono senza attributi di istanza, e quindi possono essere condivisi o riusati, allora possiamo utilizzare:

- il pattern singleton per crearli una sola volta;
- il refactoring di un concrete state affinché non si abbiano più variabili di istanza, trasportando le variabili nel contesto.

11 - Composite pattern

Composite pattern

Supponiamo di voler manipolare delle strutture di tipo **composito**: alcuni elementi sono aggregatori di altri elementi. Gli elementi all'interno non sono necessariamente omogenei, tipico per le interfacce grafiche. Tipicamente abbiamo visto librerie per aggregati di tipo omogeneo, mentre per un'interfaccia grafica si ha un contenitore principale, ossia la finestra di applicazione, e molti altri oggetti che non sono necessariamente simili alla finestra: pulsanti, menu e aree di testo sono solo esempi di questi oggetti.
Non si tratta di definire una classe per definire una finestra di applicazione, ma è qualcosa che può essere creata in mille modi diversi. Lo scopo del **composite pattern** non riguarda la creazione della struttura, ma di facilitarne l'uso: chi usa la struttura non deve far condizionare l'algoritmo alla complessità della struttura, ma bisogna poter **vedere l'intera struttura come se fosse un singolo oggetto**.

Un modo in cui può essere risolto il problema è di strutturare la classe `Window` come un ADT contenente un array di pulsanti, menu, aree di testo e container di widget:

```
class Window {
    Buttons[] myButtons;
    Menus[] myMenus;
    TextAreas[] myTextAreas;
    WidgetContainer[] myContainers;

    public void update() {
        if (myButtons != null)
            for (int k = 0; k < myButtons.length(); k++) myButtons[k].refresh();
        if (myMenus != null)
            for (int k = 0; k < myMenus.length(); k++) myMenus[k].display();
        if (myTextAreas != null)
            for (int k = 0; k < myTextAreas.length(); k++) myTextAreas[k].refresh();
        if (myContainers != null)
            for (int k = 0; k < myContainers.length(); k++) myContainers[k].updateElements();

    }
}
```

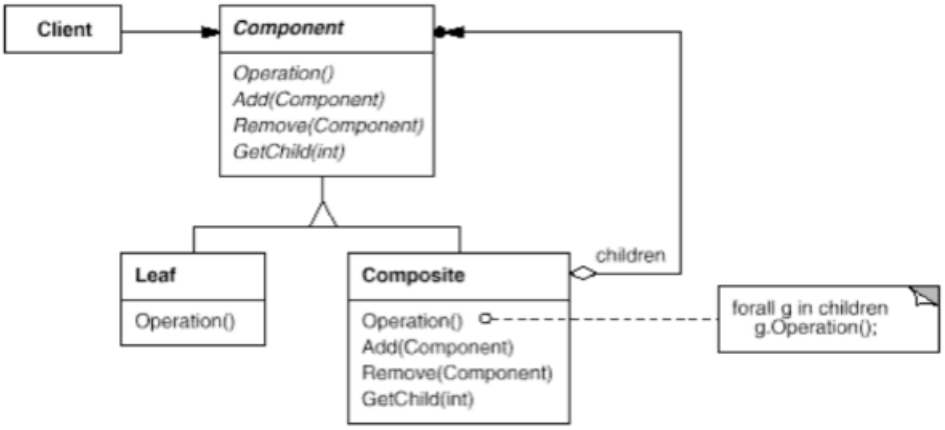
Il problema di questo approccio è che l'aggiunta di un array di elementi grafici in più richiede l'aggiunta di un nuovo ciclo `for`. Si eseguono quindi n cicli, uno per tipologia di oggetto. La composizione è limitata a tempo di compilazione. Ci sono quindi tante dipendenze.

Possiamo tuttavia utilizzare il polimorfismo, e definire un'interfaccia usata da `Buttons`, `Menus`, `TextAreas` chiamata `GUIWidget`:

```
class Window {
    GUIWidgets[] myWidgets
    WidgetContainer[] myContainers;

    public void update() {
        if (myWidgets != null)
            for (int k = 0; k < myButtons.length(); k++) myWidgets[k].update();
        if (myContainers != null)
            for (int k = 0; k < myContainers.length(); k++) myContainers[k].updateElements();
    }
}
```

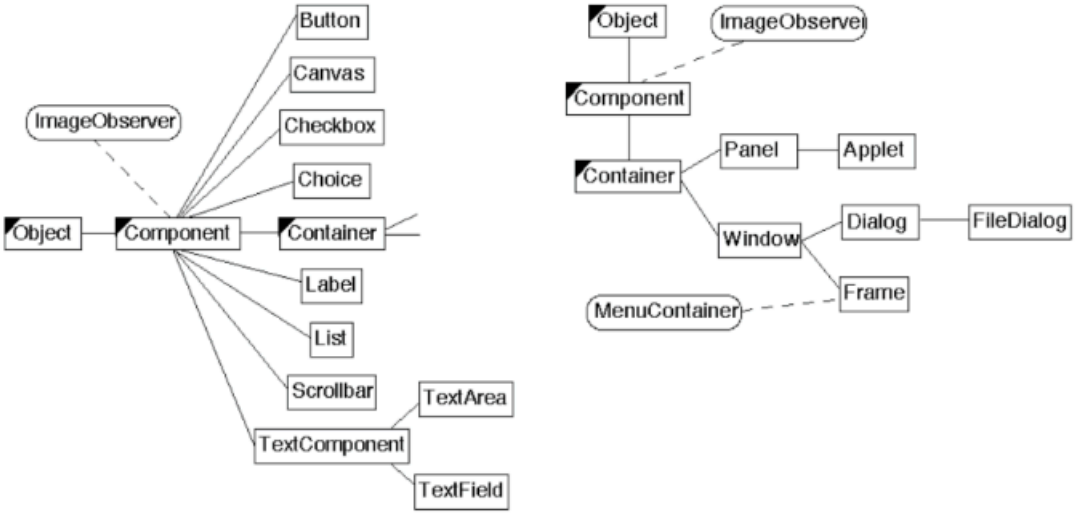
Ancora una volta, tuttavia, distinguiamo il contenitore dall'oggetto semplice. Il mio client quindi distingue un aggregato da un oggetto più semplice.
Possiamo considerare `WidgetContainer` come una speciale foglia all'interno della nostra gerarchia. D'altronde, `WidgetContainer` è pur sempre un elemento GUI, ma in più ha soltanto la capacità di contenere altri widget. Definiamo allora **un'unica interfaccia** `Component` comune a tutte le classi. Se l'oggetto è semplice, l'operazione del component è implementata. Altrimenti, se la struttura è aggregata questa sarà un aggregato di oggetti che implementano `Component` e che chiama ricorsivamente su tutti i componenti la procedura da implementare. Lo schema del pattern è il seguente:



La `Operation()` di `Composite` deve quindi ciclare ricorsivamente per eseguire le `Operation()` degli oggetti di cui è aggregato. Una cosa interessante da notare è la **ricorsione object-oriented**: se il client ha accesso alla radice della struttura, l'invocazione di `Operation()` sulla radice della struttura la richiama su tutti gli elementi della struttura. Inoltre, le chiamate possono avvenire in due modi: o dal basso, se prima si ha la chiamata ricorsiva object-oriented e poi l'effettiva operazione per la radice della struttura, o dall'alto, se l'ordine precedente è inverso.

Esempio: AWT

Il pattern Composite è ampiamente impiegato all'interno delle librerie AWT, definendo la struttura nel seguente modo:



Precisazione

Chiaramente se la classe `Composite` implementa una sua `Operation()` e viene estesa, la ridefinizione di `Operation()` richiede una invocazione di `super.Operation()`.
Nella classe `Composite` inoltre si definisce un aggregato di `Component` che è generico per tutte le sottoclassi.

Interfaccia stream

L'interfaccia stream viene utilizzata per interagire con gli aggregati. Gli stream non hanno necessariamente un aggregato di tipo sequenziale: una `Collection` infatti è iterabile, definendo un ordine implicito, ma nello stream l'implementazione dell'iterazione non è un ciclo. Lo stream quindi può eseguire iterazioni in parallelo ed operazioni sugli elementi. Lo stream definisce tre operazioni:

- `map()`, che fa corrispondere ad un elemento un altro;
- `reduce()`, che riduce l'aggregato di elementi;
- `filter()`, che filtra gli elementi da selezionare.

Conseguenze del pattern

Un client fa riferimento ad un'unica interfaccia, e può aspettarsi un elemento semplice ma riceverne uno composto. Non si vincola a design time il `Composite` all'utilizzo di alcuni tipi di Leaf ma si possono solo eseguire controlli a runtime. Infine, i tipi di foglie non devono per forza essere parenti tra loro, in quanto si sfruttano legami di composizione, per cui il pattern è **object scope**.
Bisogna inoltre definire le operazioni di `add()` e `remove()` correttamente: in teoria andrebbero definiti in `Component`. Tuttavia,

questo espone il codice ad **errori a run-time**: un `add()` su un `Leaf Component` non ha senso. Pertanto si dovrebbe gestire utilizzando una eccezione. Se invece si evita l'implementazione di `add()` e `remove()` su `Component` ma solo sul `Composite`, si può segnalare un errore in tempo di compilazione, ma questo annulla **trasparenza**.
Per navigare la struttura gerarchica si potrebbe prevedere nei `Composite` un `getParent()` in modo tale da poter risalire al contrario nella struttura. Se la struttura è complicata, si potrebbe assegnare un ruolo di **proxy** alle leaf. Il `Composite` implementa una **cache** per ridurre quindi le ricerche di valori nei child. Tuttavia, i child devono avere un metodo `getComposite()` per accedere e invalidare la cache del `Composite`.

Visitor pattern

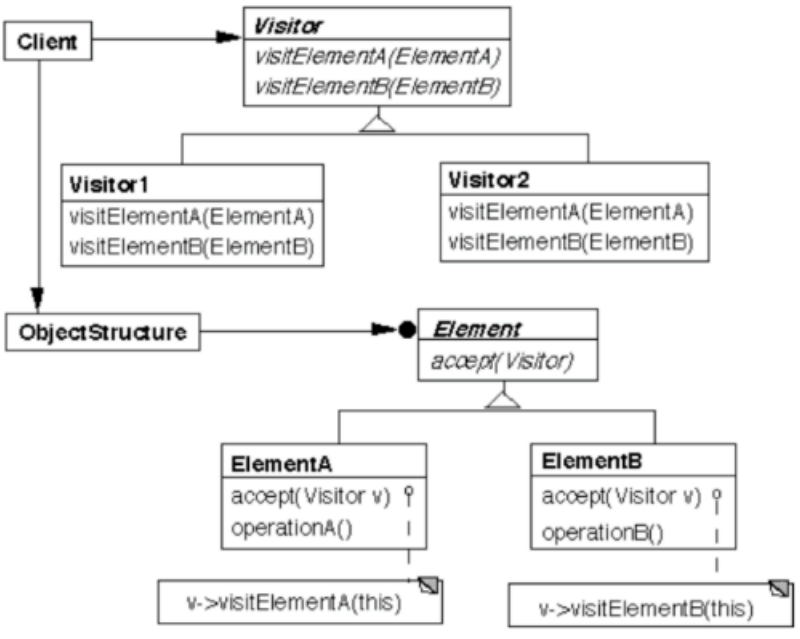
Supponiamo di avere una struttura composita eterogenea. Supponiamo di voler applicare delle funzionalità alla struttura. Sappiamo che è possibile usare il Composite pattern per creare una struttura composita espandibile e facilmente manipolabile. Le funzionalità di manipolazione erano definite nell'interfaccia `Component`, ma se volessi, ad esempio, avere molte funzionalità nella struttura composita che sono scorrelate tra loro, si ha un **affollamento** negli ADT di funzionalità che sono opzionali. E se volessi aggiungere delle funzionalità in un secondo momento, come si potrebbe fare?

Invece di utilizzare un pattern, potremmo prendere un ADT e sottoclassarlo quando è necessario aggiungere funzionalità. In questo modo il tipo di base viene derivato in più varianti, ognuna delle quali si arricchisce di una funzionalità in più. Ma in questo modo non si riesce a combinare più di due funzionalità assieme in una sola classe: infatti ad esempio in Java non si ha l'**ereditarietà multipla**. Si implementa allora il **pattern Visitor**. Il pattern utilizza la composizione, inserendo le funzionalità aggiuntive in un oggetto `Visitor` che incapsula queste. L'ADT creato è un ADT **degenere**, che incapsula solo funzionalità per operare su altri oggetti. L'idea del meccanismo è di avere un oggetto che attraversa la struttura; il visitor viene passato a ciascun elemento della struttura, il quale invoca la funzionalità del visitor.

Abbiamo tre possibilità per visitare la struttura:

1. ogni elemento sa chi è il suo successore nella visita, come se si trattasse di una lista concatenata. L'elemento quindi riceve il visitor, applica il visitor a se stesso, e successivamente applica il visitor al suo successivo;
2. si utilizza un **Iterator**;
3. il visitor sa come navigare la struttura.

Il caso generale è che la struttura supporta la visita, ma non decide l'ordine della navigazione. Non è neanche il visitor a deciderlo, ma è qualcosa di esterno sia al visitor sia alla struttura, come un iteratore. Affinché il visitor esegua la sua operazione sull'elemento si utilizza un **double dispatch**. Si suppone che ogni elemento abbia la possibilità di accettare un elemento di tipo `Visitor` dall'esterno. A sua volta il `Visitor` ha un'interfaccia con un metodo standard al suo interno `visit()`. L'elemento non conosce la funzionalità `visit()`, e passa alla funzionalità se stesso, in modo tale da dare la possibilità al visitor di accedere all'elemento. Si ha quindi una doppia invocazione: si invoca un metodo sull'oggetto per accettare il visitor, che a sua volta invoca sul visitor un metodo per eseguire la visita. Lo schema del pattern è il seguente:



Il client dipende da una `ObjectStructure`, il quale è un aggregato (indicato con un pallino, si indica chiaramente con un rombo vuoto) di `Element`, interfaccia che contiene il metodo `accept(visitor)` e che **rimbalza la chiamata a** `Visitor`. Il client fa riferimento ad un `Visitor`, un'interfaccia che supporta le visite ad elementi (in foto `visitElementA(ElementA)`,

`visitElementB(ElementB)`). Chiaramente la visita deve supportare le visite ad ogni tipo di oggetto presente nella `ObjectStructure` . Il metodo anonimo `visitElement()` può corrispondere a diverse funzionalità di visita, in virtù della implementazione di `Visitor` . Ogni implementazione concreta di `Visitor` definisce una funzionalità di visita della struttura (somma degli elementi, stampa in HTML, ecc.).
L'accesso ai dati, inoltre, potrebbe non essere sempre possibile dall'esterno. Per far sì che il visitor possa accedere ai dati esso può essere definito come inner class o può ricevere in input l'oggetto da visitare.

Esempio

Supponiamo di avere un `Visitor` per un albero binario di ricerca. Allora la visita di un elemento opera nel seguente modo:

```
Node example = new BinaryTreeNode();
Visitor traveler = new HTMLVisitor();
example.accept(traveler);

// all'interno di accept(traveler)
//...
traveler.visitBinaryTreeNode(this);
```

Limitazioni

Notare che tale tipo di legame è un tipo di legame **statico**. Il visitor quindi non è espandibile. Pur essendo un pattern basato sugli oggetti, è un pattern che si può applicare solamente a tempo di compilazione. L'interfaccia che viene data ai visitor è strettamente legata ai tipi degli elementi che sono presenti nella struttura.

Interfacca `Consumer`

Il pattern si ritrova nell'interfaccia funzionale `Consumer` . `Consumer<T>` è un'interfaccia che ha un solo metodo: `void accept(T t)` . I metodi che implementano il metodo `accept()` dell'interfaccia Consumer sono automaticamente manipolabili come **oggetti di prima classe**. Si possono quindi passare metodi a variabili o come valori attuali ai parametri di altri metodi. I metodi che implementano interfacce funzionali diventano oggetti passabili come parametri. Il pattern `Visitor` passa la funzionalità di un oggetto ad un altro oggetto per applicare la visita. Se si può definire un algoritmo come un oggetto, non si ha la necessità di definire un algoritmo per contenerlo.
Il metodo `forEach()` richiede come parametro un oggetto che implementa l'interfaccia `Consumer<T>` . Ogni metodo o oggetto che implementa `Consumer<T>` può quindi essere passato nel `forEach()` . Lo strumento delle interfacce funzionali permette quindi a un qualsiasi metodo di definire implementazioni di interfacce a runtime sotto la forma di espressioni lambda.

`accept()` non restituisce un valore, per cui o produce una modifica di stato di un oggetto, o produce un **side-effect** (la sua applicazione ha lasciato un residuo), oppure modifica lo stato interno dell'oggetto che implementa `accept()` . Nel caso del visitor, `accept()` può modificare lo stato interno del visitor.

Nelle `Collection` il `Consumer` viene utilizzato assieme ad un Iterator. Con gli stream si ha sempre il metodo `forEach` analizzato precedentemente. Anche negli aggregati iterabili esiste `forEach()` , che non è altro che un metodo `accept()` .

Applicabilità del pattern

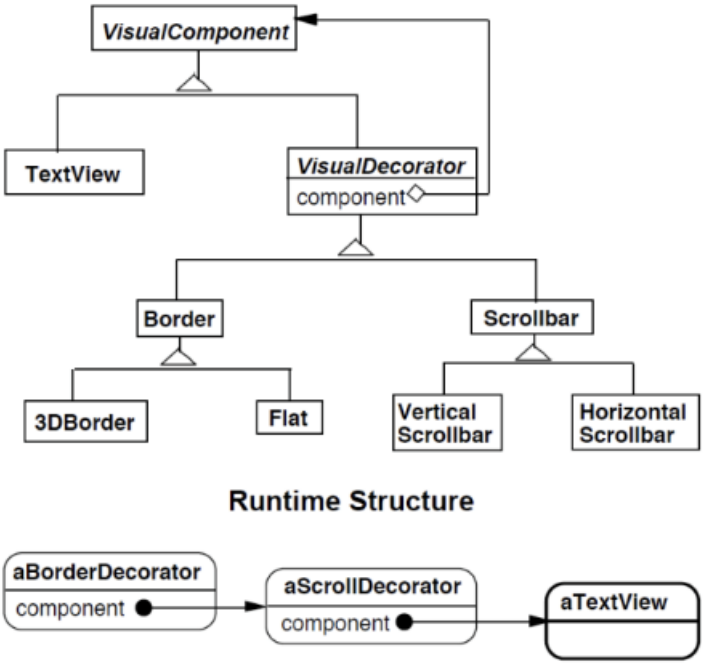
Generalmente il pattern viene applicato quando una struttura di oggetti ha molte classi con interfacce diverse e si desidera eseguire operazioni dipendenti dalle classi concrete. Si tratta di un pattern ideale anche quando bisogna eseguire operazioni distinte non correlate sugli oggetti in una struttura, evitando così di sovraccaricare le classi con tali operazioni; se le operazioni sono tante, risulta più semplice la gestione del codice. Infine, se le classi della struttura cambiano raramente ma si aggiungono nuove funzionalità: qui non vogliamo variare la struttura, ma le funzionalità da applicare a questa.
Chiaramente se aggiungiamo un nuovo elemento, dobbiamo modificare tutti i `Visitor` .

Pattern Decorator

Il pattern Decorator è un pattern strutturale che lavora sugli oggetti.
Supponiamo che un oggetto debba implementare una operazione composta da un numero molteplice e variabile di singole responsabilità. Ad esempio, abbiamo una finestra di testo che può avere opzionalmente una barra di scorriment verticale, una orizzontale, un bordo 3D o un bordo piatto. Abbiamo quindi una serie di accessori alla funzionalità base, ossia il rettangolo che contiene testo, che possiamo di volta in volta mettere o non mettere. Se ponessi tutte queste capacità dentro l'oggetto che rappresenta la finestra di testo, l'oggetto si affolla. Se l'oggetto è inoltre posizionato in alto in una gerarchia di oggetti, anche gli oggetti che si trovano in basso si affollano.

Una soluzione potrebbe essere quella di derivare una sottoclasse per ogni possibile combinazione di opzioni. Ma poiché ci pososno essere molte combinazioni, la soluzione non è accettabile. Utilizziamo allora una interfaccia generale `TextView`, che è dotata di un bordo, una barra di scorrimento orizzontale e una barra di scorrimento verticale. Tuttavia, aggiungere una opzione richiede la modifica di tutte le classi.

La soluzione risiede nel definire l'interfaccia `Component`, implementata da un componente base e da un `Decorator`. Nel `Decorator` si ha un riferimento a un componente successivo, che può essere un altro decorator o il componente base. Tutte le tipologie di opzioni e di funzionalità sono delle sottoclassi di `Decorator`. Il pattern permette di **aggiungere dinamicamente funzionalità** ad un oggetto a run-time:

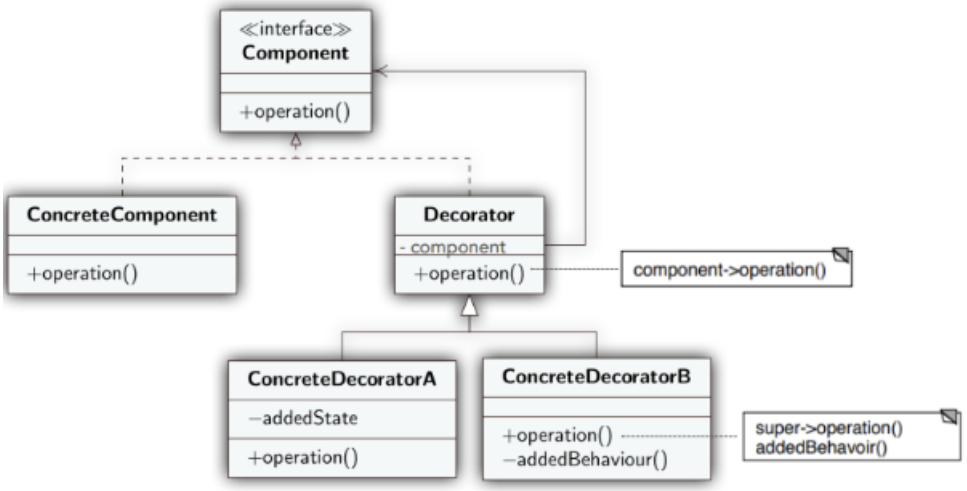


I decorator quindi aggiungono delle funzionalità in più ad un oggetto base. Si accede alle funzionalità dell'intera struttura attraverso un oggetto di accesso. Quindi nel nostro esempio, il client invocherà la funzionalità sul decoratore relativo al bordo, che invocherà le restanti funzionalità. Si tratta anche in questo caso di **ricorsione object-oriented**, quindi anche in questo caso si può decidere se utilizzare le funzionalità a partire dalla "coda" della ricorsione o dalla testa.

Applicabilità

Applicare il decorator ci permette di modificare il comportamento dinamicamente. L'oggetto che si crea non è solo un oggetto base, ma una struttura arricchita ad ogni passo. Il decorator funge da **wrapper**, avvolgendo quindi la componente base in un altro oggetto per arricchirne funzionalità.

Lo schema del pattern è il seguente:



All'interno di `Decorator` è presente un `Component` privato. Solo `ConcreteComponent` non ha un successore ma solo l'operazione. L'operazione di base nei decorator invoca la `operation()` del suo successore.
Da notare che i `ConcreteDecorator` possono modificare le loro implementazioni di `operation()`, ad esempio creando dei metodi non presenti nel `Decorator` o aggiungendo degli stati.

Una volta aver creato la struttura, essa può essere modificata a **run-time** modificando la catena di istanza. Questo può essere utile in combinazione con il **Prototype pattern**.

Si potrebbe fare un parallelismo con il pattern State. Anche in questo caso si può dire che con il Decorator il client si ritrova un Decorator che cambia tipo in base a quanti Decorator vengono aggiunti. La differenza è che nel Decorator il client vede l'oggetto cambiare tipo, mentre con lo state il cambio di oggetto è un cambio di stato interno. Inoltre, con il decorator vengono aggiunte o sottratte funzionalità, mentre con lo state si **sostituisce interamente** la funzionalità in virtù dello stato. Infine, mentre nel dacorator le istanze vengono aggiunte solo quando è necessario, nello state è necessario avere tutti gli stati per un funzionamento corretto del programma.

12 - Observer pattern e programmazione reattiva

Observer

L'observer pattern è un pattern comportamentale utilizzato per implementare relazioni tra oggetti.
Il pattern è utile quando è ricorrente nel nostro software una situazione in cui dei moduli distinti hanno tra loro relazioni di dipendenza **comportamentale** (ad esempio, se un modulo A cambia stato nel modulo B si ha una reazione): ad un certo cambiamento di stato corrisponde l'attivazione di un comportamento. Tale relazione può anche non riguardare solo due moduli, ma in generale n moduli: si può avere in generale un modulo che cambia stato e n moduli che dipendono da questo cambiamento di stato (se il cambiamento di stato occorre, devono automaticamente reagire attivando delle loro funzionalità). La classe il cui cambiamento di stato è rilevante prende il nome di **subject** in quanto è l'elemento osservato, mentre le classi che agiscono in virtù dei cambiamenti del subject prendono il nome di **observer**. Per quanto detto, ogni observer può avere una sua differente reazione allo stesso cambiamento di stato.

Vogliamo a tempo di esecuzione:

- **modificare il numero di observer a tempo di esecuzione**;
- **modificare il tipo di observer a tempo di esecuzione**.

Vogliamo quindi che la dipendenza dai cambiamenti di stato non richieda dei **legami statici** tra questi oggetti. In maniera ingenua questo pattern potrebbe essere implementato utilizzando dei riferimenti agli observer all'interno della classe subject. Il subject quindi si prende carico di attivare i meccanismi di corrispondenza azione-reazione. Questo approccio tuttavia non permette di cambiare il numero di observer, il tipo di questi e anche il comportamento per ciascun observer, in quanto la reazione è cablata nel metodo. I tipi, inoltre, devono essere conosciuti, quindi non è possibile aggiungere tipi a runtime.
Come conseguenza dell'applicazione dell'observer pattern, se un subject cambia stato e altri n oggetti reagiscono al cambiamento di stato, si riesce a far sì che ci sia una coerenza di stato tra subject e observer. Se gli observer possono conoscere il cambiamento di stato nel subject, è possibile anche fornire all'observer che tipo di cambiamento di stato è avvenuto. Gli observer possono quindi mantenersi coerenti con quello che è successo nel subject. Questo è tipico di sistemi come database.

L'observer, inoltre, può anche essere observer di più subject. In tal caso è necessario anche capire qual è il subject che ha ricevuto una modifica di stato.

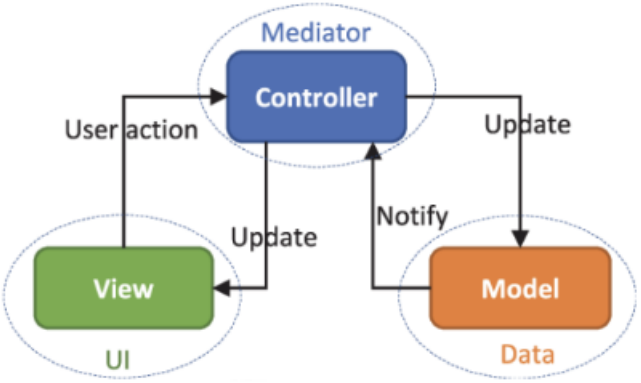
Model View Controller

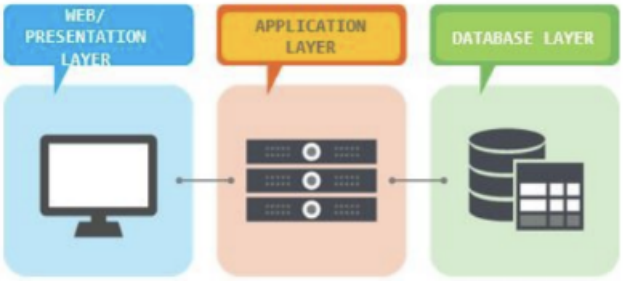
Il pattern Observer applicato all'architettura di un intero sistema prende il nome di **Model View Controller**.

Il pattern Model View Controller è un modo di implementare a livello di sistema uno schema architetturale già pensato a priori. Si divide l'applicazione in tre grossi sottosistemi, in cui si separano tre aspetti fondamentali:

- il **controller**, dove si eseguono calcoli e la **business logic** del software;
- il **modello**, la parte che si occupa della **persistenza dei dati** (su quali strutture dati vengono eseguiti i calcoli) e della **rappresentazione dei dati** in memoria;
- la **vista**, ossia la **GUI**, l'interfaccia utente grafica.

Questo schema architetturale non indica solamente i tre moduli base, ma anche come devono interagire tra loro. La struttura fornita dal modello è la seguente:



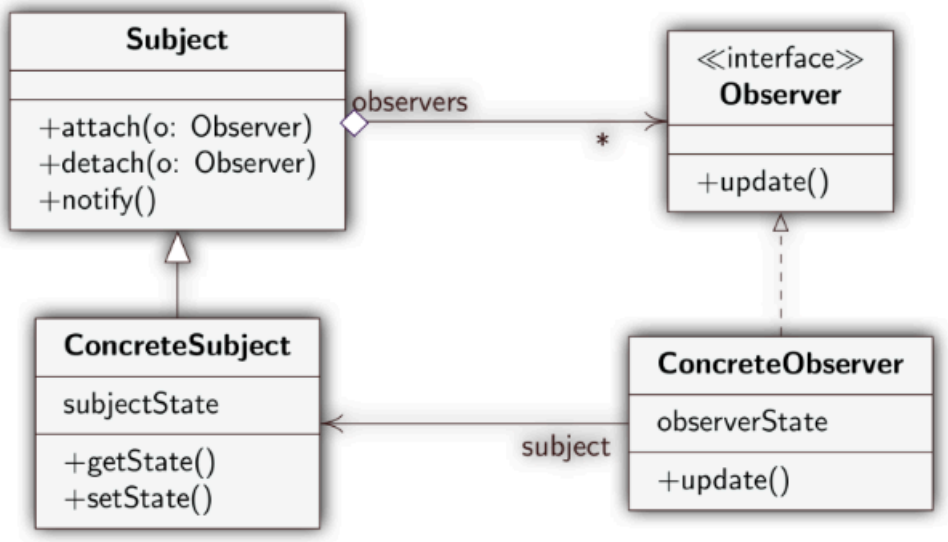


Ad esempio, ogni applicazione web utilizza l'architettura fornita dal pattern MVC. Organizzare un'applicazione web con questi tier deriva dall'idea di imporre il pattern strutturale più un controller model col vantaggio che le tre componenti possono essere **remote**. Ad esempio:

- la view è il browser;
- il controller è un'applicazione sul server web;
- il model è il database che conserva i dati del server.

Quando si esegue quindi un'operazione, il controller può protare ad aggiornamenti nella view e nel model. Il sistema opera con un meccanismo di **azione-reazione**. Pertanto l'interazione non avviene tramite **messaggi alle interfacce**, bensì come reazioni a notifiche di eventi. Questo favorisce **disaccoppiamento** all'interno del sistema software.

Schema del pattern



Il `ConcreteSubject` estende una classe predefinita `Subject`. Ogni `ConcreteSubject` è dotato di un `subjectState` e di due metodi `getState()`, `setState()` per la lettura e la modifica dello stato. La lettura dello stato è necessaria per gli observer, in quanto è necessario capire qual è stato il cambiamento di stato. La classe `Subject` prevede le seguenti operazioni:

- `attach(Observer)`;
- `detach(Observer)`;
- `notify()`.

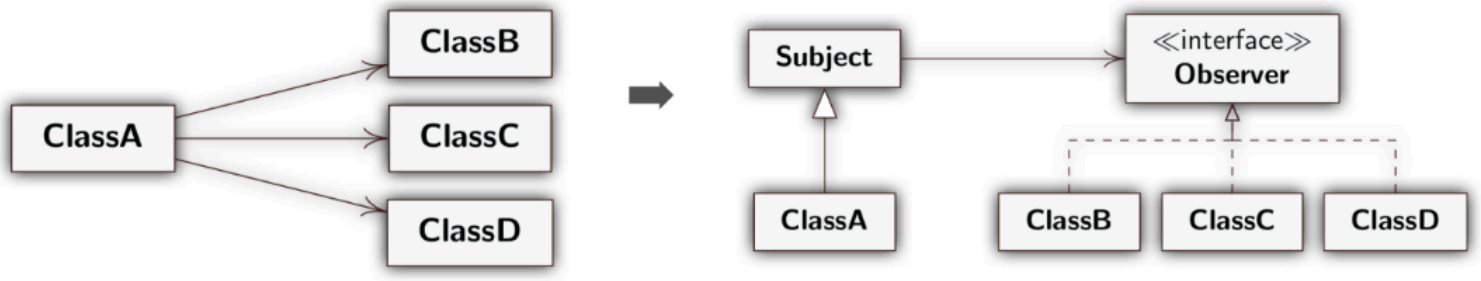
Il `Subject` pertanto deve avere un **array di** `Observer`. L'interfaccia `Observer` prevede un metodo `update()`. Ogni `ConcreteObserver` è dotato di un `observerState` e implementa la logica di `update()`. Aggiungeremo inoltre in `Subject` una

variabile booleana `changed` per ottimizzare il meccanismo di notifica agli observer (come spiegheremo successivamente).

La funzione `notify()` avvisa tutti gli observer conservati nell'aggregato `observers` presente in `Subject`. Ogni observer si aggiornerà in virtù della notifica ricevuta dal subject che invoca `notify()`.

Subject e observer sono entità **indipendenti tra loro**. Il subject inoltre sa quando i suoi algoritmi modificano in modo significativo lo stato: deciderà quindi di notificare il cambiamento di stato con il metodo `notify()`. La `notify()` si traduce nell'invocazione di `update()` su tutti gli observer.

L'applicazione del pattern snellisce le dipendenze, trasformando la dipendenza tra le classi concrete in una dipendenza tra il subject e l'observer:



Come si nota, la dipendenza di `ClassA` con le altre classi si è trasformata in una dipendenza tra `Subject` e `Observer`.

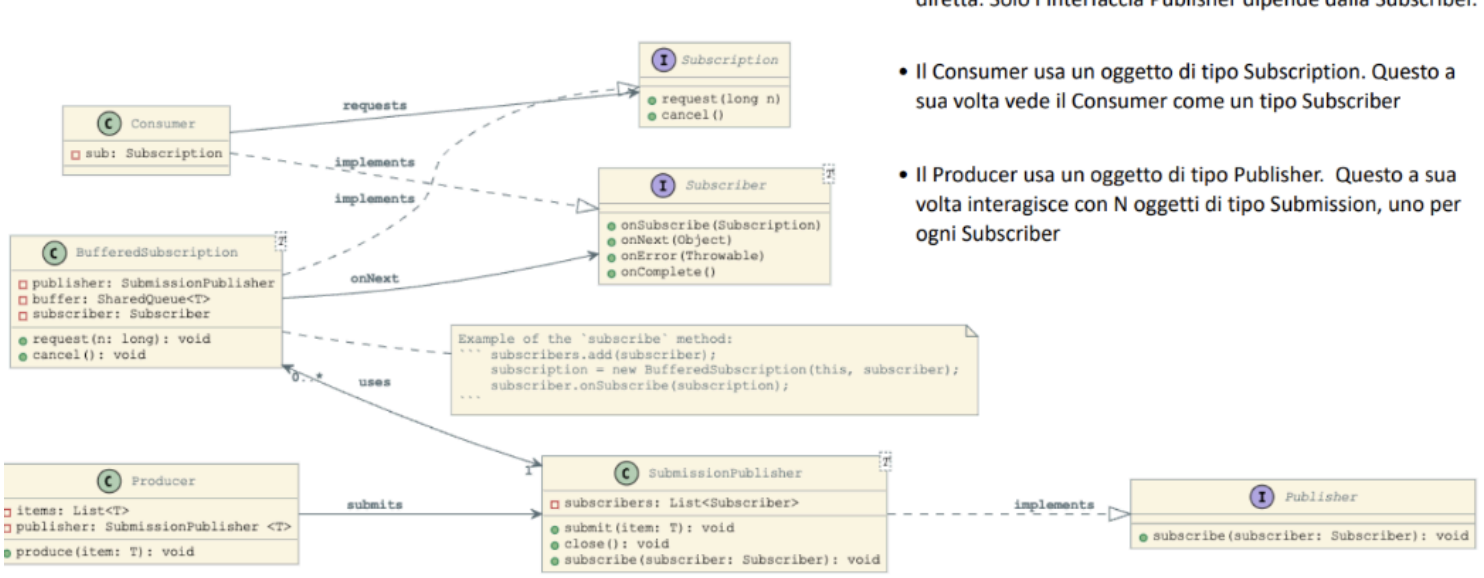
Esempio

Le interfacce grafiche fanno uso di questo pattern. Gli elementi dell'interfaccia grafica possono aggiornare altri observer come reazioni ad altri eventi.

Publish/Subscribe, reactive programming

Chiaramente se un metodo utilizza il metodo `notify()` il client deve attendere che ogni observer esegua `update()`. Si tratta di un meccanismo **sincrono**. L'implementazione pertanto rende imprevedibile l'efficienza del programma. Una soluzione è implementare un reale sistema di azione-reazione che gestisca la reattività in maniera del tutto **asincrona**.
Il framework di azione-reazione esiste come concetto nel paradigma di programmazione generale noto come **programmazione reattiva**. Sistemi come react permettono di scrivere applicazioni in cui il meccanismo di interazione tra componenti si basa sulla reazione di azioni. Standardizzare gli eventi rende possibile la comunicazione.
Java sposa il framework in una libreria che prende il nome di **Flow**, che fornisce una reattività **asincrona** tra i componenti del programma in maniera del tutto trasparente al programmatore che la utilizza. La libreria fornisce le interfacce `Publisher<T>`, `Subscriber<T>`, `Processor<R, T>` e `Subscription` per astrarre il passaggio di un flusso di oggetti da un `Publisher` a un `Subscriber` **senza che vi siano blocchi per il publisher e senza inondare il subscriber**.

Si utilizza allora una implementazione più efficiente del pattern Observer, ossa la versione **Publish/Subscribe**. Viene introdotto un ADT `Subscription` per disaccoppiare realmente l'esecuzione delle `notify()` dal subject agli observer:

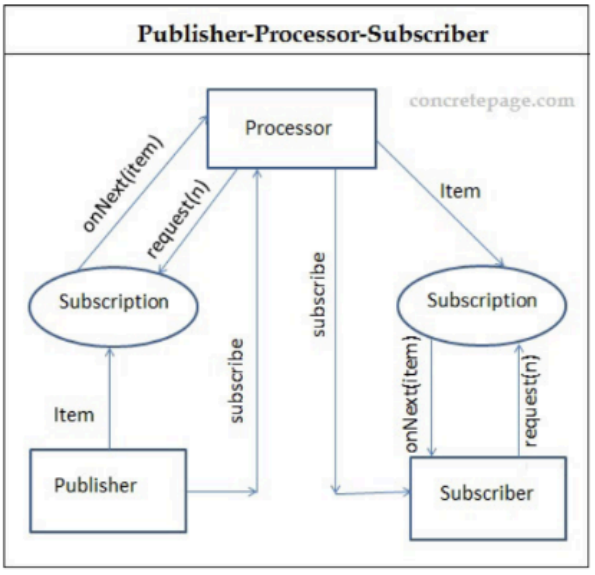


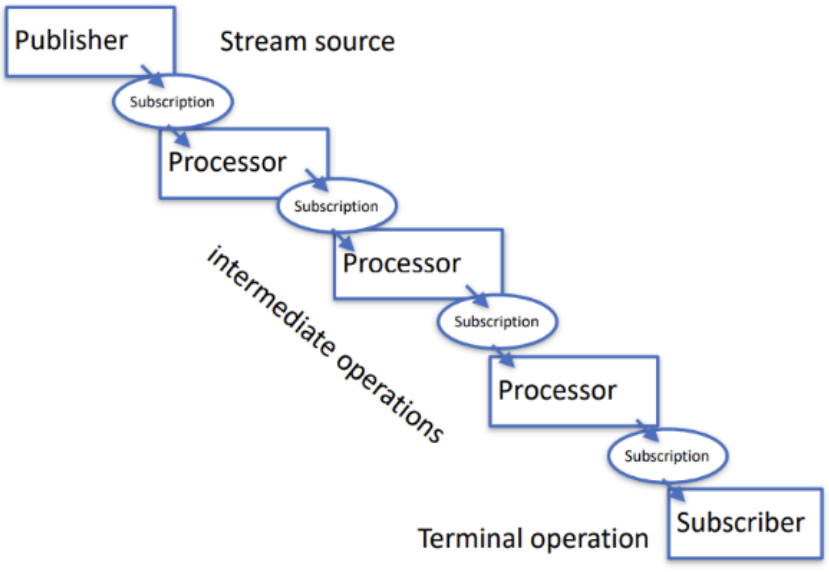
- Il Consumer usa un oggetto di tipo Subscription. Questo a sua volta vede il Consumer come un tipo Subscriber
- Il Producer usa un oggetto di tipo Publisher. Questo a sua volta interagisce con N oggetti di tipo Submission, uno per ogni Subscriber

La `BufferedSubscription` che implementa l'interfaccia `Subscription` contiene al suo interno uno spazio dove ospitare le notifiche generate dal sistema. La subscription viene utilizzata da un `SubmissionPublisher` che implementa l'interfaccia `Publisher`. Un `Producer` produce degli oggetti ed esegue dei `submit` alla `SubmissionPublisher`. In questo modo il `Producer` comunica con un `Publisher` che interagisce con `Subscription`. Ogni `Subscription` invia una notifica al `Consumer`. Lo stesso `Producer` è in grado di comunicare quindi con più `Subscription`, permettendo quindi di decentralizzare queste funzionalità e non inserirle nel `Producer`.

Ogni `Consumer` quindi può avere anche più `Subscription` in base a quanti sono i `Publisher`. Il sistema inoltre permette di gestire in maniera efficiente una relazione molti a molti tra produttori e consumatori. La comunicazione avviene grazie alle implementazioni delle interfacce `Subscriber` da parte del `Consumer` e di `Publisher` da parte di `SubmissionPublisher`

Tra un `Publisher` e un `Subscriber` si può avere un'interfaccia intermedia, ossia `Processor`, che implementa entrambe le interfacce. Il `Processor` è un elemento che serve a poter creare trasformazioni, in quanto si vede un flusso in ingresso e un flusso in uscita. I `Processor` possono essere composti per creare un **flusso di trasformazioni**. Il meccanismo illustrato rappresenta uno **stream** di elaborazioni:





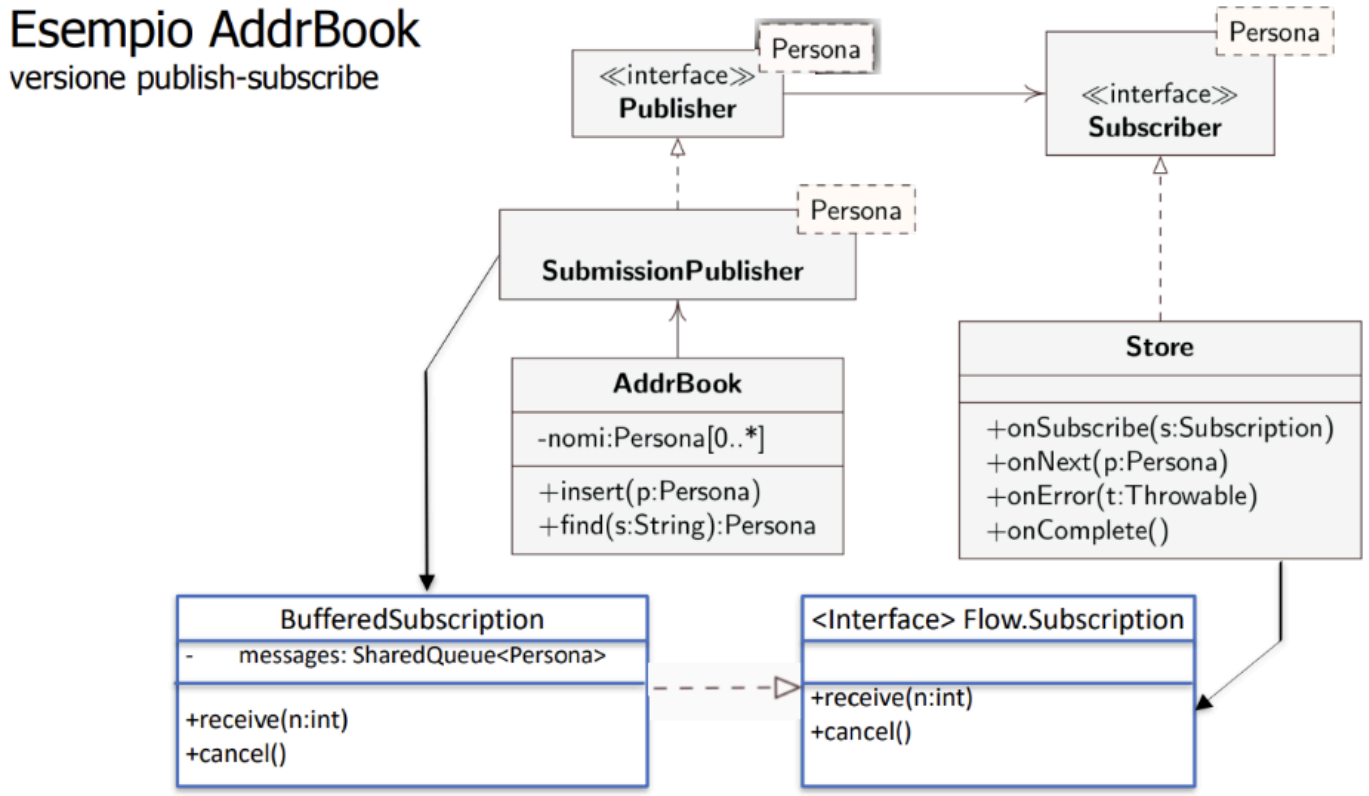
Riassumendo:

- `Publisher<T>` è l'interfaccia che definisce il metodo `subscribe(Subscriber<T>)`, analogo di `attach()` ad un `Observer` nel `Subject`. Si tratta dell'interfaccia per fornire item;
- `Subscriber` è usato per ricevere item, definendo i metodi `onNext()`, `onError()`, `onComplete()` e `onSubscribe()`. Tali metodi corrispondono a `update()` dell'interfaccia `Observer`;
- `SubmissionPublisher` implementa l'interfaccia `Publisher<T>` e invia item ai `Subscriber` **in maniera asincrona** con thread dedicati che chiamano `onNext()` dei `Subscriber` tramite il metodo `submit()`;

- `Subscription` è il collegamento tra `Publisher` e `Subscriber`, permettendo al `Subscriber` di ricevere messaggi o di fermarli. Quando un `SubmissionPublisher` chiama `subscribe`, il `Subscriber` riceve una `Subscription` da cui può ricevere le notifiche con `request(n)`, o annullare la ricezione con `cancel()`.

Esempio

Consideriamo il seguente esempio:



La classe subject utilizza una classe `SubmissionPublisher`. All'observer arriverà come dato dentro la notifica istanze di tipo `Persona`. Lo `Store` implementa un'interfaccia template generica da personalizzare con il tipo `Persona`. `Store` ha una funzione update `onNext(Persona)`, e supporta come si nota nello schema UML quattro tipi di notifiche diverse.

`AddrBook` (subject) non deve derivare da `Subject`, ma ha un **riferimento** a `Subject`: `AddrBook` utilizza `SubmissionPublisher` con il tipo `Persona`. `SubmissionPublisher` utilizza un oggetto `BufferedSubscription`, che gestisce una **coda**. Le notifiche vengono invocate su questo oggetto. Questo oggetto, inoltre, contiene i riferimenti agli oggetti. L'invocazione delle notifiche, inoltre, avviene in un thread separato rispetto al subject. In questo modo gli oggetti non si bloccano tra di loro.

Le update quindi non sono inserite in un ciclo for, ma sono smistate in una coda gestita da una `Subcription`. Il collegamento con la coda si realizza tramite un riferimento alla coda contenuto negli observer. La ricezione delle notifiche viene gestita con un metodo `receive(n)`, che permette di ricevere un numero di notifiche massimo pari ad n. In questo modo la funzione `onNext()` estrae la successiva notifica dalla coda. Si fornisce così:

- asincronia tra subject e observer;
- la possibilità di controllare la velocità di ricezione delle notifiche.

Considerazioni

`Subject` conosce solo la classe `Observer`. In questo modo:

- i `ConcreteSubject` e i `ConcreteObserver` sono **disaccoppiati** e possono essere riutilizzati e modificati più facilmente;
- gli osservatori possono essere rimossi in qualsiasi momento, in quanto il `ConcreteSubject` non sa quanti sono i `ConcreteObserver`.

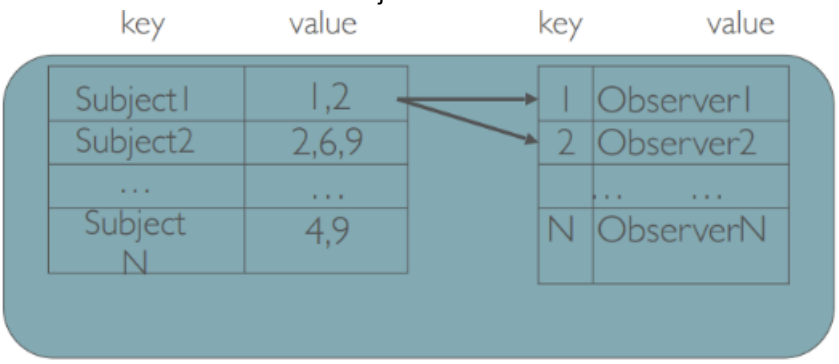
La notifica inviata da un subject può anche essere ignorata. La notifica, inoltre, può non dire agli Object cos'è cambiato, ma solamente notificare il completamento di un'operazione. In alternativa, il `Subject`:

- può passare informazioni di stato come parametro di `notify()`. Questa modalità prende il nome di **modalità push**;
- può concedere all' `Observer` l'accesso al suo stato. Questa modalità prende il nome di **modalità pull**.

Nella versione obsoleta del pattern esiste un flag di stato booleano `changed` in `Subject`. Si prevede un metodo `setChanged()` per impostare il valore di `changed`. Quando si esegue `notify()`, vogliamo far scattare meccanismi in oggetti remoti. `notify()`

segnala quindi anche cambiamenti di stato. Ma questi due momenti **non sono sempre coincidenti**. Supponiamo che nel nostro `Subject` ci siano diversi momenti in cui nell'elaborazione dell'algoritmo lo stato può cambiare. Tuttavia, magari basta semplicemente notificare che sia avvenuta almeno una modifica di stato, **ma non ogni modifica**. Non ha senso quindi notificare ogni volta gli observer.
In generale, vogliamo distinguere momenti in cui è successo un cambiamento significativo per altri oggetti, sebbene questo possa accadere in più punti dell'elaborazione. Invece di invocare `notify()`, si esegue `setChanged()` per impostare la booleana a `true`. Alla fine dell'algoritmo, se `changed` è `true` si esegue `notify()`.

Infine, se si ha una situazione di azione-reazione molti a molti può essere utile l'utilizzo di una **shared look-up table**. Se ogni subject ha tanti observer e viceversa e gli observer sono condivisi, invece di avere i riferimenti conservati tra loro negli observer, basta creare una tabella condivisa, in cui per ogni subject si inserisce la lista di observer condivisi tra loro. In questo modo si evita di duplicare i riferimenti agli observer e non si conservano subject senza observer:



13 - Lambda

Espressioni lambda

Una funzione lambda è un blocco di codice che descrive una funzione anonima non associata ad alcuna classe. Solitamente gli algoritmi vengono incapsulati in un metodo in cui ogni parametro è tipizzato e identificato da un nome. La sintassi di una funzione lambda è la seguente:

```
(lista_parametri) -> {corpo_lambda}
```

Tipicamente le funzioni lambda sono semplici, leggibili e facili da debuggare. Il costrutto rappresenta una funzionalità che è un **oggetto di prima classe**. La funzione si dice anonima perché non si sa qual è il nome, ma si può comunque passare o assegnare a qualunque variabile.
Le funzioni lambda inoltre supportano parametri astratti:

```
(x, y) -> x+y;
```

La funzione lambda soprastante definisce la somma di due elementi `x, y` il cui tipo è sconosciuto. Funzioni del tipo:

```
(x, y) -> x.length() + y.length();
```

Richiederebbe che `x` e `y` siano stringhe per funzionare correttamente (o qualunque ADT che supporti `length()`). Se i parametri passati non hanno il metodo `length()`, tramite una **deduzione di tipo a tempo di compilazione** viene segnalato un errore e il bytecode non viene prodotto.
Poiché la funzione lambda è un oggetto di prima classe, il qualificatore `this` fa riferimento al contesto in cui è definita la funzione lambda. In generale una funzione lambda può utilizzare i parametri dello scope in cui è definita:

```
class Test{
    private int testInt;
    void foo(ArrayList<Integer> arr, int p) {
        arr.forEach(d -> System.out.println(d + p));
    }
}
```

Interfacce funzionali

Un'**interfaccia funzionale** è un'interfaccia che ha **un solo metodo astratto** (ma non obbligatoriamente un solo metodo). Si utilizza l'annotazione `@FunctionalInterface` per specificare al compilatore di controllare se effettivamente si ha un solo metodo astratto nell'interfaccia definita; se ciò non accade, viene generato un errore a tempo di compilazione.

Abbiamo già visto le interfacce funzionali `Consumer` e `Comparator`. Introduciamo `Predicate<T>` come interfaccia funzionale che implementa il metodo `test(T)`. Il metodo astratto restituisce un booleano e riceve un elemento di tipo `T`.

Utilizzare le interfacce funzionali permette di definire on-spot delle classi che implementano la funzione `test(T)` e passarle a funzioni:

```
public void elaborate(Test t, Predicate<Test> func) {
    if(func.test(t)) {
        //...
    }
}

elaborate(t, new Predicate<Test>() {
    @Override
    public boolean test(Test t) { t.getValue() == 3; }
});
```

Possiamo anche fare di meglio utilizzando le espressioni lambda:

```
public void elaborate(Test t, Predicate<Test> func) {
    if(func.test(t)) {
        //...
    }
}

elaborate(t, t -> t.getValue() == 3);
```

Interfacce funzionali note sono:

- `Predicate<T>`, che produce un risultato booleano sulla proprietà di un oggetto. Contiene `boolean test()`;
- `Consumer<T>`, che consuma un oggetto ricevuto eseguendo un azione. Contiene `void accept(T)`;
- `Function<T, R>`, che trasforma `T` a `R`, creando una corrispondenza $T \rightarrow R$. Contiene `R apply(T)`;
- `Supplier<T>`, che fornisce un'istanza di `T`. Un `Supplier` non è altro che un factory. Contiene `T get()`;
- `UnaryOperator<T>`, funzione da `T` a `T`. Contiene `T apply(T)`;
- `BinaryOperator<T>`, funzione che associa `(T, T)` a `T`. Contiene `T apply(T, T)`;
- `Factory<T>`, che fornisce un metodo per istanziare oggetti di tipo `T`. Contiene `T create()`.

Queste interfacce sono presenti in `java.util.function`.

Un'interfaccia funzionale può anche prevedere l'implementazione di metodi ereditati dalla classe `Object`:

```
public interface TestInterface {
    String printInfo(TestObject test);
    boolean equals(Object obj);
}
```

Ogni classe che implementa `TestInterface` dovrà implementare `equals()`. Questa è ancora un'interfaccia funzionale in quanto `equals(Object obj)` è ereditato da `Object`, pertanto è solo un override.

Anche se ad una funzione viene passata un'espressione lambda, il compilatore analizza il tipo e l'assegnazione. Ad esempio:

```
void consumeStudent(Consumer<Student> s) {
    // logic of consuming...
}

consumeStudent(s -> System.out.println(s.getName()));
consumeStudent( () -> System.out.prinln("Test"));
```

Mentre il primo passaggio a funzione è legale, il secondo non viene accettato dal compilatore: l'espressione lambda rappresenta una funzione che non prende in input nessun parametro, ma l'interfaccia funzionale `Consumer<Student>` prevede un metodo `accept(Student)`, che richiede quindi un parametro.

L'aspetto importante nell'utilizzo delle interfacce funzionali è ricordare la firma della funzione dichiarata all'interno dell'interfaccia. In questo modo non abbiamo realmente bisogno di istanziare classi, ma possiamo utilizzare classi anonime o ancora meglio utilizzare delle espressioni lambda, rispettando tuttavia il numero di parametri in ingresso e il tipo in uscita (per le espressioni il tipo può anche essere dedotto).

Riferimenti a metodi e costruttori

In java è possibile passare un metodo esistente come parametro basta passare il nome del metodo. Per passare i metodi è necessario utilizzare l'operatore di **dereferenziazione** `::`.
Consideriamo il seguente esempio:

```
class Person {
    private String name;
    private int age;

    public int getAge() { return this.age; }
    public String getName() { return this.name; }
```

```
Person[] people = ...;
Comparator<Person> byName = Comparator.comparing(Person::getName);
Arrays.sort(people, byName);
}
```

L'interfaccia `Comparator` presenta un metodo statico `comparing` con la seguente firma:

```
static <T,U extends Comparable<? super U>> Comparator<T> comparing(Function<? super T,? extends U>
keyExtractor)
```

Tale metodo statico accetta una funzione che a partire da un oggetto di tipo `T` restituisce una chiave di ordinamento di tipo `U` e restituisce un `Comparator<T>` che esegue confronti in base alla chiave. Il `Comparator` viene utilizzato per ordinare gli array tramite la funzione `Arrays.sort`:

```
public static <T> void sort(T[] a, Comparator<? super T> c)
```

Invece di istanziare un oggetto `Function` possiamo passare un **riferiemnto ad un metodo**. Il riferimento in questo caso viene detto **riferimento ad un metodo di istanza di un oggetto arbitrario**. Esistono quattro tipi di riferimenti di metodo:

- **riferimento ad un metodo statico**: si utilizza il nome di una classe e la dereferenziazione di un metodo statico della classe. Da notare che questo vale anche quando un metodo non statico di una classe non utilizza alcuna variabile di istanza;
- **riferimento ad un metodo di istanza di un oggetto specifico**: si utilizza il nome dell'oggetto e la dereferenziazione di un metodo;
- **riferimento ad un metodo di istanza di un oggetto arbitrario di un tipo specifico**: si utiliza il nome del tipo e la dereferenziazione di un metodo;
- **riferimento ad un costruttore**: si utilizza il nome della classe e la dereferenziazione di `new`.

Metodi default

Per introdurre nuove funzionalità senza dover cambiare il codice pre-esistente, Java 8 introdusse i **metodi default**. Le librerie pre-esistenti possono ereditare le funzionalità: il metodo default infatti viene ereditato dalle altre interfacce come già implementato. Utilizzare i metodi default quindi permette di inserire nuove funzionalità nelle vecchie interfacce senza cambiare il codice, rendendo quindi il codice retro-compatibile. Ad esempio, l'interfaccia `Iterable` implementa `forEach(Consumer c)` come metodo default.

La gerarchia dei tipi in Java deve consentire un solo padre, evitando quindi l'ereditarietà multipla. Tuttavia, una classe potrebbe implementare più interfacce. A livello di ereditarietà di tipi, questo non crea ambiguità: una classe sta solo includendo più set di funzionalità. Ma dal punto di vista implementativo si possono avere conflitti: se nelle due interfacce implementate esistono metodi default con lo stesso nome, si ereditano più implementazioni dello stesso metodo. Invocare la funzionalità col nome, il nome è uno, ma le implementazioni sono molteplici. Per risolvere il conflitto, è necessario utilizzare il nome dell'interfaccia e invocare con `super`:

```
interface I1 {
    public default void display() {
        System.out.println("Interfaccia 1");
    }
}

interface I2 {
    public default void display() {
        System.out.println("Interfaccia 2");
    }
}

public class Test implements I1, I2 {
    public void display() {
        I1.super.display();
        I2.super.display();
```

```
    }
}
```

14 - Streams

Streams

L'interfaccia Stream è un'interfaccia che raccoglie operazioni astratte che operano su un flusso o aggregato di oggetti. Gli stream forniscono dei blocchi funzionali per processare meglio gli aggregati:

- utilizzando un'iterazione interna. Il ciclo scompare del tutto, in quanto non è necessariamente un ciclo, ma un'applicazione a tutto l'aggregato di un'operazione;
- supportando il paradigma di **programmazione funzionale**. Passiamo dalla definizione imperativa di algoritmi, che prevede la precisazione dell'ordine di esecuzione, ad una definizione funzionale, in cui l'ordine di esecuzione non è definito così come accade quando si valuta un'espressione matematica.

Elementi di uno stream

Gli stream sono costituiti da tre tipi di elementi:

- una **source**;
- una o più **operazioni intermedie**;
- un'**operazione terminale**.

L'operazione terminale potrebbe dare vita ad un aggregato di oggetti (non necessariamente uguale all'aggregato di partenza) oppure può essere una riduzione ad un elemento più semplice. Le operazioni intermedie, tuttavia, non lavorano su un aggregato di oggetti, bensì un oggetto di tipo `Stream`. La sorgente quindi permette di generare uno stream. Ad esempio, una `Collection` può essere trasformata in uno `Stream` dopo una trasformazione. Tale processo di trasformazione è l'invocazione del metodo `stream()` sulla collezione. Lo stream esiste solo durante l'esecuzione delle operazioni intermedie.

Dettagli interni

All'interno dell'interfaccia `Collection` è presente un metodo `stream()` che trasforma l'oggetto in uno `Stream` che ha accesso all'oggetto `Collection`. Lo `Stream` creato incapsula al suo interno la `Collection`, pertanto le funzionalità applicate allo stream si riflettono sull'aggregato incapsulato.
Molti metodi all'interno di `Stream`, inoltre, restituiscono uno `Stream`, permettendo una **concatenazione delle operazioni**:

```
result = albums.stream().filter(track -> track.getLength() >
300).map(Track::getName).collect(Collectors.toSet());
```

La funzione `filter()` prende in input un `Predicate`, filtrando gli elementi dello stream in base al predicato passato. Dopo aver applicato un filtro, utilizziamo la funzione `map()`, che prende in input una `Function`, per mappare, ossia associare, ad ogni traccia il suo nome. Infine, il metodo `collect` esegue un assemblaggio dello stream in una collezione, utilizzando la funzionalità `Collectors.toSet()`. Si ottiene quindi un insieme di tracce che hanno una lunghezza maggiore di 300.
Dietro le quinte, tuttavia, non è detto che accada quanto illustrato. In realtà, infatti, **nessuna delle operazioni intermedie viene eseguita nell'ordine specificato dalla riga di codice**. L'esecuzione in realtà comincia dalla operazione terminale: le operazioni intermedie fungono da specifica per il flusso di oggetti.

Creare uno stream

Abbiamo visto che in `Collection` è presente un metodo `stream()`. Esiste anche un metodo `parallelStream()`, che fornisce uno stream gestito in maniera parallela.
In `Arrays` esiste `stream()` poiché anche un array è un aggregato di dati. Il metodo fornisce uno stream sequenziale.

Per creare stream a partire da valori, si utilizza `of()`:

```
Stream<Integer> digits = Stream.of(3,1,4,5,6,2);
```

Uno stream vuoto può essere creato con `empty()`:

```
Stream<Student> ss = Stream.empty();
```

Si possono generare stream a partire da funzioni:

```
Random random = new Random();
Stream<Integer> randomNumbers = Stream.generate(random::nextInt);
```

`random` permette la generazione di numeri pseudo-casuali. Per questo motivo utilizziamo il riferimento di un metodo di istanza: utilizzare `nextInt` in maniera statica implica l'utilizzo di uno stesso seed di generazione che ad ogni istanza genera la stessa sequenza di numeri. Il seed è una variabile di istanza, pertanto istanziamo un oggetto `random` di tipo `Random`.
In alternativa a partire dagli array è possibile utilizzare il metodo `of()` per creare uno stream:

```
Integer[] arr = {1,2,3,4,5};
Stream<Integer> digitStream = Stream.of(arr);
```

Abbiamo già visto che con `Collection` esiste il metodo `stream()`:

```
Set<Student> ss = new LinkedHashSet<>();
Stream<Student> stream = ss.stream();
```

Si possono anche creare stream a partire dai file:

```
Stream<String> stream = Files.lines(path);
```

Infine, si possono concatenare gli stream:

```
Stream newStream = Stream.concat(stream, randomNumbers);
```

Builder pattern

Per creare uno stream, all'interno dell'interfaccia è presente un metodo `builder()` che restituisce un **builder** per uno `Stream`. Il metodo si basa sul **builder pattern**, che risolve il problema dell'assemblaggio di una struttura composta anche da altre strutture composte secondo una logica corretta. In questo caso possiamo utilizzare il metodo `builder()` per costruire lo `Stream`. Il metodo `add()` permette di aggiungere elementi allo `Stream`. Invocare infine il metodo `build()` permette di restituire lo `Stream` di oggetti. Da notare che prima di invocare `build()` è necessario passare tutti gli oggetti con `add()`. Inoltre, se passiamo un aggregato con `add()`, l'aggregato è un elemento dello stream, permettendo la creazione di **stream compositi**. Quando si processa uno stream di elementi, non è detto che il singolo elemento dello stream sia semplice.

Stream primitivi e infiniti

Può darsi che invece di avere un flusso di oggetti si abbia un flusso di dati elementari. Questo flusso di dati è comunque trattato dall'operazione terminale correttamente, ma bisogna stare attenti a distinguere `int` e `Integer`: il primo è un tipo elementare, il secondo è una classe. L'interfaccia generale è quindi `Stream`, ma ci sono delle istanze specifiche di `Stream`, come `IntStream`, `DoubleStream`, e così via.

Ad esempio, le funzioni `range(int, int)` e `rangeClosed(int, int)` permettono di generare uno stream dal valore di inizio al valore di fine (`rangeClosed()` include il valore finale).

Inoltre è possibile anche generare uno **stream infinito** a partire da un `Supplier`:

```
generate(IntSupplier)
```

Uno stream infinito può essere utile per generare una sequenza numerica. Uno stream infinito può anche essere creato con `iterate`:

```
Stream<Integer> infiniteStream = Stream.iterate(0, i -> i + 1)
```

Lo snippet soprastante genera una sequenza strettamente crescente di numeri. Chiaramente se vogliamo utilizzare il metodo `collect()` per assemblare lo stream è necessario limitare il numero di valori, altrimenti **si ottiene un'esecuzione infinita del programma**:

```
infiniteStream.limit(10).collect(Collectors.toList());
```

Lo snippet prende i primi 10 valori.

Iterazione con `Stream.iterate`

La funzione `iterate()` dell'interfaccia `Stream` permette di ottenere uno stream ordinato:

```
static <T> Stream<T> iterate(T seed, Predicate<? super T> hasNext, UnaryOperator<T> next)
```

La funzione applica iterativamente la funzione `next` passata come funzione unaria. L'iterazione continua fino a quando `hasNext` non restituisce `false`. Produce la stessa identica sequenza del ciclo for:

```
for(T index = seed; hasNext.test(index); next.apply(index)) {
    ...
}
```

Stream sequenziali e paralleli

Una collection genera tramite il metodo `stream()` uno stream sequenziale. Uno stream **parallelo** può essere utilizzato per eseguire operazioni in parallelo. Le operazioni chiaramente vengono eseguite o esclusivamente in parallelo o esclusivamente in sequenza:

```
List<Person> persons = ...;
List<Student> students = persons.stream().parallel().filter(p->p.getAge() >
18).sequential().map(Student::new).collect(Collectors.toCollection(ArrayList::new));
```

Uno stream sequenziale, specificato con `sequential()`, viene eseguito utilizzando un unico thread. Uno stream parallelo invece viene gestito utilizzando più thread, ma causa più overhead a causa della gestione di questi e dei costi del merging dei risultati.

Operazioni intermedie

Le operazioni intermedie solitamente prendono in input delle espressioni lambda, o in generale delle funzioni che sono **stateless**. L'iterazione è nascosta all'interno dell'implementazione dello stream nella libreria `Stream`.

Alcune operazioni intermedie note sono:

- `filter(Predicate)`: fitlra gli elementi in virtù del risultato di `Predicate`. Se il predicato contiene delle congiunzioni logiche, conviene utilizzare `filter()` per ogni singolo predicato, in quanto si ha un confronto semplificato, poiché gli elementi eliminati dal filtro per il primo predicato non verranno considerati per il filtro per il secondo predicato. Inoltre poiché l'ordine dei termini non è specificato per le operazioni intermedie, può darsi che il confronto separato filtri gli elementi più rapidamente, senza cambiare il risultato;
- `map(Function)`: trasforma un elemento usando `Function`;
- `mapToInt, mapToDouble, mapToLong`: utilizzate come `map()`, ma generano uno stream di tipi primitivi;
- `flatMap(Function)`: trasforma ogni elemento in un **sottostream** di zero o più elementi. Semplicemente, mentre `map()` associa ad un elemento un unico elemento, `flatMap()` associa ad un elemento anche più elementi;
- `peek(Consumer)`: esegue un azione su ogni elemento, utile per il debugging;
- `distinct()`: escluse elementi duplicati;
- `sorted(Comparator)`: restituisce uno stream ordinato. Se il `Comparator` non è specificato, restituisce un ordine naturale degli elementi;
- `limit(long)`: limita il numero di elementi dallo stream;
- `skip(long)`: salta gli elementi iniziali, il cui numero è specificato dal parametro.

Operazioni terminali

Le operazioni terminali terminano la pipeline delle operazioni eseguite sullo stream. Il processamento viene eseguito quando viene chiamata un'operazione terminale, ottimizzando la pipeline. Questa tecnica prende il nome di **lazy evaluation**. Si rimuovono inoltre operazioni ridondanti. Infine, le operazioni sono eseguite in parallelo.

`collect(Collectors)` accumula lo `Stream` in un aggregato. La collezione può essere specificata utilizzando metodi quali `Collectors.toList()` oppure `Collectors.toCollection(constructor)`, dove `constructor` è il riferimento al costruttore di una `Collection`. Esistono anche computazioni per la somma, come `Collectors.summingInt(int_value)`, oppure le stringhe `Collectors.joining(str)`, dove `str` indica di concatenare le stringhe utilizzando la stringa `str`.

È anche possibile partizionare elementi in virtù di un `Predicate`:

```
Map<Boolean, List<Student>> passingFailing = students.stream().collect(Collectors.partitioningBy(s ->
s.getGrade() >= 70))
```

La partizione viene quindi utilizzata per associare ad ogni studente un valore booleano in virtù della condizione, che in questo caso è avere un voto che sia maggiore o uguale a 70.

Optional

Per eliminare la `NullPointerException` si utilizza la classe `Optional<T>`. Operazioni come `min()` e `max()` potrebbero infatti non restituire un risultato diretto: se lo stream è vuoto, non si può determinare qual è l'elemento minimo o massimo.
Un `Optional<T>` è un container di elementi di tipo `T` che ha 0 o 1 elemento, ma non è mai `null`.

`Optional` include i metodi:

- `empty()` per creare un `Optional` vuoto;
- `of(T value)` restituisce un `Optional` che contiene `value`. Se `value` è `null`, il metodo causa una `NullPointerException`;
- `ofNullable(T value)` ha lo stesso scopo di `of()` ma se `value` è uguale a `null` restituisce un `Optional` vuoto;
- `get()` restituisce il valore contenuto in `Optional`. Se `value` è vuoto causa una `NoSuchElementException`;
- `isPresent()` restitusce se è presente un valore;
- `orElse(T defaultValue)` restituisce un valore default se `value` è empty;
- `orElseGet(Supplier<? extends T> defaultSupplier)` ha lo stesso significato di `orElse` ma è un `Supplier` che dà il valore default;
- `orElseThrow(ExceptionSupplier)` causa un'eccezione dal supplier se l'optional è vuoto;
- `ifPresent(Consumer<T> consumer)` esegue le funzionalità fornite dal `consumer` con il value se `value` non è vuoto.

`reduce()`

Oltre a `collect` esiste anche `reduce(BinaryOperator accumulator)` per eseguire una **riduzione**, ossia per raccogliere i dati dello stream in un unico risultato. `accumulator` prende in input un risultato parziale, il successivo elemento e restituisce un risultato parziale sotto forma di `Optional<T>`. Quindi `accumulator` è una funzione del tipo `(T, T) -> T`. Ci sono anche altre due versioni:

- una versione prende in input un valore iniziale;
- un'altra versione prende in input anche una funzione `(T, U) -> U` per generare un risultato con tipo `U` invece che `T`.

Altre operazioni terminali

Altre operazioni terminali utilizzate sono:

- `count()` per contare gli elementi;
- `average()`, `sum()` per eseguire media e somma;
- `allMatch()`, `anyMatch()` e `noneMatch()` per restituire risultati booleani sulla base di un `Predicate` dato in input. `allMatch()` restituisce `true` se tutti gli elementi soddisfano il predicato, `anyMatch()` se ne esiste almeno uno e `noneMatch()` se nessuno soddisfa il predicato;
- `forEach()` esegue l'operazione fornita dal consumer passato al metodo, restituendo `void`;
- `forEachOrdered()` esegue la stessa cosa di `forEach()` ma seguendo un ordine.

15 - Testing

Validazione e verifica

Nello sviluppo di software non si ha solo una complessità dovuta alla difficoltà del problema, ma anche una complessità **intrinseca** allo sviluppo. Possiamo distinguere due "tipi" di complessità:

- la complessità **accidentale**, legata principalmente all'implementazione, pertanto può essere rimossa cambiando il codice o il linguaggio impiegato;
- la complensità **intrinseca**. Questa rimane anche se tutte le complessità accidentali sono state rimosse. Può essere controllata sviluppando un comportamento a run-time più prevedibile.

Complessità intrinseca

Per quanto si possa essere chiari nello scrivere a tempo di compilazione, dietro la scrittura chiara si nasconde un insieme di comportamenti a tempo di esecuzione che sono dinamici. Il solo fatto di utilizzare astrazioni permette di rendere chiaro il codice in maniera concettuale, ma ciò che poi effettivamente accade passo dopo passo, ossia il **comportamento a tempo di esecuzione**, non è visibile.

Ad esempio la seguente riga di codice è concettualmente corretta:

```
do j++ while (a[j] < pivot)
```

Ma **non sappiamo se il ciclo termina**. La terminazione del ciclo infatti dipende dalla presenza di un elemento `a[j]` che sia più grande di `pivot`.

Facciamo quindi fronte ad un software in cui sono presenti dei difetti. Tecnicamente distinguiamo dal punto di vista lessicale l'errore dalla sua manifestazione. Il **fault** è la parte del programma che causa un errore, ed è quindi il pezzo di istruzione che causa un comportamento sbagliato del programma. Un fault non è necessariamente qualcosa che sintatticamente o semanticamente è individuabile a tempo di compilazione: potrebbe essere una scrittura sensata sia sintatticamente sia semanticamente, ma l'errore potrebbe anche essere più difficile da debuggare poiché non si riesce ad associare la fault con la **failure**. Sinonimi di fault sono "bug", "difetto" e "anomalia".
Chiamiamo **failure** un comportamento anomalo a tempo di esecuzione.

Prove di correttezza

Possiamo eseguire due prove per verificare la correttezza del software: la **validazione** (o **testing**) e la **verifica**.

Validazione

Nella fase di validazione si eseguono dei test, concentrandosi su che cosa il software ha prodotto come risultato. Si verifica il "cosa" del software (ossia, il prodotto sviluppato è corretto?), ricevendo un'indicazione sulla correttezza del risultato. La validazione analizza il punto di vista dell'utilizzatore del prodotto. I test inoltre possono essere di due tipi:

- test **a scatola nera**, ossia dove i test sono basati su input e output;
- test **a scatola bianca**, ossia test che si basano sulla conoscenza del codice.

Verifica (analisi formale)

L'analisi formale è una verifica del come viene sviluppato il prodotto. Si controllano quindi se i meccanismi interni sono corretti per lo scopo. La verifica è mollto più efficace del testing: se verifichiamo la correttezza dei meccanismi interni, la risposta valida è molto più consistente come informazione, in quanto indica che i meccanismi interni funzionano e non possono causare comportamenti sbagliati. Per dare questa risposta, non si eseguono semplicemente dei test, ma si guarda l'implementazione.

Poiché i software tendono ad essere esponenzialmente complessi e la validazione è più semplice, si esegue la verifica solo per funzionalità e porzioni di software invece dell'intero software. Alcuni strumenti inoltre permettono l'analisi di proprietà (ad esempio, la terminazione di un ciclo) tramite l'utilizzo di linguaggi formali: si trasforma il codice in una serie di predicati e si verifica che la proprietà sia una tautologia.

Testing

Scelta dei test

La fase di testing è divisa in quattro fasi:

1. scelta del software da testare e del tipo di test;
2. impostazione dell'ambiente di testing, in modo tale che il codice possa essere eseguito ripetutamente;
3. selezione di un set di input basandosi su criteri;
4. valutare gli output ottenuti.

Per capire se un set di input è buono per un test, si parte dalla **specifica dei requisiti**. I test inoltre devono essere scritti in maniera corretta, facendo sì che non si sovrappongano tra di loro, in modo tale da evitare di scrivere test non necessari. Se inoltre il principio di singola responsabilità è rispettato, è possibile scrivere dei test che validano le funzionalità di una classe.

La funzionalità analizzata dai test può essere una funzionalità singola o una funzionalità macroscopica. Individuare la funzionalità è il primo passo; per testarla correttamente, il problema principale è dare l'input, selezionando correttamente partendo da un insieme universo di input possibili. Il test inoltre, come visto precedentemente, non fornisce una risposta definitiva sulla correttezza del software: per questo motivo è necessario eseguire un numero sufficiente di test.
Il problema non riguarda solamente l'input, ma anche la deduzione dell'output. Se l'output con cui confrontarsi è complesso, vi è il problema dell'**oracolo**. Fintanto che si eseguono test di pezzi di implementazione, la specifica del software permette di dedurre il risultato atteso. Man mano che si passa dal singolo metodo all'intera classe o ad un sottosistema, fino ad arrivare all'intero sistema, ci si basa sempre sulla specifica, ma ci si allontana dal dettaglio implementativo.
Nell'esecuzione del testing inoltre, per verificare se un modulo software è corretto è necessario **isolare** il modulo. Isolare non significa tuttavia portare fuori il modulo dal suo sistema, ma inserire il modulo in un **environment** in modo tale che venga "simulata" l'interazione con l'ambiente.

Selezione dei campioni

Il testing può essere visto come una sorta di **campionamento**. Il sistema viene analizzato in virtù di campioni scelti. Il campionamento degli input tuttavia dev'essere tale da rappresentare in maniera corretta l'insieme universo degli input. In virtù del campionamento si può dedurre se il sistema è affidabile e corretto. Esistono diversi **criteri di campionamento**, ognuno dei quali fornisce una **metrica di copertura**: essa ci permetterà di capire quanti sono i test da svolgere per poter validare il software.

A questo punto si può capire la differenza tra i test black box e white box:

- se l'implementazione del sistema non è visibile (black box), i test black box sono costruiti in modo tale da associare ad un input un output. I criteri di copertura per metriche per test di tipo black box guardano solamente gli input, ragionando su come dividere il dominio dei valori di input in modo tale da ottenere un insieme di campioni rappresentativo della popolazione;
- metriche per test di tipo white box campionano nello spazio delle esecuzioni. Il codice rappresenta n flussi di comportamento, pertanto per essere sicuro che il programma lavori bene bisognerebbe provare tutti questi flussi. Si campiona in uno spazio noto come **spazio dei comportamenti**.

Terminologia

> **Unit Testing**
>
> Prende il nome di **unit testing** il testing eseguito su singole funzioni o singole classi.

> **Regression Testing**
>
> Prende il nome di **regression testing** l'esecuzione di test già eseguiti in precedenza per verificare che, dopo un cambiamento, tutto funzioni correttamente.

> **Test suite**
>
> Prende il nome di **test suite** un insieme di test di input.

> **System Verification Group**
>
> Il **System Verification Group** è un team dedicato al testing.

Il test può essere eseguito a più livelli:

- test di **unità**, che ha per oggetto le parti atomiche del sistema;
- test di **integrazione**, che testano le interazioni di funzionalità atomiche per eseguire funzionalità di più alto livello. Le interazioni possono riguardare anche singoli moduli;
- test **funzionali**, che verificano le funzionalità previste dalle specifiche che deve rispettare il software. Si analizza quindi il design del sistema;
- test di **sistema**. Spesso il test funzionale è incluso nel test di sistema, in quanto l'oggetto di testing è l'intero sistema in entrambi i casi. Nel test di sistema non si guardano solo le funzionalità, ma anche aspetti tecnici che nel test funzionale non sono presenti;
- test di **accettazione**, che verificano il sistema in scenari di utilizzo.

Gli esiti dei test sono di quattro tipi:

- positivo;
- negativo;
- falso positivo;
- falso negativo.

Gli ultimi due esiti riguardano:

- test che **dovevano fallire** ma che hanno avuto esito positivo. In questo caso si **sa già che il test fallisce**, ma il test dà esito positivo;
- test che **dovevano avere successo** ma che hanno avuto esito negativo.

Tipicamente si tende ad utilizzare dei test white box solamente quando siamo vicini al codice (ossia test di unità). Per test di integrazione o di livello superiore si eseguono dei test di tipo black box. Solitamente i test di validazione sono di tipo black box, mentre i test di verifica sono white box (si possono anche utilizzare test black box in questo caso). Si utilizzanno anche metodi agili per testare il codice durante la scrittura di questo.
Esistono inoltre strumenti che permettono la **generazione automatica di test** e l'esecuzione automatica di questi. Per Java un software di test utilizzato è la **JUnit**.
Poiché non abbiamo bisogno obbligatoriamente del codice per scrivere i test, possiamo scrivere i test **prima di scrivere il codice**: questo permette di guidare la programmazione tramite i test. Si scrivono prima i test in base alla specifica, e man mano che si implementano le funzionalità si modifica il codice in virtù degli esiti. Questo approccio è un approccio intelligente e prende il nome di **Test Driven Development** (**TDD**).
Si può anche specificare il comportamento del sistema in termini di interazione e guidare la produzione di codice in virtù del comportamento del software da produrre. Questo approccio prende il nome di **Behaviour Driven Development** (**BDD**).

Environment

L'utilizzo di un environment permette di isolare un modulo software simulando l'interazione con altre componenti. Risulta necessario rendere l'ambiente **stabile**, ossia:

- fornire valori consistenti per rendere i test ripetibili;
- fornire un ambiente controllato per testare comportamenti attesi e inattesi.

Per l'environment si costruiscono dei fantocci per un modulo. Si parla generalmente di **driver** e di **stub**. Un driver è un oggetto che sostituisce un altro oggetto nel sistema; ha il ruolo di simulare l'ambiente per il modulo da testare. Lo stub invece è un altro modulo che veniva utilizzata dal modulo oggetto di test. In generale:

- moduli che dipendono dall'oggetto di test sono i driver;
- moduli da cui dipende l'oggetto di test sono gli stub.

Integrazione del sistema e testing

Per il testing del sistema e dell'interazione tra i moduli si possono seguire due approcci: **top-down** e **bottom-up**.
Generalmente se abbiamo un sistema con diversi moduli, si comincia a sviluppare i moduli che non presentano dipendenze: questi moduli possono essere sviluppati in maniera del tutto indipendente e non richiedono dipendenze.
È anche possibile testare un sistema dall'alto verso il basso, possibile quando non si hanno strutture di dipendenze ad albero.
Solitamente si preferisce uno sviluppo e un testing bottom-up.

Automazione dei Test, JUnit

Strumenti quali JUnit e TestNG vengono utilizzati per l'automazione del testing. Altri software, come Randoop, EvoSuite e Ttuples possono essere utilizzati per la generazione di test. Utilizzeremo adesso JUnit.

Per scrivere classi per i test, ogni metodo viene annotato con la flag `@Test`. Ogni metodo di test viene trovato ed eseguito automaticamente dal framework utilizzando la riflessione computazionale. Per confrontare l'output con quello atteso, si utilizzano le **asserzioni**: la valutazione viene automata e il riscontro viene analizzato da JUnit.

La classe `Assert` fornisce i metodi usati per la valutazione dei test:

- `assertTrue(boolean condition)` verifica se `condition` è `true`;
- `assertEquals(expected, calculated)` verifica se `calculated` è uguale a `expected`. In caso contrario viene generata un'eccezione. `assertEquals` ammette un terzo parametro opzionale `delta` per una tolleranza di valore (ossia verifca se tra `expected` e `calculated` vi è una differenza minore o uguale a `delta`);
- `fail(String message)` viene utilizzato per testare le situazioni di eccezione e di failure del programma.

Poiché i metodi di test possono essere molti, è meglio avere un nome significativo per metodo di test. Per ogni test è meglio scrivere un metodo a parte.

Quando eseguiamo i test, per ogni metodo di test eseguito la classe viene istanziata nuovamente, in quanto i metodi vengono eseguiti in isolamento in un ambiente consistente. Ogni test lavora su istanze differenti della classe in cui è definito.

Esistono alcune annotazioni che possiamo utilizzare per creare delle **test suite**:

```
@RunWith(Suite.class)
@Suite.SuiteClasses({ConsumerTest.class, SumTest.class})
public class TestSuiteA{}

@RunWith(Suite.class)
@Suite.SuiteClasses({ConsumerTest.class, SumTest.class})
public class TestSuiteB{}

@RunWith(Suite.class)
@Suite.SuiteClasses({TestSuiteA.class, TestSuiteB.class})
public class MasterTestSuite{}
```

Notiamo nell'esempio anche che è possibile raggruppare test suite per formare una test suite di livello superiore.

Altre annotazioni importanti per i metodi sono:

- `@Before`, che annota un metodo in modo tale che esso venga chiamato prima di ogni test. È utile quando bisogna inizializzare lo stato di istanza della classe di test, così che tutti i test partano dalle stesse condizioni;
- `@After`, che annota metodi chiamati dopo l'esecuzione di ogni test. Viene usato per operazioni di chiusura o finali;
- `@BeforeClass` annnota un metodo statico da eseguire prima del lancio di test di una classe. A differenza di `@Before`, il metodo viene eseguito **una volta sola per classe di test**.
- `@AfterClass` annota un metodo statico da eseguire dopo il lancio di test di una classe con le modalità di esecuzione di `@BeforeClass`.

Utilizzare le flag senza `Class` permette di resettare l'environment dopo o di impostarlo prima di ogni metodo di test. Con `Class`, i metodi sono eseguiti una tantum.

Mock Objects

Poiché i test di unità devono testare la correttezza di una singola classe, si utilizzano degli **oggetti fantoccio**, ossia degli oggetti finti che restituiscono valori specifici in virtù del test. Questi oggetti implementano la stessa interfaccia del vero collaboratore, ma il comportamento dei metodi restituisce un risultato predefinito come uno stub.

In Java la libreria Mockito semplifica la scrittura di test con le annotazioni `@Mock`.

16 - Testing, 2

Tipologie di doppioni di test

Parliamo di **test doubles**, ossia di **doppioni di test** o **controfigure**, per indicare le componenti che sono simulate all'interno del sistema:

- **Dummy objects**. Gli oggetti **dummy** sono passati come oggetti per riempire i parametri di funzione, ma non sono realmente usati.
- **Oggetti finti**. Gli **oggetti finti** hanno un'implementazione funzionante, ma servono solo per implementare un'interfaccia necessaria.
- **Stubs**. Gli stub sono definiti in modo tale che i metodi di uno stub restituiscono il valore di ritorno atteso. Non rispondono alla reale funzionalità richiesta, ma semplicemente al valore che è atteso dalla computazione del modulo software analizzato.
- **Spie**. Gli oggetti spie sono oggetti stub che raccolgono delle informazioni basate su come questi vengono chiamati. Ad esempio, un servizio email potrebbe registrare quanti messaggi sono stati inviati.
- **Mock objects**. Gli oggetti mock sono oggetti che sono programmati in modo tale da sapere quante volte e in quale ordine le funzioni devono essere chiamate durante i test.

Mock e stub

Gli stub servono a fornire dei dati di ritorno alle chiamate della classe testata. Si decide l'esito dei test con delle asserzioni che verificano lo stato della classe testata.

I mock invece sono oggetti finti che controllano il comportamento delle classi sotto test. Le interazioni della classe sotto test sono tracciate con il mock, che analizza il numero, l'ordine e i valori dei parametri delle chiamate ricevute durante il test. Le asserzioni esaminano lo stato del mock, e non la classe testata, che invece comunica con la classe mock al fine di fornire le statistiche per l'oggetto mock.

Il tipo di testing in entrambi i casi è un **behavioural testing**.

Behavioural Driven Development

Il criterio di accettazione per validare le funzionalità sotto un test è uno **scenario** che segue la struttura:

- **given**, che indica il contesto iniziale dello scenario in una o più clausole. Si specifica cosa dev'essere restituito dai metodi mock quando vengono invocati;
- **when**, che indica l'evento che attiva lo scenario. La classe testata viene usata e usa in cascata il mock object;
- **then**, che indica il risultato atteso analizzando asserzioni sullo stato delle interazioni.

Mockito

La libreria Mockito permette di semplificare la scrittura dei test tramite Mock Objects. Utilizza una sintassi compatta:

```
@Mock
Collaborator mockObj;
//in alternativa...
Collaborator mockObj = mock(Collaborator.class);
```

Mockito permette di:

- indicare direttamente durante l'esecuzione del test quale valore restituire quando un metodo del mock è invocato;
- verificare con quali input un metodo del mock sia stato chiamato durante l'esecuzione del test.

Esempio

```
Seller seller = mock(Seller.class);
Shop shop = new Shop(seller);

public void shouldBuyBread() {
    //given
    given(seller.askForBread()).willReturn(new Bread());
```

```
    //when
    Goods goods = shop.buyBread();
    //then
    assertThat(goods, containBread());
}
```

Il mock object `seller` viene impostato in modo tale che quando viene invocato il metodo `askForBread()` si restituisca una nuova istanza di `Bread`. L'evento che attiva lo scenario è il metodo `buyBread()` di `shop`. Si verifica infine se `goods` soddisfa la condizione implementata all'interno di `containBread()`.

Si prevede anche l'utilizzo di un metodo `then()`, che valuta se un metodo viene effettivamente chiamato con quei valori:

```
given(summer.sum(a,b)).willReturn(5);

int res = summer.sumTwice(a,b);

then(summer).should().sum(a, b); // verifica che il metodo viene chiamato con quei valori
assertThat(res).isEqualTo(10);
```

Scrivere i test prima della codifica del software è un approccio migliore. Per verificare invece quante volte è stato chiamato un metodo, si utilizza come parametro di `should()` il metodo `times(n)`, dove con `n` si indica il numero di volte in cui il metodo dev'essere invocato. Esistono inoltre:

- `shouldHaveNoMoreInteractions(obj)`, che specifica che non bisogna avere più nessun'altra chiamata;
- `shouldHaveZeroInteractions(obj)`, che specifica che non ci devono essere chiamate a metodi di un oggetto.

Test parametrici

Poiché per eseguire i test bisogna campionare il dominio di input, vorremmo automatizzare il test su elenchi di dati molto ampi. Si definisce allora un set di dati, dove ogni dato è inserito in un array. Il set di dati statico viene annotato con `@Parameters`:

```
@Parameters
public static Collection<Integer[]> getParam() {
    return Arrays.asList(new Integer[][] {
        {1, 1, 2},
        {3, 2, 5},
        {4, 3, 7}
    });
}
```

I parametri vengono passati ad ogni metodo di classe di test. I test prevedono, in questo caso, tre parametri.

White-box testing e Black-box testing

Le principali differenze tra white box e black box testing sono illustrate nella tabella sottostante.

Black-Box	White Box
I test dipendono solo dalle specifiche	I test dipendono dall'implementazione e dalle specifiche
I test possono essere riusati se il codice viene aggiornato	I test sono invalidati nel momento in cui modifichiamo il codice
I test possono essere sviluppati prima che il codice viene scritto	I test possono essere sviluppati solamente dopo la scrittura del codice
I test non richiedono che tutto il codice venga eseguito. Possono tuttavia non controllare gli **errori di commissione**	I test non assicurano che il codice implementi tutte le specifiche. I test non controllano quindi gli **errori di omissione** (manca la funzionalità nel codice)
Alcuni strumenti forniscono misurazioni di copertura di test black box	Molti strumenti forniscono misurazioni di copertura di test white-box

Gli strumenti si completano, pertanto si utilizzano entrambe le tipologie di test, partendo dai test black box e procedendo con test white box.

Black Box Testing

Si inizia solitamente con test black box. Le tecniche più comuni per fare test black box sono tre:

- **Equivalence Partitioning**. Si partiziona il dominio di ingresso, e si rappresenta ogni partizione con un singolo test rappresentativo della partizione;
- **Boundary Values Analysis**. In questo modo si verifica la correttezza dell'operazione ai confini degli intervalli del dominio;
- **Combinatorial Interaction Testing**. Se le specifiche prevedono processamenti differenti in virtù dell'input, si eseguono test con diverse combinazioni di input.

White box testing

Il white box testing misura la copertura di statement e di branch per i test black box, permettendo il testing di tutte le parti del software da testare e migliorando quindi la fase di validazione. Si utilizzano le seguenti tecniche:

- **Statement Coverage**. Si utilizza lo statement coverage per verificare che il 100% degli statement viene eseguito;
- **Branch Coverage**. Si utilizza il branch coverage per verificare che il 100% di tutti i branch viene attraversato (in un flowchart, questo corrisponde a percorrere tutti gli archi);
- **All Paths**. Si controllano tutti i percorsi di esecuzione;
- **Definition-Use Pairs**. Si testa il codice nel caso in cui esso contenga dei pattern per l'uso di dati;
- **Condition Coverage**. Si occupa di coprire le decisioni all'interno del codice.

Fault model (definizione generica)

Un **fault model** è un modello di qualcosa che può andare storto. Per ogni tipo di test, esiste un fault model che permette di prevedere le conseguenze di un determinato fault.

Equivalence partition

Identifichiamo il dominio di ingresso e lo dividiamo in partizioni. Questo potrebbe richiedere come rappresentare i nostri parametri. Se i parametri sono di tipo base, possiamo utilizzare dei range comuni, quali valore minimo e valore massimo. Se tuttavia abbiamo tipi che non presentano un ordine naturale, dobbiamo trattare ogni singolo valore come un range separato. Oppure si potrebbero avere tipi come tipo stringa, e in generale il partizionamento dipende dalla specifica: bisogna infatti capire come la stringa utilizzata influenza il risultato dell'elaborazione. Nel caso più comune, se aspettiamo una stringa che rappresenta un qualunque elemento, distinguiamo stringhe ben formate da stringhe malformate e dalla stringa nulla.

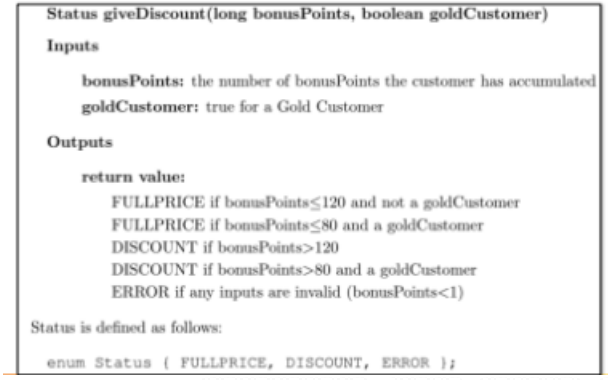
Come dominio di input non abbiamo solamente i parametri espliciti del metodo invocato, ma bisogna anche far riferimento agli attributi statici.

Una volta aver identificato il dominio e averlo diviso, si sceglie un valore per partizione. Ogni partizione per ogni input e output è un **oggetto di test coverage**. Solitamente si fornisce un identificatore unico, utilizzando come prefisso EP. Si individuano inoltre quali oggetti di test coverage prevedono fallimenti nel sistema. Se il test tuttavia fallisce e al suo interno quello che aveva stimolato erano due possibili esecuzioni in cui il sistema poteva fallire, non si può individuare correttamente quale test case ha fatto fallire il sistema. Pertanto ogni test coverage item che contiene input di errore viene testato separatamente.

Si scrivono quindi tanti test case quanti sono i test item in virtù delle specifiche del sistema e dei metodi da invocare. Il numero di test scritti dev'essere tale che esiste almeno un test per EP: in questo caso diremo che la copertura dei test è pari al 100%. Alcuni test inoltre possono essere ridondanti e pertanto possono essere eliminati.

Esempio

Supponiamo di voler testare una funzione `Status giveDiscount(long bonusPoints, boolean goldCustomer)` con le seguenti specifiche:



Status giveDiscount(long bonusPoints, boolean goldCustomer)

Inputs

bonusPoints: the number of bonusPoints the customer has accumulated
goldCustomer: true for a Gold Customer

Outputs

return value:
FULLPRICE if bonusPoints≤120 and not a goldCustomer
FULLPRICE if bonusPoints≤80 and a goldCustomer
DISCOUNT if bonusPoints>120
DISCOUNT if bonusPoints>80 and a goldCustomer
ERROR if any inputs are invalid (bonusPoints<1)

Status is defined as follows:

```
enum Status { FULLPRICE, DISCOUNT, ERROR };
```

Per `bonusPoints` possiamo allora ottenere le partizioni `MIN_VALUE, 0`, `1, 80`, `81, 120`, `120, MAX_VALUE`. Per `goldCustomer` e il valore di ritorno (che è un'enumerazione) utilizziamo semplicemente i valori discreti possibili.
Ogni oggetto di test coverage è identificato da una scelta di valore per partizione:

TCI	Parameter	Equivalence Partition	Test Case
EP1*	bonusPoints	Long.MIN_VALUE..0	To be completed later
EP2		1..80	
EP3		81..120	
EP4		121..Long.MAX_VALUE	
EP5	goldCustomer	true	
EP6		false	
EP7	Return Value	FULLPRICE	
EP8		DISCOUNT	
EP9		ERROR	

Eseguiamo allora dei test combinando i vari TCI, **cercando di minimizzare il numero di test da scrivere**. Nel nostro caso possiamo testare le funzionalità con quattro test:

ID	TCI Covered	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T1.1	EP2,5,7	40	true	FULLPRICE
T1.2	EP3,6,[7]	100	false	FULLPRICE
T1.3	EP4,[6],8	200	false	DISCOUNT
T1.4*	EP1*,9	-100	false	ERROR

Nel processo cerchiamo anche di evitare la scrittura di test duplicati, ossia test che coprono gli stessi TCI.

Fault model di EP

Il fault model dell'equivalence partitioning modella range di valori che non vengono processati correttamente. Testando almeno un valore da ogni partizione di equivalenza, in modo tale che **ogni valore dello stesso range sia processato nella stessa maniera**, si possono trovare i fault di questo tipo.

Boundary Values Analysis

Il test EP non riesce tuttavia a individuare errori relativi a valori che si trovano **agli estremi di una partizione**. Eseguiamo allora dei test **Boundary Value Analysis**.

Nell'esempio illustrato precedentemente, i valori 0, 1, 80, 81, 120, 121, `MIN_VALUE` e `MAX_VALUE` devono essere testati. Allora introduciamo altri TCI, etichettati con `BV`. Otteniamo allora i seguenti test:

ID	TCI Covered	Inputs		Exp. Results
		bonusPoints	goldCustomer	return value
T2.1	BV3,9,11	1	true	FULLPRICE
T2.2	BV4,10,[11]	80	false	FULLPRICE
T2.3	BV5,[10,11]	81	false	FULLPRICE
T2.4	BV6,[10,11]	120	false	FULLPRICE
T2.5	BV7,[10],12	121	false	DISCOUNT
T2.6	BV8,[10,12]	Long.MAX_VALUE	false	DISCOUNT
T2.7	BV1*,13	Long.MIN_VALUE	false	ERROR
T2.8	BV2*,[13]	0	false	ERROR

Statisticamente si mostra come la BVA sia in grado di trovare più errori rispetto alla EP. Poiché inoltre per ogni range si può individuare un minimo ed un massimo, la BVA fa testing di due valori aggiuntivi per ogni partizione.

Combinatorial Testing

BVA non è in grado di esplorare decisioni associate a combinazioni di input. Si utilizza allora il **Combinatorial Testing**.
Il Combinatorial Testing cerca di analizzare tutte le combinazioni di input. Tuttavia, questo può portare ad un grande numero di combinazioni. Supponendo di avere 20 variabili, ognuna con 10 valori ciascuno possibili, si ha un numero totale di test pari a 10^{20}.

Invece di provare tutte le possibili combinazioni dei parametri, possiamo ridurre l'ampiezza delle combinazioni, ad esempio decidendo di fare test di tutte le coppie di parametri. Questo perché è stato osservato che in quasi tutte le applicazioni il 60% dei fault è attivato dall'interazione di coppie di parametri. La percentuale si avvicina al 100% quando interagiamo con 6 parametri. Pertanto si può utilizzare un criterio di testing che prende il nome di **pairwise testing**, ossia si generano suite di test per coprire le combinazioni di coppie di parametri. Il ragionamento del pairwise testing permette di semplificare il numero di test. Ad esempio, avendo 5 parametri che possono avere rispettivamente 3,2,2,2,3 valori distinti, si ha un numero totale di test pari a $3^2 \cdot 2^3 = 72$. Con il pairwise testing, possiamo ridurre questo numero a 10. Si dice che il pairwise testing è un tipo speciale di **t-way coverage testing**, in cui $t = 2$. Quindi per quanto detto precedentemente, basta scegliere $t \in [2, 6]$ per ottenere un numero sufficiente di test.

Per il testing combinatorio, avendo una forza t di coverage testing, v valori possibili per parametro e k parametri è necessario costruire un **covering array** di dimensione N, che non è altro che una matrice $N \times k$ con simboli provenienti da un alfabeto di v simboli tale che in ogni sottomatrice $N \times t$ si ha che una qualsiasi combinazione di t elementi dell'alfabeto di v simboli è coperto almeno una volta. Ad esempio, supponiamo di voler testare 4 parametri binari con un test di forza 2. In teoria avremmo un numero di test pari a $2^4 = 16$ test da fare. Tuttavia, visto che ci interessano le coppie di parametri, basta considerare le coppie binarie 00, 01, 10, 11. Invece di considerare il set `0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001, 1010, 1011, 1100, 1101, 1110, 1111`, basta considerare il set `0000, 0111, 1011, 1101, 1110`. Passiamo quindi da 16 test a 5. In questo caso cerchiamo un valore N tale che N sia minimo, per ottenere quindi un covering array minimo. Nell'esempio proposto, quindi, $N = 5$ è minimo e peremtte di costruire la tabella 5×4:

Test	A	B	C	D
1	0	0	0	0
2	0	1	1	1
3	1	0	1	1
4	1	1	0	1
5	1	1	1	0

Notiamo come prendendo una qualsiasi sottomatrice 5×2 tute le combinazioni di due elementi dell'alfabeto binario $\Sigma = \{0, 1\}$ sono coperte.

Il problema risiede nel fatto che il calcolo di un covering array minimo è **NP-completo**, ossia non si ha un algoritmo che in tempo polinomiale generi il covering array, ma si ha un algoritmo che in tempo polinomiale verifica se il covering array è minimale.

Statement Coverage

Lo statement coverage permette di verificare se ogni riga di codice **eseguibile** viene eseguita durante i test, verificando intanto che l'output è corretto. Ogni statement eseguibile nel codice è un TCI.

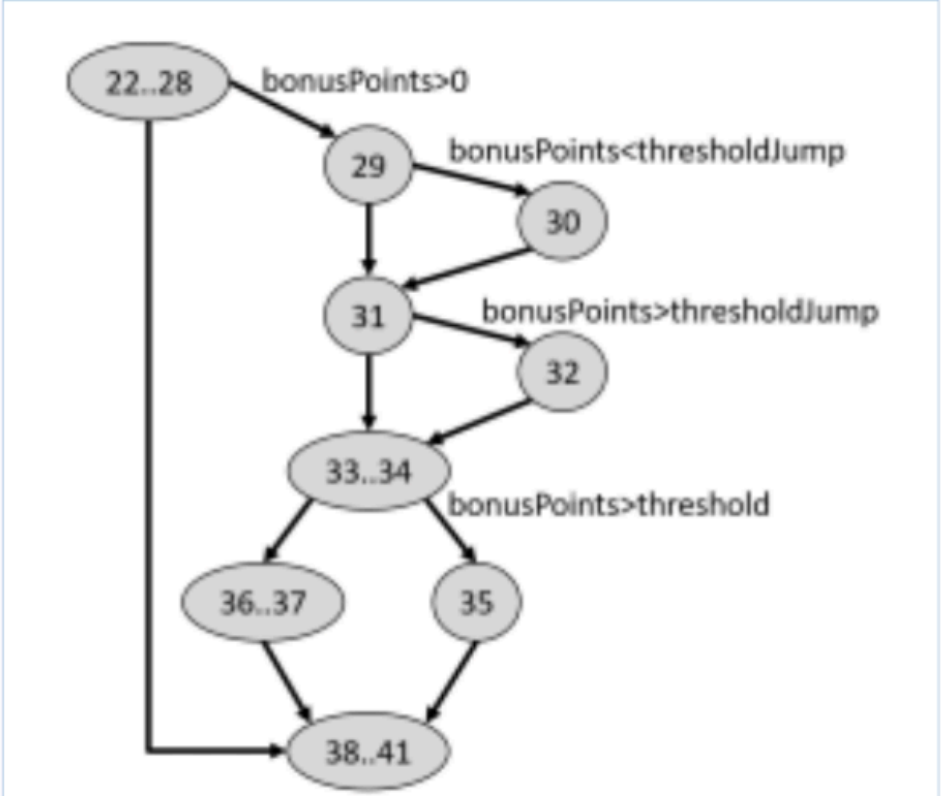
Si costruisce un grafo che descrive tutti i possibili flussi di esecuzione dell'algoritmo da testare. Si controlla manualmente tramite i TCI già coperti quali sono le parti eseguite.

Grafo di controllo di flusso

Il grafo di controllo di flusso è un grafo in cui ogni percorso da un estremo ad un altro rappresenta il flusso di esecuzione di un metodo o di un programma. Il grafo rappresenta un modello astratto dell'algoritmo in termini di flusso di esecuzione, e permette il calcolo della **complessità ciclomatica**. Ogni `if` è una biforcazione, mentre ogni `loop` è un ciclo nel grafo.

La complessità ciclomatica indica quanto è complesso un algoritmo. La complicazione cresce quando cresce il numero di branch all'interno dell'algoritmo. Questo è indipendente dal fatto che si distinguono `if` e loop. La complessità ciclomatica si calcola tramite il calcolo numero di archi - numero di nodi + 2, ed **è il numero minimo di test, ossia di percorsi distinti, per percorrere il grafo nella sua interezza**. Ma questo **non è il numero totale di percorsi di flussi**, ossia il numero di percorsi distinti.

```
22.   public static Status giveDiscount(long bonusPoints, boolean goldCustomer)
23.   {
24.       Status rv = FULLPRICE;
25.       long threshold=120;
26.
27.       if (bonusPoints<=0)
28.          rv = ERROR;
29.
30.       else {
31.          if (goldCustomer)
32.             threshold = 80;
33.          if (bonusPoints>threshold)
34.             rv=DISCOUNT;
35.       }
36.
37.       if (bonusPoints==43) // fault 4
38.          rv = DISCOUNT;
39.
40.       return rv;
41.   }
42.
43. }
```



Per la costruzione di un grafo di controllo di flusso, si può utilizzare **JaCoCo** (Java Code Coverage). JaCoCo fornisce un output in XML o HTML, e permette di identificare quali istruzioni sono state coperte o meno.

Fault model di Statement Coverage

Il fault model che è dietro lo statement coverage è l'individuazione degli errori legati al comportamento del sistema in generale. Inoltre, questi test sono in grado di notare la presenza di **codice morto** o di **statement non utilizzati**.
Questa idea tuttavia non è la più potente e alcuni errori possono restare in quanto i test nel complesso coprono gli statement, ma non hanno come obiettivo l'esecuzione completa di un percorso completo di esecuzione di un algoritmo. Per testare tutti i flussi di esecuzione, si utilizza il **branch coverage**.

Branch coverage

Il problema dello Statement Coverage risiede, oltre nel capire quali input utilizzare, nel capire come si può testare il codice in contesti inusuali. Inoltre, **non si copre il caso in cui si ha un `else` nullo**:

```
if(number < 3)
    number++
```

Lo statement coverage non copre un test case per valori che sono maggiori o uguali a 3. Per risolvere questo problema, si utilizza il **branch coverage**.
I branch che non sono stati coperti dai precedenti test potrebbero contenere dei fault. Allora si utilizza il branch coverage per coprire i casi rimanenti, analizzando i flussi di esecuzione e facendo si che ogni branch sia eseguito.

Ad esempio, dato lo snippet:

```
if(x && y)
    z = 10;
else
    z = 20;
```

Abbiamo due **branch decisionali**:

- un branch se la condizione è verificata;
- un branch se la condizione non è verificata.

In totale abbiamo **4 branch basati sulle condizioni**:

- se la condizione `x` è vera si passa a `y`;
- se la condizione `x` è falsa si passa alla riga 4;
- se la condizione `y` è vera si passa alla riga 2;
- se la condizione `y` è falsa si passa alla riga 4.

Il fault model in questo caso riguarda la copertura di branch non coperti da precedenti test che potrebbero contenere un errore. Come per lo statement coverage, in questo caso i fault si verificano in condizioni non usuali.

All-Path coverage

La forma più potente di white-box testing è la **All-path coverage**.
Per applicare l'all path coverage necessitiamo del grafo di controllo di flusso. Si enumerano **tutte le combinazioni** all'interno del grafo. Passare ad un determinato percorso significa passare a determinate condizioni in cui i valori coinvolti potrebbero avere assunto assegnamenti non compatibili. Quindi **non è detto che tutti i percorsi possano essere eseguiti**. Questo controllo tuttavia va fatto manualmente.

Il fault model del coverage all-path riguarda sequenze di operazioni (rappresentate come percorsi nel codice) che non producono il risultato corretto. Questi fault sono associati a strutture di codice dove la funzionalità scorretta è eseguita per una situazione specifica. Si trovano quindi errori funzionali che si manifestano solo con l'esecuzione di una sequenza più lunga di un singolo branch.

L'all-path coverage tuttavia:

- è un processo lungo;
- non valuta tutte le condizioni booleane in ogni decisione;
- **non esplora fault dovuti a processamenti errati di dati o errori non dipendenti dal codice**.

Dataflow coverage /Definition-Use pairs, Condition and Decision coverage

Varianti del branch coverage sono:

- **dataflow coverage** (o **definition-use pairs**): si coprono tutti i percorsi tra la definizione di un dato e il suo primo utilizzo (per questo si chiama definition-use: si considerano le coppie definizione del dato - primo utilizzo) Si tratta di un criterio di fault che cerca errori nel modo in cui si utilizzano i dati. Non si tratta quindi di un'invocazione sbagliata, ma è sbagliato il modo in cui si utilizza un valore (i criteri visti precedentemente non analizzano i valori). Si tratta di un test molto forte ma che potrebbe portare ad un numero di test molto grande;
- **condition coverage** (**CC**): si esegue un test per ogni condizione booleana. Non garantisce che la decisione è vera o falsa, ma solo la condizione booleana. Inoltre può essere difficile stabilire i parametri di input richiesti;

- **decision coverage** (**DC**): si esegue un test per ogni decisione. Equivale ad un branch coverage se la misurazione della copertura utilizza le decisioni come branch;
- **decision/condition coverage** (**DCC**): i test sono generati per ogni decisione e per ogni condizione, senza testare tutte le possibili combinazioni delle condizioni;
- **multiple condition coverage** (**MCC**): i test sono generati per testare tutte le possibili combinazioni di test booleani. Si hanno tuttavia molti TCI, in quanto per n condizioni booleane si hanno 2^n TCI;
- **modified condition/decision coverage** (**MCDC**): il numero di test di MCC viene ridotto per considerare solo le combinazioni di condizioni booleane per causare un effetto visibile sull'output. Per capire meglio consideriamo lo snippet sottostante:

```
int x = 100;
if ( (a > 10) || (b == 0) ) then x++;
return x;
```

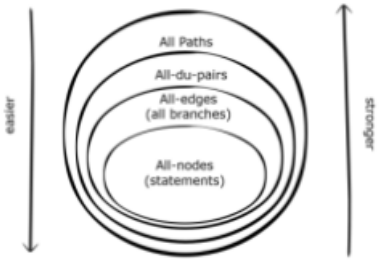
I test per lo snippet devono considerare le condizioni per i casi `true` e `false`:

- `a > 10`;
- `b == 0`.

I test inoltre devono considerare la decisione `( (a > 10) || (b == 0) )` per i casi `true` e `false`. Infine, i test devono analizzare l'effetto sul valore di output nel cambiamento del valore delle condizioni booleane.

Ranking dei test

Il ranking dei test può essere riassunto con il grafico sottostante:



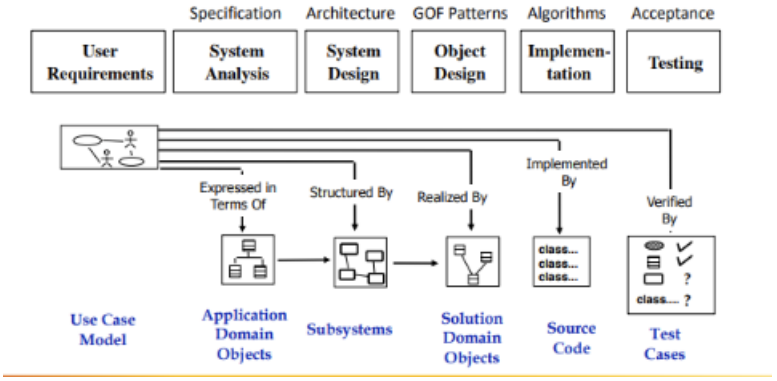
17 - Modeling e design

Modeling e Design

La prima attività che viene eseguita nella progettazione del modulo software è l'**analisi dei requisiti**. L'attività di analisi dei requisiti è la più critica, in quanto individuiamo quali sono i pezzi del nostro sistema e come sono relazionati tra di loro.
Nell'analisi dei requisiti del sistema realizziamo tre tipi di **modellazione**:

- una **modellazione funzionale**, che consiste nell'analisi del sistema a livello di cosa deve fare il sistema. Il risultato di questa modellazione è un insieme di diagrammi use case;
- una **modellazione a oggetti**, che consiste nell'analisi del sistema a livello di struttura, individuando le entità del sistema, i loro tipi e come queste siano connesse tra loro. Tale modellazione produce diagrammi di classe;
- una **modellazione dinamica**, che consiste nell'analisi del sistema a livello di reattività del sistema a eventi esterni, e come gli eventi vengano gestiti nel sistema. Tale modellazione porta alla produzione di diagrammi di sequenza, di attività e di stato.

Raccolte le informazioni funzionali, sugli oggetti e dinamiche, abbiamo ancora un'analisi formale del sistema: abbiamo infatti solo una collezione di informazioni. Lo step successivo è quello di definire com'è strutturato il sistema. Definiamo quindi uno schema complessivo del sistema, ossia un'**architettura** del sistema. Utilizziamo la **modularità** per creare un'organizzazione del sistema in parti relazionate tra loro. Questo scheletro non è solamente uno scheletro e basta, ma abbiamo anche definito il perché sono stati assemblati i pezzi del sistema in un determinato modo.
Per rendere funzionante il sistema, risulta necessario spesso anche aggiungere degli oggetti dovuti ad una successiva **object design**, ossia classi che hanno senso in virtù dell'implementazione che andremo a scrivere. Nella fase di object design applichiamo i design pattern visti.
Ottenuti i documenti che descrivono la relazione tra gli oggetti, i metodi e gli attributi di questi, siamo in grado di passare alla fase di **implementazione** di questi oggetti. Successiva all'implementazione, infine, è la fase di **testing**, utilizzando gli strumenti che abbiamo visto precedentemente.



Analisi dei requisiti

L'analisi dei requisiti prevede due step:

- **requirements elicitation**: si utilizza un linguaggio del dominio, ossia il vocabolario utilizzato è un vocabolario che viene compreso dal cliente. Il sistema viene definito in un linguaggio non tecnico e porta alla produzione del **documento dei requisiti**. Si tratta di una fase che richiede collaborazione tra gli utenti, che hanno una conoscenza relativa al dominio applicativo, e gli sviluppatori, che invece hanno una conoscenza relativa alla soluzione da sviluppare.
- **requirements analysis**: si utilizza un linguaggio prettamente tecnico, definendo le specifiche tecniche del sistema con un vocabolario compreso dallo sviluppatore, concependo quindi le **specificazioni del sistema**, ossia un modello dei requisiti.

Possiamo distinguere due tipi di requisiti:

- i requisiti **funzionali**, che sono dei requisiti che descrivono cosa deve fare il sistema, specificando le funzionalità del sistema da sviluppare, il comportamento e le azioni da intraprendere all'interno di quest'ultimo;
- i requisiti **non funzionali**, che sono dei constraint di qualità che il sistema deve soddisfare in virtù del progetto da sviluppare. In generale sono dei requisiti che riguardano come una certa elaborazione deve avvenire, e possono includere dei requisiti tecnici.

Oltre ai requisiti vogliamo anche avere una pianificazione del sistema da sviluppare, specialmente a livello temporale. Identifichiamo delle **milestone**, ossia dei punti del tempo in cui si prevede il raggiungimento di certi obiettivi (i.e. il completamento di una certa attività o di un punto di sviluppo dell'implementazione).

Una specifica importante è l'**ambiente di deployment del sistema**, ossia quello in cui opera il sistema.
Infine, è necessario anche tener conto dei **criteri di accettazione del cliente**, utilizzati per testare il sistema e vedere se i requisiti sono soddisfatti.

All'interno del documento dei requisiti non sono presenti:

- l'**architettura del sistema**, che rappresenta in realtà la prima risposta ai requisiti funzionali e non;
- la **metodologia di sviluppo**, ossia il tipo di tecnologie che si vanno a utilizzare;
- l'**ambiente di sviluppo**, in termini non solo di macchine utilizzate, ma anche di risorse umane e di competenze necessarie;
- il **linguaggio di implementazione**;
- la **riusabilità**, una qualità su cui bisogna ragionare a livello implementativo.

Modellazione degli oggetti

Per modellare gli oggetti, consideriamo uno use case e analizziamo il flusso di eventi. In particolare, procediamo con un'**analisi logica** dello use case, individuando:

- **nomi ricorrenti**, che possono determinare categorie di elementi di un dominio, ossia i **tipi** di istanze degli oggetti. Tra questi nomi possiamo individuare dei **nomi propri** che possono identificare delle entità precise, rappresentate da oggetti nel sistema;
- **verbi**, che fanno riferimento ad azioni svolte, quindi possono essere richieste di servizio o produzione di dati. Pertanto identificano **procedure** di cui il sistema deve tenere traccia.

Ad esempio, supponiamo di considerare il seguente flusso di eventi. Il cliente entra nel negozio per comprare un giocattolo per un bambino di età n. Il cliente può ricevere assistenza in meno di un minuto, e il proprietario del negozio può dare consigli al cliente in base all'età del bambino. Se il cliente sceglie un giocattolo che può essere pericoloso per un bambino di una certa età, il proprietario deve consigliare un altro prodotto. Possiamo procedere all'analisi testuale dei requisiti. Ad esempio:

- il nome comune "giocattolo" identifica la classe giocattolo;
- il verbo "comprare" indica un metodo;
- il verbo "deve" identifica un constraint;

Generalmente, inoltre, un verbo transitivo identifica un metodo, un verbo intransitivo un evento.

Quando modelliamo un oggetto, tuttavia, bisogna stare attenti alle funzionalità utilizzate dall'oggetto. Ad esempio, supponiamo che a partire dall'analisi logica dei requisiti abbiamo ottenuto una classe `BriefCase`, che modella una valigia. Supponiamo che nei requisiti sia specificata la possibilità di sedersi sull'oggetto che abbiamo modellato come una valigia. La domanda che ora ci poniamo è la seguente: **abbiamo correttamente modellato l'oggetto?**
La domanda sorge per il fatto che se effettivamente l'azione di sedersi è un'operazione ricorrente, e la valigia non viene mai aperta o chiusa, non è realmente una valigia, ma una sedia. Astraendo l'esempio, **abbiamo modellato in maniera errata l'oggetto**, in quanto non si tratta di una valigia. Per identificare correttamente gli oggetti allora è necessario far riferimento all'uso di questi, ossia alle loro **interfacce**: se partiamo dalla descrizione non funzionale, ossia dagli attributi, la modellazione potrebbe essere fuorviante. La modellazione funzionale dei requisit pertanto è fondamentale per validare la modellazione ad oggetti.

Modellazione dinamica

Modellati correttamente gli oggetti, possiamo passare alla **modellazione dinamica**, definendo un modello dinamico del sistema tramite un insieme di diagrammi di stato. Più precisamente, si utilizza un diagramma di stato per ogni classe con importanti proprietà dinamiche. L'obiettivo di questa modellazione è modellare le interazioni tra gli oggetti del sistema e i comportamenti dinamici di ogni singolo oggetto. Utilizziamo allora:

- i diagrammi di attività per descrivere gli scenari;
- i diagrammi di interazione / sequenza per modellare le interazioni tra gli oggetti;
- i diagrammi di stato per modellare i comportamenti dinamici di ogni singolo oggetto.

Si presuppone di aver eseguito una **decomposizione** degli elementi del sistema e delle funzionalità di questo. Il primo lavoro con l'analisi dei requisiti è la decomposizione.
In primis effettuiamo una **decomposizione funzionale**. Il goal principale viene diviso in funzioni più semplici. Ad ogni funzione corrisponde un **livello di astrazione**; inoltre, ogni funzione ha sempre la stessa struttura algoritmica:

- ottieni l'input;
- trasforma l'input;

- produci un output.

Quando scomponiamo le funzionalità, bisogna fermarsi ad un livello di granularità appropriato. Questa granularità corrisponde a quella degli **oggetti del dominio** già identificati. Quindi la decomposizione funzionale deve procedere fino a quando non si incontra la modellazione ad oggetti vista nello step precedente, ossia quando le funzionalità utente sono ad un livello di astrazione corrispondente a responsabilità associabili agli oggetti del sistema.

Decomporre solo funzionalmente, inoltre, non è sufficiente per la modellazione corretta del sistema. Risulta essere necessario effettuare una **decomposizione object-oriented**. Questa decomposizione focalizza l'attenzione sugli agenti che causano le operazioni. Solo in questo modo possiamo raggiungere il livello corretto di granularità.
In questo modo una volta identificate le classi e le funzionalità correttamente, possiamo mappare le funzionalità una per una dentro le classi più appropriate. Questo perché sebbene abbiamo individuato le funzionalità presenti nel sistema in gruppi, non è detto che le categorie di entità presneti nel sistema coincidano con i gruppi di funzionalità.

Complessivamente abbiamo un elenco di funzionalità. Formuliamo uno scenario per ogni funzionalità, identificando use cases del sistema. Identificati gli use cases analizziamo il flusso di eventi, e generiamo dei diagrammi di classi, con i seguenti step:

1. identificazione delle classi tramite analisi testuale;
2. identificazione degli attributi e delle operazioni;
3. identificazioni delle associazioni tra le classi;
4. identificazioni delle molteplicità;
5. identificazione dei ruoli;
6. identificazione dei constraint.

Relazioni nei modelli

Gli strumenti di astrazione e decomposizione ci portano a insiemi di classi. Vogliamo gestire la complessità del sistema mettendo in relazione tra loro queste classi. Tra le più importanti vi è la relazione di **gerarchia**, tra cui:

- gerarchia **part-of**, rappresentabile con diagrammi UML e la notazione composizione / aggregazione;
- gerarchia **is-kind-of**, ossia diagrammi di classe con solo relazioni **is-a**;
- gerarchie **uses**, ossia diagrammi di classe con solo dipendenze d'uso.

Design del sistema

Realizzare il design architetturale del sistema significa ragionare su come organizzare, raggruppare e assemblare i moduli del sistema, e come questi moduli devono essere relazionati tra loro. La fase del design di un sistema produce l'**architettura di un sistema**, che è la definizione del sistema in termini di componenti computazionali e di interazione tra questi.
Nella fase di design, dobbiamo capire:

- come selezionare i moduli;
- come definire le interfacce dei moduli;
- come definire le relazioni di uso dei moduli.

Un modulo dev'essere un'unità autonoma, indipendente. Dobbiamo applicare lo stesso principio che abbiamo applicato alle classi: **massima coesione e minimo accoppiamento**. Quello che vogliamo inoltre è che il modulo funzioni come "classe": quello che inseriamo in un modulo dev'essere protetto dall'esterno, secondo il principio di information hiding. Questo è il **golden principle** definito da Parnas: definiamo cosa vogliamo nascondere e definiamo un modulo intorno a questo. Infine, definiamo i moduli in modo tale che abbiamo un'interfaccia minimale.
La granularità dei moduli è importante, per cui si seguono alcuni principi.

Reuse/Release Equivalence Principle (REP)

La granularità del riuso è la granularità del rilascio. Il riuso richiede un tracciamento dei rilasci da un sistema di controllo dei cambiamenti.

Questo significa che il codice non dev'essere semplicemente utilizzato copiandolo e incollandolo nelle nostre classi, ma dev'essere riutilizzato **includendo il modulo**. Inoltre, non posso riutilizzare nulla che non venga rilasciato: tutto quello che dev'essere riutilizzato dev'essere anche rilasciato. Ad esempio, utilizzando le Collections di Java stiamo riusando questo modulo, il cui rilascio viene tracciato utilizzando il versioning. Quindi la granularità del riuso del modulo è uguale alla granularità del suo rilascio.

Common Reuse Principle (CRP)

Le classi in un package sono riutilizzate insieme. Se si riutilizza una classe nel package, si riutilizzano tutte le classi. Ad esempio, una classe che implementa una collection e l'iteratore corrispondente alla collection vengono inseriti nello stesso package.

Common Closure Principle (CCP)

Le classi in un package devono essere chiuse insieme quando vengono effettuati cambiamento delllo stesso tipo. Un cambiamento che influenza un package influenza tutte le classi nel package.

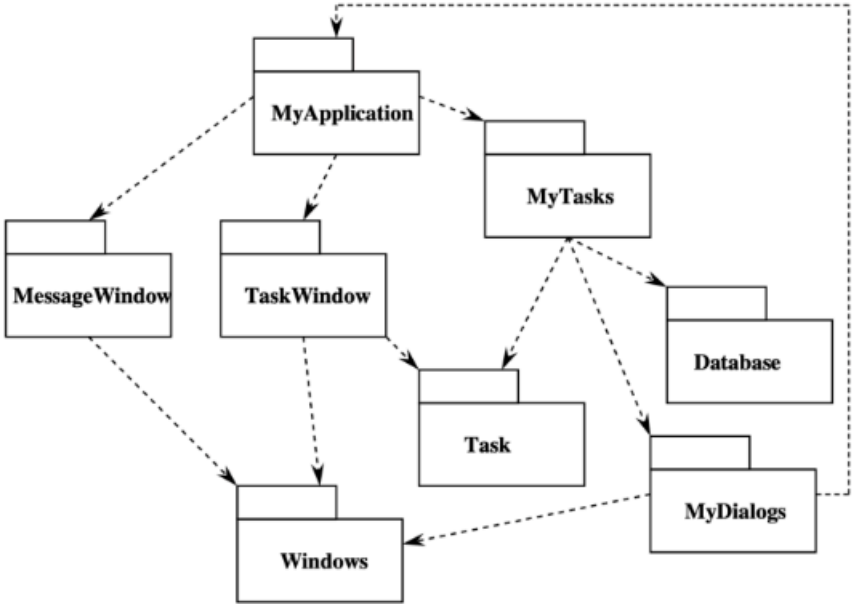
Essenzialmente si raggruppano le classi che cambiano per lo stesso motivo in un unico package, in modo tale che un cambiamento nei requisiti influenza solamente un numero minimo di classi.

Acyclic Dependencies Principle (ADP)

La struttura delle dipendenze tra i pacchetti dev'essere un grafo aciclico, ossia non ci devono essere cicli nelle dipendenze.

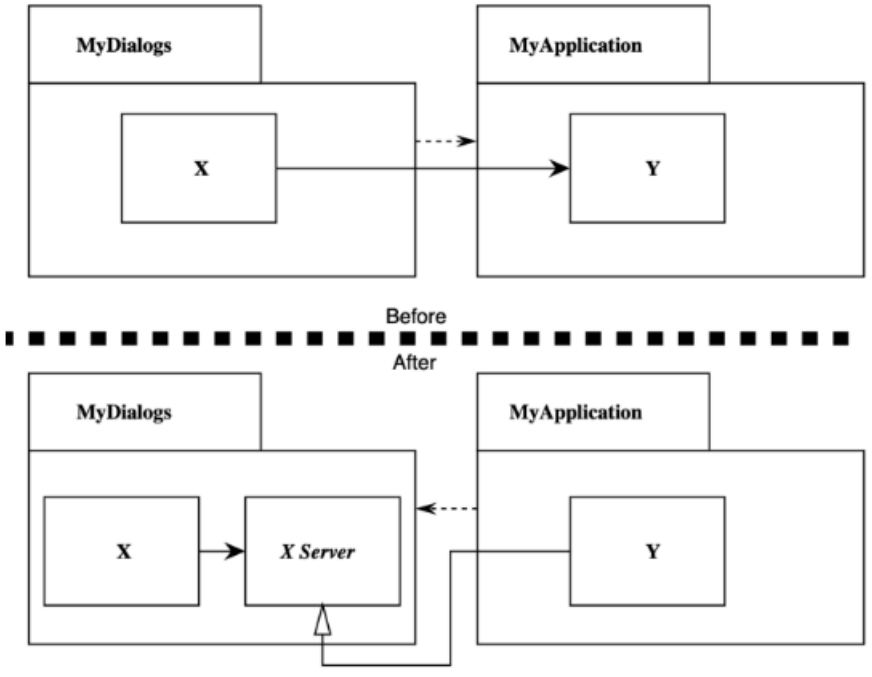
Questo garantisce che per testare un solo modulo del sistema non devo testare tutto il sistema. Un grafo ciclico delle dipendenze quindi indica che quando bisogna eseguire dei test di integrazione è necessario testare l'intero sistema, rendendo impossibile quindi i test di sottoparti del sistema.

Per rompere il ciclo, basta applicare il **Dependency Inversion Principle**: si crea un nuovo package da cui dipendono i moduli che individuano un ciclo e si inserisce in questo package le classi da cui dipendono questi moduli. Ad esempio:



\

La dipendenza da `MyDialogs` a `MyApplication` può essere invertita facendo dipendere `MyDialogs` a una classe interna a `MyDialogs`, che è sottoclasse della classe presente in `MyApplication` come segue:

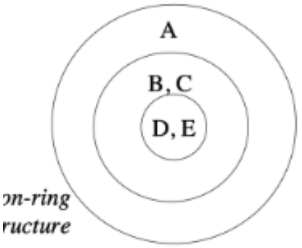


Design styles

Una volta definiti i moduli, vogliamo organizzare questi. Esistono diversi stili.

Layered system

In un sistema organizzato a **livelli** si definisce una gerarchia basata sulla relazione **use**. In particolare, ogni modulo può comunicare solo con i moduli del livello sottostante. Un esempio è la struttura a onion-ring, tipica ad esempio dei primi sistemi operativi o delle virtual machine:



In questo modo i problemi di un modulo restano interni al modulo e impattano solamente i livelli sottostanti, proteggendo i livelli più esterni. Una variante di questo approccio, spesso adottata in architetture web, è la **variante relaxed**, permettendo ai layer di bypassare il meccanismo del layering, ossia un modulo può comunicare con tutti i servizi dei livelli sottostanti, senza limitarsi al solo livello più vicino.

Pipes and filters

I moduli possono essere progettati in modo tale che non si conoscono tra di loro. Sono moduli totalmente disaccoppiati: ogni modulo è un componente che svolge un servizio prendendo un input e dando un output. Delle **pipes**, ossia dei **data flow**, trasferiscono i dati da un modulo ad un altro. Pertanto:

- i componenti sono i **filters**, che non condividono dati tra loro;
- i connettori, ossia le **pipes**, trasportano i dati da un canale di output di un componente a un canale di input dell'altro.

Esistono diversi regimi di controllo per questa architettura:

- **batch**: si processa a lotti, dando un lungo elenco di input in un file. Si processa un blocco di dati per volta;
- **sequenziale**: si processa un input attivando un solo blocco per volta. L'output è determinato da una sequenza finita di passi, per cui è consistente, garantendo anche semplicità di implementazione in quanto non si ha concorrenza;
- **concorrente**: tutti i blocchi sono attivi contemporaneamente e non si aspetta di aver concluso tutti i passi per iniziare a processare l'input successivo. I filtri vengno eseguiti in **parallelo** garantendo maggiore efficienza e lavorando in maniera asincrona.

Poiché i moduli non si conoscono tra di loro, il meccanismo:

- permette naturalmente il riuso;
- permette di modificare in maniera facile il sistema;
- permette di definire il comportamento del sistema in virtù della composizione dei moduli.

Tuttavia, ci sono alcuni svantaggi:

- non si ha persistenza. Questo tipo di architetture consiste nella definizione dinamica della struttura, pertanto l'architettura va assemblata ogni volta;
- se un'operazione dev'essere svolta più volte, l'unica opzione possibile è replicare il filtro, causando replicazione;
- tendenza ad un'organizzazione batch. Un sistema con architettura pipes and filters se usato in organizzazioni non batch permette di avere un flusso di dati che scorre continuamente attraversso i filtri. Un approccio batch obbliga il sistema a processare un blocco di dati per volta, limitanto interattività.

Repository-based system e architettura blackboard

L'architettura **repository-based** può ruotare attorno ad uno strato di **persistenza di dati**. Tutti i componenti interagiscano attraverso una **repository**, e non in maniera diretta tra loro. Un componente viene interessato in virtù di un aggiornamento in repository. Questo tipo di architettura è stato già visto nel caso della programmazione reactive.

In base al tipo di repository distinguiamo:

- il modello **database**, con repository passiva e con un componente aggiuntivo che gestisce le transazioni garantendo coerenza interna;
- il modello **blackboard**, con un repository attivo. Alle modifiche di stato corrispondono letture e scritture su altri componenti. Possiamo inserire all'interno dell'architettura una serie di componenti che dal punto di vista singolo forniscono un pezzo di funzionalità e lasciamo che questi si coordino per risolvere un problema anche molto vasto. Questa architettura permette quindi di risolvere in maniera decentralizzata un problema: non esiste una sequenza di elaborazione, ma essa è dettata dallo stato del repository.

Event-based systems

Un sistema **event-based** è una generalizzazione dell'architettura Model-View-Controller. I componenti interagiscono attraverso notifiche dirette a loro inviate da un **event manager**. Questo facilita le strategie di integrazione: i componenti non dipendono l'uno dall'altro e possono essere aggiunti al sistema facilmente. Tuttavia, si possono avere problemi di scalabilità. Infine, i componenti sono asincroni tra loro: per garantire un ordine degli eventi, si dovrebbero sincronizzare tra loro tutti gli orologi dei componenti.

18 - Metodi di sviluppo

Metodi di sviluppo

Quando il software ha cominciato a crescere sia in complessità sia in presenza nelle applicazioni quotidiane, ci si è resi conto che sviluppare i progetti software in maniera "artigianale" e basandosi sulla capacità del singolo non era affidabile. Si utilizzano oggi dei metodi per minimizzare i rischi nello sviluppo.

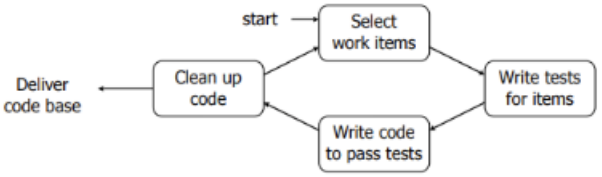
Solitamente il rischio era inteso come la possibilità di "sbagliare" prodotto, ossia che il destinatario del prodotto non era soddisfatto. I motivi tuttavia per cui un progetto può fallire vanno anche oltre errori di questo tipo; tra questi vi sono le carenze nella metodologia per il raggiungimento del risultato finale, come la progettazione dello sviluppo.

Processo

Diamo qualche definizione per i termini tecnici utilizzati.

Definiamo un **processo** come un insieme di **attività orchestrate**, ossia modi in cui si organizzano una serie di attività. Durante queste attività vengono prodotti dei risultati concreti, che possono essere di natura diversa (simulazioni, documenti, codice). Un processo deve essere in grado di identificare i **ruoli** e le **abilità** necessarie per l'esecuzione delle attività.
L'**attività** è un lasso di tempo durante il quale si lavora per la produzione di un risultato concreto a partire da **artefatti** in ingresso.
Per ogni attività è necessario avere un criterio per misurare i progressi. Questo chiaramente è un problema che a livello di processo di traduce nel monitorare i progressi da un'attività ad un'altra.
Distinguiamo un altro tipo di elemento all'interno di un modello da descrivere, ossia l'**evento**. L'**evento** è qualcosa di completamente asincrono, indipendente dalla fine di un'attività.

Esempio: processo di sviluppo iterativo



In questo schema ogni attività è un riquadro. La dipendenza delle attività la rappresentiamo con gli archi.
Lo schema è organizzato come un ciclo. L'idea dietro questo processo di sviluppo è che il lavoro procede in maniera **iterativa**, ripetendo la stessa sequenza di attività più volte. Nello schema la dicitura "Deliver code base" non è un evento. Se sono output o input di un'attività che provengono dall'interno, questi non sono eventi. Il rilascio del codice è un'output di un'attività prodotto dal sistema, e non è in se un evento. L'evento in questo caso è "start", che viene dall'esterno: è un momento asincrono la cui tempistica non è determinabile in relazione agli altri elementi del processo.

Attività

Quando pensiamo ad un'attività, possiamo scoprire diversi tipi di obiettivi. Alcuni tipi di attività sono:

- analisi dei requisiti;
- progettazione (design);
- implementazione (codifica);
- convalida (testing e approvazione);
- rilascio e manutenzione operativa (deployment e maintenance).

Alcuni approcci di sviluppo sono focalizzati sul rilascio del prodotto in tempi ottimali. Ma il vero obiettivo non è solamente produrre un sistema funzionante, ma anche far si che il sistema viva a lungo. La fase di manutenzione è la fase più renumerativa nel tempo. Infatti ad esempio un progetto può essere sviluppato in 3 mesi o più, ma l'azienda che sviluppa il progetto viene pagata per mantenere in vita il progetto.

Classe di processo

In teoria siamo anche in grado di dare indicazioni più astratte, dando **classi di processo**. Ossia, si forniscono metodologie, non dicendo esplicitamente quali attività selezionare, ma fornendo uno stile di sviluppo da seguire. Alcuni esempi sono il modello iterativo e il modello a cascata.

Framework

Definiamo come **framework** il quadro d'insieme in cui alcuni dettagli possono essere liberi. Si ha una specifica parziale dell'impostazione da dare al processo di sviluppo. Alcuni esempi sono le metodologie Scrum e Spiral.

Practice

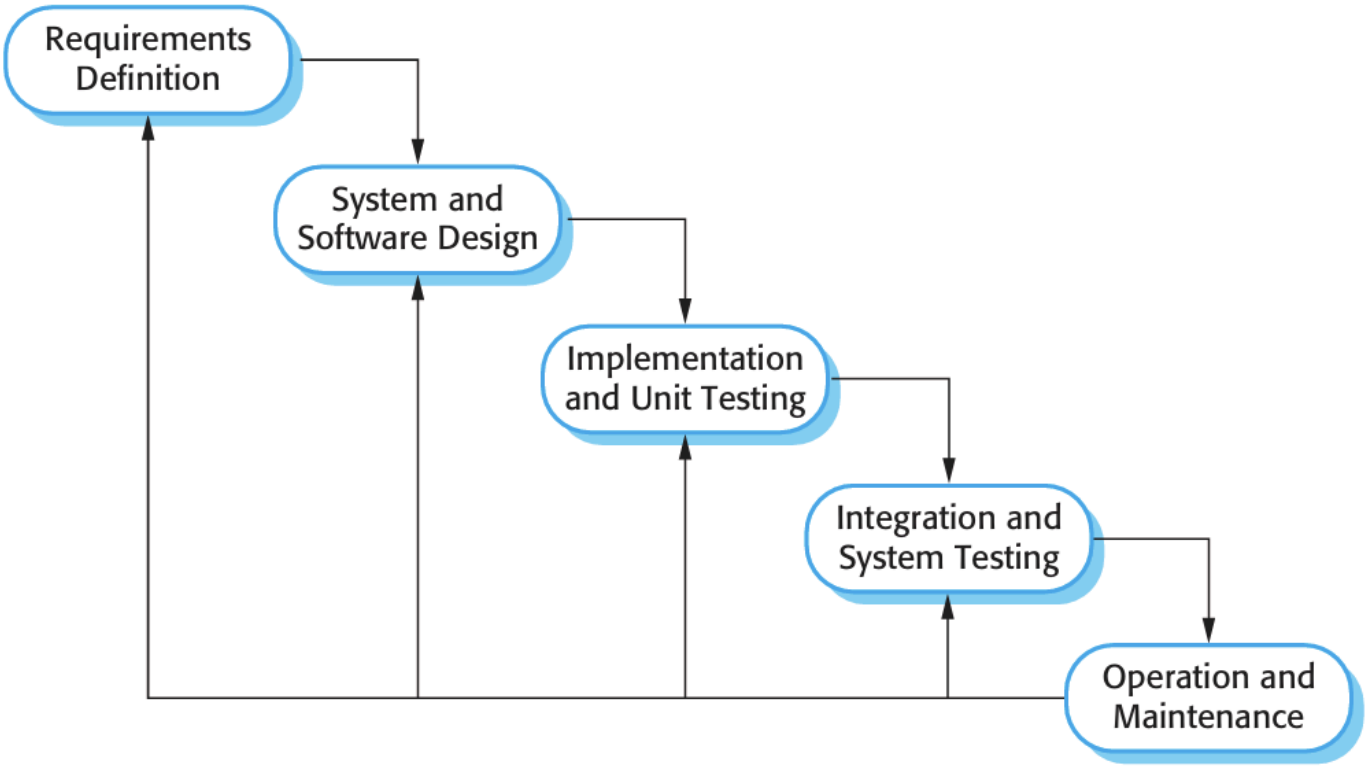
Le metodologie che non sono un processo in senso tecnicamente corretto vengono chiamate **pratiche**. Le pratiche non dicono dettagliatamente quali sono le singole attività e come si debbano comporre, ma semplicemente quali sono le tecniche concrete da applicare.
Le **best practices** sono dei modi corretti ben precisi per portare a termine ogni attività. Sono quindi indicazioni pratiche su come svolgere le attività, fornendo a livello pratico le metodologie piuttosto che a livello gestionale. Non si vede lo sviluppo come inquadrare l'attività in uno schema, ma si lascia libera la selezione dell'attività, purché si seguano le linee guida fornite.

Processo a cascata (Waterfall)

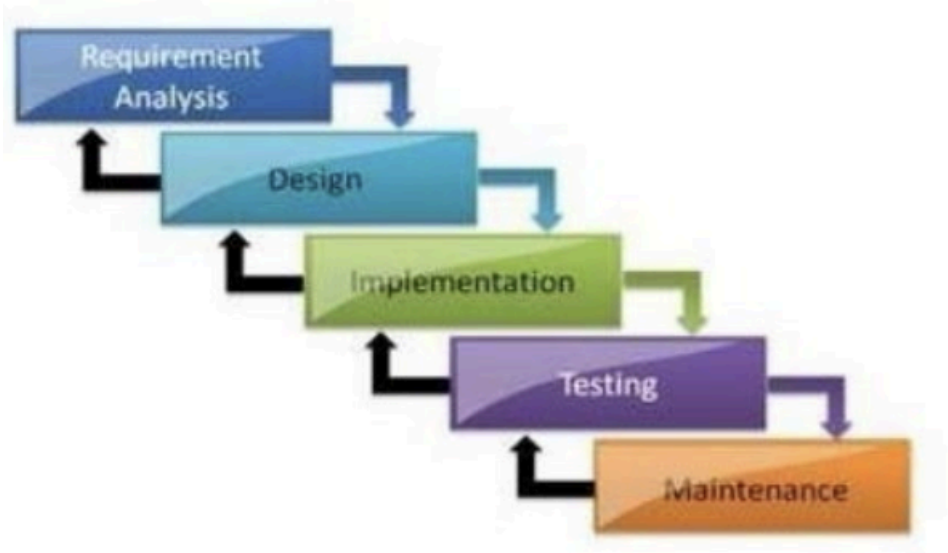
La caratteristica distintiva del **processo a cascata** è l'estrema sequenzialità. Si divide l'attività progettuale in una serie di attività distinte tra loro, ma l'idea è di azzerare l'interazione tra le attività. Le componenti interagiscono solo attraverso gli artefatti passati alla fase successiva.

Il procedimento nasce con delle frecce da sopra a sotto. Successivamente ci si è resi conto che il processo serve solo per un progetto di grosse dimensioni in cui è importante e costoso portare a termine ogni singola attività. In questi progetti è importante che non ci siano errori. Correggere uno sbaglio commesso in una delle attività può essere costoso. Pertanto, vogliamo che ogni **singola attività produca artefatti privi di errore**, ad esempio assegnando ad ogni attività dei team specializzati. Fintanto quindi che il progetto è ad alti costi, per cui ha senso che le attività vengano svolte con altrettanto impegno e allo scopo di garantire che il risultato sia corretto, il meccanismo funziona. Il modello a cascata garantisce un risultato corretto quindi se:

- il sistema è costoso;
- il tempo a disposizione è abbondante;
- **i requisiti temporali non cambiano**.



Questo tipo di approccio tuttavia non è buono se i requisiti cambiano. In più, c'è il rischio in questo approccio che qualche errore possa essere commesso. Ciò rappresenta una vulnerabilità implicita nel modello a cascata. Si aggiunge allora un meccanismo di **feedback** con le fasi precedenti. L'interazione con le attività è quindi sia "in avanti" che "all'indietro":



\

Distinguiamo allora due estremi:

- **one-shot**, ossia un meccanismo puramente sequenziale;
- **feedback**, garantendo che ogni attività possa tornare indietro, svolgendo il processo di sviluppo andando avanti e indietro tra le varie attività fino a quando non si arriva alla fase conclusiva.

Svantaggi

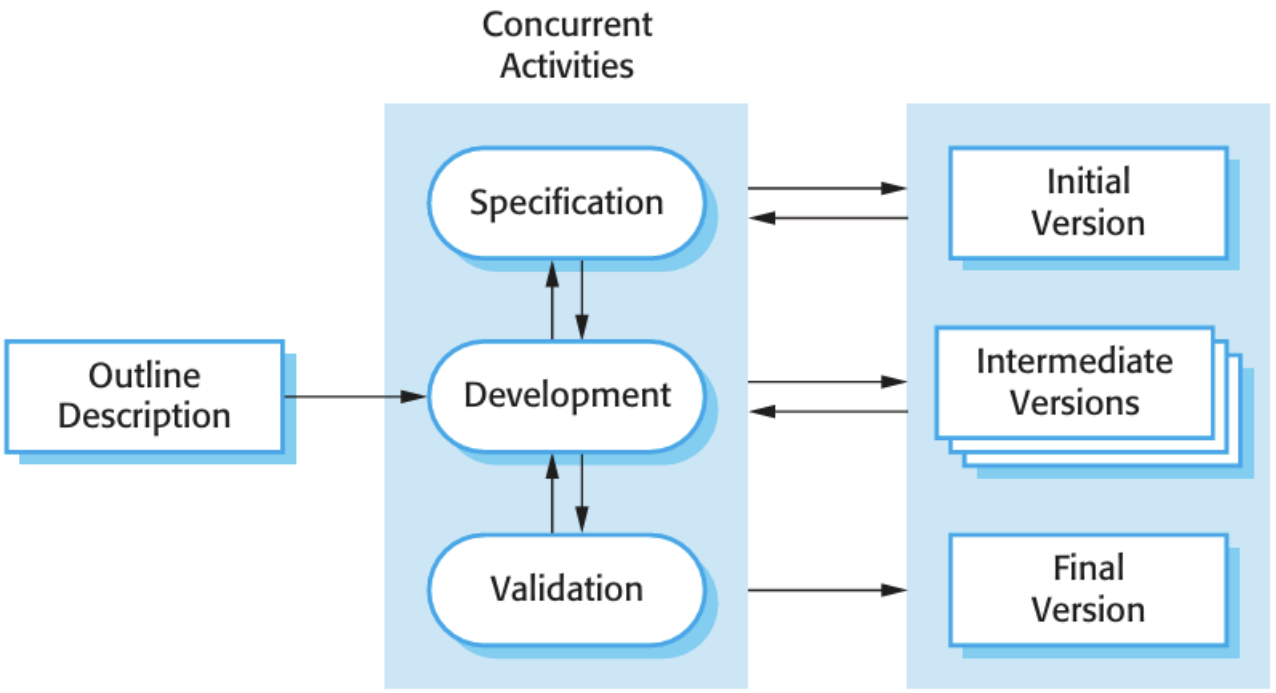
Un primo svantaggio riguarda la componente temporale: il prodotto risultante infatti può essere ottenuto dopo un grande lasso temporale. Inoltre, si ha poca interazione con il cliente, in quanto si ha solamente nella fase di richiesta dei requisiti. La poca interazione con il cliente comporta anche la difficoltà ad introdurre cambiamenti richiesti da quest'ultimo.

Vantaggi

Il modello a cascata garantisce una **consistenza** tra gli artefatti passati alle attività. Ogni attività, inoltre, produce del materiale che resta come **storico**, ossia come **prova di avanzamento** del progetto. Questo storico può anche essere un insieme di materiale intermedio, non obbligatoriamente presente nel prodotto finale, ma garantisce **documentazione**.
Il modello a cascata è particolarmente potente quando **i requisiti non cambiano**, garantendo la produzione di un prodotto di **altissima qualità**.

Processo evolutivo

Chiaramente non tutti i processi sono grandi o con disponibilità economiche e temporali infinite. Passiamo allora all'estremo opposto: il **processo evolutivo**.
Si realizza uno sketch, ossia un outline del sistema da sviluppare, e si evita di ritardare l'inizio della codifica del sistema. Essenzialmente, **si realizza il sistema in molteplici versioni** in virtù delle specifiche finali. L'idea del processo è la seguente: si ottengono dei requisiti chiari, e il cliente guida la sequenza delle evoluzioni tramite un meccanismo **build and fix**. L'idea è quindi quella di non dover richiedere requisiti finali, ma di avere un'interazione continua con il cliente che si basa sul confrontarsi con il cliente in virtù dei prodotti intermedi. Il numero di interazioni con il cliente è indefinito, e dipende da quando il cliente è definitivamente soddisfatto.



Svantaggi

La tempistica di questo processo è **assolutamente imprevedibile**. Il waterfall prevede uno studio degli obiettivi, per cui è possibile stimare gli effort per le attività, e quindi il tempo complessivo. In questo caso **non si ha idea di quando verrà completato il progetto**. Inoltre, in questo processo il lavoro è svolto su informazione informale, senza richiesta di cura dell'input del cliente. Quello che viene svolto **non è ben documentato**: viene solamente mostrato come risultato intermedio al cliente in virtù di ulteriori specificazioni. Questo modus operandi pertanto non è ben documentato. Infine, **è assente la visione di insieme**.

Vantaggi

Per sistemi estremamente piccoli, in cui tutto sommato quello che si mette in gioco è semplice e non complesso, i difetti dell'approccio non sono pesanti. Il fatto che i requisiti sono trattati con superficialità e che il lavoro di sviluppo è fatto senza particolare cura non sono problemi per il prodotto finale: un'analisi più approfondita non è richiesta. Il numero di interazioni a livello gestionale è ridotto, e appena si ottiene un risultato intermedio il prodotto viene mostrato al cliente per eventuali correzioni o per il rilascio.
Pertanto, il processo evolutivo è molto buono per progetti piccoli, per piccole parti di sistemi più grandi e per sistemi a vita breve.

Sviluppo incrementale, sviluppo component based

Se il lavoro è complesso, possiamo gestirne la complessità suddividendolo in parti. Possiamo quindi realizzare progetti più complicati, ma con il vantaggio di avere più release, ossia più momenti per confronti con l'utente finale. La **classe di processi incrementali** prevede quindi la realizzazione del progetto tramite aggiunta di funzionalità fino al completamento, ottenendo feedback su ogni elemento aggiunto.
Simile allo sviluppo incrementale vi è lo sviluppo **component-based** (**CBSE**, Component Based Software Engineering). I componenti sono diversi tra loro, ma a basso livello si somigliano tutti. I pezzi che servono per creare sistemi specifici alla fine sono tutti ottenuti utilizzando componenti a basso livello uguali. Basta allora prendere questi componenti, detti **COTS** (**Components Off The Shelf**), e assemblarli.

Chiaramente i componenti non sono modificabili in quanto preesistenti. Tuttavia, è possibile **modificare i requisiti**, realizzando pertanto un sistema che si avvicina molto al risultato finale sebbene non sia precisamente identico a quello richiesto. In questo caso è possibile realizzare degli adattatori, già visto ad esempio con l'Adapter design pattern.

Sviluppo iterativo e incrementale

L'approccio incrementale è utile quando conosciamo già qual è il risultato finale. In un procedimento iterativo, invece, ripetiamo il lavoro più volte, poiché non abbiamo le idee chiare su quello che dobbiamo fare, ma un'idea abbozzata, migliorando il prodotto fino a quando il cliente è soddisfatto. Il processo evolutivo rientra nella categoria iterativa.

Possiamo tuttavia **combinare le tecniche iterativa e incrementale**. Partiamo da un'idea superficiale, e dividiamo il lavoro in parti, iterando incrementi sulle parti da sviluppare.

Rischi relativi al progetto

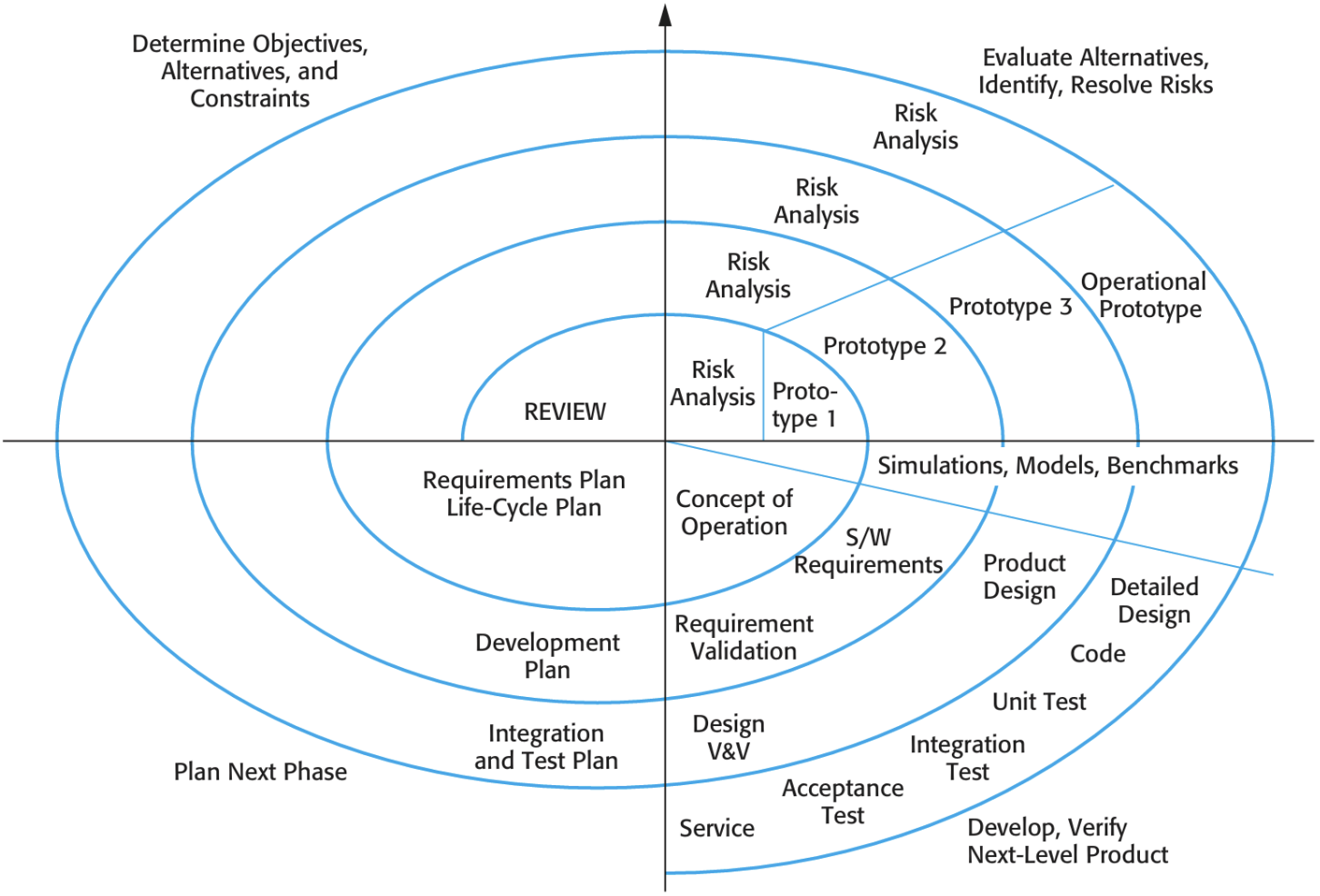
L'ingegneria del software si è evoluta molto rapidamente, e oggi si considerano molte variabili in relazione al progetto:

- ci sono aspetti legati a chi partecipa al progetto. Bisogna ad esempio garantire che il team sia soddisfatto nel lavoro di squadra;
- il team dev'essere assemblato in modo tale che ogni elemento della squadra sia coordinato, allocando le risorse necessarie. Tra queste risorse ci sono le persone, che umanamente devono essere valide e con le competenze giuste. Mettere insieme competenze non adeguate per il problema da risolvere è un rischio per il progetto, così come coordinare male la squadra;
- bisogna garantire che le tecnologie impiegate siano adeguate per la realizzazione del prodotto. Poiché un ingegnere del software non è solo un ingegnere ma anche in parte uno scienziato (inteso nell'accezione di computer scientist), può essere necessario creare gli strumenti necessari. Un altro problema è la **retrocompatibilità**: la base del sistema sviluppato, come un sistema operativo, può essere aggiornata e può potenzialmente invalidare il funzionamento del sistema;
- ci sono dei constraint da rispettare, tra cui le risorse economiche e fisiche, nonché risorse temporali.

Queste problematiche vengono affrontate tramite la **pianificazione del lavoro**, che include non solo la metodologia di sviluppo, ma anche le tempistiche e i costi. Oltre ai diagrammi use case vengono anche utilizzati i **diagrammi di Gannt**, dove sull'asse x è presente il tempo e sull'asse y le attività da svolgere. Si fornisce quindi un'indicazione temporale per ciascuna attività, collocandole nel tempo. Si concretizza quindi la metodologia di sviluppo mappata su vincoli di tempo e denaro. Ogni barra nel diagramma infatti rappresenta un **costo orario** che dipende dall'attività stessa. In un contesto aziendale questo permette anche la quantificazione dei costi: se 10 persone devono lavorare ad un'attività per un mese, ciò si traduce in un costo economico pari a 10 paghe mensili.

Framework a spirale (Spiral framework)

Il primo approccio proposto per la realizzazione dei progetti è il **framework a spirale**. Si tratta di un modello iterativo, con uno schema che prevede sempre un ciclo di attività ripetute. Gli assi indicano che c'è un parametro che cresce di intensità; di nostro interesse è l'asse y. Quello che cresce è in questo caso il **rischio**: il rischio cresce man mano che si va avanti nello sviluppo. Più si va avanti, maggiore è l'impiego di risorse e quindi maggiore è il rischio che un errore causi costi maggiori. Si parte quindi dal centro, con rischio zero:



L'asse x permette una divisione in quattro quadranti. Il primo, a partire dal quadrante in alto a sinistra, prevede la fase di analisi dei requisiti, che non è un'attività che dev'essere conclusa come nel modello a cascata, ma come nei modelli di tipo iterativo prevede una reiterazione. Successivamente, il quadrante in alto a destra è il quadrante che rappresenta le attività che portano all'identificazione delle alternative più o meno rischiose. Si valuta quindi l'alternativa con rischio minore, abbozzando una prima versione del sistema (nello schema è rappresentanto con la dicitura "Prototype 1"). Il prototipo realizzato viene analizzato a livello

matematico, ossia utilizzando delle simulazioni, dei modelli e dei benchmark. Passiamo cosi al successivo quadrante (in basso a destra), che prevede le fasi di verifica del prodotto. Infine, è necessario eseguire una pianificazione, sia a livello di requisiti che a livello di ciclo di vita, passando nel quadrante in basso a sinistra. Il ciclo viene ripetuto, ridefinendo nuovamente gli obiettivi, valutando il rischio e sviluppando un nuovo prototipo, eseguendo ulteriori simulazioni e veificando che i requisiti siano verificati. Eseguiamo ora un piano di sviluppo, e ripetiamo nuovamente la fase di analisi. Il processo continua, realizzando un ulteriore prototipo, fornendo stavolta un'architettura al sistema e pianificando i meccanismi di integrazione del sistema e di testing. Un'ultima fase prevede lo sviluppo del prodotto finale, con un prototipo funzionale, un design dettagliato e la fase di codifica e di testing.

Vantaggi e svantaggi

È evidente come l'approccio sia orientato alla gestione del rischio. Si ammette la possibilità che i requisiti possano cambiare nel tempo, ma non durante la fase di sviluppo: una volta cominciata la codifica, non si torna indietro. Tuttavia, si fa appoggio al fatto che effettivamente i requisiti sono stati analizzati ben quattro volte.
L'approccio inoltre fa appoggio ad un'analisi per lo più gestionale, dando un'importanza allo sviluppo del software solamente in fase finale. Questo tipo di approccio tuttavia ritarda lo sviluppo del software fino a quando non si è sicuri dei requisiti, richiedendo quindi una risorsa temporale non indifferente. Inoltre, il framework non è del tutto foolproof a cambiamenti nella fase finale di codifica.

19 - Metodi agili, refactoring

Metodi agili

Negli anni 2000 si è notato che è necessario avere una certa capacità a reagire al mondo esterno, evitando di chiudersi sull'idea che una meticolosa progettazione possa escludere ogni tipo di problematica del mondo reale. Quello che deve avere importanza non è quello che viene documentato, ma la realtà, ossia l'interazione con il cliente. La documentazione più dettagliata in questo caso è il codice.

Il **manifesto della programmazione agile** enuncia alcuni principi dei metodi detti **agili**. L'approccio agile evita l'utilizzo di piani, che vincola il programmatore e crea problemi, e supporta la libertà. Le informazioni devono provenire dal cliente, cercando di collaborare con questo e non bisogna negoziare per un prodotto che non sia al 100% identico a quello richiesto dal cliente. Si ha quindi la necessità di poter rispondere al cambiamento, e non di seguire ciecamente un piano.

Extreme Programming

Il concetto dell'approccio **Extreme Programming** (**XP**) nasce duranto lo sviluppo di un progetto della casa Chrysler. All'interno del processo di lavoro vennero individuate le buone pratiche, vennero raccolte insieme e venne massimizzato l'utilizzo di queste, imponendo di fare le cose in quel modo. Il modus operandi quindi viene definito dalla pratica corretta, e tale pratica dev'essere un comandamento a cui obbedire sempre. Non basta quindi che qualcuno sia bravo, ma bisogna estremizzare l'uso dei modi migliori di fare le cose.

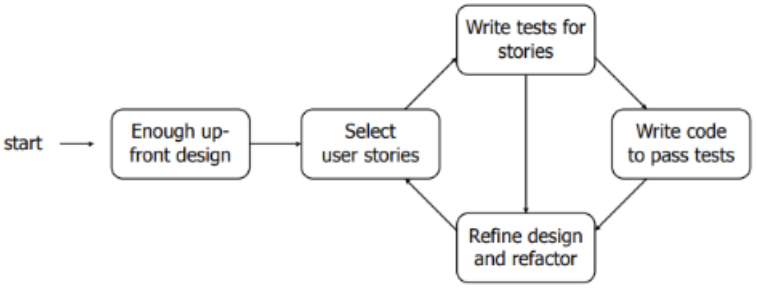
Il concetto quindi non è quello di programmare in maniera estrema, ma di estremizzare l'uso delle pratiche corrette. Oltre all'utilizzo delle pratiche da applicare costantemente, l'idea è che quello che il processo XP dà non è definito da uno schema come quelli visti, ma piuttosto è una definizione. Un progetto segue l'approccio XP quando si produce una nuova release del sistema, dove per release si intende qualcosa che si installa sul posto e che il cliente può utilizzare.

I punti chiave di XP sono i seguenti:

- **focus sul codice**: non si dà importanza al piano di sviluppo, ma allo sviluppo stesso. L'unica attività principale è la codifica, che si porta appresso attività complementari. Quindi ci sono quattro tipi di attività durante il processo XP, che sono:
  - **listening**, ossia ascoltare cosa vuole l'utente dal sistema;
  - **testing**, centrale per lo sviluppo;
  - **coding**, in particolare una scrittura di un codice semplice;
  - **design**, per ripulire il design e l'implementazione del sistema.
    L'unica documentazione è il **codice**;
- **orientato sulla gente**: la conoscenza sul prodotto è nelle persone che sono applicate a quel progetto e scambiata oralmente. Questo quindi implica un'interazione continua;
- **leggerezza**: non si hanno piani o organizzazioni, ma le persone lavorano sul progetto, rimuovendo i costi aggiuntivi e creando un prodotto di qualità con test rigorosi. Si identificano quindi gli elementi chiave e le pratiche da applicare costantemente.

Ciclo di sviluppo XP

Per schematizzare lo sviluppo XP, basta utilizzare il seguente schema:



Il cliente viene ascoltato per acquisire informazioni sul sistema. Questo viene espresso sotto la nozione di **user stories**. A partire dalle stories vengono scritti dei test per le user stories, scrivendo il codice del sistema per superare i test (seguendo quindi uno sviluppo Test-Driven). Infine, si applicano i desgin pattern e un refactoring per scrivere in maniera più pulita il codice. In maniera incrementale il sistema viene prodotto tramite continue interazioni con il cliente.

Questo tipo di approccio permette di implementare il concetto di mappare le funzionalità dentro unità di codice. La scrittura infatti delle storie è importante e permette di realizzare dei moduli ben definiti. La scelta di come inserire gli incrementi tuttavia ha una valenza per il progetto. Inoltre, notiamo che nello schema è presente un arco diretto tra la scrittura dei test per le storie e il refactoring del sistema. Infatti non è detto che le nuove storie prevedano nuove codifiche, ma semplicemente il refactoring del testo.

Il processo XP consiste nel ripetere più volte i passaggi, fino a quando non si ha un'ultima release. Ma questa ultima release non è detto che esista. Questo approccio permette lo sviluppo continuo di sistemi, aggiungendo sempre più funzionalità.

La fase di design è abbastanza trascurata. Non si ha infatti un design complessivo con una visione completa dei moduli da organizzare, ma solo un design per lavorare per un tempo limitato. La codifica di questo approccio inoltre non è proprio la codifica ideale. Idealmente vorremmo avere la definizione di un metodo e scrivere un codice che implementi nella maniera migliore il codice. Qui no: si cerca di scrivere un codice minimale, ossia una soluzione rapida per l'implementazione della funzionalità. Lo sforzo non è necessariamente giustificato e il tempo è limitato: l'obiettivo non è la qualità del codice, ma la **consegna** del prodotto. Più importante tra tutti è il **testing**, che fornisce le specifiche del sistema. Lo sviluppo, come detto, è **test-driven**, in quanto il codice da scrivere deve soddisfare i test.

L'architettura globale del software soffre di questo approccio. La poca attenzione sul design si fonda sull'idea che una fase di design eccessivamente dettagliata può portare ad effort sprecati. Tuttavia, troppo poco design può portare alla necessità di un rework del codice del sistema.
Un compromesso in questo caso è modificare il ciclo di sviluppo XP includendo un minimo di ragionamento sul design, ossia un **upfront design**, che non funge da obbligo ma da indicazione di massima.

Listening

L'attività di listening è importante. Il risultato dell'ascolto dell'utente sulle specifiche del sistema sono le **stories**. Queste stories descrivono uno scenario di utilizzo, non una categoria di funzionalità. Ci si adegua a quello che oggi è un template di base:

> As a <role> I want to <action> so that <benefit>

Ad esempio: "Come **cliente** vorrei **prelevare soldi da una ATM** così che **posso avere contanti anche quando la banca è chiusa**".

Queste stories vengono annotate su foglietti di carta piccoli in modo tale che ogni scenario sia abbastanza piccolo ma che comunque sia in grado di descrivere le esigenze dell'utente.

Testing

Dalle stories deriviamo i requisiti. Per verificare i requisiti, utilizziamo dei **test** che possono essere di più tipi:

- test per verificare il funzionamento del codice;
- test che derivano dai requisiti ricavati dalle user stories (ossia test di accettazione).

I test sono strutturati con una forma che abbiamo già visto con **Mockito**:

> Dato un <contesto> quando <evento> allora <risultato atteso>

Si possono usare anche connettivi logici per concatenare contesti (ossia far sì che si verifichino più condizioni assieme).

Refactoring

L'idea nella fase di implementazione è di scrivere un codice che non sia il migliore ma che sia rapido da scrivere. Dopo la scrittura di codice, una fase di **refactoring** ci permette di preservare la correttezza di quanto scritto, garantendo tuttavia modularità al nostro sistema, facendo sì che il principio di singola responsabilità sia garantito.
In generale, prima di passare alla prossima user story cerchiamo di dare una pulita al codice, in particolar modo:

- cerchiamo di applicare i design pattern;
- cerchiamo di migliorare la leggibilità;
- cerchiamo di dare nomi comprensibili per le variabili in virtù del loro scopo.

XP e complessità

La fase di design in XP dipende dalla complessità del prodotto e dalla stabilità dei requisiti del sistema:

- con un sistema dai requisiti stabili e con un prodotto semplice, basta un design minimale per stabilire la direzione di sviluppo;
- con un sistema dai requisiti mutevoli e con un prodotto complesso, si parte da un design modificabile e modulare;
- con un sistema che cambia molto ed è semplice, si dirottano le decisioni quando necessario;
- con un sistema che non cambia ma è complesso si pongono le proprietà del sistema come design up-front.

CRC cards

Le **CRC cards** (**Class Responsibility Collaboration cards**) documentano l'architettura in XP ed è l'unico altro tipo di documentazione accettata. Semplicemente vengono annotate:

- il nome della classe;
- le responsabilità della classe;
- le classi con cui si collabora.

Questo permette un confronto rapido e funge da class diagram informale.

Best Practices per XP

In totale ci sono 12 osservazioni su 12 aspetti diversi che costituiscono il cuore della metodologia XP. La metodologia XP può essere incarnata in altri modi. Ad esempio, Scrum organizza il lavoro in attività a livello gestionale, che trascura il modo in cui ci si occupa del software e rappresenta un modo complementare di organizzare il lavoro rispetto a XP.

Gioco di pianificazione

Il cliente scrive le stories che fungono da requisiti e assegna una priorità in termini di valore. Gli sviluppatori poi per ogni iterazione dividono ogni storia in task (se è necessario, chiaramente. Una certa funzionalità potrebbe richiedere un solo task, mentre altre più task), ne valutano il **rischio** e il **tempo di sviluppo**, dando priorità ai task più rischiosi. Il rischio viene indicato con un valore numerico, e non ha a che vedere con il **rischio gestionale del progetto**. I clienti scrivono stories per coprire da due fino a quattro settimane di lavoro. Infine, il cliente e gli sviluppatori stabiliscono quando pubblicare le release. I rilasci tuttavia devono essere a ritmi rapidi, seguendo sempre l'idea di programmazione agile, per cui solitamente si opta con ritmi di 1 o 2 settimane.

Notiamo come questo tipo di operatività si basa sull'**empirismo**, ossia sulla praticità.

Piccole release

Solitamente ad ogni iterazione si pubblica una release. Quelle che noi solitamente intendiamo come release sono le **major release**, mentre in questo caso ad ogni iterazione si pubblica una **minor release**, mentre le major sono concordate con il cliente in virtù delle funzionalità aggiunte.

Le piccole release hanno un impatto sul design del sistema, specialmente per il fatto che non esistono pianificazioni a lungo termine (e neanche a breve termine in alcuni casi).

Pair Programming

Svolgere task in coppia è considerata una pratica migliore rispetto a lavorare singolarmente. In generale avendo più persone è possibile parallelizzare il lavoro e ottimizzare i tempi.
Nel caso di XP tuttavia ogni persona decide quali story card implementare e se ne prende la responsabilità. Esistono dei micro-team, ossia delle persone si scelgono un compagno di lavoro per lo svolgimento di un task. I compagni si dividono in ruoli:

- uno svolge il ruolo di **driver**, ossia si occupa dell'implementazione dei metodi e di come questa può essere fatta nel migliore dei modi;
- l'altro svolge il ruoto di **navigator**, ossia si occupa di revisionare il codice, di scrivere i test e di vedere se tutto può essere fatto nella maniera migliore.

Sulla carta l'idea sembra molto sensata, anche perché non si lavora da soli. Tuttavia questo è vero quando c'è più armonia nella coppia: la componente comportamentale, infatti, può minare il lavoro del team.

Metafora

Una volta che abbiamo preso l'indicazione dei clienti, abbiamo le story card e abbiamo diviso i task per coppie di programmatori, tutto deve confluire insieme in un unico prodotto. Ci si deve quindi confrontare alla fine dell'iterazione anche col cliente. Ci dev'essere un modo di **comunicare** quello che abbiamo implementato. Finché la comunicazione è tra gli sviluppatori non vi è un problema; ma con il cliente, è necessario usare **metafore per il linguaggio**.

Sebbene sembri una cosa di secondaria importanza, in realtà non è così. È vero che basta scrivere il codice per soddisfare i test, ma i nomi e le metafore che usiamo per descrivere cosa fa il codice devono essere scelti con cura, in modo tale da evitare che nascano **problemi di incompresione**.

Testing

I test sono i primi ad essere scritti, anche prima del codice. Si trasformano le story card in test di accettazione, in quanto le story card fungono da requisiti richiesti dal cliente.

Prima di poter dire che il task che bisogna sviluppare è pronto, bisogna eseguire i test. Ma lo step successivo chiaramente è l'integrazione all'interno del sistema da sviluppare.

Progettazione semplice

La fase di progettazione è una fase semplice, che evita complessità inutili per mantenere il codice chiaro e leggibile ed evitare che la fase di codifica sia rimandata, a differenza di framework come il framework a spirale che rimandano l'implementazione del sistema a favore di una meticolosa progettazione dei rischi.

Integrazione continua

La tentazione sarebbe di integrare i pezzi del software quando si finisce l'iterazione. Questo tuttavia non aiuta lo sviluppo: se bisogna riunirsi, si potrebbero ritardare i tempi per evitare di integrare parti incomplete nel software.

Ogniqualvolta si sviluppa un pezzo di task, e questo pezzo supera i test, lo si installa nel sistema, **anche se l'iterazione non è completa**. Risulta quindi importante aiutare gli sviluppatori fornendo gli strumenti per automatizzare il testing. Ad esempio, Maven si integra negli ambienti di sviluppo per supportare le attività automatiche. Anche il deployment del sistema dev'essere automatizzato in XP, mentre nel processo waterfall l'installazione del sistema viene fatta in un ultimo momento.

Integrazione continua significa che nell'ambiente di lavoro bisogna avere un tool che permetta quanto detto con un clic. Si deve scaricare allora il codice condiviso, si fa l'upload e il sistema in un momento di pausa deve lanciare una batteria di test di integrazione.

Refactoring

Completato l'incremento, l'ideale sarebbe avere la release pronta. Tuttavia, poiché ogni pezzo è stato sviluppato da sotto-team diversi, bisogna unificare il lavoro: per qualche giorno, il team al completo analizza il codice con l'intento di capire se dei moduli hanno interazioni tra di loro o alcune cose potevano essere scritte meglio. L'importante è **evitare di introdurre nuove funzionalità**, in modo da poter **ri-eseguire gli stessi test scritti precedentemente** per verificare che il refactoring abbia migliorato il codice senza aver rotto alcuna funzionalità.

Cliente in sede

Il cliente deve essere sempre reperibile, o alternativamente viene delegato un rappresentante per essere sempre presente e osservare quello che viene fatto. Tipicamente il modo di procedere della XP viene apprezzato anche dal punto di vista del committente in quanto può perfettamente vedere cosa avviene dietro le quinte della sua richiesta e viene coinvolto attivamente; si può quindi rendere conto che il lavoro ha un valore e può iniziare a sviluppare fiducia su quello che ha commissionato provando i vari progressi (ossia le release).

Senza questo tipo di approccio, in altri scenari, dopo che il cliente commissiona il progetto gli sviluppatori indicano il tempo che pensano di prendere e "finisce lì": quest'ultimo deve aspettare senza poter vedere dei progressi se non chiedendo esplicitamente.

Possesso del codice collettivo

Il **possesso del codice collettivo** è un'altra best practice che permette di analizzare le integrazioni aggiunte al sistema più recentemente.

Quando si sceglie il task da implementare, lo si fa senza la presenza di un manager che gestisce lo sviluppo. Da notare che il manager può essere presente, ma solo con un ruolo di alto livello, ad esempio:

- negoziare con il cliente le macro release;
- affittare le macchine e far installare il software per lo sviluppo.

Il manager quindi gestisce le attività che non riguardano il ciclo di esecuzione. Inoltre, il possesso del codice collettivo non riguarda solamente la possibilità di avere il codice, ma anche che la responsabilità del task è la responsabilità di un team. Non è detto che si

riesca a completare il task in tempo, ma se un gruppo ha finito di lavorare ad un task prima, allora può subentrare. Se c'è un problema in un task, tutti sono responsabili e tutti possono agire.

Settimana di 40 ore

Chiaramente la programmazione è un processo creativo. Si deve limitare il carico di lavoro del programmatore, in quanto solamente così si può minimizzare lo stress. Un programmatore stressato produce codice con errori.

Usare gli standard del codice

Bisogna utilizzare notazioni standard per il codice. Poiché il nostro codice è la nostra documentazione, il codice dev'essere leggibile. Codice leggibile significa:

- indentazione corretta;
- camel casing;
- ogni identificatore ha un nome indicativo e suggestivo (ad esempio, chiamare una variabile "prezzo" probabilmente indica che la variabile memorizza il valore di un prodotto);
- evitare ripetizioni nella scrittura del codice.

Riepilogo

Abbiamo visto diversi processi di sviluppo:

- Waterfall, rigidamente sequenziale e one-pass, con una sua alternativa con feedback;
- Evolutionary, che vede la trasformazione del prodotto da una versione iniziale ad una finale;
- Incrementale, che vede l'aggiunta di funzioni partendo da un'idea chiara del prodotto finale;
- Component Based, che prevede l'utilizzo di COTS, ossia Componenti Off-The-Shelf;
- Spiral, un framework iterativo per pianificare la gestione del rischio prima dello sviluppo del codice;
- Agile, una metodologia che agevola i cambiamenti utilizzando processi più leggeri, con XP.

Refactoring

L'idea del refactoring è quella di riconoscere che è difficile ottenere fin dall'inizio una progettazione adeguata. I cambiamenti all'interno del codice permettono di incorporare nuovi requisiti più facilmente o requisiti che cambiano. Il refactoring fornisce tecniche per evolvere la progettazione in passi, riducendo il **debito tecnico**, ossia gli errori che abbiamo introdotto. Le strutture si trasformano in strutture più semplici e facili da capire.

Estrazione del metodo

Un frammento di codice può essere estratto, facendo diventare quel frammento un metodo il cui nome spiega lo scopo del frammento. Ad esempio:

```
public void printAll() {
    printBanner();
    System.out.println("Nome: ", nome);
    System.out.println("Cognome: ", cognome);
}
```

diventa:

```
public void printAll() {
    printBanner();
    printInfos(nome, cognome);
}
public void printInfos(String nome, String cognome) {
    System.out.println("Nome: ", nome);
    System.out.println("Cognome: ", cognome);
}
```

I parametri locali diventano parametri del metodo creato. Inoltre, se il codice estratto ha variabili usate solo all'interno del codice estratto, si fanno diventare variabili temporanee del nuovo metodo.

Si riduce quindi la lunghezza dei metodi, rendendo i metodi più riutilizzabili. Inoltre, le funzionalità di alto livello sono più comprensibili, in quanto sembrano costituiti da una sequenza di commenti. Infine, se i metodi sono piccoli, eseguire overriding è più semplice.

Sostituzione di variabili temporanee con metodi query

Le variabili temporanee tengono valori calcolati che non sono valori finali. Il calcolo intermedio potrebbe essere fatto non una volta, ma più volte. Se concettualmente è sempre lo stesso calcolo, il pezzo di elaborazione potrebbe essere incapsulato in un metodo a sé stante. Ad esempio:

```
double prezzoBase = quantita * prezzo;
```

diventa:

```
private double getPrezzoBase() { return quantita * prezzo; }
```

Divisione della variabile temporanea

Se la variabile temporanea è indispensabile ed è assegnata più di una volta, ma non è una variabile assegnata in un loop, si può utilizzare una variabile separata per ciascun assegnamento.

Questo può essere utilizzato per evitare ambiguità ed errori. Si assegna quindi un nome indicativo ad ogni variabile che utilizziamo. La variabile, inoltre, è dichiarata come `final`, in modo tale che il compilatore controlli dove ci sono le successive assegnazioni alla variabile.

Dopo aver applicato questo refactoring possiamo applicare la sostituzione di variabili temporanee.

20 - Scrum

Scrum

A differenza di XP, il framework Scrum è più "classico". Il processo di sviluppo descrive le attività, i ruoli e i prodotti di ogni attività, nonché le tempistiche da seguire per svolgere le attività. Scrum è una metodologia **organizzativa**, concentrata sugli aspetti gestionali del processo di sviluppo, seguendo tuttavia un approccio agile, non utilizzando la documentazione come mezzo di comunicazione principale ma l'interazione con il cliente.
Alcuni aspetti in Scrum si sovrappongono con il modo di procedere che c'è in XP. Il ciclo di attività veniva iterato in XP, e questo concetto di attività è presente anche qui.

Oggi Scrum è uno tra i framework più utilizzati. Il 66% degli sviluppatori usa solo Scrum, mentre un altro 15% utilizza Scrum insieme a XP.

Anche Scrum è un metodo iterativo. L'iterazione prende il nome di **sprint**, ed è tendenzialmente più lunga rispetto a XP. Lo sprint è lo svolgimento di un'attività che porta alla realizzazione di un goal, ossia di un obiettivo funzionale (quindi un requisito) principale nel sistema. Un requisito infatti può essere fine, ma quando si parla di goal si parla di una **macro funzionalità**, ossia di un obiettivo complessivo. Risulta normale quindi considerare uno sprint con una durata maggiore di 2 settimane, mentre con XP il ritmo dell'iterazione è di una o due settimane. **Eventi** all'interno dello sprint ne dettano il ritmo:

- il **daily scrum**, un briefing quotidiano prima di mettersi al lavoro;
- lo **sprint review**.

Questi eventi sono **time boxed**, ossia limitati a livello temporale, così come lo sprint. Ad esempio, il daily scrum deve avere una durata pari a 15 minuti.

Scrum specifica anche dei ruoli e degli artefatti. Inoltre, anche in questo framework abbiamo dei principi:

- **trasparenza**: il codice e gli artefatti sono accessibili a tutti e non si hanno livelli di accesso differenziati;
- **ispezione**: garantendo la trasparenza ci si può rendere conto se ci sono aspetti che devono essere sistemati. La potenza con cui si analizzano le informazioni non è limitata solo alle persone che hanno il privilegio di vedere il codice, ma si amplifica a tutti i membri;
- **adattamento**: poter ispezionare il codice permette di intervenire, modificano ove necessario i dati e il codice.

Ruoli

Riconosciamo diversi ruoli all'interno del team Scrum.

Product owner

Il product owner corrisponde al **cliente** del processo XP. Si tratta di colui che fornisce i requisiti del sistema richiesto, nonché feedback su quello che viene prodotto. Il product owner gestisce il **product backlog** degli item di lavoro, ossia i requisiti del sistema. Inoltre, imposta i goal che si devono raggiungere durante uno sprint.

Developer

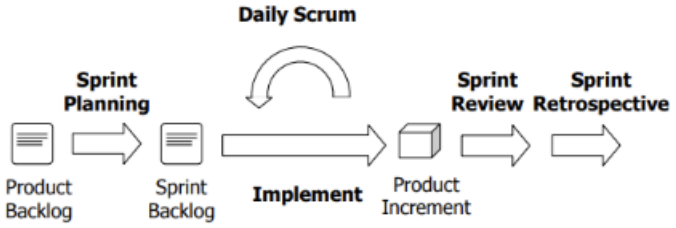
Lo sviluppatore come in XP stima il tempo per l'esecuzione di un task. Al termine di ogni sprint, dovrebbe essere sempre rilasciato un incremento del software, ossia un progresso concreto a livello dell'utente. Questo progresso concreto si traduce nell'aggiunta di funzionalità.

Scrum Master

Lo **Scrum Master** è un coach, ossia un moderatore e un aiutante. Si tratta di una figura presente, ma non è un capo che comanda. Si occupa di risolvere problemi per rendere più semplice lo sviluppo alla squadra. Se ci sono **impedimenti esterni**, ossia problemi non tecnici ma di qualunque natura che hanno ripercussioni nell'attività di sviluppo, egli si occupa di risolverli (ad esempio, se scade la licenza del software usato per sviluppare, lo scrum master si occupa di rinnovarla). Si assicura anche che le scadenze vengano rispettate.

Mentre XP si concentra sul produrre codice in maniera agile, di qualità ma senza una fase di design, Scrum si occupa di garantire che il processo di sviluppo arrivi al termine. Per quanto riguarda le attività di sviluppo, l'unico livello a cui ci si ferma è il livello gestionale.

Schema del funzionamento



Si parte da un artefatto iniziale, che è un insieme di task da completare, il **product backlog** (letteralmente "arretrato del prodotto"). A partire dai task da completare, **si pianifica lo sprint**. Una frazione del product backlog, lo **sprint backlog**, contiene i task da svolgere nello sprint.

Ogni giorno si ha un **daily scrum** (evento) a cui partecipano gli sviluppatori e lo scrum master per la pianificazione del lavoro del giorno. Questo viene ripetuto ogni giorno durante la fase di implementazione. Si produce così un **product increment**.

Infine, al termine dello sprint si hanno due eventi:

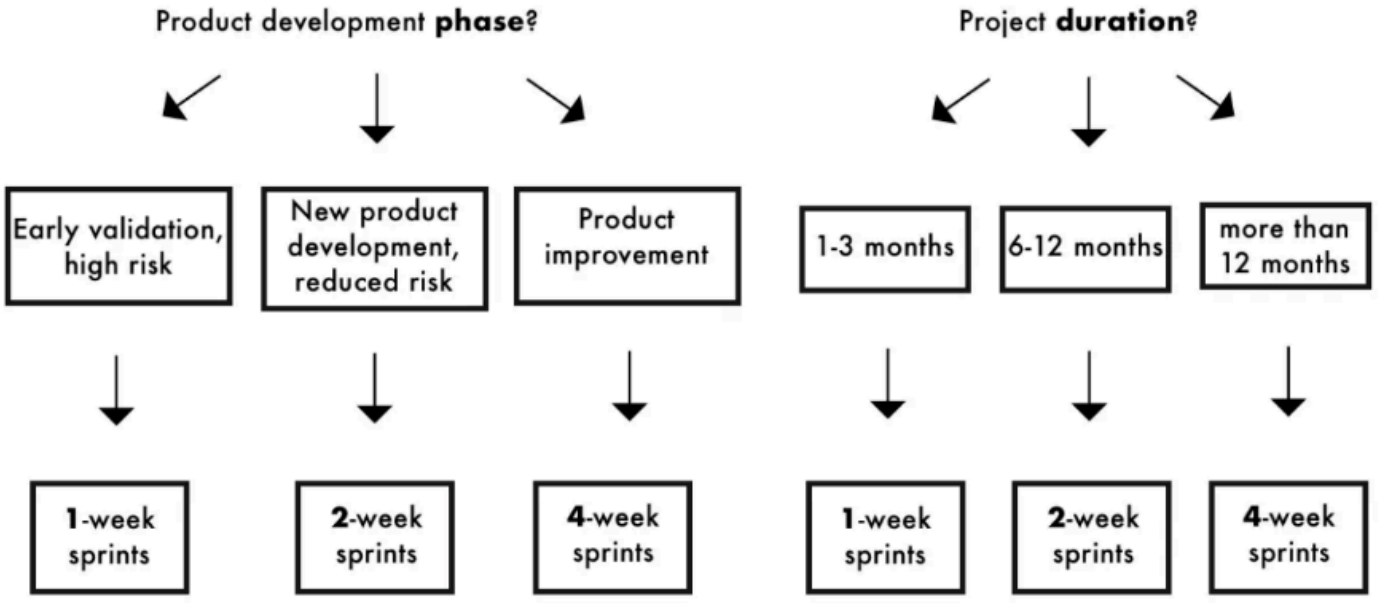
- **sprint review**, ossia una review dell'incremento prodotto;
- **sprint retrospective**, ossia una riunione in cui si ragiona sul come è stato svolto l'iterazione a livello progettuale.

Nella sprint review partecipa il product owner, in quanto si parla dell'incremento di prodotto. Nella sprint retrospective il cliente non viene coinvolto.

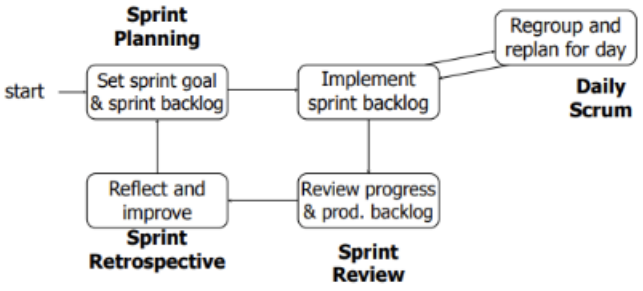
Anche le riunioni hanno una durata prefissata. Il tutto è infatti organizzato in attività con eventi di durata prestabilita che dev'essere rispettata rigidamente. Le durate degli sprint sono sempre uguali, in quanto sono fortemente legate alla natura del sistema da sviluppare. Nella realtà empirica delle cose, la durata ottimale è sempre uguale durante lo sviluppo di un oggetto. Solitamente gli elementi che si tengono in considerazione sono di due tipi:

- a che livello di maturità è il progetto che sto intraprendendo. C'è già stata una prima valutazione del rischio? È un prodotto nuovo? Ci sono assunzioni che si possono fare in sicurezza? Oppure è un prodotto che è già esistente, e quindi quello che si fa è un miglioramento?
- durata complessiva del progetto. L'intero product backlog in quanto tempo dev'essere svolto? **In quanti mesi posso portare a termine il progetto**?

In base a questi parametri si ha un'indicazione sulla durata dello sprint, che varia da una a quattro settimane. Questi numeri sono gli stessi in XP, solo che in XP quattro settimane sono da evitare, spingendo di più sulla singola settimana, ossia il minimo possibile.



Il grafico sottostante riassume quanto detto:



Sprint planning

Lo sprint planning è una riunione di massimo 2 ore per settimana moltiplicato per il numero di settimane per lo sprint. Lo sprint planning quindi, se lo sprint dura 2 settimane, dura al più quattro ore. Nello sprint planning lo scrum master si assicura che la scadenza del progetto sia rispettata, in modo tale che il non completamento del progetto non è un problema gestionale. Il rischio quindi non è legato alla gestione del processo di sviluppo.

Daily Scrum

Il daily scrum è una riunione tra gli sviluppatori e lo scrum master che assume il ruolo di moderatore. In massimo 15 minuti ogni sviluppatore deve rispondere a tre domande:

- **cosa ho fatto ieri?**;
- **cosa farò oggi?**;
- **ci sono ostacoli?**.

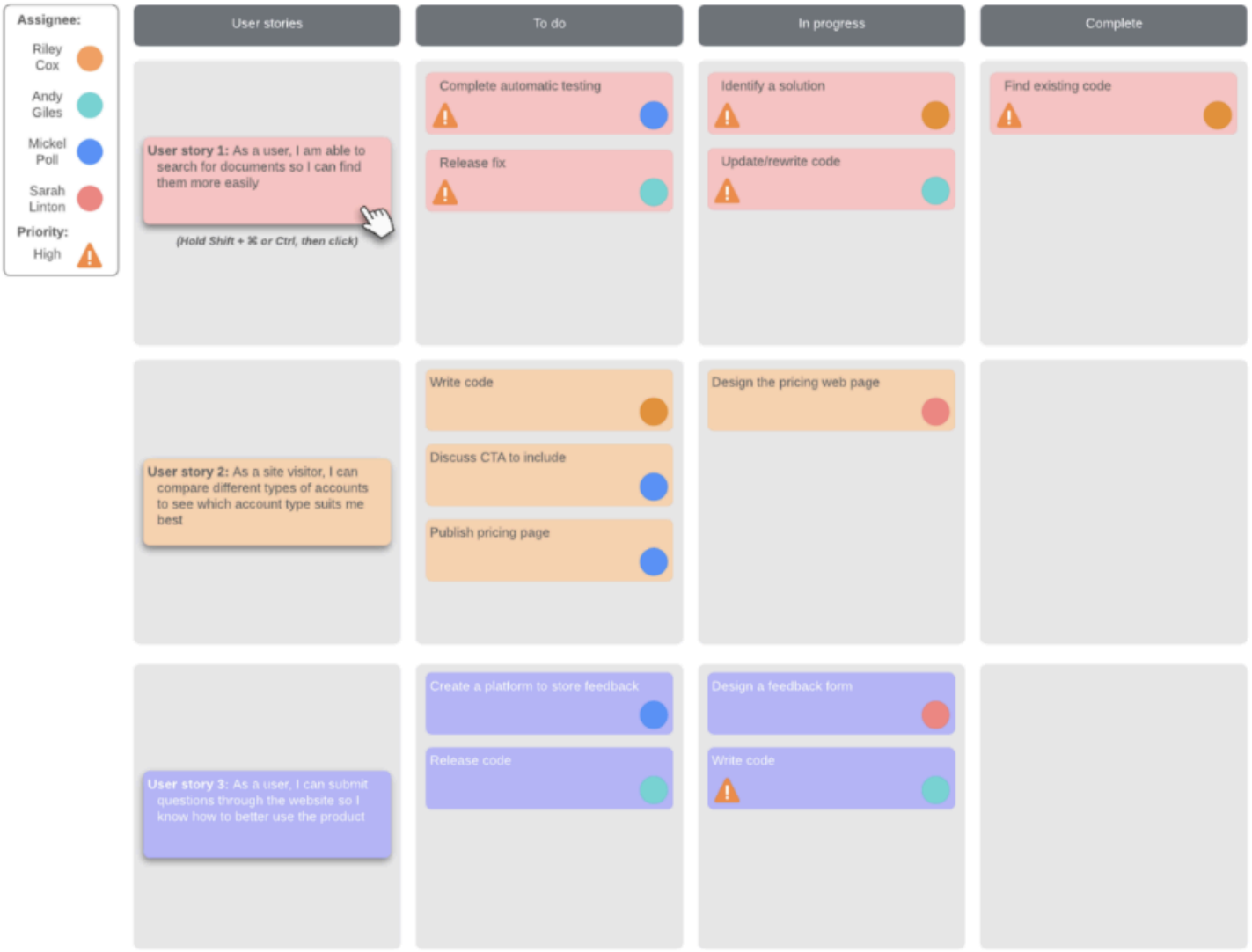
Eventuali ostacoli verranno rimossi dallo scrum master.

Sprint review

Nella sprint review è presente anche il product owner. Si analizza l'incremento prodotto, ma si ha anche un feedback dal cliente, che vede il prodotto prendere forma. Poiché intanto è passato del tempo, si ha la possibilità di modificare i requisiti, aggiornando il product backlog in base al feedback dell'utente. Sebbene si proceda per incrementi, l'utente ha un product backlog modificabile nel tempo: si ha quindi un approccio ibrido tra incrementale e iterativo.

Scrum board

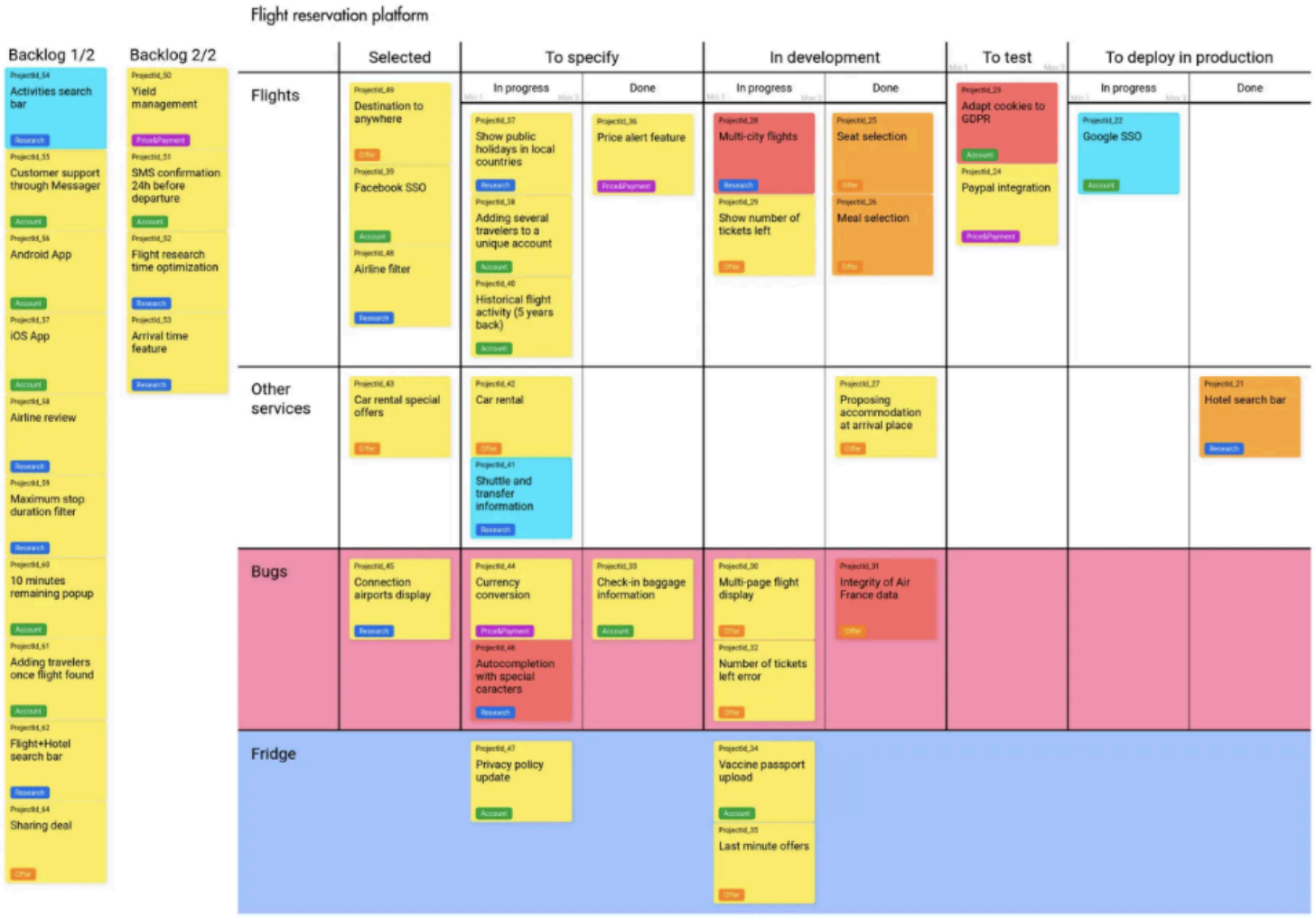
Tipicamente si usa una scrum board, che permette di gestire lo sprint backlog. Si parte dalle user stories, tracciando i task da svolgere, quelli in progress e quelli completati.



Kanban Board

In alternativa alla Scrum board si utilizza la **Kanban Board**, una metodologia agile estrema che nasce da un'esperienza industriale. La Kanvan Board è una lavagna in cui tutti possono vedere i task da svolgere. In questo modo si condivide una visione di insieme di quali sono gli obiettivi del progetto, condividendone lo stato di avanzamento.

La kanban board permette di gestire il product backlog e le attività prese in considerazione. Queste board inoltre sono più facili da adattare al contesto.



Gestione dei problemi

Scrum mira all'organizzazione funzionale del team di sviluppo pur essendo una metodologia agile tramite un development incrementale, con una negoziazione in corso d'opera dei requisiti del sistema. A differenza di XP, in cui i requisiti scritti potevano essere modificati in quanto si ha il cliente in sede, in Scrum lo sprint goal **non è negoziabile**. Infatti gli sviluppatori potrebbero sopravvalutare le loro capacità e realizzano che lo sprint backlog non può essere implementato nella sua interezza. Allora gli sviluppatori possono rinegoziare lo sprint backlog.

Scaling del framework

Finché il progetto è piccolo lo Scrum si adatta anche a team piccoli. Se i team sono molto grandi, tuttavia, il daily scrum di 15 minuti non sarebbe sufficiente. Si divide allora il team in sotto-team, dove in ogni team è presente uno scrum master. In ogni sotto-team si ha un daily scrum con il proprio scrum master.

Features non presenti

Se tutto il product backlog non può essere implementato durante il tempo schedulato per il progetto, allora:

- si svolgono prima i task con priorità più alta durante lo sprint;
- **feature non essenziali che non rientrano nello schedule possono essere scartate**;
- se le feature essenziali non rientrano nei tempi previsti, si negozia con il cliente.

Scrum e XP

Si potrebbero applicare le 12 pratiche di XP all'interno del framework Scrum. Ogni giorno dello Scrum prevede un daily scrum in cui ci si siede a lavorare. I primi giorni ci si occupa della scrittura dei test. Nei successivi si scrive il codice. Infine, si scrive il refactoring. Si passa quindi alla sprint review e successivamente allo sprint retrospective, chiudendo l'iterazione. Questa fusione permette di prendere il "buono" di XP, fornendo uno sviluppo guidato da una progettazione gestionale del progetto.

Le principali differenze sono le seguenti:

- in XP le iterazioni sono volontariamente corte, mentre in Scrum gli sprint possono essere più lunghi in virtù del progetto;

- in XP si ha una totale flessibilità con i cambiamenti, mentre in Scrum si ha più rigidità, perché i goal non vengono cambiati durante lo sprint;
- in XP il focus è sulle pratiche ottimali per dare qualità al lavoro pur essendo un lavoro di codifica svolto in un contesto di continuo aggiornamento, metnre in Scrum il focus è sul team che si organizza in modo agile ma che riesce a dare delle garanzie che le attività procederanno con le tempistiche pianificate;
- in XP il cliente determina l'ordine dei requisiti da sviluppare, mentre in Scrum il cliente decide il goal, ed è il team che decide i task del goal da svolgere prima, stabilendo delle priorità. Quindi l'ordine dei task è dettato in XP dal cliente, mentre in Scrum la priorità è un aspetto tecnico che viene gestito dagli sviluppatori.

Manutenzione

I requisiti di sistema cambieranno mentre il sistema è in sviluppo in quanto l'ambiente cambia. Quando un sistema viene installato in un ambiente, **esso cambia quell'ambiente** e quindi cambia i requisiti di sistema. Il sistema allora dev'essere mantenuto in modo tale da rimanere **utile** in un ambiente. Le modifiche da applicare sono di tre tipi:

- modifiche **riparative**. Si scopre durante il funzionamento del sistema che esso ha dei bug o dei malfunzionamenti;
- modifiche **adattative**. Uno dei cambiamenti nel contesto operativo richiede necessariamente di intervenire nel software affinché esso continui a funzionare (y2k ad esempio);
- modifiche **additive**, se bisogna aggiungere o modificare funzionalità del sistema.

Circa due terzi delle modifiche da applicare sono di tipo additivo.

Costo della manutenzione

Il costo della manutenzione è tale che esso è da due a cento volte di più del costo di sviluppo. In più, questo costo dipende da come abbiamo lavorato prima, ad esempio il costo dell'aggiunta di una funzionalità dipende da come abbiamo implementato il sistema. A prescindere da quanto possa essere fatto bene, il software **invecchia**: più tempo passa, più queste modifiche aumentano il debito tecnico. Le aggiunte non possono sempre essere fatte bene, per cui inevitabilmente si va incontro ad un momento finale.

Ciò che si risparmia quindi in tempo di sviluppo potrebbe aumentare eventuali spese di manutenzione.

Dinamica dell'evoluzione del programma

Sono state fatte delle osservazioni in relazione all'evoluzione del programma:

- **legge del cambiamento continuo**: un sistema usato nel mondo reale dev'essere necessariamente modificato, altrimenti diventerà inutile;
- **legge della conservazione di familiarità**: i cambiamenti incrementali in ogni release sono approsimativamente costanti. Ciò che cambia è una parte sempre della stessa importanza;
- **legge dell'incremento di complessità**: la struttura del sistema diventerà più complessa, e risorse extra devono essere considerate per mantenere e semplificare la struttura;
- **legge della large program evolution**: l'evoluzione del programma è un processo auto-regolatorio, per cui gli attributi del sistema sono invariati da rilascio a rilascio;
- **legge di stabilità organizzativa**: il tasso di sviluppo di un programma è approssimativamente costante;
- **legge della crescita continua**: per mantenere la soddisfazione dell'utente, le funzionalità offerte dal sistema devono crescere continuamente;
- **legge del declino di qualità**: la qualità del sistema degrada se non viene adattato ai cambiamenti dell'ambiente in cui opera.

Queste leggi, note come **leggi di Lehman**, sono applicabili generalmente a sistemi grandi sviluppati da aziende.

Impatto sui costi di manutenzione

I fattori che incidono sui costi di manutenzione sono di diversa natura:

- **stabilità di squadra**. Se si mantengono sempre le stesse persone che conoscono la storia del prodotto, si abbattono i prezzi di manuntenzione;
- **responsabilità contrattuale**. Se gli sviluppatori del sistema non sono coinvolti a livello contrattuale alla manutenzione del sistema, non percepiscono responsabilità;
- **team skills**. Le abilità del team influenzano la qualità del prodotto;
- **età del prodotto**. Più un prodotto invecchia, maggiori sono i costi.

Metriche

Abbiamo bisogno di metriche per misurare la qualità del sistema. La qualità di un sistema si può esprimere sotto molti aspetti. La maggior parte di questi aspetti sono **qualitativi**, ossia sono misurati in termini non numerici (riusabilità bassa/alta, affidabilità bassa/alta, ecc.). Analizziamo ora le metriche misurabili **quantitativamente** che riguardano la complessità del sistema. "Complessità" in questo caso è un termine lasco, ossia si misura un aspetto **volumetrico** del sistema, ossia con una dimensione spaziale.

La complessità allora dipende dalle strutture di controllo e dalle strutture dati, nonché dalla struttura dei moduli.

Righe di codice

Possiamo misurare la dimensione del sistema con le **righe di codice**. La riga di codice è una qualsiasi riga del programma che **non è un commento** o **non è vuota**. Legata alle righe di codice è la **percentuale di commenti**. Risulta molto utile commentare il software, e si suggerisce di avere una percentuale di commenti pari al 30% rispetto alla lunghezza complessiva del file. Se un file ha 100 righe di codice, almeno 30 in più dovranno essere commenti.

Metriche di Halstead

I termini più piccoli che possiamo considerare sono gli operatori e gli operandi. A partire dal numero di operatori totali e unici e dal numero di operandi totali e unici si possono derivare le seguenti metriche:

- **lunghezza** (N_1, N_2): somma di operatori totali e operandi totali;
- **vocabolario** (n_1, n_2): somma di operatori unici e operandi unici;
- **volume**, definita come la lunghezza del programma moltiplicata per il logaritmo in base due del vocabolario. Questo valore dovrebbe essere compreso per una funzione tra 20 e 1000, mentre per un singolo file da 100 e 8000. Questi numeri sono stati studiati dalla VerifySoft;
- **difficoltà**, definita come $D = \frac{n_1}{2} \cdot \frac{N_2}{2}$;
- **effort**, definito come $E = V \cdot D$;
- **tempo stimato per compresione**, definito come $T = \frac{E}{18}$;
- **bug consegnati**, definito come $\frac{E^{\frac{2}{3}}}{3000}$

Queste metriche sono **insensibili** a come scriviamo il codice. Infatti un `if` può essere scritto, ad esempio, in un'unica riga, mentre in questo caso si ha un'indicazione che non dipende da quante volte andiamo a capo, ma solo quali e quanti operatori e operandi usiamo. Queste metriche, inoltre, derivano da calcoli basati su interpolazione.

Complessità di McCabe

La **complessità di McCabe** corrisponde alla **complessità ciclomatica del grafo di controllo di flusso**. Questo numero esprime il numero di percorsi per eseguire tutte le funzionalità rappresentate dal grafo, e quindi fornisce il numero minimo di test ed è una misura della complessità del software.

Chiaramente è ovvio che il numero di McCabe **non corrisponde al numero totale di percorsi del grafo di controllo di flusso**.

Indice di manuntenibilità

Sulla base di interpolazioni si possono mettere in gioco il volume di halstead, le righe di codice e la complessità di McCabe ($V(g)$) per calcolare l'**indice di manuntenibilità**:

$$MI_1 = 171 - 5.2 \ln(V) - 0.23V(g) - 16.2 \ln(LOC)$$

Questo numero viene "corretto" utilizzando una funzione che da un peso in virtù del numero di commenti presenti nel codice:

$$MI_2 = MI_1 + 50 \sin \sqrt{2.46 \cdot CP}$$

dove CP indica la percentuale di commenti nel codice. Solitamente valori sopra 85 sono molto positivi e indicano che il codice è facilmente mantenibile.

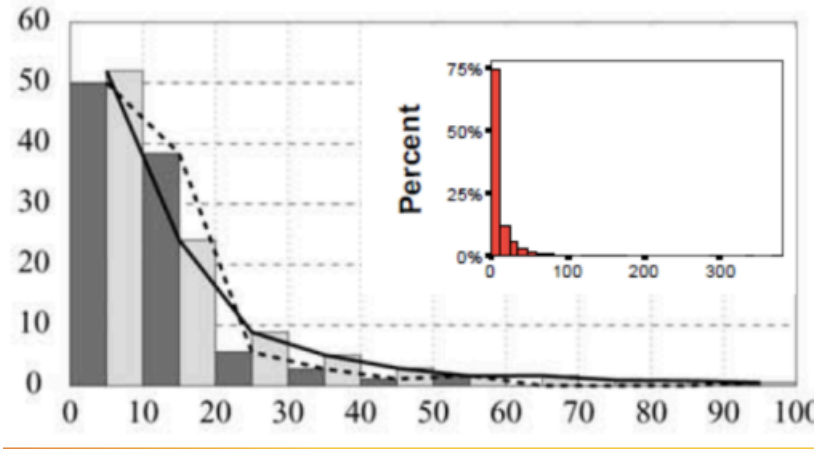
Metriche di Chidamber e Kemerer

Possiamo usare altre metriche specifiche per un sistema Object-Oriented.

Weighted Methods per Class (WMC)

Data una classe C con metodi $M_1, M_2, \ldots, M_k$ con relative misure di complessità per i metodi $c_1, c_2, \ldots, c_k$, si ha che la **Weighted Methods per Class** è data dalla somma $c_1 + c_2 + \ldots + c_k$. Se come misura si utilizza la metrica "identità", allroa la somma esprime solo il numero di metodi presenti. Più grande è il WMC maggiore è l'impatto sulle classi che ereditano questa.

Un sistema di alta qualità vede una curva della WMC con una forte distribuzione verso l'origine dell'asse y. Empiricamente inoltre si è anche visto che WMC è correlato al numero di bug.



Nel grafico soprastante notiamo che esiste un numero di classi pari a zero con una WMC alta, il che significa che maggiore è il numero di metodi, maggiore è la probabilità di avere errori.

Profondità dell'albero di ereditarietà (DIT, Depth Inheritance Tree)

Se ci sono classi che hanno tanti predecessori, l'albero dell'ereditarietà sarà più profondo, indicativo di un sistema più complesso. Empiricamente si è osservato che è meglio evitare valori superiori a 5.

Chiaramente bisogna decidere come dev'essere interpretata la profondità in virtù dell'importazione di librerie. In effetti potremmo implementare una libreria, che a sua volta estende funzionalità da un'altra liberia, che a sua volta estende altre funzionalità di un'altra libreria. Bisogna quindi decidere se:

- includere nella metrica **tutte le librerie coinvolte nell'ereditarietà**;
- includere nella metrica **solo le librerie importate**;
- **non includere le librerie importate**.

Number of children (NOC)

La metrica **number of children** conta il numero di sottoclassi di una classe. Un NOC alto potrebbe essere indicativo di un'alto riuso della classe base. Oppure potrebbe essere un segnale, se il NOC diventa troppo alto, che il sistema è progettato male: la superclasse potrebbe essere troppo astratta, può essere messa in comune con molte cose in quanto troppo generica. Quindi l'astrazione è errata. Empiricamente un NOC alto è indicazione di minori fault nel sistema.

Coupling Between Objects (CBO)

L'accoppiamento tra classi è inteso come l'utilizzo di una variabile o di un metodo di un'istanza di un'altra classe all'interno del codice della classe presa in considerazione. Si individua allora quant'è il numero di classi che sono accoppiate: tale numero è il **Coupling Between Objects**. Il CBO può essere calcolato con:

$$\mathrm{CBO} = \mathrm{FanOut} - \mathrm{FanIn}$$

Dove FanOut e FanIn sono rispettivamente il numero di classi usate dalla classe analizzata e il numero di classe che usano la classe analizzata.

Un alto CBO è sconsigliato, in particolare un CBO superiore a 14 previene il riuso. Inoltre, il mantenimeno è più difficile, visto che singole modifiche alle classi influenzano le altre.

Response For a Class (RFC)

Quando si invoca il metodo di una classe, il codice del metodo può invocare il codice di un metodo di un'altra classe. Il **Response For a Class** è il numero di metodi locali di una classe sommato al numero di metodi remoti chiamati da altri metodi (nota: la somma non dev'essere reiterata. Ad esempio, se m1->m2->m3, allora RFC per la classe con m1 è pari a 2, non a 3). RFC esprime

quanto codice viene eseguito da una classe quando viene invocato un metodo. Più è alto RFC, maggiore è la probabilità che ci siano errori e la difficoltà nel debugging e nel testing, nonché la complessità della classe.

Lack Of Cohesion in Methods (LCOM)

La metrica **Lack of Cohesion in Methods** misura la coesione all'interno dei metodi. Indichiamo con P le coppie di metodi distinti in una classe che non condividono variabili di istanza. Indichiamo con Q le coppie di metodi distinti in una classe che condividono variabili di istanza. Allora:

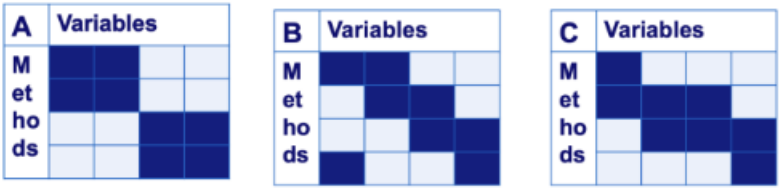
- se $P > Q \implies LCOM = P - Q$;
- se $P \leq Q \implies LCOM = 0$.

Una seconda versione di LCOM permette di ottenere valori anche negativi, definendo $LCOM = P - Q$. Anche con questa leggera miglioria la metrica è ancora migliorabile. Infatti la metrica non ci fornisce indicazioni su quanto è grande un gruppo di metodi che non sono coesi.
Indicando con v il numero di attributi, m il numero di metodi e $m(V_i)$ il numero di metodi che accedono a V_i, definiamo:

$$LCOM = \frac{\frac{1}{v} \cdot \sum_{i=1}^{v} m(V_i) - m}{1 - m}$$

Il valore di LCOM in questo caso è compreso tra 0 e 2. Quando la coesione è massima, tutti i metodi utilizzano tutte le variabili, ossia $m(V_i) = m$. Si può superare il valore 1 quando un metodo non utilizza alcun metodo: in questo caso si ha infattti che $m(V_i)$ = 0. All'estremo nessun metodo utilizza nessuna variabile, per cui dalla somma si ottien $m - \frac{1}{m-1}$, che è minore o uguale a 2.

Neanche con questa formulazione tuttavia riusciamo a distinguere situazioni in cui si hanno gruppi separati di metodi che utilizzano un sottoinsieme di variabili. La metrica non fornsice informazioni aggiuntive neanche in questo caso:



Nell'esempio `A` è meno coeso di `B` e `C`.

Si possono rappresentare le dipendenze con un grafo. Ogni nodo è un metodo e un arco indica che i due metodi accedono alla stessa variabile. Allora LCOM indica il **numero di componenti fortemente connesse nel grafo**. In particolare:

- con 0 non si hanno metodi;
- con 1 si ha una componente coesa;
- con 2 o più si ha mancanza di coesione.

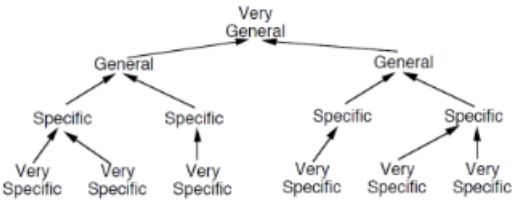
Violazione di Demetrio

La legge di Demetrio nel contesto della OOP è semplicemente la legge dell'**utilizzo di un solo punto**. Ossia, basta fare `b.m()`, e non `b.c.m()`. Nel secondo caso, significa che un oggetto che si interfaccia con `b` deve conoscere anche l'interfaccia di `c`. Questa dipendenza non è voluta, in quanto è esplicita. Le dipendenze indirettte permettono di far dipendere un livello solo da quello successivo, cosa che nel caso `b.c.m()` non è vera.

21 - Ultimi design pattern

Chain of responsibility

Supponiamo di trovarci in una gerarchia di **responsabilità**, ossia di avere un insieme di oggetti che possono gestire collettivamente un insieme di servizi. Questo insieme di servizi viene gestito con granularità differente da ogni oggetto: ai livelli bassi gli oggetti forniscono compiti molto specifici, mentre ai livelli più alti servizi più generici.
Vogliamo alle spalle di questi oggetti un'altra serie di oggetti che sono in grado di fornire per uno o più oggetti di prima linea la stessa tipologia di servizio ma in modo più generico:



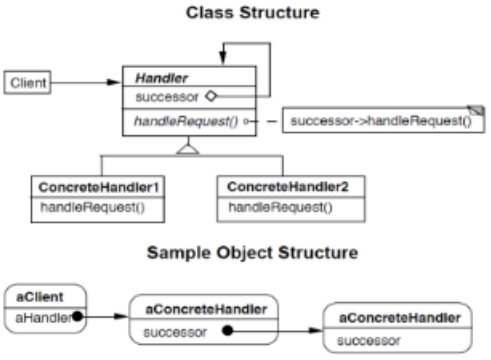
Il client allora richiede un servizio ma non saprà qual è l'oggetto specifico che soddisferà la richiesta: esso farà solo riferimento ad un'interfaccia, ma il meccanismo di catena di responsabilità è del tutto nascosto. Se inoltre il client richiede un servizio ad un oggetto più di basso livello, ma tale oggetto non è adatto a svolgere il compito, allroa la chiamata viene dirottata ad un oggetto di più alto livello che riesce a svolgere il task in una maniera un po' più generale.

Il pattern nasce dalla scrittura di sistemi di **help contestuale**, ossia messaggi di aiuto per determinate sezioni di interfacce. Se la richiesta di aiuto per un determinato pannello non è soddisfabile, questa richiesta viene dirottata sull'aiuto della finestra generale. Questo è il motivo per cui alcuni messsaggi di help specifici in realtà si rivelano più generici.

La caratteristica peculiare del pattern risiede nel fatto che avendo una catena di responsabilità, il client **non sa chi fornirà il servizio**, de-responsabilizzando l'oggetto a cui è stata fatta la richiesta nel caso in cui esso non sappia gestirla. La struttura della gerarchia è gestita tramite legami di **composizione**, per cui il pattern ha **object scope**. La struttura, inoltre, dev'essere assemblata a **run-time** e non a tempo di compilazione. L'invocazione della richiesta viene gestita tramite una ricorsione object oriented.

Il pattern propposto prende il nome di **catena di responsabilità**, sebbene in realtà si abbia una **gerarchia di responsabilità**.

Schema del pattern



I partecipanti alla catena devono estendere la classe astratta `Handler`. La classe astratta prevede un attributo privato `successor` e un metodo `handleRequest()`, che di default richiama il metodo `handleRequest()` su `successor`. Più `ConcreteHandler` definiscono il comportamento nel caso della richiesta.

Chiaramente affinché la catena di funzionalità sia gestita correttamente è necessario comporre nella maniera corretta i successori. Si prevede quindi una fase di impostazione della gerarchia.

Applicabilità

Il pattern disaccoppia l'invoker della richiesta da chi serve la richiesta. Chi riceve l'invocazione non è quindi chi soddisfa nello specifico la richiesta, ma potrebbe essere uno tra i tanti handler in grado di farlo, anche in maniera non specifica. Si ha quindi un **accoppiamento lasco**.

Possiamo anche immaginare il chain of responsibility pattern come un modo di gestire il **polimorfismo**. Nel polimorfismo un oggetto `Super` può anche essere un oggetto `Derived`, e sebbene il client lavori sull'interfaccia dell'oggetto `Super` in realtà la

chiamata a metodo può anche essere dirottata sul metodo dell'oggetto `Derived`. Il polimorfismo prevede una struttura gerarchica **immutabile** e definita **in tempo di compilazione**. Anche nel chain of responsibility abbiamo un polimorfismo. Ma questa struttura non è una struttura di tipi, ma è una struttura dinamica che componiamo durante l'esecuzione del programma, e quindi può essere creata e modificata liberamente. Riusciamo perciò ad ottenere un **polimorfismo fuori gerarchia di classe**, con una logica che può essere variata.

Chain of Responsibility e Decorator

Il fatto di poter creare la struttura in tempo di esecuzione richiama il pattern Decorator, dove l'estensione di funzionalità viene gestita con una struttura definita a run-time. Tuttavia:

- il set di funzionalità non è prefissato per un certo oggetto, mentre nel chain of responsibility si suppone che **una gerarchia implicita sia preesistente** per la gestione delle richieste;
- entrambe le strutture sono modificabili nel tempo. Tuttavia la catena di responsibilità prevede una catena di oggetti non lineare, mentre il decorator prevede una catena lineare;
- mentre nel chain of responsibility si cerca di **reindirizzare una richiesta** ad un oggetto che possa soddisfarla, nel decorator pattern ogni decoratore **contribuisce alla richiesta**.

Rappresentare le richieste

Se ci sono molteplici richieste, è possibile implementare queste con metodi con firme precise e diverse dagli altri. Questo tipo di rappresentazione delle richieste prende il nome di **richieste cablate**. Tutte le richieste gestibili però devono essere note fin dal principio. Questo però è limitante, in quanto se bisogna aggiungere una responsabilità, è necessario aggiungere questa ad ogni oggetto della catena di responsabilità.

Invece di codificare le richieste con metodi separati, è possibile implementare un metodo generico `handle()` che riceve in input un parametro che indica la richiesta. Questo parametro può essere un intero o una stringa. Ad esempio:

```
public void handle(String request) {
    switch(request) {
        case "Open" : //something
        case "Close": //something
        case "New": //something
        default:
            successor.handle(request);
    }
}
```

Questo approccio permette di codificare anche un numero variabile di richieste.

Una soluzione più intelligente risiede nell'**incapsulamento** del metodo che soddisfa la richiesta in un oggetto che viene passato alla funzione `handle()`:

```
abstract class SingleHandler {
    ...
    public void handle(Request data) {
        successor.handle(data);
    }
}

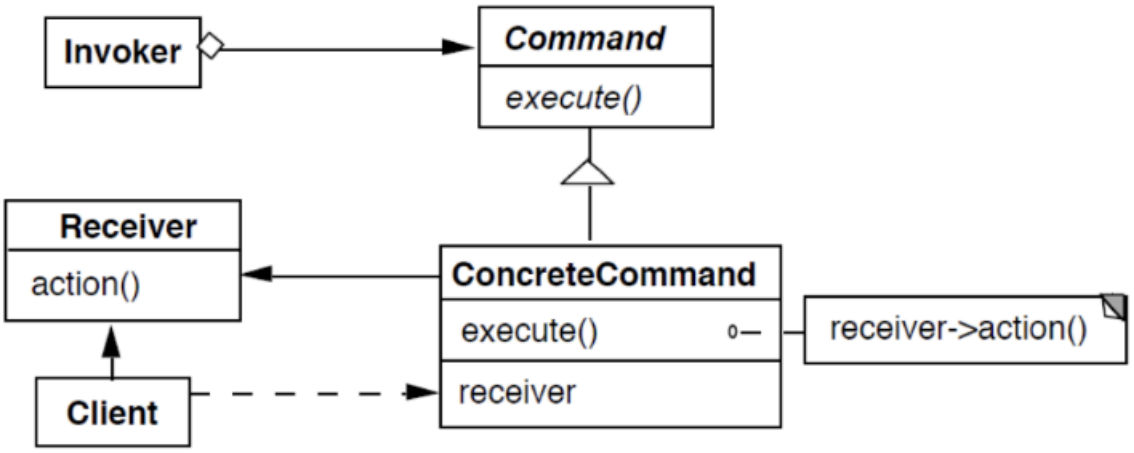
class ConcreteHandler extends SingleHandler {
    public void handle(Open data) {
        //do something with data
    }
}
```

L'ADT `Request` rappresenta una richiesta generica. Possiamo sottoclassare `Request` per indicare le diverse tipologie di richieste, spostando la gerarchia tra gli oggetti che devono gestire la richiesta nei tipi di richieste. Sono le richieste che devono essere gestite gerarchicamente.
Chiaramente per capire se il tipo di parametro ricevuto è corretto è necessario ispezionare il tipo di parametro passato utilizzando la Reflection API. Quindi una volta aver ricevuto la chiamata `handle(Request data)`, si ispeziona il tipo di `data` e si può redirezionare la chiamata correttamente.

Command

L'obiettivo del **command pattern** è incapsulare la richiesta di un'operazione in un oggetto. In questo modo si disaccoppia del tutto l'oggetto che invoca un'operazione da quell oche la esegue. In pratica:

- l'invoker invoca un servizio tramite l'interfaccia di un oggetto;
- l'interfaccia dell'oggetto prevede un metodo `execute()` ;
- l'implementazione dell'interfaccia contiene un oggetto `receiver` a cui viene dirottata la richiesta.



Se il client configura il `ConcreteCommand` in maniera differente, la stessa invocazione si traduce in un comando differente. Questo ancora una volta permette di sfruttare il mecccanismo del polimorfismo senza gerarchie. Il client è il pezzo dell'applicazione che configura i command perché funzionino. L'invoker può essere un altro elemento di un applicazione che utilizza dei command attraverso l'interfaccia `Command` .
Un `ConcreteCommand` non è altro quindi che un oggetto che implementa l'interfaccia `Command` e che utilizza funzionalità esterne di un oggetto che incapsula. Il `ConcreteCommand` può:

- semplicemente rimbalzare la chiamata all'oggetto incapsulato;
- partecipa alla creazione della funzionalità. L'oggetto incapsulato in questo caso aiuta nell'esecuzione del comando ma la funzionalità viene fornita dal comando stesso.

Il pattern è stato identificato facilmente durante la scrittura di **framework** per la gestione di applicazioni. Il command pattern infatti si trova spesso nel momento in cui bisogna implementare strumenti per creare dei menu.

Applicabilità

Una situazione simile al command pattern è riscontrata nell'**observer pattern**, in cui il metodo `update()` dell'observer corrisponde a diverse funzionalità in virtù dell'observer coinvolto. Il meccanismo del command pattern, tuttavia, è ancora più sofisticato per il fatto che non definiamo delle funzioni di callback staticamente, ma passiamo comandi a run-time.

Per avere in tempi diversi definizione, invocazione ed esecuzione di comandi l'oggetto command può differire l'esecuzione dell'azione. Il command, inoltre, può anche elaborare dei dati relativi alle richieste ricevute e ispezionare il risultato, in quanto **il command incaspula il receiver**, a differenza di altri pattern come il decorator pattern, dove ogni funzionalità è ignara del risultato delle chiamate di altre funzionalità aggiunte al componente base.

Possiamo anche assemblare funzionalità di alto livello componendo più command assieme. In questo modo funziona il meccanismo delle **macro**. Il meccanismo del command permette anche di eseguire azioni come **undo**, perché nel command siamo in grado di incapsulare il risultato dell'operazione. Quindi basta salvare l'elenco dei comandi con i rispettivi risultati.

Sebben sia un pattern che garantisce molta flessibilità, paghiamo prezzo per:

- la proliferazione delle classi;
- chiamate indirette;
- aumento della complessità. L'indirezione è un bene per disaccoppiare l'invoker dalle specifiche richieste, ma l'**invoker non potrà vedere l'effetto dell'invocazione attivata**. Il pattern ha lo scopo di rendere flessibile l'invocazione, ma l'esagerazione nell'utilizzo dei comandi può rendere incomprensibile la codifica di un algoritmo.

Implementare i comandi

Chiaramente se ci sono n comandi da implementare si ha la necessità di implementare n classi, una per comando. Possiamo fare di meglio utilizzando un'unica classe `Command` che incapsula:

- oggetto ricevente;
- il metodo del comando;
- argomenti da passare.

L'implementazione della classe utilizza la reflection:

```
public class Command {
    private Object receiver;
    private Method command;
    private Object[] arguments;

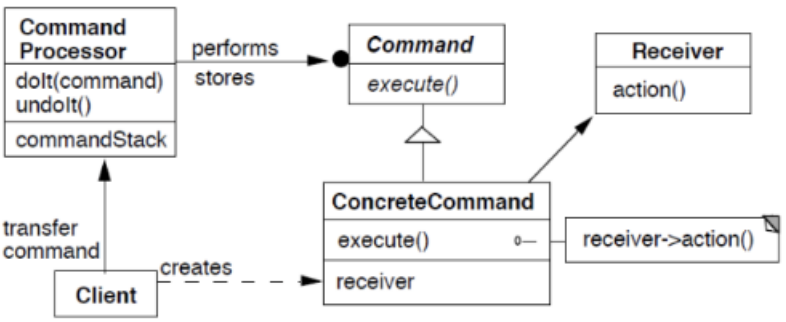
    public Command(Object receiver, Method command, Object[] arguments) {
        this.receiver = receiver;
        this.command = command;
        this.arguments = arguments;
    }
    public void execute() {
        command.invoce(receiver, arguments);
    }
}
```

Command processor

Possiamo gestire i comandi in un unico posto utilizzando un **command processor**. Il command processor si occupa di gestire i comandi di vario genere, istanziandoli ad ogni invocazione. Il command processor dispone di un set di oggetti command pre-configurati e da far corrispondere alle richieste. All'interno del command processor possiamo inserire anche delle istanze di comandi staticamente e applicare il **prototype pattern per clonare i comandi**. Infine, se vogliamo un solo command processor possiamo applicare il singleton pattern.

Il command processor quindi è utile per:

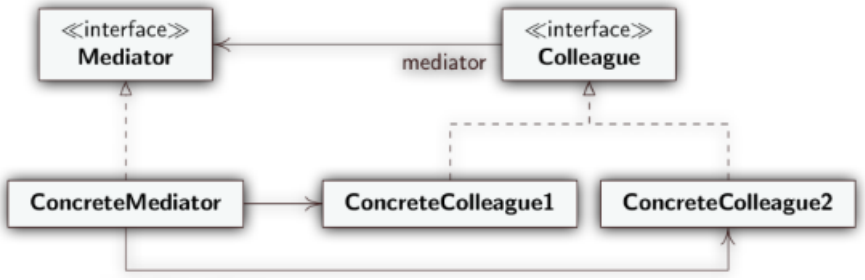
- schedulare l'esecuzione dei comandi;
- fare logging della sequenza d'esecuzione;
- conservare le istanze di tutti i comandi richiesti;
- memorizzare la sequenza di azioni, implementabile tramite una stack;
- implementare un **sistema transazionale**, rendendo una serie di comandi una transazione atomica.



Mediator

In un'interfaccia grafica solitamente abbiamo una serie di oggetti che dipendono tra loro. Chiaramente se i componenti dipendono strettamente tra loro, ogni singola modifica su un qualsiasi oggetto impatta gli altri. Vogliamo ridurre queste dipendenze in modo tale da avere un accoppiamento lasco e garantire il riuso.

Possiamo introdurre un **intermediario**, facendo si che invece di esserci un insieme di interdipendenze le dipendenze siano rivolte ad esso. Ogni componente, detto **collega**, interagisce con un mediator, e il mediator dipende dai singoli colleghi:



Nella sua interfaccia il mediator dovrà incapsulare i metodi e le interfacce esposte dai colleghi. Chiaramente il mediator **non ingloba le funzionalità, ma solo le interfacce**, per cui dirotterà le chiamate. Ad esempio, se una classe `A` chiama un metodo di una classe `B`, allora `A` fa una richiesta al mediator, che dirotta la richiesta a `B`. Il mediator pertanto ha bisogno di rifrerimenti ai colleghi.

Il mediatore concreto necessita di conoscere le classi con cui deve mediare.

Differenze con Facade

Il mediator espone un'interfaccia interna e i colleghi riconoscono il fatto che stanno lavorando con un mediator. Non si tratta di una facciata rivolta all'esterno, come accade nel pattern Facade.
Il pattern Facade, inoltre, non prevede l'aggiunta di funzionalità, mentre il mediator incorpora tutta la logica di funzionamento che prima era distribuita tra le classi, e può anche incorporare nuove funzionalità, proprio per il fatto che il suo ruolo è quello di mediare tra le classi. Il mediator quindi **trasforma un algoritmo decentralizzato**, distribuito tra le classi che richiamavano metodi tra loro, **in un algoritmo centralizzato**.
Infine, il mediator **istanzia e mantiene i rfierimenti**, coordinando l'interazione tra i colleghi, mentre il facade semplifica l'accesso ai metodi riducendo le dipendenze del client dal sottosistema a cui il facade fa da facciata.

Varianti

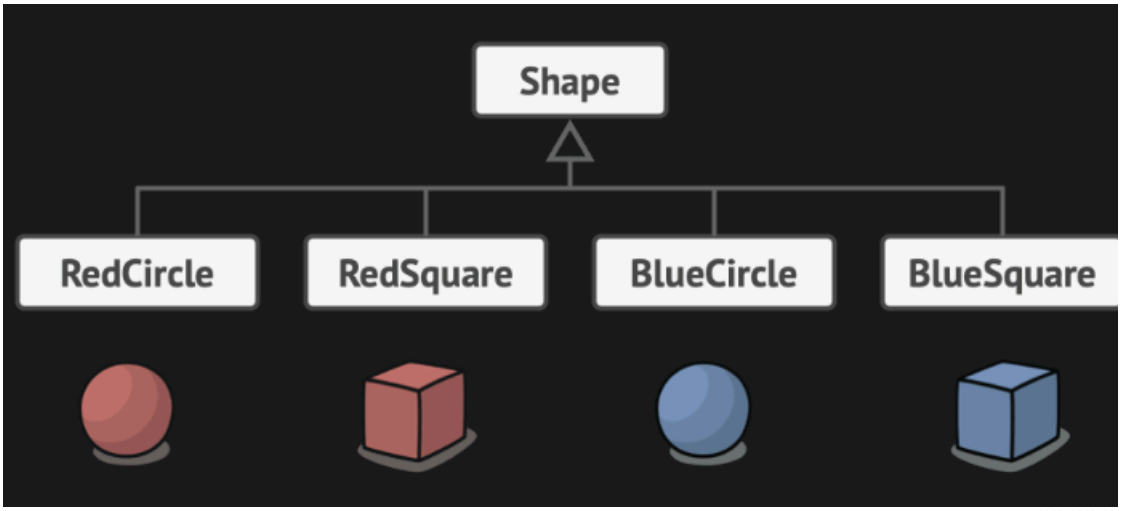
Possiamo far si che i colleghi accedano al mediator:

- implementando nel mediator dei **metodi espliciti** corrispondenti a quelli nei colleghi;
- implementando nel mediator un **metodo unico generico**, passando il colleague per individuare chi lo invoca. Il passaggio del collega mi aiuta quindi a che punto siamo dell'elaborazione;
- implementando nel mediator un **metodo con overloading**, identificando il tipo del collega passato.

Bridge

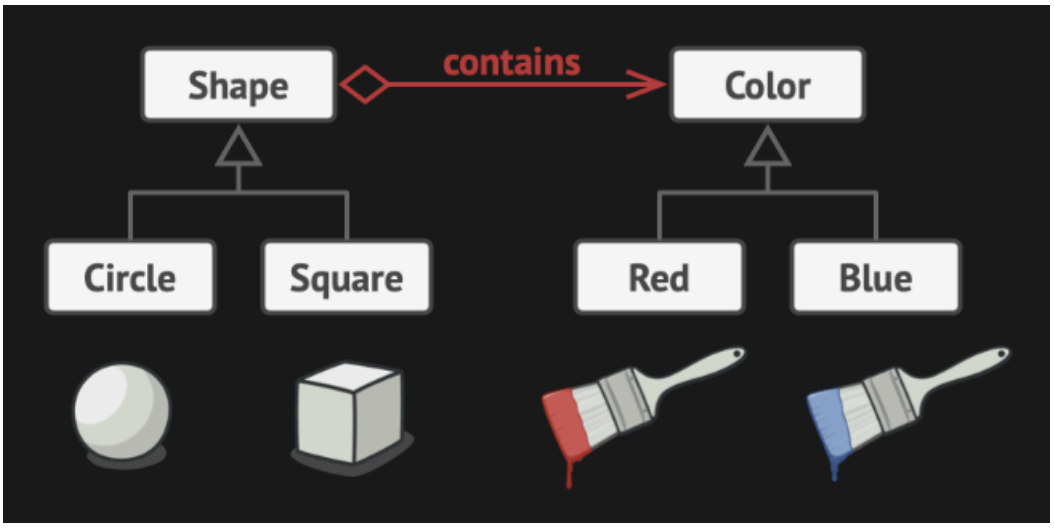
Lo scopo del pattern Bridge è quello di **disaccoppiare una gerarchia di astrazioni da una gerarchia di corrispondenti implementazioni**. Normalmente infatti quello che succede è che abbiamo un'astrazione, e l'astrazione può essere implementata in vari modi. Quello che abbiamo visto per ora è che dall'astrazione si derivano più versioni concrete, e quindi più varianti di implementazione. Facendo qussto si blocca la possibilità di espandere l'astrazione, ossia di **raffinarla**.

Supponiamo di avere una forma geometrica, ad esempio `Circle` e `Square`. Supponiamo che sia per `Circle` sia per `Square` vogliamo definire dei colori, ad esempio `Red` e `Blue`. Vogliamo allora implementare i colori per le figure definite. La struttura a cui arriviamo potrebbe essere la seguente:

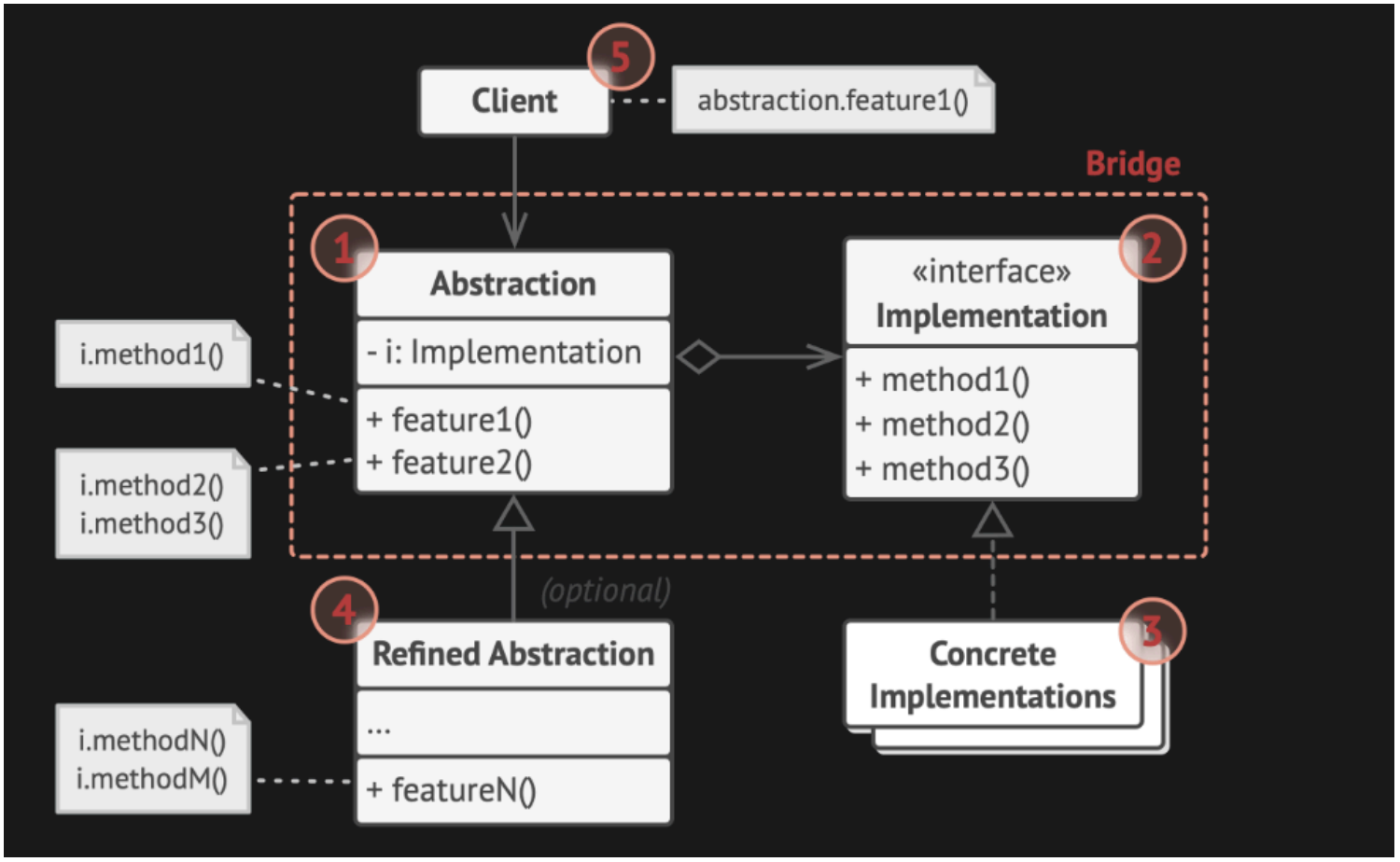


Se dovessimo aggiungere un nuovo colore, allora dobbiamo aggiungere altre implementazioni, ad esempio `YellowCircle` e `YellowSquare`. E se aggiungessimo una nuova forma, dobbiamo implementare i colori per questa nuova forma.

Il problema di questa struttura riguarda l'unificazione di astrazione e di implementazione. Vorremmo allora essere liberi di creare una qualsiasi forma e di stabilire il colore senza avere una proliferazione di classi esagerata. Possiamo allora **comporre l'implementazione nell'astrazione fornendo all'implementazione un'interfaccia**:



Il nostro client quindi può fare riferimento alla forma in relazione alle sue funzionalità, e all'interno delle funzionalità possiamo definire un comportamento in base all'implementazione che viene integrata nell'astrazione:



Questo approccio permette anche di definire un'implementazione anche a run-time in maneira dinamica.

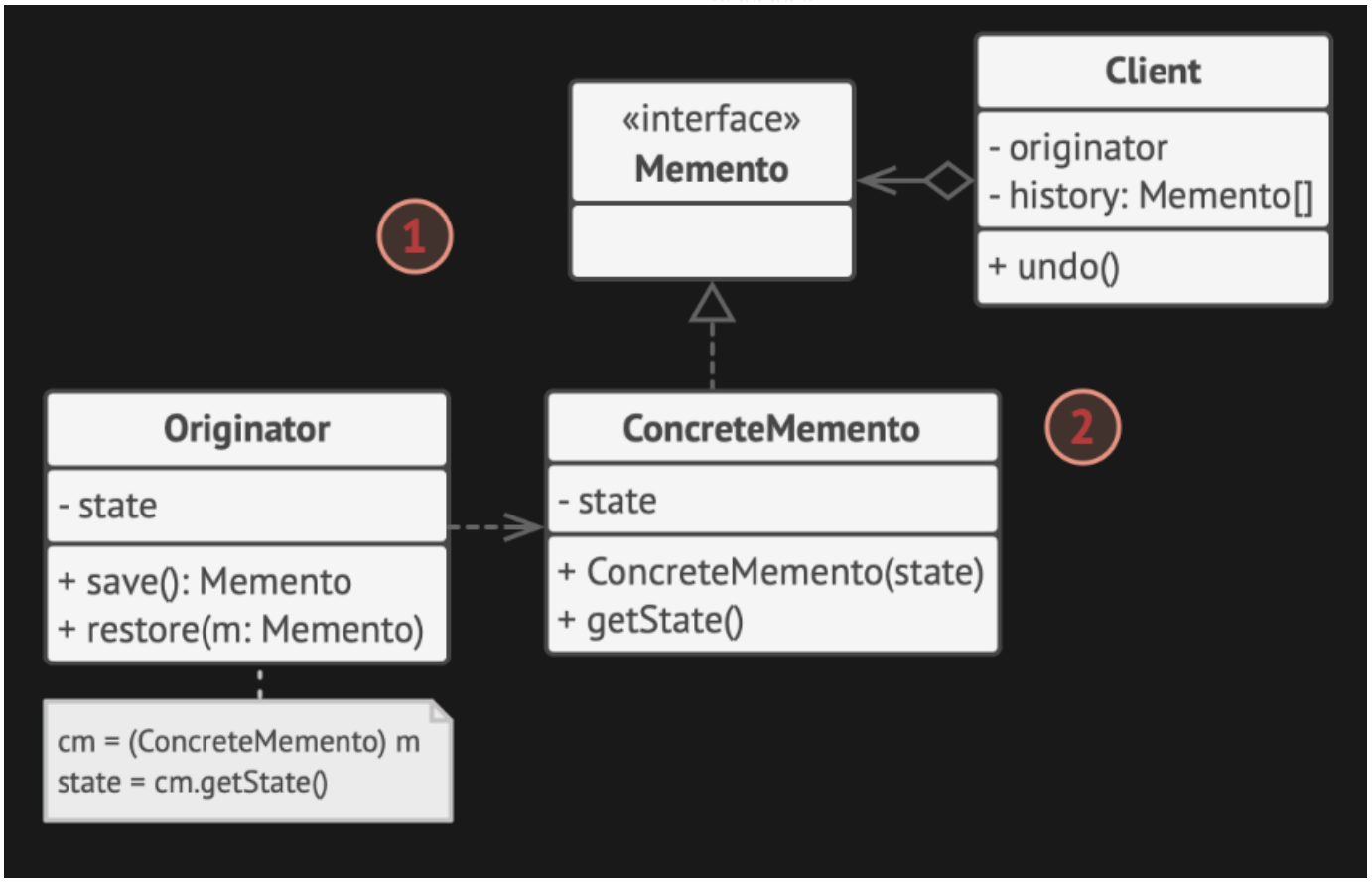
Bridge e Adapter

Il bridge è una scelta che viene applicata a **design time** per far evolvere separatamente astrazioni e implementazioni. L'adapter, invece, affronta il **problema inverso**: connettere **un**'astrazione ed **un**'implementazione, ma senza alterarle.
Il pattern adapter, inoltre, permette di far funzionare classi che sono completamente scorrelate tra loro. Inoltre, se si devono avere più alternative di implementazione, **si deve cambiare adapter**.
Il bridge permette di estendere l'astrazione a piacimento, mentre nell'adapter l'astrazione è un **punto fermo**.

Memento

Ogni oggetto ha un suo stato, però non sempre è necessario che la gestione dei salvataggi dello stato sia una funzionalità che bisogna internalizzare dentro l'oggetto stesso. Lo stato di un oggetto tuttavia dovrebbe essere la parte più privata di un oggetto, non accessibile da nessuno. Quindi da una parte si porta fuori e dall'altra si deve proteggere.

Incapsulando lo stato ottenuto in un oggetto. Utilizziamo un oggetto `Memento` che tiene la copia dello stato:



Un `Originator` istanzia e usa (freccia tratteggiata) il `Memento`, un oggetto con due metodi: `getState()` e `setState()` (non presente in foto). Le funzionalità per caricare e conservare lo stato non sono dentro l'`Originator`, ma dentro il `Memento`.
Il pattern permette salvataggi incrementali. Se la funzionalità di salvataggio è nel `Memento`, allora esso può avere una copia dello stato precedente e quindi potrebbe anche salvare solo le modifiche allo stato. Ci permette inoltre di non esporre lo stato all'esterno.

Usare i `Memento` può essere costoso se lo stato è complesso. Fare la copia degli stati potrebbe richiedere molto tempo e memoria, per cui potrebbe non essere conveniente. Tuttavia, è un metodo per permettere ad una classe di poter ripristinare uno stato senza esporlo all'esterno.

Memento e Iterator, Memento e Clone

La navigazione può essere implementata tramite gli iteratori, ma questo è anche possibile con **memento differenti che rappresentano lo stato di differenti iterazioni**. L'incapsulamneto della collezione è sempre preservato, in quanto sapere i link al prossimo elemento è accessibile solo al `Memento`.

Il `Memento` può anche essere implementato con il **prototype pattern**, per il fatto che il `Memento` può essere istanziato come il clone di se stesso.

Flyweight

Un `Flyweight` è un oggetto leggero che si può avere in un numero di istanze elevatissimo. Abbiamo già visto nel prototype pattern l'esempio di programma per la creazione di note in uno spartito. Le istanze di note erano 2 (nel nostro esempio), ma rappresentare molte note tramite clonazione può essere uno spreco di memoria.

Le note differiscono solamente per la posizione nello spartito. Altre informazioni **che non riguardano una nota precisa** potrebbero essere memorizzate in un'altra struttura dati. A questo punto mi basta tenere solo una piccola pool di istanze di queste oggetti e restituire queste istanze quando servono. In questo modo:

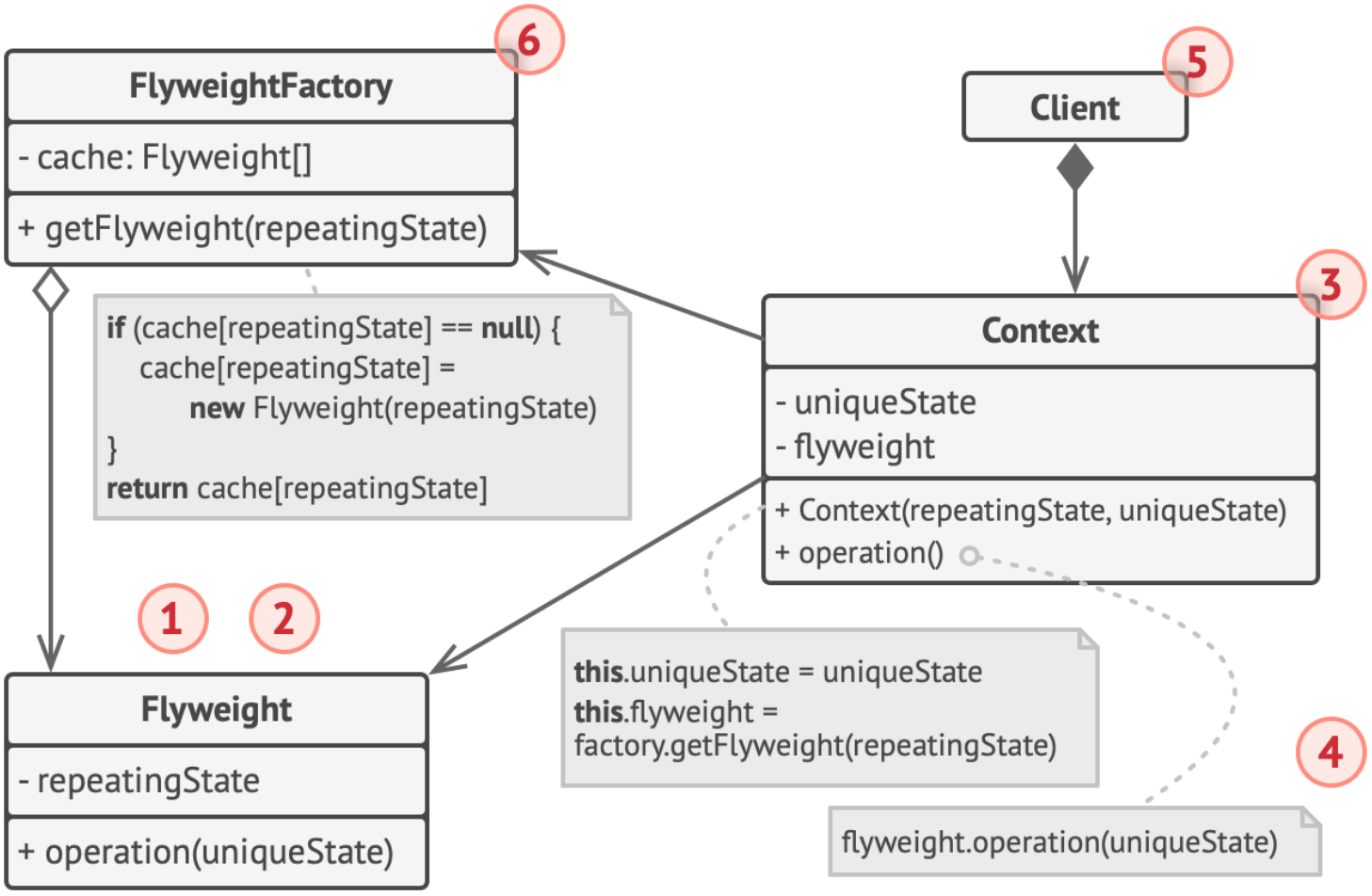
- informazioni che riguardano lo **stato intrinseco di un oggetto** sono memorizzate in una nuova istanza;
- informazioni **di carattere estrinseco** sono memorizzate in una pool predefinita, in modo tale da non ripetere le stesse informazioni in più istanze.

Ad esempio, nel caso dello spartito possiamo definire una nuova nota in virtù della sua posizione, ma salvare alcune informazioni predefinite della nota in un altro spazio, quali:

- rappresentazione del glifo;
- durata della nota.

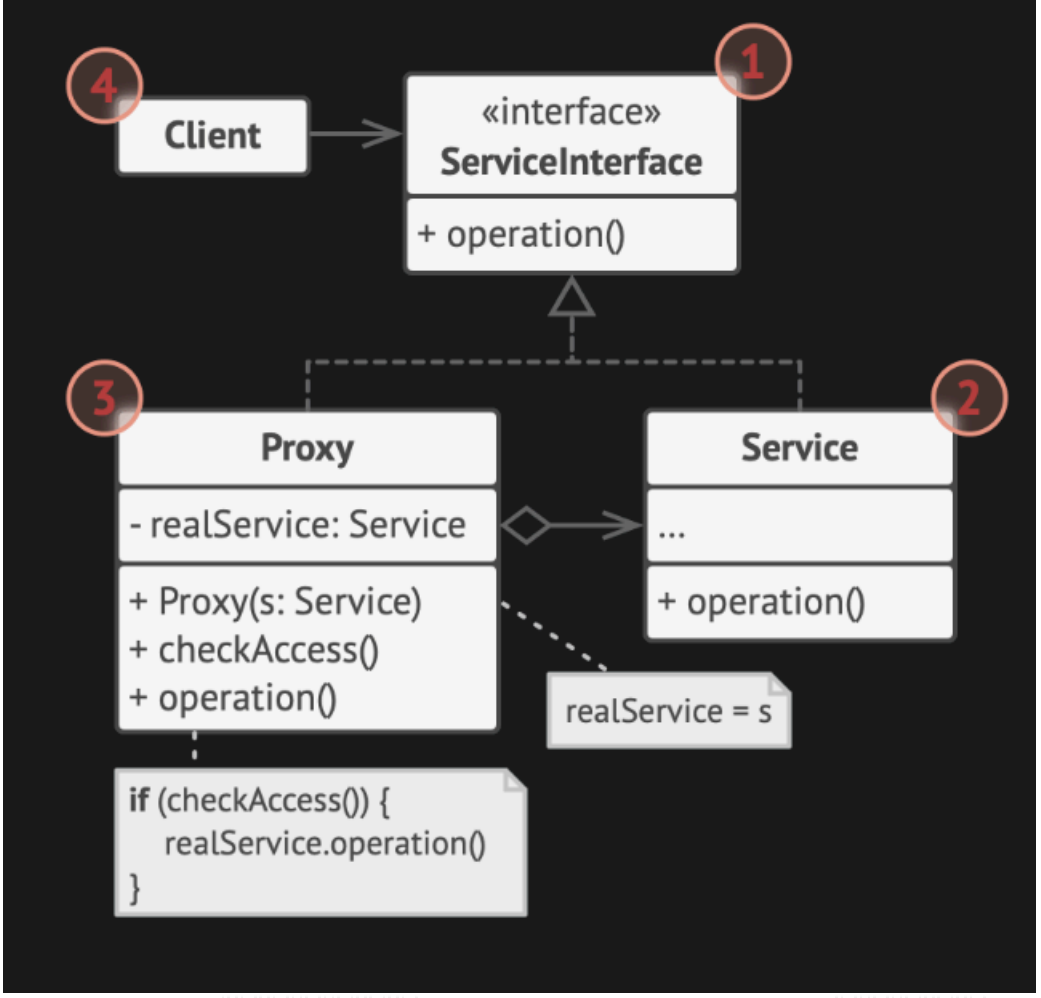
Il client fa riferimento ad un `Flyweight` astratto, selezionando quello corretto nel pool istanziato. Questo pool è gestito da un `FlyweightManager`. Inoltre, questo può essere utilizzato in coppia con il pattern memento.

Chiaramente il `Flyweight` dev'essere immutabile, perché questo oggetto viene concesso "in prestito". Inoltre, **non possiamo più fare confronti di istanze**, visto che viene sempre data la stessa istanza in prestito.



Proxy

Il proxy non è altro che un agente che opera come sostituto di un'altra entità.



Il client fa riferimento ad un'interfaccia `AbstractSubject`. L'interfaccia è implementata da un `RealSubject` e da un `ProxySubject`. Il `ProxySubject` incapsula al suo interno il `RealSubject` per redirezionare la richiesta. Il proxy è in grado di introdurre un pre-processamento e un post-processamento della richiesta senza che il client se ne renda conto.

Il proxy viene utilizzato per le Java RMI, ossia la **Remote Method Invocation**. Un client e un server immaginano di lavorare con oggetti locali, ma le richieste sono redirezionate ad oggetti su altre macchine. Gli oggetti locali fungono da proxy e redirezionano le chiamate.

Ci sono anche altri motivi per utilizzare il proxy:

- **virtual proxy**, per la lazy inizialization se la creazione di un oggetto è molto costosa;
- **protection proxy**, per fornire diversi livelli di protezione sull'accesso di un oggetto;
- **cache proxy**, per memorizzare i risultati di operazioni costose;
- **firewall proxy**, per proteggere i client locali da accessi esterni;
- **synchronization proxy**, per sincronizzare più accessi allo stesso subject;
- **counting proxy**, per contare i riferimenti agli oggetti. Questo è tipico di sistemi come i garbage collector, che contano i riferimenti agli oggetti: se non ne esistono, si elimina l'oggetto dalla memoria.

Builder

Il builder pattern consiste nell'incapsulamento di un oggetto il modo in cui si costruisce una struttura complessa. Questo pattern lo abbiamo incontrato già negli stream di Java.

Vogliamo far sì che la struttura creata possa prevedere uno **stesso strumento per la creazione** di questa, ma permettere l'ottenimento di **diverse rappresentazioni**. Un builder quindi si occupa dell'assemblamento della struttura, mentre un client invia solamente la richiesta di costruzione. L'oggetto composto viene creato a run-time, senza una definizione statica: gli oggetti vengono passati di volta in volta.

Il building presenta due fasi:

- acquisizione delle parti;
- fase di assemblaggio.

Il pattern permette:

- di separare l'implementazione della struttura dall'assemblaggio di questa;
- di isolare il codice di costruzione e rappresentazione.

