Elementi di Ingegneria del Software

Francesco Clemente

22 novembre 2018

Indice

Francesco Clemente

1 Prefazione

Questo manuale è nato come una raccolta di appunti presi durante il corso di *Programmazione ad oggetti* tenuto al *Politecnico di Torino*.

Ad un certo punto la sua stesura è caduta nel dimenticatoio, un po' per i vari impegni, un po' forse anche per pigrizia.

Ho ricevuto in ogni caso moltissimi feedback positivi per la parte completa, quella relativa all'ingegneria del software, per questo motivo ho deciso di ultimarlo escludendo la parte relativa alla programmazione con Java e lasciando solo ed esclusivamente la parte teorica riguardo l'ingegneria del software.

Nella speranza che possa essere d'aiuto a tanti colleghi.

Inizialmente questo voleva essere un progetto completamente *open source*, ma un collega ha pensato bene di considerare la possibilità di lucrare su questo lavoro, nonostante sia ampiamente spiegato nella licenza che questo non è possibile. Per questo mi vedo costretto a rilasciare soltanto il pdf per rendere la vita un po' più ardua a chi vuole guadagnare (seppur pochi spicioli) grazie a mie ore di lavoro.

2 Ingegneria del Software

Il termine *software engineering* venne usato per la prima volta in Germania nel 1968 durante una conferenza che riguardava la *crisi del software*. In particolare si discuteva del fatto che i grossi progetti software costavano sempre più di quanto si sarebbe immaginato prima della loro partenza, che venivano completati in ritardo e che non soddisfacevano i committenti.

Esistono molte definizioni di *software engineering*, una delle quali recita "**L'ingegneria del software riguarda la costruzione di un software fatto da molte persone ed in molte versioni**".

Quando si parla di software, non si parla di *"programmini"*, ma di un qualcosa di ben più completo e che coinvolge il lavoro di molte persone; consideriamo il fatto che avrà vita lunga e che quindi evolverà insieme con le esigenze e le piattaforme.

Il software si differenzia dal programma in quanto include **regole, documentazione e dati, vive a lungo, ha tante persone che se ne occupano e costa circa dieci volte di più**.

Il software non è prodotto, è sviluppato. È una disciplina che segue il software in tutta la sua vita.

3 Ciclo di vita del software

L'obbiettivo dell'ingegneria del software è di produrre il software con proprietà che siano predicibili e controllabili (costo, tempo di sviluppo, funzionalità, performance, robustezza).

La produzione prevede varie fasi:

- Raccolta dei requisiti
- Definizione di architettura e design
- Implementazione delle varie unità
- Integrazione delle varie unità

Esistono attività di **V&V**, ovvero *verification and validation* che consistono nel controllo dele varie fasi quando queste vengono completate di modo da evitare che eventuali problemi persistano nelle fasi successive.

La **verifica** è un controllo formale. La **convalida** è un controllo ad alto livello che consiste nel verificare se tutto è stato fatto nel modo corretto.

Esistono inoltre tutta una serie di attività di gestione che sono indispensabili:

- Gestione del progetto *(pianificazione e gestione del lavoro)*
- Gestione della configurazione *(documenti, codice, storia)*
- Validazione della qualità *(obiettivi, stato, risultati)*

Riassumendo, l'ingegneria del software attraversa le seguenti fasi:

- Sviluppo
- Operatività del software

Francesco Clemente

- Modifica del software
- Termine dell'utilizzo

È importante sottolineare che durante la fase di operatività avviene la fase di manutenzione e modifica del software, la quale è quella fase che occupa la maggior parte del tempo di vita del software.

La **manutenzione** prevede la correzione di difetti e l'introduzione di nuove funzionalità: ogni fase di manutenzione è considerabile come una piccola fase di sviluppo.

4 Modelli di processo

Con il termine *modelli di processo* si intende il modo, la tecnica, di approcciare all'ingegneria del software.

Esistono quattro approcci principali, di cui il primo non è definibile propriamente come modello di processo:

- **Programmazione cowboy** (o meglio "*build and fix*"): consiste nella semplice scrittura del codice. Non è applicabile a grandi progetti, non prevede stesura dei requisiti, non prevede design, non prevede fasi di validazione.
- **Basata sulla documentazione, semi-formale e basata su UML**: prevede la produzione di documentazione scritta in linguaggio semi-formale a cui segue trasformazione e controllo umano.
- **Basata sui modelli (o anche "*formale*"**: prevede la produzione della documentazione attraverso linguaggi formali a cui seguono trasformazione e controllo automatizzati.
- **Agile**: prevede attenzione orientata ad individui ed interazioni, piuttosto che ai processi ed agli strumenti; prevede inoltre una produzione precoce di software funzionante (approccio iterativo) piuttosto che di documentazione completa; si basa su una collaborazione con il cliente piuttosto che la definizione di un contratto rigido di modo da ottenere una buona risposta al cambiamento.

Esempi di modelli di processo sono:

- Basati sulla documentazione
  - Waterfall
    Prevede attività sequenziali: l'attività produce la documentazione; l'attività successiva inizia quando quella precedente è terminata altrimenti ferma il rilascio; la modifica dei documenti e/o del rilascio è un caso eccezionale
    Il problema principale di questo approccio è la mancanza di flessibilità: prevede una sequenza rigida; i requisiti rischiano di restare bloccati per lunghi periodi (non vengono effettuati cambiamenti per migliorarli, neppure per soddisfare i bisogni del clienti)

Francesco Clemente

  - V
    Simile al waterfall ma con enfasi per le operazioni di V&V; i test di accettazione sono scritti dopo o durante la stesura dei requisiti; i test di unità e integrazione sono scritti dopo o durante la fase di design
  - Incrementale, Evolutiva, Iterativa
    L'incrementale prevede una implementazione suddivisa in fasi incrementali dette *builds*; il ritardo della implementazione dipende da fattori esterni ma prevede un feedback immediato da parte dell'utente; le builds sono pianificate.
    L'evolutiva è simile all'incrementale ma i requisiti possono cambiare ad ogni iterazione: si ha un feedback immediato che porta a cambiamenti dei requisiti.
    L'iterativa prevede più iterazioni ciascuna delle quali può essere considerata come un piccolo progetto (sullo stile della waterfall)
  - Prototipazione
  - Spirale
  - Open source
  - Processo unificato

- Agile
  - Scrum
  - Extreme Programming
  - Crystal
- Metodi formali
  - Metodi formali
  - UML formale

Molto spesso è importante capire qual è la maturità del processo di produzione e come è possibile migliorare. Esistono vari modi per effettuare queste valutazioni. Un esempio è lo **Staged CMMI** il quale prevede cinque livelli di maturità per una azienda:

- Iniziale
  Processo impredicibile, poco controllato e poco reattivo
- Ripetibile
  Processo definito per i progetti e perlopiù reattivo
- Definito
  Processo definito a livello aziendale con organizzazione proattiva (controllo dei problemi prima ancora che insorgano)
- Organizzato
  Processo misurato e controllato; si interviene immediatamente appena insorge un problema
- Ottimizzazione
  Processi mirati al continuo miglioramento

Francesco Clemente

4.1 Paradigma Object Oriented e UML

Quando si parla di programmazione è opportuno sapere che esistono diversi paradigmi: procedurale, orientato agli oggetti, funzionale, logico.

Il paradigma ad oggetti prevede costrutti per la gestione dei suoi elementi fondamentali che sono **classi** ed **oggetti**.

La programmazione ad oggetti prevede l'esistenza di un modulo, quale l'oggetto, che contiene sia dati che funzioni che lavorano su questa entità; permette quindi la definizione di una interfaccia precisa e definisce nuovi tipi di relazioni basate sul passaggio dei messaggi.

Un modo di rappresentare la struttura di un programma orientato agli oggetti attraverso standard grafici è l'**UML** (*Unified Modelling Language*) il quale standardizza una notazione per specificare e visualizzare i costrutti di un sistema orientato agli oggetti.

Un diagramma delle classi, ad esempio, permette di progettare la struttura di un programma OO definendo concetti astratti, caratterizzarli definendo i dati associati, relazioni tra concetti ed i loro comportamenti.

In una applicazione OO abbiamo due livelli di astrazione:

- Astratta: concetti, entità, **classe**, categoria, tipo
  Ciò con cui si lavora durante lo sviluppo del programma
- Concreta: istanza, **oggetto**, occorrenza
  Ciò che viene "creato" durante l'esecuzione del programma

4.2 Classe

Una classe rappresenta un insieme di oggetti definendone proprietà comuni. Una istanza della classe è un oggetto avente come tipo ciò che la classe rappresenta; in fase di esecuzione, a ciascuna classe possono corrispondere più oggetti. In UML una classe viene rappresentata mediante rettangoli aventi da uno a tre scomparti in cui quello più in alto è adibito al nome della classe, quello centrale agli **attributi** e quello in basso ai **metodi**.

Gli attributi sono i dati che la classe contiene, i metodi sono le operazioni che la classe può effettuare.

È possibile definire in UML quelle che si chiamano **associazioni** tra le classi: vengono rappresentate con un collegamento ed eventualmente una etichetta e freccia per migliorarne la leggibilità; rappresentano un legame logico tra le due classi; prevede che si stabilisca la quantità delle istanze delle classi con cui queste partecipano alla relazione, attraverso la **molteplicità**; è possibile inoltre definire un **ruolo** per le entità che partecipano alla relazione in modo da definire il *ruolo*, appunto, con cui le entità partecipano alla relazione.

La molteplicità (o *cardinalità*) è rappresentata da due valori: minima e massima. Generalmente non è indicata attraverso il numero esatto ma:

- La molteplicità minima può essere 0 o 1
- La molteplicità massima può essere 1 o molti (indicato con *****)

Francesco Clemente

4.3 Oggetto

Un oggetto è la rappresentazione concreta di ciò che una classe definisce in modo astratto. È caratterizzato da identità, attributi (detti anche dati, proprietà o stato) ed operazioni che può effettuare. In UML un oggetto viene rappresentato con un rettangolo con un solo scomparto che contiene il nome e la classe di appartenenza.

La creazione di un oggetto è anche detta istanziazione.

Gli oggetti comunicano tra loro tramite "messaggi", richieste di servizio che vengono fatte ad un altro oggetto. In UML si rappresentano come connessioni tra i vari oggetti; è possibile aggiungere una freccia per definire l'orientazione della richiesta ed un eventuale etichetta che definisce il nome del metodo.

4.4 Incapsulamento

Risulta più semplice l'utilizzo di un oggetto nel momento questo viene incapsulato; ciò permette di ignorare i dettagli interni e sfruttarne le potenzialità attraverso la sua **interfaccia**.

Un oggetto incapsulato è **auto-contenuto**: una volta definita l'interfaccia, il programmatore può definire il contenuto della classe come meglio crede.

Un vantaggio dell'incapsulamento è la possibilità di una evoluzione in modo semplice: ciò che avviene all'interno è limitato all'interno, pertanto è possibile effettuare modifiche della implementazione di metodi lasciando invariata l'interfaccia.

4.5 Ereditarietà

L'ereditarietà è uno dei concetti fondamentali nel paradigma della programmazione ad oggetti: permette di definire classi come sotto-tipi di altre classi.

Una classe eredita tutti i metodi ed i campi della classe *padre*.

La classe *figlia* può effettuare un **override** della definizione dei metodi ereditati dalla classe padre, ovvero riscriverne l'implementazione.

Il codice della classe figlia (o *derivata*) consiste soltanto nei cambiamenti e le aggiunte che vengono fatte rispetto alla classe padre (o *classe base* o *superclasse*).

È possibile dire che una classe B **specializza** una classe A, intendendo che gli oggetti descritti da B hanno le stesse proprietà degli oggetti descritti da A; gli oggetti di B potrebbero avere caratteristiche addizionali rispetto a quelle di A; si dice anche che B è **un caso speciale** di A e che A è una **generalizzazione** di B (ed eventualmente anche di altre classi).

In UML si rappresenta mediante una freccia che si genera a partire dalla classe figlia e termina nella classe padre.

L'ereditarietà introduce una serie di vantaggi tra cui:

- Il riutilizzo di una classe già esistente introducendo solo una piccola quantità di modifiche
- La correzione di bug in classi padre, estende queste alle classi figlie
- La aggiunta di funzionalità alla classe base viene estesa alle classi figlie

Francesco Clemente

Esiste un sistema di misura che si chiama **DIT** (*Depth of Inheritance Tree*) la quale misura l'altezza dell'albero di ereditarietà; più il DIT cresce, più diventa difficile capire come è strutturata una classe figlia.

5 Gestione delle configurazioni

La gestione delle configurazioni (o **Configuration Management**) è la disciplina che si occupa di:

- Identificare e documentare le caratteristiche fisiche e funzionali di un oggetti di configurazione
- Controllare cambiamenti di queste caratteristiche
- Memorizzare e riportare il processo di cambiamento
- Verificare l'integrità con i requisiti

La gestione delle configurazioni inoltre gestisce le seguenti problematiche:

- Storia dei files (**versioning**)
- L'accesso ed i permessi ai files (**change control**)
- Definire il giusto insieme di documenti necessari ad uno specifico scopo (**configuration**)
- Come ottenere il sistema finale (**build management**)

Gli obiettivi che si pone consistono nell'identificare e gestire parti di software, controllarne l'accesso e permettere il rebuild di versioni precedenti del software.

5.1 Versioning

Si occupa della gestione dei **Configuration Item**: aggregati di lavoro prodotto i quali vengono trattati come una singola entità nel processo di configuration management; generalmente definito dal suo nome; tutte le sue versioni sono etichettate e conservate; l'utente può cambiare il numero di versione con specifiche operazioni (**commit**); è possibile recuperarne una qualsiasi versione precedente.

Una **versione** è una istanza di un configuration item la quale è identificata in modo non ambiguo (attraverso numeri di versione).

5.2 Configuration

Una configurazione è un insieme di configuration item, ognuno dei quali in una versione specifica.

Alcune configurazioni sono più importanti di altre e vengono chiamate **baseline**. Una baseline è una configurazione stabile che può essere di due tipi:

- Development: orientata all'uso interno
- Product: orientata al rilascio

Ad una baseline vengono assegnati dei numeri di versione aventi la sintassi *<Major>.<Minor>.<Patch>* dove:

- Major: è il numero che varia quando vengono effettuati cambiamenti estremamente significativi
- Minor: è il numero che varia quando vengono aggiunte funzionalità che siano anche retrocompatibili
- Patch: è il numero che varia quando vengono effettuati bug-fix retrocompatibili

Una **repository** è una collezione di tutte le configurazioni che appartengono ad un determinato sistema; con lo stesso termine viene anche definito il "posto" in cui questa collezione viene memorizzata.

5.3 Esempio

Un team di sviluppatori sta lavorando alla produzione di un software; molti sviluppatori hanno necessità di accedere a differenti parti del software.

Potremmo immaginare l'utilizzo di una cartella condivisa dove tutti possono leggere e scrivere i files; se si effettuano modifiche in parallelo c'è il rischio che le modifiche fatte contemporaneamente da altri vengano perdute; per evitare che ciò avvenga bisogna sempre recuperare l'ultima versione presente nella cartella e confrontarla con quella nuova che si ha prodotto (si effettua un **check-out** e poi un **check-in**).

Questo processo è gestibile in due modi:

- Lock-modify-unlock: anche detta serializzazione; soltanto una persona per volta può effettuare modificare un determinato file
- Copy-modify-merge: vengono effettuate più modifiche in parallelo ed in seguito si uniscono (**merge**)

5.4 Branches

Un branch è una linea di sviluppo che esiste indipendentemente dalle altre ma condividendone una storia comune.

Un branch prende vita come una copia di una configurazione ed avanza da quel punto in maniera indipendente.

5.5 Tools

Esistono vari tools che permettono di effettuare operazioni di **Change Control, Versioning e Configuration**; tra i più importanti troviamo:

- RCS
- CVS
- SCCS
- Subversion
- BitKeeper
- Git

Francesco Clemente

5.6 Subversion

Subversion è un sistema di controllo versione open-source: gestisce qualsiasi collezione di file e cartelle nel tempo su un repository centrale; memorizza ogni singolo cambiamento apportato ai files; permette di accedere ai suoi repository attraverso networks; lavora mediante commit atomici; permette l'utilizzo di branch e tag; utilizzabile mediante altre applicazioni.

Il repository è il centro di memorizzazioni di tutti i dati i quali vengono memorizzati sotto forma di file system; un qualsiasi numero di client possono connettersi al repository e leggere (**update**) e scrivere (**commit**).

La Working Copy è l'albero di directory che risiede sul sistema locale il quale contiene una copia dei files presenti sul repository ottenuti tramite la procedura di **checkout**; le operazioni di update e commit devono essere effettuate in modo esplicito dall'utente.

Ogni volta che il repository accetta un commit, crea un nuovo stato della gerarchia dei file che prende il nome di revisione; ad ogni revisione è assegnato un numero naturale progressivo unico.

Con **mixed revision** si intende la situazione in cui alcuni file sono ad un numero di revisione differente rispetto agli altri.

Nel momento in cui si effettua un commit di un file che è stato aggiornato da qualcun altro dopo che si è effettuato l'ultimo update, si genera quello che è chiamato **conflitto**; un conflitto deve essere risolto "fondendo" il file più recente presente nel repository e quello che si stava cercando di caricare.

In subversion non esiste un vero e proprio concetto di branch; una copia della cartella diventa un branch per il semplice fatto che è il modo in cui la interpretiamo. Per convenzione si creano all'interno di un repository, due sotto-cartelle che vengono chiamate *Trunk* (per il branch principale) e *Branches* (che contiene tutti i branch: una sotto-cartella per ognuno).

Nel momento in cui si ha completato il lavoro in un branch, questo deve essere integrato nel trunk; questa operazione prende il nome di **merge**: i due rami vengono confrontati e le differenze vengono presentate al programmatore; nel caso in cui ci siano conflitti, questi devono essere gestiti dal programmatore.

5.7 Git

Git è il sistema di versioning più utilizzato al mondo. Per una descrizione ben dettagliata di questo sistema, si rimanda al seguente LINK.

5.8 Build Management

La gestione delle Build consiste nelle fasi che portano al rilascio di un prodotto software finito.

Si articola in diverse fasi:

- Preparazione dell'ambiente
- Recupero dei componenti di terze parti
- Recupero del codice sorgente
- Compilazione

- Creazione dei pacchetti
- Testing
- Rilascio (o *deploy*)

Queste operazioni vengono generalmente fatte con l'ausilio di tool tra i quali è i più popolari sono:

- Make
- Ant
- Maven
- Gradle

Un approccio al Build Management è quello ad **Integrazione Continua** il quale consiste in:

- Mantenere una singola repository con i sorgenti
- Automatizzare i build
- Effettuare test sulle build
- Ogni build di commit viene posizionata su una macchina che si occupa dell'integrazione
- Rilascio automatizzato

Per rendere possibile l'integrazione continua è opportuno effettuare commit frequenti che compilino correttamente e che siano testati.

Tra i tool più diffusi per l'integrazione continua troviamo: Travis CI, Jenkins, Cruise Control

6 Testing

Esistono due differenti tecniche per approcciare al V&V:

- Statico
  - Fasi
    - Analisi statica della compilazione
    - Analisi del flusso di controllo
    - Analisi del flusso dati
    - Esecuzione simbolica
    - Ispezioni
  - Spesso si sfruttano processi di analisi del codice automatici: si effettuano senza eseguire il programma (durante la compilazione) e si lavora a livello di sorgente; con **code smell** si intende l'analisi del codice finalizzata a trovare strutture che violano i principi fondamentali di progettazione e ne alterano la qualità progettuale

Francesco Clemente

- Dinamico
  - Consta nel processo di utilizzare un sistema o un componente sotto specifiche condizione osservando e registrando i risultati in modo da intercettare le differenze tra il comportamento manifestato e quello previsto (**failures**).
  - È diverso dal debug che si occupa della ricerca della posizione delle cause dei fault e della loro rimozione
  - Con **test case** si intende uno stimolo dato ad un sistema eseguibile che consiste in un nome, un input ed un output atteso; con **test suite** si intende un insieme di test case correlati tra loro
  - Per produrre una buona suide di test è opportuno:
    * Avere buone probabilità di trovare un difetto
    * Non fare cose non necessarie
    * Non crearlo troppo semplice o troppo complesso
    * Che non sia ridondante
    * Renda gli errori facilmente identificabili
    * Che abbia test mutuamente indipendenti
    * Che analizzi tutte le parti del sistema

Con il termine **debito tecnico** si definisce il lavoro aggiuntivo che si genera nel momento in cui il codice viene implementato frettolosamente invece che applicando la migliore soluzione possibile.

Con il termine **test case log** si intende il registro di tutto ciò che è stato fatto durante l'esecuzione del test; contiene timestamp, esito e risultati. Il caso da testare viene fatto processare dal sistema e da un sistema di riferimento che prende il nome di **oracolo**; una volta processati questi vengono confrontati.

In una condizione ideale si avrebbe a disposizione un oracolo automatico ed un comparatore automatico; un oracolo umano potrebbe sbagliare.

Un oracolo si basa completamente sulle specifiche del programma, pertanto, nel caso in cui queste siano sbagliate, porterebbe ad errori.

Un oracolo automatico potrebbe essere: basato su specifiche formali, essere lo stesso software ma sviluppato da altre parti oppure una versione precedente dello stesso software (tecnica chiamata **regressione**). È quindi chiaro che dei test esaustivi non siano fattibili; l'obiettivo di un test è trovare difetti, non mostrare che il sistema ne sia privo.

È buona norma ricordare sempre che il software non deve invecchiare (se qualcosa funziona deve funzionare per sempre) e che è non lineare e non continuo (se con alcuni dati funziona, non è detto che funzioni con altri).

I test vengono classificati per:

- Fase/Granularità
  - Unità: test di moduli individualmente
  - Integrazione: test di moduli che lavorano insieme
  - Sistema: test del sistema nel suo complesso; mira a verificare la corrispondenza del sistema ai requisiti; considera i **profili di utilizzo** ovvero i modi più comuni di utilizzo del sistema; testa in condizioni più lontane possibili dalle conzioni di lavoro

  - Accettazione: test del sistema dal punto di vista del cliente
  - Regressione: test definiti precedentemente ed eseguiti nuovamente dopo un cambiamento (servono a verificare che il cambiamento non abbia introdotto difetti)

- Approccio
  - Black box (funzionale): serve a validare la descrizione di unità o metodi; fornisce input casuali; testa condizioni limite
  - White box (strutturale): serve a valutare la struttura del programma
  - Basato sul rischio (sicurezza)

Con il termine **driver** si intende una unità (funzione o classe) sviluppata per pilotarne un altra. Con il termine **stub** si intende una unità sviluppata per sostituirne un'altra.

7 Design Patterns

Per design pattern si intende una *soluzione* riusabile ad un *problema* noto in un *contesto* ben definito.

Esistono varie tipologie di pattern:

- Architectural patterns: si riferiscono a strutture attinenti al sistema nel suo complesso; esprimono una organizzazione strutturale fondamentale per un sistema software; mettono a disposizione una serie di componenti predefiniti con le relative responsabilità; definiscono regole e linee guida per organizzare le relazioni tra i componenti
- Design patterns: si riferiscono a meccanismi di livello più alto rispetto a quelli architetturali; definiscono schemi per rifinire componenti o le loro relazioni; descrivono strutture ricorrenti di componenti che comunicano tra loro
- Idioms: sono specifici per un determinato linguaggio; descrivono come implementare particolari aspetti di componenti o le relazioni tra questi

Esistono differenti descrizioni di un pattern:

- Coplien
  - Nome
  - Problema
  - Contesto
  - Forze: esigenze che agiscono sul problema
  - Soluzione
  - Soluzione alle forze
  - Design razionale: concetto di base da applicare nello specifico pattern
- Gang of Four

  - Nome
  - Obiettivo
  - Motivazione
  - Applicabilità
  - Struttura
  - Partecipanti
  - Collaborazioni
  - Conseguenze
  - Implementazione
  - Pattern correlai

Con il termine **pattern language** si intende che i pattern non esistono come entità assolute (spesso vengono applicati due o più pattern insieme; un pattern è utilizzato per implementare parte di un altro; un pattern può introdurre un problema risolto da un altro) e che quindi esistono **sistemi pattern**.

Secondo la Gang of Four esistono tre tipologie di pattern:

- Creazionale
  Definisce come creare oggetti secondo certe modalità per risolvere problemi standard
  Esempi:
  - Factory Method: definire un metodo a cui si passano parametri che si occupa di creare un oggetto con quei parametri e restituirlo
  - Abstract Factory: definire un modo per utilizzare una famiglia di classi correlate con diversi dettagli implementativi e predisporre una interfaccia che permetta di ignorare questi dettagli
  - Builder
  - Prototype
  - Singleton: rappresentare un oggetto unico in un'unica istanza
- Strutturale
  Descrive come strutturare le classi di modo da avere agevolazioni nella scrittura del codice
  Esempi:
  - Adapter: descrive come creare una classe che rende disponibili le specifiche richieste ma la cui interfaccia non è una delle specifiche richieste
  - Bridge
  - Composite
  - Decorator
  - Facade: predispone funzionalità che vengono implementate da un complicato gruppo di classi, senza esporre queste ultime
  - Flyweight
  - Proxy

- Comportamentale
  Definiscono strutture specifiche per determinare un certo tipo di comportamento; definiscono strutture di controllo ed assegnano responsabilità agli oggetti; non sono soltanto relativi agli oggetti ed alle classi, ma anche alla comunicazione di pattern
  Esempi:
  - Chain of responsibility
  - Command
  - Iterator
  - Mediator
  - Memento
  - Observer
  - Strategy: più classi e algoritmi hanno un nucleo stabile ma tante variazioni comportamentali
  - Visitor
  - Template method: un metodo ha un nucleo stabile e tante variazioni in punti stabiliti
  - Interpreter

  I patter comportamentali effettuano incapsulamenti delle variazioni, sfruttano oggetti come parametri per nascondere la complessità al client, definiscono politiche di circolazione dell'informazione, permettono il disaccoppiamento di *sender* e *receiver*.

Francesco Clemente