Analisi e Progettazione del Software

Riassunto tratto dagli appunti di Blue3141

Francesco Romeo

25 maggio 2023

INTRODUZIONE

Il software

Si tratta di un **programma per computer** che possiede la sua documentazione, si divide in due categorie:

- Generico, se sviluppato per diversi clienti.
- **Personalizzato**, se sviluppato per un cliente specifico.

E può essere prodotto:

- **Partendo da zero**, sviluppando quindi un nuovo programma
- **Sfruttando del software come base** per la creazione del nuovo programma.
- **Personalizzando** dei programmi pre-esistenti.

Caratteristiche

Un software deve possedere le seguenti caratteristiche:

- Deve essere **mantenibile**
- Deve essere **affidabile**
- Deve essere **efficiente**
- Deve essere **accettabile**

Ingegneria del Software

Si tratta dell'ingegneria che si occupa dello sviluppo di software di buona qualità.

Processo Software

Si tratta dell'**approccio disciplinato alla produzione di un software**. L'obbiettivo è quello di definire chi (**ruoli**) fa cosa (**attività**), quando (**organizzazione temporale**) e come (**metodologie**), tale obbiettivo viene raggiunto attraverso le **attività fondamentali**.

Attività fondamentali di processo

- Requisiti
- Analisi
- Progettazione
- Implementazione
- Validazione
- Rilascio ed installazione
- Manutenzione ed evoluzione
- Gestione del progetto

Analisi e Progettazione

- **Analisi**: capire **quale problema** si deve risolvere **e** i suoi **requisiti**.
- **Progettazione**: trovare una **soluzione** al problema che soddisfi i requisiti.

Nell'ambito OO le due definizioni diventano ben più specifiche.

- **Analisi OO**: Definire le **classi di dominio** per descrivere i concetti o gli oggetti relativi al problema.
- **Progettazione OO**: definire le **classi software** che servono a descrivere le classi di dominio all'interno di un linguaggio OO.

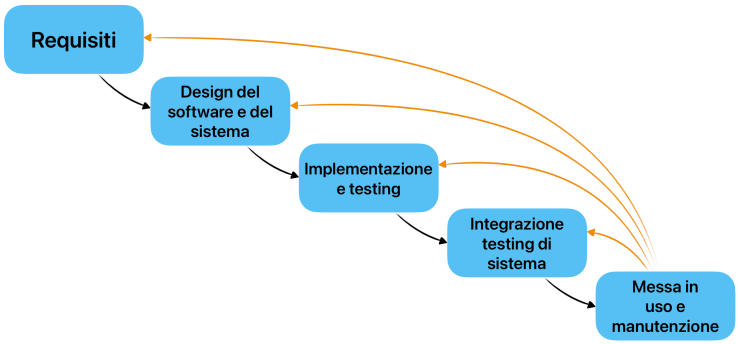
PROCESSI SOFTWARE

A Cascata

Questo particolare approccio prevede l'esecuzione a cascata, uno dopo l'altro, di alcune delle attività fondamentali:

- Def. dei **requisiti**
- **Design** del sistema e del software
- **Implementazione** e testing unitario
- **Integrazione** e testing di sistema
- **Messa in uso** e **manutenzione** che forza la ripresa da uno degli step precedenti.

Il **problema peggiore** di questo approccio è la **rigidità ai cambiamenti** per quanto riguarda i **requisiti** del cliente in quanto una loro modifica comporterebbe un quasi inevitabile fallimento.



Iterativo, Incrementale, Evolutivo

Si tratta di un **modello di organizzazione** del processo software in qui si svolgo tutte le attività di processo in una finestra di **tempo prefissata** per poi ricominciare da capo utilizzando i **risultati prodotti** durante l'iterazione precedente come **base di partenza** per quella successiva.
I vantaggi di questo approccio sono:

- **Flessibilità** rispetto alle modifiche
- **Riduzione dei rischi** maggiori in quanto subito gestiti
- **Progresso visibile** in quanto in un tempo prefissato si disporrà già di una demo.
- **Feedback precoce** in quanto il cliente potrà subito valutare la demo disponibile
- Miglior **gestione** della complessità

Iterazione

Ogni **iterazione**, come già detto, ha una **durata prefissata** e sarà impiegata solo per **svolgere parte del lavoro** complessivo, ci sarà quindi una fase di **pianificazione** dove verranno decisi gli obbiettivi da raggiungere.
Gli obbiettivi da raggiungere possono essere modificati a metà ciclo in base alla situazione che il team ha raggiunto.

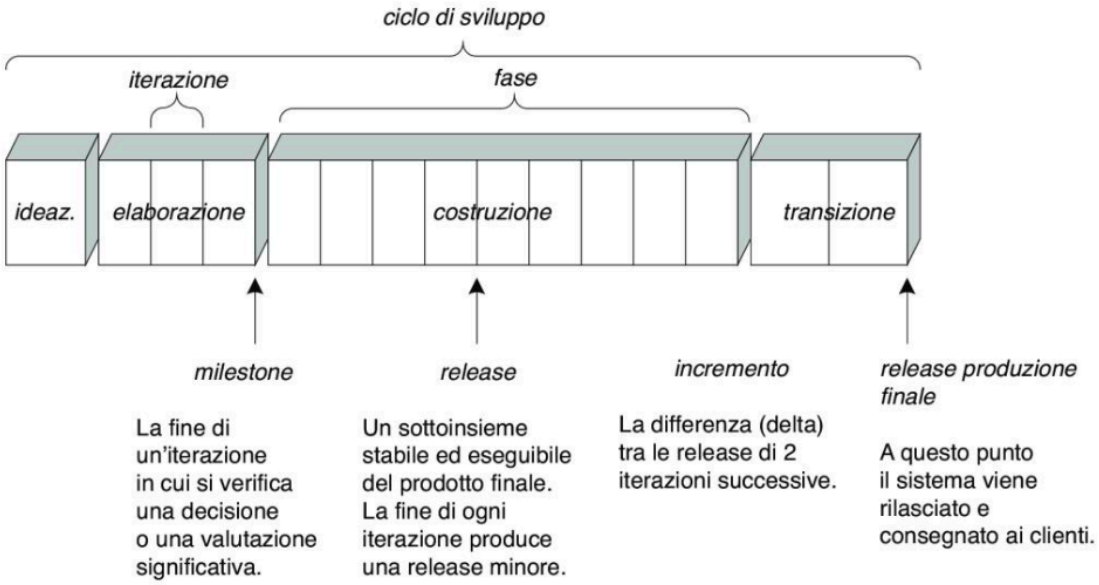
Unified Process (UP)

Si tratta di un processo iterativo, incrementale ed evolutivo impiegato per lo sviluppo di software OO, si tratta di un processo pilotato da: **casi d'uso (requisiti) e dai rischi**.

Fasi e Iterazioni

UP si divide in 4 **fasi**:

- **Ideazione**: Avvio del progetto con una visione approssimata per stimare tempi e costi.
- **Elaborazione**: identificazione della maggior parte dei requisiti.
- **Costruzione**: realizzazione delle capacità operative e preparazione al rilascio.
- **Transizione**: completamento del prodotto.



Ogni **fase del ciclo di sviluppo**, tranne *ideazione*, è **suddivisa in diverse iterazioni** all'interno dei quali si completa un mini-progetto che include:

- **Pianificazione**
- **Analisi e progettazione**
- **Costruzione**
- **Integrazione e test**
- **Rilascio, insieme di manufatti (elaborati) utili per il lavoro futuro**

Nota bene: **fase** → ideazione, elaborazione, costruzione o transizione, con *n* iterazioni.
disciplina → cosa si fa durante un iterazione.

Cattive pratiche di UP

- Tentare di **definire tutti i requisiti all'inizio.**
- **Non si testa il nucleo** dell'architettura.
- Si pensa che gli **UML** siano **dettagliati a livello di codice.**
- Si pensa che **UP** consista nel **produrre più elaborati possibile.**
- Si pensa che **ogni fase sia composta da una solo attività.**
- Si cerca di **concludere il progetto prima di implementarlo.**

Agile

Questo approccio allo sviluppo si basa sulla possibilità di **offrire una risposta rapida e flessibile ai cambiamenti**, ogni processo iterativo può adottare questa filosofia.
I principi cardine di **agile** sono:

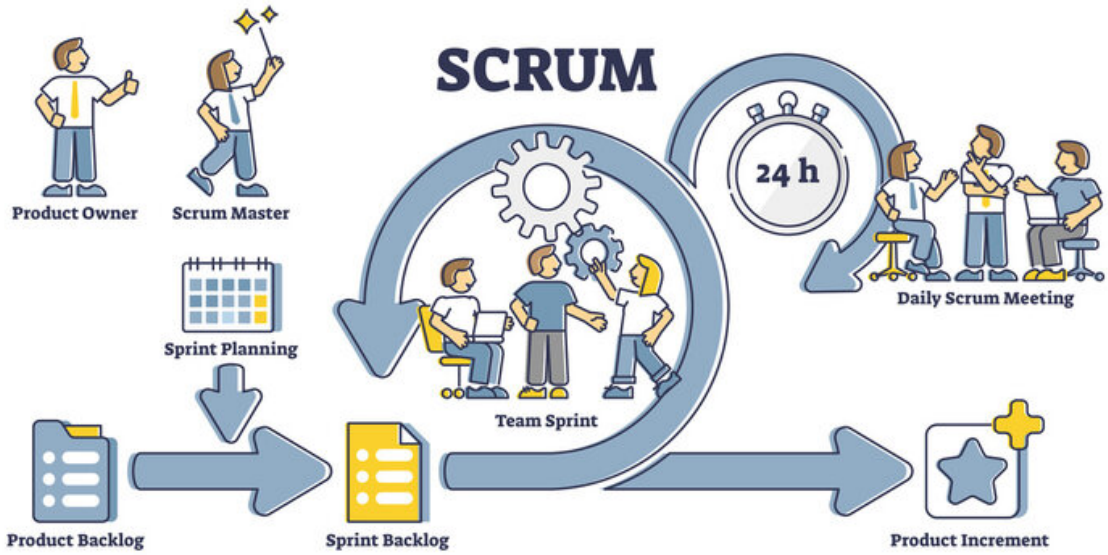
- Sviluppo e pianificazione **iterativa.**
- Consegne **incrementali**.
- Valori agili: **semplicità**, leggerezza, valore delle persone.
- Pratiche agili: **UML** solo delle parti complesse, **Test driven Dev.**, **refactoring**

UP + Agile

In poche parole: tutto UP + qualche elaborato, requisiti e progettazione si completano durante l'implementazione e anche grazie ai feedback e gli UML si fanno solo quando servono davvero.

Scrum

Si tratta di un **metodo agile** che ha come **obbiettivo** quello di creare il **software** con il più alto valore aggiunto **nel minor tempo possibile**.
Per fare ciò scrum prevede una **serie di meeting giornalieri** del team, chiamati **scrum**, dove verranno **decise le cose da portare a termine** presenti nella **backlog**, una *"lista di cose da fare"*, le **iterazioni** in scrum sono **dette sprint** e hanno una durata di 2/4 settimane, nella **fase finale** del progetto ossia quando le cose da fare sono fattibili in un **singolo sprint** esso prende il nome di **velocity**.



UP Fase di Ideazione

Lo scopo di questa fase è quello di decidere si se tratta di un progetto fattibile e di avere una visione d'insieme comune tra i membri del team.

Ideazione: portata del prodotto, visione e studio economico	
N. Iterazioni 1	**Durata realistica** Non più di qualche settimana.
Cosa si dovrebbe fare	Avere una **visione di insieme del progetto**, svolgere uno studio economico e sui tempi di sviluppo. Svolgere **l'analisi di una minima parte dei casi d'uso** e dei **requisiti non funzionali più critici**.
Documenti prodotti o modificati	• Visione e studio economico: obbiettivi e vincoli del progetto • **Modello dei casi d'uso**: i requisiti funzionali • **Specifiche supplementari**: requisiti non funzionali ad alto impatto • **Glossario**: dizionario di dati e parole chiavi del dominio • **Lista di rischi e piano di gestione** • Prototipi e proof of concept: utili per chiarire la visione • Piano di iterazione: piano per la prima iterazione del progetto • Piano delle fasi e piano di sviluppo: ipotesi sulla fase di sviluppo • Scenario di sviluppo: descrizione della personalizzazione dei passi e degli elaborati di UP per il progetto.
Errori comuni	• Dura più di qualche settimana • Definizione di troppi requisiti • Si crede che piani e stime affidabili • Definizione dell'architettura di sistema • Influenzata dal pensiero a cascata • Mancanza di visione o studio economico • I nomi di molti attori o casi d'uso non sono stati identificati • I casi d'uso non sono dettagliati

Requisiti funzionali e non

Ogni sistema software deve **risolvere uno o più problemi** e per farlo deve possedere determinate **funzionalità** e **caratteristiche** di qualità.
Essi, spesso, cambiano dopo l'ideazione.

- **Requisiti funzionali**: funzioni o servizi che il sistema deve offrire → Casi d'uso.
- **Requisiti non funzionali**: Sono vincoli di sistema a livello di affidabilità o rapidità o vincoli di standard → Specifiche supplementari.

I requisiti non funzionali si dividono in due categorie distinte:

- Gli **obbiettivi**, ossia le intenzioni generali dell'utente, che sono difficili da esprimere oggettivamente *(es facilità d'uso)*
- I **requisiti non funzionali verificabili**, ossia tutti gli altri.

I Casi d'Uso

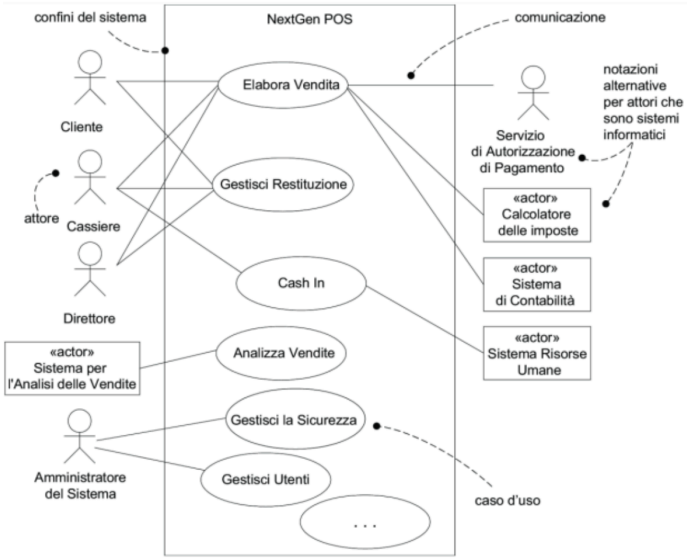
I **Casi d'Uso** sono *storie scritte* che descrivono l'i**nterazione tra un utente ed il sistema** che svolge un compito. Essi sono utili per **determinare** quali sono i **requisiti funzionali** necessari per svolgere una determinata attività e sono **facilmente comprensibili dal cliente**, ad ogni Caso d'Uso viene definito un nuovo contratto relativo al comportamento del sistema.

UP incoraggia la stesura dei Casi d'Uso ma nelle fasi iniziali del progetto solo di una piccola percentuale di essi, in particolare di quelli più influenti.
UP prevede un elaborato detto testo dei Casi d'Uso che al suo interno **contiene** tutti i **Casi d'Uso** e i diagrammi **UML** degli stessi oltre ai **Diagrammi di Sequenza di Sistema** e i **Contratti**.

Diagramma dei Casi d'Uso

Si tratta di un **diagramma di contesto** utile per avere una visione generale degli attori e delle relazioni che hanno con i vari Casi d'Uso.
Le relazioni possono essere di 3 tipi:

- ***Inclusione**: relazione tra Casi d'Uso in cui uno è sempre eseguito all'interno di un altro.*
- ***Estensione**: relazione tra Casi d'Uso in cui uno è eseguito all'interno di un altro al verificarsi di certi eventi.*
- ***Generalizzazione**: un Caso d'Uso B generalizzato da A eredita tutto quello contenuto in A e lo può modificare.*



UP Fase di Elaborazione

Lo **scopo** di questa fase è **produrre un eseguibile di qualità** che compone il nucleo dell'architettura a seguito di *n* iterazioni che prevedono diverse fasi.

Elaborazione: Costruire il modello dell'architettura, risolvere gli elementi ad alto rischio e definire la maggior parte dei requisiti e il piano di lavoro.	
N. Iterazioni 1 (2...n per l'intera fase)	**Durata realistica** 2-6 settimane (alcuni mesi per l'intera fase)
Cosa si dovrebbe fare	Si dovrebbe testare il nucleo dell'architettura e **definire la maggior parte dei requisiti**.
Documenti prodotti o modificati	• **Modello di dominio**: diagramma dei concetti del dominio. • **Modello di progetto**: diagrammi della progettazione logica, Classi Software, di Interazione etc. • Documento dell'Architettura: racchiude tutte le decisioni importanti per il progetto. • Modello dei dati: schema delle BD necessarie. • Storyboard dei Casi d'Uso, Prototipi UI: prototipi della UI etc. • **Raffinazione dei Casi d'Uso, si iniziano gli SSD e i Contratti e i si aggiorna il glossario**
Errori comuni	• Durata superiore ad "alcuni mesi" • Composta da una solo iterazione • Le fondamenta dell'architettura e gli elementi rischiosi non sono affrontati • Non produce un architettura eseguibile • Prova a fare una progettazione completa prima dell'implementazione • Si producono prototipi per mostrare idee e non la baseline del progetto

Modellazione del Business

Analisi a oggetti

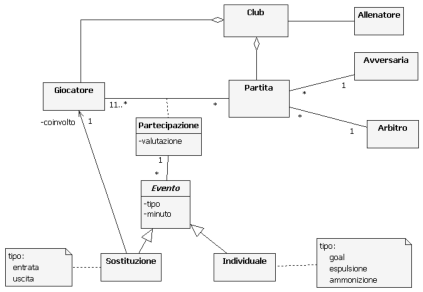
L'analisi serve per **analizzare il problema** che il sistema deve risolvere e ha come obbiettivo quello di evidenziare 3 **aspetti fondamentali**:

- Le **informazioni** da gestire → Modello di Dominio.
- Le **funzioni** → Diagramma di Sequenza di Sistema.
- Il **comportamento** → Contratti.

Modello di Dominio

Si tratta del **principale documento** prodotto nell' analisi, si tratta di una rappresentazione visiva dei **concetti del mondo reale** e dei loro legami, esso **modella il dominio** all'interno del quale opera il sistema.

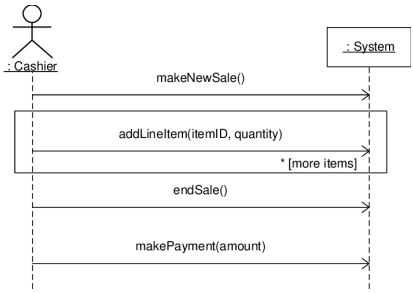
Ogni classe (box nel diagramma) è in relazione con altri e possiede degli attribuiti, esistono diversi tipi di relazioni e diverse visibilità per gli attribuiti, come in java.



Requisiti

Diagramma di Sequenza di Sistema SSD

Un SSD è un elaborato che mostra come **attori e sistema si comportano** in base a determinati input.
Il testo dei Casi d'Uso e gli eventi di sistema da essi sottintesi costituiscono l'input per la creazione degli SSD



I Contratti

I Contratti delle operazioni **descrivono i cambiamenti agli oggetti di dominio dopo l'esecuzione di un'operazione di sistema**.
I principali input per i contratti sono le operazioni di sistema identificate negli SSD e il Modello di Dominio.
UP non indica come obbligatori i Contratti quindi vanno stilati solo se sono necessari o utili.

Dai Requisiti alla Progettazione

Avendo svolto in modo corretto l'analisi si potrà porre maggior interesse alla progettazione.

Architettura software

Sono tutte le **decisioni importanti** su tutte le parti che concorrono allo sviluppo del progetto.

Architettura logica

Ha il compito di **dividere le cassi software** in strati/pakages/sottosistemi in base alla loro semantica.
Anche l'architettura software si può rappresentare tramite UML come parte del Modello di Progetto.
Solitamente in un applicazione software abbiamo questi strati:

- **UI**, interfaccia utente
- **Logica applicativa**, o staro del dominio ossia gli elementi rappresentano i concetti di dominio
- **Servizi Tecnici**

Progettazione

Progettare ad oggetti

Per approcciarsi alla progettazione gli sviluppatori possono decidere se: non produrre alcun "disegno" e passare direttamente alla codifica, sviluppare un diagramma UML e tradurlo in codice o impiegare un qualche sistema che passi in maniera automatica da "disegno" a codice.

Ci sono due tipi di modelli per gli oggetti:

- Statici → Diagrammi delle Classi, package e di deployment
- Dinamici → Diagrammi di Sequenza, Comunicazione, delle Macchine a Stati e di Attività con impiego di pattern.

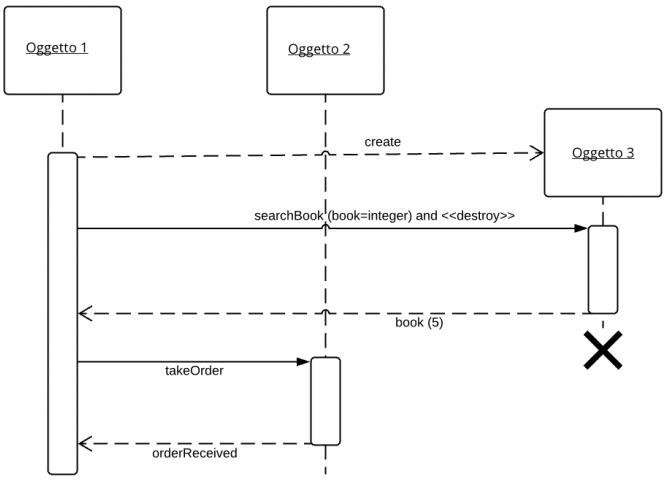
Diagrammi di Interazione

I Diagrammi di Interazione illustrano il modo in cui gli oggetti interagiscono tra di loro mediante lo scambio di messaggi.
I Diagrammi di Interazione sono di 4 tipi:

- **Diagrammi di Sequenza (SD)**
- **Diagrammi di Comunicazione (DC)**
- Diagrammi di Interazione generale
- Diagrammi di Temporizzazione

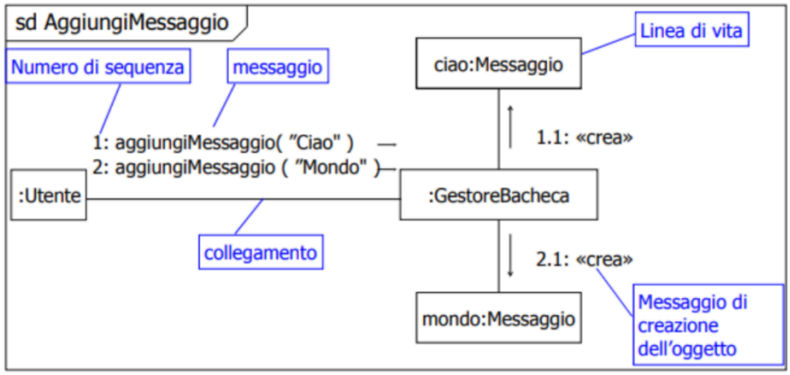
Diagrammi di Sequenza SD

Sono quasi come gli SSD con la differenza che non ci si trova più a livello software ma si analizzano iterazioni tra oggetti a livello di dominio.



Diagrammi di Comunicazione DC

Si tratta di diagrammi identici ai SD ma le linee di vita non sono rappresentate verticalmente ma dipendono dai messaggi che gli oggetti si scambiano.



Diagrammi delle Classi Software

Sono utilizzati per la modellazione delle classi, sono gli stessi usati per la Modellazione del Dominio.

Alcune definizioni

Un **oggetto** è un **istanza di una classe**, esso possiede una serie di **attributi** e di **operazioni** e sono definiti attraverso un **identificatore univoco**.

Interfacce

Un'interfaccia è un insieme di funzionalità pubbliche identificate da un nome; separa le specifiche di una funzionalità dall'implementazione della stessa.

Macchine a Stati

Si tratta di **diagrammi** utili a mostrare l'**evoluzione del ciclo di vita di un oggetto** a seguito di una serie di eventi, transizioni nel diagramma.
UP non li ritiene necessari quindi sono da scriversi se e solo se utili.

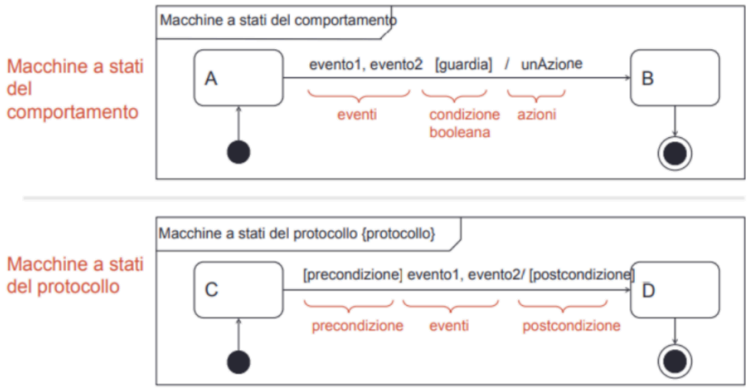


Diagramma delle attività

Si tratta di un diagramma utilizzato per mostrare le attività di un processo specifico. In UP non sono obbligatori.

Progettazione guidata dalla responsabilità (RDD)

Durante la progettazione ad oggetti è importante pensare in termini di **responsabilità**, **ruoli** e **collaborazioni**. Per **responsabilità** si intende un **astrazione** di ciò che un oggetto **fa** o **rappresenta**.
Nella programmazione OO un **progetto** è inteso come una **comunità di oggetti** con **responsabilità** che **collabora** per fornire una funzionalità.
Le responsabilità sono di due tipi:

- **Di fare** → *fare una determinata cosa*.
- **Di conoscere** → *conoscere determinate cose*.

Non confondere metodi e responsabilità: i metodi servono a soddisfare le responsabilità.

Una RDD viene svolta in modo iterativo seguendo questi passi:

- **Identificare**, una alla volta, le responsabilità
- Decidere a quale oggetto **assegnarla.**
- Capire come l'oggetto la può **soddisfare**.

Per decidere **a che oggetto assegnare** una certa responsabilità ci si può rifare al **pattern GRASP.**

Pattern GRASP

I Pattern GRASP (General resp. Assignment software patterns) descrivono dei principi base per l'**assegnazione delle responsabilità**. In generale i pattern solo delle soluzioni a problemi noti che si sono rivelate di successo in diverse situazioni.
Elenco dei Pattern GRASP spiegati nel file *"pattern grasp"*

Design Pattern (GoF)

I design pattern offrono come I GRASP delle soluzioni a problemi ricorrenti ma mostrano anche lo schema delle classi per implementare le soluzioni.
Spiegati nel file *"GoF"*

Progettare la visibilità

Per visibilità si intende la capacità di un oggetto di **vedere** o di **avere un riferimento** ad un altro oggetto con il quale vuole comunicare.

La visibilità da parte di A su B può essere ottenuta in 4 modi:

- Per **attributo**, B attributo di A
- Per **parametro**, B parametro di un metodo A
- **Locale**, B è u oggetto locale di un metodo di A
- **Globale**, B visibile globalmente

Dalla Progettazione all'Implementazione

Tutti gli elaborati realizzati durante la fase di progettazione diventano gli input per avviare la fase di implementazione.
In un linguaggio OO è necessario produrre codice per:

- Definire classi e interfacce
- Dichiarare variabili di istanza
- Definire metodi e costruttori

Testing

Sviluppo guidato dai test

Questo approccio alla scrittura di codice, ben visto in UP, consiste nel realizzare in primis i test che il codice dovrà superare e e poi procedere con la scrittura effettiva del codice.

I vantaggi di questo approccio sono:

- Effettiva stesura dei test.
- Avendo già dei test da superare il programmatore la prende come una sfida ed è più motivato.
- Chiarezza dell'interfaccia.
- Verifica autonoma, ripetibile e dimostrabile.
- Fiducia nel cambiare le cose

Tipi di test

- Test di unità → su singole classi e metodi
- Test di integrazione → per testare la comunicazione tra varie parti
- Test di sistema → per testare il collegamento tra tutte le parti
- Test di accettazione → verificare il funzionamento per l'utente

Test di unità

I test di unità consistono in **4 parti**, la **preparazione**, ossia la creazione dell'oggetto da testare chiamato fixture, **l'esecuzione**, ossia l'esecuzione della fixture, la **verifica**, ossia la valutazione dei test ed in fine il **rilascio** ossia la "pulizia" delle risorse utilizzate nei test per evitare di influenzare i successivi.

I cicli per i test unitari

Il TDD si basa su brevi cicli di lavoro in cui si:

- **Scrive un test** unitario che fallisce per mostrare l'assenza di una funzionalità.
- **Scrivere il codice** appropriato per superare il test.
- **Migliorare** il codice **o** passare alla scrittura del **test successivo (refactoring)**.

Possiamo avere anche cicli più lunghi che prevedano la scrittura dei test di accettazione e della successiva scomposizione in *n* test unitari.

Refactoring

Si tratta di un **approccio disciplinato** alla riscrittura del codice attraverso il quale si **migliora** il codice senza modificarne il comportamento.
Gli obbiettivi del refactoring sono:

- Eliminare il codice duplicato
- Migliorare la chiarezza
- Abbreviare i metodi lunghi

Quando fare refactoring

Il refacotoring si deve attuare al verificarsi di almeno una di queste situazioni:

- Si soddisfa la **regola del tre**, si utilizza tre volte la stessa porzione di codice in punti diversi del programma
- Si aggiunge una **nuova funzionalità**.
- Si corregge un **bug**.
- Si sta **revisionando** il codice.

Processo di trasformazione

Per trasformare il codice si devono seguire questi passi:

- Assicurarsi che i test siano passati
- Trovare un code smell → cattive pratiche di programmazione
- Determinare in che modo migliorare il codice
- Effettuare i miglioramenti
- Eseguire i test
- Ripetere il ciclo fino a che non si rimuove il code smell individuato

Code Smell

Si possono dividere in diverse categorie:

- Bloaters → metodi, classi o in generale codice di dimensioni troppo grandi.
- OO Abuses → Implementazioni errate della programmazione OO
- Change Preventers → Parti di codice copiate che impediscono cambiamenti rapidi
- Dispensables → codice inutile
- Couplers → classi che delegano eccessivamente