



**UNIVERSITÀ DEGLI STUDI  
DI GENOVA**

UNIVERSITÀ DEGLI STUDI DI GENOVA

# **Fondamenti di Ingegneria del Software**

*Lorenzo Vaccarecci*

# Indice

<b>1</b>	<b>Modelli di processo di sviluppo software</b>	<b>6</b>
1.1	Introduzione . . . . .	6
1.1.1	Processo prescrittivo e adattivo . . . . .	6
1.2	Modelli di processo . . . . .	7
1.3	Code and Fix . . . . .	7
1.4	Modello a cascata . . . . .	7
1.4.1	Studio di fattibilità . . . . .	8
1.4.2	Varianti del modello a cascata . . . . .	8
1.5	Modelli evolutivi . . . . .	9
1.5.1	Modelli a Prototyping . . . . .	9
1.5.2	Modelli Iterativi-Incrementali . . . . .	9
1.6	Modello a spirale . . . . .	10
1.7	Unified Process . . . . .	11
1.7.1	Le iterazioni . . . . .	11
1.7.2	Le fasi . . . . .	11
1.8	Sviluppo basato sui componenti . . . . .	11
1.9	Metodi Plan-Driven e Agili . . . . .	12
1.9.1	Come scegliere? . . . . .	12
1.10	DevOps . . . . .	13
1.10.1	Continuous Integration . . . . .	13
<b>2</b>	<b>Ingegneria dei requisiti</b>	<b>14</b>
2.1	Introduzione . . . . .	14
2.2	Classificazione dei requisiti . . . . .	14
2.2.1	Esempio: Bancomat . . . . .	14
2.3	Requirements Engineering . . . . .	15
2.3.1	Scopo . . . . .	15
2.3.2	Processo iterativo . . . . .	15
2.4	Proprietà dei requisiti . . . . .	16
2.5	Template/Schema dei requisiti . . . . .	16
2.6	Analista software . . . . .	17
2.6.1	Consigli per un'intervista . . . . .	17
2.6.2	Importanza della comunicazione . . . . .	17
2.7	Consigli finali . . . . .	18
<b>3</b>	<b>Definizione dei requisiti basata su use case</b>	<b>19</b>
3.1	Cosa sono/servono . . . . .	19
3.2	Differenza tra requisito e use case . . . . .	19
3.3	Definizione dei requisiti basata su use case . . . . .	19
3.4	Scenario . . . . .	20

3.5	Use Case . . . . .	20
3.5.1	Descrivere uno use case . . . . .	20
3.5.2	Template . . . . .	20
3.6	Gerarchia attori . . . . .	21
3.7	Relazioni tra use case . . . . .	21
<b>4</b>	<b>Design architetturale</b>	<b>22</b>
4.1	Introduzione . . . . .	22
4.1.1	Livelli di ri-uso . . . . .	22
4.2	Design architetturale . . . . .	22
4.3	Componenti . . . . .	23
4.4	Diagramma a blocchi . . . . .	23
4.5	Architettura SW . . . . .	23
4.5.1	Proprietà del sistema . . . . .	23
4.5.2	Vantaggi . . . . .	23
4.5.3	Stili architetturali . . . . .	24
4.5.4	Layered . . . . .	24
4.5.5	Repository . . . . .	25
4.5.6	Client/server . . . . .	25
4.5.7	Pipe and Filter . . . . .	26
4.5.8	Architetture eterogenee . . . . .	26
4.5.9	Microservices . . . . .	26
<b>5</b>	<b>Design delle componenti</b>	<b>28</b>
5.1	Fasi . . . . .	28
5.2	Design by contract . . . . .	28
5.2.1	Elementi di un contratto . . . . .	28
5.2.2	Vantaggi . . . . .	29
5.3	Progettazione degli algoritmi . . . . .	29
5.3.1	Notazioni . . . . .	29
5.4	Principi di progettazione . . . . .	30
5.4.1	Principi . . . . .	30
5.4.2	Principio KISS . . . . .	32
5.4.3	Sto seguendo i principi? . . . . .	32
<b>6</b>	<b>UML - Introduzione</b>	<b>33</b>
6.1	Cos'è . . . . .	33
6.2	Prospettive . . . . .	33
6.3	Modello di dominio (del business) . . . . .	33
6.4	Un po' di storia . . . . .	34
6.5	Perchè . . . . .	34
6.6	Notazione e meta-modello . . . . .	34
6.6.1	Profilo . . . . .	35
6.7	Come usare UML . . . . .	35
<b>7</b>	<b>UML - Class Diagram</b>	<b>36</b>
7.1	Introduzione . . . . .	36
7.2	Prospettive . . . . .	36
7.3	Classe . . . . .	36
7.3.1	Rappresentazione . . . . .	37
7.3.2	Attributi . . . . .	37

7.3.3	Molteplicità . . . . .	37
7.3.4	Operazioni . . . . .	38
7.4	Datatype . . . . .	38
7.5	Associazioni . . . . .	38
7.5.1	Molteplicità . . . . .	38
7.5.2	Attributi . . . . .	38
7.5.3	Implementazione (Java) . . . . .	39
7.6	Note . . . . .	39
7.7	Aggregazione . . . . .	39
7.8	Composizione . . . . .	39
7.9	Generalizzazione ed ereditarietà . . . . .	39
<b>8</b>	<b>UML - Sequence Diagram</b>	<b>40</b>
8.1	Diagrammi di interazione . . . . .	40
8.2	Notazione . . . . .	40
8.3	Sintassi dei messaggi . . . . .	40
8.4	Frame . . . . .	40
8.4.1	Cicli e condizioni . . . . .	40
8.4.2	Ref . . . . .	41
8.4.3	Par . . . . .	41
8.5	Sequence Diagram di Sistema (SSD) . . . . .	41
8.6	Communication Diagram . . . . .	41
<b>9</b>	<b>UML - State Machine &amp; Activity Diagram</b>	<b>42</b>
9.1	State Machine . . . . .	42
9.1.1	Stato . . . . .	42
9.1.2	Rappresentazione grafica . . . . .	42
9.1.3	Altri tipi di eventi in UML . . . . .	43
9.1.4	Stati composti (superstati) . . . . .	43
9.2	Activity Diagram . . . . .	44
9.2.1	Attività . . . . .	44
9.2.2	Nodi inizio e fine . . . . .	44
9.2.3	Nodi decisione e fusione . . . . .	44
9.2.4	Nodi fork e join . . . . .	44
<b>10</b>	<b>UML - Component, Package &amp; Deployment Diagram</b>	<b>45</b>
10.1	Component Diagram . . . . .	45
10.1.1	Componente . . . . .	45
10.1.2	Interfacce . . . . .	45
10.1.3	Dipendenza . . . . .	46
10.2	Deployment Diagram . . . . .	46
10.2.1	Nodi e connessioni . . . . .	46
10.2.2	Artefatti . . . . .	46
10.2.3	Manifest . . . . .	46
10.3	Package Diagram . . . . .	47
10.3.1	Package . . . . .	47
10.3.2	Visibilità . . . . .	47
10.3.3	Dipendenze . . . . .	48
10.3.4	In pratica . . . . .	48

<b>11 Software Design Patterns</b>	<b>49</b>
11.1 Altre unità di riuso . . . . .	49
11.2 GRASP . . . . .	49
11.2.1 Pattern Controller . . . . .	49
11.3 Design pattern classici . . . . .	50
11.3.1 I pattern creazionali . . . . .	50
11.4 Adapter . . . . .	50
11.5 Façade . . . . .	51
11.6 Template Method . . . . .	51
11.7 Observer . . . . .	51
11.7.1 Interfacce . . . . .	52
11.7.2 Conseguenze . . . . .	52
11.8 Model View Controller (MVC) . . . . .	52
11.9 State Pattern . . . . .	52
<b>12 Code Refactoring</b>	<b>53</b>
12.1 Legacy System . . . . .	53
12.1.1 Convivere con un Legacy System . . . . .	53
12.2 Perché? . . . . .	54
12.3 Quando? . . . . .	54
12.3.1 Code Smell . . . . .	54
12.3.2 Clone Software . . . . .	54
12.4 Il ritmo . . . . .	55
12.4.1 Catalogo dei Refactorings . . . . .	55
12.5 Come? . . . . .	55
12.5.1 Tools . . . . .	55
<b>13 Software Testing</b>	<b>57</b>
13.1 Debugging . . . . .	57
13.2 Testing . . . . .	57
13.3 Testcase & Testsuite . . . . .	57
13.4 Tipologie di testing . . . . .	58
13.5 Testing manuale VS Testing automatizzato . . . . .	58
13.5.1 JUnit . . . . .	58
13.6 White-box Testing . . . . .	58
13.6.1 Code coverage . . . . .	58
13.6.2 Limiti . . . . .	60
13.7 Black-box Testing . . . . .	60
13.7.1 Equivalence Partitioning . . . . .	60
13.8 Boundary Value Analysis (BVA) . . . . .	60
<b>14 Metodi Agili: Extreme Programming</b>	<b>61</b>
14.1 Critiche ai metodi plan-driven . . . . .	61
14.2 Agile Manifesto . . . . .	61
14.3 Extreme Programming (XP) . . . . .	61
14.3.1 Cosa è . . . . .	61
14.3.2 Requisiti . . . . .	62
14.3.3 Come si produce il desing? . . . . .	62
14.3.4 Principi di codifica . . . . .	62
14.3.5 Pair Programming . . . . .	62
14.3.6 Unit Testing . . . . .	62

14.3.7 Refactoring . . . . .	62
14.3.8 Acceptance Testing . . . . .	63
14.4 Problemi dei metodi agili . . . . .	63
14.5 Metodi Plan-Driven VS Metodi Agili . . . . .	63

# Capitolo 1

## Modelli di processo di sviluppo software

### 1.1 Introduzione

<b>Processo:</b> insieme strutturato e organizzato di attività che si svolgono per ottenere un risultato.
---

Perchè modellare il processo? Per dare ordine, controllo e ripetibilità con l'intenzione di migliorare la produttività e la qualità del prodotto.

#### 1.1.1 Processo prescrittivo e adattivo

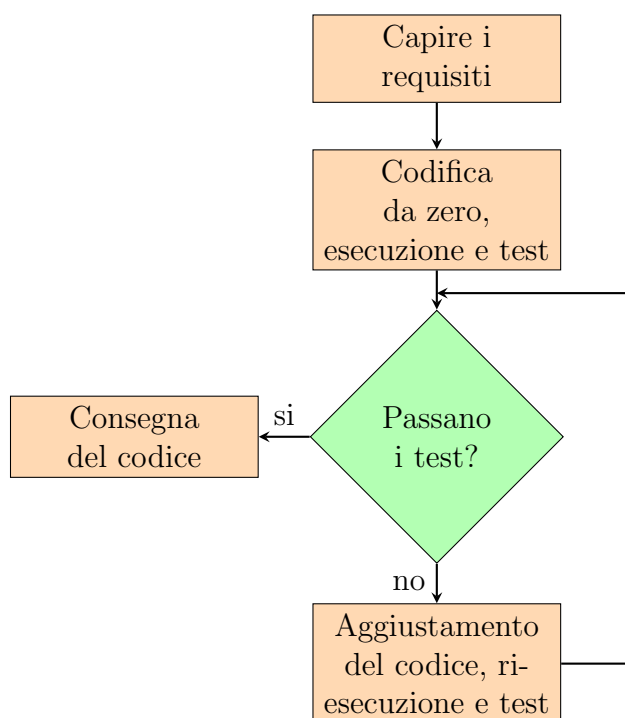
- **Processo prescrittivo:** un processo che segue un modello predefinito e rigido, con passaggi specifici e ben definiti.
- **Processo adattivo:** un processo che permette modifiche e adattamenti durante il suo svolgimento.

Perchè studiare i modelli di processo? Perchè uno dei compiti dei manager aziendali è quello di decidere il modello di processo da adottare considerando la tipologia del software da progettare e il personale disponibile.

## 1.2 Modelli di processo

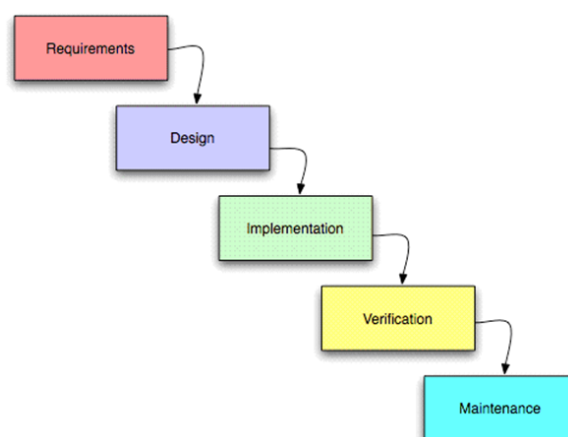
### 1.3 Code and Fix

- Si arriva al codice finale "per tentativi"
- Non adatto per progetti grandi con tanti sviluppatori
- Non è un modello di processo vero e proprio



### 1.4 Modello a cascata

- Storicamente il primo modello del processo di sviluppo software
- Ogni fase produce un prodotto che è l'input della fase successiva
- Con il modello waterfall abbiamo il passaggio dalla dimensione artigianale alla produzione industriale del software
- Molto rigido: non si può tornare indietro



Vantaggi	Svantaggi
Enfasi su aspetti come l'analisi dei requisiti e il progetto di sistema trascurati nell'approccio code & fix	Lineare, rigido, monolitico: no feedback tra fasi, no parallelismo, <b>unica data di consegna</b>
Postpone l'implementazione dopo avere capito i bisogni del cliente	La consegna avviene dopo anni, intanto i requisiti cambiano o si chiariscono: così viene consegnato software obsoleto
Introduce disciplina e pianificazione	Viene prodotta troppa documentazione poco chiara: l'utente spesso non conosce tutti i requisiti all'inizio dello sviluppo
E' applicabile se i requisiti sono chiari e stabili	Alcuni difetti superati da modello waterfall con feedback e iterazioni

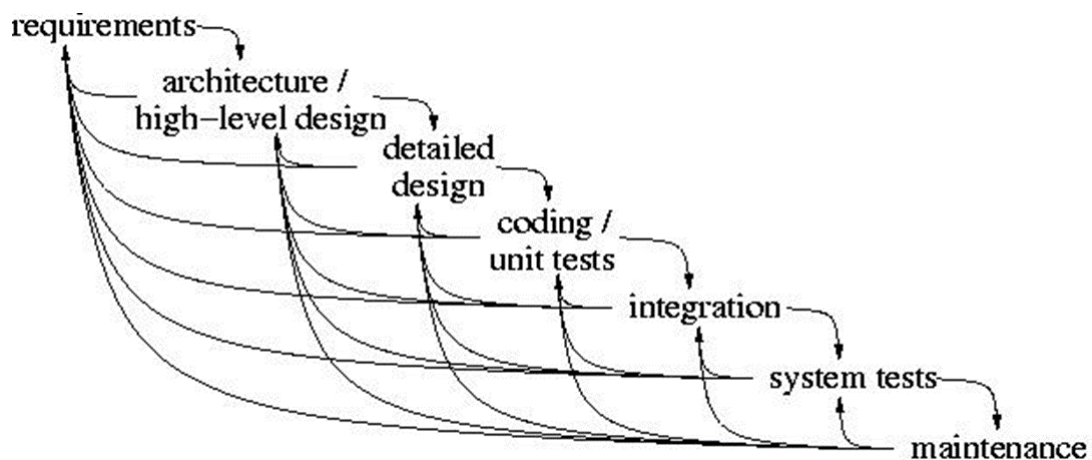


### 1.4.1 Studio di fattibilità

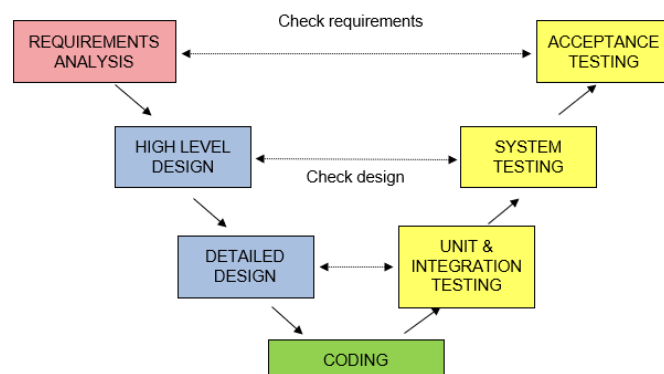
- Fase che precede lo sviluppo vero e proprio
- Viene analizzata la fattibilità e convenienza del progetto
- Stima dei costi
- Si valuta il Return Of Investment (ROI)

### 1.4.2 Varianti del modello a cascata

- Cascata con prototipazione: prima di iniziare lo sviluppo si costruisce un prototipo "usa e getta" con il solo scopo di fornire agli utenti una base concreta per meglio definire i requisiti.
- Cascata con feedback e iterazioni: posso tornare a una fase precedente.



- V-Model:
  - Enfasi sulle fasi di testing
  - Evidenzia come le attività di testing (parte destra della V) sono collegate a quelle di analisi e progettazione (parte sinistra della V)
  - Ogni controllo fatto a destra che non dia buon esito porta a un rifacimento/modifica di quanto fatto a sinistra
  - **Parallelismo**: creazione dei test e una volta che ho il codice li eseguo
  - **Problemi (anche per Waterfall)**:
    - \* Versione funzionante solo alla fine!
    - \* Errore in fase iniziale può avere conseguenze disastrose



## 1.5 Modelli evolutivi

Idea: sviluppare un'implementazione iniziale, esporla agli utenti e raffinarla attraverso successivi rilasci del SW (release)

Sottocategorie:

- Prototyping
- Modelli incrementali
- Modelli iterativi

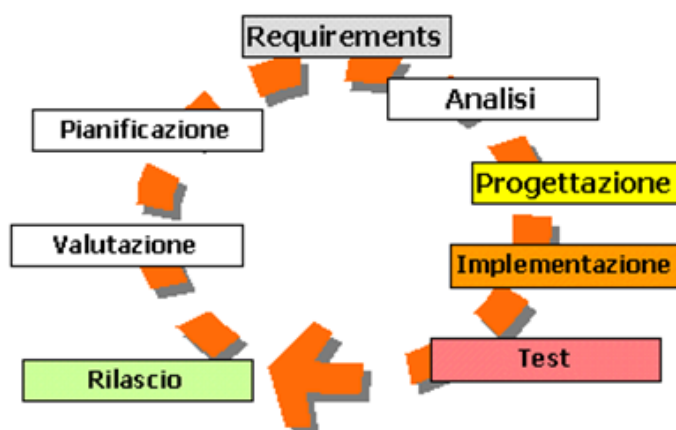
### 1.5.1 Modelli a Prototyping

- Realizzazione di un prototipo funzionante del sistema, su cui validare i requisiti (o l'architettura)
- Il prototipo ha meno funzionalità ed è meno efficiente

Vantaggi	Svantaggi
Permette di raffinare requisiti definiti in termini di obiettivi generali e troppo vaghi	Il prototipo è un meccanismo per identificare i requisiti, spesso da "buttare": problema economico e psicologico, il rischio è di non farlo e così scelte non ideali diventano parte integrante del sistema
Rilevazione precoce di errori di interpretazione	

### 1.5.2 Modelli Iterativi-Incrementali

- Sviluppo di varie release, di cui solo l'ultima è completa
- Dopo la prima release, si procede in parallelo
- Le fasi di sviluppo vengono percorse più volte



#### Modelli Incrementali

- Ogni release aggiunge nuove funzionalità
- Nella fase di pianificazione si decide il requisito/funzionalità da includere nella release successiva.
- Si trattano per prime le funzionalità ad alto rischio
- Si cerca di massimizzare il valore per gli utenti

## Modelli Iterativi

- Da subito sono presenti tutte (o buona parte) delle funzionalità che sono via via raffinate, migliorate

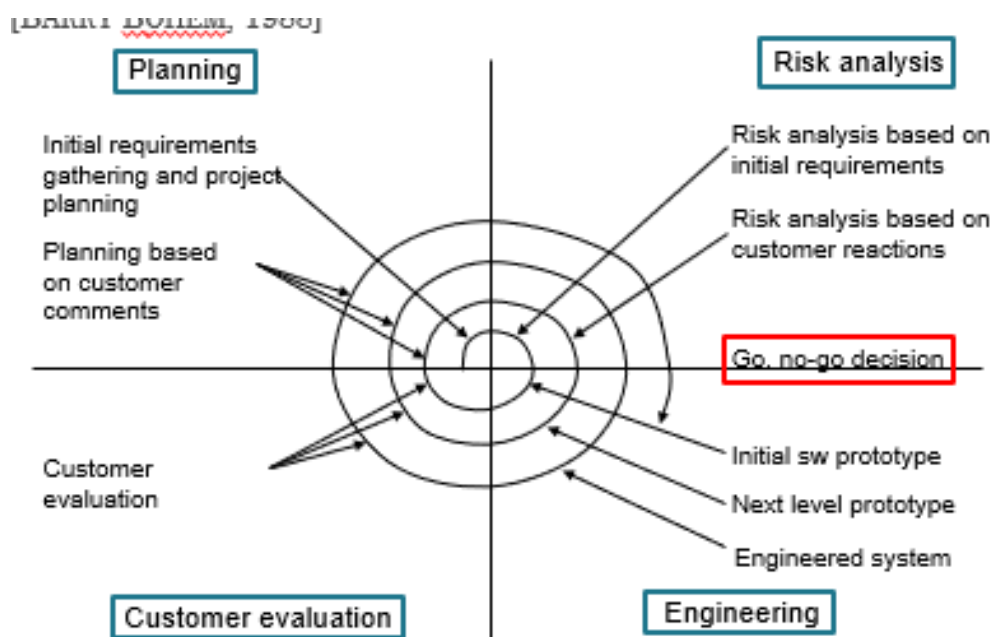
## 1.6 Modello a spirale

- Sistemi di grandi dimensioni
- Approccio "evolutivo" con interazioni continue fra cliente e developer
- Modello "risk-driver": tutte le scelte sono basate sui risultati dell'analisi dei rischi
- 'Meta-modello': dà un'idea generale ma quando si inizia a lavorare bisogna scegliere un modello esistente
  - Requisiti chiari e stabili → modello a cascata
  - Requisiti confusi → prototipo

**Rischio:** circostanza potenzialmente avversa in grado di pregiudicare lo sviluppo e la qualità del software

Ogni scelta/decisione ha un rischio associato, due caratteristiche importanti nella valutazione di un rischio sono:

- Gravità delle conseguenze
- Probabilità che si verifichi la circostanza



- **Planning:** determinazione di obiettivi, alternative, vincoli
- **Risk Analysis:** analisi delle alternative e identificazione/risoluzione dei rischi
- **Engineering:** sviluppo del prodotto di successivo livello
- **Customer Evaluation:** valutazione dei risultati dell'engineering dal punto di vista del cliente

<b>Vantaggi</b>	<b>Svantaggi</b>
Adatto allo sviluppo di sistemi complessi	Non è un rimedio universale (panacea)
Primo approccio che considera il rischio (risk-driver)	Necessita competenze di alto livello per la stima dei rischi
	Richiede un'opportuna personalizzazione ed esperienza di utilizzo
	Se un rischio rilevante non viene scoperto o tenuto a bada si inizia da zero

## 1.7 Unified Process

- Specifico per sistemi ad oggetti, con uso di notazione UML per tutto il processo
- Guidato dagli **Use Case**
- Incorpora molte delle idee 'buone' dal modello a spirale
- Meta-modello
- Supportato da tool(visuali) in ogni fase
- Processo prescrittivo per eccellenza

### 1.7.1 Le iterazioni

- Possibili diverse iterazioni che terminano con il rilascio del prodotto
- Ogni iterazione consiste di quattro fasi (anche ripetute più volte) che terminano con una milestone (= rilascio di artefatti soggetti a controllo)
- Ogni fase è costituita da diverse attività:
  - Requisiti (R)
  - Analisi (A)
  - Design (D)
  - Codifica (C)
  - Testing (T)

### 1.7.2 Le fasi

- Inception: studio di fattibilità, requisiti essenziali del sistema, risk analysis
- Elaboration: sviluppa la comprensione del dominio e del problema, gli Use Case della release da rilasciare, l'architettura del sistema
- Construction: Design (in UML), codifica e testing del Sistema
- Transition: Messa in esercizio della release nel suo ambiente (deploy), training e testing da parte di utenti fidati

## 1.8 Sviluppo basato sui componenti

Modello che va nella direzione del **riutilizzo del software**

Vantaggi	Svantaggi
Riduce la quantità di software da scrivere	Sono necessari dei compromessi: requisiti iniziali potrebbero differire da quelli che si possono soddisfare con le componenti disponibili
Riduce i costi totali di sviluppo e i rischi	Integrazione non sempre facile
Consegne più veloci	Spesso i componenti usati sono fatti evolvere dalla ditta produttrice senza controllo di chi li usa

## 1.9 Metodi Plan-Driven e Agili

Plan-Driven	Agile
Seguono un approccio classico dell'ingegneria dei sistemi fondato su processi ben definiti e ocn passi standard	Rispondere ai cambiamenti dei requisiti in modo veloce
	Filosofia del programmare come "arte" piuttosto che processo industriale
	Cosa più importante soddisfare il cliente e non seguire un piano (contratto)

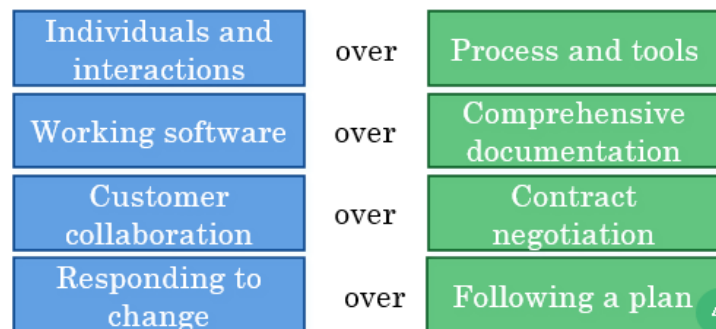


Figura 1.1: The Agile Manifesto

### 1.9.1 Come scegliere?

#### Metodi plan-driven:

- Sistemi grandi e comploessi, safety-critical o con forti richieste di affidabilità
- Requisiti stabili e ambiente predicibile

#### Metodi agili:

- Sistemi e team piccoli, clienti e utenti disponibili, ambiente e requisiti volatili
- Team con molta esperienza
- Tempi di consegna rapidi

## 1.10 DevOps

Metodo di sviluppo evolutivo

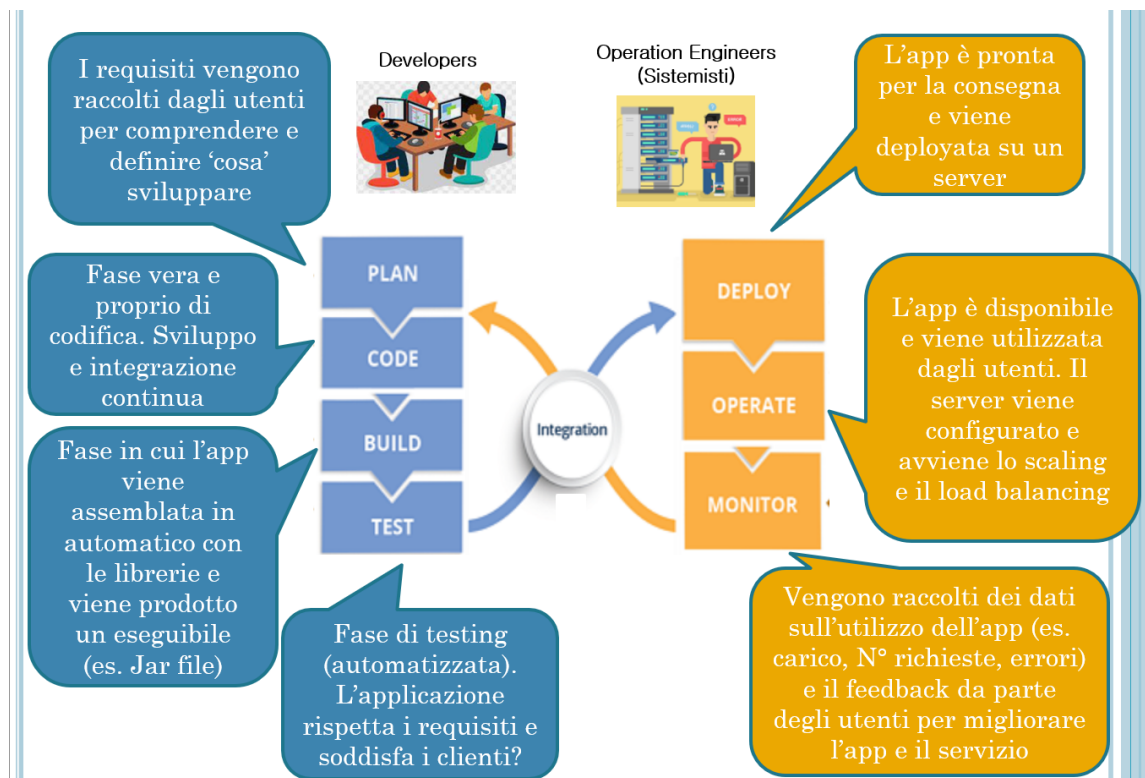


Figura 1.2: DevOps

### 1.10.1 Continuous Integration

La Continuous Integration (CI), o Integrazione Continua, è una pratica di sviluppo software in cui i programmatori integrano frequentemente il proprio lavoro (codice) nel repository condiviso del progetto, in genere diverse volte al giorno.

# Capitolo 2

## Ingegneria dei requisiti

### 2.1 Introduzione

Descrivere 'qualcosa' che il sistema dovrà fare (una funzionalità) o un vincolo a cui deve sottostare

- **Diversi livelli di astrazione:**
  - Descrizione astratta ed imprecisa del sistema
  - Descrizione dettagliata e matematica dello stesso

Che cosa il sistema farà e non come!
--------------------------------------

E' importante definire i requisiti in modo da evitare difetti in fasi avanzate del progetto, infatti i difetti dovrebbero essere scoperti il più presto possibile, ovvero a livello dei requisiti.

### 2.2 Classificazione dei requisiti

- **Requisiti utente:** descrizione in linguaggio naturale delle funzionalità che il sistema dovrà fornire e dei vincoli operativi (sono scritti per (e con) il cliente)
- **Requisiti di sistema:** descrive in modo dettagliato le funzionalità che il sistema dovrà fornire (sono scritti per gli sviluppatori)
- **Requisiti funzionali:** descrivono ciò che il sistema dovrà fare, non come ma cosa
- **Requisiti non-funzionali:** definiscono vincoli sul sistema e sullo sviluppo del sistema, in generale riguardano la scelta di linguaggi, piattaaforme, strumenti, tecniche d'implementazione, ma anche: prestazioni, questioni etiche, ...

Un requisito etico può essere ad esempio che nella realizzazione dell'applicazione verranno utilizzato solo strumenti e servizi 'non proprietari' (es. no Microsoft)

#### 2.2.1 Esempio: Bancomat

In **rosso** i requisiti funzionali, in **blu** i requisiti non funzionali

- Il sistema deve mettere a disposizione le funzioni di prelievo, saldo e estratto conto
- Il sistema deve essere disponibile a persone portatori di Handicap, deve garantire un tempo di risposta inferiore al minuto, e deve essere sviluppato su architettura X86 con sistema operativo compatibile con quello della Banca
- Le operazioni di prelievo devono richiedere autenticazione tramite un codice segreto memorizzato sulla carta
- Il sistema deve essere facilmente espandibile, e adattabile alle future esigenze bancare

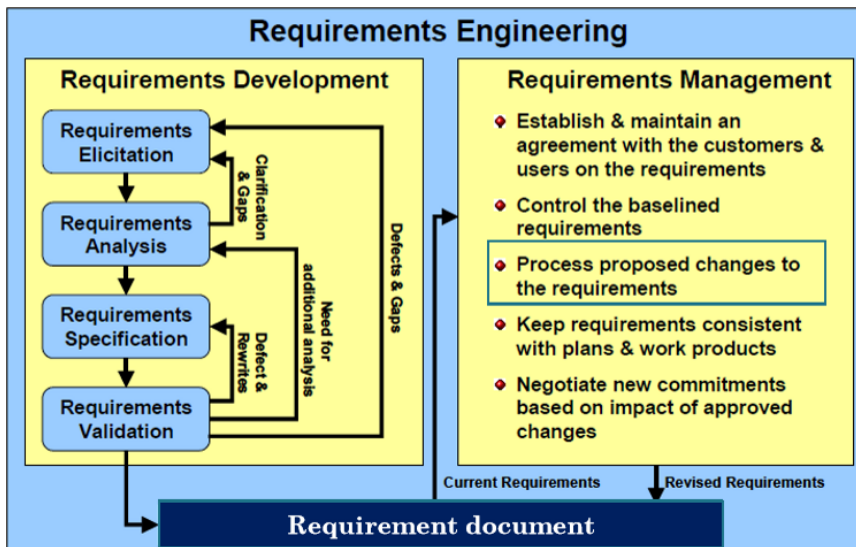
## 2.3 Requirements Engineering

E' il termine usato per descrivere le attività necessarie per raccogliere, documentare e tenere aggiornato l'insieme dei requisiti di un sistema software.

### 2.3.1 Scopo

Lo scopo primario del RE è la produzione di un documento (il requirement document) che definisca le funzionalità e i servizi offerti dal sistema da realizzare (anche tenerlo aggiornato)

### 2.3.2 Processo iterativo



- **Elicitation:**

- Ottenere, estrarre, ricavare, tirar fuori i requisiti dal cliente e da altri partecipanti
- Il primo passo è identificare gli stakeholders<sup>1</sup>
- Interviste, osservazioni sul luogo di lavoro, questionari, analisi dei prodotti dei competitors, workshop (brainstorming)
- Studio/analisi di leggi e regolamenti, help-desk reports, 'change requests' di prodotti analoghi, 'lessons learned' in progetti simili, ...

- **Analisi dei requisiti:**

- I bisogni (user needs) degli stakeholders raccolti durante la fase di elicitation sono analizzati e raffinati
- Si cerca di capire se i requisiti sono corretti
- Si cercano di identificare i "missing requirements"
- Si identificano requisiti poco chiari
- Si risolvono i requisiti "contraddittori o in conflitto"
- Viene stabilita la priorità (prioritizzazione):
  - \* Per sapere cosa "tagliare" se non tutti potranno essere realizzati
  - \* Scala numerica
  - \* Scala MoSCoW:

<sup>1</sup>Stakeholder: persona veramente interessata allo sviluppo del progetto



- **Must have:** requisiti obbligatori
- **Should have:** requisiti importanti ma non indispensabili
- **Could have:** requisiti desiderabili ma non necessari

- **Definizione e specifica:**

- Definizione dei requisiti utente: costituisce un contratto fra le parti
- Specifica dei requisiti di sistema: costituisce "starting point" per la fase di design

- **Validazione:**

- Esame della definizione/specifica dei requisiti per valutarne la qualità
- Di solito la convalida o validazione si effettua mediante 'formal peer reviews'
- Scrivere dei casi di test a partire dai requisiti
- Sviluppare un prototipo

- **Requirements Management:**

- Approvazione di alcune richieste di cambio dei requisiti
- Negoziazione con il cliente
- Impact analysis per i cambi richiesti
- Tenere allineati i requisiti e il codice (e casi di test)
- Tracciare il progresso di un progetto

## 2.4 Proprietà dei requisiti

- **Validità-correttezza**
- **Consistenza:** non ci sono requisiti contraddittori
- **Completezza:** tutti gli aspetti che il cliente vuole sono coperti nei requisiti (in teoria)
- **Realismo:** non si chiede l'impossibile
- **Inequivocabilità (Unambiguos):** ogni requisito dovrebbe avere solo un'interpretazione
- **Verificabilità:** i requisiti vanno espressi in modo che siano testabili
- **Tracciabilità:**
  - Ogni funzionalità implementata nel sistema deve poter essere fatta risalire a dei requisiti in modo semplice
  - Ogni requisito nella requirement specification deve corrispondere ad uno nella requirement definition

## 2.5 Template/Schema dei requisiti

Conviene attenersi a questo Schema

`<id> il <sistema> deve <funzione>`

Es. R1. Il sistema deve gestire tutti i registratori di cassa del negozio (non più di 20)

## 2.6 Analista software

L'analista software o di sistema è la persona che:

- si occupa dell'elicitazione dei requisiti
- analizza i requisiti
- scrive il documento dei requisiti (definizione e/o specifica)
- Comunica/spiega i requisiti a sviluppatori e altri stakeholder

Alcune competenze che un analista dovrebbe avere:

- Arte della negoziazione
- Stabilire una strategia (problem solving)
- Giusta capacità di imporsi
- Ascoltare attentamente
- Dono della sintesi
- Padronanza del linguaggio naturale
- Buona conoscenza del dominio (ad esempio in ambito medico o automobilistico)

### 2.6.1 Consigli per un'intervista

1. Fare molte domande
2. Ascoltare bene
3. Mettere in discussione i quantificatori universali: 'tutto, ogni, sempre, ...'
4. Annotare tutte le risposte

### 2.6.2 Importanza della comunicazione

- Elicitation = Attività molto delicata perchè mette in comunicazione due o più persone di realtà anche molto diverse
- Frequenti incomprensioni, che si ripercuotono sulla qualità dei requisiti

Occorre fare molta attenzione a:

- Diversità di significato che si attribuisce ai termini → possibile soluzione definizione del glossario:
  - Per la spiegazione dei termini tecnici
  - Per ridurre l'ambiguità dei termini usati
  - Per "espandere" gli acronimi
- Assunzioni nascoste (Hidden assumptions)
- Verbosità (= sovrabbondanza di parole)
- Mancanza di chiarezza/precisione

## 2.7 Consigli finali

- Riutilizzo di (parte di) requisiti
- Utilizzo di un glossario comune tra clienti, utenti e analisti
- Utilizzo di un 'buon' template/form
- Utilizzo di un software per la gestione/raccolta e analisi dei requisiti

# Capitolo 3

## Definizione dei requisiti basata su use case

### 3.1 Cosa sono/servono

- Esprimere requisiti funzionali di un sistema
- Descrivere dal punto di vista di chi lo usa un sistema, il sistema è visto come una black-box
- Totalmente indipendenti dal mondo OO
- Solo testo, formattato in modo standard (template)
- Visuale lo Use Case Diagram (UML)
- Gli use case esprimono l'interazione tra le entità (attori) che interagiscono con il sistema stesso

### 3.2 Differenza tra requisito e use case

- **Requisito:** descrive una funzionalità dal punto di vista del sistema
- **Caso d'uso:** descrive una modalità di utilizzo del sistema da parte di un utilizzatore (punto di vista dall'utente)

La differenza sostanziale è nel modo in cui è presentata l'informazione

### 3.3 Definizione dei requisiti basata su use case

- **Attore:** rappresenta un ruolo che un'entità esterna "recita" interagendo con il sistema, da non confondere un ruolo con la cosa stessa
  - **Primari:** chi guadagna qualcosa dal sistema (ad esempio un cliente Amazon)
  - **Secondari:** chi produce qualcosa (o offre un servizio) per il sistema (ad esempio Paypal)
- **Use case:** quello che gli attori 'possono fare' con il sistema
- **Relazioni:** tra gli attori e gli use case
- **Confini del sistema:** un rettangolo disegnato intorno agli use case per indicare i confini del sistema, quando si vuole costruire un sistema è la prima cosa da fare

## 3.4 Scenario

Uno scenario è una sequenza ordinata di interazioni tra un sistema e gli attori
---

Rappresenta una particolare esecuzione di uno use case (istanza), e rappresenta un singolo cammino dello use case, sono usati per il testing. Si possono avere diversi scenari, ma in tutti l'attore può avere lo stesso scopo.

## 3.5 Use Case

Insieme di scenari che hanno in comune lo scopo finale dell'attore
--

- Gli use case in genere sono dati come testo strutturato
- I passi di uno use case sono testo facile da capire
- Viene usato il vocabolario del dominio dell'applicazione
- Gli use case sono descrizioni chiare, precise, generali e indipendenti dalle tecnologie

### 3.5.1 Descrivere uno use case

- **Scenario principale:** scenario del mondo perfetto
- **Scenari secondari:** cosa può succedere di sbagliato o differente e come gestirlo

### 3.5.2 Template

- **Nome dello use case:** è il goal dello use case "**breve frase verbale attiva**" in UpperCamelCase
- **Identificatore:** di solito numerico progressivo
- **Breve descrizione:** un paragrafo che fissa l'obiettivo dello use case
- **Attori primari:** l'attore/gli attori primari dello use case
- **Attori secondari:** gli attori che "servono" per svolgere lo use case
- **Precondizioni:** vincoli sullo stato corrente del sistema
- **Scenario principale:** i passi che costituiscono lo use case
- **Postcondizioni:** condizioni che devono essere vere quando lo use case termina con successo l'esecuzione dello scenario principale
- **Scenari alternativi:** un elenco di alternative allo scenario principale

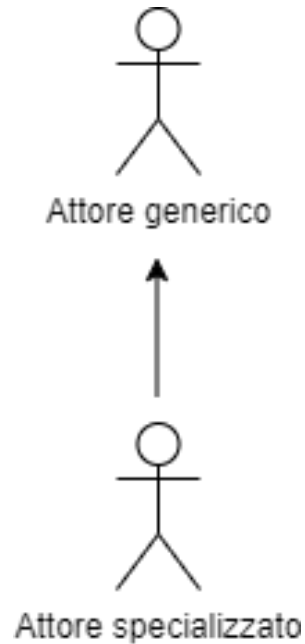
Uno scenario è costituito da un elenco di passi che devono essere concisi, numerati e ordinati temporalmente: `<numero> Il <Sistema/Attore> <qualche azione>`. Si ha una deviazione tutte le volte che 'ci si allontana' dallo scenario principale:

- **Semplici:** usare parola chiave **se** nella sequenza principale
- **Complesse:** scrivere sequenze degli eventi alternative che rappresentano errori o casi particolari che non ritornano sullo scenario principale

Si possono usare ripetizioni all'interno di una sequenza: **per** e **fintantochè**.

E' possibile che una sequenza venga attivata in qualunque momento della sequenza principale.

## 3.6 Gerarchia attori



Come per le classi di Java, l'attore specializzato eredita le relazioni dell'attore generale

## 3.7 Relazioni tra use case

- **Inclusione** <<include>>:
  - Assomiglia al concetto di procedura/funzione
  - Lo use case "principale" esegue i passi fino al punto di inclusione e passa il controllo allo use case incluso, alla fine il controllo ritorna allo use case principale
  - Lo use case principale senza use case incluso risulta incompleto
- **Estensione** <<extend>>: per estendere il comportamento di uno use case con un comportamento aggiuntivo (opzionale) rispetto allo use case base
- **Generalizzazione/specializzazione**: gli use case specializzati (figli) rappresentano delle varianti più specifiche dello use case generalizzato (genitore) da cui ereditano, i 'figli' possono
  - Ereditare i passi del genitore
  - Aggiungere nuovi passi
  - Ridefinire (modificare) i passi ereditati

# Capitolo 4

## Design architetturale

### 4.1 Introduzione

Trasforma un problem in una soluzione (come)
--

Il design definisce la struttura della soluzione invece l'implementazione la realizza, rendendola usabile.

- **Architectural design (high-level)**: mappa i requisiti su architettura SW e componenti/sottosistemi
- **Component design (low-level)**: fissa dettagli dei componenti, specificando maggiormente la soluzione

Scelte tecnologiche:

- **platform-independent design**: come pro ha il riuso
- **platform-specific design**: come pro aiuta i programmatori, per specifico si intende ad esempio nominare già le strutture dati specifiche per un linguaggio.

#### 4.1.1 Livelli di ri-uso

- **Clonazione**: si riutilizza interamente design/codice, con piccoli aggiustamenti
- **Design pattern**: buona soluzione a problema ricorrente
- **Stili architetturali**: architettura generica che suggerisce come decomporre il sistema
- **Software Frameworks**: insieme di classi e interfacce cooperanti che realizzano un design per uno specifico dominio applicativo o tipologia di app

### 4.2 Design architetturale

Processo di design per identificare:

- le macro componenti di un sistema
- come avviene il controllo e la comunicazione tra componenti

Produce una descrizione dell'architettura software.

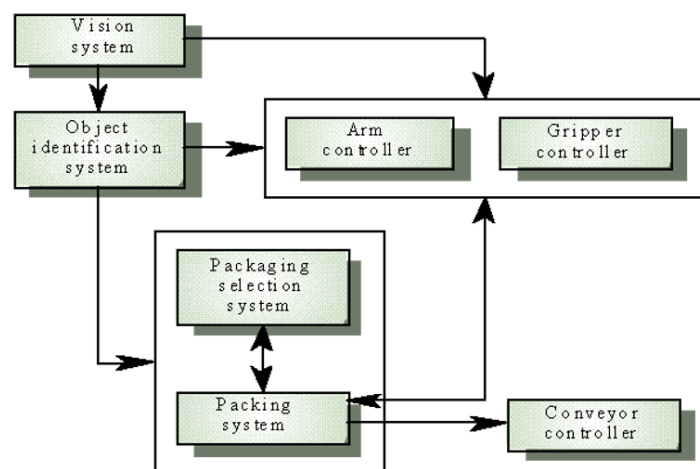
## 4.3 Componenti

Un modulo è un'unità del sistema che offre servizi ad altre unità, ma che non può essere considerato un sistema a se stante

Un sottosistema è un sistema di per sè: può essere eseguito ed utilizzato anche "da solo". Di solito i sottosistemi sono composti da moduli ed hanno interfacce ben definite, che sono utilizzate per la comunicazione con altri sottosistemi.

## 4.4 Diagramma a blocchi

Un architettura software è normalmente espressa mediante un diagramma a blocchi che presenta un "overview" della struttura del sistema. I blocchi sono i componenti, i connettori rappresentano le "relazioni" tra i componenti.



## 4.5 Architettura SW

### 4.5.1 Proprietà del sistema

- **Performance:** tempi di risposta rapidi
- **Security:** difficile da "manomettere", dati sensibili protetti
- **Safety:** non creare "disastri"
- **Availability:** 24//7/365
- **Maintainability:** semplice da mantenere/evolvere

### 4.5.2 Vantaggi

- Guida lo sviluppo ed aiuta nella comprensione del sistema
- Documenta il sistema
- Aiuta a ragionare sull'evoluzione del sistema
- Supporta decisioni manageriali
- Facilita l'analisi di alcune proprietà
- Permette il riuso (Large-scale)



### 4.5.3 Stili architetturali

L'architettura di un sistema può conformarsi a uno stile architetturale

- **Modello generico**: con caratteristiche specifiche che può essere istanziato/personalizzato
- **Layered**
- **Repository**
- **Client/server**
- **P2P**
- **Broadcast model**
- **Service Oriented Architecture**
- **Microservice**

Uno stile **strutturale** fornisce solo informazioni strutturali, uno **di controllo** anche informazioni (o solo) di controllo.

Conoscere gli stili architetturali può semplificare il problema di definire l'architettura software

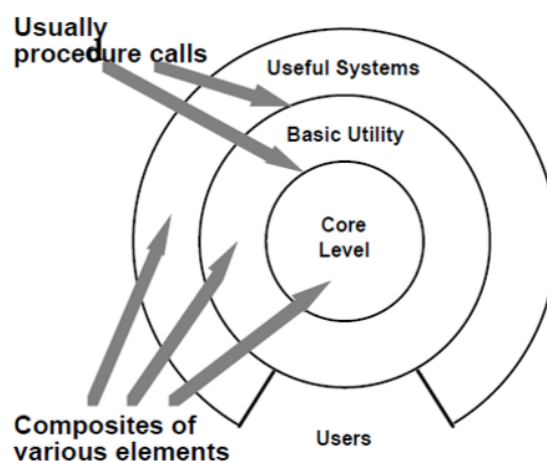
La maggioranza dei grandi sistemi sono eterogenei e non seguono un singolo stile architetturale

#### Elementi

- **Componenti**
- **Connettori**

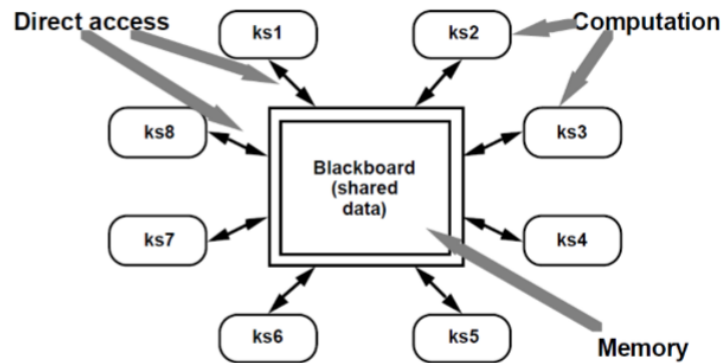
### 4.5.4 Layered

- Organizza il sistema in un insieme di livelli ognuno dei quali fornisce un insieme di servizi
- Un livello usa solo i servizi del livello inferiore



### 4.5.5 Repository

I dati condivisi sono mantenuti in un database centrale (repository o blackboard) a cui hanno accesso tutti i sotto-sistemi



### 4.5.6 Client/server

- Modello di sistema distribuito che mostra come i dati e la computazione possono essere distribuiti su:
  - Insieme di server che forniscono servizi specifici
  - Insieme di client che utilizzano tali servizi
- Esiste una rete che permette ai client di accedere ai server

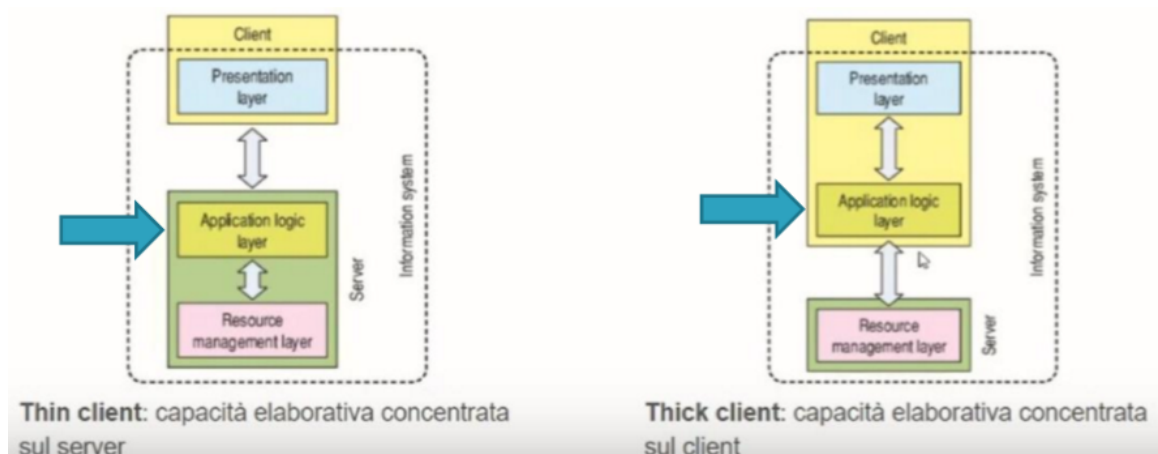
#### Two-Tier

Tre strati/componenti software:

- Interfaccia utente (presentation logic)
- Gestione dei processi e logica (business o appl. logic)
- Gestione del DB (data logic)

Distribuiti in due livelli ( $n$  client +  $m$  server)

- Client
- Server



## Three-Tier

- Sul client resta solo l'interfaccia utente
- La logica del sistema risiede sull'Application server e gestisce multi-utenti
- Gli strati di logica e gestione DB sono distribuiti su più DB server

### 4.5.7 Pipe and Filter

- I filtri effettuano trasformazioni che elaborano i loro input per produrre output
- Le pipe sono connettori che trasmettono i dati tra filtro e filtro

Ad esempio `ls -l | grep "Aug"` dove l'output di `ls -l` viene passato a `grep "Aug"` grazie alla pipe

### 4.5.8 Architetture eterogenee

Ci sono due modi di combinare gli stili ottenendo così un architettura eterogenea:

- Modo gerarchico
- Permettendo che una componente sia un mix di architetture

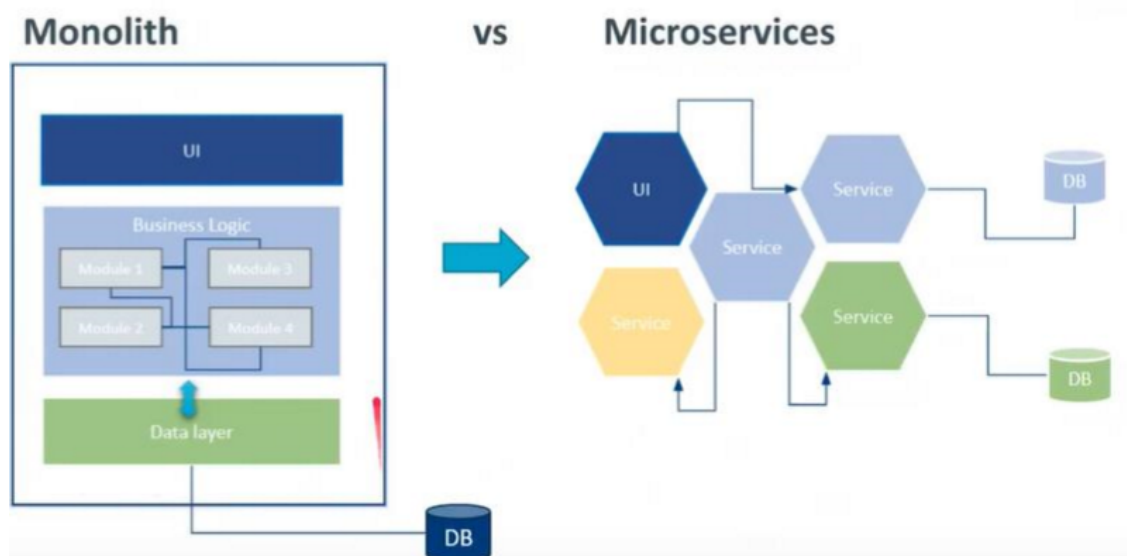
### 4.5.9 Microservices

Avere servizi separati e quindi potenzialmente più piccoli e facili da gestire (sviluppare, testare, deployare), anche dislocati su server diversi

#### Perchè?

Se la app cresce:

- La complessità aumenta
- Difficile trovare e risolvere bug
- Difficile effettuare modifiche
- Tempi estesi per il deploy
- Complesso lavorare in parallelo (team)



Ogni microservizio può essere scritto in un linguaggio di programmazione diverso ed avere diversi DBMS.

## Comunicazione

- **API Gateway:** espone un'interfaccia verso i client
- **REST:** le applicazioni basate su REST utilizzano le richieste HTTP per tutte e quattro le operazioni di CRUD (Create, Read, Update, Delete)

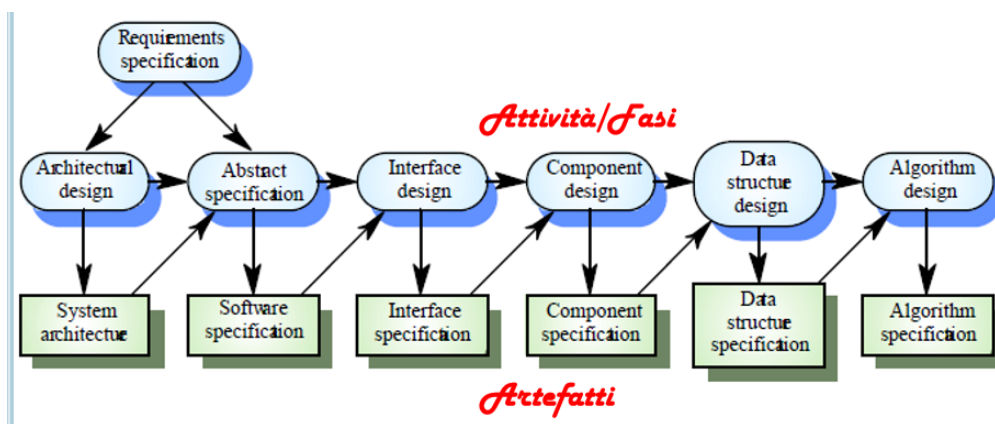
## Problemi

- Stabilire la dimensione dei micro servizi
- Sviluppo del meccanismo di comunicazione tra i servizi
- Esposizione ai disservizi di rete
- Gestione dello schema partizionato dei DB
- Difficoltà di testing
- Maggior consumo di risorse e memoria

# Capitolo 5

## Design delle componenti

### 5.1 Fasi



### 5.2 Design by contract

E' un metodo di design per il software che ha come obiettivo quello di migliorarne la qualità e prescrive che il progettista debba definire specifiche precise delle interfacce dei classi/componenti software, basandosi sulla metafora di un contratto legale (il "contratto", viene creato per ogni componente del sistema prima che sia codificato). L'idea centrale è che una componente software ha degli obblighi nei confronti delle altre componenti.

#### 5.2.1 Elementi di un contratto

- **Pre-condizione:** espressione a valori booleani rappresentante le aspettative sullo 'stato del mondo' prima che venga eseguita un'operazione
- **Post-condizione:** espressione a valori booleani riguardante lo 'stato del mondo' dopo l'esecuzione di un'operazione
- **Invariante di classe:** condizione che ogni oggetto della classe deve soddisfare in ogni momento in cui è possibile eseguire un'operazione

**Le precondizioni sono utili?** a prima vista potrebbe risultare inutile in quanto si possono aggiungere controlli all'interno del codice però chi è responsabile di questi controlli? Senza una dichiarazione esplicita potremmo avere

1. Troppi pochi controlli
2. Troppi controlli

## 5.2.2 Vantaggi

- Codifica: guida per lo sviluppatore durante la fase di codifica
- Migliorano la qualità del software: definisce quale componente è responsabile ad effettuare i controlli. Aiuta a scrivere operazioni semplici che soddisfino un contratto ben definito
- Documentazione: Pre, Post e Invarianti documentano in modo preciso cosa fa una componente/classe
- Testing: guida alla generazione di casi di test "black-box"
- Debugging: permette di trovare il "colpevole" di un malfunzionamento:
  - Le eccezioni si sollevano quando il contratto è violato

## 5.3 Progettazione degli algoritmi

E' l'attività più vicina alla codifica (spesso viene lasciata in parte o totalmente agli sviluppatori) e, di solito, si seguono i seguenti passi:

1. Si analizza la descrizione di design della classe "target"
2. Se esistono una o più operazioni che necessitano di un algoritmo, se è possibile selezionare un algoritmo noto si seleziona altrimenti occorre definire un algoritmo e si sceglie una notazione e si progetta utilizzando la nozione di "stepwise refinement"
3. Si usano i metodi formali per provare la correttezza dell'algoritmo proposto

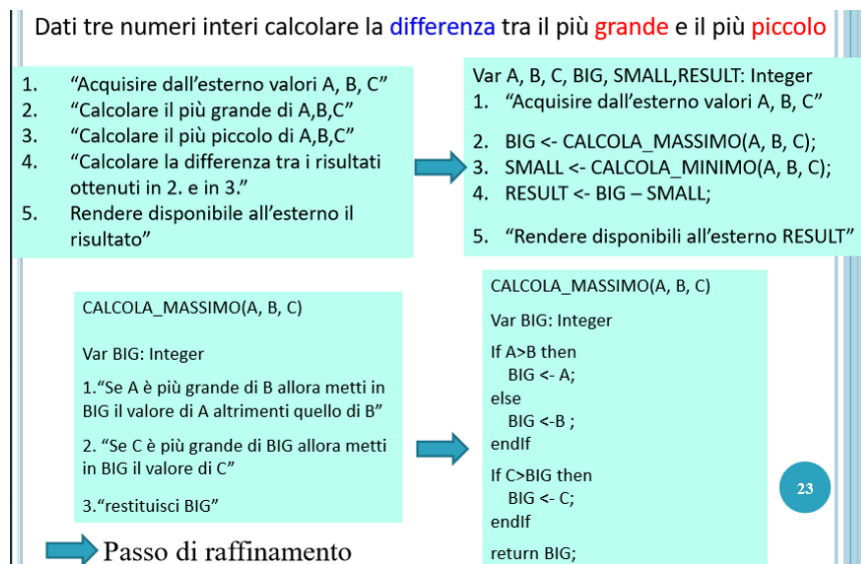


Figura 5.1: Stepwise Refinement

### 5.3.1 Notazioni

Esistono diverse notazioni utilizzate per rappresentare un algoritmo:

- **Visuali**: Activity Diagram di UML, Flowchart, Box Diagram, Structured Chart, Decision Table, ...
- **Testuali**: Program Design Language (PDL) o pseudocodice

## PDL

Il **Program Design Language** (PDL) o pseudocodice è un linguaggio semplificato che usa il vocabolario di un linguaggio naturale e la sintassi di un linguaggio di programmazione.

## 5.4 Principi di progettazione

I principi di progettazione guidano verso il raggiungimento degli obiettivi di qualità per il progetto

Porta a produrre software: manutenibile, comprensibile, semplice da testare, riusabile, riparabile e portatile

...

Si possono applicare per tutti i sistemi non solo Object Oriented!

### 5.4.1 Principi

#### Astrazione

Permette di concentrarsi su un problema ad un determinato livello di astrazione, senza perdersi in dettagli irrilevanti. Nasconde informazioni che a un determinato livello non servono.

Forme di astrazione:

- **Funzionale**:: definizione di una funzionalità indipendentemente dall'algoritmo che la implementa
- **di Dati**: definizione di un tipo di dato in base alle operazioni che su di esso possono essere fatte, senza definirne una struttura concreta
- **di Controllo**: definizione di un meccanismo di controllo senza indicarne i dettagli interni

#### Decomposizione

Cercare di risolvere un problema in una volta sola è in genere più difficile che risolverlo per parti.

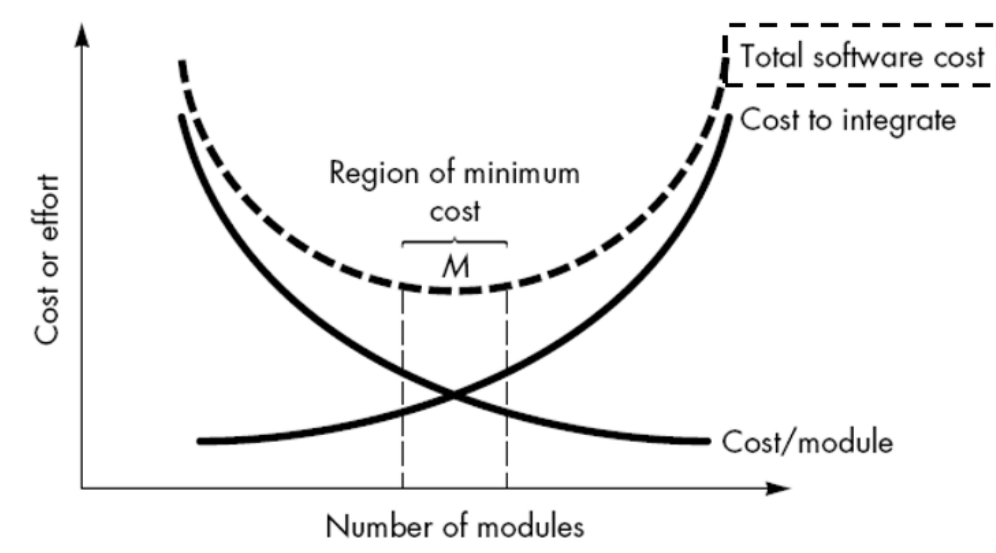
#### Legenda:

- $P$  problema
- $C(P)$  complessità di  $P$
- $E(P)$  effort (sforzo) per la risoluzione (software) di  $P$

Dati due problemi  $P_1$  e  $P_2$

- Se  $C(P_1) > C(P_2)$  allora vale  $E(P_1) > E(P_2)$
- $C(P_1 + P_2) > C(P_1) + C(P_2)$  vale empiricamente
- $E(P_1 + P_2) > E(P_1) + E(P_2)$

L'ultima disequazione porterebbe a una conclusione sbagliata che se noi dividessimo il problema e quindi il software infinite volte l'effort di sviluppo diventerebbe nullo. In realtà entrano in gioco altre variabili:



Quindi dal grafico sappiamo che non si deve dividere troppo nè troppo poco.

## Modularità

**Modulo:** è un'entità SW, identificata da un nome che può fornire servizi software; contiene istruzioni, strutture dati, controllo; può essere incluso in un altro modulo; può usare un altro modulo o parte di esso con la relazione 'dipende da'

E' una conseguenza del principio di decomposizione, l'idea è quella di tenere separati gli aspetti "un-related" di un software (**Separation of concerns**). Una valida linea guida è tenere separati il SW di generazione dati dal SW necessario alla loro presentazione e permette di cambiare la rappresentazione sullo schermo senza dover modificare il sistema di calcolo. La struttura di un'applicazione software, è spesso caratterizzata da tre livelli:

- **Presentazione:** insieme dei moduli che gestiscono l'interazione con l'utente
- **Logica Applicativa:** insieme dei moduli che realizzano la logica applicativa, implementano le funzionalità richieste e gestiscono il flusso dei dati
- **Dati e Risorse:** insieme dei moduli che gestiscono i dati che rappresentano le informazioni utilizzate

Per modularizzare sostanzialmente bisogna fare in modo che ogni modulo esegua un singolo compito e minimizzare il numero e la complessità delle interconnessioni fra moduli. Dei moduli coesi e poco accoppiati sono

- facili da comprendere
- riusabili
- semplici da modificare
- semplici da testare

Bisogna evitare i moduli "monster" e per far ciò bisogna seguire i criteri per una buona modularizzazione: moduli piccoli, separation of concerns, alta coesione e basso accoppiamento e usare delle metriche o delle view per vedere se i criteri sono stati applicati.

Un modulo **coeso** svolge un unico compito, in altre parole coesione è quando un modulo deve esprimere una sola astrazione, esistono diverse tipologie di coesione ma di solito si intende quella **funzionale**: tutti gli elementi del modulo contribuiscono ad un singolo ben definito task.



Ogni modulo non deve dipendere da troppi altri moduli, nè dipendervi in modo troppo forte

Il coupling misura (informalmente) il grado di dipendenza di un modulo dagli altri e come prima non bisogna collegarli troppo e troppo poco.

- Accoppiamento buono: chiamata di routine/metodo/funzione di altro modulo, uso di tipo di dato definito in altro modulo, inclusione o importazione di un package o di una libreria
- Accoppiamento cattivo: content coupling, un modulo modifica il valore di una variabile in un altro modulo

L'accoppiamento cattivo va evitato perchè complica enormemente la comprensione e la modifica, in un sistema OO si riduce incapsulando tutti i campi di una classe dichiarandoli privati e fornendo i metodi di get e set. Nei linguaggi 'Legacy' le cose sono più complicate...

- **Fan-in**: numero di archi entranti in un modulo
- **Fan-out**: numero degli archi uscenti da un modulo

Un alto numero di Fan-in indica un buon riuso, un alto numero di Fan-out indica eccessiva dipendenza e che il modulo "fa troppo" (va decomposto).

**Generalità**: è la proprietà di design che "migliora" il riuso di un modulo in altri progetti (in futuro), si cerca di rendere un modulo il più generale possibile in modo da poterlo usare in più contesti. Questo principio è controverso perchè 'peggiora' un altro principio quello della 'semplicità' (KISS).

### 5.4.2 Principio KISS

Il design dovrebbe essere il più semplice possibile.

- Keep It Simple, Stupid
- Keep It Short and Simple

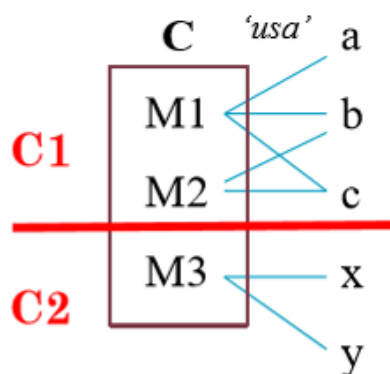
### 5.4.3 Sto seguendo i principi?

Esistono delle metriche del SW che ci permettono di misurare l'aderenza a principi che abbiamo visto:

- Metrica SW, ovvero la misura quantitativa del grado di possesso di uno specifico attributo da parte di un sistema, un componente, o un processo (es. LCOM (Lack of Cohesion))
- Esistono dei tool che forniscono metriche e viste

#### Esempio di metrica LCOM

$LCOM = \text{numero di intersezioni vuote} - \text{numero di intersezioni non vuote}$



Le intersezioni saranno  $I_1 = \{a, b, c\}$ ,  $I_2 = \{b, c\}$ ,  $I_3 = \{x, y\}$  quindi si avrà che  $I_1 \cap I_2 \neq \emptyset$  mentre  $I_1 \cap I_3 = I_2 \cap I_3 = \emptyset$  quindi avremo che  $LCOM = 2 - 1 = 1$  e visto che è  $> 0$  classe C non è coesa.

# Capitolo 6

## UML - Introduzione

### 6.1 Cos'è

L'UML è una famiglia di notazioni grafiche utili a supportare la descrizione e il progetto dei sistemi software. E' stato proposto come standard dall'OMG (Object Management Group) che ha come obiettivo di creare standard nel contesto IT.

E' unificato perchè in pratica è una collezione di varie notazioni preesistenti (Rumbaugh, Booch, Jackson, Harel), integrate e rese Object Oriented.

L'UML **non** è:

- Un metodo/processo di sviluppo → e' un linguaggio/notazione
- Un linguaggio di programmazione → è un linguaggio di modellazione
- Legato a UP (Unified Process) → è indipendente dal processo

L'UML può essere usato in diversi modi:

- **Abbozzo (sketch)**: per aiutare la comunicazione e la discussione
- **Progetto dettagliato (blueprint)**: per fornire un modello completo/dettagliato da implementare
- **Linguaggio di programmazione**: per fornire un "modello eseguibile"

### 6.2 Prospettive

- **Prospettiva software**: gli elementi rappresentati sono elementi di un sistema software, serve per descrivere il design di un sistema software e per documentare un sistema software
- **Prospettiva concettuale**: gli elementi rappresentati sono concetti del dominio e serve per modellare il dominio del business e i processi

### 6.3 Modello di dominio (del business)

E' una rappresentazione visuale di classi concettuali o di oggetti del mondo reale di un dominio ed è un glossario visuale delle astrazioni significative del dominio.

E' utile per:

- Chiarire i concetti di dominio nel quale il sistema software che dobbiamo progettare andrà inserito
- Ispirare nomi di alcune classi e attributi software nel design
- La progettazione del sistema

## 6.4 Un po' di storia

Anni '80	1980-1995	1994	1995	1996	1997	11/1997	03/2005	Adesso
Linguaggi ad oggetti iniziano a prendere piede	"babele" di linguaggi di modellazione	Rumbaugh si unisce a Booch alla Rational Software (azienda)	Booch e Rumbaugh presentano OOPSLA Unified Method v 0.8 Viene dato l'annuncio che Rational ha comprato Objectory dove lavora Jacobson	OMG entra in campo sollecitata dai venditori di software terrorizzati dalla possibilità che Rational creasse un standard de-facto	call for proposal da parte di OMG. "Tre amigos" presentano UML 1.0	Nasce UML 1.1 (standard OMG)	UML 2.0 diventa la versione ufficiale	UML 2.5.1

## 6.5 Perché

Studiare UML è utile perchè è uno standard, è usato/richiesto dall'industria (almeno a livello di abbozzo) ed è indispensabile per le figure di Analista e Progettista (architetto) software.

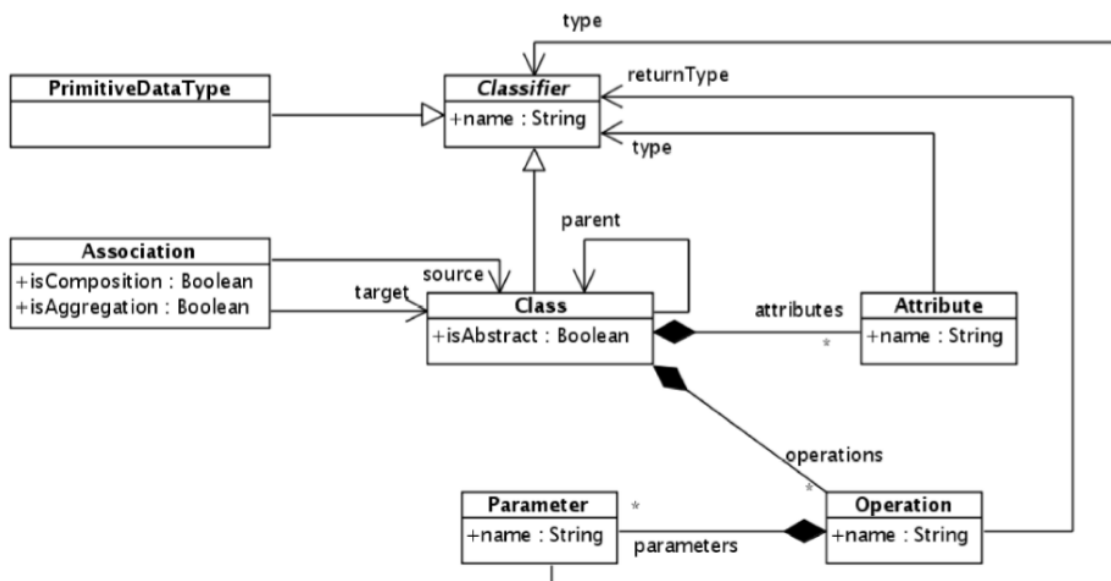
Non lo studieremo in modo approfondito perchè, ad esempio, l'approccio blueprint è poco usato nell'industria.

## 6.6 Notazione e meta-modello

UML definisce una famiglia di notazioni e un meta-modello.

Notazione = sintassi grafica del linguaggio Meta-modello = definisce i concetti stessi del linguaggio
--

**Esempio di meta-modello:** UML di una classe (generica) Java



In UML un sistema viene descritto utilizzando diverse viste e diversi diagrammi

Vista = particolare aspetto di un sistema SW dal punti di vista di uno specifico ruolo
--

Un diagramma descrive/presenta una view/subview, chiameremo **Modello** un insieme di diagrammi che descrivono aspetti diversi dello stesso sistema SW. In un modello bisogna fare attenzione a non avere viste possibilmente sovrapposte (problema di consistenza) e incomplete (problema di completezza).

Il diagramma UML non è abbastanza e per questo esistono due soluzioni:

- Si cerca o si produce un "estensione" di UML (profilo UML)
- Si usano diagrammi non UML

Facendo ciò si perde uno dei vantaggi di UML: la standardizzazione.

### 6.6.1 Profilo

Come anticipato, UML si può estendere tramite il concetto di profilo per avere la possibilità di specificare particolari domini applicativi o tipologie di applicazioni.

Un profilo è costituito principalmente da:

- **Stereotipi**: elementi aggiuntivi ottenuti modificando quelli esistenti
- **Vincoli aggiuntivi**
- **Informazioni semantiche aggiuntive** relative agli elementi aggiuntivi

Non si può guardare un diagramma UML e dire esattamente cosa farà il codice corrispondente
--

Esiste un UML "legale"? La risposta è sì e no: quello definito dalla specifica del linguaggio (documento OMG) è quello legale ma UML è molto complesso e si presta a multiple interpretazioni, così le persone lo usano adottando convenzioni particolari (per semplificare) e quindi si perde la standardizzazione. Di solito si aggiunge nei diagrammi "**non-normativo**".

## 6.7 Come usare UML

1. Capire il punti di vista dell'autore per capire i diagrammi
2. Meglio dei buoni diagrammi "illegali" che altri formalmente corretti ma "scarsi"
3. UML è molto complesso da prestarsi a multiple interpretazioni: usate un sottoinsieme e solo i diagrammi che ritenete utili (e che avete compreso bene)
4. La completezza è nemica della chiarezza
5. Qualsiasi informazione in UML può essere sempre soppressa

# Capitolo 7

## UML - Class Diagram

### 7.1 Introduzione

Definisce:

- Le classi
- Le loro feature
  - Attributi
  - Operazioni
- Le relazioni tra classi
  - Associazioni
  - Aggregazione/Composizione
  - Specializzazione/Generalizzazione
  - Dipendenze

Ovvero la parte statica di quello che intendiamo modellare.

### 7.2 Prospettive

Il significato di un class diagram e dei suoi elementi dipende dalla prospettiva:

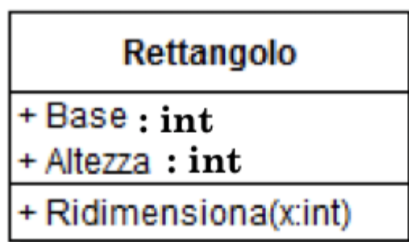
- Prospettiva concettuale: descriviamo gli elementi del "pezzo di mondo" che ci interessa modellare, la classe UML è un concetto proprio del dominio e un'operazione UML è un'azione/responsabilità
- Prospettiva software: descriviamo il design di un software, ovvero i modulo software che costituiranno l'implementazione vera e propria del sistema, la classe UML è una classe in un linguaggio OO e un'operazione UML è implementata da un metodo

### 7.3 Classe

Il concetto di classe è lo stesso dell'OO, incapsula caratteristiche comuni ad un gruppo di progetti, una classe genera oggetto con `create()` e un oggetto è un istanza di una classe. Gli attributi determinano lo stato degli oggetti mentre le operazioni descrivono il comportamento. Gli oggetti connessi tra loro possono collaborare per compiere task più complessi mediante scambio di messaggi.

### 7.3.1 Rappresentazione

Una classe in UML è semplicemente rappresentata da un rettangolo con il nome della classe all'interno, possiamo aggiungere gli attributi e le operazioni.



Se una classe è **abstract** viene indicata con il nome in corsivo.

Si ha una dipendenza tra due classi se la modifica di una classe può avere effetto sull'altra, si rappresenta con una freccia tratteggiata.

### 7.3.2 Attributi

visibilità nome: tipo [molteplicità] = default {proprietà}

- **Visibilità:**
  - +: pubblico
  - -: privato
  - #: protetto
  - ~: package
- Il nome dell'attributo è l'unica parte obbligatoria
- Il tipo dell'attributo può essere:
  - Primitivo
  - Il nome di una classe definita nello stesso modello
- Default rappresenta il valore di default dell'attributo di un oggetto appena creato
- Proprietà aggiuntive (es. `readOnly`)

Le regole di visibilità tra i vari linguaggi e UML sono spesso differenti.

Gli attributi possono essere **static** e, come per il linguaggio Java, gli oggetti di una stessa classe condividono lo stesso valore per un attributo. In UML si indica sottolineando il nome dell'attributo.

### 7.3.3 Molteplicità

La molteplicità indica il quantitativo degli attributi, alcuni valori possibili sono:

- 1 (uno e uno solo). E' il valore di default
- 0..1 (al più uno)
- \* (un numero imprecisato, eventualmente nessuno)
- 1..\* (almeno uno)
- n..m con  $m \geq 1$

Gli elementi di un attributo con molteplicità  $> 1$  sono considerati come un insieme, se essi sono dotati anche di ordine si aggiunge la proprietà **{ordered}**

### 7.3.4 Operazioni

visibilità nome (lista parametri) : tipo-ritornato proprietà

- Visibilità e nome stesse regole degli attributi
- Lista parametri contiene nome e tipo dei parametri, secondo la forma `direzione nome : tipo = default`
- Direzione: input (in), output (out) o entrambi (inout). Il valore di default è in.
- nome, tipo e valore di default sono analoghi a quelli degli attributi
- Tipo-ritornato è il tipo del valore di ritorno

Le operazioni get/set di solito non si indicano perchè è scontato che esistono

In UML esistono due tipi di operazioni:

- **Query** che ottengono un valore da un oggetto senza side efficiente
- **Modificatori** che modificano gli attributi (lo stato) dell'oggetto su cui sono chiamate

Un operazione può essere **static** e, come per il linguaggio Java, le operazioni non operano solo su una particolare istanza della classe ma sulla classe stessa. in UML si indica sottolineando il nome dell'operazione.

Un operazione viene invocata su un oggetto e corrisponde alla dichiarazione di una procedura/funzione, un metodo è il corpo di tale procedura/funzione, è l'implementazione di un operazione

Se un'operazione è **abstract** viene indicata con il nome in corsivo.

## 7.4 Datatype

In UML esiste il concetto di datatype, per sua natura un oggetto ha un identità mentre un datatype no: le istanze sono valori non oggetti. Si rappresentano come classi con lo stereotipo `<<datatype>>`

## 7.5 Associazioni

Un'associazione rappresenta una relazione tra classi

Il `>` indica in che direzione deve essere letta l'associazione, in alternativa si può indicare il ruolo di uno dei due estremi dell'associazione. In un diagramma delle classi non è consigliato aggiungere nomi delle associazioni e ruoli.

Il verso di navigazione indica in quale direzione è possibile reperire le informazioni ( $\rightarrow$ ).

### 7.5.1 Molteplicità

La molteplicità delle associazioni indica il numero di link tra gli oggetti delle classi e, come in SQL, si mettono sulla freccia ai due estremi.

### 7.5.2 Attributi

Di solito si usano attributi per tipi primitivi e datatype o associazioni se tipati da classi.

Un associazione riflessiva è quando una classe ha un'associazione con se stessa.

### 7.5.3 Implementazione (Java)

- Associazione molteplicità 1: `attributo`
- Associazione molteplicità 0..\*: `HashSet`
- Associazione molteplicità {ordered} 0..\*: `LinkedList` o `ArrayList`

Esistono anche le associazioni bidirezionali ma non possono essere implementate con un singolo attributo. Con le associazioni bidirezionali 1..1 c'è un problema di sincronizzazione.

## 7.6 Note

E' un commento in un linguaggio di programmazione, si usa anche per specificare il "behaviour" di un'operazione

## 7.7 Aggregazione

Rappresenta la relazione "tutto-parti", è una relazione di tipo "è composto da" e si rappresenta con un rombo vuoto.

## 7.8 Composizione

**Forma forte** di aggregazione: esprime la relazione "ha-un"/"è composto di", le proprietà sono:

- Se il composto/intero viene distrutto, anche le sue parti saranno distrutte
- Una parte può appartenere ad un solo oggetto intero alla volta

## 7.9 Generalizzazione ed ereditarietà

- **Generalizzazione = relazione "è un"**, ogni istanza di una classe è anche istanza della superclasse
- **Ereditarietà**: meccanismo attraverso il quale elementi specializzati incorporano la struttura ed il comportamento di elementi più generali. Nella classe erede è possibile aggiungere nuovi attributi e operazioni e ridefinire le operazioni (overriding)



# Capitolo 8

## UML - Sequence Diagram

### 8.1 Diagrammi di interazione

Descrivono la collaborazione di un gruppo di oggetti (ovvero i messaggi), i messaggi possono essere

- **Sincroni**: il mittente si pone in attesa del risultato
- **Asincroni**: il mittente continua l'esecuzione

### 8.2 Notazione

- **Messaggio trovato (input)**: si rappresenta come freccia continua che, nella coda, ha un pallino nero
- **Messaggio**: come per il messaggio trovato tranne che non c'è pallino
- **Controllo del partecipante**: si rappresenta come un rettangolo "verticale" sotto il partecipante, indica quando il partecipante ha il controllo del flusso

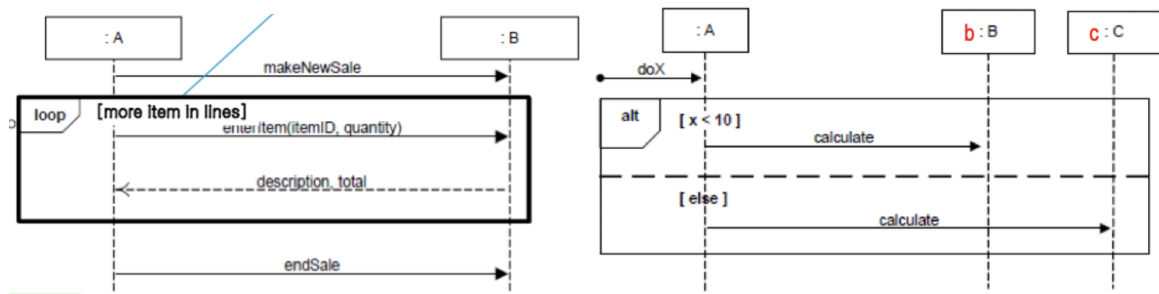
### 8.3 Sintassi dei messaggi

- **Self**: chiamata di un metodo appartenente alla classe stessa (è una freccia che parte e arriva alla stessa classe)
- **Creazione**: creazione di un oggetto (si rappresenta con una freccia tratteggiata per indicare la dipendenza da un altro oggetto)
- **Distruzione**: distruzione di un oggetto (si indica con un messaggio che finisce con una X)

### 8.4 Frame

#### 8.4.1 Cicli e condizioni

Cicli e condizioni (la logica di controllo) possono essere espressi mediante i frame di interazione, un frame è un rettangolo che racchiude un gruppo di messaggi.



Ovviamente i frame possono essere annidati.

## 8.4.2 Ref

Ref sta per **reference** è usato per semplificare i sequence diagram. Simile al concetto di "procedura" in un linguaggio di programmazione.

## 8.4.3 Par

Utile per esprimere l'interazione di oggetti complessi che possono eseguire diverse azioni/operazioni in parallelo.

I diagrammi di sequenza:

Pro	Contro
Ottimo mezzo per visualizzare l'interazione tra oggetti	Non buono per spiegare i dettagli degli algoritmi, come i cicli e i comportamenti condizionali

M. Fowler suggerisce di limitare l'uso dei frame di interazione perchè sono molto pesanti.

## 8.5 Sequence Diagram di Sistema (SSD)

Un diagramma di sequenza di sistema illustra gli eventi di input ed output del sistema relativi ad uno o più scenari di un caso d'uso.

Serve a stabilire con precisione quali sono gli eventi di input e di output del sistema e a rappresentare un punto di partenza per la progettazione delle classi e le loro operazioni e l'analisi dei contratti delle operazioni.

## 8.6 Communication Diagram

E' una variante del sequence diagram che mette in evidenza la comunicazione tra oggetti.

# Capitolo 9

## UML - State Machine & Activity Diagram

### 9.1 State Machine

Le state machine vengono usate per descrivere il comportamento di una Entità come variazione del suo stato interno quando è sottoposta a sollecitazioni dal mondo esterno (eventi)

Più semplicemente le state machine descrivono cosa fa un'entità quando riceve un input. Un'entità può comportarsi in modo diverso a seconda del suo stato interno.

Una state machine è associata a una classe.

#### 9.1.1 Stato

Rappresenta una situazione durante la quale delle condizioni vengono soddisfatte e delle attività possono essere eseguite.

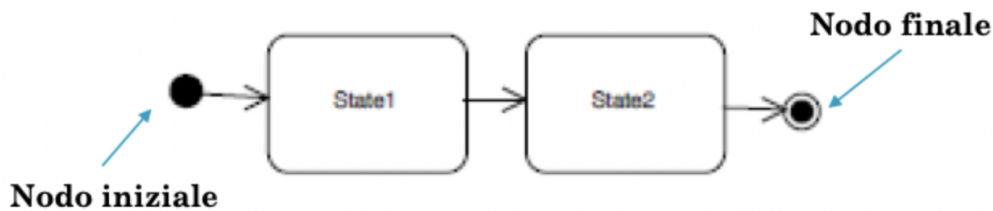
Gli stati di una macchina a stati UML sono qualcosa di più astratto: ad ognuno di essi corrisponde un diverso comportamento del sistema al verificarsi degli eventi.

#### 9.1.2 Rappresentazione grafica

Graficamente le state machine UML sono dei grafi dove i nodi sono gli stati e gli archi sono le transizioni. In questi grafi lo stato iniziale è rappresentato da un pallino nero che si collega a uno stato iniziale, sugli archi c'è scritto il nome dell'evento e se vicino al nome ci sono delle parentesi quadre dentro c'è la guardia.

#### Semantica

- Una macchina a stati è sempre associata ad un'entità e ne descrive il suo comportamento
- La macchina a stati riceve eventi che vengono salvati su una coda ed estratti uno alla volta
- **Run-to-completion:** un evento viene estratto solo dopo che il precedente è stato processato
- Se ci sono più transizioni eseguibili in un dato momento solo una viene eseguita in modo non deterministico
- Un disco nero marca il nodo d'inizio dell'esecuzione, è unico per ogni diagramma di stato
- Un disco nero bordato marca il nodo finale e possono comparire in qualunque numero all'interno di un diagramma



## Transizioni

Ogni transizione, oltre allo stato origine e destinazione **può** specificare:

- **Evento:** un 'trigger' che attiva il passaggio di stato, qualcosa che l'entità subisce
- **Guardia:** una condizione che, se vera, permette il passaggio di stato. Può essere espressa in linguaggio naturale, pseudo-linguaggio, linguaggio di programmazione o in OCL
- **Attività:** una o più azioni che sono compiute dall'entità prima di cambiare stato, possono essere espresse in modo informale (linguaggio naturale) o sotto forma di pseudolinguaggio

Una transizione avviene come risposta ad un evento, e al momento della transizione l'entità esegue l'attività specificata.

### 9.1.3 Altri tipi di eventi in UML

- **when** (evento di cambiamento): si verifica quando una condizione passa da falsa a vera
- **After, At** (evento temporale): si verifica dopo un certo tempo o ad una data/ora "precisa"

## Attività interne

Uno stato può reagire ad eventi e compiere attività anche senza una transizione ad uno stato diverso. Queste attività sono:

- **entry:** eseguita quando l'oggetto entra nello stato
- **exit:** eseguita quando l'oggetto esce dallo stato
- **do-activity:** eseguita mentre l'oggetto è nello stato, può durare per un intervallo di tempo ed essere interrotta da altri eventi

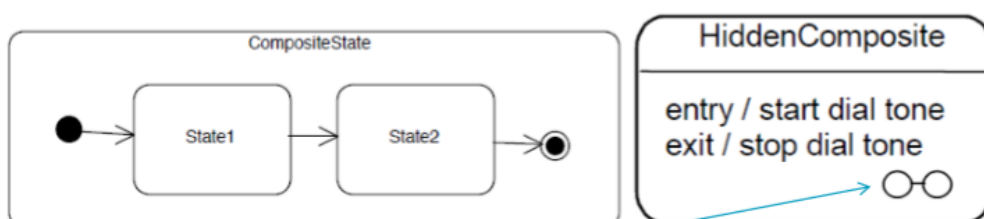
Una **self-transition** attiva sempre le entry ed exit, le attività interne no.

Un attività interna di solito si scrive:

evento/azione eseguita

### 9.1.4 Stati composti (superstati)

Uno stato composto permette di suddividere la complessità del modello, si può usare un'icona per rappresentare uno stato composto il cui comportamento interno non è mostrato:



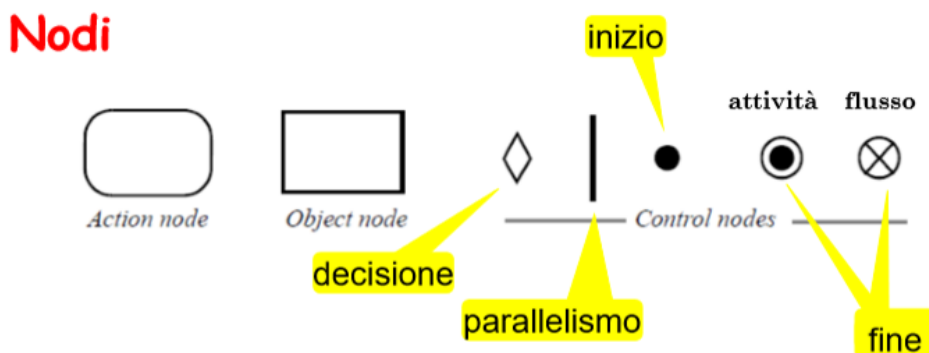
## 9.2 Activity Diagram

Descrivono come viene svolta un'attività relativa ad una qualsiasi entità, quale è il flusso di azioni che devono accadere

### 9.2.1 Attività

Un'attività è costituita da un flusso di azioni che ne sono i mattoni.

Costrutti



I flussi vengono rappresentati con frecce.

- **Nodi azione**: specificano unità di comportamento, può essere espressa in modo informale, può invocare un'altra attività, può invocare un'operazione di una classe e, volendo, si può scrivere anche un frammento di codice all'interno del simbolo di azione. A volte si può trovare il simbolo  $\text{m}$  (rake) che indica che un'azione è specificata a parte da un altro diagramma di attività.
- **Nodi oggetto**: specificano oggetti usati come input e output di azioni.
- **Nodi di controllo**: specificano il flusso delle attività.

Un nodo azione viene eseguito quando sono presenti i token su tutti gli archi in entrata, un token può essere rappresentato come un cerchio colorato.

### 9.2.2 Nodi inizio e fine

L'utilizzo dei nodi inizio e fine attività è uguale a quello dei nodi inizio e fine di stato: il nodo inizio (cerchio nero) è unico e il nodo fine (cerchio nero bordato) può essere ripetuto più volte.

Esiste anche il nodo finale **di flusso** che è rappresentato da un cerchio con una X al centro e causa la terminazione del flusso non dell'attività però al raggiungimento di un nodo finale di attività causa comunque la terminazione di tutti i flussi.

### 9.2.3 Nodi decisione e fusione

Si possono rappresentare entrambi come rombi, i nodi decisione hanno un input e vari output mentre il nodo fusione ha vari input e un output.

### 9.2.4 Nodi fork e join

I nodi fork e join si possono rappresentare come rettangoli neri, il nodo fork divide un'esecuzione in più flussi concorrenti, il nodo join sincronizza e riunisce i flussi.

# Capitolo 10

## UML - Component, Package & Deployment Diagram

### 10.1 Component Diagram

Una scatola nera il cui comportamento esterno è completamente definito dalle sue interfacce, sono rimpiazzabili e componibili

#### 10.1.1 Componente

I componenti sono entità logiche che sono realizzate da artefatti (che sono invece entità fisiche).

**Esempio:**


[Sottosistema](#) realizzato tramite un **insieme di classi**

Dove il sottosistema è l'entità logica e l'insieme di classi l'entità fisica.

**Perchè si usano le componenti per descrivere l'architettura di un sistema SW?**

Le classi sono componenti di grana troppo fine per dare una buona panoramica del sistema.

Un componente è mostrato come rettangolo con:

- Parola chiave <<component>> oppure l'icona 
- Nome

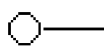
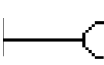
Esistono parole chiave più specifiche come:

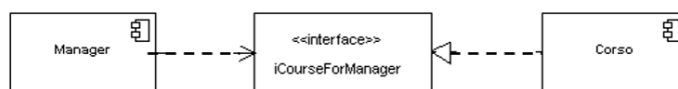
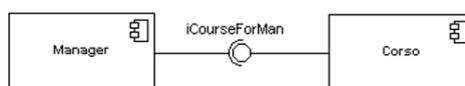
- <<Web service>>, <<Microservice>>, <<Subsystem>>, ...

Se ne possono sempre aggiungere

#### 10.1.2 Interfacce

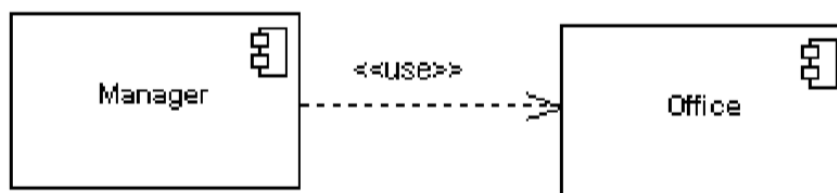
Sono composte da:

- Lollipop (fornite) 
- Socket (richieste) 

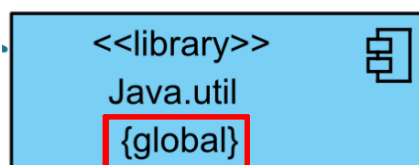


### 10.1.3 Dipendenza

I componenti possono essere connessi anche con la relazione di dipendenza (una freccia tratteggiata con scritto `<<use>>`), spesso usato quando sono utilizzate più interfacce ma non si vuole specificare quali sono.



Se in un componente c'è scritto `{global}` vuol dire che è usata da tutti i componenti.



## 10.2 Deployment Diagram

E' una vista molto implementativa che mostra la relazione tra hardware e software in un sistema SW.  
E' presente un forte link tra component e deployment diagram.

### 10.2.1 Nodi e connessioni

- **Nodi:** rappresenta un tipo di risorsa computazionale:
  - **Periferica fisica** (`<<device>>`)
  - **Ambiente software di esecuzione**

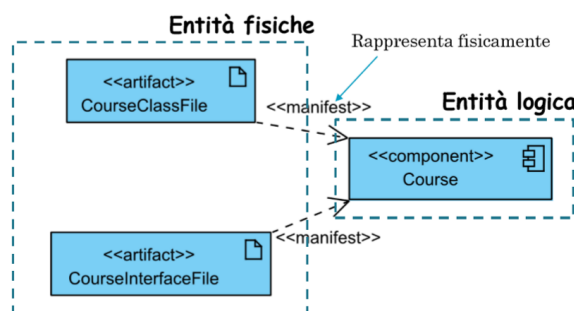
Una connessione tra nodi rappresenta un canale di comunicazione attraverso cui possono passare delle informazioni (es. TCP/IP).

### 10.2.2 Artefatti

Sono entità concrete del mondo reale (file di codice sorgente, script, ...), possono essere dislocati sui nodi. Esiste anche la relazione di dipendenza tra artefatti.

### 10.2.3 Manifest

La relazione **manifest** indica che gli artefatti sono rappresentazioni/manifestazioni fisiche dei componenti.



## 10.3 Package Diagram

Descrivono i package e le dipendenze

### 10.3.1 Package

Un package (in UML) è un costrutto che permette di prendere un numero arbitrario di elementi UML e raggrupparli insieme, può contenere sia elementi che sotto-package.

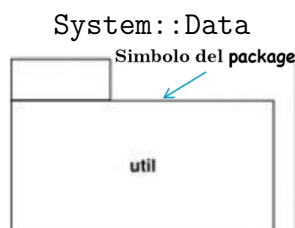
**Come suddividere le classi in package?**

Seguendo i principi di buona progettazione:

- High cohesion and low coupling
- Common Reuse Principle: le classi in un package dovrebbero essere sempre riusate assieme
- Acyclic Dependency Principle: no cicli
- ...

### Namespace

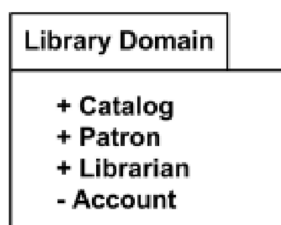
Ogni package definisce un namespace, lo scopo dei namespace è quello di evitare confusione ed equivoci nel caso siano necessarie molte entità con nomi simili in un progetto SW. Per indicare la classe a cui mi sto riferendo occorre usare un nome completamente qualificato, ad esempio:



### 10.3.2 Visibilità

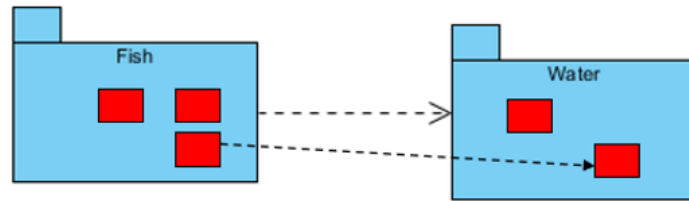
Gli elementi contenuti in un package possono avere una visibilità che indica se gli elemento sono visibili o meno ai clienti del package:

- +: pubblico
- -: privato





### 10.3.3 Dipendenze



In questo caso **Fish** dipende da **Water**: vuol dire che nel package **Fish** esiste **almeno** una classe che dipende da una classe che è nel package **Water**. La classe in **Water** deve essere visibile.

### 10.3.4 In pratica

- Progettare l'applicazione cercando di seguire il più possibile i principi di buona progettazione
- Generare il diagramma dei package a partire dal codice stesso usando tool specifici di reverse engineering
- Identificare cicli, package poco coesi e package con tante dipendenze entranti per ristrutturare il sistema

# Capitolo 11

## Software Design Patterns

E' una soluzione elegante (con un nome) di uno specifico problema di design/programmazione OO che costituisce un unità di riuso. L'idea di design pattern deriva da Kent Beck. Attualmente molti cataloghi di pattern sono in genere relativi a un particolare dominio applicativo o a una particolare tipologia di applicazioni.

I design pattern aiutano ad applicare i principi di buona progettazione OO, possono essere applicati sia durante la modellazione che la codifica e facilitano la comunicazione tra sviluppatori.

### 11.1 Altre unità di riuso

- Librerie
- Componenti riusabili (COTS)
- Framework: è un insieme di classi e interfacce cooperanti che realizzano un design riusabile e customizzabile per uno specifico dominio applicativo o tipologia di app, la differenza con la libreria è che lo sviluppatore di un'applicazione scrive le classi (in particolare le operazioni) che vengono chiamate dal framework mentre con una libreria lo sviluppatore chiama le operazioni delle classi di una libreria
- Product line
- Modelli

I design pattern, rispetto ai framework, sono più astratti, più piccoli, di solito sono meno specializzati: non relativi a un dominio applicativo specifico o tipologia di app.

### 11.2 GRASP

I pattern elementari GRASP sono pattern per l'assegnazione di responsabilità nel software, costituiscono il fondamento per la progettazione di sistemi OO.

#### 11.2.1 Pattern Controller

**Problema:** quale è il primo oggetto oltre lo strato di UI che riceve e coordina un operazione di sistema (i messaggi)?

**Soluzione:** assegna la responsabilità ad una classe che rappresenta una delle seguenti scelte:

- Sistema piccolo: classe che rappresenta il sistema complessivo
- Sistema grande: classe relativa al caso d'uso all'interno del quale si verifica l'evento, solitamente chiamata **Handler** o **Controller**

## 11.3 Design pattern classici

- Creazionali: creazione degli oggetti
- Strutturali: composizione di classi
- Comportamentali: come classi (oggetti) interagiscono tra di loro e si distribuiscono le responsabilità

Un template è piuttosto preciso, noi vedremo la versione semplificata:

- Nome (una/due parole)
- Problema: descrive quando applicare il pattern e può contenere anche delle precondizioni per poterlo applicare
- Soluzione
- Conseguenze: pro/contro della sua applicazione

### 11.3.1 I pattern creazionali

Sono pattern che hanno a che fare con la creazione di istanze
---

Di solito per risolvere questo problema ci si affida al pattern GRASP Creator, tuttavia esistono casi in cui non conviene applicare il pattern Creator.

#### Factory

Quando si vuole separare la logica di creazione dalla logica applicativa pura si ricorre al concetto di Factory in cui viene definito un oggetto specifico Factory con l'unico scopo di creare oggetti.

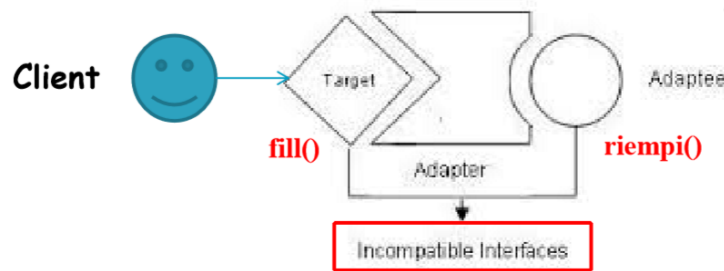
**Abstract Factory** fornisce una soluzione per creare famiglie di prodotti in modo che non ci sia necessità da parte dei client di specificare le classi concrete dei prodotti all'interno del proprio codice. E' utile in quanto separa le responsabilità, si produce un design (quindi un codice) migliore.

Conseguenze:

Pro	Contro
Isola le classi concrete	Non è facile aggiungere nuovi prodotti: può richiedere cambiamenti all'interfaccia dell'Abstract Factory e alle sue sottoclassi, richiede l'aggiunta di una nuova classe ConcreteFactory e di nuovi Abstract-Products e Products
Rende il cambio della famiglia dei prodotti facile	
Favorisce la consistenza tra i vari prodotti di una famiglia	

## 11.4 Adapter

Converte l'interfaccia di una classe in un'altra interfaccia che il cliente si aspetta, permette a delle classi di lavorare assieme anche se non potrebbero visto che hanno interfacce incompatibili.



**Problema:** usare una classe esistente, la cui interfaccia non è quella che il cliente si aspetta, si vuole creare una classe che collabora con classi non correlate o che non si conoscono ancora.

## 11.5 Façade

**Problema:** rendere più semplice l'accesso a sottosistemi che espongono interfacce complesse, fornire un'unica interfaccia per un insieme di funzionalità "sparse" su più interfacce/classi.

Conseguenze:

- Promuove un accoppiamento debole fra cliente e sottosistema
- Nasce al cliente le componenti (complesse) del sottosistema
- Il cliente può comunque, se necessario, usare direttamente le classi del sottosistema

La differenza con l'adapter è che quest'ultimo la converte un'interfaccia mentre il façade la semplifica.

## 11.6 Template Method

Definisce lo scheletro (template) di un algoritmo in un metodo posponendo la definizione di alcuni passi a delle sottoclassi.

**Problema:** implementare la parte invariante di un algoritmo una sola volta, e lasciare alle sottoclassi l'implementazione delle parti che possono variare.

Le operazioni che costituiscono la parte variabile sono chiamate **primitive**. Normalmente sono le sottoclassi a chiamare i metodi delle superclassi, con questo pattern è il metodo del template a chiamare i metodi specifici ridefiniti nelle sottoclassi.

Conseguenze:

Pro	Contro
Tecnica fondamentale per il riuso del codice	Importante chiarire bene quali operazioni devono essere ridefinite nelle sottoclassi
Realizza inversione del flusso di controllo	
Permette di avere anche più sottoclassi concrete	

## 11.7 Observer

Gli osservatori sono oggetti che catturano e informano gli altri osservatori di un cambiamento di stato di un oggetto osservato.

Gli osservatori si registrano presso l'oggetto osservato, quando l'oggetto osservato cambia stato (`cambiato() : boolean`), notifica tutti gli osservatori (`notify()`) e, quando notificato, ogni osservatore decide cosa fare (niente o `getState()`).

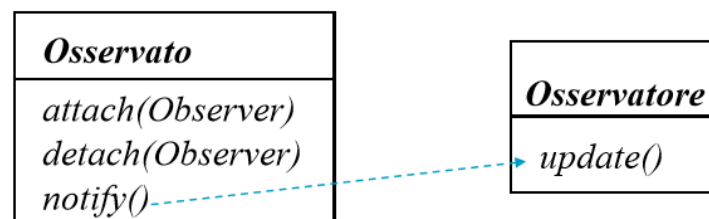
Definisce una dipendenza allentata (larga) uno a molti tra oggetti, in modo che quando un oggetto cambia stato, tutti gli oggetti che dipendono da lui sono avvertiti.

**Problema:**

- Associare più "viste" differenti ad un modello (dati)
- Implementare il broadcast
- Il cambiamento di un oggetto richiede il cambiamento di altri oggetti
- Notificare oggetti senza fare assunzioni a priori su quali siano questi oggetti

### 11.7.1 Interfacce

L'oggetto osservato deve fornire un'interfaccia standard per la registrazione, deve anche poter notificare con `notify()` che chiama tutti gli `update()` dei registrati e gli osservatori devono fornire un'interfaccia standard per la notifica `update()` che a sua volta chiamerà `getState()` dell'osservato.



### 11.7.2 Conseguenze

Pro	Contro
Supporto per la comunicazione broadcast	Modifiche inaspettate
Collegamento tra osservato e osservatore modificabile	

## 11.8 Model View Controller (MVC)

E' un pattern architetturale (quindi avremo una granularità maggiore rispetto ai design pattern) che divide un'applicazione con GUI in tre tipologie di componenti:

- Un modello: contiene i dati e le funzionalità di base
- Una o più viste: mostrano informazioni agli utenti
- Uno o più controller: gestiscono le richieste degli utenti

Un'interfaccia utente è formata da una (o più) vista e un (o più) controller.

## 11.9 State Pattern

**Scopo:** permettere a un oggetto di cambiare il suo comportamento al variare del suo stato interno. Questo permette di implementare le state machine UML.

**Come?** Si estrae la rappresentazione dello stato in classi esterne organizzate in una gerarchia, si sfrutta il polimorfismo per variare il comportamento.

Però è difficile da comprendere e mantenere (praticamente bisogna aggiungere un `if` per ogni metodo evento e ovviamente avere tanti `if` non è buono), un'alternativa è usare lo State Pattern: si usa una gerarchia di classi per rappresentare gli stati della macchina.

# Capitolo 12

## Code Refactoring

Un cambiamento eseguito alla struttura interna di un software per renderla più facile da capire è più economica da modificare senza cambiare il suo comportamento osservabile.

**Martin Fowler**

Refactoring: è una riorganizzazione, ristrutturazione del codice (di solito OO) senza modificarne il comportamento.

### 12.1 Legacy System

Sistemi per i quali l'attività di manutenzione è diventata prevalente su ogni altra.

**Problemi:**

- Sono stati implementati diversi anni fa
- La loro tecnologia è diventata obsoleta
- Sono stati mantenuti per un lungo periodo
- La loro struttura si è deteriorata e non è facile comprendere il codice
- La loro documentazione (se esiste) non è allineata
- Gli autori originali non sono più disponibili
- **Contengono 'regole di business' che non sono documentate altrove**
- **Non possono essere sostituiti facilmente**
- **Rappresentano un grosso investimento per l'azienda**

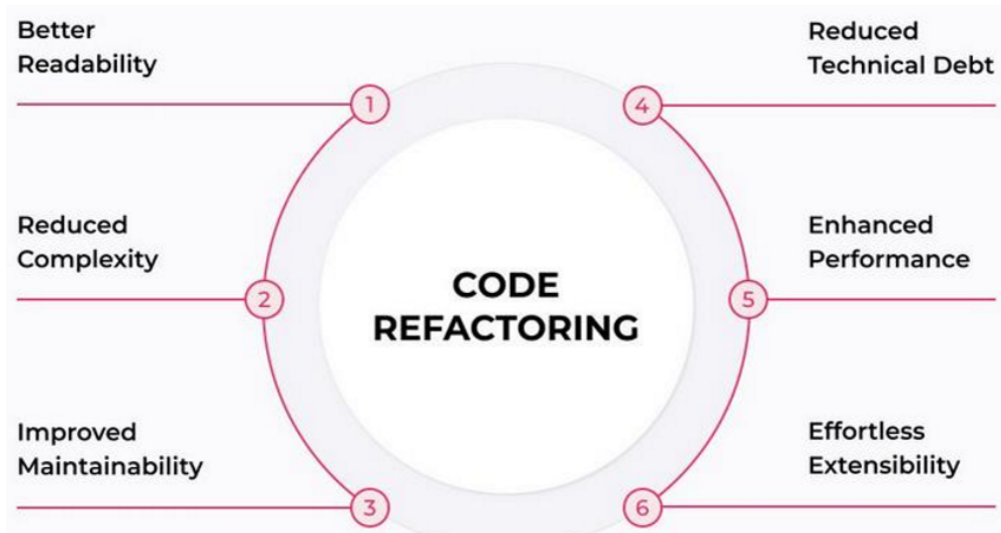
#### 12.1.1 Convivere con un Legacy System

**Obiettivo:** migliorare la qualità del SW e contenere i costi.

**Approcci principali:**

- Redocumentation: processo che mira a produrre una vista del codice 'alternativa' utile per la comprensione
- Restructuring/Refactoring: trasformazione di codice 'mal-strutturato' in codice 'ben-strutturato'
- Reverse Engineering: creazione del design e delle specifiche a partire dal codice, esiste anche la definizione forte che prevede (oltre alla rappresentazione grafica) l'inferire i requisiti/specifiche
- Re-engineering: reverse engineering + modifica di specifiche e design + forward engineering (creazione di un nuovo sistema basato su specifiche e design rivisitati)

## 12.2 Perché?



Ma anche:

- Semplificare la fase di testing
- Limitare design decay/erosion

## 12.3 Quando?

Applicare il refactoring il più spesso possibile durante lo sviluppo

- Si vuole aggiungere una nuova funzionalità al sistema
- Quando si corregge un bug
- Quando viene rilevato un **code smell**

### 12.3.1 Code Smell

Indicatore che qualcosa nel codice non va bene, potrebbe essere solo un qualcosa di **stile**, oppure che riduce la **comprensione** del codice oppure nascosto c'è un problema più grave. Spesso i tool che calcolano le metriche del software ci indicano quali sono i code smell presenti.

Esempi:

- 'Troppo' codice come ad esempio metodi lunghi
- 'Non abbastanza' codice come ad esempio le **Data class** ovvero classi con solo campi e i metodi getter/setter
- Al di fuori del codice come ad esempio commenti inutili

### 12.3.2 Clone Software

E' codice duplicato e questo può causare 'bug propagation' e problemi di manutenzione.

## 12.4 Il ritmo

1. Trova/identifica code smell
2. Modifica (piccola) del codice seguendo una procedura definita in modo preciso dal catalogo dei refactorings
3. Compila
4. Esegui i test

Ripeti per ogni code smell trovato nel codice.

### 12.4.1 Catalogo dei Refactorings

Molti refactoring sono "piccoli e semplici" (low-level refactoring) però sono i building-blocks per i refactoring complessi.

Smell	Refactoring
<b>Long Method:</b> In their description of this smell, Fowler and Beck explain several good reasons why short methods are superior to long methods. A principal reason involves the sharing of logic. Two long methods may very well contain duplicated code. Yet if you break those methods into smaller methods, you can often find ways for the two to share logic. Fowler and Beck also describe how small methods help explain code. If you don't understand what a chunk of code does and you extract that code to a small, well-named method, it will be easier to understand the original code. Systems that have a majority of small methods tend to be easier to extend and maintain because they're easier to understand and contain less duplication. [F 76, K 40]	Extract Method [F 110]
	Compose Method [K 123]
	Introduce Parameter Object [F 295]
	Move Accumulation to Collecting Parameter [K 313]
	Move Accumulation to Visitor [K 320]
	Decompose Conditional [F 238]
	Preserve Whole Object [F 288]
	Replace Conditional Dispatcher with Command [K 191]
	Replace Conditional Logic with Strategy [K 129]
	Replace Method with Method Object [F 135]
<b>Long Parameter List:</b> Long lists of parameters in a method, though common in procedural code, are difficult to understand and likely to be volatile. Consider which objects this method really needs to do its job - it's okay to make the method to do some work to track down the data it needs. [F 78]	Replace Temp with Query [F 120]
	Replace Parameter with Method [F 292]
	Introduce Parameter Object [F 295]
<b>Many Small Methods:</b> Many small methods are a sign of a poorly designed system. They are often the result of a lack of understanding of the problem domain. They are often the result of a lack of understanding of the problem domain. They are often the result of a lack of understanding of the problem domain. [F 78]	Preserve Whole Object [F 288]
	Introduce Parameter Object [F 295]

## 12.5 Come?

- Eseguire manualmente i refactoring
- Utilizzare un tool di supporto

Chiaramente usare un tool è la soluzione più comoda, tuttavia spesso i tool forniscono solo refactoring semplici. In entrambi i casi è meglio sempre controllare ri-eseguendo i casi di test.

### 12.5.1 Tools

Gli IDE di solito hanno incorporato alcuni refactoring semplici, esistono anche tool o plug-in specifici che suggeriscono refactoring più complessi.

#### Move method

Si applica quando le classi hanno 'troppo behaviour' oppure quando abbiamo classi che collaborano troppo o sono troppo accoppiate. E' un refactoring facile, il difficile però è trovare i metodi 'giusti' da spostare, dei buoni candidati sono metodi che sembrano riferirsi più ad altre classi che alla classe a cui appartengono.



## **Replace Temp with Query**

**Contesto:** le variabili locali possono essere viste solo nel contesto di un metodo così incoraggiano ad avere metodi lunghi.

**Soluzione:** rimpiazzando le variabili locali con un metodo query, ogni metodo della classe può ottenere quell'informazione.

## **Replace Parameter with Method**

**Contesto:** metodi che hanno molti parametri sono difficili da capire e la lista dei parametri dovrebbe essere ridotta il più possibile.

**Soluzione:** se un metodo può ottenere un valore che gli è stato passato allora dovrebbe fare quel calcolo per ottenerlo direttamente al suo interno.

## **Extract Class**

**Contesto:** esiste una classe che fa troppo (God class o Blob class)

**Soluzione:** si crea una nuova classe e si sposta in questa alcuni attributi e operazioni, se non si vuole modificare l'interfaccia lasciamo le operazioni "che delegano" alla nuova classe tutto il lavoro.

## **Replace Inheritance with Delegation**

Una sottoclasse usa solo parte di una superclasse e non vuole ereditare il resto

## **Replace Conditional with Polymorphism**

Esiste una condizione che sceglie differenti comportamenti a seconda del tipo/valore di una variabile

## **Separate Domain from Presentation**

Abbiamo una GUI che contiene anche la Business Logic, non c'è "separation of concerns"

# Capitolo 13

## Software Testing

E' una procedura sistematica che prevede l'esecuzione di un sistema software (SUT) con l'intento di trovare **failure**

- **Errore**: commesso da uno sviluppatore, può essere di battitura o concettuale
- **Fault**: l'errore presente all'interno del programma
- **Failure**: manifestazione di una fault durante l'esecuzione

### 13.1 Debugging

Il processo usato per trovare un bug/fault a partire dal failure e rimuoverlo

Due fasi:

- Fault localization/location
- Fault removal

### 13.2 Testing

E' un processo esaustivo e "non realizzabile" nei casi reali perchè occorre selezionare "pochi e buoni" (buoni = con alta probabilità di trovare un fault) input nel dominio di tutti gli input possibili. Gli input vengono selezionati secondo l'approccio che si decide di seguire:

- **Black box**: non è basato su una conoscenza del SUT e della sua struttura, di solito gli input sono generati a partire dai requisiti/specifiche. Viene chiamato anche **functional testing**
- **White box**: è basato su una conoscenza esplicita del SUT e della sua struttura, viene chiamato anche **structural testing**

### 13.3 Testcase & Testsuite

- **Testcase** (caso di test): un insieme di input, precondizioni e risultati attesi sviluppati per un particolare obiettivo come eseguire un particolare percorso del programma o per verificare la conformità di un requisito specifico
- **Testsuite**: una collezione di testcase

## 13.4 Tipologie di testing

- **Testing di unità** (Implementazione): l'unità viene testata dallo sviluppatore isolandola il più possibile dal resto, di solito approccio white-box
- **Testing di integrazione** (Implementazione/Integrazione): i moduli/componenti sono "testati insieme", viene testato come comunicano i moduli/componenti e quali info vengono scambiate. Le interfacce tra moduli di solito sono fonti di errori
- **Testing di sistema** (Testing di Sistema): il "testing group" verifica che il software soddisfa i requisiti e anche i bisogni dall'utente (acceptance testing), di solito black-box
- **Testing di regressione** (Manutenzione): tutte le volte che si effettua una modifica in un applicazione c'è il rischio di side effect in zone del codice che apparentemente non sembrerebbero impattate dalla modifica, lo scopo del test di regressione è verificare che non ci siano state delle regressioni (side effect/peggioramenti)

## 13.5 Testing manuale VS Testing automatizzato

Nel testing automatizzato i tester implementano dei test script (porzioni di codice in grado di interagire con il SUT, per esempio inserendo input) che sono eseguiti da un framework che dopo la loro esecuzione riporta il risultato (pass/fail).

### 13.5.1 JUnit

E' un framework di testing per programmi Java che ci permette di realizzare il test automatizzato. Sviluppato da: Erich Gamma e Kent Beck.

## 13.6 White-box Testing

### 13.6.1 Code coverage

Di solito la qualità di una testsuite  $T$  si valuta misurando la copertura di  $T$  rispetto al programma  $P$ . Ad esempio il numero di linee di codice eseguite durante la fase di testing sul totale di linee di codice. La coverage può essere basata sul codice sorgente o su modelli generati a partire dal codice (es. grafo di flusso di controllo, state machines, data flow graph, ...) o requisiti e specifiche.

#### Control Flow Graph (CFG)

Rappresenta mediante un diagramma di flusso tutti i possibili cammini che possono essere attraversati durante l'esecuzione di  $P$ . **Esistono diverse rappresentazione (e numerazioni) equivalenti di CFG.**

- Statement (node) coverage: coprire tutti i nodi del CFG
- Branch coverage: coprire tutti i branch anche chiamato decision coverage, è la copertura minima richiesta da "IEEE unit test standard"
- Multiple Condition Coverage (MCC): copre tutte le combinazioni possibili delle condizioni nei nodi decisionali
- All Paths Coverage: copertura di tutti i cammini del CFG in pratica è impossibile (ci sono i loop)

## In pratica

1. Si sceglie un criterio:
  - Seguendo gli standard o le prescrizioni aziendali (di solito branch coverage)
  - Considerando che, in generale:
    - più è la power (livello di difficoltà di implementazione) più è complesso trovare i casi di test
    - più è la power più aumenta la dimensione della testsuite
    - più è la power più aumentano i costi e i tempi di esecuzione
    - più è la power più è probabile che la testsuite riveli dei fault
2. Si sceglie una copertura
3. Si crea una testsuite seguendo il criterio scelto con tanti casi di test al fine di raggiungere la copertura voluta...

## Statement Coverage

**Criterio:** tutti gli statement devono essere coperti durante l'esecuzione della testsuite.  
E' il criterio più debole di copertura.

**Procedura:**

1. Selezionare alcuni path che coprono tutti gli statement
2. Scegliere gli input in modo da percorrere i path selezionati

## Branch Coverage

**Criterio:** tutti i branch devono essere coperti durante l'esecuzione della testsuite.  
I branch **true** e **false** degli if statement, ogni **case** in uno statement **switch** e ogni loop.

**Procedura:**

1. Selezionare alcuni path che coprono tutti i branch
2. Scegliere gli input in modo da percorrere i path selezionati

## Multiple Condition Coverage

**Criterio:** ogni condizione atomica deve essere coperta (occorrono degli input che la rendono true e false), in una condizione composta ogni combinazione delle condizioni atomiche deve essere coperta durante l'esecuzione dei casi di test.

## All Paths Coverage

**Criterio:** tutti i cammini del CFG devono essere coperti

Non applicabile!
------------------

## Problemi

- Non usabile in pratica quando ci sono dei loop (troppi cammini), di solito si semplifica trattando i loop in modo binario: viene eseguito o no
- Alcuni cammini potrebbero essere **infeasible**, potrebbe essere che non esiste una combinazione di input che ci permetta di scegliere un particolare path, questa cosa è vera anche per gli altri criteri ma è molto più complicato

## 13.6.2 Limiti

Spesso il numero di casi di test prodotti è molto grande, per questo motivo è spesso usato solo per Unit testing per piccole porzioni del sistema (Testing in the small) e non è in grado di rivelare i fallimenti dovuti a "missing feature errors" (possono essere scoperti solo considerando i requisiti/specifiche)

## 13.7 Black-box Testing

Il black box testing può essere usato con qualsiasi tipo di sistema indipendentemente dalla tecnologia/piattaforma/linguaggio usato, può essere usato per Unit, Integration e System testing (usato soprattutto per System testing). Gli sviluppatori possono scrivere casi di test non appena i requisiti del sistema sono disponibili cioè prima che il codice venga scritto.

### 13.7.1 Equivalence Partitioning

1. Partizionamento dei dati di input
2. L'input domain è suddiviso in classi tale che il risultato risulta essere una partizione
3. Le classi sono create assumendo che il SUT esibirà lo stesso behaviour su tutti gli elementi della classe
4. A questo punto si sceglie un input per ogni classe e si testa il programma (l'input scelto è rappresentativo della classe)

Di solito il primo passo è quello di dividere l'input domain in due classi:

- Input legali
- Input illegali

A loro volta queste classi possono ancora essere suddivise.

## 13.8 Boundary Value Analysis (BVA)

Si scelgono i casi di test in prossimità della frontiera (confini) delle varie classi di equivalenza (si applica dopo Equivalence Partitioning). Si basa sul seguente assunto: è più probabile commettere errori vicino alla frontiera che non all'interno delle classi.

1. Partizionare l'input domain (Equivalence Partitioning)
2. Identificare i confini per ogni partizione
3. Selezionare gli input in modo tale da comprendere i confini (e i punti vicini)

# Capitolo 14

## Metodi Agili: Extreme Programming

### 14.1 Critiche ai metodi plan-driven

In precedenza abbiamo parlato dei metodi plan-driven, però questi metodi non sono adatti per l'internet economy perchè richiede flessibilità e velocità di rispondere ai cambiamenti e lunghi cicli di sviluppo e tanta pianificazione non aiutano.

### 14.2 Agile Manifesto

*Kent Beck, Alistair Cockburn, Ward Cunningham, Martin Fowler, Ron Jeffries e Rober C. Martin*

Sono i nomi (più importanti) che hanno contribuito alla stesura dell'Agile Manifesto. I principi su cui si basa una metodologia agile che segue i punti indicati dall'Agile Manifesto sono:

Individuals and interactions	over	Process and tools
Working software	over	Comprehensive documentation
Customer collaboration	over	Contract negotiation
Responding to change	over	Following a plan

### 14.3 Extreme Programming (XP)

#### 14.3.1 Cosa è

XP è un processo di sviluppo per software OO ideato per piccoli gruppi (4-20) che cerca di mantenersi agile e flessibile. Non usa linguaggi di analisi o design (UML), ma dalla raccolta dei requisiti eseguita con le **User Stories**, passa velocemente alla codifica.

XP è stata ideata nel 1999/2000 da Kkent Beck, Ward Cunningham e Ron Jeffries e messo in pratica durante il progetto C3 (Chrysler Comprehensive Compensation System).

XP prevede frequenti rilasci di codice funzionante ogni 1-4 settimane invece di un rilascio unico.

### 14.3.2 Requisiti

I requisiti sono raccolti avendo a disposizione il cliente (**On-site Customer**) per tutta la durata del progetto, tramite le User Stories. Una User Story è un sintetico caso d'uso che viene scritto in linguaggio naturale.

### 14.3.3 Come si produce il desing?

- Metodo CRC (Class Responsibility Collaboration): si usano foglietti adesivi su cui vengono scritti il nome della classe, le responsabilità e le collaborazioni con le altre classi
- Quick design session: non si usano tool per il design e nemmeno linguaggi tipo UML, solo in casi particolari si usano degli sketch di Class Diagram (per esempio)

I documenti di design (i diagrammi) non vengono conservati, questi sono incorporati nel software e facilmente desumibili. Per avere sempre ben chiaro il design è importante che il codice sia il più possibile autodocumentato e chiaro possibile (meaningful code), questo non si ottiene con i commenti ma tramite il refactoring e assegnando alle variabili nomi significativi.

### 14.3.4 Principi di codifica

- Non si pianifica per il riuso: simple code and simple design (principio KISS)
- "Think by example": si inizia con un esempio concreto, si scrive un algoritmo ad hoc e poi si generalizza e si cercano i casi particolari
- Il codice è in comune: chiunque può modificare il codice scritto da altri (collective code ownership)
- Tutti i programmatori devono attenersi alle stesse "code conventions"

### 14.3.5 Pair Programming

I programmatori lavorano a coppie: uno digita il codice e l'altro osserva e commenta, solitamente il meno esperto scrive il codice. I punti di forza del Pair Programming sono:

- Revisione del codice continua
- Training
- Ogni modulo è conosciuto in dettaglio da almeno due persone

### 14.3.6 Unit Testing

- Per ogni classe sono scritti dei casi di test per verificarne il funzionamento
- Si usano tool/framework specifici (es. JUnit)
- Ad ogni modifica della classe tutti i casi di test sono eseguiti e devono essere tutti superati
- I casi di test sono progettati prima del codice stesso perchè aiutano nello sviluppo

### 14.3.7 Refactoring

- Per mantenere il codice semplice e meaningful occorre ristrutturarlo spesso
- Appena il codice supera i casi di test di unità si applica il refactoring
- Once and Only Once: se due segmenti di codice fanno la stessa cosa (cloni), devono essere unificati in un unico modulo
- La ristrutturazione continua è possibile perchè si dispone dei casi di test di unità

### 14.3.8 Acceptance Testing

Sono previsti dei test funzionali di accettazione basati sulle User Stories e concordati con il cliente: verificano che la user story è stata implementata correttamente.

La percentuale di test di accettazione superati è un indice del progresso nella codifica del sistema.

## 14.4 Problemi dei metodi agili

- Gestionali: la conoscenza è nella testa degli sviluppatori, si addice solo a sviluppatori molto bravi e di fatto gli customer on-site potrebbero non essere sempre disponibili
- Contrattuali/Legali: quando è soddisfatto il contratto? Di fatto non esiste...
- Manutenzione: documentazione scarsa, architettura spesso complessa determinata dalla release attuale

## 14.5 Metodi Plan-Driven VS Metodi Agili

Metodi Plan-Driven	Metodi Agili
Sistemi grandi e complessi, safety-critical o con forti richieste di affidabilità	Sistemi e team piccoli, clienti e utenti disponibili
Requisiti stabili e ambiente predicibile	Requisiti volatili e ambiente instabile
	Team con molta esperienza
	Tempi di consegna rapidi