Appunti Ingegneria del Software

Luca Seggiani
2025

1 Lezione del 22-09-25

1.1 Introduzione

Il corso di ingegneria del software si pone di fornire gli strumenti per partire dalle conoscenze tecniche relative allo sviluppo di software e arrivare allo sviluppo e alla manutenzione di **prodotti** veri e propri.

Cercheremo quindi di gestire **sistemi** *"ingegneristici"*, cioè sistemi su larga scala e stratificati, dove non possiamo usare le stesse tecniche che funzionavano su semplici prototipi per gestire ogni singola parte a sé stante. Di fondamentale importanza in questo aspetto è iniziare a vedere il software come un insieme *"vivo"* e **modulare** di più *componenti* fra di loro interconnessi, fra cui ognuno potrebbe, nel suo ciclo di vita, *evolversi* in maniera differente.

Approfondiremo quindi più **fasi** di sviluppo di un prodotto software, e le regole relative allo sviluppo di software che sia facile da documentare, aggiornare e mantenere.

In questo sfrutteremo in primis le tecniche della **programmazione ad oggetti** (**OOP**, *Object Oriented Programming*), cercando però di motivare nel contesto di un sistema più complesso l'uso di strumenti come *ereditarietà*, *polimorfismo*, ecc...

1.1.1 Interazione col cliente

Prerogativa dello sviluppo di prodotti software al giorno d'oggi è la sintonia col *cliente* che richiede il prodotto stesso, e quindi che definisce quelle che sono le **specifiche** di progetto. Diversi framework sono stati sviluppati per favorire la comunicazione fra cliente e sviluppatori (*Agile*, ecc...), e la figura del **software analyst**, cioè *analista software*, è diventata più centrale alla professione di ingegnere software di quanto lo è quella del semplice sviluppatore.

Questa situazione è dovuta al fatto che le specifiche di progetto non sono, nelle prime fasi di progettazione, chiare nemmeno al cliente. Il compito dell'ingegnere software è quindi in primo luogo **documentale**, cioè mirato a riassumere le volontà del cliente (espresse in linguaggio naturale) in una serie di specifiche e quindi una struttura di progetto (espressa in un linguaggio appositamente sviluppato per la modellazione, cioè l'*UML*).

2 Lezione del 26-09-25

2.1 Introduzione all'UML

L'**UML** (*Unified Modeling Language*) è una notazione grafica standardizzata, aperta ed estensibile, di modellazione. L'UML è pensato per modellizzare progetti che sfruttano la programmazione ad oggetti (OOP).

Viene detto *unificato* in quanto accompagna il software in tutti i suoi cicli di vita, dalle specifiche all'implementazione. Può essere usato per modellare diversi domini applicativi (dall'embedded alle applicazioni, ecc...), ed è trasparente al linguaggio e alle metodologie usate.

I modelli di UML rappresentano collezioni di **oggetti** *interagenti*. In particolare sfruttiamo modelli:

- A **struttura statica**, cioè che dettagliano quali oggetti compongono il nostro progetto in maniera statica;
- A **comportamento dinamico**, cioè dettagliamo successivamente come questi oggetti interagiscono fra loro nel tempo.

2.2 Introduzione all'UP

L'**UP** (*Unified Process*) è un processo di ingegnerizzazione software basato su 3 principi fondamentali:

1. Guidato dall'analisi dei *requisiti* e dei *rischi*;
2. Centrato sull'*architettura*, cioè finalizzato alla produzione di un'architettura robusta;
3. *Iterativo* ed *incrementale*, cioè suddivide il progetto in iterazioni incrementali che arrivano da zero al sistema funzionante.

Ciascuna iterazione è finalizzata a uno o più *sotto-obiettivi*:

- Pianificazione;
- Analisi e progetto;
- Costruzione;
- Integrazione e test;
- Release interna o esterna.

Ogni iterazione genera la cosiddetta *baseline*, cioè la versione da cui partira la prossima iterazione, e via dicendo. L'*incremento* sarà rappresentato dalla variazione da una baseline alla successiva.

Le fasi vengono implementate seguendo sostanzialmente uno o più fra 5 workflow riassunti dalla sigla **RADIT**:

- **Requirements** (requisiti);
- **Analysis** (analisi);
- **Design** (design);
- **Implementation** (implementazione);
- **Test** (test).

2.2.1 Struttura dell'UP

Il ciclo di vita del progetto si evolve in più iterazioni (ciascuna delle quali comprende i 5 workflow RADIT) . In particolare individuiamo 4 fasi:

1. **Inception** (principio): qui l'obiettivo è far partire il progetto. Dobbiamo quindi stabilire la *fattibilità* del progetto, creare un caso di business (cioè dimostrare la redditività del progetto), catturare le specifiche di base ed individuare i primi rischi critici.

   Gli workflow principali saranno requisiti ed analisi.

   Vediamo quelle che sono le *milestone* associate a questa fase: vogliamo stabilire un'associazione fra *condizioni* da soddisfare e **deliverable** (*consegnabili*) che possiamo appunto dare come ottenuti. In particolare, potremmo avere:

Condizioni	**Deliverable**
Le persone coinvolte sono d'accordo sugli obiettivi di progetto	Un documento che riassume i requisiti principali di progetto
Viene tracciata l'architettura generale	Un documento che delinea l'architettura generale
Si crea un primo piano di progetto	Il piano di progetto

2. **Elaboration** (elaborazione): è la fase dove dove si delina un'architettura eseguibile, si perfezionano i rischi valutati, si definiscono gli *attributi di qualità*, si cerca di catturare almeno l'80% delle specifiche funzionali, si crea un piano dettagliato per la fase di costruzione e si formula un offerta per il cliente che comprende risorse, tempo e staff richiesto.

   Gli workflow principali includeranno requisiti, analisi e la prima fase di design.

   Milestone in questo saranno ad esempio:

Condizioni	**Deliverable**
Viene creata un'architettura eseguibile	L'architettura eseguibile
L'architettura dimostra di aver individuato i rischi importanti	I modelli UML statico, dinamico e dei casi d'uso
Si crea un piano di progetto realistico e realizzabile	Un piano di progetto aggiornato

3. **Construction** (costruzione): in questo caso si prende l'architettura delineata in fase di elaborazione e si inizia a sviluppare il prodotto software vero e proprio.

   Il workflow principale sarà caratterizzato da design e sviluppo, nonché pesante testing.

   Le milestone includeranno:

Condizioni	**Deliverable**
Il prodotto software è sufficientemente stabile	Il prodotto software, documentazione
I committenti sono pronti per l'installazione del software	Manuali, documentazione

4. **Transition** (transizione): questa è la fase dove si risolvono i difetti delle versioni beta e si prepara l'installazione del software nell'infrastruttura dell'utente. Inoltre si realizzano i manuali utente ed eventualmente si fornisce consulenza.

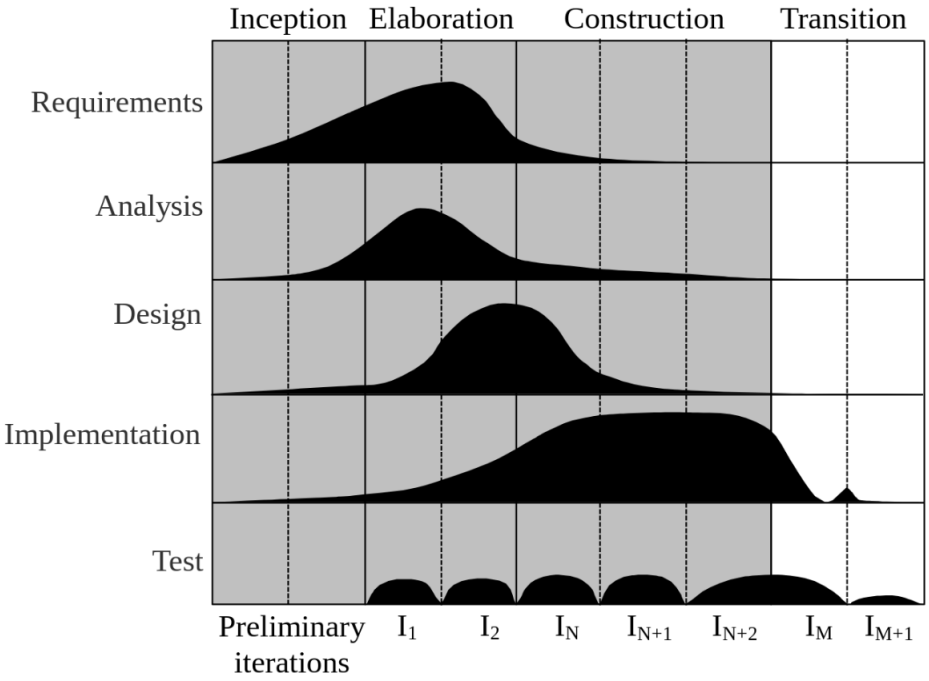
Il workflow comprenderà sviluppo e testing delle ultime funzionalità.

Le milestone saranno ristrette:

Condizioni	Deliverable
Il prodotto è stabile e (perlopiù) privo di bug	Il prodotto software finito

Ciascuna fase corrisponde a una o più iterazioni. Non è detto che lo "sforzo" (*effort*) su ogni workflow sia però lo stesso su ogni workflow nelle diverse fasi: abbiamo infatti dettagliato quali sono gli workflow più indicati per ogni fase.

Possiamo schematizzare lo sforzo sugli workflow per ogni fase con il seguente grafico:



dove per ogni fascia associata agli workflow (Requirements, Analysis, ecc...) l'andamento del carico su quel workflow per ogni iterazione (I_1, ecc...).

Notiamo come ad esempio la fase di testing diventa parte integrante di ogni iterazione praticamente dalla fase di elaborazione, se non per l'inizio dell'iterazione (dove presumibilmente stiamo progettando/implementando quanto sarà testato).

2.3 Workflow requisiti

Il workflow requisiti ha compito di individuare i requisiti del sistema. Questi sono di due tipi:

- **Funzionali**: legati a *cosa* il sistema deve fare;
- **Non funzionali**: legati a *come* il sistema deve funzionare.

Per definire i requisiti in UML possiamo usare un formato molto semplice, del tipo:

```
<id> Il <nome del sistema> deve <funzione da realizzare>
```

dove `<id>` identifica un requisito.

Quando i requisiti diventano molti, è utile raggrupparli per tipologia. 2 o 3 livelli di profondità della gerarchia sono appropiati finché non si lavora con requisiti particolarmente complessi.

Ogni requisito può essere corredato di uno o più *attributi*, cioè coppie chiave/valore associate al requisito stesso.

2.3.1 Analisi delle priorità

L'attributo più comune dei requisiti è la **priorità**. Questa si definisce secondo l'acronimo **MoSCoW**, cioè:

- **Must have**: requisiti fondamentali per il sistema;
- **Should have**: requisiti importanti che possono (dopo opportuna discussione) essere omessi;
- **Could have**: requisiti opzionali (da realizzare se possibile, cioè se c'è tmepo);
- **Want to have**: requisiti che non verranno realizzati adesso, ma al massimo in successive release.

2.3.2 Individuazione dei requisiti

I requisiti sono generati dal contesto di sistema che si vuole modellare, comprensivo di:

- Gli utenti del sistema;
- Le altre persone coinvolte (installatori, ecc...);
- I sistemi con cui il sistema deve interagire;
- I requisiti hardware del sistema e altri vincoli tecnici;
- Vincoli legali e regolamenti;
- L'obiettivo di business nostro e del cliente.

L'individuazione dei requisiti genera solitamente un documento di visione d'insieme, scritto in linguaggio naturale, che delinea i requisiti realizzabili del progetto.

Un processo che possiamo usare è quello di *deduzione* dei requisiti, tecnica dove si cerca di estrarre i requisiti dalle persone coinvolte nel progetto.

Altre metodologie sono le *interviste*, i *questionari* e i *gruppi di lavoro*.

2.3.3 Modellizzazione casi d'uso

La **modellizzazione dei casi d'uso** fa parte dell'ingegnerizzazione dei requisiti, e consiste nell'individuare come gli *attori* (sostanzialmente gli utenti e gli altri elementi coinvolti nell'uso del sistema) interagiscono con questo, all'interno del *soggetto* (il dominio di operazione del sistema).

Questo processo si svolge nel modo seguente:

- Identificare un *confine* candidato del sistema, cioè il dominio di operazione del sistema stesso. Identificare il confine del sistema significa capire cosa il sistema è e cosa non è. Questo aiuta nella definizione delle specifiche funzionali.

  In UML i confini del sistema sono chiamati **soggetto**;

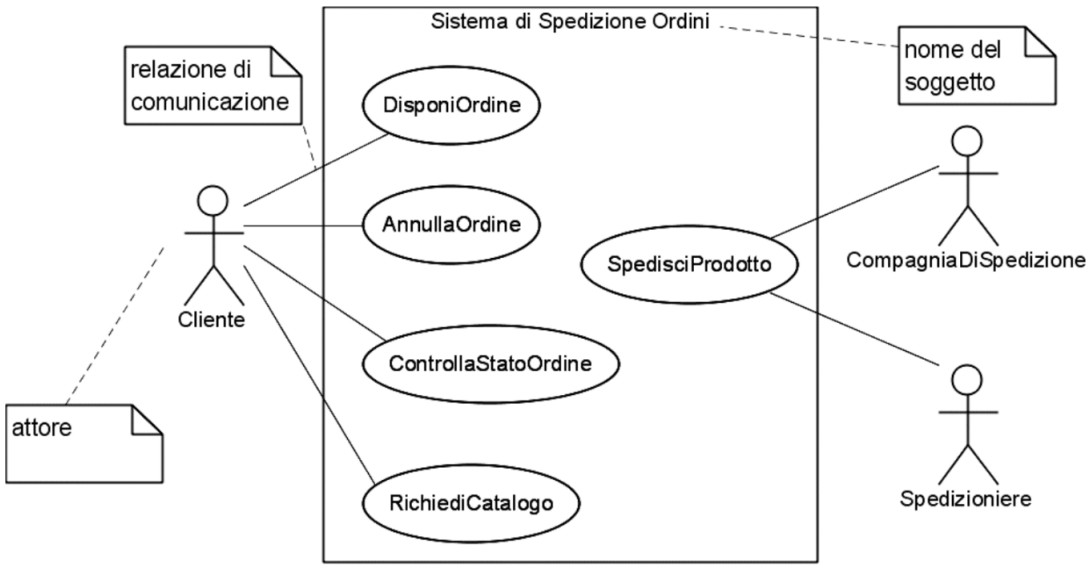
- Trovare gli *attori* coinvolti nell'uso del sistema, cioè il *ruolo* che le entità esterne assumono quando interagiscono *direttamente* col sistema.

  Notiamo che nulla vieta che la stessa entità (fisica o astratta) assuma diversi ruoli nell'interazione col sistema. Ad esempio, possiamo pensare a un utente che si comporta sia da utilizzatore che da amministratore di un certo sistema.

  In UML gli **attori** sono esterni ai *soggetti*. Potrebbe comunque essere che un sistema detiene una rappresentazione interna dell'attore (ad esempio una classe o un record di DB che mantiene i dati dell'utente);

- Trovare i **casi d'uso** del sistema, cioè il tipo di operazioni che il sistema dovrà compiere per conto degli utenti all'interno del suo dominio. A un caso d'uso è associato un *flusso* d'utilizzo del sistema da parte dell'utente. Flussi che divergono dal flusso di default vanno categorizzati e sono detti *flussi alternativi*.

Vediamo come la modellizzazione dei casi d'uso ci dà una visione d'insieme di come il sistema interagisce, nel suo dominio (*soggetto*) con le parti coinvolte (*attori*):



2.3.4 Flussi di casi d'uo

Un caso d'uso è quindi modellizzato attraverso una struttura tabulare che rispecchia la seguente:

Nome caso d'uso
Indice
Descrizione
Attore primario
Attori secondari
Precondizioni
Flusso principale: 1. Azione 1; 2. Azione 2; 3. ecc...
Postcondizioni
Flussi alternativi: • Azione 2 fallita → Azione 3; • ecc...

L'**indice** è solitamente progressivo (si possono adottare tassonomie, sopratutto nel caso di progetti complessi).

Un caso d'uso è sempre avviato da un singolo attore, l'attore **primario**. Questo non preclude il fatto che più attori possano avviare lo stesso flusso in momenti diversi. Inoltre, non preclude che altri attori vengano coinvolti: questi saranno gli attori **secondari**.

Le **precondizioni** sono le condizioni che devono verificarsi affinché il caso d'uso venga messo in moto, le **postcondizioni** le condizioni che questo lascerà una volta terminato.

Per definire i casi d'uso in UML usiamo ancora una sintassi molto semplice:

```
Il caso d'uso inizia quando un <attore> <funzione>
```

Il flusso di eventi è a questo punto una sequenza (nel caso più semplice):

```
<numero> Il <attore o altro> <azione>
```

Vediamo come è possibile prevedere dei **flussi alternativi** rispetto al **flusso principale** del caso d'uso. Questo permette una modellazione di comportamenti di errore, eccezioni, o semplici anomalie del flusso principale che si vogliono modellizzare. Per flussi più complicati ci è concesso usare altri costrutti più tipici della programmazione strutturata, cioè:

- Costrutti di ripetizione (`for`, `while`, ecc...);
- Costrutti condizionali (`if`, ecc...).

I flussi alternativi possono attivarsi in 3 modi differenti:

- Per scelta deliberata dell'attore principale;
- Attivato dopo un passo del flusso principale, in questo caso si specifica:

```
Il flusso alternativo comincia dopo il passo <numero> del flusso
    principale
```

- Attivato ad un passo qualsiasi del flusso principale, in questo caso si specifica:

```
Il flusso alternativo comincia in qualsiasi momento
```

Chiaramente, in ogni caso deve esserci una condizione che si verifica perché il flusso alternativo cominci.

2.3.5 Confronto fra requisiti e casi d'uso

Una volta terminata l'analisi dei requisiti e dei casi d'uso, si può procedere a stabilire le relazioni che collegano queste 2 categorie (una relazione molti a molti). Strumento utile in questo caso è la **matrice di tracciabilità**:

Requisiti \ Casi d'uso			
	X		X
		X	
			X

2.3.6 Glossario di progetto

Il *glossario di progetto* è uno dei deliverable principali della fase di ingegnerizzazione dei requisiti. Questo fornisce un dizionario di termini chiave e definizioni usate nel dominio di applicazione, comprensibili a chiunque sia coinvolto nel progetto. Di fondamentale importanza è individuare i **sinonimi**, che potrebbero essere innumerevoli e non apparentemente equivalenti. Non meno importanti sono gli **omonimi**, cioè parole uguali usate con significati diversi.

Un esempio di glossario di progetto, riferito ad un sistema per la gestione dei prospetti di laurea, potrebbe essere il seguente:

Termine	Significato
Unità didattica (sinonimi: **Segreteria**)	La segreteria che interagisce col sistema, fornendo corso di laurea, data di laurea e elenco matricole e ricevendo in cambio i prospetti per la commissione e la possibilità di inviare i prospetti agli studenti.
Prospetti	Gli artefatti che il sistema genera, rappresentati da un frontespizio, una tabella di esami con informazioni associate, informazioni sulle medie e nel caso di prospetti destinati alla commissione, proposte di voto. Le proposte di voto vengono calcolate secondo formule specifiche ad ogni corso di laurea.
Commissione	Coloro che ricevono il prospetto dall'unità didattica, scaricato direttamente dalla pagina del sistema. Il loro prospetto è comprensivo di proposte di voto.

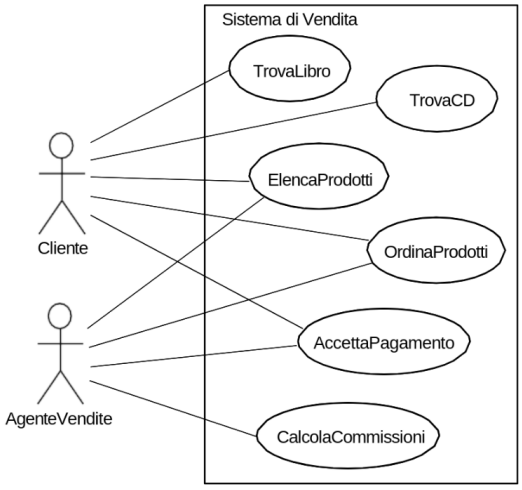
3 Lezione del 03-10-25

Cotinuiamo a parlare dell'analisi dei requisiti, in particolare riguardo ai casi d'uso, introdotti in 2.3.3.

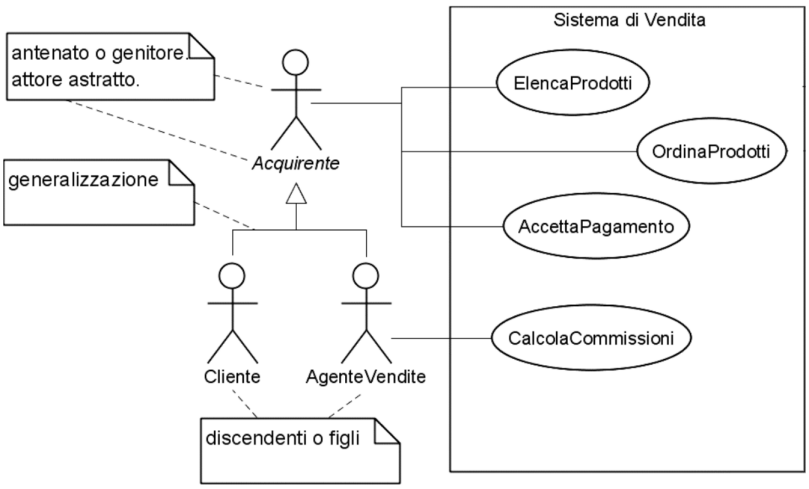
3.1 Modellizzazione casi d'uso avanzata

La modellizzazione dei casi d'uso già vista, comprensiva di *soggetto* (il dominio del sistema), *attori* (entità che interagiscono col sistema) e *casi d'uso* (modalità secondo le quali si interagisce col sistema) può essere espansa introducendo alcuni costrutti dal paradigma della programmazione ad oggetti. Vediamone alcuni:

- Può essere utile stabilire **relazioni** fra attori e casi d'uso, cioè semplicemente stabilire quali attori usino quali casi d'uso (senza negare a più attori di usare lo stesso caso d'uso). Vediamo ad esempio come a due attori diversi, *Cliente* e *AgenteVendite*, possono interessare casi d'uso differenti all'interno dello stesso sistema:

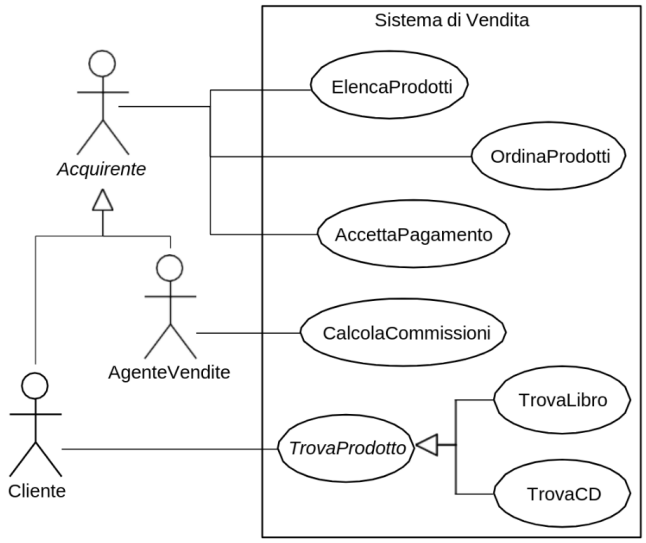


- La complessità data da più attori che interagiscono con gli stessi casi d'uso può essere ridotta introducendo la **generalizzazione degli attori**: più *sottoattori* potrebbero ereditare relazioni con casi d'uso da un *superattore* comune e *generico*. Riguardo all'esempio precedente, vediamo come si può introdurre un *Acquirente*:



Gli attori *Cliente* e *AgenteVendite* diventano a questo punto generalizzazioni di *Acquirente*. Notiamo come questo ci permette di spiegare elegantemente come mai entrambi gli attori sono interessati a casi d'uso comuni come *ElencaProdotti*, ecc...

- Come si generalizzano gli attori, potrebbe essere utile stabilire la **generalizzazione dei casi d'uso**: i sottocasi ereditano tutte le caretteristiche (relazioni, punti d'estensione, precondizioni, postcondizioni, flussi principali e alternativi) del supercaso, e possono sovrascriverle (tranne relazioni e punti d'estensione). Ad esempio, nell sistema visto finora si potrebbe voler generalizzare il caso d'uso *TrovaProdotto* sulla base del prodotto cercato:



  Potrebbe essere utile definire alcune regole su come i sottocasi possono sovrascrivere i flussi dei supercasi, assunti questi come liste di operazioni:

  - Si indicano fra parentesi le operazioni ereditate, ad esempio 3. (3.) o 4. (3.) se si è rinumerata;
  - Si prefiggono con una "o" loe operazioni sovrascritte, ad esempio 3. (o3.) o 4. (o3.) se si è rinumerata;
  - Infine, le operazioni aggiunte da zero si indicano semplicemente con numeri non usati nel supercaso.

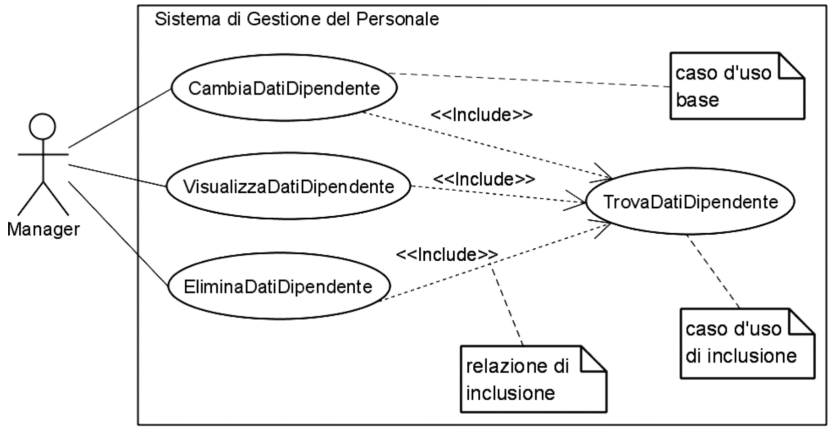
- Se si prevedono casi d'usi e attori "figli", possiamo anche prevedere casi d'uso e attori **astratti**, cioè incompleti e pensati per definire caratteristiche base. In particolare, preferiamo definire i **casi d'uso astratti** come casi d'uso con sequenze di operazioni incomplete e da definire, mentre gli **attori astratti** possono essere una variante più forte degli *attori generici* visti quando abbiamo introdotto la *generalizzazione degli attori*.

3.1.1 Relazioni `<<include>>` e `<<extend>>`

Altri due meccanismi di estensione della modellizzazione dei casi d'uso sono rappresentati dalle relazioni `<<include>>` e `<<extend>>`. Abbiamo che queste relazioni sollevano la specifica del flusso dalle relazioni di ereditarietà fra casi d'uso, mantenendo solo il collegamento semantico. Vediamole nel dettaglio:

- La relazione `<<include>>` definisce, appunto, l'*inclusione* di un caso d'uso (detto **caso d'inclusione**) all'interno di un altro (detto **caso base**). Dal punto di vista del programmatore, il caso d'inclusione rappresenta un frammento di comportamento che viene ripetuto in più casi d'uso base allo stesso modo (la corrispondenza più significativa dal lato implementativo è una banale chiamata di funzione).

  Un esempio di inclusione di casi d'uso potrebbe essere il seguente:



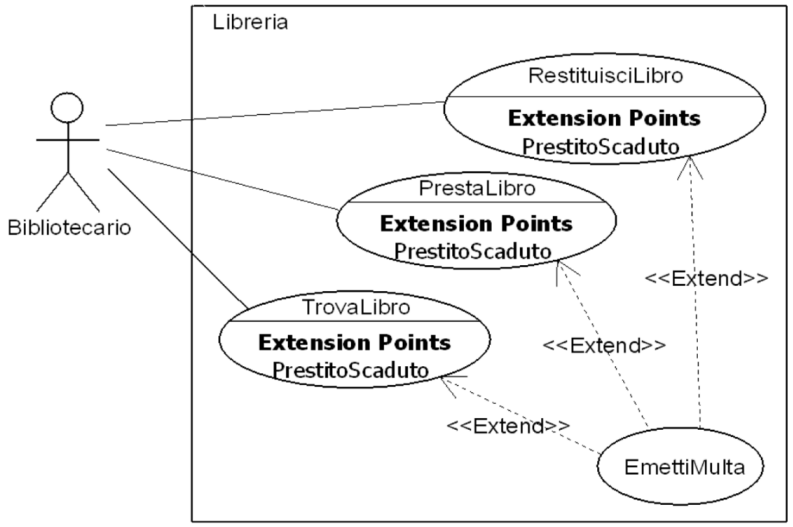
  dove il frammento di comportamento *TrovaDatiDipendente* è incluso nei casi d'uso *CambiaDatiDipendente*, *VisualizzaDatiDipendente*, *EliminaDatiDipendente*.

  Abbiamo che sicuramente i casi base non sono completi senza casi d'inclusione. Di contro, anche i casi di inclusione possono essere incompleti senza un caso base che li includa: in questo caso si dicono **non istanziabili**.

- La relazione `<<extend>>` definisce una relazione apparentemente simile all'inclusione, ma più legata alla gestione di eccezioni ai flussi principali: definiamo infatti **punti di estensione** i punti di un flusso dove si può aggiungere comportamento. In questo, i **casi di estensione** *estendono* i **casi base** nei punti di estensione. Nulla nega che lo stesso caso d'estensione estenda più punti d'estensione.

  Per approfondire, abbiamo che ogni caso d'uso d'estensione non è completo, e consiste di più frammenti detti **segmenti di inserzione**. Ogni segmento di inserzione deve combaciare con un punto d'estensione del caso base che si va ad estendere. Inoltre, si può avere che due casi di estensione estendono lo stesso punto d'estensione: in questo caso si prevedono più comportamenti d'estensione, il cui ordine di esecuzione è indeterminato.

Vediamo come si possono estendere i casi d'uso degli esempi precedenti introducendone uno, *EmettiMulta*, e usandolo per estendere i flussi alternativi di fallimento:



Infine, abbiamo che i casi di estensione possonoo essere vincolati da *precondizione*, e vincolare lo stato successivo del sistema dopo la loro esecuzione attraverso *postcondizioni*. Per completare il quadro assieme all'esempio fatto sopra, la corrispondenza implementativa più banale è quella di una chiamata condizionale di funzione.

3.2 Workflow analisi

Il workflow analisi ha il compito di creare un modello che catturi *cosa* il sistema deve fare (il *come* è prerogativa del workflow progetto). Gli artefatti prodotto in questo workflow sono:

- **Classi di analisi**: modellano i concetti chiave del soggetto;
- **Realizzazioni di casi d'uso**: illustrano com ele istanze delle classi di analisi possono interagire per realizzare i casi d'uso registrati nel workflow requisiti.

In questo caso, questa parte corrisponde con le *strutture statiche* e il *comportamento dinamico* che avevamo discusso in 2.1.

3.2.1 Classi e oggetti

Un'**oggetto** è un entità discreta con un confine ben definito che ne incapsula stato e comportamento. La struttura dell'oggetto e il modo in cui ci si interagisce sono definite dalla **classe** dell'oggetto. La classe si distingue dall'oggetto in quanto l'oggetto è un **istanza** della classe.

Tutti gli oggetti hanno **proprietà** comuni:

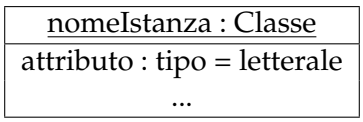
- **Identità**: un numero o qualche altro identificativo che individua univocamente l'oggetto. Per quanto ci riguarda, un *riferimento* all'istanza di classe;

- **Stato**: determinato dallo stato interno corrente dell'oggetto e le relazioni che al momento questo ha con altri oggetti. Per quanto ci riguarda, il valore dei *campi* dell'oggetto;
- **Comportamento**: le operazioni che l'oggetto può compiere. Chiaramente, questo può essere influenzato dallo stato corrente dell'oggetto. Per quanto ci riguarda, sono i *metodi* dell'oggetto.

Vediamo quindi come l'UML mira a modellizzare sistemi **orientati agli oggetti**, dove più oggetti *istanze* di *classi* interagiscono (**scambiano messaggi**) mantenendo privato il loro stato interno (**incapsulamento**).

3.2.2 Notazione di oggetti

In UML per identificare oggetti si usa un formato tabulare del tipo:



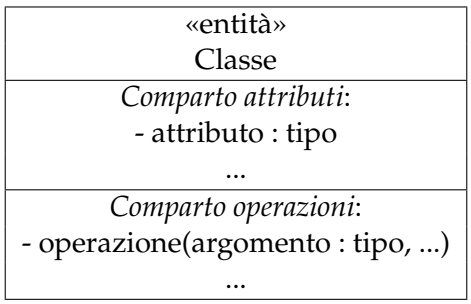
dove notiamo che:

- L'identificatore di oggetto è sempre sottolineato;
- Il nome dell'oggetto può essere risparmiato nel caso di *oggetti anonimi*;
- L'identificatore può di contro essere composto solo dal nome dell'oggetto: questo è utile in fase di analisi prima che una classe vera e propria venga definita.

Le operazioni defnite sull'oggetto non vengono riportate in quanto sono comuni a tutte le istanze di classe (vanno cercate nella definizione di classe).

3.2.3 Notazione di classi

Per identificare classi in UML si usa ancora una volta una struttura tabulare:



dove notiamo che:

- Distinguiamo in *comparto attributi* e *comparto operazioni*:
  - *Comparti attributi*: la visibilità viene solitamente omessa. A volte, si possono usare gli ornamenti (+, -, #, ~), che corrispondono a (*public*, *private*, *protected*, *package* (o *friendly*)). Inoltre, si può indicare una *molteplicità*, che per i nostri scopi definisce array;

  - *Comparti operazioni*: di queste si possono definire i parametri in entrata e in uscita usando le parole chiave `in`, `out` e `inout` nella lista degli argomenti.

Notiamo inoltre che si possono definire operazioni specifiche all'implementazione del paradigma OOP (tipicamente il **costruttore**), e che si possono sottolineare i membri, in qualsiasi comparto, che hanno **visibilità di classe**. Quest'ultima caratteristica è per noi assimilabile all'idea dei membri *statici*: come sappiamo la classe può accedere ai suoi membri statici, mentre le istanze possono accedere sia ai membri di istanza che ai membri statici.

Una volta definite classi e istanze di queste, si può usare la relazione `<<instantiate>>` (diretta da istanza a classe) per definirne la dipendenza.

3.2.4 Trovare le classi

Il problema fondamentale del design OOP diventa quindi trovare tassonomie di classi che descrivono bene il soggetto e implementano con relazioni semplici e compatte i casi d'uso. Non esiste una metodologia prefissata per lo sviluppo di tali tassonomie. Alcuni metodi che possiamo usare sono:

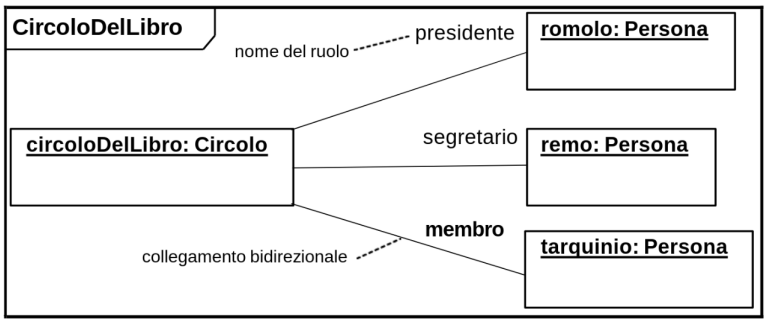
- Analisi **nomi-verbi**: i nomi ed i verbi nel testo indicano rispettivamente classi e responsabilità. Chiaramente, questo metodo richiede una buona conoscenza del soggetto, un buon glossario (in modo da individuare sinonimi ed omonimi), e potrebbe richiedere ulteriori fasi di elaborazioni delle classi candidate trovate (magari fusioni di classi simili, rimozione di classi inutili, ecc...);
- Analisi **CRC** (*Classe, Responsabilità, Collaboratore*): possiamo intendere questo metodo come estensione del precedente. Per *collaboratori* si intendono altre classi che possono collaborare con la classe definita per realizzare parte della funzionalità del sistema;
- Analisi **RUP** (*Rational Unified Process*): questa tecnica si basa sull'individuare classi sulla base di tre modelli:
  - Classi **boundary** (*confine*): mediano l'interazione tra soggetto e attori esterne. Rappresentano quindi interfacce, che siano con gli utenti, con altri sistemi o con dispositivi fisici;
  - Classi **control** (*controllo*): rappresentano controllori che mediano il comportamento associato a casi d'uso. Queste non devono essere troppo grandi, ed anzi è buona pratica suddividerle in più unit funzionali;
  - Classi **entity** (*entità*): caso principe dell'OOP, queste modellano oggetti reali all'interno del soggetto. Vengono gestite dalle classi di controllo per implementare i casi d'uso, e vengono scambiate fra le classi di confine.

Oltre a queste metodologie si possono individuare le classi sfruttando oggetti fisici, documenti d'ufficio, usando interfacce, adottando modellizzazioni in *componenti* del soggetto o entità concettuali.

3.2.5 Relazioni fra classi

Una volta individuate le classi, cioè la struttura *statica* del sistema, è opportuno definire la natura *dinamica* della collaborazione che ci aspettiamo fra le stesse. Questa si definisce attraverso **collegamenti** fra *oggetti* e **associazioni** fra *classi*.

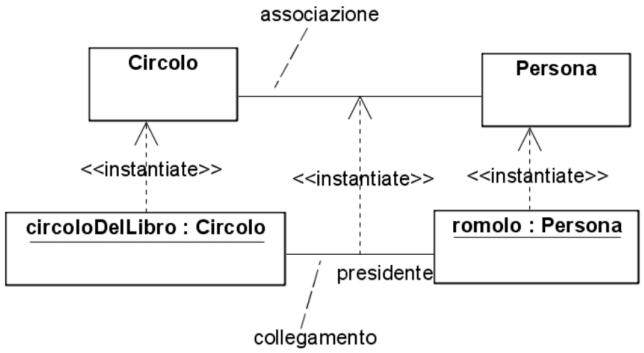
Per definire i collegamenti bisogna usare un linguaggio sollevato delle specifiche del linguaggio di programmazione, come i **diagrammi ad oggetti** dell'UML:



I *collegamenti* in UML sono **dinamici** (non fissi nel tempo), possono essere **bidirezionali** o **monodirezionali**, e devono essere supportati da una *associazione* fra classi.

Le *associazioni* sono solitamente più sofisticate nei diagrammi, e collegate ai rispettivi collegamenti da relazioni `<<instatiate>>`. Nelle associazioni vogliamo definire una stringa che definisce l'associazione, i **ruoli** delle classi coinvolte, e magari indicazioni sulla **molteplicità** dell'associazione (uno a uno, uno a molti, molti a molti e via dicendo).

Vediamo quindi come *associzioni* e *collegamenti* sono schematizzabili rispettivamente fra **classi** e **oggetti**, cioè:



La molteplicità può poi essere supportata dalla **navigabilità** dell'associazione, intesa nei nostri scopi come la possibilità di raggiungere una classe a partire da quella a cui è associata. Dovrebbe apparire chiaro come, ad esempio, una classe che contiene un riferimento ad un'altra ci permette di navigare verso questa, e non viceversa. Il sistema di navigabilità di UML 2.0 è comunque più sofisticato di quanto effettivamente necessario, e le associazioni *bidirezionali* e *monodirezionali* bastano spesso a riassumere tutta la funzionalità di cui abbiamo bisogno.

3.2.6 Associazioni ed attributi

L'assocazione fra classi può essere espressa anche come particolari attributi di classe (relazioni uno a uno sono riferimenti, uno a molto array di riferimenti, ecc...). In questo caso conviene adottare la regola: se la classe associata è parte integrante del modello, usare un'associazione, altrimenti un'attributo.

3.2.7 Classi di associazioni

Se l'associazione è molti a molti potrebbero sussistere dei campi che si perdono nella semplice specifica di associazione. In questo caso può essere utile costruire **classi di associazione**, che contengono riferimenti ad ognuna delle classi impegnate nell'associazione, e i campi associati a questa.

3.2.8 Associazioni qualificate

Le associazioni uno a molti possono essere qualificate da un particolare **qualificatore**, cioè un particolare indice che ci permette di trasformare la relazione in uno a uno, navigando verso la classe associata. L'esempio tipico è il qualificatore *codice fiscale* per una classe che si associa a più classi individuo.

3.2.9 Dipendenza

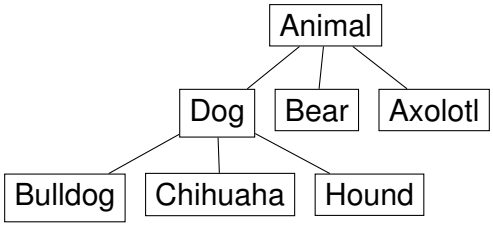
Una dipendenza è una particolare relazione tra due elementi, dove la modifica di un elemento (detto **fornitore**) può influenzare l'altro elemento (detto **cliente**). Esistono 3 tipi di dipendenze in UML:

- Dipendenza di **uso**: se il cliente usa alcuni servizi messi a disposizione dal fornitore. Questa si divide a sua volta in:
  - `<<use>>`: il cliente usa in qualche modo il fornitore;
  - `<<call>>`: il cliente invoca un'operazione eseguita dal fornitore;
  - `<<parameter>>`: il fornitore è parametro di un'operazione del cliente;
  - `<<send>>`: il cliente invia il fornitore a qualche altro bersaglio;
  - `<<instantiate>>`: il cliente è un'istanza del fornitore, già vista.
- Dipendenza di **astrazione**: se le entità sono a livelli di astrazione diversi. Questa si divide a sua volta in:
  - `<<trace>>`: cliente e fornitore rappresentano lo stesso oggetto ma in modelli diversi;
  - `<<substitute>>`: il cliente può sostituire il fornitore;
  - `<<refine>>`: simile a `<<trace>>`, ma riferito ad un'ottimizzazione all'interno dello stesso modello;
  - `<<derive>>`: il cliente può essere derivato in qualche modo dal fornitore.
- Dipendenza di **permesso**: modella la capacità di un'entità di accedere ad un'altra entità. Questa si divide a sua volta in:
  - `<<access>>`: il cliente può accedere ai contenuti pubblici del fornitore, si riferisce principalmente ai *package*;
  - `<<import>>`: simile al precedente ma con la fusione dello spazio dei nomi;
  - `<<permit>>`: permette la violazione dell'incapsulamento in linguaggi come il C++ (pensa funzioni e classi `friend`).

3.2.10 Ereditarietà

La **generalizzazione** o *ereditarietà* è la forma più forte di dipendenza. Questa indica che un entità cliente (detta *specializzata*) può essere usata in qualsiasi contesto dell'entità fornitore (detta *base*), con la differenza che l'entità cliente definisce informazioni e procedure aggiuntive rispetto all'entità fornitore.

Facciamo l'esempio classico di un **diagramma di ereditarietà** che definisce una *tassonomia* di classi (cioè) il modo in cui più classi ereditano funzionailtà l'una dall'altra, dall'alto verso il basso:



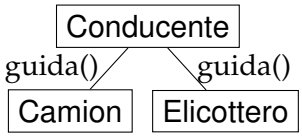
In particolare, in questo caso vogliamo usare l'ereditarietà per modellare una relazione `<<is-a>>` (*"è un"*): un bulldog *è un* cane, che a sua volta *è un* animale, e cosi via.

Sfruttando l'ereditarietà riusciamo quindi a modellare bene il tipo di categorizzazioni che incontreremo naturalmente nel soggetto del sistema che andiamo a sviulppare.

3.2.11 Polimorfismo

L'idea del **polimorfismo** è simile a quella dell'ereditarietà, e indica contesti dove si può inviare lo stesso messaggio a entità di classe differente e aspettarsi risposte semanticamente appropiate.

Per fissare i concetti, una classe base *Veicolo* può essere **generalizzata** in *Camion*, *Elicottero*, ecc... la possibilità di inviare a tutte queste il segnale *guida()* è un'esempio di **polimorfismo**:



Chiaramente, in un linguaggio di programmazione reale si dovranno usare i costrutti tipici dell'OOP per realizzare la stessa funzionalità, come classi o membri *astratti* e il meccanismo dell'*overriding*.

4 Lezione del 10-10-25

4.0.1 Ereditarietà multipla

Torniamo al discorso dell'ereditarietà. In UML è permessa l'**ereditarietà multipla**, in quanto questa esiste in linguaggi come il C++. Spesso, però, questo rappresenta un pattern da evitare ed è considerato un errore di progetto.

In prospettiva che si verificheranno errori, infatti, è meglio mantenere lineari le catene di ereditarietà in modo da minimizzare problemi.

4.1 Realizzazione casi d'uso

Abbiamo discusso finora la *modellazione delle classi di analisi*, che deve essere il primo artefatto generato dalla fase di analisi. Veniamo adesso al secondo, cioè la **realizzazione dei casi d'uso**.

Se le classi rappresentavano l'aspetto **statico** del sistema, le realizzazioni dei casi d'uso ne rappresentano l'aspetto **dinamico**.

Gli obiettivi che ci prefissiamo sono:

- Trovare quali classi interagiscono per realizzare il comportamento specificato da un caso d'uso;
- Trovare quali **messaggi** queste classi devono inviarsi per realizzare tale comportamento. In questo modo possiamo determinare quali *operazioni*, *attributi* e *relazioni* le classi dovranno avere.

Per dettagliare questo comportamento sfruttiamo 4 diagrammi, che sono:

- Diagrammi di **sequenza**: mostrano la sequenza di messaggi, dall'alto verso il basso, che due classi si scambiano per realizzare un comportamento;
- Diagrammi di **comunicazione**: mostrano le relazioni strutturali degli oggetti;
- Diagrammi di **visione generale dell'interazione**: mostrano come più comportamenti semplici implementano comportamenti complessi;
- Diagrammi di **temporizzazione**: mostrano gli aspetti in tempo reale di un'operazione.

4.1.1 Linee di vita

Una *linea di vita* (**lifeline**) rappresenta un partecipante in un'interazione, e in particolare il *ruolo* che tale partecipante intepreta in tale interazione.

Ogni linea di vita ha un nome (opzionale), un tipo e un selettore (opzionale). Il selettore è una condizione booleana che può essere usata per selezionare quali istanze partecipano all'interazione.

4.1.2 Messaggi

Un **messaggio** specifica una comunicazione fra due linee di vita in un'interazione. La comunicazione può essere:

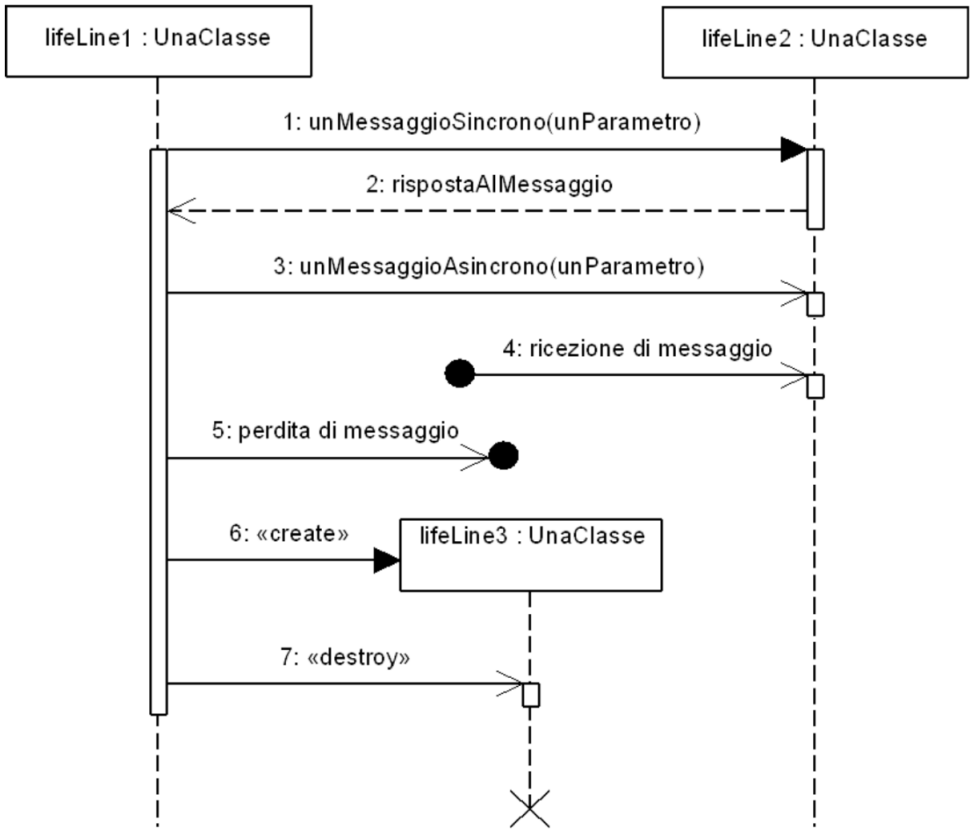
- L'invocazione di un'operazione;
- La creazione o distruzione di un'istanza;
- L'invio di un segnale.

La freccia che rappresenta il messaggio può codificare diverse informazioni:

- **Messaggi sincroni**, rappresentati da freccia piena, significano che il mittente si ferma ad aspettare che il ricevitore esegua il messaggio;
- **Messaggi asincroni**, rappresentati da freccia vuota, significano che il mittente non si ferma ad aspettare il ricevitore ma prosegue;

Esistono poi altri modelli che rappresentano informazioni come il *messaggio di ritorno*, la *creazione o distruzione di oggetti*, ecc...

Vediamo quindi un esempio di due linee vita associate a *classi* che si scambiano messaggi in UML:

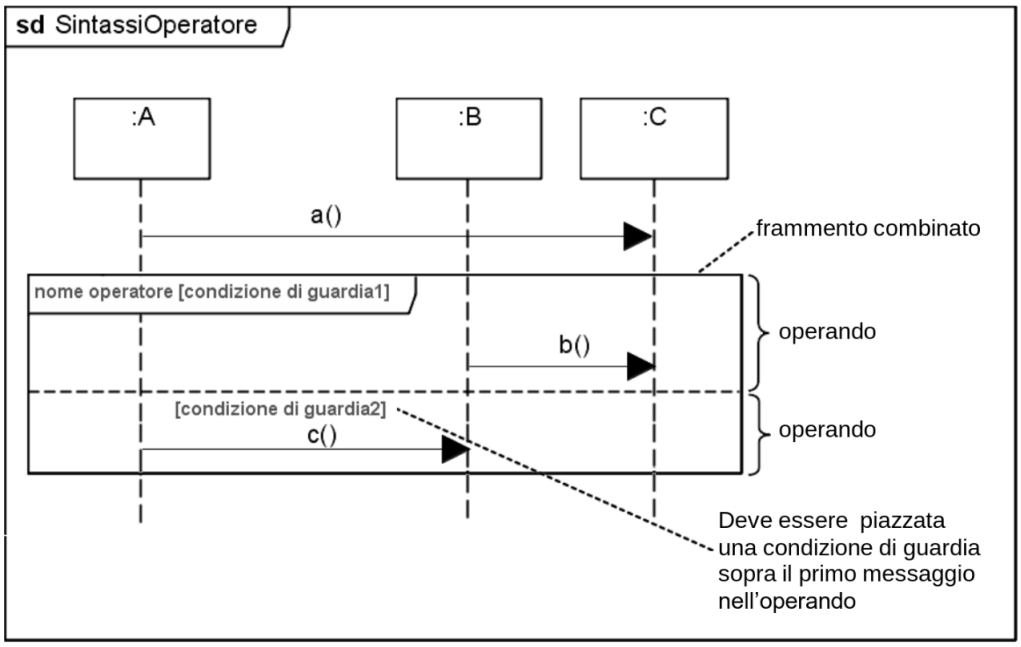


4.1.3 Frammenti combinati

I diagrammi di sequenza possono essere suddivisi in "aree" dette **frammenti combinati**. Ogni frammento combinato ha un operatore, uno o più operandi e zero o più condizioni di guardia.

L'**operatore** determina come gli operandi vengono eseguiti: solitamente gli operatori ricalcano i costrutti dei linguaggi strutturati o altre caratteristiche dell'operazioni che vengono svolte (ad esempio esecuzione *parallela*, *atomica*, ecc...).

La sintassi in UML sarà quindi la seguente:



Vediamo una tabella che riassume gli operatori principali:

Operatore	Nome	Significato
`opt`	opzione	Se la condizione è vera, viene eseguito un singolo operando
`alt`	alternative	L'operando la cui condizione è vera viene eseguito
`loop`	loop	Cicla max-min volte mentre la condizione è vera
`break`	break	Se la condizione di guardia è vera, l'operando è eseguito, ma non il resto dell'interazione
`ref`	riferimento	Il frammento combinato si riferisce ad un'altra interazione
`par`	parallelo	Tutti gli operandi vengono eseguiti in parallelo
`critical`	critico	L'operando viene eseguito atomicamente
`seq`	sequenza debole	Tutti gli operandi vengono eseguiti in parallelo con il vincolo che gli eventi che arrivano sulla stessa linea di vita da operandi differenti avvengano nella stessa sequenza degli operandi
`strict`	sequenza stretta	Gli operandi vengono eseguiti in stretta sequenza

L'operatore `loop` in particolare prevede più sintassi che rappresentano diversi tipi di iterazioni:

- Loop for, che possono essere numerati;
- Loop while, che possono essere infiniti o fino a condizione non soddisfatta;
- Loop for-each, che possono essere ripetuti su collezioni o classi.

4.1.4 Occorrenze di interazione

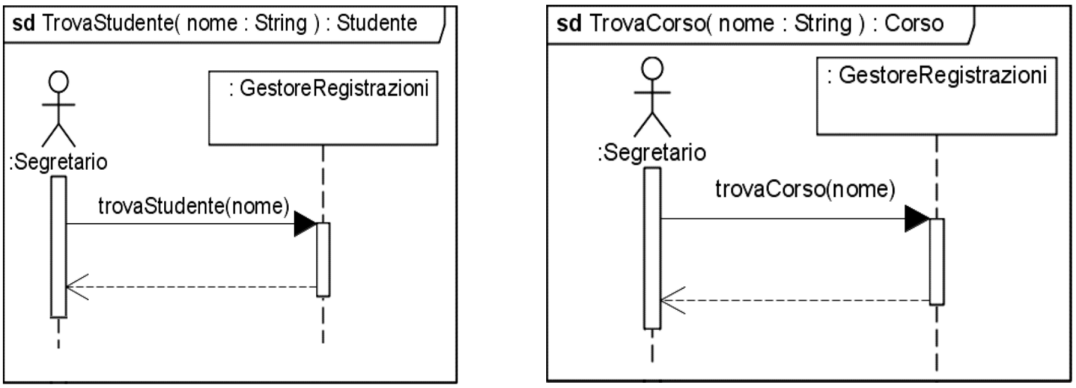
Per evitare di ripetere più volte gli stessi frammenti di interazione, l'UML mette a disposizione il riferimento ad interazione, detto **occorrenza di interazione** (che abbiamo visto come operatore `ref`).

Questo può essere usato per riferirsi a frammenti di interazione in altri diagrammi di interazione.

4.1.5 Parametri

Le interazioni possono essere **parametrizzate**. Questo permette di fornire valori diversi all'interazione ad ognuna delle sue occorrenze (meccanismo simile al passaggio di argomenti).

A scopo di esempio vediamo le interazioni parametrizzate *TrovaStudente* e *TrovaCorso* di un sistema di gestione univeristario:

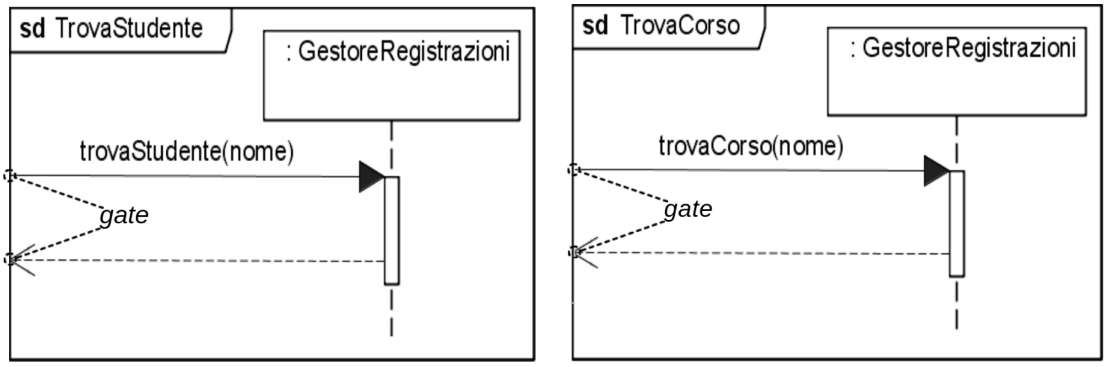


4.1.6 Gate

I *cancelli* (**gate**) rappresentano vie di accesso alle interazioni dall'esterno. Vengono quindi usati quando un'interazione è attivata da una linea di vita che non è parte dell'interazione stessa.

Vengono rappresentati graficamente come entrate ed uscite sulla finestra dell'interazione: questi punti connettono messaggi fuori dall'interazione a messaggi dentro l'interazione.

Sempre nel contesto dell'esempio sompra, vediamo come le interazioni viste possono essere realizzate anche coi gate:



Il meccanismo dei gate potrebbe sembrare simile a quello dei parametri. In sostanza si può dire che:

- I parametri rappresentano bene informazioni note all'interno dell'interazione;
- I gate rappresentano invece informazioni che provengono dall'esterno dell'interazione.

4.2 Workflow progetto

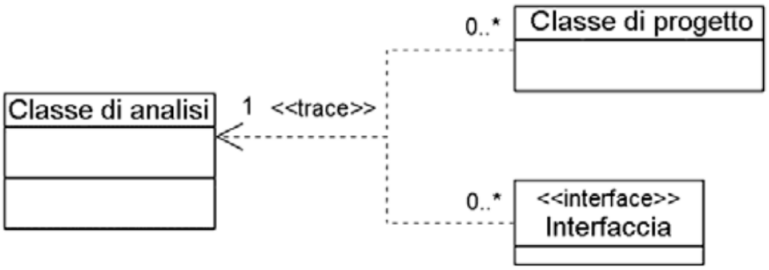
L'obiettivo del workflow **progetto** è approfondire la fase di *design* del sistema: qui si devono definire completamente tutte le funzionalità implementate per soddisfare i requisiti dell'utente.

Abbiamo studiando nel workflow *requisiti* come estrapolare i requisiti dal sistema dal dominio di sua applicazione; nel workflow *analisi* abbiamo esplorato il dominio del problema in modo da modellare strutture di classe e interazioni; adesso vogliamo tradurre questi modelli in soluzioni tecniche vere e proprie.

Abbiamo che questo workflow è tipico della fase di *elaborazione* e *costruzione* del software vero e proprio.

4.2.1 Classi di progetto

Di base, per ogni *classe di analisi* ci sono una o più **classi di progetto**, e opportunamente delle *interfacce*:



Queste non hanno più il solo scopo di modellare il dominio di problema, ma anche di essere realizzabili da programmatori veri e propri.

Abbiamo quindi che sono influenzate da:

- Il **dominio di problema**: come abbiamo visto raffinando le classi di analisi attraverso dettagli implementativi;
- Il **dominio della soluzione**: composto dall'insieme di *librerie*, *database*, *interfacce* e altre tecnologie che verranno adottate per l'implementazione vera e propria.

Campi e metodi delle classi di progetto devono essere completamente definiti (e implementabili). Per i campi, si può iniziare a specificare i tipi di variabile (se applicabile), per i metodi tipi di argomenti e ritorno. Opportunamente, è meglio dividere classi in più sottoclassi che *"ingigantire"* una classe fino a renderla inutilizzabile.

Una classe di progetto è ben definita quando risulta:

- **Completa** e **sufficiente**: le operazioni pubbliche della classe definiscono un *contratto* ben chiaro e definito con i clienti della stessa. Una classe è completa quando

offre ai suoi clienti tutto quello che i clienti si aspettano, senza operazioni in più (che non rientrano nel dominio di competenza della classe) e senza operazioni in meno (cosa che non onererebbe il contratto);

- **Primitiva**: le classi non dovrebbero implementare lo stesso comportamento più volte: dovrebbero fornire servizi singoli e atomici;
- **Alta coesione**: le classi dovrebbero modellare un singolo concetto astratto e contenere campi e metodi direttamente riconducibili a quel concetto astratto;
- **Basso accoppiamento**: le classi dovrebbero essere associate in interazioni con le sole classi con cui dividono *responsabilità*. Di base, non conviene usare metodi di altre classi solo perché il codice verrebbe altrimenti ripetuto: meglio non sacrificare la solidità strutturale per scrivere meno righe.

4.2.2 Ereditarietà

In analisi, si usa la relazione di ereditierietà se si vuole modellizzare una relazione "is a" chiara e definita. In fase di progetto, l'ereditarietà scoperta in fase di analisi può essere usata per rafforzare la struttura delle classi e risparmiare codice per funzioni a queste comuni.

4.2.3 Classi annidate

Le classi annidate sono specifiche ad alcuni linguaggi come il Java. Solitamente sono sconsiliate in quanto non ci permettono di distinguere facilmente chi sta dando problemi fra la classe madre e la classe annidata.

4.2.4 Relazioni fra classi di progetto

Veniamo al problema di rendere le **relazioni** fra classi (cioè le *associazioni*) in un'implementazione in fase di progetto.

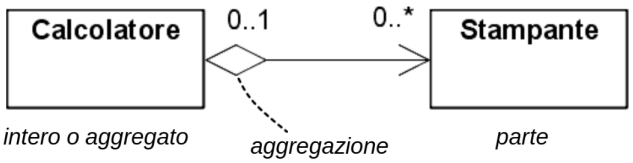
Partiamo dall'idea che nei linguaggi di progammazione che consideriamo (OOP standard, come Java o C++), non c'é nessun modo di rendere relazioni bidirezionali, o relazioni molti a molti.

Dobbiamo quindi seguire delle procedure per trasformare le associazioni di questo tipo in associazioni implementabili nel nostro dominio di soluzione.

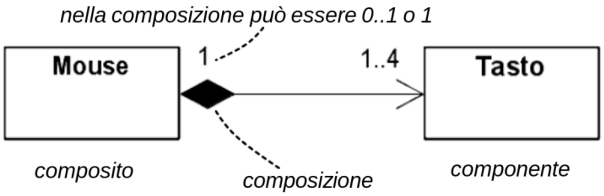
Queste procedure possono essere:

- Trasformare associazioni in relazioni di **aggregazione** o **composizione** dove appropriato:
  - La relazione di **aggregazione** è una relazione lieve fra oggetti, che lega una parte ad un'intero dove l'intero è composto da molte parti. Le parti dell'intero sono in qualche modo autosufficienti, ed esistono a priori di questo.

    Un esempio di aggregazione è la relazione fra un computer e le sue parti:

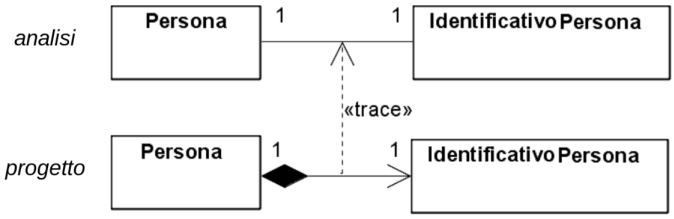


– La relazione di **composizione** è di contro una relazione molto forte, dove la parte esiste in funzione dell'intero. Ad esempio, in questo caso se muore la parte muore anche l'intero.
Un esempio di composizione è la relazione fra la parte del computer e i suoi componenti:

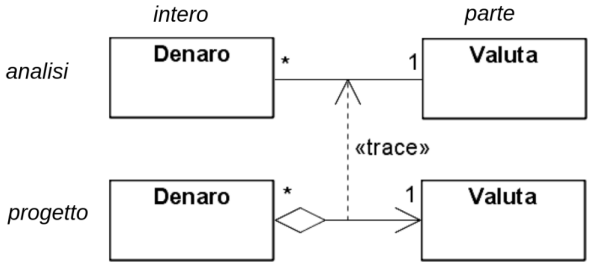


Chiaramente, bisogna distinguere prima chi nella relazione è la parte e chi l'intero, e poi risolvere problemi come capire le molteplicità (lato intero, da 0 a 1 si può comporre, oltre bisogna aggregare).

- Implementare associazioni uno-a-uno: questo si può fare sia per composizione che per aggregazione. Ad esempio:

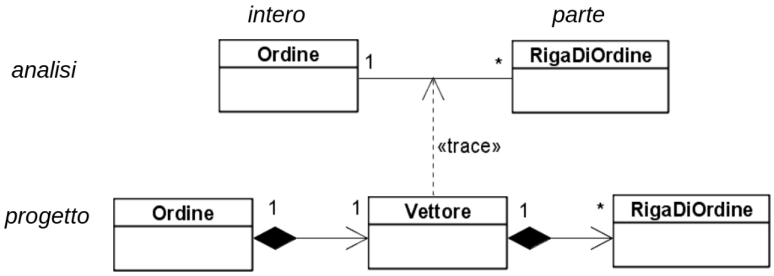


- Implementare associazioni molti-a-uno: questo si fa solitamente per aggregazione, prendendo le *molte* parti come aggregate all'*uno*. Ad esempio:



- Implementare associazioni uno-a-molti: qui bisogna usare una classe *collezione* esplicita, come ad esempio un vettore.

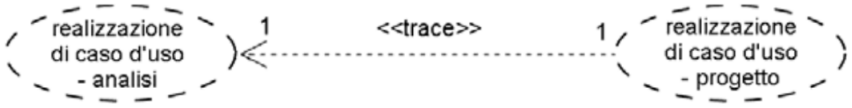
  Vediamo infatti il confronto fra la relazione molte a molti in fase di analisi alla sua implementazione in fase di progetto:



- Implementare associazioni molti-a-molti;
- Implementare associazioni bidirezionali;
- Implementare classi di associazioni.

4.3 Realizzazione casi d'uso di progetto

Come per le classi di analisi, anche i casi d'uso vanno rielaborati in **casi d'uso di progetto**:



Rispetto all'attività corrisondente in fase di analisi, abbiamo in fase di progetto alcune differenze:

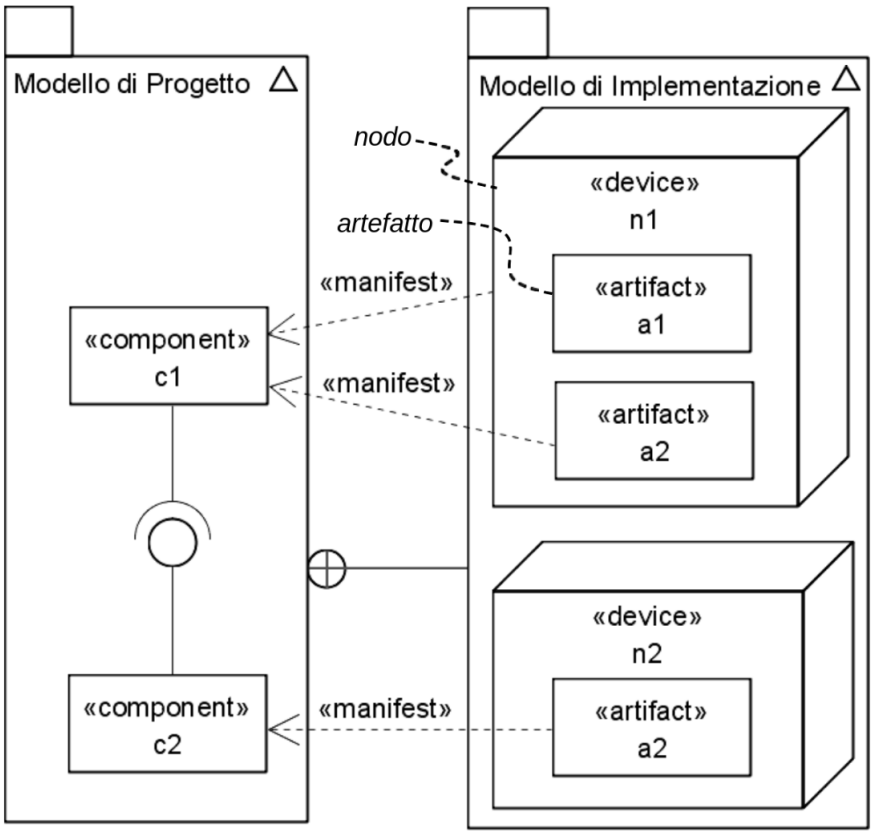
- La realizzazione dei casi d'uso coinvolge l'uso di di classi progetto, interfacce e componenti piuttosto che classi d'analisi;
- Il processo di creazione di realizzazione di casi d'uso di progetto scoprirà nuovi requisiti non funzionali e classi di progetto (atti ad interagire con il dominio della soluzione);
- Le realizzazioni dei casi d'uso aiutano a trovare i **meccanismi centrali**, cioè modi standard di risolvere un particolare problema di progetto che sono applicati consistemente nello sviluppo del sistema.

Una realizzazione di un caso d'uso nel progetto consiste in **diagrammi di interazione** e **diagrammi delle classi**.

4.4 Workflow implementazione

Veniamo quindi all'implementazione effettiva del sistema progettato in fase di progetto. Chiave di questo workflow è la scrittura di **codice**, **testing** estensivo di ogni singola funzionalità che si vuole produrre, e **dislocazione** su un'architettura reale del sistema sviluppato.

Vediamo un diagramma che spiega come vogliamo associare al modello realizzato in fase di progetto un *implementaione* vera e propria:



Nell'esempio, la relazione `<<manifest>>` significa che i componenti d'implementazione, appunto, *implementano* i relativi componenti della fase di progetto.

Solitamente nella fase di implementazione questa fase non è più necessario modellare ulteriori componenti. Eccezioni potrebbero essere le situazioni:

- Si vuole generare il codice direttamente dal modello;
- Se si sta organizzando uno sviluppo basato sui componenti (*Component Based Development*): in questo caso scegliere come disporre i componenti diventa un problema strategico.

Gli artefatti che vogliamo generare sono:

- **Diagrammi di componenti** che mostrano come gli artefatti rendono manifesti i componenti implementati;
- **Diagrammi di dislocazione** che rappresentano come gli artefatti vengono fisicamente dislocati su architetture reali per implementare il servizio richiesto.

4.5 Diagrammi di dislocazione

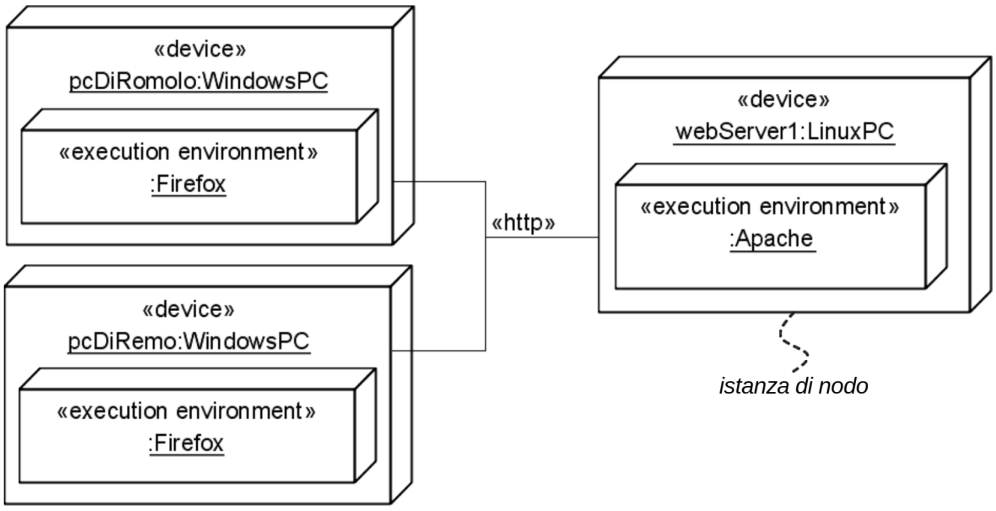
I diagrammi di dislocazione rappresentano come gli artefatti prodotto vengono, appunto, *dislocati* su istanze di nodi (macchine, ecc...). Questo processo è estremamente importante ad esempio nello sviluppo di architetture distribuite.

Esistono 2 tipi di diagrammi di dislocazione:

- **Descrittivi**: contiene nodi ed artefatti, e le relazioni che vi avvengono. Un nodo è una macchina, mentre un'artefatto è software in esecuzione su tale macchina;
- **D'istanza**: contengono istanze di nodi, di artefatti, e di relazioni. Qui le istanze di nodi sono macchine specifiche, le istanze di artefatti sono processi specifici, e così via.

  Comunque, è possibile usare nodi e artefatti anonimi quando il loro comportamento è noto a prescindere dalla specificità.

Vediamo quindi l'esempio di un diagramma di dislocazione, *d'istanza*, relativo ad un'applicazione distribuita:



4.5.1 Nodi

In dettaglio, i **nodi** rappresentano risorse computazionali che possono essere usate per realizzare il servizio. Ne distinguiamo 2 tipi:

- `<<device>>`: nodi che rappresentano dispositivi fisici;
- `<<execution environment>>`: nodi che rappresentano ambienti di esecuzione per software.

Le **associazioni** fra nodi e artefatti sono a questo punto canali di comunicazione dove informazioni possono essere scambiate.

4.5.2 Artefatti

Gli artefatti rappresentano lato software tutto ciò che viene effettivamente implementato, fra cui:

- File sorgenti;
- File eseguibili;
- Script;
- Tabelle di basi di dati;

- Documenti;
- Risultati del processo di sviluppo.

4.6 Workflow test

Veniamo quindi a discutere l'attività di **testing** che è opportuno applicare durante e dopo lo sviluppo.

Sappiamo che è impensabile sviluppare software privo di errori. Per questo dobbiamo:

- Testare il software spesso;
- Scrivere software che sia facile da testare.

4.6.1 Operazioni di test

La fase di testing si effettua vari livelli e consiste in operazioni distinte:

- Test di **unità**;
- Test di **modulo**;
- Test di **sottosistema**;
- Test di **sistema** o **integrazione**;
- Test **alfa**: prima fase dove si fornisce il sistema al cliente;
- Test **beta**: seconda fase, in cui si fornisce il sistema a più utenti;
- Test **benchmark**: il sistema viene testato in maniera standardizzata per valutarne le prestazioni;
- Test di **stress**: il sistema viene messo al prova in termini di resilienza, sicurezza, ecc... sopponendolo ad attachi o comunque stress.

4.6.2 Testing e debugging

Testing e debugging sono processi diversi fra di loro ma legati:

- Il **testing** serve a valutare la presenza di un errore;
- Il **debugging** è l'atto di risolvere tale errore.

I test di **regressione** sono utili dopo il debugging, per confermare che l'attività di debugging non abbia prodotto ulteriori errori.

4.6.3 Distribuzione di errori

Gli errori, come abbiamo detto, si verificano a prescindere dalle nostre migliori intenzioni. Possiamo adottare alcune euristiche per capire come sono distribiti:

- Gli errori tendono a concentrarsi in moduli **specifici**, solitamente i *più complessi*;
- Un modulo che contiene un errore probabilmente ne conterrà altri;

- Moduli strettamente accoppiati condivideranno più errori;
- Moduli che vengono rielaborati spesso (*refactoring* o *manutenzione*) sono più facilmente soggetti ad errori, sopratutto man di mano che il processo di refactoring si prolunga nel tempo.

5 Lezione del 24-10-25

5.1 Strategie di test

Come la progettazione, anche il testing può essere effettuato in modalità **top-down** o **bottom-up**.

- **Top-down**: si parte dalla *"testa"* del progetto e si va scendere. L'idea è di andare a simulare dalla radice fino agli altri moduli del sistema, prima che questi moduli vengano implementati. Il processo si ripete per via ricorsiva finché l'intero sistema non risulta come simulato.

  In questo caso, per testare i moduli più alti bisogna disporre dei moduli di prova, detti *simulatori* o **stub**, che vanno appunto a simulare il comportamento dei moduli più bassi.

- **Bottom-up**: si parte dai dettagli e si va a scendere. L'idea è che si vanno a collaudare prima i moduli di più basso livello nella gerarchia prodotta dalla progettazione: quando si valutano come corretti, si passa al livello superiore, e via dicendo fino a che non si è simulato il sistema intero.

  Chiaramente, testare gli stadi più bassi richiede lo sviluppo di software detti **driver** (da non confondere con i driver dei S/O). Un driver è un software che ha il compito di *pilotare* una funzionalità di livello più basso, simulando appunto come questa funzionalità potrebbe venire chiamata da eventuali stadi superiori.

  Il vantaggio sta nel fatto che si testano prima gli stadi più bassi, trovando gli errori *subito* (e non più tardi nello sviluppo, che sarebbe molto più costoso).

La strategia che si implementa più spesso nella realtà è in verità una strategia **mista**. Quando le funzionalità di basso livello risultano più complicate, è infatti utile sfruttare un approccio bottom-up, mentre se il sistema è governato da una complessa logica di alto livello, è più comodo il top-down.

5.1.1 Verifiche statiche

Il punto critico dei test è rappresentato dal costo. Una forma di testing alternativa è quella di **verifica** (o *ispezione*) **statica** del codice. Questo solitamente ci permette di individuare fra il 60% e il 90% degli errori.

5.1.2 Testing

L'ispezione statica chiaramente non basta, in quanto non intercetta gli errori dovuti all'esecuzione dinamica del programma. Occorre quindi procedere con un'apposita fase di **testing** dove si valuta il programma su più campioni di dati, in modo da verificarne il comportamento come corretto.

Dobbiamo ricordare che è il testing é:

- Mai **esaustivo**, e.g. un milione di test non equivarranno mai una verifica formale;
- **Costoso**, cioè richiede sfruttamente di risorse macchina, e sopratutto risorse umane (tempo speso a scrivere test, eseguire test, ecc...).

Per effettuare un test bisogna:

- Costruire un **test-case**, cioè disporre una situazione dove si conosce l'**input** e l'**output** corretto; I test-case vanno ricavati alle specifiche e dai documenti ricavati dalla progettazione;
- Dopo che il test-case è pronto, e si è configurato il sistema perché esegua il test, bisogna appunto eseguirlo, ottenere l'output e confrontarlo con quello desiderato;
- I test devono essere **ripetibili**: bisogna poter ottenere più test (magari variando parametri) sullo stesso test-case, in modo da essere il più esaustivi possibile.

Una prima idea per generare gli input è quello di generarli **casualmente**. Questa soluzione è economica, ma ha lo svantaggio di non essere molto uniforme per piccoli campioni. Inoltre, gli input non saranno particolarmente significativi all'applicazione vera e propria (quindi creando situazioni irreali e di difficile valutazione).

In questo risulta quindi più utile unire le specifiche del programma, la struttura stessa del programma, e sopratutto l'**esperienza** sul dominio del problema che il programma risolve. Questa spesso non viene dal programmatore, ma dagli attori che sono all'interno del dominio di problema stesso (i clienti).

5.1.3 Procedure di testing

Ci sono almeno 2 modi principali su come procedere al testing:

- Testing a **scatola nera** (o **black box** testing): in questo caso si ricavano gli input solo guardando alle specifiche del programa. Il codice scritto a questo punto non viene considerato. Questo tipo di testing è tipico delle prime fasi di testing.

  Dal punto di vista teorico, l'idea del testing a scatola nera è quello di dividere dividere i possibili input in **classi di equivalenza**, in modo che ogni possibile input abbia lo stesso **potere di testing**.

- Testing **strutturale**: unisce le specifiche di programma al codice finora scritto per la generazione di test. Ha il vantaggio di testare più precisamente alcune procedure e dettagli che test a scatola nera non individuerebbero nel codice.

  Dal punto di vista teorico, quello che vorremmo fare è sviluppare test che *eslporino* tutto lo spazio di stato del programma, cioè attivino tutte le procedure (pensa flussi alternativi), in modo da testare funzionalità a cui magari si accede più raramente.

5.2 Testing automatizzato

Chiaramente, è molto utile realizzare **test automatici**, cioè sfruttare strumenti software per effettuare automaticante i test al posto nostro.

I casi tipici di automazione dei test sono:

- Test **funzionali** per un prodotto, pensati per verificare le specifiche desiderate del prodotto;

- Test per **nuovi rilasci** di un prodotto, atti a verificare che la funzionalità precedente non sia stata compromessa.

I **simulatori** sono ambienti all'interno di cui si può effettuare il testing non solo delle procedure del software, ma della totalità delle condizioni operative che dovrebbero trovarsi all'esterno del software.

Questo chiaramente risulta molto costoso, ma può essere utile (se non fondamentale) per programmi particolarmente complessi, o per domini di problema particolarmente critici (mezzi di trasporto, industria biomedica, ecc...).