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.

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 procede 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.

  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*.

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...

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**.

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>
```

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.

2.3.4 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

2.3.5 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.

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.

3.0.1 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);
- 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*;
- 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). 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*.

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).

  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.

  Infine, 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.

3.1 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.

3.1.1 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**).

3.1.2 Notazione di oggetti

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

nomeIstanza : Classe
attributo : tipo = letterale ...

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).

3.1.3 Notazione di classi

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

«entità» Classe
Comparto attributi: - attributo : tipo ...
Comparto operazioni: - operazione(argomento : tipo, ...) ...

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.

3.1.4 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.

3.1.5 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). 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.

3.1.6 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.

3.1.7 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.

3.1.8 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.

3.1.9 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`).

3.1.10 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.

3.1.11 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*.