Ingegneria del Software 1

UML: Unified Modeling Language

Martedì, Aprile 5, 2016

Claudio Menghi, Alessandro Rizzi

April 22, 2016

Contents

1 Design Principles

Mentre si progetta il software è necessario considerare i seguenti principi:

- *Open Close Principle: le entità software (classi, moduli e funzioni) devono essere aperti all'estensione e chiusi alle modifiche.* Un design intelligente consente di aggiungere una funzionalita (aperti all'estensione) senza modificare il codice già sviluppato o con delle modifiche minimali (chiusi alle modifiche).
- *Dependency Invertion Principle: i moduli di alto livello non devono dipendere da quelli di basso livello ma da loro astrazioni.* Quando si utilizzano moduli di basso livello (e.g., classi che consentono letture da file) i moduli di alto livello non devono dipendere direttamente da questi elementi; è consigliato aggiungere un livello di astrazione (High Level Classes → Abstraction Layer → Low Level Classes) in modo tale da consentire di cambiare agevolmente le Low Level Classes
- *Interface Segregation Principle: i moduli non devono dipendere da interfacce che non utilizzano* Non aggiungete nelle interfacce metodi che non servono.
- *Single Responsibility Principle: ogni classe deve avere solo una ragione per essere modificata* Questo principio specifica che ogni classe si deve occupare solamente di una funzionalità. Se una classe si occupa di due funzionalità significa che è necessario dividere la classe in due.
- *Liskov's Substitution Principle: le classi derivate mediante estensioni (una classe che ne estende un altra) deve essere completamente sostituibile alla classe di partenza.*

2 Introduzione

UML contiene un insieme di notazioni grafiche che supportano lo sviluppatore nelle diverse fasi di sviluppo del software. È composto da una serie di notazioni che consentono di specificare la maggior parte dei costrutti forniti da un linguaggio orientato agli oggetti. Lo scopo di UML e' di fornire uno strumento per descrivere (progettare) il sistema in fase di sviluppo e viene utilizzato al fine di:

- aiutare gli *analisti* a capire le funzionalità del sistema
- *progettare* il sistema
- comunicare con i committenti

Di solito per scopi diversi vengono utilizzati modelli differenti.
Alcuni dei diagrammi piu' utilizzati sono:

- diagrammi di struttura: descrivono la "*struttura*" dell'applicazione in fase di sviluppo
- diagrammi di comportamento: descrivono il "*comportamento*" dell'applicazione in fase di sviluppo

Questa esercitazione è focalizzata sui *class diagrams* un particolare tipo di diagramma strutturale che consente di specificare l'architettura della soluzione proposta.

2.1 Diagrammi delle classi

Sono utilizzati per fornire una visione statica del sistema. Sono composti da classi e relazioni tra classi.

- *classi*: definisce un insieme di oggetti (istanze) che hanno le stesse caratteristiche. Per esempio la classe macchina definisce la struttura che devono avere gli oggetti (istanze) di tipo macchina. Un oggetto è una specifica istanza di una classe, e.g., una specifica macchina.
- *relazioni*: definiscono delle dipendenze tra le diverse classi del sistema, in particolare esistono relazioni di uso, generalizzazione (ereditarietà) e associazione (due tipi specifici di associazione sono aggregazione e composizione).

2.1.1 Classi

Le classi descrivono un insieme di oggetti. Al fine di descrivere gli oggetti di una classe e' necessario specificare:

- *Stato*: descrive un oggetto in uno specifico istante temporale. Lo stato viene descritto per mezzo di attributi.
- *Comportamento*: indicano i metodi che un oggetto può invocare. I comportameti sono descritti per mezzo di metodi. L'esecuzione di un metodo spesso comporta la modifica dello stato di un oggetto.

Ogni classe è rappresentata per mezzo di un rettangolo diviso verticalmente in tre parti contenenti il nome, gli attributi e i metodi dalla classe, rispettivamente. Per esempio, la classe rappresentata in Figura 1 ha come nome `Persona`, ha come attributi `nome`, `cognome` e `dataNascita` e come metodi `siSposa()` e `compieAnni()`. Gli attributi seguono la notazione: `<modificatore di visibilità> nome attributo : <tipo attributo>[= valore di default]`. Dove il modificatore di visibilità può essere public (+), private (−), protected (#) and package (∼). I metodi seguono la notazione: `<modificatore di visibilità> nome metodo (<parametro 1: tipo parametro1>, ...): <tipo di ritorno>`. I modificatori di visibilità sono gli stessi usati per gli attributi. Quando un attributo o un metodo è sottolineato, l'attributo o il metodo è *statico*.

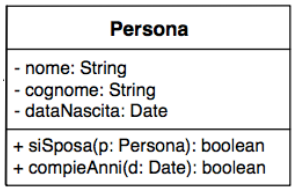

Figure 1: Una classe.

La classe mostrata in Figura 1 è mappata nel codice Java presentato in seguito.

```
public class Persona {
   private String nome;
   private String cognome;
   private Date dataNascita;

   public boolean siSposa(Persona p){
   }

   public boolean compieAnni(Date d){

   }
}
```

In un class diagram ci sono due modi possibili per indicare che una classe è astratta: scrivere il loro nome in *italico* o indicarli per mezzo della dicitura {abstract} indicata sotto il nome della classe. È inoltre possibile specificare degli stereotipi sopra il nome della classe con la seguente sintassi: $\langle\langle stereotype \rangle\rangle$. Per esempio $\langle\langle interface \rangle\rangle$ indica che la classe specificata rappresenta un interfaccia mentre $\langle\langle utility \rangle\rangle$ indica che tutti i metodi o le variabili di una classe di tipo utility sono statiche.

2.1.2 Associazione

È un tipo di relazione che descrive che due classi sono associate. Per esempio, Figura 2 presenta un associazione tra le due classi `Persona` e `Casa`[1]. L'associazione specifica la relazione `vive` la relazione tra le classi `Persona` e `Casa`. Solitamente il nome della relazione è un verbo. È anche possibile specificare alle estremità i ruoli svolti dalle classi nell'associazione e anche una espressione che specifica la molteplicità. La molteplicità si riferice ai numeri posti sulle frecce e rappresentano il numero di oggetti connessi agli altri per mezzo dell'associazione. Le moltiplicità possono essere espressi come numeri, ranges o combinazioni dei due.

- *Digit*: il numero esatto di elementi
- $*$ or 0..$*$: zero o alcuni
- 0..1 Zero or one. In Java this is often implemented with a reference that can be null.
- 1..$*$ uno o molti
- 3..5 da 3 a 5 (in alcune varianti)

Per esempio in Figura 2 viene specificato che una persona vive in una casa e che in una casa vive almeno una persona. Gli estremi di un'associazione implicano la presenza di attributi impliciti. Per esempio il class diagram descritto in Figura 2 viene mappato nel codice seguente.

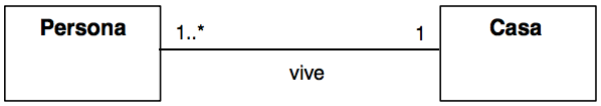

Figure 2: Associazione tra Persona e Casa.

La classe `Persona` contiene il riferimento alla `Casa` della persona.

```
public class Persona {
   private Casa casa;
}
```

La classe `Casa` contiene una collezione di persone che vivono nella casa.

```
public class Casa {
   private Collection<Persona> persone;
}
```

È possibile specificare la direzione di navigabilità dell'associazione. Per esempio, in Figura 3 l'associazione è navigabile solo nella direzione Casa→Persona. Questo implica che data una casa è possibile sapere le persone che vivono in quella casa ma non viceversa. La nuova relazione viene mappata sul codice Java seguente:

```
public class Persona {
}
```

La classe `Casa` contiene una collezione di persone che vivono nella casa.

[1] Per semplicità gli attributi e i metodi delle classi non sono mostrati nel diagramma

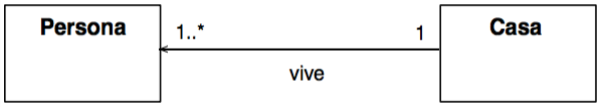

Figure 3: Associazione tra Persona e Casa.

```
public class Casa {
   private Collection<Persona> persone;
}
```

L'aggregazione è un tipo specifico di associazione che descrive una relazione "è una parte di". Nota che l'implementazione è indistinguibile da quella dell'associazione. Nell'esempio mostrato in Figura 4, un automobile aggrega un Telaio, un Motore e quattro Ruote. L'unica regola riguardante l'associazione consiste nel fatto che non è possibile creare cicli di aggregazioni.

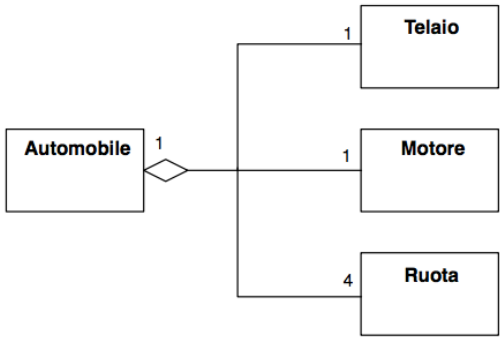

Figure 4: Aggregazione tra Automobile, Telaio, Motore e Ruota.

2.1.3 Composizione

La composizione è un altro tipo speciale di associazione. La composizione indica che il contenitore è responsabile del ciclo vita del contenuto: se il contenitore è distrutto il contenuto deve essere distrutto con esso. Nota che l'implementazione Java è indistinguibile da quella dell'associazione. Tuttavia, in questo caso, la ragione è un suo scarso utilizzo nella programmazione Java. È compito del programmatore garantire che quando il contenitore è distrutto il contenuto non deve essere più referenziato, ovvero deve essere rimosso dal garbage collector. Come per le aggregazioni non deve essere possibile creare cicli contenenti composizioni. Nell'esempio mostrato in Figura 5 quando la `Window` viene distrutta anche lo `Slider`, il `Panel` e il `Button` associati sono distrutti con essa.

2.1.4 Generalizzazione

La generalizzazione viene indicata per mezzo di una freccia non riempita. La freccia punta verso la sorgente della dipendenza. La sorgente della generalizzazione può essere una classe o un interfaccia. Se la sorgente è una classe la linea della retta è continua, se è una interfaccia la linea viene indicata mediante una linea

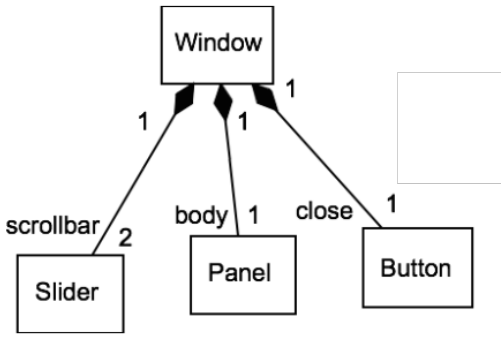

Figure 5: Composizione tra Window, Slider, Panel e Button.

tratteggiata. La generalizzazione indica che destinazione della generalizzazione `estende` o `implementa` la classe o l'interfaccia, rispettivamente. Per esempio, in Figura 6 la classe `Quadrato` estende la classe astratta `FiguraGeometrica`.

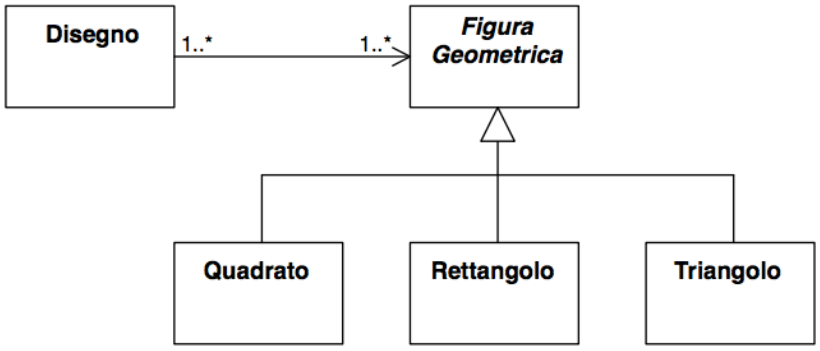

Figure 6: Generalizzazione tra FiguraGeometrica, Quadrato, Rettangolo e Triangolo.

3 Esercizi

3.1 Esercizio: Veicolo-Motore

Esercizio 1: Progettare il class diagram nella seguente situazione:

- Un Veicolo è composto da un Motore
- Veicolo: ha una targa e un numero di telaio Motore: ha una cilindrata definita su n pistoni
- Un Pullman è un tipo di Veicolo che trasporta passeggeri
- Pullman: appartiene ad una società e dispone di n posti a sedere
- Passeggero: è identificato da un nome e cognome

Identifichiamo le entità in gioco: le potenziali `classi`. Nel caso in questione identifichiamo tre potenziali

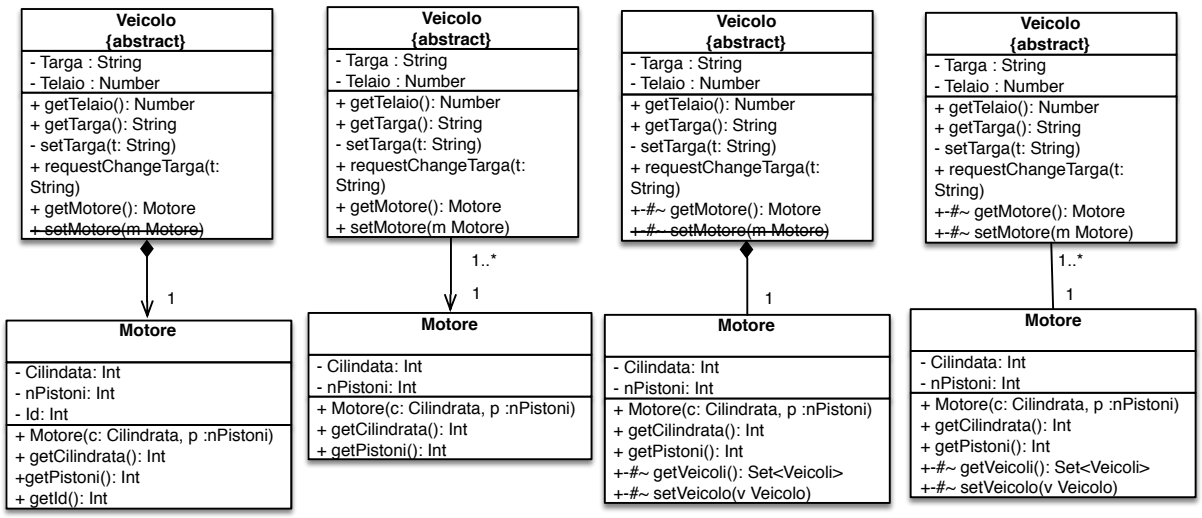

Figure 7: Un semplice esempio di gerarchia.

classi: `Veicolo`, `Pullman` e `Motore`. Iniziamo analizzando la classe `Motore`. Iniziamo discutendo la parte relativa a `Motore` e `Veicolo`. Alcune soluzoni sono presentate in Figura 7.
Il `Motore` ha una cilindrata definita su n pistoni. La cilindrata è normalmente espressa in cm^3 mediante un valore maggiore di 0, per questo scegliamo di memorizzarla per mezzo di un numero intero. Effettuiamo la stessa scelta relativamente al numero di pistoni. Cilindrata e pistoni sono accessibili mediante i metodi `getCilindrata()` e `getPistoni()`. Potrebbe sorgerci il dubbio che sia necessario memorizzare un identificativo del motore `id`, mediante per esempio un intero. Ovviamente la scelta dipende dal "problema" specifico considerato. Supponiamo di considerare un auto-officina, in tal caso il numero identificativo del motore consente di tener traccia dei motori aggiunti-rimossi dalle varie automobili, della lista di motori in magazzino etc. Se consideriamo invece un sito web, che consente di confrontare i vari tipi di autovetture, l'identificazione del numero seriale del motore non è utile per i fini prescelti.
Il `Veicolo` ha una targa e un telaio. Scegliamo di memorizzare la `Targa` per mezzo di un attributo di tipo `String` e il `Telaio` per mezzo di un attributo di tipo `Number`. Modellizziamo il `Veicolo` per mezzo di una classe `abstract`. Questo perchè ci immaginiamo ci siano delle sottoclassi, per esempio `Pullman` che estenderanno la classe astratta `Veicolo` rappresentando particolari categorie di veicoli. I metodi `getTelaio()` e `getTarga()` permettono di ottenere telaio e tarda del veicolo. Il metodo `getMotore` consente di ottenere il corrispondente motore. Possiamo fornire il metodo `setTarga()` se richesto o `requestChangeTarga()` se per esempio nella specifica è richiesto di modellizzare un procedimento nel quale per cambiare una targa bisogna contattare una motorizzazione etc[2]. L'utilizzo del metodo `setMotore` è rischioso nel caso in cui si utilizzi una composizione. Infatti nel caso della composizione bisogna garantire che nel momento della distruzione di un veicolo anche il corrispondente motore viene distrutto con esso. Bisogna stare molto attenti a chiamare il metodo setMotore visto che si corre il rischio che il reference di motore sia visibile a classi esterne. Per questo se si desidera esporre tale metodo è necessario farlo con prudenza (metterlo protected), richiamarlo solo da opportune factories etc. Inoltre, per lo stesso motivo, è bene che il metodo `getMotore` ritorni una copia del motore del veicolo.
Riguardo alla relazione tra la classe `Veicolo` e motore abbiamo tre possibili scelte: associazione "semplice", aggregazione e composizione. Con l'associazione specifichiamo che a un automobile è associato uno specifico motore. L'associazione è la relazione "più leggera" tra le tre. L'aggregazione indica che un Veicolo aggrega un motore. Per esempio potremmo dire che un veicolo aggrega un telaio, delle ruote e un motore. Infine

[2]In tal caso sarebbe necessario modellizzare anche tali elementi

la relazione più forte è la composizione. La composizione specifica che quando distruggiamo il veicolo distruggiamo anche il corrispettivo motore. Riguardo alle molteplicità è evidente che un veicolo ha uno e un solo motore. Nel caso in cui il motore rappresenti una specifica istanza di un particolare motore (identificato da un serial number), il motore appartiene a uno e un solo veicolo. In quel caso la composizione risulta la scelta più ragionevole: quando distruggiamo un auto distruggiamo anche il corrispondente motore. Tuttavia, nel caso in cui un motore rappresenti una istanza di un particolare tipo di motore (ma non un motore specifico) è possibile dire che da 1 a n macchine montano quel tipo di motore. In quel caso è richiesto l'uso dell'associazione. È infine possibile specificare la direzione di navigabilità dell'associazione. Nei primi due casi `Veicolo` ha il reference al corrispondente `Motore` ma il motore non conosce il/i veicolo/i su cui è montato. Negli ultimi due casi la relazione è navigabile in entrambe le direzioni. La traduzione in Java viene lasciata al lettore, seguendo le indicazioni date nell'introduzione.
Discutendo la classi `Pullman` e `Passeggero`. Alcune soluzoni sono presentate in Figura 8.
Il `Pullman` è uno specifico tipo di `Veicolo`, quindi `Pullman` estende `Veicolo`. `Pullman` eredita tutti i metodi e gli attributi di `Veicolo`. A tutti gli effetti un `Pullman` è un `Veicolo`. Un `Pullman` ha delle caratteristiche aggiuntive: ha un numero di posti (intero), una società (stringa) etc. I metodi `getSocietà` e `setSocietà` permettono di settare e ottenere la società relativa a un pullman. Un `Passeggero` è identificato da un nome e un cognome.
Discutiamo ora la relazione tra `Pullman` e `Passeggero`. La specifica è ambigua e si presta a varie interpretazioni che risultano in class diagram differenti. In Figura 8 ne vengono presentati alcuni. La prima interpretazione è che il nostro sistema debba tenere traccia dei passeggeri presenti in un pullman in uno specifico istante temporale. In questo caso, la prima e la seconda soluzione risultano adeguate: esiste una associazione tra le classi `Pullman` e `Passeggero`, un passeggero è su un pullman e un pullman può contenere un certo numero di passeggeri. La differenza tra le due soluzioni concerne la navigabilità dell'associazione. Nel primo caso la classe Pullman contiene un reference a ogni passeggero contenuto nel pullman, ma il passeggero non contiene il reference al pullman in cui si trova. Questa soluzione è preferibile quando l'associazione deve *solo* essere percorsa in questa direzione, infatti dato un passeggero, per sapere su quale pullman si trova devo scorrere tutti i pullman e controllare che il passeggero non sia all'interno. Nel secondo caso il passeggero contiene il reference al pullman in cui si trova, quindi dato un passeggero è semplicissimo sapere il pullman dove si trova. Lo svantaggio è in fase di aggiornamento dei passeggeri contenuti in un pullman: bisogna garantire che i reference contenuti nella classe pullman e il reference nella classe passeggero siano consistenti. Per esempio per muovere un passeggero `p` da un `pullman1` a un `pullman2` bisogna eseguire il seguente aggiornamento.

```
pullman1.delPasseggero(p);
pullman2.addPasseggero(p);
p.changePullman(pullman2);
```

Nota che senza la terza istruzione si raggiungerebbe una situazione di inconsistenza. La traduzione in Java viene lasciata al lettore, seguendo le indicazioni date nell'introduzione.

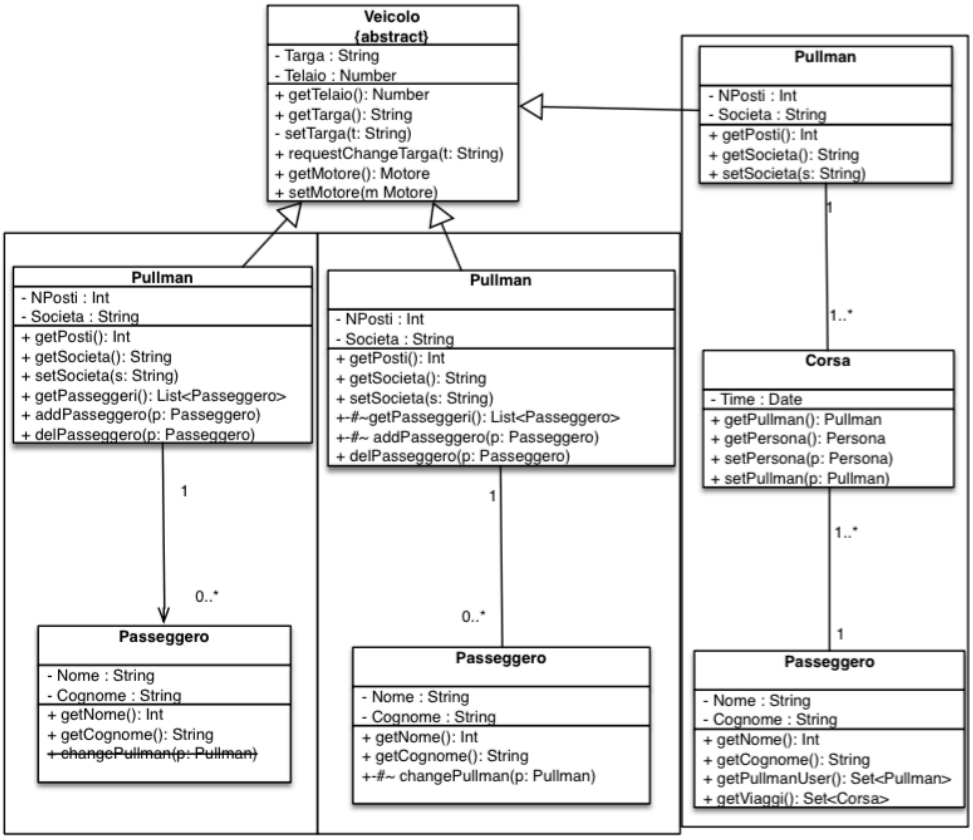

Figure 8: Pullman passeggero.

3.2 Esercizio: Giornalisti

Esercizio 2:

- *R1*: Nella redazione di una testata giornalistica ci sono tre tipi di giornalisti: gli editori, i reporter, ed i fotografi.
- *R2*: Ogni dipendente è caratterizzato da un nome e da un salario e ha diritto ad almeno un benefit (cioè un oggetto che viene concesso in uso al dipendente dall'azienda, ma che è di proprietà dell'azienda).
- *R3*: Ci possono essere vari tipi di benefit: telefono cellulare, macchina fotografica, computer (che può essere o un portatile, o un palmare).
- *R4*: Tra i benefit ci possono anche essere degli apparecchi che hanno funzionalità sia di telefono cellulare che di macchina fotografica.
- *R5*: Un telefono cellulare è caratterizzato da un numero di telefono, e offre la funzionalità di chiamata di un altro numero, e di spedizione di un testo ad un altro telefono.
- *R6*: Se il telefono ha anche funzionalità di macchina fotografica, permette anche di inviare immagini (sequenze di bit).
- *R7*: I fotografi hanno diritto, come benefit, ad esattamente una macchina fotografica.
- *R8*: Ci sono 2 tipi di reporter: i reporter junior e quelli senior.
- *R9*: I reporter junior hanno diritto ad esattamente un telefono cellulare; i reporter senior hanno invece diritto, ad un apparecchio con doppia funzionalitaà celullare/macchina fotografica.
- *R10*: Un reporter può lavorare in coppia con un fotografo, e fa riferimento ad un editore.

Questo esercizio ha lo scopo di proporre diverse soluzioni che soddisfano i requisiti specificati (in generale nella soluzione viene prestata poca attenzione ad attributi e metodi delle specifiche classi, mentre vengono considerate le loro relazioni). In generale possono esserci soluzioni equivalenti allo stesso problema. Le soluzioni in genere differiscono per la loro estendibilità. Infatti lo sviluppatore, di solito, si trova di fronte a una scelta: soddisfare nello specifico ogni minimo requisito (vincoli di relazione tra classi) rendendo l'implementazione molto relazionata al problema che si desidera risolvere o rilassare dei requisiti (vincoli di relazione tra classi) per rendere l'applicazione più astratta possibile e più facilmente estendibile. Il buon progettista è in grado di effettuare la scelta migliore in relazione al problema che si sta risolvendo. In genere possono essere valutate situazioni intermedie che vincolano il progetto ma rendono comunque l'applicazione estendibile (in certe direzioni).
Consideriamo i requisiti riguardanti i `Benefit`: *R3*, *R4*, *R5* e *R6*. La soluzione naive, che rispetta perfettamente le specifiche è presentata in Figura 9. Il requisito *R6* specifica che se il telefono ha la funzione di macchina fotografica fornisce una funzionalità che consente di mandare immagini. Il sospetto che ci sorge è che anche il computer possegga questa funzionalità di mandare immagini.
Partendo da questa ipotesi un progettista potrebbe astrare e realizzare il design presentato in Figura 10. In particolare, lo sviluppatore potrebbe specificare che un `Benefit` ha delle funzionalità. In questo caso un aggregazione o una composizione è il tipo di associazione più adatta. Le funzionalità e i benefit sono specificate in due gerarchie distinte. Ovviamente in questo caso siamo molto più flessibili, potremmo aggiungere la funzionalità chiamata a un particolare computer munito di scheda SIM, ma abbiamo meno garanzie "statiche": potenzialmente una macchina fotografica potrebbe chiamare. In particolare in questo caso è molto importante *come* gli oggetti vengono creati e quindi il creational pattern usato.

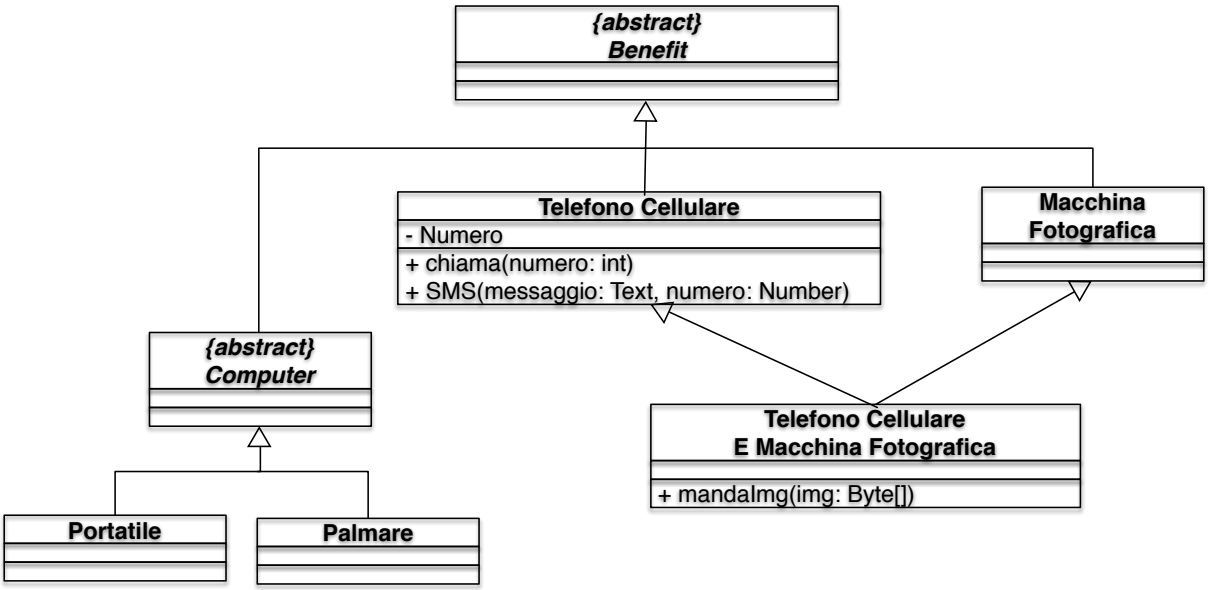

Figure 9: Benefit soluzione 1.

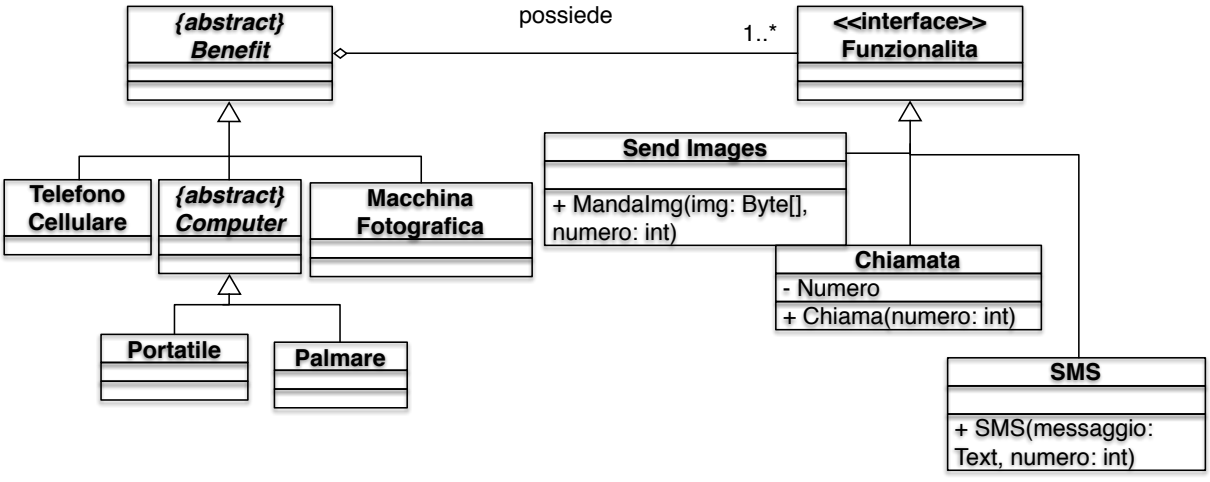

Figure 10: Benefit soluzione 2.

Consideriamo ora i requisiti relativi ai dipendenti: *R1*, *R8*, *R10*. La prima soluzione che ci viene in mente è indicata in Figura 11. Abbiamo una gerarchia di dipendenti e una associazione tra editore e reporter e fotografo e reporter. In questo caso associazione e composizione non sono adeguate.

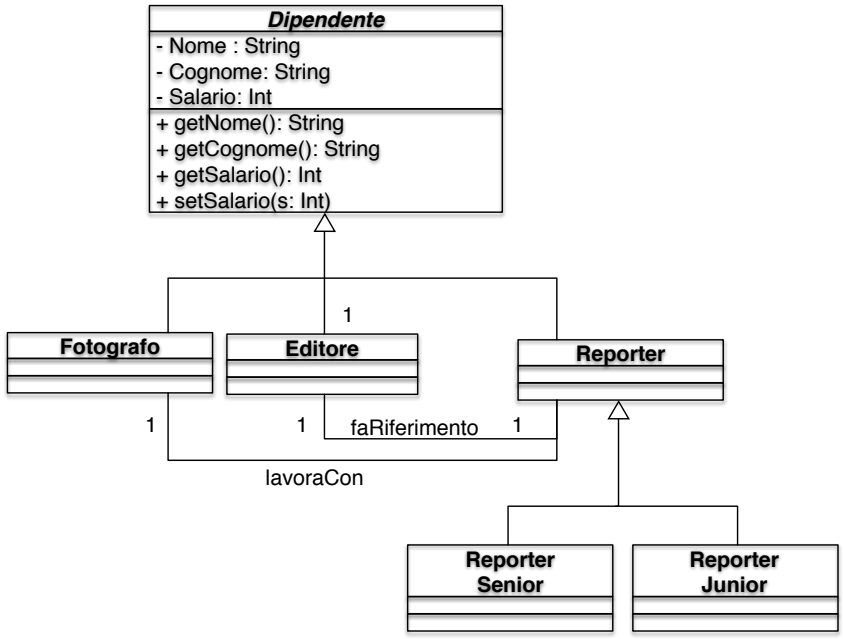

Figure 11: Dipendente.

Potremmo tuttavia cercare di astrarre spostando le relazioni *faRiferimento* e *lavoraCon* verso l'alto (sulla classe dipendente) per rendere le associazioni più generiche. Una possibile soluzione è presentata in Figura 12. La prima osservazione riguardo a questa soluzione riguarda l'enumerazione. L'enumerazione non è la scelta più adatta per il caso in questione, e il suo uso va "limitato". In generale il polimorfismo è uno strumento molto potente e ci garantisce grande estendibilità. Nel caso specifico non possiamo per esempio aggiungere il metodo `getNumeroFotografie()` che è un metodo specifico di un fotografo ma non di un editore o un reporter. Questa limitazione può essere raggirata reinserendo la gerarchia tra i diversi dipendenti e lasciando le relazioni sulla classe dipendente. Tuttavia benchè lo spostamento delle relazioni abbia reso il nostro design più flessibile può portarci a situazioni inconsistenti, dove un fotografo fa riferimento ad un altro fotografo o un fotografo lavora direttamente con un editore. In questo caso la soluzione descritta in Figura 11 è preferibile.

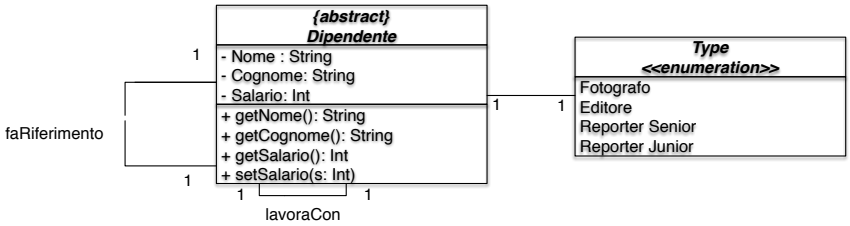

Figure 12: Dipendente.

Infine analizziamo la relazione tra impiegato e benefit: requisiti `R2`, `R7`, `R9`. La prima soluzione che soddisfa le richieste è presentata in Figura 14. Tuttavia, sembra più che naturale considerare rilassamenti di questi requisiti: anche a un reporter mandato in guerra verrà assegnata una macchina fotografica. Per questo è possibile porre un unica associazione tra giornalista e benefit, e forzare le specifiche relazioni per esempio mediante un opportuno creational pattern (la specifica del class diagram corrispondente viene lasciata al lettore).

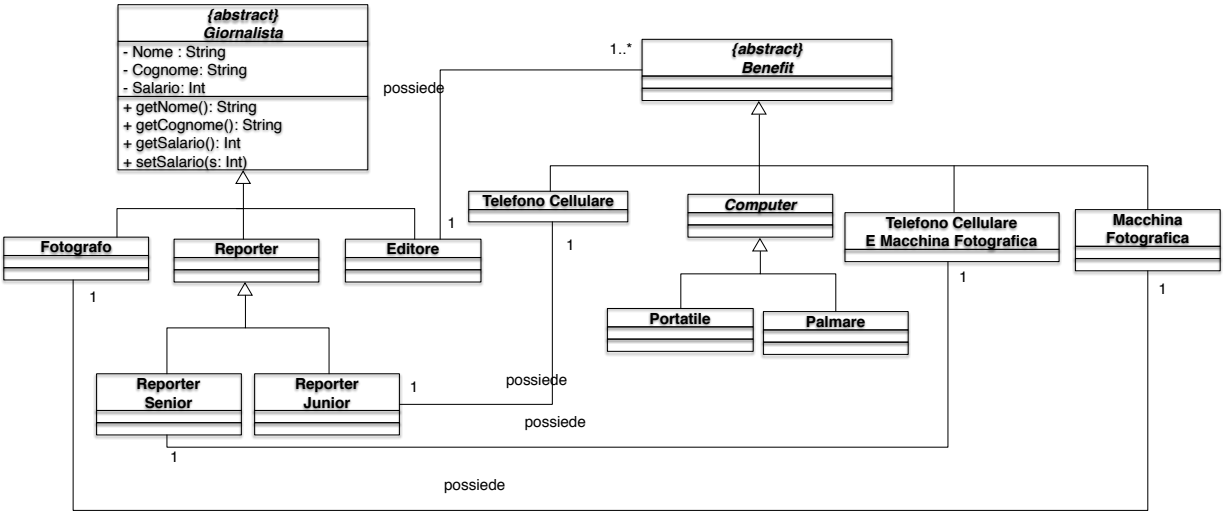

Figure 13: Dipendente.

3.3 Esercizio: Rete informatica

Esercizio 3: Disegnare un diagramma delle classi UML che rappresenti una rete di computer.

- Questa si compone di nodi, i quali possono essere di due tipi: host e router.
- Gli host sono connessi ad esattemente un router, mentre i router possono essere connessi ad un numero qualunque di host e ad almeno un altro router.
- I nodi di una rete possono essere collegati tra loro mediante link fisici.
- Un link fisico può collegare più host e più router tra loro.
- Ogni connessione tra nodi della rete e link fisici e caratterizzata da un indirizzo IP.
- Un host nella rete può offrire dei servizi.
- Ogni servizio, su un certo host, è caratterizzato da una porta.
- Inoltre, ogni servizio si caratterizza per il tipo di protocollo su cui è trasportato, che può essere TCP o UDP.

La soluzione è presentata in Figura 14. Il commento e l'analisi della soluzione è lasciata al lettore.

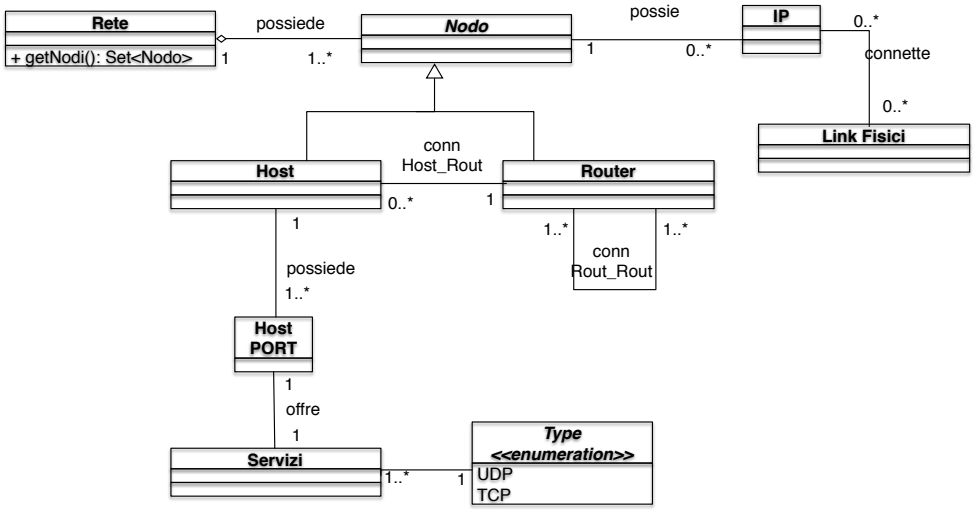

Figure 14: Dipendente.

4 Esercizi per casa

- implementare le soluzioni proposte.
- nel caso in cui venga consigliato di utilizzare un creational pattern pensare ed implementare la soluzione utilizzando il pattern considerato adeguato.