



Ingegneria del Software

---

12/02/2018

---

---

# Correzione compito in classe

---

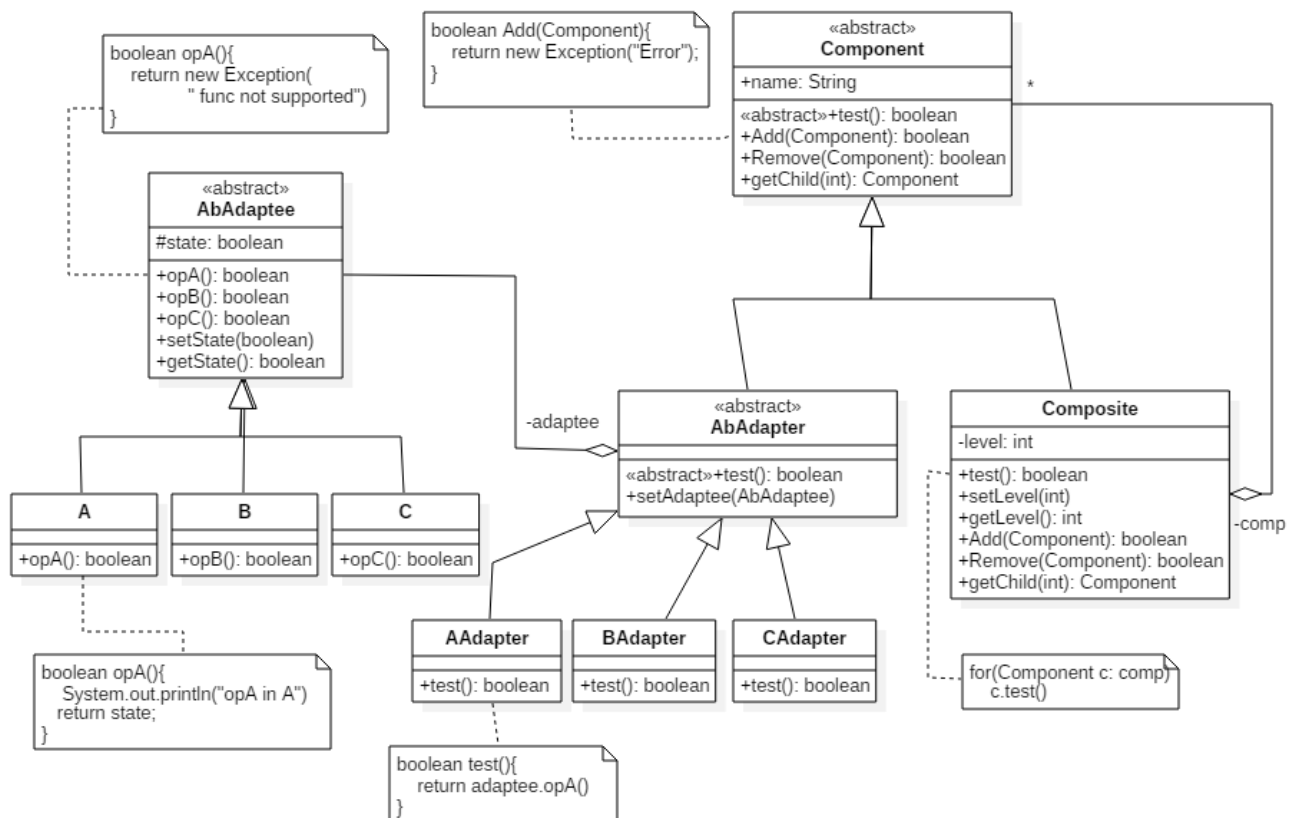
---

Emanuele Vivoli

## Es 1

**1.1** Si definisca la struttura di classi e interfacce Java che realizzano lo schema usando il pattern Composite per rappresentare a gerarchia e il pattern Adapter per adattare i componenti pre-esistenti all'interfaccia attesa per la composizione.

## Class Diagram (Object-Adapter)



---

## 1.2 *Si dettano frammenti di codice che caratterizzano l'implementazione*

---

### AbAdapter

La classe astratta AbAdapter rappresenta per il pattern Composite la struttura di Leaf, mentre per il pattern Adapter la classe Adapter (in questo primo caso con Object-Adapter).

Poiché ho deciso di rendere Component una classe astratta non si ha necessità di definire un'implementazione per i metodi Add(), Remove() e getChild(). In realtà non viene specificata nessuna implementazione anche per test(), in modo che venga ridefinito obbligatoriamente nelle classi concrete che ereditano da questa.

```
1. public abstract class AbAdapter extends Component{
2.     protected AbAdaptee adaptee;
3.
4.     public AbAdapter(AbAdaptee adap, String n){
5.         adaptee = adap;
6.         name = n;
7.     }
8.
9.     public void setAdaptee(AbAdaptee ad){
10.        adaptee = ad;
11.    }
12.
13.    @Override
14.    public abstract boolean test();
15. }
```

### AAdapter

Classe concreta implementazione di AbAdapter, implementa unicamente il metodo test() facendo forwarding sull'oggetto adaptee.

```
1. // es relativo a AAdapter
2. @Override
3. public boolean test(){
4.     try {
5.         return adaptee.opA();
6.     } catch (Exception ex) {
7.         System.out.println("operazione non consentita");
8.     }
9.     return false;
10. }
```

### AbAdaptee

Questa classe astratta rappresenta le classi che dobbiamo adattare al Target (Component), è fatta astratta per poter definire tutti e tre i metodi fasulli opA(), opB(), opC(). Ne verrà data un'implementazione solo nelle classi concrete A, B e C.

```
1. public abstract class AbAdaptee {
2.     protected boolean state;
3. }
```

```

4.      // getter and setter della variabile stato
5.
6.      public boolean opA() throws Exception{
7.          throw new Exception("Operazione non supportata! (try with opB() or opC())");
8.      }
9.
10.     // simile per opB() e opC()
11. }

```

## A extends AbAdaptee

```

1.      // in A class
2.
3.      @Override
4.      public boolean opA(){
5.          //System.out.println("A: opA()");
6.          return getState();
7.      }

```

## Component

La classe astratta Component da un'implementazione “fake” dei metodi children-related, in modo da dare la responsabilità di definirne una reale implementazione solo nel Composite.

```

1. public abstract class Component {
2.     public abstract boolean test();
3.
4.     public void Add(Component c) throws Exception{
5.         throw new Exception("Metodo non supportato nelle classi AbAdapter");
6.     }
7.
8.     // stessa implementazione per Remove() e getChild()
9. }

```

## Composite

Il Composite ha la responsabilità di trovare e notificare un errore, e specificare a che livello questo accade. Viene tutto gestito all'interno del metodo `test()` e `Add()`.

```

1. @Override
2. public boolean test() {
3.     boolean res = true, app = true;
4.     for(Component c: comp){
5.         app = c.test();
6.         if(!app && c instanceof AbAdapter)
7.             System.out.println(((AbAdapter)c).name+"Err: lev."+level-1);
8.         res &= app;
9.     }
10.    return res;
11. }
12.
13. public void setLevel(int levelSup){
14.     level = levelSup;
15.     for(Component c: comp){
16.         if(c instanceof Composite)
17.             ((Composite) c).setLevel(level -1);
18.     }
19. }
20.

```

```

21. @Override
22. public void Add(Component c){
23.     comp.add(c);
24.     if(c instanceof Composite){
25.         int max = 0;
26.         for(Component e: comp){
27.             if(e instanceof Composite)
28.                 if(((Composite) e).getLevel()>max)
29.                     max = ((Composite) e).getLevel();
30.         }
31.         setLevel(max + 1);
32.     }
33. }
34. }

```

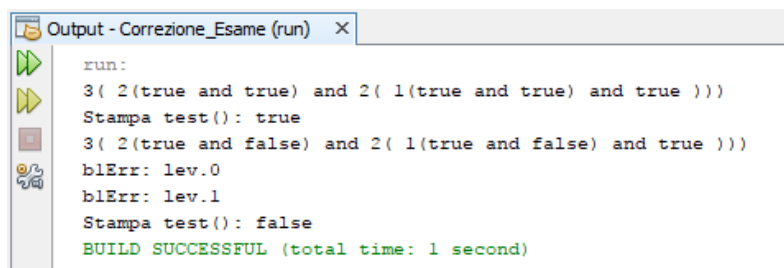
## Test

```

1. public class Client {
2.
3.     /**
4.      * @param args the command line arguments
5.      */
6.     public static void main(String[] args) throws Exception {
7.         A a = new A(true);
8.         B b = new B(true);
9.         C c = new C(true);
10.        Component a1 = new AAdapter(a, "a1");
11.        Component b1 = new BAdapter(b, "b1");
12.        Component c1 = new CAdapter(c, "c1");
13.        Component C1 = new Composite("C1");
14.        C1.Add(a1);
15.        C1.Add(b1);
16.        Component C2 = new Composite("C2");
17.        C2.Add(C1);
18.        C2.Add(c1);
19.        A aa = new A(true);
20.        Component C3 = new Composite("C3");
21.        C3.Add(C2);
22.        Component C12 = new Composite("C12");
23.        C12.Add(a1);
24.        C12.Add(b1);
25.        C3.Add(C12);
26.        System.out.println("Stampa test(): "+C3.test());
27.        b.setState(false);
28.        System.out.println("Stampa test(): "+C3.test());
29.    }
30.
31. }

```

## Output



```

Output - Correzione_Esame (run) X
run:
3( 2(true and true) and 2( 1(true and true) and true ))
Stampa test(): true
3( 2(true and false) and 2( 1(true and false) and true ))
blErr: lev.0
blErr: lev.1
Stampa test(): false
BUILD SUCCESSFUL (total time: 1 second)

```

---

### *1.3 Si discutano vantaggi e svantaggi conseguenti all'adattamento in forma Object-Adapter o Class-Adapter in riferimento al caso specifico.*

---

Ho usato per semplicità il pattern Adapter nella forma dell'Object-Adapter, che mi consente di usare la composizione piuttosto che l'ereditarietà fra le classi `AbAdapter` e `AbAdaptee`.

Ho dunque proseguito realizzando la Classe `Component` come «abstract», descrivendo nel suo codice le implementazioni “fake” per i metodi `children-related`.

I vantaggi di questa scelta sono dunque una maggior dinamicità, poiché posso aggiungere come `Adaptee` alla classe `Adapter` uno qualsiasi degli oggetti `A`, `B`, `C` senza preoccuparmi di che oggetto si tratti, e senza conoscere cosa implementano le operazioni `opA()`, `opB()`, `opC()`; consentendo anche agli oggetti `A`, `B`, `C` di fare un override dei metodi in `Adaptee`, con la sicurezza che grazie al `dynamic lookup method` si vada a richiamare sugli oggetti proprio il metodo desiderato. Inoltre la composizione non “spreca l'unica pallottola” di ereditarietà che il Java ci permette, dandoci la possibilità di estendere una classe qualora lo volessimo. Questo esempio ci dà conferma di ciò chiedendoci che `Adapter` sia proprio una `Leaf` di una struttura `Composite`. A questo proposito possiamo utilizzare `Component` come classe astratta proprio grazie alla composizione che ci lascia liberi di ereditare da un'altra classe.

Ovviamente questa scelta porta con sé anche dei difetti, come il fatto di dover istanziare ogni volta un oggetto concreto (che nel nostro caso sarebbe potuto essere `Singleton`), dover fare forwarding dei metodi (che però è il male minore) ed infine il `Self-Problem`: la classe `Adaptee` potrebbe eludere la classe `Adapter` (`Wrapper`) in un contesto di `call-back` dove restituisce un riferimento a se stessa e tutte le interazioni successive riguardano solo essa, e non la classe `Wrapper`, come dovrebbe essere.

Vediamo dunque il caso di un'implementazione mediante il `Class-Adapter`, e discutiamo vantaggi e svantaggi.

Se avessi voluto usare il `Class-Adapter` avrei certamente dovuto estendere `AbAdapter` da `AbAdaptee` e dunque implementare `Component` come interfaccia. Inoltre questa tecnica introduce il problema dell'ereditarietà per la classe `Adaptee`, cioè le classi derivate da `Adaptee` che implementano un override dei metodi di `Adaptee` non possono essere viste dal `lookup method`, poiché non vi è nessuna istanza da “ricercare”. Come soluzione però si ha che `Adapter` è una classe `Adaptee` (per via dell'`extends`) e dunque è capace di definire lei stessa dei metodi che fanno override. Sarà dunque necessario disporre `Adapter` dei metodi overrides delle classi figlie di `Adaptee`.

---

### 1.4 *Si discutano vantaggi e svantaggi conseguenti alla realizzazione del Composite come interfaccia o classe astratta.*

---

Nell'esempio del compito si è scelto di usare il Composite come classe astratta poiché in questo modo, come ripetuto svariate volte, si può fornire la classe Leaf delle implementazioni dei metodi che non le sono utili. Infatti avremmo potuto realizzare Composite come interfaccia ma nasce la necessità di implementare i metodi Add(), Remove(), getChild() anche in AbAdapter ma con una implementazione "fake" che lancia un'eccezione (poiché non si può aggiungere un figlio ad una foglia!); se si fosse certi di disporre della tecnologia di Java 8 si potrebbe utilizzare una interfaccia e definire dei metodi di default che utilizzerà direttamente l'AbAdapter (in realtà ad usufruirne saranno tutte le classi concrete che ereditano da AbAdapter).

Un'altra conseguenza che deriva dalla scelta di Component come interfaccia è che in questo modo non si permette a Composite di estendere un'altra classe, infatti una interfaccia non può estendere una classe, né astratta né concreta. Il caso specifico può essere quello in cui si necessita di un Monitor che tramite il pattern Observer possa osservare lo stato di oggetti composti. Component deve dunque essere di tipo Observable, che però è una classe astratta. Dunque Component non può essere un'interfaccia, ma solo classe astratta.

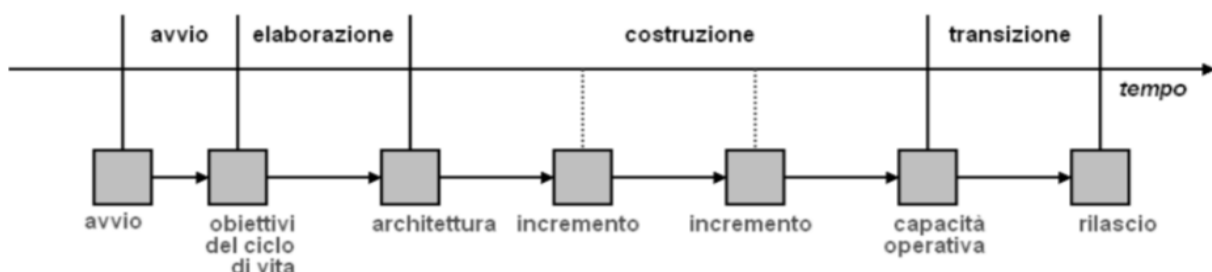
#### Nota

In questo esercizio ho sempre considerato l'Adaptee (nel nostro caso AbAdaptee) come una classe astratta già esistente, e dunque non un'interfaccia (poiché non trovavo il senso di dover definire i metodi non supportati per 3 volte con la stessa implementazione).

Questo contrariamente alla targhetta «implementation» presente nel Class Diagram del pattern Adapter (Class-Adapter), che nel compito supposevo intendesse «implements» mentre mi sono successivamente accorto del vero significato, ovvero «ereditarietà di implementazione», cioè classe e non interfaccia. (altrimenti sarebbe stato «ereditarietà di interfaccia»).

## Es 2

Unified Process è un modello di sviluppo proposto da chi ha definito UML. È iterativo ed incrementale: ogni iterazione è composta da requisiti, analisi, progettazione, implementazione e test. Inoltre è pilotato dai casi d'uso (requisiti). Il ciclo di vita è articolato in quattro fasi e ciascuna fase ha un Milestone.



1. **AVVIO**  
Obiettivi e delimitazione del progetto, Studio di fattibilità, Definizione requisiti essenziali, Studio dei rischi maggiori, Avvio con consenso manageriale.
2. **ELABORAZIONE**  
Si definiscono l'80% dei casi d'uso, Baseline eseguibile: architettura di base, Pianificazione della costruzione, Valutazione delle risorse necessarie, Perfezionamento analisi dei rischi, Definizione di attributi di qualità.
3. **COSTRUZIONE**  
Completamento dei requisiti e del modello dell'analisi, Evoluzione della Baseline in coerenza con la architettura iniziale, Capacità operativa iniziale: beta-test.
4. **TRANSIZIONE**  
Siti utente, manuali, preparazione, consulenza. Correzione di difetti, adattamento. Debriefing.

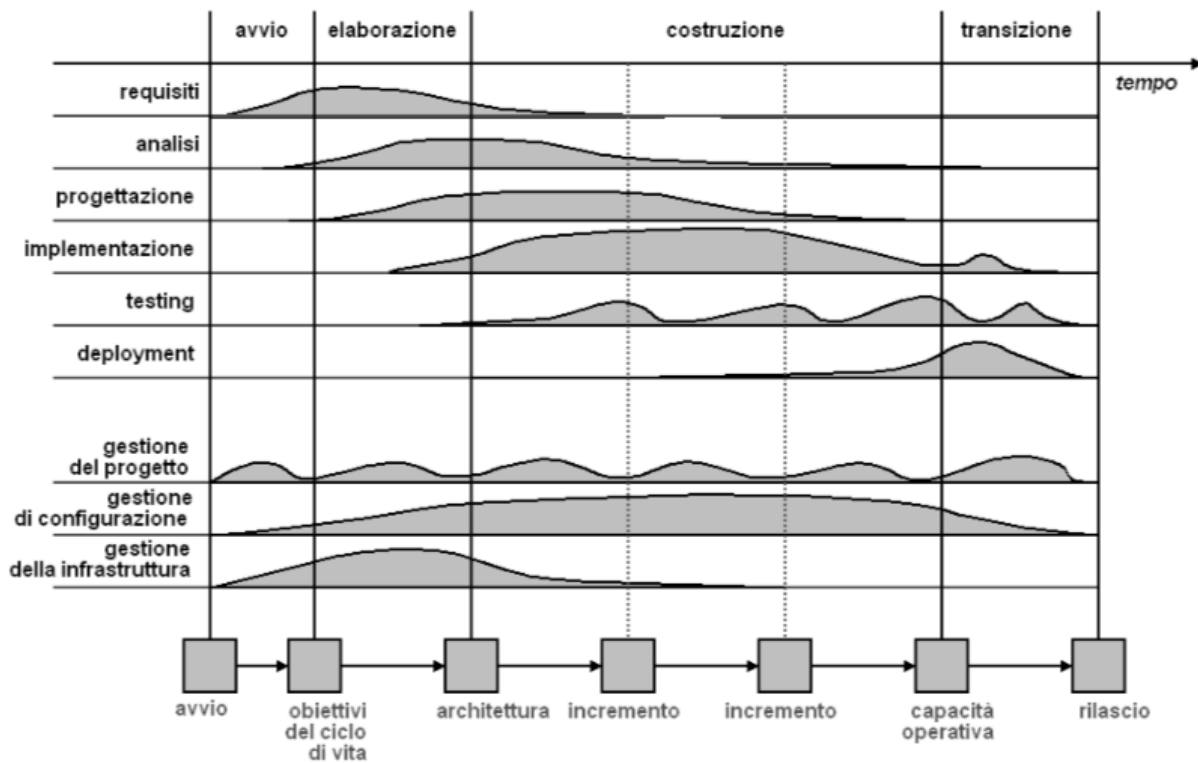
È un modello iterativo poiché si ripetono ad ogni fase le funzioni di Waterfall (è infatti una evoluzione del modello a cascata):

- **REQUISITI**  
In questa fase viene prodotto un documento dei requisiti SRS (Software Requirement Specification) ed i principali UC (Use Case);
- **ANALISI**  
In questa fase si cerca di ottenere un modello Concettuale e si rivisitano gli schemi degli UC sulla base degli attori del modello Concettuale;
- **PROGETTAZIONE**  
Si cerca di minimizzare il gap che vi è fra i documenti dell'analisi e il progetto;
- **IMPLEMENTAZIONE**  
Vengono fatte in questa fase le scelte di implementazione delle unità e si studia come realizzarle (es. vengono scelti quali Design Pattern utilizzare);
- **TEST**



I prodotti attesi che si hanno al termine di ciascun processo (poiché ogni fase è vista come un piccolo processo) sono quelli delineati precedentemente e che si trovano a dover essere approvati al raggiungimento di ogni Milestone.

Dalla tabella dei flussi possiamo notare come in fase di AVVIO sarà prevalente il flusso di “Requisiti” piuttosto che altri tipi di processi; mentre nelle fasi finali, come quella di TRANSIZIONE, si avrà prevalenza da parte del flusso di “teste pubblicazione” (ed anche emissione dei manuali e gestione della struttura).



## Es 3

---

---

*Si consideri il caso di un componente SW per la gestione delle attività di un laboratorio di ricerca.*

---

Viene richiesto di descrivere la logica di dominio attraverso un class diagram in prospettiva di implementazione/specifica orientato ad abilitare funzioni CRUD di entità, dei diversi tipi di entità.

Innanzitutto si cerca di esporre quali sono le entità principali che hanno responsabilità di CRUD su una qualche classe di oggetti:

### Laboratorio di ricerca

Dal testo si capisce che in un laboratorio di ricerca vi partecipano più Ricercatori, vi sono diverse Attività (come per esempio Progetto di Ricerca) e diversi Fondi sul quale basare le spese di tali attività.

Si decide dunque di dare la responsabilità al Laboratorio di Ricerca di creare, modificare, cercare e cancellare un Ricercatore dalla lista dei Ricercatori associati ad un certo laboratorio.

Questo ha inoltre la responsabilità di creare le attività alle quali possono partecipare i Ricercatori (sotto diverse forme) e creare i Fondi che verranno associati ad alcune Attività (le attività di tipo Progetto di Ricerca).

A seguito, il Laboratorio può anche fare operazioni di CRUD su certe spese specifiche dei Progetti, come Mobilità e Contratti extra Università, poiché sarà il Progetto a chiederne l'approvazione al Laboratorio, ma solo questo, a seguito di un'approvazione, può creare queste Spese e associarne il Ricercatore.

### Ricercatore

Il ricercatore terrà memoria di tutte le attività a cui partecipa, dividendole in `ArrayList<Attività>` differenti in base alle tipologie di partecipazione.

Il ricercatore può dunque solo iscriversi o ritirarsi da un Attività. Non può assolutamente registrarsi come responsabile di un'attività.

Dunque la modifica dell'`ArrayList<Attività>` `Att_resp` è disabilitata per il Ricercatore, ma è fornita all'Attività stessa che al momento della creazione passa ad un metodo di Ricercatore (con l'aggiunta di un token fornitogli dal Laboratorio di Ricerca) un riferimento a sé stessa in modo da aggiungere l'Attività alla lista delle Responsabilità del Ricercatore.

### Attività

L'attività ha i metodi per poter eseguire operazioni CRUD sugli `ArrayList` di Ricercatori, passando ai metodi necessari un oggetto di tipo Ricercatore.

Ha inoltre i metodi CRUD per poter creare delle Annotazioni, nelle loro forme più diverse. Il fatto che annotazione possa essere di molte tipologie ci fa pensare che debbano risiedere nella struttura dei metodi capaci di creare un'Annotazione piuttosto che un'altra. In prima analisi possiamo concludere che il Ricercatore responsabile di un'Attività sia addetto a creare dei tipi di Annotazioni (come Scadenza, Stato Avanzamento, Obbiettivi Generali) mentre la creazione e gestione di Annotazioni come Difficoltà e OpenIssue sarà possibile anche dai Ricercatori che vi partecipano, ma non da quelli Informati. Un'analisi un po' più concreta ci suggerisce di riporre il metodo vero e proprio per la creazione di varie tipologie di Annotazioni proprio in Attività (es. `createOpenIssue(pers: Ricercatore, title: String)` ).

## Progetto di Ricerca

L'ultima classe "principale" in questa applicazione è proprio Progetto di Ricerca. Questa non è altro che una classe concreta di Attività, che riprenderà l'attributo *titolo:String* definito *protected* nella classe base, e definirà un attributo tipo poiché, come specificato dal testo, tale classe può essere classificata in 3 maniere differenti: Competitivi, Commesse trasferimento, Contributi.

Il Progetto di Ricerca in quest'applicazione ha la responsabilità di gestire un Fondo, che le viene assegnato in prima istanza dal Laboratorio di Ricerca, e successivamente può essere modificato sempre con un'appropriata gestione degli accessi (bisogna essere identificati, quindi magari si permette solo con il passaggio di un token).

Il Progetto permette di creare diversi tipi di oggetti Spesa, ed anche in questo caso i metodi per la creazione risiedono nella classe Progetto di Ricerca, anche se dal class diagram risultano solo in maniera generica (es: *createSpesa(in double): Spesa* dovrà essere in realtà *createSpesaStrumentazione(Strumento, int):Spesa*; dove Strumento indica una classe non raffigurata nel diagramma che contiene attributi "costo, descrizione, ecc..." e indica lo strumento che si vuole acquisire; mentre *int* non è altro che il numero di occorrenze che si necessita).

*Nel diagramma la rappresentazione dettagliata dei metodi e delle classi è stata evitata per poter dare una visuale d'insieme e lasciare questi paragrafi dare delle spiegazioni. Per evitare dunque l'OVER DESIGN.*

