Ingegneria del Software

Correzione esame 22 gennaio 2018

Gabriele Bruni

23 Gennaio 2019

1 Esercizio 1

Nel testo del primo esercizio si richiedeva di combinare i design pattern **Observer** e **Strategy** per realizzare un meccanismo di adattamento. Entrambi appartengono alla lista dei *behavioral design pattern*. L' Observer permette ad un oggetto (il *Subject*) di mantenere una lista di oggetti (gli *Observers*) e notificare loro automaticamente l'eventuale cambiamento del suo stato. Lo Strategy permette invece di modificare dinamicamente gli algoritmi usati da un applicazione a runtime. Il testo chiedeva di registrare il *Context* dello Strategy come Observer del Subject, e di adattare di conseguenza l'algoritmo utilizzato in funzione dello stato di quest'ultimo.

Di seguito si descrive la struttura con cui vengono composti i due pattern attraverso l'uso di un class diagram:

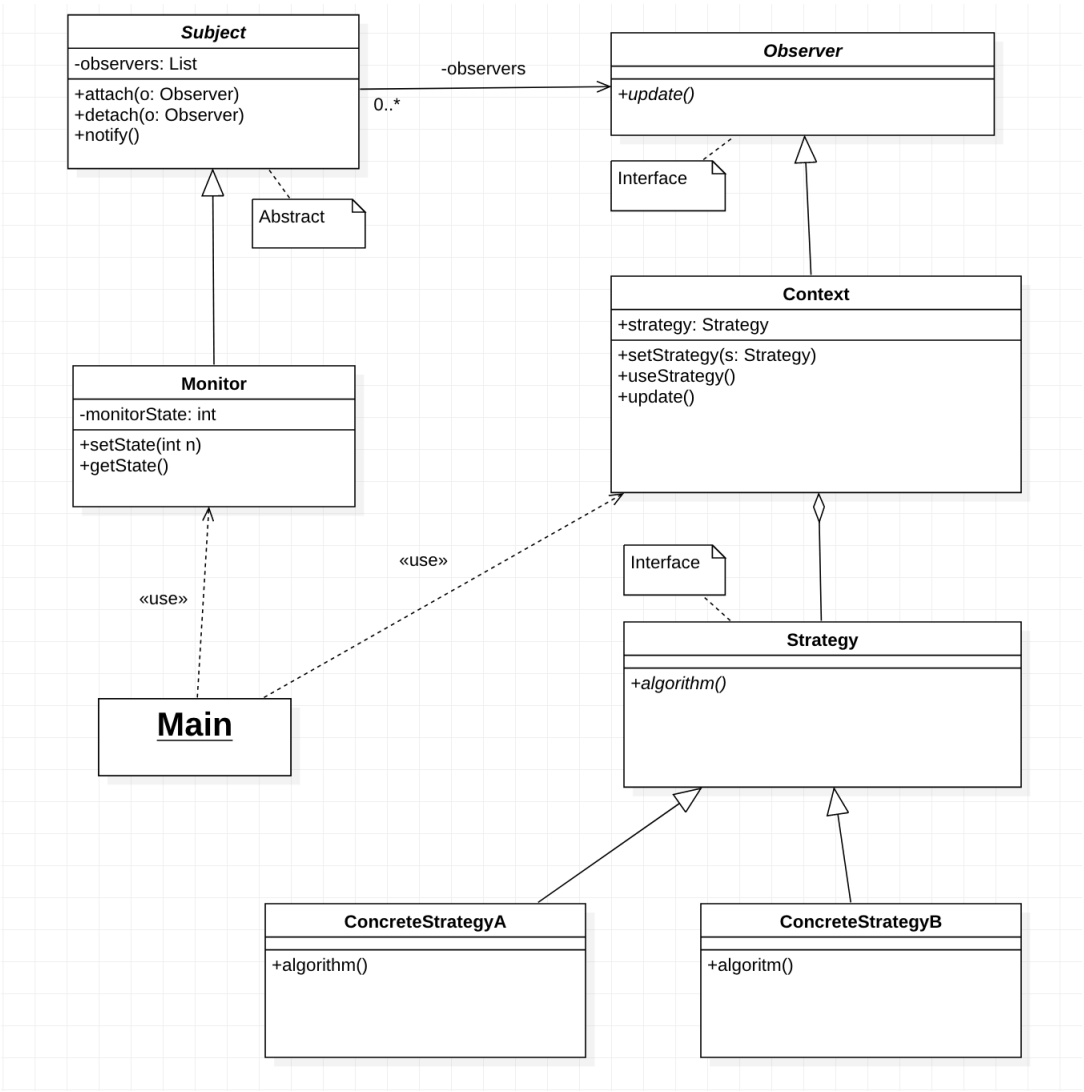

Figure 1: ClassDiagrams

L'esercizio chiedeva inoltre di illustrarne il funzionamento in uno scenario di uso tipico. Di seguito riportiamo il sequence diagrams in uno scenario di test:

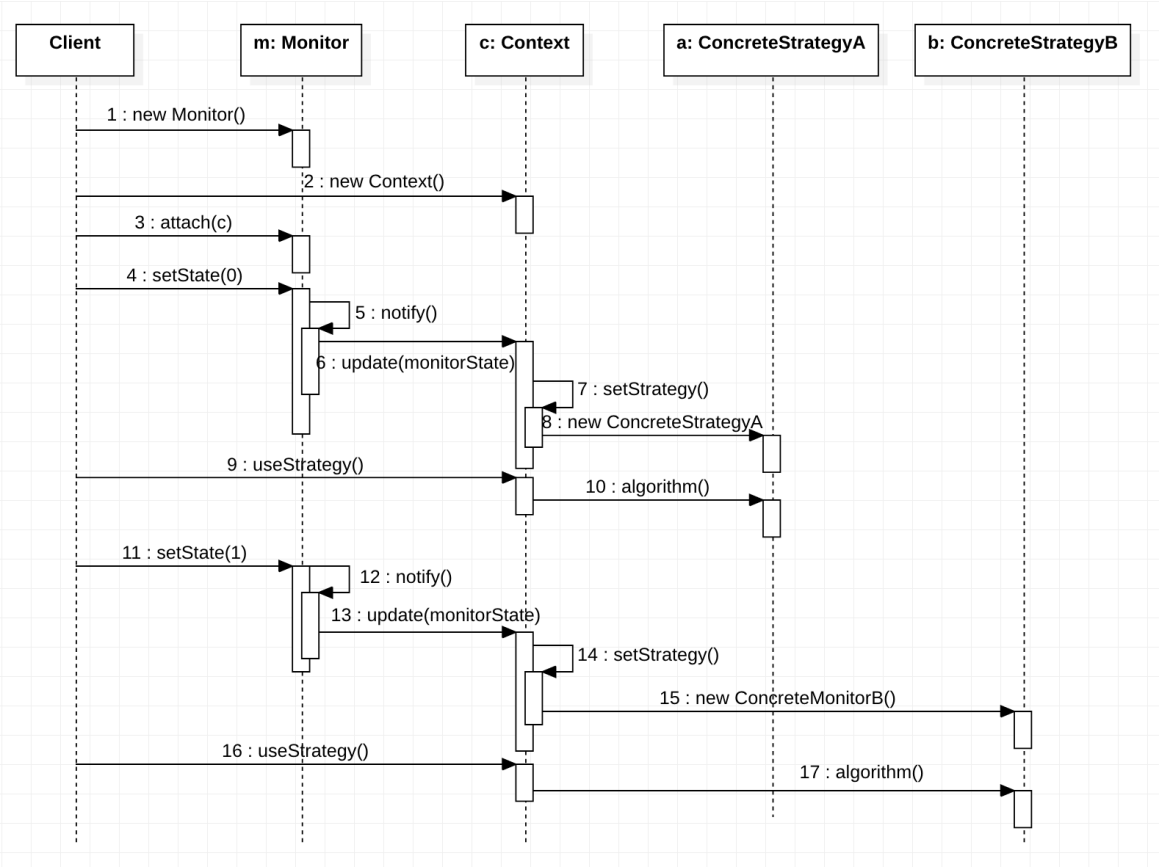

Figure 2: Sequence Diagrams

L'ultima richiesta era relativa all'implementazione in java dello scenario d'uso sopra riportato. Di seguito riportiamo il codice e l'output prodotto:

```
package com.company;

import java.util.ArrayList;
import java.util.Iterator;

public class Main {

    public static void main(String[] args) {
        // write your code here
        Monitor m = new Monitor();
        Context c = new Context();

        m.attach(c);

        try {
            m.setState(0);
            c.useStrategy();
```

```
            m.setState(1);
            c.useStrategy();

            m.setState(3);
            c.useStrategy();
        }catch (IllegalStateException e){
            System.out.println(e.getMessage());
        }


    }
}

abstract class Subject {
    protected ArrayList <Observer> observers = new ArrayList<>();


    public void attach(Observer o){
        observers.add(o);
    }

    public void detach(Observer o){
        observers.remove(o);
    }

    abstract void notifyObserver() throws IllegalStateException;

}

class Monitor extends Subject{
    private int monitorState;

    public void setState(int newState) throws IllegalStateException {
         monitorState = newState;
         notifyObserver();
    }

    public int getState(){
        return monitorState;
    }

    @Override
    public void notifyObserver() throws IllegalStateException {
        Iterator it = observers.iterator();

        while (it.hasNext()){
            Observer o = (Observer) it.next();
            o.update(monitorState);
        }
    }
}
```

```
interface Observer{
    public void update(int state) throws IllegalStateException;
}

class Context implements Observer{
    private Strategy strategy;

    @Override
    public void update(int state) throws IllegalStateException {  //add push vs pull co
        if(state<0 || state>1){
            throw new IllegalStateException();
        }
        else if(state == 0){
            setStrategy(new ConcreteStrategyA());
        }
        else if(state == 1){
            setStrategy(new ConcreteStrategyB());
        }
    }

    private void setStrategy(Strategy s){
        strategy = s;
    }

    public void useStrategy(){
        strategy.algorithm();
    }
}

interface Strategy{
    public void algorithm();
}

class ConcreteStrategyA implements Strategy{
    @Override
    public void algorithm() {
        System.out.println("You are now using strategy A!");
    }
}

class ConcreteStrategyB implements Strategy{
    @Override
    public void algorithm() {
        System.out.println("You are now using strategy B!");
    }
}

class IllegalStateException extends Exception{
    public IllegalStateException(){
        super("ERROR: state allowed are 0 and 1");
    }
```

}

Output:

```
You are now using strategy A!
You are now using strategy B!
ERROR: state allowed are 0 and 1
```

1.1 Note sull'implementazione

Per quanto riguarda l'implementazione dell'Observer, è venuto più naturale implementarlo secondo una strategia *push*: il Monitor invia, ad ogni sua variazione, il suo stato a tutti gli Observer. Si è assunto infatti che tutti gli Observer necessitano di adattare la propria strategia per tutti i cambiamenti di stato del monitor: l'implementazione con metodologia push consente di evitare un ulteriore call-back al server per ricevere lo stato, garantendo prestazioni migliori in questo caso d'uso. Si è scelto, dove possibile, di utilizzare interfacce invece di classi astratti, per evitare o problemi che l'ereditarietà si porta dietro. Si è deciso di utilizzare una classe astratta solo per l'implementazione di Subject: ogni concreta implementazione di Subject avrebbe utilizzato lo stesso codice per le funzioni `attach()` e `detach()` se implementato con interfacce. La scelta della classe astratta permette l'eventuale riuso di codice. Inoltre il metodo `notify()` richiedeva l'utilizzo dello stato di ogni singolo Subject. Si è deciso di non implementare nella classe astratta tale metodo ed il relativo attributo `monitorState` per garantire un livello di visibilità minore a quest'ultimo (se definito nella classe astratta, tale metodo avrebbe dovuto avere visibilità `protected` invece che visibilità `private`.

2 Esercizio 2

L'eXtreme Programming è una metodologia di sviluppo del software adattata in quei contesti dove i requisiti possono cambiare rapidamente. Il team di sviluppo è solitamente formato da 4 programmatori, 1 team leader, ed 1 responsabile di progetto. Ruolo centrale riveste il committente, che deve fornire continui feedback sulla realizzazione del progetto. Due programmatori sviluppano su una singola workstation: il primo si occupa dell'aspetto "modellistico", il secondo scrive il codice. Le release vengono rilasciate una volta circa ogni due settimane, in modo da fronteggiare repentivamente i cambi di requisiti. Tutti i membri del team di sviluppo devono avere accesso all'intero codice, sono incoraggiati a revisionarlo ed eventualmente a migliorarlo.Si cerca di evitare di produrre documentazione in eccesso, usando prevalentemente use-case diagrams come fonte di documentazione. La settimana lavorativa è strutturata in modo da non superare le 40 ore e si privilegiano come luoghi di lavoro gli "open-space", in modo che tutti possano scambiarsi facilmente informazioni. Si da molto importanza alla fase di test del codice ed ai test stessi, che vengono scritti primi di implementare il codice sorgente vero e proprio del prodotto. Tuttavia tale metodologia non sempre è applicabile: il committente spesso non si trova nello stesso luogo di lavoro per fornire feedback e non è detto che siano presenti open-space. Si creano inoltre difficoltà contrattuali.

3 Esercizio 3

Di seguito diamo una descrizione generale delle classi identificate per abilitare i casi d'uso identificati nella descrizione del problema. Si immagina che l'applicazione che supporta le attività di gestione delle attività presso un laboratorio didattico fornisca una interfaccia di login attraverso il quale gli attori inseriscono le proprie credenziali. Una volta inserite le credenziali, il sistema provvederà a rendere disponibili automaticamente le funzioni che abilitano i casi d'uso richiesti da ciascuno attore. In particolare le classi `Tecnico`, `Tecnico di laboratorio`, `TecnicoResponsabileDiElaborato`, e `Docente`, implementano i vari casi d'uso. Di seguito forniamo una descrizione di queste classi:

- **Docente**: il Docente di un corso può registrare un nuovo elaborato tramite il metodo `addElaborato()`, il quale crea un nuovo oggetto `Elaborato`, aggiunge gli studenti che partecipano all'elaborato tramite il metodo `addStudenti()`, e definisce il programma di lavoro svolto tramite il metodo `addProgrammaDiLavoro()`. Al termine di tutto ciò, `addElaborato()` chiama il metodo `insertElaborati()` di `Progetto` e registra così l'elaborato all'interno di un progetto. Per registrare la chiusura di un elaborato al suo completamento, il docente tramite il metodo `chiudiElaborato()` ricercherà fra i progetti l'elaborato precedentemente registrato e ne modificherà la variabile booleana `isCompleted`. Una volta che questa è posta al valore `true`, non sarà più possibile modificare l'elaborato. Il campo `ruolo`, descrive i vari ruoli ricoperti dal docente. Ad esempio, un professore potrebbe assumere il ruolo di responsabile scientifico sia per un progetto, sia per un assegnista e, contemporaneamente potrebbe anche fare da Advisor per un assegnista.

- **Tecnico**: un Tecnico può ricercare in quali elaborati sono in uso quali dispositivi semplicemente attraverso i metodi `ricercaElaborato()` e `ricercaDispositivo()` i quali si interfacciano con `DbRecord` e restituiscono i record contenenti l'elaborato o il dispositivo registrato. Il tecnico può inoltre contattare il docente responsabile dell'elaborato tramite il metodo `contattaDocente()`, il quale utilizza il metodo precedentemente definito `ricercaElaborato()` per accedere al record corrispondente all'elaborato, da qui accedere all'istanza dell'elaborato, ricavare il relativo riferimento al docente responsabile e chiamare il metodo di Docente `getRichiesta()`, il quale inoltra la richiesta al Docente. Il metodo `contattaStudente()` esegue analoghe procedure per contattare lo studente.

- **Tecnico responsabile Elaborato**: il tecnico responsabile dell'elaborato può registrare i dispositvi assegnati a ciascuno studente tramite il metodo `registraDispositivo()`. Questo accede alla lista degli studenti tramite il metodo `getStudenti()` del suo attributo `elaborato`, (che contiene un riferimento all'elaborato di cui è responsabile), e, per ogni Studente, crea un nuovo oggetto `Record` che poi inserisce in `DbRecord`, una sorta di Database dove sono contenuti tutti i Record. Ogni record associa ad uno studente una lista di dispositvi a lui assegnati con relativo periodo di assegnazione (si ipotizza che i dispositivi siano assegnati in blocco e che tutti siano assegnati nello stesso periodo), e l'eventuale attività di formazione svolte dallo studente ai fini della sicurezza. Il tecnico accede alla lista di dispositivi disponibili nel laboratorio tramite il rifermento all'inventario rappresentato dall'entità `Inventario`. L'inserimento di un dispositivo nella lista dei dispositivi assegnati comporta lo spostamento nella lista `dispositviInUso` dell'Inventario. Terminato l'elaborato il tecnico provvederà inoltre a spostare di nuovo manualmente il dispositivo nella lista `dispositiviDiponibili` tramite il metodo `freeDispositivo()`.

- **Tecnico laboratorio**: il tecnico di laboratorio può individuare i dispositivi che non sono stati riconsegnati al termine di un elaborato tramite il metodo `checkDispositivi()`. Questo accede al database dei record `DbRecord`, seleziona i record degli elaborati il cui campo `isCompleted` è settato a true, accede alla relativa lista dei dispositivi, controlla in quale lista dell'inventario sono contenuti tramite il metodo `getList()` del proprio riferimento ad `Inventario`, e restituisce i dispositivi contenuti nella lista `dispositiviInUso`.

  Il tecnico può poi successivamente contattare lo studente tramite il metodo `contattaStudente()` della sua classe base `Tecnico`.

Di seguito riportiamo lo use-case diagrams ed il class diagrams dell'esercizio:

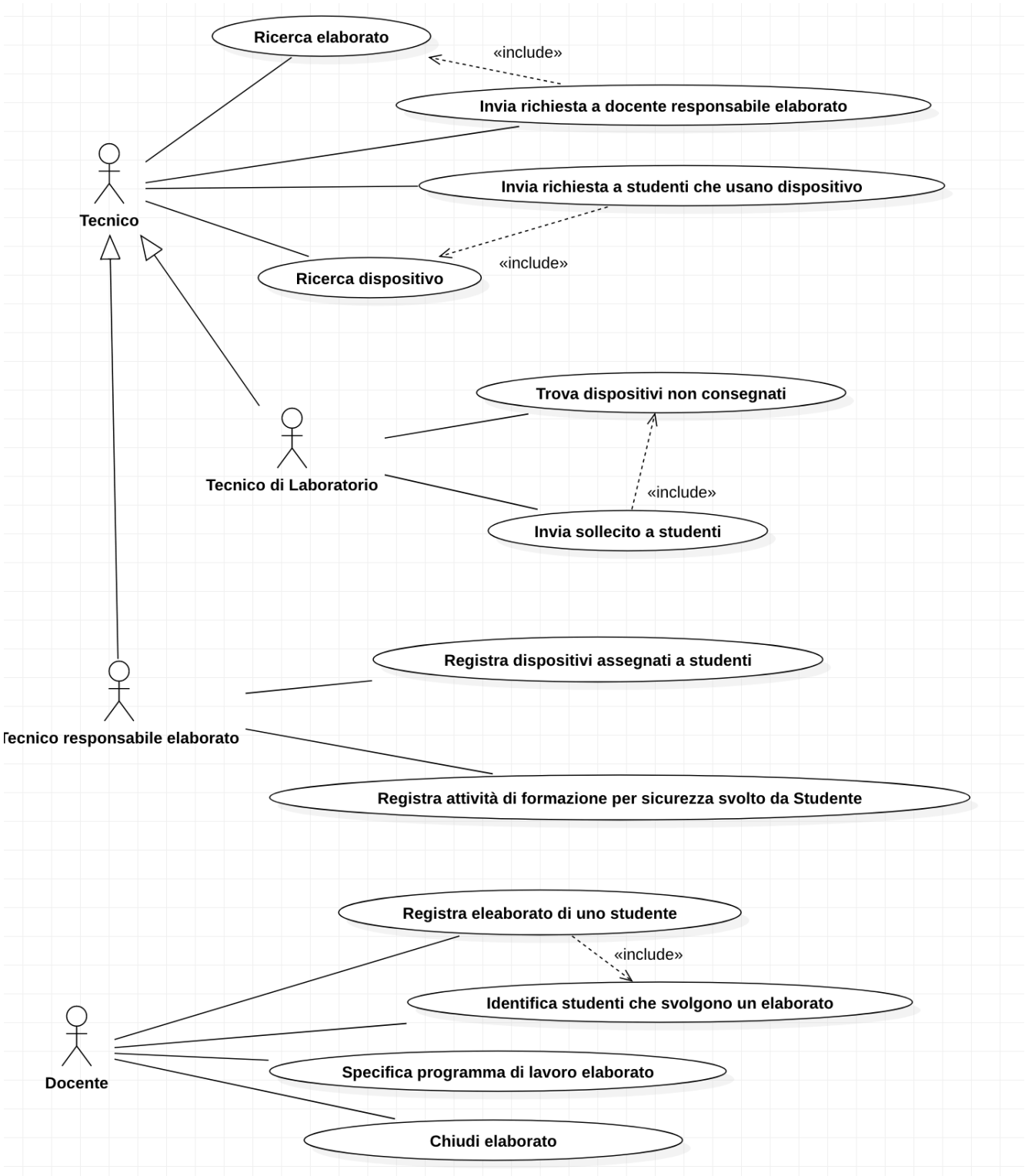

Figure 3: Use-case Diagrams

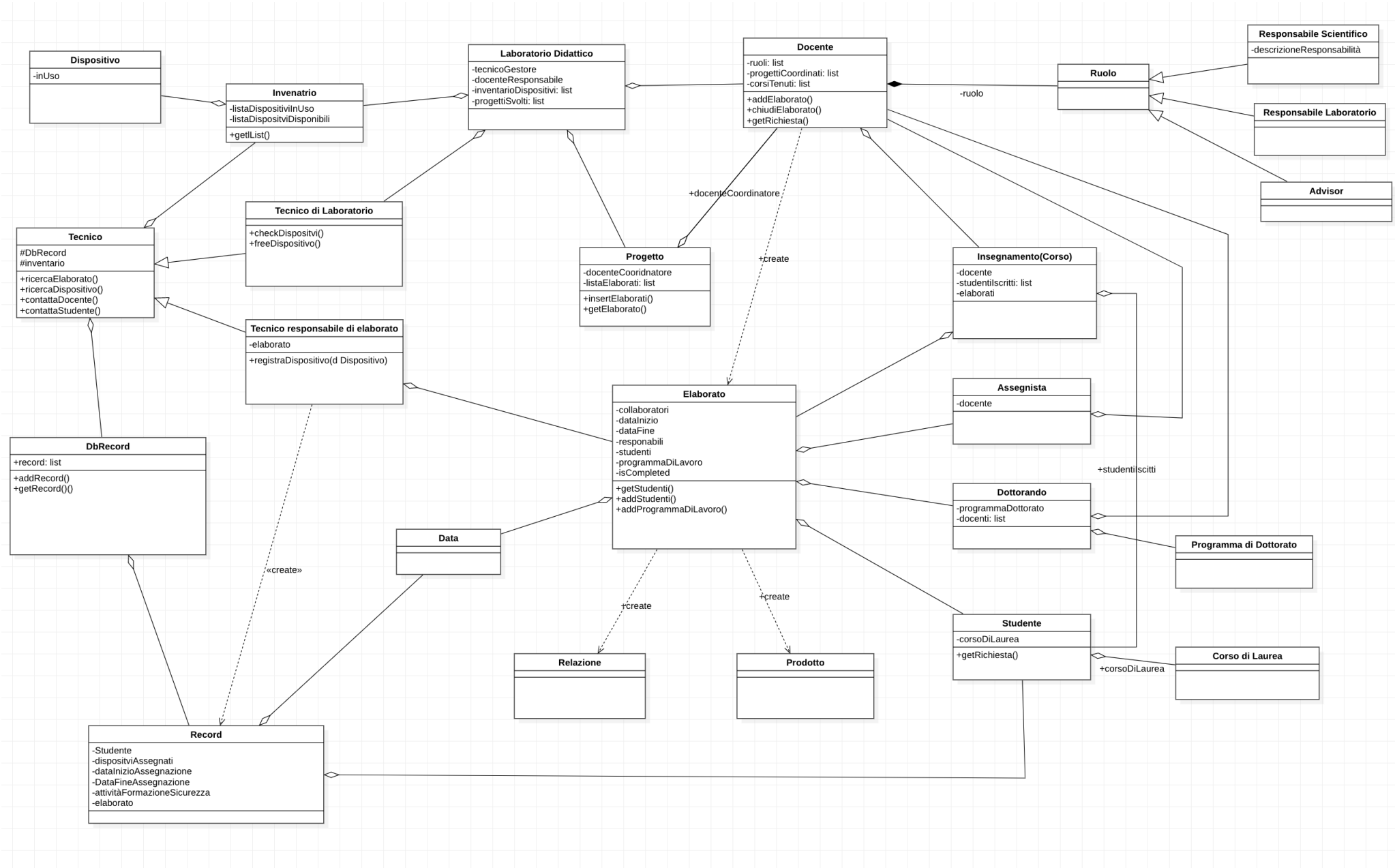

Figure 4: Class Diagrams