Implementing Many-to-Many Associations

Contents

[1 Introduction 1](#_Toc143940644)

[2 Implementation 1](#_Toc143940645)

[3 Associations 2](#_Toc143940646)

[4 Enforcing Uniqueness 2](#_Toc143940647)

[5 Adding a Support Class, *Registrar* 3](#_Toc143940648)

[6 Adding a *Registration* 3](#_Toc143940649)

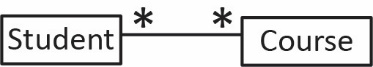
[7 Setting Grades 4](#_Toc143940650)

[8 Generating a Report Card 5](#_Toc143940651)

[9 Unit Testing a Package Visible Method 5](#_Toc143940652)

# Introduction

Consider the situation where a *Student* class is in a *many-to-many* relationship with a *Course* class. Thus, each *Student* can have many *Course*sand each *Course* can have many *Student*s.

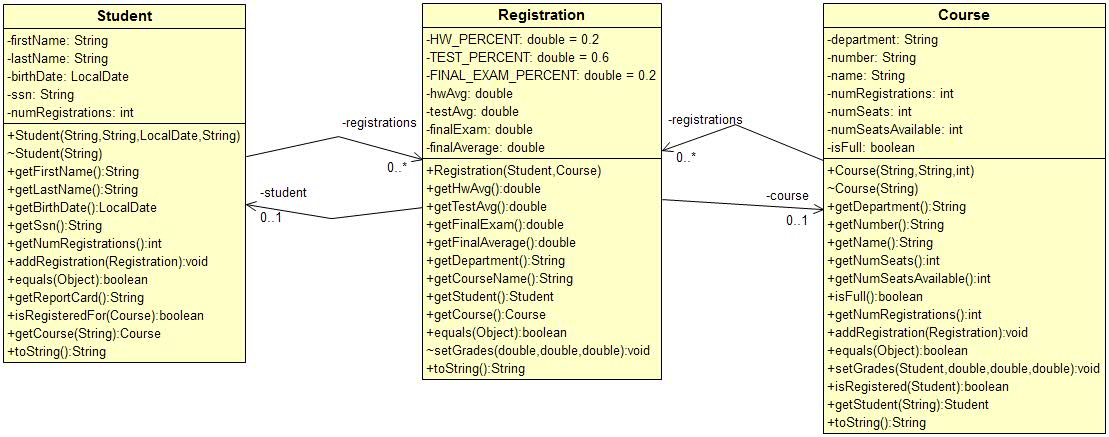


We introduce the *Registration* class to hold data associated with a student enrolled in a course: *hwAvg, getAvg, finalExam*.



# Implementation

Consider the class diagram below. In what follows, we discuss how to implement this. The code for this example is found on the Schedule. There are two packages: *ver4,* which contains the code and *ver4\_test* which contains the unit tests.



# Associations

We assume that the *ssn* in the *Student* class is required to be unique as well as the *name* in the *Course* class. The *name* is composed of the *department* and the *number.* For example, *department=”CS”* and *number=”4321”,* then the *name=”CS 4321”*. Thus, we use a map to implement the relationship between *Student* and *Registration*, where the key is *ssn*:

**public** **class** Student {

...

**private** TreeMap<String,Registration> registrations;

Similarly, we also use a map to implement the relationship between *Course* and *Registration,* where the key is *name*:

**public** **class** Course {

...

**private** TreeMap<String,Registration> registrations;

Finally, the *Registration* class maintains associations with both *Student* and *Course*:

**public** **class** Registration {

...

**private** Student student;

**private** Course course;

# Enforcing Uniqueness

The *equals* method is overridden in each of the classes to enforce uniqueness if needed. In this example, I don’t believe they are used. If there were methods, that return lists, then these implementations of *equals* would be useful as they support the list operations: contains(Object o), indexOf(Object o), lastIndexOf(Object o), remove(Object o).

|  |  |
| --- | --- |
| *Student* Class | *Course* Class |
| **public** **boolean** equals(Object o) {  **if**(o **instanceof** Student s) {  **return** **this**.ssn.equals(s.ssn);  }  **return** **false**;  } | **public** **boolean** equals(Object o) {  **if**(o **instanceof** Course c) {  **return** **this**.name.equals(c.name);  }  **return** **false**;  } |

|  |
| --- |
| *Registration* Class |
| **public** **boolean** equals(Object o) {  **if**(o **instanceof** Registration r) {  **return** **this**.student.equals(r.student) &&  **this**.course.equals(r.course);  }  **return** **false**;  } |

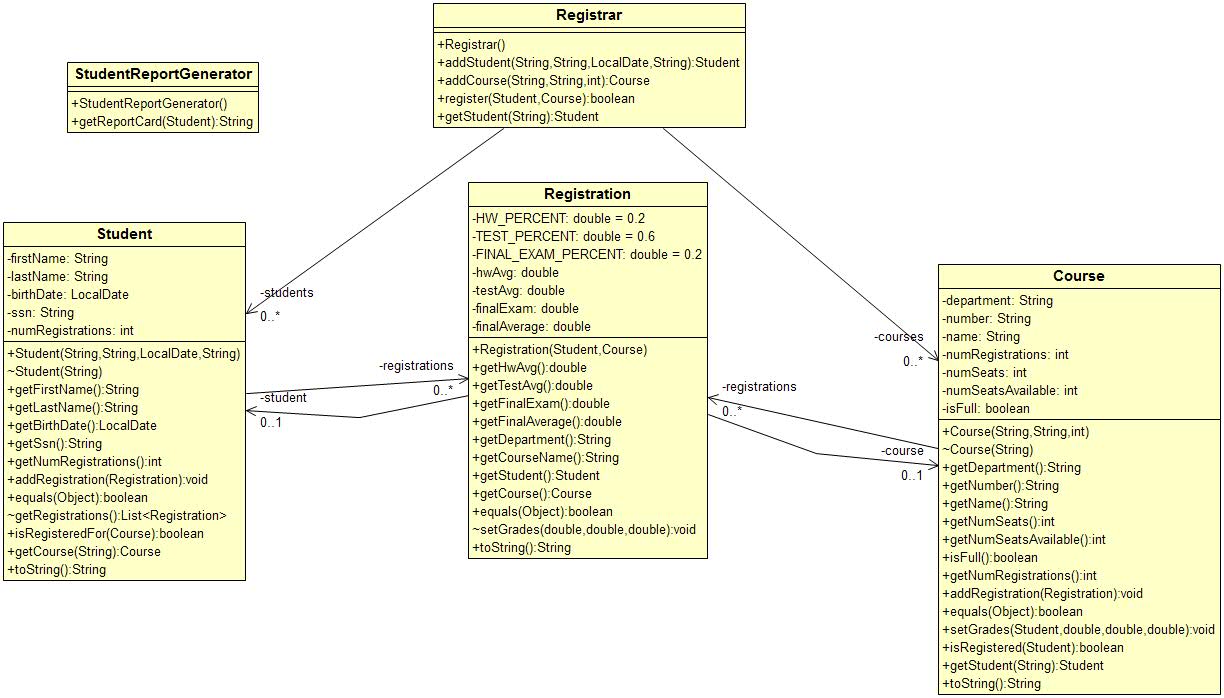
# Adding a Support Class, *Registrar*

In addition to the domain classes mentioned above, we introduce a *Registrar* class which maintains a map of *Student*sand a map of *Course*sas well as a method, *register* to register a student in a course.

**public** **class** Registrar {

**private** TreeMap<String,Student> students;

**private** TreeMap<String,Course> courses;



# Adding a *Registration*

The *Registrar* class has a *register* method:

**public** **boolean** register(Student s, Course c) {

// Registrar must contain reference to student and course before registering,

// and student must not already be registered for course.

**if**(!students.containsKey(s.getSsn()) ||

!courses.containsKey(c.getName()) ||

s.isRegisteredFor(c) ) {

**return** **false**;

}

Registration r = **new** Registration(s,c);

s.addRegistration(r);

c.addRegistration(r);

**return** **true**;

}

The *Student* class has an *addRegistration* method:

**public** **boolean** addRegistration(Registration r) {

**if**(registrations.containsKey(r.getCourseName())) {

**return** **false**;

}

registrations.put(r.getCourseName(),r);

**return** **true**;

}

The *Course* class has an *addRegistration* method:

**public** **boolean** addRegistration(Registration r) {

**if**(registrations.containsKey(r.getStudent().getSsn())) {

**return** **false**;

}

**if**(isFull()) {

**return** **false**;

}

registrations.put(r.getStudent().getSsn(),r);

numSeatsAvailable--;

**if**(numSeatsAvailable==0) {

isFull = **true**;

}

**return** **true**;

}

# Setting Grades

The *Course* class is responsible for setting grades:

**public** **boolean** setGrades(Student s, **double** hwAvg, **double** testAvg, **double** finalExam) {

**if**(!registrations.containsKey(s.getSsn())) {

**return** **false**;

}

Registration r = registrations.get(s.getSsn());

r.setGrades(hwAvg, testAvg, finalExam);

**return** **true**;

}

The *Registration* class records the grades and computes the *finalAverage* as shown below. The method is given package level visibility to prevent other classes outside the package from directly modifying the grades.

**void** setGrades(**double** hwAvg, **double** testAvg, **double** finalExam) {

**this**.hwAvg = hwAvg;

**this**.testAvg = testAvg;

**this**.finalExam = finalExam;

**this**.finalAverage = ***HW\_PERCENT*** \* hwAvg +

***TEST\_PERCENT*** \* testAvg +

***FINAL\_EXAM\_PERCENT*** \* finalExam;

}

# Generating a Report Card

We introduce the *StudentReportGenerator* class to build a report card for a student:

**public** **static** String getReportCard(Student s) {

StringBuilder sb = **new** StringBuilder();

sb.append("Report Card\n-------------------\n");

**if**(s.getNumRegistrations()==0) {

sb.append("No courses registered for");

**return** sb.toString();

}

**int** i=0;

**for**(Registration r : s.getRegistrations()) {

String line = String.*format*("%d. %s, final avg=%.1f\n", ++i, r.getCourseName(), r.getFinalAverage());

sb.append(line);

}

**return** sb.toString();

}

Above, we utilized the *getRegistrations* method found in the *Student* class. We’ve given the class package level visibility and have it return an unmodifiable list in order to practice defensive programming:

// In practice, may or may not need to be unmodifiable

List<Registration> getRegistrations() {

List<Registration> list = **new** ArrayList<>(registrations.values());

**return** Collections.*unmodifiableList*(list);

}

# Unit Testing a Package Visible Method

The *setGrades* method in the *Registration* class has package level visibility. Assuming the unit tests are in a separate *test* package, then reflection must be used to access the method.

@Test

**void** testSetGrades() **throws** NoSuchMethodException, IllegalAccessException,

InvocationTargetException {

String date = "1999-04-26";

LocalDate bDate = LocalDate.*parse*(date);

Student s1 = **new** Student("Dave", "Gibson", bDate, "222348834");

Course c1 = **new** Course("CS", "3326", 3);

Registration r = **new** Registration(s1, c1);

Method m = getSetGradesMethod();

m.invoke(r, 90, 80, 70);

**double** actualFinalAverage = r.getFinalAverage();

**double** expectedFinalAverge = 0.2\* 90 + 0.6 \* 80 + 0.2\*70;

*assertEquals*(expectedFinalAverge, actualFinalAverage);

}

**private** Method getSetGradesMethod() **throws** NoSuchMethodException {

Method method = Registration.**class**.getDeclaredMethod("setGrades", **double**.**class**,

**double**.**class**, **double**.**class**);

method.setAccessible(**true**);

**return** method;

}