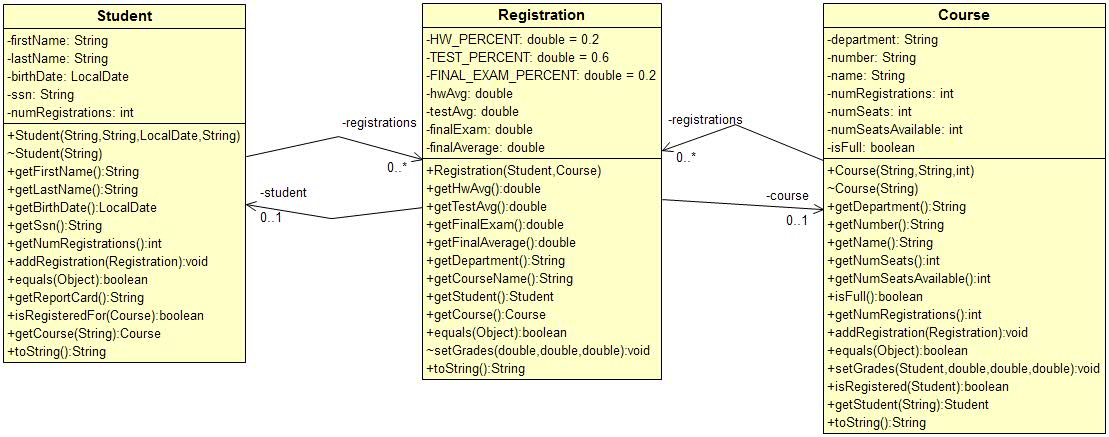
**Implementing a Many-to-Many Relationship**

In this section we consider the implementation of a many-to-many relationship between classes.

**Version 1**

1. Consider a *many-to-many* relationship between *Student* and *Course* which we resolve by introducing the *Registration* class.



1. Both *Student* and *Course* have a list of *Registrations*:

**private** List<Registration> registrations;

And, *Registration* has associations with *Student* and *Course*:

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

...

**private** Student student;

**private** Course course;

...

}

1. Both *Student* and *Course* have an *addRegistration* method to add a *Registration* instance to the respective lists. To register a student in a course (see code in *InformalTest*)

// Create student

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

// Create course

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

// Create registration

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

// Register student

s1.addRegistration(r);

c1.addRegistration(r);

Note that these steps should be encapsulated in method. Later, we introduce a *Registrar* class that has this *register* method:

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

**if**(!s.isRegisteredFor(c)) {

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

s.addRegistration(r);

c.addRegistration(r);

**return** **true**;

}

**return** **false**;

}

1. Grades are stored in the *Registration* class. We chose to write a *setGrades* method in the *Course* class. We will explore how this is implemented as we go along. First, to add grades for a student (see code in *InformalTest*):

// Add grades

c1.setGrades(s1, 90.0, 75.0, 80.0);

1. Before we look at *setGrades,* note that all three classes override *equals*:
2. *ssn* is unique for a *Student*

**public** **boolean** equals(Object o) {

Student s = (Student)o;

**return** **this**.ssn.equals(s.ssn);

}

1. *courseName* is unique for a *Course*, and

**public** **boolean** equals(Object o) {

Course c = (Course)o;

**return** **this**.name.equals(c.name);

}

1. A *Registration* is unique for a given student and course

**public** **boolean** equals(Object o) {

Registration r = (Registration)o;

**return** **this**.student.equals(r.student) &&

**this**.course.equals(r.course);

}

1. The *Course* class has a method to *setGrades*

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

Registration rKey = **new** Registration(s,**this**);

**int** loc = registrations.indexOf(rKey);

Registration r = registrations.get(loc);

r.setGrades(hwAvg, testAvg, finalExam);

}

Note the following:

1. We must find the *Registration* associated with the student. To do this we create a *Registration* object, *rKey* which we use to search (*indexOf*) the list of registrations, *registrations.*
2. The *Registration* class has a *setGrades* method to record the grades in the *Registration* class. Note that it has package-level visibility so that it can not be directly called outside the package

**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;

}

1. Given a student, we can search for a course by supplying the *courseName*.

// Get a Course

Course c = s1.getCourse("CS 3326");

1. The *Student* class’s *getCourse* method uses a similar mechanism as above to find the corresponding *Registration,* and once it has it, use it to retrieve the *Course:*

**public** Course getCourse(String name) {

Course c = **new** Course(name);

Registration rKey = **new** Registration(**this**,c);

**int** loc = registrations.indexOf(rKey);

Registration r = registrations.get(loc);

**return** r.getCourse();

}

However, note that we must create a “dummy” *Course*. We do this using a package-level constructor (not the full constructor) which just takes the *courseName* as this is all we need because *courseName* is unique.

Course(String name) {

**this**.name = name;

}

1. The *Student* class has a *getReportCard* method which simply loops through all the *Registrations* using them to retrieve the course name (*Registration* must retrieve it from the *Course*) and final average (stored in *Registration*).

**public** String getReportCard() {

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

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

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

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

**return** sb.toString();

}

**int** i=0;

**for**(Registration r : registrations) {

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

r.getCourseName(), r.getFinalAverage());

sb.append(line);

}

**return** sb.toString();

}

**Version 2**

1. A better implementation of this is to leverage the fact that *ssn* and *courseName* are unique. Thus, we can use a map to hold the registrations. In the *Course* class we use *ssn* as the key, and in the *Student* class we use *courseName* as the key:

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

Thus, when the *Student* class adds a registration:

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

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

}

Note that the *Registration* class has a convenience method to retrieve the course name:

**public** String getCourseName() {

**return** course.getName();

}

And when the *Course* class adds a registration:

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

**if**(!isFull) {

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

numSeatsAvailable--;

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

isFull = **true**;

}

}

}

1. Use of the map to hold the *registrations,* simplifies any method where we need to access a particular *Registration*. For example, the *setGrades* method in the *Course* class

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

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

r.setGrades(hwAvg, testAvg, finalExam);

}

Or, the *Student* class’s *getCourse* method:

**public** Course getCourse(String name) {

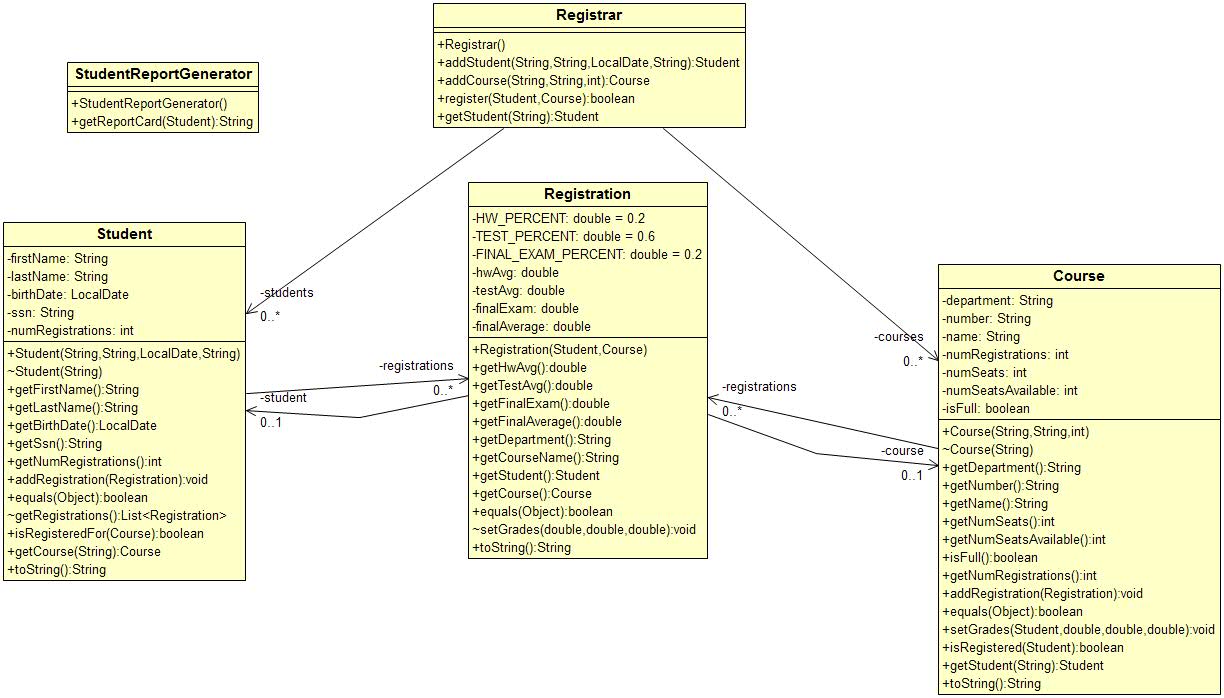
Registration r = registrations.get(name);

**return** r.getCourse();

}

**Version 3**

1. Version 3 introduces two new classes: *Registrar* and *StudentReportGenerator*. We explain these next.



1. We need a class to hold all the courses and students, so we introduce the *Registrar* class. There, we use maps to hold these collections:

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

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

1. The *addStudent* method makes sure students are unique, returning a reference to the *Student* if the add was successful, and *null* if the student already exists:

**public** Student addStudent(String firstName, String lastName, LocalDate birthDate,

String ssn) {

Student s = **new** Student(firstName, lastName, birthDate, ssn);

**if**(!students.containsKey(ssn)) {

students.put(ssn, s);

**return** s;

}

**return** **null**;

}

Now, client code (*InformalTest*) doesn’t need to create students directly, it uses the *Registrar*:

Registrar registrar = **new** Registrar();

LocalDate bDate = LocalDate.*parse*("1999-04-26");

Student s1 = registrar.addStudent("Dave", "Gibson", bDate, "222348834");

1. We remember from Version 1 and 2 that the *Student* class had a *getReportCard* method. In version 3, we remove this method and place it in a *StudentReportGenerator* class as a static method:

**public** **class** StudentReportGenerator {

**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;

// Had to add getRegistrations method in Student class.

// Returns an unmodifiable list.

**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();

}

}

The reason for this is that the *Student* class is likely to become cluttered with longish methods to build various reports (and the same for the *Course* class). A class should have one purpose (we call this *high cohesion*). The *Student* class’s purpose is to manage its data (ssn, name, etc, and the registrations). Report generation is an additional responsibility.

Another class design principle is: “Separate the things that vary from the things that stay the same.” Reports are likely to change: need a different format, different data, new reports, *etc*. Thus, it is better to encapsulate those in a separate class: *StudentReportGenerator*. We would likely need a *CourseReportGenerator* class as well.

We make this (and subsequent) methods static as the class holds no data.

To use this method, we simply pass it a student:

String reportCard = StudentReportGenerator.*getReportCard*(s1);

1. Note above (see the commented code), that we need all the student’s registrations to build the grade report. Thus, we had to introduce a *getRegistrations* method in the student class.

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

List<Registration> getRegistrations() {

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

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

}

We should practice *defensive programming* which means trying to prevent others from using our code in ways that are unintended. A student’s list of registrations should not be modified anywhere except inside the *Student* class. Thus, we return an unmodifiable list of registrations.