Student Registration Example
Ralph E. Johnson

This document is a chapter from a textbook that is supposed to use patterns to teach introductory programming. The textbook will use Java, and is supposed to teach good style and good design from the beginning. Patterns are not the purpose of the book, but are a means to the end. The purpose of the book is to teach programming.

Since these patterns are the first one that students learn, they are "elementary programs". This chapter does not describe the patterns, but uses them as if they were described elsewhere in the book. Most of the patterns are described in the paper "Some Patterns of Novice Programs" by E. Wallingford, D. Steinberg, R. Duvall, R. Johnson, which is another PloP paper. It should probably be read before this one. In practice, students will skip between them.

In this document, pattern names are underlined. This is partly because pattern names need to be easily distinguishable, and partly because the book is being developed on the web (at http://wiki.cs.uiuc.edu/ElementaryPatterns).

The book is going to teach test-first programming and refactoring, but does not make a big deal about it. The goal is to make these techniques obvious and natural.

Chapter 3

Representing information

One of the main uses of computers is to keep track of things. Computers keep track of everything you buy on a credit card, and print out your bill. They keep track of the phone calls you make on your cell phone. They keep track of the books in a library, and know when they have been checked out and where you should be able to find them. Your doctor's computer might keep track of your medical history. An engineer's computer might store the design of an automobile or a radio, and policemen are able to use computers that keep track of criminals.

When you are designing an information system, it is helpful to separate the question "what does the system know?" from the question "what does the system do?" However, one of the secrets of object-oriented programming is that a single object will both know things and do things. Earlier ways of programming tended to have separate systems for representing information and for doing things with it. Having individual objects do both makes it easier to understand and change a system. Nevertheless, it is often useful to first think about what a system knows, and then what it does. In general, each thing that a system knows will be an object, and what it does will be methods on one of those objects.

In contrast, some computer systems are not so much about information as about doing things. Early cell phones stored little information. You'd type in a number, and the phone would call it. Most of the software in a cell phone was devoted to converting voice into radio signals, and controlling the display and the keypad. But now cell phones keep track of call history and an address book. Some of them keep calendars and remember different tunes for different callers. So, cell phones are much more like information systems than they used to be, but a lot of the software in a cell phone is still about doing, not about remembering.

This chapter will teach you how to make systems that are primarily information, and other sections will teach you how to make other kinds of system.

A Student Registration System

Many of you register for courses online. The computer knows all the courses that are available each semester, and also knows which ones you are taking. It makes sure that you can't register for too many courses, that you can't take the same course twice, that you have taken the prerequisites, and that a course is not oversubscribed.

A registration system has many users. You've probably only seen the system from the point of view of a student. You want to know what courses you signed up for, and maybe how many more hours you need to take to have a full load. But you don't need to know

who is signed up for a particular course. This is only needed by teachers. Teachers will want a course listing, and will need to assign grades to each student in a course. But teachers don't decide which courses are going to be offered each semester. This is done by some administrator. So, there are at least three views of a student registration system; the student's view, the teacher's view, and the administrator's view.

A student registration system keeps track of students, courses, and who registers for which one. Students don't really register for a course, they register for a section. A section is for a particular semester, and keeps track of the students registered for the course that semester. A course has a title like "Introduction to Tourism" and a name like "TO101". It belongs to a department. The department named "Tourism" might have courses named "TO101", "TO201", "TO202", "TO301", "TO302", and "TO304". In fall of 2015, the department is offering TO101 (as always), TO201 and TO301. So, each of those courses has a section for Fall 2015. A course might have several sections for one semester, but offered at different times. But none of these courses are that popular.

We are going to develop a student registration system bit by bit. At first, we are not going to check for a student taking the same course twice, or whether the course is full. We are just going to figure out how to represent the basic information. We are going to provide a way to create new courses and students, and to say that a student is registering for a particular section of a course. After that, we'll make the system check that it is being used correctly, and then we'll make it able to provide reports and to answer questions. We'll allow teachers to record grades.

The first version of the system will let you

- define a student
- define a department
- define a course
- define a course section
- register a student to a section of a course

The first four operations would only be done by an administrator. The last operation would only be done by a student. This first iteration will not have any of the operations needed by a teacher.

First design of registration system

Given the description so far, an expert object-oriented designer would say that there are four obvious classes; Student, Course, Section, and Department. Why would she say that?

One of the standard ways of finding classes is to read a specification and look for nouns. Student, course, section and department are all nouns in the specification. Teacher and administrator are also nouns, though, and they are not on the list. Why not?

First, classes from nouns is one way to learn classes, but not the only one. Usually we'll start with a few obvious classes and get on with the design, and then add some more

classes as needed. The first classes are usually nouns in the specification, but later classes are more likely to be solutions to problems that come up.

In the same way, not all nouns in the specification become classes in the design. Sometimes two different nouns end up as the same class, either because the nouns are synonymns or because they are names of different instances of the same class. Sometimes nouns refer to things outside the system that interact with the system in some way. Both Administrator and Teacher are good examples of things that are outside the registration system. Later on we will decide to record the teacher of each section, but at the moment, Teacher is not part of the system.

A good class should have one major responsibility. When a class has several responsibilities, it is usually a sign that you need to split it into pieces. Our classes and their responsibilities are

- Student - keep track of sections for which students have registered.
- Course - keep track of course sections
- Section - keep track of students in the course
- Department - Name, courses that are held by the department.

None of these classes has only one responsibility; each knows its name, sections know their course, and courses know their departments. But these are minor responsibilities, which will turn out to be trivial to implement. So we can say that each of these has one major responsibility.

The next step is to think about the interface of these classes. How will we use them? A good way to learn how to use them is to write some test cases. So, how would we create students, courses, sections, and departments?

In general, the way to create an object in Java is to call the constructor of its class. For example, the way to create a student object is "new Student()" and the way to create a Department is "new Department()", assuming that the constructors of Student and Department do not have arguments. But Student and Department almost certainly need arguments for the constructors. Both a Student and a Department need to have a name so we can distinguish one from the other. So, we should create a new Student with an expression like "new Student("Marvin Minsky")"

Make a test for creating a Student as follows:

```
public void testStudentCreation() {
    Student s;
    s = new Student("Marvin Minsky");
    assertTrue(s.getName() == "Marvin Minsky");
}
}
```

This test implies that we need to be able to read the name of the student, which is not too surprising. Even so, it is a pretty wimpy test, and an expert would be likely to not bother writing it, but it will get us started.

The test will not compile (you can try it and see) because there is no Student class. We can write a simple Student class as follows:

```
public class Student {
     public Student(String name) {
         this.name = name;
     }
     public String getName()  {
         return this.name;
     }
     String name;
}
```

This defines Student as a class with a constructor and one method, getName(). The class Student defines one instance variable, "name". This means that each Student has a name, which is a string. This should allow the test to compile and run.

The test runs because it is nearly trivial. A more interesting test of Student would be to register a student for a section. But we can't do that yet, because we don't have sections. Or courses. Or departments. So, let's implement them and then come back to Student.

Building three classes at once is a lot of work. Never do a lot of work without checking it. test before code is a good way to check your work. But writing tests for three classes is a lot of work. So, let's simplify things a little. Let's assume that Students take classes, and forget sections and departments. After all, a lot of classes only have one section per semester. This lets us write a test like

```
public void testAddingStudent(){
  Course c;
  Student s;
  c = new Course("Tourism 101");
  s = new Student("Donald Knuth");
  c.addStudent(s);
  assertEquals(c.students.size(), 1);
}
```

Define the class Course and its constructor. It is just like Student.

```
public class Course {
    private String name;
    public Course(String name) {
               super();
               this.name = name;
       }
}
```

But how do we implement addStudent()? First, addStudent() implies that a Course has a set of Students. How does one object keep track of a number of other objects? In Java, the usual way is to have a Vector. Vector is a standard Java class. A Vector is an object that holds zero or more other objects. Some of its methods are addElement(anElement),

which add anElement to the vector; removeElement(anElement), which removes anElement from the vector; and elementAt(i), which returns the i'th element of the vector.

We need to give Course an instance variable that holds a Vector. We need to initialize it in the constructor of Course and add methods addStudent() and students() to Course. The method addStudent() will add the student to the vector (using the vector method addElement()) and the method students() will return the vector.

```
public class Course {
    private String name;
    private Vector students;
    public Course(String name) {
               super();
               this.name = name;
               this.students = new Vector();
       }
    public void addStudent(Student s){
       this.students.addElement(s);
    }
    public Vector students(){
       return this.students;
    }
}
```

The testAddStudent test is enough to make us implement addStudent, but it is incomplete. What if addStudent() made the list longer, but added something else, and not a student? We aren't even checking that the student that we added is really there. So, let's add another assertion to testAddStudent(). Vector has a method elementAt() that returns a particular element, so we can ask it whether its first element is the student that we added.

```
assertEquals(c.students().elementAt(0), s);
```

It is good to play with a test to convince yourself that it is right. If you change the variable "s" to a 0 then the line will not type-check, because assertEquals expects both arguments to be an Object. But if you change it to "new Student("Bob")" then it will run, but the test will fail because the course contains a different student. So, now we know that the course has one student, and that it is the right one.

Suppose a course had many students. How would we check that it contained a particular one? One possibility is to make a method on Course called containsStudent() that takes a student and returns a boolean. Then we could write the following test.

```
public void testManyStudents(){
  Course c;
  Student s1;
  Student s2;
  c = new Course("Tourism 101");
  s1 = new Student("Donald Knuth");
  c.addStudent(s1);
  s2 = new Student("Marvin Minsky");
  c.addStudent(s2);
  assertEquals(c.containsStudent(s1));
```

```
}
```

The interface for containsStudent is not realistic. It assumes that we have a variable that points to the student, which is OK for a test in which we just created the student. But in the final registration system, the information about students will be created somewhere else and last for years. There should either be a way to find a student with a particular name, or there should be a way to ask a Course for a student with a particular name. Let's take the second approach. Then the test would be

```
public void testManyStudents(){
  Course c;
  Student s1;
  Student s2;
  c = new Course("Tourism 101");
  s1 = new Student("Donald Knuth");
  c.addStudent(s1);
  s2 = new Student("Marvin Minsky");
  c.addStudent(s2);
  assertEquals(c.containsStudentNamed("Donald Knuth"));
}
```

The temporary variables s1 and d2 are each used once, and are not needed. The test is shorter and easier to read if we get rid of them as follows:

```
public void testManyStudents(){
  Course c;
  c = new Course("Tourism 101");
  c.addStudent(new Student("Donald Knuth"));
  c.addStudent(new Student("Marvin Minsky"));
  assertTrue(c.containsStudentNamed("Donald Knuth"));
}
```

Now all we need to do is to write containsStudentNamed()

```
public boolean containsStudentNamed( String aName) {
         for (int i=0; i< students.size(); i++) {
           if (((Student)
(students.elementAt(i))).getName().equals(aName)) return true;
         }
         return false;
       }
```

containsStudentNamed returns true if it contains the student, and false otherwise. It shows the classic pattern for searching a vector. It contains a for loop that examines each element in the vector. When it finds the right element, it returns true. If it examines each element without finding the right one then the vector does not contain the element being looked for, so it returns false. Vectors can hold any kind of object, so elementAt() is declared to return an Object, not a Student. But we know it only contains Students because that is all that we put in there. So, we "cast" the result of elementAt to Student with the phrase "(Student)", which tells the Java compiler to treat the result as a Student so we can call its getName() method. testManyStudents should work now. It tests that a course knows the students who have been added to it. But a student should know the

courses he or she is taking. So, a student needs to keep track of a number of objects, too. It does this the same way that a course keeps track of students. # a student has a vector of courses # Student has an addCourse() method for adding courses # Student has a isTakingCourseNamed() method for indicating whether it is taking a particular course. Here is a test for a student taking several courses.

```
public void testManyCourses(){
  Student s;
  s = new Student("Donald Knuth");
  s.addCourse(new Course("TO101"));
  s.addCourse(new Course("TO102"));
  assertTrue(s.isTakingCourseNamed("TO101"));
}
```

You should be able to implement this because it is another example of the keep track of a number of objects pattern.

Adding a student to a course should be equivalent to adding a course to a student. In other words, the following version of the previous test should work, too:

```
public void testManyCourses(){
  Student s;
  s = new Student("Donald Knuth");
  (new Course("TO101")).addStudent(s);
  (new Course("TO102")).addStudent(s);
  assertTrue(s.isTakingCourseNamed("TO101"));
}
```

To make this test work, we need to make sure that adding a course to a student also adds a student to a course. In other words, the addCourse and addStudent method are either both called or both not called. The way to do this is to have make sure that one of them is called in only one place, and that one place is the other method. Let's make addStudent be the method that calls addCourse. Then addCourse shouldn't be called anywhere else. The test cases will call only addStudent. Then addStudent will look like

```
public void addStudent(Student s){
   this.students.addElement(s);
    s.addCourse(this);
}
```

Now that we have some tests for Student and Course, let's look at Department. A Department has a set of Courses, so it needs to keep track of a number of objects But Course is a little different from Student because a Course has only one Department, but a Student can take many courses. You can have a Student who isn't taking any courses, but you can't have a Course that isn't in a Department. (Well, you can't at most schools, but you can at some. You can't at JavaLand University, which is what we are building this for.) If we add a Department to a Course using setDepartment then there will be a period of time when the Course is illegally without a department. The solution is to make sure that the Course is initialized properly in its constructor, which means that the Department needs to be a parameter of the constructor of Department. The following test case illustrates.

```
public void testCourseCreation() {
   Department d;
   Course c;
   d := new Department("Tourism");
   c := new Course("TO101",d);
   assertTrue(d.getDepartment() == d);
   assertTrue(g.getName() == "TO101");
}
```

This builds a department and a course within that department, and then checks that the course actually holds the department and its own name.

Implementing this test case is just another use of the pattern to keep track of a number of other objects.

Checking input

This section will show how to prevent errors like allowing a student to sign up for too many courses, or allowing too many students to sign up for a course.

Grades and grade averages

This section will show how to record grades and compute grade averages. It will show how to compute grade average both iteratively and incrementally.