

CS1Ah Lecture Note 11

Case Study: Modelling using objects

This note has two main aims:

1. To illustrate the ideas of the previous two lectures on a slightly larger example.
2. To introduce a simple approach to the creation of a collection of objects that attempt to provide a “model” of a real world situation.

Creating a model of the system we intend to implement can be a difficult and involved task. You will revisit it a number of times during this course and in subsequent years. This note barely scratches the surface. We give a simple example and consider one or two issues associated with models. Getting the model right is a prerequisite to a good implementation and it is a crucial aspect of the design of systems.

Modelling the world with data

In constructing an object-oriented model of a system we need to decide on what objects we require to include in our system, how those objects relate to one another, and what methods we need to maintain the collection of objects in the system in the correct relationship. In later notes we will consider how to capture the relationships between objects and how to decide on what methods we need. Here we just concentrate on deciding on a collection of objects and how they model the world.

The simplest approach to identifying the collection of objects we need in a system is to comb through the description of the system looking for likely candidates. Usually candidates for objects correspond to the *noun phrases* in the description. For example *colorless green ideas* is a noun phrase while *sleep furiously* is not. Each noun phrase in a description is a candidate for a kind of object. After we have identified the candidates we reduce the number by identifying when different noun phrases are referring to the same thing or by discarding vague phrases or by discarding things we think are outside the system. This is best illustrated via an example.

An example

Here is a short description of the kind of record keeping that is typical in a University setting. We have italicised at least one instance of each noun phrase in the description.

The *Informatics Teaching Office* maintains a record of every *student* taking *Informatics courses*. This includes *basic contact and identification information* on the student, the student's *Director of Studies*, and the student's *programme of study*. The programme of study is a *record* of which courses the student is taking that *year*.

Considering the list of noun phrases we might decide that the Informatics Teaching Office is outside the scope of the system, that we can unify "Informatics courses" with "courses", that "record" and "year" are too vague and that the contact, identification and programme information is really part of the student object. This process would leave us with a list of candidate objects: Student, Director and Course. Being tuned in to *inheritance* we might then note that students and directors share some aspects in common, and decide that they can inherit these from a common Person superclass.

We might also consider the relationships between the objects. For example, each Student has exactly one Director, each Programme comprises a number of courses (someone will typically take 6 half courses in a year, perhaps more, perhaps fewer), each Director will direct many students. This kind of data is the beginning of a detailed model of the system. In later lectures we will consider how to capture these relationships more systematically.

This model is still quite crude. The next step is to begin to develop it by deciding what information is required in each object. For example, the Student object requires at least forename, surname, matric number, addresses, Director of Studies and Programme of Study, but at least the first two of these are also needed for Directors. We can decide what information is required in the other objects to arrive at a more refined model that includes the attributes of each object.

Validating models

Validating a model is the process of checking it agrees sufficiently well with the world that an implementation based on the model will be capable of supporting all the activities expected to be supported by the system. In this simple example this comes down to ensuring that each object carries the correct data adequately to represent the object in the world it is intended to represent. Our example is simple because it does not go into details. In particular it omits many constraints that would be included in a real system. For example, the choice of courses should not contain a clash of lecture times, programmes of study should be capable of leading to a final degree, etc.

In more complex situations we will see that we expect objects to maintain relationships between groups of objects and the validation process will include checking that the objects can indeed maintain those relationships.

Adding user interfaces

Once we have constructed an implementation of an object model we need to construct user interfaces that allow users access to the model. There may be many different classes of user and each class of user may have a different interface. The interface is where the methods of the objects in the system are coordinated to provide the functions required by the users. The design of user interfaces is an entire sub-area of Computer

Science and Software Engineering and is out of the scope of this lecture. To keep things simple (and to help with our validation) we will write a (very) short test program which creates and manipulates some of our objects. In a real system, the method calls which are made directly in this program would be generated behind the scenes by the user interface code, in response to typing and mouse clicks by a user.

```
1  class ITodemo {
2      public static void main(String[] args) {
3          Director d1, d2;
4          Student  s1, s2, s3;
5          Course   c1, c2, c3;
6
7          d1 = new Director("Murray", "Cole", "mic", 505154);
8          d2 = new Director("Paul", "Jackson", "pbj", 505131);
9
10         s1 = new Student("Freddy", "Furby", "ff", 208729);
11         s2 = new Student("Pika", "Chu", "pc", 293923);
12         s3 = new Student("Thomas", "TankEngine", "tt", 1);
13
14         c1 = new Course("Apld Txtting");
15         c2 = new Course("Track Maintenance");
16         c3 = new Course("Sitting around");
17
18         s1.setDirector(d2);
19         s2.setDirector(d1);
20         s3.setDirector(d2);
21
22         s1.addCourse(c1);
23         s1.addCourse(c2);
24         s2.addCourse(c3);
25         s2.addCourse(c2);
26         s3.addCourse(c2);
27
28         d1.showDirectees();
29         d2.showDirectees();
30
31         s2.showCourses();
32
33         s1.addCourse(c3);
34         s1.showCourses();
35
36         c3.changeTitle("Hard work in the Library");
37         s1.showCourses();
38     }
39 }
```

Implementing in Java

Having defined a model that includes the objects we require, their attributes and relationships, we can implement the model fairly directly in Java. We introduce classes for each kind of object and create enough objects of the right class to construct the model.

Starting with simple things first we design our class for courses. In a fuller example we might include a code, lecture times, pre-requisites and so on, but for now we choose just to note the course's name as a string. We will protect this with constructor and accessor methods in good object-oriented style. Notice that we include a method for changing a course's name. Thinking ahead, we would like any such change to be reflected immediately in the programme of study of any students taking the course.

```
1  class Course {
2      private String title;
3      public Course(String title) {
4          this.title = title;
5      }
6
7      public String getTitle() {
8          return title;
9      }
10
11     public void changeTitle (String newTitle) {
12         title = newTitle;
13     }
14 }
```

Now we consider the people in our model. We will use Java's *inheritance* mechanism to define a class which captures what all people (in the model) have in common, then extend this for the specifics of students and directors. Notice that there is no way to change a person's name once created. We also provide a handy `toString` method.

```
1  class Person {
2      private String forename;
3      private String surname;
4      private String email;
5
6      public Person (String fname, String sname, String email) {
7          this.forename = fname;
8          this.surname  = sname;
9          this.email    = email;
10     }
11
12     public String toString () {
13         return forename + " " + surname;
14     }
15 }
```

Next we implement our student class as a subclass of person (please don't take this sentence personally!).

```
1  import java.util.*;
2  class Student extends Person {
3      private int matricno;
4      private Director dos;
5      private Vector programme;
6
7      public Student(String fname, String sname, String email,
8                      int matricno) {
9          super (fname, sname, email);
10         this.matricno = matricno;
11         programme = new Vector();
12     }
13
14     public void addCourse(Course course) {
15         programme.addElement(course);
16     }
17
18     public void setDirector(Director mydos) {
19         dos = mydos;
20         mydos.assignDirectee(this);
21     }
22
23     public void showCourses() {
24         Iterator i = programme.iterator();
25         Course c;
26
27         while (i.hasNext()) {
28             c = (Course) i.next();
29             System.out.println(c.getTitle());
30         }
31         System.out.println();
32     }
33 }
```

There are a number of points of interest. Firstly, we use **extends** to capture the inheritance. Notice that the constructor invokes the constructor for the superclass `Person` before carrying out student specific actions. Secondly, the constructor forces a student to have a full name, matric number and email address, but not a DoS or any courses, so that these can be added separately later. Thirdly, notice how the `setDirector` method has a knock-on effect in the given director object (using **this** to refer to the current student object). This will help to maintain the consistency of the data. Finally, the `showCourses` method uses an iterator to print out the student's current set of courses. The iterator methods make this work however many courses a student has, without explicitly counting them.

We now turn to the remaining class, for Directors of Studies.

```
1  import java.util.*;
2  class Director extends Person {
3      private int extension;
4      private Vector directees;
5
6      public Director(String forename, String surname,
7                      String email, int extension) {
8          super(forename, surname, email);
9          directees = new Vector();
10     }
11
12     public void assignDirectee(Student student) {
13         directees.addElement(student);
14     }
15
16     public void showDirectees() {
17         Iterator i = directees.iterator();
18         Student s;
19
20         while (i.hasNext()) {
21             s = (Student) i.next();
22             System.out.println(s); //implicitly invoke "toString"
23         }
24         System.out.println();
25     }
26 }
```

The situation here is similar to `Student`, being a different subclass of `Person`. Notice that we don't try to reflect the assignment of a new directee by updating the student object as well (why might this be a bad idea?). Also notice that in the `showDirectees` method, the printing of the student object `s` automatically invokes the student `toString` method which we wrote earlier. If we hadn't written this we would have printed the result of the generic `toString` method inherited from the `Object` class, which wouldn't do what we might hope (you might like to find out what it *would* do).

Summary

- We can construct a crude object model by considering the noun phrases in the description of the system.
- Validating models involves checking how good they are at reflecting the real world.
- We can implement object models fairly directly in Java.

Murray Cole, 2002/09/06 14:24:21.