24. Multiple Inheritance, Factories & Facades

In real life it is possible to inherit from multiple sources – for e.g. genes from multiple parents and money from multiple benefactors. We will consider here the issue of inheriting multiple times – both for interfaces and classes. The problem used to understand this issue will be used also to show the benefits of defining factory classes and method for instantiating objects and for creating façade objects that combine multiple objects into one simpler whole.

Multiple Interface Inheritance

Consider a modification of the course problem of earlier chapters, where we wish to gather statistics about the courses queried by users. In particular, every time a user request for a course matching a title is successful, we want to increment a counter associated with the course, and provide a method to access this counter. We do not want to directly modify the previous solution, as some might prefer to use the version that does not gather and store statistics for efficiency and other reasons. For example, they might feel that these statistics give misleading information and do not allow an unpopular course to become popular. Thus, we would like to inherit rather change the previous code.

Let us begin with the previous course interface:
```java
class ACourse {
    String title, dept;
    public ACourse (String theTitle, String theDept) {
        title = theTitle;
        dept = theDept;
    }
    public ACourse () {} // constructor
    public void init (String theTitle, String theDept) {
        title = theTitle;
        dept = theDept;
    }
}
```

We wish to retain all of the previous methods, so we will create a new interface that inherits from it:
```java
class LoggedCourse extends ACourse {
    public int getNumberOfQueries();
}
```

Now let us consider the previous abstract class, which provided implementations shared by the two kinds of classes:
```java
class ACourse {
    String title, dept;
    public ACourse (String theTitle, String theDept) {
        title = theTitle;
        dept = theDept;
    }
    public ACourse () {} // constructor
    public void init (String theTitle, String theDept) {
        title = theTitle;
        dept = theDept;
    }
}
```

1 © Copyright Prasun Dewan, 2005.
Factories/facades

}  
public String getTitle() {return title;}
public String getDepartment() {return dept;}
public Course matchTitle(String theTitle) {
    if (title.equals(theTitle)) return this;
    else return null;
}
}

We want to retain this implementation and add to it the ability to store, change, and query the desired statistics. We can extend this class, adding a counter to store the number of successful matchTitles invoked on the course, a method to return the value of the variable, and an overriding matchTitle() that calls the super classes matchTitle(), and in case of a match, increments the counter:

    package courseTree;
    public abstract class ALoggedCourse extends ACourse implements LoggedCourse {
        int numberOfQueries = 0;
        public ALoggedCourse (String theTitle, String theDept) {
            super (theTitle, theDept);
        }
        public ALoggedCourse () {
        }
        public int getNumberOfQueries() {
            return numberOfQueries;
        }
        public Course matchTitle(String theTitle) {
            Course course = super.matchTitle(theTitle);
            if (course != null) numberOfQueries++;
            return course;
        }
    }

So far, this is just more use of single-inheritance. More interesting is the issue of how the interface of a regular course should be defined. Previously, it was:

    package courseTree;
    public interface RegularCourse extends Course {
        public void init (String theTitle, String theDept, int theCourseNum);
    }

The only difference between this interface and the desired one is that it needs to inherit the getNumberOfQueries() method from new Course interface. Thus, we can define it as:

    package courseTree;
    public interface LoggedRegularCourse extends LoggedCourse {
        public void init (String theTitle, String theDept, int theCourseNum);
    }

The init method defined by LoggedRegularCourse is identical to the one defined by RegularCourse. We can avoid this duplication by making this interface extend RegularCourse instead of LoggedCourse:

    package courseTree;
    public interface LoggedRegularCourse extends RegularCourse {
        public int getNumberOfQueries();
    }

But now we are duplicating the method in RegularCourse. The solution here is to inherit from both RegularCourse and LoggedCourse:
The interface is empty! All it does is combine the two interfaces, thereby not duplicating any code. The adjectives in its name were a hint that we needed multiple inheritance. The following figure summarizes the discussion so far.

**Figure 1 Inheriting from multiple sources**

Similarly, the new freshman seminar interface, LoggedFreshmanSeminar, also inherits from two sources: LoggedCourse, the new common interface, and FreshmanSeminar, the previous freshman seminar interface.

**Multiple Class Inheritance**

Let us now consider the class implementing regular courses and freshman seminars. The previous version defined constructors, implemented an init method, and implemented the dynamically dispatched getNumber() method:

```java
public class ARegularCourse extends ACourse implements RegularCourse {
    int courseNum;
    public ARegularCourse (String theTitle, String theDept, int theCourseNum) {
        init (theTitle, theDept, theCourseNum);
    }
    public ARegularCourse () {}  
    public void init (String theTitle, String theDept, int theCourseNum) { 
        super.init (theTitle, theDept);  
        courseNum = theCourseNum;
    }
    public int getNumber() { 
        return courseNum; 
    }
}
```

If we restrict ourselves to single inheritance, then new class would be the same except that it would inherit from ALoggedCourse instead of ACourse:

```java
public class ALoggedRegularCourse extends ALoggedCourse implements LoggedRegularCourse {
    int courseNum;
    public ALoggedRegularCourse (String theTitle, String theDept, int theCourseNum) {
        init (theTitle, theDept, theCourseNum);
    }
    public int getNumberOfQueries() {
    }
    public void init (String theTitle, String theDept, int theCourseNum);
}
```
Factories/facades

Thus, this class duplicated all the method implementations of the previous one. The natural thing to so at this point would be to use multiple class inheritance to remove the code duplication problem:

```java
public class ALoggedRegularCourse extends ALoggedCourse, ARegularCourse implements LoggedRegularCourse {}
```

Figure 2 Ambiguities created by multiple class inheritance

As it turns out, this creates an ambiguity shown the in the figure above. The problem is that two implementations of the `matchTitle()` method are inherited, one from each superclass. Which should be used? Some other languages such as C++ provide programmers with constructs to disambiguate in such situations, but Java’s has taken the philosophy of preventing such ambiguities by disallowing multiple class inheritance. This means you will often end up duplicating code as we have done in the example above.

**Impact of single class inheritance on polymorphism**

Another side effect is that the class duplicating the code does not have an IS-A relationship with the class whose code is duplicated. In our example, it is not the case that `ALoggedRegularCourse` IS-A `ARegularCourse`. Thus, given the following method:

```java
void print (ARegularCourse regularCourse) { … }
```

the following usage is not allowed:

```java
print (new ALoggedRegularCourse());
```

This problem arises because we broke our rule of using interfaces rather than classes to type variables. Had we used the interface name to type the parameter of the `print()` method:

```java
void print (RegularCourse regularCourse) { … }
```

we could indeed have made the method call above, as `ALoggedRegularCourse` IS-A `LoggedRegularCourse` IS-A `RegularCourse`. This is another reason for using interfaces as types: multiple inheritance is allowed for interfaces but not for classes!
**Class vs. Interface Inheritance**

In our example, why did multiple inheritance cause problems for classes but not for interfaces? In the class case, there were two alternative definitions for a method, matchTitle(), which was not the case for interfaces. In both cases, several definitions/implementations was inherited twice, from the two super types. In the interface case, all methods defined in Course were inherited twice, once from LoggedCourse and the second time from RegularCourse, as shown in Figure 1. In the class case, implementations of all methods except matchTitle() were inherited twice from ACourse. Java can determine when two method definitions/implementations are the same and thus do not conflict.

Are there examples of interface inheritance that create ambiguities? Let us change the interfaces as shown in the figure below:

![Diagram showing interface inheritance](image)

**Figure 3 Creating a duplicate method header**

Here we have duplicated in LoggedCourse the init method definition in Course. This is analogous to adding a new matchTitle() implementation the subclass. As it turns out, this change also does not cause problems. The two inherited definitions, though not the same in that they come from two different sources, are equal. An interface is a set of method definitions, and set addition ensures no duplicates are added.

Now consider the following variation of the example shown in Figure 4. This time the init() method in LoggedCourse is identical to the one in the supertype, Course, except that its return type is different. Including both of them in the interface is creates overload resolution problems. Therefore, just as Java will not let you implement two methods that differ only in return, it will not let you inherit two definitions that differ only in return types.

To summarize, like multiple class inheritance, multiple interface inheritance can also cause ambiguities though they are rarer. Java chooses to ban multiple class inheritance because of the possibility of creating ambiguities, but uses a more flexible approach for interfaces, preventing only those multiple inheritance scenarios in which ambiguities actually arise.
Implementing vs. Extending Multiple Interfaces

The extended interface, we created is empty in that it has no methods. An alternative to extending multiple interfaces (to create an empty interface) is to implement multiple interfaces. In our example, instead of making ALoggedRegularCourse implement a single empty interface, LoggedRegularCourse, that extends the two interfaces LoggedCourse and RegularCourse, we make the class directly implement the two interfaces:

```java
public class ALoggedRegularCourse implements RegularCourse, LoggedCourse {
    ...
}
```

Implementing multiple interfaces, $I_1, I_2, \ldots I_N$, is equivalent to implementing a single empty interface $I$ that extends $I_1, I_2, \ldots I_N$.

Which approach is better? The extension approach causes us to define more interfaces. In the worst case, for every possible set of interfaces, we would have to define a new empty interface that extends each member of the set, leading to a combinatorial explosion. In practice, however, we would a small number of these sets would need an extended interface. Moreover, implementing multiple interfaces leads to cast between the extended interfaces, which is generally to be avoided as it can lead to runtime errors.

Consider the following situation. We have two methods, one that takes parameters of type LoggedCourse and another of type RegularCourse:

```java
void print (RegularCourse regularCourse) { ... }
void appendToLog(LoggedCourse loggedCourse) { ... }
```

Now consider a variable that is assigned an instance of ALoggedRegularCourse implementing the two interfaces:

```java
LoggedRegularCourse loggedRegularCourse = new ALoggedRegularCourse();
```

If the variable is declared as a RegularCourse:

```java
RegularCourse loggedRegularCourse;
```

then it must be cast to a LoggedCourse before being passed to appendToLog():

```java
appendToLog((LoggedCourse) loggedRegularCourse);
```

It could have been declared to be of type LoggedCourse:

```java
LoggedCourse loggedRegularCourse;
```
Factories/facades

But this time it must be cast to RegularCourse before printing:

```
print((RegularCourse) loggedRegularCourse);
```

The extension approach gives us a type that combines both interfaces:

```
LoggedRegularCourse loggedRegularCourse
```

thereby avoiding the casting problems above.

This is not to say that casts do not occur with the extension approach. Consider the following two print methods:

```
void print (RegularCourse regularCourse) { … }
void print(LoggedCourse loggedCourse) { … }
```

and a variable declared to be of the extended interface:

```
LoggedRegularCourse loggedRegularCourse;
```

Which print() should be called in the following invocation?

```
print (loggedRegularCourse)
```

The type assignment rules allow both. Therefore, the programmer can “cast” to disambiguate

```
print((RegularCourse) loggedRegularCourse);
```

This cast is safe in that it is guaranteed to not lead to a runtime error. Its only purpose is to disambiguate
during compile time. For this reason, it is not a cast of the sort we have seen so far, whose purpose has been
to provide information about the runtime object assigned to a variable. In fact it is not really needed, as we
could declare an extra variable to disambiguate:

```
RegularCourse regularCourse = loggedRegularCourse;
print( regularCourse);
```

It simply serves as a shorthand for such a declaration.

In summary, the approach of implementing multiple interfaces can result in runtime errors due to casting
between the interfaces, while the approach of extending multiple interfaces can lead to a proliferation of
interfaces. As there is a tradeoff to be made in choosing between the two approaches, no one approach can
be considered superior. I personally prefer the extension approach as the extra programming overhead is
worth the reduced chance of runtime errors. This is not a universally accepted principle however.

Factories

As mentioned above, it is useful to create both the normal and logged versions of the courses as different
sites may wish to use different versions. It would be nice if it were easy to switch between these sets.
Fortunately, by using interfaces, we have already taken a step towards this goal. For example, if we look at
ACourseList, we see no code that is aware of the exact class of the stored courses. he class refers only to the
TreeNode interface, which is implemented by both sets of course classes. Even the main program is fairly
oblivious to the exact class – the print() methods assume only the Course interface, which is again
implemented by both configurations. The only code that is aware of the exact classes is the one in the
fillCourses() method of the main class that creates instances of the course classes and puts them in the
course list:

```
static void fillCourses() {
    CourseList prog = new ACourseList();
    prog.addElement (new ARegularCourse("Intro. Prog.", "COMP", 14));
    prog.addElement (new ARegularCourse("Found. of Prog.", "COMP", 114));
    courses.addElement(prog);
    courses.addElement (new AFreshmanSeminar("Comp. Animation", "COMP"));
    courses.addElement (new AFreshmanSeminar("Lego Robots", "COMP"));
}
```

How could we make it easy to change the course configuration in classes such as the main class
above that need to create courses? A related question is: How do we ensure that these classes do not
add incompatible objects to a course list such an unlogged regular course and a logged freshman
seminar?
The answer is to create, for each configuration of classes, a special class whose task is to instantiate the classes. Factories that instantiate alternative configurations, such as the unlogged and logged course sets, implement the same interface. As a result it is possible to easily switch between factories.

In our example, the common factory interface must define two methods, one for returning a regular course and another for returning a freshman seminar. It must also use, for the return types of these methods, interfaces that are common to the logged and unlogged classes, as shown below:

```java
package factories;
import courseTree.RegularCourse;
import courseTree.FreshmanSeminar;
public interface CourseFactory {
    public RegularCourse getRegularCourse();
    public FreshmanSeminar getFreshmanSeminar();
}
```

We must now create a factory class implementing this interface for the unlogged courses:

```java
package factories;
import courseTree.RegularCourse;
import courseTree.FreshmanSeminar;
import courseTree.ARegularCourse;
import courseTree.AFreshmanSeminar;
public class ACourseFactory implements CourseFactory {
    public RegularCourse getRegularCourse() {
        return new ARegularCourse();
    }
    public FreshmanSeminar getFreshmanSeminar() {
        return new AFreshmanSeminar();
    }
}
```

and one for the logged courses:

```java
package factories;
import courseTree.RegularCourse;
import courseTree.FreshmanSeminar;
import courseTree.ALoggedRegularCourse;
import courseTree.ALoggedFreshmanSeminar;
public class ALoggedCourseFactory implements CourseFactory {
    public RegularCourse getRegularCourse() {
        return new ALoggedRegularCourse();
    }
    public FreshmanSeminar getFreshmanSeminar() {
        return new ALoggedFreshmanSeminar();
    }
}
```
As we see above, defining a factory class is trivial—it simply instantiates classes in the configuration with which it is associated. We see the benefit of using factory classes in the modified implementation of fillCourses() below:

```java
static void fillCourses() {
    CourseFactory courseFactory = new ACourseFactory();
    CourseList prog = new ACourseList();
    RegularCourse introProg = courseFactory.getRegularCourse();
    introProg.init("Intro. Prog.", "COMP", 14);
    RegularCourse foundProg = courseFactory.getRegularCourse();
    foundProg.init("Found. Prog.", "COMP", 114);
    ...
}
```

Instead of directly instantiating a course class, the method calls a method in the factory class that returns an instance of the class. If the method wants to switch configurations, all it has to do is replace the line that creates the factory for unlogged courses:

```java
CourseFactory courseFactory = new ACourseFactory();
```

with code that creates the factory for logged courses:

```java
CourseFactory courseFactory = new ALoggedCourseFactory();
```

The alternative was to replace all instantiations of the unlogged regular and freshman seminar classes with the logged ones, a tedious and error prone task.

Notice that the factory used the parameterless constructors to instantiate the class. This is because it does not know how to initialize instances of the class, which only the user of the class knows. The user calls init() methods to initialize the returned objects, as shown above. An alternative might have been to define methods in the factory interface that take parameters whose values are supplied by the user:

```java
package factories;
import courseTree.RegularCourse;
import courseTree.FreshmanSeminar;
public interface CourseFactory {
    public RegularCourse getRegularCourse();
    public FreshmanSeminar getFreshmanSeminar();
    public RegularCourse getRegularCourse(String theTitle, String theDept, String theCourseNum);
    public FreshmanSeminar getFreshmanSeminar(String theTitle, String theDept);
}
```

The disadvantage of this approach is that each time a new kind of constructor is added to a class, the factory creating it must be changed. With the init method, only the class definer and class user need to worry about initialization parameters—the factory does not. The advantage is that object creation requires fewer calls.

**Factory Selector**

What if we wanted the main class to be completely oblivious of the exact configuration. We can define another class, called a factory selector, which is essentially a factory that creates a factory. Its interface is shown below:

```java
package factories;
public interface CourseFactorySelector {
    public CourseFactory getCourseFactory();
}
```
As before, the implementation is straightforward

```java
package factories;
public class ACourseFactorySelector {
    public CourseFactory getCourseFactory() {
        return new ACourseFactory();
    }
}
```

Instead of instantiating the course factory class, `fillCourses()`, now instantiates the factory selector class and calls a method in it to obtain a factory:

```java
static void fillCourses() {
    CourseFactory courseFactory = (new CourseFactorySelector()).getCourseFactory();
    ...
}
```

If logged course configuration is now desired, we change only the factory selector class:

```java
package factories;
public class ACourseFactorySelector {
    public CourseFactory getCourseFactory() {
        return new ALoggedCourseFactory();
    }
}
```

instead of the main and all other classes that create course configurations. Thus, a factory selector defines settings that are shared by all other classes. We will later see how we can define application-specific settings.

**Switching between vs. creating consistent configurations**

In the above example, factories were used for two purposes – to switch between course configurations and to create consistent course configurations. A factory user may be interested in one but not the other. For example, we can create a variation of the main class above that is bound to the logged course configuration, displaying the number of queries for each course. Such a class is not interesting in switching configurations. However, it may still use the factory to ensure that it does not create incompatible courses.

Such a class is not well served by factory methods whose return types do not reflect the specific object needed by it. Our new main class, for example, needs values of type `LoggedFreshmanSeminar` and `LoggedRegularCourse` but, instead, gets values of types `FresherSeminar` and `RegularCourse`. It could cast these values, but this process is always error prone. The solution is to have in a factory multiple creation methods, all instantiating the same class, but typing the returned object to various degrees of specificity. Thus, our logged course factory would be changed as follows:

```java
package factories;
import courseTree.RegularCourse;
import courseTree.FreshmanSeminar;
import courseTree.LoggedRegularCourse;
import courseTree.LoggedFreshmanSeminar;
import courseTree.ALoggedRegularCourse;
import courseTree.ALoggedFreshmanSeminar;
public class ALoggedCourseFactory implements LoggedCourseFactory {
    public RegularCourse getRegularCourse() {
```
Factories/facades

```java
return new ALoggedRegularCourse();
}
public FreshmanSeminar getFreshmanSeminar() {
    return new ALoggedFreshmanSeminar();
}
public LoggedRegularCourse getLoggedRegularCourse() {
    return new ALoggedRegularCourse();
}
public LoggedFreshmanSeminar getLoggedFreshmanSeminar() {
    return new ALoggedFreshmanSeminar();
}
}

A user who wishes to be able to switch between the two configuration would use the original set of methods while one that wishes to be bound to logged courses would use the other.

**Factory Pattern**

What we have seen above is an example of the factory pattern, shown in the diagram below:

![Figure 5 Factory pattern](image)

Different class configurations are created by different factory classes implementing the same interface.

We used the term “factory” earlier for a class. Here we use it for an object that instantiates other objects. Which usage is closer to a real world factory?

A class is closer to a real-world factory as it defines a blueprint for its instances. The factory class is a really an intermediate broker ordering objects from the real factories on behalf of its customers to insulate them from these factories and to ensure that consistent objects are obtained from them. A factory selector chooses between these brokers. Here is an analogy. I ask my IT department to get me a 4lb laptop and a couple of extra batteries and battery chargers. It decides to choose the IBM CCI configuration rather than some other configuration. The order is then sent to the office dealing with the CCI configuration, which orders a laptop and matching batteries and chargers from their manufacturers. Here the manufacturers correspond to classes, CCI office to a factory class, and the IT department to a factory selector. Taking the car analogy, a factory class is like a dealership that makes sure it has parts consistent with the car it sells.

Should objects always be created through factories? Just as we use intermediate brokers only when necessary, we use factories only when we need to switch between alternate configurations or we need a consistent set of related objects.

Toolkits area good example of substitutable configurations of consistent objects. Consider the Java AWT toolkit. It has a different implementation for each windowing system. There is one for Microsoft Windows and another for X windows. However, the programs we write are not aware of the exact implementation.
This is because Java has essentially defines a different factory for each configuration and automatically chooses the factory based on the underlying window system.

It also makes sense to allow programmers to select between toolkits. Java provides two toolkits: the AWT and Swing toolkits providing competing implementations for buttons, textfields, panels and other widgets. If each toolkit is associated with a factory, then a programmer-defined factory selector could choose between them to determine the widget set for all applications using the selector.

Unfortunately, it is not possible to write such factories. The reason is that alternatives provided by the two toolkits do not implement a common interface defining their common methods. For example, the getText() and setText() methods of the AWT and Swing textfield widgets are not defined in a common interface implemented by them. As a result, it is not possible to define common methods to create these widgets as there is no interface to use as the return type of these methods. This is yet another scenario motivating the need to define interface.

**Facades**

Let us look again at the main class of the course displayer:

```java
package main;
...

public class AFactoryBasedCourseDisplayer {

    static CourseList courses = new ACourseList();

    public static void main(String[] args) {
        // do I/O
        while (true) {
            System.out.println("Please enter course title:");
            String inputLine = readString();
            if (inputLine.equals("."))
                break;
            else {
                Course matchedCourse = courses.matchTitle(inputLine);
                if (matchedCourse == null)
                    System.out.println("Sorry, this course is not offered.");
                else {
                    printHeader();
                    print(matchedCourse);
                }
            }
        }
    }

    static void fillCourses() {
        CourseFactory courseFactory = (new CourseFactorySelector()).getCourseFactory();
        CourseList prog = new ACourseList();
        RegularCourse introProg = courseFactory.getRegularCourse();
        introProg.init("Intro. Prog.", "COMP", 14);
        RegularCourse foundProg = courseFactory.getRegularCourse();
        foundProg.init("Found. Prog.", "COMP", 114);
    }

    // I/O methods
}
```

Besides taking care of input and output, it composes the course list object with the two kinds of courses by filling courses in the course list. The composition and I/O tasks are independent – yet
we are doing them in a single class. If we were later to create a graphical user interface for the same composition, we would not be able to reuse the composition code.

Thus, it makes sense to create a separate class for the composition task. Such a class allows its users to see a new object that represents the combination of a set of individual objects. It is called a façade because, like the façade of a building, only selected aspects of its units are available to the outside. In our example, the façade needs to give its users a way to map input titles to courses. It does not have to bother the users with details of instantiating courses or putting them into course lists. Here is an interface to do this:

```java
package facades;
import courseTree.Course;
public interface TitleToCourseMapper {
  public String getTitle();
  public void setTitle (String newVal);
  public Course getCourse();
}
```

It is possible to define an even simpler interface that provides the following single method:

```java
public Course matchTitle(String title);
```

essentially exporting the matchTitle() method of a course list. An interface with getters and setters has the advantage that it allows the building of more flexible user interfaces, as we will see later.

Here is an implementation of the interface:

```java
package facades;
import courseTree.Course;
import courseTree.RegularCourse;
import courseTree.FreshmanSeminar;
import courseTree.CourseList;
import courseTree.ACourseList;
import factories.CourseFactory;
import factories.ACourseFactorySelector;
public class AFactoryBasedTitleToCourseMapper implements TitleToCourseMapper {
  CourseList courses = new ACourseList();
  public AFactoryBasedTitleToCourseMapper() {
    fillCourses();
  }
  String title = "";
  public String getTitle() {return title;}
  public void setTitle (String newVal) {
    title = newVal;
    course = courses.matchTitle(title);
  }
  Course course;
  public Course getCourse() {
    return courses.matchTitle(title);
  }
  void fillCourses() {
    CourseFactory courseFactory = (new ACourseFactorySelector()).getCourseFactory();
    CourseList prog = new ACourseList();
    RegularCourse introProg = courseFactory.getRegularCourse();
    introProg.init("Intro. Prog.", "COMP", 14);
  }
}
```
Factories/facades

```java
prog.addElement(introProg);
RegularCourse foundProg = courseFactory.getRegularCourse();
foundProg.init("Found. of Prog.", "COMP", 114);
prog.addElement(foundProg);
courses.addElement(prog);
FreshmanSeminar compAnimation = courseFactory.getFreshmanSeminar();
compAnimation.init("Comp. Animation", "COMP");
courses.addElement(compAnimation);
FreshmanSeminar legoRobots = courseFactory.getFreshmanSeminar();
legoRobots.init("Lego Robots", "COMP");
courses.addElement(legoRobots);
}
}

This class worries about the task of getting and using a factory for creating the courses, instantiating the course list, and putting courses in the course list.

The main class now only performs the task of implementing the user interface:
```java
package main;
...
public class AFactoryBasedCourseDisplayer {
    static TitleToCourseMapper titleToCourseMapper = new ATitleToCourseMapper();
    public static void main(String[] args) {
        while (true) {
            System.out.println("Please enter course title:");
            String inputLine = readString();
            if (inputLine.equals("")
                break;
            else {
                titleToCourseMapper.setTitle(inputLine);
                Course matchedCourse = titleToCourseMapper.getCourse();
                if (matchedCourse == null)
                    System.out.println("Sorry, this course is not offered.");
                else {
                    printHeader();
                    print (matchedCourse);
                }
            }
        }
    }
}
...
```

It first creates the façade object. Then, for each input string entered by the user, it sets the title of the façade to the string, gets from the façade the corresponding course, and calls print to display it.

The following diagram illustrates the general notion of a façade and its use in our example. A façade is a boundary around a set of component objects that exports the combined functionality. It is responsible for creating the composed objects, connecting them, and exporting methods representing the combined functionality. It connects an object to another by passing a reference to one of them to a method implemented by the other. Thus, in our example, it connects a course object to a course list by passing a reference to the course object to the addElement() method of the list. This connection is single-directional as the course object, on the other hand, is not given a reference to the course list. In some cases, it may be necessary to create bi-directional links between connected objects.

Factories/facades

A compiler is a good example of a program that creates a façade. It consists of several components including a scanner, which converts the input program to a token stream, a parser, which converts the token stream to a tree structure, and a code generator, which visits the tree. The user of the compiler, however, is not interested in these internal components. All it needs is a façade that converts the input string to the generated code. The compiler is responsible for creating and connecting these components.

**Factory Methods**

Consider again portions of the factory-based façade class:

```java
public abstract class AFactoryBasedTitleToCourseMapper implements TitleToCourseMapper {
    ...
    void fillCourses() {
        CourseFactory courseFactory = (new ACourseFactorySelector()).getCourseFactory();
        RegularCourse introProg = courseFactory.getRegularCourse();
        FreshmanSeminar legoRobots = courseFactory.getFreshmanSeminar();
        ...
        ...
    }
}
```

The class uses the factory selector to determine which configuration of courses is to be used. If a different configuration is needed for this application, we can change the selector class. What if other applications do not wish to share the configuration? In other words, the factory selector allows us to create a global configuration shared by all classes (using the selector). What if each application wishes to choose the configuration independently?

The general answer is to create an abstract class representing the application that defines and uses abstract methods choosing the configuration. For each alternative configuration in which we are interested, we create a concrete subclass that implements the abstract methods to choose the classes of the configuration. Thus, in the example above, we convert the above façade to the following abstract class:

```java
public abstract class AnAbstractTitleToCourseMapper implements TitleToCourseMapper {
    ...
    abstract RegularCourse getRegularCourse();
    abstract FreshmanSeminar getFreshmanSeminar();
    void fillCourses() {
        RegularCourse introProg = getRegularCourse();
        FreshmanSeminar legoRobots = getFreshmanSeminar();
        ...
        ...
    }
}
```
The class is identical to the previous one except that it defines an abstract method for each kind of course that returns the course. Instead of using a factory and factory selector to obtain a course, the class calls the abstract method. Even though the method is not implemented by the abstract class, the class can refer to it as it is guaranteed to be implemented by every concrete subclass class.

Now for the non logged courses, we define one concrete class:

```java
package facades;
import courseTree.Course;
import courseTree.RegularCourse;
import courseTree.FreshmanSeminar;
import courseTree.ARegularCourse;
import courseTree.AFreshmanSeminar;
public class ATitleToCourseMapper extends AnAbstractTitleToCourseMapper {
    RegularCourse getRegularCourse() {
        return new ARegularCourse();
    }
    FreshmanSeminar getFreshmanSeminar() {
        return new AFreshmanSeminar();
    }
}
```

and for the logged versions we define another concrete class:

```java
package facades;
import courseTree.Course;
import courseTree.RegularCourse;
import courseTree.FreshmanSeminar;
import courseTree.ALoggedRegularCourse;
import courseTree.ALoggedFreshmanSeminar;
public class ATitleToLoggedCourseMapper extends AnAbstractTitleToCourseMapper {
    RegularCourse getRegularCourse() {
        return new ALoggedRegularCourse();
    }
    FreshmanSeminar getFreshmanSeminar() {
        return new ALoggedFreshmanSeminar();
    }
}
```

If we wish to use the version choosing the non logged versions, we instantiate the first concrete class and if we wish to choose the logged versions, we instantiate the second one. Our selection does not influence any other application.

The following figure illustrates the general idea behind the factory method pattern and its use in our example:
Factories/facades

**Figure 7 Factory method pattern (left) and its use in course façade (right)**

The class structure we created, in our example, does not capture the fact that the second concrete subclass, ATitleToLoggedCourseMapper, is a special case of the first concrete class, ATitleToCourseMapper. The reason is that the objects returned by the factory methods of the second class are a special case of the objects returned by the factory methods of the first concrete class. As a result, ATitleToLoggedCourseMapper, can be used wherever ATitleToCourseMapper class can be used, as their names suggest. Thus, ideally speaking we should have created the class hierarchy shown in the following figure:

This hierarchy diligently follows the factory method pattern as described in the patterns book, but has the disadvantage of creating an apparently redundant abstract class. The only purpose if this, as far we can see, is to be subclassed by a single concrete class. Whenever a class has only one purpose - to be used in another class without itself ever being instantiated, then it should be combined with the other class to reduce the class proliferation problem. Thus, we can create an alternative structure that meets our goal of switching configurations without leading to class proliferation, shown on the right of the figure below. On the left we see the general pattern.

**Figure 8 Generalized factory method pattern and its use in the course façade**

Our first concrete class now defines, implements, and uses the factory methods:
Factories/facades

```java
public class ATitleToCourseMapper implements TitleToCourseMapper {
    ...
    RegularCourse getRegularCourse() {
        return new ARegularCourse();
    }
    FreshmanSeminar getFreshmanSeminar() {
        return new AFreshmanSeminar();
    }
    void fillCourses() {
        RegularCourse introProg = getRegularCourse();
        FreshmanSeminar legoRobots = getFreshmanSeminar();
        ...
    }
}
```

The second concrete class simply overrides the methods:

```java
public class ATitleToLoggedCourseMapper extends ATitleToCourseMapper {
    RegularCourse getRegularCourse() {
        return new ALoggedRegularCourse();
    }
    FreshmanSeminar getFreshmanSeminar() {
        return new ALoggedFreshmanSeminar();
    }
}
```

You will find many applications of this pattern in assignment 6.

**Factories vs. Factory Methods**

Both factories and factory methods make it possible to write code that obtains new objects without being aware of the exact class of these objects. As a result, the classes can be changed without changing the code. Factories (and factory selectors) are useful when we need to make a global choice applicable to all applications and factory methods are useful when we wish to localize the choice to a particular application. More specifically, if class C implements abstract factory method defined in super class S, then the configuration chosen by C applies to C, and all of its subclasses and all of its superclasses between C and S that do not override/implement it.

One of the problems with factory methods is that each application may have to define and implement a large number of methods. Moreover, it must ensure that compatible objects are returned by these methods. Imagine creating toolkit objects through factory methods. Each class that uses a toolkit must define and implement, for each kind of widget, a factory method creating the widget and ensure that incompatible widgets are not created.

Fortunately, this problem can be solved by combining the ideas of factory methods and factory classes. Instead of defining multiple factory methods directly instantiating different kinds of classes, we create a single method that simply chooses the factory. Methods that need the objects simply use the factory selected by the method. Thus, factory methods take the place of the factory selector rather than factories themselves. This is shown in the modified class below:

```java
public class ATitleToCourseMapper implements TitleToCourseMapper {
    ...
    RegularCourse getCoursefactory() {
        return new ARegularCourseFactory();
    }
    void fillCourses() {
        CourseFactory courseFactory = getCourseFactory();
    }
}
```
Factories/facades

RegularCourse introProg = courseFactory.getRegularCourse();
FreshmanSeminar legoRobots = courseFactory.getFreshmanSeminar();
...
}
}

The following picture shows the general pattern combining factory methods and factories:

Figure 9 Combining factory methods and factories

More real-world analogies
Before we end this chapter, here are some more real world analogies from the appliances world for factories and facades.

Imagine a VCR interface defining methods to power, set and get channels, record and play, and connectToTV. Imagine also a CHANNEL constant in it that determines the channel to which a TV must tune to receive VCR signals. Such an interface can be subtyped for VCRs that connect via RF (Coaxial jack) and those that connect via S-Video (RCA jack).

A TV interface could similarly offer methods to power, and get and set channels. It can also be subtyped for TVS that connect via RF and S-Video jacks.

Factories could be created for the RF and S-Video cases to ensure that incompatible TVs and VCRs and never bought. Moreover, a VCR-TV combo can be simulated by a façade object. In its constructor the facade connects the devices together and sets the TV channel. It exposes the VCR methods to set and get channels. Also, it offers a power() method to power() both devices. The convenience of combo units illustrates the usefulness of facades.