UNBC Laboratory Assignment
Computer Science 101—Winter 2016

Factory Methods, Clones, and Automation

Due Date:

This assignment is due Friday 2013-04-08.

Purpose:

To extend the Payroll system developed in Laboratory Assignment 3, and thereby gain familiarity with advanced Java idioms for object construction and polymorphism.

Copy constructors:

A copy constructor is a constructor whose sole argument is another object of the same class. Although not as important as in C++, copy constructors greatly simplify the operation of creating copies of objects.

Clone functions:

One of the disadvantages of copy constructors (and constructors in general), is that they cannot provide run-time polymorphism through overriding. On the other hand you *can* ask an object to make a copy of itself in a polymorphic way. This is the purpose of a `.clone()` method.

One generic technique for creating `.clone()` functions is to build the clone function from a copy constructor.[1]

```
public XXX clone() { return new XXX(this) ; }
```

(Note that the `.clone()` method is an override of a method in the `Object` class, despite the change in return type.)

Exercise 1

⇒ Add copy constructors and `.clone()` functions to the `Employee` class and each of the `Employee` subclasses of Laboratory Assignment 3.

⇒ Show that your modified code works at this stage.

[1] Another more specifically Java-based technique is to implement the `Cloneable` interface and use `super.clone()` to make a shallow copy, and then add whatever code is needed to provide a deep copy. This is technically tricky.

```
21  public static void processReport(Scanner employeeScanner, Report report)
22      {
23      String code = "--" ;
24      Employee current = new UnclassifiedEmployee() ;
25      bool employeeFound = false ;
26      bool done          = false ;
27
28      while (!done)
29          {
30          if (employeeFound)
31              {
32              report.print(current) ;
33              employeeFound = false ;
34              }
35
36          done = !employeeScanner.hasNext() ;
37          if (done) continue ;
38
39          switch(code = employeeScanner.next())
40              {
41              case "FT":  current = new Fulltime() ; break ;
42              case "PT":  current = new Parttime() ; break ;
43              case "CO":  current = new Contract() ; break ;
44              default:
45                  current = new UnclassifiedEmployee(code) ;
46                  break; ;
47              }
48          current.readDetailsFrom(employeeScanner) ;
49          employeeFound = true ;
50          }
51      return ;
52      }
```

Figure 1: Somewhat simplified `processReport`

```
54  public static void processReport(EmployeeFactory employeeFactory,
55                                   Scanner employeeScanner, Report report)
56      {
57      String code = "--" ;
58      Employee current = new UnclassifiedEmployee() ;
59      bool employeeFound = false ;
60      bool done          = false ;
61
62      while (!done)
63          {
64          if (employeeFound)
65              {
66              report.print(current) ;
67              employeeFound = false ;
68              }
69
70          done = !employeeScanner.hasNext() ;
71          if (done) continue ;
72
73          code = employeeScanner.next() ;
74          current = employeeFactory.getFreshEmployee(code) ;
75          current.readDetailsFrom(employeeScanner) ;
76          employeeFound = true ;
77          }
78      return ;
79      }
```

Figure 2: `processReport` with an `EmployeeFactory`

Employee Factories

Look at Figure 2 of Laboratory Assignment 3 (*p.* 5). Removing the construction of the scanner and report this can be simplified to Figure 1 on the preceding page.

We want to abstract away from the switch logic used to set the variable `current` used in Figure 1. To do this, we introduce the idea of a *factory*, that is, an object for generating other objects.

This gives us Figure 2. What is an `EmployeeFactory`? Well, let's make it an interface, so that we can supply increasingly sophisticated version of the idea. We clearly need a method `.getFreshEmployee`. We'll also add a method `.registerExample`, for reasons to be explained later. This interface is shown in Figure 3 on the following page.

It's easy to implement this interface, as shown in Figure 4 on the next page.

```
15 public interface EmployeeFactory
16 {
17   abstract public void registerExample(String employeeCode, Employee example) ;
18   abstract public Employee getFreshEmployee(String employeeCode) ;
19 }
```

Figure 3: The `EmployeeFactory` interface

```
1  public class FixedFactory implements  EmployeeFactory
2  {
3    public void registerExample(EmployeeCode c, Employee example) {}
4    public Employee getFreshEmployee(EmployeeCode code)
5        {
6        switch(code)
7            {
8            case "FT":  return new Fulltime() ;
9            case "PT":  return new Parttime() ;
10           case "CO":  return new Contract() ;
11           default:
12               return new UnclassifiedEmployee(code) ;
13           }
14       }
15 }
```

Figure 4: A simple `EmployeeFactory`

```
1  public class FlexibleFactory implements  EmployeeFactory
2  {
3    private java.util.TreeMap<String, Employee> myLinks ;
4
5    public FlexibleFactory()
6        {
7        myLinks = new java.util.TreeMap<String, Employee> () ;
8        }
9
10   public void registerExample(String ec, Employee example)
11       {
12       myLinks.put(ec, example) ;  // TODO! check for repeated registration.
13       }
14
15   public Employee getFreshEmployee(String code)
16       {
17       Employee e = myLinks.get(code) ;
18       return (e==null) ? new UnclassifiedEmployee(code) : e.clone() ;
19       }
20 }
```

Figure 5: A better `EmployeeFactory`

Exercise 2

⇒ Rework your main logic so that new employees are generated by an `EmployeeFactory` of some kind.

⇒ Show that your modified code works at this stage.

Better Factories

The factory used in Figure 4 is particularly simple. We would like a more flexible approach that allows us to change the factory settings on the fly. In order to do this in an automatated way, we need to have a data structure that relates `String`s to `Employee`s. The abstract `java.util.Map<K,V>` class and the concrete instantiation `java.util.TreeMap<K,V>` can be used to do this. This is shown in Figure 5.[2,3] Using this factory, we can get the equivalent of the previous one by writ-

[2]Look at the online documentation for the `java.util.Map` interface and the `java.util.TreeMap` class if the code does not seem clear. The most important methods are `containsKey`, `get`, and `put`.

[3]Note that we need `.clone()` to be sure that factory generates a new `Employee` every time that `getFreshEmployee` is called.

ing something like:

```
    private static EmployeeFactory setupFactory()
        {
        EmployeeFactory ef = new FlexibleFactory() ;
        ef.registerExample("CO", new Contract()) ;
        ef.registerExample("FT", new Fulltime()) ;
        ef.registerExample("PT", new Parttime()) ;
        return ef;
        }
```

Plug and Play

In fact we are now almost in a position to do something way more exciting: automate the `EmployeeFactory` so that it automatically picks up all of the `Employee` subclasses that have been compiled into a single directory.

This means that we can add a new class of `Employee` to our payroll system simply by writing the appropriate subclass `.java`-file and compiling it!

Figure 6 on the next page shows how you can construct an `ArrayList` of all of the classes that subclass from `Employee` and have a zero argument constructor. Here are some key points of the code

1. `fileNames` is a list of all of the files in `directory`.
2. `classNames` is a list of all of the files that end in `.class`, minus the `.class` part.
3. Line 21 attempts to construct a `Class` object for the corresponding `.class` file. The argument to `.forName` needs to have the qualifying package prefix if there is one.
4. Assuming that line 21 does not throw an exception, line 22 attempts to use the zero argument constructor of the corresponding class to build an object. (If there is no zero argument constructor an exception is thrown.)

Exercise 3

⇒ Add a `String getCode()` method to each `Employee` subclass so that given an Employee object, you can determine its employee code.

⇒ Using that method, an `EmployeeFactory` like that in Figure 5, and code like that shown in Figure 6, rewrite your code so that it dynamically determines the `Employee` subclasses available and uses them to build an appropriate `EmployeeFactory` to process a file.

Note that you'll need a `.getCode()` function in `Employee` subclasses so that you can call `.registerExample` correctly.

This part is hard.

```
1  private static ArrayList<Employee> employeesFromDirectory(
2      java.io.File directory)
3      {
4      String [] fileNames = directory.list() ;
5      ArrayList<String> classNames = new ArrayList<String> () ;
6      for (String name:fileNames)
7          {
8          if (name.endsWith(".class"))
9              {
10             classNames.add(name.substring(0,name.length()-6)) ;
11             }
12         }
13     ArrayList<Employee> employees = new ArrayList<Employee> () ;
14     for (String classString:classNames)
15         {
16         Object o = null ;
17         try
18             {
19             // next line assumes class is in package payroll
20             // adjust if necessary
21             Class c = Class.forName("payroll."+classString) ;
22             o = c.getConstructor().newInstance() ;
23             if (o instanceof Employee)
24                 employees.add((Employee) o) ;
25             }
26         catch (Throwable x) {}
27         }
28     return employees ;
29     }
```

Figure 6: Finding `Employee` subclasses dynamically

Exercise 4

⇒ Implement a `Commission` class that meets the description below. Compile your `Commission.java` file, and *without making any other changes to your code* see if your program can now handle `Commission` employees.

⇒ Create a `.jar` file that contains sub-directories for each of the Exercises.

Description of the Commission Employees

Commission Commissioned employees work 37.5 hours per week for a varying hourly rate. In addition they are payed a commission on sales, and possibly other expenses related to travel costs. A typical commission employee input record looks like

```
CM 19.31 4000.00 150.00
```

indicating an employee that earns $19.31 per hour, and had commissions totalling $4000.00 and other expenses totalling $150.00.

Deductions for commissioned employees are the same as for part time employees on hourly wages, together with a flat 15% on commissions earned. There are no deductions on the other payments.

Summary

Congratulations!

If you have made it this far, you have created a JAVA program that can be modified to handle arbitrary new classes of `Employees` *solely by adding* `.class`-files for those new classes.

This lab touches a number of new ideas:

- Factories;
- Dynamically loading `.class` files (implicitly through `Class.forName`); and
- Runtime reflection (`Class.forName`, `Class.getConstructor`, `Constructor.newInstance`).

Unfortunately, there is not time to do more!