UNIVERSITY OF JOHANNESBURG

# Computer Science 2A

## Practical Assignment 09

2016-04-28

Time: Deadline — 2016-05-03 12h00 Marks: 70

This practical assignment must be uploaded to eve.uj.ac.za **before** 2016-05-03 12h00. Late or incorrect submissions **will not be accepted**, and will therefore not be marked. You are **not allowed to collaborate** with any other student.

Good coding practices include a proper coding convention and a good use of JavaDoc comments. Marks will be deducted if these are not present. Every submission **must** include a batch file. See the reminder page for more details.

The Java Development Kit (JDK) has been installed on the laboratory computers along with the Eclipse Integrated Development Environment (IDE).

## This practical aims to solidify your understanding of the Abstract Factory Design Pattern.

The story continues from Practical08. After making the much-needed upgrades to the system, by implementing the Visitor Design Pattern, it is now necessary to implement upgrades to the way in which **CrewMember** are produced for use in the system. In order to do this, you must implement the Abstract Factory Design Pattern to decouple the use of **CrewMembers** from the way in which they are produced.

Start by creating a new package called ***acsse.csc2a.model_factory*** to separate the production of **CrewMembers** from the ***model*** which defines them.

Inside of the ***acsse.csc2a.model_factory*** package, create an empty interface called **EntityProduct** which has no constants or method signatures. (This interface will serve as a maker to determine which objects can be produced by our factory.)

Next, go to each of the classes which derrive from **CrewMembers** (**CrewCombat**, **CrewEngineer**, **CrewMedic**, **CrewPsychic**, and **CrewScience**) and set them to implement the **EntityProduct** interface. (Thereby marking them as being producible.) As there are no methods in the interface, there are no further changes required in these classes.

Inside of the ***acsse.csc2a.model_factory*** package create an abstract class called **EntityFactory**. The **EntityFactory** must have five(5) abstract methods inside of it (one for each of the different types of **CrewMember**). These methods are called ***ProduceCrewPsychic***, ***ProduceCrewScience***, ***ProduceCrewCombat***, ***ProduceCrewEngineer***, and

***ProduceCrewMedic*** respectively. Each of these methods must return an object of type ***EntityProduct***. Each of these methods accepts seven(7) parameters detailed as follows:

- A ***String*** for the crew member's ID
- A ***E_CREW_RANK*** for the rank of the crew member
- A ***String*** for the crew member's surname
- A ***String*** for the crew member's type
- A ***String*** for the crew member's special
- An ***int*** for the crew member's level
- A ***String*** (*OR* an ***int*** depending on the type of crew member) for their special value

Still inside of the ***acsse.csc2a.model_factory*** package, create a class called **CrewFactory** which extends **EntityFactory**. In **CrewFactory** implement each of the five methods from the abstract base class. These methods return a new **CrewMember** of each of the particular types. Therefore, the ***ProduceCrewPsychic*** will return a new **CrewPsychic** as its product. Do likewise for each of these methods.

You will have to make changes to the **CrewRoster** class in order to make use of the Abstract Factory Design Pattern. Inside of the ***readRoster*** method create a variable called *factory* of type **EntityFactory** (the abstract factory) then initialise it with a new concrete **CrewFactory** instance.

Once you have read a valid line from the roster file and determined what type of **CrewMember** to produce, change the way in which the **CrewMember**s are produced. (Currently, you statically create a new instance of **CrewPsychic**, **CrewCombat**, etc. using the ***new*** key word.) You must change this to get the ***factory*** instance to ***Produce*** the **CrewMember** for you. This can be done by calling one of the five produce methods specified in the **CrewFactory**. Which method is called is dependant on the type of **CrewMember** you need to create. Store the product produced by the **CrewFactory** in the ArrayList of **CrewMember**s exactly as it was before. ***(Note: You will have to cast the Product produced by the factory to the type of CrewMember it is or you will get a TypeMismatch error.)***

The remaining classes are left unchanged.

Note: Once you have made all of the specified changes, your program should run and look exactly the same as it did before!

Bonus: Implement the Singleton Design Pattern to ensure that only one instance of the **CrewRoster** can ever be created. In order to do this you will have to make changes to the **CrewRoster** class to implement the Singleton Design Pattern as well as change the way in which the **ShipFrame** works by making use of the single **CrewRoster** instance rather than calling static methods in the **CrewRoster**. [Note: You have not been taught the Singleton Design Pattern, you will have to research it yourself. No help will be given for this practical bonus.]

# Mark sheet

1. **EntityProduct** interface with no constants or methods. [02]
2. Each type of **CrewMember** implements **EntityProduct**. [05]
3. Abstract **EntityFactory**: 5 methods which return **EntityProducs** + correct parameters. [05]
4. Concrete **CrewFactory**: Implements 5 methods + return the correct type of **CrewMember** (**EntityProduct**). [05]
5. **CrewRoster**
   (a) Has **EntityFactory** reference. [01]
   (b) Initialises **EntityFactory** with a **CrewFactory**. [02]
   (c) Gets **CrewFactory** instance (factory) to ***Produce*** the correct type of **EntityProduct**. [10]
   (d) Casts **EntityProduct** from the factory to the correct type of **CrewMember** to store in the ArrayList. [05]
6. Packages [05]
7. Coding convention (structure, layout, OO design) and commenting (normal and JavaDoc commenting). [10]
8. Correct execution (implementation of working Abstract Factory). [20]
9. Bonus: Correct implementation of Singleton Design Pattern (max 10 marks). [00]

# NB

## Submissions which **do not compile** will be capped at 40%!

Execution marks are awarded for a correctly functioning application and not for having some related code.

# Reminder

Your submission must follow the naming convention as set out in the general learning guide.

```
SURNAME_INITIALS_STUDENTNUMBER_SUBJECTCODE_YEAR_PRACTICALNUMBER
```

**Example**

| | |
|---|---|
| **Surname** | Berners-Lee |
| **Initials** | TJ |
| **Student number** | 209912345 |
| **Module Code** | CSC2A10 |
| **Current Year** | 2016 |
| **Practical number** | P00 |

```
Berners-Lee_TJ_209912345_CSC2A10_2016_P00
```

Your submission must include the following folders:

- `bin` - *(Required)* Should be empty at submission but will contain runnable binaries when your submission is compiled.
- `docs` - *(Required)* Contains the batch file to compile your solution, UML diagrams, and any additional documentation files. Do not include generated JavaDoc.
- `src` - *(Required)* Contains all relevant source code. Source code must be places in relevant sub-packages!
- `data` - *(Optional)* Contains all data files needed to run your solution.
- `lib` - *(Optional)* Contains all libraries needed to compile your solution.

## NB

Every submission **must** include a batch file. This batch files must contain commands which will compile your Java application source code, compile the associated application JavaDoc and run the application. **Do not** include generated JavaDoc in your submission. All of the classes/methods which were created/updated need to have JavaDoc comments.