Innopolis University | Software Architecture | Spring 2016

The Poporo Social Network

Software Design Patterns

Néstor Cataño

Project 03 (10%)

Issued: April/04/2016
Due: April/29/2016

Problem Description

The goal of this project is to design and implement the *Poporo* social network. You will do so by combining several *software design patterns*. During this project deliverable we will be exploring and discussing issues related to software implementation. The two previous deliverables exercised architectural aspects. For the sake of simplicity of this project deliverable, we will abstract away from the architectural issues related to Poporo and rather focus on the software patterns necessary to implement it. If you suppose that Poporo is implemented using an MVC pattern, then the M part must includes Poporo's core functionality and business rules. If you think of Poporo as adhering to a typical client-server architecture, then you might want to assume that we want to implement all the services provided by the server side, which are used by the clients of the social network site, which we also want to implement. For this project deliverable we will be manipulating and exercising software structures in memory and not featuring a database for manipulating persistent data. This of course is inviable in practice due to the high number of users that social networking sites typically have, however, we are here rather interested in looking at the use of various software patterns to prototype Poporo.

The Poporo social network features personal information (e.g. gender, birthday, family situation), media content (personal photos and videos), and a continuos stream of social network activity logged from actions taken on Poporo's site (such as messages sent, status updated, games played). The privacy and the security of the information manipulated by Poporo is a significant concern, and thus measures are to be taken to prevent the unauthorized manipulation of media content. Typically, users may want to upload media they wish to share with specific friends, but do not wish to be widely distributed to the social network as a whole.

What makes Poporo different from other social networks, e.g. Facebook, Sapo and many others, is its peculiarity on the way it implements security and privacy polices on content. Poporo enforces such a policies by using a correct-by-construction approach whereby the manipulation of network content is carried out through operations (methods) with well-established software contracts. In this project deliverable you will be required to implement and to demonstrate the correctness of some these software contracts by using Hoare logic and Weakest Precondition calculus (WP) techniques.

Poporo's security and privacy polices are realized through the use of *access permissions*. Social network users may have *edit* or *view* permissions on content items. These two types of permissions keep a hierarchy. Hence, if a network user has an *edit* permission on some content item, then the user is certain that he has a *view* permission on that content too. The converse property does not always hold, hence, it might be the case that a user may see a content item but cannot modify it.

Poporo features three types of friendship relations, namely, *best-friends*, *social-friends* and *acquaintances*, which have an implicit hierarchy. The *best-friend* relation is a stronger relationship than

social-friend. That is, any of my best-friends will have any of the permissions that any of my social-friends can have on any of the contents that I own. The *social-friend* relationship is a stronger relationship than *acquaintance*.

Social networking users *own* social network content, and hence, any network user that owns a content item will (always) be able to edit it, so view it. Content items are mapped (stored) into user's pages. You might have a visible content item on your homepage, but that does not necessarily mean that you own that content item. Someone else may own that content, which you happen to have view permission upon.

Homepages have a *wall*, a feature that's common to most typical social network websites. People can comment on content items (e.g. photos) and those comments will show up in the wall. Content items (comments, e.g. text) that appear on the wall enjoy the same types of privileges than other types of content items, e.g. users may have edit or view permissions on wall content items too.

A list of software requirements for Poporo are shown below. You might want to consult slides of session 06 (*06-SA-Reliability.pdf*) for a complete presentation on those requirements. They are grouped into functional, security, and safety invariant properties. The list is intentionally left incomplete. In particular, the list does not list no requirements about the wall.

FUN-1: The social network shall have users and data (content)

FUN-2: The user who uploads data shall be classified as the owner of the data

FUN-3: The users of the social network shall upload data

FUN-4 Users might choose what data available to them is viewed by them

SEC-1: The users shall have controlled access to the data on the network based on permissions

SEC-2: The following permissions (privileges) over a given data may be given to a user:
i. the permission to view the data
ii. the permission to edit the data

INV-1: Users that can edit data must also be able to view it

INV-2: The owner of some data has all the permissions on it

Step-by-Step Implementation of Poporo

The following steps will take you through the implementation of a prototype of the Poporo social network. The implementation will be carried out on an incremental basis. Each step implements a different aspect of Poporo. You will combine several software design patterns to complete the prototype implementation of Poporo. You will also need to prove the correctness of some aspects of Poporo. You **must** implement this prototype using the Eclipse IDE. This will help us grading your project deliverable more easily. You will need to turn in your project implementation as an Eclipse project in the end.

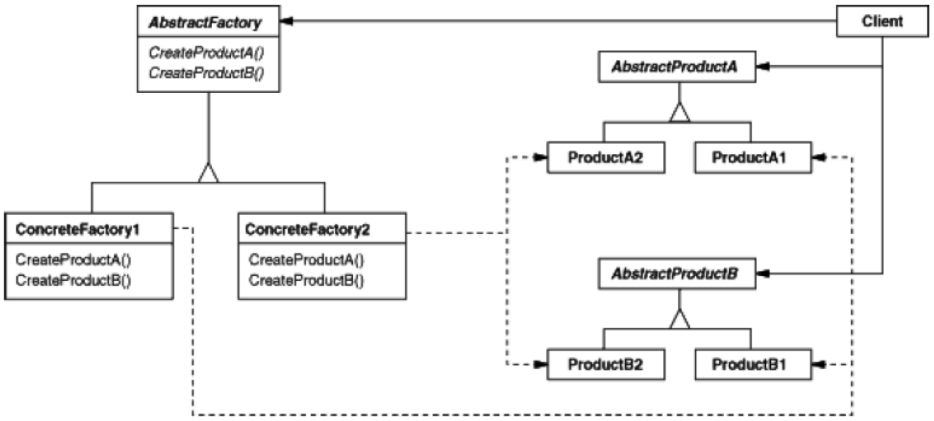

Figure 1: The Abstract Factory Software Design Pattern

1. Create an Eclipse project for your application. You **must** use Eclipse Luna, and Eclipse IDE for Java EE developers. You can refer to the first Lab of this course for a complete introduction to Eclipse. Assuming that you're working with Java, select *File*, *New*, *Project*, *Java Project*. Name your project The name of your project must be **poporo-<Student-Number-01>-<Student-Number-02>...<Student-Number-05>**. Next, in Eclipse right-click *src* and select *New*, *Java Package*, and create a package called `innopolis.poporo`. Use this package to store your implementation of the Poporo social network. More concretely, put within this package any class that you create.

2. You will use the Abstract Factory (AF) software pattern to provide Poporo with an initial structure. This structure will be extended during the subsequent steps. Figure 1 presents the schema of the AF software pattern. For a complete presentation on AF, you're invited to consult lecture 12 of the course Software Architecture, the slide presentation `12-SA-AFactory.pdf`, and the book *Design Patterns: Elements of Reusable O-O Software*. You might also want to check the code of the Widget Factory example that was presented during lecture 12. AF is a *creational* design pattern. This means that AF is mainly used to abstract object creation and instantiation processes. In this step you will use AF to design the creation of objects in the Poporo social network. Poporo manipulates the following types of objects: content items (class `ContentItem`), social network users (class `User`), user pages (class `Page`), and social network accounts (class `Account`). Use AF to implement the creation of those objects. You **must** stick to the following name convention for object creation. The name of the method that creates an object of type `ContentItem` must be `createContentItem`, the name of the method that creates an object of type `User` must be `createUser`, etc.

   Name `PoporoFactory` Poporo's abstract factory class. Poporo implements two types of concrete factory classes (see Figure 1), namely, `StandardPoporoFactory`, and `EnchantedPoporoFactory`. The first class implements Poporo's standard social networking functionality, the second one incorporates games. Stick to the standard name convention suggested by AF in Figure 1. For instance, for each object class, two additional classes are to be created, one for the standard version of Poporo and one for the enchanted version. Therefore, you will need to create and implement classes `StandardContentItem` and `EnchantedContentItem` for class `ContentItem`, `StandardUser` and `EnchantedUser` for class `User`, etc.

   The wall won't be designed and implemented as an AF (Abstract Factory) object, but as a decorator through the use of the Decorator design pattern later on.

3. Use class `Client1` the current functionality of your implementation of the Poporo social network. `Client1` will permit you to check if your code compiles and runs correctly or not. The code of class `Client1` is available from Moodle. Download class `Client1` and add it to your Eclipse project. Then, right-click `Client1` and select *Run As*, *Java Application*. You might want to add some `System.out.println` instructions to `createContentItem`, `createPage`, and `createUser` to observe outputs when `Client1` is running. Make sure that class `Client1` compiles within your project and runs without producing errors (see the Grading Criteria section for more information on this).

```
package innopolis.poporo;

public class Client1 {
  private PoporoFactory pf = null;

  public Client1(PoporoFactory p) {
    pf = p;
  }

  public void run() {
    ContentItem c = pf.createContentItem();
    Page p = pf.createPage();
    User u = pf.createUser();
  }

  public static void main(String[] args) {
    EnchantedPoporoFactory epf = new EnchantedPoporoFactory();
    Client1 c = new Client1(epf);
    c.run();
  }
}
```

4. During this step you will refine parts of your prototype implementation of Poporo. You will edit your implementation of a *user page* (class `Page` and its sibling classes). Sibling classes are either sub-classes or classes that might use class `Page`. Notice that user pages must include information of the user who owns the page and the content items that are part of the page. You might want to code that information as class attributes. Keep in mind that we have decided to implement the creation of objects in memory (e.g. by using data structures like lists, bags, arrays or vectors) rather than using a database engine to make data persistent. Also keep in mind that a content item cannot appear duplicated in a user's page.

   Make the following editions to class `Page` and/or sibling classes.

   (a) Implement method `void upload(ContentItem c)` that adds `c` to the user's page.

   (b) Implement method `boolean remove(ContentItem c)` that removes `c` from the list of content items of the user's page. The method returns `true` if the content already existed in the user's page, and `false` if it didn't.

   (c) Implement method `public boolean isEmptyPage()` that returns `true` if only if there is no content item in the user's page.

   (d) Implement method `public boolean containsContentItem(ContentItem c)` that returns `true` if and only if `c` is in the user's page.

(e) Implement a constructor `public StandardPage(ContentItem c)` for class `StandardPage` that sets the page content to contain a sole content item `c`. Add a similar constructor `public EnchantedPage(ContentItem c)` to class `EnchantedPage`.

At this point, your implementation of class `Page` must look like below.

```
public abstract class Page {
 public abstract void upload(ContentItem c);
 public abstract boolean remove(ContentItem c);
 public abstract boolean isEmptyPage();
 public abstract boolean containsContentItem(ContentItem c);
}
```

5. The goal of this step is to validate your implementations of classes `StandardPage` and `EnchantedPage`. JUnit test classes `StandardPageTest` and `EnchantedPageTest` are available from Moodle. Download them, include them in your poporo project, and run them (see the Grading Criteria section). Use TDD techniques to make sure that your implementation successfully passes all the JUnit tests defined in the above JUnit classes. The main validations performed by the JUnit classes are *i.*) a user's page-content is never empty, *ii.*) if a content item is uploaded to a page, then the page contains that content item, and *iii.*) if a content item is uploaded and then removed from a page, then the page no longer contains that content item. The code below shows some parts of the last validation. To run the tests in Eclipse, right-click `StandardPageTest` or `EnchantedPageTest`, then select *Run As*, *JUnit Test*, *Use configuration specific settings*, *Eclipse JUnit Launcher*, *o.k.*

```
@Test
public void testUploadRemove() {
  p = new StandardPage(c); // c is defined in another scope

  StandardContentItem c1 = new StandardContentItem();
  p.upload(c1);
  assertTrue(p.containsContentItem(c1));
  p.remove(c1);
  assertFalse(p.containsContentItem(c1));
}
```

6. The goal of this step is to complete the implementation of user accounts (class `Account` and siblings). An account features information about the user that owns the account and its page. You might want to encode such a featured information as class attributes of proper classes. Class `Account` is shown below. You must implement classes `StandardAccount` and `EnchantedAccount` to override the implementation of methods `openAccount`, `closeAccount`, `transmit`, and `contains`. Method `openAccount` returns the `User` to whom the account was created. Method `openAccount` must adhere to LSP (Liskov Substitution Principle), hence (at least) `openAccount`'s return type in classes `StandardAccount` and `EnchantedAccount` must be *variant* with respect to its return type in class `Account`. Method `closeAccount` deallocates all the resources that have been allocated to the account. Method `transmit` uploads a content item `c` into the page associated to the account. Method `contains` returns `true` if and only if the user's page contains content item `c`.

```
public abstract class Account {
 public abstract User openAccount();
```

```
 public abstract void closeAccount();
 public abstract void transmit(ContentItem c);
 public abstract boolean contains(ContentItem c);
}
```

7. During this step you will check if your implementation for user accounts compiles and runs fine. This step is a first and basic checking; a more complete checking is carried out during next step. Download class `Client2` from Moodle, and add it to your Eclipse project. In Eclipse, right-click `Client2` and then select *Run As*, *Java Application*. You can add some `System.out.println` instructions to your implementations of user accounts so as to observe outputs produced by your running code. Make sure that `Client2` compiles and does not produce no runtime errors. This is one criterion mentioned in the Grading Criteria section.

8. The goal of this step is to validate your implementation of classes `StandardAccount` and `EnchantedAccount` (see the Grading Criteria section). Download the JUnit test files `StandardAccountTest` and `EnchantedAccountTest` from Moodle. Include them in your Poporo project, and run them. Use TDD techniques to make sure that your implementation successfully passes all those JUnit tests. To run the tests in Eclipse, right-click `StandardAccountTest` or `EnchantedAccountTest`, then select *Run As*, *JUnit Test*, *Use configuration specific settings*, *Eclipse JUnit Launcher*, *o.k.*

9. The goal of this step is to use the *Decorator* software pattern to embellish Poporo's interface. You must use the Decorator pattern to decorate user pages (class `Page` and siblings) with a *wall*. You must achieve this by using a `WallDecorator`. Figure 2 presents the general schema of the Decorator software design pattern. For a complete presentation on the Decorator software pattern, you're invited to consult lecture 13 of the course Software Architecture, the slide presentation `13-SA-Decorator.pdf`, and the book *Design Patterns: Elements of Reusable O-O Software*.

   (a) For the purpose of visualizing user pages, add a method `Draw()` to your definition of class `Page`. At this point, your implementation of class `Page` must look like below. Method `Draw()` is central to the decoration of user pages.

   ```
   public abstract void upload(ContentItem c);
    public abstract boolean remove(ContentItem c);
    public abstract boolean isEmptyPage();
    public abstract boolean containsContentItem(ContentItem c);
    public abstract void Draw();
   }
   ```

   (b) Define a **class** `Decorator` to decorate elements of type `Page`. This class must include a *concrete* method **public void** `Draw( ) { ... }`. Short Note: in this context, *concrete* means that the method is not **abstract**, but it rather has an actual implementation within the class.

   (c) Define a **class** `WallDecorator` to decorate pages with a wall. This class must include a method **public void** `DrawWall( ) { ... }`

   Download class `Client3` from Moodle. This class will permit you exercise your implementation of the social network wall. Copy the class to your Poporo project. Make sure `Client3` compiles fine and produces no runtime errors (see the Grading Criteria section). I present the code of class `Client3` below.

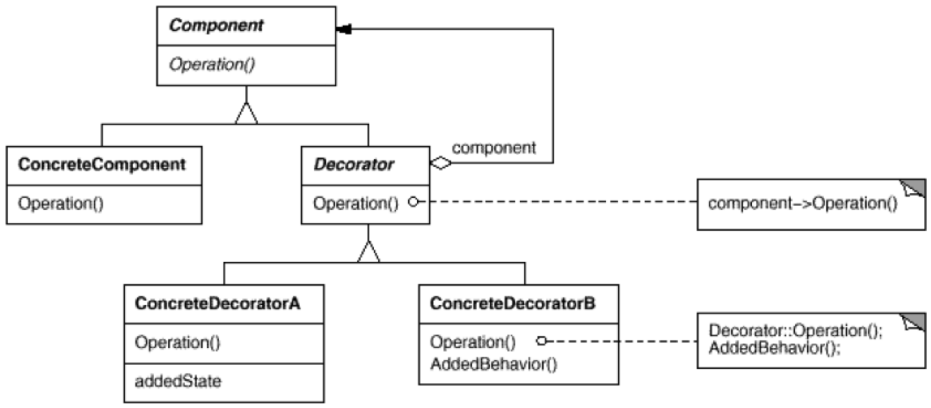

Figure 2: The Decorator Software Design Pattern

```
package innopolis.poporo;

public class Client3 {
 public static void main(String[] args) {
  StandardContentItem c = new StandardContentItem();
  StandardPage p = new StandardPage(c);

  Page pageDecorated = new WallDecorator(p);
  pageDecorated.Draw();
 }
}
```

10. Some aspects of Poporo are left unimplemented. These aspects may be subject of further evaluation during the Final Exam. Aspects that are left unimplemented include, access permission, friendship, content owner, and the use of Hoare and WP precondition calculus for program correctness.

Grading Criteria

You will be graded according to the following algorithm. Therefore, any project with compile time errors will be graded 0. The maximum number of marks given to a project is 10. In the algorithm below, `poporo` refers to your implementation of Poporo the Poporo social network. Classes `Client1`, `Client2`, `Client3`, `StandardAccountTest`, `EnchantedAccountTest`, `StandardPageTest`, and `EnchantedPageTest` are all reachable from Moodle. Download them and install them in your Poporo project.

```
marks = 0;

if(poporo compiles) {
  if(Client1 compiles and produces no runtime errors) {
    marks = marks + 1;
    if(Client2 compiles and produces no runtime errors) {
```

```
      marks = marks + 1;
      if(poporo passes all the tests in StandardAccountTest) {
        marks = marks + 1;
      }
      if(poporo passes all the tests in EnchantedAccountTest) {
        marks = marks + 1;
      }
      if(poporo passes all the tests in StandardPageTest) {
        marks = marks + 2;
      }
      if(poporo passes all the tests EnchantedPageTest) {
        marks = marks + 2;
      }
      if(Client3 compiles and produces no runtime errors) {
        marks = marks + 2;
      }
    }
  }
}
```

What to Turn In?

You should send us your Eclipse project as a Zip file. The name of your project file must be **poporo-<Student-Number-01>-<Student-Number-02>...<Student-Number-05>.zip**. To create a Zip file out of your project in Eclipse, you should right-click your project and select *Export ...*, *General*, *Archive File*. Then, click *Next* and name your file as above. Finally, click *Finish*.