patternCoder for BlueJ – Overview

1. INTRODUCTION

This document gives an overview of the patternCoder extension for BlueJ, which has been developed at Glasgow Caledonian University. The overview includes: installation instructions; a description of how the extension implements patterns; and a description of a learning activity which makes use of it.

2. INSTALLATION

The basic distribution of the patternCoder extension can be downloaded from *www.patterncoder.org* in a single archive, *patterncoder_<version>.zip* or *patterncoder_<version>.tar.gz*. You must have BlueJ itself installed before adding the extension. To install the extension, simply extract the contents of the archive into a suitable location, and copy the file called *patterncoder.jar* and the folder called *patternfiles* into your BlueJ *extensions* folder, which will usually be *<BLUEJ_HOME>/lib/extensions* (Windows and Linux) or *<BLUEJ_HOME>/ /BlueJ.app/Contents/Resources/Java/extensions* (Mac – to access this you should Control-click *BlueJ.app* and choose *Show Package Contents*).

The *patternfiles* folder includes a set of XML pattern files. You can use these patterns, modify them as you wish, or add your own. The intention of the distribution is to provide examples rather than a full catalog of patterns. The standard distribution contains two sets of patterns in separate subfolders:

- *StarterPatterns:* a collection of simple versions of well-known design patterns - this limited set of patterns ties in with sample teaching materials which are available for download)
- *BasicRelationships:* a collection of implementations of basic binary class relationships – again these tie in with sample teaching materials.

Each folder contains XML files and two subfolders, *images* and *templates*.

In order to be available at run-time, pattern files must be located in the root *patternfiles* folder. You should copy the pattern files which you want to make available from the relevant subfolder into the root *patternfiles* folder. You should also copy the relevant files from the *images* and *templates* folders within the subfolder to the corresponding folder in the root *patternfiles* folder.

A couple of things to note:

1. There is already a copy of the starter patterns in the root *patternfiles* folder – you can delete these if you don't want to use them.
2. We intend to implement proper support for reading from subfolders, but this has not been done yet.

If you have installed successfully, then after starting BlueJ you should find a new menu option *PatternCoder* in BlueJ's *Tools* menu.

3. THE PATTERNCODER EXTENSION FOR BLUEJ

The BlueJ IDE allows the integration of third party extensions by way of an application program interface (API). The design patterns extension is a pure java application, and makes use of the BlueJ extensions API in order to integrate itself into BlueJ.

3.1 Pattern source files

The extension is distributed with a basic set of patterns. However, it was decided that it is important that anybody wishing to use the extension be able to add, remove or alter the design patterns that are available. To achieve this, all information about each of the patterns and the steps to be taken during the wizard is stored in an external source file. The pattern source files have two functions within the design patterns extension:

1. To provide information about the pattern and define each of the components that must be created as part of that pattern.
2. To define the structure and sequence of the wizard process.

The pattern source files are written in XML, so it is possible to edit files easily in any text editor or xml authoring tool that the user wishes.

3.2 Defining a pattern

The Design Patterns extension loads each of the installed source files when it is selected in the first stage of the wizard. When it is loaded, the information is extracted from the file and used to display information about the currently selected pattern. Information about the components and wizard steps is also extracted and used to alter the behaviour of the wizard. The following extract shows how the pattern is defined initially:

```
<?xml version="1.0"?>
<pattern patternName="Decorator"
 patternImage="decorator.bmp">
    <patternDescription>
        <![CDATA[ Description of decorator
         pattern]]>
    </patternDescription>
```

The `<pattern>` XML tag is the outermost tag of any source file, and all information about the pattern is contained within this tag. The above information is used by the extension to show details of the pattern when the pattern is selected on the initial selection screen of the wizard.

3.3 Defining components

The source files, as well as supplying descriptive information about the pattern, also define the components that will be created when it is adopted. A component is defined within the XML tag `<class>`; all information related to a component is found within this element.

In order to reference a component within the source file, a unique *id* is assigned to each component that is defined within a particular source file. This *id* is used to associate particular components with a step in the wizard, and also for component renaming purposes.

For each component of the pattern, it is also necessary to detail any other components with which it may have an association or dependency. This is necessary in order for the renaming of components during the wizard process to work correctly. This is done by referencing the class *id* of the components to which it associates. The following extract is taken from the Decorator pattern, and shows how a class element would be constructed. In this example the component in question has an association with a class of *id* ="1":

```
<class classId="3" compType="Decorator"
 defaultName="Decorator"
 classDescription=”Description”
 classTemplate="Decorator/Decorator.TMPL">
    <dependantClass value="1"/>
</class>
```

Other attributes of the class element deal with descriptive information about the component which can be displayed in the wizard screens. The `classTemplate` attribute details the template file that the component should use to create the java source file. This is explained in more detail in section 2.5.

3.4 Defining wizard steps

The source file defines the steps that will be carried out during the wizard process. The overall wizard procedure is enclosed within the `<wizard>` XML tag. Further `<step>` elements are then defined, representing an individual step of the wizard.

As with the component definitions, each step must be assigned a unique *id*. In the case of the wizard, this is used to uniquely identify each step and link all the steps together to produce an order in which to display them. The attributes `nextStepId` and `previousStepId` correspond to the unique *ids* of other steps defined in the wizard element. If no step is in front or behind, then attributes are left blank, indicating that it is either the start or end of the wizard process. The following extract shows the definition of one step:

```
<wizard>
    <step stepId="1" type="class" compId="1"
     nextStepId="2" previousStepId=""
     stepName="Rename Component"
     stepDesc="Use the text field to enter a
     relevant name for the Component
     class.">
    </step>
</wizard>
```

There are further descriptive attributes for each step, `stepName` and `stepDesc`. These can be edited to display appropriate title and short description at the top of the wizard dialogue box.

3.5 Template files

To store information about the structure of classes that are to be implemented, a way of storing information about each class was required. Template files are already used by the BlueJ environment in order to store the structure of

classes, abstract classes, interfaces etc. that can be added to a user's project. In keeping with the BlueJ approach, similar template files were created for each pattern component. The following shows an excerpt from the Decorator template used within the decorator pattern:

```
public class $CLASSNAME implements $DEPENDANT1
{
   protected $DEPENDANT1 component;

   public $CLASSNAME($DEPENDANT1 component)
   {
      this.component = component;
   }
```

The template files store the Java source that should be generated for that particular component to work within the pattern once it is implemented. As the user can change the name of these components during the wizard process, any reference to a components name is replaced by a keyword. The keyword "$CLASSNAME", represents the name of this component, and when the source file is created, after completing the wizard steps, this keyword is replaced by the assigned name.

A pattern involves classes which have well-defined relationships. Therefore, unlike the BlueJ templates, the pattern component templates require knowledge of the name of other components within the same pattern. As these can also be renamed, the name can not be written into the templates, instead an additional keyword "$DEPENDANT" was created. Where this keyword is present, it indicates that it should be replaced by the name of another component. The required component is specified by appending the components unique *id* to the keyword, i.e. "$DEPENDANT1", shows an association with a component with a unique *id* of 1.

A template file has the name <*class_name*>.*TMPL*, where <*class_name*> is the generic name for the pattern component which the template represents. The set of templates for a pattern is contained in a folder with the same name as the pattern.

3.6 Image files

For each pattern, an image is provided to illustrate the components of the pattern and their relationship in the form of a class diagram. These files are simply bitmap files, and can be created using a diagramming tool. Some of the provided images have been prepared simply by creating a BlueJ project and taking a screenshot of the BlueJ class diagram. The image file for a pattern has the name <*pattern_name*>.*bmp*, where <*pattern_name*>.*xml* is the name of the pattern source file.

3.7 Installing patterns

In order install a new pattern which you have created yourself or downloaded, you simply need to copy the relevant files into the following folders:

- XML pattern source file <*pattern_name*>.*xml* - <*BLUEJ_HOME*>*/lib/extensions/patternfiles*
- Tempate files – contained in folder <*pattern_name*> in <*BLUEJ_HOME*>*/lib/extensions/patternfiles/templates*
- Image file <*pattern_name*>.*bmp* - <*BLUEJ_HOME*>*/lib/extensions/patternfiles/images*

4. USING PATTERNCODER

The patternCoder extension is intended to be used as a learning tool within learning activities which are designed to encourage students to apply their knowledge of design patterns within problem solving contexts. The role of the tool within such an activity includes providing support in producing a correct pattern-based implementation and providing insight into the roles and operation of the constituent classes during assembly of the solution and by inspection and interaction with the classes post-creation.

We illustrate this role with a walkthrough of the application of the tool to an activity based on the Decorator pattern. The example is loosely based on the scenario used by Freeman et. al. [4] to introduce this pattern. A coffee shop sells many different kinds of beverage, with many combinations of attributes and extras. For example any kind of coffee can be ordered as regular or large; and with extras such as milk, mocha, whipped milk, and so on. This scenario could be modeled as a straightforward inheritance tree, but the number of subclasses required to account for all the possible products which could be ordered (e.g. *LargeDarkRoastWithMilkAndMocha*, *EspressoWithWhip*) could be very large. However, by using combinations of a few different kinds of decorators, you can create many different combinations of behaviour.

The aim of the problem solving activity is to produce a set of classes which can model all the possible beverages which can be created with as few classes as possible. The solution should allow new extras to be added to the menu easily. Students should select a suitable pattern and produce a working implementation which must allow instantiation of any beverage, which should then be able to calculate its cost. Each beverage should have a *description* attribute. The information given is the list of beverage types and extras and their individual costs. Completion of this activity would include the actions described in the following sections.

4.1 Select a pattern

The patternCoder extension adds a group of classes based on a pattern to an existing project. In this case, all the classes required are part of the pattern, so the wizard is invoked on a new empty project. The first step of the wizard (figure 1) gives a brief summary of each of the installed patterns, assuming that a suitable summary has been included in the pattern description file, which may provide sufficient information to prompt a suitable selection. Decorator is one of the patterns included in the basic distribution of the extension.

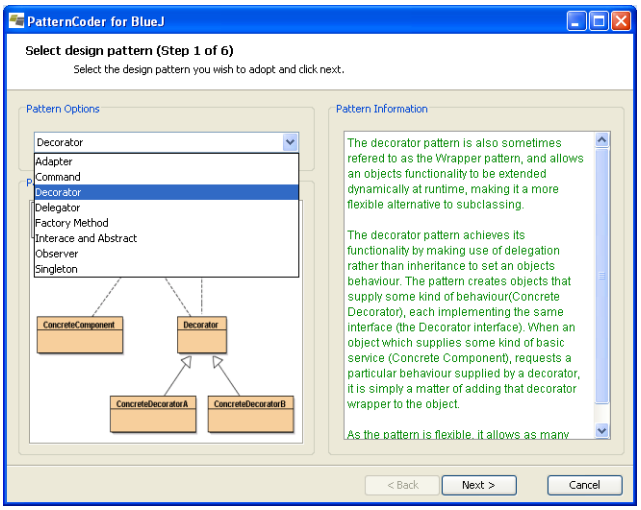

Figure 1. Selecting the appropriate pattern

4.2 Name the components

This action requires the students to determine the role played by each component of the pattern in the scenario. The pattern diagram is available at each step (figure 2) along with prompts which describe the role of each component. For example, the student should decide that the *Component* interface which represents a decoratable object should correspond to the concept of *Beverage* in the coffee shop scenario.

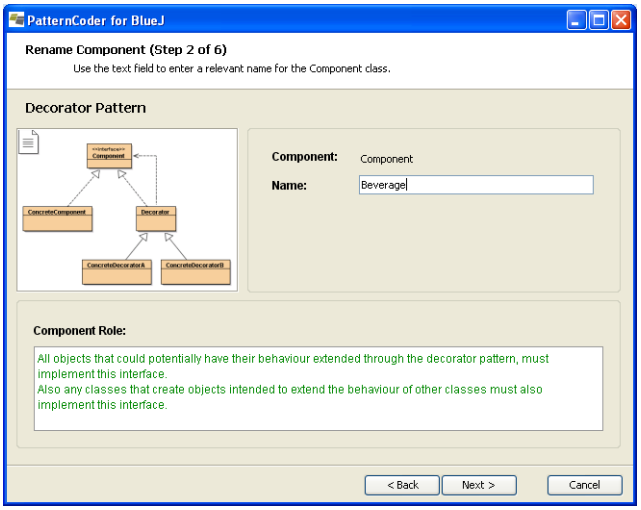

Figure 2. Naming the components

Clearly the pattern diagram which is shown in figures 1 and 2 contains a limited number of classes – only one concrete component and two decorators. The actual number of classes required will vary between scenarios. The current version of the wizard does not allow the number of classes to be varied. Rather, it allows a basic set of classes to be generated, which can then be used as models for extending the model to implement the scenario fully. In this case, a single beverage type (e.g. DarkRoast) and two extras (Milk and Mocha) are created initially – these can then be used as sample code when creating additional classes.

4.3 Create the basic classes

Completion of the wizard causes a basic set of classes to be generated and added to the project. This supports the student by ensuring that if the previous step has been completed correctly (requiring that the student has understood the pattern and its components correctly) an outline set of classes is created with the correct relationships and a basic set of methods which operate correctly 'out of the box'. Creation of the basic classes might otherwise be a time consuming, error-prone step in which the flow of understanding within the activity is interrupted. The student will still have to understand the structure of the code in order to modify it to model the

scenario correctly. The need to understand the detail of the code has been shifted to a later point in the activity, allowing creation of 'live' objects at an earlier stage. The same effect could be created if the instructor simply provides the basic code, but if this is done the students do not gain the sense of having created the code as a result of their own decisions and understanding.

After completion of the wizard and compilation, the project should look like figure 3. Note that the layout of the BlueJ class diagram will usually need to be rearranged manually.

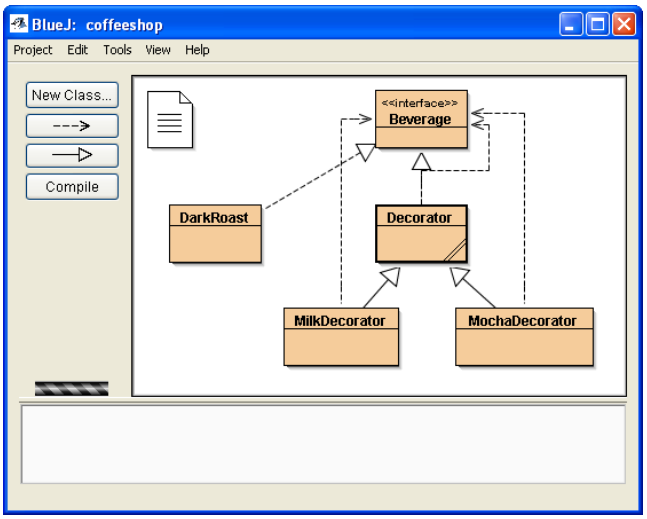

Figure 3. Initial classes

4.4 Initial inspection and testing

It should now be possible to create some objects using the classes created using the wizard. This allows the students immediately to explore the interactions between the classes, which will aid understanding of the pattern in detail and help to determine what modifications are required to fit the scenario. A suitable initial test would be to create a *DarkRoast* object and inspect it to see what attributes and methods it has. Initially it has no attributes, and it has an operation (figure 4) which prints the following:

```
This is basic operation of DarkRoast
```

Requesting creation of a *MilkDecorator* object results in a prompt for a *Beverage* object (figure 5), emphasizing that a decorator can only exist as a wrapper for another object. The *DarkRoast* object already created can be used.

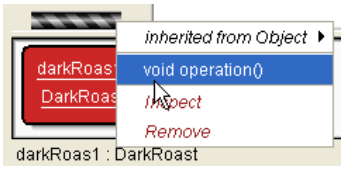

Figure 4. Operations of DarkRoast object

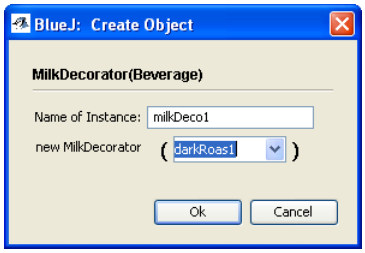

Figure 5. Creating a decorator object

Inspecting the new object shows that it has an attribute which is the *Beverage* it decorates (figure 6).

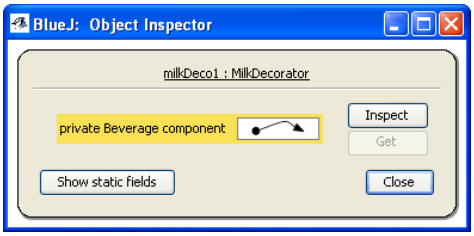

Figure 6.. Attributes of decorator object

The new object has two operations, representing the behavior of the *Beverage* (possibly overridden) and additional behavior provided by the decorator (figure 7).

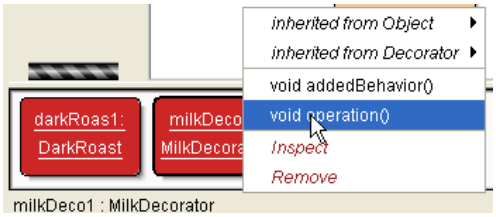

Figure 7. Operations of decorator object

The *operation* method initially outputs the following:

```
This is basic operation of DarkRoast
This is added behavior of MilkDecorator
```

Note that in the initial implementation, the *addedBehavior* method of the decorator is included in its *operation* method. The code is as follows:

```
public void operation()
{
	super.operation();
	addedBehavior();
}

protected void addedBehavior()
{
	// Do something new.
	System.out.println("This is added behavior of MilkDecorator");
}
```

4.5 Modify to solve the problem

The classes created using the wizard provide a basis for a solution to the problem, but do not provide a complete solution. Students must now modify and add classes as appropriate. The original problem required that a *description* attribute must be included. The students must determine how this should be provided, and how its value should be set correctly for every possible type of product. This is a straightforward but not entirely trivial task.

Additionally, each product must calculate its own cost. Students could use the *operation* method here as a model for a *cost* method. For example, the *cost* method in *MilkDecorator* could be as follows, if milk adds 20 cents to the total cost:

```
public double cost(){
	return component.cost() + 0.20;
}
```

Again the task is not particularly complex, but requires some thought, some understanding of the pattern and of basic issues such as inheritance, interfaces and access levels.

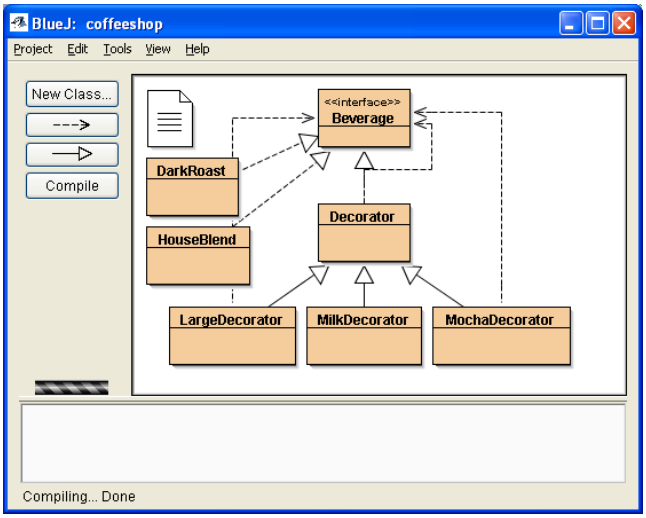

Figure 8. Additional classes added to the project

Finally, the complete scenario includes more options than the ones created initially, so additional classes must be added manually to the project. The *DarkRoast* class can be used as a model for other basic beverage types, such as *HouseBlend* or *Espresso*, while the decorator classes can be used as models for further decorators, such as *LargeDecorator* or *WhipDecorator*, as shown in figure 8.

.