The Syntax and the Semantics of FUDA Templates

1 Definition

A template is a representation of the *implementation steps* that are necessary to instantiate a given concept. A part of a template for the concept drawing a figure in a GEF editor on top of the Eclipse GEF framework is shown in Figure 1. A FUDA template has the form of a textbook-like example in Java-based pseudo-code.

2 The Templates' Content

Templates specify the following implementation steps:

- Packages to import (l. 1–4 in Figure 1);
- Framework classes to subclass (l. 5, l. 22);
- Interfaces to implement (l. 15);
- Methods to implement (l. 16);
- Objects to create (l. 7, 17, 26, 29, 31); and
- Methods to call (l. 9, 11, 12, 19, 28, 30, 32).

In addition to the basic implementation steps, the template also reflects:

- Call nesting, e.g., `setModel()` is called directly or indirectly by `createEditPart()` (l. 19);
- Call order, e.g., `paletteRoot` (l. 29) is first created before calling its `setDefaultEntry()` method (l. 30);
- Parameter passing patterns, e.g., the `defaultEditDomain` object passed to the `setEditDomain()` method (l. 32) is obtained by a prior call to **`new`** `DefaultEditDomain()` (l. 31).

```
1                   import org.eclipse.gef.editpolicies.GraphicalNodeEditPolicy;
2                   import org.eclipse.gef.EditPart;
3                   import org.eclipse.gef.EditPartViewer;
4                   import org.eclipse.ui.part.WorkbenchPart;
5  /* FRL_01 */ public class AppComponentEditPolicy extends ComponentEditPolicy {
6  /* FRL_02 */     public Command createDeleteCommand(GroupRequest) {
7  /* FRL_03 */         AppGraphicalNodeEditPolicy appGraphicalNodeEditPolicy =
8                           new AppGraphicalNodeEditPolicy();
9  /* FRL_04 */         EditPart editPart = appComponentEditPolicy.getHost(); // REPEATED!
10                          /* UNKNOWN ORDER FOR THE FOLLOWING INSTRUCTIONS */
11 /* FRL_05 */         Object object1 = editPart.getModel(); // MAY REPEAT!
12 /* FRL_06 */         EditPart editPart = editPart.getParent();
13                      }
14                  }
15 /* FRL_07 */ public class AppditPartFactory implements EditPartFactory {
16 /* FRL_08 */     public EditPart createEditPart(editPart)||(appAbstractGraphicalEditPart) {
17 /* FRL_09 */         AppAbstractGraphicalEditPart appAbstractGraphicalEditPart =
18                          new AppAbstractGraphicalEditPart();
19 /* FRL_10 */         appAbstractGraphicalEditPart.setModel(object1)||(editPart); // REPEATED!
20                      }
21                  }
22 /* FRL_11 */ public class AppGraphicalNodeEditPolicy extends GraphicalNodeEditPolicy {
23                  }
24                  public class SomeClass {
25                      public void someMethod() {
26 /* FRL_12 */         TemplateTransferDragSourceListener templateTransferDragSourceListener =
27                          new TemplateTransferDragSourceListener(EditPartViewer)||(EditPartViewer,Transfer);
28 /* FRL_13 */         EditPartViewer.addDragSourceListener(templateTransferDragSourceListener);
29 /* FRL_14 */         PaletteRoot paletteRoot = new PaletteRoot();
30 /* FRL_15 */         paletteRoot.setDefaultEntry(ToolEntry);
31 /* FRL_16 */         DefaultEditDomain defaultEditDomain = new DefaultEditDomain(IEditorPart);
32 /* FRL_17 */         GraphicalEditor.setEditDomain(defaultEditDomain);
33                      }
34                  }
```

Figure 1: Part of a template generated by FUDA for the concept drawing figures in a GEF editor

Note that the specified steps involve only the elements of the framework API and implementation steps that are specific to a particular sample application are not reflected in the template.

It is also worth mentioning that the classes of the template only include the methods that are actually executed when the desired concept is invoked at runtime. For example, the body of the class `AppGraphicalNodeEditPolicy` is empty and it shows that none of its methods is called when the figure drawing concept is executed.

2.1 The Meaning of Comments

- The comments `REPEATED` (e.g., l. 9) and `MAY REPEAT` (l. 11) indicate that the commented step appeared more than once in every or some of the traces used to generate the template, respectively.

- The comments in the form of `/* FRL_n */` are used to provide traceability between the templates and the sample applications' source code from which the templates are generated. In these comments, *n* represents the number of the implementation step in the template. More specifically, we have manually com-

mented all the lines of the sample applications' source code with `/* FRL_n */` that correspond to the n^{th} implementation step in the template. Therefore, you can search the sample applications' source code with `FRL_n` to see how that template's implementation step is actually implemented in the provided sample applications.

- Some templates have a comment in the form of `/* UNKNOWN ORDER FOR THE FOLLOWING INSTRUCTIONS */` (e.g., l. 10). This comment shows that FUDA was not able to automatically identify an exact order for the instructions that follow this comment.

3 Differences from Ordinary Java

There are two main differences between templates and ordinary Java programs:

1. *Using the notion of* '||': FUDA uses a special syntax to show that a method with a given name was called with different argument types. For example, `setModel(object1)||(editPart)` (l. 19 in Figure 1) is due to multiple calls to `setModel()` with different arguments. As another example, l. 27 illustrates that the class `TemplateTransferDragSourceListener` can be instantiated with different types of arguments.

2. *All variables are global:* What appears to be a local variable declaration in Java, such as `object1` (l. 11), actually has global meaning in the template. For that reason, `object1` can be used as a method argument in another method scope (l. 19).

4 The class `SomeClass` and the method `someMethod` in the Template

The method `someMethod` of class `SomeClass` hosts instructions that FUDA was unable to automatically identify an exact place for them. **Please note that the instructions in this method may or may not come together in the concept's implementation.** You may specify the place of each instruction yourself.

5 Possibility of False Positives and False Negatives

A template is an approximation of the necessary implementation steps, and it can be incomplete or unsound or both. In particular, implementation steps can be missing (*false negatives*) or unrelated steps (*false positives*) can be present in some cases. Given two sample applications, FUDA will filter out any steps that are not common to both sample applications. However, based on the experiments we did, false positives are more likely to happen than false negatives.

Furthermore, some implementation detail is still missing in a template. For example, the presented template in Figure 1 only presents the implementation steps on top of the Eclipse GEF framework. However, there might be some instructions on top of other frameworks that are necessary to see the correct functionality of the drawing figure concept. Since they are on top of other frameworks (e.g., draw2d framework), they are not included in the template. As another example, whereas the calls in lines l. 9 and l. 11 are marked as candidates to be repeated, the template does not reflect the fact that they should be repeated as a block, rather than individually.