17 Tool Support Beyond Preprocessors and Feature Modules

In feature-oriented software product lines, products can be generated completely automatic for each valid configuration. In Parts III and IV, we already focused on two implementation techniques, namely, conditional compilation and feature-oriented programming, which enable such automatic generation. In this chapter, we give an overview on further implementation techniques that are supported within FeatureIDE.

Implementation techniques for feature-oriented product lines are typically distinguished into composition-based and annotation-based techniques (Apel et al. 2013a). Annotation-based techniques, such as conditional compilation, use annotations to remove certain artifacts or parts thereof. In contrast, with composition-based techniques, such as feature-oriented programming, a selection of partial artifacts is composed into the artifacts of a product. In this sense, conditional compilation and feature-oriented programming are representative techniques used in previous parts. Nevertheless, there is no consensus in research what is the best technique and, thus, other techniques have been integrated into *FeatureIDE* additionally.

In this chapter, we focus on runtime variability resolving variability decisions at runtime (cf. Sect. 17.1), black-box frameworks enabling compile-time variability in terms of plug-ins (cf. Sect. 17.2), and aspect-oriented programming weaving crosscutting concerns into a base application (cf. Sect. 17.3). Plug-ins and aspects are considered composition-based techniques (Apel et al. 2013a), whereas runtime variability is closer to conditional compilation. In the following, we illustrate how those techniques are supported within FeatureIDE.

DOI 10.1007/978-3-319-61443-4_17

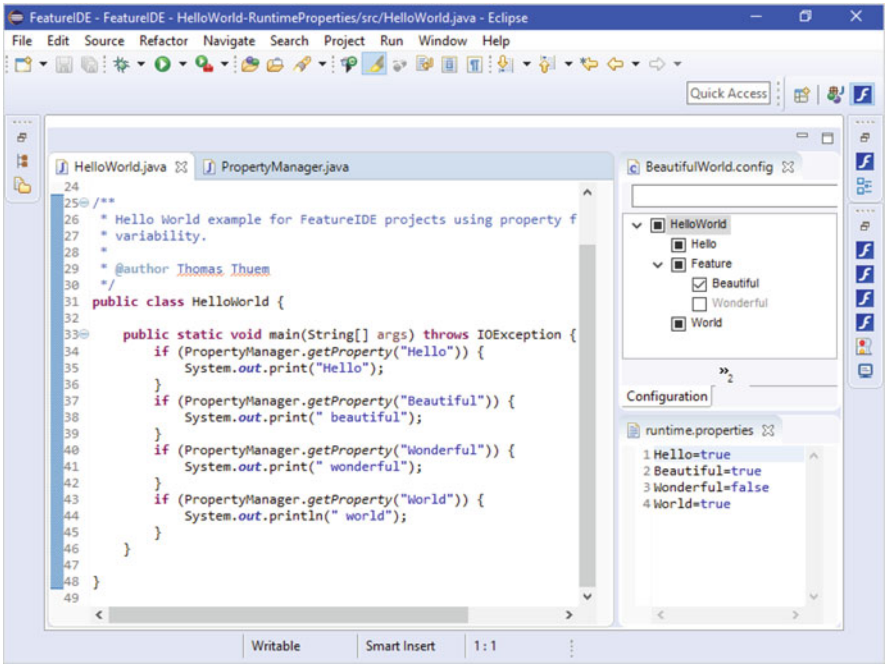
17.1 Product-Line Implementation with Runtime Variability

With runtime variability, we refer to the ability of software to be customized with branching in the language of choice (e.g., Java or C). That is, parts of the source code are activated or deactivated based on a given configuration. Which source code is activated or deactivated is resolved at runtime. Hence, it is called *runtime variability*.

Instruction 17.1 (Open Example with Property Files)

1. Open the example wizard by any of the following options:
   - Press *New* → *Example* in the context menu of the Project/Package Explorer
   - Press *New* → *Examples* → *FeatureIDE* → *FeatureIDE Examples* in the upper-left menu toolbar of *Eclipse*
   - Press *File* → *New* → *Example* in the menu bar of *Eclipse*
2. Select *FeatureIDE Examples*
3. Press *Next*
4. Select *Book Example* tab
5. Select the project *Part V* → *Chapter 17* → *HelloWorld-RuntimeProperties*
6. Press *Finish*
7. Open the class `HelloWorld` to see how features are implemented
8. Open the class `PropertyManager` generated by *FeatureIDE* to inspect how feature values are accessed from Java source code
9. Open the file `runtime.properties` to see how feature values are stored
10. Open, change, and save the current configuration to understand when and how the file `runtime.properties` is updated
11. Run `HelloWorld`

Before a program with runtime variability can be started, we need to decide how to make the configuration accessible in the source code. That is, a mechanism is required such that the selection value of each feature can be evaluated in branching statements. FeatureIDE currently supports two strategies to propagate configuration decisions to runtime variability, namely, property files and global variables. With property files, whenever the configuration is updated, it is also stored in a property file. This property file is then loaded on the start-up of the program. With global variables, FeatureIDE creates a class with boolean constants representing the features of the product line. Whenever the configuration changes, those constants are assigned to true or false. During runtime, those constants can be evaluated by the program.


FeatureIDE - FeatureIDE - HelloWorld-RuntimeProperties/src/HelloWorld.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
HelloWorld.java
PropertyManager.java
BeautifulWorld.config
Configuration
runtime.properties
Writable
Smart Insert
1 : 1

```

/**
 * Hello World example for FeatureIDE projects using property f
 * variability.
 *
 * @author Thomas Thuem
 */
public class HelloWorld {

    public static void main(String[] args) throws IOException {
        if (PropertyManager.getProperty("Hello")) {
            System.out.print("Hello");
        }
        if (PropertyManager.getProperty("Beautiful")) {
            System.out.print(" beautiful");
        }
        if (PropertyManager.getProperty("Wonderful")) {
            System.out.print(" wonderful");
        }
        if (PropertyManager.getProperty("World")) {
            System.out.println(" world");
        }
    }

}

```

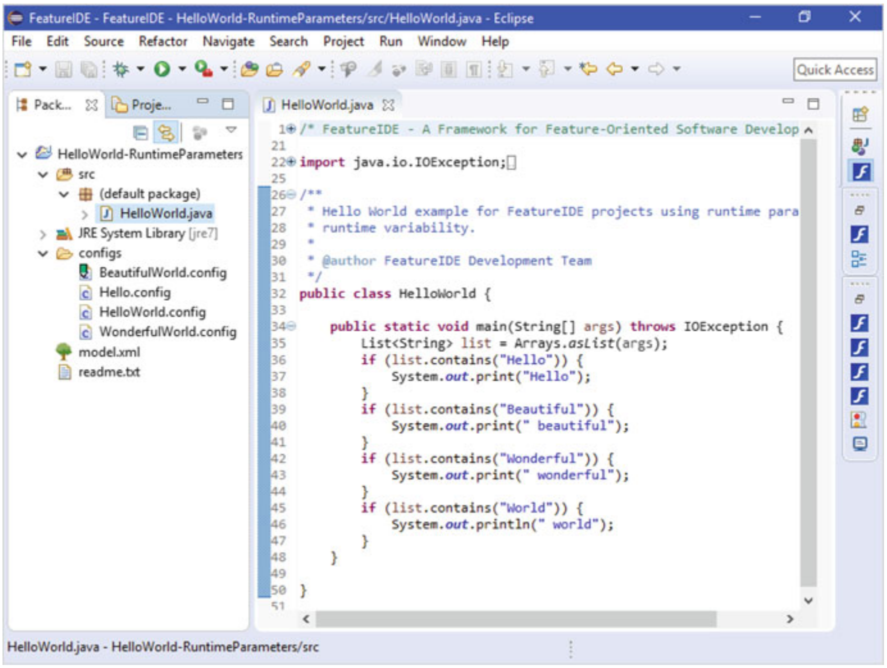
```
HelloWorld
    Hello
    Feature
        [✓] Beautiful
        [ ] Wonderful
    World
```

```
Hello=true
Beautiful=true
Wonderful=false
World=true
```

Fig. 17.1 `HelloWorld` example with runtime variability in terms of property files

In this chapter, we illustrate all implementation techniques using a smaller example than the elevator for brevity. In particular, we will use examples similar to the `HelloWorld` product line already used in Chap. 3 on Page 19. In those tiny examples, we can focus on the main differences of each technique rather than getting lost in details.

Follow Instruction 17.1 to understand how property files are used in *FeatureIDE* to implement product lines with runtime variability. In Fig. 17.1, we show some relevant parts of the example. The current configuration is not only stored in a config file but also in a property file called `runtime.properties`. The property file is updated once the current configuration is changed and saved. The product-line code refers to features by means of the automatically generated class `PropertyManager`, which reads feature values from the property file and gives an easy-to-use interface to access feature selection states.


FeatureIDE - FeatureIDE - HelloWorld-RuntimeParameters/src/HelloWorld.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Pack... Proje...
HelloWorld-RuntimeParameters
src
(default package)
HelloWorld.java
JRE System Library [jre7]
configs
BeautifulWorld.config
Hello.config
HelloWorld.config
WonderfulWorld.config
model.xml
readme.txt
HelloWorld.java
HelloWorld.java - HelloWorld-RuntimeParameters/src

```
/* FeatureIDE - A Framework for Feature-Oriented Software Develop

import java.io.IOException;

/**
 * Hello World example for FeatureIDE projects using runtime para
 * runtime variability.
 *
 * @author FeatureIDE Development Team
 */
public class HelloWorld {

    public static void main(String[] args) throws IOException {
        List<String> list = Arrays.asList(args);
        if (list.contains("Hello")) {
            System.out.print("Hello");
        }
        if (list.contains("Beautiful")) {
            System.out.print(" beautiful");
        }
        if (list.contains("Wonderful")) {
            System.out.print(" wonderful");
        }
        if (list.contains("World")) {
            System.out.println(" world");
        }
    }

}
```

Fig. 17.2 `HelloWorld` example with runtime variability in terms of parameters

Instruction 17.2 (Open Example with Runtime Parameters)

1. Open the example wizard by any of the following options:
   - Press *New* → *Example* in the context menu of the Project/Package Explorer
   - Press *New* → *Examples* → *FeatureIDE* → *FeatureIDE Examples* in the upper-left menu toolbar of *Eclipse*
   - Press *File* → *New* → *Example* in the menu bar of *Eclipse*
2. Select *FeatureIDE Examples*
3. Press *Next*
4. Select *Book Example* tab
5. Select the project *Part V* → *Chapter 17* → *HelloWorld-RuntimeParameters*
6. Press *Finish*
7. Open the readme.txt and follow instructions to set up the project
8. Open the class `HelloWorld` to see how features are implemented
9. Run `HelloWorld`
10. Run different configurations of the example by switching to another configuration

Similarly, runtime variability can be implemented with command-line parameters (for short runtime parameters) instead of property files. Follow Instruction 17.2 to understand how runtime variability with parameters is supported in *FeatureIDE*. In Fig. 17.2, we show how the parameters of the main method can be used to implement runtime variability. In the example, we simply check whether certain feature names have been passed as a parameter to the main method or not. *FeatureIDE* makes sure that the parameters passed to the main method are those features being selected in the current configuration.

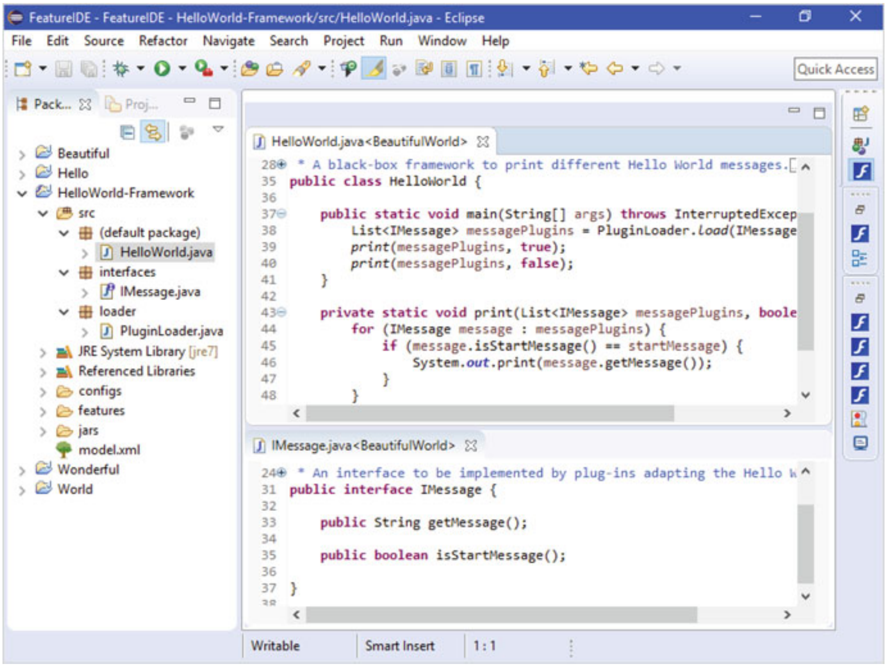
Runtime variability is not to confuse with dynamic software product lines (Hallsteinsen et al. 2008), in which the configuration can be switched even during runtime. Here, we only illustrate what is also known as load-time variability: a configuration is already fixed before loading the program and then never changed during execution. Dynamic product lines additionally need a mechanism to translate the current program state of one configuration safely into another configuration, which is so far not addressed by FeatureIDE.

17.2 Product-Line Implementation with Black-Box Frameworks

A further common way to implement variability is *black-box frameworks*. Similar to feature-oriented programming, the implementation is modularized according to features. In contrast to feature-oriented programming, in which all classes and methods can be refined, all future extensions have to be anticipated and enabled by means of extension points [cf. preplanning problem (Apel et al. 2013a)]. An extension point provides one or several interfaces, which can be implemented by extensions. Besides the framework itself, all source code is modularized into plug-ins. Each plug-in implements a number of extensions and may itself provide extension points to other plug-ins.

Eclipse itself is considered a black-box framework implemented based on OSGi (Gruber et al. 2005). Each plug-in is developed in a special kind of Java project. All extensions and extension points of a plug-in are specified in XML, which enables *Eclipse* to load plug-ins only once their source code is accessed by the framework or other plug-ins. However, *Eclipse* does neither provide a mechanism to specify valid configurations with a feature model nor does it allow to check whether certain configurations are valid. In particular, we cannot easily switch between numerous configurations within *Eclipse*. Rather, we have to manually install and uninstall features. In the *Eclipse* world, a feature is basically a set of plug-ins and provides a unit being used for installation.

FeatureIDE aims to close this gap by combining black-box frameworks with feature modeling. Feature models and configurations are created and maintained as discussed before. During the creation of a *FeatureIDE* project, we have to select *Framework* as composer. The created *FeatureIDE* project is supposed to contain all artifacts (i.e., source code) of the framework itself. For each feature, developers have to create a separate Java project, which is then being compiled as a jar file and only loaded during the start-up of the framework, if the according feature is



```
 * A black-box framework to print different Hello World messages.[]
public class HelloWorld {

    public static void main(String[] args) throws InterruptedExcep
        List<IMessage> messagePlugins = PluginLoader.load(IMessage
        print(messagePlugins, true);
        print(messagePlugins, false);
    }

    private static void print(List<IMessage> messagePlugins, boole
        for (IMessage message : messagePlugins) {
            if (message.isStartMessage() == startMessage) {
                System.out.print(message.getMessage());
            }
        }
```

```
 * An interface to be implemented by plug-ins adapting the Hello W
public interface IMessage {

    public String getMessage();

    public boolean isStartMessage();

}
```

Fig. 17.3 `HelloWorld` example implemented as black-box framework

selected in the current configuration. *FeatureIDE* provides a simple plug-in loader implementation, which is automatically added to the project on creation.

Follow Instruction 17.3 to understand how *FeatureIDE* supports black-box frameworks with plug-ins. In Fig. 17.3, we show the source code of the class `HelloWorld` being the framework and its interface `IMessage`. The latter is a standard Java interface, which can be implemented by plug-ins to add part of the HelloWorld message. The class `HelloWorld` retrieves a list of all plug-ins from the plug-in loader and iterates over them to print a message by each of them.

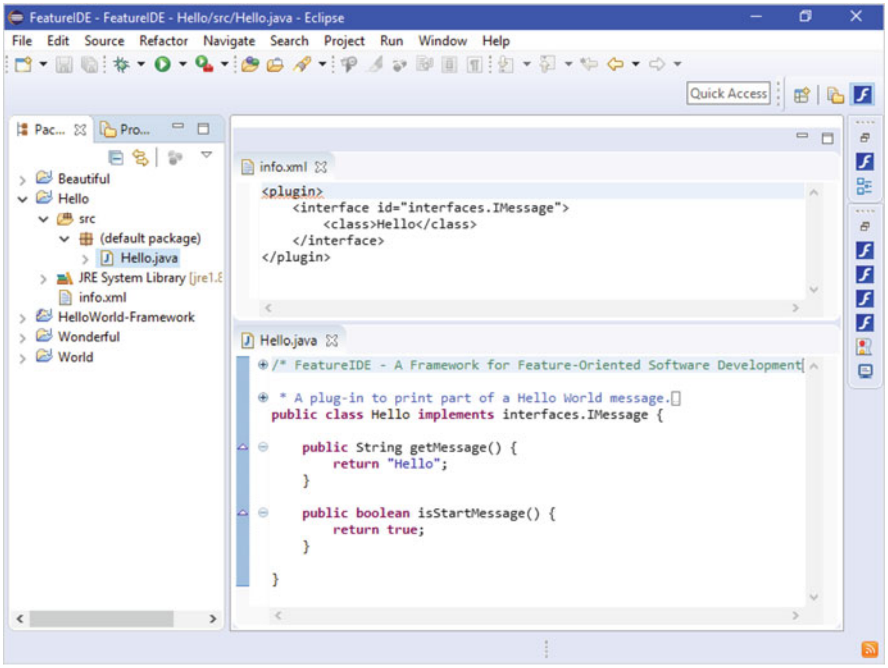
In contrast to feature-oriented programming, the order in which plug-ins are loaded is typically not specified using a total order and rather depends on the particular plug-in loader. In contrast to conditional compilation and runtime variability, the order of features matters for frameworks. *FeatureIDE*'s plug-in loader loads plug-ins in alphabetical order. Hence, the `HelloWorld` example is slightly more complex due to the fact that the order of plug-ins loaded simply depends on their name. To print the word `Hello` before the word `beautiful` (against alphabetical order), we introduced the method `isStartMessage` to print the word `Hello` before all other plug-ins. A more general solution could be to assign plug-ins priorities (i.e., integer values) and load or process them according to their priorities. In our example, two different priorities were simply enough (i.e., start message or not).

Instruction 17.3 (Open Example with Black-Box Frameworks)

1. Open the example wizard by any of the following options:
   - Press *New* → *Example* in the context menu of the Project/Package Explorer
   - Press *New* → *Examples* → *FeatureIDE* → *FeatureIDE Examples* in the upper-left menu toolbar of *Eclipse*
   - Press *File* → *New* → *Example* in the menu bar of *Eclipse*
2. Select *FeatureIDE Examples*
3. Press *Next*
4. Select *Book Example* tab
5. Select the project *Part V* → *Chapter 17* → *HelloWorld-Framework*
6. Press *Finish*
7. Open the classes `HelloWorld` and `IMessage` to see how the framework itself is implemented
8. Open the class `Hello` and the file `info.xml` in the project Hello to see how plug-ins are implemented
9. Run `HelloWorld`
10. Inspect how *FeatureIDE* sets the classpath based on the current configuration by:
    - Press *Run As...* → *Run Configurations...* → *Java Application* → *HelloWorld* → *Classpath* in the upper menu toolbar of *Eclipse*
    - Press *Run As* → *Run Configurations...* → *Java Application* → *HelloWorld* → *Classpath* in the context menu of the Project/Package Explorer
    - Press *Run (Ctrl + F11)* → *Run Configurations...* → *Java Application* → *HelloWorld* → *Classpath* in the menu bar of *Eclipse*
11. Open the class `PluginLoader` generated by *FeatureIDE* to inspect how plug-ins are loaded

In Fig. 17.4, we show all artifacts necessary to implement the plug-in `Hello`. The Java code consists of a single class implementing the interface `IMessage` in a straightforward manner. However, *FeatureIDE*'s plug-in loader relies on the fact that plug-ins make their extensions to given interfaces explicit in a simple XML file. The file `info.xml` contains the knowledge needed to know when loading all classes for a particular interface. In our example, the plug-in extends only the interface with the full qualified name `interfaces.IMessage` by means of the class named `Hello`. Given that the according feature is selected, the plug-in loader will load the class `Hello` if the framework asks for classes implementing the interface `IMessage` (cf. first line of the main method in Fig. 17.3).

In the context of software product lines, it is not intended to always load all plug-ins. To only load plug-ins for features selected in the current config-


FeatureIDE - FeatureIDE - Hello/src/Hello.java - Eclipse
File Edit Source Refactor Navigate Search Project Run Window Help
Quick Access
Pac... Pro...
Beautiful
Hello
src
(default package)
Hello.java
JRE System Library [jre1.8
info.xml
HelloWorld-Framework
Wonderful
World
info.xml
Hello.java

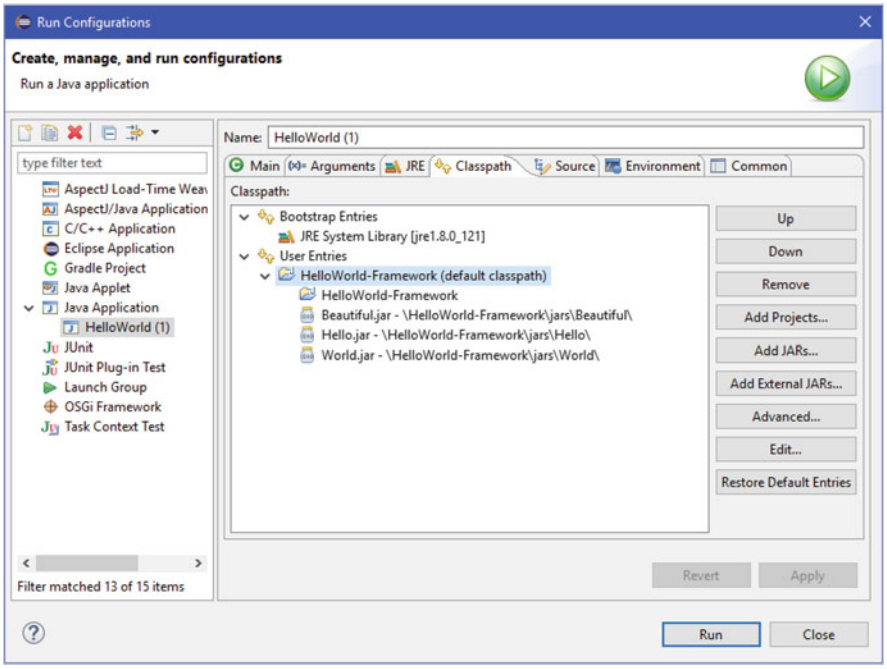
```
<plugin>
    <interface id="interfaces.IMessage">
        <class>Hello</class>
    </interface>
</plugin>
```

```
/* FeatureIDE - A Framework for Feature-Oriented Software Development
 * A plug-in to print part of a Hello World message.
public class Hello implements interfaces.IMessage {

    public String getMessage() {
        return "Hello";
    }

    public boolean isStartMessage() {
        return true;
    }
}
```

Fig. 17.4 Plug-in `Hello` of the `HelloWorld` framework

uration, *FeatureIDE* modifies the classpath of the Java project representing the framework. In Fig. 17.5, we show how the classpath looks if all features except the feature `Wonderful` are selected. *FeatureIDE* creates a jar for each plug-in and adds only those jars to the classpath, for which the according feature is selected in the current configuration. On each change of a configuration, the classpath is updated. Similarly, on each change of a plug-in, the respective jar is updated.

Readers might wonder why this implementation technique is called black-box framework. In black-box frameworks, the framework itself and each plug-in can compiled separately (Apel et al. 2013a). In our example, each plug-in is even a separate Java project that can be compiled without having the source code of any other part. Plug-ins, however, need at least the class files of the framework for compilation. Hence, the source code of plug-ins does not need to be available for the framework and vice versa. In contrast, in white-box frameworks, any extension needs access to the source code of the framework and cannot be compiled separately (Apel et al. 2013a).


Run Configurations

Create, manage, and run configurations

Run a Java application

type filter text

- AspectJ Load-Time Weav
- AspectJ/Java Application
- C/C++ Application
- Eclipse Application
- Gradle Project
- Java Applet
- Java Application
  - HelloWorld (1)
- JUnit
- JUnit Plug-in Test
- Launch Group
- OSGi Framework
- Task Context Test

Filter matched 13 of 15 items

Name: HelloWorld (1)

Classpath:

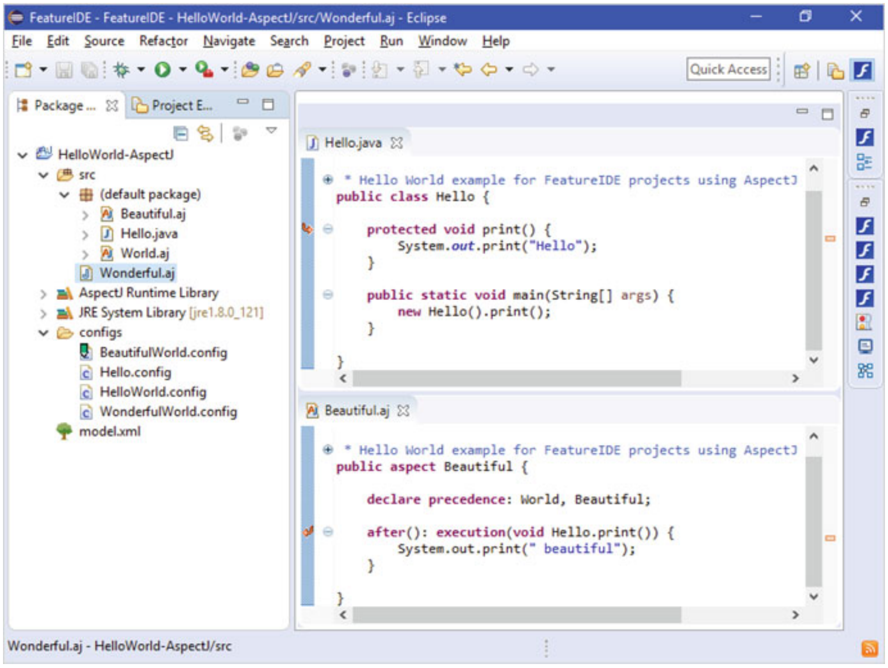
```
Bootstrap Entries
    JRE System Library [jre1.8.0_121]
User Entries
    HelloWorld-Framework (default classpath)
        HelloWorld-Framework
        Beautiful.jar - \HelloWorld-Framework\jars\Beautiful\
        Hello.jar - \HelloWorld-Framework\jars\Hello\
        World.jar - \HelloWorld-Framework\jars\World\
```

Fig. 17.5 *FeatureIDE* automatically includes and excludes plug-in jars into the framework's classpath depending on the current configuration

17.3 Product-Line Implementation with Aspect-Oriented Programming

Aspect-oriented programming is a further technique, which can be employed to implement product lines (Apel et al. 2013a). However, the initial intention was not to be used for product lines, but rather to decompose an application into its concerns or to change an existing application without modifying its source code (Kiczales et al. 1997). In particular, we have a base application implemented in an object-oriented language (e.g., Java), and then aspects are implemented in an aspect-oriented extension of that language (e.g., AspectJ).

In contrast to conditional compilation and runtime variability, the goal is to modularize source code with respect to features. In this respect, aspect-oriented programming is similar to feature-oriented programming and black-box frameworks. The main difference to frameworks is that extensions do not have to be anticipated in advance [cf. preplanning problem (Apel et al. 2013a)]. In contrast to feature-oriented programming, aspect-oriented languages typically have a complex language to make any possible change to an existing application without changing its source code.



```
 * Hello World example for FeatureIDE projects using AspectJ
public class Hello {

    protected void print() {
        System.out.print("Hello");
    }

    public static void main(String[] args) {
        new Hello().print();
    }

}
```

```
 * Hello World example for FeatureIDE projects using AspectJ
public aspect Beautiful {

    declare precedence: World, Beautiful;

    after(): execution(void Hello.print()) {
        System.out.print(" beautiful");
    }

}
```

Fig. 17.6 `HelloWorld` example implemented using aspect-oriented programming with AspectJ

Follow Instruction 17.4 to understand how *FeatureIDE* extends the AspectJ Development Tools with support for product lines. In Fig. 17.6, we show the class `Hello` and an aspect called `Beautiful`. In this example, the base application merely consists of the class `Hello`, which essentially prints `Hello` to the command line. Aspects are defined similarly to classes but are indicated by the keyword `aspect` instead of `class`. The aspect `Beautiful` alters the base application in the following way; after every execution of the method `print` in the class `Hello`, the word `beautiful` is printed to command line. The according three lines include a piece of advice (i.e., the code to execute additionally) and a pointcut (i.e., a predicate identifying when to execute this additional code). In essence, advice is similar to method refinement in feature-oriented programming, but AspectJ allows more sophisticated ways to change the control flow of a given Java Application. The aspects `World` and `Wonderful`, however, do not need more language constructs than aspect `Beautiful`.

A further language construct that is required for our HelloWorld example is related to the ordering of aspects. Similar to feature modules and plug-ins, the order in which aspects are composed can influence the result. In contrast to feature modules and plug-ins, AspectJ provides a dedicated language construct to define a partial order on aspects. The keyword `declare precedence` is followed by a list of aspects, whereas aspects are woven into the application from right to left.

In our example, it is sufficient to specify only the order of two pairs, namely, that `Wonderful` comes before `World` and that `Beautiful` comes before `World`. The order of `Beautiful` and `Wonderful` is simply irrelevant, as they can never be selected together due to the constraints in the feature model.

Due to the large number of language constructs of AspectJ, this section can only give a brief introduction and cannot provide a full overview of all language features of AspectJ. Similarly, the tool support provided by the AspectJ Development Tools is not provided in detail here. We refer interested readers to dedicated literature at the end of the chapter.

Instruction 17.4 (Open Example with Aspect-Oriented Programming)

1. Open the example wizard by any of the following options:
   - Press *New* → *Example* in the context menu of the Project/Package Explorer
   - Press *New* → *Examples* → *FeatureIDE* → *FeatureIDE Examples* in the upper-left menu toolbar of *Eclipse*
   - Press *File* → *New* → *Example* in the menu bar of *Eclipse*
2. Select *FeatureIDE Examples*
3. Press *Next*
4. Select *Book Example* tab
5. Select the project *Part V* → *Chapter 17* → *HelloWorld-AspectJ*
6. Press *Finish*
7. Open the class `Hello` to see the base application, which essentially prints `Hello`
8. Open the aspects `Beautiful`, `Wonderful`, and `World` to inspect how aspects modify the base application
9. Run `Hello`
10. Select another configuration as current configuration and see in package explorer how aspects are added and removed from the classpath
11. Remove or change a line in the aspect `Beautiful` or the aspect `Wonderful` starting with `declare precedence` and investigate the effect by running the example under different configurations

As a side remark, the AspectJ Development Tools seem not to be actively developed anymore. In particular, there was not any stable release until April 2017 for *Eclipse* 4.4 or newer versions. In other words, the latest stable version has been released in 2013. Hence, AspectJ can be used with *Eclipse* 4.4 or newer versions only using a development build of the AspectJ Development Tools, which, according to our experience, we cannot recommend for productive development. If there are no further stable releases, we sadly recommend to use *Eclipse* 4.3 or older versions.

17.4 Summary and Further Reading

In this chapter, we discussed further techniques to implement feature-oriented software product lines. In particular, we focused on runtime variability, frameworks, and aspect-oriented programming. While frameworks and aspect-oriented programming like most techniques in this book enable compile-time variability, with runtime variability, the source code of every product is essentially the same. Runtime variability uses branching depending on feature selections to implement differing behavior. *FeatureIDE* provides two mechanisms for runtime variability, namely, property files and command-line parameters. Besides conditional compilation and feature-oriented programming, *FeatureIDE* supports black-box frameworks with plug-ins and aspect-oriented programming with AspectJ for compile-time variability. For plug-ins and aspects, variability is achieved by modifying the classpath to add or remove certain plug-ins or aspects according to the current configuration. The main difference of both techniques is that frameworks require to make extension points explicit in terms of interfaces, whereas each aspect can modify arbitrary positions in the control flow by means of pointcuts.

Parts III and IV extensively focused on *FeatureIDE*'s tool support for conditional compilation and feature-oriented programming. This chapter was devoted to other implementation techniques for which tool support exists in a similar fashion, although not all functionalities are implemented for all implementation techniques yet. In Chap. 19 on Page 227, we give an overview on all implementation techniques and to which extent they are supported in *FeatureIDE*. Whereas this chapter focuses on some implementation techniques that have been neglected in previous parts, the next chapter is devoted to further development tasks that *FeatureIDE* supports, such as refactoring or specification.

A more extensive discussion of those implementation techniques and their advantages and disadvantages can be found elsewhere (Apel et al. 2013a). For aspect-oriented programming, we also refer to the original work proposing it (Kiczales et al. 1997) and a language description of AspectJ (Kiczales et al. 2001).