FeatureIDE in a Nutshell

3

FeatureIDE implements a general support to implement feature-oriented software product lines. In this chapter, we give a general overview on the functionalities of *FeatureIDE*. To get a first impression of *FeatureIDE*, we use a small "Hello World" application. As *FeatureIDE* supports all phases of the feature-oriented software development process, we introduce how all these phases are realized.

First, we explain how to configure *Eclipse* to display the *FeatureIDE* perspective in Sect. 3.1. In Sect. 3.2, we show how to load an example product line which we will use in this chapter for illustration purposes. Based on this example, we explain the general structure of *FeatureIDE* projects in Sect. 3.3. The support for domain engineering in the form of feature models is introduced in Sect. 3.4. In Sect. 3.5, we show how to implement variability using feature-oriented programming. After implementing software variability, the program can be configured by selecting the required features. How product configuration is supported is explained in Sect. 3.6. Finally, we show in Sect. 3.7 how the configured program can be generated and executed.

3.1 Opening the FeatureIDE Perspective

For an optimal functional support, *Eclipse* offers several specialized perspectives. A perspective predefines views, editors, and menu entries that provide dedicated support for specific tasks (e.g., debugging) or programming languages (e.g., Java).

FeatureIDE provides a perspective to support product-line development. The standard representation consists of several views, such as *FeatureIDE Outline*, *Collaboration Diagram*, *Feature Model Edits*, and *FeatureIDE Statistics*, which are discussed in detail later in the book. Furthermore, besides others, the perspective also consists of the editor and the package explorer that is also used by other plug-ins. To select the *FeatureIDE* perspective, follow Instruction 3.1. After the *FeatureIDE* perspective is selected, it appears in the top-right corner of *Eclipse* (Fig. 3.1).

J. Meinicke et al., *Mastering Software Variability with FeatureIDE*,
DOI 10.1007/978-3-319-61443-4_3

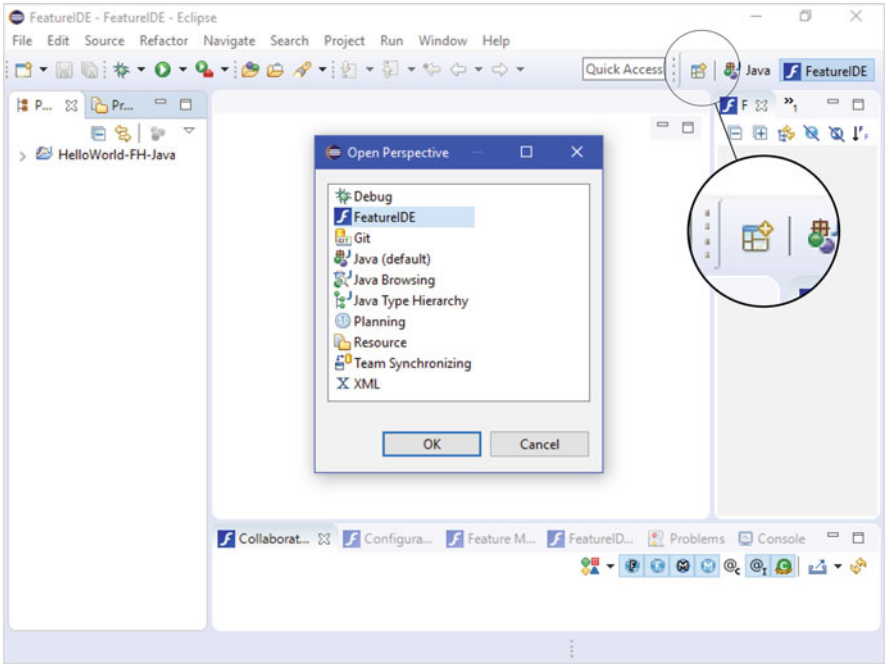

Fig. 3.1 Open the perspective selection dialog via the button (*right*). Select the *FeatureIDE* entry (*left*) *FeatureIDE*

Instruction 3.1 (Open the *FeatureIDE* Perspective)

1. Open the perspective selection dialog by any of the following options:
   - Press the button with the tooltip *Open Perspective* in the upper-right menu toolbar of *Eclipse*
   - Press *Window* → *Perspective* → *Open Perspective* → *Other...* in the menu bar of *Eclipse*
2. Select the entry for *FeatureIDE*.

3.2 Loading FeatureIDE Examples

To introduce to the *FeatureIDE*, we use a simple hello world product line. To ease the comprehension and to get an optimal overview, we use an existing example that can be installed via *FeatureIDE*'s example wizard. It is required that the plug-in *FeatureIDE example project* is installed.

In Fig. 3.2, we present an overview of the example wizard which provides all example projects of *FeatureIDE*. Using the pages (e.g., *Composer* or *Book*

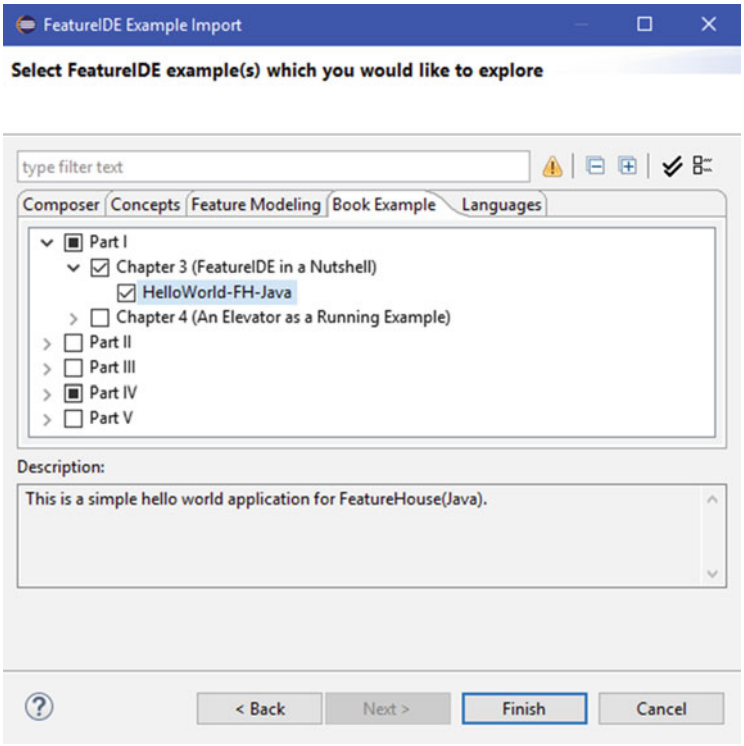

Fig. 3.2 Overview of existing projects in the *Example Wizard* of *FeatureIDE*

Example), we cluster the example into categories. In this book, we always refer to the examples in the page *Book Example*. In this chapter, we use a *HelloWorld* project created with the composition tool *FeatureHouse* (Apel et al. 2013b). It is possible to directly select the project in the project overview, or to filter the overview by the filter text box. If the user identified a specific project of interest, the project must be checked and confirmed using the *Finish* button.The *Example Wizard* always presents all *FeatureIDE* example projects regardless of the installed *FeatureIDE* plug-ins.

The projects of the Example Wizard can be in different states: *normal* (black), *warning* (yellow), and *error* (red). In detail, projects have an error state if they would result in an error after or during the installation step. It is not allowed for a developer to install projects with an error state (i.e., the *Finish* button is not selectable). For instance, if a project is already installed in the current workspace, it is not possible to install this project without an error. By contrast to projects with an error state, the warning state does not block an installation of the specific project. However, it is also likely that the project will not correctly work with the given *Eclipse* setting. The warning gives some suggestions to solve the project's problem. For instance, the warning state can be the result of a missing plug-in that has to be installed before we can use the project.

In the remainder of this chapter, we use the project *HelloWorld-FH-Java*. How to check out the project using the example wizard is explained in Instruction 3.2. Using this example, we will explain the main parts of *FeatureIDE* to understand its general process to develop product lines, namely, the project structure, feature modeling, implementation, configuration, product generation, and execution.

Instruction 3.2 (Import the HelloWorld-FH-Java Example)

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 I* → Chapter 3 → *HelloWorld-FH-Java*
6. Press *Finish*

3.3 Structure of FeatureIDE Projects

With *FeatureIDE*, we aim to support all phases of feature-oriented software development. Independent of the generation mechanisms, we need (a) a *feature model* to define features and their interdependencies, (b) *configurations* to define specific features representative of a certain product, (c) files that implement the feature's functionalities, and (d) generated source files that represent the product for the selected configuration. In the following, we give an overview on how *FeatureIDE* supports all these phases using the HelloWorld project.

After loading the HelloWorld project using the Example Wizard, we take a first look into the *FeatureIDE* project structure. In the following, we assume that the *FeatureIDE* perspective is active as some buttons may not be available in other perspectives. In this perspective, the loaded project is shown in the package explorer (cf. Fig. 3.3a). In the structure, we can see all main parts of a *FeatureIDE* project: (1) the `model.xml` for the domain modeling, (2) the directory `features` for the feature implementations, (3) the directory `configs` that consists of product configurations, and (4) the folder `src` that contains the generated product. The *current configuration*, marked with a green pencil (cf. the file `BeautifulWorld.config`), is used to generate the product using the feature implementation. Depending on the generation tool, the directory for feature implementations can be the same as for the generated product (e.g., for Antenna and AspectJ).

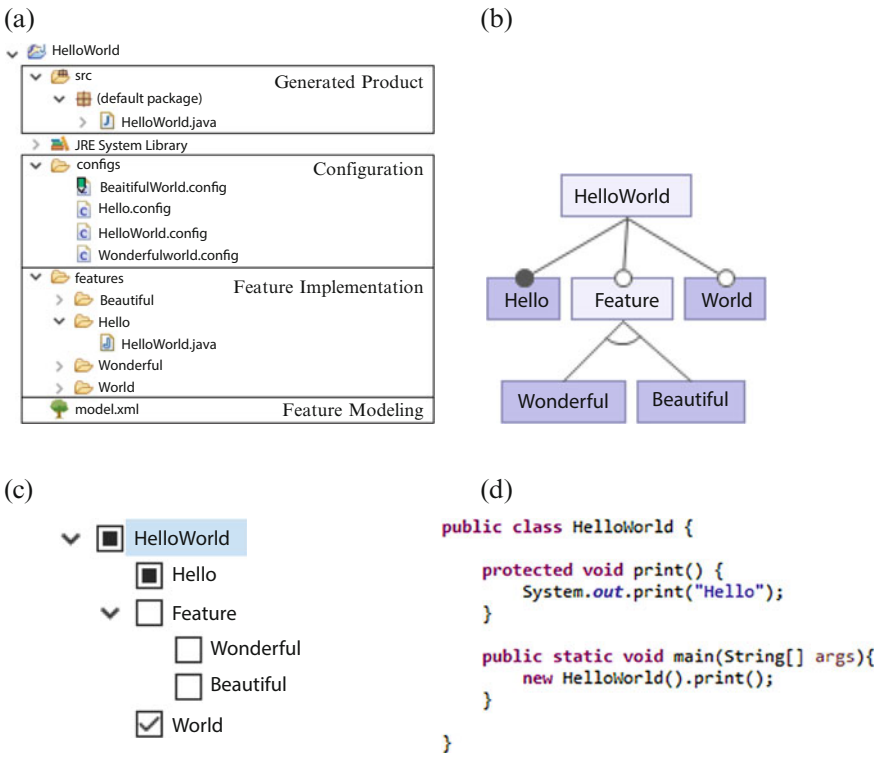

Fig. 3.3 Default project structure of a *FeatureIDE* project based on feature-oriented programming and Java, including feature modeling, configuration feature implementation, and generation of products. (**a**) General structure of FeatureIDE projects. (**b**) Feature modeling. (**c**) Configuration. (**d**) Feature implementation

The `model.xml` represents the domain model of a software product line using a feature model. Feature models are used to describe the common and variable parts of a software product line. *FeatureIDE* can use these information for other parts of the development cycle (e.g., domain implementation, product generation). By default, the `model.xml` is connected to the *Feature Model Editor* (cf. Fig. 3.3b). Using this editor, we can edit the feature model in a graphical manner. In Sect. 3.4, we present a brief overview of the facilities using the *Feature Model Editor*. In addition, we present a closer look in Chap. 5 on Page 43.

Let us consider the directory `configs` that is used for a product configuration. This directory contains a set of files with the extension *config*. Each of them is one specific product configuration in compliance to the feature model that is described in the `model.xml`. In addition, one of the existing files is marked with a pencil symbol (in Fig. 3.3 the file `BeautifulWorld.config`), which indicates that this product is selected for product generation and execution (see Sect. 3.7). *FeatureIDE* offers a specialized editor (i.e., the *Configuration Editor*)

which allows a developer to configure specific products and to prevent mistakes during the configuration process (cf. Fig. 3.3c). The *Configuration Editor* is the default editor of the configuration files (i.e., automatically opened using double-click). We present a brief overview on the configuration and execution of products in Sect. 3.6. More details are given in Chap. 6 on Page 63.

The directory `features` represents the domain implementation of *FeatureIDE* projects based on *feature-oriented programming* (FOP; see Chap. 13 on Page 143 for more details). FOP separates the implementation of each feature that we introduced in the `model.xml` in a dedicated subdirectory of the directory `features`. The directory is separated into folders for each feature, which each contains the corresponding program artifacts (cf. Fig. 3.3d which implements the feature `Hello`). Each file in the directory *features* is connected to the default *Eclipse* editor. Thus, if we double-click on a Java file, the *Java Editor* will be opened. Afterward, it is possible to edit each file in a straightforward manner as given in an *Eclipse* Java project.

Finally, the selected features from the current configuration are generated into the source folder `src`. This folder is then compiled by the underlying compiler (e.g., the Java compiler for Java projects). Thus, the generated product can be executed as usual in *Eclipse*.

3.4 Modeling Variability with Feature Models

Instruction 3.3 (Opening the Feature Model Editor)
Open the feature model editor by:

- Double-click on the model.xml in the Project/Package Explorer

As described in the last section, the feature model of a product line is stored in the file *model.xml* of each *FeatureIDE* project. In this section, we describe the default editor for the *model.xml*, the *Feature Model Editor*. To open the *model.xml* using the *Feature Model Editor*, follow Instruction 3.3.

In Fig. 3.4, we depict the *FeatureIDE Feature Model Editor* using our example project *HelloWorld-FH-Java*. This editor offers three tabs that can be used for

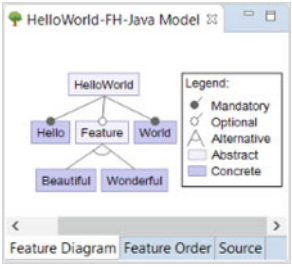

Fig. 3.4 The *Feature Model Editor* allows product-line developers to edit feature models, such as the *Hello World* feature diagram of the example *HelloWorld-FH-Java*

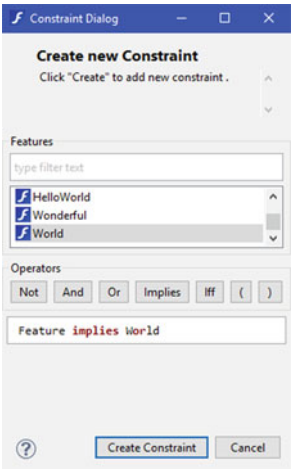

Fig. 3.5 The *Constraint Dialog* supports the product-line developers during the creation of cross-tree constraints

editing. First, the editor offers a tab for graphical editing of feature diagrams. The second tab allows a developer to define the feature order, which may be needed to ensure a correct product generation of a *FeatureIDE* project (more details are given in Chap. 5 on Page 43). Third, the *Feature Model Editor* also offers a tab that allows a developer to directly edit the textual representation as **.xml* file.

Let us take a closer look into the graphical editor. In Fig. 3.4 on the next page, we depict the feature model of the project *HelloWorld-FH-Java* with the page *Feature Diagram*. With the editor, a developer is able to add, remove, and change features and their dependencies. For instance, it is possible to rename the feature `Feature` to `Specification` or to add the feature `Perfect` as a child feature to the existing Alternative-group. Furthermore, we can change dependencies so that, for instance, the feature `Feature` is a mandatory feature. In particular, it is also possible to add *cross-tree constraints* to the feature model. The developer can describe an arbitrary propositional formula based on the set of existing features. Therefore, *FeatureIDE* offers an additional dialog that ensures the syntactical correctness of described cross-tree constraints (see Fig. 3.5 on the following page). The dialog can be opened using the context menu or a double-click on an existing cross-tree constraint. Using the *Constraint Dialog*, the developer immediately gets feedback about the correctness of the constraint to prevent the creation of incorrect constraints.

3.5 Implementation of Software Variability

The implementation of a product line and the respective implementation procedure differs according to the used programming language (e.g., Java, C++) and generation mechanism (e.g., preprocessors). In this chapter, we only focus on an introduction based on the programming language Java and the generation mechanism *feature-oriented programming* (FOP) using the *FeatureIDE* project

HelloWorld-FH-Java. In Chap. 17 on Page 199, we give some further insights into the support of other languages and paradigms.

As presented in Sect. 3.3 on Page 22, a *FeatureIDE* project based on FOP consists of two source folders: the folder *src* for generated source files and the folder *features* for implementation artifacts. Thus, the editable implementation artifacts of a product line based on FOP are located in the folder *features*. In contrast, the folder *src* is only the output folder for the generator and the content changes by each product generation (i.e., build process). Therefore, it is not intended to manually change the files of folder `src`. Nevertheless, the folder `src` can be helpful if program failures occur and more details are needed to find the error. To avoid accidentally changing the generated code, the files are marked as derived and the user gets a warning when trying to modify them.

Let us take a look into the folder *features*. The folder consists of a set of subfolders that represents the feature modules of the specific FOP project. In detail, each concrete feature of the feature model described in the *model.xml* is represented by one subfolder in which implementation artifacts and respective program files can be described. For instance, the example project *HelloWorld-FH-Java* (see Fig. 3.4 on the previous page) consists of four concrete features (`Hello`, `World`, `Beautiful`, and `Wonderful`) that are represented as subfolders in the source folder *features* (see Fig. 3.3 on Page 23). Each of the feature modules contains implementation artifacts, Java files, which we can edit to change the behavior of the product line's products. To open a source file, follow Instruction 3.4.

Instruction 3.4 (Open a Source File for a Specific Feature in FeatureHouse)
Open a feature implementation by:

- Double-click on *features* → <*feature*> → <*class*>.*java* in the Project/Package Explorer

3.6 Creating Configurations

Instruction 3.5 (Opening the Configuration Editor)
Open the configuration editor by:

- Double-click on a *.config file in the config folder of the Project/Package Explorer

Before we are able to run a specific product of the *Hello World* product line, we need to select all features that should be included. Therefore, *FeatureIDE* provides configuration files, in which the selection is stored. As described above, all existing configurations of a project are stored in the directory `config` and the

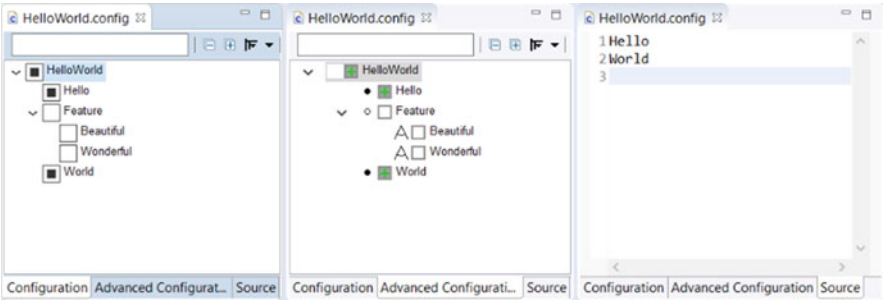

Fig. 3.6 The *Configuration Editor* with *Configuration*, *Advanced Configuration*, and *Source* tab to support the config file editing

active configuration (i.e., the product that is used for the build process) is marked by a *green pencil*. Typically only the active configuration is built automatically on each change. To open a configuration with the *Configuration Editor*, follow Instruction 3.5.

A developer can use *FeatureIDE*'s *Configuration Editor* to have a look into the selected features of a **.config* file and to change the selection. Therefore, the Configuration Editor provides three pages, (a) a *Configuration Page*, (b) an *Advanced Configuration Page*, and (c) a *Source* tab for the textual representation of the file. Whereas the Source tab presents all selected features in a textual manner, the Configuration and Advanced Configuration tabs support the configuration process and ensure that the selection does not lead to invalid configurations. In Fig. 3.6, we depict the Configuration and Advanced Configuration tabs using our *Hello World* example. In Chap. 6 on Page 63, we present a more detailed description of the configuration process.

3.7 Product Generation and Execution

Instruction 3.6 (Executing the Current Configuration)
Run the Current Configuration by any of the following options:

- Press *Run As...* → *Java Application* in the upper menu toolbar of *Eclipse*
- Press *Run As* → *Java Application* in the context menu of the Project/Package Explorer
- Press *Run* → *Run (Ctrl + F11)* in the menu bar of *Eclipse*

Once the feature modeling, feature implementation, and product selection is done, we can start to build and run a specific *Hello World* product. Therefore, *FeatureIDE* reuses all well-known procedures that *Eclipse* provides for a project build and launch. Thus, *FeatureIDE* allows a developer to use all ways to create and run an *Eclipse Run Configuration* for *FeatureIDE* projects.

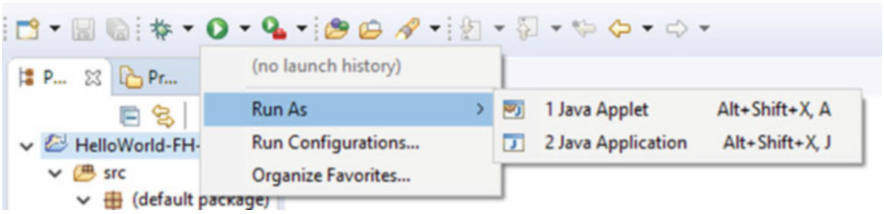

Fig. 3.7 Create a *Run Configuration* for *FeatureIDE* projects using the toolbar

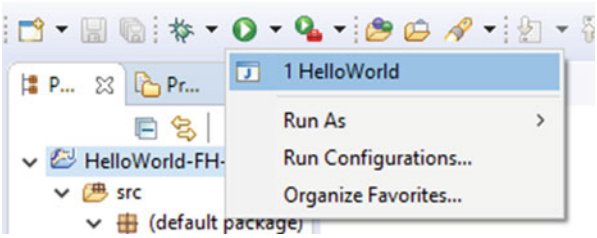

Fig. 3.8 Rerun the *Run Configuration*

Even if the concept of *Run Configurations* is well known by developers who use *Eclipse* as an integrated development environment, we give a short overview on how to create and use it. As mentioned above, *Eclipse* provides multiple ways to create a *Run Configuration* as described in Instruction 3.6 (cf. Fig. 3.7). Depending on the programming language, the submenu varies slightly. In this example, the *FeatureIDE* project is based on a Java project. Thus, the submenu allows us to create and launch a *Run Configuration* for Java. Afterward, we can reuse the created *Run Configuration* to relaunch the project's configuration (see Fig. 3.8).

Due to false settings, such as possible start parameters, the created *Run Configuration* may not launch the project correctly. In this case, we have to set up the created *Run Configuration*. Therefore, we use the menu entry *Run Configurations...* (cf. Fig. 3.7). Using this menu entry, we can open the default dialog for *Eclipse* configurations (cf. Fig. 3.9) that allows us to edit or create all kinds of configurations. Depending on the type of the *Run Configuration*, we can define all needed start information, such as the starting class or start parameters.

3.8 Summary and Further Reading

In this chapter, we gave a general overview on the basic functionalities of *FeatureIDE*. We introduced to the support of *FeatureIDE* for the main phases of feature-oriented software development. We showed how *FeatureIDE* supports domain engineering with support to create and edit feature models. We explained the general process of implementing variability in software. Then, we introduced

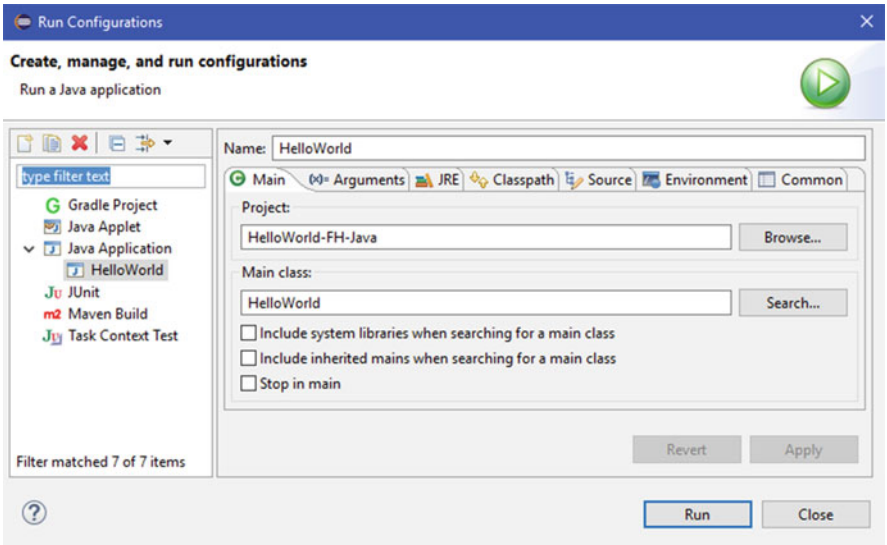

Fig. 3.9 Setup of *Run Configurations*

how *FeatureIDE* provides support to configure products. And finally, we showed how these products can be generated and executed.

As this chapter's purpose is only to give a general overview on the functionalities of *FeatureIDE*, we give detailed descriptions and more specialized support in the rest of this book. Support for feature modeling and product configuration will be discussed in Part II. How *FeatureIDE* supports implementation with conditional compilation (aka preprocessors) is explained in Part III. The support of feature-oriented programming as used in this chapter is explained in Part IV. In Part V, we shortly describe support for further generation mechanisms, namely, runtime variability, black-box frameworks, and aspect-oriented programming. We also give an overview on the purpose of all *FeatureIDE* views and editors in the last part.

http://www.springer.com/978-3-319-61442-7

Mastering Software Variability with FeatureIDE

Meinicke, J.; Thüm, Th.; Schröter, R.; Benduhn, F.; Leich, Th.; Saake, G.

2017, XII, 243 p. 50 illus., 30 illus. in color., Hardcover
ISBN: 978-3-319-61442-7