Masaryk University
Faculty of Informatics

PerfCake Plugin for Integrated Development Environments

Master's Thesis

Bc. Jakub Knetl

Brno, Spring 2017

This is where a copy of the official signed thesis assignment and a copy of the Statement of an Author is located in the printed version of the document.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Jakub Knetl

Advisor: Mgr. Martin Večeřa

Acknowledgement

I would first like to thank my adviser Mgr. Martin Večeřa for his helpful ideas, insights, and advice. I would also like to thank Ing. Pavel Macík for his valuable opinions.

Abstract

The main goal of this thesis is to provide a PerfCake plugin for IDEs, which contains a graphical editor for PerfCake scenarios designing. This plugin improves the PerfCake user experience mainly by simplifying the scenario definition process.

The first part of this thesis examines the PerfCake framework and the plugin development process for IntelliJ IDEA, NetBeans, and Eclipse. The second part introduces the design of the solution and describes the implementation of a universal plugin, which can be integrated into the IDEs, including details about integration into IntelliJ IDEA.

The resulting plugin has a universal design which allows integration into the IDEs. Complete integration of the plugin into IntelliJ IDEA demonstrates the plugin functionality.

Keywords

PerfCake, IDE plugin development, user interface design, graphical editor design, Eclipse, IntelliJ IDEA, NetBeans, Java

Contents

List of Figures

1 Introduction

Performance testing is an important part of many software projects. Various applications and programs can be accessed by a huge number of clients simultaneously; thus, they are required to behave properly under a heavy load. There are multiple performance testing tools and frameworks available, which can verify software behavior under a heavy load. One of these tools is the *PerfCake* framework. PerfCake is a lightweight performance testing framework, which enables testing of multiple protocols and permits its users to extend a set of supported protocols easily. In order to use PerfCake, users are required to write a *scenario* which defines a course of a test. The scenario definition must follow specified syntax rules based on a selected language[1].

Writing a scenario is a complex task for users. Providing a dedicated editor for scenario designing significantly improves the user experience. As a result, four editors were developed as plugins into integrated development environments (IDEs), namely into IntelliJ IDEA[2], Eclipse, and NetBeans. These plugins significantly enhance the PerfCake user experience and speed up scenario development.

On the other hand, the existing editors have several problems. Firstly, these plugins are consistent neither in their user interface controls nor in the provided functionality. Secondly, the plugins were developed in separation. Therefore they do not share any codebase which makes maintenance of these plugins very hard. Lastly, PerfCake is an active software project which constantly evolves and provides new functionality. Consequently, all of these plugins has become outdated.

The main goal of this thesis is to provide a universal scenario designer library, which is pluggable into the mentioned IDEs. The library should unify all the existing plugins and provide a uniform codebase and functionality. By this approach, the consistency of the plugins should be increased and the maintenance should be simplified. Moreover, updating one universal library according to new PerfCake features will require much less effort than maintaining three different

1. Scenarios can be written in XML, DSL, or Java API.
2. Two of the editors were plugins for IntelliJ IDEA.

projects. The second goal of the thesis is to demonstrate the functionality of the library by integrating it completely into IntelliJ IDEA.

The next chapter of this thesis contains a description of the PerfCake framework. The third chapter briefly describes basics of plugin development for the IDEs and underlines differences between their platforms. The fourth chapter defines features and benefits of the scenario designer. The fourth chapter also contains descriptions of the existing designers, their comparison, issues, and proposed solution. The fifth chapter describes the high-level design of the solution. In the last chapter, main concepts of the solution are analyzed in more detail.

2 PerfCake

In this chapter, the PerfCake framework is described. In the first section, basic concepts of PerfCake are covered. The next part outlines PerfCake architecture along with its main components. Lastly, usage of PerfCake is illustrated with emphasis on the user experience.

2.1 What is PerfCake?

PerfCake is a performance testing tool which focuses on simplicity. The PerfCake authors define PerfCake in the following way: "PerfCake is a lightweight performance testing tool and a load generator with the aim to be minimalistic, easy to use, provide stable results, have minimum influence on the measured system, be platform independent, use component design, allow high throughput." [1]. These properties make PerfCake suitable for various performance testing use cases. Additionally, the properties allow users to use PerfCake for an arbitrary type of performance testing [1].

PerfCake is written in the Java programming language[1], but it does not mean that it is limited to testing performance of applications written in Java. In fact, it allows testing of an arbitrary system which exposes an interface or a protocol. Support for numerous common protocols is included directly in PerfCake. In case that a target application defines a special protocol, users are required to provide PerfCake with an extension implementing the protocol to enable PerfCake to communicate with the target. More details on extending PerfCake can be found in section 2.3.2.

Although the PerfCake contributors make efforts into developing PerfCake, there are still few enhancement requests in the issue tracker which are not implemented in the latest stable version[2] [2]. One of the requests is to provide an up-to-date plugin for various integrated development environments (IDEs), which would make the user experience even better. This issue is one of the goals of this thesis.

1. There is also a very small amount of code written in Groovy.
2. In the time of writing this text the last version is 7.4.

2.2 Architecture

This section introduces PerfCake core components in order to describe its architecture and define important terms for the next chapters. However, this thesis introduces only the architecture outline. For obtaining more details this topic, I recommend the PerfCake Users' Guide [3], PerfCake Developers' Guide [4] or PerfCake source code [5]. All of these three resources served as the primary source for the rest of this chapter.

PerfCake has component architecture. Components are specified through well-defined interfaces; thus, it is easy to provide a new component implementation by a third party. The additional implementation is simply loaded into PerfCake and that is one of the reasons why PerfCake is easily extensible. More details are described in section 2.3.2.

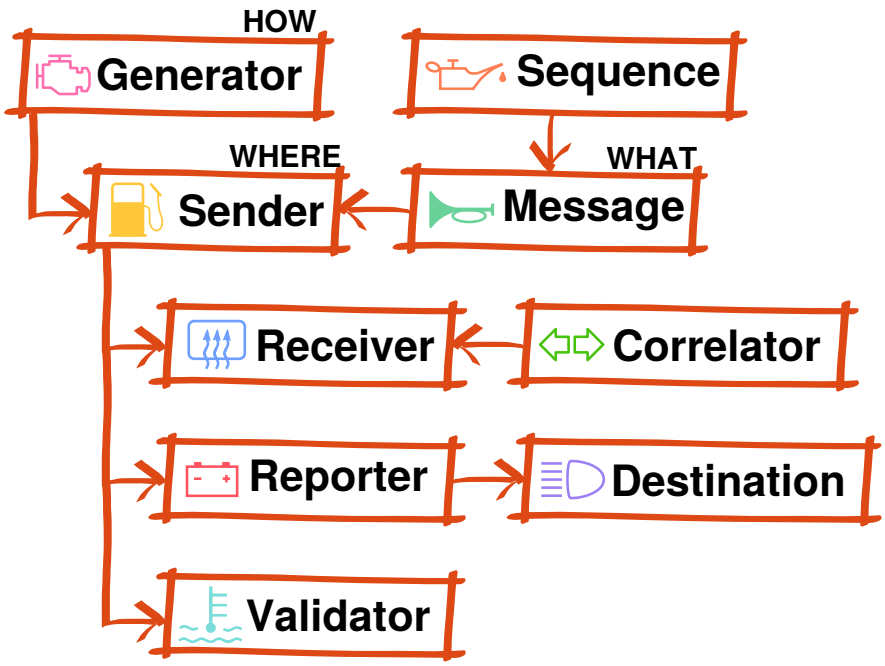

Figure 2.1: PerfCake components overview. Source: [3]

The main and the most important term is a *scenario*. The scenario describes a particular performance test. It contains information such as how a load is generated, how a *target system*[3] is reached, what the load contains, what quality will be measured by the test and other additional information. The scenario consists of many other components. Each of this more low-level components defines some particular aspect of a test. Figure 2.1 outlines all PerfCake component types. In the rest of this chapter, the PerfCake components will be defined. This thesis will go one by one and describe each component in more detail.

2.2.1 Generator

After defining the scenario component, a *generator* comes as one of the most important components. The purpose of the generator is to generate a load for a target system [3].

The generator manages a thread pool for generating a load and uses a *sender* component to transfer messages to a target system. As a result, generator implementations are crucial to be able to measure a target system performance. In case that a generator implementation is not good enough, then PerfCake reports about performance of a target system may be degraded. Therefore, users are advised to use existing generator implementations [4]. PerfCake provides four generator implementations by default. Namely, there is a `DefaultMessageGenerator`, a `ConstantSpeedMessageGenerator`, a `CustomProfileGenerator` and a `RampUpDownGenerator`. However, users are still able to implement their own generator by implementing the `MessageGenerator` interface.

Generators are configured to generate a load for a period of time or to perform a number of iterations. In each iteration, a sender component is used to query a target with a request. Additionally, it is also important to thoughtfully define a number of threads which will be used for the load generation as it will influence test execution.

3. Target system (or just target) is a system which is being tested, sometimes called system under test (SUT).

2.2.2 Sender

The sender is invoked by a generator to transfer a message to a target system [3]. In other words, the sender is responsible for implementing a protocol which can be used for communication with a target system. Additionally, sender can also receive a response from a target [4].

PerfCake provides several sender implementations; thus, users are able to communicate with a target through a number of protocols out of the box. For instance, there are senders for communicating with target system using HTTP (Hypertext Transfer Protocol), JDBC (Java Database Connectivity), MQTT (Message Queuing Telemetry Transport), sockets, files on a file system and many others. Furthermore, users are encouraged to develop additional senders by implementing the `MessageSender` interface.

2.2.3 Message

A message represents actual data which are transferred using a sender to a target [3]. By default, senders transfer one message per iteration. However, it can be configured to transfer an arbitrary number of messages in each iteration.

A payload of a message is usually specified in a separate file. A content of the file is filtered. Therefore, it is possible to replace part of the message dynamically during scenario execution. In other words, the content of the file is a template which can contain placeholders. These placeholders are replaced with actual values at runtime. A value is resolved from a PerfCake property, a system property, or an environment variable.

2.2.4 Sequence

A *sequence* component provides a mechanism which enables to modify a content of a message, influence behavior of a sender, or to send messages to a different target [3]. A goal of a sequence component is to generate a sequence of values. Subsequently, these values can be injected into a message content template so that every generated message has different content.

Similarly, as for the other components, PerfCake provides multiple sequence implementations. For example, users can use a sequence

of numbers, random sequence or timestamp sequence. The sequence interface is very simple. As a result, users can implement their own sequence easily.

2.2.5 Reporter

In order to record results of target performance, it is required to use a *reporter* component. Reporters allow measuring some software quality such as response time, a number of iterations per second or memory usage.

Each iteration of a sender creates a `MeasurementUnit` with some qualities of the iteration (for instance response time) [4]. These measurements are immediately propagated to a reporter. Consequently, the reporter implementation processes these results as needed to be able to record specified qualities. These quality measurements are subsequently recorded into one or more *destinations*.

2.2.6 Destination

A destination is used by a reporter to publish its measurements. Results of reporter measurements are outputted periodically. The destination component contains a specification of a *period*, which serves as a trigger for publishing the output. The period can be defined using an amount of time or by a number of iterations.

PerfCake contains multiple destination implementations. Test results may be reported simply into a console or a logger. On the other hand, more complex destinations can be used such as charts or elastic search database[4].

2.2.7 Receiver

In order to measure qualities like response time, PerfCake needs to get a response. Similarly, as a sender is used for transferring messages to a target, a *receiver* is used for transferring response messages from a target to PerfCake [3]. In simple scenarios, a sender can be used to both send a request and receive the response in the same channel. To be more specific, suppose there is the HTTP sender in a scenario. The

4. See ElasticSearch reference [6] for more details.

sender sends a request to a target and immediately gets the HTTP response for the request which is measured.

Nevertheless, in the more complicated scenarios, a response comes through a completely different channel. For instance, an HTTP request may result in adding a new record into a database. In order to measure this operation, a receiver must be defined because the HTTP sender cannot query the database to measure when the target responded.

To summarize, receivers are responsible for getting a response from a target in a different channel from the one through which the request has been sent. Additionally, every receiver needs a *correlator* component to work properly.

2.2.8 Correlator

A correlator component is used in conjunction with a receiver. In performance tests, there is a significant number of messages being sent to a target. As it is stated above, receivers are responsible for acquiring responses in a different channel. In performance tests, there is a large number of requests as well as a large number of responses. In order to measure the time between a particular response and the corresponding request, a receiver must know which response is related to which request. That is exactly what a correlator component is responsible for. The correlator matches responses to corresponding requests. Such a matching is usually achieved by recording a *correlation ID*[5] of a request and comparing it against a correlation ID of a response. Hence, a receiver is able to pair requests and responses together using a correlator.

2.3 How to Use PerfCake

Having covered PerfCake architecture, this section describes steps which are required from users to do performance testing using PerfCake. There are multiple ways how to make use of PerfCake in performance testing. Namely, users can use the PerfCake binary distribution, the Maven plugin or the Java Application Programming Interface (API)

5. ID which is included in both request and response. Additionally, the correlation ID is different for each request.

to run PerfCake scenarios [3]. Rest of this chapter will focus on using the binary distribution to illustrate the user experience of PerfCake.

At first, users need to install PerfCake. Installing PerfCake is as simple as downloading the latest version and unpacking an archive. However, in order to run PerfCake, users need to have installed a Java environment in the version 1.8 or later[6] [3]. The binary distribution contains all required dependencies and it also contains scripts which serve as the command line interface (CLI). There are scripts for multiple environments; including Bash scripts for the UNIX environment and batch files for the Windows environment.

The second step requires defining a scenario. The scenario definition is the most complex and the most critical part of successful performance testing with PerfCake. For this reason, it will be covered in the separate chapter 2.3.1.

The next step involves actual scenario execution. The CLI interface is very straightforward and simple. It takes several options but the only required option is the `-s` option followed by a scenario name. Subsequently, PerfCake loads the scenario from a scenarios directory. By default, the scenario directory is located in the PerfCake installation directory in `PERFCAKE_HOME/resources/scenarios`, where `PERFCAKE_HOME` denotes the directory to which PerfCake has been installed. The location of scenario directory can be altered by providing the `-sd` option. Consequently, after invoking PerfCake through the CLI it starts scenario execution.

The last step in the testing procedure is to inspect test results. This step is heavily dependent on a particular scenario definition. Depending on kind of a reporter component used in the scenario, users need to inspect a log file, a chart or another output of specified reporters to determine if the application performance is good enough.

2.3.1 Scenario definition

The scenario definition is the most challenging part for users. Defining a scenario involves choosing proper components, configuring the components, and assembling them together into the scenario definition.

6. In the time of writing this paper Java 1.8 is the latest stable release.

There are currently three ways how to define a scenario [3]. For the binary distribution, users can define a scenario using the extensible markup language (XML) or the domain specific language (DSL). In case that a scenario is executed directly from the Java API without using the binary distribution then the scenario is allowed to be defined also using the PerfCake fluent API. Examples of XML and DSL scenario files are shown in the attachment A.1.

For the purpose of writing a scenario, users need to open a text file in their preferred text editor or an IDE and edit the file manually. Regardless of a particular scenario definition language, writing a scenario is a complex task for users. As it requires knowledge of the syntax for at least one of the languages. Moreover, users also need to have knowledge of the PerfCake components and their parameters. Furthermore, users should know which components are required and components are optional.

Additionally, the majority of component implementations have a significant number of properties which alter the component behavior. These options are usually specific for a particular component implementation and do not apply for the other ones. For example, the HTTP sender has the *method* property that decides which HTTP method is used for communication (e.g. GET, POST, PUT, ...) [3]. What is more, some of these properties are required for a component implementation. In contrast, some properties have a default value and users do not need to specify them explicitly unless some tweaking of the value is required.

As a result, users need to take all above-mentioned parts of the definition into account. It makes writing a scenario hard and error-prone, especially for new users. On the other hand, the scenario definition is as simple as possible to preserve flexibility and generality of PerfCake. Thus, scenario definition cannot be simplified. However, the process of designing a scenario can be simplified significantly.

In general, one of the goals of this thesis is to design and develop a graphical scenario editor as a library which can be integrated into multiple IDEs. The editor should simplify the scenario definition process. Although there are tools which enable such a graphical editing, they have multiple drawbacks. These drawbacks are described in section 4.3.

2.3.2 Adding PerfCake Component Implementation

Although PerfCake provides a large number of components by default, it is possible that no suitable component exists in order to test a particular system. In that case, users extend PerfCake with a new implementation of the components mentioned in section 2.2. In order to add a new component implementation, users need to implement an interface of the component, compile the source code, bundle the compiled code as a JAR (Java Archive) package, and provide PerfCake with the package.

3 Plugin Development for Selected IDEs

The main goal of this thesis is to build a universal scenario designer library which is pluggable to various IDEs. In order to do that, this thesis needs to consider all supported IDE abilities and restrictions. This chapter describes and compares some aspects of a plugin development in the IntelliJ IDEA, Eclipse, and NetBeans. The reason why these three IDEs were chosen is described in section 4.1.

3.1 IntelliJ IDEA

3.1.1 User Interface

The IntelliJ IDEA user interface is written in Swing and provides its own look and feel[1] [8]. The user interface is composed of many parts. The most important parts are described in the following paragraphs.

The *Editor window* displays an edited file. It is the main part of the IDE. The window can have multiple tabs; therefore, it allows several files to be opened at the same time. A specific editor is chosen by a file type of the edited file. Some editors provide two or more representations of edited data so that it allows users to switch between representations.

Various *tool windows* are another important part of the UI (User Interface) [9]. These tool windows allow users to accomplish numerous file editing or programming tasks without leaving the IDE. The *project* window is one of the most important tool windows. The window displays files and directories in a project. The *structure* window displays a structure of an edited file, whereas the *run* window displays output of an executed code.

Tool windows can be organized within IntelliJ IDEA in different layouts [9]. A window can be pinned to a side of a screen or even maximized over all other windows and editors.

Lastly, there is a context menu and a toolbar, which allow users to invoke additional actions related to file editing, programming, the IDE configuration, or opening hidden tool windows.

1. See [7] for more details.

3.1.2 Plugin Structure

A plugin for IntelliJ IDEA is a standard JAR (Java Archive) package which contains a file `META-INF/plugin.xml` [10]. The file declares metadata about the plugin. It collects basic information such as a name, version, description, vendor, or dependencies on other plugins. The `plugin.xml` file also defines *extensions* and *extension points* [10], which are covered in the following section.

If a plugin has external dependencies then the JAR structure is different. The plugin archive contains a `lib` directory, where multiple JARs can be placed [10].

3.1.3 Extension Mechanism

Extensions and extension points are used to extend the functionality of the IDE or another plugin. Each extension point defines an interface for extending the functionality of a feature [9]. If another plugin wants to change or add behavior of the feature then it must define an extension for a particular extension point. If a plugin wants to provide a way to change its behavior, then it must define an extension point.

Another important concept of extending IDEA are *actions*. These actions can add items to existing menus and toolbars inside of the IDE. An action implementation is a Java class which implements the `AnAction` interface [9]. An action implementation defines what happens in a specific context of action activation.

3.1.4 Automation of Plugin Build Process

IntelliJ IDEA allows building plugins directly from the IDE. Since a plugin for IDEA is a standard JAR, an arbitrary Java build automation tool, such as Apache Maven[2], can be used for automating the build process. However, IntelliJ does not provide its API in any Maven repository. Therefore, in order to use Maven, it is required to ensure that JARs from the IntelliJ runtime are accessible in the build process. This can be achieved either by installing the IDEA API into a Maven repository or by providing Maven with a path to the IntelliJ runtime.

2. Description of Maven can be found on Maven website [11].

3.2 Eclipse

3.2.1 User Interface

Eclipse defines a *workbench* which refers to the whole user interface in the Eclipse window [12]. The Eclipse IDE uses SWT (Standard Widget Toolkit) instead of Swing [12]. Hence, it delivers native components for a particular system.

The most dominant part is an editor. Eclipse allows several editors to be opened at the same time. A particular editor is chosen for a particular file based on the file type. Eclipse also provides *multi-page* editors to allow users to view a content of a file in different representations.

The workbench also contains *views*. Views can serve for various purposes such as project navigation, displaying a file structure, displaying variables while debugging, console output, and much more.

Eclipse also introduces a concept of *perspectives*. A perspective prescribes what views are visible and their layout. It is based on the concept that users need a different set of views for different tasks. By switching perspectives, windows and their layout can be changed quickly.

Finally, Eclipse contains a toolbar and a context menu which allows performing supplementary tasks.

3.2.2 Plugin structure

Eclipse is based on the *OSGi*[3] technology [12]. As a result, Eclipse plugins are packaged as OSGi *bundles*. An OSGi bundle is very similar to a standard JAR package. In comparison with a standard JAR, an OSGi bundle must contain a manifest file with mandatory metadata [13]. This metadata includes a bundle name, bundle version, bundle activator, bundle dependencies, bundle API, and others.

OSGi favors modular applications [13]. This means that the majority of Eclipse plugins are usually formed by a group of OSGi bundles. Consequently, Eclipse defines a concept of *features*. A feature allows grouping a set of Eclipse bundles into one plugin, which simplifies the plugin installation and distribution.

3. OSGi is a framework which allows building dynamic modular application in Java. See [13] for detail description.

3.2.3 Extension Mechanism

In Eclipse, there is a similar concept of extensions and extension points as in IntelliJ IDEA. A plugin extends the IDE by defining an extension for a particular extension point [12]. Similarly, a plugin provides extension points in order to allow modifying its behavior by other plugins.

These extensions and extension points are defined in the `plugin.xml` file [12]. Naturally, the `plugin.xml` file has a different syntax than the `plugin.xml` file in IntelliJ IDEA.

3.2.4 Automation of Plugin Build Process

Eclipse plugins are usually built directly from the IDE which takes care of bundle manifest generation. It is also possible to use Apache Maven for building an Eclipse plugin.

In order to employ Maven, it is required to ensure that the OSGi manifest is provided inside of a bundle. It is possible to write the manifest manually, but it can be demanding task to capture all dependencies in proper versions. The other option is to use a specialized plugin, developed in the Tycho project [14], which allows building Eclipse plugins and features. Tycho also generates the manifest file. However, when Tycho is used for building a plugin, it is restricted to dependencies defined through the manifest. In other words, no dependencies defined through the Maven standard dependency management should be used.

3.3 NetBeans

3.3.1 User Interface

The NetBeans UI is also written in Swing and it uses the default Swing look and feel. Similarly, as both Eclipse and IDEA, NetBeans provides multiple windows in the IDE.

The main window is the editor window which is used for an edited file. The editor window also provides the possibility to display a file content in more representations. Other windows have similar func-

tionality as in IDEA and Eclipse such as the file window, the project window, the console window and others.

NetBeans also provides a toolbar and a context menu for performing additional actions.

3.3.2 Plugin Structure

The NetBeans IDE incorporates its own modularity system [15]. A NetBeans module is a JAR package which must define special entries in the JAR manifest file. These entries contain metadata about the module such as dependencies of the module and the module classpath.

A NetBeans plugin is a NetBeans module which additionally defines a set of *registrations* [16]. Registrations allow extending the NetBeans platform with additional features such as menus, controls, and editors.

Multiple plugin modules can be bundled together with their dependencies into one module. Thus, it allows distributing a plugin combined from multiple modules as one unit.

3.3.3 Extension Mechanism

The NetBeans IDE is extended using registrations. Although registrations are located in an XML file bundled in a JAR archive, they are not defined in a file. Plugin developers define registrations using Java annotations on source code classes and methods [16]. During the build process, these annotations are converted into the XML file with registrations.

Each annotation has its own type and multiple fields. Some of the annotation fields are required, other do not have to be defined.

3.3.4 Automation of Plugin Build Process

The NetBeans IDE allows building NetBeans modules directly from the IDE. NetBeans also provides a plugin for Apache Maven called *nbm-maven-plugin*, which enables to build NetBeans modules and plugins using Maven. Moreover, NetBeans provides its own Maven repository with all NetBeans API modules; therefore, the build process can be completely automated using Maven. In comparison with the

Eclipse IDE, the NetBeans Maven plugin allows using standard Maven dependencies, which are not packaged as NetBeans modules.

4 Scenario Designer

In order to simplify the process of the scenario definition, demands on users' knowledge should be minimized. Users should not be required to remember actual component names and component attributes. Users also should not be required to consult a component reference section in the Users' guide [3] to find out a particular parameter of a particular component. Such context switching between the documentation and scenario definition slows users down. Moreover, typing or copying of component names and parameters is error prone.

A convenient *scenario designer* should guide users through the scenario definition process so that users need to be aware only of component types (e.g. what is a purpose of the generator component). The designer should enumerate particular component implementations, allowing users to choose a suitable one. To simplify users decision making, description of components should be presented to users. By such enhancements, users are no longer required to search additional information in the documentation.

The scenario designer should also be able to display implementation specific properties, which brings multiple advantages to users. Firstly, users see what behavior of a component can be altered at first glance. Secondly, the designer does not allow supplying a property which is not supported by a specific component. Lastly, the designer is able to display all component-specific properties values even if they were not defined explicitly in a scenario. In that case, designer displays the default values for a specific component.

For example, if a user defines the HTTP sender in a scenario without any additional property, the designer displays that there are two properties supported by the HTTP sender in the sender component settings. In more detail, the designer displays form with the *method* and the *expectedResponseCodes* properties along with its default values which are *POST* for the *method* and *200,202* for the *expectedResponseCodes*.

4.1 Designer Platform

One of the PerfCake goals is to provide a designer [1] as mentioned in the previous section. The designer should use an IDE as its base platform. Furthermore, several IDEs should be supported. In other words, the designer should be implemented as a plugin for various IDEs.

Choosing an IDE as a platform has several benefits. Firstly PerfCake is used mostly by software engineers. Software engineers are usually familiar with some kind of an IDE. Hence, users are allowed to use tool which they are accustomed to. Furthermore, if a Java IDE is chosen, users are allowed to start a debug session right away from the IDE.

There are multiple surveys which show the popularity of different IDEs [17][18]. These surveys have significantly different results. However, in all surveys, there are always Eclipse, IntelliJ IDEA and NetBeans in the list of the most popular IDEs for Java developers [19]. That is the reason why one of the requirements for the plugin is to support these three IDEs.

Another PerfCake goal is to provide a web-based graphical designer for scenarios. The designer would run in a cloud and users would access the designer using their web browser only. Nevertheless, designing a web-based editor is not a goal of this thesis and will not be further considered.

4.2 Existing Designers

At the time writing this thesis, four scenario designers exist as plugins for a particular IDE. However, each of them has some issues. In the rest of this section, the existing designers are shortly introduced. The designers are ordered chronologically.

4.2.1 PerfClipse

PerfClipse is a plugin for the Eclipse IDE. PerfClipse was written to bring the support of PerfCake 2.0 into Eclipse 4.3[1]. Later, it was

1. In the time of writing this text, current version of Eclipse is 4.6.

updated to support PerfCake 3.3. There were also attempts to update PerfClipse to support PerfCake 6.2, but it has not been finished yet [20].

PerfClipse equips the Eclipse IDE with the various functionality. It provides a component scanner which detects PerfCake component implementations at runtime [21]. Additionally, PerfClipse integrates wizards into Eclipse, which guide users through the new scenario definition process. Mainly, it extends the Eclipse IDE with the graphical editor for scenario modeling. The editor allows users to drag and drop additional components from the palette into the graphical representation of a scenario. What is more, PerfClipse allows users to run PerfCake scenarios directly from the IDE. PerfClipse can edit only scenarios defined in the XML format.

PerfClipse uses GEF 3 (Graphical Editing Framework 3) for the visual representation of scenarios. GEF 3 is a graphical framework based on top of SWT (Standard Widget Toolkit).

4.2.2 PerfCakeIDEA

The PerfCakeIDEA project was established when PerfClipse was in a development stage. During a significant amount of time, PerfCakeIDEA and PerfClipse was developed simultaneously. PerfCakeIDEA provides a PerfCake plugin for IntelliJ IDEA 14.0.2[2] [22]. PerfCakeIDEA uses PerfCake version 3.3.

PerfCakeIDEA provides very similar functionality to PerfClipse. Even though PerfCakeIDEA uses Swing[3] instead of SWT, the editor is nearly identical with the PerfClipse editor. Actually, one of the PerfCakeIDEA design goals was to be consistent with the existing PerfClipse plugin.

4.2.3 Pc4idea

Pc4idea is another plugin for IntelliJ IDEA. The main goal of the pc4idea project was to update the PerfCakeIDEA plugin so that it also supports editing of scenarios defined using DSL and support PerfCake 4.0 [23]. Moreover, pc4idea was developed to be compatible with IntelliJ IDEA 14.1.4 [24].

2. Latest version is 2017.1.1 in the time of writing this paper.
3. For more details on swing see [7].

The design of the graphical editor is also very similar to both PerfClipse and PerfCakeIDEA. However, pc4idea delivers functionality which is not present in PerfClipse or PerfCakeIDEA. Pc4idea enhances scanning of component implementations. It scans not only component implementations at runtime but also their properties.

In addition, the designer displays all supported properties for a component implementation and its default values. Hence, it is simple for users to overwrite default values. This is a great improvement for the user experience because the designer does not require users to type long property names. This feature also accentuates what can be configured in a specific component. Moreover, it also explicitly displays effective values for properties, even in the case that no values are defined in a scenario and default values are used.

4.2.4 Pc4nb

Pc4nb is a plugin for the NetBeans IDE 8.0[4]. It was developed against PerfCake 6.0-SNAPSHOT in order to provide the latest PerfCake functionality [25]. Nevertheless, it was not updated to use the version 6.0 after the PerfCake 6.0 release.

The pc4nb project is very similar to pc4idea, but it integrates PerfCake into another IDE. Pc4nb also provides similar functionality. However, it cannot edit scenarios defined in DSL.

4.3 Designers Issues

Whereas the existing designers fulfill their purpose, they have multiple drawbacks. These drawbacks decrease plugins quality, value, and deteriorates the user experience.

4.3.1 Outdated Product Versions

One of the most significant drawbacks of all the plugins is the fact that they are not up to date. Each plugin should maintain compatibility with the latest PerfCake and the latest IDE version. As it is apparent from the existing plugin description in section 4.2, neither of them

4. At the time of writing the latest version is the 8.2.

satisfies this constraint. Therefore, users cannot use the latest PerfCake features or they are forced to install obsolete IDE versions in order to be able to install the plugins properly.

4.3.2 User Experience

The graphical editors in the plugins are very similar in the major concepts. All of them visualize a scenario directly by reflecting a structure of an underlying XML file. As shown in figure 4.1, XML elements are represented as rectangles. A nested element is represented as a rectangle nested into another rectangle. Consequently, the scenario visualization is a hierarchy of rectangles which is same as a hierarchy of elements in an XML file. Although this straightforward approach is convenient for developers, it does not simplify a scenario structure for PerfCake users.

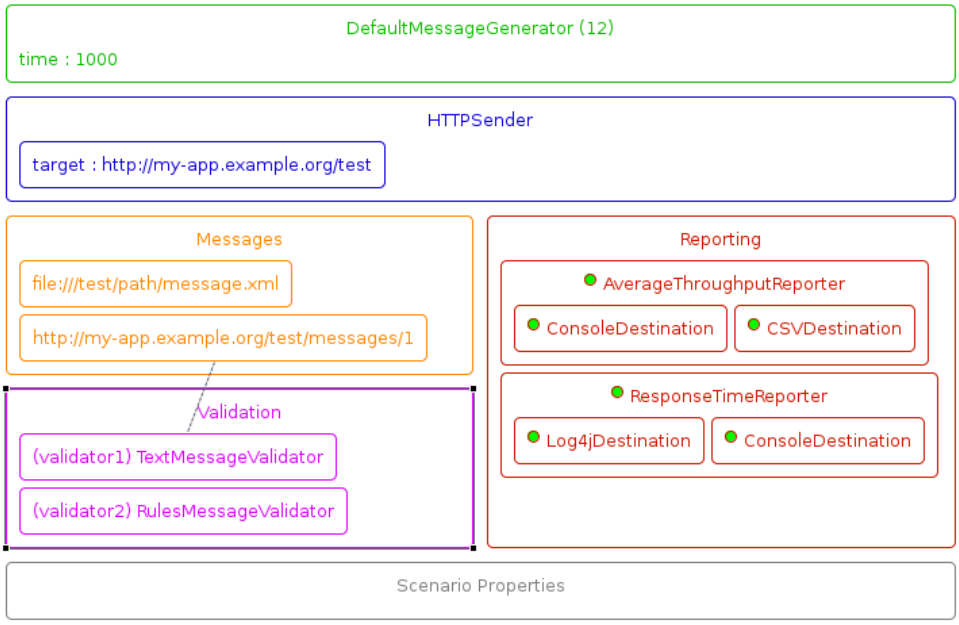

Figure 4.1: An example of a scenario in the PerfClipse designer.

In addition, the existing designers insufficiently help users to decide which component is more appropriate. In the same way, the

designers do not expose description of various component properties to users. Therefore, users must know the properties in advance in order to use them properly. A superior editor design should assist users with components definition.

4.3.3 Consistency

The plugins should have a consistent user interface so that users can transition between different IDEs without changing their editing habits. Although the plugins are consistent in the graphical scenario editor main concepts, they differ greatly when component detailed settings are opened or when a new scenario wizard is launched.

Similarly, the provided functionality by the plugins for different IDEs should be same. Some of the later plugins added a new functionality. Consequently, such a functionality is not present in plugins for other IDEs. The plugins should be unified in a way to provide a same set of features.

4.3.4 Maintainability

The plugins do not share any codebase. Even if the plugins provide a similar set of features, these features are not implemented once and are not shared between the plugins. Every plugin implements everything on its own which implies major drawbacks. From the user experience point of view, features provided by the plugins are similar, but the plugins do not behave exactly same.

From the developer point of view, missing a shared codebase makes the maintenance significantly more difficult. In case that a feature needs to be changed, it has to be changed three times, once for each IDE plugin. It also increases demands on developer or maintainer knowledge since a developer must understand a structure and an architecture of three different plugins.

Despite the plugins are well prepared for an automated building, none of them is built completely automatically. The plugins do not use any CI (Continuous Integration) system. It also conduces to maintainability issues. It is much easier for a maintainer to fix small problems immediately when they are detected than to fix large problems late when a plugin does not work completely.

In conjunction with the CI, the plugins do not define a sufficient set of tests which would detect problems early. For example, PerfClipse and pc4nb do not define any test suite. Other plugins, such as pc4idea, defines a set of tests. These test suite should be also consistent to ensure that the same set of features works properly in different IDEs. But since there is no shared codebase it is not easy to provide a unified set of tests.

4.4 Proposed Solution

As there are many problems in the existing IDEs, there are also more ways how to remove them.

The most straightforward solution would be to go through each plugin and try to eliminate above-mentioned issues one by one. Such an approach would require to update compatibility with the latest PerfCake and the latest IDE versions, define features which are missing in the plugin, and modify some features so that they are consistent with the other plugins. These changes would enhance the user experience and the consistency. From the maintainability point of view, the build process could be unified and automated within this approach. A set of tests would have to be implemented for each project. These enhancements would allow detecting problems and incompatibilities for future versions immediately. Nonetheless, if some of the editor features needs to be changed, then it will be necessary to update the features three times. Furthermore, it would be required to maintain each implementation compatible with the latest PerfCake and the latest IDE. To summarize, this approach could be used to eliminate the majority of problems, but it would not simplify the maintenance.

Therefore, it is highly beneficial to have a shared codebase, which would be general enough to allow integrating the shared codebase with all above-mentioned IDEs. Shared implementation greatly reduces the maintenance complexity. Accordingly, another approach includes extracting a shared codebase from the existing plugins. Though it would be the very complex task because the plugins use different technologies and were not designed to be general enough.

Because of above-mentioned reasons, this thesis proposes a completely new graphical scenario editor called *pc4ide*. Pc4ide is designed

carefully; therefore, it is completely independent of any IDE. In other words, the pc4ide serves as a library which can be integrated easily into the all three supported IDEs[5]. Moreover, designing the completely new editor enables to make significant changes in the user interface, in order to enhance the user experience.

Consequently, the very thin layers which provide integration of pc4ide into the IDEs are defined. These integration layers map IDE specific actions to the pc4ide editor actions. These layers are desired to be as thin as possible in order to make the maintenance of the plugins simple. Having the integration layers thin implies that some of the IDE specific features cannot be easily utilized. However, the advantage of the simple maintenance is more valuable.

Lastly, the implementation of the pc4ide library contains set of tests for integration with PerfCake. The build process is completely automated and triggered automatically. This facilitates to discover incompatibilities early when a new version of PerfCake is about to be released.

5. Theoretically also to other IDEs which supports plugins written in Java.

5 Design of the Pc4ide Plugin

This thesis proposes designing and implementing a new scenario editor named *pc4ide* (as a shortcut for PerfCake for IDEs). The pc4ide editor is implemented as a universal library, which allows integrating integrate pc4ide as plugins into various IDEs (namely Eclipse, IntelliJ IDEA, and NetBeans). As a part of this thesis, the pc4ide library is also completely integrated into IntelliJ IDEA as the plugin called *pc4ide-intellij*.

5.1 Design Considerations

The implementation of the graphical editor must be pluggable to the multiple IDEs. Each IDE provides its own API and a set of tools which can be utilized for a plugin development. However, this IDE dependent API and tools cannot be ported to another IDE easily. Each of the existing PerfCake plugins is highly dependent on the API of the corresponding IDE. For this reason, it is not easy to merge the existing plugins into a universal one. Therefore, this thesis proposes a design of a new universal scenario editor library called pc4ide, which can be easily integrated into various IDEs.

The pc4ide library tries to have as low dependencies on a particular IDE as possible. Actually, it is designed independently of any IDE. Although the plugin is not expected to be distributed as a separate application, it is possible to run pc4ide outside of any IDE[1]. This approach has two major benefits. Firstly, such a design makes sure that no IDE specific tool is present in the main part of the universal plugin. Secondly, it simplifies pc4ide testing and development, because it does not require to install and configure development tools and development platform for a particular IDE.

The pc4ide library provides its own API which can be used to integrate the graphical editor into all three IDEs. The integration layer between a core of the library and the IDE plugin is designed to be as thin as possible. Such an approach minimizes code duplication be-

1. However, such a non-IDE runtime has neither project file management support and settings.

tween the plugin distributions for different IDEs because the majority of logic is located in the shared part of the library. The integration layer only binds this logic to specific parts of an IDE.

5.2 User Interface Design

User interface (UI) design is very important in graphical applications. This chapter describes the user interface of the scenario editor in the pc4ide library.

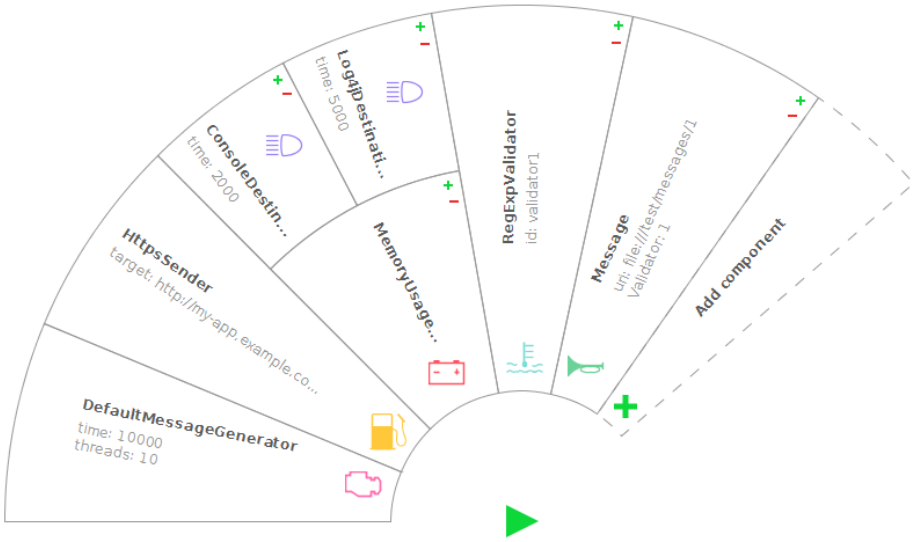

Figure 5.1: An example of a scenario displayed by the graphical panel.

The scenario editor consists of two panels. A graphical panel is the main panel. As shown in figure 5.1, it graphically visualizes a scenario. The visualization displays only the most important information about the scenario. For example, what components are defined in the scenario and their most important properties. Users can add components to the scenario and remove components from the scenario using the graphical panel. Components in the graphical panel are organized into circular sectors which should evoke a car speedometer. After all,

the pc4ide plugin is also used to measure "speed"[2] of applications. Each component is associated with an icon as a visual cue so that users can recognize similar components quickly. In order to support the speedometer vision, the icons are inspired by icons in a dashboard of the Škoda 742[3] cars produced from late 70's up to 1990 by former AZNP, today known as Škoda Auto [26].

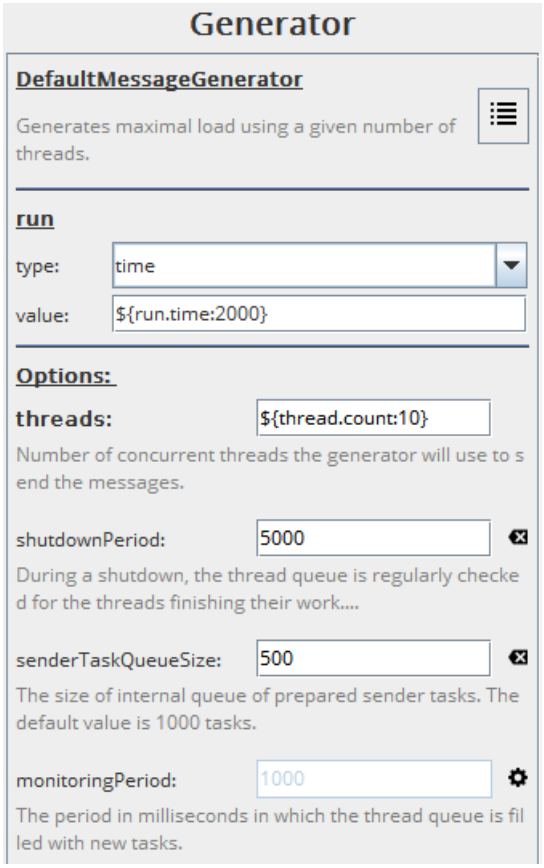

Figure 5.2: An example of a generator displayed by the form panel.

The second panel is a form panel. The form panel displays details of a component which is currently selected in the graphical view.

2. Though, performance is more precise term.
3. For example, Škoda 105 and Škoda 120.

Example of the form panel is shown by figure 5.2. If no component is selected, then a structure of a scenario is displayed. The form panel displays all component options and details, which can be configured for a component.

If a component option is not set explicitly in a scenario, then a default value for the option is displayed. Users can see at first glance that the value is not set explicitly because the value text field is grayed out, as it is depicted in the figure.

Furthermore, if a user enters an invalid value for an option, then the entered value turns red, signaling the user that the value is wrong. For example, if the user enters a non-numeric value for the shutdownPeriod option, then its field turn red. However, the option value validation is able to detect placeholder. As a result, the threads option is not visualized as invalid in the figure, even if the value is not strictly numeric.

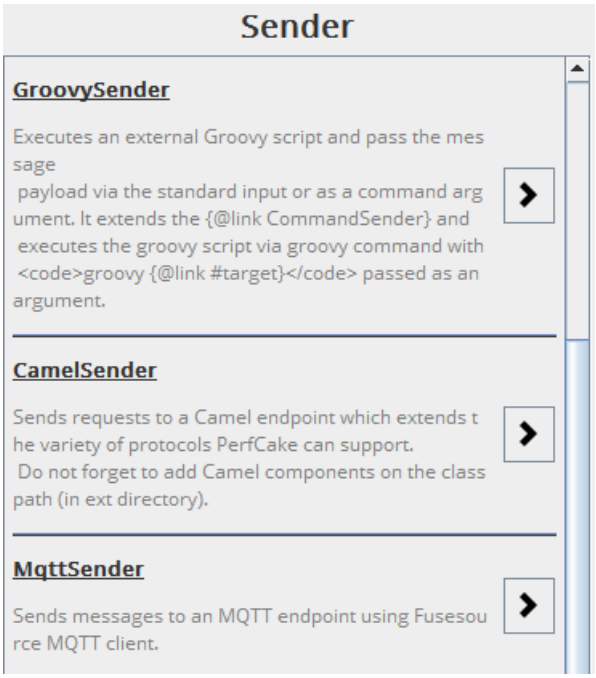

Figure 5.3: An example of choosing an implementation of a destination.

Since the form knows valid values for particular options. It is also capable of showing proper control element in the form. To be more specific, the type field of the run settings is displayed as a combo box. This feature prevents users to input invalid values.

In addition, the form panel also contains descriptions of component options and a description of a component implementation. This helps users to set options properly, even if they did not know options in advance. The documentation is obtained automatically from the PerfCake sources at the build time. Hence, if there is a change in the PerfCake source code, the documentation will be updated simply by rebuilding the application.

Similarly, descriptions of whole components are shown, when users are choosing a component implementation. However, in this case, full descriptions are shown in order to help users with choosing a proper implementation. The form for component implementation selection is shown in figure 5.3.

5.2.1 Design Rules

The user interface of the scenario editor tries to follow good practices for UI design. This subsection goes through some design rules used in the scenario editor.

First of all, *consistency* is a very important rule to follow. The graphical panel of the editor displays all components very similarly. Moreover, the form panel displays component properties with the same controls and same order based on a property type. This allows users to familiarize with the editor quickly.

Equally important, there is a group of design rules called Gestalt principles. They are based on the fact that people tend to perceive shapes and figures as whole structured objects rather than a set of shapes and areas [27]. The design of the editor took the Gestalt principles into account. For example, the *similarity* principle is used in the form panel, which displays all properties of the same type in the same manner, which makes them appear as a group rather than single items. Additionally, use of the *proximity* principle, by grouping these similar items together, emphasize their grouping even more. The *figure/ground* principle is used to separate what is a figure and what is a ground in the graphical editor.

According to Johnson [27], users should not be required to read long texts. On the other hand, one of the pc4ide goals is to guide users and explain various component options. Some of these descriptions are long. For this reason, pc4ide displays descriptions with very pale font hue in the form panel just below the property options. This approach makes description texts appear less significant and it does not distract users. Moreover, if a property description is long, then only part of the description is displayed by default. Users must navigate a mouse on the description in order to display the whole text. As a result, users are able to display descriptions easily, but in a non-distracting way. However, there is an exception in the pc4ide library. When users choose a component implementation, they are presented with full component description. The reason is that choosing a proper component implementation is very important for every scenario. Therefore, presenting a complete description of a component is considered helpful.

Johnson also recommends choosing colors in user interfaces carefully. Colors should be presented so that they are easily distinguishable. On the other hand, user interfaces should "separate strong opponent colors" [27]. The reason is that placing strong opponent colors together causes disrupting of users. Consequently, the colors were inspired by Kempson's base16 color schemes [28].

Another rule recommends not to require users to remember lots of information. Human working memory[4] is very limited. As a result, only a limited set of information can reside in users' working memory [27]. Users should be concentrated on their goals or tasks which need to be performed. If users need to remember lots of additional information given by the user interface (e.g. instructions), then they are often unable to focus on their goals [27]. Additionally, if users' tasks are complex, they need to use nearly all their attention to the tasks. For this reason, the pc4ide avoids burdening users with lots of information.

Similarly, another rule related to the working memory suggests to "use visual cues to let users recognize where they are" [27]. Therefore, the graphical panel highlights a component which is currently displayed in the form panel in order to help users to keep track of which component is being edited at the moment.

4. Also known as short-term memory.

5.3 UI Framework

Pc4ide must enable simple integration into IntelliJ IDEA, NetBeans, and Eclipse. Hence, its UI framework must be compatible with all these IDEs. Plugins for all of the mentioned IDEs are written in Java; thus, the graphical framework should be implemented in Java. As a result, the SWT, JavaFX, and Swing frameworks were considered as a framework for graphical components such as buttons and text fields.

The SWT option was rejected because it uses native system libraries. This causes different look and feel on different platforms which would complicate designing the UI so that it look properly in all environments. Furthermore, it would complicate the build process and distribution of the plugins due to native libraries which should be included.

JavaFX was the preferred choice since it was created by Oracle as the Swing successor [7]. JavaFX provides multiple enhancements over Swing like skinning using CSS (Cascading Style Sheets) and better API. What is more, JavaFX provides a canvas which could be utilized for the visualization of a scenario. However, two issues were encountered during experiments with utilizing JavaFX. These issues are summarized in section 5.3.1.

Even though some JavaFX enhancements would simplify the development, it would complicate plugin usage for some users because of the mentioned issues. As a result, the Swing framework is employed for the graphical components. This choice brings also multiple advantages. Mainly, Swing is part of every JRE so it works on every platform, providing that Java is supported on that platform. Additionally, two of three supported IDEs use Swing for their user interface; thus, it integrates with them perfectly.

Eclipse IDE needs a special effort for the integration between SWT and Swing. It is required to use a bridge which is implemented in SWT in order to enable plugging Swing components into Eclipse IDE.

The scenario visualization is implemented using the Java 2D technology. Similarly as Swing, Java 2D is part of JFC (Java Foundation Classes). Therefore, it is part of every JRE and it works on every platform which supports Java.

5.3.1 JavaFX Issues on Linux

Firstly, applications which contain both JavaFX and SWT can hang or crash on Linux platforms. According to information in OpenJDK issue tracker [29], the issue is caused by the fact, that JavaFX internally uses GTK 2 (GIMP Toolkit) on Linux, but latest versions of SWT uses GTK 3. These two native system libraries cannot coexist at the same time in one application. This issue would cause problems with integrating the editor library into the Eclipse IDE, which uses SWT. Additionally, if the IntelliJ IDEA or NetBeans is configured to use GTK look and feel, then the same problems could arise.

Secondly, the JavaFX is not part of the OpenJDK packages in the majority of Linux distributions. That would prevent users who prefer OpenJDK over Oracle JDK to use the plugin.

5.4 High Level Architecture

The pc4ide is structured into multiple modules in order to encapsulate related logic and decrease coupling between the code in different modules. The plugin consists of following modules:

- pc4ide-core
- pc4ide-editor
- perfcake-docs
- pc4ide-testing-components
- pc4ide-intellij
- pc4ide-eclipse
- pc4ide-netbeans

Rest of this section goes through the modules and describes briefly their main functionality.

5.4.1 Pc4ide-core module

The core module contains core features of the plugin. The core module is responsible mainly for maintenance of a scenario data model, conversion between different data models, persistence of the model, execution of a scenario using PerfCake, and PerfCake component detection.

Additionally, the core module also contains unit tests for the main features of the module. The pc4ide-core module does not contain any graphics or user interaction related code. It also does not contain any IDE specific dependencies.

5.4.2 Pc4ide-editor

Similarly as the core module, the editor module is used in the plugins for all IDEs. Hence, this module is also not dependent on features of any IDE.

The editor module contains the graphical part of the plugin. The main responsibility of the editor module is to manage the graphical editor and the forms for tweaking component values and options. This module also controls user interaction and handles various actions. The pc4ide-editor module uses the core module in order to access the data model.

5.4.3 Perfcake-docs

The perfcake-docs module is responsible for parsing and persisting the PerfCake documentation from the source code. It is not part of any IDE plugin. It is used before building the plugins in order to create a file which contains descriptions of all PerfCake components and their options. The documentation is then used in the user interface to provide users with more information about a particular component.

5.4.4 Pc4ide-testing-components

The pc4ide-testing-components module contains custom implementations of some PerfCake components. This module is used for testing purposes only. For example, it implements a custom sender component.

Consequently, this sender component is used for testing of custom component loading.

5.4.5 IDE Related Modules

The pc4ide-intellij, pc4ide-eclipse, and pc4ide-netbeans modules serve as a glue between IDE-agnostic modules and a particular IDE. Their goal is to provide a thin layer which integrates the core and editor modules into a corresponding IDE. Each IDE also has its own plugin structure requirements, so these modules are also responsible for automating build process of the plugins, and bundling the compiled code in the required IDE plugin structure so that the plugins can be easily installed into the IDEs.

Pc4ide-intellij integrates pc4ide into IntelliJ IDEA. It is fully functional layer which integrates all features provided by the core and editor modules into IDEA.

The pc4ide-eclipse and pc4ide-netbeans modules serve as a proof of concept that the library can be integrated also into other IDEs. However, only part of the functionality provided by pc4ide is covered by these layers.

6 Implementation

This chapter describes the architecture of the universal plugin in more detail than chapter 5. It focuses on the main points of a design and implementation of the plugin. For each point, this chapter summarizes how the design problems have been solved as well as reasoning why each solution have been chosen.

Several figures and diagrams are used in this chapter to better illustrate various concepts. Please note that UML (Unified Modeling Language) class diagrams are simplified. For brevity, diagrams are not complete and do not contain all fields and methods. They contain only information which is necessary to explain a particular concept.

6.1 Scenario Model Management

Maintaining the scenario model is one of the most important tasks of the whole plugin. Therefore, the design of the model is very important and influences all other parts of the plugin.

PerfCake uses three different scenario models. Firstly there is a run-time model, which is used for actual scenario execution. The other two models are an XML model and a DSL model. These two models are used for scenario designing and persistence. When PerfCake is launched, these models are deserialized and converted to the run-time model.

The run-time model cannot be easily used as a model for the scenario editor since it is strongly typed. As a result, all placeholders in a scenario definition file must be resolved before the scenario is represented using the run-time model. This constraint holds for PerfCake execution because all placeholders must be known at runtime. However, placeholder values are not known at design time. Hence using the run-time model would prevent the scenario editor from using placeholders for values which are not represented using the String type (for example integer values).

Scenario models of all existing PerfCake plugins are based on the XML model. The PerfCake XML model uses POJO (Plain Old Java Object) classes as model objects. As a result, it is easy to work with the XML model. In the XML model, each component has its own Java type.

For each property of a component, there is a specific setter or getter[1] method. Additionally, some PerfCake components behavior can be modified by implementation-specific properties, which are applicable only for a particular component implementation. For this purpose, the XML model of majority of components has also a sequence of general purpose properties. In this sequence, an arbitrary property can be specified using its name and value. An impact of this general property is dependent on a particular component implementation.

However, pc4ide defines its own scenario model. Besides of setting and getting values of modeled component properties, the pc4ide model provides additional features which simplify development of other parts of the plugin.

Firstly the pc4ide model is observable. It acts as a subject in the observer design pattern[2]. This allows the editor to bind to the model dynamically and redraw the editor on a model change in a way, that the model is not dependent on the observers at all.

The second feature of the pc4ide model is that it is very generic. This allows manipulating all PerfCake components (such as Sender, Generator, Destination, etc.) in a unified way. In the pc4ide model, all components inherit from the same interface.

The third feature allows a modeled PerfCake component to provide metadata about its supported properties dynamically. Therefore, it is possible to find which properties are supported, what are their types, how many times they can appear in a scenario, and if they are mandatory.

The fourth feature of the pc4ide model allows detecting the implementation specific properties at runtime, without knowing all component implementations in advance.

The main idea behind creating an additional model representation is to be able to process the model in a generic way. For example, It allows the form panel, which displays component settings, to be implemented only once in a generic way so that one implementation is able to display an arbitrary component. Otherwise, it would be required to implement the form panel for each component separately.

1. Setter and getter methods allow to set or get a value of an object field.
2. Refer to Design Patterns [30] for more information on the observer pattern.

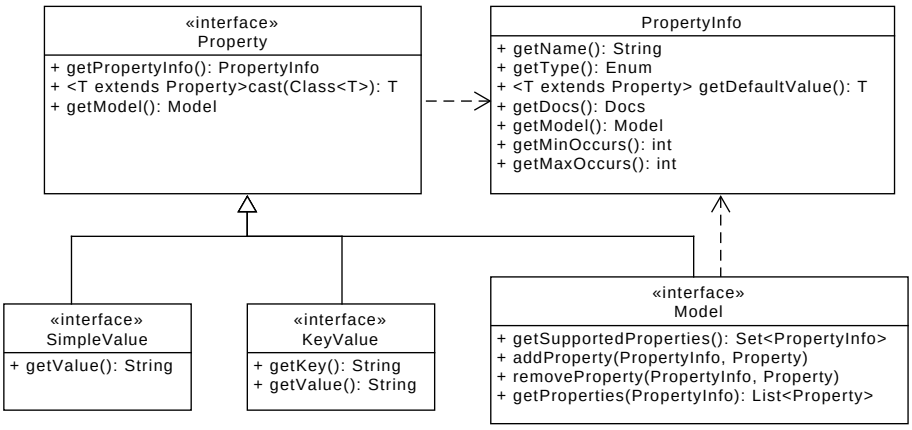

Figure 6.1: Class diagram of some model classes. For brevity, only the most important methods and fields are shown.

6.1.1 Model interfaces

Figure 6.1 illustrates the most important parts of a `Model`, `Property`, and `PropertyInfo` interfaces. Each PerfCake component is represented by the `Model` interface. Component properties are modeled using the `Property` interface. The pc4ide implementation contains three property types. There is a `Value` type which represents a property with simple value, a `KeyValue` which represents a property with two values (key and value), and last property type is the model type itself because some PerfCake components act as a property of another component.

Each Model maintains a collection of its supported properties. A supported property is represented by a `PropertyInfo` interface, which contains all required metadata about a property. For each supported property, the model also maintains a collection of actual properties defined in the model.

Not only that such a design allows working with all components uniformly, it also allows working uniformly with all components properties, no matter if a property is just a simple value or a very complex component.

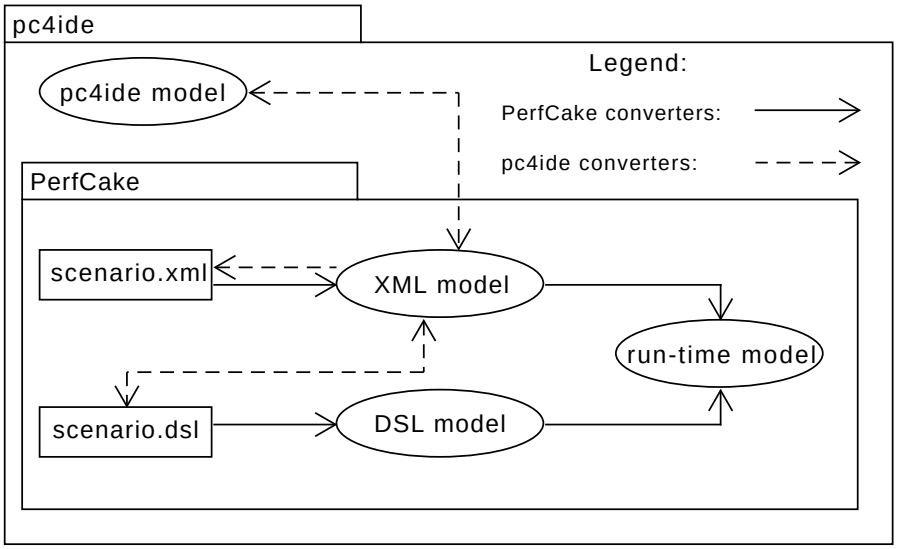

Figure 6.2: Conversions between different models.

Although the pc4ide model brings multiple advantages over the PerfCake XML model, using the separate model enforces an additional requirement. The plugin must be able to convert between the pc4ide model and the PerfCake model. For that purpose, the core module contains a converter which allows converting between the pc4ide model and the PerfCake XML model. Additionally, there is also another converter for converting the DSL model to the PerfCake XML model. The diagram 6.2 depicts relations between the models. Using a combination of the PerfCake converters and the pc4ide converters, the pc4ide model is loaded from an XML or a DSL scenario definition file and it can be consequently stored in the XML and DSL format. The serialization and deserialization of the pc4ide model are covered by classes implementing the `ScenarioManager` interface.

6.2 Scenario Execution and Debugging

This thesis requires the plugin to be able to start a PerfCake scenario directly from the scenario editor. The main design problem related to scenario execution was whether to execute PerfCake scenarios using the PerfCake API directly from the plugin as a separate thread or executing PerfCake binaries as an external process.

Both options were considered and the latter approach has been chosen. This approach does not require all PerfCake dependencies to be on the classpath, which would make the plugin unnecessarily bloated. A disadvantage of this approach is that users must install PerfCake manually and configure a PerfCake installation location in plugin settings.

There are two important classes related to Scenario execution. Firstly, there is a `PerfCakeExecutor` class. It uses the builder pattern[3]to set all scenario execution options. Then PerfCakeExecutor uses the standard Java `ProcessBuilder` class in order to start PerfCake in an external process. The second important class is `ExecutionManager`. ExecutionManager is created by the PerfCake executor when PerfCake is started. The execution manager allows controlling scenario execution. It is also able to obtain PerfCake debug information from the PerfCake debug agent through the JMX protocol[4]. ExectionManager is also observable using the observer pattern, so other interested parts of the editor receive notifications about debug values changes.

6.3 Component Detection

The plugin is able to detect PerfCake components implementations on the classpath at runtime. It uses the Reflections library [32] which is able to scan a classpath for Java classes based on their type. For IntelliJ and NetBeans plugins the Reflections library worked as is. For the OSGi environment within Eclipse, the Reflections library needed to be extended with a special Bundle URL type.

---

4. For more detail about JMX, refer to the Oracle tutorial [31]
4. Refer to Design Patterns [30] for more information on the builder and the command pattern.

6.4 Commands

Commands encapsulate an action of a scenario modification. Every action should be wrapped in a command. The `Command` interface allows to execute a command, undo a command, and determine whether a command is undoable. The commands are implemented using the command design pattern[4].

There is also a `CommandInvoker` which is used to invoke commands. The command invoker also manages a history of executed commands in a stack. As a result, it can easily undo commands. These concepts allow to implement undoing and redoing user actions easily.

6.5 Graphical Editor Structure

The graphical editor is the most important part of the pc4ide library. The editor visualizes a scenario and allows users to modify it. In order to separate concerns, several design patterns have been used in the implementation of the graphical editor.

From a high-level perspective, the graphical editor is implemented using the MVC (Model-View-Controller) class structure, which involves multiple design patterns. MVC was used in order to encapsulate separate concerns into separate classes. Figure 6.3 displays basic interaction in the MVC structure. A model, which is implemented by the pc4ide model as mentioned in section 6.1, is responsible for maintaining scenario data. Controller objects are responsible for managing user interaction and view objects. Lastly, View objects serve as a graphical representation of the model.

6.5.1 Controller

Controller objects subscribe itself as observers to the model. Controllers manage user interaction and update the view objects when the model fires a notification about a change. Controllers also handle user interaction. If a user performs an action, then a controller handles the action. Consequently, the controller also updates its model so that it corresponds to the performed action. This makes the model to notify all affected controller objects about the change. Subsequently,

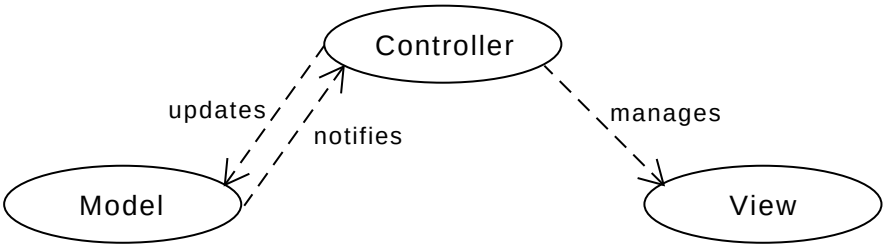

Figure 6.3: MVC interactions

the controllers update their views. and the updated visualization is displayed to the user.

Controllers are nested one into another using the composite design pattern[5] to represent a hierarchy of objects. The controller also implements the visitor design pattern, which allows performing an action on a subset of controllers easily.

6.5.2 View

The view is responsible for displaying data to users. Similarly as the controller, the view classes also implement the composite pattern. Such a design allows manipulating with multiple views, which are nested one into another, as a whole.

Drawing of a view object is implemented using Java 2D. Among other features, Java 2D allows drawing graphics primitives and texts into a Swing component[6]. View class implementations use these primitives and combine them together into more complex structures. Consequently, a PerfCake component is represented by a view class.

Figure 6.4 depicts the main concepts in the `View` and `LayoutManager` classes. The most important capability of a view object is to draw itself. However, view objects must be provided with an instance of `LayoutData` in order to know a location and a size of the drawing. The LayoutData class maintain information about positioning of a com-

5. Refer to Design Patterns [30] for more information on the composite pattern.
6. More details on Java 2D may be found in Java 2D tutorial [33].

ponent and component size. The graphical editor places components into circular sectors. Therefore, the LayoutData class consists of information about an angle (start angle and angular extent), a radius, and total dimensions of the editor. Using this information, the view objects draw themselves onto the editor surface.

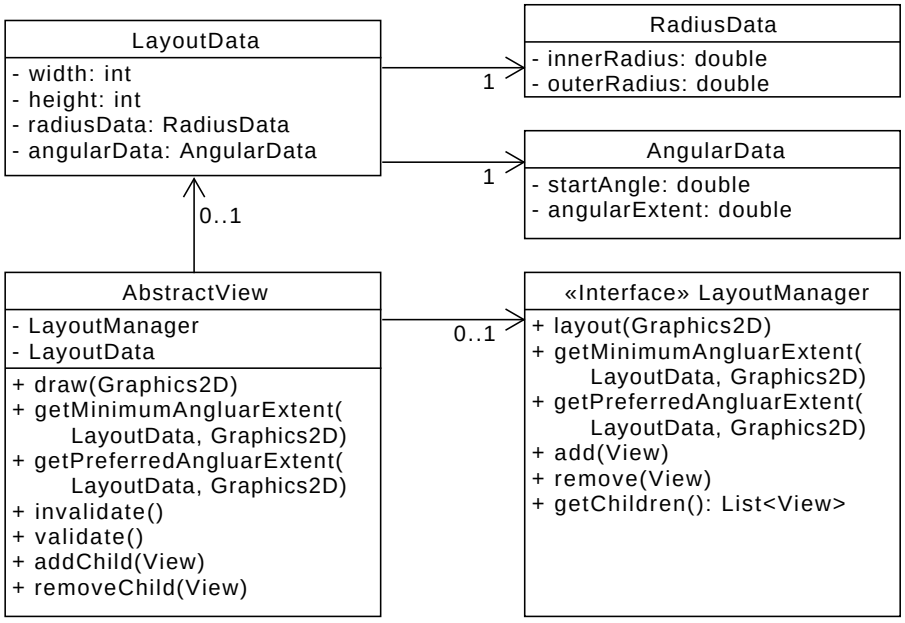

Figure 6.4: View and LayoutManager simplified class diagram.

The LayoutData object for a view is computed by its parent view. The top level view in the root of the hierarchy uses the whole editor panel as a layout data. Every view either compute the layout data for its children by itself or it can use a `LayoutManager` object which is responsible for managing and computing layout for nested views. The layout manager implementation may query children views in order to determine their preferred or minimum sizes. The preferred and minimum sizes are determined only as an angular extent because a radius is given by a window size. However, LayoutManager is allowed to ignore information from a view. The preferred and minimum sizes provided by a component view is considered only as a hint,

not as a constraint. A view object is not allowed to draw itself using a different location than provided by a LayoutData object.

When a view object changes, it may require to redraw itself. However, the change can influence its preferred or minimum size. Subsequently, the view marks itself as invalid, which is considered as a request for the parent view to validate its children. This request will propagate towards the root of the view hierarchy causing recomputing positions and sizes of the view objects and redrawing the views with proper sizes.

6.5.3 Icons

All icons use vector graphics. As a result, the icons can be resized without loosing quality. All used icons were defined in the SVG (Scalable Vector Graphics) format, even though Java 2D is not able to draw the SVG format directly. The SVG icons were converted into Java 2D using Flamingo SVG Transcoder [34]. Flamingo SVG Transcoder allows converting SVG files into Java classes by translating SVG instructions into Java 2D drawing primitives. Subsequently, it is possible to draw the pc4ide icons in arbitrary size without loosing quality.

6.6 Forms Structure

Besides the graphical editor, the form panel is very important part of the plugin. The graphical editor displays only the most important data about PerfCake components. Additional component details are displayed inside of the form, which also allows users to modify these data. A design of the form classes takes advantage of the dynamic pc4ide model and its metadata.

Form related classes design is displayed by figure 6.5. The main interface is `FormManager`, which controls what component is being displayed on the form panel. The form is allowed to have multiple pages at the same time, but always only one page is displayed. The form manager aggregates `FormController` objects which represent individual pages. FormController is responsible for maintaining the model of a component which is displayed on the page. Furthermore, FormController is able to draw all necessary details about the compo-

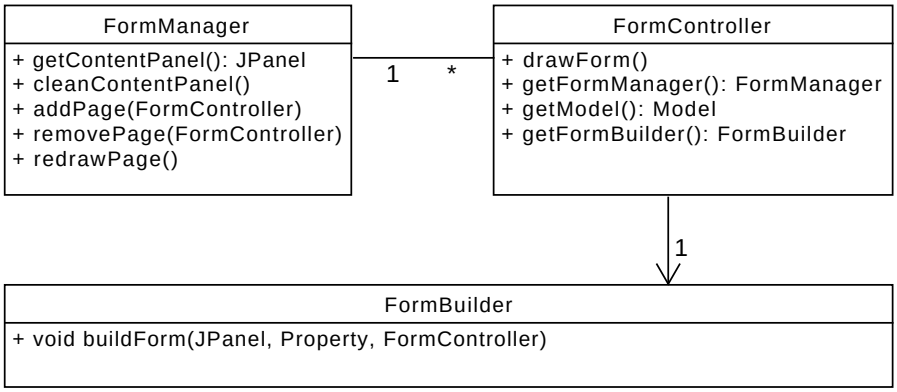

Figure 6.5: Class diagram of some form related classes. For brevity, only the most important methods are shown

nent to the form panel. Actually, all FormControllers delegate a task of drawing a form to a `FormBuilder` object. The FormBuilder class encapsulates concern of drawing a form. The FormBuilder class is able to draw a form for an arbitrary pc4ide model property [7] by obtaining metadata about components from the pc4ide model. This design allows adding new PerfCake components without modifying the code for the forms.

6.7 IntelliJ IDEA Integration

The IntelliJ integration uses implementation of the core and editor modules described in the previous chapters and binds them into IntelliJ IDEA. This involves implementing interfaces from IntelliJ IDEA API and delegating logic to the pc4ide library. The implementation classes of the IntelliJ API are then referenced in the `plugin.xml` file which defines extensions to the IntelliJ user interface. This section goes through some aspects of the integration.

7. Thus, also for the whole model.

6.7.1 Project Management

IntelliJ IDEA manages whole projects as basic units. Therefore, a new *PerfCake project* type is introduced by the plugin. This project type provides dedicated directories for PerfCake resources such as messages and extensions. Users can create the PerfCake project using a *new project* wizard. As a result, managing the PerfCake resources with PerfCake project type is the easiest way how to manage PerfCake resources. For example, if users want to use their own implementation of a component, it is enough to place the component to the lib directory in the project structure. The plugin will detect and load the implementation automatically. The file structure of the PerfCake project type is shown in figure 6.6.

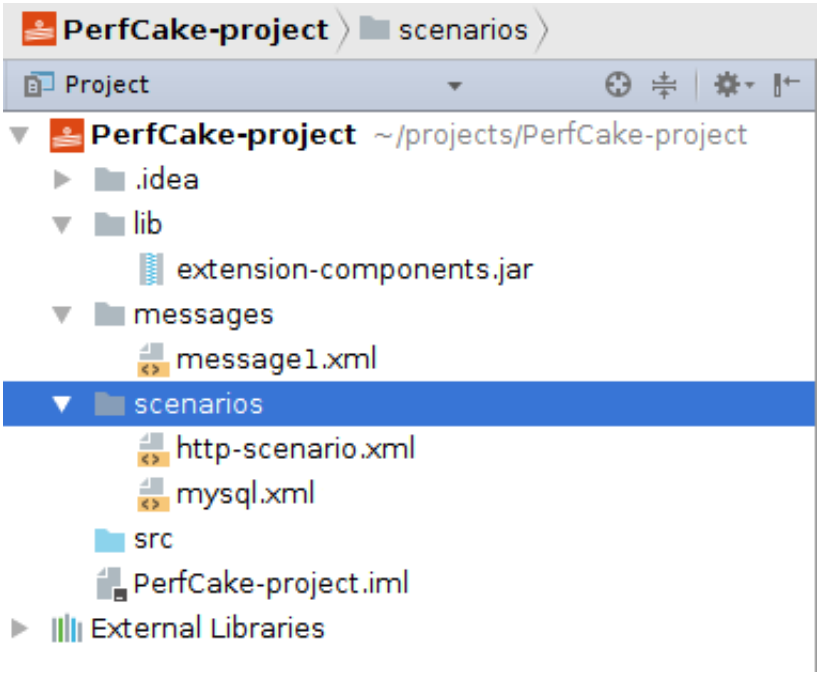

Figure 6.6: The PerfCake project structure.

On the other hand, it is not convenient to force users to use the PerfCake project type. Users are allowed to keep scenarios in an arbitrary IDEA project type such as the maven or java project type. The

only disadvantage of such an approach is that the plugin is not able to determine where the PerfCake additional resources are stored; thus, users are required to configure their location manually.

6.7.2 Editors

In order to use the scenario editor in IntelliJ IDEA, it was required to implement two classes from the IntelliJ API. Firstly, a `FileEditorProvider` which is responsible for creating an IntelliJ editor for a file. Secondly, a `FileEditor` interface has been implemented. The FileEditor interface is responsible for managing a state of the file, providing a graphical representation, and handling various events.

The only cumbersome point of the pc4ide editor integration was caused by integration of managing scenario edit history. IntelliJ IDEA provides its own mechanism for handling undo and redo actions, which is based on storing a state of an actually edited file. Consequently, the `CommandInvoker` interface from the plugin library could not be used directly, because the IntelliJ interface does not allow integrating own history management easily. To mitigate this problem, an IntelliJ IDEA specific CommandInvoker was introduced. Although this implementation satisfies the command invoker interface, it actually does not manage undo and redo command actions. This invoker is used only for executing commands. During a command execution, the invoker additionally modifies a state of the edited file which causes IntelliJ IDEA own mechanism to store the file history. This allows users to undo commands using the standard IntelliJ history management mechanism.

6.7.3 Execution

In order to be able to manage execution configurations with standard IntelliJ IDEA dialogs, several interfaces from the IntelliJ API were implemented. However, these classes do not perform any logic. They only adapt the execution classes from the pc4ide library to match the IntelliJ IDEA interface. They delegate all logic to the pc4ide-core module.

6.7.4 Problems with Plugin Library Integration

There were no major problems during the pc4ide universal editor integration into IntelliJ IDEA. Nevertheless, there were few complications which deserve attention. Firstly, IntelliJ IDEA manages the editor history using states of an edited file as described in section 6.7.2.

Secondly, the universal library uses the `Path` interface from `java.nio` introduced in Java 7 for file location representation. On the other hand, IntelliJ IDEA uses its own `VirtualFile` interface for a file representation. These two representations are not compatible. Consequently, the issue was solved by implementing a converter, which is able to convert between these two interfaces.

Thirdly, IntelliJ IDEA uses its own Swing components look and feel which causes a little different appearance of swing components. Additionally, IntelliJ provides its own Swing component implementations which visually fits into IntelliJ IDEA interface. For this reason, the editor module defines a `SwingFactory` interface, which is used through the editor in order to create swing components. This allows IDE specific plugins to create its own implementation of the swing factory which adjusts components so that they integrate properly into a user interface of an IDE.

6.8 Build Process

The pc4ide project uses Apache Maven as a build tool. The top level maven project is called pc4ide-parent. Every module, as mentioned in section 5.4, is a separate maven module. Maven is responsible for downloading all dependencies, compiling modules, and building modules. However, the IntelliJ IDEA API is not located in the maven central repository. Therefore, IntelliJ IDEA dependencies must be provided to the build process.

The build process also triggers a set of tests. These tests verify integration between pc4ide and PerfCake. Pc4ide also uses continuous integration system Travis CI [35]. Travis CI automatically builds the project on every commit to the pc4ide GitHub repository[8] as well as it executes tests. This allows detecting incompatibilities with PerfCake

8. The repository can be found at `https://github.com/PerfCake/pc4ide`

in early stages. Because of the automatic trigger, the tests are run on every change; thus it does not allow developers to forget to run the tests and fail to notice a regression.

7 Conclusion

The main purpose of this thesis is to design a user-friendly PerfCake scenario designer, implement the designer, and demonstrate its functionality by integrating the designer as the IntelliJ IDEA plugin. Additionally, the goal requires that the plugin is universal enough to be pluggable to additional IDEs.

In first two chapters, this thesis analyses the problem, describes the PerfCake framework, and outlines plugin development processes for IntelliJ IDEA, Eclipse, and NetBeans. The fourth chapter examines the existing scenario designer plugins, describes their imperfections, and proposes an improved solution. The fifth chapter is devoted the plugin design. The last chapter covers the implementation of the plugin.

The designer was implemented as a universal library called pc4ide. The pc4ide library was completely integrated as an IntelliJ IDEA plugin named pc4ide-intellij.

The scenario designer simplifies usage of the PerfCake framework. Among others, the pc4ide designer provides an easy way how to define a scenario using graphical cues, displays all PerfCake components and their options, presents descriptions of the components and their options, and allows users to test a scenario right away from an IDE. Moreover, it greatly reduces maintenance complexity in comparison with other existing plugins and makes the designer consistent between various IDEs.

The pc4ide library also contains subprojects pc4ide-eclipse and pc4ide-netbeans which demonstrate that pc4ide may be integrated also into Eclipse and NetBeans. However, these projects are not implemented completely. They are able only to display the scenario editor inside of IDEs. Hence, more integration and development of these modules is required.

Future pc4ide development goal may be to spread between users and build a community around the pc4ide designer in order to help enhance and maintain the editor so that it reflects new PerfCake and IDE versions.

Although some users prefer a local development environment, the other users might embrace a web-based designer. The pc4ide project may also serve as a template for designing a web-based scenario editor.

Bibliography

[1] PerfCake Contributors. *PerfCake: A Lightweight Performance Testing Framework*. Online. 2016. URL: https://www.perfcake.org/ (visited on 11/05/2016).

[2] PerfCake Contributors. *PerfCake Issue Tracker*. Online. 2016. URL: https://github.com/PerfCake/PerfCake/issues (visited on 11/05/2016).

[3] Martin Večeřa Pavel Macík. *PerfCake 7.x: User Guide*. URL: https://www.perfcake.org/docs/perfcake-user-guide.pdf (visited on 11/05/2016).

[4] Martin Večeřa Pavel Macík. *PerfCake 7.x: Developers' Guide*. URL: https://www.perfcake.org/docs/perfcake-developers-guide.pdf (visited on 11/05/2016).

[5] PerfCake Contributors. *PerfCake Source Code*. Online. 2016. URL: https://github.com/PerfCake/PerfCake (visited on 11/05/2016).

[6] ElasticSearch devolopers and documentators. *ElasticSearch Reference 5.0*. Online. 2016. URL: https://www.elastic.co/guide/en/elasticsearch/reference/5.0/index.html (visited on 11/05/2016).

[7] Oracle. *Java Platform, Standard Edition (Java SE) 8, Client Technologies*. Online. 2016. URL: http://docs.oracle.com/javase/8/javase-clienttechnologies.htm (visited on 11/19/2016).

[8] JetBrains s.r.o. *IntelliJ IDEA 2017.1 Help*. Online. 2017. URL: https://www.jetbrains.com/help/idea/2017.1/.

[9] Jarosław Krochmalski. *IntelliJ IDEA Essentials*. Birmingham, UK: Packt Publishing Ltd., 2014. ISBN: 978-1-78439-693-0.

[10] JetBrains s.r.o. *IntelliJ IDEA Platform SDK Documentation*. Online. 2017. URL: http://www.jetbrains.org/intellij/sdk/docs/index.html.

[11] The Apache Software Foundation. *What is Maven?* Online. URL: http://maven.apache.org/what-is-maven.html.

[12] The Eclipse Foundation. *Eclipse Documentation: Release Neon*. Online. URL: http://help.eclipse.org/neon/index.jsp.

[13] Richard Hall et al. *Osgi in Action: Creating Modular Applications in Java*. 1st. Greenwich, CT, USA: Manning Publications Co., 2011. ISBN: 1933988916, 9781933988917.

[14] Eclipse. *Tycho*. Online. URL: https://eclipse.org/tycho/ (visited on 12/03/2016).

[15] Oracle. *Modules API*. Online. 2016. URL: http://bits.netbeans.org/dev/javadoc/org-openide-modules/org/openide/modules/doc-files/api.html.

[16] Oracle Corporation. *NetBeans Platform Learning Trail*. Online. URL: http://help.eclipse.org/neon/index.jsp.

[17] StackOverflow. *Developer Survey Results*. Online. 2016. URL: http://stackoverflow.com/research/developer-survey-2016 (visited on 11/19/2016).

[18] PYPL. *Popularity of Programming Languages*. Online. 2016. URL: http://stackoverflow.com/research/developer-survey-2016 (visited on 11/19/2016).

[19] Eugen Paraschiv. *The Market Share of Java IDEs in Q2 2016*. Online. 2016. URL: http://www.baeldung.com/java-ides-2016 (visited on 11/19/2016).

[20] PerfClipse Developers. *PerfClipse Source Code Repository*. Online. 2016. URL: https://github.com/PerfCake/PerfClipse (visited on 11/17/2016).

[21] Jakub KNETL. "Editor Scénářů Projektu PerfCake pro Eclipse 4.3". [Czech]. Bachelor's thesis. Masaryk University, Faculty of Informatics, Brno, 2014. URL: http://is.muni.cz/th/396062/fi_b/.

[22] PerfCakeIDEA Contributors. *PerfCakeIDEA Source Code*. Online. 2016. URL: https://github.com/PerfCake/PerfCakeIDEA-deprecated (visited on 11/18/2016).

[23] Stanislav KALETA. "Editor Scénářů Projektu PerfCake pro IntelliJ IDEA". [Czech]. Bachelor's thesis. Masaryk University, Faculty of Informatics, Brno, 2015. URL: http://is.muni.cz/th/369525/fi_b/.

[24] Pc4idea Contributors. *Pc4idea Source code*. Online. 2016. URL: https://github.com/PerfCake/pc4idea (visited on 11/19/2016).

[25] Andrej HALAJ. "Editor Scénářů Projektu PerfCake pro NetBeans". [Czech]. Bachelor's thesis. Masaryk University, Faculty of Informatics, Brno, 2016. URL: http://is.muni.cz/th/410220/fi_b/.

[26] Wikipedia Contributors. *Škoda 120*. Online. URL: https://en.wikipedia.org/wiki/%C5%A0koda_120.

[27] Jeff Johnson. *Designing with the Mind in Mind: Simple Guide to Understanding User Interface Design Rules*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2010. ISBN: 9780123750303, 012375030X.

[28] Chris Kempson. *Base16*. Online. 2017. URL: http://chriskempson.com/projects/base16/.

[29] David Hill. *[SWT,Linux] FXCanvas will Hang or crash on startup with swt-4.4 on Linux*. Online. 2015. URL: https://bugs.openjdk.java.net/browse/JDK-8089584.

[30] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995. ISBN: 0-201-63361-2.

[31] Oracle. *The Java Tutorials: Java Management Extensions (JMX)*. Online. 2017. URL: https://docs.oracle.com/javase/tutorial/2d/index.html.

[32] Reflections Contributors. *Reflections Source Code*. Online. 2017. URL: https://github.com/ronmamo/reflections.

[33] Oracle. *The Java Tutorials: 2D Graphics*. Online. 2017. URL: https://docs.oracle.com/javase/tutorial/2d/index.html.

[34] Emmanuel Bourg, Kirill Grouchnikov. *Flamingo SVG Transcoder*. Online. 2017. URL: http://ebourg.github.io/flamingo-svg-transcoder/.

[35] Travis CI, GmbH. *Travis CI*. Online. URL: https://travis-ci.org/.

A Appendix

A.1 Scenario Examples

A.1.1 XML Scenario Example

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<scenario xmlns="urn:perfcake:scenario:7.0">
    <run type="${perfcake.run.type:time}"
         value="${perfcake.run.duration:120000}"/>
    <generator class="DefaultMessageGenerator"
               threads="${thread.count:1}"/>
    <sender class="HttpSender">
        <target>http://${server.host}/post</target>
        <property name="method" value="POST"/>
    </sender>
    <reporting>
        <reporter class="IterationsPerSecondReporter"
                  enabled="true">
            <destination class="ConsoleDestination"
                         enabled="true">
                <period type="time" value="1000"/>
            </destination>
        </reporter>
    </reporting>
    <messages>
        <message uri="file://plain.txt"/>
    </messages>
</scenario>
```

A.1.2 DSL Scenario Example

```
scenario "dsl-example.dsl"
  qsName "test" propA "hello"
  run 25000.ms
  generator "DefaultMessageGenerator" threads "10"
  sender "DummySender"
  reporter "IterationsPerSecondReporter" enabled
    destination "ConsoleDestination" every 5000.ms enabled
  message content:"Hello world" send 1.times
  validation fast disabled
    validator "RegExpValidator" id "text" pattern "Hello
       world"
end
```

A.2 Digital Attachments

As a part of this thesis, various digital resources were created. All Digital attachments are stored in the thesis archive of Masaryk University.

A.2.1 Pc4ide Source Code

The pc4ide source code is located in the `pc4ide-sources.zip` file. For convenience, the sources are also located in the pc4ide GitHub repository[1]. The repository contains also a description of steps required for the source compilation.

A.2.2 Pc4ide IntelliJ IDEA Plugin

The pc4ide plugin for IntelliJ IDEA built from the sources is located in the `pc4ide-intellij-plugin.zip` file. The file can be also obtained from the GitHub repository. The repository also contains steps required for installation of the plugin.

1. The repository can be found at `https://github.com/PerfCake/pc4ide`