

DANMARKS TEKNISKE UNIVERSITET

System for development of functional accept-test specifications

Svante T. H. Jørgensen
Kongens Lyngby 2012

IMM-B.Eng-2012-13

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk
IMM-B.Eng-2012-13

1 Foreword

I would like to thank associate professor at DTU Edward A. Todirica for his support and good advice during this project. If not for his willingness to take extras time to understand this obscure tool called Cucumber and what I wanted to do with it, I am sure my project would have been much less successful.

2 Summary

In this project I develop tools to help people write software tests more efficiently.

More specifically I will analyze the current problems with writing tests for the Cucumber test tool and design and implement solutions to those problems. The main focus will be on automating the steps where most time can be saved with least effort.

The result of the project is a plug-in for the Eclipse Integrated Development Environment (Eclipse IDE) that assists users in writing Cucumber tests.

3 Intended audience

This report is intended for developers who want to use the Cucumber test tool more efficiently and want to learn ways of making tools for this purpose in the Eclipse IDE.

The reader is expected to have some experience with development and test of non-trivial computer systems and the general knowledge that this implies. Knowledge of Eclipse and Java is helpful but not required.

Because the Cucumber tool is relatively obscure and not yet used very widely in the industry, I will go into some details about how it works before I introduce my solutions.

4 Report organization and conventions

This report is divided in four main chapters.

Introduction chapter

Context of the problems and the Cucumber tool is introduced and explained.

Objectives chapter

Problems and how they should be solved is discussed.

Design chapter

Design of the solution is explained.

Implementation chapter

Implementation is presented and discussed.

In the end I will document the tests of my solution and reflect on its usefulness and future potential.

4.1 External reference convention

When external sources are referenced, they are marked with a superscript number corresponding to their number in the References chapter.

Example: I used a book named Eclipse Plug-ins¹.

The above example means that “Eclipse Plug-ins” is listed as number 1 in the References Chapter.

5 Table of Contents

1	Foreword	ii
2	Summary.....	iii
3	Intended audience.....	iv
4	Report organization and conventions	iv
4.1	External reference convention	iv
6	Introduction.....	1
6.1	Scope	2
6.2	Cucumber and the Gherkin Language	2
7	Objectives	4
7.1	Gherkin syntax highlighting (The Obscurity Problem).....	4
7.2	Context Sensitive Assistance (The Commonality Problem).....	5
7.2.1	Overview of the process	5
7.3	Template System (The Multiplicity Problem).....	6
7.3.1	Creating a Template	6
7.3.2	The Step Def string	7
7.3.3	Using a Template	8
8	Design	10
8.1	Terminology.....	10
8.2	System overview.....	10
8.3	Gherkin syntax highlighting	11
8.4	Context Sensitive Assistance	11
8.4.1	Levenshtein Distance design	13
8.5	Template System	14
8.5.1	Create Template	16
8.5.2	Insert from Template.....	18
9	Implementation.....	21
9.1	Terminology.....	21
9.2	Tools and frameworks	21
9.2.1	Eclipse Software Development Kit	21
9.2.2	Eclipse Integrated Development Environment	22
9.3	Creating a Text Editor plug-in in Eclipse	22
9.4	Gherkin syntax highlighting	22

9.5	Context Sensitive Assistance	23
9.5.1	The Levenshtein distance Algorithm	24
9.6	Template System	25
9.6.1	Create Template system.....	26
9.6.2	Insert from Template system	28
10	Test	31
10.1	Test methodology.....	31
10.2	Test Results.....	31
11	Result analysis	33
11.1	Gherkin Syntax Highlighting	33
11.2	Context Sensitive Assistance	34
11.3	Template system	34
12	Discussion of future improvements	36
13	Conclusion	36
14	References	37
15	Table of Figures	38
16	Appendix 1: GherkinEdit Installation instructions.....	i
17	Appendix 2: User Manual	ii
17.1	Setting up an Eclipse project to use GherkinEdit	ii
17.2	Using Syntax highlighting.....	iii
17.3	Using Context Sensitive Assistance	iv
17.4	Using the Template system	vi
17.4.1	Creating a Template	vi
17.4.2	Inserting from a Template	vi

6 Introduction

A common software development challenge is how to make sure you build the product that your customer want, and not just what he says he wants. Furthermore, it is usually difficult and/or very time consuming to make sure that completed features do not break as a result of further development on new features or changes to old ones.

Without a clear picture of what you need to build and a good architecture for your system there is little hope of meeting these challenges. However, there are ways to make sure that you find and correct misunderstandings and errors as soon as possible.

In this area agile software development methods have been very successful. Developing in short cycles with tests and live demos at the end of each cycle will, in the worst case, quickly let you know that there are problems with the software for whatever reason, or in the best case give you confidence and assurance that what you are building is correct.

The faster you find your errors the less time will have been spent building on those errors, and you quickly realize that to get the most out of this approach, you have to test and demo as frequently and broadly as possible while using as little time as possible doing it.

A good approach for saving time on test, is test automation. Test automation tools are basically programs that assist in developing computer-executable tests that would otherwise be done by a human tester or user. All automatic tests have in common that they take longer to build than manual tests. But after they have been built they can be executed again and again, often in a fraction of the time required by a human.

One of these tools is called Cucumber. It's premier advantage over other test automation tools is that the steps executed in a test is defined in a natural language with very few grammatical rules on top of the common language (English or other national languages). This language is called Gherkin. The natural language serves to narrow the gap between the technologists understanding of what the system under development is supposed to do and the understanding of the non-technologists, who just want the system to "work". One of the problems with Cucumber is that there are few tools to help with writing Gherkin and lots of time is wasted doing steps that could be done automatically.

While the Cucumber test automation tool offers good functionality and great potential, it is in its early development, and is still very much a developer's tool. Not much thought has been put in how to help the customer or non-technical specification professionals write these test definitions.

So what are the steps that could be done automatically, and what would a tool look like that automated these tasks? That is what this project will try to answer.

6.1 Scope

This project will only look at Cucumber on the Java platform in the form of the Cucumber-Java library. The focus will be on the process of writing Gherkin sentences and not on implementing them in test code, running the tests, or all the automatic things that the Cucumber tool does to tie these things together behind the scenes.

6.2 Cucumber and the Gherkin Language

The Cucumber test framework, as illustrated in Figure 1, consists of the system under test, the test code, and the Gherkin sentences. Cucumbers job is basically to read the Gherkin sentences and execute the matching test code. The test code will in turn perform the tests on the system under test.

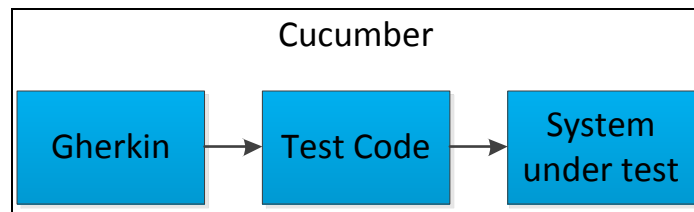


Figure 1 - Cucumber system overview

The idea behind Gherkin is to use natural language to formulate tests or functional specifications, which in turn can be linked to machine-executable code representing the tests. English is the main language, but many other natural languages are supported.

In Gherkin, a test or functional specification is called a "Scenario" (see Figure 2, point 2). Each Scenario consists of zero or more sentence describing the starting conditions (called a "Given sentence" – see Figure 2, point 3), at least one sentence describing an action (called a "When sentence" see Figure 2, point 4) and at least one sentence describing the desired outcome of the action (called a "Then sentence" – see Figure 2, point 5).

```
1 Feature: As a system administrator,  
   when I log in to the system, the system log is displayed.  
2 Scenario:  
3 Given I am at the log in screen  
4 When I log in as a system administrator  
5 Then the status of the system is displayed
```

Figure 2 - Gherkin Example

The grammatical rules are the keywords "Feature:", "Scenario:", "Given", "When" and "Then". The text after the Feature keyword (see Figure 2, point 1) is a short description, only meant for human understanding and is thus free form. The text after the Given, When and Then keywords are also free form text, but are captured and matched against pieces of test code which will actually do what the text describes.

On top of the basic syntax there are also so-called “Tags” which have the function of declaring features or scenarios so they can be referenced later. This is very useful when the number of tests grow and you wish run only a subset of the tests. A Tag is defined as starting with a @-sign and ending with a space or line break (whichever comes first). An example could look like this:

```
@VeryNiceTag
```

Furthermore there are Comments, which will be ignored by the text matcher. Comments start with a #-sign and ends with a line break. An example could look like this:

```
#This is a comment
```

Each “Given”, “When” and “Then” sentence must have a matching piece of code to actually do what they describe. This is done with methods with annotations called “step definitions”. An example can be seen in Figure 3. Each method has an annotation (beginning with a @-sign) which contains a regular expression which in turn is matched with the Gherkin sentences in the feature. If they match the code in the method is run. Each matche

```
import cucumber.annotation.en.*;

public class GherkinSteps2 {
    @Given("^I am at the log in screen$")
    public void goToTheLogInScreen() {
        // Navigate to login screen
    }

    @When("^I log in as a system administrator$")
    public void logInAsSystemAdministrator() {
        // Log in with System Administrator credentials
    }

    @Then("^the status of the system is displayed$")
    public void checkStatusDisplayed() {
        // Find the text "System Status: "
    }
}
```

Figure 3 - Example of step definitions

The point of Gherkin, which creates a layer of abstraction over the code that actually executes the test steps, is to have a common definition of system features between customer and developer. It is important because it provides a common language in which both developer and customer can express their expectations of what the system should do.

One other neat thing this makes possible is to use the test definitions as documentation and accept-test of the system. When the developer and customer have to agree if the system actually does what it is supposed to, these test definitions can be used to impartially judge if it is the case.

7 Objectives

In this chapter I will try to identify the most critical problems with writing Gherkin features in terms of how much time and energy is used on things that could be automated.

The Obscurity Problem

When writing long features, it can be hard to get a good overview of the feature structure and much time can be wasted on reading and formatting the text to get a good overview. An easy way to identify specific parts of the feature is critical in any Gherkin editor.

The Commonality Problem

Many steps are similar, like “Given I am at the log in screen”, “Given I am at the settings screen” and “Given I am at the help screen”, where the corresponding code to execute that step also often will be very similar and could be generalized. There is currently no tool for this kind of reuse in any Gherkin-capable editors.

The Multiplicity Problem

Every unique step (Given ..., When ..., Then ...) in the test definitions require a set of machine instructions for it to be executed. This should encourage reuse of steps, as to minimize the amount of new code required, but there is currently no software which supports the writer of Gherkin features in this, other than to look through the code base.

The solution

So how could these problems be solved, or at least mitigated? I propose a new editor for writing Cucumber features called “GherkinEdit” with the following features.

7.1 Gherkin syntax highlighting (The Obscurity Problem)

To give the user of the editor a better overview of a features structure, the keywords should be highlighted with colors to visually separate them from the other text.

The colors chosen are strong primary colors (red, green, blue) to give the best possible contrast.

Highlighting the example from Figure 2 - Gherkin Example, it would look something like this:

```
#Remember to write good comments
```

```
@TaggedFeature
```

```
Feature: As a system administrator, when I log in to the system, the system log is displayed.
```

```
Scenario:
```

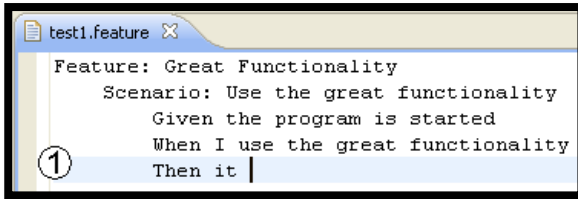
```
    Given I am at the log in screen  
    When I log in as a system administrator  
    Then the status of the system is displayed
```

7.2 Context Sensitive Assistance (The Commonality Problem)

To help the user reuse and remember Gherkin sentences that has already been used before and implemented as test code, GherkinEdit should provide Context Sensitive Assistance. The Context Sensitive Assistance function is activated by writing the beginning of a sentence and pressing the Ctrl+Space keys. A list with similar sentences that has already been implemented in code appears, and the similar sentences can be selected and automatically inserted in the document to finish the sentence the user was writing.

7.2.1 Overview of the process

1. The user writes the beginning of a Gherkin sentence



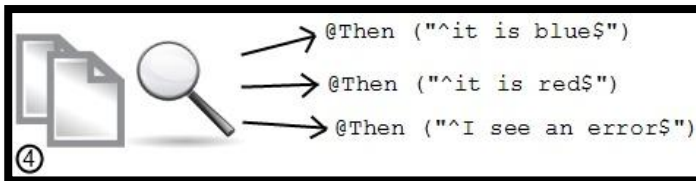
2. The user presses a key combination that activates the Context Sensitive Assistant



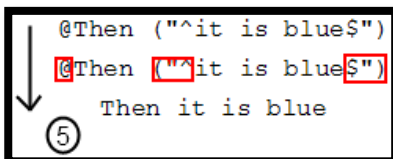
3. The Context Sensitive Assistant captures the written sentence before the cursor



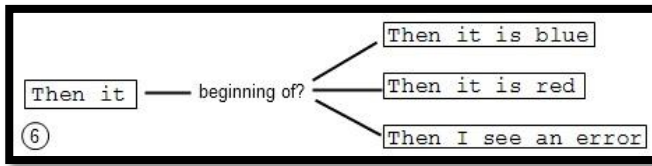
4. All source files in the project is searched and all Cucumber annotations are captured



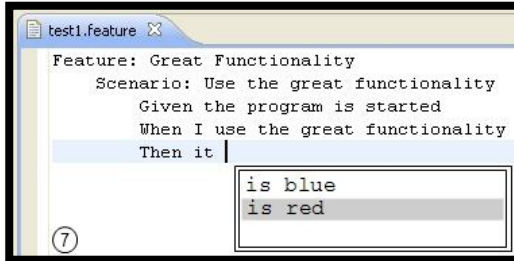
5. The Cucumber annotations have their special syntax stripped so they look like normal Gherkin sentences



6. Each Cucumber annotation is compared with the written sentence to see if the written sentence matches the beginning of the Cucumber annotation



7. If they match, the Cucumber annotation is displayed as a suggestion to complete the written sentence.



8. If none of the scanned Cucumber annotations starts with the written sentence, then all annotations are suggested to the user – but sorted according to their Levenshtein distance to the written sentence. (see Levenshtein Distance design chapter)

7.3 Template System (The Multiplicity Problem)

When writing Cucumber features, step definitions can be written very narrowly and only support a specific step in a feature. But this is very time consuming when writing many step definitions, and steps will often be very similar to previously written steps – for example “When I press the ‘Home’ button” and “When I press the ‘Next’ button”. This makes it an advantage to make the step definitions more general and one step definition could likely cover all forms of button pressing.

This is all well and good, but when the steps and the step definitions are written by different people, as is often the case, it can be very difficult to keep track of what kind of steps are already supported by step definitions.

In this project I propose a new system to solve this problem which I call the “Template System”. The Template system is a way for the person writing the step definition (the step definition writer) to tell the person writing the Cucumber feature (the feature writer), which general step definitions he has written, and how to use/reuse them in different contexts. The idea is that the step definition writer creates Template, which is a kind of skeleton for a feature sentence, and the feature writer fills in the blanks to create a full feature sentence which is guaranteed to be supported in test code. This will save a lot of time for the feature writer, since he does not have to look through all the test code to find the exact sentences that are supported. This would otherwise be necessary even if he has a very good idea of what he wants to express, because Cucumber matches character by character.

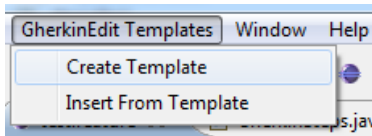
7.3.1 Creating a Template

To create a template, the system will go through two general steps:

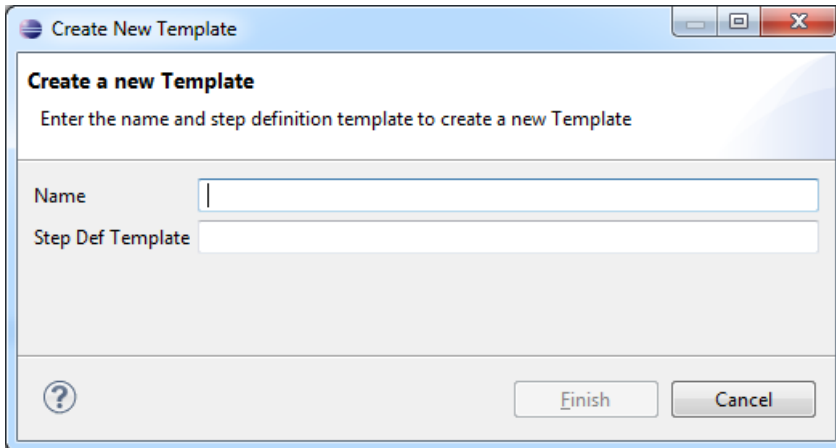
1. Gather information from the user about what the step definition supports
2. Store the information in a format so it can be used to help another user create steps supported by the step definition.

The process

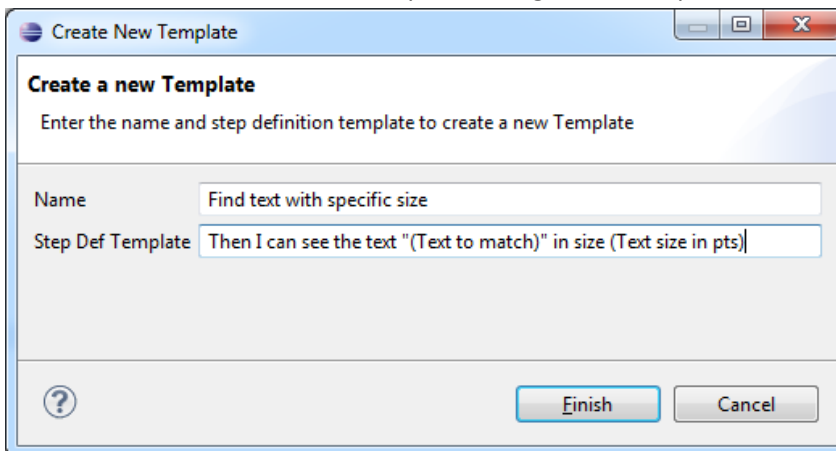
1. The user selects the "Create Template" command:



2. The user is presented with the "Create Template" dialog:



3. The user enters the name and Step Def string of the template:



4. The information is converted to XML and saved as a file in the workspace.

7.3.2 The Step Def string

The template Step Def string can be thought of as a text with a series of blocks which can be either text or input. The text blocks provide context and information so you know what is going to happen, and the input blocks gives the template flexibility in what data it is going to work on.

As an example take the following Step Def:

Then I can see the text "(Text to match)" in size (Text size in pts)

The string contains two text blocks, **Then I can see the text "** and **" in size.** They tell us that the test is going to test if it can find a specific text with a specific size, and the test will be positive if it finds it.

The string also contains two input blocks marked with parentheses, **(Text to match)** and **(Text size in pts)**. They tell us that the test wants a text to search for and the size of the text it should search for.

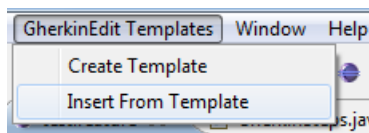
7.3.3 Using a Template

To use a template, the system will go through four general steps:

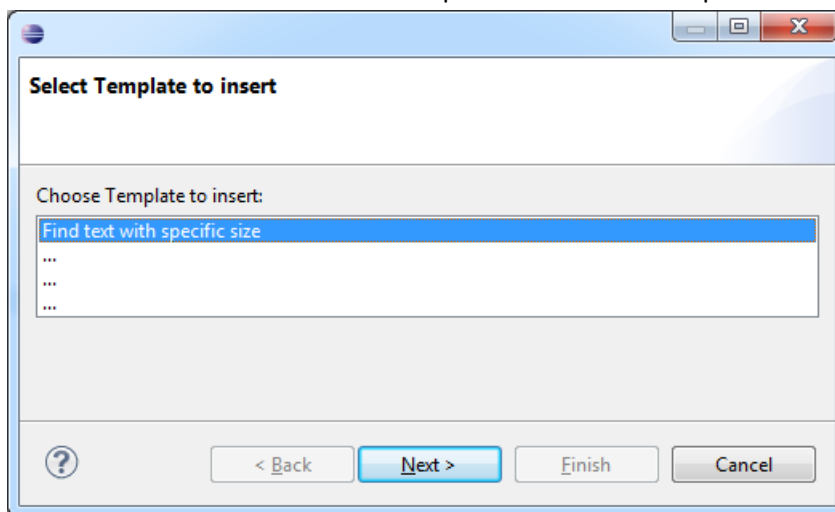
1. Present the user with a list of available templates and ask him which one he wants to use
2. Retrieve the information on how to use the supported step definition
3. Guide the user in using the step definition as it is intended
4. Write the resulting step to the feature

The process

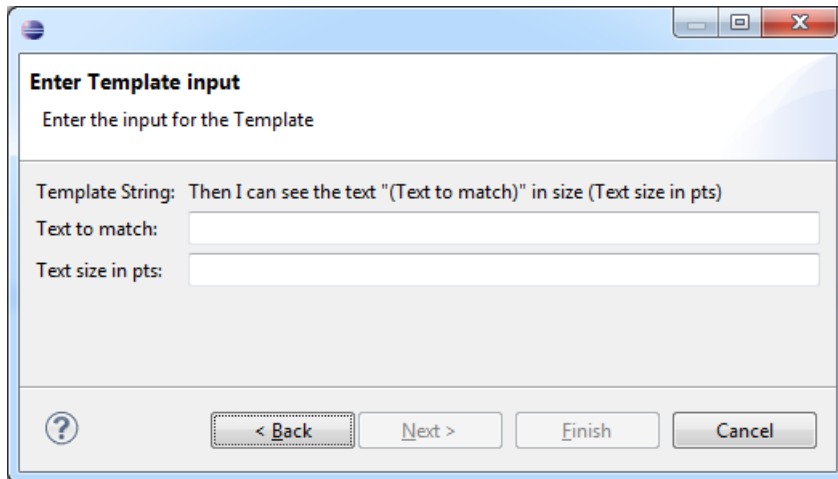
1. The user selects the “Insert From Template” command:



2. The editor finds all the available templates and the user is presented with the list of templates:



3. The selected template is parsed by the system and presented as an input dialog for the user:



Enter Template input
Enter the input for the Template

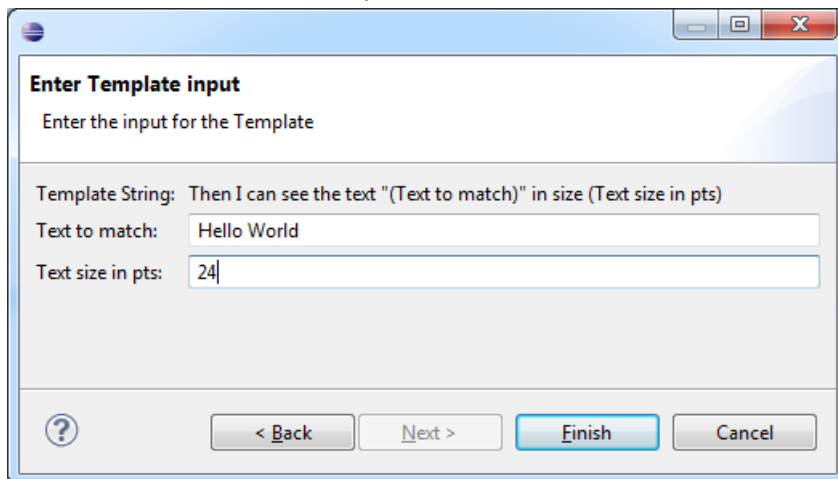
Template String: Then I can see the text "(Text to match)" in size (Text size in pts)

Text to match:

Text size in pts:

? < Back Next > Finish Cancel

4. The user enters the desired input:



Enter Template input
Enter the input for the Template

Template String: Then I can see the text "(Text to match)" in size (Text size in pts)

Text to match:

Text size in pts:

? < Back Next > Finish Cancel

5. The system writes the resulting step in the editor:

```
Then I can see the text "Hello World" in size 24
```


8 Design

This chapter will describe the design choices in building the GherkinEdit editor and the alternatives if applicable.

8.1 Terminology

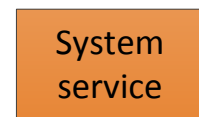
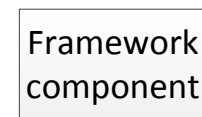
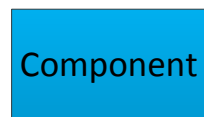
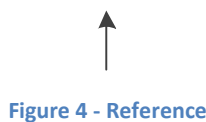
In this chapter several symbols and colors are used to represent parts of the system and relationships between them.

A black arrow means a reference. See Figure 4. The component at the beginning of the arrow uses and/or depends on the component at the end of the arrow.

A box with a blue background is a component of the feature being described. See Figure 5.

A box with a white background is a component independent of the feature being described. Typically a Framework component. See Figure 6.

A box with an orange background is a system service, such as File System or Network Adapter. See Figure 7.



8.2 System overview

One of the first design choices was to base the editor on the Eclipse Integrated Development Environment (Eclipse IDE). Eclipse IDE is specialized in tools for code development, and it has a plug-in framework which is very modular and encourages extensions and reuse of functionality. It provides a basic text editor with all the features you would expect from a Notepad-style program, such as text editing, file saving and loading, and undo/redo functionality. These features are essential for most text editors and provide a great platform to build on.

To keep the complexity as low as possible, the proposed features are split up into three independent parts. But since I don't want users to install a lot of different plug-ins they are wrapped in the GherkinEdit Plug-in Package, which will let the features be installed as one plugin, as shown in Figure 8.

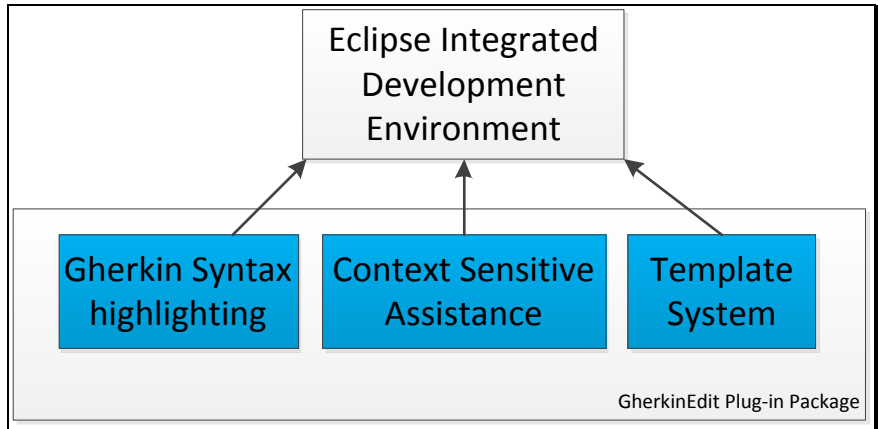


Figure 8 - GherkinEdit system overview

8.3 Gherkin syntax highlighting

The purpose of the syntax highlighting function is to make it easier for a user to read and write Gherkin features. With syntax highlighting it is easier to spot keywords, separate sentences and recognize text with special meanings such as Comments and Tags.

The syntax highlighting function consists of three main parts; Document Reconciler, Document Damager and Document Repairer. See Figure 9.

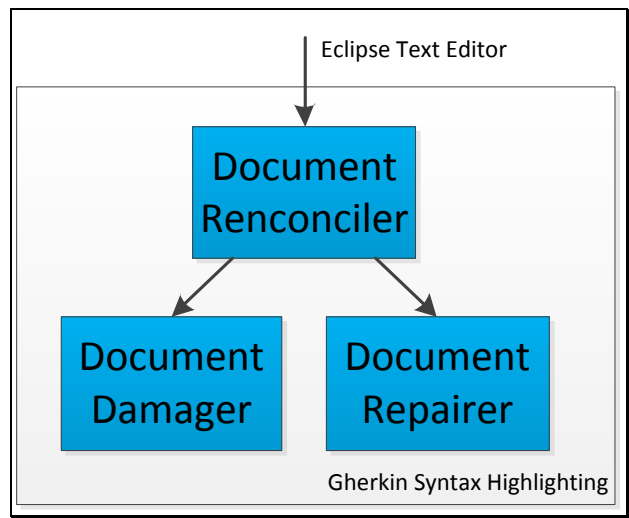


Figure 9 - Gherkin Syntax Highlighting

Every time the user changes the document (the feature being written in the editor), the Document Reconciler is called. The Document Reconciler first asks the Document Damager which parts of the document have changed so much that it needs to be repaired. Then it executes the Document Repairer on the “damaged” parts of the document. The Document Repairer recognizes the patterns in the changed parts that need to be painted and paint them in specific colors.

8.4 Context Sensitive Assistance

The purpose of the Context Sensitive Assistance feature is to make it easier to find Gherkin sentences that are already supported by test code. Normally the person writing the Gherkin sentences would have to look

through the test code base or talk to the developer every time he wants to write a sentence and be sure it is already supported. The Context Sensitive Assistance feature tried to make this process significantly easier by searching through the test code base, and matching the users sentence to sentences supported in the test code base.

The Context Sensitive Assistance feature consists of 3 main components. See Figure 10.

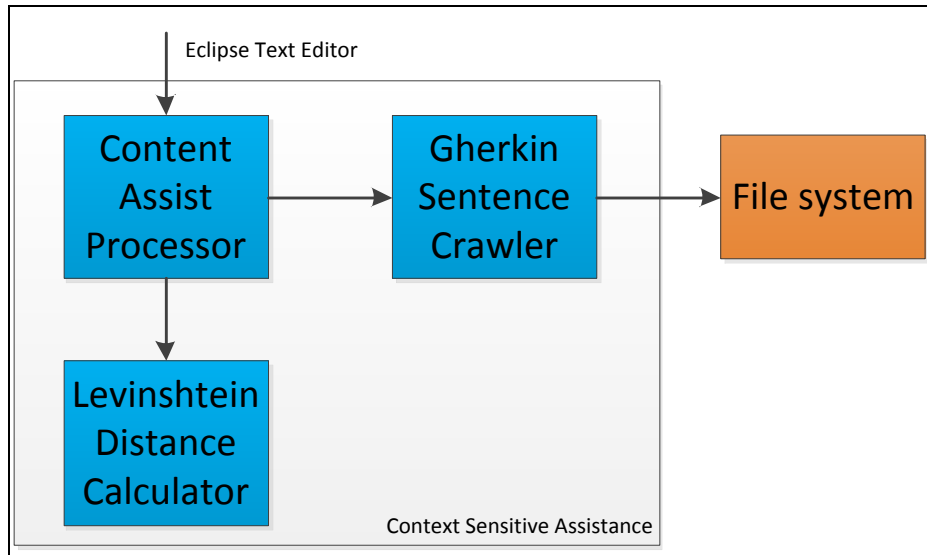


Figure 10 - Content Sensitive Assistance

When the user activates the Context Sensitive Assistance function, the line marked by the cursor is captured (see Figure 11) and sent to the Content Assist Processor.



Figure 11 - Text capture

First the Content Assist Processor requests all implemented Gherkin sentences from the Gherkin Sentence Crawler. The Gherkin Sentence Crawler requests the file system for all the .java files in the project directory tree, and captures all the Cucumber annotations. See Figure 12.

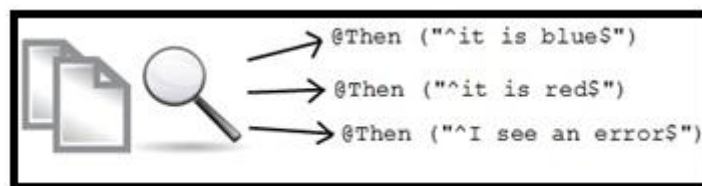


Figure 12 - Finding Cucumber annotations

Next, the Gherkin Sentence Crawler strips the annotation syntax from the annotations, and it returns the list of implemented Gherkin sentences to the Content Assist Processor. See Figure 13.

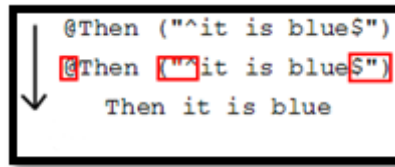


Figure 13 - Removing annotation syntax

With the list of supported Gherkin sentences, the Content Assist Processor now filters the list so only the sentences which starts with the String that was sent to the Content Assist Processor is left. If there are any left after the filtering, the list is returned to the user to choose from, and the chosen sentence is inserted in the document.

However, if none of the supported Gherkin sentences starts with the users String, the supported Gherkin sentences are then sorted by their Levenshtein Distance to the users String, and then returned to the user to choose from. For more on this, see the “Levenshtein Distance design” chapter below.

8.4.1 Levenshtein Distance design

The Levenshtein distance between two strings is equal to the lowest number of character insertions, deletions and substitutions it takes to make one string match the other.

For example, the distance between “dogs” and “dung” is 3, because to get from “dogs” to “dung” using character delete, insert or substitute, I have to use 3 steps:

0. original	dogs
1. substitute the “o” for a “u”	dugs
2. insert a “n” after the “u”	dungs
3. delete the “s”	dung

The Levenshtein distance is a good measure of similarity between strings and the algorithm can be run relatively quickly on even a large collection of strings.

The context assist uses the Levenshtein distance algorithm if no direct matches can be found between the written sentence and the Cucumber annotations in the project.

1. Each annotation has its Levenshtein distance to the written sentence calculated
2. All the annotations gets sorted according to their Levenshtein distance (ascending, so highest similarity first)
3. The list is presented as suggestions to the user

8.5 Template System

The purpose of the Template System is to let the test code developer easily describe how to use the test code, and for the Gherkin sentence writer to take advantage of this when writing sentences.

When writing test code to support Gherkin sentences, it is often an advantage to generalize the step definitions. Take the following Gherkin sentence as example:

“Then I see the text ‘Hello World’ in size 16”

Now, the fast and easy way to support this in test code would be to write:

```
@Then("^I see the text 'Hello World' in size 16$")
public void findHelloWorldInSize16() {
    // Find 'Hello World' in size 16
}
```

But what if I change my mind and want to see the text in size 20? Or if I want to see the text ‘Hello Denmark’ instead? With this way of doing it I end up with a lot of test code.

Luckily the Gherkin annotation supports regular expression capturing. This is automatically done when the regular expression contains a so called “group” which is marked with parentheses and another regular expression. This makes the following test code possible, which can handle any text in any size:

```
@Then("^I see the text '([a-z]*)' in size ([0-9]*)$")
public void findTextInSize(String text, String Size) {
    // Find String text in size String size
}
```

The `([a-z]*)` capture group means “capture all letters here from a to z”, and the `([0-9]*)` capture group means “capture all numbers here from 0 to 9”.

Now the test code can handle finding any combinations of text and size, but how does the person writing the Gherkin sentences find this information? Normally he would have to look through the code base and read and understand all the regular expressions. In the Template system the person writing the test code can now create a template representing the general test code and guide the person writing Gherkin sentences in using it.

The Template System consists of 2 main parts. The Create Template feature and the Insert From Template feature. See Figure 14.

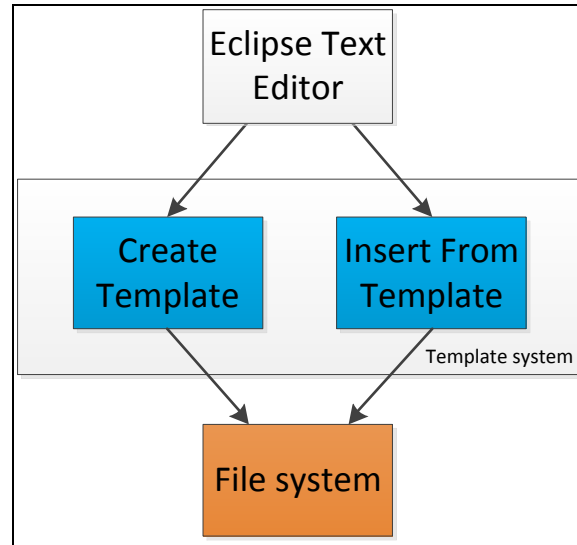


Figure 14 - Template system overview

The only place where the Create Template and Insert From Template features overlap is in the design of the data structure used to save and load Templates.

The special data structure is needed to save the Templates persistently beyond the scope of runtime. Templates are saved in files with the ending `.getemplate` (short for “GherkinEdit Template”).

The content of the file is plaintext XML in the form shown in Figure 15.

```

<template>
  <name>Find text with size</name>
  <templateString>Then I can see the text &quot;(text to match)&quot; in size (size to match).</templateString>
</template>
  
```

Figure 15 - GherkinEdit Template Example

The XML have one main element called “template”, which contains two elements “name” and “templateString”. The “template” element is a container which marks the start and end of the template. The “name” element is a short description of the template, and the “templateString” element is the definition of the template.

If we take a look at the template string in Figure 15, four parts can be identified. The first part “Then I can see the text ";” is a static text that cannot be changed by the user of the template. The next part “(text to match)” is an input field which the users of the template have to fill with the desired input. It can be identified by the parentheses. The text inside the parentheses is used as a description of the input the user is expected to input as shown in Figure 16 - Input prompt when using Template.

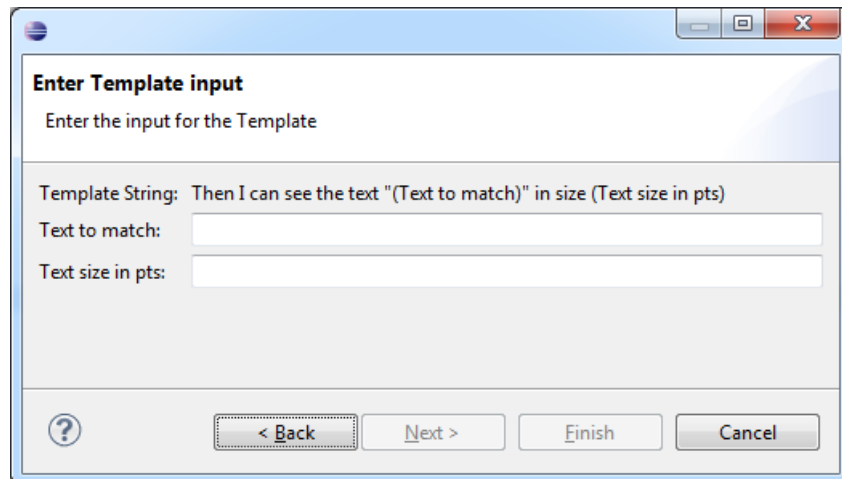


Figure 16 - Input prompt when using Template

The parentheses also result in the limitation that static text cannot contain parentheses. This type of marking is chosen because it is already a limitation in Gherkin sentences not using the template system, where parentheses have to be escaped with the backslash character. This is because it is a reserved character in regular expressions used for grouping as mentioned earlier.

The reason for choosing XML is that it is a well-known standard with many free tools and libraries to read and write it, which makes it easier to implement, but also easier for Cucumber tool developers to understand, interface with or adopt. It is also very easy to extend in case future templates need more (or less) data to define them.

8.5.1 Create Template

The purpose of the Create Template system is to let test code developers express and communicate how their test code can and should be used, and as a result save time for the person writing the Gherkin features.

The Create Template system consists of a Command Handler, Template, XML Handler, Wizard and Page for the Wizard. See Figure 17 - Create Template System.

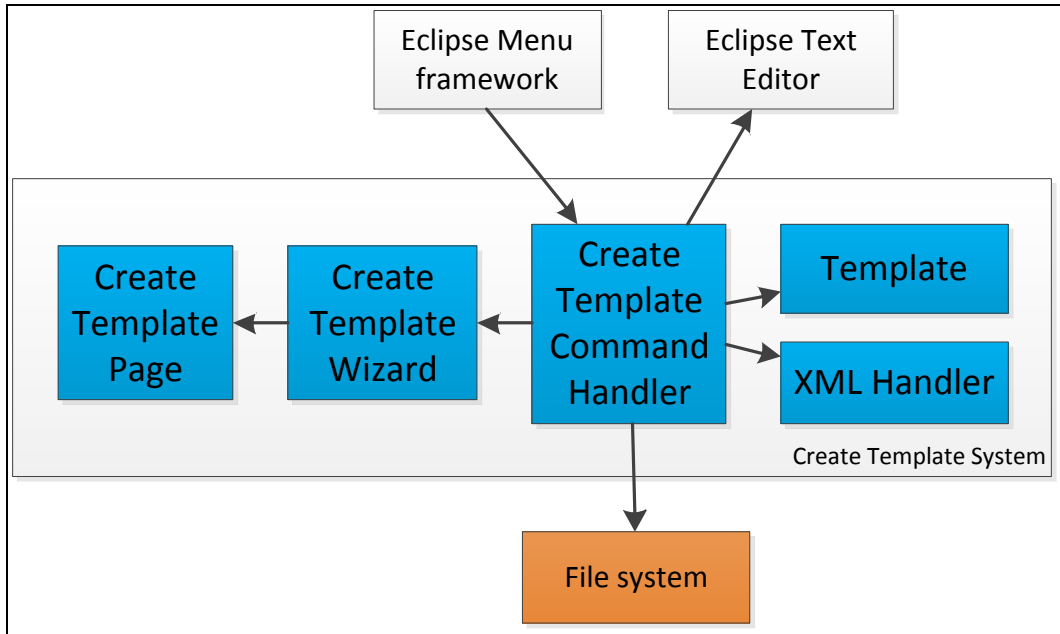


Figure 17 - Create Template System

The core of the Create Template system is the Command Handler. The Command Handler is activated (and registered on) the Eclipse Menu framework. This creates a drop-down menu in Eclipse with the item “Create Template” in it as illustrated in Figure 18 - Create Template Eclipse menu item.

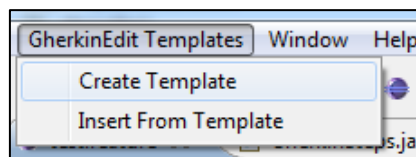


Figure 18 - Create Template Eclipse menu item

When the Create template menu item is activated, the Create Template Command Handler is called. First it calls the Create Template Wizard, which in turn displays the Create Template Page as illustrated in Figure 19.

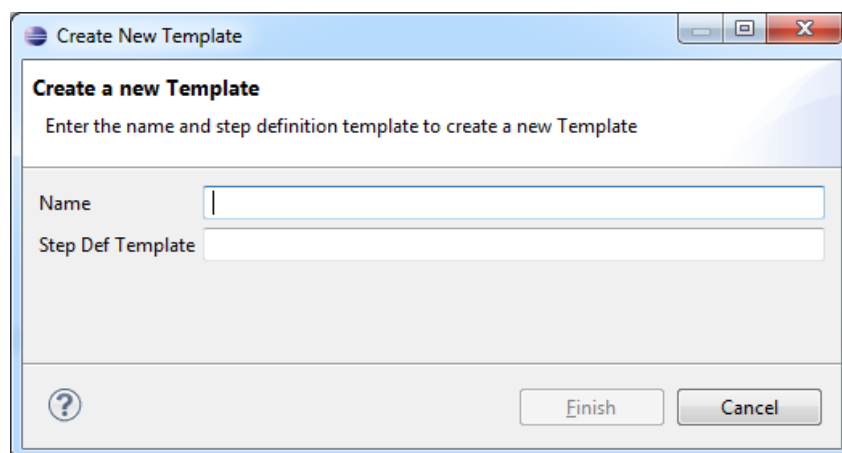


Figure 19 - Create Template Page GUI

After capturing the Name and Step Definition of the new Template a Template object is created with the data and the XML handler is used to convert the object into a XML representation as illustrated in Figure 15. As the last step the XML String is saved as a file in the file system in a designated directory in the Eclipse project root directory.

8.5.2 Insert from Template

The purpose of the Insert from Template System is to help the user use an existing Template to create a Gherkin sentence that is guaranteed to be supported by the test code and used as the test code developer intended.

The Insert from Template System is designed very similarly to the Create Template System. See Figure 20.

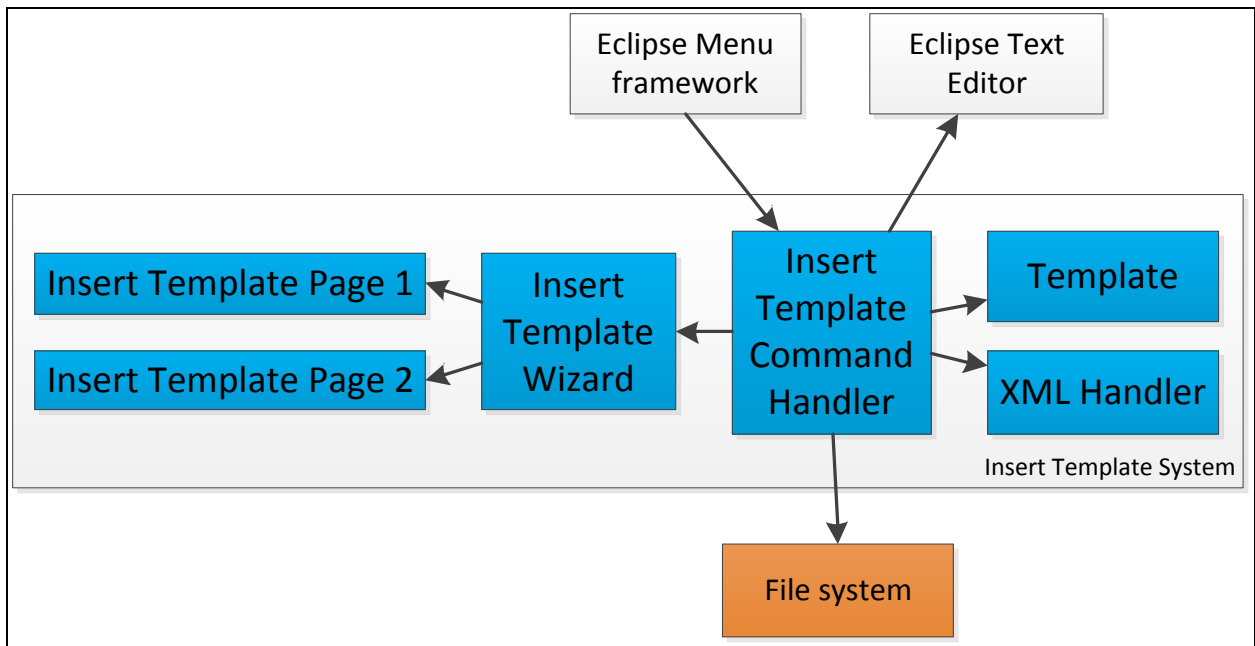


Figure 20 - Insert from Template System

When the Insert from Template menu item is activated, the Command Handler reads all the Template files in the designated Template directory in the Eclipse projects root directory. All the XML-encoded templates are parsed by the XMLhandler and a Template object is created for each stored Template. This list is then sent to the Insert Template Wizard which displays the Insert Template Page 1. Here the user is asked to choose the Template to be inserted in the current document in the Eclipse Text Editor. See Figure 21.

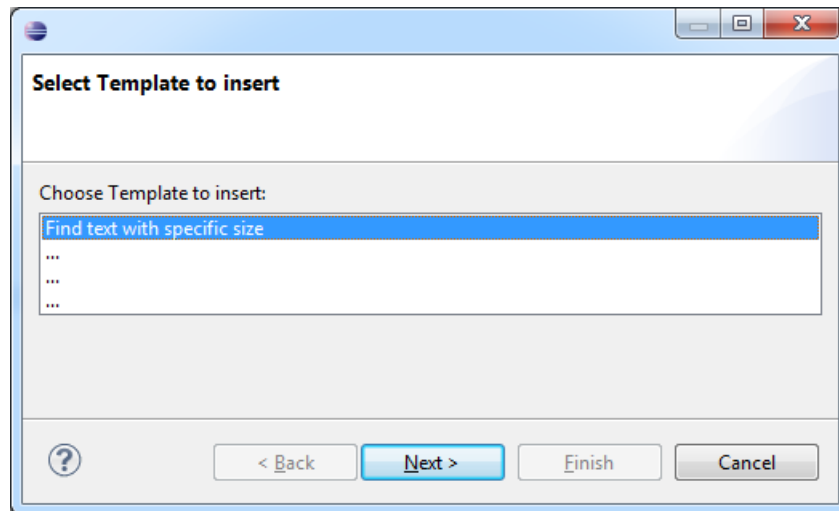


Figure 21 - Selecting Template to be inserted

When the user has selected the template, page 2 of the Wizard is shown with the details of the template and input fields required for the given template. See Figure 22.

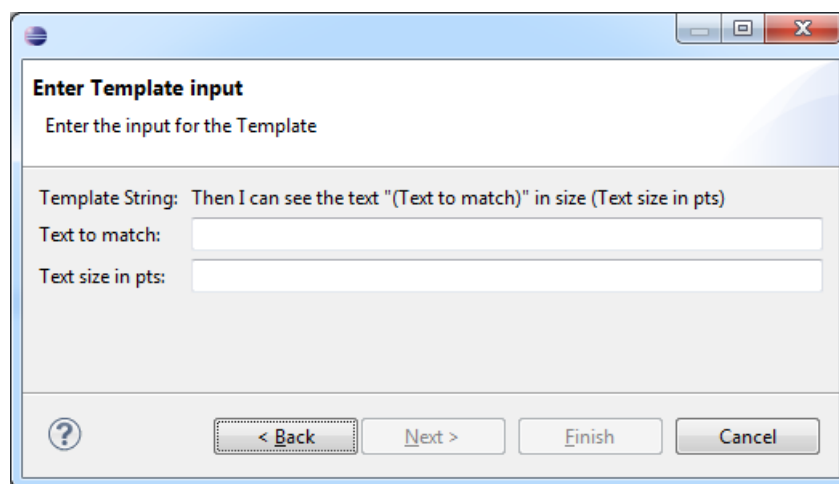


Figure 22 - Prompting user for template input

Finally the Command Handler receives the user input and chosen template, compiles the resulting Gherkin sentence, and inserts the sentence in the document in the Eclipse Text Editor. The Gherkin sentence is now inserted in the editor as illustrated in Figure 23.

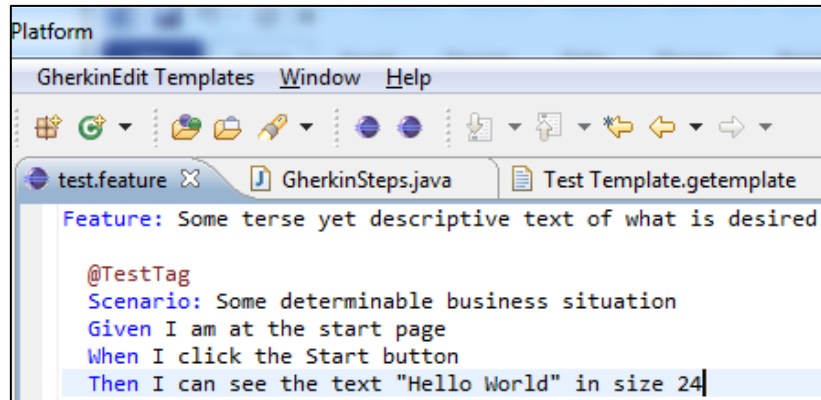


Figure 23 - Final result of inserting a template

9 Implementation

This chapter will describe the choices made in implementing the solution, and the core steps to build it.

9.1 Terminology

In this chapter several symbols and colors are used to represent parts of the system and relationships between them.

A white arrow means an inheritance. See Figure 24. The class at the beginning of the arrow is a child of the class at the end of the arrow.

A black arrow means a reference. See Figure 25. The object at the beginning of the arrow uses and/or depends on the object at the end of the arrow.

A box with a blue background is a class in the part of the program being described. See Figure 26.

A box with a white background is a class independent of the feature being described. Typically a Framework Component. See Figure 27.

A box with an orange background is a system service, such as File System or Network Adapter. See Figure 28.



Figure 24 - Inheritance



Figure 25 - Reference

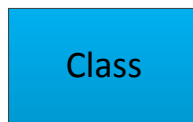


Figure 26 - class

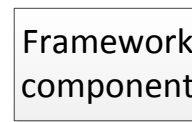


Figure 27 - Framework component



Figure 28 - System service

9.2 Tools and frameworks

In this chapter I introduce the Tools and frameworks I used for the implementation of the GherkinEdit plugin.

9.2.1 Eclipse Software Development Kit

When it comes to building a specialized text editor, few frameworks look as promising as the Eclipse SDK. The whole Eclipse software suite is extremely modular and extendable, and with software development in mind Eclipse SDK provides a lot of help with common text editor features such as copy/paste, undo/redo, opening and saving files and so forth.

The code that reads and manipulates text is completely open and extendable, and the extensions can easily be packaged as a plugin that other people can install in their Eclipse installation, or with a bit more effort packaged as a stand-alone application (a so-called Rich Client).

9.2.2 Eclipse Integrated Development Environment

In theory it is possible to develop Eclipse Plug-ins in any editor you can write Java in, but almost all Eclipse plug-ins are written in Eclipse since it provides a host of features that makes development easier. So in practice it does not make much sense to develop Eclipse plug-ins in anything else than Eclipse.

All my experience with Cucumber has been from command line or Eclipse. Eclipse is open source, free and multiplatform compatible. But best of all I know that Eclipse plays well with Cucumber and I see it as a big advantage to be able to write Cucumber features and the definitions that run them in the same development environment.

9.3 Creating a Text Editor plug-in in Eclipse

To create the base of the Text Editor plug-in I followed the excellent tutorial “Building an Eclipse Text Editor with JFace Text”⁴, and read “Chapter 2. A Simple Plugin Example”¹ from the book Eclipse Plug-ins by Eric Clayberg for a more thorough understanding of the basics.

After creating the basic Text Editor plug-in, which has functionality comparable to Notepad (edit text, undo/redo and save/load files in the file system), working with the Eclipse Framework is mostly a matter of finding which class provides the functionality you want to tap into, extend the class and override the correct methods. Finding these methods and figure out how to change them to get the desired effect proved quite time consuming and gave a steep learning curve. But after figuring it out it was a pleasure to work with.

9.4 Gherkin syntax highlighting

The Gherkin syntax highlighting feature consists of four classes (blue) as illustrated in Figure 29.

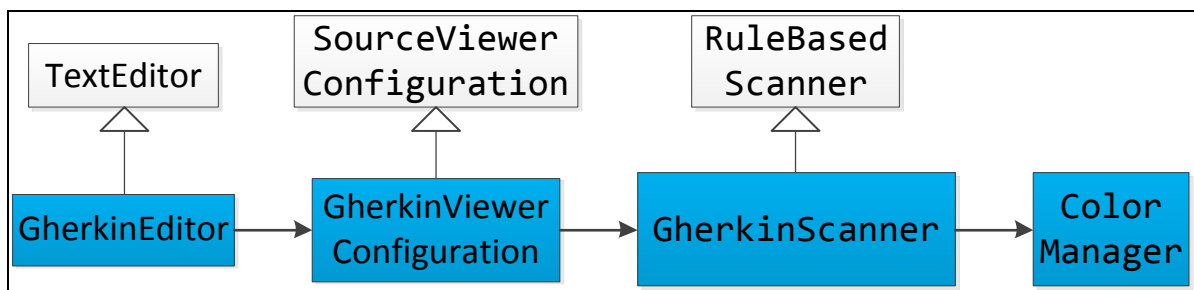


Figure 29 - Implementation of Syntax highlighting

To use the Eclipse TextEditor the class GherkinEditor is created and declared as extending TextEditor. In GherkinEditor the GherkinViewerConfiguration class is set as the new SourceViewerConfiguration. This is done with the following command in the GherkinEdit Constructor:

```
setSourceViewerConfiguration(new GherkinViewerConfiguration(colorManager));
```

The GherkinViewerConfiguration instructs the GherkinEditor to use the GherkinScanner which in turn applies the rules that Gherkin keywords, comments and tags is painted in their respective colors, with the following code:

```

6 public class GherkinScanner extends RuleBasedScanner {
7
8     public GherkinScanner(ColorManager manager) {
9         IToken comment = new Token(new TextAttribute(manager.getColor(IGherkinColorConstants.COMMENT)));
10        IToken tag = new Token(new TextAttribute(manager.getColor(IGherkinColorConstants.TAG)));
11        IToken keyword = new Token(new TextAttribute(manager.getColor(IGherkinColorConstants.KEYWORD)));
12
13        IRule[] rules = new IRule[10];
14
15        //Add Comment rule
16        rules[0] = new EndOfLineRule("#", comment);
17        //Add Tag rule
18        rules[1] = new SingleLineRule("@", null, tag);
19        //Add Feature rule
20        rules[2] = new SingleLineRule("Feature:", " ", keyword);
21        //Add Scenario
22        rules[3] = new SingleLineRule("Scenario:", " ", keyword);
23        //Add Given rule
24        rules[4] = new SingleLineRule("Given", " ", keyword);
25        //Add When rule
26        rules[5] = new SingleLineRule("When", " ", keyword);
27        //Add Then rule
28        rules[6] = new SingleLineRule("Then", " ", keyword);
29        //Add And rule
30        rules[7] = new SingleLineRule("And", " ", keyword);
31        //Add But rule
32        rules[8] = new SingleLineRule("But", " ", keyword);
33        // Add generic whitespace rule
34        rules[9] = new WhitespaceRule(new GherkinWhitespaceDetector());
35
36        setRules(rules);
37    }

```

The RuleBasedScanner requires the rules to implement IRule. Instead of implementing it from scratch the EndOfLineRule, SingleLineRule and WhitespaceRule classes are used. The EndOfLineRule is a rule that applies for a specific pattern (here a "#") and everything following it until the next line delimiter. The SingleLineRule is a rule that matches a String beginning with the first parameter and ends with the second parameter or a line delimiter, whichever comes first.

To generate the color objects, GherkinScanner uses the ColorManager.

9.5 Context Sensitive Assistance

As with the syntax highlighting, plugging in the Context Sensitive Assistance is also a matter of overriding existing functionality. The Eclipse TextEditor component already supports content assistance, it is just not enabled by default. See Figure 30.

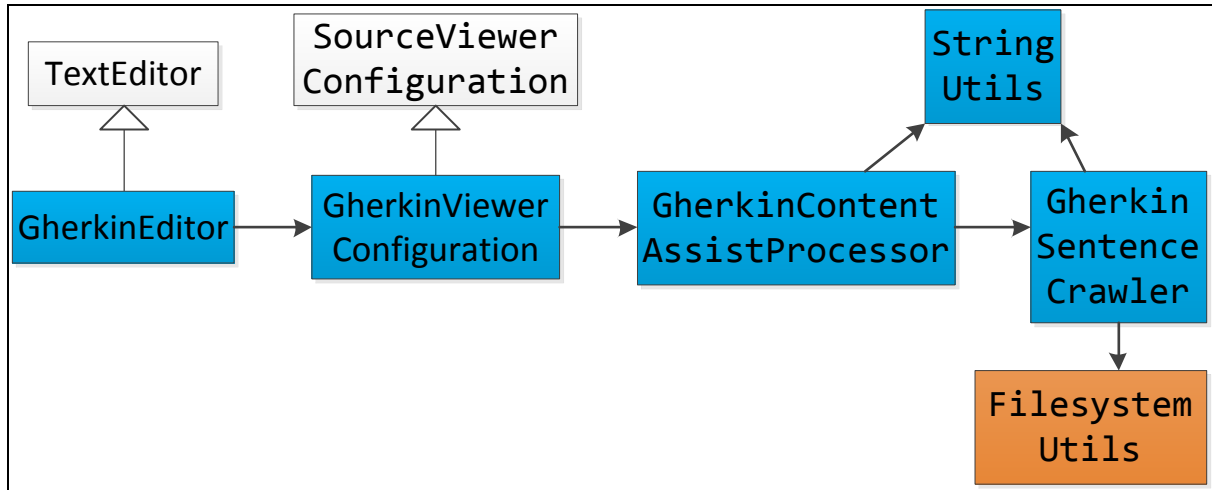


Figure 30 - Context Sensitive Assistance implementation

The GherkinContentAssistProcessor is registered with the GherkinViewerConfiguraion which in turn is registered with the GherkinEditor. This results in the GherkinContentAssistProcessor being asked for a list of suggestions whenever the user presses the hotkey for content assistance (Ctrl+Space).

The GherkinContentAssistProcessor then asks the GherkinSentenceCrawler for a list of the supported Gherkin sentences. First the GherkinSentenceCrawler uses the FileSystemUtils to read all java files in the Eclipse project directory and collects all the test code annotations and return them to the GherkinContentAssistProcessor.

Now, the GherkinContentAssistProcessor uses the StringUtils to strip the annotations of the annotation syntax and are left with the pure Gherkin sentences. If the users String sent to the GherkinContentAssistProcessor is the beginning of any of the collected Gherkin sentences, all sentences where this is true will be sent back to the GherkinEditor and presented as suggestions.

However, if none of the Gherkin sentences starts with the users String, each Gherkin sentence is compared with the users String and given a score of how similar it is. This score is calculated with the Levenshtein distance Algorithm which is described in the chapter below.

Finally the Gherkin sentences are returned as suggestions sorted by their similarity to the users String.

9.5.1 The Levenshtein distance Algorithm

To let the Context Sensitive Suggestion feature find the Cucumber annotations similarity to the written sentence, this project implements the Levenshtein distance algorithm. This chapter will explain how it is done.

Let's take "dogs" and "dung" as an example.

		d	o	g	s
d					
u					
n					
g					

1. The two strings' characters are put in a matrix(i,j) with one string horizontally in the top and the other vertically to the left, both with 2 spaces from (0,0).

		d	o	g	s
	0	1	2	3	4
d	1				
u	2				
n	3				
g	4				

2. The characters' position in the string (one-based) is written under the first string and right of the other.
3. For each remaining cell, calculate the cost as: if the character above the cell and the character to the left of the cell are the same, then the cost is 0. Otherwise it is 1.
4. Calculate the value of the cell as the minimum of
 - a. The above cell-value plus 1 ($matrix(i-1,j)+1$)
 - b. The left cell-value plus 1 ($matrix(i,j-1)+1$)
 - c. The cell-value above and to the left plus the cost ($matrix(i-1,j-1)+cost$)

		d	o	g	s
	0	1	2	3	4
d	1	0	1	2	3
u	2	1	1	2	3
n	3	2	2	2	3
g	4	3	3	3	3

5. After all the cells have been calculated (step 3 and 4 repeated), the result will be in the lower right corner cell.

9.6 Template System

The Template system consists of two parts. The Create Template function and the Insert from Template function. These can be accessed from the Eclipse menu bar as illustrated in Figure 31.

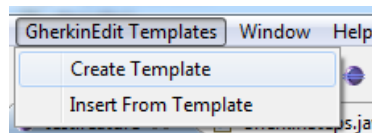


Figure 31 - Accessing the Template system functions

To insert menu's and menu items in the Eclipse menu bar, the menu and menu items are registered in the plugin's plugin.xml file. The plugin.xml file is a special file with, among other things, information on which

extension points the plug-in plugs in to and commands registered for the plug-in. Commands are an Eclipse Framework concept which is used to let plug-ins interface with other plug-ins.

To add the “GherkinEdit Templates” menu to the menu bar, the following XML is inserted in plugin.xml:

```
44 <extension
45     point="org.eclipse.ui.menus">
46     <menuContribution
47         locationURI="menu:org.eclipse.ui.main.menu?after=additions">
48         <menu
49             id="testtest.menus.TemplateMenu"
50             label="GherkinEdit Templates"
51             mnemonic="M">
52             <command
53                 commandId="gherkinedit.commands.CreateTemplateCommand"
54                 id="gherkinedit.menus.CreateTemplateCommand"
55                 mnemonic="C">
56             </command>
57             <command
58                 commandId="gherkinedit.commands.InsertFromTemplateCommand"
59                 id="gherkinedit.menus.InsertFromTemplateCommand"
60                 mnemonic="I">
61             </command>
62         </menu>
63     </menuContribution>
64 </extension>
```

Line 45 defines the extension point, which is the Eclipse menu bar. Line 49-50 inserts the “GherkinEdit Templates” menu in that extension point, and lines 53-54 and 58-59 inserts the “Create Template” and “Insert from Template” menu items in the menu as well as defines the command ID the action should trigger.

9.6.1 Create Template system

When the Create Template system is activated through the Eclipse menu item, the CreateTemplateCommandHandler is called. See Figure 32.

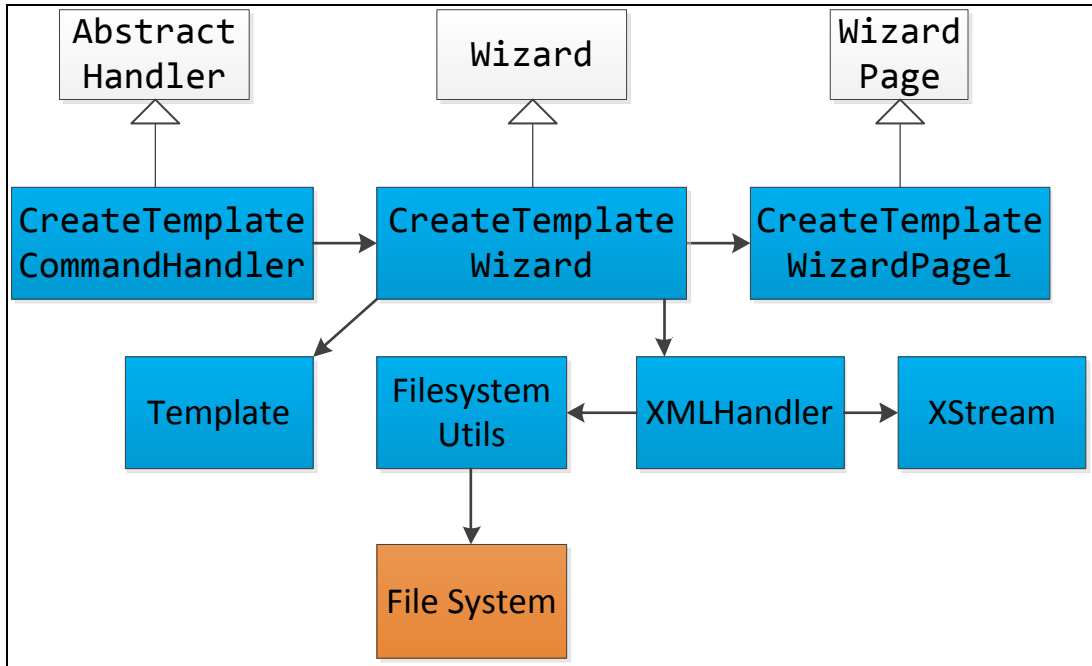


Figure 32 - Create Template implementation

The CreateTemplateCommandHandler then calls the CreateTemplateWizard. The Wizard framework is part of the Eclipse Framework and makes it easier to create GUI's in a wizard form. A wizard is a popular graphical linear dialog concept where the user can move back and forward between pages, will be presented with information and asked to provide input. Figure 33 illustrates the Create Template Wizard.

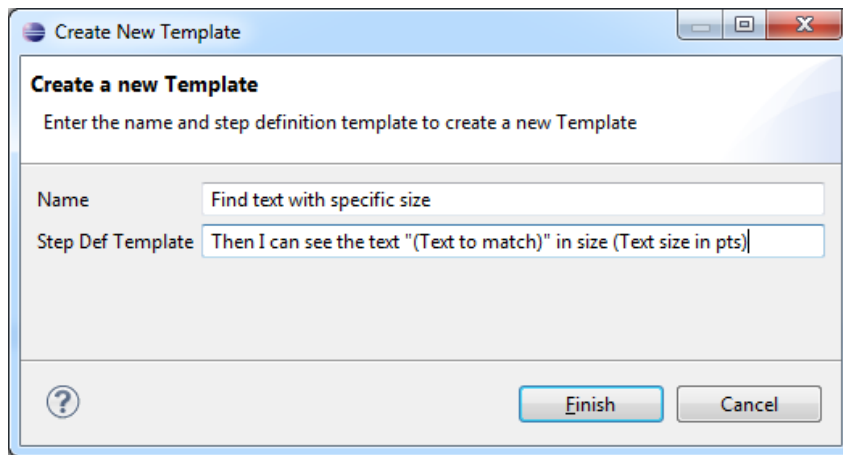


Figure 33 - Create Template Wizard

This particular Wizard only has one page, but if there were more pages a “Back” and a “Next” button would be visible left of the “Finish” button.

The Create Template Wizard is created from CreateTemplateCommandHandler with the following code:

```

IWorkbenchWindow window = HandlerUtil.getActiveWorkbenchWindowChecked(event);
WizardDialog wizardDialog = new WizardDialog(window.getShell(), new CreateTemplateWizard());
wizardDialog.open();
    
```

The CreateTemplateWizard then adds the Wizard page with the following code:

```
@Override
public void addPages() {
    page1 = new CreateTemplateWizardPage1();
    addPage(page1);
}
```

The CreateTemplateWizardPage1 class' job is to create the labels and input fields for the wizard page that captures the input of the user required for creating the Template.

When the user presses "Finish", the text in the input fields are sent back to the CreateTemplateWizard class which first creates a Template object and then sends it to the XMLHandler to save it to a file.

The XMLHandler uses the XStream⁵ library to convert Java objects to XML code, and the FileSystemUtils class to save it to a file:

```
public static void saveTemplate(Template template) {
    XStream xstream = new XStream();
    xstream.alias("template", Template.class);
    String xml = xstream.toXML(template);

    String relativePath = "\\GherkinEditTemplates\\";
    String filename = template.getName() + ".getemplate";
    FileSystemUtils.saveStringAsFileInProject(relativePath, filename, xml);
}
```

The resulting file looks something like this:

```
<template>
  <name>Find text with size</name>
  <templateString>Then I can see the text &quot;(text to match)&quot; in size (size to match).</templateString>
</template>
```

9.6.2 Insert from Template system

The objective of the Insert from Template system is to let the user use Templates to create Gherkin sentences quickly and efficiently, with the guarantee that the sentences are supported by the test code.

The implementation of the Insert from Template system is quite similar to the Create Template system. See Figure 34.

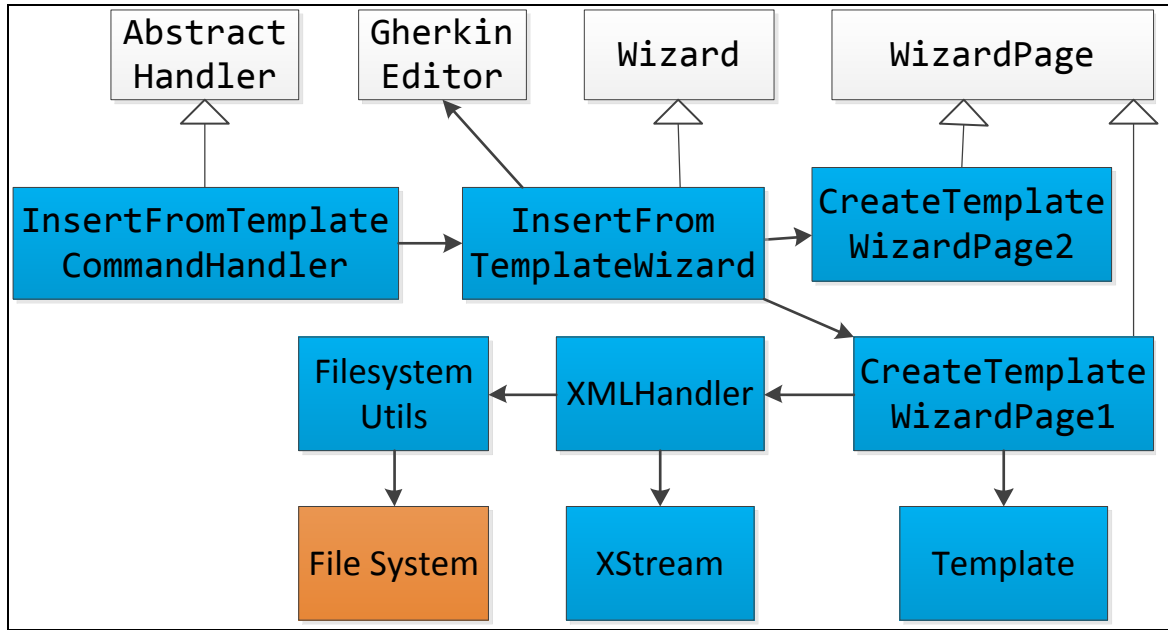
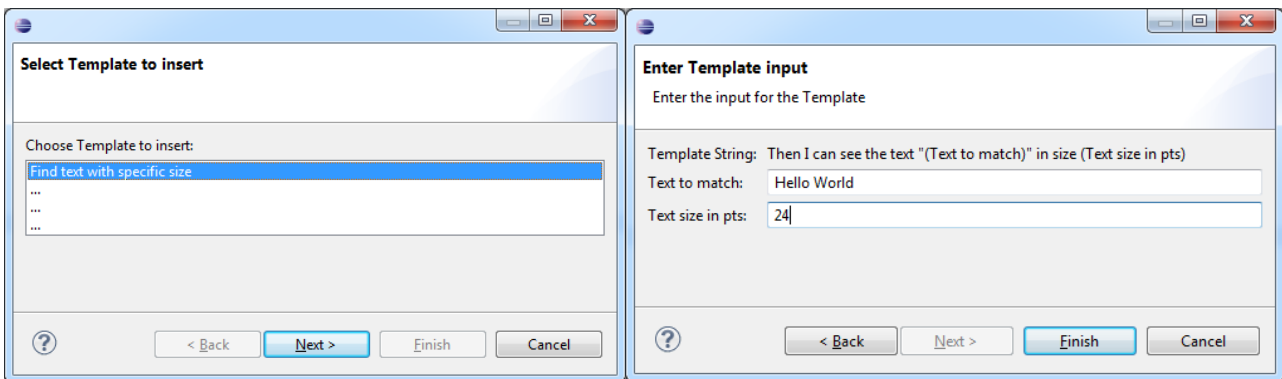


Figure 34 - Insert from Template implementation

Again, the CommandHandler receives the command from the user activating the Eclipse menu item. The CommandHandler starts the Wizard, but this Wizard has two pages:



The first page asks the user to choose which Template to use. The next page is generated based on the chosen template and asks the user for the required input.

To get the list of Templates, CreateTemplateWizardPage1 asks the XMLHandler for a list of all the created Templates. The XMLHandler in turn uses the FilesystemUtils class to read all the Template files, and the XStream⁵ library is used to turn the XML from the template files into Template objects with the following code:

```
public static List<Template> getTemplates() {
    List<File> templateFiles = FilesystemUtils
        .getFilesByExtension(FilesystemUtils.getCurrentProjectRootDirectory(), "getemplate");
    List<Template> templates = new ArrayList<Template>();

    XStream xstream = new XStream();
    xstream.setClassLoader(Template.getClassLoader());
    xstream.alias("template", Template.class);

    for (File file : templateFiles) {
        templates.add((Template)xstream.fromXML(file));
    }

    return templates;
}
```

Of special note is the `xstream.setClassLoader(...)` call which is necessary because the Eclipse framework, which the plug-in is executed in, is based on the Open Services Gateway initiative⁶ (OSGi) framework. This means that the `Template` class is not guaranteed to be loaded by the same class loader as the `XStream` class. This is why `XStream` must be specifically told what class loader to use.

10 Test

10.1 Test methodology

Because this is a one-man project of relatively short duration I decided not to do automated testing. In my experience small projects tend to benefit a lot less from automated testing than bigger projects. There is a certain overhead in setting up automated testing, and finding tools that let your test code interface with the running program can be a project in itself. In the case of Eclipse plug-ins, Eclipse has its own Eclipse Test Framework⁷. Because the single most time consuming part of this project was figuring out the Eclipse Plug-in Framework, I thought it was probably not worth a big part of my project time to learn another Eclipse framework at the same time.

So in the end I decided that my test methodology would be to just manually test the functionality as the project grew.

10.2 Test Results

Gherkin Syntax Highlighting

Test	Result	Comment
Highlights "Feature: " keyword blue	✓	
Highlights "Scenario: " keyword blue	✓	
Highlights "Given " keyword blue	✓	
Highlights "When " keyword blue	✓	
Highlights "Then " keyword blue	✓	
Highlights Comments green	✓	
Highlights Tags red	✗	Only works after user presses Enter after the Tag

Context Sensitive Assistance

Test	Result	Comment
Retrieves and suggests implemented Gherkin sentences	✓	
Inserts selected Gherkin sentence	✓	
Inserts selected sentence instead of the selected line in the feature file	✗	It inserts it after the selected line instead
Only shows the sentences that begin with the selected line	✓	For some reason this works when the plug-in is run from the GherkinEdit project, but this functionality does not work when exported to a .jar file to be distributed.
If no sentences begin with the selected line, the sentences are ranked by their Levenshtein distance to the selected line	✓	For some reason this works when the plug-in is run from the GherkinEdit project, but this functionality does not work when exported to a .jar file to be distributed.

Template System

Test	Result	Comment
Template menu is shown in Eclipse menu bar	✓	
Create Template wizard receives user input	✓	
Create Template wizard creates template file after successful completion	✓	
Create Template wizard does not create template file if wizard is aborted	✓	
Insert from Template wizard loads all existing Templates and present them to the user	✓	
Insert from Template wizard asks the user for correct input for selected template	✓	
Insert from Template wizard inserts the chosen Gherkin sentence with user input in document after successful termination	✓	
Insert from Template wizard does not insert anything in the current document if the it is canceled.	✓	

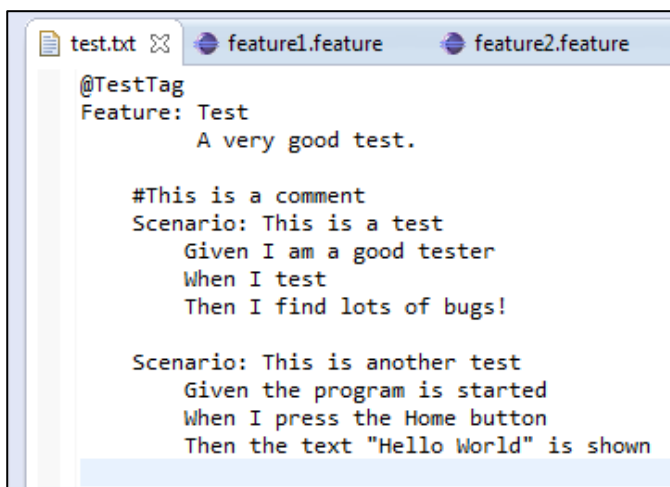
11 Result analysis

Even though not all tests passed, I am still quite satisfied with the result. I have not had time to iron out all the bugs but I think GherkinEdit clearly shows the promise of the tools I have created.

In the following chapters I will compare the old ways of making Cucumber tests to the way it can be done with GherkinEdit. The scenarios of how it is traditionally done is based on my experience in a student job where I worked on a project where Cucumber were used for all automated tests.

11.1 Gherkin Syntax Highlighting

Before Gherkin Syntax Highlighting, the features would just look like this:

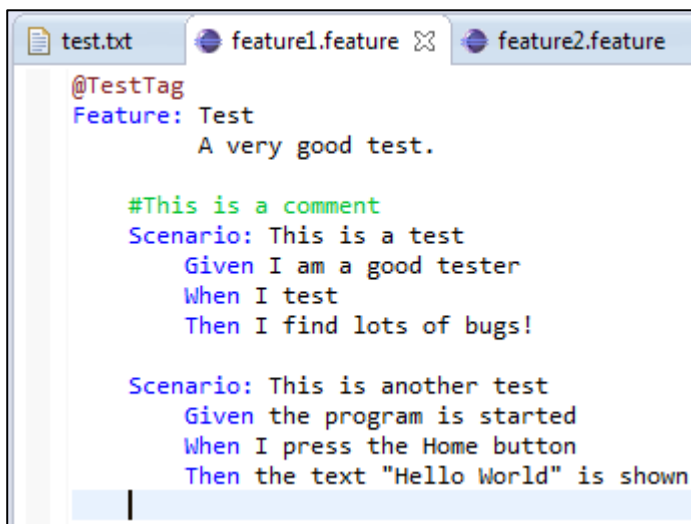


```
test.txt feature1.feature feature2.feature
@TestTag
Feature: Test
    A very good test.

    #This is a comment
    Scenario: This is a test
        Given I am a good tester
        When I test
        Then I find lots of bugs!

    Scenario: This is another test
        Given the program is started
        When I press the Home button
        Then the text "Hello World" is shown
```

Now, they look like this:



```
test.txt feature1.feature feature2.feature
@TestTag
Feature: Test
    A very good test.

    #This is a comment
    Scenario: This is a test
        Given I am a good tester
        When I test
        Then I find lots of bugs!

    Scenario: This is another test
        Given the program is started
        When I press the Home button
        Then the text "Hello World" is shown
```

It is in my opinion a vast visual improvement that makes it much easier to recognize which part of the test does what and how to interpret it.

11.2 Context Sensitive Assistance

Before, when you wanted to reuse implemented Gherkin sentences, you had to read through the entire (sometimes huge) test code base and identify the Gherkin sentences they supported:

```
public class GherkinSteps {
    @Given("^the webpage ([a-z]*) exists$")
    public void theWebpageExists(String webpage) {
        //check that the webpage exists
    }

    @When("^I press the ([a-z]*) link$")
    public void pressLink(String link) {
        //press a link
    }

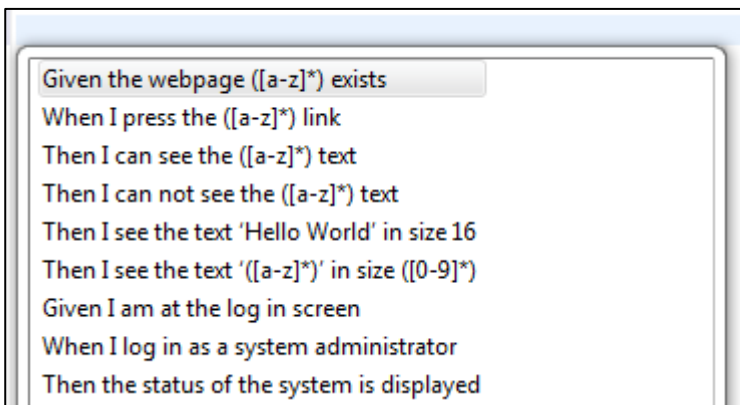
    @Then("^I can see the ([a-z]*) text$")
    public void checkForText(String text) {
        //check text
    }

    @Then("^I can not see the ([a-z]*) text$")
    public void checkThatTextIsNotThere(String text) {
        //check that text is not there
    }

    @Then("^I see the text 'Hello World' in size 16$")
    public void findHelloWorldInSize16() {
        // Find 'Hello World' in size 16
    }

    @Then("^I see the text 'Hello World' in size ([0-9]*)$")
    public void findHelloWorldInSize16(ct, String Size) {
        // Find 'Hello World' in size 16
    }
}
```

With the Context Sensitive Assistance system, all the sentences are available by pressing Ctrl+Space:



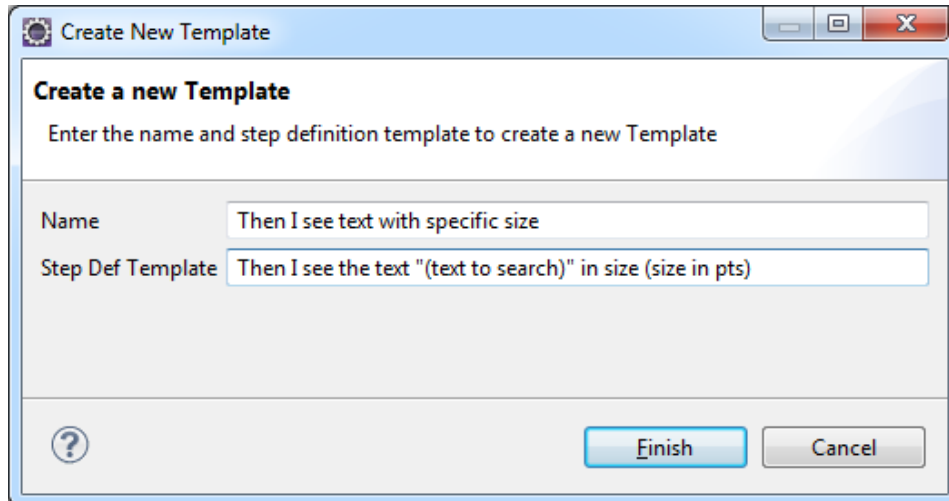
It even tries to guess which sentences you want by analyzing the line you are currently writing, and displays them first.

11.3 Template system

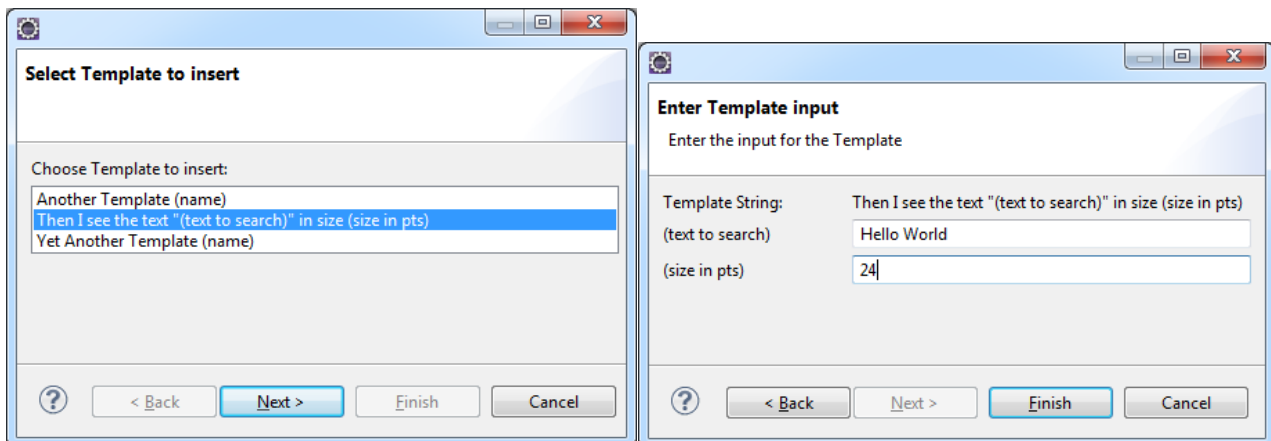
In traditional Cucumber test development, expressing how to use the test code to write supported Gherkin sentences is quite time consuming. The developer either have to explain it in person to the test writers, write documents or draw illustrations. In the case of explaining it in person, this has to be repeated every time a new person wants to write tests. In the case of documents and drawings, first of all they are time

consuming to make and second they need to be attached to the specific code project being worked on or indexed in some other way.

With GherkinEdit the Template system is right where the users need it – in the code project itself. There is only one dialog needed to create a description:



And only two dialogs needed to use it:



12 Discussion of future improvements

In the Gherkin Syntax highlighting there are a few situations where it does not behave as expected, for example to color the tag string you need to press Enter after writing it. This is obviously something I would like to fix. Also it would be nice to add support for customizing the colors and choosing which language features should be highlighted and which should not.

Additionally, Cucumber supports many other languages than English, and it would be nice to support syntax highlighting for these as well.

As for the Context Sensitive Assistant, it is the most unstable part of GherkinEdit. But besides fixing the bugs so it works completely as intended, I don't think there is much more I could ask it to do.

With the Template system I think there is a lot of potential for improvements. Right now all user inputs when inserting a Template is Strings only. It would be great to be able to specify if an input should be plain text, number or Enums (a specific set of options). Furthermore a system for organizing the Templates in categories instead of just each project having a long list of Templates would also be a good improvement.

13 Conclusion

In this project I have analyzed the workflow of developing Cucumber tests, identified steps that were well suited for automation, and proposed tools that could be used to automate these steps. The tools I proposed was a syntax highlighting tool, an intelligent suggestion tool and a template tool. I then designed, implemented and tested the tools.

In the end I reflected on the usefulness of the tools and how they could be improved in the future.

One of the most useful things I learned was how to use the Eclipse Plug-in framework. It has a very steep learning curve and many novel design ideas which takes time to understand, but the usefulness and ease with which plug-ins can be created to change or add almost any functionality in Eclipse more than makes up for the effort.

I am also quite happy with my GherkinEdit plug-in. It saves a lot of time in writing Cucumber tests, and even though it has some rough edges none of them makes it take longer to write tests. It is definitely a tool I will use in the future.

14 References

1. Eclipse Plug-ins, 3. Edition, Eric Clayberg, 2008 (ISBN-13: 978-0-321-55346-1)
2. Levenshtein Distance, in Three Flavors (<http://www.merriampark.com/ld.htm>)
3. Levenshtein distance, Wikipedia page (http://en.wikipedia.org/wiki/Levenshtein_distance)
4. Building an Eclipse Text Editor with JFace Text (<http://www.realsolve.co.uk/site/tech/jface-text.php>)
5. XStream (<http://xstream.codehaus.org/>)
6. Open Services Gateway initiative framework (OSGi) – (<http://en.wikipedia.org/wiki/OSGi>)
7. Eclipse Test Framework
(<http://dev.eclipse.org/viewcvs/viewvc.cgi/org.eclipse.test/testframework.html?view=co>)

15 Table of Figures

Figure 1 - Cucumber system overview	2
Figure 2 - Gherkin Example.....	2
Figure 3 - Example of step definitions	3
Figure 4 - Reference	10
Figure 5 - Component.....	10
Figure 6 - Framework component	10
Figure 7 - System service	10
Figure 8 - GherkinEdit system overview.....	11
Figure 9 - Gherkin Syntax Highlighting	11
Figure 10 - Content Sensitive Assistance.....	12
Figure 11 - Text capture.....	12
Figure 12 - Finding Cucumber annotations	12
Figure 13 - Removing annotation syntax.....	13
Figure 14 - Template system overview.....	15
Figure 15 - GherkinEdit Template Example	15
Figure 16 - Input prompt when using Template.....	16
Figure 17 - Create Template System	17
Figure 18 - Create Template Eclipse menu item	17
Figure 19 - Create Template Page GUI	17
Figure 20 - Insert from Template System	18
Figure 21 - Selecting Template to be inserted.....	19
Figure 22 - Prompting user for template input	19
Figure 23 - Final result of inserting a template	20
Figure 24 - Inheritance	21
Figure 25 - Reference	21
Figure 26 - class	21
Figure 27 - Framework component.....	21
Figure 28 - System service	21
Figure 29 - Implementation of Syntax highlighting	22
Figure 30 - Context Sensitive Assistance implementation	24
Figure 31 - Accessing the Template system functions	25
Figure 32 - Create Template implementation	27
Figure 33 - Create Template Wizard.....	27
Figure 34 - Insert from Template implementation.....	29

16 Appendix 1: GherkinEdit Installation instructions

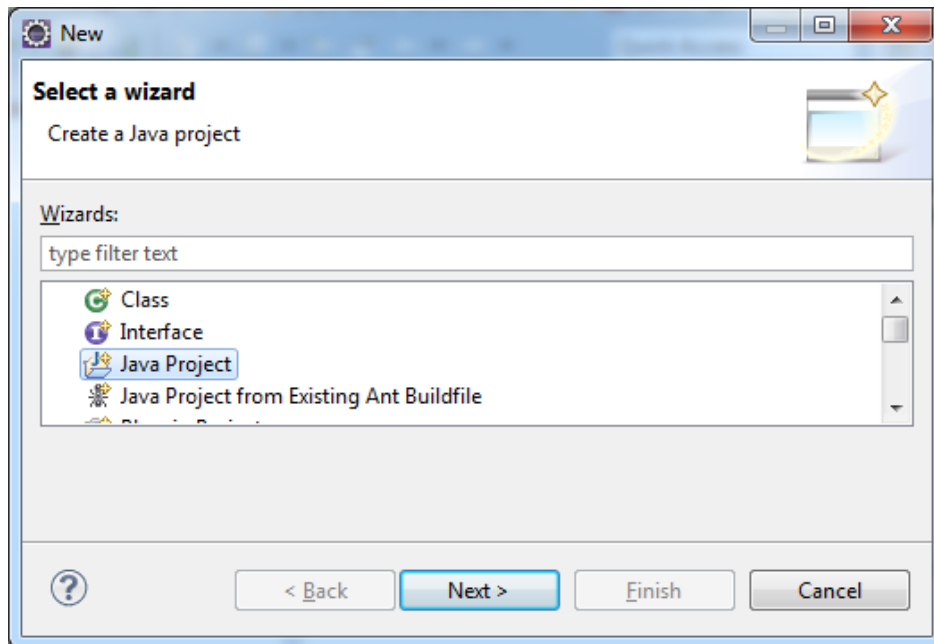
1. Download Eclipse from <http://www.eclipse.org/downloads/>
(GherkinEdit been tested with the Windows builds of “Eclipse IDE for Java EE Developers” Indigo and Juno in both 32 and 64 bit versions)
2. Install (extract) Eclipse to any directory
3. Put GherkinEdit_0.1.0.alpha.jar and XStream_1.4.2.jar in the plugins directory in the Eclipse installation directory.
4. Start Eclipse

17 Appendix 2: User Manual

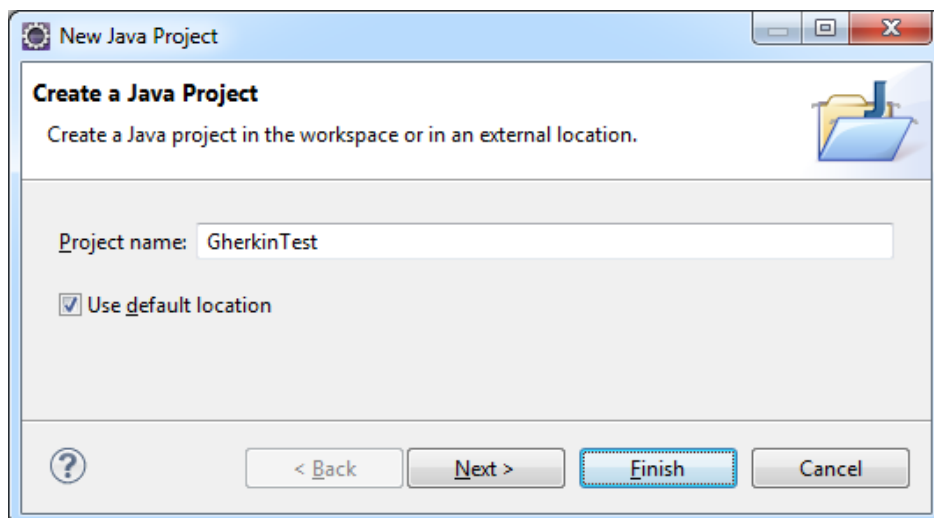
This user manual will guide you through setting up an Eclipse project and using the different features of the GherkinEdit plug-in. This user manual assumes that you have Eclipse IDE installed with the GherkinEdit plugin (if not, please refer to Appendix 1).

17.1 Setting up an Eclipse project to use GherkinEdit

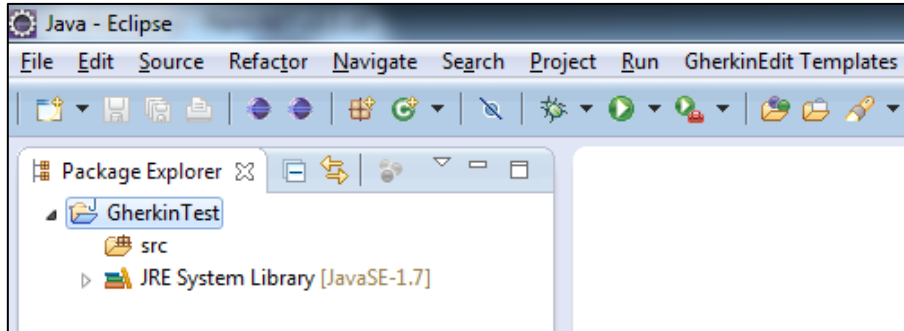
1. Start Eclipse and choose any directory for workspace (preferably an empty one)
2. Click File -> New -> Other... and choose "Java Project" and press Next



3. Write a project name and press finish.



4. If asked if you want to open the Java perspective, just answer Yes.
5. Expand your new project in the project explorer.

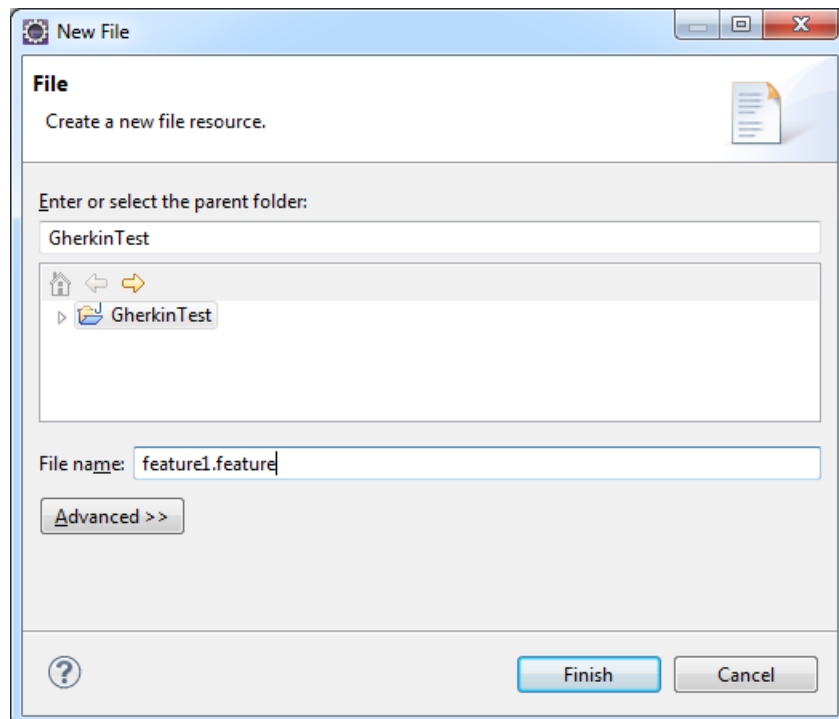


6. (Optional, but strongly recommended) install the cucumber-java library which avoids compile errors when using Cucumber annotations in the Java test code:
 - a. Download the latest cucumber-java .jar file from <https://oss.sonatype.org/content/repositories/releases/info/cukes/cucumber-java/>
 - b. Right-click your project (GherkinTest in this example) and select Build Path -> Add External Archives..
 - c. Select the cucumber-java.x.x.x.jar file you downloaded and press Open.
Cucumber-java should now be located in your project under “Referenced Libraries”

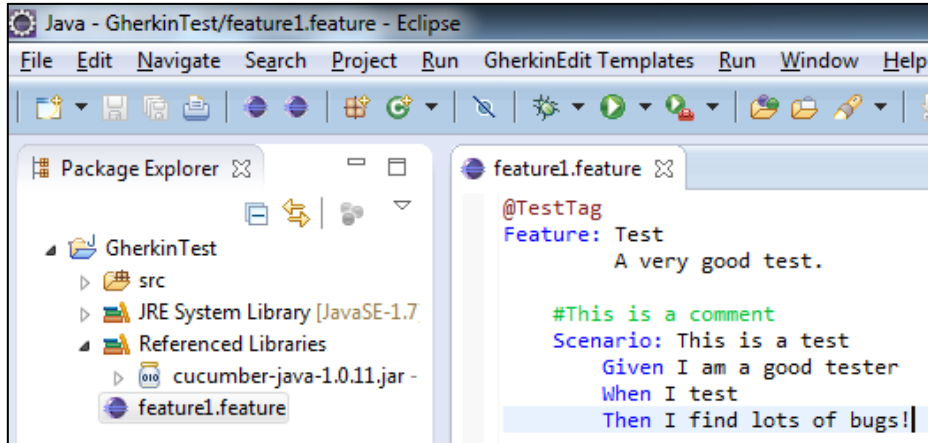
17.2 Using Syntax highlighting

The syntax highlighting is done automatically in all files ending on “.feature”.

1. Right-click on your project (GherkinTest in this example) and press New -> File...



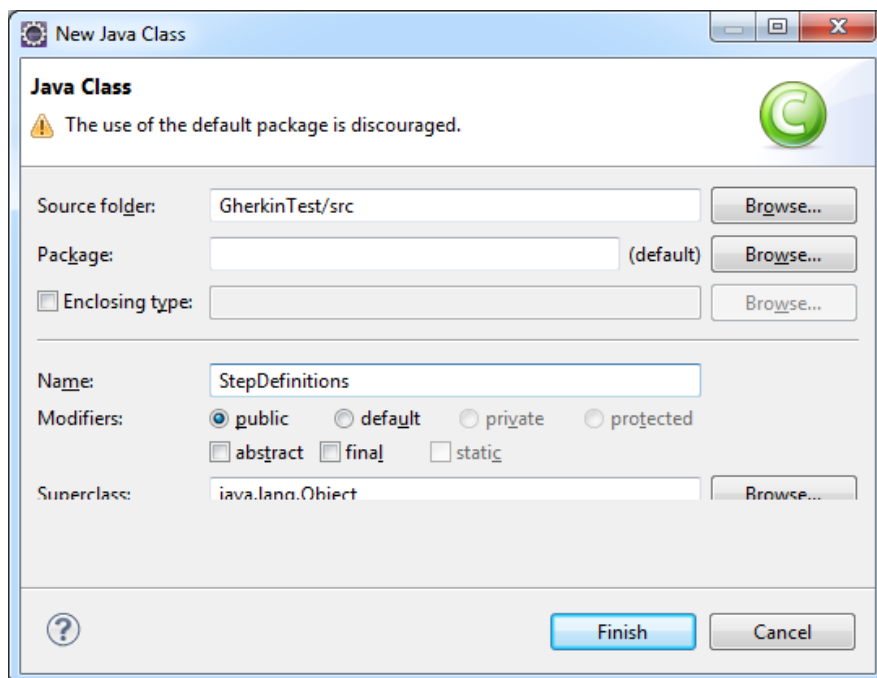
2. Type any file name ending on .feature and press Finish.
3. When you edit the file the syntax highlighting should work like this:



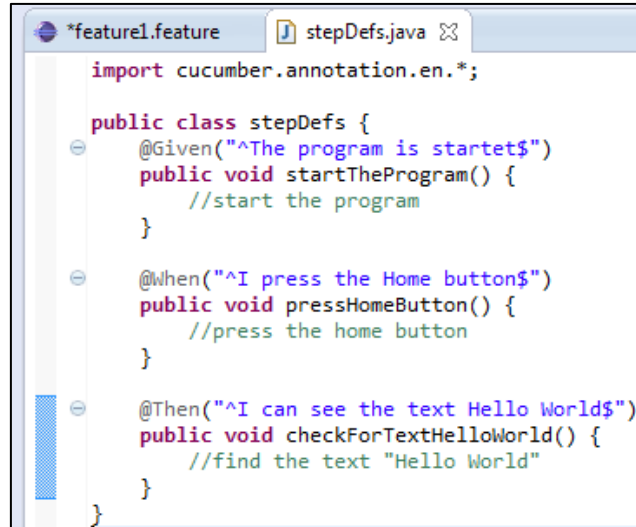
17.3 Using Context Sensitive Assistance

Because the Context Sensitive Assist function only recommends Gherkin sentences that have already been implemented in test code, we first need to create some annotated tests.

1. Right-click on the “src” folder and choose New -> Class
2. Type a name and click Finish:



3. Write a couple of test implementations of Gherkin sentences:



```

import cucumber.annotation.en.*;

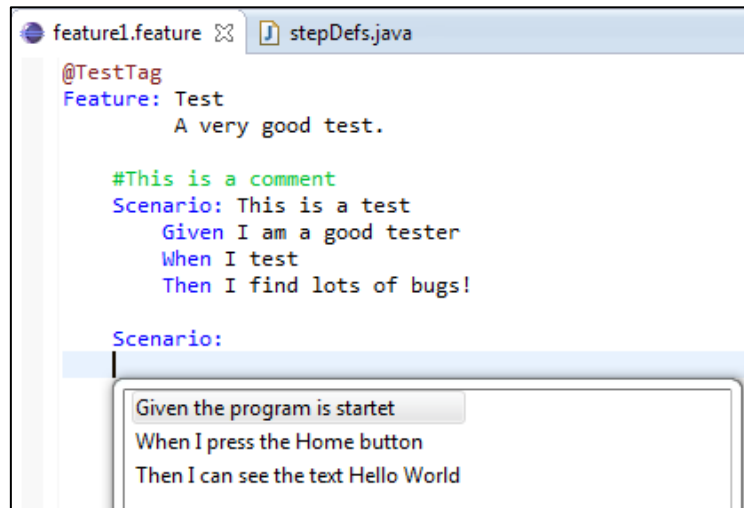
public class stepDefs {
    @Given("^The program is startet$")
    public void startTheProgram() {
        //start the program
    }

    @When("^I press the Home button$")
    public void pressHomeButton() {
        //press the home button
    }

    @Then("^I can see the text Hello World$")
    public void checkForTextHelloWorld() {
        //find the text "Hello World"
    }
}

```

4. Go back to the .feature file and press Ctrl+Space to get a list of the implemented sentences:



```

@TestTag
Feature: Test
    A very good test.

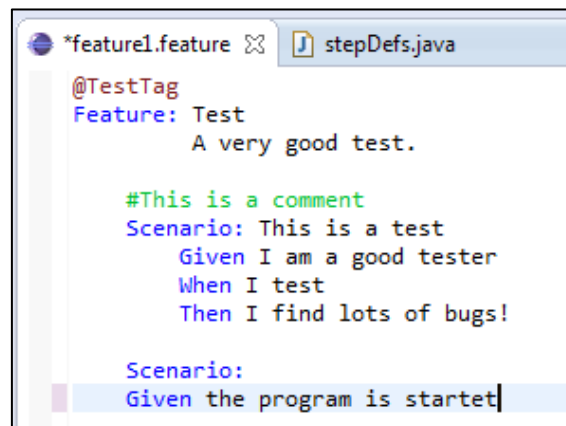
#This is a comment
Scenario: This is a test
    Given I am a good tester
    When I test
    Then I find lots of bugs!

Scenario:

```

Given the program is startet
When I press the Home button
Then I can see the text Hello World

5. Choose the desired sentence to have it inserted into the document:



```

@TestTag
Feature: Test
    A very good test.

#This is a comment
Scenario: This is a test
    Given I am a good tester
    When I test
    Then I find lots of bugs!

Scenario:
    Given the program is startet

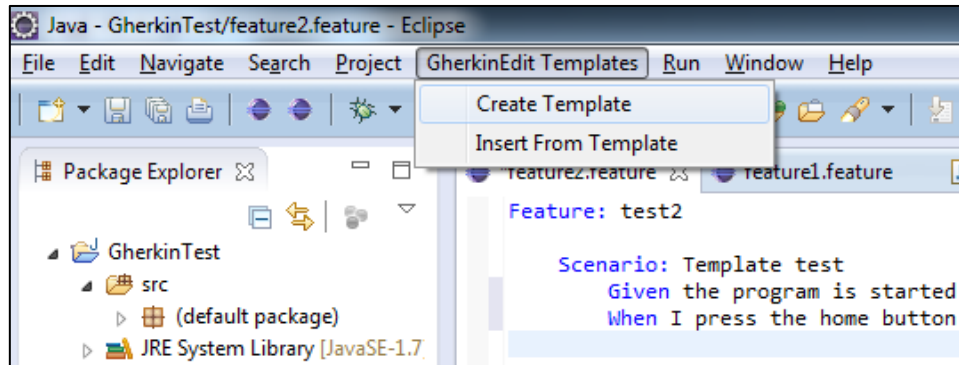
```

17.4 Using the Template system

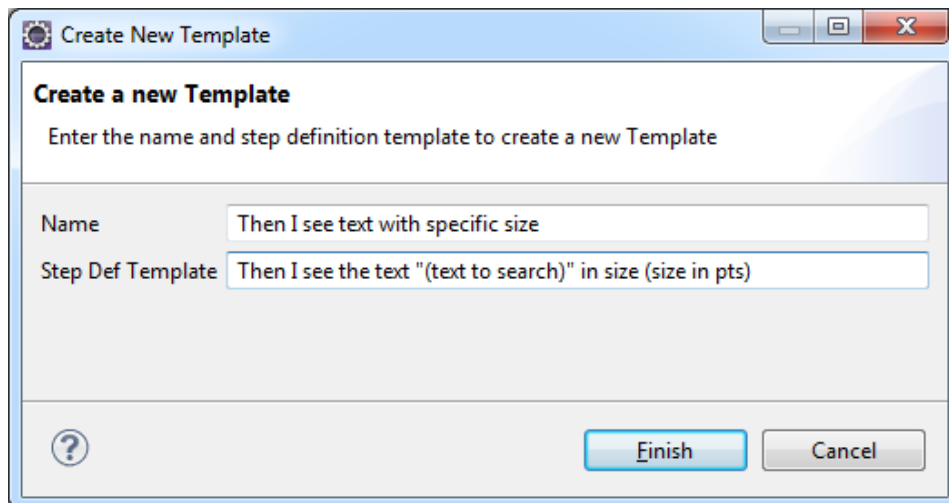
To use the template system, you only need an open .feature file.

17.4.1 Creating a Template

1. In the Eclipse menu bar, press GherkinEdit Templates -> Create Template:



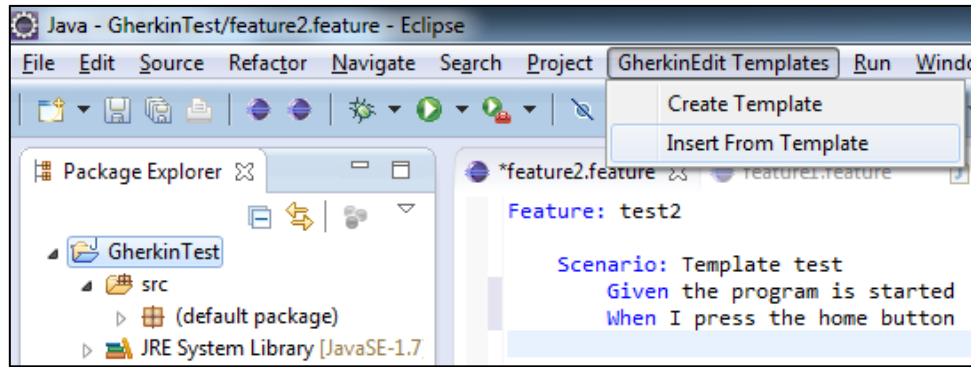
2. Write the name of the Template and its Step Def Template and click Finish:



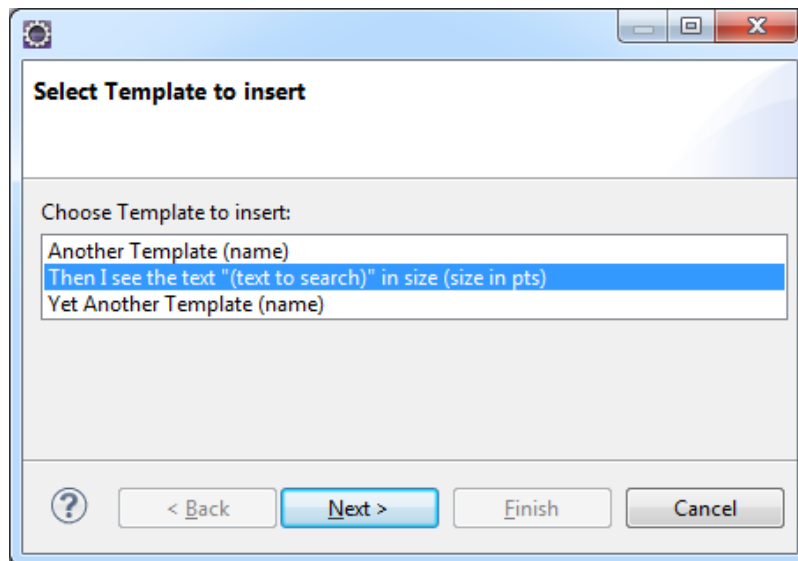
The Template should now have been created. If you select your project and press F5, there should now be a folder in your project called "GherkinEditTemplates" with the file representation of the template in it.

17.4.2 Inserting from a Template

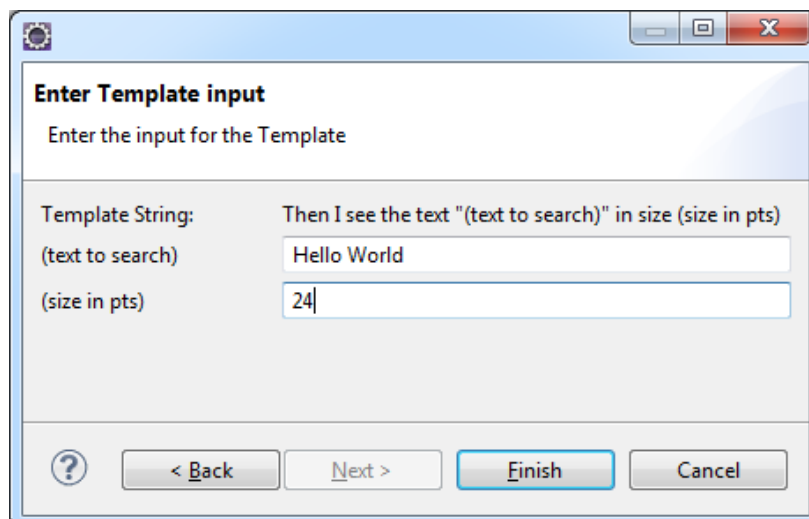
1. In the Eclipse menu bar, press GherkinEdit Templates -> Insert From Template:



2. Select the desired Template to insert:



3. Fill in all the input fields and press Finish:



The new sentence should now be inserted in the open feature file:

