

Development of an Eclipse Plug-In
for Tasklets

Bachelorarbeit
von

DANIEL FLACHS

MATRIKELNUMMER 1360598

19. Mai 2015

Vorgelegt am Lehrstuhl für Wirtschaftsinformatik II
Universität Mannheim
Gutachter: Prof. Dr. Christian Becker
Betreuer: Janick Edinger, M. Sc.

Eidesstattliche Erklärung:

Hiermit versichere ich, dass diese Arbeit von mir persönlich verfasst wurde und dass ich keinerlei fremde Hilfe in Anspruch genommen habe. Ebenso versichere ich, dass diese Arbeit oder Teile daraus weder von mir selbst noch von anderen als Leistungsnachweise andernorts eingereicht wurden. Wörtliche oder sinngemäße Übernahmen aus anderen Schriften und Veröffentlichungen in gedruckter oder elektronischer Form sind gekennzeichnet. Sämtliche Sekundärliteratur und sonstige Quellen sind nachgewiesen und in der Bibliographie aufgeführt. Das Gleiche gilt für graphische Darstellungen und Bilder sowie für alle Internet-Quellen.

Ich bin ferner damit einverstanden, dass meine Arbeit zum Zwecke eines Plagiatsabgleichs in elektronischer Form anonymisiert versendet und gespeichert werden kann. Mir ist bekannt, dass von der Korrektur der Arbeit abgesehen werden kann, wenn diese Erklärung nicht erteilt wird.

Mannheim, 19. Mai 2015

Daniel Flachs

Contents

List of Figures	v
List of Tables	vi
List of Listings	vii
List of Abbreviations	viii
1. Introduction	1
1.1. Motivation	1
1.2. Goal of this Work	2
1.3. Structure	2
2. Fundamentals	3
2.1. The Tasklet Project	3
2.1.1. System Overview	4
2.1.2. The Tasklet Language	8
2.2. The Eclipse Platform	10
2.2.1. General Overview	11
2.2.2. Basic Components	11
2.2.3. Plug-Ins and Features	13
2.2.4. Extensions and Extension Points	14
3. Plug-In Design	16
3.1. Requirements	16
3.1.1. Writing and Editing Source Code	16
3.1.2. Compilation	17
3.1.3. Documentation of Source Code	18
3.2. Component Design	18
3.2.1. The Editor	20
3.2.2. The View	20
4. Plug-In Implementation	22
4.1. General Plug-In Properties	22
4.2. Package and Folder Structure	24
4.3. The CMM Editor	25
4.3.1. Document Provider, Partitioner and Partition Scanner	26

4.3.2. Editor Configuration	26
4.4. The CMM Code Documentation View	28
4.4.1. Data Model	29
4.4.2. Main View Class	31
4.4.3. Content Provider, Label Provider and Editing Support . .	35
4.4.4. Model Builder	37
4.5. The CMM Perspective	38
4.6. CMM Source File Compilation	38
5. Summary	40
5.1. Conclusion	40
5.2. Outlook	41
5.2.1. Open Issues	41
5.2.2. Additional Features	42
Appendix A. Plug-In Installation Instructions	45
Appendix B. Code Listings	47
B.1. Plug-In-specific Listings	47
B.2. Java Source Code Listings	47
B.3. C-- Source Code Listings	49
Bibliography	50

List of Figures

2.1.	3-Tier Architecture of the Tasklet System	5
2.2.	Overlay Network Structure of the Tasklet System	6
2.3.	Eclipse workbench window with components	12
3.1.	Standard layout of the C-- perspective	19
4.1.	The view's data model	30
4.2.	View initialization	32

List of Tables

2.1. Tasklet Language Keywords	9
------------------------------------------	---

List of Listings

- 4.1. Excerpt of “plugin.xml” 23
- 4.2. Excerpt of “CMMCodeScanner.java” (simplified) 27
- B.1. MANIFEST.MF 47
- B.2. CMMPcedure.serialize (simplified) 47
- B.3. CMMPcedure#deserialize (simplified) 48
- B.4. primes.cmm 49

List of Abbreviations

API	Application Programming Interface
CMM	C--, the Tasklet Language
cmm	File extension for a C-- source file
GUI	Graphical User Interface
IDE	Integrated Development Environment
JDT	Java Development Tools
JSON	JavaScript Object Notation
PDU	Protocol Data Unit
SWT	Standard Widget Toolkit
TBC	Tasklet Byte Code
TCP	Transmission Control Protocol
TDF/tdf	Tasklet Documentation File (file extension in lowercase letters)
TF	Tasklet Factory
TP	Tasklet Protocol
TVM	Tasklet Virtual Machine
TVMM	Tasklet Virtual Machine Manager
UI	User Interface
e. g.	<i>exempli gratia</i> , for example
i. e.	<i>id est</i> , that is

1. Introduction

1.1. Motivation

These days, software programmers can rely on a vast variety of tools which support them in writing source code and in developing software. Integrated Development Environments (IDEs) are quite popular tools which consist, for instance, of source code editors and compilation support features. Thereby, they integrate the overall work flow from writing the first lines of code to deploying the final software product. Hence, when a new programming language emerges, it is highly desirable to provide a suitable IDE for software developers who want to write code in that language.

The goal of the Tasklet project is to build a system which allows the execution of fine-grained computation units, so-called *Tasklets*, on a large number of heterogeneous devices in a distributed environment. Tasklets can be executed either locally or remotely on another machine and are written in “C--” (also “CMM”), the *Tasklet Language*, which was built to fulfill exactly the requirements of the project. The design of C-- allows distributed and lightweight computation, while the syntax of the language is based on the programming language C and offers some of C’s functionality.

Since C-- is a new programming language which was designed and developed within the project, no IDE support exists so far. Tasklet code has to be written with standard system editors, such as “Notepad”, which do not provide any language-specific support, e.g., syntax highlighting. Hence, the work of a C-- developer is quite uncomfortable, compared to, e.g., IDE-aided Java development. Especially when it comes to debugging the source code, IDE support for C-- would be of great help.

Quite powerful IDEs for various programming languages already exist. A lot of them can be extended by custom functionality. Hence, it seems unnecessary

and exaggerated to implement a completely new IDE solely for C++ development. Therefore, this work describes the development of a plug-in for the IDE *Eclipse* in order to provide Tasklet language-specific support, such as the invocation of the C++ compiler on click, or a tool to support the programmer when he/she wants to document his/her source code.

1.2. Goal of this Work

This thesis aims to develop an Eclipse plug-in which considerably simplifies the development of Tasklets for programmers. The plug-in is supposed to integrate and interface with the Eclipse IDE, just like any other Eclipse plug-in. The overall goal is to integrate the Tasklet development work flow in one single IDE in a comfortable and user-friendly manner.

1.3. Structure

The remainder of this work is organized as follows: Firstly, the fundamentals of both the Tasklet project and the Eclipse framework as underlying IDE are provided in chapter 2. Secondly, in chapter 3, the design of the overall plug-in is presented, with respect to, e.g., the user interface. Afterwards, the concrete implementation of the functions will be introduced in chapter 4. Finally, this work is concluded along with addressing open issues and additional desirable features in chapter 5.

2. Fundamentals

This chapter provides an overview of both the basic aspects of the Tasklet project and the fundamentals of the Eclipse framework. These two areas are covered because, on the one hand, the Tasklet project and especially the Tasklet language are the reasons for which the implementation of a plug-in is necessary: The work of C++ developers should be supported and simplified by an IDE. On the other hand, the Eclipse framework provides such IDE support because its functionality can be extended by plug-ins.

2.1. The Tasklet Project

The Tasklet project is settled at the Chair of Information Systems II at the University of Mannheim. It aims for a new abstraction for computation, addressing the heterogeneous and distributed nature of today's landscape of computation devices.

Nowadays, an almost innumerable quantity of different electronic devices capable of performing computations exists. Personal computers, cloud services, embedded or mobile devices, like smartphones, are just some examples. They are all heterogeneous in terms of computation power, workload, availability or reliability. For instance, a laptop and a smartphone differ substantially regarding their hardware endowment, although they both have to perform complex computations, such as image processing. Furthermore, many devices exhibit quite an amount of idle time during which their resources remain unused.

The goal of the Tasklet project is to make use of this heterogeneity and the free resources by providing a system which is capable of executing computations either locally or on a remote device. To achieve this, a common level of abstraction for computation is needed in order to be suitable for all devices involved.

One of the vital concepts within the project are so-called *Tasklets* which are fine-grained computation units that can be issued for local or remote execution. All program parts or algorithms which are implemented as Tasklets can utilize both parallelism and idle resources: The programmer can start multiple Tasklets at the same time (parallelism) and execute one or more Tasklets on remote devices currently having excess capacities. As a consequence, the most resource-consuming parts of a program should be moved to Tasklets to gain flexibility (with respect to the location of execution) and performance increase. In contrast to cloud services, such as Amazon Web Services¹ or Google Cloud Platform², which also offer resource sharing and distributed computing, the system does not require a fixed architecture of (homogeneous) cloud servers. Instead, the system can make use of nearly any device, ranging from embedded devices over smartphones and PCs to high-performance servers. By defining so-called *Quality of Computation* (QoC), it is possible for the programmer to specify the execution conditions of a Tasklet, for instance reliable execution, so that a result is returned in any case.

The integration of Tasklets into host applications is provided by programming language-specific libraries, which already exist for Java SE and Android.

2.1.1. System Overview

The following paragraphs provide an overview of the Tasklet system. Basically, the architecture of the system can be divided into three tiers (see also Figure 2.1): Execution Tier, Distribution Tier and Construction Tier.

From an overlay network point of view, the Tasklet system is organized as a hybrid peer-to-peer network with three types of entities: *Resource providers* offer their idle resource to *resource consumers*, so that a consumer's Tasklets are executed on a provider's resources. Providers and consumers are not necessarily disjoint, meaning that one entity can be producer and consumer at the same time, i.e., Tasklet execution can happen locally as well. On each provider, one or more Tasklet Virtual Machines (TVMs) are running, depending on the provider's free resources. TVMs constitute the execution environment for Tasklet byte code. The third type of entities, called *resource brokers*, are centralized servers within

¹<http://aws.amazon.com/de/>

²<http://cloud.google.com/>

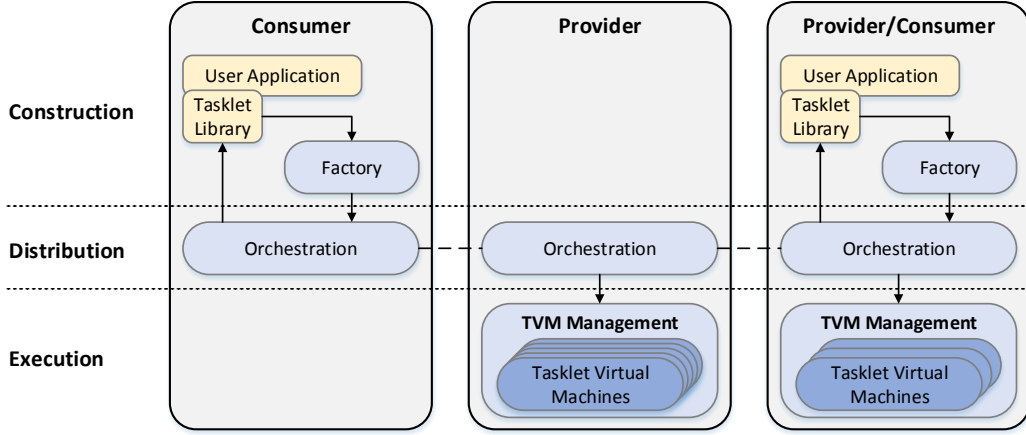


Figure 2.1.: The 3-Tier Architecture of the Tasklet System

the system responsible for managing the system’s resources. Providers offer their idle resources by registering to a broker. A consumer which wants to execute Tasklets remotely can request resources on a provider by contacting the broker. Brokers are only responsible for matching “demand and supply” – in contrast, Tasklets to be executed and the corresponding results are exchanged directly between consumers and providers. Figure 2.2 shows an overview of the system architecture.

Execution Tier

The *Tasklet Virtual Machine* is the execution point, i.e., the interpreter, for compiled Tasklet byte code (TBC). In order to run on as many devices as possible, including such with minimal hardware equipment, e.g., embedded systems, the TVM process is lightweight regarding memory usage. TVMs execute one incoming Tasklet after the other, in a batch processing manner and without interruption. They do not support multi-threading natively, however, one resource provider can host any nonnegative number of TVMs, depending on the idle resources. Furthermore, any TVM can be terminated at any time, e.g., when the provider allocates the formerly free resources otherwise, thereby possibly dropping the currently executed Tasklet. Since Tasklet execution can happen on any remote device, the TVM does not allow platform-dependent system access, e.g.,

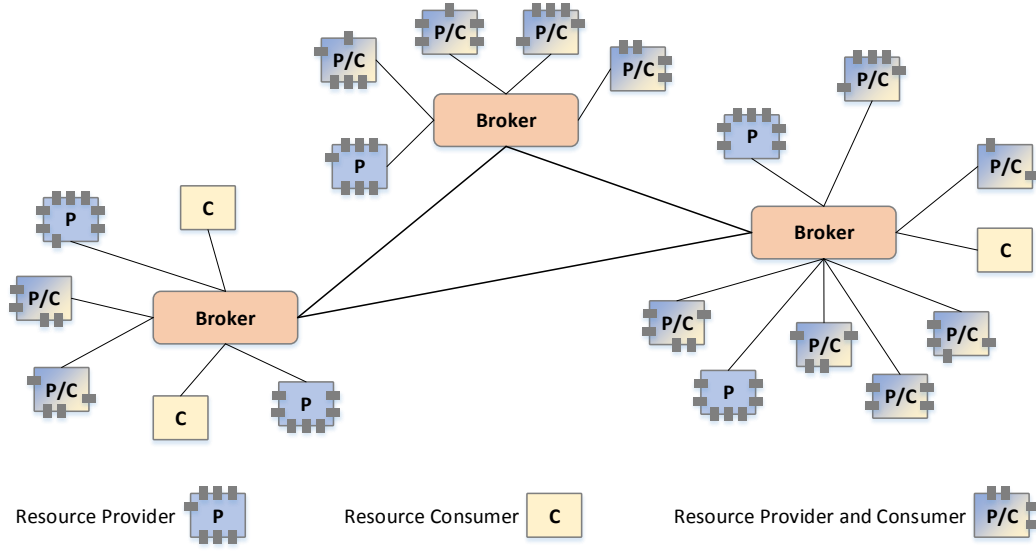


Figure 2.2.: The Overlay Network Structure of the Tasklet System

calls to the operation system. Furthermore, neither Tasklet code nor parameters nor other data ever leave the TVM's memory, which is located in the volatile RAM of the physical machine. If an error occurs during a Tasklet's execution, the TVM simply drops the respective Tasklet. This is called “best-effort computation”: One cannot be sure if a Tasklet issued to a TVM will ever return a result.

Each resource provider runs one *Tasklet Virtual Machine Manager* (TVMM) which is responsible for starting and stopping the TVMs on that machine. The number of launched TVMs is determined in consideration of the physical machine's status, namely the device's number of CPU cores, the current CPU usage (respectively the number of idle cores), network connectivity or – with mobile devices – battery charge. Furthermore, the TVMM has two “modes of operation”: Either a specific, static number of resources is assigned for the TVMs, or creation and termination of TVMs depends dynamically on the excess capacity of the device.

Construction Tier

Tasklets are written in the Tasklet language “C--” and compiled in the *Tasklet Factory* (TF). Since the Tasklet language and its syntax are of high importance regarding the development of the Eclipse plug-in, they will be described in detail in Section 2.1.2, whereas this section focuses on the compilation process.

The Tasklet factory is located on the resource consumer side of the system. Its task is to compile C-- source files (file extension “cmm”) and output the corresponding Tasklet byte code (file extension “tbc”), which, later on, can be executed by a Tasklet virtual machine on the provider side. The compiler creates parameterized Tasklet byte code and conducts a static type safety check. The locations of the parameters in the byte code are marked to allow fast exchange in case the same source code with a different set of parameters has to be compiled more than once. This especially speeds up parallel execution of the same Tasklet with different parameter values. The TF’s error handling is straightforward: If a compilation error occurs, the responsible Tasklet is dropped, the TF returns an error code and resets itself to accept the next Tasklet for compilation. After the compilation process, the Tasklet byte code is handed over to the orchestration in the distribution tier.

Distribution Tier

The *Distribution Tier* is responsible for the messages which are exchanged both among the system’s components and between them and the resource management. The protocol data units (PDUs) which are exchanged within the system are defined by the *Tasklet Protocol* (TP), which is based on the connection-oriented, reliable Transmission Control Protocol. Two categories of messages – messages within the system and interface messages between the host application and the Tasklet system – can be distinguished. Basically, there are three types of *System Internal Messages*, which carry either Tasklets for execution, the corresponding results or messages to trigger the execution of a Tasklet. The two types of *Interface Messages* are used to transfer Tasklets and results between the Tasklet factory and the host application.

As already mentioned, *resource management* is provided by resource brokers.

Each provider's TVMs register with a broker by means of a "vmUpMessage". Deregistration is realized via analogous "vmDownMessages". The broker keeps track of the registered and available TVMs and assigns only as many Tasklets to a provider as it can handle in order to prevent queuing. Whenever a consumer wants to execute a Tasklet on a remote TVM, it issues a *resource request* to its broker whereupon the broker selects an appropriate provider. The provider's contact information is then delivered to the requesting consumer. Furthermore, the broker decrements the number of available TVMs for the associated provider. As soon as the TVM on the selected provider finished executing the Tasklet, it signals this fact to the broker by sending a "vmUpMessage".

Under normal, faultless conditions, providers notify their broker when they are about to exit the system. In order to handle leaving as a result of an error as well, such as loss of network connection or crash of the device, additional *heartbeat channels* are established between brokers and providers. Providers without a heartbeat are removed from the broker's resource pool.

2.1.2. The Tasklet Language

The Tasklet Language called C-- was designed and built from scratch as part of the Tasklet project. It is intended to feature distributed, lightweight and generic computation and uses the imperative and procedural programming paradigm. C--'s type system provides strong typing and static type checking, i. e. each variable is assigned a data type when declared. The language is based on the programming language C – thus, the name is not a coincidence – and offers a fraction of C's functionality. Table 2.1 shows an overview of C--'s reserved keywords. An example of how a C-- program can be used to find prime numbers is located in the Appendix, see Listing B.4.

Code Structure Tasklet source code is supposed to have the following structure: First, constants and global variables are declared. After that, an arbitrary number of procedures can be defined. Finally, statements which can be considered the "main function" of the Tasklet follow. The main function is the starting point of the execution where previously declared variables can be initialized and procedures can be called. Procedures can also be invoked within other proce-

Group	Reserved Words
Data Types and Variable Declaration	void, bool, char, int, float const
Control Flow and Procedures	if, else, while procedure, return
Constants	true, false
Standard Functions	length, nroot, random, sqrt, log, logII, logX

Table 2.1.: Tasklet Language Keywords

dures. Blocks, e. g., procedure bodies, are surrounded by curly brackets, whereas indentation and whitespace are syntactically irrelevant.

Data Types C-- defines integers (`int`) and floats (`float`) as numeric data types, characters (`char`) and string literals to represent text, a Boolean type (`bool`) to represent logical values and the “empty” type `void`. All of the preceding data types can be used for variables, arrays, procedure parameters and procedure return values. The keyword `const` can be used to declare constants.

Statements and Operators A statement, such as declaring and/or assigning a variable, ends with a semicolon. C-- supports an operator for variable assignment (`:=`), basic arithmetic operators (`+`, `-`, `*`, `/`, `%`) and operators for comparison (`=`, `#` (inequality), `<`, `<=`, `>=`, `>`).

Control Flow Statements The Tasklet Language supports decision-based branching with the common `if/else` construct, which is known from many other programming languages. In the current version, the Boolean condition of the `if` statement only supports simple logical expressions. For instance, combining two conditions with logical AND (\wedge) has to be expressed by means of two nested `if` statements.

Loops can be implemented using the `while` statement, which represents a top-controlled loop. The loop is repeated as long as the loop condition is true.

Procedures Since C-- is a procedural programming language, it supports the declaration of procedures within a Tasklet. Procedures are identified by a unique name. They have a typed parameter list with any number of parameters and a return value type (including `void`). Procedures have to be defined before the main function of the Tasklet and can then be invoked from there. C-- also supports recursive procedure calls.

Standard Functions C-- provides a set of standard functions for basic operations, including the calculation of random numbers (`random`), n^{th} roots (`nroot`, `sqrtn`), logarithms (`log`, `log10`, `log2`) and the determination of an array's length (`length`).

I/O Operations To access execution parameters from within the Tasklet and to provide return values to the issuer of the Tasklet, the Tasklet language has two special operators. Both the input parameters and the results are included and transferred in byte arrays. When issuing a Tasklet, the host application has to build a byte array which contains all input parameters for the Tasklet. During execution, the parameters can be read from the source code by means of the *input operator* `>>` and stored in variables within the program. To deliver the result values, the *output operator* `<<` has to be used to add the current value to the result byte array. After the Tasklet has been executed, the result byte array is transferred back to the issuer of the Tasklet.

Comments In the current language version, inline comments in the source code are not possible yet. In future versions, both single-line comments (starting with `//`) and multiline comments (starting with `/*` and ending with `*/`) will be possible.

2.2. The Eclipse Platform

This section presents Eclipse as a modular platform which can be customized and extended by plug-ins. This foundation is needed in order to provide the basis for designing and developing an Eclipse plug-in for C-- support as described in Chapters 3 and 4.

2.2.1. General Overview

Eclipse is well-known as a popular Integrated Development Environment (IDE), especially for the programming language Java. The Eclipse IDE is part of the Eclipse project which is “a community for individuals and organizations who wish to collaborate on commercially-friendly open source software” [Ecl15a]. Eclipse provides a workbench and a plug-in model instead of concrete end user functionality, i.e. it is a platform which can be enriched by various modular features, namely *plug-ins*, for a large number of applications [Ecl15d]. The Eclipse plug-in/component model called *Equinox* is based on OSGi, a framework which provides a modular system for Java [Ecl15c, OSG15]. Since Eclipse is open source software, everyone can contribute plug-ins and make them available for other users in the community. Furthermore, the fact that the platform itself is written in Java, a well-established programming language, makes it easy to extend. Although Eclipse is maybe best-known as an IDE for Java (JDT, Java Development Tools), support for numerous other programming languages exists as well, such as C/C++, PHP or Python. All these features are implemented as plug-ins, using the platform as a basis.

The Eclipse platform’s architecture is built to provide a small runtime which acts as an underlying basis. The platform hides the complexity of handling, for instance, different operating systems or accessing file resources on disk from the plug-in programmer. Plug-ins are built on top of this groundwork in a layered manner, so that one plug-in can extend another. As a consequence of this encapsulation, plug-in developers can focus on the specifics of their plug-in instead of dealing with such general tasks as, e.g., resource handling [Ecl15d].

2.2.2. Basic Components

Eclipse distinguishes two basic user interface components with semantic behavior: *views* and *editors*. A view displays information for a given object, mostly in a hierarchical manner [Ecl15d], e.g. the “Outline View” to show the current editor’s source code structure. If a view is able to accept user input, this input is normally stored directly by updating the view’s underlying model. The resulting change is immediately reflected by the workbench. In contrast, editors are used to display and edit objects, normally files on disk. An editor is directly connected to these

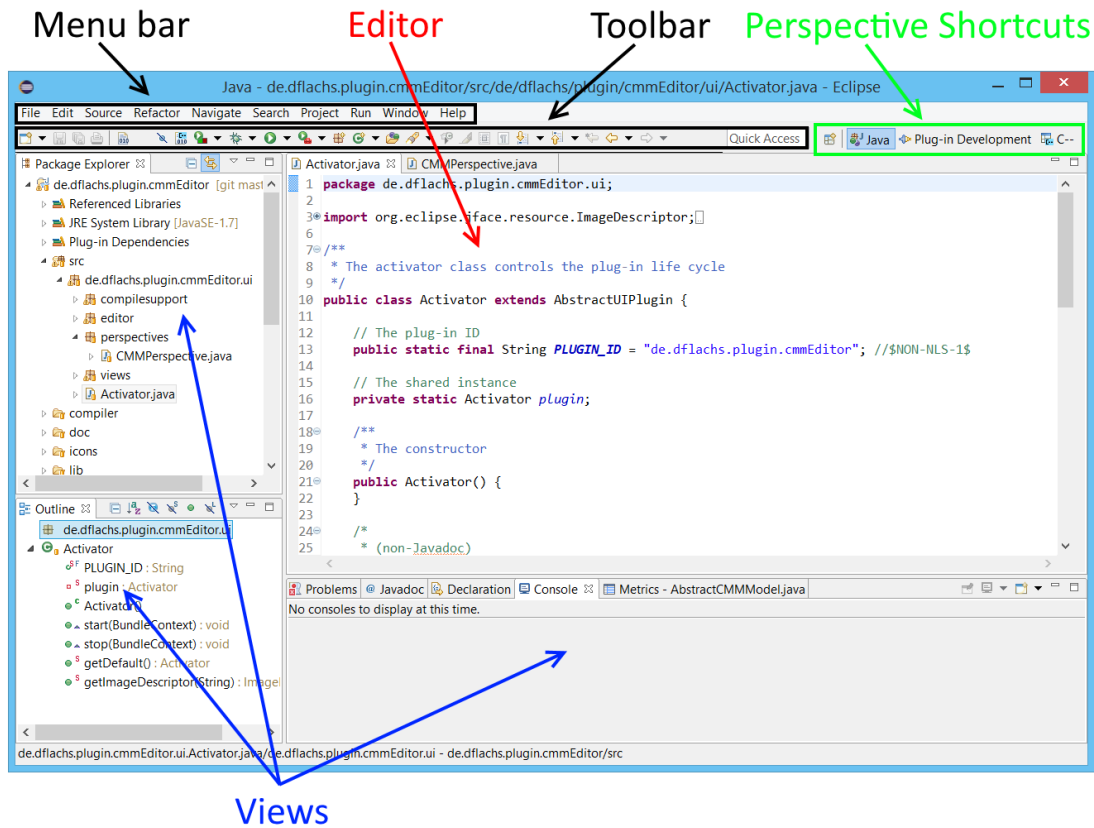


Figure 2.3.: Eclipse workbench window with its components, compare [Ecl15d]

so-called input objects and can be compared to text editors of the operating system. Unlike views, changes made to the input file are not stored on disk until the user explicitly orders the editor to do so. Figure 2.3 shows a screenshot of the Eclipse workbench window, including views and editors.

For building GUI elements, for instance inside a view, Eclipse offers a graphical widget library called *Standard Widget Toolkit* (SWT), which interfaces with the GUI components of the operating system. To simplify the development of certain GUI elements, the UI toolkit *JFace* provides a layer of abstraction which combines SWT basic components as more complex widgets in order to provide support for “features that can be tedious to implement” [Ecl15d].

In order to show the user the specific set of visual components – editors, views, toolbars and menus – he currently needs for his task, Eclipse provides the concept of *perspectives* [Ecl15d]. A perspective specifies the initial layout of the visual

components within the workbench window, e. g., the views to be shown, including their size and location, and thus the allocation of space in the workbench window. The user can customize these default settings, for instance by dragging existing views around or by ordering the workbench to open a new view. Resetting views to their defaults is possible at any time. Examples for perspectives are the “Java perspective” (shows tools useful for Java development) or the “Debug perspective” (shows tools beneficial for debugging programs). Using perspectives, the user can switch between different tasks quite quickly, without the need to open and close the corresponding UI components, such as views, one by one.

2.2.3. Plug-Ins and Features

As mentioned before, the platform itself does not provide any functions for the end user. Its main task is to clear the way for plug-ins. The plug-ins are then responsible for providing the ultimate user functionality. Plug-ins exist for various purposes: Development support for common programming languages, support for team collaboration (e. g. GIT or Subversion) or builders for graphical user interfaces are just some examples. If a desirable feature does not exist yet, it can be contributed to the platform by any user. In order to assist the programmer in writing plug-ins, Eclipse provides an extra plug-in called *Plug-in Development Environment* (PDE).

The Eclipse documentation describes a plug-in as follows: “a structured component that describes itself to the system using an OSGi manifest (`MANIFEST.MF`) file and a plug-in manifest (`plugin.xml`) file. The platform maintains a registry of installed plug-ins and the functionality they provide” [Ecl15d]. Via the `plugin.xml` file, which contains high-level information about the plug-in, such as name, category or a unique ID, the Eclipse runtime can determine the functions a specific plug-in provides. Plug-ins are dynamically loaded by the runtime if their functionality is requested by the user. Hence, plug-ins which are installed but not loaded do not influence the Eclipse application’s memory footprint and thus the user experience. `MANIFEST.MF`, in contrast, contains rather low-level information, e. g., what other plug-ins this plug-in requires (plug-in dependencies) or the plug-in’s classpath. This file constitutes the interface to the OSGi Framework, using the OSGi bundle terminology. Moreover, it points to the plug-in’s

so-called *activator class*. This class provides methods to manage the life cycle of the plug-in, i.e. for starting up and shutting down the plug-in or to manage resources, such as images or settings. The activator class can be omitted – e.g. for very simple plug-ins. A standard activator implementation will then be provided by the platform. If implemented by the developer, it has to extend the Eclipse API class `org.eclipse.ui.plugin.AbstractUIPlugin`.

When deploying a plug-in, it is basically simply exported and packed as a Java Archive (JAR-File) by a wizard inside Eclipse. Multiple plug-ins can be bundled as a *feature*. Features natively provide installation support, i.e. they can be installed via Eclipse’s built-in installation wizard, whereas plug-ins have to be integrated manually.

In order to test a plug-in during development, the programmer does not need to export and install the plug-in again and again. Eclipse provides a feature which starts a new instance of `eclipse.exe` with a test workspace and runs the plug-in in it. On top of that, when executing in “debug mode”, it is possible to make some modifications to the running plug-in, which are then hot-swapped into the running application. Furthermore, debug output of the test instance can be written on the host Eclipse’s console.

2.2.4. Extensions and Extension Points

Extension points are one of the main concepts in Eclipse’s plug-in model. The platform’s workbench defines a set of basic extension points which can be addressed by plug-ins, declaring that the plug-ins contribute a certain functionality to the platform [Ecl15d]. For instance, the `org.eclipse.ui.views` extension point is used by plug-ins which want to contribute a view to the workbench; the `org.eclipse.ui.editors` extension point indicates that the plug-in adds an editor to the workbench. One plug-in can extend an arbitrary number of extension points, thereby contributing multiple functions, e.g. both an editor and a new menu entry.

Extension points have to be listed in the `plugin.xml` file, so that the platform is able to register the plug-in’s functions and load the plug-in if needed. By adding and defining certain XML tags and attributes, the plug-in developer can

specify particular properties of the extension, e.g., the Java class that provides the concrete functions or simply the editor's icon for the user interface.

Since the plug-in model is layered, a plug-in cannot only extend extension points of the workbench but also define its own extension points, which can then be extended by other plug-ins. That way, the aforementioned layered plug-in structure is generated, so that one plug-in depends on one or more others.

3. Plug-In Design

This chapter deals with the high-level design of the C-- plug-in, namely the design of both the C-- editor and the C-- code documentation view. The overall design decisions are derived from the subsequently determined requirements.

3.1. Requirements

This section reveals and describes the requirements for the plug-in in order to support developers using C--.

3.1.1. Writing and Editing Source Code

Writing source code is the main task for programmers who want to develop software in a certain programming language. In order to do so, they can use a text editor which allows them to enter source code. Every source file is basically a simple plain text file stored on hard disk. A text editor is a piece of software which is capable of modifying this file, i. e., load the data, display it to the user, accept the user's modifications and write the modified file back to disk. Furthermore, an editor can be expected to provide features which simplify the modification of files, such as copy, cut and paste.

The user experience when using an editor is considerably influenced by the visual presentation of the file contents. With respect to a source code editor for a programming language, this leads directly to a feature called *syntax highlighting*: The source code is visually presented using different colors and font styles, thereby making it easier for humans to read. Syntax highlighting comprises, e. g., reserved keywords of the language, comments, control-flow statements and variables. With respect to C--, the editor is supposed to provide basic syntax

highlighting of the CMM-specific static keywords. Furthermore, visual accentuation of corresponding brackets is desirable because of the possibility of deeply nested control flow statements (if/else and loops).

Auto indentation denotes a feature which is supposed to make source code easier to read by indenting coherent blocks of code such that the indentation matches the program's structure. In practice, the C-- editor is supposed to feature elementary auto indentation in a way that the current indentation level of the currently edited code block is maintained when the user enters a new line.

Another feature is *autocomplete* which helps the programmer to write code more quickly and efficiently. Moreover, it is supposed to reduce typing mistakes and thus compiler errors. Basically, autocomplete provides a dictionary of all language-specific expressions, i.e., keywords, which are valid in the given context. With respect to C--, if, for instance, the user starts typing “pro”, the editor is supposed to automatically suggest the keyword “procedure”, which the user can insert with only one keystroke.

3.1.2. Compilation

Without IDE support, the work flow for writing and compiling a source file is as follows: write the source code, save the file, open the command line interface (e.g. `cmd.exe` for Windows or `bash` for Linux) and invoke the compiler with the source file. This is quite tedious, as the developer has to switch between different programs and type the compile command with the arguments manually. This process needs to be executed for every re-compilation after every change of the source. In contrast, in an IDE, clicking the “compile” button is all that is required. Hence, the IDE needs to know the location of the compiler in order to automatically pass the current source file to it. The C-- plug-in is supposed to provide some UI component, e.g., a button, which starts the compiler (i.e. the Tasklet factory) with the currently opened source file. Furthermore, the documentation information for a Tasklet (see beneath, Section 3.1.3) should be included in the compiler's output as a compiler comment to ensure that users who do not have the source code but only the byte code can access the documentation anyway.

3.1.3. Documentation of Source Code

Writing functional and maintainable source code does not only include writing correct statements in the source language, but also documenting the code properly. This can be done by inline comments or by external tools. When using Java, *JavaDoc*, for instance, represents a hybrid approach: The programmer includes special comments in his source code which are later parsed and combined as a summarized documentation of the whole code. External tools have the advantage that the documentation is independent from the source files: If a library is built and then deployed as compiled byte code and not in the source format, users of the library cannot read the comments in the source code anymore. If the documentation is independent, they can still access the API specification.

In order to properly document Tasklet source code, an external tool is required for two reasons: Firstly, the language does not provide inline comments yet. Secondly, and most importantly, in the future, it should be possible for users to reuse Tasklets which other users have provided. In order to search for suitable Tasklets in some sort of “Tasklet Market”, descriptions of the Tasklets’ functions have to be accessible.

From a developer’s point of view, the documentation of C++ code should be made possible by some visual component within the Eclipse IDE. All possible programmatic structures of the language – source files, input and output parameters, procedures, parameters and return values – should be representable. This can be compared to the “Outline View”, which Eclipse already natively provides for several programming languages, extended by the documentation feature.

3.2. Component Design

This section presents the general components which have been selected to comprehend the features implied by the desired requirements and shows how these components are designed.

As mentioned in Section 2.2.2, Eclipse provides two basic components for different purposes: views and editors. Editors are suitable whenever editing of source files from disk is required. Taking this into account and including the fact that most

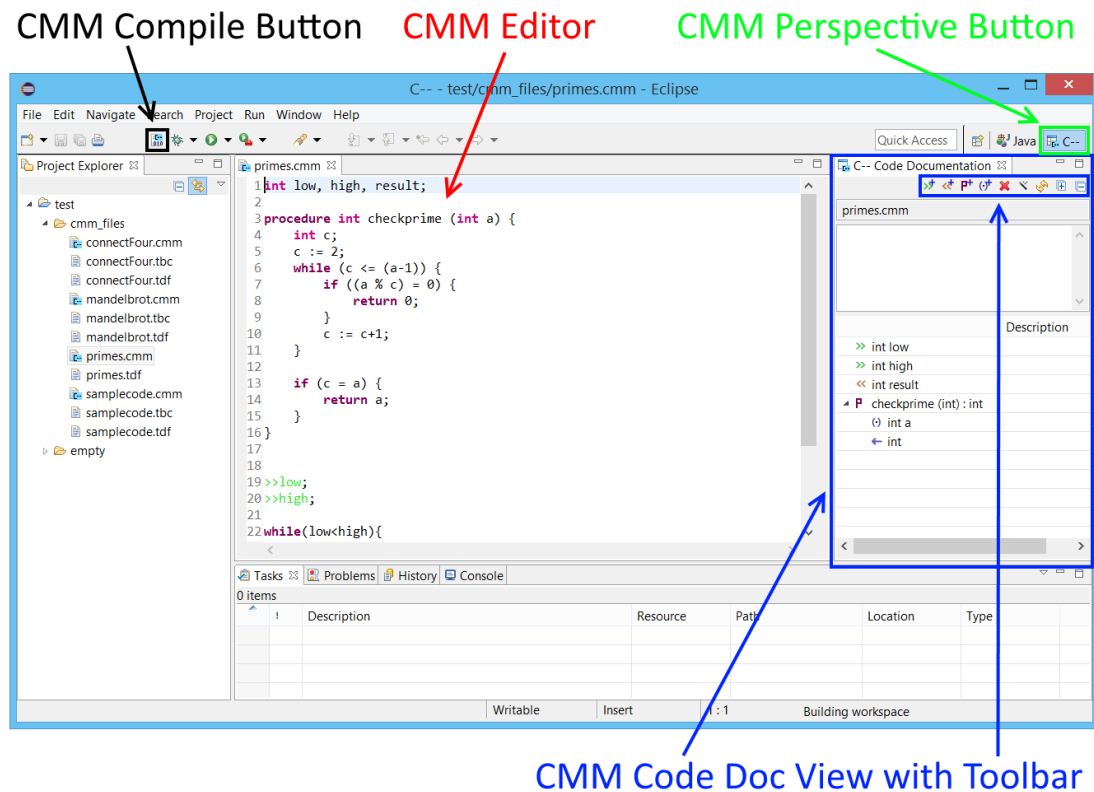


Figure 3.1.: Standard layout of the C-- perspective

likely every plug-in providing support for a certain programming language uses an editor for editing source files, it is quite obvious that an editor should be used to fulfill the “Editing Source Code” requirement. In contrast, in order to meet the “Documentation” requirement it is necessary to visualize the hierarchical source code structure while allowing the user to add descriptive texts to certain source code elements. This is exactly the purpose of Eclipse’s view concept – hence, a view has been selected for the code documentation feature.

To easily allow the user to display all UI components necessary for Tasklet development, a special Eclipse perspective for C-- development was created. It contains the following views: the project explorer on the left side, a console at the bottom and the view for C-- code documentation on the right side. The C-- editor is located between these three views. A button to start the compiler was added to the main toolbar in order to satisfy the “Compilation” requirement. Figure 3.1 provides an overview of the default layout of the C-- perspective.

3.2.1. The Editor

The C-- editor constitutes the central component to edit Tasklet source files. It supports basic operations one would expect from a simple text editor: The user can open a Tasklet language source file, type in, delete and edit text, and save the file. Furthermore, copy, cut and paste operations are supported. On top of that, there is of course programming language-specific support. To achieve that, the editor internally stores a model of the source file which can then be processed, modified and saved.

A syntax highlighting feature is responsible for visually emphasizing C--'s keywords, comments and strings in different font types and colors (see Section 2.1.2). Moreover, when the user places the cursor behind a bracket, the corresponding opening or closing bracket is highlighted by a rectangle enclosing that bracket. The editor also provides context-sensitive content assist, i.e., when the user starts typing and requests content assist via a keyboard shortcut, it is recognized whether the user's input fits any keywords of C--. If so, it gives suggestions and the user can insert one of them by selecting it and hitting "Enter".

3.2.2. The View

The C-- code documentation view is the complementary component to the editor and serves the purpose to enable the user to document the source code he produces in the nearby C-- editor. For each C-- source file the user opens in the editor, the view represents the source file's code structure by an (initially empty) *data model*. The model has a tree structure so that the source file constitutes the (implicit) root of the tree. This is suitable since a C-- source file is a structured object which contains other structured objects (parameters and procedures) in a hierarchical manner. Thus, child elements of the root can be Tasklet input parameters, output parameters and procedures. Only procedures have further children; I/O parameters are always leaves of the tree. A procedure must have one child representing its return value (e.g., `int` or `void`) and can have an arbitrary number of children representing procedure parameters. An example of a model containing three I/O parameters and one procedure can be found in Figure 3.1 on the right side. The user can manually add any of the aforementioned elements to the model by clicking the corresponding icons in the toolbar. In order to simplify

the documentation process, it is also possible to let the view automatically create the data model from the opened source file: When the programmer has finished writing his program, he can tell the view to build the model from his source code. As an addition of the tree structure, a table is used to extend the amount of information which can be displayed by the view: The tree is located in the first column of the table and each tree node represents one table row. The second column is used to provide a description for each tree item. The user can edit all displayed elements – both tree elements and corresponding description fields – by double-clicking. Furthermore, deletion of elements and expanding/collapsing the whole tree is possible via clicking the associated buttons. The view also provides a larger text field for a description of the overall source file. Each user change to the data (adding, deletion, modification) is immediately reflected in the underlying model. One source file is associated with exactly one model. When the user switches between C++ editor tabs (meaning between different source files), the model of the “old” source file is stored for later reloading and the model for the newly opened/activated source file is loaded and displayed.

4. Plug-In Implementation

While the previous chapter deals with the overall and high-level design of the C-- plug-in, this chapter addresses the question how the plug-in's concrete implementation is realized.

4.1. General Plug-In Properties

The following paragraphs describe how the C-- plug-in interfaces with the Eclipse platform, which is mainly defined in the plug-in-specific files `plugin.xml` and `MANIFEST.MF` (see Section 2.2.3 for a general introduction).

Amongst other things, `MANIFEST.MF` specifies the `Bundle-Activator` for the plug-in, i.e. the Java class responsible for the plug-in's life cycle management. Furthermore, next to `Require-Bundle`, a list of bundles (i.e. plug-ins) this plug-in depends on is given. The `Bundle-Classpath` specifies the plug-in's classpath, which is important if external libraries are included. The complete `MANIFEST.MF` file can be found in the Appendix, see Listing B.1.

The C-- plug-in uses a total of six extension points, defined in `plugin.xml`:

`org.eclipse.ui.editors`: This extension point indicates that the plug-in contributes an editor to the Eclipse platform. Via the XML attributes, the editor's name ("C-- Editor for Tasklet Language"), display icon and supported file extensions (`*.cmm`) can be specified. Moreover, a unique ID must be assigned to globally refer to the editor. The `class` attribute points to the associated Java class which is responsible to finally implement the concrete editor functions, such as Tasklet language-specific syntax highlighting. Listing 4.1 shows the `editors` extension point as an example.

`org.eclipse.ui.views`: This extension point announces that a new view is added to the Eclipse workbench – in this case to support C-- code documentation. Just like the `editors` extension point, it is mandatory to

Listing 4.1: Excerpt of “plugin.xml”

```
1 <extension
  point="org.eclipse.ui.editors">
  <editor
    name="C-- Editor for Tasklet Language"
5    extensions="cmm"
    icon="icons/cmm_file.ico"
    contributorClass="org.eclipse.ui.texteditor.
      BasicTextEditorActionContributor"
    class="de.dflachs.plugin.cmmEditor.ui.editor.CMMEditor"
    id="de.dflachs.plugin.cmmEditor.ui.editor.CMMEditor">
10  </editor>
</extension>
```

provide a globally unique ID, a name (“C-- Code Documentation”), an icon and a responsible Java class. Furthermore, Eclipse’s views are organized in categories to group associated views together. Thus, the C-- view has to be added to a category as well. Since there are no pre-existing views for Tasklet development support, a new category called “C--” was created and the C-- view was added to it.

org.eclipse.ui.perspectives: By adding this extension point, new perspectives are added to the workbench. The precise layout of the perspective, in this case the “C-- perspective”, is provided by the Java class specified by the `class` attribute of the extension point.

org.eclipse.ui.perspectiveExtensions: In contrast to the extension point perspectives, `perspectiveExtensions` does not add a new perspective but extends an existing one. Eclipse is widely-used especially for Java development and, on top of that, a Java library for the Tasklet system already exists. Therefore, it is reasonable to assume that both the Java and the C-- perspective are likely to be used in parallel by developers. Hence, two menu shortcuts for the C-- perspective and the C-- view were added to the Java perspective, so that both the view and the perspective can be displayed via `Window >> Open Perspective` and `Window >> Show View`.

org.eclipse.ui.menus: This extension point is used to add menus, toolbars and buttons to the workbench window. The extension points menus and commands are tightly coupled, so that the behavior of a menu entry or

button is defined by the associated command. In this case, a toolbar with a single button, which calls a command responsible for compiling C++ source files, is added to the Eclipse main toolbar.

org.eclipse.ui.commands: According to the Eclipse documentation, a command is “an abstract representation of some semantic behavior, but not its actual implementation” [Ecl15d]. The concrete implementation is provided by `org.eclipse.ui.handlers`, so that the same command can have different implementations, depending on the context. Here, the command’s default handler executes the compile action independently from the context. Hence, no specific handlers except the default handler have been defined.

4.2. Package and Folder Structure

The Java package `de.dflachs.plugin.cmmEditor.ui` is the plug-in’s root package. Since the plug-in contributes functions and elements to the GUI, the last subpackage is called “ui”. This follows the Eclipse convention for naming plug-in packages. Each of the four subordinate packages encapsulates one of the plug-in’s high-level functions: The *compilesupport* package encompasses all classes in charge of the compilation process; the packages *editor*, *perspectives* and *views* offer exactly what the names already imply: the editor’s, the perspective’s and the view’s classes with their specific functions. Since the editor package consists of quite many classes with different purposes, it contains two subpackages: *cmm* (Tasklet language-specific classes) and *util* (general utility classes). Furthermore, the view package also has a subpackage called *model* which provides the classes for the underlying data model of the view.

In addition to the Java source files, four folders containing external resources are included in the plug-in: The folder *compiler* contains the C++ compiler as an executable file, which can be invoked by the user clicking the “compile” button. *lib* contains the external libraries which are used by the plug-in: “JSON.simple” for JSON serializing and parsing, and “Apache Commons I/O” for handling paths and file names. The folder *icons* stores all graphical icons which are used to represent plug-in functions or data objects to the user in an intuitive manner. *doc* contains the automatically generated JavaDoc files for the Java classes of the plug-in.

4.3. The CMM Editor

This section deals with the C-- editor and thus all Java classes in the package `de.dflachs.plugin.cmmEditor.ui.editor`. The description starts with the central editor class and discusses all other associated classes afterwards.

The editor's main class `CMMEditor` inherits from the Eclipse API class `org.eclipse.ui.editors.text.TextEditor` which constitutes the most concrete version of Eclipse's text editor classes. Since the C-- editor is a "normal" text editor with certain language-specific adaptations, this API class was chosen, hereby using all of its pre-implemented functions, such as copy, cut and paste. This does not affect flexibility much because most of the inherited methods can be overridden if needed. There are various other, more abstract classes above `TextEditor` in the API's type hierarchy. This is a common design approach of the Eclipse API: It provides multiple interfaces and classes which are organized hierarchically. The "root class" is the most abstract one and provides the least functionality but has the most possibilities for developer-made customization. Descending in the type hierarchy, the classes become more concrete and functional, whereas the options for implementation adjustment decrease. The following example shows the hierarchy of Eclipse's text editor classes (without interfaces) [Ecl15b]:

```
java.lang.Object
  org.eclipse.core.commands.common.EventManager
    org.eclipse.ui.part.WorkbenchPart
      (*) org.eclipse.ui.part.EditorPart
        org.eclipse.ui.texteditor.AbstractTextEditor
          org.eclipse.ui.texteditor.StatusTextEditor
            org.eclipse.ui.texteditor.AbstractDecoratedTextEditor
              org.eclipse.ui.editors.text.TextEditor
```

An Eclipse editor has to implement the interface `IEditorPart`. In the above class hierarchy, `EditorPart` is marked with an asterisk because it is the first concrete class which represents a thorough and functional editor.

In order to add functions and to configure the editor, it is linked to a couple of other classes, which will be discussed in the subsequent paragraphs.

4.3.1. Document Provider, Partitioner and Partition Scanner

Every editor needs a *document provider* which is responsible for supplying the editor with its input. The Eclipse API has two classes representing documents and editor inputs respectively. On the one hand, an `IDocument` represents “text providing support for text manipulation, positions [...]” [Ecl15b] and various listeners. On the other hand, `IEditorInput` “is a light weight descriptor of editor input, like a file name but more abstract. [...] It is a description of the model source for an `IEditorPart`” [Ecl15b]. Here, the class `CMM-DocumentProvider` constitutes the editor’s customized provider. It extends `org.eclipse.ui.editors.text.FileDocumentProvider`, the Eclipse API’s standard implementation of a provider specialized for file resources, such as source code files. This class only overrides one method, namely `createDocument`, which creates a document from a given input element (i. e. a representation of a file on disk). Furthermore, this method sets up the *document partitioner*: Whenever Eclipse handles a document, it parses and partitions it. Partitioning means that the document is divided into non-overlapping sections, so that every section has a specific content type. The C-- editor only defines two content types: multiline comments¹ and “everything else”, the default content type, which is normal C-- code. For each content type, a different editor behavior can be implemented. For instance, some keywords for multiline comments could be defined, so that certain common expressions in a comment could be highlighted.

Each document has its own partitioner which is connected to the document and vice versa. Partitioning of a document is realized by a *partition scanner* which operates on the document with a defined set of partition rules. Here, the plugin class `CMMPartitionScanner` is responsible for this behavior. The scanner supplies the partitioner with tokens, so that different content types generate different tokens, which can then be distinguished by the partitioner.

4.3.2. Editor Configuration

Most of the editor customization is done via the editor’s configuration class. With respect to the C-- editor, the configuration class is called `CMMConfiguration`.

¹Multiline comments are not part of the language yet, nevertheless they are already implemented in the plug-in.

Listing 4.2: Excerpt of “CMMCodeScanner.java” (simplified)

```
1 public CMMCodeScanner() {  
    List<IRule> rules = new ArrayList<IRule>();  
  
    IToken keywordToken = new Token(new TextAttribute(  
5      new RGB(127, 0, 85), null, SWT.BOLD));  
    IToken otherToken = new Token(new TextAttribute(  
      new RGB(0, 0, 0)));  
  
    WordRule wordRule = new WordRule(new WordDetector(), otherToken);  
10    wordRule.addWord("while", keywordToken);  
    rules.add(wordRule);  
  
    IRule[] result = new IRule[rules.size()];  
    rules.toArray(result);  
15    setRules(result);  
}
```

It sets up the content assist feature as well as the editor’s syntax highlighting behavior and some minor editor properties, e.g., how many “undo” steps are possible.

The editor’s capability for *syntax highlighting* is probably the most noticeable visual feature. To understand how highlighting the syntax of source code works, one has to realize how Eclipse text editors modify text: When the user applies changes to the source code, parts of the text have to be redisplayed in order to update the UI. The task of calculating which parts of the text have to be updated is known as calculating damage. As soon as the damaged parts are determined, they have to be “repaired” to keep the UI and the text document synchronized. The whole process is known as “reconciling” [Ecl15d]. This can be achieved by a so-called `DefaultDamagerRepairer` – an API class – which also defines how the text is displayed. This means that syntax highlighting is tightly coupled with the reconciling process. Each content type can have its own damager/repairer, so that it is again possible to handle different content types differently.

Similar to document partitioning, a damager/repairer also depends on an associated scanner which supplies it with tokens. Since the scanner for C++ source code (the “default content type”) is the more interesting one, the `CMMCodeScanner` will be presented as an example. The scanner class defines rules to recognize the following syntactical constructs: C++ keywords, data types, single-line comments,

character and string literals, and I/O operations (>> and <<). Each rule is associated with a token which is returned if the rule applies. Tokens also carry information about how to highlight content which is marked with this token. For instance, the keyword `while` is printed in bold font and colored purple. Listing 4.2 shows a simplified version of how the rule for this keyword is set up.

Another feature activated by the editor configuration is *context-sensitive content assist*, which supports the developer in writing C-- source code by offering suggestions for certain language-specific keywords. The class responsible for this functionality is called `CMMCompletionProcessor`, which implements the API interface `IContentAssistProcessor`. Basically, the user types something, for instance “pro”, and requests suggestions by pressing `Ctrl`+`Space`. Next, the completion processor tries to find the beginning of the word by going backwards from the current cursor position, until a whitespace character is encountered. The string between the current cursor position and the beginning of the word is stored as the prefix. After that, the dictionary containing all of C--’s reserved words is searched for keywords that start with the prefix typed by the user. If one or more words apply, they are added to a list of suggestions which is then displayed to the user. If there are no suggestions available, the user is simply given a list of all words in the dictionary. The user can select one of the suggestions. After hitting the return key, it is calculated which string (called “completion”) needs to be added to the user’s prefix so that a concatenation of the prefix and the completion equals the selected suggestion. Finally, the completion string is inserted after the prefix and the user can continue to write his source code. Instead of manually requesting suggestions, it would also be possible to tell the completion processor to automatically activate the suggestions pop-up when the user enters a certain character. However, this behavior does not seem reasonable in the current version of the plug-in since this would require a lot more intelligence and rules with respect to context sensitivity of the feature.

4.4. The CMM Code Documentation View

This section presents the implementation of the C-- code documentation view along with the involved Java classes in the package `de.dflachs.plugin.cmmEditor.ui.views`. From a plug-in developer’s point of view, the C-- view

is probably the most complex part of the plug-in, not only considering total lines of source code. This is due to the fact that there is no API standard implementation of a view supporting code documentation which can simply be adapted, as it is the case for the editor. The view provides a lot of very CMM-specific functions which of course all have to be implemented by hand. In contrast, the Eclipse API offers quite extensive support for GUI elements which are used within the view.

Firstly, the view's underlying data model is presented. Afterwards, the central view class is explained, followed by other associated classes.

4.4.1. Data Model

The view's purpose is to display the code structure of the currently opened C-- source file. Furthermore, it should allow the developer to add comments and descriptions to certain structural elements of the source code which are represented by the view. In order to store, manipulate and display the documentation information given by the user, it is reasonable to build a data model describing a Tasklet source code file. General information about the model has already been given in the "Design" Chapter, see 3.2.2. Instances of the five model classes represent the C-- source file itself, its I/O parameters and its procedures with their parameters and return values. Figure 4.1 shows the structure of the data model. The abstract superclass of all concrete classes is `AbstractCMMModel`. It defines the string attribute "description" which all model classes have in common and which stores the content of the description field for each object given by the user. Furthermore, the superclass orders its subclasses to implement two methods: `getParent()` returns the immediate parent of an object, e.g., the associated procedure for a `CMMProcReturn` object. `getRoot()` always returns the `CMMSourceFile` the given object belongs to. It is also possible to ask a model object for its children. I/O parameters and procedures are child elements of a source file object; parameters and return values are children of a procedure object. All model classes are equipped with getter and setter methods for their attributes; when reasonable, the setter methods perform logical checks. For instance, they test whether an input and not an output parameter is passed to the method which is responsible for setting an input parameter.

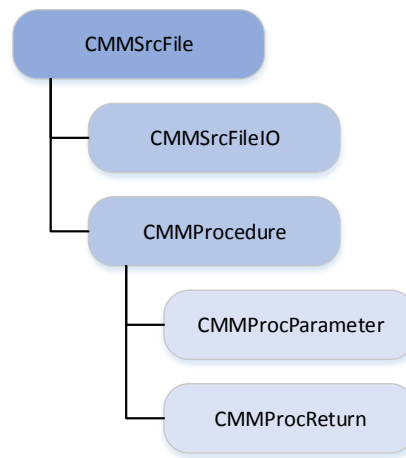


Figure 4.1.: The view's data model

Since a data model is always associated with one C++ source file, it has to be stored and loaded dynamically, depending on the source file which is currently displayed in the editor. Saving the model is realized via serialization and, i.e., the model with all its objects is transformed into a unique string, which can be stored in a plain text file on the hard disk. Vice versa, when a model has to be loaded from disk, the process is performed backwards: Objects are built from their string representation. This is called deserialization. The string representation has a special, structured format called JSON (short for JavaScript Object Notation), which is a standard data transmission format [JSO15]. JSON was chosen for two reasons: Firstly, it is standardized, language-independent (despite the name) and libraries in different programming languages already exist. Secondly, it is human-readable and thus easier to modify and debug. Each of the five model classes has two methods for serialization and deserialization: An object method `serialize()` which returns a `JSONObject` and a static method `deserialize(JSONObject object)` which returns an instance of the model class for a given `JSONObject`. If, for instance, a C++ source file object is serialized, all of its children are serialized recursively by calling their `serialize` methods. The Listings B.2 and B.3 in the Appendix show the methods for (de)serialization of the `CMMProcedure` class. The plain text file storing the model has the file extension “tdf” (Tasklet Documentation File) and is located in the same folder as

the associated source file (extension “cmm”).

As the model is supposed to be dynamic, meaning that it can be modified, any model change has to be announced. This is necessary because the GUI which displays the model has to be updated after every change, for example. Furthermore, model changes have to be stored permanently. Therefore, after every modification, the model has to be serialized and a new version of the linked tdf file must be written. To achieve that behavior, the *observer pattern* was implemented: `AbstractCMMModel` extends `java.util.Observable`, so that classes implementing the interface `java.util.Observer` can register to any model class as observers. After every model modification, the observable’s inherited methods `setChanged()` and `notifyObservers()` are called to announce a change. This causes the observers to call their local `update()` method. With respect to the plug-in, the only model observer is the tree viewer’s content provider, which will later be described in more detail.

4.4.2. Main View Class

The central class which constitutes the C++ code documentation view is called `CMMCodeDocView` and extends `org.eclipse.ui.part.ViewPart` from the Eclipse API, which is the view equivalent to the editor’s `EditorPart`. The view basically consists of four different GUI widget classes: At the top, there are two instance of an SWT text widget; one to display the name of the C++ source file currently associated to the view, and one as an input field where the user can type a general description for the Tasklet currently open in the editor. Beneath these two fields, the data model is displayed by a combination of a `JFace TreeViewer`, an SWT `Tree` and SWT `TreeColumns`. Since these components must be initialized whenever the workbench wants to open the view, a method is required for these preparation actions: `createPartControl` is inherited from a superclass and needs to be overridden. The initialization of all involved components, i.e., both GUI and model elements, have to be executed here. This initialization method performs six consecutive steps which are depicted in the following Figure 4.2.

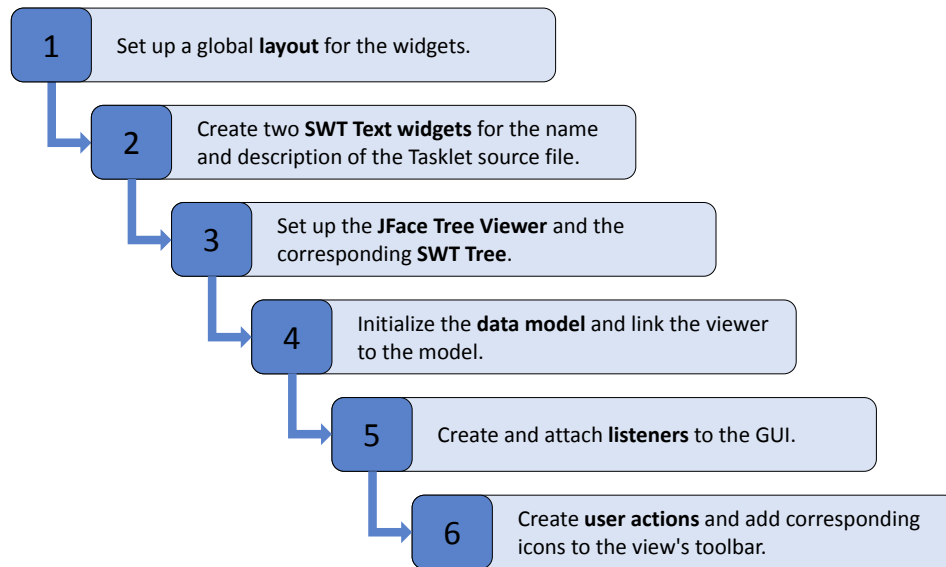


Figure 4.2.: The steps for the initialization of the view

Subsequently, the initialization steps are explained in more detail.

Firstly, setting up the view's global layout in **step 1** initializes a new grid layout which arranges the two text fields and the tree viewer vertically, one below the other.

Secondly, the two SWT text widgets are created in **step 2**. The upper text widget is only used to display the file name of the current C++ file, thus, editing is disabled. In contrast, the second text widget is enabled for editing since it allows the user to enter a description for the source file. The text content of the widgets is initialized with a standard text which is overwritten as soon as the model is loaded.

Next, in **step 3**, several required components of the UI frameworks are initialized. This is the most extensive initialization step since it requires interconnecting instances of three different API classes: At first, a new `org.eclipse.swt.widgets.Tree` – an SWT widget to display a hierarchical tree structure – is created. Next, the tree is encapsulated by an `org.eclipse.jface.viewers.TreeViewer`, which provides a higher abstraction level than the low-level SWT tree. Lastly, two instances of `org.eclipse.swt.widgets.TreeColumn` are

created in order to provide a layout with two columns for the tree: one for the tree and one for the associated description fields. Furthermore, “editing support” is added to the two columns. This enables editing and defines how values of different tree items are set. Finally, both a label provider and a content provider, which determine which content (i.e. model data) is displayed how, are added to the JFace viewer. A more detailed description of the providers and the editing support can be found in Section 4.4.3. It should be noted that at the end of this step, the viewer is empty and does not contain any model data yet.

Afterwards, **step 4** sets the data model for the viewer, i.e. it provides it with content to display. The model is obtained as follows: As the view is always connected to the currently displayed C++ source file, the model for this active source file must be loaded. If the active editor is not a C++ editor, the view remains disabled because it is only useful when combined with a C++ editor. Otherwise, the active C++ editor is asked for its document’s path and file name, which is for instance “code.cmm”. From the source file’s name, the documentation file’s name is constructed by removing the source file extension “cmm” and appending the doc file extension “tdf”. In the example, the result would be “code.tdf”. Next, the model setup method checks whether the doc file “code.tdf” exists. If so, the file contains the previously saved model as a JSON string which is handed over to the model’s `deserialize` method (see Section 4.4.1). The reconstructed model is then set as the input data for the viewer. The method which sets the viewer input also equips the new model with an observer so that modifications can be recognized.

The subsequent **step 5** attaches listeners to several parts of the plug-in. These listeners are solely used to enable the plug-in to react to events which are triggered by user input and have nothing to do with the model change listeners from above. The most important listener is a `org.eclipse.ui.IPartListener2`, which listens “to part lifecycle events” [Ecl15b]. With this listener, the view is informed about every workbench part, like a view or an editor, that changes in the workbench window. This is important since the use of the C++ view is only reasonable if the user is editing a Tasklet source file. `IPartListener2` prescribes several methods from which two are meaningful for the behavior of the plug-in: `partActivated` is called whenever a view or an editor is activated, i.e., it is opened and/or gains focus, e.g., because the user clicks on it. In the plug-in’s

implementation, the method will only start to take action if the part which has just been activated is an editor. If the editor is not a `CMMEditor`, the view will be disabled, since this indicates that the user is not editing a C-- source file at the moment, i.e. the view is not needed. Otherwise, it is checked whether the C-- editor which has just been activated equals the previous editor. This step is necessary because if the user switches between several views and one and the same C-- editor, every re-activation of this editor triggers an activation event. However, this event does not need to be handled since the model associated to the editor's source file is still valid. In contrast, if the previous editor does not equal the current one, the model for that editor's source file has to be obtained. To do that, the same actions as in step 4 are performed. The second important method, `partClosed`, is called when a workbench part is closed. If the part which has just been closed was a C-- editor, it is checked whether another C-- editor is now active. If not, the view is disabled. This is especially the case if the closed C-- editor was the last editor, so that no other editor gains focus afterwards.

The **6th and last step** in the initialization process is responsible for setting up the GUI elements and the underlying functions for the view's toolbar, e.g., for adding and deleting model elements. Firstly, the logic for the user actions are created, i.e., the implementation of everything that has to happen within the view and the model – for instance, if the user wants to add a new procedure. Secondly, the actions' implementations are linked to the GUI by supplying them with intuitive icons and adding them to the view's toolbar. The following actions are implemented and can be invoked by the user: The “Add Input Parameter” action adds a new Tasklet input parameter (`>>`) to the model. Respectively, “Add Output Parameter” is used to add a new Tasklet output parameter (`<<`). Similarly, procedures and procedure parameters can be added. Source file parameters and procedures are always automatically added to the root of the model, the source file. Procedure parameters can only be added to an existing procedure, thus a procedure has to be selected beforehand. Any of the aforementioned actions creates a new, generic model entry of the particular type. For instance, if an input parameter is added, the new entry is displayed as “DATATYPE NAME”. The first word is a placeholder for one of C--'s data types and the second is supposed to be replaced by the parameter's identifier. There is no pop-up or the like requesting

name and data type of the newly added parameter. This is very straightforward and does not force the user to work through one input dialogue after the other. In addition to extending the model with new elements, it is also possible to remove elements from the model. When deleting an element which has child elements, the children are removed recursively. This is the case for procedures and their return values and parameters. Furthermore, the user can select multiple model items at once to delete all of them. The deletion action can be executed either by clicking the icon or by pressing the **Del** key. The user action “Parse Model from Source File” is supposed to significantly simplify the documentation process for Tasklet developers. This action takes the source code of the current C++ editor, parses it for parameters and procedures which can be represented in the view and automatically builds the model. The detailed implementation of the source code parser is documented in Section 4.4.4. Moreover, there are also two actions for expanding and collapsing the whole model tree in order to display the minimum or the maximum set of model elements. The command is delegated to the JFace viewer’s standard implementations `expandAll()` and `collapseAll()`. Furthermore, there is an action for manually refreshing the complete tree viewer, i.e., to reload the entire model and re-initialize the viewer with it. Normally, this should not be necessary, but it can be helpful if the view does not update certain parts in special cases which might not be covered by the listeners.

4.4.3. Content Provider, Label Provider and Editing Support

The three classes described in this section are associated with the C++ view and are vital for its successful operation.

As the name implies, the *content provider* (class `CDVContentProvider`) is necessary to supply the JFace tree viewer with content obtained from its model and to enable it to navigate through the model hierarchy. For instance, it implements methods returning the parent or child elements of a given arbitrary model element. The logic to handle distinct types of model elements differently has to be implemented by the plug-in programmer since the provider cannot “know” the model and its structure natively. For instance, the method `getChildren` returns procedure parameters and one return value for a given procedure, but `null` for a Tasklet input parameter.

Moreover, the content provider implements the interface `Observer` from the `java.util` package, which poses the counterpart to the observable model objects. Hence, the provider has an `update` method which is called whenever a model object, such as a procedure, notifies its observer that some of its attributes, e.g. the description text, has changed. Depending on the type of model object, the update method decides which parts of the view have to be refreshed. For instance, if a procedure's parameter changes, not only the parameter's entry has to be refreshed but also the parent procedure, as changing a parameter also affects the procedure's signature.

Furthermore, the provider possesses a method which writes a given model to a certain tdf file, and another method which reads a model from a tdf file. For this, it accesses the model's methods for serialization and deserialization.

Lastly, the `inputChanged` method is called as soon as the viewer's input, i.e. the model, is set via its `setInput` method. In this case, we use the provider's method to dispose of the old model's listeners and attach new listeners to the new input.

While the content provider is responsible for supplying and navigating the model on a logical level, the *label provider* (class `CDVLabelProvider`) offers the visual representation of tree items. For a given tree item (meaning every model object), the label provider can be asked for an icon visually representing the item and for a string textually representing the item's data. This function is implemented in the provider's methods `getImage` and `getText`. For instance, an output parameter is depicted by an icon showing two orange angle brackets pointing to the left. An output parameter's text is "`<data type> <identifier>`". Since the view combines a tree viewer and a table with two columns, it is necessary to distinguish images and texts for the two columns. The first column is supposed to contain the tree, in which each model item is depicted by its icon and its string representation. The second column contains the corresponding description text. To achieve that, two methods are used: `getColumnImage` simply returns `getImage` for the first column and `null` for the second. `getColumnText` returns `getText` for the first column and the model element's description text for the second column.



It should be possible for the Tasklet developer to edit the model data displayed in the two columns of the view, e.g., to add descriptive texts to the proce-

dures. The user can edit a cell in the viewer's columns by double-clicking it. To enable editing, each column is provided with a customized version of `org.eclipse.jface.viewers.EditingSupport`. There are two essential methods, `getValue` and `setValue`. The former is called when the user double-clicks an entry in order to edit it. The latter is invoked when the user finishes editing and stores the user's modifications by updating the corresponding model element. For Tasklet I/O parameters and procedure parameters, the `set` method expects the user to type in two words, separated by a space character. The former will be interpreted as the parameter's data type and the latter as its identifier. If there is no space character, the modification will be rejected. With all other model elements, the method is more tolerant, so that user input validation is not performed.

4.4.4. Model Builder

The class `CDVModelBuilder` provides functionality to automatically create a C-- data model, i.e., an instance of `CMMSrcFile`, from a given and syntactically correct C-- source file. This feature is expected to significantly decrease the Tasklet developer's documentation effort since it is not necessary to add every single parameter and procedure by hand. Instances of this class are initialized with a `CMMEditor`. They request the editor's document (i.e. the source code). Afterwards, parsers based on regular expressions are used to find Tasklet input and output parameters as well as procedure signatures within the code. The parsing results are used to build a new model which can be requested by the `getModel` method. The class's functionality is divided into several methods: `parseSrcFileParameters` scans the source code for expressions like `>>identifier;` or `<<identifier;` and returns a list of Tasklet input or output parameters. Finding the expressions and extracting the identifier is not that difficult. However, not only the identifier of an I/O parameter is needed but also its data type. To achieve this, the auxiliary method `getIdentifierDatatype` was implemented to find the declaration of a given variable's identifier and to extract the data type. `parseProcedures` searches the source code for procedure signatures like `procedure void execute (int a)` and returns a list of completely initialized `CMMProcedures`, including all parameters and the return type.

4.5. The CMM Perspective

The C-- perspective is implemented with only one class which is located in the package `de.dflachs.plugin.cmmEditor.ui.perspectives`. The class implements `org.eclipse.ui.IPerspectiveFactory` and thus the method `createInitialLayout`. Successively, views, action sets (in principle toolbars), and shortcuts for views and wizards are added to the perspective by this method. The project explorer view is displayed on the left-hand side and the C-- code documentation view on the right. Between the two, there is space for showing the editors. A combined view containing Eclipse's views for task lists, problems and the local history is shown beneath this editor placeholder, together with a generic console view. The initial set of action sets is quite minimalist as it only comprises the Eclipse default launch action set (the buttons for running and debugging the application). Furthermore, shortcuts to access associated perspectives and views quickly are added for the Java, the debug and the team synchronization perspectives as well as for the C-- code doc view, the history view, the console view and the outline. Shortcuts for wizards for new projects, folders and files are added to the   menu.

4.6. CMM Source File Compilation

This section explains the interaction with the C-- compiler (i.e. the Tasklet factory as an exe file) in order to compile source files on click. As a preliminary remark, it has to be mentioned that there is currently no functional “cmmcompiler.exe” since the Tasklet factory is implemented using sockets. Therefore, an executable “dummy compiler” is used to simulate the real compiler's behavior.

The class responsible for this functionality is called `CMMCompileSupport` and is located in the `de.dflachs.plugin.cmmEditor.ui.compileSupport` package. It implements the interface `org.eclipse.core.commands.IHandler`, so it can be executed by an Eclipse command – in this case, the “compile” button in the main toolbar. The class's two methods `isEnabled` and `isHandled` return Boolean values which indicate whether the handler is enabled and capable of handling the event. They only return true if the currently active

workbench editor is an instance of `CMMEditor`. As a consequence, clicking the “compile” button does not have any effect if the current source file is not a C++ file.

The compiler takes two parameters: a folder path to the location of the cmm source file (and implicitly the associated tdf doc file) and the name of the source file without file extension. When invoked, the compiler takes the file path and searches for files with the given name and the extensions “cmm” and “tdf”.

The handler’s essential method is called `execute`, which is invoked as soon as the user clicks the “compile” button. Firstly, the method obtains the current source file from the workbench’s active editor. Then, the source file’s absolute folder path and the source’s file name without its extension are determined. Next, it is checked whether a documentation file corresponding to the source file already exists. If not, it is created with an empty model in order to prevent compiler errors because of the missing file. Next, the `compiler.exe` is invoked with the two previously determined command line parameters, thereby forwarding the compiler’s output to the console view for the user to see. If no errors occur, the compiler generates the Tasklet byte code and stores it along with the information from the documentation file in a `tbc` file in the same folder as the source file.

5. Summary

This chapter summarizes the content of the thesis and gives an outlook of how the plug-in could be improved and extended in future versions with respect to known open issues and additional, desirable features.

5.1. Conclusion

The thesis documents the design and development of an Eclipse plug-in which is supposed to support developers in writing source code in the Tasklet language C--. The plug-in satisfies the high-level requirements from Section 3.1: It is possible to edit C-- source files with the C-- editor integrated in the Eclipse IDE. Considering features like autocompletion or syntax highlighting, the plug-in's editor is significantly more comfortable than any generic text editor of the operating system. Furthermore, the Tasklet developer is able to document his source code using the nearby C-- code documentation view. The view has a simple and easy-to-use user interface. The data model representing the source code is shown by means of a tree structure in a hierarchical manner. All necessary basic modification operations, such as adding, renaming and deleting model objects, are provided to the user. Moreover, the model can be automatically obtained from the corresponding source file by parsing. For each source file, the model information is converted to a JSON string and stored in a text file. Thereby, it is not only accessible and readable for the plug-in but also for other applications, since JSON is a standardized data exchange format. After development and documentation have been finished (or during that process), the programmer can invoke the C-- compiler which creates a Tasklet byte code file from both the source code and the documentation file. In the current version, the compiler is only a dummy with no mentionable functionality, which will be replaced with a real compiler in future versions of the plug-in. The plug-in is completed by the C-- perspective which comprises all aforementioned features and arranges them visually.

5.2. Outlook

5.2.1. Open Issues

In summary, the plug-in extends the Eclipse IDE essentially with respect to supporting C++ development. However, the plug-in still has to demonstrate its capability in everyday use, outside the plug-in development environment. An organized and comprehensive testing process was outside the scope of this thesis, so this could be a starting point for future work. Nevertheless, two existing issues have already been identified:

Firstly, when the user instructs the view to automatically build a model from the C++ source file, the `CMMModelBuilder` does not recognize a certain special case regarding the syntax. For instance, it is syntactically possible to write the following statement: `<<proc()`, where `proc` is a Tasklet procedure. The procedure is called and its return values are added to the array of output parameters. This expression cannot be found by the currently implemented regular expression which is used to parse the source code. Moreover, the model is not intended to reflect this special case explicitly. Normally, the identifier after the `<<` operator is expected to refer to a variable, not a procedure. It can be considered another problem of the model builder that it does not perform advanced syntax checks of the source code. In the current implementation, for instance, it would be possible that a procedure signature is recognized although there is no associated body surrounded by `{` and `}`.

The second issue concerns the performance of parsing a tdf file and rebuilding the model from it. When the model size increases, e.g., with long source files containing many procedures, the tdf file grows as well. Since JSON, which is used to store the model in the tdf file, consists of key/value pairs and is human readable, it is also quite verbose and produces overhead for each new key identifier. On the one hand, the conversion from model data, which are in principle Java objects, to JSON works quite fast. On the other hand, the opposite direction, creating Java objects from a JSON string, takes a long time if the model is sufficiently large. In the current implementation, the user interface is blocked while the tdf is parsed. To improve that, it should be checked whether it is possible to load the model in another thread in order to perform this process in the background and in parallel. As serialization and deserialization are handled by the `JSON.simple` library,

the internal processes cannot be optimized. Nevertheless, it would be possible to search for other JSON libraries which probably offer a higher performance.

5.2.2. Additional Features

Beyond the issues, there are also ideas for additional, desirable features, which could be tackled by future work and might then be included in the next versions of the plug-in. With regard to functionality, there is practically no limit for imagination since the existing plug-ins for the well-known programming languages are very advanced and highly comfortable.

In the current version, the C++ plug-in does not provide any possibility for customization from the user side. For instance, normally, the compiler is stored somewhere in the file system, not at a specific location in the Eclipse plug-in folder. Currently, it is not possible for the user to specify the compiler's location. This could be solved by a *preferences page* where all possible settings are shown and can be modified. Other plug-ins' preferences pages can be found in the `Window` » `Preferences` menu. Of course, other settings could also be customized, e.g., the user could be allowed to define his favorite colors for the syntax highlighting feature.

As soon as a functioning compiler is fully integrated, i.e. it produces “real” Tasklet byte code, it would be desirable to show compiler error messages and warnings inline. This means that the plug-in analyzes the compiler message, extracts the lines of source code responsible for the error and marks them in the editor. For example, this feature is provided by the plug-in “Java Development Tools” and can be extended by a feature for dynamic compilation. This means that Eclipse implicitly runs the compiler in the background after every change of the source code. Accordingly, syntax errors are marked immediately and not only when the users manually starts the compiler. This behavior would make debugging and trouble-shooting during development much easier for the programmer.

Regarding the **view**, it would be useful for the developer to link the model directly to the editor's source file, so that any change in the source code is directly transferred to the model without the user having to add new model elements manually. Consequently, the user would not have to deal with the creation of the model anymore. Instead, he could focus on the immediate development and documentation tasks. However, this functionality seems very powerful and thus

might require a lot of plug-in development effort. Hence, it should be carefully considered if the benefits are worth the effort, considering that Tasklets are normally not that long, compared to Java classes, for instance.

It seems feasible to extend the current model builder with some intelligence: Instead of overwriting the whole model with every run of the builder, one could think of a solution where the parser simply adds newly discovered elements to the model and leaves pre-existing elements untouched if they are still present in the source code. Like that, the user-generated description texts are not overwritten when the source file is parsed again. Of course, it is assumed that the user did not manually change any identifiers or procedure signatures in the meantime since this would render recognizing pre-existing model elements impossible. Moreover, one could think of a possibility to automatically extract certain information from the source code and use it to fill the model elements' descriptions. For instance, special "documentation comments" could be defined which are automatically parsed and added to the model, similarly to the generation of JavaDocs.

Next, when performing model modifications, especially deletions, one can easily make mistakes. Hence, an "Undo" button to revoke the last modification(s) would be a useful feature. On top of that, a stricter user input validation would be reasonable since in the current version of the plug-in an identifier can contain whitespace characters, for instance, which makes no sense syntactically.

In order to further simplify the use of the view, a set of hot keys could be defined, so that the developer can work more with the keyboard and less with his mouse. At the moment, the user is required to double-click a model element in order to edit it. The general Eclipse shortcut for editing/renaming is **F2**, which could also be used for the view. In contrast, deletion of elements is already possible using the **Del** key.

As a last improvement for the view, one could consider to make the individual visual component more scalable, e.g., the SWT text widget for the source file description has a fixed height of five lines. This could be made more flexible to enable the user to drag the widget to his preferred size.

Concerning the **editor**, it would be highly comfortable if the plug-in was capable of automatically formatting C++ source code. This feature is probably well-known and appreciated by many programmers who use, e.g., the plug-in "Java Development Tools". By pressing **Ctrl**+**⇧**+**F**, Eclipse re-formats the source code with

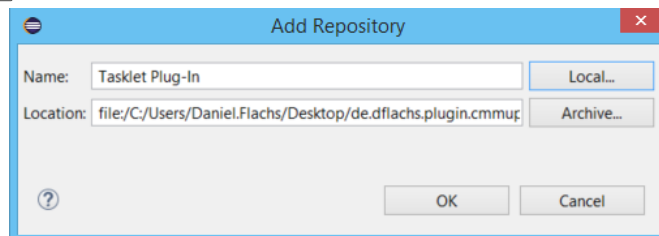
respect to maximum line length, empty lines and correct indentation level. As poorly formatted as the source code might be, this feature cleans up all that and returns source code which is not only optically more attractive but also easy to read and implicitly has a standardized format. This feature does not exist for C++ yet, but would certainly be very helpful for the developers.

Another editor feature which was inspired by JDT is an extension to the existing content assist feature: During Tasklet development, the programmer writes procedures and declares variables. Somewhere else in the source code, he will most definitely refer to them via their identifiers. Consequently, it would be helpful if the content assist feature did not only suggest static keywords of C++, but also names of procedures and variables defined in the current source file. To enable this feature, the source code would have to be parsed, so that identifiers – names of procedures and variables – could be obtained. Concerning the procedures, the logic for this is virtually already implemented by the `CMMModelBuilder`. The obtained identifiers could then dynamically be added to the list of possible suggestions.

Appendix A. Plug-In Installation Instructions

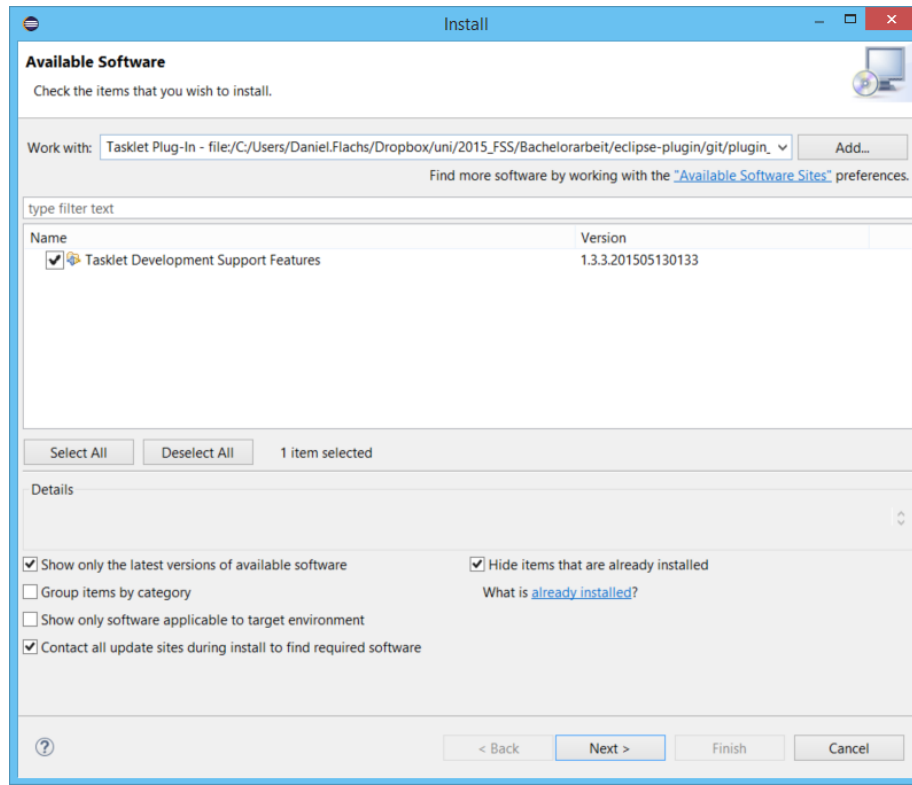
This short manual describes how the C-- plug-in can be installed in the Eclipse IDE. It is assumed, that while Eclipse is installed, there is no other version of the C-- plug-in present in Eclipse.

1. Store the ZIP file containing the plug-in locally at a location of your choice and unpack it.
2. Start Eclipse.
3. Click **Help** > **Install New Software**.
4. In the right upper corner, click on **Add...**.
5. Set a name for the local “Software Site”, e. g. “Tasklet Plug-In”.
6. Click on **Local...**, find and select the unpacked ZIP folder in your file system and click **OK**.



7. If the list in the middle of the window remains empty, uncheck “Group items by category”.

8. Check “Tasklet Development Support Features” and click **Next**.



9. Confirm the next step by clicking **Next**.
10. Accept the (non-existing) license and click **Finish**.
11. Please wait while the plug-in is being installed. During the process, a warning pops up because of the “unsigned content”. Dismiss it by clicking **OK**.
12. To finish the installation, you have to restart Eclipse. Please do so when you are asked to.

To open the C-- perspective for the first time after the installation, go to **Window** **Open Perspective** **Other...** and then click on “C--”.

Appendix B. Code Listings

B.1. Plug-In-specific Listings

Listing B.1: MANIFEST.MF

```
1 Manifest-Version: 1.0
  Bundle-ManifestVersion: 2
  Bundle-Name: C-- Tasklet Language Editor
  Bundle-SymbolicName: de.dflachs.plugin.cmmEditor;singleton:=true
5 Bundle-Version: 1.3.2.qualifier
  Bundle-Activator: de.dflachs.plugin.cmmEditor.ui.Activator
  Bundle-Vendor: DFlachs
  Require-Bundle: org.eclipse.ui,
    org.eclipse.core.runtime,
10    org.eclipse.jface.text,
    org.eclipse.ui.editors,
    org.eclipse.ui.console,
    org.eclipse.jdt.ui,
    org.eclipse.jface,
15    org.eclipse.core.resources
  Bundle-RequiredExecutionEnvironment: JavaSE-1.7
  Bundle-ActivationPolicy: lazy
  Bundle-ClassPath: .,
    lib/json-simple-1.1.1.jar,
20    lib/commons-io-2.4/commons-io-2.4-javadoc.jar,
    lib/commons-io-2.4/commons-io-2.4-sources.jar,
    lib/commons-io-2.4/commons-io-2.4.jar
```

B.2. Java Source Code Listings

Listing B.2: Method for serialization of class CMMPProcedure (simplified)

```
1 public JSONObject serialize() {
    Map<Object, Object> map = new HashMap<>();

    map.put("proc_name", this.getName());
```

```

5  map.put("proc_description", this.getDescription());
   map.put("proc_return_datatype", this.getReturnInfo().getType());
   map.put("proc_return_description",
           this.getReturnInfo().getDescription());
   map.put("proc_paramcount", this.getParameters().size());
10
   for (int j = 0; j < this.getParameters().size(); ++j) {
       CMMProcParameter currProcParam = this.getParameters().get(j);
       map.put("proc_param_" + j, currProcParam.serialize());
   }
15
   return new JSONObject(map);
}

```

Listing B.3: Method for deserialization of class CMMProcedure (simplified)

```

1  public static CMMProcedure deserialize(JSONObject object) {
   Map<?, ?> map = object;
   CMMProcedure p;
   String name, description, returnType, returnDescription;
5  int paramCount;

   name = (String) map.get("proc_name");
   description = (String) map.get("proc_description");
   returnType = (String) map.get("proc_return_datatype");
10  returnDescription = (String) map.get("proc_return_description");
   paramCount = Integer.parseInt(
       map.get("proc_paramcount").toString());

   p = new CMMProcedure(name, description, returnType);
15  p.getReturnInfo().setDescription(returnDescription);

   for (int i = 0; i < paramCount; ++i) {
       CMMProcParameter current = CMMProcParameter
           .deserialize((JSONObject) map.get("proc_param_" + i));
20  current.setParentProc(p);
       p.addParameter(current);
   }

   return p;
25 }

```


B.3. C-- Source Code Listings

Listing B.4: Tasklet Source File `primes.cmm`

```
1  int low, high, result;

   procedure int checkprime (int a) {
       int c;
5      c := 2;

       while (c <= (a-1)) {
           if ((a%c) = 0) {
               return 0;
10          }
           c := c+1;
       }

       if (c = a) {
15          return a;
       }
   }

   >>low;
20  >>high;

   while (low < high) {
       result := checkprime(low);

25      if (result # 0) {
           <<result;
       }

       low := low + 1;
30  }
```

Bibliography

- [Ecl15a] Eclipse Foundation. About the Eclipse Foundation, 2015. <http://www.eclipse.org/org/> (Accessed: 03/05/2015).
- [Ecl15b] Eclipse Foundation. Eclipse API Specification, 2015. <http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Findex.html&org.eclipse/jface/databinding/dialog/package-summary.html> (Accessed: 10/05/2015).
- [Ecl15c] Eclipse Foundation. Eclipse Equinox, 2015. <http://eclipse.org/equinox/> (Accessed: 05/05/2015).
- [Ecl15d] Eclipse Foundation. Eclipse Platform Plug-in Developer Guide, 2015. <http://help.eclipse.org/luna/index.jsp?nav=%2F46> (Accessed: 04/05/2015).
- [JSO15] JSON. Introducing json, 2015. <http://www.json.org/> (Accessed: 10/05/2015).
- [OSG15] OSGi Alliance. OSGi Main Page, 2015. <http://www.osgi.org/> (Accessed: 04/05/2015).