HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

COMPUTER SCIENCE

IFS INSTITUTE FOR SOFTWARE

Conan for Cevelop

Term Project

Department of Computer Science
University of Applied Science Rapperswil

Fall Term 2017

Author(s): Giovanni Heilmann, Pascal Schweizer
Advisor: Thomas Corbat
Technical Advisor: Felix Morgner
Project Partner: Institute for Software, Rapperswil

Table of Contents

Assignment.....5
Supervisor and Expert.....5
Students.....5
Introduction.....5
Goals of the Project.....5
Documentation.....6
Important Dates.....6
Evaluation.....6
References.....7
Abstract.....8
1 Management Summary.....9
1.1 Roles.....9
1.2 Introduction.....9
1.3 Objective.....9
1.4 Result.....10
1.5 Outlook.....10
2 Introduction.....11
2.1 About this Document.....11
2.2 Scope.....11
2.3 Overview.....11
3 Analysis.....12
3.1 Requirements.....12
3.1.1 Use Case 1: Manage Conan profiles.....12
3.1.2 Use Case 2: Manage Conan remotes.....12
3.1.3 Use Case 3: Manage project dependencies.....12
3.1.4 Use Case 4: Browse Conan packages.....12
3.2 Conan Install.....12
3.2.1 Conan Build Settings.....13
3.3 Conan Installation Path.....15
3.4 Conan Remotes.....15
3.5 Conan Profiles.....16
3.6 The Parsers.....16

3.6.1 conanbuildinfo.txt 17
3.6.2 Conan Profiles 17
3.6.3 Conan Remotes (registry.txt) 18
4 Design 19
4.1 Architecture 19
4.2 Conan Install 19
4.3 Conan Installation Path 21
4.4 Conan Remotes 22
4.5 Conan Profiles 23
4.5.1 Edit Profiles 24
4.5.2 Saving Changes 26
4.5.3 Project Specific Profile 26
4.6 The Parsers 27
5 Implementation 29
5.1 Conan Install 29
5.1.1 Storing Build Information 29
5.1.2 Remembering Which Build Information Was Added 30
5.2 Conan Installation Path 30
5.2.1 ConanPreferencePage 30
5.2.2 ConanNotFoundDialog 30
5.2.3 ConanCliCommand 30
5.3 Conan Remotes 31
5.3.1 Default and Active Model 31
5.4 Conan Profiles 32
5.4.1 Profile List 32
5.4.2 Edit Profile View 32
5.4.3 Project Specific Profile 33
5.5 The Parsers 34
5.5.1 SectionParser 34
5.5.2 NamedSectionParser 35
5.5.3 OrderedSectionParser 37
6 Conclusion & Outlook 39
7 Project Management 40

7.1 Organization 40
7.2 Process 40
7.3 Quality Assurance 40
8 References 41
9 Figures 42
10 Listings 43
Appendix A: Installation Manual 44
A.1 Requirements 44
A.2 Plug-in Installation 44
A.3 Installation Problems 50
Appendix B: User Manual 52
B.1 Introduction 52
B.2 Install packages 52
B.3 Set Conan installation path 52
B.4 Manage Conan Remotes 53
B.5 Manage Conan Profiles 54
B.6 Project Specific Profile 55

Assignment

Supervisor and Expert

This term project will be developed for the Institute for Software at HSR internally. It will be supervised by Thomas Corbat (tcorbat@hsr.ch) and Felix Morgner (fmorgner@hsr.ch), HSR, IFS

Students

This project is conducated in the context of the module “Studienarbeit” in the department “Informatik” by

- Giovanni Heilmann (gheilman@hsr.ch)
- Pascal Schweizer (pschweiz@hsr.ch)

Introduction

Cevelop is an Eclipse CDT based integrated development environment (IDE) for C++, implemented and maintained by the Institute for Software at HSR. [1] The IDE is responsible for providing various tools to ease the development of C++ software. It is also used to manage dependencies of the source code at hand to external libraries. Conan is a package manager for C++ [2]. It is a flexible command line tool to also specify dependencies to specific packages. As Cevelop is currently lacking a proper integration of Conan, the tool has to be configured and invoked though a console explicitly.

Goals of the Project

The goal of this term project is the proper integration of Conan and a subset of its functionality as a plug-in for Cevelop. This includes detection of a Conan configuration for the project, invocation of the CLI tool and synchronization of the dependencies of the C++ project.

The main features of the plug-in:

- Configuration of main settings for Conan, like installation location, working directory, repository, etc. on workspace and project level.
- Selection of active Conan profile.
- Management of the dependencies specified by Conan in the Cevelop project. Updates in the Conan
- configuration should be accommodated and refresh the configuration of the project. This includes parsing Conan files and adapting source, header and library locations as well as defined symbols.
- Invocation of the most important Conan commands. The output and result can be shown in an Eclipse console as is.
- Configuration for Release and Debug builds have to be respected.

The goals above can be adapted with a reasonable explanation if suitable.

Extended goal (optional): If the students make quick progress on the implementation of the main features, the plug-in could be extended by a Conan file editor. This editor might provide Conan-specific syntax highlighting and auto completion. Furthermore, it could query Conan repositories for available packages.

There will be weekly meetings with the supervisors. Additional meetings might be scheduled as required by the students. All Meetings, except for the kick-off meeting, will be prepared by the students with an agenda that is sent to the supervisors at least one day before the meeting. During the meeting the current progress will be presented (What has been done? What has been achieved? How much time did it take? What is planned for the subsequent week?). Decisions of the meeting must be recorded by the students.

At the beginning a project plan has to be devised for with milestones for the semester. This plan is used as a guide line to check the progress compared to the estimation. The project plan will be updated according to the actual execution, including time reports and tasks. The students get feedback for accomplished milestones. The final mark will be given based on the eventual results handed-in by the deadline at the end of the semester.

Documentation

This project has to be documented according to the guide lines of the "Informatik" department [3]. This includes all analysis, design, implementation, project management, etc. sections. All documentation is expected to be written in English. The project plan also contains the documentation tasks. All results must be complete in the final upload to the archive server [4]. Two copies of the documentation have to be handed in:

- One in color, two-sided
- One in B/W, single-sided

Important Dates

18.09.2017	**Start of the term project.**
19.12.2017	**Hand-in of the abstract to the supervisor for checking. Information about accessing the corresponding web tool will be given by the department office.** **Hand-in (by email) of the A0 poster to the supervisor.**
22.12.16, 17.00	**Final hand-in of the report**

Evaluation

A successful term project counts as 8 ECTS points per student. The estimated effort for 1 ECTS is 30 hours. (See also the module description [5]). The supervisor will be in charge for all the evaluation of the project.

Criterion	Gewicht
1. Organisation, Execution	1/5
2. Report (Abstract, Management Summary, technical and personal reports) as well as structure, visualization and language of the whole documentation	1/5
3. Content	3/5

Furthermore, the general regulations for term projects of the department "Informatik" apply.

References

[1] https://www.cevelop.com
[2] https://www.conan.io
[3] https://www.hsr.ch/Allgemeine-Infos-Diplom-Bach.4418.0.html
[4] https://archiv-i.hsr.ch
[5] http://studien.hsr.ch/allModules/24386_M_SAI14.html

Abstract

Conan is a CLI tool for C++ development that can conveniently install packages for any given build configuration. The tool automatically downloads the libraries from remote databases and prepares the build information for any tool-chain like, but not limited to, gcc or cmake. Using profiles, the user can define different build environments and Conan chooses the right packages accordingly. This makes Conan a very flexible tool, as it works well in many environments.

Cevelop is an IDE based on Eclipse C/C++ Development Tooling (CDT). Currently, users need to manually call the CLI commands and write the build information to the respective settings in the Cevelop project. Thus, this term project's goal is to write an Eclipse plug-in that fixes this problem. It must handle installing packages and keeping them up-to-date. What's more, it should manage Conan profiles and remotes so that the user does not need to use the command-line or open files directly. Everything should be done in Cevelop alone.

Conan for Cevelop has the following features: First off, the user needs only to click a button and the plug-in installs all designated packages. The plug-in also manages remotes and profiles. The user can manage them from within Cevelop. Currently, only one file needs to be edited by hand, which is used by Conan to know which packages to add. In the future, a package browser may be added to the plug-in. This would remove the need to edit the conanfile.txt manually.

1 Management Summary

1.1 Roles

The roles for this project are distributed as can be seen in Table 1.

Graduates	Giovanni Heilmann, Pascal Schweizer
Examiner	Thomas Corbat
Technical Advisor	Felix Morgner
Subject area	Software-Engineering
Project partner	Institute for Software

Table 1 Roles

1.2 Introduction

There are many ways to handle package dependency management. From manually sharing library files among the development team to using dependency managers that automatically install all related libraries in an easy and simple way. The Conan C++ package manager belongs to the latter category. It is a command line interface (CLI) tool for C++ development that can conveniently install dependencies for any given build configuration. The different build configurations are managed by Conan in user-definable profiles. The tool handles downloading the right files and preparing build information for any build tool chosen by the user. The files can be downloaded from public Conan remotes, although users can also set up their own private databases, e.g. for confidential source code. These features make Conan a very flexible tool, as it works well in many environments.

1.3 Objective

Cevelop, developed at the Institute for Software (IFS), is an integrated development environment based on Eclipse C/C++ Development Tooling (CDT). The IFS have been working with Conan for some time now, and the workflow proves to be suboptimal at best. Currently, there is no integration of Conan for Cevelop-managed projects. Developers need to manually call the CLI commands in the shell and insert the build information to the respective settings in the Cevelop project.

Therefore, this term project's goal is to implement an Eclipse plug-in to remedy this problem. It must handle installing package dependencies and keeping them up-to-date. What's more, it should manage Conan profiles and remotes so that the user does not need to use the command-line or edit configuration files directly. Everything should be done in Cevelop alone.

The main motivation for this project are convenience, ease-of-use, and time-efficiency as the current workflow lacks any of these.

1.4 Result

The developed plug-in has the following features:

- Conan for Cevelop handles the installation of all package dependencies in a very convenient way. The user needs only to click a button and the plug-in handles the rest. Notably, all Eclipse build information is automatically updated. This is the plug-in's key feature and saves a lot of time for the developer. Furthermore, using this solution is less error-prone. The user will not forget to remove an obsolete setting. The benefits are tremendous.
- Conan for Cevelop manages Conan remotes. The user may add, edit or delete remotes from within Cevelop and does not need to edit any files or open a command-line tool.
- Conan for Cevelop manages Conan profiles. The user may add, edit or delete profiles. Furthermore, the user may select a workspace-wide default profile. Individual Cevelop projects may choose to use the workspace profile, or they may select a different active profile.

1.5 Outlook

Currently, only one file needs to be edited by hand: conanfile.txt, which is used by Conan to know which packages to add. In the future, a package browser may be added to the plug-in. It would remove the need to edit the conanfile.txt manually.

Additionally, the plug-in only edits build settings for the active configuration. The user should be able to choose different profiles for different configurations, for example a release profile for the release configuration and a debug profile for the debug profile.

There are no known functional faults in the software, and the core features are implemented. Thus, Conan for Cevelop is ready to be shipped with Cevelop.

2 Introduction

2.1 About this Document

This document describes the functionality provided by the Conan plug-in as well as how it was designed and implemented, including the reasons and decisions leading to this implementation.

2.2 Scope

This document is valid for the whole project duration. Only the most recent version is valid.

2.3 Overview

There are many ways of handling the library dependencies of a C++ project. A widely known and used method is the Conan package manager [1]. However, a big downside of this package manager is that there is no integration for common IDEs like Eclipse [2] or Cevelop [3]. And that is exactly what this plug-in provides: The integration of Conan's core functionality into Eclipse and Cevelop. It works for both IDEs because Cevelop is based on Eclipse C++ Development Tooling (CDT) and is thus able to use plug-ins written for Eclipse CDT.

Conan's core functionality is its install command [4] that handles the download and installation of the project dependencies, combined with the Conan remotes [5] and profiles [6] that are used to configure the install command.

This is how a workflow of someone using the plug-in could look like:

1. Create a new C++ project in Eclipse
2. Add a conanfile.txt to the project root, containing a list of the dependencies
3. Optional: Configure the Conan remotes through the Eclipse preferences
4. Optional: Configure/select the Conan profile through the Eclipse preferences
5. Run the Conan plug-in's install command through the project explorer's context menu

And these are the things the plug-in does for you in the background when running the install command:

6. Invokes the actual Conan install command to download and install the declared dependencies
    a. Looks for packages in the configured remotes
    b. Applies the currently selected profile to the install command
    c. Prints the command's output on the Eclipse console
7. Adds all installed libraries, include paths, build flags, etc. to the corresponding Eclipse project's build settings
8. Updates the CDT index so Eclipse can handle syntax highlighting.

How exactly the plug-in does these things and why we designed them that way, is described in the further chapters of this document.

3 Analysis

This chapter describes the analysis that the design decisions are based on.

3.1 Requirements

Conan is a command line interface (CLI) tool that provides several useful functions for package dependency management of C++ projects. Its core feature is its install command which handles the download and installation of the dependencies for different build configurations. These dependencies are downloaded from remote servers, or remotes as they are called in Conan. The build configurations are handled through Conan's profiles that enable saving and reusing of different configurations.

The install command, remotes and profiles are the three main features Conan provides. Thus, they are the focus of this project and will be integrated into the plug-in.

3.1.1 Use Case 1: Manage Conan profiles

The user can create, edit and delete Conan profiles, configure a default profile for all C++ projects and override this default in each project.

3.1.2 Use Case 2: Manage Conan remotes

The user can modify Conan's list of remotes, meaning he can add, edit and remove remotes and also reorder them.

3.1.3 Use Case 3: Manage project dependencies

The user can install and uninstall project dependencies.

3.1.4 Use Case 4: Browse Conan packages

The user can browse the Conan packages on configured remotes and the ones that are already installed on his machine.

3.2 Conan Install

Conan's install command, that handles the download and installation of the project dependencies, contains an option called "--generator xxx" which enables the use of so-called generators [7]. These generators are named after build tools, including gcc, cmake, etc. and will generate files containing build information that can be used to build the project using the corresponding build tool. These files differ greatly depending on the generator used and are hardly human-readable, because they need to comply with the requirements of the corresponding build tool.

There is also an extra generator called txt [8] that does not correspond to a specific build tool and generates a file called conanbuildinfo.txt which presents the build information in a human-readable form that is quite similar to that of ini-files [9]. An example of such a file that was generated for a project with zlib as its only dependency can be seen in Listing 1.

```
1:  [includedirs]
2:  /home/user/.conan/data/zlib/1.2.11/conan/stable/package/72446892ca242ab/include
3:
4:  [libdirs]
5:  /home/user/.conan/data/zlib/1.2.11/conan/stable/package/72446892ca242ab/lib
6:
7:  [builddirs]
8:  /home/user/.conan/data/zlib/1.2.11/conan/stable/package/72446892ca242ab/
9:
10: [libs]
11: z
12:
13: [cppflags]
14:
15: [...]
16: ...
```

Listing 1 conanbuildinfo.txt example

Assuming you have a C++ project with multiple dependencies and installed all these dependencies using Conan. If you want to build this project using Cevelop, you first need to configure the project's build settings so that it knows where to find its dependencies, what compiler flags to use, etc. All this information can be found in conanbuildinfo.txt. The sections contained in the file can be mapped 1:1 to the build settings of Cevelop. All you need to do is copy and paste the values to the right places. But depending on the amount of dependencies and your knowledge of build settings, this can prove to be quite difficult and error prone. Or in other words: This is something that can and should be automated through the plug-in.

3.2.1 Conan Build Settings

As mentioned above, the build settings can be mapped 1:1. To do this, it is necessary to know where they can be found in Cevelop.

As build settings are project specific, it can be found under the project properties, when right-clicking on a given project and selecting "Properties." There, one needs to navigate to the "Settings" page, which is a sub-page of "C/C++ Build". All build settings can be found on this page in the "Tool Settings" tab. Here, the settings are categorized by function like certain compilers, and linker), which are in turn divided further as can be seen in Figure 1.

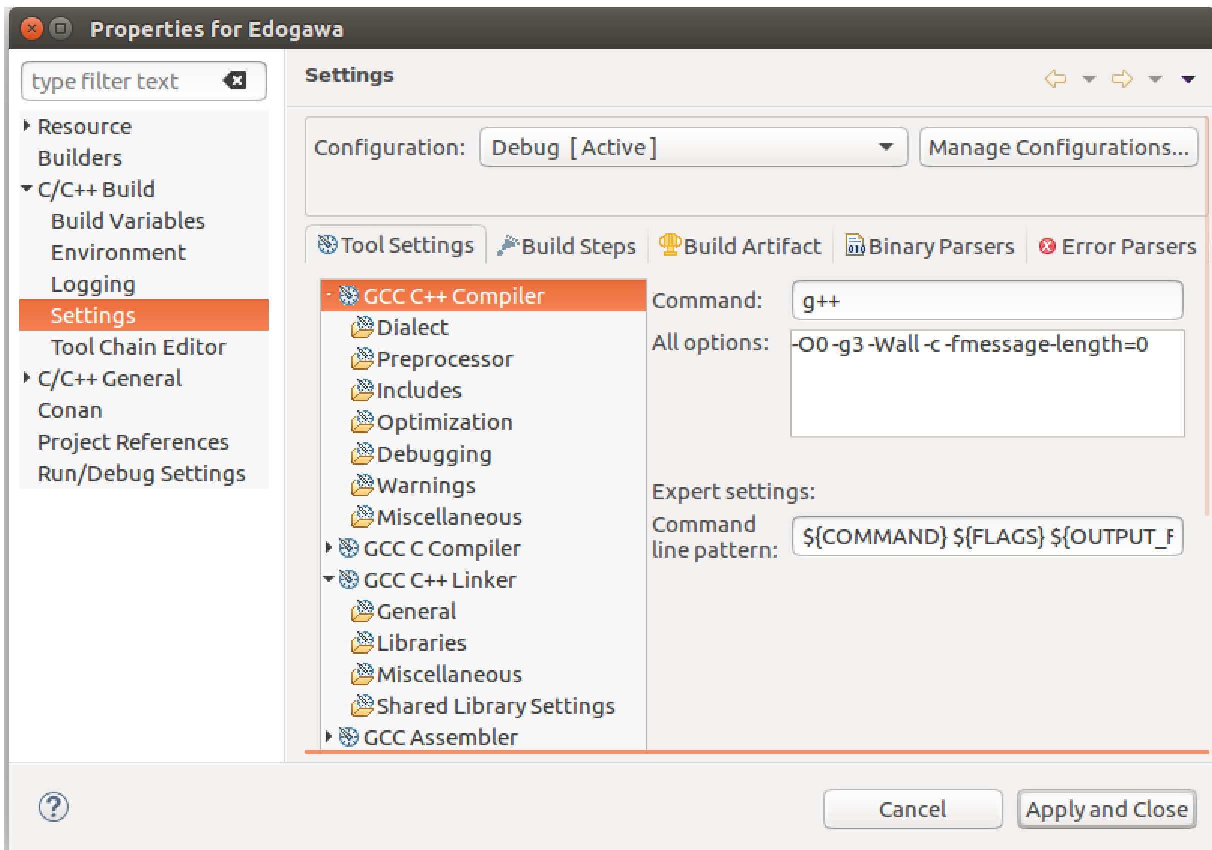


Figure 1 Project properties

3.2.1.1 GCC C++ Compiler

Here are settings related to the GCC C++ compiler, used when building C++ projects. Certain Conan settings need to be added here.

Preprocessor symbols

Conan can define some preprocessor symbols in the section “defines.” This must be mapped to the “Defined symbols” field under “Preprocessor.”

Include paths

Conan’s “includedirs” corresponds to Cevelop’s “Include paths” under “Includes.”

These are relevant for symbol resolution and syntax highlighting/autocompletion in Cevelop.

C++ flags

Custom compiler flags defined by Conan are listed in the “cppflags” section. These must be appended to “Other flags” under “Miscellaneous.”

3.2.1.2 GCC C Compiler flags

Conan also defines some C compiler-specific flags “cflags”, that need to be added to “Other flags” under “Miscellaneous” of the GCC C Compiler.

3.2.1.3 GCC C++ Linker

The last few relevant settings can be mapped to Cevelop’s GCC C++ linker settings.

Library directories

Conan’s “libdirs” is mapped to Cevelop’s “Library search path” under “Libraries.”

Cevelop needs this to find where the libraries are located.

Libraries

Conan’s “libs” is mapped to Cevelop’s “Libraries” under “Libraries.”

Cevelop needs this to know which libraries to include.

Link flags

Conan’s “sharedlinkflags” and “exelinkflags” are mapped to Cevelop’s “Linker flags” under “Miscellaneous.”

These are flags that must be added depending on whether the project is an executable or library.

3.3 Conan Installation Path

Since the plug-in uses some Conan commands, like Conan install, it needs a Conan installation to execute them. That means there are two options:

1. Automatically install Conan together with the plug-in.
2. Leave the installation of Conan to the user.

The first one would be more user friendly since they would not have to manually download and install Conan themselves. But there is also a problem: Conan is a Python application. And the problem therein is that Python applications do not work without a Python interpreter, meaning that Python would also need to be shipped with the plug-in since that is not something that is preinstalled on most operating systems.

Another problem that arises from using a Conan installation is that the plug-in needs to know its installation location. But since the user is free to install or move Conan anywhere on his file system, there is no way of knowing that location except for searching through the entire file system. But this is not a feasible solution.

3.4 Conan Remotes

One of the most important configuration options Conan provides are the so-called remotes. They can be edited through the Conan remote command and are stored in a file called registry.txt, which looks like the example shown in Listing 2:

```
1:  conan-center https://conan.bintray.com True
2:  conan-transit https://conan-transit.bintray.com True
3:  ...
```

Listing 2 registry.txt example

The entries shown above consist of three values:

1. A unique name to identify the remote.
2. A URL pointing to a web-server where Conan packages can be downloaded from.
3. A Boolean that indicates whether Conan should verify the SSL certificates for that remote server.

These remotes are used for various Conan commands. When, for example, the Conan install command is invoked, Conan first checks if the required packages are already installed and if not, sequentially looks for them in the configured remotes from top to bottom.

3.5 Conan Profiles

Conan allows the use of profiles for easy handling of different build configurations. These files – like the conanbuildinfo.txt-files – look like ini-files, and most sections use key-value pairs separated by a '=' symbol, but one uses just simple lines of text (i.e. the build_requires section). The first (nameless) section is comprised of include declarations of other profiles, and variable declarations that can be used further down in the file and other profiles that include this one. An example can be seen in Listing 3.

```
1:  Include(other)
2:  CLANG=/usr/bin/clang
3:
4:  [settings]
5:  setting=value
6:
7:  [options]
8:  MyLib:shared=True
9:
10: [env]
11: zlib:CC=$CLANG/clang
12:
13: [scopes]
14: scope=value
15:
16: [build_requires]
17: Tool1/0.1@user/channel
```

Listing 3 Conan profile example

Calling the Conan CLI tool has a noticeable delay, because it needs to load up the python interpreter and Conan. This will take half a second. What's more, the profile manipulation command only allows to set single options. Thus, to edit a whole profile, one would need to wait several seconds just to save changes. This is relevant for design choices when designing the UI, as responsiveness is a key quality.

3.6 The Parsers

Conan uses multiple configuration files that are either used or created during the execution of various Conan commands. Some of these files, like the ones containing the remotes and profiles, are also used for the Conan install command, meaning that their content would be important for the plug-in. And if the content of a file is to be used in an application, it needs to be parsed. However, that is only possible if the file has a fixed structure, which is why we analyzed the structures of the most important configuration files.

3.6.1 conanbuildinfo.txt

This is one of the files generated by Conan's install command. It contains build information like include paths, compiler flags, etc. that can be used when building the corresponding C++ project. Its structure is shown in Listing 4.

```
1:  [SectionName]
2:  entry 1
3:  entry 2
4:  entry n
5:
6:  [NextSection]
7:  more entries...
```

Listing 4 conanbuildinfo.txt structure

The file consists of multiple sections each with a name put in [square brackets] followed by the entries belonging to that section. This structure is very similar to that of ini-files [9].

3.6.2 Conan Profiles

Conan's profiles are stored in separate files, one file per profile. All these files follow the same structure, which is pretty similar to the one of conanbuildinfo.txt. An example can be seen in Listing 3.

They contain multiple named sections with multiple entries each. But there are also two major differences:

1. The sections contain three different types of entries:
    a. key-value-pairs
    b. "normal" entries without keys
    c. includes of other profiles
2. There is an extra unnamed section at the beginning of the file containing
    a. the profile includes
    b. variable definitions that can be used within the profile

3.6.3 Conan Remotes (registry.txt)

The file containing the remotes is called registry.txt and is made up of two lists. The first one contains all configured remotes and the second one all installed packages, as shown in Listing 5.

```
1:  remote 1
2:  remote 2
3:  remote n
4:
5:  package 1
6:  package 2
7:  package n
```

Listing 5 Conan remotes file structure (registry.txt)

It also consists of multiple sections each containing multiple entries. But the sections are not named. Instead they have a predefined order: remotes first, installed packages next.

4 Design

This chapter describes the design details for each function point mentioned in the analysis chapter.

4.1 Architecture

As a plug-in, Conan for Cevelop interfaces with Eclipse through certain extension points. Furthermore, it uses some of Eclipse CDT's functionality. This section will give an overview of the surroundings.

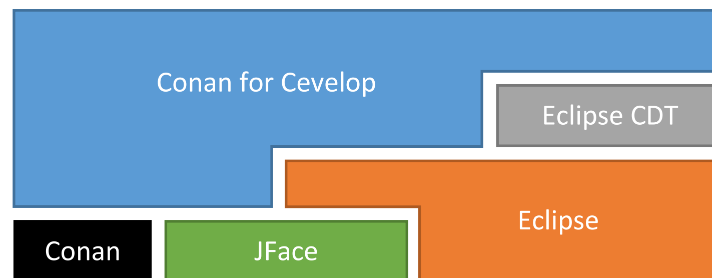


Figure 2 Architecture layers

As can be seen in Figure 2, Conan for Cevelop depends on several other components. A general description about the interface between the plug-in and the underlying component is described in the following list. For more information on how they are used, the following sections in this chapter will provide more insight.

- **Conan** is used for running its install command to conveniently install all packages.
- **Eclipse** is the underlying framework for the GUI. It is interfaced by way of extension points.
- **Eclipse CDT** is used to access the build settings in Eclipse.
- **JFace** is a GUI toolkit for Java. It builds on SWT and is used for several key GUI features.

The first three points are essential for this plug-in. Without them, it would not be feasible to implement Conan for Cevelop. JFace was chosen, because it provides a solid toolset for GUI development and ease of use. The alternative would have been to work with SWT directly. This would have meant more design time, because it is much more difficult to use. JFace was designed specifically to alleviate the drawbacks of SWT.

4.2 Conan Install

Since the Conan install command is Conan's core feature that was also the first feature we wanted to add to our plug-in. The command is used to manage project dependencies, meaning that it is tightly coupled with a project and its properties and, because of that, always has to be invoked in the context of a project. This was our main concern when deciding where exactly we should add this command to the IDE.

There are many different options in Eclipse to add new functionality, like the menu or tool bars at the top, various context menus, etc. What we thought would fit best was the project explorer's context menu (see Figure 3). Because, as the name already suggest, it is a menu in the context of a project and is only available by right-clicking either a project itself or a file that belongs to a project.

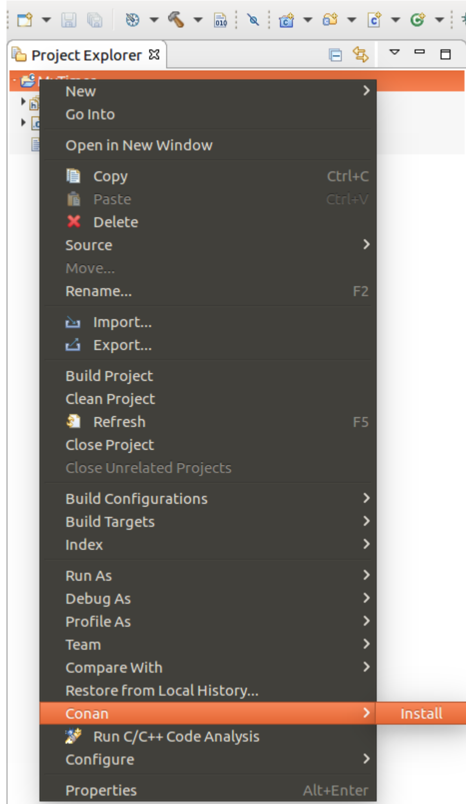


Figure 3 Conan install

The next question was: What to do with the output of Conan install? Since Conan is a command line tool, it generates plain text output that is usually printed on the console used to run a Conan command. This output is quite important as it provides useful information about what the command did and whether its execution was successful or not. That is why we decided to also show this output through the plug-in by printing it on Eclipse's own console.

The next issue was writing the build settings to the right locations. To do this programmatically, the plug-in needs to interface with Eclipse CDT's code, since its build information is stored there. The class ManagedBuildManager handles reading and writing this build information from and to Eclipse.

Eclipse CDT stores build settings per configuration, namely Debug and Release. Every configuration has certain toolchains, like linux gcc, mingw. These are separated into tools. The linux gcc for example has the following: compiler, linker and assembler. These tools are further divided into options. There are three basic kinds of options as defined by Eclipse CDT: Boolean, strings, and string lists. To store Conan's build settings, they need to be allocated to the right CDT option in the right toolchain in the right configuration.

Currently, the Conan plugin does not include all sections supplied by Conan, because they are not needed. Namely, bindirs, resdirs, and builddirs and all package specific sections, signified by a section-suffix containing its name, e.g. "libs_Poco". They are not necessary, because the only needed build information is library locations and the library names to build a project in Cevelop.

Furthermore, the plug-in needs to know which build information belongs to Conan and which is user-defined. This is important, because when rerunning the install command, the plug-in should isolate the right information and replace it with the updated one. The risk when not done correctly is that settings are duplicated, or worse yet, user settings are lost. There is no way in Eclipse CDT to do this directly when reading and writing. Therefore, Eclipse preferences are used as a memory of what was added by the plug-in. Upon rerunning the install command, the stored build information is recalled and first removed, before the new information is added.

Eclipse preferences are persisted information between application restarts. They are stored as key-value pairs where the key is always a string and the value may be any primitive type. These preferences can be saved in different scopes, to grant different ranges of visibility within the application. The most notable ones are instance scope and project scope. Instance scope is visible workspace-wide and project scope – as the name suggests – is specific to each project. Since the build settings are project-specific as well, Conan uses the project scope to store them.

Since the Eclipse preferences can only store primitive types, a list of settings like libraries or include directories cannot be saved directly to the preferences. Therefore, the list is joined into a single string delimited by an appropriate delimiter, in this case ";".

4.3 Conan Installation Path

Since we did not want to ship our plug-in with a Python installation, we decided to also leave the installation of Conan to the user. The problem of the unknown installation location of Conan was also solved in a pretty simple manner: By adding a configuration option to Eclipse's preferences. As shown in Figure 4 below, this option was added to a newly created preference page called Conan that also contains other options related to the plug-in.

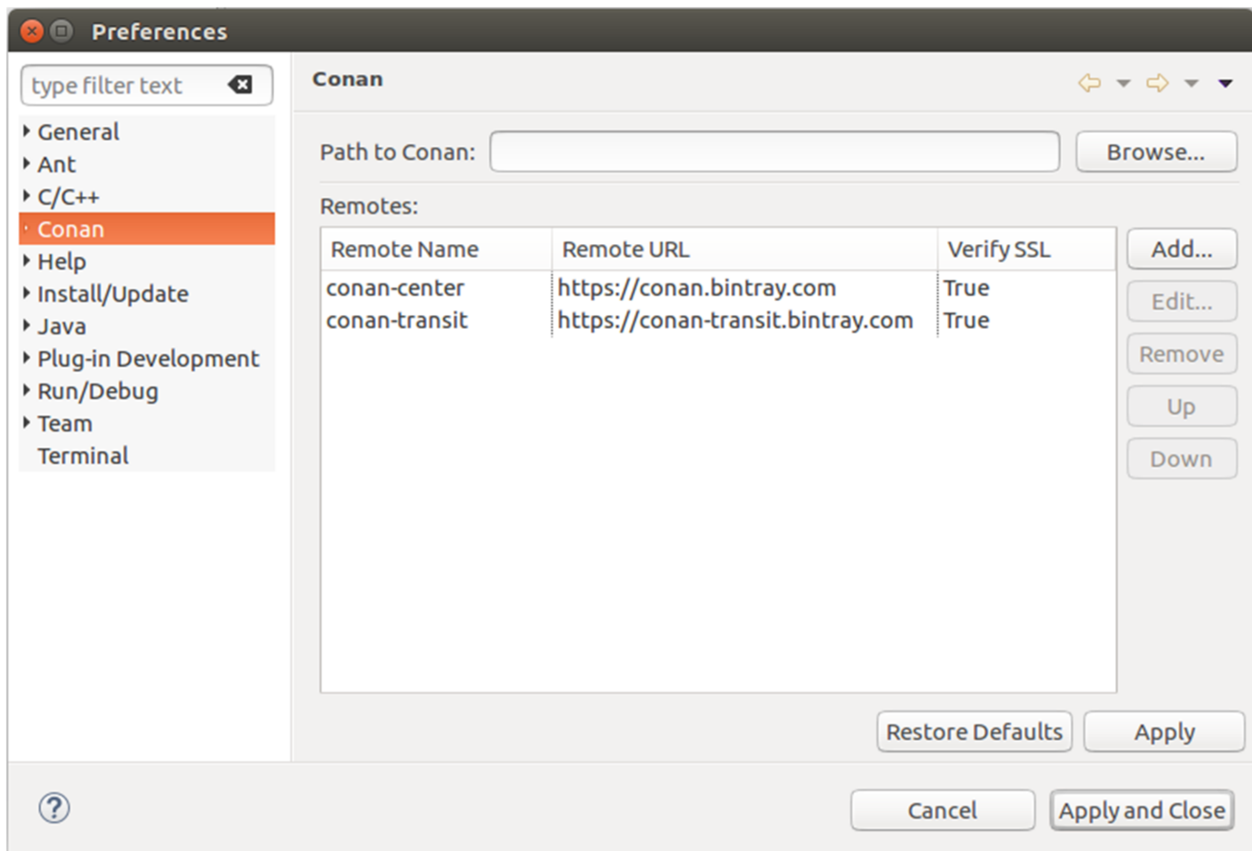


Figure 4 Conan preference page

The user can either type or copy the path into the text field or use the browse button to open a file explorer where he can select the Conan executable. Although the preference page does not let the user apply a path that does not point to an executable, it is possible that a path becomes obsolete after applying it if, for example, the selected file is moved or deleted. If that is the case or no path was configured, the plug-in will use the fallback option which is the system's PATH variable and search for a Conan installation there.

Whenever the user invokes a Conan command through the plug-in, the only one being the install command at the moment, it will check whether the configured path is valid or use the PATH variable as a fallback. If both do not work, the dialog shown in Figure 5 will pop up, prompting the user to either provide a valid path or abort the command:

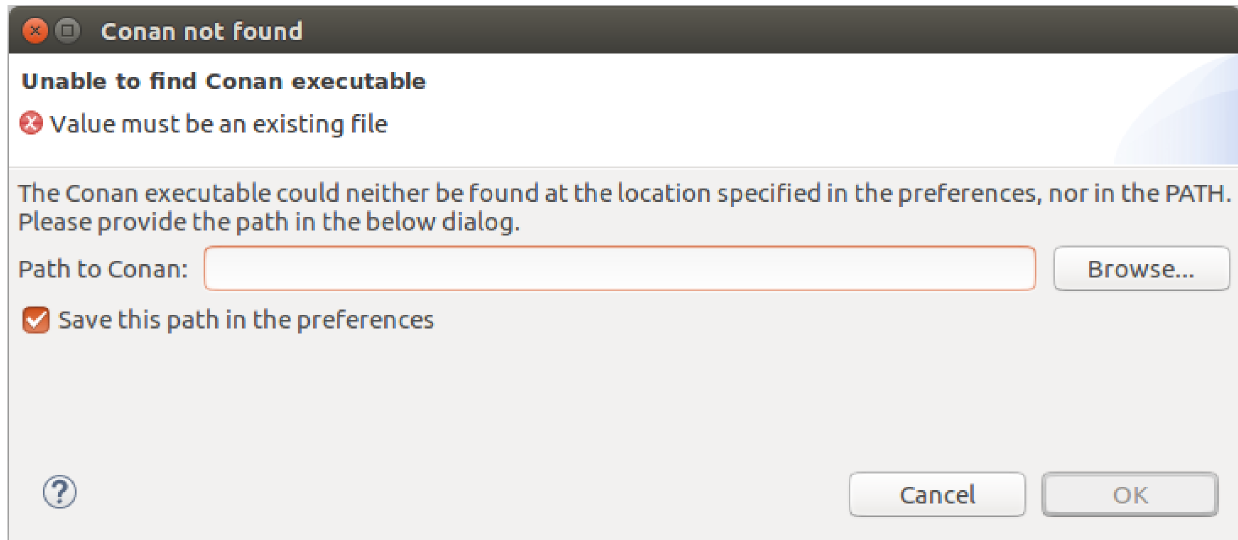


Figure 5 Conan not found dialog

4.4 Conan Remotes

Since the remotes are an important part of Conan and are also used for the core feature of this plug-in, we decided to provide a simple way of configuring the remotes through the plug-in. When it comes to configuration tools in Eclipse, the first thing that comes to mind are the workspace preferences and project properties. Since Conan handles the remotes globally, in a single file and not project-wise, we decided to do the same and went with the workspace preferences.

The next question was: How do we access/edit the remotes? This could either be done through the Conan remote command or by modifying the registry.txt file directly. We decided to modify the file directly, because of the following reasons:

- Conan is a Python application, meaning that using a Conan command will start up a Python interpreter which is much slower than editing a text file.
- The Conan command can only edit one remote at a time, whereas modifying the file has no such restrictions.
- We already had a parser for the build-info that we could adjust and reuse.

As shown in Figure 4, the configuration of the Conan remotes is located on the same preference page as the installation path. The reason for that is that there was a lot of free space and we did not want to create unnecessarily many separate (sub-)pages.

Similar to some other Eclipse preferences, the page shows a list of all configured remotes with a few buttons on the right hand side to perform common operations like adding, editing, removing or reordering entries. To add or edit a remote, this popup dialog was created:

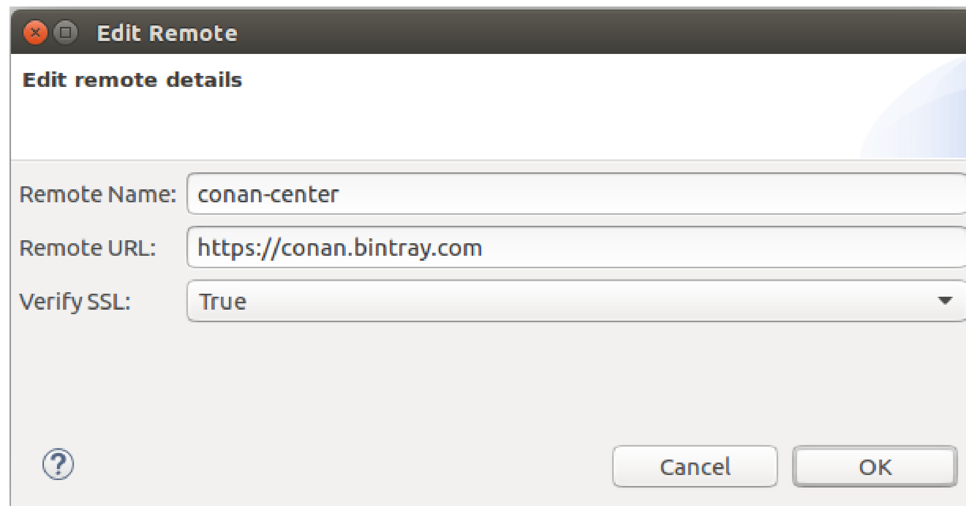


Figure 6 Edit remote dialog

This dialog performs some basic input checks to make sure that only valid values are stored in the file containing the remotes. These checks involve:

- Only letters, numbers, - and _ can be used for the remote name.
- The URL must begin with http: or https: and cannot contain special characters like []<>\"^…
- No spaces are allowed in any values, because that is the delimiter used to separate them.

They are not very elaborate because the expected user of this plug-in is a programmer that knows what he is doing.

4.5 Conan Profiles

To manage Conan profiles, a preference page was added. There, the user can create new profiles, edit existing ones and delete obsolete ones. When deleting the default profile, the user is warned, but may still proceed if he so chooses.

We decided against showing everything in one view, because it would feel very cluttered. Therefore, we added a pop-up to edit profile details.

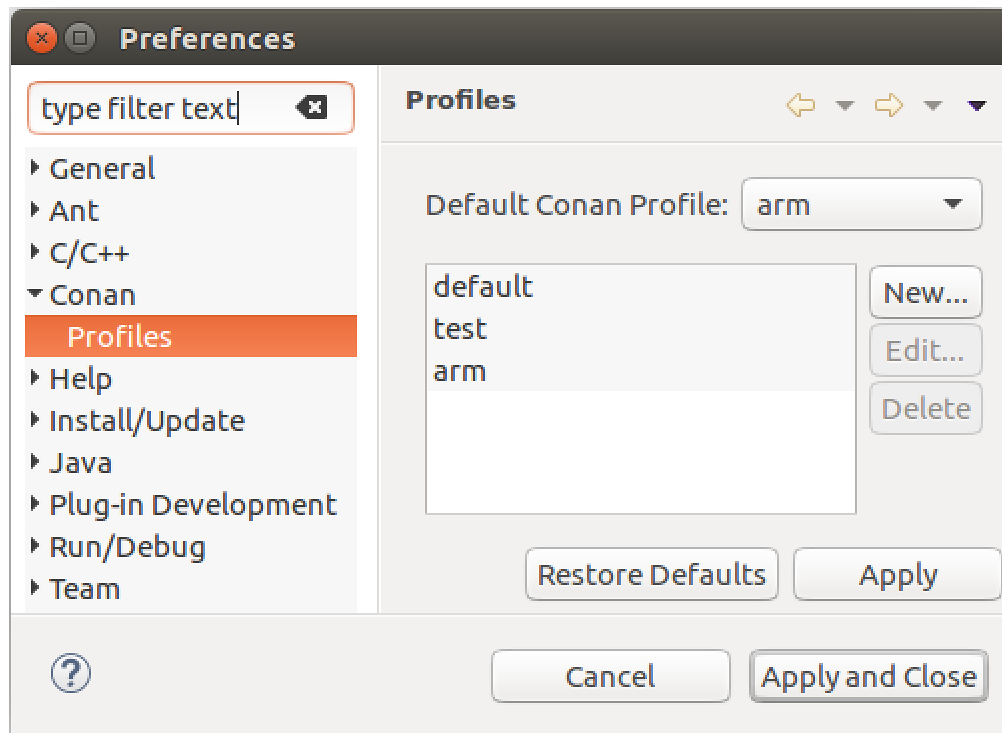


Figure 7 Conan profiles preference page

4.5.1 Edit Profiles

When selecting a profile, and clicking the edit-button, a new window is opened (see Figure 8). In this window, the user can edit a profile without having to edit the source file. The plug-in will handle everything in the background. Namely, the user may add or remove profile dependencies, define variables, and so on.

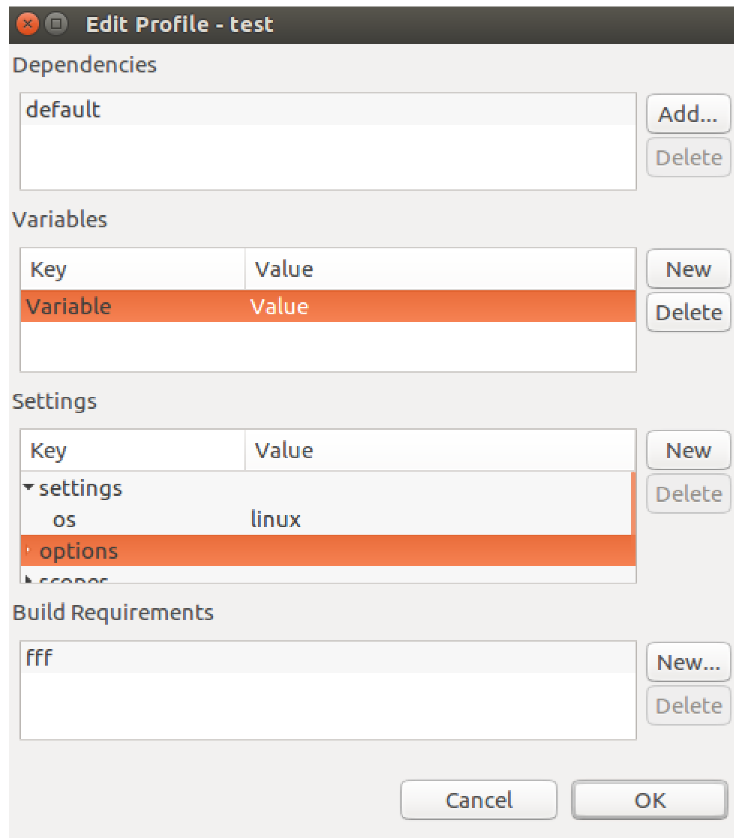


Figure 8 Profile detail view

To edit an entry, the user can double-click on its key or value cell to edit the respective field. For easy editing, the user may tab through all visible entries and does not need to double-click each cell individually.

There are no checks to prevent duplicate entries, since this is not a critical issue. If there are duplicate settings, only the last one will be respected by Conan.

The only part that behaves differently, is adding dependencies. They may only be added or removed, not edited. When adding a dependency, a new window opens, and the user may pick one of the existing profiles that are not already in its dependency list. The plug-in then checks if there are any cycles. If there are, the user is notified and is prevented from adding the profile (Figure 9). This is done because there is no check by Conan. Having dependency cycles will lead to a stack-overflow when executing the install command with the affected profile selected.

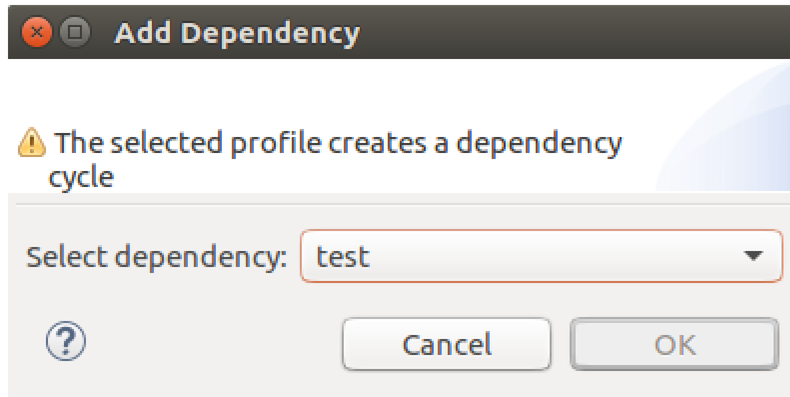


Figure 9 cycle detected

4.5.2 Saving Changes

No changes are saved permanently until the apply-button in the preference page is pressed. When it is pressed, the current state will be saved to all changed files. If necessary, new profile-files will be created or deleted.

When deciding whether to save the profiles to files or store them internally in Eclipse preference stores, we found that the latter would not make any sense, because Conan needs access to them anyway and storing them in a preference store would just add unnecessary redundancy.

As discussed in 3.5, using the Conan CLI is rather slow. Directly editing the file would speed up the process tremendously. Therefore, the parser used for build-info is used here as well.

4.5.3 Project Specific Profile

As mentioned before, the dialogs shown above are located in the Eclipse preferences and used to configure different profiles and select a workspace default. However, since the user will not want to use the same profile for all his projects and changing the default profile every time he switches to another project is quit a hassle, we also added the possibility to select a different profile for each project.

Since this is configured on project level and not on workspace level, a new property page was added to the project properties, see Figure 10.

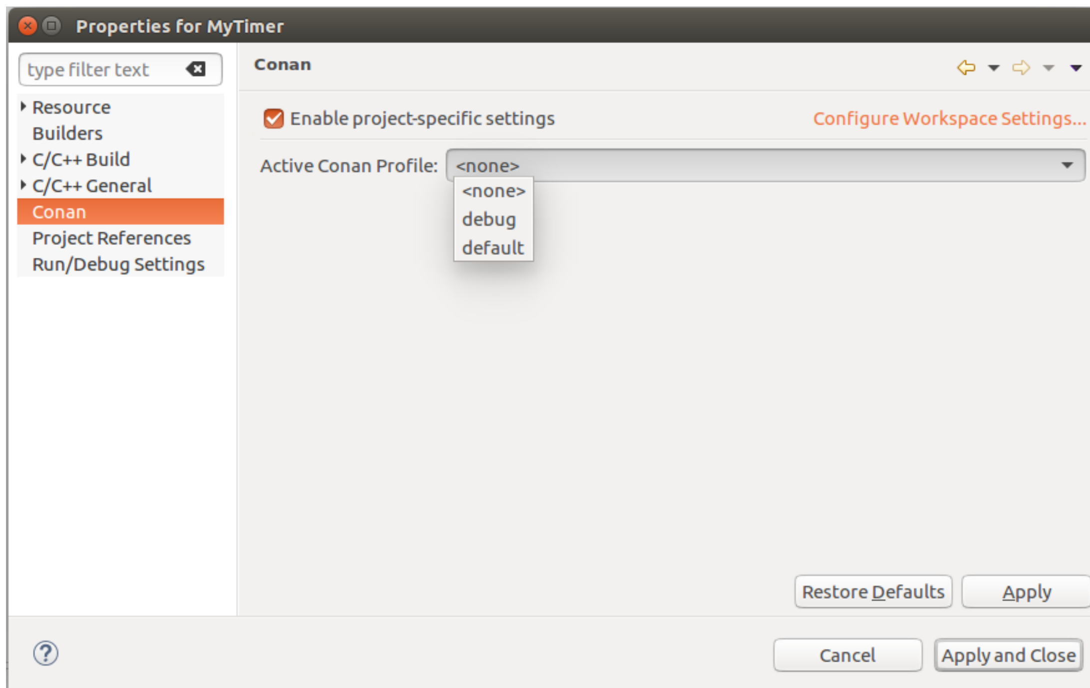


Figure 10 Conan properties page

The page provides three different configuration options:

1. Disable the project specific settings → Use the workspace default
2. Select <none> as the active profile → Use Conan's default profile
3. Select an actual profile from the dropdown → Use the selected profile

The link shown in the top right corner of the property page is a shortcut to the plug-in's preference page where the profiles can be modified.

4.6 The Parsers

As mentioned in the analysis chapter, there are multiple files we have to parse, the first one being conanbuildinfo.txt. Its structure is very similar to that of so-called ini-files, see Listing 4. Because of that similarity we thought of using a 3rd-party ini-parser called ini4j [10] instead of creating our own. This idea ended in failure as our section entries did not conform to the ini-format, as shown in Listing 6.

```
ini-file entries:  key=value
Conan entries:  value
```

Listing 6 ini-file entries vs conanbuildinfo.txt entries

Since ini-parsers did not solve our problem we decided to create our own. But at that time we did not expect that there would be more files that we would need to parse, which is why we created a parser specifically for this type of file. It scans the file line by line and associates all entries with their corresponding section's name and provides a method that returns a list of all entries for a given section name.

Later on, during the creation of the preference pages for Conan remotes and profiles, we encountered two more file-types that needed to be parsed. The ones containing the profiles and remotes. Because of the differences in the file structures, see **Fehler! Verweisquelle konnte nicht gefunden werden.** and Listing 5, the initial parser needed to be updated. The three main differences are:

1. The files contains different types of entries
2. The profiles can contain an unnamed section at the beginning of the file
3. The sections in the file containing the remotes have a fixed order instead of names

The first difference did not matter too much. We decided to treat all types of entries the same, as simple strings, meaning that we did not split key-value-pairs into keys and values. If that ever needed to be done, it could be done by the code using the parser instead of by the parser itself. Currently the only place this this is done is in the profile preference page.

The second difference meant that the parser would have to be adapted so that it would associate all entries occurring before the first section name with an extra section without a name.

The third difference was the one that posed the greatest problem. The fact, that the sections in the file containing the remotes do not have names and are instead identified by their order, makes the file structures too different from each other to be handled by a single parser implementation.

But since all those files still have a lot in common and we did not know if there were other files we would have to parse in the future, we decided to create more generalized versions of the initial parser that are able to handle multiple file-types with slight differences between each other. To do that we split the three files we had to parse in groups and tried to name these groups in a way that indicates what all files of a group have in common. We came up with the following grouping:

Named Sections

- conanbuildinfo.txt
- Conan profiles

Ordered Sections

- Conan remotes (registry.txt)

As the names imply, the first group consists of files that contain named sections, whereas the second group’s files contain sections with a fixed order.

The last decision we had to make was whether a parser should also provide the ability to modify the parsed file or if that should be done somewhere else. When we still only had our conanbuildinfo.txt-parser we did not need that functionality because there was no need to make any changes to that file. But the profiles and remotes should be editable through the plug-in, meaning that the plug-in has to be able to save these changes back to the files. After some research we found that parsers being able to not just read but also write files is quite common. That made sense to us since a parser already has all the information about a file that it needs to change it. For that reason we decided to extend the parsers and enable them to modify the files they parse.

5 Implementation

This chapter describes the implementation details for each function point mentioned in the design chapter.

5.1 Conan Install

As mentioned before, the install command was added to the context menu of the project explorer. Adding a command to any kind of context menu is actually quite simple and done so in a file called plugin.xml. Only a brief explanation is given here. For a more detailed description of Eclipse's menu contributions, refer to [11].

In plugin.xml a new menu contribution must be added to Eclipse's menu extension point "org.eclipse.ui.menus". This menu contribution must contain a location URI that defines the menu to which it is added and a command whose id points to the class handling its execution. In our case that class is called InstallHandler.

InstallHandler defines the execute-method, which is called whenever the command is invoked. This method can be split into four main steps:

1. It gets the project to which the right-clicked project explorer entry belongs to.
2. Creates a new Eclipse console called Conan, where the output of the command will be shown.
3. Executes Conan's install command by using our helper class ConanCliCommand, which handles the execution of all Conan commands. ConanCliCommand is described in more detail in the chapter "Conan Installation Path".
4. Stores the build information provided by Conan's install command in the IDE's build settings.

5.1.1 Storing Build Information

CDT's ManagedBuildManager handles saving build info on a per-project basis. With setOption(..) any option can be set programmatically; with saveBuildInfo(..) changes can be saved. However, by doing only that, the index will not be updated, which means that symbol resolution will not work. The most obvious symptoms are that the project will seem full of errors and syntax highlighting will not work – the project will still build though. Therefore, one must invoke the command in Listing 7 to update the index.

```
1:  CCorePlugin.getIndexManager().reindex(CoreModel.getDefault().create(project));
```

Listing 7 Update the index

5.1.1.1 The Intricacies of Eclipse CDT

As discussed in 4.1, Eclipse CDT uses the class Option to save build settings. Options can have different value types beyond the basic value types. There is a type for libraries, for library paths, for include paths, and so on. When calling getStringListValue(), the method checks whether the concrete option-type corresponds to the non-base value type STRING_LIST. Meaning, if its type is anything but STRING_LIST, it will throw an exception – see Listing 8. And since for example the option for include directories has value type INCLUDE_FILES, calling getStringListValue() will throw an exception. To circumvent this, getBasicStringListValue() must be called in this case. The reason being that it would make the code less maintainable and readable if one were to use the specific method for each value type – for reference: the specific method for include directories would be getIncludePaths().

For the same reason, the method getBasicValueType() must be called to check if the type is STRING_LIST or STRING, to distinguish between the two types. The distinction is necessary, because there is not just a single get-method, they are different for each type. But more importantly, there are different overloads of the ManagedBuildManager.setOption(..) method. This makes it impossible to get rid of the duplicated code.

```
1:  if (getValueType() != STRING_LIST) {
2:      throw new BuildException(ManagedMakeMessages
                .getResourceString("Option.error.bad_value_type")); //$NON-NLS-1$
3:  }
```

Listing 8 getStringListValue() type check

5.1.2 Remembering Which Build Information Was Added

When saving Conan build info, the plug-in stores everything it added to the preferences of the current project scope with the help of ConanBuildInfoPrefStore. Upon the next execution of Conan install, it recalls the stored values, deletes them from the current CDT options and adds the new ones received from the updated conanbuildinfo.txt file.

5.2 Conan Installation Path

There are mainly three classes involved in handling the configuration and validity check of the Conan installation path.

5.2.1 ConanPreferencePage

ConanPreferencePage is the preference page where the user can configure the installation path (see Figure 4). It uses a simple FileFieldEditor which provides all of the needed functionality and UI controls:

- A label, text field and browse button
- Validity check of the given path
- Storing the path in a preference store

5.2.2 ConanNotFoundDialog

Just like the preference page, this dialog (see Figure 5) uses a FileFieldEditor to get the user input. Additionally there is a checkbox that controls whether the given path will be saved in the Eclipse preference store or just used once.

5.2.3 ConanCliCommand

ConanCliCommand is responsible for the execution of all Conan commands.

```
public ConanCliCommand(final String[] arguments)
```

The constructor takes the command arguments that will be added to the Conan command. It also checks if the configured installation path is valid and if not, tries to find the Conan executable through the PATH variable. If even that does not work, an empty path is used and the command will not be able to execute, which is why classes creating a ConanCliCommand should always call canExecute() before actually executing the command.

```
public boolean canExecute()
```

This method returns whether the command will be able to execute or not, meaning it checks if the currently used path is valid. If not, it will return false and the caller can choose what to do next. That can involve aborting the command and providing an error message for the user or opening the ConanNotFoundDialog and setting the path given by the user through ConanCliCommand's setExecutable-method.

---

public boolean execute(**final** OutputStream out, **final** OutputStream err)

As the name implies, this method is responsible for the execution of the command. It uses Eclipse CDT's CommandLauncher class to do that and writes all output or errors to the corresponding streams. By using MessageConsoleStreams the output can be printed on the Eclipse console. It also returns a Boolean indicating whether the execution was successful, which is indicated by the exit code 0 of the command. This method should not be used for commands that take a long time to complete like, for example, the install command, since executing it on the UI thread will block the UI. For such cases the executeAsync-method should be used.

---

public void executeAsync(**final** OutputStream out, **final** OutputStream err, **final** Consumer<Boolean> callback)

This method does the same as the execute-method, but uses a separate thread for the command execution, meaning that it will not block the UI. Since this is an asynchronous method, instead of returning a Boolean to the caller, it will execute the callback with said Boolean as a parameter. The callback is run on the UI thread.

5.3 Conan Remotes

The list and its corresponding buttons used to manage the Conan remotes (see Figure 4) are handled in the class RemoteTableViewer. As the name suggests, this class creates a JFace TableViewer to present the remotes, which uses the MVC pattern to update its view based on the model. To get this to work, we mainly needed to do two things:

1. Use the JFace ViewerSupport's bind-method to create a data binding between the viewer and model, so that the table shown on the preference page is updated whenever the list containing the table entries is altered.
2. Implement PropertyChangeSupport for the table entries, which are instances of the class Remote, so that the view is notified of changes to properties of individual entries.

Since PropertyChangeSupport is also needed in other models it was moved to the ModelBase-class that all models inherit from.

5.3.1 Default and Active Model

The preference page contains two models for the TableViewer: The one that is currently active and being modified by the user and the one that represents the default values.

Whenever the preference page is opened, the file containing the remotes is parsed with the OrderedSectionParser and stored as the active model. This model is then deep-cloned to create the default model. It has to be deep-cloned because it contains object references, meaning that a shallow clone of the active model would still reference the same objects, thus causing the default model to contain changes of the active model. When a user presses the "Restore Defaults" button, a clone of the

default model, which has not changed, becomes the new active model, thus returning the TableViewer to its initial state.

The default model always contains the exact same values as the file, meaning that whenever a user saves his changes to the file using the “Apply” button, the active model is cloned anew to create an updated default model.

5.4 Conan Profiles

Initially, we removed the apply and default buttons, because we felt like there was no need for them, for we intended to apply any changes directly to the respective files. But now, changes are not committed to file immediately, but only after pressing the apply-button. If the user closes the preferences page and reopens it or hits the defaults-button, any changes are reverted.

5.4.1 Profile List

To list the profiles, a simple JFace ListViewer is used. If more information is needed at a glance (e.g. what build configuration the profile is for), it may still be substituted for a TableViewer with fairly low effort.

To implement the default button functionality, a backup model is used that is cloned from the model in use. When the user presses the button, the current model is overwritten with the default model. If the apply-button is pressed, the current model is cloned to the backup model.

5.4.2 Edit Profile View

When opening the EditProfileDialog, the selected profile is cloned for easy rollback if the user does not want to commit changes made. If the user hits OK in this window, the original profile instance is replaced with its clone.

Initially, we decided to use a TreeViewer to display the profile details for the built-in hierarchy, so we could display the different sections. However, since not every section was the same (as discussed in the analysis chapter), we split up the view. Specifically, entries in sections like environment variables are key-value-pairs separated by an equals sign. Profile inclusions and build requirements on the other hand are not. Thus, it does not make sense for them to be in the same table with key and value columns.

Now, there is a ListViewer for profile dependencies. When developing the dependency cycle check, it was discovered that using a list of ConanProfile references led to inconsistencies. This is due to the deep cloning strategy used for the default model and when opening the edit dialog.

As can be seen in Figure 11, when the dependency list is implemented as a ConanProfile-list with references to the actual profiles, there is a problem. Due to the deep cloning strategy, dependencies references will point towards outdated model instances. This has the consequence that when checking for cycles, actual cycles will not be detected.

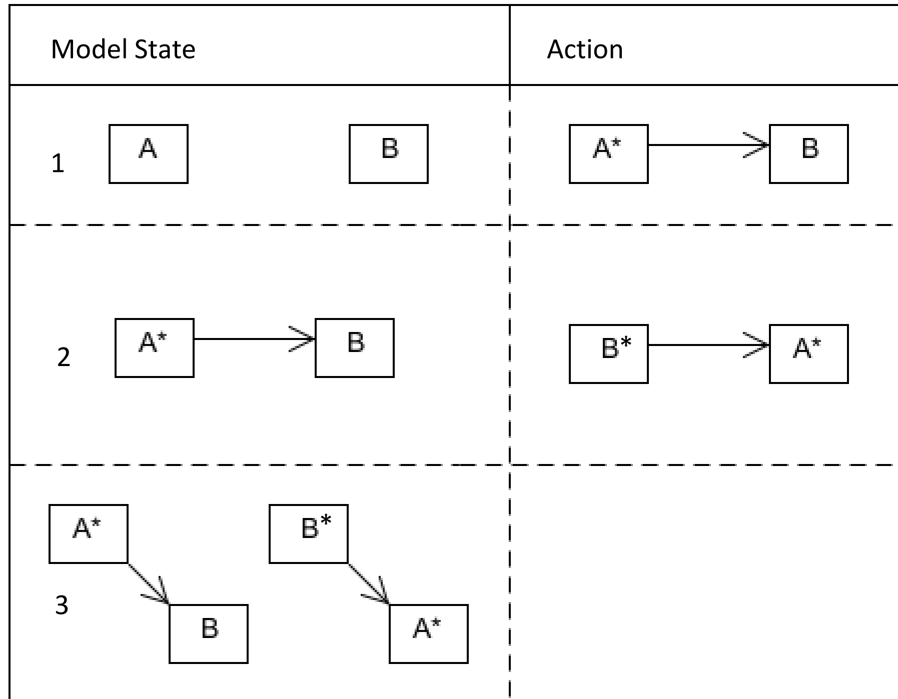


Figure 11 Adding dependencies sequence

Thus, it was decided to use a list of strings for dependencies. This is not an elegant solution and is only an attempt to mend the problem. If in the future a different default mechanism is used, that handles references better, this should be revisited.

A single-column table is used for the build requirements list. Using a ListViewer is not possible, because there is only editing support for ColumnViewer. Since ListViewer does not inherit from ColumnViewer, TableViewer must be used. This is important because we wanted to make the experience as homogenous as possible.

5.4.3 Project Specific Profile

ConanPropertyPage is the class handling the property page containing the configuration of the project specific profile. This page is rather simple as it only contains two controls:

1. A checkbox to enable/disable the use of project specific settings
2. A combo box to select one of the profiles (or <none>)

Both controls are linked to the plug-in's preference store where the last applied settings are saved as a Boolean or String respectively. An important thing to note here is that the selected value of the combo box will not always be saved as is. If the checkbox is disabled or the <none>-entry is selected, the value saved in the preference store will always be the empty string. This simplifies the process of getting the active profile of a project, which is done through the class ConanProfilePreferenceUtility. This class simply checks whether the project specific settings were enabled and then returns the corresponding profile (enabled: project specific, disabled: workspace default). Since only the empty string or a profile name can be stored in the preferences, there is no need for additional checks like for example if <none> was selected.

5.5 The Parsers

There are currently two concrete parser implementations that inherit from one common base class:

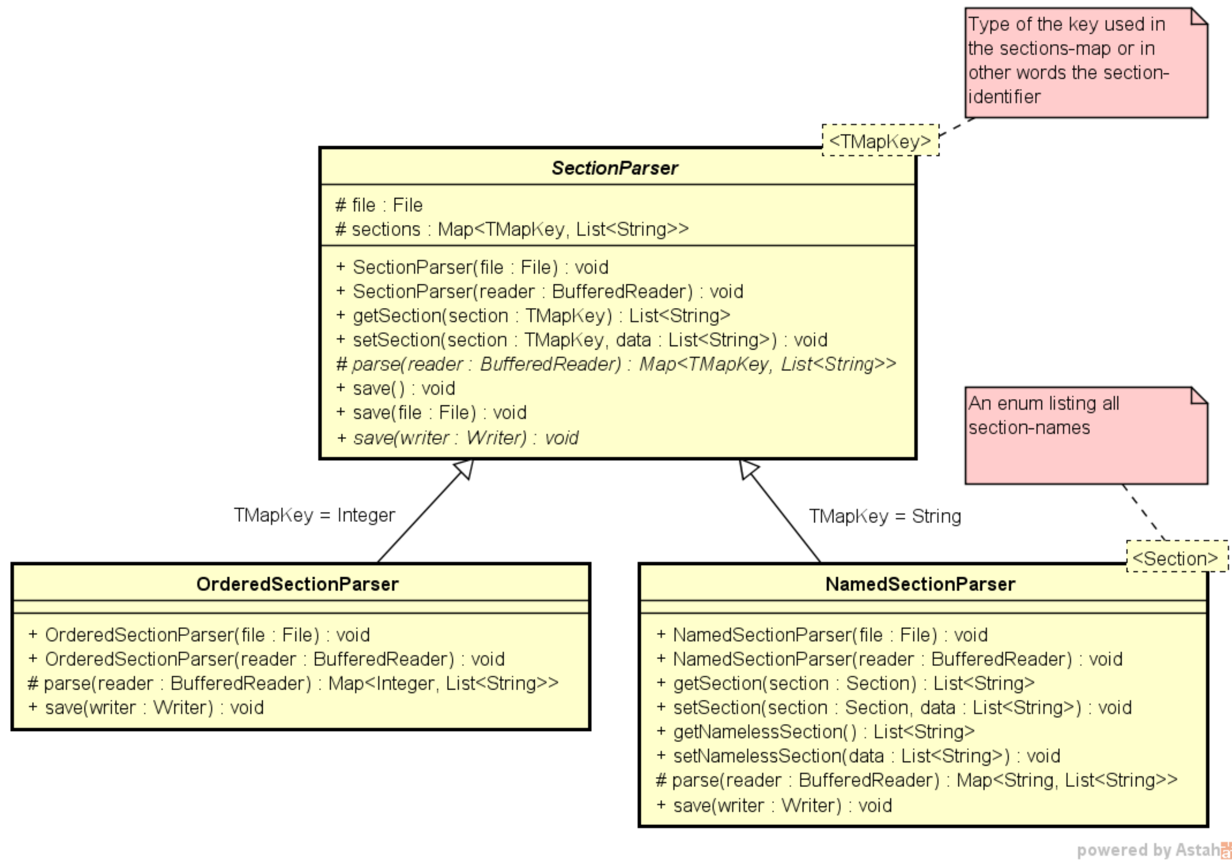


Figure 12 SectionParser(s) class diagram

Note: The parsers define multiple constructors and save-methods. This was done to enable unit-testing of the parsers without the use of actual files through dependency injection.

5.5.1 SectionParser

SectionParser is the abstract base class for the two concrete implementations. It is abstract because it only defines the basic structure that all parsers should implement and some default implementations for simple methods like get- and setSection.

It does not provide any algorithms for parsing files and writing the sections back to a file, which is why the parse- and save-method are declared as abstract. That is the job of the concrete parser implementations.

---

public SectionParser(File file)

The constructor takes a File as a parameter, wraps a BufferedReader around it and then invokes the parse-method. That means that the whole file is parsed just once, when the parser is created. This is much more efficient than parsing each section individually when it is needed, since reopening the file and jumping to a certain section's position each time the getSection() is called takes a lot of time and in most cases the program will need all sections anyways (half a profile is not really useful). The generated

Map containing the section identifiers and their corresponding entries is then saved in the member-variable **protected** Map<TMapKey, List<String>> sections. TMapKey is the type of the keys that identify the different sections. For our two implementations that is either String (named) or Integer (ordered), as shown in the class diagram above (Figure 12).

protected abstract Map<TMapKey, List<String>> parse(BufferedReader reader)

The parse-method is supposed to parse the whole input coming from the BufferedReader and return a Map containing Lists of entries mapped to their corresponding section keys. It is declared as protected because it should only be called once from the constructor, so that the file is parsed only once. Since this is already done in SectionParser's constructor, subclasses should always call **super**(file) in their own constructor.

public abstract void save(Writer writer)

The save-method is responsible for writing all sections currently present in the parser's Map to the given Writer in a certain format, whereas the format is determined by the file that parser was made for. Just like the parse-method it is supposed to write all sections at once, not each section individually. This is important, because in most cases the Writer will be a FileWriter and writing to a file each time setSection() is called is much slower than changing the Map and writing the file only once, when this method is called.

5.5.2 NamedSectionParser

As the name suggests, NamedSectionParser is responsible for all files containing named sections, which currently include conanbuildinfo.txt and the Conan profiles (see Listing 4 and **Fehler! Verweisquelle konnte nicht gefunden werden.**).

public class NamedSectionParser<Section **extends** Enum<Section>> **extends** SectionParser<String>

This class extends SectionParser using String as the type parameter meaning that the section identifiers/names are represented as Strings. Its own type parameter, called Section, is an Enum that contains the names of the sections that will be used by the part of the program creating the parser-instance. This, for example, is the Enum used for the parser-instance reading the Conan profiles:

```
public enum Section {
	settings,
	options,
	scopes,
	env,
	build_requires
}
```

Listing 9 Section-Enum for Conan profiles

We added this type parameter and the get- and setSection-methods taking an Enum-value as a parameter so that the sections did not have to be accessed by using plain Strings. The benefit of this is that renaming a section only has to be done in a single location and it also prevents typos in section names.

Note: The parser still parses all sections, not just the ones given in the Enum. And the fact that a section name is contained in the Enum also does not mean that it will be present in the parsed file.

The get- and setSection-methods of the parent class taking a String as a parameter can of course still be used if necessary.

public List<String> getNamelessSection()
public void setNamelessSection(List<String> data)

Since some files (e.g. profiles) can contain an extra section without a name at the beginning of the file (see **Fehler! Verweisquelle konnte nicht gefunden werden.**), there must be a way to store and retrieve this section. We decided to store it in the Map, just like all other sections, using an empty String ("") as the key and provide an extra getter and setter for it. The benefit of this is that all sections are stored in the same place and no extra variables need to be considered when looping over all sections like, for example, in the save-method.

Important: The nameless section is very special because, unlike the other sections, it has a fixed place (beginning of the file). This must be respected when writing the sections back to a file. Currently this is done by using the characteristics of the Java HashMap and String.hashCode(). The hash code of an empty String will always be 0, meaning that using this as a key for a HashMap-entry will always result in it being the first element when iterating over said Map's entries. But this is only true under one condition: There must be no other entry present in the Map's 0-bucket when it is inserted, since new entries are always inserted at the end of the list of entries in a bucket. This is ensured by parsing and storing the nameless section first and not providing the ability to remove it from the map, since removing and reinserting it would move it to the end of the bucket's list. Replacing the entry, which is what setSection() does, does not alter its position.

protected Map<String, List<String>> parse(BufferedReader reader)

As mentioned before, this is the method that actually parses the given file, or whatever kind of input the BufferedReader is wrapping. The algorithm is actually very simple. It just parses the input line by line, creating a List for each section name encountered and storing all non-empty lines in the corresponding Lists. The first list created is always the one for the nameless section, which can be seen as something like a default section containing all entries that do not correspond to an actual section. An example of a file being parsed and stored in a Map is shown in Listing 10.

File		Map
1: an entry 2:	Nameless section	{ "": ["an entry"
3: [SectionA] 4: 5: entry a1 6: entry a2 7:	SectionA	], "SectionA": ["entry a1", "entry a2"
8: [SectionB] 9: entry b1 10: 11: entry b2	SectionB	], "SectionB": ["entry b1", "entry b2"
12: [EmptySection] 13:	EmptySection	], "EmptySection": [ ] }

Listing 10 Parsing of file with named sections

As you can see, all empty lines, no matter where they occur, get ignored.

The Map created by this method is a HashMap. We chose this Map implementation because it is very fast and the order of the sections does not matter (except for the nameless section, as described above).

---

public void save(Writer writer)

This method does pretty much the opposite of the parse-method and in a just as simple manner. It uses two loops, the outer one iterating through the Map (sections) and the inner one through the Lists (entries), to create a file looking like the one shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**. That means the file content will conform to the following properties:

- Section names are put in [square brackets]
- An empty line after each section, which means ...
  - after its last entry
  - after the section name, if it has no entries
- No empty lines ...
  - within sections ...
    - between entries
    - between the section name and its first entry
  - at the start or end of the file

5.5.3 OrderedSectionParser

OrderedSectionParser is responsible for files containing sections with a fixed order. There is currently only one such file that needs to be parsed, which is the one containing the configured Conan remotes.

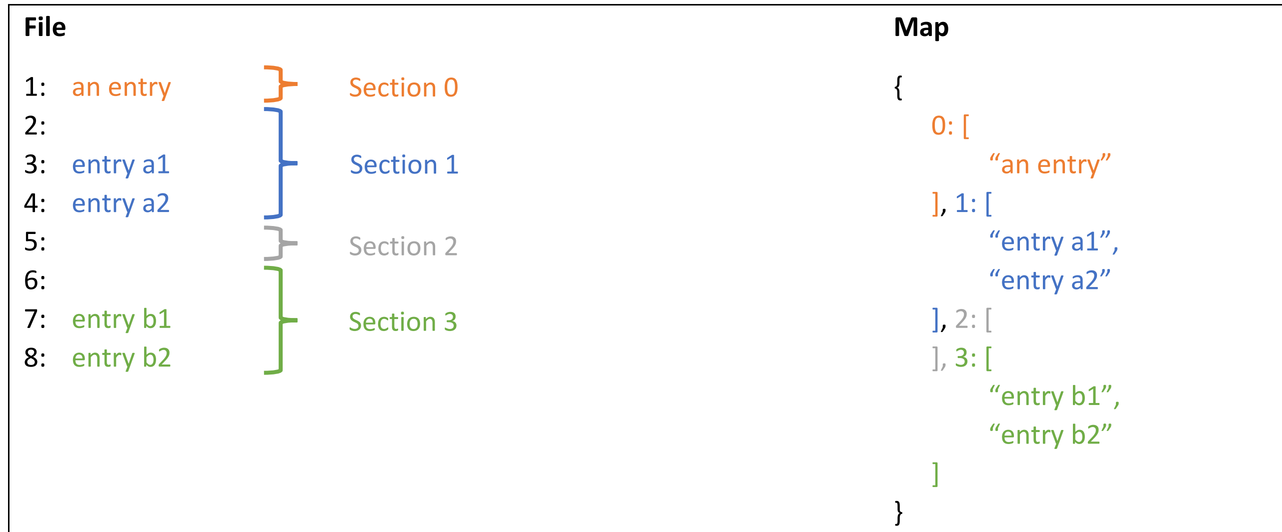
---

public class OrderedSectionParser **extends** SectionParser<Integer>

This class extends SectionParser using Integer as the type parameter meaning that the sections are identified through Integers, which makes sense since the sections have a fixed order and no names.

protected Map<Integer, List<String>> parse(BufferedReader reader)

This parse method is very similar to the one used in NamedSectionParser. But since the two file types do not really have anything to do with each other and combining both implementations would have resulted in an unnecessarily complex algorithm, we decided to keep them apart. The main difference is that empty lines actually matter in the OrderedSectionParser, because they indicate the start of a new section (see Listing 5). An example of how the parsing looks like can be seen in Listing 11.



Listing 11 Parsing of file with ordered sections

As you can see, each empty line causes the parser to create a new section. Even if there are two empty lines in succession, both will cause a new section to be created, with the first one having no entries (see lines 5 & 6 and section 2 above).

public void save(Writer writer)

Just like the parse-method, this method is also quite similar its counterpart in NamedSectionParser and they were not combined for the same reasons. It also uses two loops to iterate through the Map and its Lists, creating a file that looks like the one shown in Listing 5. That means the file content will conform to the following properties:

- An empty line after each section, which means ...
    - after its last entry
    - the section itself is only an empty line, if it has no entries
- No empty lines ...
    - within sections, since that would indicate a new section
    - at the end of the file, meaning that if the last sections in the Map are empty, they basically get ignored

6 Conclusion & Outlook

The following features have been implemented according to the project's assignment:

The user can install Conan packages with minimal effort using the plug-in. There is no need to copy the build information from the conanbuildinfo-file to Cevelop by hand anymore. The command is designed in a way that the caller does not need to worry about inconsistencies or lost user build settings that are not set by the Conan-plug-in. It isolates the build information of Conan only and replaces it with the new one. Functionally, the source code is where it should be. However, its style is lacking due to its dependency on Eclipse CDT. When Eclipse CDT is updated, the ConanBuildInfoManager class should be revisited for optimization.

The plug-in allows the user to manage Conan remotes and profiles from within Cevelop. Basic CRUD operations are available. The user may select a workspace default profile that can be overridden for any projects. The profile dependency cycle check solution is not optimal, as described in the corresponding design chapter. This may be remedied as described by a different cloning strategy or a different default model strategy. What's more, the entries in the profile's settings section may only have certain values. Thus, it would be beneficial to adjust the table to allow for the value column to be a combo-field with a predefined set of values. Potentially, that means splitting up the TreeViewer into multiple TableViewers. Then there would be one TableViewer per section.

A way to browse and install packages through Cevelop has not yet been implemented. The concequence of that is that the user must still edit the conanfile by hand. The idea was that it would look similar to the nuget GUI in Visual Studio with one tab for browsing all the available Conan remotes. Then the user could choose which packages he wanted and add it to the dependencies list with one click. A second tab would show all the installed packages. From there the user could update or remove them.

A second improvement would be assigning profiles not just for the active build configuration as it is done currently. Ideally, the user could assign one profile to the debug configuration and a different one to the release configuration. This would make the plug-in more flexible. Currently, to achieve the same effect the user must switch the profile when switching the build configuration. This was not done in this term project, because Eclipse CDT is very archaic, and it is very difficult to understand where the right functions are and how to use them. It was the goal to get the functionality to a working point, from where one could expand.

7 Project Management

7.1 Organization

The people involved with the project and their roles and responsibilities can be seen in Figure 13. Giovanni Heilmann and Pascal Schweizer are the developers actively working on the development, whereas Thomas Corbat and Felix Morgner have a supportive role.

Giovanni Heilmann	Pascal Schweizer	Thomas Corbat	Felix Morgner
• Documentation • Redmine	• Code Base (git) • CI Server (Jenkins)	• Supervisor	• Technical Advisor

Figure 13 Project organization

7.2 Process

The project duration amounts to 14 weeks, each week consisting of 17 working hours per developer, summing up to 34 hours per week and 480 hours in total.

A simplified version of the Agile Unified Process (AUP) [12] was used during this project. Simplified because the inception phase was not needed as that was already done by the supervisor, before the students started working on this project. As shown in Figure 14, the 14 weeks were split up into the remaining three traditional AUP phases, namely elaboration, construction and transition, with a sprint duration of two weeks.

Elaboration				Construction								Transition	
SW1	SW2	SW3	SW4	SW5	SW6	SW7	SW8	SW9	SW10	SW11	SW12	SW13	SW14
Sprint 1		Sprint 2		Sprint 3		Sprint 4		Sprint 5		Sprint 6		Sprint 7	

Figure 14 Project process

7.3 Quality Assurance

To ensure that the code is of adequate quality, a mutual code review was done at the end of the construction phase. Both developers reviewed each other's code and checked it for correctness, style and code smells. A list of all issues found during this review, including the corrections that are to be made, was then given to the respective developers. They then worked through these lists and applied all needed changes to the code.

8 References

[1] Conan: https://www.conan.io/
[2] Eclipse: https://www.eclipse.org/home/index.php
[3] Cevelop: https://www.cevelop.com/
[4] Conan install: http://docs.conan.io/en/latest/reference/commands/consumer/install.html
[5] Conan remote: http://docs.conan.io/en/latest/reference/commands/misc/remote.html
[6] Conan profile: http://docs.conan.io/en/latest/reference/commands/misc/profile.html
[7] Conan generators: http://docs.conan.io/en/latest/reference/generators.html
[8] txt-generator: http://conanio.readthedocs.io/en/latest/reference/generators/text.html
[9] ini-files: https://en.wikipedia.org/wiki/INI_file
[10] ini4j: http://ini4j.sourceforge.net/index.html
[11] Eclipse menu contributions: https://wiki.eclipse.org/Menu_Contributions
[12] Agile Unified Process: https://en.wikipedia.org/wiki/Agile_Unified_Process

9 Figures

Figure 1 Project properties 14
Figure 2 Architecture layers 19
Figure 3 Conan install 20
Figure 4 Conan preference page 21
Figure 5 Conan not found dialog 22
Figure 6 Edit remote dialog 23
Figure 7 Conan profiles preference page 24
Figure 8 Profile detail view 25
Figure 9 cycle detected 26
Figure 10 Conan properties page 27
Figure 11 Adding dependencies sequence 33
Figure 12 SectionParser(s) class diagram 34
Figure 13 Project organization 40
Figure 14 Project process 40

10 Listings

Listing 1 conanbuildinfo.txt example 13
Listing 2 registry.txt example 15
Listing 3 Conan profile example 16
Listing 4 conanbuildinfo.txt structure 17
Listing 5 Conan remotes file structure (registry.txt) 18
Listing 6 ini-file entries vs conanbuildinfo.txt entries 27
Listing 7 Update the index 29
Listing 8 getStringListValue() type check 30
Listing 9 Section-Enum for Conan profiles 35
Listing 10 Parsing of file with named sections 37
Listing 11 Parsing of file with ordered sections 38

Appendix A: Installation Manual

A.1 Requirements

The plug-in requires the following tools in the respective minimum version. It might, but is not guaranteed to work with older versions of these tools:

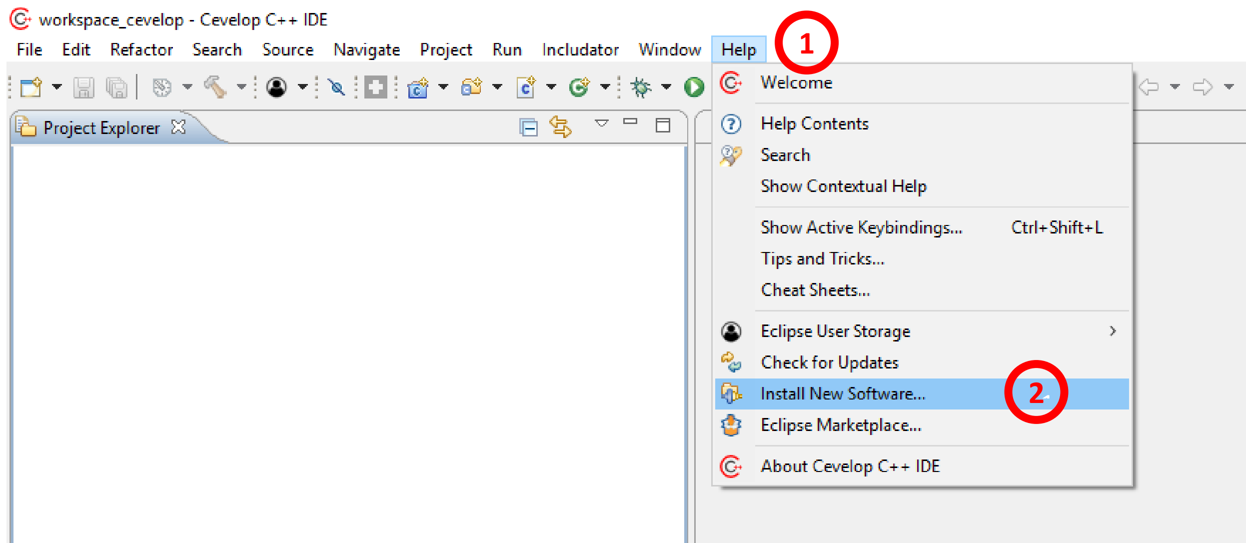
- Conan 0.27.0
  https://conan.io/downloads

- Python 2.7.12
  https://www.python.org/downloads/

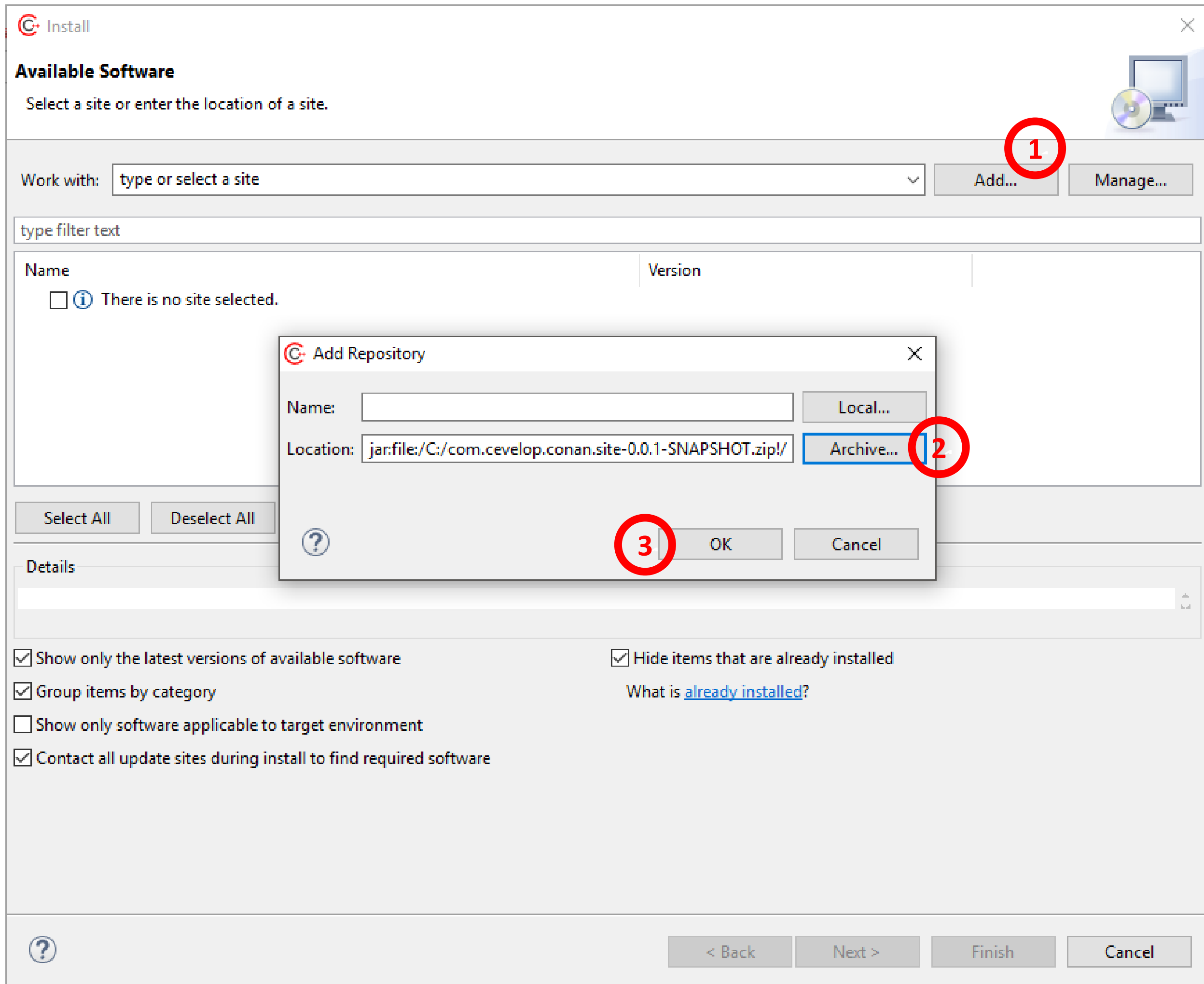
- Cevelop 1.8.0
  https://www.cevelop.com/download/
  OR
  Eclipse Oxygen 4.7.1 & Eclipse CDT 9.3.2
  http://www.eclipse.org/downloads/packages/eclipse-ide-cc-developers/oxygen2
- Java 1.8.0
  https://java.com/de/download/

A.2 Plug-in Installation

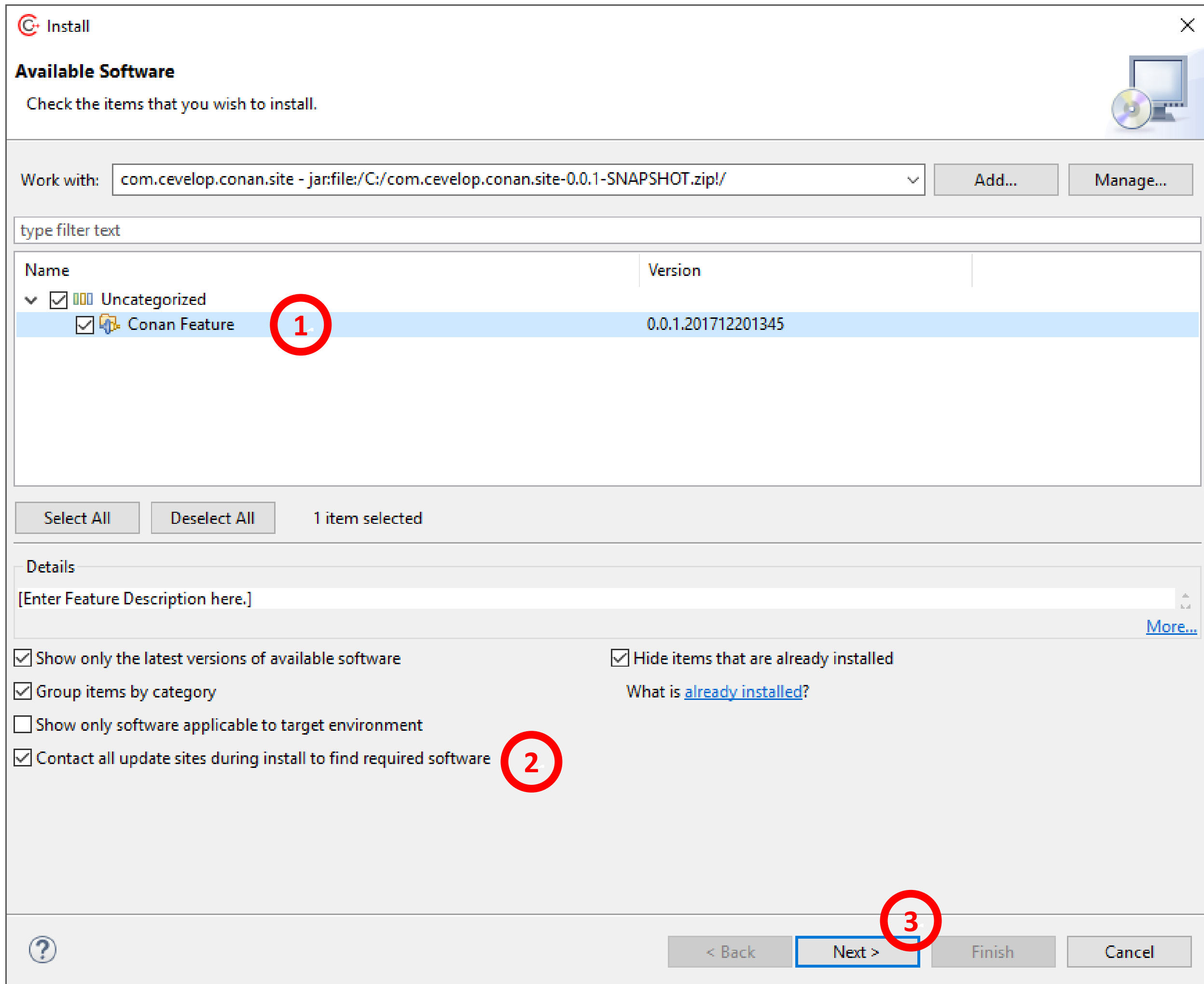
The plug-in installation procedure is the same for both Cevelop and Eclipse. The entry point to install plug-ins into the IDE can be found under “Help -> Install new Software...”.



In the window that pups up, press “Add...” to add a new update site or repository where plug-ins can be installed from. Then use the “Archive...”-button to select the zip-file called “com.cevelop.conan.site-0.0.1-SNAPSHOT.zip” containing the Conan plug-in and press “OK”.

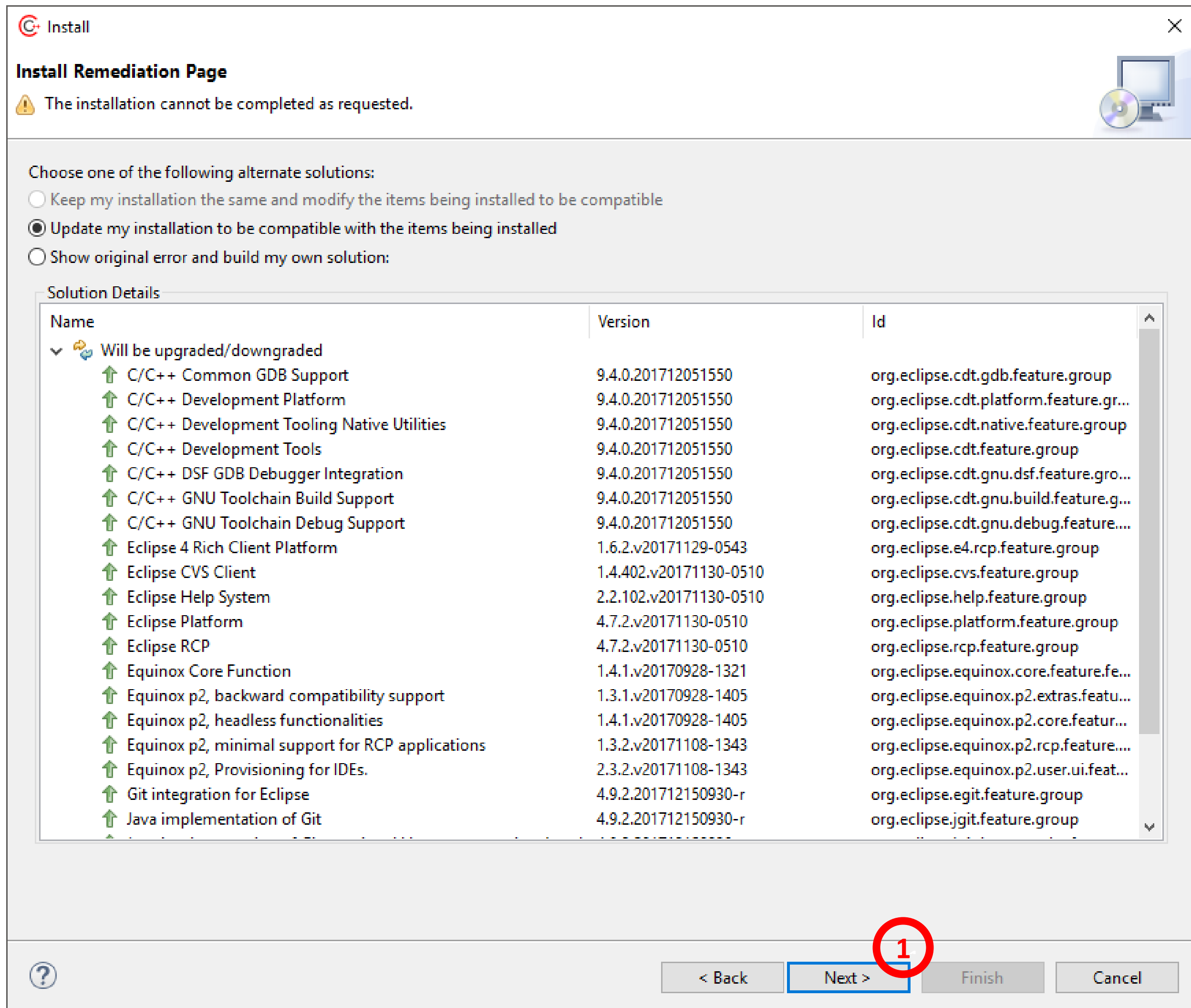


You will then be presented a list of all plug-ins available from the selected zip-file, which consists solely of the Conan Feature. Select it from the list. Make sure that the checkbox at the very bottom is checked, so that the installer is able to download all plug-in requirements. Then proceed by pressing “Next >”.

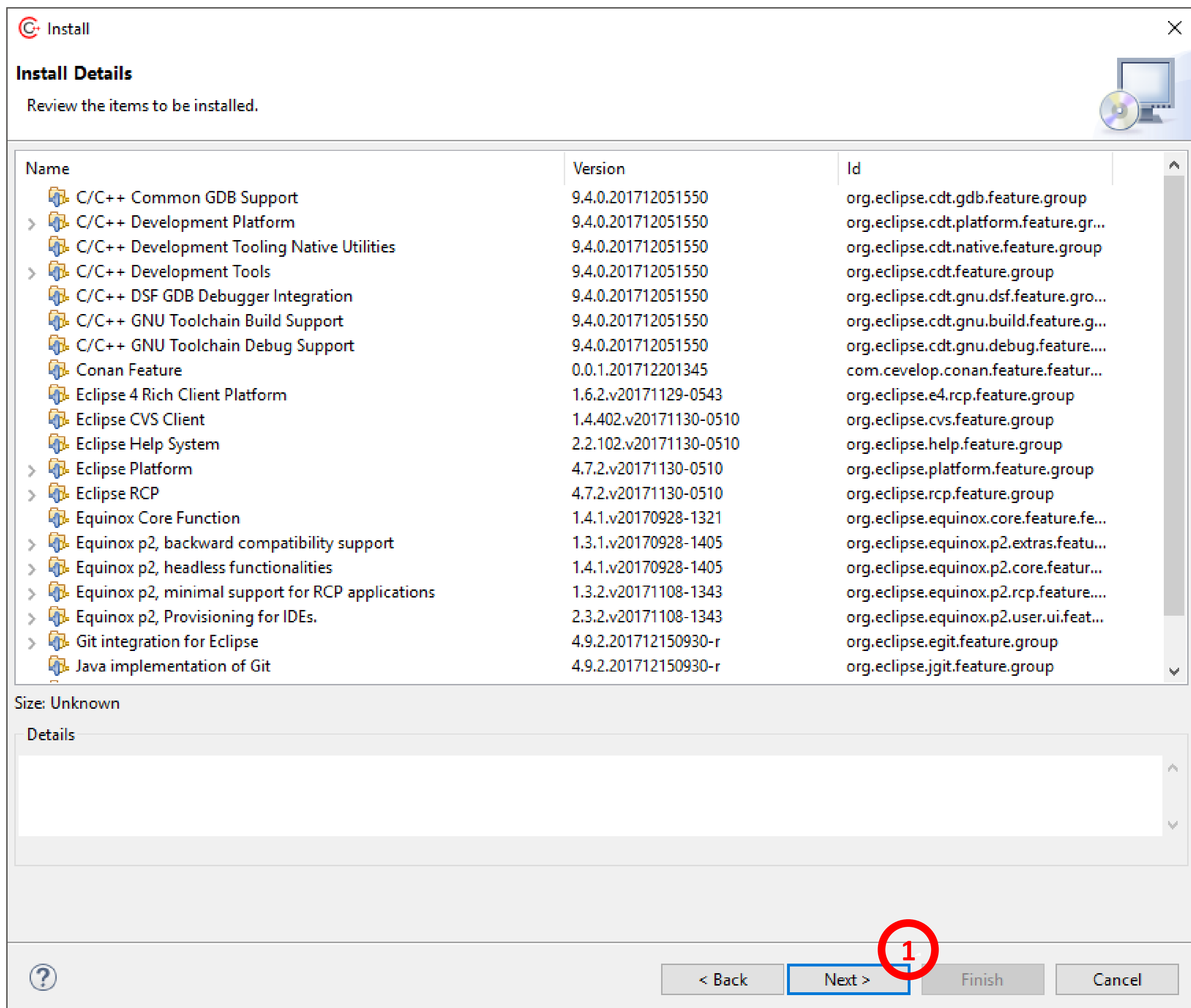


If everything was ok, you should then see a list showing what will be up-/downgraded so that the IDE is compatible with the plug-in you want to install. Press “Next >” two times and finish the installation by accepting the license agreements.

In case something went wrong and you are presented an error message, refer to section A.3 for a potential solution to your problem.



Press “Next >”.



Accept the license agreements and finish the installation.

Install

Review Licenses

Licenses must be reviewed and accepted before the software can be installed.

Licenses:

- [Enter License Description here.]
- Eclipse Foundation Software User Agreement

License text:

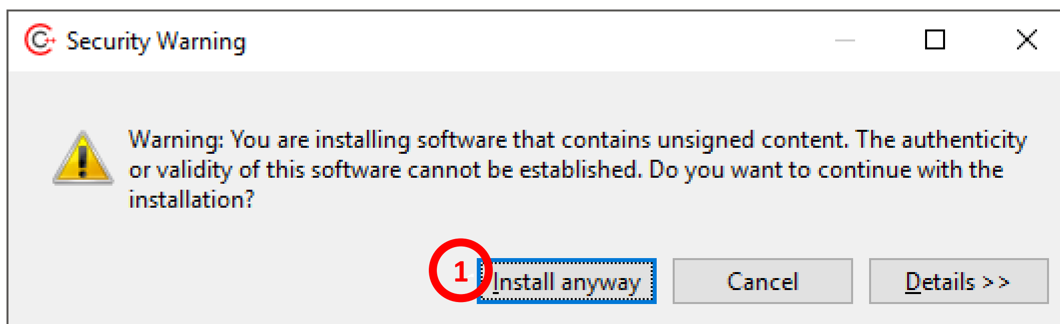
[Enter License Description here.]

(1) ◉ I accept the terms of the license agreements

○ I do not accept the terms of the license agreements

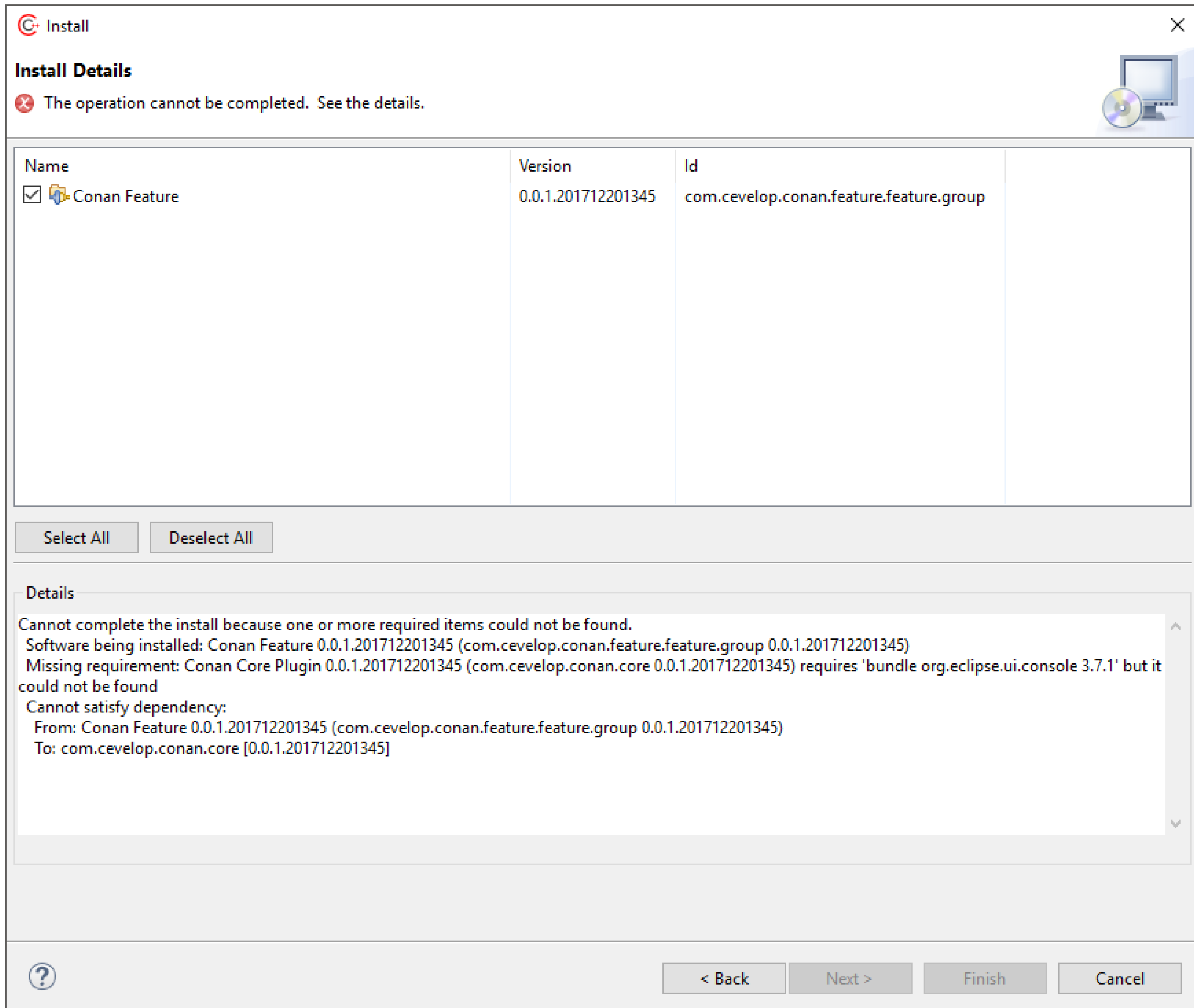
< Back | Next > | Finish (2) | Cancel

Since the plug-in is not signed, you will get a security warning during the installation process. You can safely ignore it and proceed by pressing “Install anyway”.



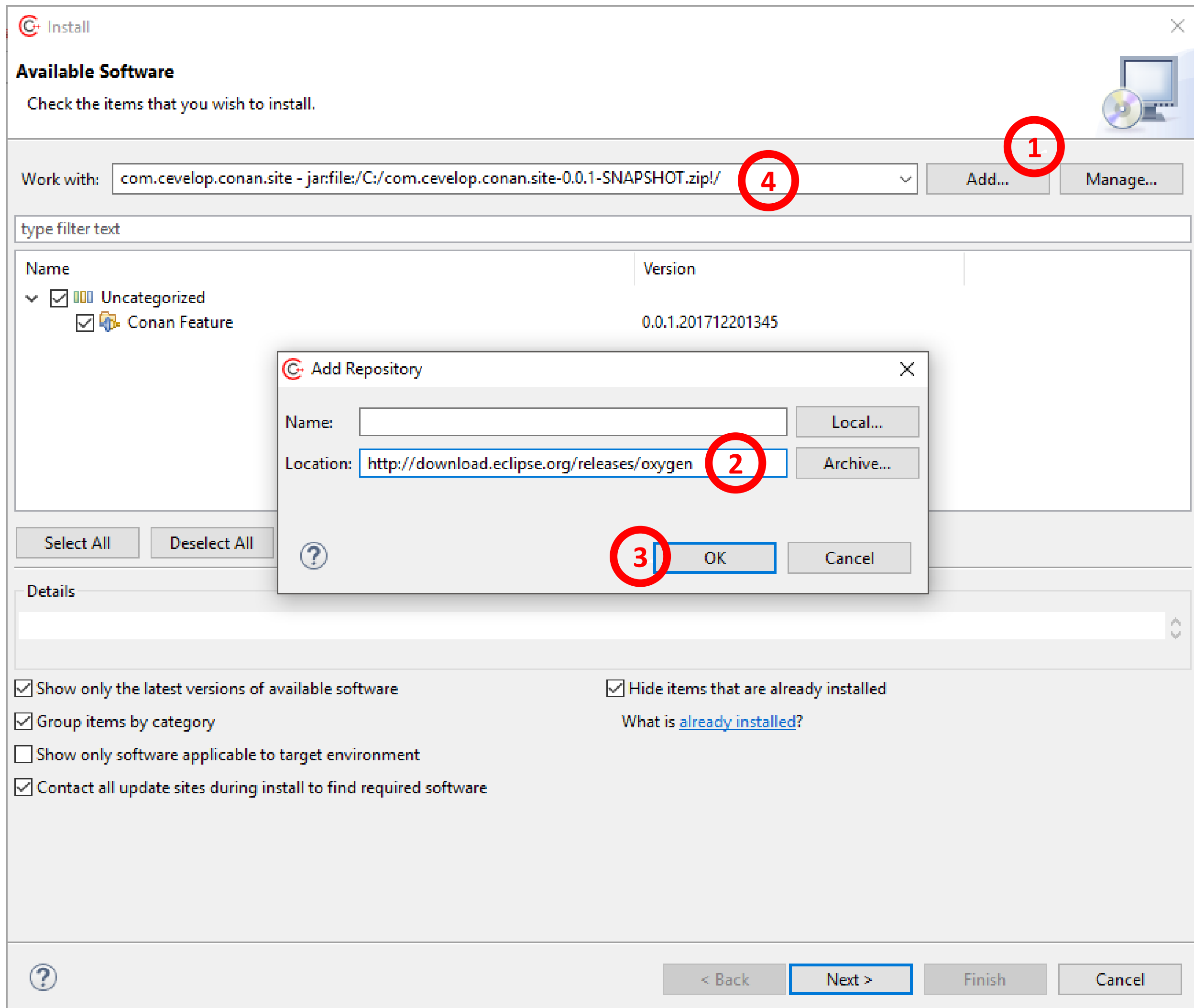
A.3 Installation Problems

A very common problem is the one shown in the picture below. It arises if the installer is unable to find one of its requirements, in this case "org.eclipse.ui.console". The cause of this can either be a broken internet connection or the lack of an update site from which the installer could download the missing requirement.



If your internet connection is fine, try going back to the previous screen, where you selected the zip-file and add Eclipse Oxygen's update site to the repositories. Click the "Add..."-button and then use "http://download.eclipse.org/releases/oxygen" as location. After pressing "OK" you will have to wait a few seconds until Eclipse is done gathering information about the update site. This is indicated with a green progress bar.

After that you can try to install the plug-in again by selecting the zip-file from the dropdown left of the "Add..."-button.



Appendix B: User Manual

B.1 Introduction

This document explains how to use the Conan for Cevelop plug-in. It describes all functionality of the plug-in. To use the plug-in, please install it first following the installation manual. To understand the explanations of this document, it is expected that you know basic functions of Conan and Eclipse as they will not be explained.

B.2 Install packages

Conan for Cevelop does not manage the conanfile. It is solely the user's task to maintain this configuration file. The plug-in can handle conanfile.txt or conanfile.py. If you need help with creating the file, please consult the conan.io manual at http://docs.conan.io/. The conanfile must be located in the respective project's root path, otherwise the plug-in will not recognize it.

To install all designated packages, click on the project in the project or package explorer and select Conan -> Install in the context menu, see Figure 15. The plug-in will automatically install the packages to Cevelop.

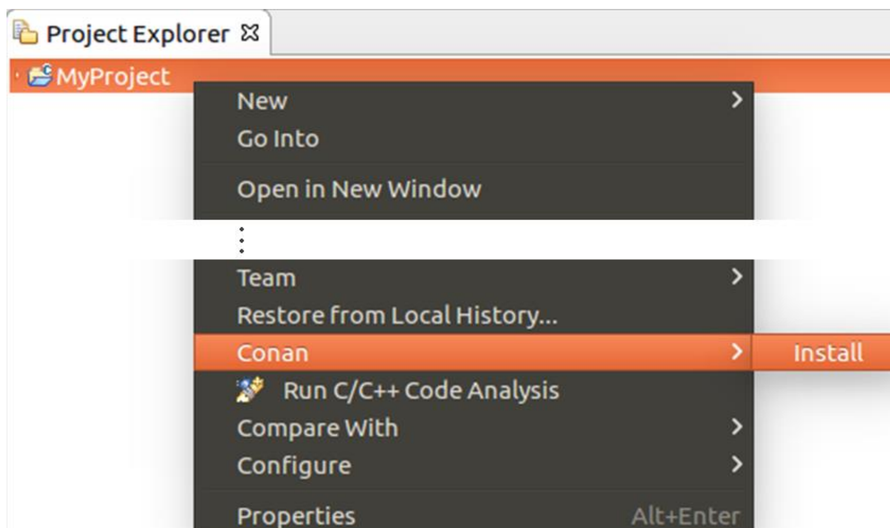


Figure 15 - Conan install

B.3 Set Conan installation path

In the Conan preference page, you may select the path to Conan if you do not wish to add it to the PATH symbol. See the field "Path to Conan" in Figure 16.

After you have made your changes, press "Apply" or "Apply and Close" to commit your changes.

Please note that this is optional if you have added Conan to the PATH symbol.

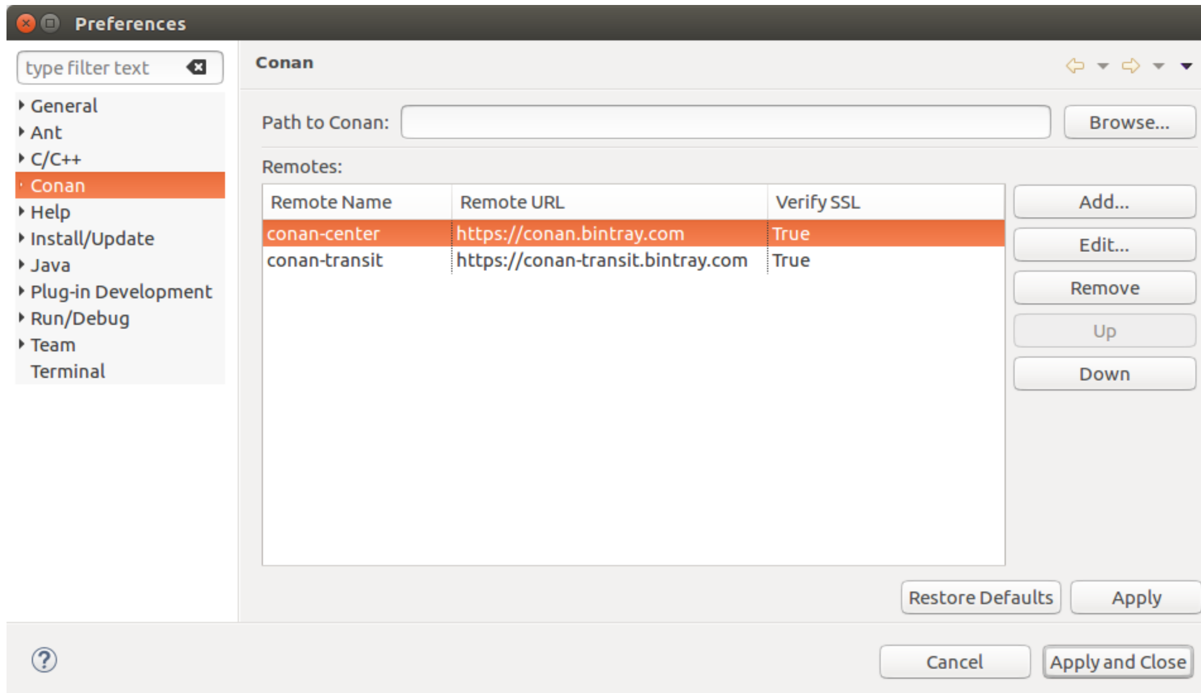


Figure 16 - Conan preference page

B.4 Manage Conan Remotes

You can manage your remotes in the Conan preference page as can be seen under “Remotes” in Figure 16. In the window that opens when adding or editing remotes, see Figure 17, enter the remote’s information.

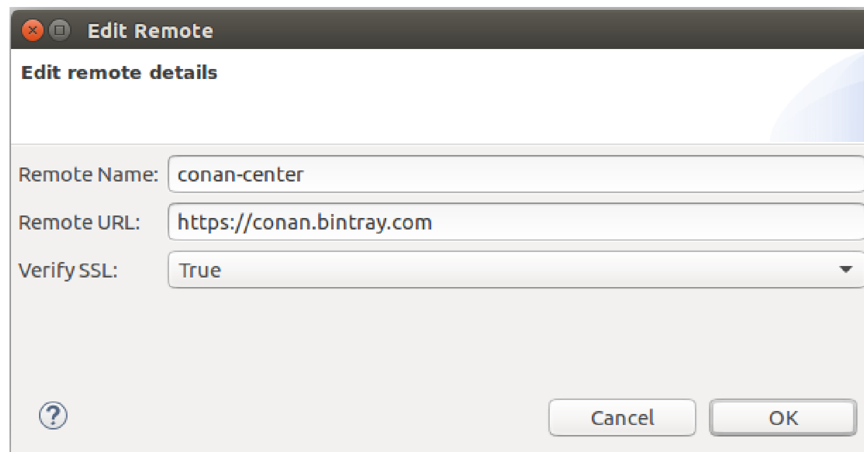


Figure 17 - Edit remote window

You may also reorder the remotes by selecting the desired remote and pressing the Up or Down button. When installing dependencies through the plug-in it will look for them sequentially in the configured remotes.

After you have made your changes, press “Apply” or “Apply and Close” to commit your changes.

B.5 Manage Conan Profiles

You can manage Conan profiles in the Conan profile preference page. It is a sub-page of the Conan preference page. Here you can select a default profile for the whole workspace.

After you have made your changes, press “Apply” or “Apply and Close” to commit your changes.

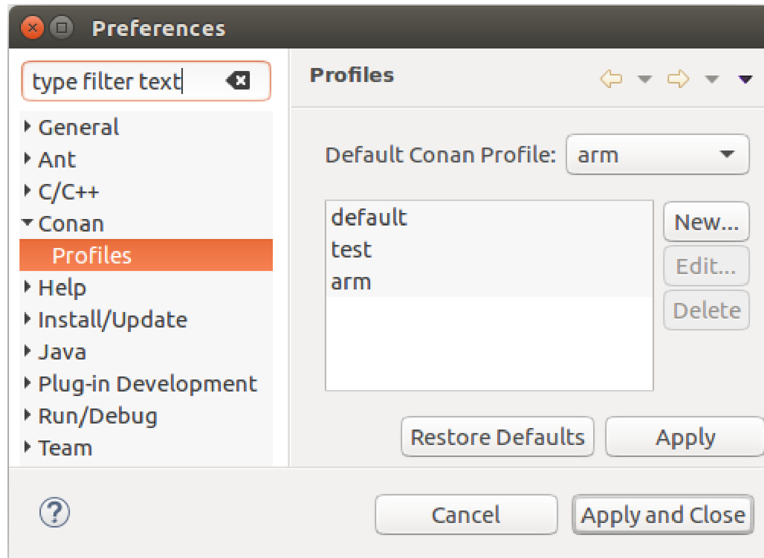


Figure 18 - Profiles preference page

To add a profile, follow these steps:

- Press the New button. This will take you to a new window.
- Enter a profile name.
- Press OK in the new window.

To edit a profile, select it and press Edit. This will take you to a new window, seen in Figure 19.

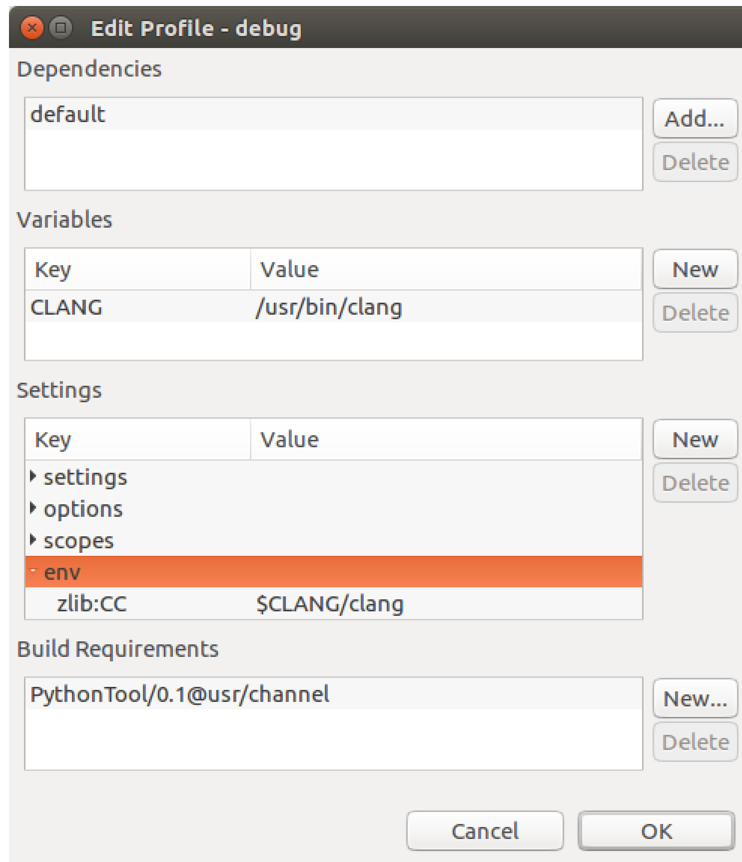


Figure 19 - Edit profile window

Here you can edit all profile settings.

When adding a profile dependency, the plug-in will check whether there exists a cycle in your dependencies and warn you accordingly. Therefore, you do not need to manually check for cycles. Please note that you cannot add a dependency if a cycle exists.

To edit the variables, settings, and build requirements you can double-click the respective field and edit it directly.

B.6 Project Specific Profile

It is possible to override the default profile set in the profile preference page, see Figure 18. This can be done in the project properties, which are accessed by right-clicking a project and then selecting "Properties" in the context menu that pops up. In the Conan property page, see Figure 20, you can

choose to either disable the project-specific settings and use your workspace default or enable them and select a profile from the dropdown below.

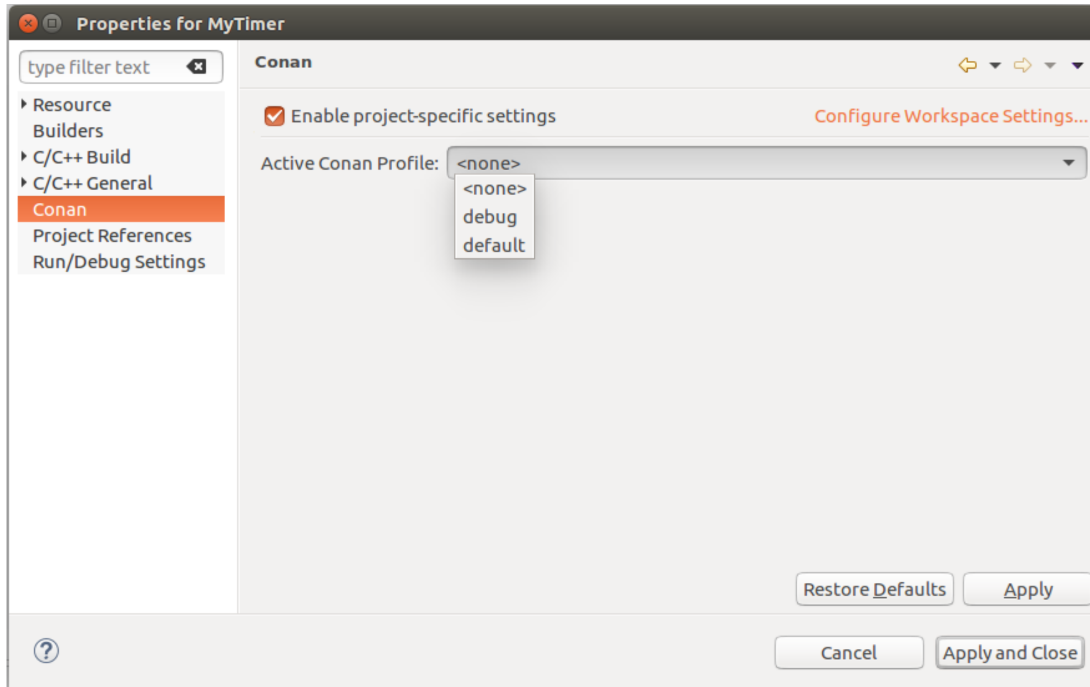


Figure 20 - Conan property page