Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

D2.2 Code Co-Development

Sandra Schröder

Work Package:	WP2
Responsible Institution:	Universität Hamburg
Date of Submission:	October 2019

Contents

Chapter 1

Introduction

Software engineering is often neglected in computational science [Kel07]. However, it can increase productivity by providing scaffolding for the collaborative programming, reducing the coding errors and increasing the manageability of software. In work package 2 (WP2), Task 2.1, suitable concepts from the software engineering perspective have been collected. A subset of these concepts have been chosen for evaluation that are considered suitable and useful for scientists during their programming tasks. These concepts have then been discussed with the scientists and they adopted them in their everyday work. In order to teach the concepts, they carried out a tutorial that teaches the selected concepts.

This report constitutes an experience report on the code co-development process.

The following paragraph is taken from the project proposal for describing the purpose of code co-development:
"In this service, a few joint efforts with scientists are implemented to evaluate and utilize new software concepts such as programming languages and tools but also understand the potential of novel architectures and processing systems. This is achieved by inferring knowledge from existing studies and by providing simplified performance and cost models for those alternatives. Together with scientists we establish pilot studies to support re-write of existing codes and document those results as success stories. While we conduct the co-development, we will periodically capture the fraction of work time spent in different tasks, e.g., design, programming and runtime. This will allow to understand the required programming time and combined with our cost analysis, the cost-efficiency of novel approaches can be made more visible. To allow other scientists to conduct similar studies, we will develop and publish a quality control method for conducting surveys that assess the benefit of systematic performance engineering."

The experiences and conclusions of the code co-development process are described and reflected in this report. A summary of the results, the advantages, and challenges of adopting the selected concepts and of the whole process is given. Additionally, some solutions for further improvement of the code co-development process are proposed.

Chapter 2

Code Co-Development

This chapter reports about the results of Task 2.5. First, the code co-development process how it was performed is described. Second, the experiences made during the code co-development process are described including a list of expected advantages and challenges of adopting the selected concepts.

2.1 Describing Success Stories

In the following chapter, we describe a success story about introducing software engineering concepts to scientists who are asked to use them throughout their development tasks. The success story is described according to this structure. As new success stories are collected, this structure can be reused. The structure contains the following elements:

Software Engineering Concepts This part describes the software engineering concepts that have been selected for the code co-development. The detailed list of software engineering concepts is given in D2.1.

Procedure This part outlines the concrete steps and used methods (e.g., tutorials, workshops, inspections) that have been performed for code co-development and to collect the results.

Results This part lists and explains the results obtained through code co-development. This section should also especially emphasize advantages and disadvantages of the applied concepts. This includes a description of challenges that appeared during the code co-development process (was it difficult to introduce/use concepts, why was it difficult?)

Material This part contains supplementary material that have been used in the code co-development process. For example, this part can contain step-by-step tutorial descriptions, task assignments for hands on sessions, user surveys, cheat sheets etc.

Chapter 3

Success Story

In the following, we describe a success story about introducing software engineering concepts to scientists who are asked to use them throughout their development tasks. To describe the success story, we use the suggested structure from Section 2.1.

3.1 Software Engineering Concepts

The following practices have been chosen and performed by the scientists.

Integrated Development Environment Normally, scientists use text editors like *vim*[1], *emacs*[2] or *nano*[3] to write the source code. While those editors provide rich support of source code highlighting, they lack advanced features like debugging, automatic refactoring (e.g., extract method/function, rename variable), code structure views, or formatting support, just to name a few. That is why the Eclipse IDE (Integrated Development Environment) has been chosen as an exemplary IDE. In Section 5.2, the tutorial for introducing the IDE is presented.

Refactoring Refactoring is defined as a *"technique for restructuring an existing body of code, altering its internal structure without changing its external behavior."*[4]. It supports to remove so-called bad smells [Fow99] that indicate bad code design. As an exemplary refactoring, the extract function is applied to reduce the complexity and readability of functions.

Coding Style A consistent coding style enhances the understandability of the source code. The participants were asked to conform to indentation of code statements and to conform to naming conventions.

Documentation Documentation is an appropriate means to preserve different kind of knowledge about a software system [Som01]. The participants were asked to add comments in source code that explain the functionality of functions. This technique aims for making the code clearer to the reader who is not the original author of the code.

Debugging Debugging is a very helpful and supportive method to find and resolve defects within a program. In Section 5.4, the corresponding tutorial for explaining the

[1] https://wiki.ubuntuusers.de/VIM/
[2] https://www.gnu.org/software/emacs/
[3] https://wiki.ubuntuusers.de/Nano/
[4] https://refactoring.com/

main concepts of debugging is presented. In this tutorial, a debugger for Python programs is used as an example, since the scientists participating in the tutorial write their programs in Python.

Unit Testing Testing is a process for software quality assurance [NT11]. In this tutorial, unit testing [HT03] [Osh09] was chosen as a specific technique. Unit testing is used to ensure that a part of an application - the unit - meets its requirements, i.e., that it behaves as intended.

3.2 Procedure

3.2.1 Tutorial Setup and Execution

Introduction of practices The selected concepts are discussed with and presented to the participants. For each selected concept, a tutorial has been designed that has been used by the participants to learn the most important principles. The tutorial contains an explanation of important terms and a step-by-step guide. The participants were asked to follow each step of the tutorial.
User Feedback Survey The participants were asked to fill out a survey after each tutorial part. This survey captures feedback on the perceived usefulness on applying the selected practices referred in the respective tutorial part. Based on the feedback, we collect experiences on the code co-development process. The complete survey is given in Section 5.5.

3.3 Results

This section reports on the experiences obtained by performing code co-development. Experiences are collected based on the feedback from the user survey and by interviewing the participants. First, the perceived advantages of the concepts are presented. Second, the challenges as perceived by the participants when applying the concepts and practices are discussed.

3.3.1 Perceived Advantages

Coding Style Indentation and naming conventions helped to enhance and keep the understandability of the code. The participants used descriptive names for variables. The names helped them to better understand the purpose of a specific variable. From their point of view, introducing naming convention does not require high effort, but provides a great benefit during development.

Refactoring Participant found that dividing the code into functions of shorter length keeps the structure of the code and improves understandability. However, as will be illustrated in the next section, refactoring sometimes could not be performed.

Documentation Documentation in form of code comments improved understandability of code. However, as reported in the next section, documentation requires a lot of effort.

3.3.2 Challenges in Code Co-Development

During the code co-development process, we found that is challenging to introduce software development methods into the development process of scientists in the context of high performance computing. Mainly, this is due to the focus of scientists on producing and publishing new research results. The effort in applying software development methods is too high from their point of view. Interviewed scientists fear that following SE practices might slow down the entire research process. Scientists mostly write code without having sustainability in mind and without considering that the code will be shared with other scientists. That is why most scientists do not see any value for most of the proposed practices.

Especially code **documentation** was considered time consuming. Interrupting the coding process in order to add code comments was perceived as *"disruptive"*. Participants mentioned that an appropriate balance between the amount of code comments and the clarity of the code is important.

Automatic **refactoring** such as *extract function* was not applicable for all code parts. For instance, one software system is written as a mixture of C and a Domain-Specific Language (DSL). For the C programming language, a rich support for refactoring is provided by the IDE. However, refactoring for DSL code is not provided. That is why, refactoring the DSL code is a manual process. Manual refactoring can become time consuming and is error prone.

Chapter 4

Summary and Conclusions

In this deliverable, we have summarized an experience report on applying the code co-development process. We have selected a subset of software engineering concepts and asked scientists to apply them in their everyday work. We designed a tutorial to teach them the most important principles of the software engineering practices. In order to collect the experiences of the code co-development process, the participant needed to fill out a survey. Additionally, we interviewed the participants.

In this report, several challenges of introducing software engineering concepts to the HPC field are described. Those challenges are due to several reasons. Often, scientific software is written by PhD students that use their programs in order to produce results for their research. The software is only developed for the scope of their thesis. The software is not intended to be developed and used by others after the thesis is finished. However, it is very likely that future students make use of the software for answering new research questions based on existing results obtained with this software. It seems that this is a situation scientists are not aware of. At this point, software engineering methodologies and techniques can be prove to be beneficial in order to increase the productivity of scientific software development processes.

Despite the challenges reported in the previous section, we think that scientists in the HPC field can greatly benefit from such methods. It is necessary to convince them that software engineering practices can greatly support them in their programming tasks. For this, it is necessary to collect and provide more success stories of scientists that have applied the selected concepts.

Chapter 5

Material

5.1 Tutorial Introduction

5.1.1 How to use this guide?

Each part starts with an explanation/description of terms used in order to understand the following instructions. Please read them carefully and make yourself familiar with the terms. Please capture the time when you start the respective part and capture the time when you finished the part. At the end of each part, please provide some impressions about the benefits and challenges that you experienced while you performed the tasks. Please provide the information in the survey `User-Feedback.pdf` (either handwritten or in the corresponding tex file).
This guide contains three parts:

Part 1 Foundations of Python IDE, introducing Eclipse and Python
(file: `1tutorial-eclipse.pdf`)

Part 2 Debugging Python programs with PyDev
(file `2tutorial-unit-testing.pdf`)

Part 3 Test Automation and Unit Tests in Python with Eclipse
(file `3tutorial-debugging-eclipse.pdf`)

Please work through the parts in order.

Actions that need to be performed by the reader are marked with **ACTION**.

5.2 Part 1: Eclipse

5.2.1 Terms

Workspace The Workspace is the physical location (file path) where certain meta-data and the development artifacts (projects, source files, images and other artifacts) are stored. The meta-data stored for the workspace contains preference settings, plug-in specific meta data, logs etc.

Perspectives, views, and editors Views are used to navigate and change content. Views and editors are grouped into perspectives. For example, the python perspective contains python specific views and editors. A view is typically used to work on a set of data. This data might be a hierarchical structure. If data is changed via the

view, the underlying data is directly changed, without the need to save. For example, Project Explorer view allows you to browse and modify files of Eclipse projects. Any change in the Project Explorer is directly applied to the files, e.g., if you rename a file, the file system is directly changed. Editors are typically used to modify a single data element, e.g., the content of a file or a data object. For example, the python editor is used to modify python files. An editor with files modified by the user (a dirty editor) is marked with an asterisk left to the name of the modified file.

Outline The Outline view shows the structure of the currently selected source file.

Project/package explorer The Package Explorer view allows you to browse the structure of the projects and to open files in an editor via a double-click on the file. It is also used to change the structure of the project. For example, files and folders can be renamed or moved via drag and drop. A right-click on a file or folder shows the available options.

Problems view The Problems view shows errors and warning messages. Sooner or later you will run into problems with your code or your project setup. To view the problems in your project, you can use the Problems view which is part of the standard Java perspective. If this view is closed, you can open it via Window>Show View>Problems.

5.2.2 Setting up the Python Eclipse Plugin PyDev

Installation of the software is OS dependent, follow the instructions in the appropriate subsection. After you have installed the software, make yourself familiar with the user interface of Eclipse, e.g., investigating the package explorer, the views of the python perspective etc.

5.2.3 Windows

Please follow the steps below in order to setup Eclipse and the PyDev plugin.

1. Download and install latest eclipse (Eclipse Photon)
2. Eclipse requires Java to be executed. If not already installed on your system, please do so.
3. Start Eclipse, choose a workspace. If there does not exist a workspace on that path, it will be created. When chosen, Eclipse starts and shows a welcome page.
4. Install Python Plugin: Help > Eclipse Marketplace; Find: PyDev > Install
5. Restart Eclipse.
6. Finished.

5.2.4 Linux (Debian and derivatives)

Use `apt` to install the required packages:

```
$ sudo apt install eclipse eclipse-pydev
```

5.2.5 Create a Python Project

In this part, you will create your first python project. Please perform the following steps:

1. Open the PyDev Perspective: Window > Perspective > Other Perspective > PyDev
2. File > New > Project...
3. New window: PyDev > PyDev Project
4. Click Next.
5. Choose a project name, e.g., “Test”.
6. Choose a python version that is installed on your computer, e.g., 3.6 or 2.7.
7. Choose the interpreter (same as on your computer or other).
8. Click Finish.

5.2.6 Create a Python Module

In this part, a python module is created. This will be a simple python script that will print “Hello” to the Eclipse console. For this, do the following:

1. First create a source folder.
2. For this, right click on the python project that you have just created. Then choose New > Source Folder; The name of the source folder should be “src”.
3. Create a python module. For this, Right click on folder “src” and choose New > PyDev Module.
4. Choose a Name for the Python module, e.g., “TestModule”.
5. Accept eclipse default settings for PyDev.
6. Choose a template for the module: Module:Main.
7. This creates a python main module stub that can be executed. There are also other templates available, e.g., Python Class template. You should see this source code:

```
1 if __name__ == '__main__':
2     pass
```

8. In the line below `if __name__ == '__main__':`, replace the `pass` command with `print('Hello')`, resulting in:

```
1 if __name__ == '__main__':
2     print("Hello")
```

5.2.7 Run a Python Module

1. Right click on the created python module in the package explorer.
2. Run as > Python Run
3. Console in Eclipse should show the text `Hello`.

5.2.8 Import an existing project

For this tutorial, an existing project is provided that implements the Quicksort algorithm. The next parts of the tutorial (Part 2 and Part 3) are based on this example. That is why the project needs to be imported into the workspace. The project is delivered as a zip archive. In order to import the archive into the workspace, perform the following steps:

1. Right click in the package explorer, choose import.
2. Choose General > Existing Projects into Workspace; Next.
3. Check "Select archive file".
4. Browse to the archive file containing the project.
5. Choose the path where you have stored the archive.
6. Click the "Finish" button.
7. The project is now imported into the workspace.

5.3 Part 2: Unit Testing

5.3.1 Terms

Unit Test: A unit test is an automated piece of code that invokes a unit of work in the system and then checks a single assumption about the behavior of that unit of work.

Test framework: A test framework provides specific methods in order to verify assumptions about the behavior of the unit (assert methods). The framework *unittest* for Python will be used in this tutorial.

Test Case: A test case is the individual unit of testing. It checks for a specific response to a particular set of inputs.

Test Fixture: A test fixture represents the preparation needed to perform one or more tests, and any associate cleanup actions. This may involve, for example, creating temporary or proxy databases, directories, or starting a server process.

Test Suite: A test suite is a collection of test cases, test suites, or both. It is used to aggregate tests that should be executed together.

Test Runner: A test runner is a component which orchestrates the execution of tests and provides the outcome to the user. The runner may use a graphical interface, a textual interface, or return a special value to indicate the results of executing the tests.

5.3.2 Assertions

Each test needs to call an assert method in order to verify the assumption about the behavior of a unit. Python's testing framework uses Python's built-in assert() function which tests a particular condition. If the assertion fails, an AssertionError will be raised. The testing framework will then identify the test as Failure. Other exceptions are treated as Errors. The following assertions are basic assertions:

assertEqual/assertNotEqual(arg1,arg2): Test that arg1 and arg2 are equal/not equal. If the values do not compare equal/do compare equal, the test will fail.

assertTrue/assertFalse(expr): Test that expr is true/false. If false/true, the test fails.

assertIs/assertIsNot(arg1,arg2): Test that arg1 and arg2 evaluate to the same object/do not evaluate to the same object.

assertIsNone/assertIsNotNone(expr): Test that expr is/is not None.

assertTrue, assertFalse: verify a condition

assertRaises: verify whether a specific exception has raised

5.3.3 Writing a Test for the Quicksort Algorithm

In the following, the Quicksort Algorithm from the exemplary project will be tested. For this, create a unit test class:

1. Create a new python module “quicksort_test.py”
2. Choose the Unittest Template for the module

You should see the following code:

```
import unittest

class Test(unittest.TestCase):

    def testName(self):
        pass

if __name__ == "__main__":
    unittest.main()
```

A testcase is created by subclassing `unittest.TestCase`. Initially, one test is defined in this template, namely `testName`. The name of any tested function starts with the letters “test”. This naming convention informs the test runner about which methods represent tests. The final block shows a simple way to run the tests. `unittest.main()` provides a command-line interface to the test script. Execute the test by right clicking on the editor of the test class and choose Run As > Python unit-test.
You should see the following output:

```
Finding files... done.
Importing test modules ... done.

----------------------------------------
Ran 1 test in 0.001s

OK
```

Since the test method is not implemented yet, it will run successfully.

In a next step, the test class will be completed with two test cases. It should be tested that

1. The length of the resulting, sorted list is the same after sorting even when the list contains duplicated entries, e.g., [1,0,0,2] should result in [0,0,1,2] and not in [0,1,2].

2. The list is sorted after calling the quicksort method.

Since `unittest` prints only the name of the successful/failed test method, it is important to choose a descriptive name that allows readers to quickly grasp what the method is supposed to test.

ACTION: Please add two test method with a meaningful name that best represent the tests mentioned above.

As a next step, the test methods will be implemented. In the test, an unsorted, arbitrary list with integers is taken as input. To test the Quicksort algorithm, the expected output needs to be compared with the actual output.

ACTION: Implement the test methods. Use an appropriate assertion method to test the Quicksort algorithm.
The tests should fail, since there is a (rather simple) bug in the implementation of the algorithm. The bug will be discovered in the next part using debugging.

5.3.4 Survey

Please answer the questions provided in the tutorial's introduction.

5.4 Part 3: Debugging

Debugging is used to find and resolve defects within a program. In this part, we will use the Eclipse debugger in order to locate the bug currently implemented in the example project.

5.4.1 Terms

Breakpoint A breakpoint in the source code specifies where the execution of the program should stop during debugging. Once the program is stopped you can investigate variables, change their content, etc.

Debug Perspective Eclipse provides a Debug perspective which gives you a pre-configured set of views. Eclipse allows you to control the execution flow via debug commands.

Debug View The Debug View allows you to manage the debugging or running of a program in the workbench. It displays the stack frame for the suspended threads for each target you are debugging.

Breakpoints View The Breakpoints view allows you to delete and deactivate breakpoints and watchpoints (not covered in this tutorial).

Variables/Expression View The Variables view displays fields and local variables from the current executing stack. Please note that you need to run the debugger to see any variables in this view.

Stepping Stepping is the core feature of any debugger which allows you to execute (step through) your code line by line. This allows you to examine each line of code in isolation to determine whether it is behaving as intended. The eclipse debugger provides the following stepping modes:

- F5: Go to the next step in the program. If the next step is a method / function, this command will jump into the associated code.
- F6: Step over function calls, e.g., it will call a method / function without entering the associated code.
- F7: Step out of the current function. So this will leave the current code and go to the calling code.
- F8: Resume execution. If no further breakpoint is encountered, the program will continue its work, and possibly exit normally.

5.4.2 Debugging the Application

The tests in the previous tutorial part revealed that the Quicksort algorithm contains a bug. In this part of the tutorial, the bug will be discovered using the debugging features of Eclipse.

In order to find the location of the bug, a breakpoint needs to be set. To create a breakpoint at a specific line, double click in the left margin in the Python editor.

1. Create a breakpoint at `if len(numbers) == 0:` (the first line of the quicksort implementation).
2. Then right click on the file "quicksort_test.py" in the package explorer (or right click in the editor) and choose Debug As > Python Run.
3. Eclipse will ask to change to the Debug perspective. Confirm that you want to change the perspective (click "Switch"); You can make eclipse remember your decision.

Eclipse will run the code until it reaches the breakpoint. Eclipse shows a blue arrow in the line where the breakpoint has been reached. The line has not been executed at this point. After stopping at the breakpoint, we can investigate the current program state using the Eclipse debugger, e.g., current variables in the variable view.

ACTION: Continue investigating the program by stepping through it using F6 (Step Over).

In the variables view you see how variables (e.g., `first`, `numbers`, and `remaining`) are added and filled with values as you proceed with F6.
In the debug view, the stack frame is depicted showing how the quicksort function is called recursively as you step through the program.
Try to find the location that lets the test fail that verifies that the length of the list stays the same by investigating the code step by step and using the variable view as support.

Fix the code and re-run the test to verify whether they pass successfully.

5.5 User Survey

5.5.1 Part 1

Start _____

End _____

Do you think you could apply the IDE in your project? Please indicate why (or why not) this is the case.

Will you start using an IDE in your project / or do you think that this might become interesting in a future project?

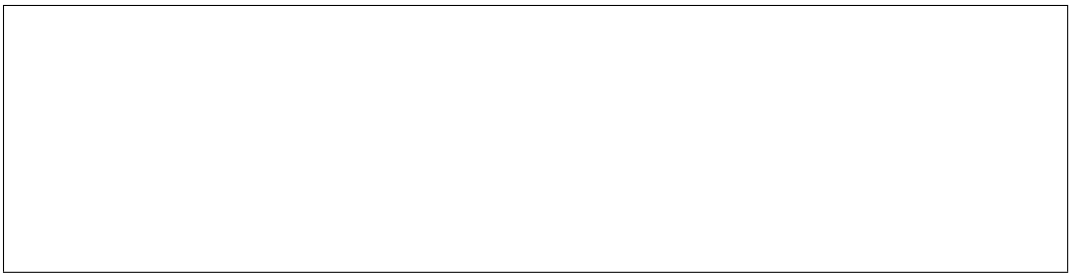
5.5.2 Part 2

Start _____

End _____

Do you think you could apply test-driven development in your project? Please indicate why (or why not) this is the case.

Will you start applying test-driven development in your project / or do you think that this might become interesting in a future project?



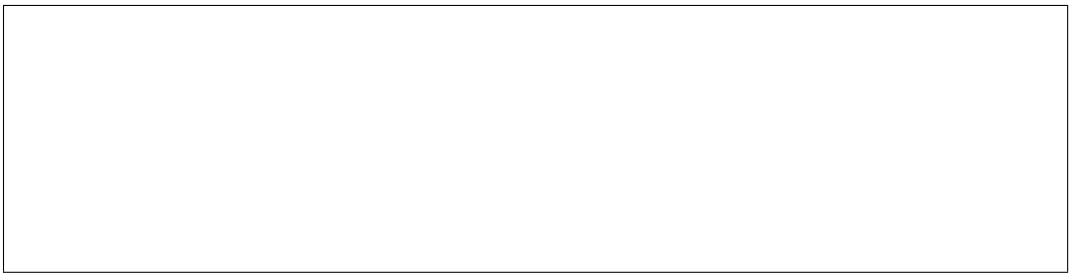
5.5.3 Part 3

Start _____

End _____

Do you think you could apply a debugger, e.g., the Eclipse debugger, in your project? Please indicate why (or why not) this is the case.

Will you start applying a debugger in your project / or do you think that this might become interesting in a future project?



Acknowledgement

The PeCoH project has received funding from the German Research Foundation (DFG) under grants LU 1353/12-1, OL 241/2-1, and RI 1068/7-1.

Bibliography

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[HT03] Andy Hunt and Dave Thomas. *Pragmatic unit testing in Java with JUnit*. The Pragmatic Bookshelf, 2003.

[Kel07] D. F. Kelly. A software chasm: Software engineering and scientific computing. *IEEE Software*, 24(6):120–119, Nov 2007.

[NT11] Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[Osh09] Roy Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2009.

[Som01] Ian Sommerville. Software documentation. *Software engineering*, 2:143–154, 2001.