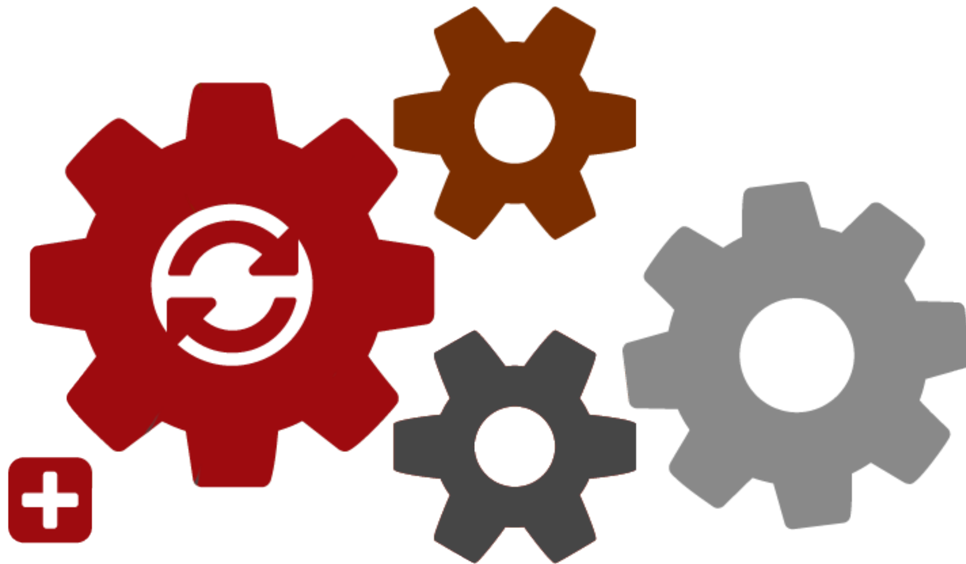
CHALMERS
UNIVERSITY OF TECHNOLOGY



Software maintenance for Discrete-Event Simulation Models

Developments of snippets and a workflow using Git to support maintenance of discrete-event simulation projects

ANDREAS LU
YULI HUA

Department of Product and Production Development
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2018

Master's thesis 2018

Software maintenance for Discrete-Event Simulation Models

Developments of snippets and a workflow using Git to support maintenance of discrete-event simulation project

ANDREAS LU
YULI HUA

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Product and Production Development
Division of Production Systems
Chalmers University of Technology
Gothenburg, Sweden 2018

Software maintenance for Discrete-Event Simulation Models
Developments of snippets and a workflow using Git to support maintenance of discrete-event simulation project
Andreas Lu
Yuli Hua

© ANDREAS LU & YULI HUA, 2018.

Supervisor:
Camilla lundgren, Department of Product and Production Development
Leo Adelbäck, ÅF Industry

Examiner: Anders Skoogh, Department of Product and Production Development

Master's Thesis 2018
Department of Product and Production Development
Division of Production Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2018

Software maintenance for Discrete-Event Simulation Models
Developments of snippets and a workflow using Git to support maintenance of discrete-event simulation project
Andreas Lu
Yuli Hua
Department of Product and Production Development
Chalmers University of Technology

Abstract

In an ever-growing world with various manufactures and industries increasingly striving to expand and becoming more high-tech and advance, it's of great importance to have intelligent and inexpensive tools for decision-making. A powerful tool such as a discrete event simulation (DES) offer the ability to perform analyses and experiments virtually, avoiding potential damages and cost to the real physical world. The simulation team at ÅF are looking for ways to improve their way of dealing with DES projects; mainly developed in AutoMod. With the increasing demand of discrete event simulations as well as with businesses continuously changing, it's becoming a great challenging to maintain a model, from a long-term perspective. Problems such as making it time-efficient and easy to resume previous models in case of a future project inheritance or enabling concurrent collaboration among other team members are essentials topics that are now receiving more importance. The thesis has combined previous experience in developing in AutoMod with a case study, literature studies and interviewing session with the simulation team at ÅF. The case study involves an old simulation model developed in AutoMod with a history of multiple authors, making it hard to break down and update. The results are a variety of tools designed to be applicable to both previous models as well as new incoming projects. Key tools developed are the wide range of snippets, customized workflow using Git, and the incorporated well-defined documentation. Using a version control system, Git, has proved useful when trying to manage implementation of new model-features, review changelog, keeping track of bugs and the overall documentation. Snippets have showed tremendous advantages when trying to eliminate the software related challenges such as inconsistency in writing and reviewing source code.

Keywords: AutoMod, software maintenance, snippets, Git, discrete event simulation, VS Code.

Acknowledgements

We would like to thank our examiner Anders Skoogh and our supervisor Camilla Lundgren from Chalmers University of Technology. The door to Camilla Lundgren's office was always open whenever we needed help with our research or writing. She consistently allowed this paper to be our own work, but steered us in the right direction whenever she thought we needed it. We would also like to thank our supervisor Leo Adelbäck at ÅF Industry who continuously helped us understand both intricate details of the given simulation model and AutoMod related issues. We are also thanking the rest of the simulation team at ÅF Industry providing invaluable feedback and encouragement during the interview process.

Big thanks to the engineers at Arla; Ola Allvin, Peder Jonsson, Shabib Khattak, Ulf Tollerud and Jonas Graneràs, for their great help in explaining the different concept of their system at their factory. Without their passionate participation and input, the process of fully understanding the simulation model could not have been made possible. Finally, we must express our gratitude to Pär Ström from ÅF Industry for providing us with this thesis proposal.

This accomplishment would not have been possible without the combined help of the above-mentioned peers. A profound thank you.

Gothenburg, June 2018

Andreas Lu and Yuli Hua

Contents

List of Figures **viii**

1 Introduction **2**
1.1 Purpose 4
1.2 Project aim 4
1.3 Delimiters 4
1.4 Research questions 5
1.5 Brief explanation of our "case" 5
1.6 Report Outline 5

2 Theory background **7**
2.1 Discrete event simulation 7
2.1.1 Components of DES 7
2.1.2 AutoMod 8
2.2 Software maintenance 8
2.2.1 Software maintenance importance 9
2.2.2 Software maintenance issues 9
2.2.3 Differences among operations, development and maintenance . 10
2.2.4 Software developer doing software maintenance 10
2.3 Snippets 11
2.3.1 Using snippets to learn coding languages. 11
2.3.2 Snippets in order to improve compilation time. 12
2.3.3 What to think about writing snippets 13
2.3.4 Snippet management 13
2.3.5 Snippets in runtime 13
2.3.6 Summary snippets 13
2.4 Version control 14
2.4.1 Git 14
2.4.2 Basic concepts of Git 15
2.4.3 Commits 15
2.4.4 The past 16
2.4.5 The present 16
2.4.6 Git history and commit log 17
2.4.7 Git commit messages 18
2.4.7.1 Be useful 18
2.4.7.2 Be detailed (enough) 18

2.4.8 Be consistent . . . 19
2.4.9 Active voice . . . 19
2.4.10 Guidelines to writing git commits summarized . . . 19
2.4.11 Branches . . . 20
2.4.12 Merge branches . . . 21
2.4.13 Work-flows . . . 21
2.4.14 Centralized work-flows . . . 21
2.4.15 Feature branch work-flow . . . 22
2.4.16 GitFlow . . . 22
2.4.17 Summary version control . . . 23

3 Methodology - Work process **25**
3.1 Identify problems . . . 26
3.1.1 Code-related issues . . . 26
3.1.2 Documentation-related issues . . . 27
3.2 Literature review . . . 27
3.3 Creation of AutoMod Extension . . . 28
3.3.1 AutoMod Editor limitations . . . 28
3.4 First generation of snippets . . . 29
3.4.1 Snippet . . . 29
3.5 First generation of the workflow using git . . . 30
3.6 Interview ÅF . . . 31
3.7 Final update to snippets and workflow using Git . . . 32

4 Result **33**
4.1 AutoMod language support extension . . . 33
4.1.0.1 Syntax highlighting and auto-completion . . . 33
4.2 Snippets . . . 34
4.2.1 Snippets for different process types . . . 34
4.3 Workflow using Git . . . 36
4.3.1 Anatomy of workflow . . . 37
4.3.1.1 Bug branch . . . 37
4.3.1.2 Quick-fix branch . . . 37
4.3.1.3 Features branch . . . 37
4.3.2 Internal rules . . . 38
4.3.2.1 Committing . . . 38
4.4 Input from the Interview and final update . . . 38
4.4.1 Regarding the Snippets . . . 38
4.4.2 Documentation to declaring different processes . . . 39
4.4.3 Regarding the Git workflow . . . 40
4.4.4 Feature and Bug tracker . . . 41

5 General Discussion **43**
5.1 Results . . . 43
5.2 Interview feedback regarding snippet management . . . 44
5.3 Interview feedback regarding version control systems . . . 44
5.4 Why a dedicated software maintenance team is not needed at ÅF . . 47

5.5 Sustainability aspect 48

6 Conclusion 49

7 Future work 51

Bibliography 52

A The interview guide I

B Figures III

List of Figures

2.1 Two examples of low-level programming languages and functions. Both calculates the n:th Fibonacci number. On the left, code is written in hexadecimal representation of 32-bit x86 machine code, whereas on the right code is written in x86 assembly language using MASM. [1] . . . 12
2.2 The following screenshot explains the file status life cycle [2, p. 14] . 15
2.3 The following diagram explains how commits can be viewed as past and present [2, p. 18] . . . 16
2.4 The following diagram depict a modified working directory and an empty index [2, p. 19] . . . 16
2.5 The following diagram depict the result when the modified files from working directory has been added to the index [2, p. 19] . . . 17
2.6 The following diagram explains how commits can be viewed as past and present [2, p. 20] . . . 17
2.7 Two examples of git commit logs, 1. has no consistency while 2. uses conventions which provide more consistency thus improved readability 19
2.8 A common example of how a new and unexperienced user of git will commit messages. 1. describes the actual commiting, whereas 2. is how the commit should have been done . . . 20
2.9 An abstraction of how branching can be viewed, "C1-C4" represents commits and "MASTER" and "NEW WORK" are branch names. [2, p. 31-32] . . . 21
2.10 Centralized work flow example [2, p. 104] . . . 22
2.11 Example of feature branch based work flow [2, p. 105] . . . 22
2.12 Example of GitFlow [2, p. 106] . . . 23

3.1 Flowchart of the work process . . . 26
3.2 An example of intellisense, suggestion is automatically inserted upon selection (auto-completion) . . . 29

4.1 An example of a snippet designed for break down procedures in AutoMod. Procedures are automatically inserted upon selection . . . 34
4.2 An example of a snippet designed for while loop routines in AutoMod. Procedures are automatically inserted upon selection . . . 34
4.3 The top part displays the intellisense prompt of the three implemented processes type snippet. Below is the auto-completed version function, procedure and subroutine snippet content respectively . . . 36

4.4 The final version of process snippets with improved introductory documentation. . . . 40
4.5 An example of a feature tracker sheet in excel. By adding a new feature through the button "Add new feature", a new row containing an incremented feature number, current date, status and other placeholder values are auto-filled. The status colors green, yellow and red represent done, in progress or not started respectively . . . 41
4.6 An example of a bug tracker sheet in excel. By adding a new bug through the button "Add new bug", new row containing an incremented bug number, current date, status and other placeholder values are auto-filled. The status colors green, yellow and red represent done, in progress or not started respectively . . . 42

5.1 An example visualizing a git work flow using no branch features and no commits token for different types of commit. . . . 45
5.2 An example visualizing a git work flow using branch features and has commits token for different types of commit. . . . 46
5.3 Filtering options of branches . . . 46
5.4 Filter out branch f2-storage and shows the commit history included in f2-storage . . . 47

6.1 Flowchart of the work process showing where the main identified issues could be solved . . . 50

B.1 Example of final process specific snippet, function F_CalcDay. Every subject field is filled except for Private variables, which is left blank . III
B.2 Example of final process specific snippet, function F_CalcTime. Every subject field filled . . . IV

1 Introduction

In an ever-growing world with various manufactures and industries continuously expanding and becoming more high-tech and complex, it's highly demanded to have an intelligent and inexpensive tool for decision-making; a discrete event simulation (DES) could be such a tool. Some of the prominent usages of a DES is the ability to perform extensive experiments and analyses in a non-physical environment.

To give an example, one can imagine a warehouse consisting of several aisles of stocks, where truck drivers move in-between to pick products into their roll cages or pallets. There are also loading sites where the drivers go to when they are done order picking. Here they unload their filled roll cages or pallets, which are subsequently moved onto trailers for upcoming customer shipment. With a DES model of this particular warehouse management, one could consider an interesting analysis to be a scenario where one is constraining the amount of the concurrent and active trucks. With this restriction, one could study the overall efficiency and throughput, the amount of less or additional breakdowns or how much time would now be needed to finish all customer shipments etc. Another analysis could be, to swap the position of a specific article from one stock to another. By doing so, the drive path to collect the article would alter; thereby affecting the overall traffic, in a good or bad way. These are only two examples of analysis which can be experimented thanks to DESs.

By making prompt investments or working in a non-proactive manner, the result can be highly expensive and eventually lead to trailing consequences. Using a powerful tool such as a DES to setup different scenarios beforehand to both analyze and confirm certain investments, one can potentially increase the productivity, reduce the cost, improve the flow in the shop-level or reduce storage capacity demands etc. These are highly desired and beneficial outcomes.

However, dealing with DES projects are not always an obvious task. In fact, DES projects can involve many several intricate steps throughout its life cycle. Depending on the project at hand, varying sub task can be set up differently due to the handful methodologies that are applicable. Furthermore, bigger DES projects can be regarded as a software related project, meaning there are additional software related challenges one has to consider when designing a simulation model. Some of the software challenges could be ensuring consistency in the source code and developing an effective workflow. Disregarding these challenges could lead into problems when handing over or resuming the project in the future.

A great challenge by companies is thus to maintain DES projects in a sustainable aspect as when project stretches over a longer period of time, it can be hard to get in contact with the former responsible(s). For instance, a poorly documented project would mean that one would have to spend valuable time reiterating the same time-consuming process of understanding all the details of a project. Additionally, if there are no guidelines as to how projects ought to be carried out in a maintainable regard, it's to no one's surprise that the project will sooner or later self-degenerate due to lack of consistency and structure. Imagine the consequences of building a construction using different materials for every sub parts. No consistency will eventually cause failure. In general, good documentation encourages improvements and updates to a DES project. From a simulation standpoint this is important, since building a good simulation model implies that one allow for both minor and major updates. In the end, having guidelines for the initial setup procedure, how projects are to be carried out and continuously documented, are key processes that ensure more sustainable and manageable DES projects.

Banks methodology is a well-known methodology describing and providing guidance for modeling a DES project [3]. It covers the various steps of how a simulation study should be carried out. When it comes to documentation, Banks too, highlights the importance of it. He argues that with adequate documentation, the future usages and future developments of a model is vastly facilitated. Moreover, by having an extensive overview of the project history, in terms of backlogs and changelogs, it becomes easier for developers to determine whether a project is progressing towards a set goal or not. However, although the encouragement of rigid documentation by Banks, his methodology lack the information on how a model ought to be maintained for continuous improvement and integration. Guidelines and knowledge concerning how to maintain a model for these causes are essential to develop a relevant, yet long-lived, model.

With a better maintained model it's easier to conduct project handovers. And one of the more important aspect of project handovers includes the knowledge transferring part. From the article of "knowledge transfer" the author argues that the most effective way of transferring knowledge is through in-person communications [4]. This is not always possible though. In fact, when this is not possible, it's mentioned that adequate documentation can many be the only or best compensation for this.

Another way of keeping up with the challenge of knowledge transferring is to reduce the complexity of the knowledge that needs to be transferred in the first place. This is further discussed in the article of "successful knowledge transfers" [5]. The main idea is to have standards for how projects ought to be setup, how similar problems should be solved, and other internal rules for how things ought to be carried out in general. This provides a more consistent workflow with predefined set of rules that reoccur in every project. This makes the process of knowledge transfers, as well as project handovers, simpler, more lightweight and more digestible, as seen from a long-term perspective.

ÅF, a consultancy company in Gothenburg and the main collaboration for this thesis, wants to extend their skills in how to properly document simulation models for future operation and facilitate the accessibility and understanding for future developers. The main idea is that the research will improve the maintenance of simulation models and in a long run save valuable time and reduce unnecessary cost spent on code reviewing.

1.1 Purpose

Utilizing discrete event simulations as a diverse and cost-efficient decision-making tool is becoming a reality for various businesses in the manufacturing industry. As a consequence, there is an increasing demand and interest of performing new discrete event simulations, as well as continuously keeping up with existing ones. Challenges as to long-term maintain a model are emerging. It's desired to incorporate time-efficient and simple work methods to resume previous models in case of a future project inheritance, or enabling concurrent collaboration among other team members. The purpose of the thesis is to study this matter; how different tools such as snippets and workflows using Git, can help mitigate software maintenance challenges.

1.2 Project aim

The three main objectives of the project are:

1. Identify the challenges of understanding an inherited simulation model.
    - Perform interviews
    - Analyze the model from the the case study
2. Discover alternative solutions for maintaining a DES project regarded as a software related project.
    - Develop snippets.
    - Develop a workflow for git.
3. Take previous developments and propose a customized solution for ÅF.

1.3 Delimiters

Data collections/observances are solely done on ÅF, hence only issues highlighted there are considered. This project does not include development or experimentation of the model, but the simulation model from the case study will be used as an additional source to identify issues. The decision of what method will be used and what solution is the best one, will be based on how it suits ÅFs working methods and the AutoMod language.

1.4 Research questions

The following research questions will be touched upon and are key objectives of the thesis.

- What are the central topics in order to perform maintenance of a software project, and how can it be applied to DES projects
- How can software maintenance help cultivate and transfer gained knowledge?
- How can snippets be combined with smart documentation in order to aid the maintenance work for DES projects.
- How can a workflow using Git be designed to serve useful and effective for maintaining DES projects, especially for smaller groups of 2-4 persons.

1.5 Brief explanation of our "case"

ÅF has inherited a DES project from Arla Foods. The project involves a simulation model of their warehouse and distribution center. The model has been handed over to several developers from different companies throughout its lifetime. During this time it has become a challenge to keep it structured and easy for the next developer to continue with the model. Lack of documentation regarding the source code, project history and general output data, between the different iteration of the model and between developers are general reasons for the issues. To be able to start further development in an efficient way as well as to allow for easier transition for future developers, a maintenance work applied to the model is of interest. The existing and working model will serve as a perfect inspiration source for this thesis.

1.6 Report Outline

Chapter 1 - Introduction: In this chapter an introduction will be given regarding the many maintenance challenges of discrete event simulation projects. Additionally, information about the purpose, research questions, delimitation of the thesis, as well as a description of the case study and the report outline will be given.

Chapter 2 - Theory: All the appropriate theory and material necessary for understanding the concepts in the subsequent chapters are presented here. The theory will also serve as discussion material for designing and developing the different implementations. The main material of this chapter will consist of general subjects in various field that can serve as a foundation to build up a robust maintenance for DES projects.

Chapter 3 - Methodology: In this chapter, how the work ought to be carried out are presented in a chronological order. Some of the subjects that will be discussed are; The VS Code integration, the AutoMod extension and the different generations of implementations of snippets and workflow using Git. Separate implementation-specific discussion will be given in order to guide the read to understand why different

approaches was chosen.

Chapter 4 - Result: The result will include the developed AutoMod extension, created snippets, proposal of a workflow using Git, and finally inputs from the interview and a revamping of the implementations.

Chapter 5 - General Discussion: This chapter gives a more broad and general discussion of the result and about the feedback and input from the interview. As well as, general discussion of areas where the researched theory was not directly applied to the different implementations and the sustainable aspect.

Chapter 6 - Conclusion: The final conclusions to the research questions and the project as a whole are presented here.

Chapter 7 - Future work: In this chapter a list of future work is listed, both non finished and non started work.

2
Theory background

This chapter describes different theories of subjects that are included in this work. Mainly tools that can be used for maintenance of a DES project. DES projects will be regarded more as a software project, as the code base for the simulation model play a central role. The theories includes Discrete event simulation, Software maintenance, Snippets and Version control

2.1 Discrete event simulation

A discrete event simulation (DES) is a powerful tool that describe companies' dynamic systems and can be used to perform extensive experiments to evaluate future implementations or changes in their current layout. As the name suggest, the simulation is performed on discrete time, when events occur. Events, in simulation term, are changes occurring in the system like movement of a part from machine to transporter or initiating a operation process. That means the time in-between events are unimportant and not included in the model. Model is a representation of the real-time system, which also following quote concur, "models address a wide range of manufacturing system design and operational issues and are therefore essential tools in many facets of the manufacturing system design process". [6]

The usage of discrete event simulation or simulation in general with computers is due to time limit when managing large data sets. Using analytical approach is time-consuming and in some cases impossible. Simulation can be summzarized as,

1. Answers to numerous questions about the system
2. A representation of the system and its behavior
3. Better knowledge of different systems in the world.

[7]

2.1.1 Components of DES

There are several important components in a discrete event simulation model and the most common one's are the following. [8]

- **Entities** are objects that are direct representations from the real-time system. Changes or activities occurring in the system are due to entities, either by the entity is making an action, in this case the entity is a human operator, or action is made on the entity, in this case the entity is a product. Entities can

be operators, product, machines, etc.

- **Attributes** describes entities. Each entity can have their own unique attributes, but some attribute can also be the same for a group of entities. Attributes can be size, name, type, etc.

- **Events**, as mentioned previously, are changes occurring in the system and what the simulation model is composed of. Commonly there are a list of events where the time of the event is specified.

- **Queue** are used to represent buffers and storages. It is also commonly to use queue to solve some modelling problem, thus it do not has to represent anything from the real-time system.

- **List** can be used to control the flow in the system. If the machine should process the products in any particular order or in different batch-sizes, it can be setup by a list.

2.1.2 AutoMod

AutoMod is a discrete event simulation tool which is commonly used to simulate manufacture- and logistic systems. It is used by many companies in diverse field of area such as, vehicle, traffic, food, medicine, metal, mine, wood industry. AutoMod is flexible when it comes to being able to model, simulate and analyze both simple and complex system. [9]

An model in AutoMod can be setup by writing the source code of the different process. The process includes all the events which are broken down to several of code lines. The source code is then used to essential build the graphical part of the model. AutoMod's library has some base component to represent some entity, such as trucks and human operator. But component such as the layout of the facility can be created in another software and then imported to AutoMod. Furtheremore, AutoMod has build-in functions to support modelling of system such as, path mover, conveyer, bridge crane, kinematics, etc.

The simulation can then be conducted in AutoMod Runtime where the animation of the system can be viewed. Various kinds of reports are presented in Runtime, such as total throughput, utilization statistics, waiting time, etc. To simulate different experiments or scenarios in a more efficient way, there exist a add-on to AutoMod called AutoStat. Multiple simulation can be setup and run with different settings, and the outcome of those can easily be compared.

2.2 Software maintenance

There are many various definitions of software maintenance. Here are some quotes presented to name a few:

- “changes that are done to a software after its delivery to the user” [10]
- “a software product does not wear out from repeated usage, hence does not need to be ’maintained’ the way a car or a TV does. In fact, the word is used by software people to describe some noble and some not so noble activities. The noble part is modification: as the specifications of computer systems change, reflecting changes in the external world, so must the systems themselves. The less noble part is late debugging: removing errors that should never have been there in the first place.” [11]
- “The totality of the activities required in order to keep the software in operational state following its delivery” [12]
- “Maintenance covers the software life-cycle starting from its implementation until retirement” [13]

As can be read above, many sources state that the software maintenance definition involves the process of making modifications after the delivery of a product, but there are also sources [14] that state that software activities are not only performed on the post delivery stages, but could also be done during the pre-delivery stages.
In summary, software maintenance is a part of a Software Development Life Cycle. Its main purpose is to modify and update software application after or berfore delivery to correct faults and to improve performance. A software is a model of the real world. When the real world changes, the software requires alteration whenever and wherever possible [14].

2.2.1 Software maintenance importance

Due to various reasons such as a growing number of programmers and the popularity of digitization among other things, new software projects are being born and developed on a regular basis. Hence, the need for comprehensive software maintenance has never been greater. Lehman [15] states that “it’s unavoidable; changes force software applications to evolve, or else they progressively become less useful and obsolete”. For an ever changing organization with employees using their application software(s) on a daily basis, it is essential to have software maintenance as a part of the organization.

2.2.2 Software maintenance issues

While a software is in operation, a natural fallout is failures. Failures can be regarded as defects that have been unintentionally implemented into production by the developers. In many cases rigid testings and verification efforts have been carried out to mitigate failures but it’s both difficult and time-consuming, therefore costly, to develop and perform testing. Failures can also be discovered as a result of subsequent implementation, where the customer has requested new features or requirements that was not considered in the previous specification and infrastructure. Maintaining software projects is a complicated procedure and in bigger projects it’s not uncommon to have specialized software maintainers.

2.2.3 Differences among operations, development and maintenance

There is sometimes confusion and one can experience hard times in differentiating who is supposed to conduct the maintenance work. Where does it start and end? According to [16] in an organizational context, the daily maintenance and development of a software developed internally are typically the responsibility of an operational support unit within the software organization. Whereas, software acquired externally typically need maintenance done by the original developers and not the organization using the software.

2.2.4 Software developer doing software maintenance

In the early days of the software industry, there was no difference between software development and software maintenance. In the beginning, legacy source code was rather small and one could rewrite it every time there was a big change needed. Only in the 1970s the life-cycle models specific to the software maintenance process, as well as the software maintainers started to appear [16].

Nowadays software maintainers are essential for bigger software organizations. In fact, there are a number of disadvantages to letting the development team maintain the software after it has been put into production.

- Developers do not like performing maintenance and are more likely to leave for more interesting work.
- New hires in the development team will be both surprised and dissatisfied to discover that they also need to maintain existing software.
- Developers are often re-assigned to other development projects and prefer this kind of work.
- When the individuals who developed the software leave the other employees will probably not be qualified to maintain it.

Maintainers are often confronted with hundreds or even up to million lines of code. Source code mostly written by somebody else. Within a short period of time they have to quickly familiarize themselves with the code and data. To make it all more challenging, and if the software is newly developed, a number of urgent changes might be pending because the developers could not include in the initial implementation, due to either time constraint or other prioritization. From an optimal perspective the software maintenance is managed on an organizational level. It requires that a management system for software maintenance is in place, and in order for the best impact and to optimize the cost of software maintenance activities, it is required that the whole organization are committed to the continuous improvement of software maintenance.

2.3 Snippets

Writing code can many times be a rather tedious and an error prone process. Depending on the programming language and author, the code can be difficult to read and break down. One way of solving this is to use snippets. According to Liu [17] snippets are chunks of source code which can be organized for copy and paste usage, among other things. The overall goal is to ease the process of writing code and mitigate the error prone mistakes. With snippets the manual effort to type in (repetitive) source code is eliminated, and instead it is possible to re-use well defined and developed existing lines or blocks of code.

Jiang [18] states that snippets can be used in any software projects where there is a need of producing portion of code content more than once, rather than rewriting it all over. A snippet can be as short or as long as you want. They can contain just a few words, a couple lines of codes or even a whole paragraph (class). Snippets are not limited to it's content type, in fact, except for plain text they can also include images, lists or even other snippets. [18]

There are various types of snippets. Snippets are typically classified as static, dynamic or scriptable. [17] Static snippets are normally associated with the fixed chunk of source code or text that can be copied and pasted directly. It's normally inserted at the cursor position at the same pace someone is typing code. Dynamic snippets contain dynamic elements which are filled upon insertion of the snippet. The dynamic elements can be placeholder values that are computed depending on the users input or the context of the code. Finally, scriptable snippets are dynamic snippets that can invoke other snippets or internal commands. Presented below are example snippets of each individual category.

- Static snippet: static for-loop, while-loop, template for a code function.
- Dynamic snippet: user is presented with options to fill in placeholder values that yield in a final encompassed snippet.
- Scriptable snippet: math operation is inserted or casting is done dependent on the context of code.

2.3.1 Using snippets to learn coding languages.

As mentioned in [18] developers are often put against programming tasks where the implementation of the given task is an unfamiliar process. They normally face a problem where they start to search for code examples online and try to learn the usage patterns by studying the related Application Program Interfaces (APIs) from the corresponding homepages. After finding the code examples, they then proceed to copy-paste it in their code environment and start modifying it as desired. However, Jiang [18] argues that snippets can be a solution for this problem. He argues that snippets can function as high-quality code examples, specially designed to teach and demonstrate a particular purpose, whether it be a snippet whose purpose is to teach

how a typical function or method ought to be written in a particular language, or simply a snippet that explains the process of implementing a specific task following a predefined architecture.

2.3.2 Snippets in order to improve compilation time.

According to [1] a low-level programming language is a programming language that provides little or no abstraction from a computer's instruction set architecture. Meaning, commands or functions to the corresponding languages map closely to the processors instructions. Common examples of low-level programming languages are machine code or assembly languages.

```
8B542408 83FA0077 06B80000 0000C383
FA027706 B8010000 00C353BB 01000000
B9010000 008D0419 83FA0376 078BD989
C14AEBF1 5BC3
```

```
fib:
    mov edx, [esp+8]
    cmp edx, 0
    ja @f
    mov eax, 0
    ret

    @@:
    cmp edx, 2
    ja @f
    mov eax, 1
    ret

    @@:
    push ebx
    mov ebx, 1
    mov ecx, 1

    @@:
        lea eax, [ebx+ecx]
        cmp edx, 3
        jbe @f
        mov ebx, ecx
        mov ecx, eax
        dec edx
    jmp @b

    @@:
    pop ebx
    ret
```

Figure 2.1: Two examples of low-level programming languages and functions. Both calculates the n:th Fibonacci number. On the left, code is written in hexadecimal representation of 32-bit x86 machine code, whereas on the right code is written in x86 assembly language using MASM. [1]

In contrast to low-level programming, in computer science high-level programming language has a strong level of abstractions. For a human it's easier to manage and the language is overall more understandable, relative to a low-level language. But due to possible use of natural language elements and the diversity in the language, the code readability can vary depending on the code author.

According to [1] it is possible to utilize and produce snippets than can be used to provide prepared and specialized designed blocks of source code, that are carefully

designed in order to effectively improve memory allocation or other compiler specific advantages. In some programming language it can be difficult to express semantics of high level programming languages in a concise manner. But, by using specially designed snippets these problems can be alleviated, and the end result can minimize the overall the compilation time overhead. On the contrary, writing semantics in a low-level programming language can be difficult due to the poor abstractions. Having snippets that will provide you with the appropriate code upon request will evidently save you a lot of time. Ultimately, snippets can also express low-level semantics of high-level operations, and vice versa. This can serve useful if the snippets are written in the same level as the compiler is compiling.

2.3.3 What to think about writing snippets

The snippet should be developed in a generic manner [18]. When using a snippet on multiple locations, a change in the snippet context will result in a global change to all places where the snippet is used. This prevent you from making same changes on multiple locations.

2.3.4 Snippet management

The term snippet originates from the domain of text editors [17]. Today, most text editors allow for viewing, editing, categorizing, and storing snippets in a repository of re-usable source code fragments. Depending on the text editor or Integrated Development Environment (IDE) the management and the diversity of functionally of snippets can vary.

2.3.5 Snippets in runtime

Recent research shows a number of snippet implementation programs that generate evaluates snippets at run time. For instance, in [19] Galen talks about their program that, at runtime can quickly evaluate and generate an appropriate list of suggested snippets for the user. From the same source it is argued that these snippets tools improves the programmers productivity by more than a factor of two. The user can save immense time by avoiding searching and studying APIs or the web, instead the tool does it using its own algorithms. It can pre-set the snippets with variable values, which in turn help the user to decide which snippet is most appropriate for the situation. The tool is also interactive which makes the result more accurate, as the user can connect more constraints and information, to refine the candidate code fragments. [19]

2.3.6 Summary snippets

Snippets will be a valuable and one of the core tool for this thesis. The concept of snippets will support and guide the software maintenance goal towards the right direction. The initial designs of the snippets will involve a combination of static and dynamic architecture. The are aimed to be both educational, short and concise.

2.4 Version control

The concept of version control is a well-known subject that has its origin from software management [2, p. 1]. Most (software) developers has been using it to some extent, whether it be an amateur or a professional. Having the flexibility to add new features, fix broken ones, or stepping back to previous states of a project are important steps and daily routines to anyone working with digital copies. For this reason, a powerful tool that can allow a team to move and manage a project quick- and easy is of high value. On the market there are many tools for this job. Both open source and proprietary. Two common systems are **Centralized Version Control Systems (VCS)** and **Distributed Version Control Systems (DVCS)**. The tools are normally categorized as either centralized or decentralized. Some examples of centralized tools are **Concurrent Version System (CVS)** and **Subversion (SVN)**. While some examples of DVCS are **Mercurial** and **Git**. The main difference between the two families is the constraint of centralized systems. To have a remote server, a directory and location where you put all your files, it is essential and required to have the network enabled. If it is down, no communication with the server can be done in a centralized system. On the contrary, in DVCSs network availability is not an as crucial factor. One can decide to have none, a single or even several remote servers, while also not having to rely on the network availability. It is possible to work online as well as offline in DVCSs. The beauty of DVCSs is that all modifications are locally recorded and stored, and can at any time be synced online at a later time [2, p. 1].

2.4.1 Git

Git is today the DVCS that has grown from being a niche tool to becoming the mainstream. It has grown rapidly and gained public favor compared to other DVCSs. It is the second famous child of Linus Torvalds, who, after creating the Linux kernel, forged this versioning tool. Git is essentially a tool for versioning files, originally designed as a tool to let hundreds of contributors help and work on Linux kernel. It is understandable that Git has been built up with collaboration in mind, and there is a very robust consideration behind sharing data among computers. In Git, like in many other DVCSs it is possible to work both locally and remotely. A Git remote can be another computer that has the same repository you have on your computer. In basic terms, every computer that hosts the same repository on a shared network can also be the remote of other computers. As already mentioned, a great difference between Git, and other DVCSs, to classical centralized versioning systems such as SVN, is that there is no central server where you can give custody of your repository. However, in DVCSs you can have many remote servers. This yield a more fault tolerant and more flexible system, as there are multiple working copies where one can communicate and get information from. Using remote servers as a contact point for different developers promote work done in a distributed manner. There are many remotes and free online services that offer room for Git repositories. GitHub and Bitbucket are two commonly used remotes [2, p. 53-54].

2.4.2 Basic concepts of Git

In Git context, a folder that contains an initialized Git repository is called a **working directory**. Initially, files and folders are **untracked** in the working directory. This means Git has no idea of what files are important and which one should be treated and perhaps synchronized. However, the fact that files are untracked don't mean that Git is not noting that there is something new or changed in the working directory, in fact, Git is still monitoring everything. However, in order to start tracking a file and include it in a **repository**, it is required to **stage** the file using an add-command. Once added, the file enters the **staging area** (also called index). When a repository is created for the first time and subsequently added, the files are in the state **unmodified**, but once they become edited they switch status to **modified**. That a file is in the staging area basically means that the file is ready to be committed to the repository. The staging area is a virtual place that holds a collection of all the desired files and modifications, before next commit. So a rule of thumb is: any modified or created files that is desired to be committed needs to be staged first [2, p. 13-14]. The following screenshot explains the status life cycle of a file:

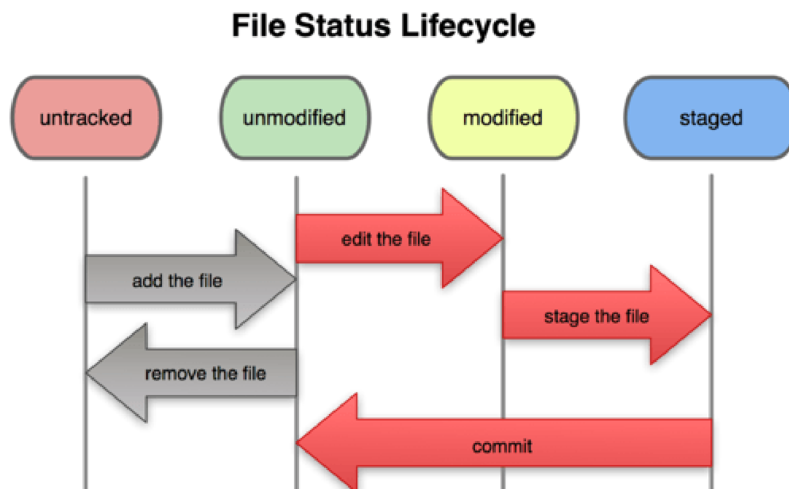


Figure 2.2: The following screenshot explains the file status life cycle [2, p. 14]

2.4.3 Commits

Another central concept in the context of Git are **commits**. Every commit build up the repository. They are in a ordered sequence and can be identified as an acyclic directed graph. Below is a more detailed explanation to commits given by doing a time metaphor.

2.4.4 The past

Previous commits can be represented as "the past", as shown by the commits C1 and C2 in the diagram below, (figure 2.3). A reference to the last commit as well as the parent of the next commit is called the "HEAD" pointer. By knowing the position of the HEAD pointer, one can step back and navigate in the past.

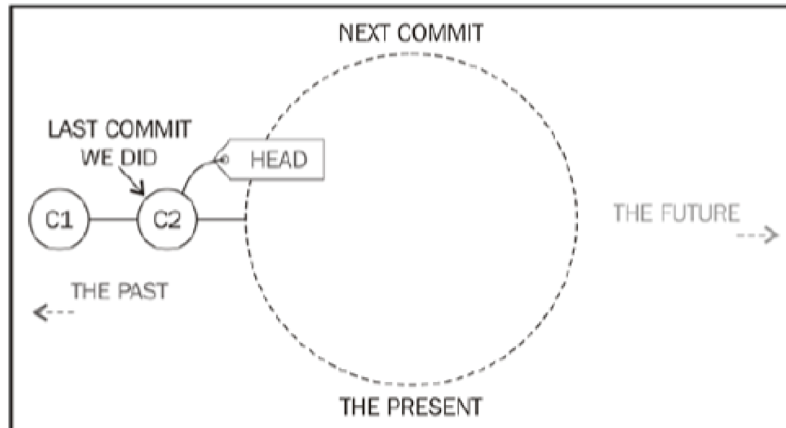


Figure 2.3: The following diagram explains how commits can be viewed as past and present [2, p. 18]

2.4.5 The present

As shown in the consecutive diagrams below (figure 2.4-2.5), the "next commit" is the child of HEAD pointer, and encircles both working directory and items in the staging area (index). Files that are created or modified can be marked and added to the index using the "git add" command. The next commit will only consider the modifications collected in the index and the rest non staged items remains in the working directory as is.

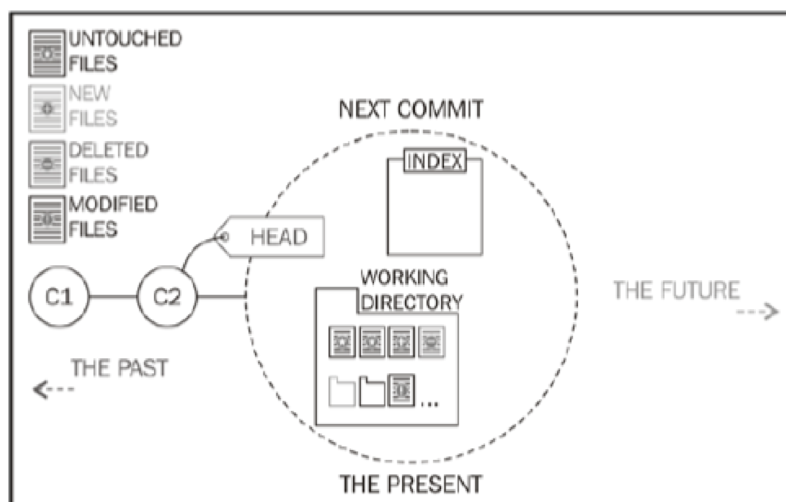


Figure 2.4: The following diagram depict a modified working directory and an empty index [2, p. 19]

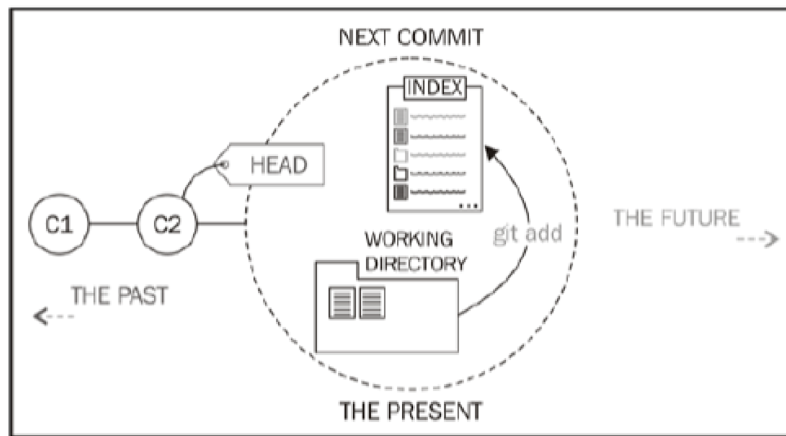


Figure 2.5: The following diagram depict the result when the modified files from working directory has been added to the index [2, p. 19]

Finally, once the commit operation is executed, the processed commit becomes the new HEAD reference as well as part of the past. The index is emptied and the working directory comes back to the initial state and everything can repeat, as shown in figure 2.6:

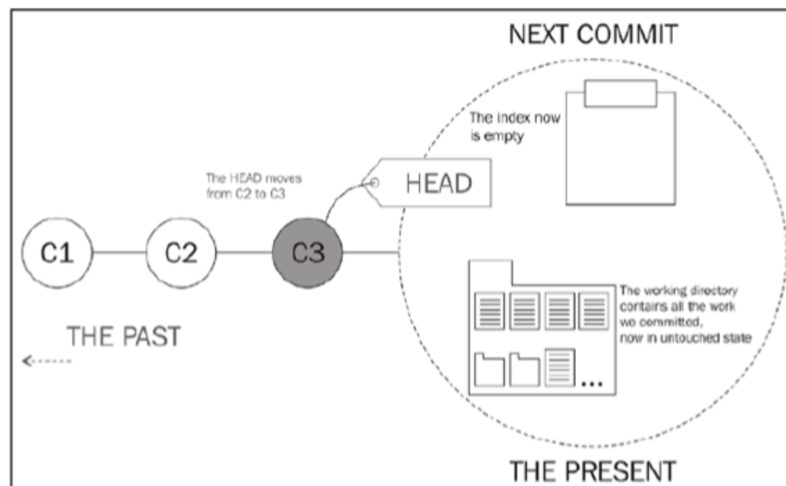


Figure 2.6: The following diagram explains how commits can be viewed as past and present [2, p. 20]

2.4.6 Git history and commit log

A commit can be regarded as a snapshot that wraps all the bundled files collected in the index during that specific commit. In order to review and back track the life of a file one can browse and go back to the different commits in the past. Furthermore, every time a commit is done, a commit message is needed. Commit messages can give a brief explanation and some context to the added files or modifications. This in turn provides a more detailed and easy to follow changelog/history of the life cycle of a working directory. The historical collection of Git commits are called commits log.

2.4.7 Git commit messages

In the book "Git for Humans [20], David Demaree argues that a Git's commit log can be viewed as though it was a newsfeed for your project. He gives the similarity that a commit can be regarded as a headline in a newspaper. The same way a good headline doesn't tell you about the whole story, but sufficient enough to describe what the story is about (without the need of having to read it), is the same way a commit message should be formulated. He also continues to say that if the project it worked by one person alone, or a minor group, the commit log may seem interesting for a historical purpose, but when it comes to bigger collaborations then the commit logs become more valuable as it can be a way to check-in and see what have happened while you haven't been active within the project.
Strictly speaking, there is no hard limit when to comes to character or line numbers when committing a commit message. It can be as detailed and as long as you want [20], [2]. There are a few hard rules for crafting effective commit messages that David is recommending. He argues that with good practice and obedience to the rules below, one can master the elements of producing long lasting and effective commit messages. Below are the interpretation of the rules and guidelines explained by David Demaree[20].

2.4.7.1 Be useful

The principal purpose of a commit message is to summarize a change, in a way that help you and your team members to understand what is going on in a project. The information must be valuable and useful to the people who will read it. In addition, the messages should be as short and as clear as possible. No messages should include text like "stupid bug solved" or "fixing a bug". These are examples of commit messages that are rather useless and will degenerate the project history over time. It's also common to involve a bug or issue number if the team is incorporating a bug tracking system in their work-flow. Like this:

```
"Add missing attribute (A_foo) ;fixes #1357"
```

Some bug trackers, like the one built-in into every GitHub project, are hooked into Git so a commit message like above would automatically mark the bug 1357 as done, after merging into master.

2.4.7.2 Be detailed (enough)

Although the temptation to include lots of details it's highly recommended to refrain from this. While details can be very important for understanding a change, there's almost always a more general reason for a change that can be explained succinctly. Another reason to why it's recommended to write meaningful, yet short, messages is because of the environment the commit log is displayed. A normal Git user will be using the terminal window and due to lack of space a commit log consisting of

short commit messages is much easier to scan. And then there is the space issue, shorter messages also save space. A good rule of thumb is to keep the "subject" portion of a commit message to one line, roughly 70 characters. Exceeding details worth including can still be included as an separate paragraph, however, the subject line is the primary message that will be displayed. Below is an example of a commit message including a short subject line and a trailing paragraph.

```
$: git commit -m "Updated Ruby on Rails version because security

                  Bumped Rails version to 3.2.11 to fix JSON security bug.
                  See also http://weblog.rubyonrails.org/2013/1/8/Rails-3-2-
                  11-3-1-10-3-0-19-and-2-3-15-have-been-released/"
```

2.4.8 Be consistent

Although commit messages may seem very short, there are minor conventions and consistency applicable. Having a short template or wiki page with some examples of good and bad commit message will help things run more smoothly. With a small set of rules it will become easier to write commit messages as well as review them. Below is an examples of 3 commit messages written without consistency or convention, followed by a more consistent way. It gives an understanding to why consistency in writing commit message can be of advantage.

```
* 1.
"added a function F_foo that can convert dollars to sek"
"Adding a new function F_foo2 that can round up integers."
"fixing bug #314 by adding a new function F_foo3 that calculates remainder."

* 2.
"Add function F_foo to convert dollar to SEK."
"Add function F_foo2 to round up to integers"
"Add function F_foo3 to calculate remainder; fixes #314"
```

Figure 2.7: Two examples of git commit logs, 1. has no consistency while 2. uses conventions which provide more consistency thus improved readability

2.4.9 Active voice

Both Santacroce and David recommend writing commmit messages with a imperative present tense. This is because it's basically shorter. Example: "Fix xxx" rather than "fixing xxx" or "fixed xxx". Of course, any tense is allowed, the key is to always stick with the same one.

2.4.10 Guidelines to writing git commits summarized

Ultimately it's up to the team to decide what conventions and rules is most suitable for a particular project. But by trying to incorporate these rules, will most likely support and make the committing more painless. Resulting in a more clean and

easy to follow commit log that can be very valuable for both current and future colleagues.

```
* 1.
# Making the last homepage update before releasing the new site
$: git commit -m "Version 1.0"
# Ten minutes later, after discovering a typo in your CSS
$: git commit -m "Version 1.0 (small error)"
# Forty minutes later, after discovering another typo
$: git commit -m "Version 1.0 (oh, another error)"

* 2.
$: git commit -m "Update homepage for launch"
$: git commit -m "Fix typo in screen.scss"
$: git commit -m "Fix misspelled name on about page"
```

Figure 2.8: A common example of how a new and unexperienced user of git will commit messages. 1. describes the actual commiting, whereas 2. is how the commit should have been done

2.4.11 Branches

Git also allows for branching of a repository. This come handy when one want to make experimental changes to the files in a working directory. As a programmer, branching a repository is useful because there might already exist working code, and by having the opportunity to perform a branch, it is possible to try something new without fearing to break the existing working code. In [2] Santacroce argues that there are no exact rules on how to use branches, in fact one can use them to keep track of extensive work, for development of a new feature or a way to structure and maintain different versions of releases of a project.

Initially the default branch, by convention, is the master branch. This means that the first commit in a brand new repository is done in the master branch. A common habit is to branch from the master branch as soon as possible and only work directly in the master branch if necessary. [2, p. 29]

Differently from other versioning systems like Subversion, a new branch, would result in a new folder. For Git it's different. Branches are interchangeable and, files and folders of the previous branch are replaced with the content of the "swtiched-to branch". In order words, the working directory is constantly a mirror of the current branch. To view the content of another branch one has to "check out", switch, to another branch [2, p. 28-33].

Worth noting is that a repository is a sequence of ordered commits, when a new branch is created, all commits prior to the creation is passed on to the new branch. This is depicted in figure 2.12

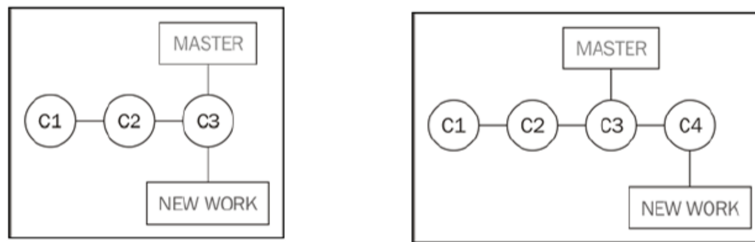


Figure 2.9: An abstraction of how branching can be viewed, "C1-C4" represents commits and "MASTER" and "NEW WORK" are branch names. [2, p. 31-32]

2.4.12 Merge branches

It is often desired to implement the modifications of a feature branch to the original master branch. This concept is called merging. However, this can sometimes be a complex procedure depending on the amount of developers working on the same repository at the same time. For instance, someone might have edited the same file at the same line number as someone else. This will cause a conflict or collision. In this case, Git is unable to do an auto-merge, which basically means Git merge commits from different branches as they were done subsequently in the same branch. Upon collision, Git provide special conflict markers to the affected areas of the file(s), where the user can then manually solve the situation by editing to one's need. To avoid conflicts and major manual work, it is recommended to merge branches frequently. It's not needed to wait before complete work is "done" on a branch prior to merging. Because merging after a few week or months where lots of modifications have been done, can result in a very intricate situation. Too many changes, additions or deletions are happening in the same files etc [2, p. 37-39].

2.4.13 Work-flows

Git is essentially a versioning tool, and there are many well-known and designed work-flows adopted for using Git. Common ones are centralized work-flow , feature branch work-flow and GitFlow among others.

2.4.14 Centralized work-flows

Even in Git one can adopt a centralized way of working. In teams, it is common and often necessary to share repositories with one another. Having a common point of contact is essentially indispensable. A scenario where this work-flow can be used is when you have multiple developers in an office. Someone usually initializes the remote repository on a local Git server such as GitHub or Bitbucket, and other team members clone the original repository on their computer and start working. Work is then pushed to the remote once it is ready/done, to make it available for other colleagues. This method of work-flow require internal rules and detailed defined patterns or else it can be hard to manage and keep every file in sync [2, p. 104].

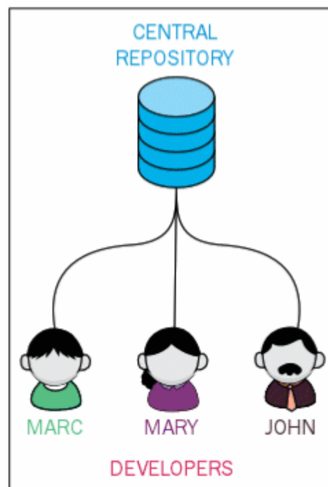


Figure 2.10: Centralized work flow example [2, p. 104]

2.4.15 Feature branch work-flow

A feature branch based work-flow is all about branching. Every single feature the developer is working on require its own branch and when the work is ready, the feature branch is merged onto the master branch.

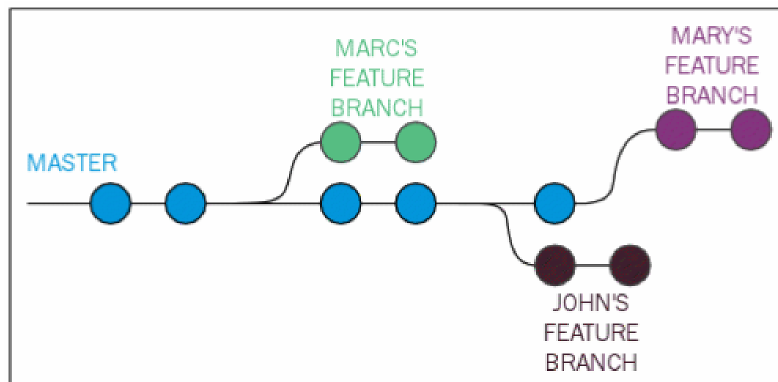


Figure 2.11: Example of feature branch based work flow [2, p. 105]

2.4.16 GitFlow

A more sophisticated work-flow using git is the so called GitFlow. This work-flow has gained success over the years, to the point that many developers, teams and companies are starting to use it [2, p. 105]. Similar to the first work-flow presented, Gitflow also embeds a centralized work-flow. The core concept of GitFlow are the "main branches", that is master-, hotfix- , development- , release and feature branches etc. Each main branch hold a specific role. The master branch is the so called, production-ready branch, which means when you fix or add something new in either a feature branch or bugfix branch, and then merge it onto the master branch, one can assume the changes will be up and running in a matter of hours. The hotfix branches is always derived from the master branch and are, by definition, created

to resolve bugs. The development branch is a sort of staging branch, a platform for a feature branch. And, finally, the release branch contains the next release and one may not add new feature to this branch. The GitFlow approach can be regarded as having different departments with specific roles but are simultaneously closely collaborating with one another. In general, GitFlow, is highly dependant on the team members and "the social contract", meaning members continuously support, comment and test each others code before merging onto the master branch [2, p. 105-109].

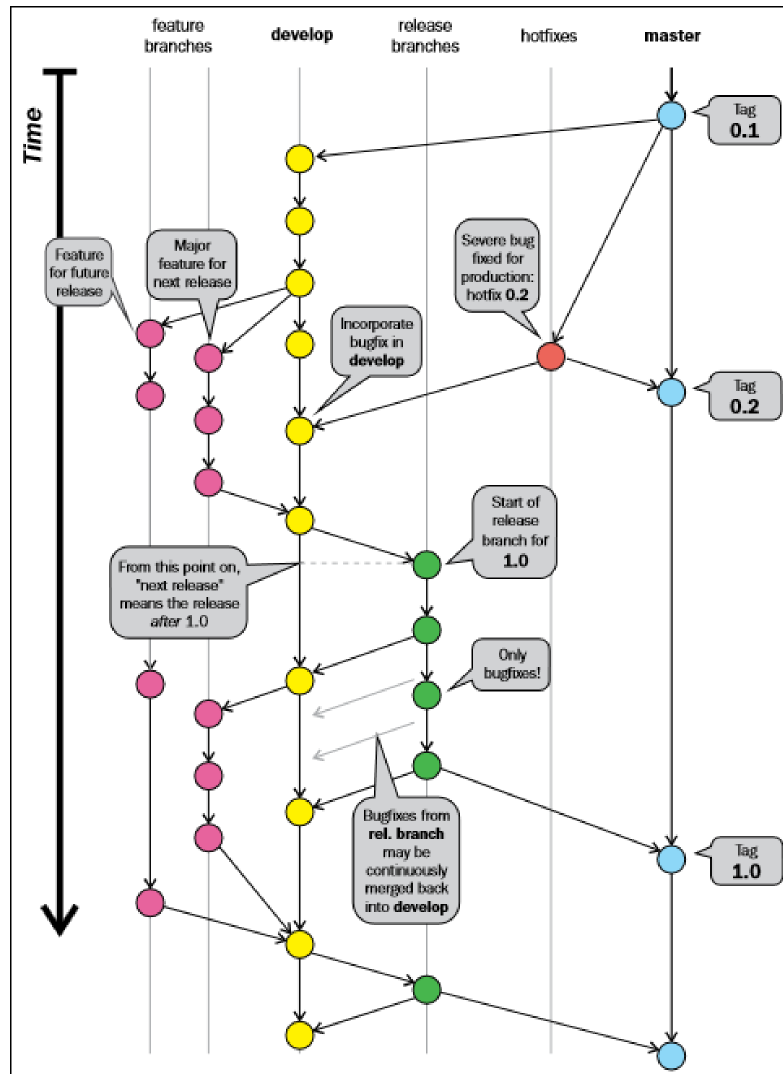


Figure 2.12: Example of GitFlow [2, p. 106]

2.4.17 Summary version control

In this thesis a customized work-flow will be designed appropriate to discrete event simulation projects. A particular interest of the development will be in regard to a work flow suitable for the simulation team at ÅF. An initial version of git work-flow will combine a centralized and feature branch approach. And, depending on the

feedback from ÅF and with the experience gained out of the first version, a more sophisticated approach such as GitFlow will be considered.

3
Methodology - Work process

In this chapter , the steps that are included in the methodologies are presented in a chronological order. All the theories applied here are derived from the theory chapter, unless referred explicitly. Shorter but more detailed discussions will be given for each implementations.

Important to note is that the implementations have revolved around the inherited discrete event simulation model issued by the simulation team at ÅF Industry. The model has involved several hand-overs from different authors and companies, thus with the years has become self-degrading. Moreover, the implementations and workflow have been appropriately adopted to incorporate AutoMod.

The work have been divided in the following steps can be seen in the flowchart 6.1

1. The identification and categorization of maintenance challenges with the model
2. A literature review about software maintenance
3. The creation of AutoMod Extension
4. The first version of snippets
5. The first version of workflow using Git
6. Interview at ÅF regarding maintenance of DES projects using AutoMod.
7. Final version of snippets and workflow using git, with the interview feedback taken into account

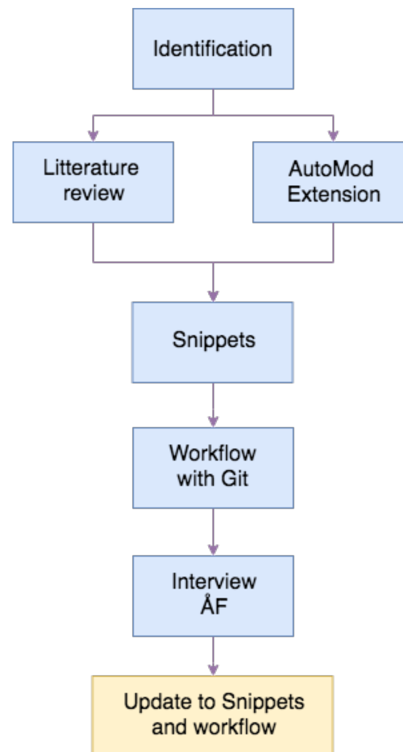


Figure 3.1: Flowchart of the work process

3.1 Identify problems

The initial step of the work involved actions such as extensively identify and analyze the simulation model. The most immediate and apparent issues, as well as potential red flags, were documented and categorized as either "code related issues" or "documentation related issues".

3.1.1 Code-related issues

The main targeting questions for the source code of the simulation model was "What affected the source code's readability negatively, making it hard to break down?".

One of the more contributory cause to the code-related issues were the apparent presence of multiple authors. This in itself, resulted in major inconsistency problems, Meaning, there were difference in naming conventions, different ways of solving problems and varying ways of writing comments etc. Below are the most elevated code-related issues due to inconsistency of the source code:

1. Different way of naming variables
    - Mixing lower- and upper case for both prefixes and sufixes
      `(E.g. v_item, v_Item2, V_item3, V_Item4)`
    - Inconsistency in using prefixes

`(E.g. 'V_time', 'Vi_time')`

2. Different way of naming entities (same as above)
3. Different way of writing code to solve the same problem
4. Different choices in choosing data structures for similar problems
5. Inconsistency in using placeholder variables
6. Redundant and lengthy code (both vertically and horizontally)

3.1.2 Documentation-related issues

From the examination of the simulation model it was further observed that the overall documentation of the model could be improved. A big challenge involved navigating and tracing different items from the project. There were no primary or comprehensive documentation of the project history, and no details or guidelines describing how work ought to be carried out in order to secure a maintainable project. Below are the documentation-related issues, or items missing, that made the traceability of the simulation project difficult.

1. A general change-log for waiting, pending or fixed bugs
2. A general document for all previously (frequently) asked questions
3. A document explaining the different terminology
4. Flow charts illustrating the whole- or sub systems
5. More documentation to previous states and versions of the simulation model

3.2 Literature review

When the challenges had been identified the next step was to carry out a literature review in order to research different fields of theory that could help resolve and mitigate these issues. Relevant literature and other sources was found mainly using the following:

- Chalmers library databases
- Google Scholar
- Research Gate
- ProQuest

The three main keywords that were look into were:

- Software maintenance
- Snippets
- Git

3.3 Creation of AutoMod Extension

It was decided that a few actions and tools had to be developed and incorporated with AutoMod.

Methods and tools such as snippets and git integration are software maintenance tools that would serve very useful later on, but before any of those techniques were introduced there was a prominent bottleneck that had to be taken care off. Although AutoMod's powerful ability to describe and simulate discrete event systems, it was argued that it had a flaw in terms of its developing environment. It was early on in the project concluded that in order to apply the different software maintenance tools, something had to be done with the way of programming in AutoMod. The default AutoMod editor was regarded as too limited in terms of flexibility and customization and would only prevent the implementation of desired maintenance techniques. Below are some of the limitations of AutoMod's editor described. Some arguments are also given as to why a third party editor was consequently chosen.

3.3.1 AutoMod Editor limitations

While programming in the AutoMod language it is recommended that one uses the AutoMod naming conventions. For instance, common entities and parameters are recommended to be named with the following prefixes:

- P_ (Process)
- R_ (Resource)
- F_ (Function)
- V_ (Variable)
- A_ (Load Attribute)

This will ensure that the AutoMod editor is able to recognize the specific standard prefix for entity names, and consequently help speeding up the variable declaration process [21]. This constraint is helpful and will mitigate the concerns regarding inconsistency in naming variables. However, this rule is not mandatory, and by simply failing to follow the rule will yield in a misinterpretation by the AutoMod compiler and the user has to manually declare the variable with the original intention. In a big project where new variables have to be introduced and the dimensions have to be adjusted regularly, this process will eventually become tedious. Fortunately, AutoMod is not limiting developers to only develop code inside the AutoMod editor. In fact, AutoMod allow developers to program in any preferred text editor.

The example above is only one scenario where the AutoMod editor is limited. In fact, being able to work outside AutoMod's editor will not only result in a quicker way of programming, as declaring variables, modifying the data structures such as dimension and type can instead be done directly in the source code from the text editor. Additional advantages are the possibilities to refractor names of entities efficiently

and to use popular text editor features such as intellisense and auto-completion. The two latter features encourages and enables a more consistent programming manner. This is because names of variables are predicted and offered as the user is typing. By being provided with predictions, valuable time is saved from searching or coming up with a new name. Also by selecting the proposed auto-completion suggestion, one eliminate common mistakes such as misspellings. To see an example of intellisense, also known as intelligent code completion, see figure 3.2.

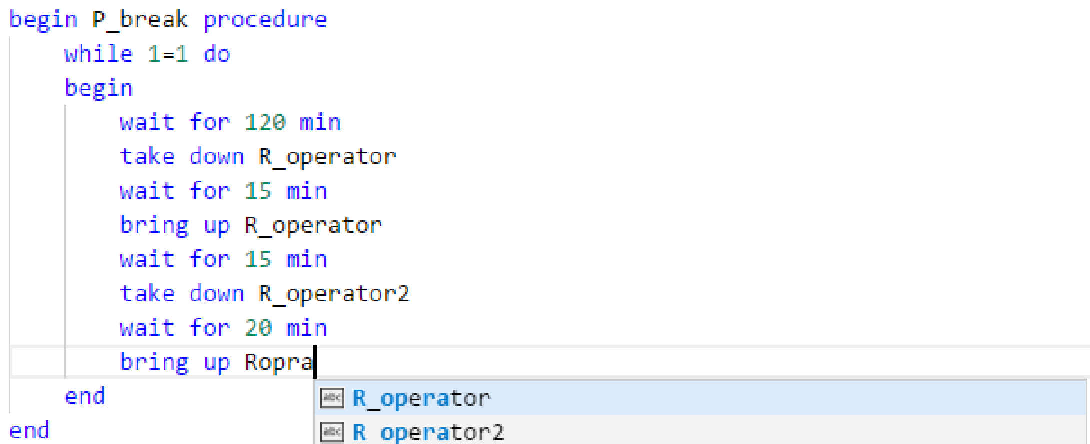


Figure 3.2: An example of intellisense, suggestion is automatically inserted upon selection (auto-completion)

3.4 First generation of snippets

With a more flexible and richly featured developer environment as well as having identified the most prominent and observable problems of the simulation model, it was time to start the process of development that could help prevent the remaining challenges.

It is argued that issues 1-2 from the code-related issues list presented previously, will be alleviated more or less thanks to the built-in features such a intellisense and auto-completion in VS Code. Unfortunately these features will not be sufficient enough for issues such as 3-5 in the list.

Instead, snippets are introduced. With snippets one can implement educational/guiding templates that can be used consistently throughout the source code. Preventing issues such as coding differently for solving the same problem and inconsistent choice of data structures etc.

3.4.1 Snippet

From the observation of the source code of the simulation model, there were quite a few of inconsistencies noticed. First of all, there were misleading and poorly named

placeholder variables and counter indexes. For instance, variables were named differently but had the same purpose:

```
E.g. 'V_temp', 'Vs_String', 'Vi_i', 'j', 'V_i' etc.
```

One can argue that these placeholder variables and index counters are not so self-explanatory and it's required that the code reviewer holds some prior knowledge or has a good understanding of the code, in order to analyze the given content. The main consequence of having inconsistencies and misleading information like this yields a situation where unnecessary time is put on understanding things that could have been self-explanatory in the first place.

Another observation made was the numerous occurrences of code written differently but ultimately solved the same problem, e.g. something that could and had been solved in 5 steps was solved with 7 or 15 steps later on elsewhere. This obviously caused some confusion and made the code more lengthy than needed. To help mitigate these two challenges, snippets proved to be very successful.

3.5 First generation of the workflow using git

Further challenges with the given simulation model was the documentation of the source code. It was argued that the project history and change log were inadequate. Inside the folder of the simulation model there were existing documentation with the intent to keep track of these changes, but due to reasons such as time constraint or poor habits they were neither complete nor up to date. With a continuously updated change log it is possible for the developer to easily stay up to date with what has been implemented or is currently on the pending list. If a developer is inactive or have to come back to a project after a couple of weeks the developer can easily catch up by reviewing the change log. This also means that a future colleague less familiar to the project, or any code reviewer really, can more easily and quickly dive into the project and understand what still needs to be done or is already marked as finished.

Another observation of the given simulation model, was the amount of inconsistency in commenting in the source code. There were occurrences where code-comments were both verbose and redundant, but also times were crucial and appropriate comments were missing. Due to lack of proper guidelines of commenting, different developers had aggravated the source code of the simulation model by coding "quick-and-dirty" workarounds. As a result new features and solutions problems were built on top of each other rather than re-using and incorporating the already existing code foundation. This have made the source code much longer and more complex than needed. Again, the main reason for this was the lack of documentation and an inadequate change log record.

Tools such as intellisense, auto-completion and snippets are useful for alleviating the

coding challenges in 6. But in order to solve for the challenges such as recording and tracking an updated change log will require other means. The distributed version control system Git is chosen as a candidate to help solve this.

Git provides good support for a change log (commits logs), and it is also designed to be developer-friendly, meaning it is aimed to reduce the effort and create intuitive habits for the developer to document changes of a project. By following a set of best practices a developer can effortlessly and efficiently commit changes or implementations to the project, thereby contributing to a continuous and rigorous record of changes.

Git will also help the developer to be more descriptive thanks to the commit messages. More crucial and explanatory comments can now be found in the commits logs rather than inside the source code. There are many different workflows that are applicable using Git as stated in the theory. The chosen one will be presented in the result chapter

3.6 Interview ÅF

The developers of ÅF's simulation team presented a couple of issues revolving simulation model maintenance and project inheritance, and suggested some part-solution and research area to look into. That together with literature study form the first version of AutoMod extension, snippets and Git workflow, presented previously. Later on in the project, an more planned and well-thought interview was held with same team from ÅF to gather more information. Since the different tools and method are developed solely for the usage of the ÅF individually, their opinions are vital when forming the solution. There are to no use if the developer has misinterpret the issues and the solution does not give any value, or if they find it to be redundant or bothersome to use. The goals of the interview were to:

- Get their experience on dealing with model maintenance and project inheritance.
- Get their feedback/input on what has been researched/developed. Let them emphasize additional issues that have surface.
- Summarize the feedback and develop the final solution accordingly to the input.

Since the interviewers at that state had a solid knowledge about issue from studying literature and by speaking to the individual in ÅF, the questions was designed heavily focused on getting the data needed but still letting the participants give their personal opinions. Therefore, the interview was conducted as a Semi-structured interview. [22]

An interview guide was created before the interview and had the purpose to keep

the interview going in the right direction and have questions to fall back on. The interview guide follows the guidelines presented in [22] and is presented fully in the appendix A.

3.7 Final update to snippets and workflow using Git

A final update to the snippets and workflow using Git was carried out with the focus of satisfying the feedback given during the interview and these are presented in the Result chapter.

4 Result

The following sections presents the developed AutoMod extension, developed snippets, proposal of a workflow using Git and a final update considering the given feedback from the interview.

4.1 AutoMod language support extension

The programming environment of choice for this thesis is a continuously growing and popular text editor called Visual Studio Code (VS Code). The choice of editor was chosen due to it's flexibility in managing and implementing personal extensions. Moreover, VS Code is a free and open source text editor that have all the features described above, smart refractoring, intellisense, auto-completion, and many more. Using VS Code as the primary development environment also give the developer the opportunity to run and compile AutoMod code directly in a integrated terminal, and finally, VS Code also allow for easy implementation of custom snippets and integration of Git. That being said, an AutoMod language support extension was created in order to primary incorporate syntax highlighting, auto-completion rules and snippets management.

4.1.0.1 Syntax highlighting and auto-completion

Syntax highlighting is the feature that displays certain text or source code in different colours and fonts according to a set of rules and predefined keywords [23]. When coding outside of AutoMod's editor there is no syntax highlighting available by default. However, in VS Code one can implement a custom syntax highlighting grammar by following their Application Programming Interface (API). In addition to AutoMod's editor it is now possible to modify syntax color or even apply new or modify the rules of the syntax highlighting, as preferred. All of the figures involving snippets show an example of the syntax highlighting that has been created.

In addition to the syntax highlighting feature a library of all AutoMod's reserved keywords were added to the VS Code extension. The result can be regarded as an extended database of auto-completion words, meaning the developer is now able to look up AutoMod's keywords directly from VS Code and insert it upon selection. Being able to look up and auto-complete keywords mitigate common errors such as misspelling a keyword. This also adds the possibility to search for a specific action on the fly. Some of the beginning reserved keywords of AutoMod starting with the

letter "a" are the following:

```
'absent','absolute','ac','acc','acceleration','activation','active',
'after','aisle','aisles','align','all','along','among', ... etc
```

4.2 Snippets

The first generation of snippets have been designed to be guiding, yet easy to follow, short and concise. The example in figure 4.1 and figure 4.2 below are great examples of how using snippets can aid the developer in avoiding variable misuse and overall inconsistency and uncertainty when coding.

```
36
37 §breakd
38      §breakdown1_resource
        §breakdown2_motors
        §breakdown3_triangular
        §breakdown4_print_to_label

'Example 1.' (AutoMod)
begin <P_breakdown> arriving
 wait for <5> min        // start in 5min
 while 1=1 do begin
  take down <R_machine> // user-defined
  move into <Q_machine>
  wait for <15> min      // MTTR
  bring up <R_machine>
  wait for <7.75> hr     // MTTF
 end
end
```

Figure 4.1: An example of a snippet designed for break down procedures in AutoMod. Procedures are automatically inserted upon selection

```
36
37 §while
38      §while_double
        §while_double_wait
        §while_single
        §while_single_wait

'Example 2.' (AutoMod)
set i = 1
while i < <n> do begin
    //do something
    set j = 1
    while j < <n2> do begin
        //do something
        inc j by 1
    end
    inc i by 1
end
```

Figure 4.2: An example of a snippet designed for while loop routines in AutoMod. Procedures are automatically inserted upon selection

4.2.1 Snippets for different process types

Creating processes (procedures, functions, subroutines etc) is one of the building blocks in AutoMod programming, to make the procedure of creating processes more

autonomous, efficient and less error prone some specialized snippet templates have been designed. "The different process snippets" are built up in three separate parts, see figure 6.1. The first line is the code for declaring the process, the next block of code is the introductory-documentation and the rest is the body content. The snippets are also a combination of static and dynamic. Static in the sense that the majority of the content is always the same, and dynamic in a sense that minor parts of the content is also dynamically changing. These minor text spots are situated at different locations of the content, and as the user is typing the input values are altered across the multiple occasion inside the content. For instance, if "begin_function" were to be selected in the figure 6.1, then the snippet content would be auto-completed and the "<F_Name>" would be highlighted and selected on two locations (row 6 and row 12). If the user would start typing then "<F_Name>" would be replaced on the two locations according to the user edited input. The placeholder text "<F_Name>" is to inform the user that the function name should be starting with the prefix "F_".

```
1   §begin_
2           §begin_function  Descriptions: 'Describe main con..
3           §begin_procedure
4           §begin_subroutine
5
6   FUNC name <F_Name> type Integer PARAM name <ArgInt_Parameter> type Integer
7   /**********************************************************************
8    // Descriptions: todo
9    // Parameters:   todo
10   // Example:     todo
11   **********************************************************************/
12  begin <F_Name> function
13
14    return 0
15  end
16
17  PROC name <P_Name> 0 traf Infinite nextproc die
18  /**********************************************************************
19   // Descriptions: todo
20   // Parameters:   todo
21   // Example:     todo
22   **********************************************************************/
23  begin <P_Name> arriving procedure
24
25  end
26
27  SUBRTN name <S_Name>
28  /**********************************************************************
29   // Descriptions: todo
30   // Parameters:   todo
31   // Example:     todo
32   **********************************************************************/
33  begin <S_Name>
34
35  end
```

Figure 4.3: The top part displays the intellisense prompt of the three implemented processes type snippet. Below is the auto-completed version function, procedure and subroutine snippet content respectively

4.3 Workflow using Git

The git workflow is currently non-existent and is designed to be dedicated to discrete event simulation projects, in particular the team at ÅF, was based on the two first concepts presented in the version control section of the theory. In other words, it combines a centralized- and a feature-branch based workflow. The chosen remote server is a platform called Bitbucket, and is already used at ÅF, and will hereby serve as the central contact point for this particular simulation project.

Worth noting is the fact that the workflow will be designed in consideration to AutoMod being the main programming language.

4.3.1 Anatomy of workflow

The workflow is designed to require new branching of the master branch for every considered feature, bug fixes and quick-fixes. The principle is that the master branch should always have the latest stable version in order to always have a latest working version of the project, which multiple developers can access. Therefore it is important to always pull from the remote master branch before merging and pushing to it, in case changes have been made to the master branch. In order to keep the project history clean and easily traceable for the future, it is required that a branch is only limited to a single change (e.g. single bug or single feature).

Although the possibility to include two features in a branch it's required to either first merge the current feature branch or to branch out from the feature branch itself. Having too many commits waiting to be pushed on a branch can be an example and signal that the branch needs to be merged or branched out further. When the content in a branch is ready to be merged, it should be merged to the "closest branch (the branch it has been branched out)" for better structure and easier for tracking.

4.3.1.1 Bug branch

A bug branch is where the developer handle all the bugs throughout the project. Since a bug is usually solved by a single developer, the branch does not need to exist on the remote server. Furthermore, a project may include countless of bugs and "make no sense to push all the bug branch to the remote server, unclear, a lot of branches". Commits on bug branch shall start with a commit token followed up with the commit message; b<number>: <subject line>. Once the bug has been solved, the commits message of the merging process shall include "b<number> solved". The index number in the commit message can be tracked to an excel-document where a more detailed explanation about the bug is presented.

4.3.1.2 Quick-fix branch

A quick-fix branch is used to solve minor changes of the code such as; Deletion of code, code refactoring, replacing names of data structure or entities, commenting/uncommenting etc. Similar to bug branch, a quick-fix branch is shall only be created as a local branch and merge into master once changes are done. Commits on quick-fix branch shall start with a commit token followed up with the commit message; qf: <subject line>.

4.3.1.3 Features branch

New features shall be handled in individual feature branches. Depending on the feature size and if multiple people are collaborating on that feature, the branch shall be pushed to the remote server. Thus the master branch will not be affected by the development of the feature and the feature branch on the remote server can be used as a access point for the involved developer. Bugs and quick-fixes within a feature should be branched out further and "follow the same procedure,

above". Commits on feature branch shall start with a commit token followed up with the commit message; f<number>: <subject line>. Commits on bug and quick-fix branches follows the same structure adding feature token in front, e.g. f<number1>/b<number2>: <subject_line>.

4.3.2 Internal rules

In order to keep the communication and structure maintainable it is required to introduce some internal rules in addition to the information above [2, p. 37-39].

4.3.2.1 Committing

From the theory in section it's advised to keep a detailed and structured way of performing commits. Here is a set of best practices for commits designed for this workflow.

- Should have a length limited to 70 characters
- Should answer "I just did a thing, a thing I did was:"
- Should be written in imperative present tense
- Should start with lower case character, e.g. "fix breakdown of clamp truck"
- Common starting verbs of subject lines: Fix, Add, Replace, Remove, Refractor
- Issue number will be written in the beginning, e.g "f/b<#>:<subject_line>"
- If commit involved something "missing", add what is missing to the subject line
- If something was solved, point out what was solved rather than how it was done

In VS Code there is a integrated GUI panel that cover and offer the most common commands of Git. Among other commands, there is a message field specially dedicated for commit messages. This field will prompt a warning as soon as the user exceeds 70 characters.

4.4 Input from the Interview and final update

The interviews were held with two members from ÅF's simulation team. It was held individually and both answered all the questions listed in appendix A and gave their personal opinions on the matter. Following sections describes the summarized outcome of the interviews, regarding the snippets and proposed workflow using Git, and final updates that was done according to the feedback.

4.4.1 Regarding the Snippets

From the interviewing session it was learned that there were already some preexisting snippets available from ÅF's part. There were only a few of them and they were still not officially distributed to the team. There were future intentions of implementing more ones and undeveloped snippet concepts pending. One challenge

regarding the snippets, however, was the management and integration of it. The distribution of the snippets among team members were currently non-existent due to lack of centralized repository. A suggestion was given to the simulation team at ÅF. In order to manage the snippets in a distributed way, the snippets ought to be stored on a Git repository on Bitbucket or similar. This way anyone with access to the repository can contribute to changes, new implementation or deletion of snippets. Furthermore, because Git integrates nicely with VS Code it is also possible to easily access the repository and different snippets directly from VS Code.

Some of the pending ideas as well as already developed snippets were some concepts of shorter control flow statements like: for-each loops and while loop and order-list statements. Other pending snippet concepts they had were ideas such as larger modules and sub-system that could prove handy and save a lot of time if designed as snippets. E.g. conveyor system, scheduling systems or sub systems.

4.4.2 Documentation to declaring different processes

Special focus and input was given towards the snippets of different processes. Previously the process snippets, see figure 6.1, had all the same type of introductory documentation. In other words they contained the topics "Descriptions, Parameters and Example". However, from the interview some feedback was given regarding this. Because the processes have different main purposes the introduction documentation should be distinct depending on the process type.

The following final updates were made to the process snippets:

```
1   §begin_
2         §begin_function  Descriptions: 'Describe main con…
3         §begin_procedure
4         §begin_subroutine
5
6   FUNC name <F_Name> type Integer PARAM name <ArgInt_Parameter> type Integer
7   /***********************************************************************
8    * Descriptions:
9     * Private variables / attributes
10      -
11    * Parameters
12      -
13
14   * Example:
15   ***********************************************************************/
16  begin <F_Name> function
17
18    return 0
19  end
20
21  PROC name <P_Name> 0 traf Infinite nextproc die
22  /***********************************************************************
23   * Descriptions:
24    * Public variables / attributes    (static)
25      -
26    * Private variables / attributes
27      -
28    * Inherited variables / attributes
29      -
30    * Subroutines
31      -
32   ***********************************************************************/
33  begin <P_Name> arriving procedure
34
35  end
36
37  SUBRTN name <S_Name>
38  /***********************************************************************
39   * Descriptions:
40    * Public variables / attributes    (static)
41      -
42    * Private variables / attributes
43      -
44    * Inherited variables / attributes
45      -
46   ***********************************************************************/
47  begin <S_Name>
48
49  end
```

Figure 4.4: The final version of process snippets with improved introductory documentation.

4.4.3 Regarding the Git workflow

There were less feedback given towards the thesis's suggestion of workflow for using Git. The simulation team were still deciding on a version control system appropriate for their working method, and they had only started to experiment with Git. Hence,

there were neither a workflow adapted nor internal rules set up yet, and when they were using Git they had previously only committed on the master branch. For that reason, the initial impression of the suggested workflow was mixed feelings. They saw positively on the intention of the workflow, that is, to create more structure and offer a better history of projects. However, having no experience with branching and the requirement to follow internal rules made the whole workflow suggestion also somewhat discouraging. A more rigid discussion concerning this is given in the Discussion chapter.

4.4.4 Feature and Bug tracker

It was further discussion in the interview that in order to implement Git rigorously, then it was needed that internal rules had to be formed. Among other things the commit messages had to be short and concise, and because of this it was decided that a bug and feature tracking system will be needed. This means that for each bug and feature that is added to the model there will be a new item added to a database. The database is nothing but an excel document containing the bug or feature identification number, description/explanation of the bug or feature and other comments. See figure 4.5 and figure 4.6 below.

Add new feature

Feature ID	Date	Status	Name	Description	Comments
1	8/20/2018		Commisioner	Involves all the commissioning, drivining logics and item distribution.	3 commisioner cranes, 120 racks
2	8/20/2018		picking system	Involves all the picking for KPVs and Pallets, carrier driving routes and logic and item distribution.	16 alleys and 30 racks per alley.
3	8/22/2018		roller conveyour	Involves roller conveyour logics and item distribution	5 roll lanes and 3 levels.
4	8/23/2018		<name>	<description>	<none>

Figure 4.5: An example of a feature tracker sheet in excel. By adding a new feature through the button "Add new feature", a new row containing an incremented feature number, current date, status and other placeholder values are auto-filled. The status colors green, yellow and red represent done, in progress or not started respectively

Add new bug

Bug ID	Date	Status	Name	Problem	Outcome	Cause	Comments
1	8/20/2018		gate 19	First load that reaches gate 19 will not continue, subsequent arriving loads are queued	Deadlock on gate 19	Duplication of routeIds in Vi_RouteIdsToGate	second time routeId 785 arrives it will delay the
2	8/20/2018		double routeIds	Duplication of routeIds in last order release	Deadlock in gates on day 8	Reoccuring routeIds from day 1 happens on day 8	the simulation model runs day 8 (Sat) as day 1 (Sat)
3	8/22/2018		departure time	Departure time for each carrier in Vi_Carriers is not sorted in an ascending order.		S_Release is called even though all values in Vi_Carries are 0	make sure to check if carriers are loaded before calling S_Release
4	8/23/2018		<name>	<problem description>	<outcome description>	<cause unknown>	<none>

Figure 4.6: An example of a bug tracker sheet in excel. By adding a new bug through the button "Add new bug", new row containing an incremented bug number, current date, status and other placeholder values are auto-filled. The status colors green, yellow and red represent done, in progress or not started respectively

5
General Discussion

In the following sub sections there will be discussions given regarding the result, some further comments concerning the interview feedback, topics where the implementation didn't fully apply the researched theory and the sustainability aspect.

5.1 Results

Banks [3] is helpful for describing and initially defining a discrete event simulation model, but when it comes to project handovers or maintaining them, the methodology can be rather insufficient. It has been previously mentioned by Lehman that software maintenance is highly important due to the fact that when the real system changes the corresponding software system needs alteration accordingly [15]. In other words, the model/source code of a DES project of a manufacture needs to be updated and modified as the manufacture itself is implementing new changes. The reason is to keep the model active and up to date, making it possible to continuously perform relevant analyses and experiments. Lehman [15] and Jiang[18] have both argued that Snippets support software maintenance by reducing the manual effort of carrying out repetitive tasks required for building DES models. With this reduction it also help mitigate easily done mistakes and more frequent human errors such as misspelling and inconsistent coding. From an industrial point of view, snippets are valuable thanks to the less time spent coding recyclable and error prone code, and by having an overall more consistent source code the model end up being more maintainable. This encourages quick and continuous improvements which in the long run save time and money [18].

At the same time from an industrial point of view, git as a version control tool [2], keeping files stored on a shared directory as well as allowing high flexibility in terms tracing files and saving them in different "states", support the software maintenance work additionally. This may many times be a major lifesaver and can nevertheless contribute advantageously for the knowledge keeping and knowledge transfer.

With both snippets and Git as software maintenance tools integrated to bank's methodology, one can argue that the outcome is an extended version of bank's methodology that support the insufficient parts of software maintenance challenges involving DES models.

5.2 Interview feedback regarding snippet management

During the interview it was discussed whether it should be a single person, multiple persons or the whole team who ought to be handling the snippet management. Previously, there were no centralized directory for keeping the snippets. In fact, they were stored locally as text files and when needed the snippets were introduced to a project/model by a copy-and-paste method. In other words, if a developer "Mr. X" had 1-3 snippets then the same developer had to be responsible for distributing the snippets among the team members. At the same time, the team members had to make sure they had the latest version of the corresponding snippet from "Mr. X", in order to avoid using an outdated snippet. The distribution of the snippets was normally done by email, and that way of handling snippets was impractical when it came to regular modification of the snippets. Thanks to the proposal of integrating the snippet management in VS Code these issues are mitigated. The snippets are instead stored on a Bitbucket repository using Git and there is no need to manually look for the separate snippet text files. The snippet integration will now prompt the user with any appropriate snippet option, directly in VS Code. Having the library of snippets stored online on a Git repository makes it easy to add, modify and delete snippets by any liable developer. This also means it's easy to pull down the latest modification from the repository, making it easy to make sure one have the latest snippets available. It's is also argued that having a web page explaining basic best practices of how to create snippets makes it easy for a developer to add new ones without spoiling the consistency. This will be regarded as one of the desired future work.

5.3 Interview feedback regarding version control systems

When the subject of version control systems arose during the interview it was said that Git had been used previously. However, the simulation team at ÅF were still experimenting with various version control systems, and the experience of using Git to manage a simulation model was minor. But they were looking forward and gave promising feedback towards a better Git integration in their team. Prior to any of their version control systems, there had been mostly email communication in order to distribute different simulation model versions among another. This was a non sustainable solution, in regard to keeping the model up to date and easily accessible. The minor experience with Git also meant that the simulation team at ÅF had little experience with Git and had not yet a workflow for Git nor any internal rules to commit messages or branching etc. When the thesis's workflow of using Git and the accompanying internal rules were presented, it gave rise to mixed opinions. The team at ÅF argued that there had been challenges of even introducing Git among all team members, and thus far committing from a single branch (master) had been

sufficient.

There were discussion of whether the mentioned branches (feature, bug, quick-fix) were needed at all, as these could possible cause confusion and eventually end up spoiling the infrastructure or deterring team members to start using Git.

The main purpose of the workflow is designed to keep a project more maintainable both for current usage and future resuming work. However, if the internal rules are not explicit and clear enough this can spoil the essence of the workflow. To give an example, if the internal rules regarding merging are not followed correctly due lack of understanding or poorly formulated guidelines, this can cause merge conflicts in the simulation model which will eventually cause bottlenecks in developing code.

One of the major obstacles from ÅF's side was the introduction of using branches versus performing all the commits on a single (master) branch. It was debated though that if a model contain a lot of different sub systems, these systems should be branched as a separate feature branch. By doing so it's much cleaner and easier to read the commit logs for each and individual sub system. With just a single master branch all of these information would be cluttered and there wouldn't be possible to filter the logs in any way. Below are examples given illustrating the advantages and clarity using branches.

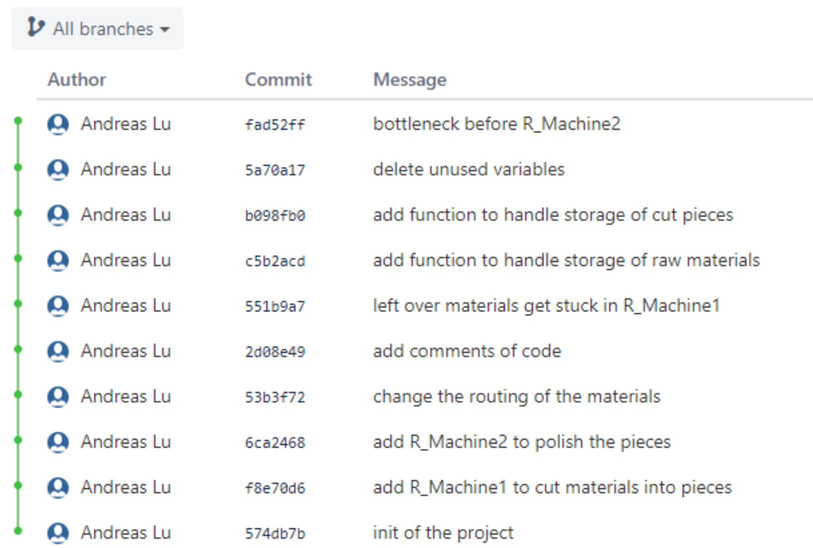


Figure 5.1: An example visualizing a git work flow using no branch features and no commits token for different types of commit.

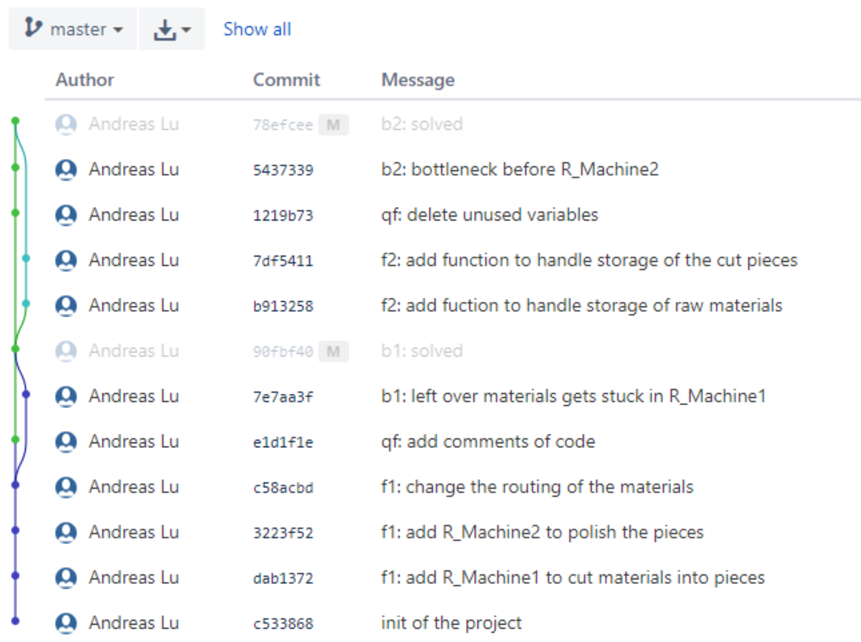


Figure 5.2: An example visualizing a git work flow using branch features and has commits token for different types of commit.

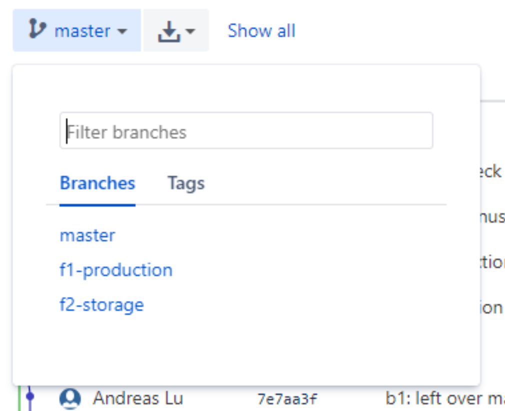


Figure 5.3: Filtering options of branches

f2-storage Show all

Author	Commit	Message
Andreas Lu	7df5411	f2: add function to handle storage of the cut pieces
Andreas Lu	b913258	f2: add fuction to handle storage of raw materials
Andreas Lu	90fbf40 M	b1: solved
Andreas Lu	7e7aa3f	b1: left over materials gets stuck in R_Machine1
Andreas Lu	e1d1f1e	qf: add comments of code
Andreas Lu	c58acbd	f1: change the routing of the materials
Andreas Lu	3223f52	f1: add R_Machine2 to polish the pieces
Andreas Lu	dab1372	f1: add R_Machine1 to cut materials into pieces
Andreas Lu	c533868	init of the project

Figure 5.4: Filter out branch f2-storage and shows the commit history included in f2-storage

The simulation team at ÅF agreed that with a mutually developed guide line for internal rules, it was worth putting the master thesis's workflow to a test. The agreement was to initially try to make everyone start using Git, and to begin with, have most work on a master branch with no mandatory branching. When everyone has become more comfortable with the fundamentals of Git, such as how and when to commit, it's believed that the remaining essentials of the proposed workflow will come naturally.

5.4 Why a dedicated software maintenance team is not needed at ÅF

From the theory chapter about software maintenance it was mentioned that it's recommended to have a software maintainer or even a whole team dedicated for software maintenance. This is because of reasons such as developers not being fond of the maintenance work or there's a lack of knowledge of how the work is done etc. At ÅF the situation is currently limited and introducing a whole division dedicated for software maintenance is not a viable option. Instead, it's argued that if the maintenance work can be made in a more simplistic and more intuitive manner, using snippets and incorporating Git among other things, this will yield a very good initial foundation for maintaining the DES projects. Moreover, the size and work of the DES projects are often manageable by 2-3 persons alone and the team normally stay within close range among each other so the communication is fairly easy and practical. If the project required many more developers and if the developers were situated at scattered locations, then it would be helpful with someone or multiple individuals dedicated to software maintenance. But that is something that ÅF can and will look into in the future.

5.5 Sustainability aspect

The proposed thesis solution contributes to an improvement of simulation regarding development and long-term usage of a simulation model. Making simulations more viable in terms of being more time-efficient and less work-intensive, make them more attractive in the market for other companies.

Simulations can be used for analyzing and performing experiments of a non-physical system without affecting the real systems. Therefore, costs from test failures, stop of productions, bad investment, etc. can be reduced. Radical changes can be thoroughly thought-out using simulations prior to having them applied, which can be regarded as positively from an environmental perspective.

With that said, this project was set to improve simulation in ÅF and through that increases the usage of simulation, and doing so a contribution has been made.

6
Conclusion

The project started out with the challenges such as long-term maintaining a simulation model and the desire of incorporating a time-efficient and simple work method for both resuming less active simulation models while also enable concurrent collaboration among multiple colleagues. In order to come up with a solution for these challenges research were carried out combining literature studies, examination of a simulation model and an interviewing study. The main results were tools such as an AutoMod extension for VS Code, a wide range of snippets and a workflow using Git including internal rules and a bug and feature tracker systems. These have been developed with the aim to mitigate maintenance challenges that arise when working with discrete event simulation projects. It has been proven and demonstrated that snippets and Git can be used to educate or guide new incoming developers as well as helping them to quickly put themselves into an old simulation model. Furthermore, special types of snippets were carefully designed with documentation blocks, making code reviewing, less painful, easy and more effective, see figure 4.4. With a clean and extensive record of the project history, thanks to Git integration, it's easy to gain knowledge as a new developer, and likewise, as easy for a previous developer to transfer knowledge on the go. Having all these tools gathered at one single location, that is in the VS Code editor, makes the workflow seamlessly and more stimulating.

Overall the group of snippets that has been developed handles the inconsistency issues when it comes to naming conventions, choice of solutions for the same problem or data structure and the way of commenting. Meanwhile the proposed work-flow using Git solves the documentation-related issues, mainly when it comes to having a better trace-ability option. Figure 6.1 shows which of the implemented tools solves the respective identified issues.

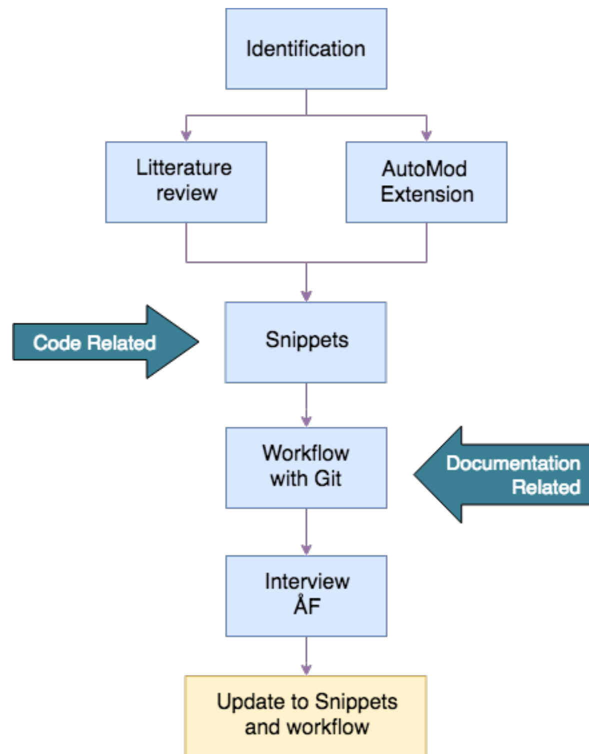


Figure 6.1: Flowchart of the work process showing where the main identified issues could be solved

This thesis has served very helpful for our personal development. Additionally, it has resulted as a feasible decision material for the growing simulation team at ÅF Industry and shown the great importance and indisputable challenges emerging from software maintenance.

7

Future work

Most of the methodologies and tools developed in this thesis are still in its infancy stage, meaning there is still a lot of testing and future evaluation to be done. Hence, future work could be to study different outcomes and reaction resulting from the usage of the tools. Subsequently perform recursive analyses and additional research to further improve the tools.

There are already a set of internal rules developed for the workflow using Git. But there are currently no place where the instructions are stored online. Thus, it's desired to create a web page dedicated to the enlisting the following:

- Internal rules for the workflow using Git
  - How/When to write commits
  - How/When to branch and merge
- Snippets
  - How to add/delete/update snippets
- Other
  - How to incorporate AutoMod with VS Code
  - Installation of AutoMod extension

Finally, there is also a desire in the future to implement a linter tool to the AutoMod VS Code extension. A linter is basically a tool to analyze the source code to flag programming errors, bugs, stylistic error and suspicious constructs [24]. Thus, with a linter one would add yet another layer of protection in regard to mitigating the code related challenges listed in list 6. For instance, if developers are still mixing lower- and upper case, or naming entities and variables inconsistently or any of the other code related issues for that matter, then a linter will be helpful. It will help mark the code with an error in the same manner a spell checker shows a red wriggled underline under a misspelled word in any common text editor, e.g. MS Word. It could also be useful to have the linter intelligently provide the developer with information where and when an appropriate snippet should be used.

Bibliography

[1] Doug Simon, Christian Wimmer, Bernhard Urban, Gilles Duboscq, Lukas Stadler, and Thomas Würthinger. Snippets: Taking the high road to a low level. *ACM Transactions on Architecture and Code Optimization (TACO)*, 12(2):20:1–20:25, 2015.

[2] Ferdinando Santacroce and Ebook Central (e-book collection). *Git essentials: create, merge, and distribute code with Git, the most powerful and flexible versioning system available.* Packt Publishing, Birmingham, [England], 2015.

[3] Jerry Banks. *Discrete-event system simulation.* Pearson Education, Upper Saddle River, N.J, 5., international edition, 2010.

[4] Chung-Jen Chen, Yung-Chang Hsiao, and Mo-An Chu. Transfer mechanisms and knowledge transfer: The cooperative competency perspective. *Journal of Business Research*, 67(12):2531–2541, 2014.

[5] Jeffrey L. Cummings and Bing-Sheng Teng. Transferring r&d knowledge: the key factors affecting knowledge transfer success. *Journal of Engineering and Technology Management*, 20(1):39–68, 2003.

[6] Ronald G Askin and Charles R Standridge. *Modeling and analysis of manufacturing systems.* John Wiley & Sons Inc, 1993.

[7] Lars Holst, Lund University, Department of Electrical, and Information Technology. *Discrete-Event Simulation, Operations Analysis, and Manufacturing System Development: Towards Structure and Integration.* PhD thesis, 2004.

[8] George S. Fishman. *Discrete-event simulation: modeling, programming and analysis.* Springer, New York, 2001.

[9] AutoMod simulering av produktion och logistik. `http://www.automod.se/index.html`. Accessed: 2018-06-10.

[10] J. Martin and C.L. McClure. *Software maintenance: the problem and its solutions.* Prentice-Hall, 1983.

[11] B. Meyer. *Object-oriented software construction.* Prentice-Hall international series in computer science. Prentice-Hall, 1988.

[12] N.B. Standards, National Institute of Standards, and Technology (U.S.). *Federal Information Processing Standards Publication: Guideline on Software Maintenance.* NIST federal information processing standards publication; NIST FIPS. U.S. Department of Commerce, National Institute of Standards and Technology, 1984.

[13] A. Von Mayrhauser. *Software Engineering: Methods and Management.* Suny Series Frontiers in Education. Academic Press, 1990.

[14] A. Abran and J.W. Moore. *Guide to the software engineering body of knowledge.* IEEE Computer Society, 2004.

[15] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[16] Alain April, Alain Abran, and IEEE Xplore e-books (e-book collection). *Software maintenance management: evaluation and continuous improvement*. Wiley Interscience, Hoboken, N.J, 1 edition, 2008.

[17] Ling Liu, M. T. Özsu, and SpringerLink (e-book collection). *Encyclopedia of database systems*. Springer, New York, 2009.

[18] He Jiang, Liming Nie, Zeyi Sun, Zhilei Ren, Weiqiang Kong, Tao Zhang, and Xiapu Luo. Rosf: Leveraging information retrieval and supervised learning for recommending code snippets. *IEEE Transactions on Services Computing*, pages 1–1, 2016;2017;.

[19] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: dynamic and interactive synthesis of code snippets. pages 653–663. ACM, 2014.

[20] David Demaree. *Git For Humans*. Jeffrey Zeldman, 2016.

[21] Applied automod 12.6.1 user's guide. pages 2.23–2.24. Applied Materials, 2015.

[22] Chauncey Wilson, Books24x7 (e-book collection), ScienceDirect (e-book collection), and Inc Books24x7. *Interview techniques for UX practitioners: a user-centered design method*. Morgan Kaufmann, Amsterdam;Boston;, 2014;2013;.

[23] J. D'Anjou, S. Fairbrother, D. Kehn, J. Kellerman, and P. McCarthy. *The Java Developer's Guide to Eclipse*. Addison-Wesley, 2005.

[24] I.F. Darwin. *Checking C Programs with Lint*. C programming utility. O'Reilly & Associates, 1991.

A

The interview guide

This is the interview format used during the interview and all the questions that were asked:

Introduce the subject: The subject of this interview is about maintenance of a simulation model and how it can be affected by different matters.

Explain the purpose of the interview: The purpose of the interview is to improve the current solution by gathering information of the participants experiences and opinions of the subject.

Present the setup of the questions: The questions are divided into two waves. The first wave of questions are general questions about how the participants experience and how they have dealt with maintenance of simulation mode. The second wave of questions mainly focuses on gathering their inputs on the suggested solution. The questions itself are bunched together in different categories.

Questioning:

General in DES-project:

- How do you guys work in bigger projects (at least two developer for the simulation model) in regard to maintenance of simulation model (documentation, version control, workflow, etc.).
- What do you want/What do you think is important, as a developer, when you take over a DES-project?
- What is your experience of inheriting a project from another developer? Following question, Did you use any tool/method to ease the process of understanding the model?

Code-related:

- Have you use any method/tool when you develop a model in order to facilitate understanding of logical expression, automatize workflow, avoid miss-spelling or miss-usage of variables, etc.?
- How do you solve coding similar logic multiple times?
- How do you avoid solving similar problems differently? (E.g. using different data structure)
- How do you avoid writing long codes?

- How do you avoid inconsistent variable- entities naming such as mix of lover/upper case or inconsistent prefix?
- Do you have any experience on using/creating snippets? Following question, to what purpose?

Version control tools:
- What version control tool are you using/have you used? Following question, Pros and cons of the following tools?
- How much experience do you have with git? Following question, Are you using git regular for all the DES-project? Are you using en git work-flow?

Snippets
- What do you think about our current snippets?
- Are there any standard functions/process that you could think of, adding to the snippets?
- Do you have any other snippets idea?

Git workflow
- What do you think of using a combination of git feature branch based and centralized workflow?
- Can you see any prons and cons using this method?
- Do you think our git workflow will satisfy the need in ÅF regarding version control?
- Do you have any idea or suggestion on how to improve our git workflow and make it more appealing for users to manage?
- Do you have any idea or suggestion on different approach?

Finish the interview: Ask the participants for any final comments. Let them highlight any issues that was not considered by the interviewees or if the find issues particular important which was not shown during the interview.

B

Figures

```
33    FUNC name F_CalcDay type Integer PARAM name ArgInt_Day type Integer
34    /*****************************************************************************
35     * Descriptions: Day conversion: input 1-7(Mo-Su) to model-specific 1-7(Sa-Fr)
36      * Private variables
37        -
38      * Parameters
39        - ArgInt_Day: 1-7 (Monday-Sunday)
40
41     * Example:      Fi_CalcDay(6) = 1
42     *****************************************************************************/
43    begin F_CalcDay function
44        if (ArgInt_Day = 6) then return 1
45        else if (ArgInt_Day = 7) then return 2
46        else if (ArgInt_Day = 1) then return 3
47        else if (ArgInt_Day = 2) then return 4
48        else if (ArgInt_Day = 3) then return 5
49        else if (ArgInt_Day = 4) then return 6
50        else if (ArgInt_Day = 5) then return 7
51        return 0
52    end
```

Figure B.1: Example of final process specific snippet, function F_CalcDay. Every subject field is filled except for Private variables, which is left blank

```
2    FUNC name F_CalcTime type Integer PARAM name ArgReal_Time type Real
3    /*************************************************************************
4     * Descriptions: Excel format conversion 0.284(General) -> 650(Digital)
5                       (06:50:00 AM is 0.2847222 in General format in Excel)
6      * Private variables
7          - Vi_paramSec
8          - Vi_paramMin
9          - Vi_paramHour
10     * Parameters:
11         - ArgReal_Time: 0-1 (excel time in general format)
12
13    * Example:      Fi_CalcTime(0.2847222) = 650
14                    1. 0.2847222 * 24*60*60 = 24600
15                    2. check for integer division
16                    3. 410 = 24600/60 (minutes)
17                    4. 6 (6.8333) = 410 / 60 (hour)
18                    5. 50 = 410 - 6 * 60 = 410 - 360
19                    6. 650
20    *************************************************************************/
21   begin F_CalcTime function
22       set Vi_paramSec = ArgReal_Time * 24*60*60
23       if ((Vi_paramSec % 60) <> 0)
24           then inc Vi_paramSec by (60 - (Vi_paramSec % 60.0))
25       set Vi_paramMin = Vi_paramSec / 60
26       set Vi_paramHour = Vi_paramMin / 60
27       set Vi_paramMin = Vi_paramMin - Vi_paramHour * 60
28       return Vi_paramHour*100 + Vi_paramMin
29   end
```

Figure B.2: Example of final process specific snippet, function F_CalcTime. Every subject field filled