Towards Higher Code Quality in Scientific Computing

MALIN KÄLLÉN
Abstract

In scientific computing and data science, computer programs employing mathematical and statistical models are used for obtaining knowledge in different application domains. The results of these programs form the basis of among other things scientific papers and important decisions that may e.g. affect people's health. Consequently, correctness of the programs is of great importance. To reduce the risk of defects in the source code, and to not waste human resources, it is important that the code is maintainable, i.e. not unnecessarily hard to analyze, test, modify or reuse. For these reasons, this thesis strives towards increased maintainability and correctness in code bases for scientific computing and data science.

Object-oriented programming is a programming paradigm that facilitates writing maintainable code, by providing mechanisms for reuse and for division of code into smaller components with restricted access to each others data. Further, it makes extending a code base without changing the existing code possible, increasing flexibility and decreasing the risk of breaking existing functionality. However, in many cases, object-orientation trades its benefits for performance. For some scientific computing programs, performance is essential, e.g. because the results are unusable if they are produced too late. In the first part of this thesis, it is shown that object-oriented programming can be used to improve the quality of an important group of scientific computing programs, with only a small impact on performance.

The aim of the second part of the thesis is to contribute to understanding of, and improve quality in, source code for data science. A large corpus of Jupyter notebooks, a tool frequently used by data scientists for writing code, is studied. Results presented suggest that cloned code, i.e. identical or close to identical code that recurs in different places, is common in Jupyter notebooks. Code cloning is important from a perspective of maintenance as well as for research. Additionally, the most frequently called library functions from Python, the language used in the vast majority of the notebooks, are studied. A large number of combinations of parameters for which it is possible to pass values that may lead to unexpected behavior or decreased maintainability are identified. The existence and consequences of occurrences of such combinations of values in the corpus are evaluated. To reduce the risk of future defects in source code calling these functions, improvements are suggested to the developers of the functions.

Malin Källén, Department of Information Technology, Division of Scientific Computing, Box 337, Uppsala University, SE-751 05 Uppsala, Sweden. Department of Information Technology, Computational Science, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

© Malin Källén 2021

ISSN 1651-6214
ISBN 978-91-513-1107-4
urn:nbn:se:uu:diva-430134 (http://urn.kb.se/resolve?urn=nbn:se:uu:diva-430134)
List of papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I Malin Källén, Sverker Holmgren, and Ebba Þ. Hvannberg. Impact of Code Refactoring using Object-Oriented Methodology on a Scientific Computing Application. Technical report 2020-004, Department of Information Technology, Uppsala University, 2020. This is an extended version of a peer-reviewed paper with the same name published at IEEE International Workshop on Source Code Analysis and Manipulation 2014 [1].


IV Malin Källén, Tobias Wrigstad. To Err or Not to Err?: Subtle Interactions Between Parameters for Common Python Library Functions. Submitted to *The Art, Science, and Engineering of Programming*.

Reprints were made with permission from the publishers.
## Contents

1 Introduction ................................................................. 7  
  1.1 Scientific Computing .................................................. 7  
  1.2 Software Engineering .................................................. 8  
  1.3 Applying Knowledge from Software Engineering in Scientific Computing .................................................. 9  
  1.4 Contributions of this Thesis ........................................... 9  
      1.4.1 Outline .......................................................... 10  

2 Background ................................................................. 11  
  2.1 Maintainability ......................................................... 11  
  2.2 Object-Oriented Programming ........................................ 11  
      2.2.1 Encapsulation .................................................... 12  
      2.2.2 Inheritance ..................................................... 12  
      2.2.3 Polymorphism ................................................... 12  
  2.3 Object-Oriented Design ............................................... 13  
      2.3.1 The SOLID Principles .......................................... 13  
      2.3.2 Design Patterns ............................................... 14  
  2.4 Refactoring ............................................................ 16  
      2.4.1 Code Smells .................................................... 16  
  2.5 Software Metrics ...................................................... 17  
  2.6 C++ and Java .......................................................... 18  
      2.6.1 The Java Virtual Machine ..................................... 18  
      2.6.2 Object-Orientation ............................................. 19  
  2.7 Parallel Programming ................................................ 21  
      2.7.1 Shared Memory Parallelization ................................ 21  
      2.7.2 Distributed Memory Parallelization ......................... 22  
      2.7.3 Hybrid Parallelization ....................................... 22  
  2.8 Python ................................................................. 22  
  2.9 Jupyter Notebooks ..................................................... 23  
  2.10 Code Cloning .......................................................... 23  
  2.11 Statistical Methods .................................................. 24  
      2.11.1 Basic Concepts ............................................... 24  
      2.11.2 Errors ......................................................... 25  
      2.11.3 Statistical Power .............................................. 25  
      2.11.4 Different Types of Statistical Tests ....................... 25  
      2.11.5 Parametric Tests .............................................. 26  
      2.11.6 Non-Parametric Tests ....................................... 27
2.11.7 Adjustment of p values or Significance Levels ........... 27
2.11.8 Correlation Tests .............................................. 27

3 Towards Higher Code Quality in Scientific Computing ............. 29
  3.1 Software Engineering in Scientific Computing .................. 30
    3.1.1 History and Present State .................................. 31
    3.1.2 Scientific Computing Software .............................. 31
  3.2 Maintainability of Code for Scientific Computing ............... 32
    3.2.1 The Importance of Maintainability ......................... 33
    3.2.2 The Importance of Performance ............................. 34
    3.2.3 Object-Orientation as a Means to Code Quality ............ 34
    3.2.4 A Potential Problem of Object-Orientation ................. 35
    3.2.5 Work Presented in this Thesis ............................. 35
  3.3 Jupyter Notebooks ............................................... 36
    3.3.1 Code Cloning ............................................... 36
    3.3.2 Error Suppression in Python ................................ 37
    3.3.3 Work Presented in this Thesis ............................. 38
  3.4 Summary of Papers .............................................. 39
    3.4.1 Paper I ..................................................... 39
    3.4.2 Paper II ..................................................... 40
    3.4.3 Paper III ................................................... 41
    3.4.4 Paper IV .................................................... 42
  3.5 Concluding Remark ............................................... 42

4 Summary in Swedish ................................................ 44
  4.1 Objektorienterad programmering ................................ 45
  4.2 Jupyter notebooks .............................................. 46
  4.3 Slutnot ............................................................ 48

References ............................................................... 50
1. Introduction

This thesis touches upon both scientific computing and software engineering. In the first chapter, these subjects are introduced and the contributions of the thesis are briefly presented.

1.1 Scientific Computing

Scientific computing is a subject where we perform computer simulations to study processes that would be dangerous, unethical, expensive or even impossible to reconstruct in physical experiments. Examples are car crashes, earthquakes, and processes in the body of a patient put on a ventilator. The latter example is of immediate interest with the current COVID-19 pandemic. Scientific computing is a wide subject, involving setting up models, developing numerical methods, and implement and test the code.

A subject that has many similarities with, and partly overlaps with, scientific computing is data science. Data science involves programmatically analyzing large amounts of data to extract knowledge from it. It has gained a lot in attention and practice the last couple of years.

Both subjects involve writing computer programs that apply mathematical and/or statistical methods, often in an application domain different than math and statistics. The application domain can for example be physics, medicine or geology. Many code bases for scientific computing are designed to be run once or twice [3]. Similarly, an analysis of a sample of Jupyter notebooks, a tool often used by data scientists to write code, suggests that notebooks are generally not further developed once they are written and committed to a code repository the first time. The analysis is further described in Section 3.3.3. This suggests that many code bases written for projects in scientific computing and data analysis are mainly used within the project that they are developed for. In other words, they are not further developed to be used in subsequent projects, building on the first one.

This thesis focuses on some aspects of the computer programs mentioned above. The circumstances in which these programs are written are in many ways common across the two subjects. Therefore, and for the sake of a more concise discussion, hereinafter the term “scientific computing” is used to refer to both subjects. When it is important to distinguish between data science and the computer simulations part, it is explicitly stated which one is discussed.
In data science, programs are often constructed using high-level languages such as Python and R. In addition to having a comparatively low learning threshold, these languages contain a wide range of functions that are useful for data scientists, e.g. different plotting and statistics functions. On the other hand, these languages have little focus on factors that increase the performance of a program, such as efficient data representation and memory management.

High-level languages are not only used in data science, but for some of the previously mentioned computer simulations as well. However, performing a simulation requires lots of computations, and for large simulations, performance is an important aspect. With more and more powerful computers, we can run faster or larger simulations. Still, for some programs, the execution times are counted in days or weeks, even when run on large clusters of computers. When we work with computations of this size, we talk about high performance computing.

As could be expected, execution time is an important aspect in high performance computing, both to speed up the workflow of researchers, and to reduce the electricity cost, which is a substantial part of the expenses for a compute center. Consequently, some researchers put a large effort in optimizing their programs before running them. Optimizing the programs involves tweaking details in the code and the researcher needs to have detailed knowledge about the programming language used. This programming language is typically a compiled language where you have control over low-level details, for example C or C++.

1.2 Software Engineering

While, in scientific computing, software is often seen as a means to other ends [4], software engineering is a subject that targets software more directly. To cite Ian Sommerville, it is “an engineering discipline that is concerned with all aspects of software production” [5]. This includes requirements engineering, design, implementation, testing, documentation and maintenance of software. Software is treated as an artifact that will be used in different contexts and has a long lifespan. This view makes software engineers focus more on software quality. For instance, the software should be reliable and secure, and it is important not to make maintenance, testing and further development of it unnecessarily hard. Resources for software development are often limited, and productivity is another important aspect, not only in the maintenance phase, but also in the development phase. Consequently, methods for efficiently producing high-quality software are studied and developed in software engineering.
1.3 Applying Knowledge from Software Engineering in Scientific Computing

The interdisciplinary nature of scientific computing demands a wide range of competences of the practitioners. Often, it is necessary to have knowledge in math/statistics, computer program construction (which also covers software engineering) and an application domain. Naturally, it is hard to be an expert in all of these areas, which may explain why earlier research [6] suggests that many practitioners of scientific computing have limited knowledge about many software engineering practices. Likewise, the study suggests that the usage of these practices is low in scientific computing.

This thesis argues that in scientific computing, we could benefit considerably from applying knowledge from software engineering, not least about software quality. Imagine, for example, that we put more effort in making the code readable, organized it into reusable units, and facilitated testing – i.e. we increased its maintainability. This would make it a lot easier to come back to the code after a period of time and extend it to support experiments for a new study, instead of writing a new program from scratch. Not having to rewrite the software, we could expect saving time, and the time set free could be spent performing new research instead of reinventing the wheel. More maintainable code could also open up for other researchers to reuse the code, which would increase the impact of its developer(s).

Moreover, this thesis argues that code is likely to have fewer defects if it is maintainable and/or reused, see Section 3.2.1. Fewer defects in the code would make results based on the output of the code more reliable.

1.4 Contributions of this Thesis

This thesis strives towards the goal of applying more knowledge from software engineering in scientific computing, as argued for in Section 1.3. However, as explained in Sections 1.1 and 1.2, both scientific computing and software engineering are broad subjects. Therefore, it is impossible to cover more than a fraction of them in a single thesis. In this thesis, a subset of the two subjects are dealt with as described below.

First, as a step towards more maintainable programs in high performance computing, the possibility to apply object-oriented programming in this field is explored. As argued in Section 2.2, object-oriented programming is a technique that may be used to create modular and flexible code. However, in many cases it trades the modularity and flexibility for performance. The conflict between maintainability and performance is discussed, as well as two possible solutions.

Next, the focus is on Jupyter notebooks, a tool frequently used for writing high-level scientific computing code. As a basis for future research, basic
characteristics of, as well as the amount of code cloning in, a large number of notebooks are studied. Cloned code refers to identical or close to identical snippets of code that can be found in more than one place. Code cloning is important to be aware of and take into consideration when performing studies on source code, in order to reduce bias in the studies. This holds if the research targets software quality, as well as other aspects of the code.

Last, a number of Python functions frequently used in the notebook corpus are discussed. Combinations of argument values that indicate defects in the calling code, or errors in input data, but that do not lead to a warning or an error are identified. Actual occurrences of such argument combinations are found in the notebooks corpus. A number of issues are filed in the GitHub repositories of the modules where the functions in question reside, arguing that the argument combinations described above should induce an error or a warning.

1.4.1 Outline

The thesis is arranged as follows: In Chapter 2, important concepts discussed in the thesis are explained. Chapter 3 contains a summary of the work presented. Chapter 4 presents a brief summary in Swedish. Last, the papers presenting the scientific work are included.
2. Background

In this chapter, a number of concepts discussed in the thesis are introduced. Some understanding of these concepts is necessary in order to read and appreciate Chapter 3 and the scientific papers included in the thesis.

2.1 Maintainability

In Section 1.3, it is argued that scientific computing would benefit from more maintainable code and some examples of maintainability increasing actions are given. In this section, maintainability is presented a little bit more comprehensively. Maintainability is defined in an ISO and IEC standard: According to ISO/IEC 25010 [7], this characteristic represents

the degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers

where modifications can include

- corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications.

The standard defines five sub-characteristics of maintainability: modularity, reusability, analysability, modifiability and testability. A modular system is one that can be divided into parts that can be changed with minimal impact on other parts. Hence, before we say that our code is maintainable, it should be split in smaller components that can be changed with minimal impact on each other (modular). Moreover, we should be able to reuse the code, or parts of it, in more than one system. Furthermore, it should not be too hard to analyze deficiencies, the effect of changes of the code, or the origin of failures. Likewise, when we modify the code, we should not run a high risk of degrading its quality or introducing defects. Last, test criteria should not be unnecessarily hard to identify and implement.

2.2 Object-Oriented Programming

This section introduces the concept of object-oriented programming. The topic is also discussed in Section 3.2.
As explained below, object-oriented programming can be used to write flexible (modifiable) and modular code, and it provides a mechanism for reuse. In an object-oriented program, the code is divided into different entities referred to as classes. Run-time instantiations of the classes are called objects. Often, classes represent concepts from the application domain of a program, but may also act e.g. as abstractions of concepts from the programming domain.

2.2.1 Encapsulation
An object has a state, represented by values of a number of attributes (or, to be more concrete, variables). In a well-designed object-oriented code base, each object is a well-defined unit which keeps track of its own state, and operations on the state are made by the object itself. The object interacts with other objects through interfaces that do not expose low-level details, e.g. of the state of the other objects. This is known as encapsulation, which is a central concept in object-oriented programming. It increases the modularity of, and thereby the ease of analyzing and modifying, a code base. When objects do not operate on each others state, it is possible to modify the code of a class without having to worry about unexpected side effects on the state of objects of other types. Likewise, analyzing the behavior of a class is much easier when we know that all changes of its state is made by the class itself. Encapsulation has been a target of many scientific studies [8].

2.2.2 Inheritance
Another central concept in object-oriented programming is inheritance. Inheritance is a mechanism that makes it possible to create new (typically more specialized) classes based on existing ones, which facilitates reuse of code. If a class \( D \) inherits from another class, \( B \), public operations and attributes of \( B \) exist in \( D \) as well. In many object-oriented languages it is also possible to specify that an operation or attribute should be accessible by the class itself and its derived classes only. The class denoted \( B \) above and the class denoted \( D \) above are called base class and derived class respectively.\(^1\) Classes that can be instantiated are called concrete classes. There are also abstract classes, which can not be instantiated. Abstract classes often serve as a base for several derived classes and contain aspects that are common across the derived classes. They may also serve as polymorphic interfaces as described next.

2.2.3 Polymorphism
In a language supporting polymorphism, you can specify one interface (e.g. an abstract class) for accessing an object in your program and let it have sever-

\(^1\)They may also be referred to as super class and sub class, or parent class and child class.
eral different implementations. Below, non-abstract implementations of the interface are referred to as concretizations. Thanks to the common interface, it is possible to interact with all concretizations in a consistent way, not even knowing until run-time which actual concretization is behind the interface. This brings an opportunity to write very flexible programs, see further Section 2.3.

Static Polymorphism
The fact that the actual concretization is not known in compile-time prevents the compiler from making a number of optimizations. This may be mitigated using static polymorphism, that is keeping the interfaces but also provide compile-time information about the concretization. In this work, static polymorphism is implemented using template parameters in C++.

2.3 Object-Oriented Design
Two well-known tools for designing object-oriented programs are the SOLID principles and the notion of design patterns, both of which are presented in this section.

2.3.1 The SOLID Principles
The SOLID principles [10] were assembled by Martin [11]. The name consists of the initial letters in the names of the principles, which are:
- Single-Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface-Segregation Principle (ISP)
- Dependency-Inversion Principle (DIP)

These are described and motivated below.

The Single-Responsibility Principle
The SRP tells us that a class should have only one responsibility, which in this case means one reason to change. If a class is affected by different types of changes, it is time to split it into smaller ones. Otherwise, making a change of one type may break functionality connected to the other responsibility. Moreover, too many responsibilities on a single class may cause unnecessary dependencies.

The Open-Closed Principle
The OCP states that a code entity should be open for extension but closed for modification. In other words, it should be possible to add functionality without

\[^2\text{Note that this is the case also when the polymorphism is replaced by conditionals [9].}\]
changing the existing code, since changing code brings a risk to break something that is currently working. In practice, this can be obtained by writing a class in a way that makes it possible to extend its functionality by simply adding new sub classes to the class itself or to interfaces that it uses.

The Liskov Substitution Principle
The LSP, which is based on work by Liskov [12], tells us that every sub type should be substitutable for its base type. In an object-oriented context, this means that whenever a variable is declared as a certain class, C, it should be possible to initialize it with an instantiation of any subclass of C. Not following this principle would deteriorate the analyzability of the code.

The Interface-Segregation Principle
According to the ISP, interfaces should be small and well-defined. Interface in this context means the interface through which you use a class, typically an abstract, possibly indirect, base class. Interfaces that are too heavy should be split into smaller ones, in order not to force other code entities to depend on, or implement, code that is not needed.

The Dependency-Inversion Principle
Last, a commonly referred but somewhat simplified version of the DIP states that we should depend on abstractions rather than concretizations. When a code entity depends on an abstraction instead of a concrete class, you may change implementation details in the concrete class, and add new concretizations, without affecting the dependent code entity.

2.3.2 Design Patterns
The notion of design patterns for object-oriented software was introduced by Gamma et al. [13], commonly referred to as “The Gang of Four”. A design pattern is a recurring solution of a design problem. The inspiration comes from the architect Christopher Alexander, who has cataloged patterns in the construction of buildings and cities. In the book by Gamma et al., 23 different patterns for object-oriented software design are documented. However, there are many more software design patterns than these. More than two decades after the release of the book, design patterns is still an active area of research, and there is an entire scientific conference, “PLoP” (“Pattern Languages of Programming”) [14] devoted to this.

Below, seven of the patterns described by Gamma et al., are discussed. They are all used to increase the maintainability of the code discussed in Paper I.

Template Method
Assume that you have an algorithm with a number of sub steps and there are different ways to perform the sub steps. Then you may write a method that ex-
ecutes this algorithm, but it defers the varying sub steps to the sub classes. (If there are invariant sub steps of the algorithm, these should naturally enough be implemented in the base class.) This is known as the Template Method pattern. Using it, you can vary the algorithm without having to duplicate the common parts, and you may add new versions of the algorithm without changing the existing code.

**Strategy**

An alternative way to vary an algorithm, or parts of it, is by applying the Strategy pattern: Provide an interface for the algorithm and use different implementations of the interface for the different alternatives. Then you can make the code that uses the algorithm independent of the concrete classes and you may easily change the behavior without affecting the caller. An advantage to Strategies over Template Methods is that you can combine different sub steps without having to write a new class for each combination. On the other hand, using a Strategy when there is only one sub step that will vary can result in a large number of classes, which may bring needless complexity.

**Factory Method**

A design pattern that is often used together with Template Method is the Factory Method pattern. Here, the base class provides a method for creating objects that it (or the code using it) needs, but the implementation of the method is deferred to its sub classes. Then, the base class only depends on the abstract type of the objects, and the concrete types used may be varied between sub classes.

**Abstract Factory**

Now, assume that you have a whole family of objects that are created and used together. In order to be able to use different families, you may encapsulate the creation of the objects in a number of different classes, and provide a common interface for them. Then, the calling code only needs to know about the interface, and you may vary which family is used by simply plugging in different classes.

**Decorator**

If you have a set of classes with a common interface, I, and you want to be able to extend the functionality of the objects dynamically, you may want to use the Decorator pattern: Create a new implementation of I, a decorator, which holds a reference to an instance of I. When an operation is called on the decorator, it makes sure that the corresponding operation is performed by the just mentioned I instance, and it adds the behavior specific to itself. This lets you add the same functionality to different implementations of I without having to create a new sub class for each implementation. It also makes it easy to extend an object with several different behaviors.
**Composite**
If you want to organize objects in a tree structure, it may be wise to apply the Composite pattern: If the leaf objects and the interior node objects are different classes, you can still provide a common interface for them. Then, the interior node objects can treat its children the same way regardless of whether they are leafs or not. Likewise, other code using a tree does not have to bother whether the root node is a leaf or not. This makes it possible to write the code in a more concise way.

**Iterator**
Suppose that you have a composed data type and you want to be able to traverse the elements in different order, or you want to separate iteration from other logic to adhere better to the single responsibility principle. Then, you may make use of the Iterator pattern, where you define a separate class, an iterator, which is used to traverse the data. By writing different iterator implementations, you can easily alter the traversal scheme.

### 2.4 Refactoring
Often, the design of a program does not get perfect from the beginning, and even if it does, when the program evolves, the structure will most likely gradually drift away from the original design. Hence, eventually, a need for refactoring can be expected to appear. Refactoring means restructuring program code in a way that improves its internal structure without changing its behavior. A more detailed discussion of refactoring can be found in Paper I.

#### 2.4.1 Code Smells
As an instrument for identifying code units that could benefit from refactoring, Fowler [15] introduces the notion of code smells: patterns in (object-oriented) source code that suggest that the code structure may be improved by refactoring. Three such smells, which are all used in Paper I, are described below.

**Shotgun Surgery**
If changing one thing in a code base forces you to make changes in several different classes, your code smells of shotgun surgery. It is a sign that it is time to merge the functionality of the different classes into one or several new ones. A well-defined update of the code should ideally only result in one class being changed.

**Divergent Change**
Divergent change can be seen as the opposite of shotgun surgery. Here, changes that are unrelated to each other result in updates of the same class. This breaks
the Single-Responsibility Principle, and the smell indicates that the class in question should be split into two or more different classes, each which is changed for one reason.

**Inappropriate Intimacy**

When a class, A, is too interested in the private parts of another class, B, that is A is operating on the instance variables of B, your code smells of inappropriate intimacy. Put differently, you are violating the encapsulation. In a well-designed object-oriented code base, each class should operate on its own attributes. If not, it is time to move either methods of A into B, or attributes of B into A.

### 2.5 Software Metrics

Having refactored a code base, we might want to evaluate the effect quantitatively. Since refactoring aims at improving the internal structure of the code, we somehow have to judge whether the new structure is better than the old one. Better is a subjective attribute that has lots of aspects, but presumably everybody in the field would agree that the code structure is better if it is more maintainable.

Which characteristics make a code base maintainable is debatable, but it is hard to argue against less complex code being easier to analyze and modify. Depending on the type of reuse and the testing methodology, the less complex code may also be more reusable and testable. Hence, if we can show that our code base is less complex, we can argue that it is more maintainable. Furthermore, if the new code base is smaller, there is less code that needs to be maintained, which should facilitate maintainability.

Certainly, there are more aspects of maintainability than size and complexity, of which some are not measurable at all. For example, relevant naming of functions and variables is essential for readability, and thereby analyzability and modifiability, but objectively measuring the adequacy of names is not possible. Hence, despite giving some indication of the level of maintainability, measures of size and complexity of a code base need to be combined with a qualitative analysis in order to give a more complete picture.

Accordingly, in this thesis, maintainability is evaluated both quantitatively and qualitatively. For the quantitative analysis, measures of size and complexity are used. As argued above, these reflect maintainability to some extent. When it comes to size of source code, the most commonly used metric is *lines of code* [16], which can be measured on different levels (such as entire code base, class or function).

A metrics suite specifically designed to measure the complexity of object-oriented code is developed by Chidamber and Kemerer [17]. It is extensively
used in the literature [18]. The suite contains the following metrics, which are all measured at class level:

- **Depth of Inheritance Tree (DIT)** measures the longest chain of ancestor classes (parent, parent’s parent and so on) of a class.
- **Number of Children (NOC)** measures the number of direct sub classes of a class.
- **Coupling Between Objects (CBO)** measures the number of connections between a class and other classes.
- **Lack of Cohesion in Methods (LCOM)** measures how well different parts of a class relate to each other.
- **Weighted Methods per Class (WMC)** is the sum of the complexity of the methods in a class, where the choice of definition of complexity is left to the user of the metric.
- **Response For a Class (RFC)** is the number of methods that can potentially be invoked as a result of a call to a method in the class.

Exact definitions of the metrics can be found in the paper by Chidamber and Kemerer [17].

### 2.6 C++ and Java

Two popular languages for object-oriented programming are C++ and Java. C++ is built upon the procedural language C, while Java is designed to be a purely object-oriented language. The lion’s share of the code that the results presented in this thesis rely on is written in C++ and Java. Moreover, a number of aspects of the two languages are compared in Paper II.

#### 2.6.1 The Java Virtual Machine

C++ and Java are both statically typed and have a similar syntax, but they differ in an important aspect: C++ is a compiled language that runs directly on the operating system. On the contrary, most modern Java compilers translate code into object code which is run on a virtual machine, the JVM (Java Virtual Machine). Certainly, the usage of a virtual machine brings some overhead when it comes to memory usage as well as program execution. However, the JVM comes with a number of benefits. Three of them, which are also discussed in Paper II, are explained next.

First, Java performs checks of array bounds at run-time: In C++, if you allocate memory for an array and populate it with data whose size exceeds the allocated memory, you will overwrite other data, which may lead to defects that are very hard to debug. On the contrary, in such a case Java will throw

---

3 As of November 5, 2020, the paper presenting this suite has 7180 citations.
an ArrayIndexOutOfBoundsException, clearly telling you which mistake you have made and where.

Second, the virtual machine provides automatic memory management: In C++, memory needs to be explicitly allocated and deallocated in many cases, which Java handles automatically under the hood. In particular, Java has a feature called garbage collection, which means that the JVM keeps track of references to all objects and automatically deallocates those that are no longer referenced. While bringing some overhead, this facilitates programming and almost eliminates the risk for memory leaks.

Third, the JVM contains a just-in-time compiler, often referred to as a JIT compiler. The JIT compiler compiles the most frequently executed parts of the program (“the hot code”) into more optimized machine code, possibly in several steps. There is a great variety of optimizations that can be made by a JIT compiler. An example is inlining of methods, which eliminates method call overhead and, more importantly, enables the compiler to optimize bigger chunks of code. This can also be done by the compiler of pre-compiled language like C++, if the methods are statically bound. When a method is statically bound, it is known at compile time which method is referred to by a specific method call. The opposite is a dynamically bound method. Methods in the interfaces discussed in Sections 2.2 and 2.3 are dynamically bound and can not be inlined by a C++ compiler\(^4\). However, the JIT compilation is performed at run-time, which means that it can use run-time information, such as the concrete types hidden behind interfaces, to optimize the code. Hence, the JIT compiler can inline both statically and dynamically bound method calls.

Later on, this thesis will discuss the time needed for warm-up of a Java program. This is the time it takes for the program to reach steady state, that is a state where two subsequent executions of the same part of the code take equally long time.

2.6.2 Object-Orientation

In addition to classes, C++ supports a data structure named “struct”. C++-structs (as opposed to structs in C) resemble classes, and can be used more or less in the same way as such. Structs are not present in Java. In addition to this, the support for object-oriented programming in C++ and Java has other differences that affect the work presented in this thesis.

First, the methods in a Java class are dynamically bound by default, and you have to explicitly state when a method should be statically bound. On the contrary, in C++ methods are statically bound by default, and dynamically bound methods must be explicitly defined as such.

\(^4\)unless there is only one concretization of the interface, but then the usage of an interface may be questioned
Second, in C++, a class can inherit from several other classes. Here, the interfaces discussed in Sections 2.2 and 2.3 are implemented using abstract classes. In Java, a class can only inherit from one other class. However, Java has an additional concept called interfaces, of which a class can “inherit from” more than one. Therefore, both abstract classes and Java interfaces are used as object-oriented interfaces in Java. Since Java 8, the difference between an interface and an abstract class is small. Explaining the difference would require this thesis to get too much into technical details, but the interested reader can refer to Oracle’s Java interface tutorial [19].

The fact that a C++ class can inherit from multiple classes calls for another feature of the language, namely virtual inheritance. Inheritance in C++ is implemented such that the sub class holds a reference to an instantiation of the base class. Now, consider the case illustrated in Figure 2.1. Here, the class D inherits from the classes B and C, which both inherit from the class A. This means that each of B and C keeps a reference to an instance of A. In the default case, when D accesses instance variables of A, it is impossible to know which instance should be used, the one kept in B or the one kept in C. Therefore, this is not allowed. If, instead, B and C inherit virtually from A, only one instance of A is created, and both B and C point to that instance. Thus, when virtual

---

5The actual term is implement.
Inheritance is used, D can access A:s instance variables in an unambiguous way.

Last, both languages have support for class templates, but with an important difference. A class template resembles a class, but takes one or several template parameters, which specify one or several properties of the class. A property may e.g. be the type of an instance variable. Using the template, one may create objects that are all of the type of the template, but have different values of the template parameter. The important difference between the two languages is the types allowed for the template parameters. In Java you may only use types as template parameters. On the contrary, a C++ class template parameter may also represent a value, e.g. of an integer.

2.7 Parallel Programming

As explained in Section 1.1, some of the scientific computing programs written in C++ have very high demands for performance. In extreme cases, an execution can go on for days or weeks. Programs with such execution times are typically run on large clusters of computers. In order to make full use of the computer capabilities of these clusters, one needs to parallelize the programs. Also, when running e.g. on an ordinary work station, you might want to parallelize your program, in order to make use of both/all cores of the computer. In this thesis, two types of parallelization are used: shared memory parallelization and distributed memory parallelization.

2.7.1 Shared Memory Parallelization

First, focus is on parallelization of programs that are run on processing units (typically processor cores) having a common memory area. Here, computations are divided among threads that are run on the different processing units. A library often used for shared memory parallelization of C and C++ programs is OpenMP [20]. Using OpenMP, you can annotate sections in the code that should be run in parallel, and the work done in each annotated section is split between different threads.

There is a Java version of OpenMP called OMP4J [21], but it is a student project that lacks much of the functionality of OpenMP. At the time of writing (October 29, 2020), only three commits have been made to the GitHub repository [22] since the end of 2015. The standard Java way to parallelize a program for a shared memory environment is to define tasks, chunks of work to be done, that are assigned to different threads.
2.7.2 Distributed Memory Parallelization

Next, parallelization of programs that are run on processing units (typically processors) of which each has its own memory space is described. As in the shared memory parallelization case, the work done by the program is divided, and each share is handled in a separate process. The processes are run on the different processing units. All values that have been computed by one process and are needed by another must be explicitly communicated.

A commonly used standard for the inter-process communication in C++ is MPI (“Message Passing Interface”) [23]. The standard contains functions for sending and receiving data. Send operations as well as receive operations can be blocking or non-blocking. Using the latter, a process may initialize communication of a chunk of data, and do some other work while waiting for it to complete (as long as it does not use data being received or update data being sent). Then we say that communication is overlapped with computations. Certainly, when large data chunks are sent, overlapping communication with computations may speed the execution up.

MPI is most commonly used for C, C++ and Fortran code. However, the MPI implementation named OpenMPI [24] also has support for Java code.

2.7.3 Hybrid Parallelization

Assuredly, it is possible to combine shared memory parallelization with distributed memory parallelization. Then the work to be performed by the program is divided between different processes running on different processing units with distributed memory. The work performed by each process, in turn, is divided between threads that are executed on processing units sharing memory.

2.8 Python

In addition to C++ and Java, a language that is used, and in particular studied, in this thesis is Python. Python is a high-level, interpreted and dynamically typed language. According to the TIOBE Index [25], Python is one of the most popular languages in the world, and its popularity is rapidly growing. There is an extensive number of modules containing third-party code available for Python. Three commonly used such modules, which are discussed in this thesis, are numpy, pandas and matplotlib.pyplot. The dot in the latter name means that pyplot is a submodule of matplotlib.

The module numpy provides support for multi-dimensional arrays and high-level mathematical operations, while matplotlib.pyplot implements plotting

---

6It is also possible to use this type of parallelization for processing units that share memory. Then the memory is split between the units and they can not access each others memory area.
functionality similar to that of MATLAB. Through pandas, we have support for reading, manipulating and analyzing (possibly heterogeneous) data sets. For performance reasons, parts of numpy and pandas are implemented in C, but all functions are called via a Python interface.

2.9 Jupyter Notebooks

Python is commonly used in Jupyter notebooks. A Jupyter notebook [26] is a file that can contain both text on markdown format, and executable code. It is a tool commonly used by data scientists.

The text and the code of a notebook are stored in a sequence of cells, where each cell is either a text cell or a code cell. Each code cell is executable in isolation, but they all share the same program state. In addition to the code itself, an executed code cell may contain the results of its last execution. The result can consist of both text and visualizations.

A Jupyter notebook is stored in JSON format, and can be accessed, manipulated and run through a web interface where the cells are rendered in a user-friendly way. The notebooks are rendered in a similar way when uploaded on and viewed through the popular online code repository GitHub.

When executed in the web interface, a code cell is executed by a kernel, which governs which programming language can be used. Originally, there were Jupyter notebook kernels supporting Julia, Python and R. (The name is a fusion of the names of these three languages.) However, today, there are kernels for over 40 different programming languages.

Jupyter notebooks are also described in Papers III and IV.

2.10 Code Cloning

As will be seen in Paper III, code cloning is common in Jupyter notebooks. A code segment (e.g. a function, a file or a block of code) is a code clone if it is identical, or nearly identical, to another code segment of the same type. Different studies deal with clones on different levels, where the level indicates the extent of the cloned segments. For example, in a study of clones at function level, functions that are clones of each other are studied. In this thesis, clones are studied at the level of code cells in Jupyter notebooks.

The following classification of clones, based on their level of similarity, is frequently used in the literature [27, 28, 29, 30, 31, 32, 33, 34]:

- Type 1 clones are identical except possible differences in comments and whitespaces.
- Type 2 clones may, except from the differences allowed for type 1 clones, also differ in literals, types and identifiers.
• Between type 3 clones, entire statements may differ as long as most of the code is the same. These may also be referred to as “near-miss clones”.

This classification is also used in this work, but for practical reasons, another clone type is introduced, referred to as CMW clones. CMW is an abbreviation of “Copy Modulo White space”. These clones may only differ in whitespaces, which means that the CMW clones of a code base are a subset of its Type 1 clones. Code clones are further discussed in Section 3.3.1 and in Paper III.

2.11 Statistical Methods

Assume that we have collected quantitative measures of a characteristic (such as software metrics or clone frequencies) of different objects (such as versions of a code base) or groups of objects (such as notebooks written in a particular language). Almost certainly, we will get different values for the different objects/groups. We may make several measurements and compute the mean value for each object or group, but still the probability is that we get different values for them. It may be hard to tell whether the difference is due to random variations or if it represents a real difference in the characteristic. Here, statistical methods stand in good stead, since they can indicate whether the differences are just random fluctuation or not.

In this section, statistical methods that are used in this thesis are briefly described. The aim is not to give a complete description, but mainly the aspects relevant for the thesis are discussed. Usages of the methods that are not applied here are left out. For more details on the methods presented, see e.g. Quinn & Keough [35].

2.11.1 Basic Concepts

In this section, the (groups of) objects mentioned above will be referred to as “study objects”. In statistical terms, all possible values of a characteristic of a study object is the population for that characteristic. The collection of measurements that we actually collect is called a sample.

Applying a statistical test on our data, we get a value, p, which represents the probability that the differences we see are only due to random variations. If p is below a prechosen value $\alpha$, we say that the difference is statistically significant, or simply “significant” on the level of $\alpha$, and conclude that there is an actual difference between the study objects. It is common practice to set $\alpha$ to 0.05, 0.01 or 0.001.
2.11.2 Errors
Concluding that there is a significant difference when the differences are only random fluctuations is called a Type I error. The risk of getting a Type I error is the prechosen value $\alpha$ described above. The opposite, not finding a significant difference when there is an actual difference between the study objects, is called a Type II error. Often, the risk of a Type II error is much higher than that of a Type I error. Consequently, when we do not find a significant difference, we refrain from drawing any conclusions.

2.11.3 Statistical Power
The probability of actually finding a significant difference, if it exists, is called the statistical power, or simply “power”, of a test. The power increases with
- the size of the difference in the characteristic
- the level of $\alpha$
- the sample size, that is the number of measures we make
while it decreases with the standard deviation of our measures. The standard deviation is a measure of the size of the differences between different values of a characteristic for a certain study object.

2.11.4 Different Types of Statistical Tests
When choosing a statistic test for an evaluation, there are a number of different aspects to consider. First, a test may be parametric or non-parametric. Parametric and non-parametric tests are discussed in Sections 2.11.5 and 2.11.6 respectively.

**Paired and Unpaired Tests**
Second, a statistical test may be paired or unpaired. When we study one characteristic for two different study objects, we use an unpaired test. However, we may want to study two different characteristics, $C_1$ and $C_2$ for one group of objects instead. Then, we have to choose between a paired and an unpaired test. An unpaired test tells us if there is a difference between $C_1$ and $C_2$ in general, while a paired test tells us if there is generally a difference between $C_1$ and $C_2$ for each object in the group.

**One- and Two-Sided Tests**
Third, a test may be one-sided or two-sided. Assume that we want to make a statistical test of a characteristic, C, of two study objects, A and B. If we do not know which study object has the largest value of C, we perform a two-sided test, which makes no assumption about the order of the values of C. On the contrary, if we know that C can not be smaller for A than for B, we may instead make a one-sided test. A one-sided test has higher power
than the corresponding two-sided test, but it can only detect differences in one direction. Hence, if $C$ is smaller for A than for B, the one-sided test will not detect this.

2.11.5 Parametric Tests

As mentioned above, there are both parametric and non-parametric tests. Generally, the former have higher power, but they require more properties to hold for the underlying distribution of the population. For example, the well-known $t$-test, which is described next, requires that the characteristic under study is normally distributed and has the same variance (which is the standard deviation squared) in the two populations. However, it is relatively robust to violations of the second condition if the sample sizes are equal. If, in addition to that, the distributions are symmetric it is also robust to violations of the first condition.

The $t$-test

The $t$-test is a parametric test commonly used for comparing a characteristic of two different study objects. When performing a $t$-test, we use our sampled values to compute a statistic called $t$. The value of $t$ can then be used to determine the p value of the test.

ANOVA and MANOVA

There are cases when we want to compare a characteristic for several different study objects, not only two as described above. A common parametric test for this situation is ANOVA, which is an abbreviation of “ANalysis Of VAriances”. As is the case with the $t$-test, ANOVA gives us a p value from which we may judge whether there is a statistically significant difference between the objects. However, if the result is significant, ANOVA does not tell us between which study objects there is a significant difference. To find out, we may e.g. perform pairwise tests (such as $t$-tests) between all study objects.

Sometimes, we study not only one, but several different characteristics. It is possible to make an ANOVA for each characteristic to look for differences between the study objects. Another possibility is to analyze all characteristics at once using an multivariate ANOVA (MANOVA). The MANOVA tells us whether all these characteristics, as a whole, differ for our study objects.

Transformations of Sampled Values

If the preconditions of the test that we want to make are not met, we might transform the sampled values such that we perform the test on e.g. the logarithm or the square root of the characteristic, if it fulfills the preconditions.
Resampling Tests
Another approach that can be used if the preconditions of a test are not met is to use a resampling method after having got a significant result. Here, we resample our sampled values a large number of times and compute the statistic of interest each time. For a t-test, the statistic of interest is the t statistic. From these values, we can determine a probability distribution for the statistic, from which we can compute a new p value. This value represents the probability that the first, low, p value did not arise by chance, but represents an actual statistically significant difference.

2.11.6 Non-Parametric Tests
If the distribution of the characteristic that we study does not fulfill all preconditions of a parametric test, another approach is to use a non-parametric test For comparing one characteristic from two study objects, two commonly used non-parametric tests are the Wilcoxon rank-sum test and the Wilcoxon signed-rank test. These build on the ranks of the measurements rather than the actual values. The former is an unpaired test, while the latter is paired. Like the t-test, these tests give us a p value that can be used to determine if there is a significant difference or not.

The non-parametric equivalent of ANOVA is called Kruskal-Wallis ANOVA.

2.11.7 Adjustment of p values or Significance Levels
Making many statistical tests on the same data increases the risk of a Type I error. (Statistically, if we set $\alpha = 0.05$, we will have a Type I error for one significant result out of twenty.) To mitigate this risk, one may adjust (that is increase) the p values such that the risk of a Type I error decreases for each individual test. An equivalent method is to adjust (decrease) the significance level $\alpha$. A well-known procedure for adjustment of the significance level is the Bonferroni procedure, which is relatively simple, but also very conservative (i.e. the power is low).

2.11.8 Correlation Tests
It is not always the case that we want to judge whether a certain characteristic differ between study objects. We may also want to know if and how much two characteristics co-vary. In those cases, if the relation that we want to test for is linear, we may perform a correlation test. The correlation test will give us a p value and a correlation coefficient. Similar to the ones described above, the p value is the probability that there is a linear relation between the two characteristics. The correlation coefficient tells us how strong the relation is. It is normalized to a value between -1 and 1. The larger the absolute value,
the stronger the co-variation. A positive value means that there is a positive relation between the characteristics, while a negative value implies a negative relation.

In this thesis, a non-parametric correlation test called Spearman’s rank correlation test is used. The correlation coefficient calculated by this test is denoted by $\rho$. 

In 2013, the Nobel prize in chemistry was awarded three researchers whose work has laid the foundation of current computer models of chemical reactions [36]. This demonstrates the importance of computer models for research, which is also stressed in a literature study on the interaction between software engineering and scientific software, written by Storer [37]. Carver et al. [6] state that software development skills are becoming as important as laboratory skills for science. Further, two literature studies from the last six years [3, 4], both discussing the usage of software engineering in scientific computing, argue that the importance of in silico experiments\(^1\) is increasing. In short, scientific computing software is of great importance for research, and its importance is increasing.

In scientific computing software, small mistakes may introduce defects that make the results unusable. To the worst, the results are used anyway, because the users of the software are unaware of the defects. Miller [38] reports on a scientist who had to retract three papers from *Science*, and report that two papers in other journals contain errors. The reason is that the home-made program he used for data analysis swapped two columns in his data. The program was used to analyze proteins, and the most influential of the retracted papers was of great clinical interest since the protein plays an important role in drug resistance. Before it was retracted, it was cited in 364 other publications.

A more recent example of a small mistake that caused a big error can be found in a program used by the Public Health Agency of Sweden [39]. The program estimated that there were 4 million residents of the Swedish capital Stockholm who were infected by undetected COVID-19. Since Stockholm has no more than 2.4 million residents, this was obviously wrong. The defect stemmed from an incorrect parameter value in the simulation. In this case, the error was easily detected, but the result could as well have been probable but still considerably wrong. That could lead to the error never being detected and decisions being taken based on incorrect data.

Defects that are likely to have caused heavy monetary losses are reported by Hatton [40]: Together with a co-author, he studied the accuracy of programs processing seismic data, used for siting oil wells. They found that there were accuracy-deteriorating defects that effectively made the decision-making process random. As Hatton [41] puts it, “this could lead to a 20 million dollar well being drilled in the wrong place”.

\(^1\)i.e. experiments run on computers
3.1 Software Engineering in Scientific Computing

Software for scientific computing is often developed by researchers in scientific computing or in the application domain. In this chapter, these will be referred to both as researchers and as developers of scientific computing software.

The three examples above shed light on the importance of correctness of software for scientific computing. Moreover, given the extent of the development of scientific computing software in research, productivity is an important aspect. If the software is developed inefficiently, a large amount of time is wasted. Here, software engineering practices stand scientific computing in good stead, since software engineering aims at high productivity in software development and high quality of the developed artifacts. In a paper by Carver et al. [6], the authors report on a researcher (who is also co-author of the paper) starting to apply software engineering best practices to his scientific computing software projects. He observed significant improvements in productivity as well as in the quality of the software produced. Further, in their literature study, Heaton & Carver [3] conclude that both software engineers and scientific computing software developers believe that application of software engineering practices could result in higher quality of scientific computing software.

However, few practitioners of scientific computing have any formal training in software engineering [6]. Instead, they are mostly self-taught [6], or learn from colleagues – who do not have any formal training either [3]. Still, in a survey of developers of software for scientific computing [6], Carver et al. find that most respondents think that their knowledge in software engineering is “mostly sufficient” for their work in the scientific computing projects. However, when asked about specific software engineering practices, they indicate that their knowledge and use are relatively low. This indicates that scientific computing software developers are unaware of many tools from software engineering. Interestingly, the respondents seem to have higher confidence in their own software engineering skills than in those of other researchers in the field.

A reason for the lack of formal training may have been given by Storer [37], Johanson & Hasselbring [4] and Heaton & Carver [3]. These three literature studies on the usage of software engineering techniques in scientific computing state that the goal in scientific computing is to produce new knowledge, not to develop software. The software is generally considered a means to obtaining more knowledge. According to the most recent of these studies [4], software development skills have lower status than domain knowledge. Here, domain means the domain from which the problems to be solved using the computer program originate. Training in software engineering is considered an extra burden upon all other duties of a researcher.
3.1.1 History and Present State
The view of software as a means to other ends might also explain a fact estab-
lished in a systematic mapping study on software engineering practices for
scientific software development written by Arvanitou et al. [42]: In the
eighties and nineties, research on scientific computing software did not fo-
cus on software engineering practices. Similarly, Heaton & Carver [3] point
out a historically low emphasis on quality of scientific computing software.
However, during the first decade of this century, the number of studies dis-
cussing software engineering for scientific computing increased consider-
ably. According to Johanson & Hasselbring [4], this change can be explained by
two factors: Firstly, scientific computing software developers experienced a
“productivity crisis”. The hardware on which the programs were run became
more complex, and the programs needed to be modified to be able to lever-
age the capabilities of the computers. Meanwhile, the models underlying the
programs became more complex and more researchers were involved in the
development. This obstructed keeping the development of large and insuffi-
ciently maintainable code bases productive. Secondly, the credibility in scien-
tific computing software was questioned, not least after the Climategate scan-
dal, were leaked emails revealed the limitation of programming skills among
climate researchers.
Since then, the software engineering practices of version control and is-
issue tracking have been adopted by the scientific computing community [3].
However, there is still room for improvement regarding e.g. testing [3, 4, 37],
reuse [4] and maintainability [4, 43]. The fact that the challenges related to
usage of software in research persist after more than a decade is interpreted
by Storer [37] as suggesting that there are no straight forward solutions. All
three literature studies mentioned above [3, 4, 37] argue that existing software
engineering practices must be adapted for scientific computing, since software
for scientific computing is different than other software.

3.1.2 Scientific Computing Software
According to Johanson & Hasselbring [4], software development in scientific
computing is done on all levels, from scripts for small-scale data analysis to
large and complex simulation software that is run on large compute clusters.
The large span is also visible in the examples in the beginning of this chapter:
The code for estimating the number of undetected cases of COVID-19 is about
1000 lines [39], while the seismic data processing programs contain hundreds
of thousands of lines of code [41].
According to Heaton & Carver [3], there are two types of software in sci-
cientific computing. The first type is constructed to be run once or twice, while
the second is aimed to be run many times, often outside the research group
that constructed it. They state that most scientific computing software is only
used in the research group where it is developed, a view shared with Johanson & Hasselbring [4].

In the literature [37, 6, 4, 3], there are different ideas whether scientific computing code is long-lived or short-lived. Presumably, both types of code exist. Consider the two types of software described by Heaton & Carver above. It is reasonable to think that code bases of the first type have a short life-span while those of the second type live for a longer time. The view that scientific computing software is long-lived seems to be more common than the opposite. A possible explanation for this is that research about software engineering practices in scientific computing more often focuses on long-lived code, since it can provide the researchers with more data.

According to Heaton & Carver [3], usability is not a priority for software that only is used within the developer’s research group (that is most scientific computing software), which in turn reduces maintainability. Similarly, Carver et al. [6] claim that many large scientific computing code bases are hard to maintain. They report that the general knowledge and use of e.g. refactoring is low among their respondents and conclude that this may be an explanation for the low maintainability. Heaton & Carver [3] report that scientific computing software developers see redesign of their programs as a waste of time. In addition, scientific computing software is most often developed without focus on intelligibility from the start, according to Johanson’s & Hasselbring’s study [4].

Johanson & Hasselbring state that this makes researchers rewrite others’, or their own, code rather than trying to understand and reuse existing code. The lack of reuse is backed up by Heaton & Carver [3]. They write that the main reasons are that the software does not meet the researchers’ requirements closely enough, and that it is hard to understand and poorly documented. They also report that developers of software for scientific computing are not convinced that reuse of existing code will save them effort in the development.

### 3.2 Maintainability of Code for Scientific Computing

Due to the low degree of reuse, practitioners of scientific computing waste considerable effort re-inventing existing technologies [4] instead of reusing code. Heaton & Carver [3] list a number of qualities needed for a reusable code component. Among these, ease of understanding and modification can be found. Both are important aspects of maintainability (analyzability and modifiability respectively). The importance of maintainability, for reuse as well as for correctness, is also argued for in the following section.
3.2.1 The Importance of Maintainability

According to the literature cited above, one reason for the lack of reuse is that code for scientific computing often is hard to understand. Consequently, the problem might be mitigated if scientific computing code was made more maintainable. If the programs that are written to be used once or twice were easier to understand and modify, its developers could more easily extend them. Then the code could be reused in future projects building on the first one. Thereby, researchers could save time that could be used for other work, which they, according to the literature, consider more important than software development. If the code was maintainable, and made available, also other researchers could use it, which would increase the impact of the researcher who developed it. If this was common practice, scientists could build upon each others work instead of redoing it, which is how science is supposed to be performed\(^2\).

Additionally, writing more maintainable code, we as researchers in scientific computing could decrease the risk of defects as those described in the beginning of the chapter: The more complex something is, the higher the risk for mistakes during its creation and further development. Therefore, keeping the code modular and comprehensible will reduce the risk of errors in the coding process. Moreover, if we write the code once instead of \(n\) times, there are \(n - 1\) fewer chances for defects to accidentally be inserted, and we have \(n - 1\) more chances to detect the defects that are still introduced. Further, testability\(^3\) would increase our chances to find and fix these defects.

For code bases that are already large and long-lived, improved maintainability is at least as important as for the ones that currently are not designed to be reused. A large code base that is not modular, analyzable, modifiable and testable will require the scientists updating it to spend a vast amount of time and effort. Moreover, updates will be more error-prone the less maintainable the code is.

In their systematic mapping study, Arvanitou et al. [42] report that factors that can increase the productivity of scientific computing software developers are frequently studied. Their interpretation is that there is a high interest in being able to work more efficiently with the software. Reusing more code, introducing fewer defects (that need to be fixed) and spending less time each time our code needs an update, we would most likely be more productive. In other words, there should be a strong interest in making scientific computing code more maintainable, and indeed Arvanitou et al. find that maintainability is the third most studied quality attribute, preceded by performance and productivity. Still, less than 2\% of the studies covered in their paper discuss software maintenance.

---

\(^2\) unless the aim is to reproduce earlier studies

\(^3\) Recall that testability is a sub-characteristic of maintainability!
3.2.2 The Importance of Performance

Higher interest in performance than in maintainability is not only reported by Arvanitou et al.. Also, Johanson & Hasselbring [4] find that practitioners of scientific computing give higher priority to performance than to maintainability. This thesis agrees that performance is an important aspect of software for scientific computing, in particular for high performance computations. As mentioned in Section 1.1, there are high performance computing programs that take days, weeks or even months to run. Higher performance of these programs will shorten down the time we need to wait for the results of the simulations, not to mention reduce the electricity cost. Alternatively, it may allow us to run larger simulations without increasing the execution time and power consumption.

3.2.3 Object-Orientation as a Means to Code Quality

The challenge of producing high quality code has been discussed in software engineering for decades. Different solutions have been presented, of which one is object-oriented programming. Certainly, programming object-orientedly is not the only way to obtain high quality code, but it is considered interesting to analyze in this thesis for the reasons explained below.

Encapsulation and flexibility are listed by Heaton & Carver [3] as qualities necessary for reusable components, and as described in Section 2.2, encapsulation is a central concept in object-oriented programming. Moreover, as explained in Sections 2.2 and 2.3, object-oriented programming and design can aid in writing flexible code. This suggests that object-orientation can be beneficial for reusability.

Further, Johanson & Hasselbring [4] as well as Tu et al. [43] argue that better separation of domain logic and implementation details is needed in scientific computing software. Object-orientation supports this separation naturally, since domain logic and implementation details can easily be separated into different classes. Low-level details can be hidden behind well-defined interfaces instead of being exposed to classes operating on a higher level. This is one advantage of the modularity that can be obtained using object-oriented programming. Modularity is also considered important in three of the studies covered by Heaton & Carver [3], which all come to the conclusion that the use of an object-oriented language was necessary for the success of their projects.

Also Arge et al. [44] discuss the advantages of object-orientation for scientific computing. They point out that it is possible to create abstractions in software that precisely correspond to mathematical abstractions. Details such as matrix sizes and storage formats can be encapsulated in the class, and algorithms can be expressed in a more general (reusable) way. This separation of implementation details and domain logic also makes it possible to change the implementation details without affecting other classes. The authors also
mention the fact that general method names such as `print`, `init` and `add` can be reused in different classes, eliminating the need to include e.g. type information in the method names.

### 3.2.4 A Potential Problem of Object-Orientation

Johanson & Hasselbring [4] argue that we should use software engineering approaches to separate concerns, but that it should not be done at the expense of performance. Similarly, Arvanitou et al. [42] consider that performance should be taken into account for in studies concerning scientific software, also when other quality attributes are the primary interest. This is a potential problem of object-oriented programs, since frequent creation (and destruction) of objects, as well as the use of dynamically bound methods, may hamper performance.

### 3.2.5 Work Presented in this Thesis

This section presents work related to object-oriented programming, which is not the only topic dealt with in this thesis. Work done on the other topic, code quality in Jupyter notebooks, is described in Section 3.3.3.

Paper I and II deal with the potential conflict between maintainability and performance of object-oriented code for scientific computing. The most performance critical parts of a program for high performance computing is refactored into more object-oriented code, for the sake of improved maintainability. The studies are made on one single program, but the program belongs to an important class of applications for scientific computing and the results might be generalizable to other applications in this class.

The original code is a mixture of code expressing high-level algorithms and implementation details. It consists of functions that sometimes take very long lists of parameters, representing different indices and data containers. The code is written in C++, but the usage of classes is limited, and the classes that do exist do not encapsulate any state. Further, the part of the code targeted by the refactoring does not make use of any inheritance or polymorphism.

Two versions of the refactored code are developed, one using dynamic polymorphism and one using static polymorphism. In the first-mentioned version, the (by far) most frequently executed lines in the code contain a number of calls to dynamically bound methods. In the latter, these dynamically bound methods are made statically bound. The effects on maintainability and performance of object-orienting the code and of using static polymorphism are analyzed.

Next, the performance critical parts are extracted from the two refactored versions of the code, and two corresponding implementations are made in Java, which inherently uses dynamic binding. The motivation for this is that Java’s JIT compiler may be able to use run-time information to optimize the
code as if the dynamically bound methods were statically bound. This effect, as well as other performance effects of using Java instead of C++ for this type of applications, are studied.

All performance evaluations are made for serial, multi-threaded as well as distributed executions of the code bases.

### 3.3 Jupyter Notebooks

For researchers who write their code in Jupyter notebooks, performance is much less of a concern. Here, the computations performed are much smaller than in high performance computing, and Python is the most commonly used language. Generally, an execution of Python code is much slower than an execution of the corresponding C or Java code.\(^4\) On the other hand, Python reads as pseudo code and has a relatively low learning threshold.

When it comes to code reuse, Python shows its paces. The language contains many high-level functions for performing common tasks. Moreover, currently (December 14, 2020), the official Python third-party software repository, PyPI, contains over 275 000 packages.

#### 3.3.1 Code Cloning

Python programmers clone code to a large extent [45, 46]. Cloning is important to consider when it comes to correctness and maintainability of code. For one thing, code duplication increases the amount of code that needs to be maintained. Moreover, if a defect is found in cloned code, it has to be fixed in all clone instances. According to a study by Juergens et al. [47] it is not uncommon that one instance in a clone pair is overlooked when the other one is changed. Their study covers cases when the code containing the two clones is maintained by the same developers. It can be expected that the situation is even worse when the two code segments are maintained by different people. Juergens et al. also note that inconsistently changed clone pairs contain more defects than code in general.

Cloning is also important from a scientific point of view. When we, as researchers, study code corpora, clones may bias our results if we do not take them into account. As an example, a Python module may be imported in a notebook that is later copied a large number of times. It may for example be part of the material for a course where the students have put the notebook in their own repository just to have it available as an example, or as a starting

\(^4\)Note that this is the general case. Some Python libraries, e.g. `numpy` are C code that is called through a Python interface, and could be expected to perform at the level of C. Further, there are Python implementations containing a JIT compiler, which could be expected to speed Python programs up.
point for their work. If this is the case, counting the number of imports of different Python modules in code downloaded from public online repositories, one might conclude that Python programmers often choose to use this module, which is not necessarily the case. The risk that the conclusions of studies of software repositories are biased by clones is also pointed out by Lopes et al. [46].

3.3.2 Error Suppression in Python

Considering correctness, this thesis argues that Python disregards possibilities to help users identifying errors. This is particularly unfortunate for scientific code, since researchers tend to think that the problems in their results stem from the scientific theory [3]. Python is a very allowing language and when a variable differs from what could be expected, Python often tries to make sense of the developer’s code instead of signaling that something may be wrong.

For example, consider a function that takes a list as its argument. Assume that the user of this function passes a string instead of a list, e.g. because there is a defect in his/her code or his/her input data is not what he/she expects. Unless a check of the type of the argument is explicitly done in the function, Python will use the string as an array where each character is one element. In this case, the result may not at all be what the user intended.\(^5\) This type of errors may very well go undetected, yielding incorrect results of the program. If, instead, Python issued an error or a warning, the user would find out about the problem and have a chance to fix it.

Another example is the function `show` in `matplotlib.pyplot`. When this function is called, all open figures are displayed. The function takes a parameter, `block`, which is supposed to be a boolean value specifying whether the program should block until the windows in which the figures are shown are closed.\(^6\) When `pyplot` is run in interactive mode, `block` defaults to `True`. Otherwise, it defaults to `False`. In Python, when used as a condition, initialized objects are interpreted as `True`, while `None` is interpreted as `False`. There are users of `show` who pass a plot or figure object as an argument, instead of passing a boolean value, see Paper IV. Presumably, these users expect their programs to show the figures passed as arguments. Instead, all open figures are shown. Further, the blocking behavior may not be what the user expected, leading e.g. to programs that unexpectedly halt their executions, or to plots with which the user cannot interact as expected. For a user who is convinced that the figure to be shown should be passed as an argument to `show`, the cause of these problems may be hard to find. If `show` issued a warning when being passed values that are likely not to be intended to be used as boolean values,

\(^5\)A C developer might argue that it is natural to regard a string as a character array, but Python developers generally work on a higher level of abstraction.

\(^6\)This is a slightly simplified explanation. For more details, see the documentation [48].
the user would get valuable information about his/her mistake, which could help resolving the defect faster.

3.3.3 Work Presented in this Thesis

This section presents work on Jupyter notebooks, which is not the only topic dealt with in this thesis. Work done on the other topic, the usage of object-oriented programming for high performance computing, is described in Section 3.2.5.

In Paper III and IV, a large corpus of Jupyter notebooks is analyzed. In particular, code cloning and possibly unexpected behavior of frequently called Python functions are discussed. The corpus consists of 2.7 million Jupyter notebooks downloaded from GitHub spring 2019. First, basic characteristics of the corpus are investigated, which forms a basis for the subsequent work. Moreover, the presence of cloned code cells is analyzed, both at the level of CMW clones (i.e. exact copies modulo whitespace) and of near-miss clones (which are mostly equal, but may contain small differences). To the best of knowledge of the authors of the paper, to date, this is the largest study of cloning in Jupyter notebooks. Information about the clones is used to improve the quality of the results of the remainder of the work on Jupyter notebook in this thesis.

After the clone analysis, the modules that are most frequently imported in the Jupyter notebook corpus are identified. The usability of the most frequently called functions from these modules is discussed: Parameter combinations for which nonsensical combinations of values are accepted without any warning or error are identified. Moreover, the prevalence of such combinations of values is studied. Last, a number of issues are reported to the developers of the modules in which these functions reside, with suggestions of better argument checking. The functions discussed are all very usable in scientific computing, in particular in data science.

Commit Rates for Jupyter Notebooks

During the work with the Jupyter notebook corpus, the author of this thesis did a pilot study which never made its way to any published paper. Nevertheless, the results are relevant in the context of this thesis. Therefore, the study is described below.

From a corpus of 2.7 million Jupyter notebooks downloaded from GitHub spring 2019, 100 notebooks were randomly sampled. Commit information was downloaded for the repositories were these notebooks resided, if the repository still existed, which 97 of them did. For each of the 97 repositories, the

---

7For a description of the corpus, see Paper III.
number of notebooks included and the number of times a notebook had been committed were counted.\textsuperscript{8}

It turned out that in total, the repositories contained 4915 Jupyter notebooks, which were committed 6303 times altogether. Hence, a notebook in the sample is only committed 1.28 times on average. Since each notebook must have been committed at least once (when it was first added to the repository), at least 72\% of the notebooks have no more than one commit. In other words, for most notebooks in the sample, there is only an initial commit, and no committed updates, which suggests that the notebooks are generally not further developed once they are written and committed the first time.

3.4 Summary of Papers

This section contains a summary of each of the four papers written in this PhD project.

3.4.1 Paper I

Paper I is a technical report which is an extended version of a paper published in the proceedings of IEEE International Workshop on Source Code Analysis and Manipulation 2014 [1]. The published paper is not included in the thesis.

In this paper, the most performance critical parts of a high performance computing code base written in C++ are refactored using an object-oriented approach. Since the refactoring introduces a considerable number of calls to dynamically bound methods, two versions of the refactored code are written. One employs dynamic polymorphism, while the other one employs static polymorphism, replacing the above-mentioned dynamically bound methods with statically bound methods. The performance as well as the maintainability of the three code versions are compared. A quantitative analysis using software metrics suggests that the refactored code is more object-oriented and contains shorter methods. Moreover, a qualitative discussion concludes that the two refactored code versions are more maintainable than the original code and out of them, the code employing dynamic polymorphism is most maintainable. It is also concluded that the refactored versions of the code adhere well to the single responsibility principle, the open-closed principle, the Liskov substitution principle and the interface segregation principle. Moreover, the version that employs dynamic polymorphism adheres well to the dependency-inversion principle, which is not the case for the version using static polymorphism. Except from worse adherence to the dependency-inversion principle, the code with static polymorphism has a couple of other issues when it comes to maintainability, which are discussed in the paper. The difference in performance

\textsuperscript{8}Commits containing $X$ notebooks were counted $X$ times.
of the original code and the code with dynamic polymorphism is statistically significant. As could be expected, the original code is considerably faster that the dynamic polymorphism version. Between the original code and the code employing static polymorphism, a statistically significant performance difference is found as well, but the absolute difference is small. The original code performs best in two experimental setups, while the code with static polymorphism is faster in four other experimental setups.

Contributions
The author of this thesis refactored the code and designed and ran all experiments. All three authors contributed to the text of the conference paper, while the author of this thesis made the extensions for the technical report.

3.4.2 Paper II
Paper II is a technical report which is an extended version of a paper published in the proceedings of ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes 2019 [2]. The published paper is not included in the thesis.

In this paper, the performance effects of using Java instead of C++ for the application described above are evaluated. Since the performance degradation of the code studied in Paper I stems from dynamic binding, a language with a JIT compiler might yield faster code. In this type of code, targets of the calls to dynamically bound methods may vary from one execution of the code to another, but not during a single execution. Hence, the JIT compiler might be able to use run-time information to optimize the code as if the methods were statically bound.

This is evaluated as follows: The most performance critical parts are extracted from the two refactored versions of the code, and two corresponding implementations are done in Java. One implementation uses OMP4J for shared memory parallelization, while the other is parallelized the standard Java way, see Section 2.7.1. For distributed memory parallelization, both versions use MPI, just like the C++ implementations. The implementation using OMP4J is compared to the C++ implementations. It appears to be faster than the C++ code employing dynamic polymorphism and slower than the one with static polymorphism. The differences are statistically significant.

A deeper analysis indicates that the main reason for the difference between the C++ implementation with static binding and the Java implementation is the way data to be communicated using MPI is handled. When the time needed for communication is excluded from the analysis, the difference decreases considerably. In one experimental setup the difference is only 4%.

In scientific computing, Java has a reputation of being slow because of the overhead of the virtual machine, and features such as just-in-time compilation, dynamic memory management and array bounds checking. In this study,
the time spent in the warm-up phase appears to be negligible compared to the execution times needed for this type of application. Dynamic memory management does not seem to be a problem either: This type of application allocates more or less all memory it needs just after the program starts and uses it through the whole execution. Hence, the performance impact of dynamic memory management should be minimal. On the contrary, array bounds checking is done to a large extent. Actually, all computations in the part of the program where almost the entire execution time is spent are performed on array elements. Still, a detailed analysis suggests that not more than 4% of the execution time is spent on array bounds checks, demonstrating that such protection is compatible with high performance.

Contributions
The author of this thesis read up on related work, wrote the source code under study and designed and ran all experiments. Both authors contributed to the text.

3.4.3 Paper III
Paper III is the largest study performed on Jupyter notebooks to date (as far as the authors of the study know). It deals with basic properties and cloning in a corpus of 2.7 million notebooks. The corpus consists of all notebooks that were available on GitHub when it was downloaded in spring 2019. The notebooks are generally small with a median of 47 non-empty lines of code and 9 code cells. However, the size distributions are highly right-skewed and there are a few very large notebooks. It appears that the code is written in Python in more than 95% of all notebooks.

The clone analysis shows that more than 70% of all code cells are CMW clones of each other, and around 50% of the notebooks consist solely of CMW cloned code cells. For a majority of these, 48% of all notebooks in the corpus, there is at least one notebook containing an identical sequence of code cells (if CMW clones are considered identical). Further, around 80% of the Python code cells are near-miss clones, and this is most likely an under-estimate. Only 13% of the notebooks contain no code cells that are CMW clones, and 5% contain no code cells that are near-miss clones.

By far, most clones (of both types) are inter repository clones, but the single repository from which a notebook contains most clones is the one where itself resides.

Contributions
The author of this thesis read up on related work, designed and performed all analyzes of the downloaded data and wrote the lion’s share of the text.
3.4.4 Paper IV

In this paper, the analysis of the Jupyter notebook corpus downloaded for Paper III is continued. However, from each group of notebooks where each code cell is a CMW clone of the corresponding cell in the other notebooks, only one notebook is included. Imports of external modules are analyzed and the most frequently called functions in the most frequently imported modules are identified. Each notebook imports 6.36 modules on average. The most commonly imported modules are (in order) `numpy`, `pandas` and `matplotlib.pyplot`. These are all important for scientific computing, not least for data science.

The documentation of the seven most frequently called functions from the top ten imported modules is studied in order to find combinations of parameters for which a user could pass values that indicate an error in the code or the input, without getting a warning or an error. These are referred to as risky parameter combinations. 111 risky parameter combinations (of which 16 are targeted in the documentation) are identified, and divided into 5 different categories. Cases where the users do pass error-indicating combinations of values for the risky parameter combinations are identified in the notebook corpus. Last, issues are filed in the GitHub repositories where the functions having risky parameter combinations reside. The issues suggest that the functions should warn or err when being passed error-indicating values for the risky parameter combinations. The intent is to help future users of the functions to find defects in their code. This would facilitate debugging, and prevent defects from going undetected, when such argument combinations are used.

Contributions
The author of this thesis read up on related work, implemented and performed all analyses, including the identification and categorization of risky parameter combinations, and wrote the lion’s share of the text in this paper.

3.5 Concluding Remark

This thesis has contributed to the field of scientific computing by showing that it is possible to use object-oriented programming to write maintainable code with high performance. Hopefully, this will contribute to increased reuse of source code and tests in the field, which would lead to a lower risk of defects in the code and less waste of human resources.

It has also contributed to understanding of Jupyter notebooks, in particular regarding code clones. Knowledge of clones is important for code quality as well as for research. If not taken into consideration, clones may bias the results of scientific studies. Last, impact of the culture to suppress errors in Python has been analyzed for the most frequently called functions in the most frequently imported modules in a large corpus of Jupyter notebooks. A number of issues have been reported to the developers of the functions, suggesting
warnings or errors to be issued when arguments passed to the functions suggest an error in the user’s code or input, instead of suppressing the error.
4. Summary in Swedish


Resultaten från program som utvecklas och används inom beräkningsvetenskap ligger till grund för bland annat vetenskapliga artiklar och viktiga beslut, som exempelvis kan påverka människors hälsa. Därför är det viktigt att dessa program ger korrekta resultat.

Inom beräkningsvetenskap anser ovan nämnda datortillämpningar ofta inte vara viktiga i sig själva, utan fungera som ett medel för ny kunskap [3, 4, 37]. Ett ämne som däremot fokuserar på mjukvara (vilken involverar programkod) är programvaruteknik. Här ses programkoden som något som ska finnas under lång tid och användas upprepade gånger, vilket ofta inte är fallet med beräkningsvetenskaplig kod. Det gör att fokus på kodens kvalitet är högre i programvaruteknik än i beräkningsvetenskap. Inom programvarutekniken läggs stor vikt vid att program ska bete sig korrekt och vara förvaltningsbara, det vill säga inte vara onödigt svåra att testa, analysera, (vidare)utveckla och återanvända.

Beräkningsvetenskaplig kod utvecklas ofta av forskare inom beräkningsvetenskap eller det tillämpningsomrade inom vilket programmet ska bidra till ny kunskap. Dessa forskare är experter på sitt område, och utöver det har de inte sällan djupa tekniska kunskaper om det programspråk de använder. Däremot är kunskaperna om och användningen av metoder från programvaruteknik ofta lägre [6]. Detta leder bland annat till en onödigt hög risk för fel i koden, och till att värdefull forskningstid behöver läggas på att göra om programmeringsarbete som redan är gjort, alternativt på att förstå kod som hade kunnat vara skriven på ett mer lättbegripligt sätt.
4.1 Objektorienterad programmering


En teknik som används i omskrivningen av ovan nämnda program, och som är central inom objektorientering, kallas för polymorfi. Polymorfi refereras även till som ”dynamisk polymorfi” nedan. Tekniken gör programmet mer flexibelt, vilket är en fördel ur förvaltningsbarhetssynpunkt. Däremot är det en nackdel för kompilatorn eftersom det gör att koden innehåller mindre information som kan användas vid optimeringen.


Ett programspråk som har många likheter med C++ är Java, men generellt kompileras Java-koden på ett annorlunda sätt än C++-kod: de flesta Java-kompi- latorer utför bara det första steget av dem som utförs vid kompilering av C++-kod, och resultatet blir så kallad bytekod. För exekvering av denna bytekod används i de flesta moderna Java-versioner en annan typ av program, som kallas

Java har även fördelar jämfört med C++ ur ett förvaltningsbarhetsperspektiv. Bland annat får användare hjälp med hantering av datorns minne på ett sätt som en C++-användare inte får. Detta minskar risken för fel som i vissa fall är mycket svåra att lokalisera.


4.2 Jupyter notebooks


Med slutmålet att bidra till att avhjälpa fel i kod för dataanalys studeras en korpus på över 2,7 miljoner Jupyter notebooks nedladdade från GitHub, en webplats som är populär för lagring av kod. Analysen i det första delarbetet
visar att de flesta notebooks innehåller en relativt liten mängd kod. Mediana antalet snippetar i en notebook är 9, medan medianen för antal kodrader i en notebook ligger runt 50. Koden i Jupyter notebooks kan skrivas i ett 40-tal olika språk, men snippetarna i en enskild notebook ska skrivas i ett och samma språk. Analysen av korpusen i det första delarbetet visar att en stor majoritet av notebooken, drygt 95%, innehåller kod skriven i programspråket Python. Python är ett språk som har vuxit mycket i popularitet de senaste åren, och det är ett av världens populäraste språk [25].

Även mängden kodkloning i korpusen analyseras i det första delarbetet. Två snippetar är varandras kloner om de består av identisk, eller nästan identisk kod. I den här avhandlingen används begreppet CMW-kloner för kloner som innehåller identisk kod, möjligen med skillnader i antal och placering av blanksteg. För att beskriva kloner med små skillnader i koden används det väletablerade begreppet near-miss-kloner.


Kodklonsanalysen visar att kloner är vanliga i korpusen. Av alla snippetar har drygt 70% en eller flera CMW-kloner. Andelen snippetar som har en near-miss-klon bedöms vara ungefär 80%, men det är med största sannolik en underskattning.

Programkod delas normalt upp i mindre enheter som kallas för funktioner.1 En funktion kan anropa en annan, det vill säga få den andra funktionen att exekveras. Detta ger mer strukturerad och överblickbar kod, vilket gör koden lättare att både skriva och förvalta. En ännu större fördel med funktioner är dock att de kan användas för återanvändning av kod. En funktion kan anropas hur många gånger som helst, och från hur många olika ställen som helst. Kod som behöver exekveras flera gånger och i olika sammanhang bör därför läggas in i en egen funktion som kan anropas i de olika sammanhangen. Det är vanligt att man slår ihop flera funktioner till en enhet som kallas för en modul eller ett bibliotek. Dessa kan i sin tur användas av flera olika kodbaser istället för att behöva skrivas om och om igen av olika programmärene.


1Beroende på språk och programmeringsparadigm kan också termen “metoder” användas.
I det andra delarbetet i studien av Jupyter notebooks analyseras de av notebooksen i korpusen där koden är skriven i Python (drygt 95%). I den ovan beskrivna klonanalysen identifierades alla grupper av notebooks där samtliga snippetar är CMW-kloner av varandra. Endast en notebook från varje sådan grupp inkluderas i analysen.


För de sju funktionerna återfinns 111 möjligheter att ange de nyss beskrivna argumentkombinationerna. Ett antal förekomster av sådana argumentkombinationer identifieras i korpusen. Slutligen lämnas rapporter till utvecklarna av några av funktionerna, där det föreslås att funktionerna ska signalera till användaren när denne anger dessa potentiellt felaktiga argumentkombinationer. Sådana signaler kan bidra till att fel i användarnas kod hittas snabbare och i större utsträckning.

4.3 Slutnot

Denna avhandling har bidragit till beräkningsvetenskapen genom att visa att objektorienterad programmering kan användas för att skriva förvaltningsbar kod med god prestanda. Förhoppningen är att detta kommer att leda till ökad återanvändning av programkod i ämnet, vilket skulle leda till en lägre risk för fel och mindre slöseri med mänskliga resurser. Den har också bidragit till kunskaper om Jupyter notebooks, inte minst med avseende på kloner, något som är viktigt både för forskningen och ur ett kvalitetsperspektiv. Därutöver har den bidragit genom att identifiera tillfällen där ett antal frekvent anropade Pythonfunktioner försummar möjligheter att hjälpa användare att identifiera fel i deras kod, och föreslagit förbättringar för utvecklarna av dessa funktioner.

---

2Här avses en kurva som illustrerar hur två variabler samvarierar, inte en graf i datavetenskaplig mening.
Acknowledgements

First and foremost, I want to thank my main supervisor, Tobias Wrigstad, for your positive attitude and for many advises, ideas and new angles of approach regarding my work! Also, thank you for good discussions and for your pedagogical explanations! I have learned a lot from you. Further, thanks to both Tobias and to Dave Clarke for taking on supervising me without even having met me, and for showing great interest in my work.

Next, I would like to thank my supervisor Sverker Holmgren for helping me developing my ability of critical thinking and for initializing my first research project. I am also grateful to Michael Thuné for good discussions about science and for valuable feedback on my work.

Moreover, I want to thank my co-author of Paper I, Ebba Hvannberg. It was a pleasure to work with you! Also, thanks to Magnus Grandin for patiently answering a great number of questions (and questioning), and to Göran Arnqvist for your help on statistics!

Lina von Sydow has provided invaluable support during the toughest parts of this journey. Many thanks for that, and for your great leadership! Also, thanks to Britt Övragård for great support and encouragement during the writing of this comprehensive summary!

Last but not least, I want to thank Patrik for helping and supporting me in a countless number of ways, and for being the most fantastic person in the world! Without you, I would not have made it to this point.
References


Acta Universitatis Upsaliensis

Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2000

Editor: The Dean of the Faculty of Science and Technology

A doctoral dissertation from the Faculty of Science and Technology, Uppsala University, is usually a summary of a number of papers. A few copies of the complete dissertation are kept at major Swedish research libraries, while the summary alone is distributed internationally through the series Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology. (Prior to January, 2005, the series was published under the title “Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology”.)