Comparing Code Quality in Software Paradigms

A thesis presented
by
Marc Coquand
to
The Department of Computer Science

for a degree of
Master of Science in Engineering
in the subject of
Interaction Technology and Design

Umeå University
Supervised by Anders Broberg
June 11, 2019

Abstract

This study's goal is to compare approaches to functional programs and object-oriented programs to find how it affects software quality. By looking at 3 cases, we analyze, how does a functional approach to software architecture compare to an OOP (Object-oriented programming) approach when it comes to maintainability and code quality? TO BE REPLACED WITH CONCLUSION

Acknowledgements

TODO I want to thank myself for my amazing work

Contents

Chapter 1

Introduction: Software paradigms and complexity

Different schools of thoughts have different approaches when it comes to building applications. There is one that is the traditional, object oriented, procedural way of doing it. Then there is a contender, a functional approach, as an alternative way to build applications. Early programming languages were based around procedure calls. Procedures are the series of steps the computer needs to perform the computations desired. [1] These programming languages were turing complete. A programming language is Turing complete if is able to calculate everything that is calculatable. [2] Object-oriented programming is used to make code more reusable and structured. Java, amongst most languages, is a popular language that is turing complete, object-oriented and imperative. [3]

Functional programming originates from 1936 from Lambda calculus. Lambda calculus is a theory for functions and evolved from Church and Curry to create an alternative foundation for mathematics. [4] Even though Turing machines and Lambda calculus developed separately, the church-turing thesis proves that any computational problem that is solvable with Lambda Calculus is also solveable for Turing machines and vice versa. [5] What this means in practice is any program written in the functional paradigm can be written in the procedural, object-oriented paradigm. However it does not mean that the paradigms are the same as one solution might be very complex in a procedural language and simple in a functional language and vice versa as long as they are turing-complete.

As software engineers, one factor of concern is software quality. Software quality can be divided into two different subparts: software functional quality and software structural quality. Software functional quality reflects how well our system conforms to given functional requirements or specification and the degree of which we produce correct software. To check that the software is correct, software engineers create tests. [6] If a software engineer has to write more tests it could potentially lead to affecting software quality. By looking at cyclomatic complexity, described in Section 3.3.1, one can find out how the different software paradigms affect the amount of tests we need to write.

Software structural quality refers to how well the software adheres to non-functional requirements such as robustness and maintainability. [6] Some of the maintainability aspects, such as usability, is hard to measure quantitatively. By looking at the cognitive dimensions, described in Section 3.3.3, one can use an expert analysis to find how the two approaches affect the usability for the developer. So by looking at different case studies this study aims to find if the different software paradigms affect software quality.

Chapter 2

Background

This chapter will look at how concerns and requirements made impact the maintainability and testability. It aims to establish what are the current methods of developing applications and how software engineers today approach testing.

2.1 Architecture

When developing large scale applications, often the requirements are as follows:

- There is a team of developers
- New team members must get productive quickly
- The system must be continuously developed and adapt to new requirements
- The system needs to be continuously tested
- System must be able to adapt to new and emerging frameworks

Two different approaches to developing these large scale applications are microservice and monolithic systems. The monolithic system comprises of one big "top-down" architecture that dictates what the program should do. This is simple to develop using some IDE and deploying simply requires deploying some files to the runtime.

As the system starts to grow the large monolithic system becomes harder to understand as the size doubles. As a result, development typically slows down. Since there are no boundaries, modularity tends to break down and the IDE becomes slower over time, making it harder to replace parts as needed. Since redeploying requires the entire application to be replaced and tests becomes slower; the developer becomes less productive as a result. Since all code is written in the same environment introducing new technology becomes harder.

Enter microservices, in a microservice architecture the program comprises of small entities that each have their own responsibility. There can be one service for metrics, one that interacts with the database and one that takes care of frontend. This decomposition allows the developers to easier understand parts of the system, scale into autonomous teams, IDE becomes faster since codebases are smaller, faults become easier to understand as they each break in isolation. Also long-term commitment to one stack becomes less and it becomes easier to introduce a new stack.

The issue with microservices is that when scaling the complexity becomes harder to predict. While testing one system in isolation is easier testing the entire system with all parts together becomes harder.

2.1.1 Unit testing

Unit testing is a testing method where the individual units of code and operating procedures are tested to see if they are fit for use. A unit is informally the smallest testable part of the application. To deal with units dependence one can use method stubs, mock objects and fakes to test in isolation. The goal of unit testing is to isolate each part of the programs and ensure that the individual parts are correct. It also allows for easier refactoring since it ensures that the individual parts still satisfy their part of the application.

To create effective unit tests it's important that it's easy to mock examples. This is usually hindered if the code is dependant on some state since previous states might affect future states.

2.1.2 Property-based testing

Property-based testing tests the properties that a function should fulfill. A property is some logical condition that a specification defines the function should fulfill. Compared to unit testing where the programmer creates the mock values; property based testing generates values automatically to find a contradiction. For example a function $reverse : [a] \rightarrow [a]$, which takes a list and reverses it's items, should have the property $reverse \circ reverse\ x = x$. A unit test would check that $reverse[1, 2, 3] = [3, 2, 1]$; a property-based test would instead generate a list of values randomly and check it's properties hold.

2.1.3 Integration testing

Whereas unit testing validates that the individual parts work in isolation; integration tests make sure that the modules work when combined. The purpose is to expose faults that occurs when the modules interact with each other.

2.1.4 End-2-End Tests

An End-2-End test (also known as E2E test) is a test that tests an entire passage through the program, testing multiple components on the way. This sometimes requires setting up an emulated environment mock environment with fake variables.

2.1.5 Challenges

No clear guidelines exists on how many tests should be written. Google advocates for a 10% E2E-tests, 20% integration tests and the rest to be unit tests. [7] Spotify advocates instead to have a low amount of unit tests and instead focus on integration tests and having 80% integration tests, 10% E2E-tests, 10% unit tests. [8]

When writing unit tests that depend on some environment, for example fetching a user from some database, it can be difficult to test without simulating the environment itself. In such cases one can use dependency injections and mock the environment with fake data. Dependency injection is a method that substitutes environment calls and returns data instead. The issue with unit tests is that even if a feature works well in isolation it does not imply that it will work well when composed with other functions.

The challenge in integration and E2E-tests comes with simulating the entire environments. Given a server connected to some file storage and a database it requires setting up a local simulation of that environment to run the tests. This results in slower execution time for tests and also requires work setting up the environment. Thus it ends up being costly. Also the bigger the space that is being tested the less close the test is to actually finding the error, thus the test ends up finding some error but it can be hard to track it down.

Readability

The hypothesis of this thesis is that after designing a system to be testable the code becomes much harder to understand and more complex. The thesis investigates if the different language paradigms then end up having vastly different complexity and being much less readable after creating testable systems and if paradigms factor into that complexity.

2.2 Complexity and relation to testability

To establish how to look at complexity and readability, there exists different metrics for measuring software quality. Lines of Code (LOC) can be used as a measure in the software to find defects. [9] As this study will measure different programming languages and software paradigms where syntax is vastly different, it follows that using LOC to measure fault is flawed. Thus, here are some other ways to measure complexity:

Cyclomatic Complexity Described further in Chapter 3

Halsteads metric A metric that relates to the difficulty of writing or understanding code related to operators and operands. [10]

Chidamber and Kemerer Metrics [11] A complexity measure for Object-oriented programs. Since this study compares functional to object-oriented progams this metric is ill suited as it does not allow for comparative analysis.

Berg Van Der Klaas explored Halsteads metric and compared OOP and functional programming. The study notes that the psychological complexity needs to be taken into account and noted that the use higher order functions may affect results for functional programs for the Halstead metric. To measure complexity of software, a program could instead be measured at a lexical level. No quantitative measure was found for measuring the linguistic structures of the program and the comprehension. However a qualitative measure called Cognitive Dimensions was found will be used instead. Cognitive Dimensions inspects fourteen different aspects and they are not always applicable for all projects. Prior research suggests omitting some which has been taken into account and is explained further in Chapter 3. [12]

Chapter 3

Theory

This chapter will cover how the system architectures are implemented in practice for Object-oriented systems and functional systems to address the concerns established in Chapter 2. This chapter also aims to introduce different ways we can compare these software paradigms on complexity, both in testability and mental complexity. This way

Exact definitions exist of OOP but not for Functional programming, however both's stated goal is creating maintainable programs. The way OOP aims to make maintainable software is by emphasising encapsulation of state into *objects* and message passing. Functional programs emphasise moving state to the edges of the program, making the core logic of the program pure and using immutable data (defined in Section 3.1). Data that is immutable can not be changed after initialization. This is explained further in Section 3.1 and Section 3.2. To prevent defects, tests need to be written to check that the behavior works as expected, thus testability is of concern when creating maintainable software.

While paradigms define how we build our applications, we still need design patterns for structuring the source code. A design pattern is a template for the developer when structuring their code to solve certain problems. For instance, without design patterns, one could couple dependencies with logic which affects maintainability. Coupling logic and dependencies causes problems later when changing the dependency since that means that logic needs to be changed as well. This study will present the patterns for OOP and functional programming in their respective section.

3.1 Characteristics of Functional Programming

While different definitions exist of what Functional programming means, we define functional programming as a paradigm that uses of pure functions, decoupling state from logic, using trait-based polymorphism and immutable data.

Purity When a function is pure it means that calling a function with the same arguments will always return the same value and that it does not mutate any value. For example, given $f(x) = 2 \cdot x$, then $f(2)$ will always return 4. It follows then that an impure functions is either dependant on some state or mutates state in some way. For example, given $g(x) = currenttime \cdot x$, $g(5)$ will yield a different value depending on what time it is called. This makes it dependant on some state of the world. Or given $x = 0$, $h() = x + 1$. Then $h()$ will yield $x = 1$ and $(h \circ h)()$ will yield $x = 2$, making it impure. [13]

Trait-based polymorphism OOP inherits classes that contain methods and attributes. [14] Functional programs instead define classes that describe the actions that are possible. For example, a class `Equality` could contain a function `isEqual` that checks if two data types are equal. Then any data type that implements the interface Equality, for example lists or binary trees, would be able to use the function isEqual. This is known as type-classes in Haskell[1], mixins in Javascript[2] or traits in Scala[3].

Immutable data by deafult Immutable data is data that after initialization can not change. This means if we initialize a record, `abc = {a: 1, b: 2, c: 3}` then `abc.a = 4` is an illegal operation. Immutable data, along with purity, ensures that no data can be mutated unless it is specifically created as mutable data.

Decoupling state from logic Even if functional programs emphasise purity applications still need to deal with state somehow. For example a server would need to interact with a database. Functional programs solve this by separating pure functions and effectful functions. Effects are observable interactions with the environment, such as database access or printing a message. While various strategies exist, like Functional Reactive Programming[4], Dialogs[5] or

[1] `www.learnyouahaskell.com/types-and-typeclasses`
[2] `www.typescriptlang.org/docs/handbook/mixins.html`
[3] `https://docs.scala-lang.org/tour/traits.html`
[4] Read more: `en.wikipedia.org/wiki/Functional_reactive_programming`
[5] Read more: `stackoverflow.com/questions/17002119/haskell-pre-monadic-i-o`

uniqueness types[6], the one used in Haskell (the language used in this thesis to construct the programs) is the IO monad. For the uninitiated, one can think of Monads as a way to note which functions are pure and which are effectful and managing the way they intermingle. It enables handling errors and state.[7].

As a strategy to further separate state and logic, one can construct a three-layered architecture, called the three layer Haskell cake. Here, the strategy is that one implements simple effectful functions, containing no logic as a base layer. Then on a second layer one implements an interface that implements a pure solution and one effectful solution. Then on the third layer one implements the logic of the program in pure code. The way the second layer is implemented is explained further in Section 3.1.7.

So while no exact definition of Functional programming exist, this thesis defines it as making functions pure and inheritance being based around functionality rather than attributes.

3.1.1 ADTs: Sum types and product types

A type is in Haskell a *set* of possible values that a given data can have. This can be *int*, *char* and custom defined types. A *sum type* or *union type* is a type which is the sum of types, meaning that it can be one of those it's given types. For example the type `type IntChar = Int | Char` is either an Int or a Char. A useful application for sum types are enums such as `type Color = Red | Green | Blue`, meaning that a value of type Color is either red, green or blue. A sum type can be used to model data which may or may not have a value, by introducing the Maybe type: `type Maybe value = Just value | Nothing`.A product type is a type which is the product of types, for example `type User = User Name Email`. Informally, a product type can be likened to a record in Javascript. This allows us to model computations that might fail. For example given $sqrt(x) = \sqrt{x}$, $x \in \mathbb{Z}$ then $sqrt(-1)$ is undefined and would cause Haskell to crash. Instead by introducing a function `safeSqrt`, where `safeSqrt x = if x > 0 then Just (sqrt x) else Nothing`, the program can force the developer to handle the special case of negative numbers. Sum types are useful for implementing the Interpreter pattern, explained in Section 3.1.7.

[6]Read more: `https://en.wikipedia.org/wiki/Clean_(programming_language)`

[7]This is simplified as Monads are notoriously difficult to explain.

3.1.2 GADT

a GADT is a *generalized abstract data type*. They specify, depending on the input, what the output should be of that type. GADT enables implementing *domain-specific languages* (DSL). A DSL is a language with a limited scope for specific applications. For example a parsing library or a calculator.

```
data Calculator = Number Int
  | Add Calculator Calculator
  | Multiply Calculator Calculator
```

Figure 3.1: A Calculator GADT with two operations add and multiply.

```
mathExpression = (Number 5 ‘Add‘ Number 3) ‘Multiply‘ (
    Number 4 ‘Add‘ Number 3)
```

Figure 3.2: A mathematical expression constructed using the GADT in figure 3.1

Figure 3.1 defines a GADT for a calculator. The calculator can do two operations, add and multiply. This allows us to construct mathematical expressions. The expression in Figure 3.2 can translates to $(5+3)*(4+3)$ by defining a way to evaluate the expression. Figure 3.3 defines an evaluation for the program.

```
evaluate :: Calculator -> Int
evaluate (Add expr1 expr2) = evaluate expr1 + evaluate expr2
evaluate (Multiply expr1 expr2) = evaluate expr1 + evaluate
    expr2
```

Figure 3.3: Evaluator for the calculator

3.1.3 Type classes

A type class is a construct that allows for ad hoc polymorphism. This allows to create constraints to type variables in parametrically polymorphic types. In English, that means that it allows creating interfaces that must be implemented for the types. For example the equality type class, defined in Figure 3.4

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Figure 3.4: Equality type class in Haskell.

By defining an Equality type class one can create general functions that can be used for anything that is "equalable". For example Figure 3.5 is a function that prints a text if two items are equal. This function can be used for floats, ints, tuples and everything else that implements the `Eq` type class. Other uses for type classes is Num which implements numeric operations for floats and integers. This is useful for implementing the MTL technique which will allow us to implement the Interpreter pattern which will be described in the following sections.

```
printIfEqual :: Eq a => a -> a -> IO ()
printIfEqual a b =
    if a == b then
        putStrLn "They are equal"
    else
        putStrLn "They are not equal"
```

Figure 3.5: A function that prints a text if the two items are equal.

3.1.4 Functors and Contravariant Functors

A useful type class is the Functor type class. It defines a function $map : (a \to b) \to m\ a \to m\ b$. So every type that can be mapped over is a Functor. Examples of this are lists, where map morphs every value in the list from a to b. Another example is for Maybe, defined in 3.1.1. A Functor for Maybe checks if the value is $Just\ a$, if so it morphs that value to $Just\ b$, otherwise it returns $Nothing$.

Not every polymorphic type is a Functor. For example the type $Predicate = a \to Boolean$, is a function that when given some value a returns a boolean. This type can not be a functor due to the type parameter being the *input* of the function. When the polymorphic value of the type is the input it is in negative position and the type is *contravariant*. Similarity when the type parameter is the output of a function, it is in positive position and the type is covariant. A type can be a Functor if covariant.

Contravariant Functors are type classes that define a function $contramap : (a \to b) \to m\ b \to m\ a$. These are useful for defining how the value should be *consumed*. So for example a type $encoder : a \to encoded$, defines an encoder. The contravariant functor would allow transforming the encoder into intermediate value.

3.1.5 Brief introduction to Monads for side effects

Monads[8] are a way to sequence computations that might fail while automating away boilerplate code. Figure 3.6 shows how Monads are implemented as a typeclass in Haskell. It implements the function `return`, the function bind `(»=)`, the function sequence `(»)` which is bind whilst ignoring the prior argument and `fail` which handles crashes.

```
class Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
    (>>) :: m a -> m b -> m b
    fail :: String -> m a
    fail msg = error msg
```

Figure 3.6: Monad type class in Haskell.

Informally, Monads are as a design pattern that allows us to sequence different computations. Without them the developer would have to explicitly check if a computation has failed. For example, given the function $unsafeSqrtLog = sqrt \circ log$, then $unsafeSqrtLog(-1)$ would throw an error since log and $sqrt$ are undefined for -1. Section 3.1.1 showed how the `Maybe value` type could be used to create a safe computation `safeSqrt`. To sequence that computation with a function `safeLog`, the user would have to manually check that `safeSqrt` returned a value `Just result` and not `Nothing`. Monads allows sequencing these computations without explicitly writing this check, so composing `safeSqrt` and `safeLog` using bind becomes `safeSqrtLog n = safeSqrt n »= safeLog`. The same idea applies for effectful computations such as fetching data from a database.

[8]`en.wikipedia.org/wiki/Monad_(functional_programming)`

3.1.6 Strong Profunctors

Some types are both contravariant and covariant, such as the type *computation* $a\ b = a \to p\ b$, since computation has a type parameter in positive and negative position. A Profunctor is a type class that contains the function $dimap : (a \to b) \to (c \to d) \to p\ b\ c \to p\ a\ d$. *dimap* both contramaps and maps the type at the same time. So since *computation* is both covariant and contravariant it is also a profunctor.

A Strong Profunctor defines two functions, $first : p\ a\ b \to p\ (a, c)\ (b, c)$, $second : p\ a\ b \to p\ (c, a)\ (c, b)$. This reveals the core strength of Profunctors, to compute two separate values and then merge those together. By defining a function $merge : Strong\ p\ \Rightarrow p\ a\ (b \to c) \to p\ (a, b) \to c$, a Strong Profunctor gives a "memory" to the function, so that values can be computed both on the input and the output of the function. How this is useful will be demonstrated in the functional REST server chapter.

3.1.7 Interpreter pattern for testability

The beginning of this chapter briefly mentioned that design patterns are important to create maintainable software. In this study a design pattern called the Interpreter pattern will be used to structure functional programs. An interpreter is something which interprets input of some format, modifies it and transforms it into some output. Informally, the interpreter pattern is a way to create smaller composable compilers that when added together make one big application. A compiler is a program that takes some input, interprets the input and then does some output. A server, for example, would take some request, interpret that request and then turn it into a response. The server could integrate itself with the database, which would take some query, interpret that query and then return an object. [15]

To implement this pattern in Haskell, create an Abstract Syntax Tree (AST) using a sum type, of the program that contains all the available commands that the program is capable of doing. See Figure 3.7 for an example of a to-do list AST. Once we have the AST we can encode the logic of the program as instructions. Then the final step is to implement an interpreter for the program that evaluates those commands. So if we have the commands in Figure 3.7, we implement a function `eval` that takes a command and computes some effectful code. The command `Add Item (Item -> next)` could, for example, be executed as add an Item to a database.

```
        data TodoList next
             = Add Item (Item -> next)
             | Mark Item next
             | Remove Item next
             | End --^ Terminates the program
```

Figure 3.7: AST for a to-do-list. We can derive a functor instances from ASTs for deriving Free instances. [16]

Hiding the implementation behind an AST allows us to separate effectful code (like output a string or send a http request) with the logical instructions. This simplifies our testing, since we can hide the environment (for example database) behind an interface that we can swap out for testing. So we can implement two interpreter functions, one for our real environment and one for testing.

3.1.8 Using MTL for the interpreter pattern

Implementing interpreter pattern can either be done using sum types or a design pattern called MTL. The idea of MTL is to substitute dependencies by using a type class for one pure and one effectful instance. This allows for a pure instance that can be used for testing.

For example, if one wants to implement an authentication system for a server where the tokens expire within one week then one could do it as shown in Figure 3.8. Figure 3.8 is hard to test as it couples effectful code with pure code. The function `token` calls `Time.Posix.getPOSIXTime` which depends on the current time, making unit testing more difficult.

```haskell

addWeek :: POSIXTime -> POSIXTime
addWeek currentTime =
    currentTime + oneWeek
    where
        oneWeek = 604800000

token :: Key User -> IO (Maybe WebToken)
token user =
    do  currentTime <- Time.Posix.getPOSIXTime
        let expirationDate = addWeek currentTime
        let maybeToken = encode $ Token user expirationDate
        return maybeToken
```

Figure 3.8: Example of a function that, given the ID of a user, generates a unique token that can be used for authentication.

Instead, by using the MTL technique, one decouples the effectful code from it's dependencies. In Figure 3.8 the effectful code is the function `Time.Posix.getPOSIXTime`, which fetches the current time. So the type class will be a class `MonadTime` that contains one method `getTime`. By abstracting away the effectful code it becomes trivial to implement a pure and effectful instance. This is done in Figure 3.9.

```haskell
class MonadTime m where
    getTime :: m POSIXTime

instance MonadTime IO where
    getTime = Time.Posix.getPOSIXTime

instance MonadTime ((->) POSIXTime) where
    -- Allows us to call functions with the constraint MonadTime
    -- with an extra argument containing a mock value.
    getTime = id

addWeek :: POSIXTime -> POSIXTime
addWeek currentTime =
    currentTime + oneWeek
    where
        oneWeek = 604800000

token :: MonadTime m => Key User -> m (Maybe WebToken)
token user =
    do  currentTime <- getTime
        let expirationDate = addWeek currentTime
        let maybeToken = encode $ Token user expirationDate
        return maybeToken
```

Figure 3.9: An implementation of the token generation following MTL and interpreter pattern.

With the implementation in Figure 3.9, testing the function `token` becomes trivial with the instance `MonadTime ((->) POSIXTime)` as it allows one to substitute `getTime` with any value. Thus we separate effectful code from logic and mocking becomes trivial.

3.2 SOLID principles

A poorly written Object-oriented system can lead to rotten design. Martin Robert, a software engineer, claims that there are four big indicators of rotten design. Rotten design also leads to problems that were established in Chapter 2, such as making easy unit tests. Thus the Object-oriented system should follow the following indicators.

Rigidity is the tendency for software to be difficult to change. This makes it difficult to change non-critical parts of the software and what can seem like a quick change takes a long time.

Fragility is when the software tends to break when doing simple changes. It makes the software difficult to maintain, with each fix introducing new errors.

Immobility is when it is impossible to reuse software from other projects in the new project. So engineers discover that, even though they need the same module that was in another project, too much work is required to decouple and separate the desirable parts.

Viscosity comes in two forms: the viscosity of the environment and the viscosity of the design. When making changes to code there are often multiple solutions. Some solutions preserve the design of the system and some are "hacks". The engineer can therefore easily implement an unmaintainable solution. The long compile times affect engineers and makes them attempt to make changes that do not cause long compile times. This leads to viscosity in the environment.

To avoid creating rotten designs, Martin Robert proposes the SOLID guideline. SOLID mnemonic for five design principles to make software more maintainable, flexible and understandable. The SOLID guidelines are:

Single responsibility principle Here, responsibility means "reason to change". Modules and classes should have one reason to change and no more.

Open/Closed principle States we should write our modules to be extended without modification of the original source code.

Liskov substitution principle Given a base class and an derived class derive, the user of a base class should be able to use the derived class and the program should function properly.

Interface segregation principle No client should be forced to depend on methods it does not use. The general idea is that you want to split big interfaces to smaller, specific ones.

Dependency inversion principle A strategy to avoid making our source code dependent on specific implementations is by using this principle. This allows us, if we depend on one third-party module, to swap that module for another one should we need to. This can be done by creating an abstract interface and then instance that interface with a class that calls the third-party operations. [17]

Using a SOLID architecture helps make programs that are not as dependent on the environments, making them easier to test (swapping the production environment to a test environment becomes trivial). When investigating the testability, it is important to look at programs that are written in such a way that all parts are easy to test. Thus choosing a SOLID architecture for OOP based programs will allow making more testable software.

3.3 Measuring testability and complexity

The previous sections defined the programming paradigms and how to write software in a testable and maintainable manner. Since the aim of the study is to find which of the two paradigms allows us to write the most maintainable software, looking at the *Cyclomatic complexity* of software aloows finding the amount of tests needed to test every possible outcome of the software. It follows then that if one of the paradigms have a lower complexity, it would require less tests. If less tests need to be written, the amount of lines of code (LOC) needed for the software should be lower. Although further research would be needed to prove this, lower LOC correlates with less defects in software. [9] Note that LOC is a bad measurment for comparing two languages it is a good measure for comparing source code written in the same language.

Looking at the testability of code for maintenance is not enough. While testability is an important metric, it can also be important to look at other factors like how cognitive complexity affects the maintenance. For instance, a program written in the language Whitespace, an esoteric language consisting only of whitespace[9], could have a low LOC and cyclomatic complexity. But for humans knowing only the English language, a text consisting of whitespace would be illegible. By using a framework called Cognitive dimensions to do a qualitative evaluation of software one can qualitatively the structural software quality. This is described further in Section 3.3.3.

3.3.1 Measuring testability: Cyclomatic Complexity

Cyclomatic complexity is a complexity measure to measure the amount of paths through a program. The Cyclomatic complexity number is an upper bound for the number of test cases required for full branch coverage of the code.

[9]Description of Whitespace: `https://en.wikipedia.org/wiki/Whitespace_(programming_language)`

Definition. *The cyclomatic number $v(G)$ of a graph G with n vertices, e edges and p connected components is $v(G) = e - n + p$.*

Theorem. In a strongly connected graph G, the cyclomatic number is equal to the maximum number of linearly independent circuits. [18]

Informally, cyclomatic complexity is a way to measure the amount tests a program needs to reach full branch coverage by constructing a graph that branches out based on the control flow. For example, given `f(bool) = if bool then ‘‘hello’’; else ‘‘hola’’` the function `f` will either evaluate to “hello” or “hola”. To get full branch coverage for `f` two tests are needed. The cyclomatic complexity therefore becomes 2. The nodes of a cyclomatic graph represents processing tasks and edges represent control flow between the nodes.

Given the code found in example figure 3.10. To calculate the complexity of this function, first construct a graph as seen in figure 3.11. From the graph we find $n = 4, e = 5, p = 2 \Rightarrow v(G) = e - n + p = 5 - 4 + 2 = 3$ is the cyclomatic number.

```
void foo(void)
{
  if (a)
    if (b)
      x=1;
  else
      x=2;
 }
```

Figure 3.10: Multi if function foo.

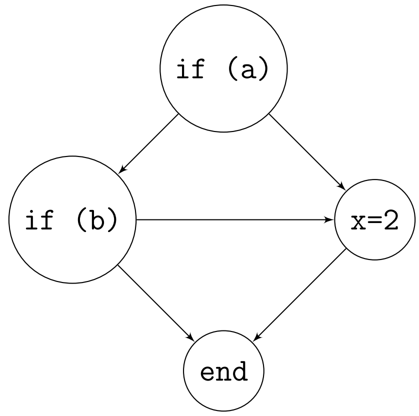

Figure 3.11: Cyclomatic complexity graph for Figure 3.10

3.3.2 Cyclomatic Complexity in Functional Programming

The definition of cyclomatic complexity in Section 3.3.1 is not ideal for functional programming. Cyclomatic complexity is calculated by creating graphs based on control flow operations such as while loops and if statements. In functional programming everything is a function, thus the cyclomatic complexity will always tend to 0 using this definition. So we define a different method of calculating the cyclomatic complexity for functional programs.

Definition. *The cyclomatic complexity number, in functional programming, is equal to 1 plus the sum of the left hand side, called LHS, plus the sum of the right hand side, called RHS. RHS is the sum of the number of guards, logical operators, filters in a list comprehension and the pattern complexity in a list comprehension. LHS is equal to the pattern complexity. The pattern complexity is equal to the number of identifiers in the pattern, minus the number of unique identifiers in the pattern plus the number of arguments that are not identifiers. In summary:*

```
Cyclomatic complexity = 1 + LHS + RHS

LHS = Pattern complexity

Pattern complexity
    = Pattern identifiers
    - Unique pattern identifiers
    + Number of arguments that are non identifiers

RHS = Number of guards
```

```
          + Number of Logical operators
          + Number of filters in list comprehension
          + Pattern complexity in list comprehension
```

Instead of cyclomatic graphs in functional programs one constructs flowgraphs, such as the one seen in Figure 3.13, to model the functions.

```haskell
split :: (a -> Bool) -> [a] -> ([a], [a])
split onCondition [] = ([], [])
split onCondition (x:xs) =
    let
        (ys, zs) = split onCondition xs
    in
        if (onCondition x) then
            (x:ys, zs)
        else
            (ys, x:zs)
```

Figure 3.12: Recursively split a list into two based on a given condition in Haskell. For example `split (>3) [1,2,3,4,5] = ([4,5],[1,2,3])`.

In Haskell $(x : xs)$ denotes an item x at head of a list of items xs. Given the Haskell code in Figure 3.12. To calculate LHS we find two pattern identifiers which are $onCondition$ and $(x : xs)$. there is one unique pattern identifiers which is $(x : xs)$. There is also one non identifier which is []. There is also one guard, an if statement, and no list comprehensions on RHS. Thus the cyclomatic complexity is $1 + (2 - 1 + 1) + 1 = 4$.

In this method to calculate cyclomatic complexity, do not count the *otherwise* and *else* clauses. [10]

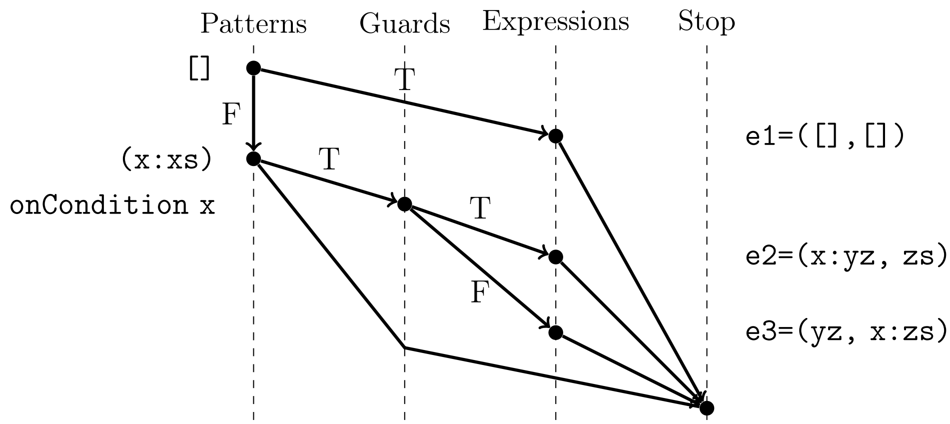

Figure 3.13: Flowgraph for split function, defined in Figure 3.12.

Cyclomatic complexity finds how many tests are needed to get full branch coverage of source code. A low complexity for a program means that less tests have to be written. Note that if a program has a low cyclomatic complexity it does not imply that the program is easier to test. If a program is written in such a way that it depends heavily on the environment it can also lead to difficulty testing. So to make cyclomatic complexity a better metric this study will use design patterns to make testable code.

3.3.3 Mental complexity: Cognitive Dimensions

Cognitive Dimensions is a framework for evaluating the usability of programming languages and to find areas of improvements. [19] It allows us to evaluate the quality of a design and explore what future improvements could be made. As part of the Cognitive Dimensions, 14 different Cognitive Dimensions of Notation exist. A notation depends on the specific context, in this case the notation is the languages themselves and their architecture. The author of the framework recommends omitting the dimensions that are not applicable to the notation. We give a brief description of the dimensions.

Viscosity How much work does it take to make small changes? How easy is the code to refactor? If small changes requires consequent adjustments then that is a problem. As a viscous system cause a lot more work for the user and break the line of thought.

Visibility How easy is it to navigate the source code to find the parts that you want?

Hidden dependencies Are there hidden dependencies in the source code. Does a change in one part of the source code lead to unexpected consequences in another part of the code. Every dependency that matters to the user should be accessible in both directions.

Role-expressiveness How obvious is each sub-component of the source code to the solution as a whole?

Abstraction What are the levels of abstraction in the source code? Can the details be encapsulated?

Secondary notation Are there any extra information being conveyed to the user in the source code?

Closeness of mapping By looking at the source code, how close do we find it to be to the case we are solving?

Consistency Once Object-oriented procedural programming and Functional programming has been learned. How much of the rest can the user guess successfully?

Diffuseness or terseness How much space and symbols does the source code need to produce a certain result or express a meaning?

Hard mental operations Where does the hard mental processing lie? Is it more when writing the source code itself rather than solving the case, I.E. the semantic level? Does one sometimes need to resort to pen and paper to keep track of what is happening?

Provisionality How easy is it to get feedback of something before you have completed the entire system?

Progressive evaluation How obvious the role of each component of the source code in the solution as a whole?

Error proneness To what extent does the programming paradigm and language help minimise errors? Are there any structures or complexities that lead to it being easier to make errors?

For this study we will investigate the following dimensions:

- Diffuseness or terseness
- Progressive evaluation
- Closeness of mapping
- Hard Mental Operations
- Visibility
- Hidden dependencies
- Abstraction
- Error-proneness

We omit the other dimensions as related work concluded that the other dimensions did not bring much weight when evaluating the different paradigms. [12]

In summary, cognitive dimensions allow us to look at different aspects of a programming language to evaluate how complex they are cognitively.

3.4 Functional servers

3.4.1 RESTful servers

Servers are applications that provide functionality for other programs or devices, called clients. Services are servers that allow sharing data or resources among clients or to perform a computation.

REST (Representational State Transfer) is a software architecture style that is used to construct web services. A so called RESTful web service allow requesting systems to access and manipulate textual representations of web services by using a set of stateless operations. The architectual constraints of REST are as follows:

Client - Server Architecture Separate the concerns between user interface concerns and data storage concerns.

Statelessness Each request contains all the information neccessary to perform a request. State can be handled by cookies on the user side or by using databases. The server itself contains no state.

Cacheability As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable or not to prevent clients from getting stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.

Layered system A client can not tell if it is connected to an end server or some intermeditary server.

Code on demand Servers can send functionality of a client via exectuable code such as javascript. This can be used to send the frontend for example.

Uniform interface The interface of a RESTful server consists of four components. The request must specify how it would like the resource to be represented; that can for example be as JSON, XML or HTTP which are not the servers internal representation. Servers internal representation is therefore separated. When the client holds a representation of the resource and metadata it has enough information to manipulate or delete the resource. Also the REST server has to, in it's response, specify how the representation for the resource. This is done using Media type. Some common media types are JSON, HTML and XML.

A typical HTTP request on a restful server consists of one of the verbs: GET, POST, DELETE, PATCH and PUT. They are used as follows:

GET Fetches a resource from the server. Does not perform any mutation.

POST Update or modify a resource.

PUT Modify or create a resource.

DELETE Remove a resource from the server.

PATCH Changes a resource.

A request will specify a header "Content-Type" which contains the media representation of the request content. For example if the new resource is represented as Json then content-type will be "application/json". It also specifies a header "Accept" which informs which type of representation it would like to have, for example Html or Json.

A request will also contain a route for the resource it is requesting. These requests can also have optional parameters called query parameters. In the request route:

```
/api/books?author=Mary&published=1995
```

the ? informs that the request contains query parameters which are optional. In the example above it specfies that the request wants to access the books resource with the parameters author as Mary and published as 1995.

When a request has been done the server responds with a status code that explains the result of the request. The full list of status codes and their descriptions can be found here: `https://en.wikipedia.org/wiki/List_of_HTTP_status_codes`

3.4.2 Implementation concerns for REST apis

A REST api has to concern themselves with the following:

- Ensure that the response has the correct status code.
- Ensure that the correct representation is sent to the client.
- Parse the route and extract it's parameters.
- Parse the query and extract it's parameters.
- Handle errors if the route or query are badly formatted.
- Generate the correct response body containing all the resources needed.

Every type of error has a specific status code, these need to be set correctly.

3.5 Formal implementation of a server in functional programs

A server is a function that takes a request of parameter a and transforms it into a response of parameter a. I.E. $Server : Request\ a \to Response\ b$. The parameter a is the resource requested by the client.

A *Response* is a record consisting a status code, a set of headers, a content type, a function $body : a \to encoded$, encoding.

```
type Response a = {
  code: StatusCode,
  headers: Header ,
  contentType: MediaType,
  body: a -> encoded,
  encoding: Encoding.t,
};
```

The body of a response is a function that transforms the resource into it's requested representation. If the request specifies accept as `application/json` then the body function turns the body into a json format.

A *Request* is a monad that parses the incoming request. It transforms that into a request handler that then feeds the result into a response. A handle is a function $a \rightarrow Responseb$. The extended definition of the server is then $Server : Requesta \rightarrow (a \rightarrow Responseb) \rightarrow Responseb$.

3.6 Cause, a functional REST framework

Cause is a high-level web framework that allows writing composable REST frameworks in Reasonml. In Cause, a REST api is a *specification*, where the user specifies endpoints. An endpoint in a spec contains the following:

- An ordered set of required parameters to execute a handler.
- An ordered set of optional query parameters to execute a handler.
- A set of accepted representations for a resource as well as an accompanying function that encodes the resource to the represenation.
- A handler to execute, where a handler must take the correct parameters and return the correct resource.
- A set of http request methods supported by the endpoint.

An endpoint is composable, meaning that you can create a *connector endpoint* which consists of two *subendpoints*. The following operations exist for endpoints

oneOf : $list(endpoint)$ Creates a connector endpoint out of a list of subendpoints.

is : $string \rightarrow endpoint$ Checks that the request contains a given string in it's path.

int : $endpoint$ Extracts an integer from the request path and feeds it into the handler.

text : $endpoint$ Extracts a text from the request path and feeds it into the handler.

contentType : $list(MediaType, a \rightarrow encoded)$ Checks that the request contains one of the supported media types and sets the appropriate encoder.

accept : $list(MediaType, encoded \to a)$ Takes a set of supported mediatypes and a way to transform the encoded value into the value.

($\rightarrowtail$) : $endpoint \to endpoint \to endpoint$ An operator for composing endpoints.

get : $handler \to endpoint \to endpoint$ Connects an endpoint to a handler and ensures that endpoint accepts only GET.

delete : $handler \to endpoint \to endpoint$ Connects an endpoint to a handler and ensures that endpoint accepts only DELETE.

put : $handler \to endpoint \to endpoint$ Connects an endpoint to a handler and ensures that endpoint accepts only PUT.

post : $handler \to endpoint \to endpoint$ Connects an endpoint to a handler and ensures that endpoint accepts only POST.

Example 3.6.1. A RESTful api User manages user data for a server at the path `/user`. It has the handlers $getUser : id \to list(User)$, $postUser : User \to result$, $deleteUser : id \to result$. The user endpoint then becomes

```
userSpec = is("user") ↣ oneOf(
    [int ↣ contentType([(Json,encodeJson)]) ↣ get(getHandler),
    ,accept([(Json,decodeJson)]) ↣ post(postHandler),
    ,int ↣ delete(deleteHandler),
    ])
```

Chapter 4

Method: Case studies

Chapter 3 defined what programming paradigms are and how to evaluate their testability and their cognitive complexity using Cyclomatic complexity and Cognitive dimensions. Section 3.3 also explained how Cyclomatic complexity and Cognitive dimensions is tied to the maintainability and testability of software. The aim of this study is to find if the functional paradigm or the OOP paradigm is more maintainable than the other and in which situations. This study will compare the cyclomatic complexity and cognitive complexity by looking at three different cases. If the Cyclomatic complexity is lower in one of the paradigms or the other, then the amount of tests write for full branch coverage will be lower for that paradigm than the other. Evaluating the paradigms using cognitive dimensions finds if one paradigm is easier to maintain for the developer. The cases were chosen based on how they are often subproblems in bigger applications. The solutions for functional programming will be implemented using Haskell and for OOP Java will be used.

Java is a programming language that is class-based and object-oriented. The aim of the language is that you should be able to write the code and run it anywhere. [3] Haskell is a purely functional programming language, it also features lazy evaluation which allows composing functions easier. For example, given the function *three* where $three = take\ 3 \circ cycle\ 5$, where cycle is a function that generates an infinitely long list consisting of a number. *three* only compute the first three values, I.E. $[5, 5, 5]$, of the infinite list. In a non-lazy program the function would never terminate as *cycle* would run forever. [20] The reason these languages where chosen is because of the authors familiarity with those languages.

Simplified chess game

Chess is a famous game and in this report it is assumed that the reader know how it works.[1] The aim is to implement a simplified variant of it.

- Only pawns and horses exist.
- You win by removing all the other players pieces.

The player should be able to do the following:

- List all available moves for a certain chess piece.
- Move the chess piece to a given space
- Switch player after move
- Get an overview of the board
- Get an error when making invalid moves

To interact with the game, a user types commands through a command line prompt. An example of basic interaction with the game is displayed in Figure 4.1.

```
> list (a,2)
White pawn at (a,2)
> move (a,2) (a,3)
White pawn moved to (a,3)
> listall
White pawn at (a,3)
White pawn at (b,2)
...
White Horse at (a,1)
...
Black pawn at (a,8)
...
```

Figure 4.1: An example of the interaction in the chess game.

[1] Rules of chess: `en.wikipedia.org/wiki/Rules_of_chess`

To-do List

A common task in programming is to create some kind of data store with information. A to-do list is a minimal example of that. It consists of a list of items that can be used to remember what to do later. The user should be able to:

- Create a new item in the to-do list.
- Remove an item from the to-do list.
- Mark an item from the to-do list as done.
- See all items in the to-do list.

The interface to this program will be a text interface, displaying each item in the todo list. To navigate and mark an item in the list the user presses up and down and presses `x` to mark it.

Kudos bot

When working in big teams we need to automate common tasks. A common application for communication is Slack[2]. The task here is to create an bot that allows people to send kudos to collegues when they have done a good job

- The user should be able to type "/kudos Username 'msg"' where username is the name of the person they want to thank and msg is the message.
- The receiving user should be notified
- By typing "/kudos all" the user should be able to see the score of all kudos that have been given.

[2] See more: `www.slack.com`

Chapter 5

Results

Once these programs are constructed then cyclomatic complexity can be evaluated by summing the cyclomatic complexity of each function. We can also evaluate the cognitive complexity using the cognitive dimensions framework. Thus we get can see if smaller subproblems in big applications require more tests and have a bigger mental complexity depending on the different paradigms.

5.1 Limitations

TODO

5.1.1 Improvements to implementation

TODO

5.2 future work

5.2.1 Relations to cardinality

TODO

Bibliography

[1] T. Ishida, Y. Sasaki, and Y. Fukuhara, "Use of procedural programming languages for controlling production systems," in *[1991] Proceedings. The Seventh IEEE Conference on Artificial Intelligence Application*, vol. i, pp. 71–75, Feb 1991.

[2] A. M. Turing, "On computable numbers, with an application to the entscheidungsproblem," *Proceedings of the London Mathematical Society*, vol. s2-42, no. 1, pp. 230–265, 1937.

[3] G. S. G. B. A. B. James Gosling, Bill Joy, "The java language specification," Febuary 2015. `https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf`, accessed Feb 2018.

[4] D. A. Turner, "Some history of functional programming languages," in *Proceedings of the 2012 Conference on Trends in Functional Programming - Volume 7829*, TFP 2012, (New York, NY, USA), pp. 1–20, Springer-Verlag New York, Inc., 2013.

[5] B. J. Copeland, "The church-turing thesis," in *The Stanford Encyclopedia of Philosophy* (E. N. Zalta, ed.), Metaphysics Research Lab, Stanford University, winter 2017 ed., 2017.

[6] R. S. Pressman, *Software Engineering: A Practitioner's Approach.* New York, NY, USA: McGraw-Hill, Inc., 6 ed., 2005.

[7] M. Wacker, "Just say no to more end-to-end tests," 2015. `https://testing.googleblog.com/2015/04/just-say-no-to-more-end-to-end-tests.html`, accessed Mar 2019.

[8] A. Schaffer, "Testing of microservices," 2018. `https://labs.spotify.com/2018/01/11/testing-of-microservices/`, accessed Mar 2019.

[9] H. Zhang, "An investigation of the relationships between lines of code and defects," pp. 274–283, 10 2009.

[10] K. Berg, "Software measurement and functional programming," *Current Sociology - CURR SOCIOL*, 01 1995.

[11] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, pp. 476–493, June 1994.

[12] E. Kiss, "Comparison of object-oriented and functional programming for gui development," master's thesis, Leibniz Universität Hannover, August 2014.

[13] "Pure function," Aug 2018. `https://en.wikipedia.org/wiki/Pure_function`, accessed Feb 2019.

[14] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[15] Source Making, "Interpreter design pattern." `https://sourcemaking.com/design_patterns/interpreter`, accessed Feb 2019.

[16] "Support for deriving functor, foldable, and traversable instances." `https://ghc.haskell.org/trac/ghc/wiki/Commentary/Compiler/DeriveFunctor`, accessed Feb 2019.

[17] M. C. Robert, "Design principles and design patterns," September 2015. originally `objectmentor.com`, archived at `https://fi.ort.edu.uy/innovaportal/file/2032/1/design_principles.pdf`, accessed Feb 2018.

[18] T. J. McCabe, "A complexity measure," in *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, (Los Alamitos, CA, USA), pp. 407–, IEEE Computer Society Press, 1976.

[19] T. Green and M. Petre, "Usability analysis of visual programming environments: A 'cognitive dimensions' framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131 – 174, 1996.

[20] "Haskell." `https://www.haskell.org/`, accessed Feb 2018.