Comparing Testability and Usability 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
February 10, 2019

Abstract

This study's goal is to compare approaches to functional programs and object-oriented programs to find how it affects maintainability and code 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

I want to thank myself for my amazing work. TODO

Contents

1 Introduction

This is time for all good men to come to the aid of their party! TODO

2 Objective

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. When building applications testability is of high concern to ensure that the application functions properly. By looking at cyclomatic complexity, described in Section 4.1, we can find out how the different approaches affect the amount of tests we need to write to get full branch coverage. By looking at the cognitive dimensions, described in Section 4.2, we can find how the two approaches affect the mental complexity for the developer. So by looking at different case studies this study aims to find if the different software paradigms affect the cyclomatic complexity and if there are any cognitive benefits to one approach over the other.

3 Theory

3.1 Characteristics of Functional Programming

Functional programming is a software paradigm. While different definitions exist of what it means, we define functional programming as:

- Removing state to the outwards of the program
- Make functions pure, produce no side effects
- Using trait-based polymorphism rather than class-based

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 if you have $f(x) = 2 \cdot x$, 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. [1]

3.1.1 Interpreter pattern

When structuring large functional applications there is no official way to do it. A design pattern that we will use for this study is the Interpreter pattern. Informally, we can think of it as 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.

To do this in Haskell we create an Abstract Syntax Tree (AST), using a union type, of our program that contains all the available commands that this program is capable of doing. Afterwards we create an interpretor of that AST that then outputs the program. This 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 make sure that the right instructions is returned when testing a function without calling effectful code, like database access.

Note: *This part assumes familiarity with Haskell and uninitiated can ignore it.*

3.2 Free Monads

Free Monads To give an example, say we want to create an application, a To-do-list. In this To-do-list the user should be able to add an item, remove an item and mark an item as complete. We then define the union type in Figure 3.2. [2]

```
1        data TodoList next
2             = Add Item (Item -> next)
3             | Mark Item next
4             | Remove Item next
5             | End --^ Terminates the program
```

Figure 1: AST for a to-do-list. We can derive a functor instances from ASTs for deriving Free instances. [3]

This allows us to create a "domain specific language" (DSL) for our program. So now we can create instructions, for example `addAndMark item =`

`Add item $ \item -> Mark item End`. Then from that we can create an interpreter for that DSL. To do so we create a Free Monad instance of our AST. In order to instance a Monad we first need a Functor instance, seen in Figure 3.2.

```
instance Functor TodoList where
  fmap f (Add item k) =
    Add item (f . k)
  fmap f (Mark item next) =
    Set item (f next)
  fmap f End =
    End
```

Figure 2: An Functor instance for Figure 3.2. This can automatically be derived using DeriveFunctor.

3.3 Object Oriented Programming

Object-oriented programming (OOP) is a programming paradigm based around the idea of objects that contain attributes and methods. The general idea is that you model the world through objects, where the programming objects represent the objects in the real world. A shopping cart might have an object "Customer" and an object "Shopping list", for example. Inheritance is also an important concept in OOP, where one object can inherent methods and attributes of another object. For example, given an object Animal with the attributes health and the method attack, another object Alligator could inherit those properties. [4]

3.3.1 SOLID principles

If the design of a system is done wrongly, it can lead to rotten design. Martin Robert, a software engineer, claims that there are four big indicators of rotten design.

First one is rigidity. 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.

Second is fragility. 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.

Third is immobility. 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.

Last is viscosity. Viscosity comes in two forms: the viscosity of the environment and the viscosity of the design. When the engineer makes changes they are often met with multiple solutions. Some that preserve the design and some that are "hacks". So the engineer can easily do the wrong thing.

When the compile times are long the engineers will 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 guidelines. 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. [5]

When we compare our solutions we want to test them after principles used in the industry, as then we can use the results when making technical decisions for our softwares.

4 Methods

4.1 Measuring testability: Cyclomatic Complexity

Cyclomatic complexity is a complexity measure that allows us 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.

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. [6]

Informally, we can think of cyclomatic complexity as a way to measure the amount tests a program needs to reach full branch coverage. We construct a graph that branches out based on when the control flow in our source code branches out. For example, given `f(bool) = if bool then 1; else 2`, the function `f` will either be 1 or 2. The function `f` will need two tests in order to have full code coverage. The cyclomatic complexity in this case is 2. The nodes of the graph represents processing tasks and edges represent control flow between the nodes.

```
1         void foo(void)
2         {
3           if (a)
4             if (b)
5               x=1;
6           else
7               x=2;
8          }
```

Figure 3: Multi if function foo.

If we have the code found in example figure 3. To calculate the complexity of this function we first construct a graph as seen in figure 4. 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.

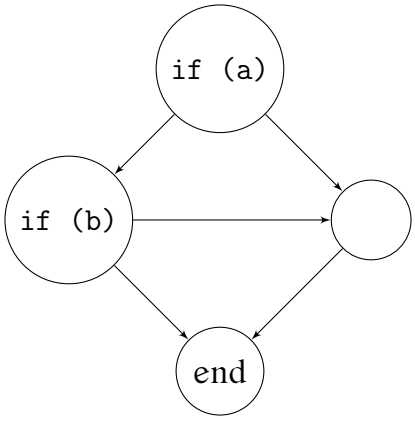

Figure 4: Cyclomatic complexity graph for Figure 3

4.1.1 Cyclomatic Complexity in Functional Programming

The definition of cyclomatic complexity in Section 4.1 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:*

```
1       Cyclomatic complexity = 1 + LHS + RHS
2
3       LHS = Pattern complexity
4
5       Pattern complexity
6           = Pattern identifiers
7           - Unique pattern identifiers
8           + Number of arguments that are non identifiers
9
10      RHS = Number of guards
```

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

Instead of cyclomatic graphs we instead construct flowgraphs, such as the one seen in Figure 4.1.1 to model our function.

```
1     split :: (a -> Bool) -> [a] -> ([a], [a])
2     split onCondition [] = ([], [])
3     split onCondition (x:xs) =
4         let
5             (ys, zs) = split onCondition xs
6         in
7             if (onCondition x) then
8                 (x:ys, zs)
9             else
10                (ys, x:zs)
```

Figure 5: 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 4.1.1. 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 []. We also find one guard, an if statement, and no list comprehensions on RHS. Thus the cyclomatic complexity is $1 + (2 - 1 + 1) + 1 = 4$. <- NEEDS VERIFICATION FROM SOMEONE, THE SOURCE IS VERY VAGUE HERE...

We do not count the $otherwise$ and $else$ clauses as a guard, just as how we do not count the $else$ statement in normal procedural cyclomatic complexity. [7]

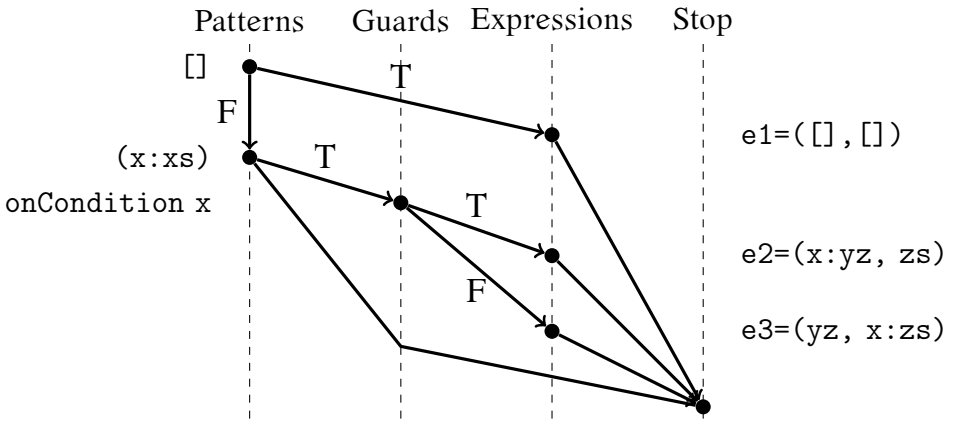

Figure 6: Flowgraph for split function, defined in Figure 4.1.1.

4.2 Mental complexity: Cognitive Dimensions

Cognitive Dimensions is a framework for evaluating the usability of programming languages and to find areas of improvements. Used as an approach to analyse the quality of a design, it also allows to explore what future designs could be possible. As part of the Cognitive Dimensions, 14 different Cognitive Dimensions of Notation exist. A notation depends on the specific context, in this case will be the languages themselves and their architecture.

4.2.1 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.

4.2.2 Visibility

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

4.2.3 Premature Commitment

Are there any architectural decisions that must be made before all the necessary decisions have been made? Can we correct those decisions later, how

safe is the refactoring and how much is needed? If you imagine a word processor which required how many pages of text would be used before being written then that would be premature commitment.

4.2.4 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.

4.2.5 Role-Expressiveness

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

4.2.6 Abstraction

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

4.2.7 Secondary Notation

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

4.2.8 Closeness of Mapping

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

4.2.9 Consistency

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

4.2.10 Diffuseness or terseness

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

4.2.11 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? Do you sometimes need to resort to pen and paper to keep track of what is happening?

4.2.12 Provisionality

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

4.2.13 Progressive Evaluation

How obvious the role of each component of the source code in the solution as a whole? [8]

4.3 Case studies

To compare the different paradigms using Cognitive dimensions and measuring Cyclocmatic complexity we will look at three different cases. The cases were chosen based on how they are often subproblems in bigger applications.

4.3.1 Simplified chess game

Chess is a famous game and assumed that the reader know how it works. Aim is to implement a simplified variant of it. This is not ordinary chess but a simplified version:

- 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 4.3.1.

```
> 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 7: An example of the interaction in the chess game.

4.3.2 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.

4.3.3 Chatbot engine

Oftentimes when developing applications we have to deal with complex information input. One of those cases is when we have chat bots. Chat bots are interactive programs that respond with a text answer to the users input. For this application we will implement the following:

- Interpretor that can handle semi-complex inputs and deal with errors.
- Give answers to those inputs in form of text messages.

5 Results

We worked so hard, yet achieved very little. TODO

6 Limitations

TODO

6.1 Improvements to implementation

TODO

7 future work

7.1 Relations to cardinality

TODO

References

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

[2] W. Swierstra, "Data types à la carte," *J. Funct. Program.*, vol. 18, pp. 423-436, July 2008.

[3]

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.