Comparing Maintainability 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 12, 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

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. When building applications testability is of high concern to ensure that the application functions properly. By looking at cyclomatic complexity, described in Section 2.3.1, 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 2.3.3, 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 testability and if there are any cognitive benefits to one approach over the other.

Chapter 2

Theory

Software paradigms are ways to construct software. There exists many but the two we will look at are functional programming and object-oriented programming. Historically and today OOP has been the most popular language for the commercial software industry. Java is an OOP language that is, in February 2019, the most popular language in the software industry. Languages that are functional, like Haskell and F#, are less used. [1] However functional concepts have become more popular in multi-paradigm languages like Javascript. For example React, based on Functional Reactive Programming [2], has gained a lot of popularity. [3]

Exact definitions exist of OOP but not for Functional programming. Both emphasise creating bug free programs. OOP emphasises encapsulating 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 (defined in Section 2.1) and using immutable data. Immutable data is data whose state can not change once initialized. This is explained further in Section 2.1 and Section 2.2.

While paradigms define how we build our applications, we still need design patterns for structuring the source code. For instance, without design patterns it is possible to couple dependencies with logic. This causes problems if we later want to change the dependency since that means we also have to change the logic. To increase maintainability, we want to use design patterns. This study will present the patterns for OOP and functional programming in their respective section. When you create applications that you use once then maintainability is not of high concern. However since the design patterns might affect the metrics we will be looking at, we still employ them in this study.

2.1 Characteristics of Functional Programming

Functional programming is a software paradigm. While different definitions exist of what it means, we define functional programming by the use of pure functions, moving state to the outwards of the program, 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 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. [4]

Trait-based polymorphism In OOP we inherit classes that contain methods and attributes. [5] For functional programs, we 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, mixins in Javascript or traits in Scala.

Immutable data Immutable data is that that upon being initialized can not be changed.

Moving state to the outwards of the program Even if functional programs emphasise purity certain 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. While various strategies exist, like Functional Reactive Programming[1], Dialogs[2] or uniqueness types[3], 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 also allows us to handle errors and state.[4].

[1]Read more: `en.wikipedia.org/wiki/Functional_reactive_programming`
[2]Read more: `stackoverflow.com/questions/17002119/haskell-pre-monadic-i-o`
[3]Read more: `https://en.wikipedia.org/wiki/Clean_(programming_language)`
[4]This is simplified as Monads are notoriously difficult to explain.

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 2.1.1.

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

2.1.1 Interpreter pattern for testability

When structuring large functional applications there is no official way to do it. For this study we will use a design pattern called 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. The server could integrate itself with the database, which would take some query, interpret that query and then return an object. [6]

To implement this pattern in Haskell we create an Abstract Syntax Tree (AST), using a union type, of the program that contains all the available commands that the program is capable of doing. See Figure 2.1.1 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 2.1.1, 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.

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

Figure 2.1: AST for a to-do-list. We can derive a functor instances from ASTs for deriving Free instances. [7]

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.

2.1.2 Extending applications with Free Monads

If we implement an interpreter for an AST we will run into the Expression problem. The expression problem states:

"The goal is to define a data type by cases, where one can add new cases to the data type and new functions over the data type, without recompiling existing code, and while retaining static type safety."

This means, if we try to add another operation to our AST we would have to add new cases to every function using that data type. For small programs this is not a problem but as this scales it could mean adding cases to thousands of functions. To resolve this we will create a Free Monad for our AST. A Free Monad is a Monad that we get "for free" if we instance an AST as a Functor. With the do-notation a Free Monad gives an interface we can use to write our program. So we start off by instancing it as a Functor (example in Figure 2.1.2). Functors for ASTs can automatically be derived using the DeriveFunctor extension. [7]

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

```
1          instance Functor TodoList where
2            fmap f (Add item k) =
3              Add item (f . k)
4            fmap f (Mark item next) =
5              Set item (f next)
6            fmap f End =
7              End
```

Figure 2.2: A Functor instance for Figure 2.1.1. This can automatically be derived using DeriveFunctor.

From this we can define `type FreeAST = Free FunctorAst`. Lastly we lift all the members of the union type as operations and create an interpreter. See Figure 2.3 for a full example.

```
type TodoListF = Free TodoList

add :: Item -> TodoListF Item
add item =
    liftF $ Add item id
mark :: Item -> TodoListF ()
mark item =
    liftF $ Mark item ()
end :: TodoListF r
end =
    liftF $ End

eval :: TodoListF r -> IO r
eval (Pure r) = return r
eval (Free (Add item next)) = addItemToDb item >>= eval next
eval (Free (Mark item next)) = markItem item >> eval next
eval (Free End) = exitSuccess

fakeItem :: Item

exampleProgram :: TodoListF ()
exampleProgram =
    do  item <- add fakeItem
        mark item
        end

main =
    eval exampleProgram
```

Figure 2.3: An example implementation for our To-do list data type defined in Figure 2.1.1. Requires the haskell package "free" (`hackage.haskell.org/package/free-5.1`)

So to summarize, if we just implement an evaluate function for our AST we run into the expression problem. To mitigate this we create Free monads from our ASTs and to get an extendable interface that we can use with the do-notation. By using the interpreter pattern, our program becomes independent from the environment that it runs on. This allows us to easier test all parts of the program.

2.2 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 attribute health and the method attack, another object Alligator could inherit those properties. [5]

2.2.1 SOLID principles

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

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.

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.

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.

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 guideline. SOLID mnemonic for five design principles to make software more maintainable, flexible and understandable. The SOLID guideline is

- 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. [8]

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

2.3 Measuring testability and complexity

With the definitions of the software paradigms, we can now look at how we can evaluate them in regards of testability and complexity. Since we want to find which of the two paradigms allows us to write the most maintainable software, it is important that we have testable software so that finding defects is easier. While we can not measure testability, by looking at the *Cyclomatic complexity* of our software we can find out how many tests would be 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, it is logically sound to assume that the amount of lines of code (LOC) for our software would be lower. Lower LOC correlates with less defects in software. [9]

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[5], could have a low LOC and cyclomatic complexity. But for humans knowing only the English language, a text consisting of whitespace would be illegible. While measuring cognitive complexity is hard, we can use a framework called Cognitive dimensions to do a qualitative evaluation of software. We describe this further in Section 2.3.3.

2.3.1 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. [10]

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 ‘‘hello’’; else ‘‘hola’’` the function `f` will either evaluate to “hello” or “hola”. To get full branch coverage for `f` we will therefore need two tests. The cyclomatic complexity then becomes 2. If we construct a graph for the control flow of our program, the nodes of the graph would represent processing tasks and edges represent control flow between the nodes.

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

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

Figure 2.4: Multi if function foo.

If we have the code found in example figure 2.4. To calculate the complexity of this function we first construct a graph as seen in figure 2.5. 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.

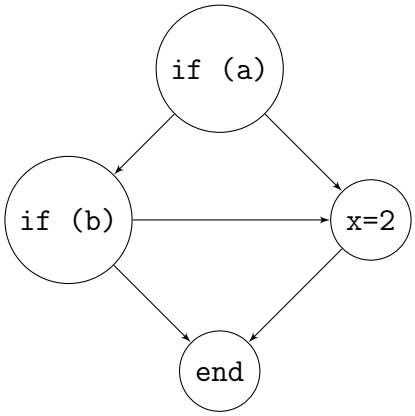

Figure 2.5: Cyclomatic complexity graph for Figure 2.4

2.3.2 Cyclomatic Complexity in Functional Programming

The definition of cyclomatic complexity in Section 2.3.1 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 we instead construct flowgraphs, such as the one seen in Figure 2.3.2 to model our function.

```
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 2.6: 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 2.3.2. 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. [11]

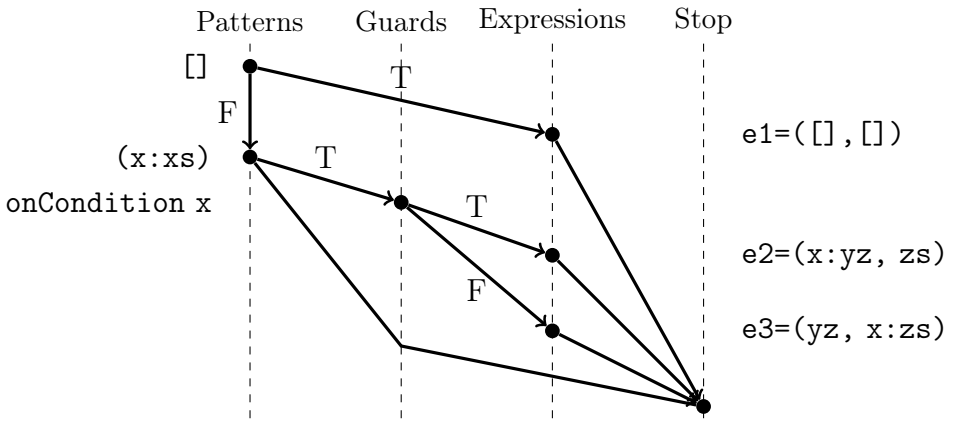

Figure 2.7: Flowgraph for split function, defined in Figure 2.3.2.

Cyclomatic complexity allows us to look at programs in a quantitative way to find how many tests we need to write for our programs to get full branch coverage. A low complexity for a program means that less tests have to be written. Note that simply because 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. We mitigate this difficulty by choosing a SOLID architecture for OOP based programs and the interpreter pattern for functional based programs.

2.3.3 Mental complexity: Cognitive Dimensions

Cognitive Dimensions is a framework for evaluating the usability of programming languages and to find areas of improvements. 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 software paradigm and language help minimise errors? Are there any structures or complexities that lead to it being easier to make errors? [12]

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. [13]

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

Chapter 3

Method: Case studies

In the Chapter 2 we defined what software paradigms are and how we can evaluate their testability and their cognitive complexity using Cyclomatic complexity and Cognitive dimensions. In Section 2.3 we 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 we find that the Cyclomatic complexity is lower in one of the paradigms or the other, we can conclude that the amount of tests write for full branch coverage will be lower for that paradigm than the other. By evaluating the paradigms using cognitive dimensions we can find 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 we will use Java.

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. [14] Haskell is a purely functional programming language, it also features lazy evaluation which allows you to compose functions easier. For example, the function $three = take\ 3 \circ cycle\ 5$, where cycle is a function that generates an infinitely long list consisting of a number, would only compute the first three values, I.E. $[5, 5, 5]$, of the infinite list. In a non-lazy program it would never evaluate as *cycle* would run forever. [15] 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] 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 3.

```
1          > list (a,2)
2          White pawn at (a,2)
3          > move (a,2) (a,3)
4          White pawn moved to (a,3)
5          > listall
6          White pawn at (a,3)
7          White pawn at (b,2)
8          ...
9          White Horse at (a,1)
10         ...
11         Black pawn at (a,8)
12         ...
```

Figure 3.1: 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.

Chat-bot 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.

Once these programs have been 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.

Chapter 4

Results

We worked so hard, yet achieved very little. TODO

4.1 Limitations

TODO

4.1.1 Improvements to implementation

TODO

4.2 future work

4.2.1 Relations to cardinality

TODO

Bibliography

[1] TIOBE, "Tiobe index." https://www.tiobe.com/tiobe-index/, accessed Feb 2019.

[2] Code Magazine, "How functional reactive programming (frp) is changing the face of web development." https://www.codemag.com/article/1601071/How-Functional-Reactive-Programming-FRP-is-Changing-the-Face-of-Web-Development, accessed Feb 2019.

[3] J. Hannah, "The ultimate guide to javascript frameworks." https://jsreport.io/the-ultimate-guide-to-javascript-frameworks/, accessed Feb 2019.

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

[5] 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.

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

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

[8] 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.

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

[10] 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.

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

[12] 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.

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

[14] 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.

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