Comparing Testability and Code Quality in Software Paradigms

Marc Coquand
Department of Computer Science
Umeå University

January 29, 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

1 Introduction

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

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.

3 Theory

3.1 Characteristics of Functional Programming

Expressions and functions

3.1.1 Iterator pattern

3.2 Object Oriented Programming

Uses variables, commands and procedures

3.2.1 SOLID principles

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 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. [3]*

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 1: Example for cyclomatic complexity.

If we have the code found in example figure 1. To calculate the complexity of this function we first construct a graph as seen in figure 2. 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.

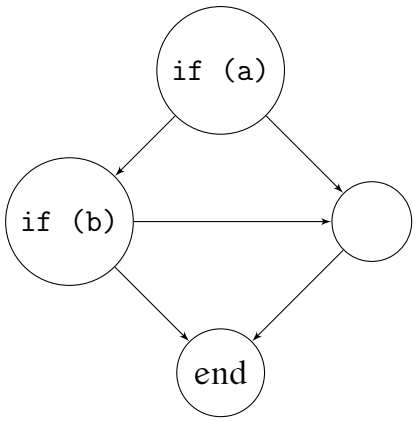

Figure 2: Cyclomatic complexity graph for figure 1

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, for a function 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 3: 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. 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. [1]

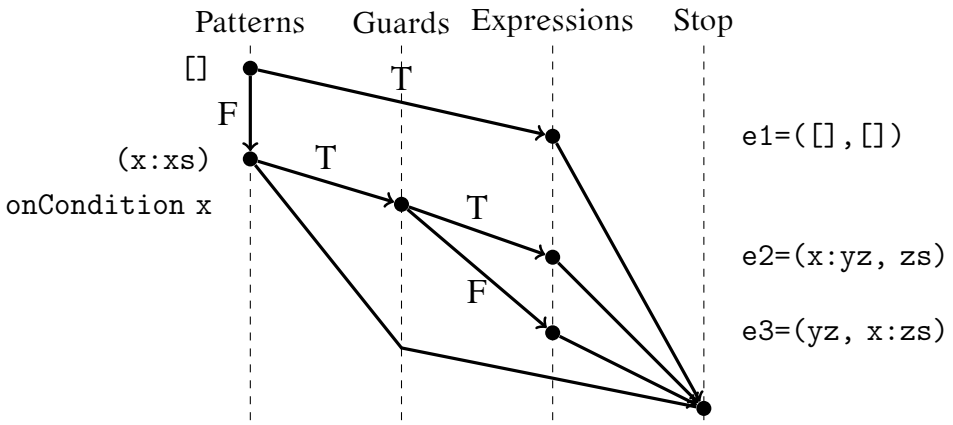

Figure 4: Flowgraph for split.

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. [2]

4.2.1 Viscosity

How much work does it take to make small changes? How easy is the code to refactor?

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?

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.

4.2.5 Role-Expressiveness

4.2.6 Abstraction

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

4.2.7 Secondary Notation

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

4.2.13 Progressive Evaluation

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

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 5: 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.
- See all items in the to-do list.
- Update an item from the to-do list.

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.

6 Limitations

6.1 Improvements to implementation

References

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

[2] T. Green and M. Petre. Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, 7(2):131 - 174, 1996.

[3] T. J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407-, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.