Comparing Testability and Code Quality in Software Paradigms

Marc Coquand
Department of Computer Science
Umeå University

January 28, 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?

1 Introduction

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

2 Theory

2.1 Characteristics of Functional Programming

Expressions and functions

2.1.1 Iterator pattern

2.2 Object Oriented Programming

Uses variables, commands and procedures

2.2.1 SOLID principles

3 Methods

3.1 Measuring testability: Cyclomatic Complexity

Cyclomatic complexity is a complexity measure that looks to measure the amount of paths through a program. The Cyclomatic complexity is an upper bound for the number of test cases required for 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. [2]*

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`. To test `f` we would need two tests, one where the if statement is false and one where true, so the cyclomatic complexity is 2. We represent that as two different edges in the graph. In summary the nodes of the graph represents processing tasks and edges represent control flow between the nodes.

```
i = 0;
n=4; //N-Number of nodes present in the graph

while (i<n-1) do
j = i + 1;

while (j<n) do

if A[i]<A[j] then
swap(A[i], A[j]);

end do;
i=i+1;

end do;
```

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 $n = 9, e = 7, p = 2 \Rightarrow v(G) = e - n + p = 9 - 7 + 2 = 4$ is the cyclomatic number.

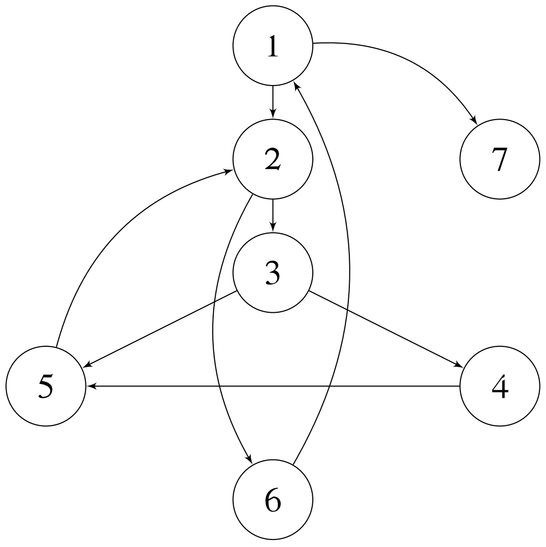

Figure 2: Cyclomatic complexity graph for figure 1

3.1.1 Cyclomatic Complexity in Functional Programming

The definition of cyclomatic complexity in Section 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, 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 of LHS. 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.

Instead of cyclomatic graphs we instead construct flowgraphs, such as the one seen in Figure 3.1.1 to model the function.

```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: 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])`.

For example, given the Haskell code in Figure 3. To calculate LHS we find three pattern identifiers which are $onCondition$, x and xs, there are two unique pattern identifiers which are x and xs, and there are two non identifiers which are [] and (:).

We do not count the *otherwise* clause, similar to how we do not count the *else* statement in normal procedural cyclomatic complexity. [1]

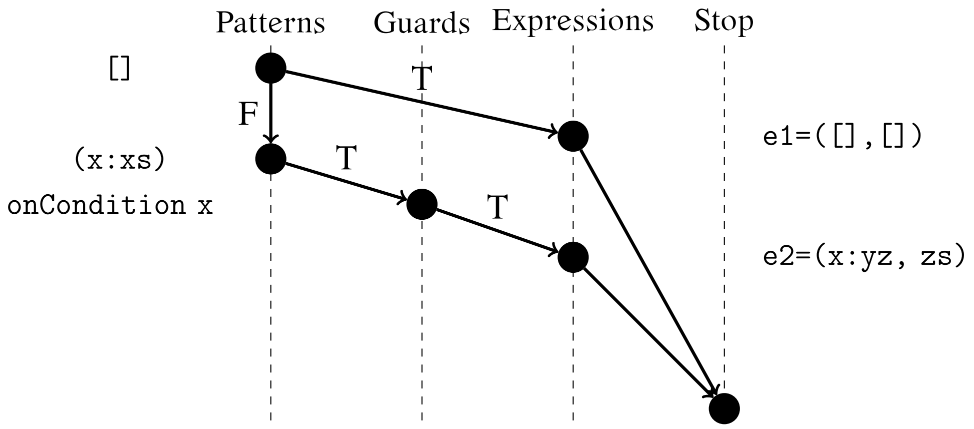

Figure 4: Flowgraph for split.

3.2 Cognitive Dimensions

3.3 Case studies

3.3.1 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

3.3.2 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.

3.3.3 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.

4 Results

We worked so hard, yet achieved very little.

5 Limitations

5.1 Improvements to implementation

References

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

[2] 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.