# How languages affect design patterns

*A comparison of Object-Oriented and Functional design patterns*

Victor Nascimento Bakke

Thesis submitted for the degree of
Master in Informatics: Programming and Systems Architecture
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2021

# How languages affect design patterns

*A comparison of Object-Oriented and Functional design patterns*

Victor Nascimento Bakke

# Abstract

Software design patterns are a common tool to solving a common set of problems, and are widely used in object-oriented programming. Likewise, functional programming has its own set of patterns and techniques that solve common problems in the functional programming space.

In this study, we have examined eight design patterns from Gang of Four's seminal work *Design patterns: Elements of Reusable Object-Oriented Software*, and how they apply to functional programming, which functional programming patterns can be used to implement these design patterns, and how applicable are they to an functional programming language.

We implemented these patterns in the object-oriented programming language Java and the functional programming language Haskell.

From this study, we found that while some design patterns are applicable to functional programming, some were not, and only a few are reasonably useful in an functional programming context.

# Contents

# List of Figures

# List of Tables

# Preface

I would like to thank my supervisor Eric Bartley Jul for invaluable guidance throughout this thesis work, especially for being accommodating with Zoom meetings in these COVID times.

# Part I

# Introduction

# Chapter 1

# Introduction

Ever since the inception of programming as a field of engineering and science, techniques to handle program complexity have been explored, discovered and refined to optimize the legibility, maintainability and efficiency computer programs. In the pursuit of these techniques, software developers borrowed the term "Design Patterns" from architecture [1] to describe reusable, composable and extensible solutions to common programming problems.

## 1.1 Introduction to design patterns

Although the term "Design Pattern" most often describes a solution aimed for object-oriented programming (OOP) systems, the definition itself is applicable to many kinds of patterns and techniques found in programming. The Gang of Four (GoF), Gamma et al., authors of the well-known book *Design patterns: Elements of Reusable Object-Oriented Software*, describe design patterns as follows:

> [They are] descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context. Gamma et al.[4, p. 24]

In this thesis we use GoF's definition of OOP design pattern. A design pattern has four essential elements:

- **Name**: A pattern must have a name, most often a word or two, that refers to the pattern, its problem and the solution. This is to increase our design language and better facilitate higher level design abstractions.
- **Problem**: The pattern must have a context or problem in which it is applied. The problem might describe a set of conditions, a set of properties that must be upheld, or a set of invariants to enforce, for the pattern to be applicable.
- **Solution**: The solution describes how the problem is mitigated, its contexts, its relationships, its responsibilities and the interconnected

components that make up the solution to the problem. The solution is not a concrete definition of how to solve the particular problem in question, but rather a basic template which can be adapted to the myriad of similar problems that fit the design pattern in question.

- **Consequences**: The positive and negative aspects of applying the design pattern to the problem. These range from performance trade-offs, readability improvements, flexibility, portability, memory constraints etc.

As well as design patterns, we use Haskell type classes as defined in *Haskell 2010 Language Report*[11], as well as the more up-to-date *Typeclassopedia*, which is a refinement of Yorgey seminal article 'The Typeclassopedia' in The Monad.Reader journal.

Even though the term "design pattern" is rarely used in the functional programming (FP) field, we will group patterns, techniques, and solutions in the FP field under "functional patterns".

## 1.2 Motivation

While design patterns are a widely known subject in terms OOP languages, the term "design patterns" rarely comes up when discussing solutions to general programming problems in FP. Functional patterns have a tendency to go by different names, and many originate from mathematics and other studies of mathematical structures. GoF's design patterns might not apply directly to FP, the majority of the problems described by GoF do occur in FP as well. Exactly how these kinds of problems are solved in FP, and what functional patterns are used to solve these problems in FP, is what this thesis aims to examine.

So for a given GoF design pattern, which functional patterns are used in FP to solve the described problem, and how do these improve the qualities of the code in question.

While different programming languages have different strengths and weaknesses in and of their own, different programming paradigms have different strengths and weaknesses as well. What might be a major problem requiring a distinct pattern to maintain readability and flexibility in one paradigm might be entirely trivial in another paradigm, or it might not even exist as a concern altogether.

Differences in patterns between OOP and FP have rarely been studied in academic papers, and even less so as empirical studies. We believe there is more to learn from performing these comparisons and measurements.

## 1.3 Problem Statement

In this thesis, we generate and analyse data to learn problems solved by design patterns in OOP translate to similar problems in FP. We limit this thesis work to two languages that are characteristic of their programming paradigm. We also limit the design patterns for OOP to a subset of those found in

*Design patterns: Elements of Reusable Object-Oriented Software* by Gamma et al. [4]. When comparing FP type classes to design patterns, we restrict ourselves to only those type classes contained in 'The Typeclassopedia' by Yorgey [23] as well as any type classes contained in Haskell's `base` package which ships with the Glasgow Haskell Compiler (GHC).

Our problem statement is thus as follows:

> How are the problems solved by the object-oriented programming Gang of Four design patterns solved in functional programming, and which attributes of these paradigms cause these differences?

## 1.4 Goal

The goal of this thesis is to compare the ergonomics between OOP and FP when solving the problems solved by design patterns. We contrast and compare various measured qualities between the generated code of the languages.

## 1.5 Approach

This thesis is an empirical study containing multiple sub-studies. Each sub-study has one design pattern to be studied. We define requirements, and implement a program in each of the two languages that satisfies these requirements. We then gather data through metrics and code, and perform a brief analysis. We then draw a conclusion for that specific design pattern.

## 1.6 Work Done

For the purposes of writing this thesis, we have investigated and examined a number of design patterns for OOP and various type classes and techniques for FP. We have looked into how others have compared these two paradigms, and especially how others have compared OOP design patterns with FP patterns.

## 1.7 Evaluation

For each selected design pattern, an implementation is made for each langauge. We gather data from these implementations through some metrics defined further in Section 5.3. We also gather data through qualitative observations made during the implementations, as well as afterwards when evaluating them in a more complete state.

## 1.8 Results

The measurements and results made during this study are difficult to summarize in this introductory chapter. The results for each case is available in the case sections in Chapter 6. The summarized and collated results are available in Chapter 7.

As a very brief summary: the Java implementations required more code, while the Haskell implementations were generally more dense in complexity.

The code produced as part of this thesis work is available at https://github.com/Gipphe/PLMasterThesis.

## 1.9 Conclusion

Our main findings from this study are as follows. These conclusions are described in more detail in Chapter 9.

**Problem space** Some problems in OOP simply do not exist in FP. As such, some patterns are not applicable to FP because there is no problem to solve.

**Sizes and complexities** Some patterns are simple to implement in OOP, while the equivalent solution in FP might be complex, or require more than one pattern. Some patterns require a sizeable amount of code in Java, while being trivial to implement in Haskell. There is a clear mixture of both cases.

**Concerns and values** Most FP patterns are smaller than OOP patterns. FP is heavily focused on composability, while OOP is more focused on imperative procedures.

## 1.10 Limitations

While we feel confident in the results and conclusions in this study, it is merely a master's thesis. We acknowledge the limitations of our own capabilities, as well as the short timescale in which this study was performed. Had we more time, we would have examined all of the GoF design patterns instead of just the selected eight.

We describe our observed limitations in further detail in Section 8.6.

## 1.11 Outline

This thesis consists of four parts.

1. Part I is an introduction and overview of the study. It aims to present the motivation for this thesis, as well as an overview of the thesis contents.

2. Part II contains the background and theoretical knowledge necessary for discussing the results of this thesis.

3. Part III describes the methods used to gather data and evaluate the results.

4. Part IV describes the cases and implementations presented. It also describes the results for each individual case.

5. Part V examines the results from Part IV, and draws conclusions based on this data.

# Part II

# Background

# Chapter 2

# Programming paradigms

## 2.1 Introduction to programming paradigms

Programming languages can be categorized in a multitude of ways. From the level of abstraction from the actual machinery in the computer, how programs written in the language are structured, strictness and verbosity, and many other qualities can be used to qualify a given programming language. Some programming languages strive to disallow invalid programs through the use of type systems, contract-based programming and pre- and post conditions, while others take a more hand-off approach, allowing the programmer full control over the life cycle, correctness and resource usage of the program, coupled with the increased discipline those responsibilities demand. Some languages prefer a specific paradigm of programming, while others allow for multiple paradigms of programming, often interchangeably or even all at once.

Before we tread too far forwards, we have to define "paradigm" in programming languages. According to the online dictionary Merriam-Webster, the general term "paradigm" means, among other definitions:

> "A philosophical and theoretical framework of a scientific school or discipline within which theories, laws, and generalizations and the experiments performed in support of them are formulated." [13]

We personally find this definition quite fitting for the term "programming paradigm" as well. In the scientific school of software development we have theories, laws and generalizations which follow specific philosophical processes to develop programs. These philosophical processes can be in conflict with each other, with differing values that do not coexist well in a program. They can be in harmony with each other, furthering the quality of the program more when used in tandem than if used on their own. Such philosophical processes and conventions are what constitutes programming paradigms. Some have clear definitions on how a program should be structured, while some merely describe a general style or pattern of programming. Some define how information is treated, and how data is transformed, during the life cycle of the program. How the program is split up and how program behaviour is represented is tied to the paradigm of the program in question, and subsequently the paradigm the programming language used is able to represent well. [3]

The two programming paradigms we take a look at in this study is the "Object-oriented programming (OOP)" and the "Functional programming (FP)" paradigms.

## 2.2 Object-Oriented Programming

Object-oriented programming, as its name implies, focuses on the concept of objects containing data, where each object is a container with its own data, and is responsible for manipulations and interactions with its contained data. Objects then communicate with each other through the publicly available methods and operations available on each object. [14]

While the specific meaning of "object-oriented programming" has changed over the years, OOP encompasses some key concepts:

- An object is a collection of operations that share state. These operations may interact with the contained state, but said state is inaccessible to the outside world, and may only be interacted with through the collection of operations. This property is referred to as "encapsulation", and allows objects to define a definitive way for the outside world to interact with its state. [22]
- Classes, serving as templates for creating new objects of a specific type, serve as a formal method of instantiating new objects of a specific shape. [22]
- Through inheritance, one class may serve as the basis for other classes, which may in turn serve as the basis for further more classes, building a hierarchy of behaviour and operations. [22]
- Any inheriting class (subclass) can be treated as if it was any inherited class (superclass) in its line of inherited classes. [22]

In 1967, Ole-Johan Dahl and Kristen Nygaard developed what is today recognized as the first OOP language, Simula 67. It introduced what we today consider core OOP concepts such as objects, classes, inheritance and dynamic binding. [6]

## 2.3 Functional Programming

What differentiates FP from OOP, and more conventional Procedural Programming languages, is that FP focuses more on functions, and composing functions together into entire programs. In FP, "function" has a very specific meaning, and describes a more mathematical function from one value to exactly one other value, providing a mapping between the two sets of possible values. FP emphasizes that a function should be "pure", and cannot perform any side-effects. A function should only take in data as its arguments, and return the result of its computation. It should not be necessary to know exactly how this return value is computed from the outside. Some FP languages enforce this restriction on functions, and are thus dubbed "pure" languages, while those that do not enforce purity are deemed "impure" languages. [20] In pure FP languages, purity has multiple implications on how FP languages work:

- Data structures have to be immutable, because mutating data changes that piece of data in all places it is referenced, which is considered a side-effect. Thus, to update a piece of data, a new instance of that data is created with the requested changes applied.

- Functions are referentially transparent, meaning that for any given input, the function will always return the same result regardless when, how or how many times it is called with the piece of input.

- In languages that support non-strict (lazy) evaluation, functions might not even be called if their result is not required. An example of this in Haskell:

```haskell
add1 x = x + 1

main = do
    let y = add1 4
    return ()
```

  Without describing Haskell's syntax quite yet, here in `main`, we first compute `add1 4`, which returns `5`, but then we disregard that returned value and simply `return ()`, terminating the program. Haskell supports lazy evaluation, meaning - in simple terms - that only the values that are required by the program are actually computed. Here, `y` is never used, and so is never computed, which means `add1` is never called.

Treating these properties as invariants on functions themselves, it is possible to run a function exactly how many times is necessary, whether it be zero, one or more times, and it may even be run in parallel without interference since the functions do not interact with the outside world.

Functions in FP can themselves be treated as values to be passed around a given program. Functions can take other functions as arguments, and use them in their definition. Functions that take other functions as arguments, and functions which return functions as their results, are called "higher-order function (HOF)", while a function that is able to be passed as a value is called a "first-class function", where "first-class" denotes whether the value in question can be passed around and treated like any other value in the program.

Programs in FP tend thus to revolve around data types with a set of functions that interact with said datatype.

## 2.4 Summary of programming paradigms

In this chapter, we have presented and described programming paradigms. We have described in detail the two programming paradigms OOP and FP, and how briefly how they differ from each other.

# Chapter 3

# Languages

## 3.1 Introduction to languages

A programming language is a method of communicating instructions to a computer for the purposes of performing a set of computations or a set of actions. The exact actions performed by the computer by a given instruction depends on the programming language. A computer's base instruction set is defined by the computer's processor and its architecture. While it is possible to write computer programs directly in this instruction set, the instruction set is very bare-bones and difficult to reason about for larger, or even medium-sized, programs. As such, programmers have developed higher level languages that can be translated by a "compiler" down to this instruction set. These higher level languages abstract away the complex and hard-to-reason-about details of the instruction set, and allow the programmer to focus their efforts elsewhere. This instruction set is often called simply "machine code" to differentiate it from human-readable code.

The higher level a language is, the more removed from the machine code it is. While machine code consists of raw bytes being executed by the processor directly, a programming language is often written in plain human-readable text, with various keywords and symbols carrying specific predefined meaning in the language.

## 3.2 Static typing

Most programming language has some notion of "types". The type of a piece of data describes the structure of the data in question. A number is a different type from a piece of text, for example.

Programming languages are either statically typed or dynamically typed. A statically typed language requires that the types of the program's data is known before the program is executed. A dynamically typed language has no such requirement.

With static types, a compiler or a runtime system can analyse the results of operations and transactions within the program ahead of time, and discover whether there are mismatches between what is expected from an operation and how the operation is defined. In regards to types, these kinds of mismatches are referred to as “type errors”. In a dynamically typed language, these “type errors” would instead end up becoming either undefined behaviour, or runtime errors.

## 3.3 Java

Java is a statically typed, compiled, class-based, object-oriented programming language with heavy emphasis on portability and architectural independence. Java’s core philosophy is “write once, run anywhere”, abbreviated “WORA”. Java makes this possible by being a compiled language that compiles to bytecode executable by a Java virtual machine (JVM). As long as there exists a JVM implementation for a specific platform, it will be able to execute any given Java program, as long as said program does not utilize platform-specific binaries or libraries. [9]

With a heavy emphasis on OOP and its concepts, Java enforces all files in a Java program to contain exactly one public class at the top level of the file, with the same name as the file in question. Thus, each module in Java is a class, and can thus be instantiated.

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, world");
    }
}
```

### 3.3.1 Inheritance

Here we use the access modifier “public”, which means the class in question (as well as the contained method |main|) is available for any consuming method or object. Java supports a total of four access levels:

- private: only the containing class has access to the item in question.
- package-private (the default when no access level is specified): the containing class and all members of the package.
- protected: the containing class, all members of the package and all subclasses of the class.
- public: everything has access.

Java also has the concept of a “package”, which in simplified terms is the folder the source file is located in. A class in a sub-package has access to package-specific content in its parent packages, but not the other way around.

Inheritance is the core mechanism of code reuse, and Java supports single inheritance from classes and multiple-inheritance from interfaces.

Inheriting from a class inherits all of that class’ methods, both its own methods and inherited methods.

```java
public class Foo {
    public int foo(int x) { /* ... */ }
}

public class Bar extends Foo {
    public boolean bar(boolean x) { /* ... */ }
}

public class Main {
    public static void main(String[] args) {
        Bar bar = new Bar();
        int fooRes = bar.foo(12)
        boolean barRes = bar.bar(true);
    }
}
```

Because |Bar| inherits from |Foo|, it inherits |Foo|'s |foo| method as well. Bar can override |Foo|'s |foo| method by defining a |foo| method of its own:

```java
public class Bar extends Foo {
    public boolean bar(boolean x) { /* ... */ }

    public int foo(int x) { /* ... */ }
}
```

And thus |Bar|'s |foo| method would be called instead.

Java's interfaces describe a contract that an implementing class must adhere to for the program to be considered valid. This contract describes the publicly visible methods of the implementing class and their signature. Any method or operation may thus depend on this interface, and any implementing class should suffice.

```java
public interface Baz {
    int quack(int quux);
}

public class Foo implements Baz {
    public int quack(int x) { /* ... */ }
}
```

For `Foo` to be a valid instance of `Baz`, it *must* implement at minimum `int quack(int)`.

### 3.3.2 Generics

Java is a statically typed language. There are two kinds of types in Java: primitive and non-primitive. All non-primitive values are objects, while the primitive values are built-in for performance reasons. [2] Modern Java also supports "Generics", which allows you to pass types to a class or interface and let the class or interface describe signatures and operations in relation to this passed type.

```java
public class Foo<T> {
    private T x;

    public Foo(T x) {
        this.x = x;
    }

    public T getX() {
        return this.x;
    }
}

public class Main {
    public static void main(String[] args) {
        Foo<Integer> foo = new Foo(12);
        Integer x = foo.getX();
        System.out.println(x);
    }
}
```

The above program prints `12` to the console. Here, the type `T` is generic. Any fully qualified type can be passed, but note that we pass the type "|Integer|". This is a boxed type for `int`, and is required for passing an `int` to `Foo` because primitive types cannot be passed to generic constructs. Java supports automatically boxing primitive values to their boxed variants, and as such we do not need to write |Integer.valueOf(12)| instead of simply `12`. [16] Java's implementation of generics is considerably more advanced than this description indicates, but for simplicity's sake we will omit the remaining details.

## 3.4 Haskell

Haskell is a statically typed, compiled, "pure" functional programming language [7] with its roots as an academic language, but has seen extensive use in practical and industrial applications.

Haskell by various researchers in academia during the 1990s as a means to simplify writing about research and theorems in computer science (CS) without having to describe the language used in code examples in their articles. [8] The Haskell Committee was formed to steer the design of the language.

A Haskell program is required to at minimum have a function called `main` with the type `IO ()`. The simplest "Hello world" program in Haskell looks like this:

```haskell
main :: IO ()
main = putStrLn "Hello␣world"
```

Including the function's type signature is optional, but recommended. Type signatures are defined separate from the function, and use the `::` operator, which can be read as “has type”, to bind the type on the right-hand side to the identifier on the left-hand side.

While Java’s syntax might be very familiar to the reader, Haskell’s syntax warrants some explanation. Haskell’s syntax is simple yet extensible; Haskell lets the programmer create user-defined operators with customizable precedence rules. An example alias for the modulo operator (`%`):

```
(#) :: Int -> Int -> Int
x # y = x % y

main = print (3 # 2)
```

This would print `1` to `stdout`.

Because of Haskell’s functional nature, function application is just the function name and its arguments, separated by white space, optionally encapsulated by parentheses - or not optional if you pass the result of one function to another function:

```
foo :: Int -> Int -> Int
foo x y = x + y

main = print (foo 1 2)
```

Operators are merely binary functions in Haskell, and regular named binary functions can be treated in much the same way as an operator by using backticks (`‘`)s:

```
main = print (1 ‘foo‘ 2)
```

Using square brackets, you can form a list:

```
foo :: [Int]
foo = [1, 2, 3, 4]
```

And with parentheses and commas, you can form tuples:

```
foo :: (String, Double)
foo = ("Hello", 2.43)
```

Function types are defined with `->`, while functions themselves can be defined in two ways:

```
foo :: Int -> Bool
foo x =  x > 3

bar :: Int -> Bool
bar = \x -> x > 3
```

Functionally, `foo` and `bar` are identical.

All functions in Haskell are “curried”, meaning they take one argument and return a function which takes the rest of the arguments. Since function application is only space-separating the function from its argument, this effect is transparent when applying multiple arguments at a time, but when partially applying a function by only supplying some of its arguments, you create new functions that await the rest of the arguments.

```haskell
add :: Int -> Int -> Int
add x y = x + y

double :: Int -> Int
double x = add x x

increment :: Int -> Int
increment = add 1
```

Here, `add` takes two arguments and simply adds them together, returning the result. `double` takes one argument, and doubles it through the use of `add`. But `increment` takes one `Int`, and returns an `Int`, but there is no argument defined in `increment`'s function definition. `increment` is written in an "eta-reduced" style: since `add 1` returns a function `Int -> Int`, and since `increment` is a function `Int -> Int`, then `add 1` is a sufficient definition for `increment`.

### 3.4.1 Types

Haskell has many built-in data types that might seem familiar to programmers coming from other languages, as well as some types which might seem foreign. The list is too long for this essay, but some types which might seem unfamiliar to programmers not used to FP are:

- `()`: called the "unit" type, it has only one value, namely `()`. [8] Useful when a function performs some action in an environment (for example, prints to the console), but doesn't return a meaningful value.
- `Maybe a`: can either be a `Nothing`, which denotes the absence of a value, or `Just a` which denotes the presence of a value of type `a`. [10]
- `Either a b`: can either be a `Left a` or a `Right b`. Often used when a function can fail to perform its task, thus returning a `Left someError`, or a `Right result` if the operation completed successfully. [10]
- `(a, b)`: a two-tuple containing an element of type `a` and an element of type `b`. Tuples can be of varying sizes, and each are considered separate types. `(a, b)` is separate from `(a, b, c)`, and they cannot be treated the same. A function applied to a three-element tuple will not compile if it expects a two-element tuple, and vice versa. There is theoretically no upper limit to the number of elements a tuple can have, and it might vary from implementation to implementation, but the limit is at least fifteen, as stipulated by the Haskell 98 report. [8]
- `Void`: a completely empty type. There exists no values with the type `Void`, meaning any type consisting of `Void` must have an alternative value that it can return. Example:

```haskell
foo :: Either Int Void
foo = Left 12

```

```haskell
bar :: Either Int Void
bar = Right ???
```

foo is a valid value of type Either Int Void, since the Left value can be Int, but bar cannot exist because there is no value for type Void, and as such we have nothing to replace ??? with. Haskell has a concept of “bottom”, formally represented as ⊥. Bottom refers to a computation which never completes due to some error or an infinite loop, or a value which can never exist at runtime. undefined encapsulates the concept of “bottom” in Haskell, and allows programs which could otherwise not exist to compile. If undefined is ever encountered at runtime, however, the program will raise an error and halt.

```haskell
bar :: Either Int Void
bar = Right undefined
```

So if bar’s value is ever required to complete a computation, then the runtime system would encounter undefined and raise an error.

One of Haskell’s strong points is its powerful and extensive type system which allows the programmer to make assumptions and set pre- and post conditions for their programs. This type system is built upon, among many other things, “Algebraic Data Types”. In this sense, an “Algebra” refers to how such a data type is made up of “algebraic” operations, namely “sums” and “products”. The number of possible values a type can have is called the “cardinality” of that type. A “sum” in this sense is the sum of the type in question’s contained types’ cardinalities. Likewise, “product” is the product of the contained types’ cardinalities.

Defining new data types is done through the use of the data keyword.

```haskell
data C = CS Int | CT String
```

Following the data keyword, we have the name of the new type in question, here D. The right-hand side of = denotes the possible values of this new type, here CS Int and CT String. CS and CT in this case act as data constructors for our new type. CS 12 and CT "foo" are thus valid C values.

Data types can also have one or more “type variables” attached to them. Type variables resemble generics in Java, but are more closely related to the mathematics of type-level programming.

```haskell
data D a b = DS a | DT b
```

Here, D is a type constructor, taking two types as its arguments. So D Int String is equivalent to our C type above. Thus, to make a value of type |D String Bool|, you can either use DS String or DT Bool, so DS "foo" and DT False are valid D String Bool values.

While Haskell does not have “objects” in the OOP sense, it does have “records”, which are a collection of data indexed by accessor functions:

```haskell
data Foo = Foo
    { foo :: Int
    , bar :: String
    }

f :: Foo -> String
f x = bar x ++ "␣" ++ show (foo x)

main = print (f (Foo { foo = 12, bar = "foobar" }))
```

When defining a record, Haskell automatically creates accessor functions for its fields, here `foo` and `bar`, which have the types `foo :: Foo -> Int` and `bar :: Foo -> String` respectively.

### 3.4.2 Type classes

Haskell's core method of abstracting behaviour is using "type classes". Without going into too much detail on the implementation and specifics of type classes, they enable overloading functions and operators for each type that implements the type class. [21] An example of this is the `Num` type class. `Num` encompasses multiple operators, but key among them are the `+` and `-` operators. These operators, because they are defined in terms of the `Num` type class, work on any types that implement the `Num` type class. Number types like `Int` and `Double` both implement `Num`. The `+` operator has the type `Num a => a -> a -> a`, where `a` is the type of the values being added together, and as such the expression `2 + 2` can either have the type `Int` or the type `Double`, and the exact mechanism of adding the two numbers together would be handled by their corresponding type's instance for that type class.

### 3.4.3 Example data type

Example: the list type in Haskell. Lists in Haskell are singly-linked cons-lists, meaning each element consists of the value in that element of the list, coupled with the rest of the list. The list can also be empty, of course.

```haskell
data List a
    = Empty
    | Cons a (List a)
```

So to create a value of this list of ours, we can either use `Cons` to create a list from a value and another list, and `Empty` to create an empty list.

```haskell
foo :: [Int]
foo = [1, 2, 3]

bar :: List Int
bar = Cons 1 (Cons 2 (Cons 3 Empty))
```

Thus, though rather verbose, `bar` is equivalent to the list `foo`.

Next we want to define equality for our list. Equality is handled by the `Eq` type class. `Eq` has many functions, but we only have to define one to get it to work: `==`. But how should comparing a list for equality work? Naturally, we come to the conclusion that we need to compare the elements of the two lists being compared element-for-element. That means the elements also need to be instances of Eq. Using “pattern matching” here, we can specify how == reacts to each possible value of List a. We use pattern matching by specifying the function for each pattern to match, much like how you would specify equations in mathematics. Underscore, _, simply means we ignore the value in that position:

```
1 (&&) :: Bool -> Bool -> Bool
2 True && True = True
3 _    && _    = False
4
5 instance Eq a => Eq (List a) where
6 Empty     == Empty     = True
7 Empty     == _         = False
8 _         == Empty     = False
9 Cons x xs == Cons y ys = x == y && xs == ys
```

Recursive structures in Haskell are very common, and our list is indeed a recursive data structure. Thus, we first need a base case to handle, and terminate the recursion. Comparing two empty lists will be one of the base cases in this instance, since two empty lists are indeed equal to each other. Then, if either list is empty while the other is not empty, they are not equal. On line five, we have the more interesting case that handles actual non-empty lists. We use pattern matching to get the element in the Cons, as well as the rest of the list. We compare the two values x and y, and compare the rest of the lists xs and ys, boolean AND-ing the results together. Thus, comparing the list Cons 1 (Cons 2 (Cons 3 Empty)) with itself expands into the following (shortening “Cons” to “C” and “Empty” to “E” to save horizontal space):

```
C 1 (C 2 (C 3 E)) == (C 1 (C 2 (C 3 E)))
1 == 1 && (C 2 (C 3 E) == C 2 (C 3 E))
1 == 1 && (2 == 2 && (C 3 E == C 3 E))
1 == 1 && (2 == 2 && (3 == 3 && (E == E)))
1 == 1 && (2 == 2 && (3 == 3 && True))
1 == 1 && (2 == 2 && (True && True))
1 == 1 && (2 == 2 && True)
1 == 1 && (True && True)
1 == 1 && True
True && True
True
```

Creating new type classes is done with the class keyword:

```
class Foo a where
    bar :: a -> Int

data Baz = MkBaz Int

instance Foo Baz where
    bar (MkBaz x) = x
```

```haskell
foo = round (1 + limit 10 (((23 + 5) / 4)))
bar = round $ (+) 1 $ limit 10 $ (/) 4 $ 23 + 5

p = mkReport
    ( avg
        ( take 10
            ( sort
                ( fetchInvoices "url"
                )
            )
        )
    )

q = mkReport
    $ avg
    $ take 10
    $ sort
    $ fetchInvoices "url"
```

Figure 3.1: Function application operator example

Here we define the type class `Foo`, which has the function `bar`. `Baz` then implements this type class through the `instance` declaration, and describes the implementation of `bar` for `Baz`.

### 3.4.4 Operators

An "operator" in Haskell is just a binary function whose "name" is made up of symbol characters that can be used in infix position. They hold no special meaning to the compiler, outside of how some operators are defined for more primitive data types. This means we can define our own operators, and there are a myriad of operators already defined in Haskell's `base` package. Operators can be used in their normal infix position, `1 + 2`, or they can be used like regular function, enclosed in parentheses, `(+) 1 2`. The lattern form can be used to partially apply an operator like a reglar function, `increment = (+)1`.

Apart from the commonly used addition (`+`), subtraction (`-`), multiplication (`*`) and division (`/` for floating point numbers and `‘div‘` for integral numbers), Haskell comes with a few operators not commonly seen outside of Haskell and FP:

- `($) :: (a -> b)-> b -> a`: Function application. `f $ a` is equivalent to `f a`. While this might seem useless, `$` has a specific use because of its low operator precedence. This allows the programmer to reduce the use of parentheses.

  In Listing 3.1, `foo` and `bar` are equivalent, and `p` and `q` are equivalent.

```haskell
p x = foo (bar x)
q = foo . bar
```

Figure 3.2: Function composition operator example

The function application operator can make Haskell code look very foreign to programmers not used to Haskell. As such, we will minimize its use in this thesis work to only places where the parentheses would make things harder to read.

- `(.) :: (b -> c)-> (a -> b)-> (a -> c)`: Function composition. This is equivalent to the mathematical function composition operator ($\circ$).

  In Listing 3.2, `p` and `q` are equivalent.

  The function composition operator allows Haskell code to be rather terse and hard to read. As such, we will minimize its use in this thesis work to only places where the expanded form would be less readable.

- `(<$>) :: Functor f => (a -> b)-> f a -> f b`: Functor function application. Essentially the `($)` operator lifted into a functor. Explained further in Section 4.3.3.

- `(>>=) :: Monad m => m a -> (a -> m b)-> m b`: Monadic bind. Used for chaining monadic actions. Explained in Section 4.3.3.

### 3.4.5 Purity

As mentioned previously, Haskell is a "pure" programming language. Here, "purity" refers to how a function in Haskell cannot interact with the outside world in any way.

Haskell programs are composed of instructions for the underlying runtime system on how to interact with the world. If the programmer want to interact with the outside world in a function, the function has to return instructions on what to do with the outside world. This is encapsulated in the `IO a` type. `IO a` is, in simplified terms, a record containing the state of the world outside of the program, coupled with some value `a` that is available after the interaction with the outside world has come to pass. When calling a function that returns `IO a`, that function must be evaluated in regards to previous `IO`-values. Haskell has a type of notation that simplifies writing these operations (as well as many others) called `do`-notation:

```haskell
getLine :: IO String
getLine = ...

putStrLn :: String -> IO ()
putStrLn str = ...

main :: IO ()
main = do
    input <- getLine
    putStrLn input
```

getLine returns the type IO String, meaning a new state of the world (where a String has been read from stdin) coupled with the read String. Using <-, we unwrap the String from IO, and the state of the world in IO is carried on from getLine. While getLine does not have much impact on the world directly, putStrLn does, by printing the passed String to stdout in its world state. This world state is then evaluated, the String is printed to stdout, and the result is simply (). Out program is finished at this point, and thus exits normally, having read from stdin and printed to stdout.

We must stress that this is a gross and somewhat incorrect oversimplification of what is actually happening in the runtime system.

### 3.4.6 Laziness in brief

Haskell is a **lazily computed** programming language, and as such Haskell supports infinite data structures, as well as built-in short-circuiting, since only the elements of the structure that are required to perform the program's tasks are actually used.

```haskell
foo :: [Int]
foo = take 6 (repeat 9)
```

Here we first use repeat to make an infinite list of repeating 9s, followed by take, taking the first 6 elements of that list. Since take only uses the first six elements of the list, the rest of the list is never evaluated. Even more so: if foo is never used, it is never evaluated either, and neither repeat nor take are ever called.

### 3.4.7 Language extensions

There are a large number of available language extensions that augment the Haskell language, change the behaviour of data types, further enhance the type system etc. While many of these are used very, very often when writing modern Haskell, we limit ourselves to as few of these language extensions as possible for simplicity's sake.

These are the language extensions used in the implementations in this study:

- ExistentialQuantification: This allows us to conceal the actual type of something in a datatype.

  ```haskell
  data Hidden = forall a . MkHidden a
  ```

  Here, Hidden has one constructor, MkHidden, which will take any type whatsoever. After creating a value of type Hidden, we can get to this concealed value through pattern matching:

  ```haskell
  foo :: Hidden -> _
  foo (MkHidden x) = undefined
  ```

In foo, we have x available to us, but we have “forgotten” what x is. According to Hidden, x is of type a, and nothing more. As such, there is little we can do with x in this example, and we cannot even return it from the function because a isn’t defined outside of Hidden, so it will never type check.

Inside of the data declaration using existential quantification, we can add constraints to the contained type.

```
data AnyShow = forall a . Show a => MkAnyShow a
```

To construct a value of type AnyShow using its constructor MkAnyShow, the supplied value must be an instance of Show. Since this “proof” is remembered by AnyShow, it will be available to us when we pattern match as well.

```
foo :: AnyShow -> String
foo (MkAnyShow x) = show x
```

We use ExistentialQuantification to allow us to make lists containing items of different types, as long as they have the necessary type class instances. In Java, this is a very common pattern:

```
interface Display {
    String display();
}

class Foo implements Display {
    public String display() { return "Foo"; }
}

class Bar implements Display {
    public String display() { return "Bar"; }
}

class Baz implements Display {
    public String display() { return "Baz"; }
}

public class JavaExample {
    public static void main(String... args) {
        Display[] list = new Display[] {
            new Foo(),
            new Bar(),
            new Baz()
        };
        for (Display x : list) {
            System.out.println(x.display());
        }
    }
}
```

To have this kind of list in Haskell, we have to use ExistentialQuantification:

```haskell
{-# LANGUAGE ExistentialQuantification #-}

import Control.Monad (forM_)

class Display a where
    display :: a -> String

data AnyDisplay =
    forall a . Display a => MkAnyDisplay a

instance Display AnyDisplay where
    display (MkAnyDisplay x) = display x

data Foo = Foo

instance Display Foo where
    display Foo = "Foo"

data Bar = Bar

instance Display Bar where
    display Bar = "Bar"

data Baz = Baz

instance Display Baz where
    display Baz = "Baz"

main :: IO ()
main = forM_ list (putStrLn . display)
  where
    list :: [AnyDisplay]
    list =
        [ MkAnyDisplay Foo
        , MkAnyDisplay Bar
        , MkAnyDisplay Baz
        ]
```

- MultiParamTypeClasses: according to the original *Haskell 2010 Language Report*, type classes in Haskell can have a single type parameter.

```haskell
class OneParam a where
    foo :: a -> a
```

This is considered a bit of an unfortunate oversight, and multiparameter type classes can be enabled through MultiParamTypeClasses.

```
class TwoParam a b where
    bar :: a -> b -> b
```

We use this as an alternative to more complicated language extensions to allow us to do more with type classes, as MultiParamTypeClasses is adequate to achieve the simple type classes we define.

- FlexibleInstances: normally, type class instances must be in the form `instance Foo T` or `instance Foo (S a)`. Instances such as `instance Foo (S T)` are not allowed. FlexibleInstances remedies this by allowing arbitrarily nested types, as well as type synonyms, in type class instances.

- FlexibleContexts: Similarly to `FlexibleInstances`, normally, a type's context can only have type variables in its constraints. This means the following definitions are disallowed:

```
instance Foo Int a => Bar a

foo :: Bar n String => n -> String

bar :: Foo Int => Int -> Int
```

  `FlexibleContexts` relaxes this restriction, making the definitions above valid. This is especially useful when using `MultiParamTypeClasses`.

## 3.5 Differences between the langauges

Java and Haskell were both born at roughly the same time, but with widely different purposes and use-cases in mind. Java was designed for portability and OOP, while Haskell was designed for mathematical expressiveness and CS research.

### 3.5.1 Objects vs. records

In Java, everything (except the primitive types) is an object, while in Haskell the closest thing to objects is record types, which do not have methods and are not mutable. In Haskell, functions inside a record do not have special access to the record they are contained in, while in Java methods can access their object through `this`.

### 3.5.2 Inheritance vs. subtyping

Java's omnipresent inheritance is completely absent from Haskell. The closest Haskell has to any sort of inheritance is subtyping:

```
data Foo = Foo Int
```

where Foo's type class instances can be specified in terms of the contained Int:

```
instance Num Foo where
    Foo x + Foo y = Foo (x + y)
    Foo x - Foo y = Foo (x - y)
    ...
```

### 3.5.3 Lambda expressions vs. higher-order functions

There are however various qualities from each language which has been added to the other language. Most notably Java's lambda expressions and streams, while not directly inspired by Haskell in particular, stem from the FP world.

Taking a look at lambda expressions in particular, whenever you want to pass just a function to another method, you had to wrap this function in a class because functions themselves are not first-class values in Java. The method expecting the function in question would then also have to know statically the type of the argument object, and what methods it needs. As such, you have to define an interface to describe the structure of this anonymous class as well.

```
interface Arg {
    boolean isGood(int x);
}

public class WithoutLambda {
    static boolean doStuff(Arg arg) {
        return arg.isGood(12);
    }

    public static void main(String[] args) {
        boolean res = doStuff(new Arg() {
            public boolean isGood(int x) {
                return x > 10;
            }
        });
        System.out.println(res);
    }
}
```

This is exceedingly verbose. The Arg interface has a single abstract method defined, meaning it is considered a "functional interface". Java supports lambda expression syntax for functional interfaces, meaning you need only write a heavily abbreviated method, (x) -> x > 10, and pass that to the method in question.

```
interface Arg {
    boolean isGood(int x);
}

```

```
public class WithLambda {
    static boolean doStuff(Arg arg) {
        return arg.isGood(12);
    }

    public static void main(String[] args) {
        boolean res = doStuff(x -> x > 10);
        System.out.println(res);
    }
}
```

In addition to supporting functional interfaces in this way, Java also comes with multiple functional interfaces built-in for these kinds of purposes. One of them is the `Predicate<T>` interface:

```
interface Predicate<T> {
    boolean test(T t);
}
```

Which further simplifies our toy program, allowing us to completely omit `Arg`:

```
import java.util.function.Predicate;

public class PredicateExample {
    static boolean doStuff(Predicate<Integer> pred) {
        return pred.test(12);
    }

    public static void main(String[] args) {
        boolean res = doStuff((x) -> x > 10);
        System.out.println(res);
    }
}
```

### 3.5.4 Imperative style

While Haskell has strived to stick to its FP nature, some concessions have been made in terms of syntax. One of these concessions is the do-notation previously mentioned and demonstrated. Do-notation emerged from the need to perform effectful actions in sequence, especially `IO`-actions, mimicking a more imperative programming style. Previously, the `IO` example above would have been written as follows:

```
main = getLine >>= \line -> putStrLn line
```

with a new `>>= \x ->` for each value you want to carry on forward to subsequent actions. It was recognized that this pattern is just an imperative sequence of actions, and as such the do-notation was implemented in the Haskell standard. Do-notation is, however, simply syntactic sugar for the above pattern, and it is transformed at compile time.

```
main = do
```

```
    line <- getLine
    putStrLn line
```

### 3.5.5 Design patterns vs. type classes etc.

The GoF design patterns are all applicable to Java, while barely any of them are even possible to implement in Haskell without using unconventional techniques. Haskell has “patterns” of its own, however, in the form of type classes and other techniques.

### 3.5.6 Eager vs. lazy

Java is an eagerly computed language, with only simple forms of lazy computation such as short-circuiting in the boolean operators:

```
boolean x = False && getSomeOtherValue();
boolean y = True || getSomeDifferentValue();
```

Here, neither `getSomeOtherValue` nor `getSomeDifferentValue` would ever be invoked.

In comparison, Haskell is a lazily computed language, and such short-circuiting as above will occur in many different scenarios as values are only computed as they are needed. As a result of this, Haskell suffers from space leaks where “thunks” of computation that have yet to be evaluated are built up in memory, effectively acting as a memory leak. These “thunks” represent computations which might never even be needed, and thus will never be computed, but the garbage collector is unable to release them because long-lived references to them might still exist.

## 3.6 Summary of languages

In this chapter we have taken a look at static typing, as well as the two languages we will use in this thesis. We have also summarized the key differences between the two languages.

# Chapter 4

# Design Patterns

## 4.1 Introduction to design patterns

In pursuit of furthering the study and practice of software engineering, many methods have emerged and are available as recipes on how to structure a program to achieve or preserve some desired property of the source code. These methods are often called "Design patterns", and they describe reusable structures and techniques with the aim to improve the quality of a program, whether it is applied to just a subset or the entirety of the program.

In OOP, design patterns are an oft-mentioned source of programming wisdom, gaining popularity in 1994 with the publishing of the book *Design patterns: Elements of Reusable Object-Oriented Software* by Gamma et al., colloquially referred to as the "Gang of Four". In the book, they outline three major categories of design patterns; Behavioral patterns, Structural patterns and Creational patterns; as well as describing various design patterns within those categories. [4] Even today, in 2021, this book stands as the core literature on design patterns in OOP.

In FP, on the other hand, functional patterns are as pervasive as in OOP, but they are rarely explicitly called "design patterns". Since patterns in FP tend to be considerably smaller than in OOP, it is sometimes difficult to establish whether a technique or method in FP is a "design pattern". Some patterns are widely used, and have become so ingrained in the programming culture of the programming language, or FP as a whole, that it is difficult to identify whether the pattern exists as a result of thought and consideration or just because it is the most sensible way to structure a the program.

## 4.2 Object-Oriented Design Patterns

In the following sections, we present our chosen 8 OOP design patterns.

### 4.2.1 Abstract Factory

"Provide an interface for creating families of related or dependent objects without specifying their concrete classes." [4, p. 109]

The Abstract Factory pattern is used to define a way to create a set of related objects without having to know their concrete classes. The Abstract Factory defines a set of methods which themselves return abstract classes of products. The concrete products from these methods is decided by the concrete Abstract Factory and its implementation. The consumer can thus be supplied with an Abstract Factory, and have no dependence on the concrete classes used for neither the factory nor its products.

### 4.2.2 Adapter

> "Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces." [4, p. 159]

The Adapter pattern is used as a compatibility layer between two incompatible interfaces or systems. The adapter class usually either wraps or extends the adapted, altering its observable interface to that which is desired. The adapter aims to minimize changing the observable behaviour of the adapted.

### 4.2.3 Command

> "Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations." [4, p. 251]

The Command pattern is used to represent actions and commands as data that can be manipulated. An individual command is then responsible for performing its design action upon invocation of its "execute" (or equivalent) method. Thus, a system that can trigger a command to execute does not need to depend on the components the command uses to execute its action, and it is up to the creator of the command object to instantiate it with the required dependencies.

### 4.2.4 Composite

> "Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly." [4, p. 181]

The Composite pattern is used to represent a hierarchy of objects in a uniform way, whether the current hierarchy has child nodes or not. Thus, it is inconsequential to the client which part of the hierarchy it is currently holding, and it will be able to perform the same actions on that part of the hierarchy regardless. This simplifies dealing with hierarchies of objects and classes by having compositions of objects be responsible for dealing with their child objects as necessary, while leaf objects only have to deal with themselves as necessary.

### 4.2.5 Decorator

> "Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality." [4, p. 193]

The Decorator pattern is a method of extending and/or altering the behaviour of an object in a way that is transparent to the client. A decorator wraps the objects it intends to alter, and overrides one or more methods of the wrapped object, either altering the result returned by the wrapped object or replacing it entirely. Since a decorator implements the same interface as the wrapped object, decorators can be nested, each wrapping another decorator until the innermost decorator, which wraps the original object.

### 4.2.6 Iterator

> "Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation." [4, p. 275]

The Iterator pattern is a method of providing access to the elements of a collection without having to depend on the structure or implementation of said collection. This means structures like arrays, lists, trees, sets, etc. can be accessed in the same way, provided they can produce an associated iterator.

### 4.2.7 Prototype

> "Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype." [4, p. 137]

The Prototype pattern is a way of creating objects from an existing object instance. This existing object must provide some method of copying itself. Especially in languages with poor object instantiation syntax, this can simplify the act of creating multiple objects with the same configuration, and can also be used to provide a subset of preconfigured objects for duplication wherever they are needed.

### 4.2.8 Strategy

> "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it." [4, p. 331]

The Strategy pattern is used to abstract families of algorithms over a common interface. This allows us to decide at runtime which concrete algorithm to use for a given scenario. The client using the strategy object does not directly depend on the implementation of the strategy being used, only that it satisfies the interface for the strategy.

## 4.3 Functional Patterns

While OOP has many well-defined design patterns that solve various problems encountered in OOP, FP has patterns of its own that are used to solve problems encountered in FP. The following patterns are the functional patterns we will use in this thesis.

### 4.3.1 Newtype and smart constructor

If you want to specify some behaviour to a subset of values in a specific type you can make a `newtype` of that type and limit the creation of this new type.

For example: we want to represent positive numbers. If we have a value of this "positive numbers" type, it should never accidentally be a negative number.

```
module Positive
    ( Positive
    , getPositive
    , mkPositive
    ) where

newtype Positive n = MkPositive { getPositive :: n }
    deriving (Show)

mkPositive :: (Ord n, Num n) => n -> Maybe (Positive n)
mkPositive x
    | x >= 0    = Just (MkPositive x)
    | otherwise = Nothing
```

This is a pattern which goes by many names, but most often it is called "Newtyping" or "smart constructors". Here, `mkPositive` is a "smart constructor", which only gives us an actual `Positive` value if, and only if, (iff) `x` is a positive number.

Smart constructors allow us to create types with invariants that can be enforced.

### 4.3.2 Defunctionalization

Given a specific context, many functions can be represented as data. Let's say we have a function that sorts a list given some comparison function:

```
data Ordering
    = Eq
    | Lt
    | Gt

sortList :: (a -> a -> Ordering) -> [a] -> [a]
sortList f xs = ...
```

If we want to do more with the passed sorting function other than apply it, we have to supply this extra information through other means, like additional arguments. We also cannot serialize this function, since it is a function. There are also no guarantees that the passed function returns a sensible result, although such concerns should not lie with `sortList` itself, and instead is a concern for `sortList`'s caller.

Instead of passing a function, we can pass a data type which represents what we want to achieve.

```haskell
data Sort
    = Asc
    | Desc

sortList :: Sort -> [a] -> [a]
sortList sort xs = case sort of
    Asc -> ...
    Desc -> ...
```

The onus is then on `sortList` to implement the actual sorting, but it presents a clearer interface to the caller where it's clear that there are two, and only two, options: ascending and descending order. This also makes it simpler to test `sortList`, since there are only two kinds of sorting that need to be tested.

### 4.3.3 Type classes

In addition to actual code patterns, Haskell has type classes. Type classes are in essence design patterns as a language feature, to a certain extent. They group together types and values that can be used in the same way, abstracting over the actions we can perform on these types.

While Haskell has a lot of built-in type classes, we utilize only the following type classes from [19] when implementing solutions in FP.

#### Functor

> "A simple intuition is that a Functor represents a 'container' of some sort, along with the ability to apply a function uniformly to every element in the container." [23, p. 18]

Functors abstract the action of applying a function to the contents of a context or container. In Haskell, a functor has an implementation for the `fmap`[1] operation.

```haskell
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

To read this function signature in a rather verbose manner: if you give `fmap` a function `(a -> b)` and a functor that has `a`s, it will give you the same type of functor, but every element is replaced by applying the passed function to the contained `a`s, making them into `b`s.

Consider a singly linked list:

[1]It should have been named `map`, but the name `map` was already taken by list's specific mapping function by the time the Functor type class made its way into Haskell.

```haskell
data List a = Cons a (List a) | Nil
```

To “apply a function uniformly to every element” in the list, we can simply build a new list while applying the function to each element recursively.

```haskell
instance Functor List where
    fmap :: (a -> b) -> List a -> List b
    fmap _ Nil         = Nil
    fmap f (Cons x xs) = Cons (f x) (fmap f xs)
```

Another example with Maybe:

```haskell
data Maybe a = Just a | Nothing
```

```haskell
instance Functor Maybe where
    fmap :: (a -> b) -> Maybe a -> Maybe b
    fmap _ Nothing  = Nothing
    fmap f (Just x) = Just (f x)
```

As we can see, functor is mostly about unwrapping the functor and applying the function, as long as it makes sense to do so. It should be noted that there doesn’t even have to be anything in the container to map over:

```haskell
data Empty a = Empty
```

```haskell
instance Functor Empty where
    fmap :: (a -> b) -> Empty a -> Empty b
    fmap _ Empty = Empty
```

For Empty, the passed function will never be called, but you will get a Empty b back if you pass it a function (a -> b). Empty’s functor instance doesn’t even evaluate the function, meaning we can pass in undefined without issue.[2]

The Functor is the basis for many other type classes, and understanding it is paramount to understanding all other type classes that depend on it.

### Monad

```haskell
class Applicative m => Monad m where
    return :: a -> m a
    (>>=) :: m a -> (a -> m b) -> m b
```

[18]

Monads have two operations:

- return: puts a value of any type into a monadic context.
- >>=: Pronounced “bind”, this is the operation that distinctly defines the Monad class. In lay terms, >>= unwraps the contained a and gives it to the supplied function a -> m b, and returns that result. If, depending on the actual Monad in question, an a value cannot be supplied, m must be able to change from m a to m b through other means.

[2]While it might be difficult to see the use of a data type like Empty, it can be used to pass along type information to other functions.

Monads are an extension of Functors[3] in the way that it abstracts working with contexts in a similar fashion. Consider the similarities between Functor's `fmap` and a flipped version of Monad's bind (`>>=`)[4]:

```
fmap  :: Functor m => (a ->   b) -> m a -> m b
bind' :: Monad m   => (a -> m b) -> m a -> m b
```

With a functor, the function is not allowed access to the context of the functor whatsoever. With a monad, however, the function is allowed to return a new monadic context, which will be used as the result of the operation.

Using `Maybe` as an example:

```
instance Monad Maybe where
    return :: a -> Maybe a
    return x = Just x

    (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
    Just x  >>= f = f x
    Nothing >>= _ = Nothing
```

`return` does nothing out of the ordinary: it puts a value inside a `Maybe`. `>>=`, on the other hand, is the focus of monads. To call the provided function in `>>=`, we have to be able to provide it a value. As such, we pattern match on the `Maybe a`, extracting the `a` and calling the provided `f :: a -> Maybe b` function on it. The result of this function call is thus returned as the result of `>>=`.

Exactly how `>>=` is defined depends entirely on the monad in question, and can be radically different for different monads. The essence is just that we chain functions that return monadic contexts, whatever those contexts might contain. They could be error states, additional information, concurrency primitives, or even just plain values.

Haskell's `do`-syntax is merely syntactic sugar for `>>=`. The following two functions are equivalent, and the former is translated into the latter before compilation.

```
foo :: IO ()
foo = do
    x <- getLine
    y <- getLine
    let result = fmap toUpper x <> fmap toLower y
    putStrLn result

bar :: IO ()
bar =
    getLine >>= (\x ->
```

[3]In practice, they are an extension of *Applicative Functors*, but this thesis is already struggling with its page count as it is.

[4]Monad's bind operation normally has the type `(>>=) :: Monad m => m a -> (a -> m b)-> m b`, but it is easier to visualize the similarities between Functor and Monad with this flipped version.

```
    getLine >>= (\y ->
        let result = fmap toUpper x <> fmap toLower y
        in putStrLn result
    ))
```

### Semigroup

Semigroup describes things that can be combined into one.

Examples of this are numbers, where $1 + 1 = 2$ having the type `Int -> Int -> Int`, and string concatenation where `"foo" + "bar" = "foobar"`. Semigroups are defined as having "a binary, associative operation" defined for them.

```
class Semigroup a where
    (<>) :: a -> a -> a
```

If we define the semigroup instance for integers to be addition, then the following would be true: `1 <> 2 = 1 + 2`. Since Semigroups are so abstract, many types have more than one possible Semigroup available for them. Haskell does not allow a single type to have multiple instances of one type class, and we instead have to use `newtype`s to distinguish between the semigroup instances we want to use.

Considering `Int`s again, there are two possible Semigroup instances: addition and multiplication. Subtraction and division are not Semigroup instances, since they are not associative.

```
newtype Sum = Sum { getSum :: Int }
```

```
newtype Product = Product { getProduct :: Int }
```

These `newtype`s are just wrapping `Int`s, and for the semigroup and monoid instances we can just unwrap them to achieve the desired result:

```
instance Semigroup Sum where
    Sum x <> Sum y = Sum (x + y)
```

```
instance Semigroup Product where
    Product x <> Product y = Product (x * y)
```

List concatenation is also a semigroup:

```
instance Semigroup [a] where
    (<>) :: [a] -> [a] -> [a]
    []       <> ys = ys
    (x : xs) <> ys = x : (xs <> ys)
```

Semigroups allows us to abstract over combining things together.

### Monoid

Monoids are semigroups which also have a value which makes the combining operation return the other argument. This value is called the "identity" value of the monoid.

```haskell
class Semigroup a => Monoid a where
    mempty :: a
```

So for integers, the addition and multiplication instances have 1 and 0 as identity values respectively.

```haskell
instance Monoid Sum where
    mempty :: Sum
    mempty = Sum 0
```

```haskell
instance Monoid Product where
    mempty :: Product
    mempty = 1
```

For lists, the identity value is the empty list.

```haskell
instance Monoid [a] where
    mempty :: [a]
    mempty = []
```

Monoids appear in many shapes and sizes, and can be considered anything that can be combined (a semigroup), with some special value which does not affect the outcome.

### Foldable

A foldable value is a value that can be “folded” into another value. It is a generalization of converting from zero or more values into a different value, while not enforcing any constraints as to what this resulting value can be.

The Foldable class in Haskell has a lot of operations in its declaration, but only one of two need to be implemented to create a Foldable instance.

```haskell
class Foldable t where
    foldr :: (a -> b -> b) -> b -> t a -> b
    foldMap :: Monoid m => (a -> m) -> t a -> m

    foldl :: (b -> a -> b) -> b -> t a -> b
    foldr1 :: (a -> a -> a) -> t a -> a
    foldl1 :: (a -> a -> a) -> t a -> a
    elem :: Eq a => a -> t a -> Bool
    maximum :: Ord a => t a -> a
    minimum :: Ord a => t a -> a
    sum :: Num a => t a -> a
    product :: Num a => t a -> a
```

The main operations for Foldable are `foldr`, `foldl` and `foldMap`, where `foldl` folds the values from the left side, while `foldr` folds them from the right. `foldMap` converts the values to Monoids and combines them from the left. [19]

The minimum required to make a type foldable is to implement either `foldr` or `foldMap`. For simplicity, we will only consider `foldl` when using Foldable in this thesis, as folding from the left is the most common way to consuming the elements of a list.

Foldable does not have any strict laws that instances must adhere to, and as a result there exists a lot of strange and unusual instances of foldables. A prominent example of this is the foldable instance for two-tuples, `(a, )`.

```
instance Foldable ((,) e) where
    foldr :: (a -> b -> b) -> b -> (e, a) -> b
    foldr f z (_, x) = f x z
```

It is debatable whether tuples are foldable as they do not represent a series or a collection of values in the normal sense, but rather an indexable pair of values of possibly differing types. The function `length :: Foldable t => t a -> Int` returns the number of elements in the passed foldable. For two-tuples, it returns `1`, since that is the number of foldable values in a two-tuple. Most would surmise that there are two elements in a two-tuple, but Foldable says otherwise.

For lists, however, the Foldable instance makes much more sense.

```
instance Foldable [a] where
    foldr :: (a -> b -> b) -> b -> [a] -> b
    foldr _ z []       = z
    foldr f z (x : xs) = f x (foldr f z xs)
```

### MonadIO

For any Haskell program to interact with the outside world, whether it be printing to the screen or sending an HTTP request, the function making this interaction must invariably end up using `IO`. However, if we are in a different monad than `IO`, how would be use an `IO` action while still sticking to this other monad of ours?

`MonadIO` and its function `liftIO` allows a monad to "lift" `IO` actions into itself, where they will be executed as needed. As long as this other monad is defined in such a way that `IO` action can be performed, it can have an instance of `MonadIO`.

```
class Monad m => MonadIO m where
    liftIO :: IO a -> m a
```

`MonadIO` has two laws that instances must abide by to be considered valid `MonadIO` instances:

- `liftIO . return = return`: `liftIO` should not do any more work than `return` would do.
- `liftIO (m >>= f)= liftIO m >>= (liftIO . f)`: monadic bind should work equivalently in `IO` as in the instance monad.

These two laws ensure `MonadIO` merely acts as a transformer between `IO` and other monads.

To better illustrate this, assume we have a dedicated monad for sending HTTP requests

```haskell
newtype HTTP a = HTTP { runHTTP :: IO a }

get :: String -> HTTP ByteString
get url = ...

post :: String -> JSON -> HTTP ByteString
post url payload = ...
```

and assume we have a function that uses HTTP and prints the result

```haskell
import HTTP (HTTP, get)

printInvoices :: HTTP ()
printInvoices = do
    invoices <- get "/invoices"
    print invoices
```

the compiler would complain that print returns a value of type IO (), which cannot be performed in this do-block because it is of type HTTP.

To circumvent this we can lift IO into HTTP

```haskell
instance MonadIO HTTP where
    liftIO io = HTTP io
```

and use liftIO to embed IO actions in HTTP

```haskell
import HTTP (HTTP, get)

printInvoices :: HTTP ()
printInvoices = do
    invoices <- get "/invoices"
    liftIO (print invoices)
```

Thus, we can have IO effects and actions in other environments, as long as IO can be embedded in the other environments.

## 4.4 Summary of Design Patterns

In this chapter we have described and presented design patterns both in OOP and FP. We have presented the specific

# Part III

# Method

# Chapter 5

# Methodology

## 5.1 Introduction to methodology

This section describes the methods and practices performed during this thesis' main experiment. The validity and significance of both the experiment itself as well as the resulting data relies on understanding the limitations, biases and process of the experiment and the data. It is therefore imperative that the reader is provided as detailed of an understanding of the experiment and the methodology used during the gathering process.

## 5.2 Cases

This section presents the outline of the experiment. In this study, we have chosen eight design patterns to examine: Abstract Factory, Adapter, Command, Composite, Decorator, Iterator, Prototype and Strategy. For each of these design patterns, have designed a case with a high-level description of a system to implement using said pattern. We have then implemented the pattern in Java, satisfying the description provided in the design pattern's case. We have then also implemented the pattern in Haskell, using zero or more combinations of other functional patterns as we have seen fit, in an attempt to replicate the solution provided by the design pattern. Thus, we have two implementations for each design pattern, written in Java and Haskell, representing the OOP and FP programming paradigms respectively.

After the implementations are finalized, we collect data on the implementations, using the metrics described in Section 5.3. We also present our qualitative observations made when implementing the cases.

## 5.3 Metrics

### 5.3.1 Introduction

In this section we present the metrics used for data gathering and numeric analysis for the cases.

Differences in solution implementation, whether it be with a design pattern in OOP or other patterns in FP, are varied and numerous. Thus, we strive to include as many of these differences and qualities in our metrics when evaluating the differences between the implementations, while also limiting ourselves to those most applicable and impactful.

To achieve this we want to select metrics that are cheap to evaluate and only concerns one area of differences with as little overlap between metrics and the qualities they measure as possible.

### 5.3.2 Example

Through the rest of this section, we will be using two examples to illustrate the qualities of the metrics used.

```haskell
{-# LANGUAGE ExistentialQuantification #-}

-- | Example code used to explain the metrics used
-- in this thesis.

import Data.Foldable (traverse_)

class Speaker a where
    speak :: a -> IO ()

newtype Canine = Canine { sound :: String }

instance Speaker Canine where
    speak (Canine sound) = putStrLn sound

type Dog = Canine

type Wolf = Canine

mkDog :: Dog
mkDog = Canine "Woof"

mkWolf :: Wolf
mkWolf = Canine "Awoo"

newtype Cat = Cat { size :: Int }

mkCat :: Int -> Cat
mkCat = Cat

instance Speaker Cat where
    speak (Cat size) = if size > 2
        then rawr
        else meow
      where
        meow = putStrLn "Meow"
        rawr = putStrLn "Rawr"
```

```haskell

-- ExistentialQuantification used here
data SomeSpeaker =
    forall a . Speaker a => SomeSpeaker a

instance Speaker SomeSpeaker where
    speak (SomeSpeaker x) = speak x

main :: IO ()
main = do
    let
        zoo =
            [ SomeSpeaker (mkCat 1)
            , SomeSpeaker (mkCat 3)
            , SomeSpeaker mkDog
            , SomeSpeaker mkWolf
            ]
    traverse_ speak zoo
```

Listing 5.1: Haskell example code

```java
// Example code used to explain the metrics used
// in this thesis.


interface Speaker {
    public void speak();
}

abstract class SpeaksWithSound implements Speaker {
    protected String sound;

    protected SpeaksWithSound(String sound) {
        this.sound = sound;
    }

    public void speak() {
        System.out.println(this.sound);
    }
}

class Dog extends SpeaksWithSound {
    public Dog() {
        super("Woof");
    }
}

class Wolf extends SpeaksWithSound {
    public Wolf() {
        super("Awoo");
    }
}
```

```java

class Cat implements Speaker {
    private static final String SMALL_SOUND = "Meow";
    private static final String BIG_SOUND = "Rawr";
    private int size;

    public Cat (int size) {
        this.size = size;
    }

    public void speak() {
        if (size > 2) {
            System.out.println(BIG_SOUND);
        } else {
            System.out.println(SMALL_SOUND);
        }
    }
}


class MetricExample {
    public static void main(String... args) {
        Speaker[] zoo = new Speaker[] {
            new Cat(1),
            new Cat(3),
            new Dog(),
            new Wolf()
        };

        for (Speaker animal : zoo) {
            animal.speak();
        }
    }
}
```

Listing 5.2: Java example code

Both of these programs output the following:

```
Meow
Rawr
Woof
Awoo
```

### 5.3.3 Source Lines of code

Source lines of code (SLOC) is an extremely common metric of program size measured by simply counting the number of lines the code spans in a given program. The significance of SLOC depends heavily which lines specifically are counted, as well as how the code is structured. By defining a set of rules governing how we count SLOC, we can paint a picture of how *large* a program is. In this thesis, we use the rules described by Nguyen et al. in ‘A SLOC Counting Standard’ [15]. They define two types of SLOC to count: PSLOC and LSLOC.

**Physical SLOC** Counts all lines that are neither blank nor contain only comments. [15] The lines which make up the significant syntax of the language (provided blank lines are not a part of the syntax of the language). PSLOC can thus be considered the physical size of the code in question, and how much visual space it occupies.

**Logical SLOC** Counts lines according to their contents and significance in the language and/or syntax. [15] This is where OOP and FP start to differ. [15] only specify a set of LSLOC counting rules for C, C++, Java and C# (and by extension most C-like languages). As such, we define a set of LSLOC counting rules for Haskell, derived from the rules specified in [15], in table 5.1

| Structure | Precedence | Logical SLOC Rules |
|---|---|---|
| Selection statements: `if then`, `else if then`, `else` , `catch`, `case of`, pattern matching | 1 | Count once per occurrence. Nested statements are counted in a similar fashion. |
| Expression statements: function call, assignment, operators, record syntax, ADT constructor application | 2 | Count once per occurrence. Includes TemplateHaskell expressions. Do not count bare expressions with no function application. |
| Block delimiters, braces: block of code, `do` block | 3 | Count once per `do` block. Function definitions are counted once, since they constitute a "block". Count once per lambda function, since they effectively constitute a function declaration. |
| Compiler directive, CPP pragma | 4 | Count once per occurrence |
| Data declaration | 5 | Count once per occurrence. |
| Type declaration: Type synonym, `data` declaration, `newtype` declaration | 6 | Count once per occurrence. Count once per `data` constructor, and once per `newtype`. |

Table 5.1: Rules for counting LSLOC in Haskell

Long lines of text is sometimes wrapped over multiple lines in source code to prevent horizontal scrolling while viewing the source code. As such, we add another rule to the aforementioned LSLOC rules that any string concatenation where both the operands are string literals will not be counted as a LSLOC. Thus, the following would just be counted as one LSLOC:

```
return "foo" + "bar" + "baz";
```

While the following would indeed be counted as two:

```
return "foo" + bar + "baz";
```

Assuming `bar` is some defined variable.

### 5.3.4 Cyclomatic complexity

The CC of a program is a measure of program complexity. First introduced in [12], it is applied to the control flow graph of the program, where the nodes of the graph represent groups of commands separated by control flow statements like conditionals and decision points. Two nodes `A` and `B` would then be considered connected if `B` can be executed immediately following `A`.

For a program `P`, the CC of program `P`, `v(P)`, is defined as follows:

$$\begin{aligned} v(P) &= e - n + 2p \\ e &= \text{Number of edges in the graph} \\ n &= \text{Number of nodes in the graph} \\ p &= \text{Number of connected components in the graph} \end{aligned} \tag{5.1}$$

CC does not provide an adequate picture on its own as to how complex a program is; a program with a low complexity number might be more complex than a program with a high complexity number, and vice versa. CC only describes how many different paths there are in the code, and not how chaotically they might be connected.

In regards to our example programs,

### 5.3.5 Execution time

Comparing programming languages by execution time is difficult, and most often a detriment to the comparison of the languages. It probably is in this case as well. While both languages are compiled languages, Haskell compiles to x86 instructions while Java compiles to JVM instructions, which are then interpreted and compiled "just-in-time" at runtime.

Using benchmarking programs, we can measure the total run time of a given program. In this thesis, we use the benchmarking program "Hyperfine" [17]. In all measurements in this thesis, we invoke Hyperfine with the option `--warmup 10`, which informs Hyperfine to execute the program ten times before measuring to ensure the program is benchmarked against hot CPU caches, file caches, and the like, minimizing deviation between the first run of the benchmark from the rest.

Using Hyperfine, the example code in listing 5.1 and 5.2 are measured at 9.9 ms and 89.0 ms, respectively. These numbers by themselves do not properly communicate the actual run time of the two example programs. An entirely empty Java program, with only an empty `main` method, takes 74.0 ms to execute to completion, while an empty Haskell program takes 6.9 ms. Java has the JVM which needs to do its work, while Haskell has its runtime which marshals IO, so the comparison is very hard to make with very small programs like this.

As such, we can assume there will always be at least an 74.0 ms overhead to Java programs and a 6.9 ms overhead to Haskell programs. However, the actual overhead of the runtime system for each language varies depending on the compiled code and what actions we perform in the program. This mostly depends on when the value is calculated, which varies wildly between the eager JVM and the lazily computed Haskell runtime.

### 5.3.6 Cyclomatic complexity density

Measuring CC and LSLOC by themselves gives some insight into the differences between the languages. Large programs with high LSLOC tend also to have a higher CC. It can thus be stated that for any given program, as LSLOC increases, so does CC, most of the time. The correlation between CC and LSLOC, also called cyclomatic complexity density (CCrho) can be given as follows, for a given case $C$ in a given programming language $P$:

$$\begin{aligned} CC_\rho(C,P) &= \frac{CC(C,P)}{LSLOC(C,P)} \\ CC(C,P) &= \text{CC for language P in case C} \\ LSLOC(C,P) &= \text{LSLOC for language P in case C} \end{aligned} \tag{5.2}$$

### 5.3.7 Code

Comparing the programs presented in this thesis by the above metrics alone would barely paint even a faint picture of the qualities that make up their differences. As such, we compare and contrast various language constructs as they appear in the programs. The code produced in this thesis is available in full at https://github.com/Gipphe/PLMasterThesis.

## 5.4 Summary of methodology

In this chapter we have presented and described the methodology and approach for this study. We have outlined how the cases are structured, and the various metrics that will be used to collect data on them.

# Part IV

# Cases

# Chapter 6

# Cases

## 6.1 Introduction to cases

In this chapter we present the design patterns examined for this thesis, the case wherein we have utilized the design pattern in question, the implementations made to satisfy the case, as well as our findings and measured metrics for the implementations. Finally, we summarize our findings and compare between the cases.

In the case description, we define a set of requirements for the case's implementation. This description is high-level and not specific to any particular thought model or paradigm, as OOP and FP often have very different ways to model a domain. We present these requirements as a "case statement" to be satisfied by the implementations.

Along with the case statement, we present a UML class diagram of the design pattern in question, tailored to the case in question. This allows us an overview of the relationships between the different components of the case. Although FP does not have classes and objects in the same way as OOP; and thus a UML class diagram is not as applicable to an FP context; data types, functions and constructors will correspond to the elements of the diagram in most cases, and as such provides an adequate overview of the relationship between elements of the FP implementation as well.

All code produced and referenced in this chapter is contained in the repository described in Subsection 5.3.7.

## 6.2 Abstract Factory

The Abstract Factory pattern is concerned with object creation, where a factory creates sets of dependent objects that satisfy abstract interfaces.[4]

### 6.2.1 Abstract Factory case

In this case we model a retailer for cutlery and table setting.

> Implement a system where a client can "order" sets of cutlery. One set of cutlery includes a fork and a knife, which can then be

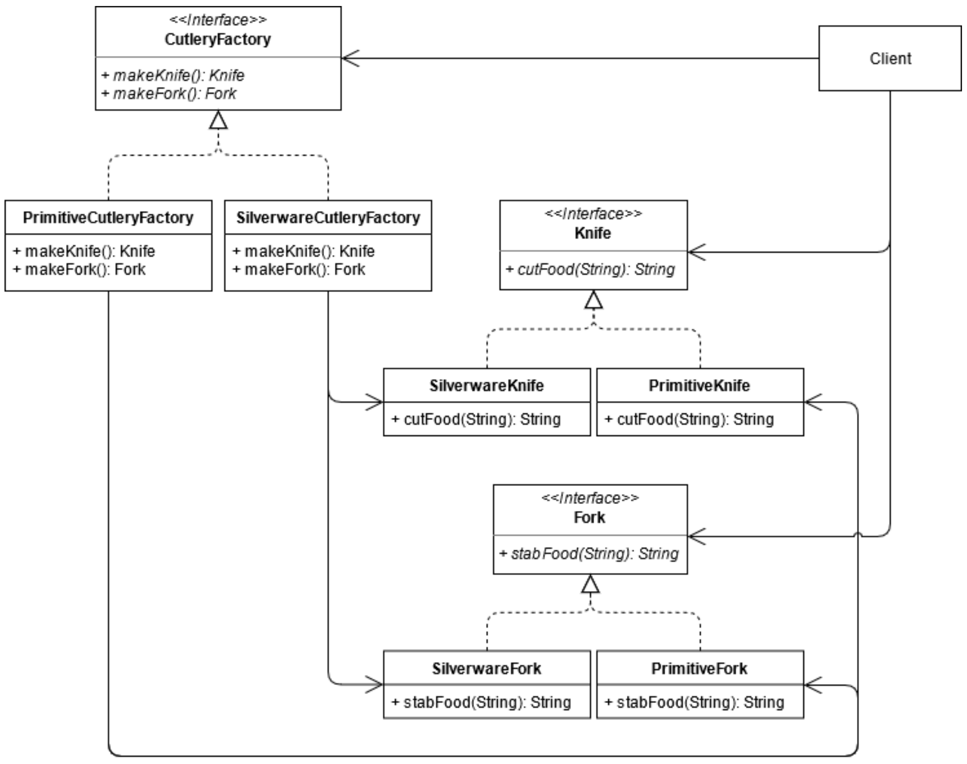


Figure 6.1: UML class diagram for the Abstract Factory case.

used to eat with. Each piece of cutlery must be usable on any food.

A mock client must be implemented, asking for "silver" cutlery and "primitive" cutlery, and using each type of cutlery for a meal.

The result of running the program must be a string of text describing eating the meals with the various cutlery, with each piece of cutlery supplying a distinct description from the rest.

### 6.2.2 Implementations for Abstract Factory

The code for the Java implementation is contained in the repository's "Java/Implementations/Abstract Factory" folder, while the Haskell implementation is contained in the "Haskell/Implementations/Abstract Factory" folder.

**Java implementation** To implement this case in Java, we created a base class `CutleryFactory` containing the factory methods `makeKnife` and `makeFork`, which produce the abstract `Knife` and `Fork` respectively. `PrimitiveCutleryFactory` and `SilverwareCutleryFactory` implement `CutleryFactory` using `PrimitiveKnife`, `PrimitiveFork`, `SilverwareKnife` and `SilverwareFork`. `Main` is our mock client, which we use to create an instance of `SilverwareCutleryFactory` and

PrimitiveCutleryFactory, and subsequently consume one meal with each set of cutlery.

**Haskell implementation** To implement this case in Haskell, the interface CutleryFactory is instead a record of functions, but since they do not take any parameters whatsoever the makeFork and makeKnife instead just return a set of pre-made forks and knives. Due to immutability, we do not need "object instances", and thus need not create a new Fork and Knife for each makeFork and makeKnife call. Fork and Knife are also just records with "silver" and "primitive" specific properties set on creation. silverwareCutleryFactory and primitiveCutleryFactory are thus merely CutleryFactory records with specific Fork and Knife smart constructors. This set up allows us to treat all CutleryFactorys, Forks and Knifes identically, preserving their implementation details to the smart constructors used to create them. Like in the Java implementation, the Main module acts as our client, specifying the type of cutlery to create and the meals to eat with the cutlery.

### 6.2.3 Metrics for Abstract Factory implementations

In this subsection we list the metrics results for the Abstract Factory implementations, and briefly describe our observations for this particular case.

#### Source lines of code for Abstract Factory implementations

In this section we list the LSLOC and PSLOC metrics for the Abstract Factory implementations.

**Logical source lines of code for Abstract Factory implementations** In Table 6.1 we list the LSLOC for each Abstract Factory implementations' files.

The Java implementation is $88 \div 58 = 1.52$ times larger than the Haskell implementation in terms of LSLOC.

**Physical source lines of code for Abstract Factory implementations** In Table 6.2 we list the PSLOC for each Abstract Factory implementations' files.

The Java implementation is $132 \div 114 = 1.16$ times larger than the Haskell implementation in terms of PSLOC.

#### Cyclomatic complexity for Abstract Factory implementations

In Table 6.3 we list the CC for each Abstract Factory implementations' files.

The Haskell implementation's total CC is 14, while the Java implementation's is 24. The Java implementation CC is $24 \div 14 = 1.71$ times greater than the Haskell implementation CC.

| **LSLOC for Abstract Factory** | |
|---|---|
| **File** | **LSLOC** |
| **Haskell** | |
| Cutlery.hs | 9 |
| Fork.hs | 11 |
| Knife.hs | 9 |
| Main.hs | 29 |
| Total | 58 |
| **Java** | |
| CutleryFactory.java | 0 |
| Fork.java | 10 |
| Knife.java | 10 |
| Main.java | 34 |
| PrimitiveCutleryFactory.java | 7 |
| PrimitiveFork.java | 5 |
| PrimitiveKnife.java | 5 |
| SilverwareCutleryFactory.java | 7 |
| SilverwareFork.java | 5 |
| SilverwareKnife.java | 5 |
| Total | 88 |

Table 6.1: Logical source lines of code for Abstract Factory implementations

| **PSLOC for Abstract Factory** | |
|---|---|
| **File** | **PSLOC** |
| **Haskell** | |
| Cutlery.hs | 22 |
| Fork.hs | 32 |
| Knife.hs | 30 |
| Main.hs | 30 |
| Total | 114 |
| **Java** | |
| CutleryFactory.java | 4 |
| Fork.java | 15 |
| Knife.java | 15 |
| Main.java | 32 |
| PrimitiveCutleryFactory.java | 8 |
| PrimitiveFork.java | 14 |
| PrimitiveKnife.java | 12 |
| SilverwareCutleryFactory.java | 8 |
| SilverwareFork.java | 12 |
| SilverwareKnife.java | 12 |
| Total | 132 |

Table 6.2: Physical source lines of code for Abstract Factory implementations

| **CC for Abstract Factory** | |
|---|---|
| **File** | **CC** |
| **Haskell** | |
| Cutlery.hs | 3 |
| Fork.hs | 2 |
| Knife.hs | 2 |
| Main.hs | 7 |
| Total | 14 |
| **Java** | |
| CutleryFactory.java | 0 |
| Fork.java | 3 |
| Knife.java | 3 |
| Main.java | 6 |
| PrimitiveCutleryFactory.java | 2 |
| PrimitiveFork.java | 2 |
| PrimitiveKnife.java | 2 |
| SilverwareCutleryFactory.java | 2 |
| SilverwareFork.java | 2 |
| SilverwareKnife.java | 2 |
| Total | 24 |

Table 6.3: Cyclomatic complexity for Abstract Factory implementations

| **Execution times for Abstract Factory** | | | | |
|---|---|---|---|---|
| **Command** | **Mean [ms]** | **Min [ms]** | **Max [ms]** | **Relative** |
| Java | $90.09 \pm 1.23$ | 87.43 | 91.57 | 12.21 |
| Haskell | $7.38 \pm 0.17$ | 7.05 | 8.19 | 1 |

Table 6.4: Execution times for Abstract Factory implementations

### Execution time for Abstract Factory implementations

In Table 6.4 we list the execution time for each Abstract Factory implementation.

The Java implementation is on average approximately $90.09 \div 7.38 = 12.21$ times slower than the Haskell implementation.

Accounting for the execution time overhead calculated in Section 5.3.5, we get the following results:

$$\begin{aligned}\text{Java: } 90.09 - 74.0 &= 16.09\\ \text{Haskell: } 7.38 - 6.9 &= 0.48\end{aligned} \tag{6.1}$$

With these adjusted numbers, the Java implementation is $16.09 \div 0.48 = 33.52$ times slower than the Haskell implementation.

**Cyclomatic complexity density for Abstract Factory implementations**

Using the results from our LSLOC and CC measurements for this case, we calculate the CC density as follows:

$$\begin{aligned} CC_\rho(\text{Abstract Factory}, \text{Java}) &= \frac{CC(\text{Abstract Factory}, \text{Java})}{LSLOC(\text{Abstract Factory}, \text{Java})} \\ &= \frac{24}{88} \\ &= 0.2727 \end{aligned} \tag{6.2}$$

$$\begin{aligned} CC_\rho(\text{Abstract Factory}, \text{Haskell}) &= \frac{CC(\text{Abstract Factory}, \text{Haskell})}{LSLOC(\text{Abstract Factory}, \text{Haskell})} \\ &= \frac{14}{58} \\ &= 0.2414 \end{aligned} \tag{6.3}$$

The Java implementation is approximately $0.2727 \div 0.2414 = 1.1297$ times more logically dense than the Haskell implementation for the given LSLOC and CC measurements.

### 6.2.4 Comparison of Abstract Factory implementations

In this section, we summarize our observations made while making the implementations for Abstract Factory.

Both implementations feature roughly the same abstractions:

- Both abstract over some general `CutleryFactory`, where Java uses an interface with classes as concretions, while Haskell uses a data type with smart constructors as concretions.
- Both abstract over general `Knife` and `Fork` in the same way they abstract over `CutleryFactory`.

But they feature some key differences:

- Java creates new objects for each `makeKnife` and `makeFork` call, while Haskell returns the same `Fork` and `Knife` for each concrete call. Java objects are immutable, and as such returning new instances of the objects makes more sense especially in a context in which the provided `Fork` or `Knife`, or whatever the abstraction might be, can change. In Haskell, these changes always result in a new `Fork` or `Knife` due to immutability, and as such "changing" one `Fork` will not change that given `Fork` any other place it has been referenced.

In FP, there are multiple ways to achieve the same level of decoupling and flexibility that Abstract Factory offers. One of them, which we used here, is to have a general data type with a hidden constructor act as an "interface", with smart constructors being the concrete implementations of that "interface". Abstract Factory can be summarized as "an interface creating interfaces", where you depend on an abstract interface for a factory to produce objects which implement some abstract interfaces. In Haskell, we achieve this by "nesting" the "smart constructors" pattern.

To summarize: the FP solution to Abstract Factory's problem is (among others) the "smart constructor" pattern.

## 6.3 Adapter

The adapter pattern is concerned with having incompatible interfaces and classes communicate though a compatibility layer or class. This compatibility layer or class is sometimes called a "shim", or even a "transformer".[4]

### 6.3.1 Adapter case

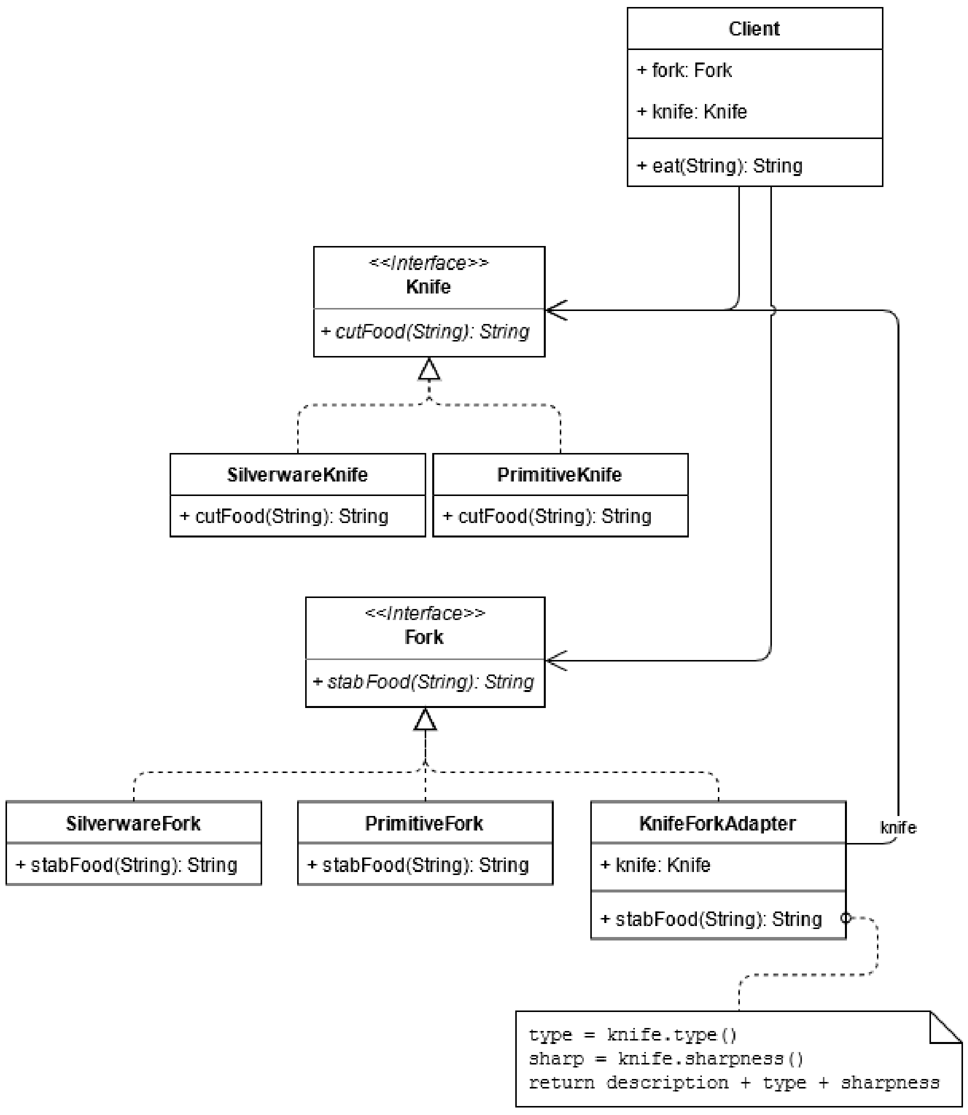
We adapt the Abstract Factory case to centre more around the usage of cutlery rather than their creation.

> Implement a system where a client can hold a fork and a knife, and use the fork and knife to eat food. There must exist native knives and forks that can be used by the client.
>
> Much like in the real world (and especially when you are a lazy student who does not wash their dishes often enough), it must also be possible to use a knife as a make-shift, single-pronged fork. As such, there must exist a compatibility layer that allows knives to be used as forks by a client.
>
> The result of running the program must be a string of text describing eating the meals with the various cutlery, with each piece of cutlery supplying a distinct description from the rest.

### 6.3.2 Implementations for Adapter

The code for the Java implementation is contained in the repository's "Java/Implementations/Adapter" folder, while the Haskell implementation is contained in the "Haskell/Implementations/Adapter" folder.

**Java** To implement this case in Java we have a "client" class, `Cutlery`, which requires a `Fork` and a `Knife`. We have defined three concrete `Knife` and `Fork` classes, `SilverKnife`, `SteelKnife`, `WoodenKnife`, `SilverFork`, `SteelFork` and `WoodenFork`. These are the "native" cutlery. In addition to this, we have an adapter class, `KnifeForkAdapter`, for using `Knife` objects as `Fork` objects, as per the stated requirements. We build up a set of `Cutlery` objects (some containing two knives, where one of them is adapted into a `Fork`) in `Main`, and `eat` with each of these `Client`s.


Client
+ fork: Fork
+ knife: Knife
+ eat(String): String
<<Interface>>
Knife
+ cutFood(String): String
SilverwareKnife
+ cutFood(String): String
PrimitiveKnife
+ cutFood(String): String
<<Interface>>
Fork
+ stabFood(String): String
SilverwareFork
+ stabFood(String): String
PrimitiveFork
+ stabFood(String): String
KnifeForkAdapter
+ knife: Knife
+ stabFood(String): String
knife

```
type = knife.type()
sharp = knife.sharpness()
return description + type + sharpness
```

Figure 6.2: UML class diagram for the Adapter case.

**Haskell** To implement this case in Haskell we have an existential datatype `Client` which takes a value that implements the `Fork` type class and a value that implements the `Knife` type class, where `Fork` and `Knife` define the necessary functions for using the cutlery. Because `Client` is existential, we can build up a list of `Client`s that each have different types of knives and forks. We then have "concretions" of `Knife` and `Fork` in the form of the data types `SilverKnife`, `SteelKnife`, `WoodenKnife`, `SilverFork`, `SteelFork` and `WoodenFork`. Our knife-to-fork adapter is the `KnifeForkAdapter` data type. This data type takes one type argument. Its adapter-functionality comes forth in its type class instance, which requires said type argument to be an instance of `Knife` for itself to be a valid `Fork` instance. Like in the Java implementation, we build up a list of `Client`-food pairs, and call `eat` on each of them.

### 6.3.3 Metrics for Adapter implementations

In this subsection we list the metrics results for the Adapter implementations, and briefly describe our observations for this particular case.

#### Source lines of code for Adapter implementations

In this section we list the LSLOC and PSLOC metrics for the Adapter implementations.

**Logical source lines of code for Adapter implementations** In Table 6.5 we list the LSLOC for each Adapter implementations' files.

The Java implementation is $165 \div 100 = 1.65$ times larger than the Haskell implementation in terms of LSLOC.

**Physical source lines of code for Adapter implementations** In Table 6.6 we list the PSLOC for each Adapter implementations' files.

The Java implementation is $185 \div 137 = 1.35$ times larger than the Haskell implementation in terms of PSLOC.

#### Cyclomatic complexity for Adapter implementations

In Table 6.7 we list the CC for each Adapter implementations' files.

The Haskell implementation's total CC is 31, while the Java implementation's is 32. The Java implementation CC is $32 \div 31 = 1.03$ times greater than the Haskell implementation CC.

#### Execution time for Adapter implementations

In Table 6.8 we list the execution time for each Adapter implementation.

The Java implementation is on average approximately $99.85 \div 7.30 = 13.68$ times slower than the Haskell implementation.

Accounting for the execution time overhead calculated in Section 5.3.5, we get the following results:

| **LSLOC for Adapter** | |
|---|---|
| **File** | **LSLOC** |
| **Haskell** | |
| Client.hs | 8 |
| Fork.hs | 31 |
| Knife.hs | 38 |
| Main.hs | 20 |
| Util.hs | 3 |
| Total | 100 |
| **Java** | |
| Client.java | 12 |
| Fork.java | 0 |
| Knife.java | 0 |
| KnifeForkAdapter.java | 20 |
| Main.java | 51 |
| SilverFork.java | 15 |
| SilverKnife.java | 15 |
| SteelFork.java | 14 |
| SteelKnife.java | 11 |
| Utils.java | 6 |
| WoodenFork.java | 10 |
| WoodenKnife.java | 11 |
| Total | 165 |

Table 6.5: Logical source lines of code for Adapter implementations

| **PSLOC for Adapter** | |
|---|---|
| **File** | **PSLOC** |
| **Haskell** | |
| Client.hs | 11 |
| Fork.hs | 43 |
| Knife.hs | 59 |
| Main.hs | 17 |
| Util.hs | 7 |
| Total | 137 |
| **Java** | |
| Client.java | 14 |
| Fork.java | 5 |
| Knife.java | 6 |
| KnifeForkAdapter.java | 21 |
| Main.java | 34 |
| SilverFork.java | 17 |
| SilverKnife.java | 18 |
| SteelFork.java | 16 |
| SteelKnife.java | 17 |
| Utils.java | 6 |
| WoodenFork.java | 14 |
| WoodenKnife.java | 17 |
| Total | 185 |

Table 6.6: Physical source lines of code for Adapter implementations

| **CC for Adapter** | |
|---|---|
| **File** | **CC** |
| **Haskell** | |
| Client.hs | 1 |
| Fork.hs | 9 |
| Knife.hs | 15 |
| Main.hs | 4 |
| Util.hs | 2 |
| Total | 31 |
| **Java** | |
| Client.java | 2 |
| Fork.java | 0 |
| Knife.java | 0 |
| KnifeForkAdapter.java | 4 |
| Main.java | 3 |
| SilverFork.java | 3 |
| SilverKnife.java | 4 |
| SteelFork.java | 3 |
| SteelKnife.java | 4 |
| Utils.java | 2 |
| WoodenFork.java | 3 |
| WoodenKnife.java | 4 |
| Total | 32 |

Table 6.7: Cyclomatic complexity for Adapter implementations

| **Execution times for Adapter** | | | | |
|---|---|---|---|---|
| **Command** | **Mean [ms]** | **Min [ms]** | **Max [ms]** | **Relative** |
| Java | $99.85 \pm 6.85$ | 90.24 | 106.43 | 13.68 |
| Haskell | $7.3 \pm 0.16$ | 6.98 | 8.26 | 1 |

Table 6.8: Execution times for Adapter implementations

$$\begin{aligned} \text{Java: } 99.85 - 74.0 &= 25.85 \\ \text{Haskell: } 7.30 - 6.9 &= 0.40 \end{aligned} \tag{6.4}$$

With these adjusted numbers, the Java implementation is $25.85 \div 0.40 = 64.63$ times slower than the Haskell implementation.

#### Cyclomatic complexity density for Adapter implementations

Using the results from our LSLOC and CC measurements for this case, we calculate the CC density as follows:

$$\begin{aligned} CC_\rho(\text{Adapter}, \text{Java}) &= \frac{CC(\text{Adapter}, \text{Java})}{LSLOC(\text{Adapter}, \text{Java})} \\ &= \frac{32}{165} \\ &= 0.1939 \end{aligned} \tag{6.5}$$

$$\begin{aligned} CC_\rho(\text{Adapter}, \text{Haskell}) &= \frac{CC(\text{Adapter}, \text{Haskell})}{LSLOC(\text{Adapter}, \text{Haskell})} \\ &= \frac{31}{100} \\ &= 0.3100 \end{aligned} \tag{6.6}$$

The Haskell implementation is approximately $0.3100 \div 0.1939 = 1.5988$ times more logically dense than the Java implementation for the given LSLOC and CC measurements.

### 6.3.4 Comparison of Adapter implementations

In this section, we summarize our observations made while making the implementations for Adapter.

- Both abstract over `Knife` and `Fork`, where Java uses interfaces while Haskell uses type classes. Interfaces and type classes are very similar in shape a lot of the time.
- Both have a concrete `Client` which requires a fork and a knife that fit the `Fork` and `Knife` interfaces/type classes.

In practice, there are very few differences between how the adapter pattern is handled in each of these implementations. This is mainly because the Haskell implementation uses type classes to abstract over `Fork` and `Knife` functionality. Had a different abstraction been used, the result would have probably been considerably different from the Java implementation.

Exactly how to solve the Adapter problem in FP depends on how the two incompatible interfaces are structured and implemented. In the case given in this thesis, both were type classes, and as such the adapter would just be a `newtype` pattern around a `Knife`, which adapted the `Knife` type class to function as a `Fork` type class.

## 6.4 Command

The Command pattern is concerned with making actions and operations more flexible, and to abstract over them as data instead of hard-coded method calls.[4]

### 6.4.1 Command case

We model a very rudimentary Email client, used for writing, signing and sending Emails.

> Implement an Email application. This application must be operable by a mock "operator". This operator must write the message subject, write the message body, have the email cryptographically signed and send the email. These actions that the operator perform must allow grouping. The operator interacts with the system by issuing these actions to the system, which will then run the action as a `Command` object.
>
> Once a given Email has been sent, it cannot be edited further; sent Emails are immutable.

### 6.4.2 Implementations for Command

The code for the Java implementation is contained in the repository's "Java/Implementations/Command" folder, while the Haskell implementation is contained in the "Haskell/Implementations/Command" folder.

**Java** To implement this case in Java we have the "client" class `Operator`. `Operator` just has two static methods that emulate a "human" operator. These methods build up lists of `Command`s, using the various `Command` implementations: `SignEmailCommand`, `SendEmailCommand` and `TextCommand` . The first two commands have rather self-explanatory names. `TextCommand` takes a text string and a `Supplier` object. When executed, it sends the text string to the `Supplier` object. As a result, we can use `TextCommand` to update the body and subject of the email as it is being edited. `Operator` dispatches the `Command` objects through a `User`. Each of these commands use `EmailEditor` and `MailServer` instances to do their work, which are implemented by `Email.Builder` and `ConsoleMailServer` respectively.

**Haskell** To implement this case in Haskell we have the "client" module `Operator`. Much like the Java implementation, `Operator` uses a `User` to build up lists of `Command`s which are then dispatched through the `User` module. These commands are dispatched

### 6.4.3 Metrics for Command implementations

In this subsection we list the metrics results for the Command implementations, and briefly describe our observations for this particular case.

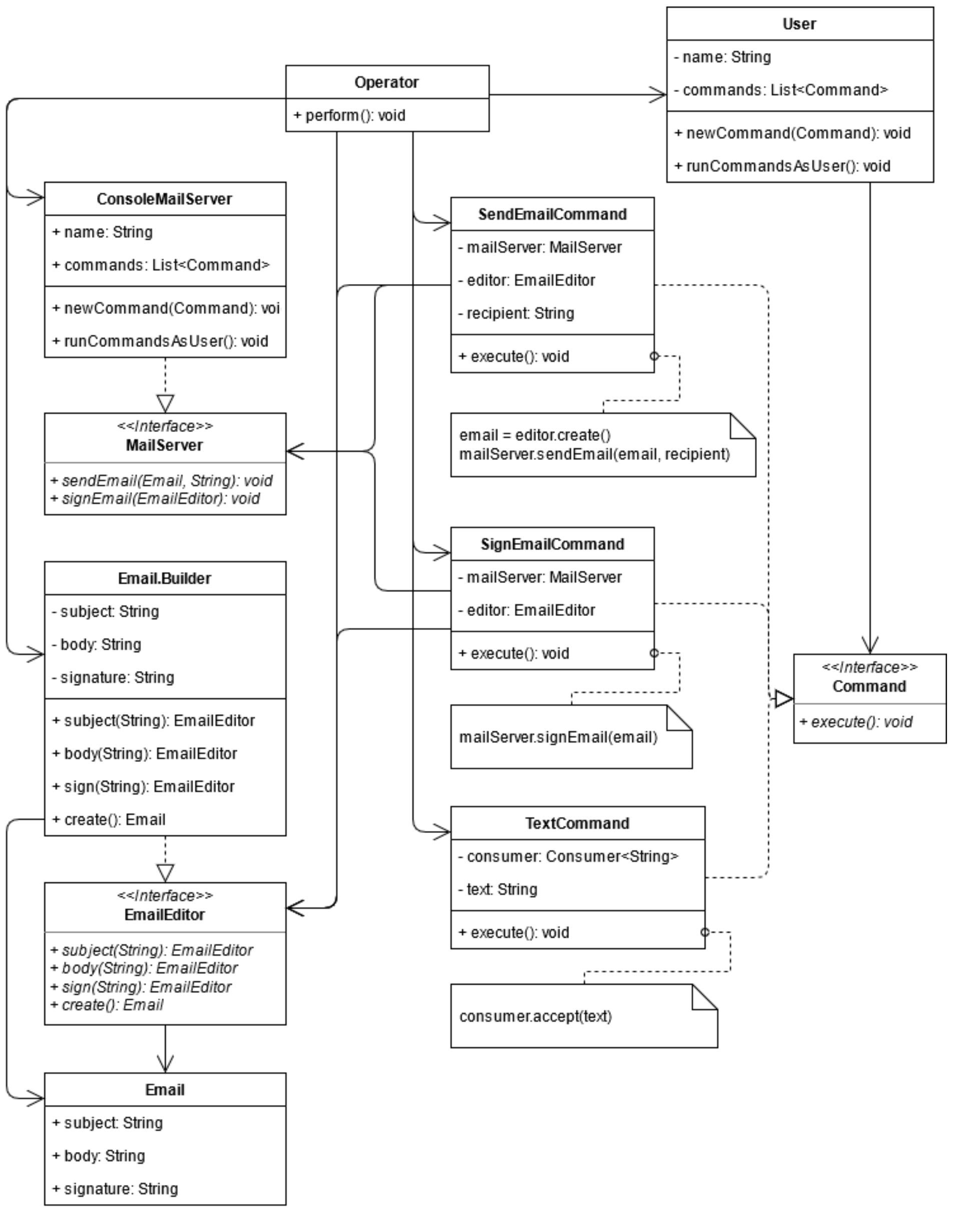


Figure 6.3: UML class diagram for the Command case.

| **LSLOC for Command** | |
|---|---|
| **File** | **LSLOC** |
| **Haskell** | |
| Command.hs | 21 |
| Email.hs | 24 |
| Main.hs | 3 |
| Operator.hs | 93 |
| User.hs | 15 |
| Total | 156 |
| **Java** | |
| Command.java | 0 |
| ConsoleMailServer.java | 17 |
| Email.java | 24 |
| EmailEditor.java | 0 |
| MacroCommand.java | 10 |
| MailServer.java | 0 |
| Main.java | 10 |
| Operator.java | 71 |
| SendEmailCommand.java | 12 |
| SignEmailCommand.java | 8 |
| TextCommand.java | 9 |
| User.java | 15 |
| Total | 176 |

Table 6.9: Logical source lines of code for Command implementations

### Source lines of code (SLOC) for Command implementations

In this section we list the LSLOC and PSLOC metrics for the Command implementations.

**Logical source lines of code for Command implementations** In Table 6.9 we list the LSLOC for each Command implementations' files.

The Java implementation is $176 \div 156 = 1.13$ times larger than the Haskell implementation in terms of LSLOC.

**Physical source lines of code for Command implementations** In Table 6.10 we list the PSLOC for each Command implementations' files.

The Java implementation is $209 \div 147 = 1.42$ times larger than the Haskell implementation in terms of PSLOC.

### Cyclomatic complexity for Command implementations

In Table 6.11 we list the CC for each Command implementations' files.

The Java implementation's total CC is 35, while the Haskell implementation's is 40. The Haskell implementation CC is $40 \div 35 = 1.14$ times greater than the Java implementation CC.

| PSLOC for Command | |
|---|---|
| **File** | **PSLOC** |
| **Haskell** | |
| Command.hs | 26 |
| Email.hs | 30 |
| Main.hs | 6 |
| Operator.hs | 67 |
| User.hs | 18 |
| Total | 147 |
| **Java** | |
| Command.java | 3 |
| ConsoleMailServer.java | 17 |
| Email.java | 31 |
| EmailEditor.java | 6 |
| MacroCommand.java | 13 |
| MailServer.java | 4 |
| Main.java | 9 |
| Operator.java | 69 |
| SendEmailCommand.java | 14 |
| SignEmailCommand.java | 11 |
| TextCommand.java | 14 |
| User.java | 18 |
| Total | 209 |

Table 6.10: Physical source lines of code for Command implementations

| CC for Command | |
|---|---|
| **File** | **CC** |
| **Haskell** | |
| Command.hs | 5 |
| Email.hs | 4 |
| Main.hs | 1 |
| Operator.hs | 27 |
| User.hs | 3 |
| Total | 40 |
| **Java** | |
| Command.java | 0 |
| ConsoleMailServer.java | 4 |
| Email.java | 5 |
| EmailEditor.java | 0 |
| MacroCommand.java | 3 |
| MailServer.java | 0 |
| Main.java | 1 |
| Operator.java | 10 |
| SendEmailCommand.java | 2 |
| SignEmailCommand.java | 2 |
| TextCommand.java | 4 |
| User.java | 4 |
| Total | 35 |

Table 6.11: Cyclomatic complexity for Command implementations

| Execution times for Command | | | | |
|---|---|---|---|---|
| **Command** | **Mean [ms]** | **Min [ms]** | **Max [ms]** | **Relative** |
| Java | $90.22 \pm 1.24$ | 87.53 | 92.32 | 13.4 |
| Haskell | $6.74 \pm 0.18$ | 6.46 | 7.57 | 1 |

Table 6.12: Execution times for Command implementations

#### Execution time for Command implementations

In Table 6.12 we list the execution time for each Command implementation.

The Java implementation is on average approximately $90.22 \div 6.74 = 13.40$ times slower than the Haskell implementation.

Accounting for the execution time overhead calculated in Section 5.3.5, we get the following results:

$$\begin{aligned} \text{Java: } 90.22 - 74.0 &= 16.22 \\ \text{Haskell: } 6.74 - 6.9 &= -0.16 \end{aligned} \tag{6.7}$$

With these adjusted numbers, the Java implementation is $16.22 \div -0.16 = -101.38$ times slower than the Haskell implementation.

#### Cyclomatic complexity density for Command implementations

Using the results from our LSLOC and CC measurements for this case, we calculate the CC density as follows:

$$\begin{aligned} CC_\rho(\text{Command, Java}) &= \frac{CC(\text{Command, Java})}{LSLOC(\text{Command, Java})} \\ &= \frac{35}{176} \\ &= 0.1989 \end{aligned} \tag{6.8}$$

$$\begin{aligned} CC_\rho(\text{Command, Haskell}) &= \frac{CC(\text{Command, Haskell})}{LSLOC(\text{Command, Haskell})} \\ &= \frac{40}{156} \\ &= 0.2564 \end{aligned} \tag{6.9}$$

The Haskell implementation is approximately $0.2564 \div 0.1989 = 1.2891$ times more logically dense than the Java implementation for the given LSLOC and CC measurements.

### 6.4.4 Comparison of Command implementations

In this section, we summarize our observations made while making the implementations for Command.

There are significant differences in the structure of the implementations this time in comparison to previous cases. This stems mostly from Haskell's way of explicitly handling side-effects in the `IO` monad, while Java allows side-effects anywhere in code. The Command pattern suits functional programming rather well, but it can be executed in a multitude of ways. The solution we implemented uses an ADT to represent a command to be executed, and uses `dispatch` (and `dispatchList`) to execute the commands. We have to manually thread the email that is being edited through each call to `dispatch`.

Using a custom monad for handling the state of the email while it is edited could be used, but requires multiple methods and techniques this thesis does not have the page count to explore.

Other than this, they feature similar abstractions in terms of how the commands are created and issued:

- Both have a general sense of a `Command`, which can be `execute`d or `dispatch`ed.
- Both build up lists of commands to execute in on a `User`.
- Both execute these commands *as* the `User`.

Summarizing the key differences:

- Java allows the programmer to mutate state wherever they please. In Haskell, updating state is more explicit through function chaining.
- As a result of the above point, it makes sense to use the `EmailEditor` interface to edit the email before it is made "immutable" in the Java implementation. In Haskell, the email is technically always immutable, so we do not necessarily need `EmailEditor`, or any equivalent, and leave the `Email` module the responsibility of editing, signing and sending the email in question.

There are a myriad of ways to implement the Command pattern in FP, because "passing behaviour as data" is the essence of FP: functions as data. Has we more time to rethink this implementation, we would instead opt for a simpler approach that is more comparable to the Java implementation in complexity and scope.

## 6.5 Composite

The Composite design pattern is concerned with aggregating and streamlining how a group of components are treated in comparison to a single component. It is mainly suitable for creating part-whole hierarchies.[4]

### 6.5.1 Composite case

We model the hardware components of a computer, and how they are assembled.

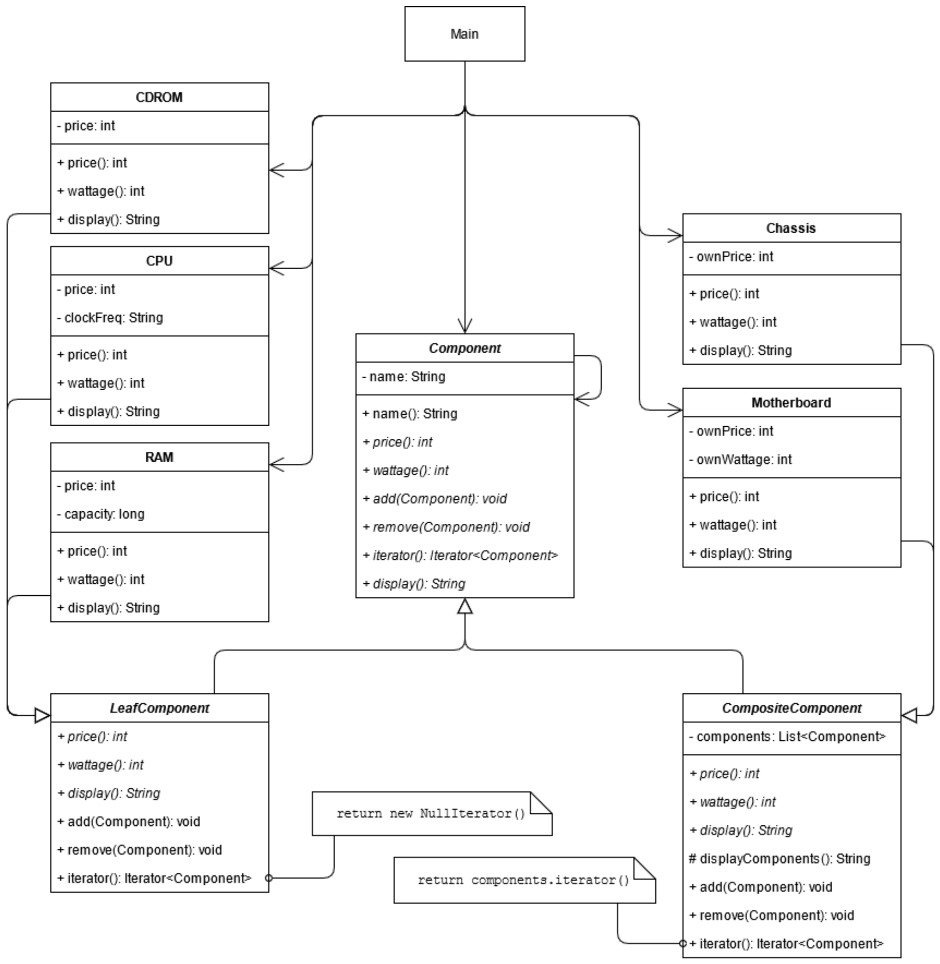


Figure 6.4: UML class diagram for the Composite case.

Implement an system for assembling a modern desktop computer. This application will be assembled and displayed by a client. The components of the computer must fit the Composite design pattern in the sense that they form a hierarchy of components, where the chassis of the computer is expected to be the root component of the system, with a CDROM and motherboard attached. The motherboard has further more components in the form of a CPU and RAM.

When displaying the computer, a hierarchical overview of the computer and its parts should print to the console.

### 6.5.2 Implementations for Composite

The code for the Java implementation is contained in the repository's "Java/Implementations/Composite" folder, while the Haskell implementation is contained in the "Haskell/Implementations/Composite" folder.

**Java** We model the components as a loose hierarchy. All concrete components are transitive extensions of the base `Component` abstract class. Leaf components, those components that cannot have children, extend `LeafComponent`. Composite components extend `CompositeComponent`. Other than that, we have concrete leaf components in the form of `CDROM`, `CPU` and `RAM`, as well as concrete composite components in the form of `Chassis` and `Motherboard`. The computer is assembled in `Main`, and subsequently displayed, utilizing the Composite pattern to display the entire tree of assembled components.

**Haskell** The Haskell implementation is very similar to the Java implementation. Instead of an abstract class, we use a recursive ADT to define a `Component`. We keep `Component` polymorphic in the type of the contained used for child components to allow for different implementations of component collections. We use `BasicInfo` to keep the information that would in Java's case be kept in super classes. Otherwise, the `displayComponent` function uses pattern matching to figure out how to display each component, and recursively calls `displayComponent` on child components to get their displayed output as well.

Neither of these implementations are safe from circular dependencies though, and would loop indefinitely if for example a component is the parent of itself:

```
c = new Chassis();
c.addComponent(c);
c.display();
```

In the case above, the program would never terminate on line 3, or would encounter a stack overflow.

### 6.5.3 Metrics for Composite implementations

In this subsection we list the metrics results for the Composite implementations, and briefly describe our observations for this particular case.

#### Source lines of code for Composite implementations

In this section we list the LSLOC and PSLOC metrics for the Composite implementations.

**Logical source lines of code for Composite implementations** In Table 6.13 we list the LSLOC for each Composite implementations' files.

The Java implementation is $266 \div 255 = 1.04$ times larger than the Haskell implementation in terms of LSLOC.

**Physical source lines of code for Composite implementations** In Table 6.14 we list the PSLOC for each Composite implementations' files.

The Java implementation is $246 \div 178 = 1.38$ times larger than the Haskell implementation in terms of PSLOC.

| LSLOC for Composite | |
|---|---|
| **File** | **LSLOC** |
| **Haskell** | |
| Component.hs | 2334 |
| Main.hs | 22 |
| Total | 255 |
| **Java** | |
| CDROM.java | 24 |
| Chassis.java | 48 |
| Component.java | 6 |
| CompositeComponent.java | 26 |
| CPU.java | 28 |
| LeafComponent.java | 5 |
| Main.java | 15 |
| Motherboard.java | 52 |
| NullIterator.java | 5 |
| RAM.java | 29 |
| Utils.java | 28 |
| Total | 266 |

Table 6.13: Logical source lines of code for Composite implementations

| PSLOC for Composite | |
|---|---|
| **File** | **PSLOC** |
| **Haskell** | |
| Component.hs | 159 |
| Main.hs | 19 |
| Total | 178 |
| **Java** | |
| CDROM.java | 21 |
| Chassis.java | 35 |
| Component.java | 16 |
| CompositeComponent.java | 28 |
| CPU.java | 24 |
| LeafComponent.java | 14 |
| Main.java | 11 |
| Motherboard.java | 37 |
| NullIterator.java | 9 |
| RAM.java | 24 |
| Utils.java | 27 |
| Total | 246 |

Table 6.14: Physical source lines of code for Composite implementations

| **CC for Composite** | |
|---|---|
| **File** | **CC** |
| **Haskell** | |
| Component.hs | 54 |
| Main.hs | 7 |
| Total | 61 |
| **Java** | |
| CDROM.java | 4 |
| Chassis.java | 6 |
| Component.java | 2 |
| CompositeComponent.java | 6 |
| CPU.java | 4 |
| LeafComponent.java | 4 |
| Main.java | 1 |
| Motherboard.java | 6 |
| NullIterator.java | 2 |
| RAM.java | 4 |
| Utils.java | 7 |
| Total | 46 |

Table 6.15: Cyclomatic complexity for Composite implementations

| **Execution times for Composite** | | | | |
|---|---|---|---|---|
| **Command** | **Mean [ms]** | **Min [ms]** | **Max [ms]** | **Relative** |
| Java | $90.46 \pm 4.2$ | 87.6 | 91.8 | 12.13 |
| Haskell | $7.46 \pm 0.15$ | 7.18 | 8.24 | 1 |

Table 6.16: Execution times for Composite implementations

### Cyclomatic complexity for Composite implementations

In Table 6.15 we list the CC for each Composite implementations' files.

The Java implementation's total CC is 46, while the Haskell implementation's is 61. The Haskell implementation CC is $61 \div 46 = 1.33$ times greater than the Java implementation CC.

### Execution time for Composite implementations

In Table 6.16 we list the execution time for each Composite implementation.

The Java implementation is on average approximately $90.46 \div 7.46 = 12.13$ times slower than the Haskell implementation.

Accounting for the execution time overhead calculated in Section 5.3.5, we get the following results:

$$\begin{aligned} \text{Java: } 90.46 - 74.0 = 16.46 \\ \text{Haskell: } 7.46 - 6.9 = 0.56 \end{aligned} \tag{6.10}$$

With these adjusted numbers, the Java implementation is $16.46 \div 0.56 = 29.39$ times slower than the Haskell implementation.

**Cyclomatic complexity density for Composite implementations**

Using the results from our LSLOC and CC measurements for this case, we calculate the CC density as follows:

$$\begin{aligned} CC_{\rho}(\text{Composite}, \text{Java}) &= \frac{CC(\text{Composite}, \text{Java})}{LSLOC(\text{Composite}, \text{Java})} \\ &= \frac{46}{266} \\ &= 0.1729 \end{aligned} \tag{6.11}$$

$$\begin{aligned} CC_{\rho}(\text{Composite}, \text{Haskell}) &= \frac{CC(\text{Composite}, \text{Haskell})}{LSLOC(\text{Composite}, \text{Haskell})} \\ &= \frac{61}{255} \\ &= 0.2392 \end{aligned} \tag{6.12}$$

The Haskell implementation is approximately $0.2392 \div 0.1729 = 1.3835$ times more logically dense than the Java implementation for the given LSLOC and CC measurements.

### 6.5.4 Comparison of Composite implementations

In this section, we summarize our observations made while making the implementations for Composite.

Apart from the difference of interface vs ADT, the implementations are remarkably similar without significantly deviating from their respective paradigm.

- Both group components under a single abstraction.
- All components can be treated similarly.
- Components delegate to child components.

There is one key difference though: in the Java implementation, the components can be treated as the concrete component that they are if need arise, but they cannot easily be treated this way in the Haskell implementation. In Haskell, all the components **are** the same type, and don't just implement the same type.

While the implementations are very similar in structure, the Haskell implementation has a significantly higher CC density than the Java implementation, and is considerably more complex accoring to the CC metric.

This case could also be implemented with a `Component` type class instead of an ADT, which would let us treat each component as separate types if necessary.

## 6.6 Decorator

The Decorator design pattern is focused on altering the behaviour of an object in a transparent, stackable manner by wrapping the object in question. To the consumer of a decorated object, the decorator will be practically invisible.

### 6.6.1 Decorator case

We model a system for displaying shapes in various ways.

> Implement a system for displaying shapes. It must be able to decorate the displayed shapes with various concrete decorators. These decorators must follow all the qualities of decorators described by the Decorator pattern: they must be nestable, must not change the visible interface of the underlying object, and must preferably only have a single responsibility.
>
> The shapes will be constructed and decorated by a client, which then sends them to a "screen" to be displayed. This screen may only require that the passed object is displayable, and should make no further assumptions on the passed objects.

### 6.6.2 Implementations for Decorator

The code for the Java implementation is contained in the repository's "Java/Implementations/Decorator" folder, while the Haskell implementation is contained in the "Haskell/Implementations/Decorator" folder.

**Java** The `Screen` class is responsible for actually displaying the shapes on the screen. In this particular case, it just prints them to the console. For a class to be displayable, it must implement the `Display` interface. Each of the concrete shapes, `Rectangle`, `Square` and `Circle`, implement the `Display` interface. In addition to these concrete shape classes, we have various decorators for augmenting the display output for each shape: `BorderDecorator`, which adds a border to the display output; `ColorDecorator`, which colors the displayed output; `LowerCaseDecorator`, which turns all displayed text lower case; and `UpperCaseDecorator`, which turns all displayed text upper case. In `Main`, we build up a list of decorated shapes, and display them using `Screen`. `Screen` makes sure colors are reset between each shape.

**Haskell** The `Screen` module is responsible for actually displaying the shapes on the screen. Mimicking the Java implementation, the shapes are printed to the console. For `Screen` to be able to display a shape it needs to be an instance of the `Display` type class. Each of the concrete shapes, `Rectangle`, `Square` and `Circle`, are instances of the `Display` type class. In addition to these concrete shapes, we have the same decorators as in the Java

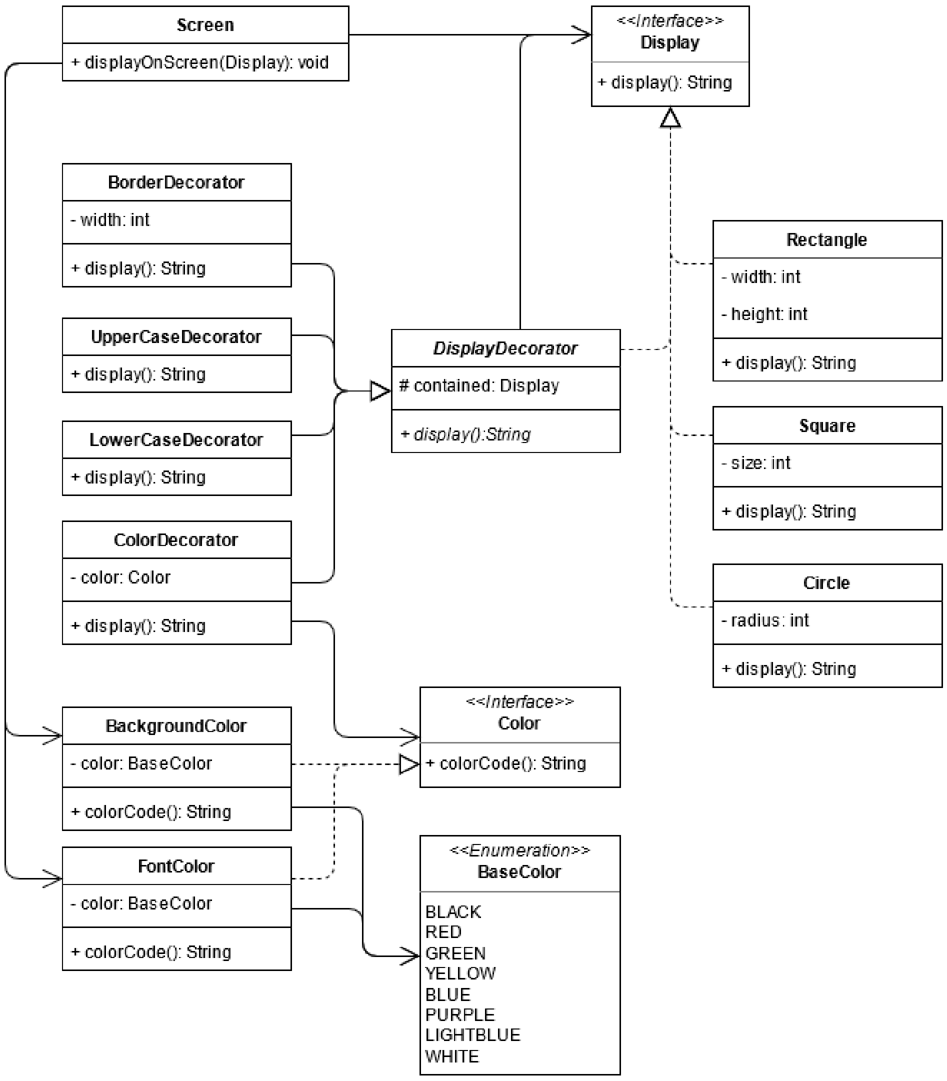


Figure 6.5: UML class diagram for the Decorator case.

implementation. These decorators effectively use the Newtype pattern, even though some of them are not `newtypes`, to alter the output of the contained `Display` instance. We use `ExistentialQuantification` to build up a list of `Display` values, and display them using `Screen`'s `displayOnScreen` function. `displayOnScreen` makes sure colors are reset between each shape.

### 6.6.3 Metrics for Decorator implementations

In this subsection we list the metrics results for the Decorator implementations, and briefly describe our observations for this particular case.

#### Source lines of code for Decorator implementations

In this section we list the LSLOC and PSLOC metrics for the Decorator implementations.

**Logical source lines of code for Decorator implementations** In Table 6.17 we list the LSLOC for each Decorator implementations' files.

The Java implementation is $246 \div 188 = 1.31$ times larger than the Haskell implementation in terms of LSLOC.

**Physical source lines of code for Decorator implementations** In Table 6.18 we list the PSLOC for each Decorator implementations' files.

The Java implementation is $248 \div 201 = 1.23$ times larger than the Haskell implementation in terms of PSLOC.

#### Cyclomatic complexity for Decorator implementations

In Table 6.19 we list the CC for each Decorator implementations' files.

The Java implementation's total CC is 56, while the Haskell implementation's is 65. The Haskell implementation CC is $65 \div 56 = 1.16$ times greater than the Java implementation CC.

#### Execution time for Decorator implementations

In Table 6.20 we list the execution time for each Decorator implementation.

The Java implementation is on average approximately $106.79 \div 7.21 = 14.82$ times slower than the Haskell implementation.

Accounting for the execution time overhead calculated in Section 5.3.5, we get the following results:

$$\begin{gathered}\text{Java: } 106.79 - 74.0 = 32.79 \\ \text{Haskell: } 7.21 - 6.9 = 0.31\end{gathered} \tag{6.13}$$

With these adjusted numbers, the Java implementation is $32.79 \div 0.31 = 105.77$ times slower than the Haskell implementation.

| **LSLOC for Decorator** | |
|---|---|
| **File** | **LSLOC** |
| **Haskell** | |
| BackgroundColor.hs | 15 |
| BaseColor.hs | 1 |
| BorderDecorator.hs | 43 |
| CharMapDecorator.hs | 4 |
| Circle.hs | 5 |
| Color.hs | 0 |
| ColorDecorator.hs | 5 |
| Display.hs | 0 |
| FontColor.hs | 15 |
| Main.hs | 34 |
| Rectangle.hs | 29 |
| Screen.hs | 12 |
| Square.hs | 25 |
| Total | 188 |
| **Java** | |
| BackgroundColor.java | 23 |
| BaseColor.java | 1 |
| BorderDecorator.java | 78 |
| Circle.java | 9 |
| Color.java | 0 |
| ColorDecorator.java | 9 |
| Display.java | 0 |
| DisplayDecorator.java | 4 |
| FontColor.java | 23 |
| LowerCaseDecorator.java | 7 |
| Main.java | 24 |
| Rectangle.java | 26 |
| Screen.java | 11 |
| Square.java | 24 |
| UpperCaseDecorator.java | 7 |
| Total | 246 |

Table 6.17: Logical source lines of code for Decorator implementations

| **PSLOC for Decorator** | |
|---|---|
| **File** | **PSLOC** |
| **Haskell** | |
| BackgroundColor.hs | 21 |
| BaseColor.hs | 12 |
| BorderDecorator.hs | 25 |
| CharMapDecorator.hs | 7 |
| Circle.hs | 12 |
| Color.hs | 5 |
| ColorDecorator.hs | 8 |
| Display.hs | 5 |
| FontColor.hs | 21 |
| Main.hs | 36 |
| Rectangle.hs | 19 |
| Screen.hs | 16 |
| Square.hs | 14 |
| Total | 201 |
| **Java** | |
| BackgroundColor.java | 23 |
| BaseColor.java | 10 |
| BorderDecorator.java | 64 |
| Circle.java | 9 |
| Color.java | 3 |
| ColorDecorator.java | 10 |
| Display.java | 3 |
| DisplayDecorator.java | 7 |
| FontColor.java | 23 |
| LowerCaseDecorator.java | 8 |
| Main.java | 31 |
| Rectangle.java | 20 |
| Screen.java | 11 |
| Square.java | 18 |
| UpperCaseDecorator.java | 8 |
| Total | 248 |

Table 6.18: Physical source lines of code for Decorator implementations

| **CC for Decorator** | |
|---|---|
| **File** | **CC** |
| **Haskell** | |
| BackgroundColor.hs | 17 |
| BorderDecorator.hs | 11 |
| CharMapDecorator.hs | 1 |
| Circle.hs | 1 |
| ColorDecorator.hs | 1 |
| FontColor.hs | 17 |
| Main.hs | 3 |
| Rectangle.hs | 7 |
| Screen.hs | 2 |
| Square.hs | 5 |
| Total | 65 |
| **Java** | |
| BackgroundColor.java | 11 |
| BaseColor.java | 0 |
| BorderDecorator.java | 15 |
| Circle.java | 2 |
| Color.java | 0 |
| ColorDecorator.java | 2 |
| Display.java | 0 |
| DisplayDecorator.java | 1 |
| FontColor.java | 11 |
| LowerCaseDecorator.java | 2 |
| Main.java | 2 |
| Rectangle.java | 3 |
| Screen.java | 2 |
| Square.java | 3 |
| UpperCaseDecorator.java | 2 |
| Total | 56 |

Table 6.19: Cyclomatic complexity for Decorator implementations

| **Execution times for Decorator** | | | | |
|---|---|---|---|---|
| **Command** | **Mean [ms]** | **Min [ms]** | **Max [ms]** | **Relative** |
| Java | $106.79 \pm 1.21$ | 103.43 | 108.18 | 14.82 |
| Haskell | $7.21 \pm 0.14$ | 6.96 | 8.12 | 1 |

Table 6.20: Execution times for Decorator implementations

**Cyclomatic complexity density for Decorator implementations**

Using the results from our LSLOC and CC measurements for this case, we calculate the CC density as follows:

$$\begin{aligned} CC_\rho(\text{Decorator}, \text{Java}) &= \frac{CC(\text{Decorator}, \text{Java})}{LSLOC(\text{Decorator}, \text{Java})} \\ &= \frac{56}{246} \\ &= 0.2276 \end{aligned} \tag{6.14}$$

$$\begin{aligned} CC_\rho(\text{Decorator}, \text{Haskell}) &= \frac{CC(\text{Decorator}, \text{Haskell})}{LSLOC(\text{Decorator}, \text{Haskell})} \\ &= \frac{65}{188} \\ &= 0.3457 \end{aligned} \tag{6.15}$$

The Haskell implementation is approximately $0.3457 \div 0.2276 = 1.5189$ times more logically dense than the Java implementation for the given LSLOC and CC measurements.

### 6.6.4 Comparison of Decorator implementations

In this section, we summarize our observations made while making the implementations for Decorator.

Much like with the Composite case, the two implementations for this case are very similar. The main differences are language-specific, with the Haskell implementation using type classes while the Java implementation uses interfaces to abstract over `Display`. Each decorator wraps another decorator or concrete shape in both implementations. However, the form of wrapping used in the Haskell implementation is often called the Newtype pattern.

Newtype pattern is very similar to how the Decorator pattern is usually applied, so the similarities are high between the two implementations.

The Haskell implementation seems to be slower than usual, with the Java implementation only being 11.15 times slower. Compared to previous cases, the Haskell implementation is about half as fast as usual when compared to the Java implementation.

## 6.7 Iterator

The Iterator design pattern is concerned with consuming the elements of a collection as they are needed. Instead of converting the entire collection to an array, or having to consume the elements of the collection in an implementation-specific way, the Iterator pattern allows us to consume a collection lazily regardless of the structure of the collection.

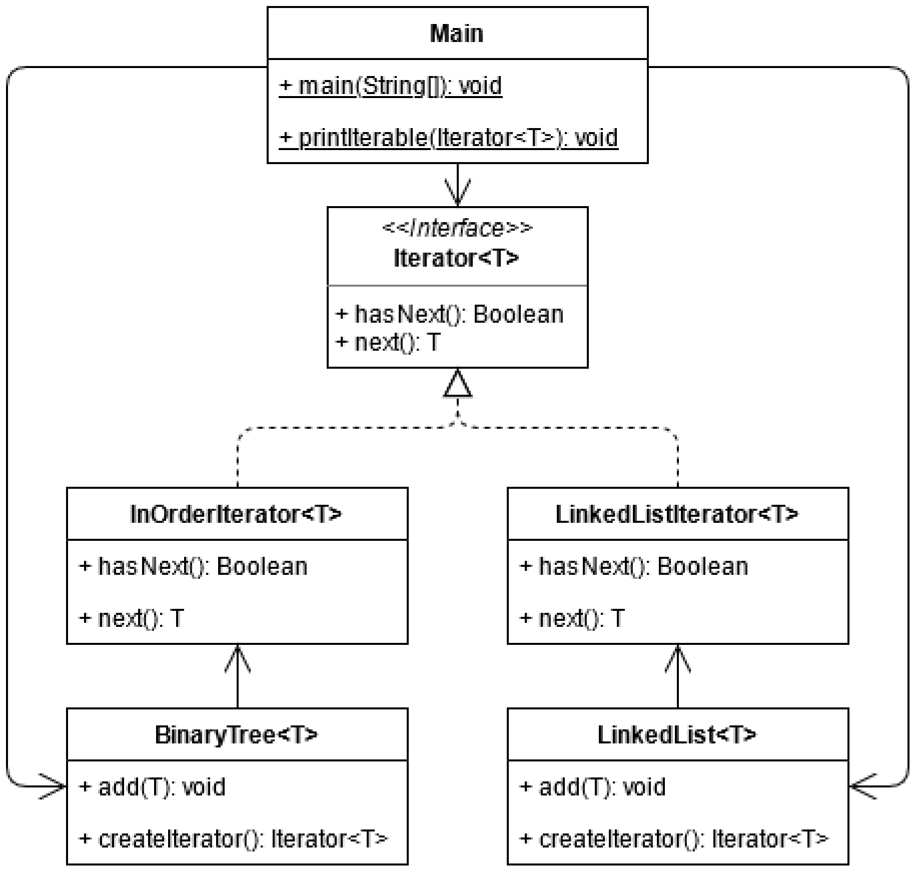


Figure 6.6: UML class diagram for the Iterator case.

### 6.7.1 Iterator case

We implement iterators for a linked list and a binary tree.

> Implement a linked list and a binary tree. These implementations must also be able to provide an iterator to consume the collection in question as we need the elements.

### 6.7.2 Implementations for Iterator

The code for the Java implementation is contained in the repository's "Java/Implementations/Iterator" folder, while the Haskell implementation is contained in the "Haskell/Implementations/Iterator" folder.

**Java** We implemented a binary tree and a linked list, and provided iterators for each of these. The iterator for the binary tree traverses the tree in order from lowest to highest. The iterator for the linked list traverses the list from the beginning to the end of the list.

**Haskell** We implemented a binary tree and a list, and made these collections instances of the `Foldable` type class. The default list in Haskell is a linked

| **LSLOC for Iterator** | |
|---|---|
| **File** | **LSLOC** |
| **Haskell** | |
| Add.hs | 0 |
| LinkedList.hs | 13 |
| Main.hs | 38 |
| Tree.hs | 19 |
| Total | 70 |
| **Java** | |
| BinaryTree.java | 68 |
| Coll.java | 7 |
| LinkedList.java | 34 |
| Main.java | 34 |
| Total | 143 |

Table 6.21: Logical source lines of code for Iterator implementations

list, but to keep the measurements similar we implemented a linked list of our own.

### 6.7.3 Metrics for Iterator implementations

#### Source lines of code for Iterator implementations

In this section we list the LSLOC and PSLOC metrics for the Iterator implementations.

**Logical source lines of code for Iterator implementations** In Table 6.21 we list the LSLOC for each Iterator implementations' files.

The Java implementation is $143 \div 70 = 2.04$ times larger than the Haskell implementation in terms of LSLOC.

**Physical source lines of code for Iterator implementations** In Table 6.22 we list the PSLOC for each Iterator implementations' files.

The Java implementation is $173 \div 81 = 2.14$ times larger than the Haskell implementation in terms of PSLOC.

#### Cyclomatic complexity for Iterator implementations

In Table 6.23 we list the CC for each Iterator implementations' files.

The Haskell implementation's total CC is 18, while the Java implementation's is 37. The Java implementation CC is $37 \div 18 = 2.06$ times greater than the Haskell implementation CC.

#### Execution time for Iterator implementations

In Table 6.24 we list the execution time for each Iterator implementation.

The Java implementation is on average approximately $81.79 \div 7.01 = 11.67$ times slower than the Haskell implementation.

| PSLOC for Iterator | |
|---|---|
| **File** | **PSLOC** |
| **Haskell** | |
| Add.hs | 6 |
| LinkedList.hs | 21 |
| Main.hs | 31 |
| Tree.hs | 23 |
| Total | 81 |
| **Java** | |
| BinaryTree.java | 83 |
| Coll.java | 13 |
| LinkedList.java | 44 |
| Main.java | 33 |
| Total | 173 |

Table 6.22: Physical source lines of code for Iterator implementations

| CC for Iterator | |
|---|---|
| **File** | **CC** |
| **Haskell** | |
| LinkedList.hs | 6 |
| Main.hs | 6 |
| Tree.hs | 6 |
| Total | 18 |
| **Java** | |
| BinaryTree.java | 21 |
| Coll.java | 3 |
| LinkedList.java | 9 |
| Main.java | 4 |
| Total | 37 |

Table 6.23: Cyclomatic complexity for Iterator implementations

| Execution times for Iterator | | | | |
|---|---|---|---|---|
| **Command** | **Mean [ms]** | **Min [ms]** | **Max [ms]** | **Relative** |
| Java | $81.79 \pm 7.33$ | 74.7 | 91.33 | 11.67 |
| Haskell | $7.01 \pm 0.19$ | 6.74 | 8.05 | 1 |

Table 6.24: Execution times for Iterator implementations

Accounting for the execution time overhead calculated in Section 5.3.5, we get the following results:

$$\begin{aligned} \text{Java: } 81.79 - 74.0 &= 7.79 \\ \text{Haskell: } 7.01 - 6.9 &= 0.11 \end{aligned} \tag{6.16}$$

With these adjusted numbers, the Java implementation is $7.79 \div 0.11 = 70.82$ times slower than the Haskell implementation.

**Cyclomatic complexity density for Iterator implementations**

Using the results from our LSLOC and CC measurements for this case, we calculate the CC density as follows:

$$\begin{aligned} CC_\rho(\text{Iterator}, \text{Java}) &= \frac{CC(\text{Iterator}, \text{Java})}{LSLOC(\text{Iterator}, \text{Java})} \\ &= \frac{37}{143} \\ &= 0.2587 \end{aligned} \tag{6.17}$$

$$\begin{aligned} CC_\rho(\text{Iterator}, \text{Haskell}) &= \frac{CC(\text{Iterator}, \text{Haskell})}{LSLOC(\text{Iterator}, \text{Haskell})} \\ &= \frac{18}{70} \\ &= 0.2571 \end{aligned} \tag{6.18}$$

The Java implementation is approximately $0.2587 \div 0.2571 = 1.0062$ times more logically dense than the Haskell implementation for the given LSLOC and CC measurements.

### 6.7.4 Comparison of Iterator implementations

In this section, we summarize our observations made while making the implementations for Iterator.

Given that Haskell has language support for ADTs the `LinkedList` and `Tree` implementations in the Haskell implementation are considerably simpler than the Java implementation's `LinkedList` and `BinaryTree` implementations in LSLOC, PSLOC and CC.

While the Haskell implementation is considerably "simpler" than the Java implementation in terms of CC, PSLOC and LSLOC, it sustains a higher CC density than the Java implementation, if even by only 6.13%. The Haskell implementation's execution time is also considerably worse when comparing it to previous cases, while the Java implementation seems to be rather fast. This might be due to lazy data structures in the Haskell implementation.

The Iterator pattern can be implemented in two main ways in FP: `Foldable`, like we did here, and conversion to `List`s. In Haskell, the default list is a lazy linked list. Each element of the list is evaluated as they are needed, and will otherwise remain unevaluated if consumption of the list halts before reaching every element. As such, Haskell lists are themselves iterators according to the Iterator pattern's definition. Converting a collection to a list would then provide a generalized, lazy way to consume the collection regardless of its implementation.

## 6.8 Prototype

The Prototype pattern is focused on keeping complicated and inefficient object initialization to a minimum by copying existing "prototype" objects instead of initializing new ones from scratch.

### 6.8.1 Prototype case

We model a system for creating space ships in a game.

> Implement a system for creating modular space ships. Each part of a space ship must implement a method for deep cloning, whereby any objects and attributes within the object are cloned as well. The space ship will be the client using the Prototype pattern to clone the parts as they are added to the ship. The ship and the parts must also implement a method to display themselves on the screen.

### 6.8.2 Implementations for Prototype

The code for the Java implementation is contained in the repository's "Java/Implementations/Prototype" folder, while the Haskell implementation is contained in the "Haskell/Implementations/Prototype" folder.

**Java** The specification by Gamma et al. for the Prototype pattern is very simple: have objects implement an interface with a method for cloning the object. We have a `Part` abstract class which implements a `clone` method for use by sub classes, along with a `display` method for displaying the part on the screen. The `SpaceShip` class holds a list of `Part` objects, and uses these parts when displaying itself on the screen. We have four concrete parts: `GunPart`, `StructurePart`, `ThrusterPart` and `WindowPart`. Each of these concrete parts extend the `Part` abstract class, implementing a `display` method of their own. In `Main` we initialize a weak thruster, a strong thruster, a weak gun, a strong gun, a normal structure, a strong structure, and a window. We then assemble a small and a large ship with various combinations of these parts, using `SpaceShip`'s `addPart` method, which makes sure to `clone` the passed part before adding it to its list.

**Haskell** In Haskell, the need to copy objects is not present to the same degree as in Java and other OOP programming languages. Because values are immutable by default, we can reuse values as we like without fear of

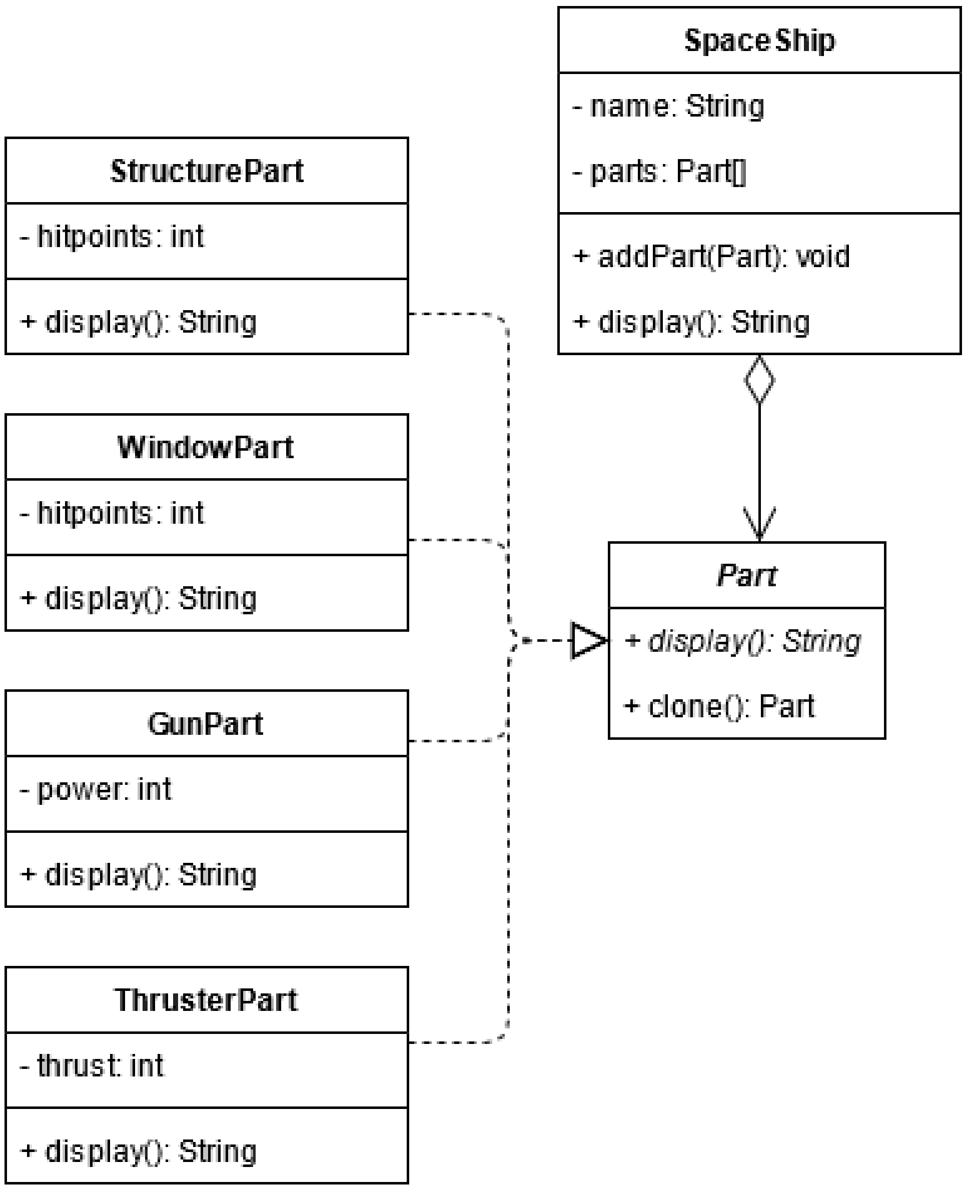


Figure 6.7: UML class diagram for the Prototype case.

| **LSLOC for Prototype** | |
|---|---|
| **File** | **LSLOC** |
| **Haskell** | |
| Display.hs | 0 |
| Gun.hs | 5 |
| Main.hs | 59 |
| SpaceShip.hs | 26 |
| Structure.hs | 5 |
| Thruster.hs | 5 |
| Window.hs | 5 |
| Total | 105 |
| **Java** | |
| GunPart.java | 11 |
| Main.java | 44 |
| Part.java | 0 |
| SpaceShip.java | 25 |
| StructurePart.java | 11 |
| ThrusterPart.java | 11 |
| WindowPart.java | 11 |
| Total | 113 |

Table 6.25: Logical source lines of code for Prototype implementations

mutating state in other places of the program. The Haskell implementation is thus similar to the Java implementation, with the exception of the `clone` method, which we omit entirely in the Haskell implementation. We have a `Display` type class with one operation, `display`, for displaying values on the screen. `SpaceShip` uses the existential data type `Part` for its list of parts. `Part` only requires its value to be an instance of `Display`, which allows us to have a list of parts of different types. Each of the concrete parts are implemented as separate data types which are instances of the `Display` type class. In `Main`, we initialize the same part variations as in the Java implementation, and build the same small and large ships.

### 6.8.3 Metrics for Prototype implementations

In this subsection we list the metrics results for the Prototype implementations, and briefly describe our observations for this particular case.

#### Source lines of code for Prototype implementations

In this section we list the LSLOC and PSLOC metrics for the Prototype implementations.

**Logical source lines of code for Prototype implementations** In Table 6.25 we list the LSLOC for each Prototype implementations' files.

The Java implementation is $113 \div 105 = 1.08$ times larger than the Haskell implementation in terms of LSLOC.

| **PSLOC for Prototype** | |
|---|---|
| **File** | **PSLOC** |
| **Haskell** | |
| Display.hs | 5 |
| Gun.hs | 9 |
| Main.hs | 39 |
| SpaceShip.hs | 27 |
| Structure.hs | 9 |
| Thruster.hs | 9 |
| Window.hs | 9 |
| Total | 107 |
| **Java** | |
| GunPart.java | 12 |
| Main.java | 38 |
| Part.java | 4 |
| SpaceShip.java | 26 |
| StructurePart.java | 12 |
| ThrusterPart.java | 12 |
| WindowPart.java | 12 |
| Total | 116 |

Table 6.26: Physical source lines of code for Prototype implementations

**Physical source lines of code for Prototype implementations** In Table 6.26 we list the PSLOC for each Prototype implementations' files.

The Java implementation is $116 \div 107 = 1.08$ times larger than the Haskell implementation in terms of PSLOC.

### Cyclomatic complexity for Prototype implementations

In Table 6.27 we list the CC for each Prototype implementations' files.

The Java implementation's total CC is 19, while the Haskell implementation's is 19. The Haskell implementation CC is $19 \div 19 = 1.00$ times greater than the Java implementation CC.

### Execution time for Prototype implementations

In Table 6.28 we list the execution time for each Prototype implementation.

The Java implementation is on average approximately $90.70 \div 6.91 = 13.13$ times slower than the Haskell implementation.

Accounting for the execution time overhead calculated in Section 5.3.5, we get the following results:

$$\begin{aligned} \text{Java: } 90.70 - 74.0 &= 16.70 \\ \text{Haskell: } 6.91 - 6.9 &= 0.01 \end{aligned} \tag{6.19}$$

With these adjusted numbers, the Java implementation is $16.70 \div 0.01 = 1670.00$ times slower than the Haskell implementation.

| **CC for Prototype** | |
|---|---|
| **File** | **CC** |
| **Haskell** | |
| Gun.hs | 1 |
| Main.hs | 11 |
| SpaceShip.hs | 4 |
| Structure.hs | 1 |
| Thruster.hs | 1 |
| Window.hs | 1 |
| Total | 19 |
| **Java** | |
| GunPart.java | 3 |
| Main.java | 3 |
| Part.java | 0 |
| SpaceShip.java | 4 |
| StructurePart.java | 3 |
| ThrusterPart.java | 3 |
| WindowPart.java | 3 |
| Total | 19 |

Table 6.27: Cyclomatic complexity for Prototype implementations

| **Execution times for Prototype** | | | | |
|---|---|---|---|---|
| **Command** | **Mean [ms]** | **Min [ms]** | **Max [ms]** | **Relative** |
| Java | $90.7 \pm 0.86$ | 89.19 | 92.27 | 13.15 |
| Haskell | $6.91 \pm 0.15$ | 6.64 | 7.69 | 1 |

Table 6.28: Execution times for Prototype implementations

**Cyclomatic complexity density for Prototype implementations**

Using the results from our LSLOC and CC measurements for this case, we calculate the CC density as follows:

$$\begin{aligned} CC_\rho(\text{Prototype}, \text{Java}) &= \frac{CC(\text{Prototype}, \text{Java})}{LSLOC(\text{Prototype}, \text{Java})} \\ &= \frac{19}{113} \\ &= 0.1681 \end{aligned} \tag{6.20}$$

$$\begin{aligned} CC_\rho(\text{Prototype}, \text{Haskell}) &= \frac{CC(\text{Prototype}, \text{Haskell})}{LSLOC(\text{Prototype}, \text{Haskell})} \\ &= \frac{19}{105} \\ &= 0.1810 \end{aligned} \tag{6.21}$$

The Haskell implementation is approximately $0.1810 \div 0.1681 = 1.0767$ times more logically dense than the Java implementation for the given LSLOC and CC measurements.

### 6.8.4 Comparison of Prototype implementations

In this section, we summarize our observations made while making the implementations for Prototype.

Haskell's immutable values are a language feature, and we require no pattern or technique to "implement" the Prototype pattern in Haskell. We have not implemented any sort of pattern in the Haskell implementation whatsoever.

The Java implementation cannot use `Object`'s `clone` method because it merely makes a shallow copy of the object. As such, we define our own `clone` method in `Part`, which also forgoes the checked exception `CloneNotSupportedException` thrown by `Object`'s `clone`.

In terms of the actual pattern, there are practically no similarities between the two implementations because the Haskell implementation does nothing in particular to achieve the desired result. Modifying a value or record in Haskell always automatically copies it, so we effectively get the intention of the Prototype pattern for free.

We can thus safely state that the Prototype pattern barely exists in Haskell, if at all. This statement can be extended to FP in general, but there are other FP languages that more readily support mutable values and objects.

## 6.9 Strategy

The Strategy pattern abstracts over a family of algorithms, providing polymorphism in terms of the concrete algorithm used by the client.

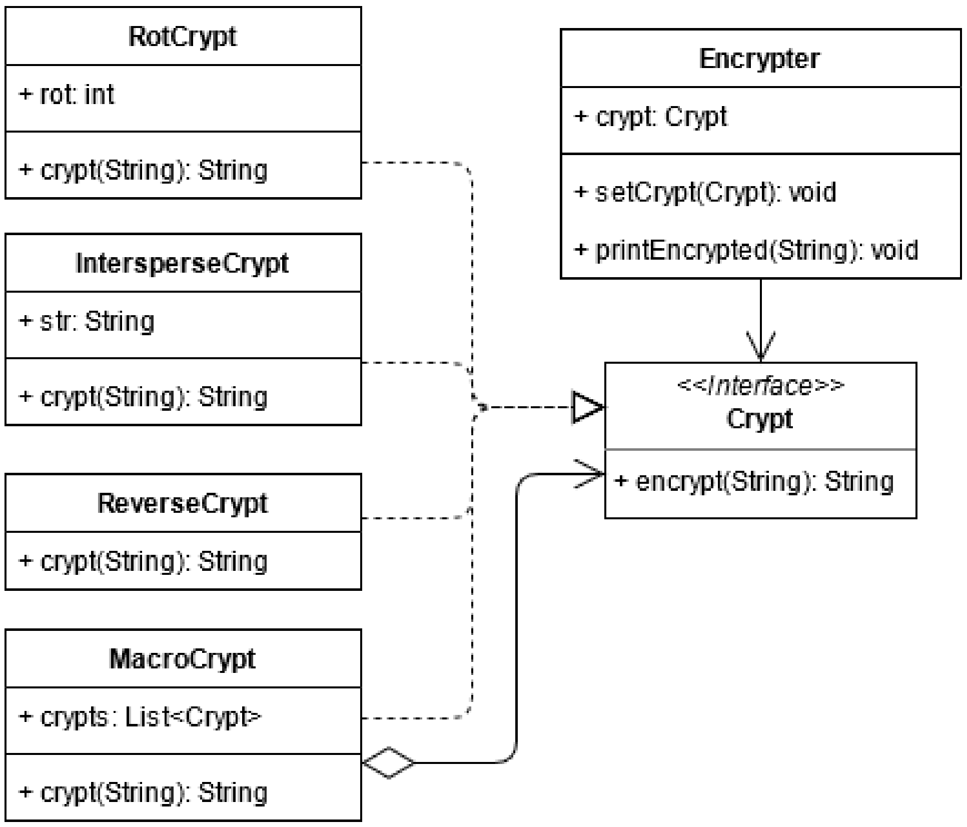


Figure 6.8: UML class diagram for the Strategy case.

### 6.9.1 Strategy case

We model a system for encrypting text.

> Implement a system for encrypting strings of text in various ways. Each way of encrypting text should be a self-contained class or method. These encrypting algorithms must be used by an "encrypter", which will take an encryption algorithm, an input string, and print the encrypted result to the console.

### 6.9.2 Implementations for Strategy

The code for the Java implementation is contained in the repository's "Java/Implementations/Strategy" folder, while the Haskell implementation is contained in the "Haskell/Implementations/Strategy" folder.

**Java** We implemented a `Crypt` interface to generalize the act of encrypting a piece of string. We have four concrete implementations for `Crypt`: `IntersperseCrypt`, which intersperses a piece of text between each character of the input string; `MacroCrypt`, which passes the input string through each `Crypt` in its list of `Crypt`s; `ReverseCrypt`, which reverses the input string; and `RotCrypt`, which shifts each character in the input string by a number

| **LSLOC for Strategy** | |
|---|---|
| **File** | **LSLOC** |
| **Haskell** | |
| Crypts.hs | 37 |
| Encrypter.hs | 6 |
| Main.hs | 19 |
| Total | 62 |
| **Java** | |
| Crypt.java | 0 |
| Encrypter.java | 12 |
| IntersperseCrypt.java | 14 |
| MacroCrypt.java | 12 |
| Main.java | 18 |
| ReverseCrypt.java | 6 |
| RotCrypt.java | 22 |
| Total | 84 |

Table 6.29: Logical source lines of code for Strategy implementations

of places. `Encrypter` takes a `Crypt`, and has a method `printEncrypted` to print a string to the terminal after encrypting it with the passed `Crypt`.

**Haskell** We implemented the four concrete `Crypts` from the Java implementation as normal functions, and the `printEncrypted` method from `Encrypter`, having it take a string and a function of type `String -> String`. Other than that, `Main` resembles the Java implementation's `Main` when it comes to initializing some of the encryption functions.

### 6.9.3 Metrics for Strategy implementations

In this subsection we list the metrics results for the Strategy implementations, and briefly describe our observations for this particular case.

#### Source lines of code for Strategy implementations

In this section we list the LSLOC and PSLOC metrics for the Strategy implementations.

**Logical source lines of code for Strategy implementations** In Table 6.29 we list the LSLOC for each Strategy implementations' files.

The Java implementation is $84 \div 62 = 1.35$ times larger than the Haskell implementation in terms of LSLOC.

**Physical source lines of code for Strategy implementations** In Table 6.30 we list the PSLOC for each Strategy implementations' files.

The Java implementation is $92 \div 49 = 1.88$ times larger than the Haskell implementation in terms of PSLOC.

| PSLOC for Strategy | |
|---|---|
| **File** | **PSLOC** |
| **Haskell** | |
| Crypts.hs | 24 |
| Encrypter.hs | 5 |
| Main.hs | 20 |
| Total | 49 |
| **Java** | |
| Crypt.java | 3 |
| Encrypter.java | 12 |
| IntersperseCrypt.java | 15 |
| MacroCrypt.java | 15 |
| Main.java | 16 |
| ReverseCrypt.java | 6 |
| RotCrypt.java | 25 |
| Total | 92 |

Table 6.30: Physical source lines of code for Strategy implementations

### Cyclomatic complexity for Strategy implementations

In Table 6.31 we list the CC for each Strategy implementations' files.

The Haskell implementation's total CC is 16, while the Java implementation's is 20. The Java implementation CC is $20 \div 16 = 1.25$ times greater than the Haskell implementation CC.

### Execution time for Strategy implementations

In Table 6.32 we list the execution time for each Strategy implementation.

The Java implementation is on average approximately $104.58 \div 6.97 = 15.00$ times slower than the Haskell implementation.

Accounting for the execution time overhead calculated in Section 5.3.5, we get the following results:

$$\begin{aligned}\text{Java: } 104.58 - 74.0 &= 30.58 \\ \text{Haskell: } 6.97 - 6.9 &= 0.07\end{aligned} \tag{6.22}$$

With these adjusted numbers, the Java implementation is $30.58 \div 0.07 = 436.86$ times slower than the Haskell implementation.

### Cyclomatic complexity density for Strategy implementations

Using the results from our LSLOC and CC measurements for this case, we calculate the CC density as follows:

| **CC for Strategy** | |
|---|---|
| **File** | **CC** |
| **Haskell** | |
| Crypts.hs | 10 |
| Encrypter.hs | 1 |
| Main.hs | 5 |
| Total | 16 |
| **Java** | |
| Crypt.java | 0 |
| Encrypter.java | 3 |
| InterspersseCrypt.java | 3 |
| MacroCrypt.java | 4 |
| Main.java | 1 |
| ReverseCrypt.java | 1 |
| RotCrypt.java | 8 |
| Total | 20 |

Table 6.31: Cyclomatic complexity for Strategy implementations

| **Execution times for Strategy** | | | | |
|---|---|---|---|---|
| **Command** | **Mean [ms]** | **Min [ms]** | **Max [ms]** | **Relative** |
| Java | $104.58 \pm 12.77$ | 89.84 | 121.86 | 15 |
| Haskell | $6.97 \pm 0.14$ | 6.74 | 7.68 | 1 |

Table 6.32: Execution times for Strategy implementations

$$\begin{aligned} CC_\rho(\text{Strategy}, \text{Java}) &= \frac{CC(\text{Strategy}, \text{Java})}{LSLOC(\text{Strategy}, \text{Java})} \\ &= \frac{20}{84} \\ &= 0.2381 \end{aligned} \tag{6.23}$$

$$\begin{aligned} CC_\rho(\text{Strategy}, \text{Haskell}) &= \frac{CC(\text{Strategy}, \text{Haskell})}{LSLOC(\text{Strategy}, \text{Haskell})} \\ &= \frac{16}{62} \\ &= 0.2581 \end{aligned} \tag{6.24}$$

The Haskell implementation is approximately $0.2581 \div 0.2381 = 1.0840$ times more logically dense than the Java implementation for the given LSLOC and CC measurements.

### 6.9.4 Comparison of Strategy implementations

In this section, we summarize our observations made while making the implementations for Strategy.

In Java, we implemented the Strategy pattern using a very simple interface with one method, while in Haskell we just specified that `printEncrypted` required a function of type `String -> String`. This simplifies the implementation because we merely pass functions to `printEncrypted`, and require no special data types whatsoever. The encryption algorithms are just functions, some partially applied before being passed to `printEncrypted`.

Apart from their algorithm implementations and the `printEncrypted` functions, the two implementations barely resemble each other.

The Strategy pattern is one of the patterns that most resembles FP. Because of this, the Strategy pattern is not exactly a "pattern" in FP, but more a core philosophy of the paradigm: HOFs, functions that take other functions as arguments. The Haskell implementation is overall considerably smaller in both SLOC measurements, less complex in CC, and only marginally more complex in terms of CC density.

## 6.10 Summary of cases

In this chapter we have presented the various cases designed for this study. We have described their requirements, how they were implemented, and the various measurements made for each case. We have also described various qualitative observations made for each case's implementations.

**Part V**

# Discussion and conclusion

# Chapter 7

# Analysis

## 7.1 Introduction

In this chapter we analyse the data collected from the cases as a whole. The focus is to analyse the results in relation to other cases. We present our measurements for the various metrics defined in Section 5.3.

We have used the metrics defined in Section 5.3 when collecting data on the various implementations created for this study.

## 7.2 Logical source lines of code summary

Table 7.1 collates the LSLOC for each implementation. The Java implementations have on average higher LSLOC than the Haskell implementations.

Especially for the Iterator case, the Java implementation has more than double the Haskell implementation's LSLOC. The most similar case is the Composite pattern, with the Java implementation only being 1.04 times larger in LSLOC than the Haskell implementation.

On average, the Haskell implementations are 76% the size of the Java implementations.

| **LSLOC** | | | |
|---|---|---|---|
| **Pattern** | **Haskell** | **Java** | **Haskell / Java** |
| **Abstract Factory** | 58 | 88 | 0.66 |
| **Adapter** | 100 | 165 | 0.61 |
| **Command** | 156 | 176 | 0.89 |
| **Composite** | 255 | 266 | 0.96 |
| **Decorator** | 188 | 246 | 0.76 |
| **Iterator** | 70 | 143 | 0.49 |
| **Prototype** | 105 | 113 | 0.93 |
| **Strategy** | 62 | 84 | 0.74 |
| **Average** | 124 | 160.13 | 0.76 |

Figure 7.1: LSLOC for each implementation

| PSLOC | | | |
|---|---|---|---|
| **Pattern** | **Haskell** | **Java** | **Haskell / Java** |
| **Abstract Factory** | 114 | 132 | 0.86 |
| **Adapter** | 137 | 185 | 0.74 |
| **Command** | 147 | 209 | 0.70 |
| **Composite** | 178 | 246 | 0.72 |
| **Decorator** | 201 | 248 | 0.81 |
| **Iterator** | 81 | 173 | 0.47 |
| **Prototype** | 107 | 116 | 0.92 |
| **Strategy** | 49 | 92 | 0.53 |
| **Average** | 127 | 175.13 | 0.72 |

Figure 7.2: PSLOC for each implementation

| CC | | | |
|---|---|---|---|
| **Pattern** | **Haskell** | **Java** | **Haskell / Java** |
| **Abstract Factory** | 14 | 24 | 0.58 |
| **Adapter** | 31 | 32 | 0.97 |
| **Command** | 40 | 35 | 1.14 |
| **Composite** | 61 | 46 | 1.33 |
| **Decorator** | 65 | 56 | 1.16 |
| **Iterator** | 18 | 37 | 0.49 |
| **Prototype** | 19 | 19 | 1.00 |
| **Strategy** | 16 | 20 | 0.80 |
| **Average** | 33 | 33.63 | 0.93 |

Figure 7.3: CC for each implementation

## 7.3 Physical source lines of code summary

Table 7.2 collates the PSLOC for each implementation. The Java implementations have on average higher PSLOC than the Haskell implementations. The largest difference is in the Iterator case, while the smallest difference is in the Prototype case. The Java implementations are on average 1.38 times larger in terms of PSLOC than the Haskell implementations.

## 7.4 Cyclomatic complexity metric

Table 7.3 collates the CC for each implementation. The Haskell implementations have on average higher CC than the Java implementations. The cases where this is not the case. The implementation with the lowest CC for Haskell is Abstract Factory, and for Java it is Prototype. Strategy is the simplest case overall, with an average CC between the two languages at 18. Decorator was the most complex case for both, with an average CC of 53.5.

| **Execution time [ms]** | | | |
|---|---|---|---|
| **Pattern** | **Haskell** | **Java** | **Haskell / Java** |
| **Abstract Factory** | 7.38 | 90.09 | 0.0819 |
| **Adapter** | 7.30 | 99.85 | 0.0731 |
| **Command** | 6.74 | 90.22 | 0.0747 |
| **Composite** | 7.46 | 90.46 | 0.0825 |
| **Decorator** | 7.21 | 106.79 | 0.0675 |
| **Iterator** | 7.01 | 81.79 | 0.0857 |
| **Prototype** | 6.91 | 90.70 | 0.0762 |
| **Strategy** | 6.97 | 104.58 | 0.0666 |
| **Average** | 7.12 | 94.3100 | 0.0760 |

Figure 7.4: Execution time [ms] for each implementation

| **Normalized time [ms]** | | | |
|---|---|---|---|
| **Pattern** | **Haskell** | **Java** | **Haskell / Java** |
| **Abstract Factory** | 0.48 | 16.09 | 0.0298 |
| **Adapter** | 0.40 | 25.85 | 0.0155 |
| **Command** | -0.16 | 16.22 | -0.0099 |
| **Composite** | 0.56 | 16.46 | 0.0340 |
| **Decorator** | 0.31 | 32.79 | 0.0095 |
| **Iterator** | 0.11 | 7.79 | 0.0141 |
| **Prototype** | 0.01 | 16.70 | 0.0006 |
| **Strategy** | 0.07 | 30.58 | 0.0023 |
| **Average** | 0.22 | 20.3100 | 0.0120 |

Figure 7.5: Normalized time [ms] for each implementation

## 7.5 Execution time summary

Table 7.4 collates the execution times for each implementation, while Table 7.5 collates the execution times in relation to the measured minimum time from Section 5.3.5.

The Java implementations take on average an order of magnitude more time to execute to completion in comparison to the Haskell implementations. The Iterator case is the closest comparison, where the Java implementations takes *only* 9.06 times more time than the Haskell implementation. The biggest difference is in the Strategy case, where the Java implementation takes 11.39 times more time than the Haskell implementation.

The normalized execution times in relation to the measured minimum are considered invalid data. The Command implementation was measured to take 0.16 ms less time to complete than the reference empty program. As such, we cannot in good faith draw any sensible conclusions from this data.

| CC density | | | |
|---|---|---|---|
| **Pattern** | **Haskell** | **Java** | **Haskell / Java** |
| **Abstract Factory** | 0.2414 | 0.2727 | 0.8852 |
| **Adapter** | 0.3100 | 0.1939 | 1.5988 |
| **Command** | 0.2564 | 0.1989 | 1.2891 |
| **Composite** | 0.2392 | 0.1729 | 1.3835 |
| **Decorator** | 0.3457 | 0.2276 | 1.5189 |
| **Iterator** | 0.2571 | 0.2587 | 0.9938 |
| **Prototype** | 0.1810 | 0.1681 | 1.0767 |
| **Strategy** | 0.2581 | 0.2381 | 1.0840 |
| **Average** | 0.2611 | 0.2164 | 1.2288 |

Figure 7.6: CC density for each implementation

## 7.6 Cyclomatic complexity density summary

Table 7.6 collates the CC density for each implementation. The Haskell implementations have on average 1.21 times the CC density of the Java implementations.

## 7.7 Summary of analysis

In this chapter we have presented a general overview of our findings from Part IV, and noted the extreme points for each metric, as well as the average.

# Chapter 8

# Discussion

## 8.1 Introduction to discussion

In this chapter we discuss our findings from Part IV and Chapter 7.

## 8.2 Case evaluation

### 8.2.1 Introduction

Some cases are simple in Java, while others are simple in Haskell. There is a general structure to the patterns that were simple and complicated in each language. None of the patterns were notably difficult to implement in either language, but some patterns were exceedingly trivial. This level of ease in implementing the patterns in each language could stem from a variety of qualities about the case, the languages and our programming capabilities.

### 8.2.2 Case differences

We present the observed differences in the case implementations.

**Abstract Factory** Simple in both languages. Using polymorphism through interfaces and type classes, both implementations were simple.

**Adapter** A bit of a contrived case objective. Neither language had any issues in terms of implementing this pattern, and they both resemble each other to a large degree. If we briefly consider interfaces and type classes to be identical constructs, the resemblance is uncanny.

**Command** Very difficult case to implement in Haskell in a form that was comparable to the Java implementation. Haskell's handling of side-effects complicated matters in the sense that we were unsure exactly how to proceed with executing the commands once they had been initialized. The solution we settled on we found to be satisfyingly simple and representative uses regular functions instead of any type classes or Newtype-trickery. Implementing the case in Java, on the other hand, proved no issue.

**Composite** Neither the Java nor the Haskell implementations proved difficult to implement for this case. They resemble each other quite a lot again, but the Haskell implementation does not have as high level of polymorphism as the Java case does, as the Haskell case requires that children of the composite components are of the same type, while in the Java implementation the children are only required to implement an interface.

**Decorator** Very simple for both implementations, but ended up requiring a lot of code for both implementations. The implementations are somewhat weighed down by the code used to display the elements in the terminal, and thus have a somewhat high "signal to noise ratio" if we consider anything not directly related to the design pattern to be noise.

**Iterator** Simple for Haskell, complex for Java. Binary trees with ADTs are very simple to implement in Haskell, while in Java it is less so. Java requires a lot of looping to find the right nodes, forcing us to consider the state of the methods before, during and after looping, and accurately assign the right values to the right places depending on where we are in the tree. `Foldable` applies the provided function on each value of the collection as they are iterated through, while the Iterator pattern requires the iterator to relinquish control in between calls to its `next` method, meaning we have to keep state between each of these calls.

**Prototype** Trivial to implement in Haskell, easy to implement in Java. This pattern does not exist in Haskell, and barely exists in general in FP. Since FP focuses on immutable values, most FP languages by default copy values when they are modified, and Haskell is not exception. In Java, on the other hand, we implement our own copying method to achieve deep copying.

**Strategy** Trivial to implement in Haskell, easy to implement in Java. This is one of the simpler patterns, and it shows. The case we decided upon brought some noise in the form of `RotCrypt`. We initially didn't realize the complexity of performing the Caesar cipher, at least in comparison to the rest of the implementations.

### 8.2.3 Case bias

While we have observed some strong correlations in our results, we should not trust correlations blindly in such a study as this. The cases presented were selected and iterated through qualitative opinions and criteria, with a human bias guiding the final implementations. The measurements and observations made represent choices made based upon qualitative criteria as well.

While we have aimed to minimize this bias and subjectiveness in the design procedure of these cases, we cannot deny that there might still exist an element of bias in the cases. We aimed to select case requirements that resemble the examples given in [4], we aimed to select case implementations that represent the programming language they were implemented in, and we aimed to measure and observe the qualities of these implementations in a balanced manner.

As such, we think the observations made in this thesis is of value. The correlations observed in our results would be a very improbable coincidence if not for some underlying phenomena or quality of the languages and paradigms. However, it should be noted that to dismiss the presence of a bias would be unwise.

## 8.3 Applicability

In this section we discuss the applicability of the design patterns to each language and their respective programming paradigms.

All the design patterns suit Java quite well, but only a few are worth considering when programming in Haskell. We describe these details for each pattern below.

### 8.3.1 Applicability of Abstract Factory

The Abstract Factory pattern suits Java exceedingly well due to Java's language support for interfaces and abstract classes. We are able to easily implement the pattern, with little overhead, and the pattern brings tangible value in solving the problems it sets out to solve.

On the other hand, the pattern is not particularly helpful in Haskell. While data creation is definitely a concern in Haskell, data creation in the way proposed by the Abstract Factory pattern is less flexible in Haskell and FP. In Haskell, it is more common to focus on data manipulation, since manipulating existing data automatically copies it. As such, instead of the Abstract Factory pattern, Haskell uses its built-in "Prototype" pattern to create data where it's needed.

### 8.3.2 Applicability of Adapter

The Adapter pattern suits Java very well, with Java being able to support the pattern in multiple ways. The pattern can be applied verbatim in problem areas whenever necessary.

The pattern is also applicable to Haskell and FP, but not necessarily in the form proposed in [4]. It can be applied as a simple function, transforming data into another format, or as a Newtype-pattern, wrapping an existing type and replacing or extending its type class instances.

### 8.3.3 Applicability of Command

The Command pattern, we feel, is somewhat difficult to reason about, but is nonetheless simple to implement in Java. The pattern reinforces Java's reliance on side-effects, and might make it hard to track how, why and when something happened in a system. Even so, it is possibly the best solution to the problem described in the problem space described in [4].

The Command pattern is used quite a lot in Haskell, but in a different form. It is quite common to want to track the effectful operations performed by a function. As such, it is common to build a set of domain-specific languages (DSLs) for dealing with these effects. These DSLs can then be used to issue commands that can later be interpreted into real effectful operations. This method of abstracting out effects can be done in a multitude of ways, but the most common techniques are called “mtl-style”[5], using the brilliant Haskell package `mtl`, and “Freer monads”, which use a specific data type that, provided a functor, can form a “free” monad. The theory and specifics of these two methods are left up to the reader to discover further.

### 8.3.4 Applicability of Composite

The Composite pattern is simple to implement, and simple to use, in Java. It is a simple way to encode recursive and/or hierarchical structures. Our only qualm about the pattern is that methods defined in the interface for the composite that are meant for the compositions are not implementable for the non-compositions, and so they must either throw an exception or not do anything. This feels like a leaky or incomplete abstraction to us, but it is exactly this way it is described in [4].

In Haskell, seeing as it’s common to catch invalid cases in types and force them to be compilation errors rather than runtime exceptions, the composite pattern is not directly applicable. Through the use of ADTs, it is simple to implement the pattern, but having operations that should only work on compositions rather than leaf values complicates matters. It is more common to attempt to write “complete” functions, that have a result for any value of the input type, as opposed to “partial” functions, that might not have a solutions for all input values. Partial functions complicates matters, as catching exceptions from pure functions is error-prone, cumbersome and not particularly performant. As such, it is more common to treat compositions and leaf values distinctly, and instead build a hierarchy through ADTs if one is desired.

### 8.3.5 Applicability of Decorator

The Decorator pattern is a useful pattern for Java. The pattern is a nice way of extending a class selectively and through composition instead of inheritance.

This pattern is used in Haskell almost verbatim to how it is defined in [4] in the form of the Newtype pattern. Extending an existing type’s type class instance by wrapping it in a `newtype` and defining a different instance for it is common practice in Haskell.

### 8.3.6 Applicability of Iterator

The Iterator is so well-suited to Java that it is a part of its standard library as `java.util.Iterator`. While the implementation does not correspond exactly to the definition in [4], it serve the same purpose, and can be used in mostly the same way.

In Haskell, there are multiple ways to achieve the same effect as the Iterator pattern. We opted to use the Foldable type class, but a simpler definition would be to just convert the collection to a list, which is lazy by default. So while iterator objects themselves do not fit Haskell, the pattern's problem space and solution are so similar in Haskell and FP to the definition in [4] that we conclude the pattern fits Haskell very well.

### 8.3.7 Applicability of Prototype

The Prototype pattern is less useful in Java than most of the other patterns due to Java's simple object instantiation mechanisms. It is nice to use when object instantiation is complicated, or if there is a predefined set of objects that exhibit an archetype to create multiple objects of.

The Prototype pattern exists "by default" in Haskell and FP: when values are copy-on-modify, which is the case for Haskell, the prototype pattern is effectively the default behaviour when modifying a value in general.

### 8.3.8 Applicability of Strategy

The Strategy pattern is innately very simple, and implementing it in Java is about as simple as implementing it in Haskell. It is at its core just HOF with objects instead of first-class functions, and it achieves this effect very succinctly given the usefulness of the pattern.

HOFs are widely used in FP, but is not considered a pattern in and of itself in FP since functions are first-class values in FP, and passing functions as arguments is one of the core tenets of FP.

## 8.4 Code size

Differences in LSLOC could be because of differences in Java and Haskell's standard libraries. Java's standard library has a small number of utility "functions", but is more focused on utility classes. While these utility classes are helpful in situations that fit them, they are applicable to fewer situations than Haskell's standard libraries' utility functions, which aim to be completely agnostic to the environment they are used in. As such, we have inevitably ended up using more of the standard library in the Haskell implementations than in the Java implementations. Unfortunately, we did not foresee this exact issue when initially planning this study.

Differences in PSLOC is mostly caused by Java's C-like syntax, with curly braces to delimit blocks of code. The closing curly brace is on a dedicated line in Java, and as such a block of code in Java will always count 1 more PSLOC than an equivalent block of code in Haskell.

Haskell, and FP in general, is focused on composing together functions. This often means that a function using two other functions could be written in one line, à la `foo x = bar (baz x)`, while in Java it is more common to write these on separate lines, especially for longer chains of functions.

Even with these considerations, we can clearly see a consistent trend of the Java implementations having a higher PSLOC than the Haskell implementations.

## 8.5 Complexity

Java has only marginally higher CC, but Haskell has significantly higher CC density. The Haskell implementations have on average 22.9% higher CC density than the Java implementations, even though they have the same CC score. As such, it is safe to say that the Haskell implementations are not any less complex than Java implementations, but they spread their complexity over a smaller area. This is most likely as a result of the Haskell implementations' high number of functions, which each count at least 1 CC, while in Java it's more common with fewer, more complicated methods. The Haskell implementations and their code might then seem more compact, and more obtuse when compared to the Java implementations, with more things happening per line of code. This is to be expected, given that Haskell focuses on composing smaller functions into bigger functions.

The case with the largest difference in CC density is the Adapter case, where the Haskell implementation is 60% more dense, even though the Adapter pattern is a pattern we find to be fitting for Haskell through using the Newtype pattern. Both languages have roughly the same CC in this case, but the Haskell implementation is considerably smaller in terms of LSLOC, giving it such a high CC density.

Both languages have very low CC density in the Prototype case, cementing our concern that the case requirements might have been too simple.

## 8.6 Discussion of limitations

In this section we describe the limitations of this thesis work, and details about this study that can be done better or differently.

### 8.6.1 Master's thesis

This is merely a master's thesis, and even though we are passionate about the work we have done during this thesis work, our abilities are inadequate to draw proper and definitive conclusions on this matter.

### 8.6.2 Functional programming bias

We have more experience working in Haskell and with FP from other languages like Javascript and Elm than with OOP. As such, our views here are heavily biased towards FP.

### 8.6.3 Code quality

The code written for this thesis work is most likely sub-optimal in both languages. We have little experience with production-ready Java code, and our Haskell code was probably heavily influenced by the Java code since we wrote the Java implementations first, and then tried to mimic them in Haskell.

### 8.6.4 Subjectivity

The qualitative observations made are rather subjective, and reflect qualities we personally find “good” and “bad” in a program. As such, they might not reflect what commonly constitutes “good” and “bad” programs, although that is a rather muddled and difficult topic in and of itself.

### 8.6.5 Inadequate metrics

The metrics used when comparing the implementations do not measure every important quality about the programs.

Using Hyperfine to measure the execution time for the implementations does not give a very accurate result of the written programs’ execution times, especially when comparing them between environments, since the JVM does a considerable amount of work on start-up compared to the Haskell runtime, since JVM does just-in-time (JIT) compilation.

### 8.6.6 Small programming paradigm scope

This study compared only OOP and FP, and then only two languages, which we deemed “representative of their paradigm”. Not all OOP languages are like Java, and not all FP languages are like Haskell. Frankly speaking, there are rather few languages that are like those two languages, and they might not represent these other different languages adequately.

### 8.6.7 Standard library pollution

By only limiting available packages and modules to those in the standard library for each language, we have muddled the results from our implementations by not only comparing the languages but also their standard libraries against each other. One standard library might contain functions, classes and utilities the other does not, which might give one language an unfair advantage in a given implementation.

## 8.7 Summary of discussion

In this chapter we have discussed our findings from this study. We have discussed the various correlations and observations we have made in this study, as well as drawn conclusions as to the cause of these correlations and how they affect the applicability of these design patterns. We have also described the limitations we see in our work, even though we are confident in the conclusions we have arrived at.

# Chapter 9

# Conclusion

## 9.1 Introduction to conclusion

For this thesis, we wrote multiple simple programs in two different programming paradigms, implementing some of the commonly known design patterns from both paradigms, and compared them in terms of various metrics and qualitative observations.

## 9.2 Highlights

We summarize our key findings from our qualitative observations and metrics.

**Problems** Some problems in OOP simply do not exist in FP. As such, many of the design patterns from OOP do not apply to FP, and vice versa. OOP is concerned with abstracting the structure of objects and interfaces, while FP is concerned with abstracting behaviour and functions and composing them together. Both are concerned with reducing coupling and producing maintainable code, but the structure of an OOP system is so different from an FP system that they rarely encounter similar problems.

**Solutions** Some patterns are simple to implement in OOP, while the equivalent solution in FP might be complex, or require more than one pattern. In OOP, it is not uncommon for one part of a system with its own high-level responsibilities, composed of multiple classes and interfaces, to use between one and five patterns. In FP, since type classes are a form of design pattern, it is more common to have more than 10 or 20 patterns applied in one part of the system. OOP design patterns are more often than not ways to structure code, and implicitly dictate some behaviour, while FP patterns are mere functions to be applied wherever they might be needed, with only a few patterns resembling the OOP structure of a design pattern.

**Sizes and complexities** FP patterns are smaller than OOP. FP patterns are functions or otherwise very simple structures concerning one thing at a time, while OOP patterns most often concern multiple related concretions and interfaces at a time and how they interoperate. This amounts to a sizeable difference in both SLOC and LSLOC, as well as the number of parts needed for operation.

**Concerns and values** Patterns in each paradigm are solutions to problems for that paradigm and that paradigm's problem space. A given problem in an OOP context might be considered a considerably smaller problem in an FP context, and vice versa. So while most OOP patterns are implementable in FP, they bear a lower, if not non-existent, value there. While FP focuses on composing expressions, functions and programs together, OOP focuses more on imperative operations and object representation.

## 9.3 Summary of conclusion

In this chapter, we have discussed our conclusions and outlined our reasoning for arriving at those conclusions.

# Chapter 10

# Future work

## 10.1 Introduction to future work

In this chapter we discuss improvements and additions to the findings in this thesis.

## 10.2 More languages

Using more languages, possibly from more than two paradigms, might provide more value to the comparisons.

## 10.3 More design patterns

More of the design patterns can be explored, both in OOP, FP and other paradigms we have not explored in this thesis.

## 10.4 Better metrics

The metrics used in this study proved to be of varying quality. A future study of this particular topic would be wise to choose metrics more carefully. Special emphasis is put on attempting to find metrics that are comparable across languages and paradigms, something we had trouble with.

## 10.5 Summary of future work

The possible improvements and additions are endless, and leave much to be explored in terms of comparing programming paradigms, their design patterns, and programming languages.

## Acronyms

| | | |
|---|---|---|
| **OOP** | object-oriented programming | 3 |
| **ADT** | algebraic data type | |
| **FP** | functional programming | 4 |
| **lazy** | non-strict | 13 |
| **HOF** | higher-order functions | 13 |
| **CS** | computer science | 18 |
| **JVM** | Java virtual machine | 16 |
| **SLOC** | source lines of code | 50 |
| **GoF** | Gang of Four | 3 |
| **PSLOC** | physical source lines of code | vii |
| **LSLOC** | logical source lines of code | vii |
| **CC** | cyclomatic complexity | vii |
| **HOF** | higher-order function | 13 |
| **DSL** | domain-specific language | 114 |

# Bibliography

[1] Christopher Alexander. *A pattern language: towns, buildings, construction*. Oxford university press, 1977. URL: https://books.google.no/books?id=mW7RCwAAQBAJ&lpg=PR5&ots=fx43W5r9S-&dq=A%20Pattern%20Language%3A%20Towns%2C%20Buildings%2C%20Construction&lr&pg=PR5#v=onepage&q&f=false (visited on 26/05/2020).

[2] Ken Arnold et al. *The Java programming language*. Vol. 2. Addison-wesley Reading, 2000. URL: http://www.academia.edu/download/56545110/1arnold_k_gosling_j_holmes_d_the_java_programming_language.pdf (visited on 26/05/2020).

[3] Ole-Johan Dahl, Edsger Wybe Dijkstra and Charles Antony Richard Hoare. *Structured programming*. Academic Press Ltd., 1972.

[4] Erich Gamma et al. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994, p. 395. ISBN: 0-201-63361-2.

[5] Andy Gill, Jeff Newbern and Andriy Palamarchuk. *mtl: Monad classes, using functional dependencies*. URL: https://hackage.haskell.org/package/mtl (visited on 10/05/2021).

[6] Jan Rune Holmevik. 'Compiling SIMULA: a historical study of technological genesis'. In: *IEEE Annals of the History of Computing* 16.4 (1994), pp. 25–37. URL: https://www.idi.ntnu.no/grupper/su/publ/simula/holmevik-simula-ieeeannals94.pdf (visited on 26/05/2020).

[7] Paul Hudak and Joseph H. Fasel. 'A gentle introduction to Haskell'. In: *ACM Sigplan Notices* 27.5 (1992), pp. 1–52.

[8] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003. DOI: 10.1.1.63.4393.

[9] Nick Langley. 'Write once, run anywhere?' In: *ComputerWeekly* (2nd May 2002). URL: https://www.computerweekly.com/feature/Write-once-run-anywhere (visited on 18/05/2020).

[10] libraries@haskell.org. *Basic Libraries: Prelude*. 2001. URL: https://hackage.haskell.org/package/base-4.12.0.0/docs/Prelude.html (visited on 21/05/2020).

[11] Simon Marlow et al. *Haskell 2010 Language Report*. URL: https://www.haskell.org/definition/haskell2010.pdf (visited on 19/04/2021).

[12] Thomas J. McCabe. 'A Complexity Measure'. In: *Proceedings of the 2nd International Conference on Software Engineering*. ICSE '76. San Francisco, California, USA: IEEE Computer Society Press, 1976, p. 407. DOI: 10.5555/800253.807712.

[13] Merriam-Webster, ed. *Paradigm*. Merriam-Webster. URL: https://www.merriam-webster.com/dictionary/paradigm (visited on 25/05/2020).

[14] Bertrand Meyer. *Object-oriented software construction*. Vol. 2. Prentice hall New York, 1988. URL: https://www.academia.edu/download/32169601/Object-Oriented_Software_Construction.pdf (visited on 25/05/2020).

[15] Vu Nguyen et al. 'A SLOC Counting Standard'. In: *COCOMO II Forum 2007*. 2007. URL: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=057D73DBE4DC4E993A2EDA27351F9EA8?doi=10.1.1.366.196&rep=rep1&type=pdf (visited on 12/04/2021).

[16] Oracle. *The Java™ Tutorials. Autoboxing and Unboxing*. URL: https://docs.oracle.com/javase/tutorial/java/data/autoboxing.html (visited on 28/05/2020).

[17] David Peter. *Hyperfine*. URL: https://github.com/sharkdp/hyperfine (visited on 13/04/2021).

[18] The University of Glasgow. *Monad type class documentation*. URL: https://hackage.haskell.org/package/base-4.14.1.0/docs/Prelude.html#t:Monad (visited on 11/03/2021).

[19] *Typeclassopedia*. URL: https://wiki.haskell.org/Typeclassopedia (visited on 12/03/2021).

[20] Philip Wadler. 'Monads for functional programming'. In: *International School on Advanced Functional Programming*. Springer. 1995, pp. 24–52. URL: https://homepages.inf.ed.ac.uk/wadler/papers/marktoberdorf/baastad.pdf (visited on 28/05/2020).

[21] Philip Wadler and Stephen Blott. 'How to make ad-hoc polymorphism less ad hoc'. In: *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, pp. 60–76. URL: https://dl.acm.org/doi/abs/10.1145/75277.75283 (visited on 28/05/2020).

[22] Peter Wegner. 'Concepts and Paradigms of Object-Oriented Programming'. In: *SIGPLAN OOPS Mess.* 1.1 (Aug. 1990), pp. 7–87. ISSN: 1055-6400. DOI: 10.1145/382192.383004. URL: https://doi.org/10.1145/382192.383004 (visited on 25/05/2020).

[23] Brent Yorgey. 'The Typeclassopedia'. In: *The Monad.Reader* (13 12th Mar. 2009). Ed. by Wouter Swierstra. URL: https://wiki.haskell.org/wikiupload/8/85/TMR-Issue13.pdf (visited on 11/03/2021).