Abstract

Software design patterns, descriptions of design structures that provide reusable solutions to frequently occurring problems in software engineering, are widely used in modern object oriented programming. However, in order to use patterns effectively, it is important to recognise which patterns are best suited for specific contexts and languages.

In this study, we examined five commonly used design patterns; Abstract Factory, Factory Method, Prototype, Strategy, and Flyweight. Implementations of these were compared across four popular programming languages; Java, C#, Python, and JavaScript, to test their efficiency and ease of implementation.

From these comparisons, we determined that the chief deciding factor in design patterns’ applicability in various languages, especially those patterns that rely heavily on inheritance and interfaces, is whether the language in question is statically or dynamically typed. The majority of patterns examined were of the creational category, and all of these exhibited this trait.
Acknowledgements

I would like to thank the student café Escape for its endless supply of coffee and stimulating discussions, without which I would never have made it this far.

Thank you to my therapist Brynjar Fyhn for helping me find the mental fortitude to push through the dark times and complete this project, albeit after several extensions.

Thank you to Emilie, Kasper, Kyle, and James for their continued moral support and for keeping my feet on the ground.

Finally, a very special thank you to my supervisor, Eric Jul, for being a tremendous source of guidance, wisdom, and humour.

Blindern, February 1st, 2017
Vebjørn Ring Sjølyst
# Contents

## I Introduction

1. **Introduction**  
   1.1 Definition  
   1.2 Motivation  
   1.3 Goal  
   1.4 Approach  
   1.5 Work Done  
   1.6 Evaluation  
      1.6.1 Complexity  
      1.6.2 Runtime  
   1.7 Results  
   1.8 Conclusion  
   1.9 Outline

## II Background

2. **Introduction to Design Patterns**  
   2.1 The History of Design Patterns  
   2.2 Creational Patterns  
      2.2.1 Factory Method  
      2.2.2 Abstract Factory  
      2.2.3 Prototype  
   2.3 Structural Patterns  
      2.3.1 Flyweight  
   2.4 Behavioural Patterns  
      2.4.1 Strategy  
   2.5 Summary

3. **Languages**  
   3.1 Statically Typed Languages  
      3.1.1 Java  
      3.1.2 C#  
   3.2 Dynamically Typed Languages  
      3.2.1 Python  
      3.2.2 JavaScript  
   3.3 Summary
List of Figures

2.1 UML class diagram for Factory Method pattern. . . . . . . . . 12
2.2 UML class diagram for Abstract Factory pattern. . . . . . . . . 14
2.3 UML class diagram for Prototype pattern. . . . . . . . . . . 15
2.4 UML class diagram for Flyweight pattern. . . . . . . . . . . 16
2.5 UML class diagram for Strategy pattern. . . . . . . . . . . 17
List of Tables

4.1 Ease of implementation for each pattern . . . . . . . . . . . . . 49
Part I

Introduction
Chapter 1

Introduction

As computer systems grow ever more complex, using standardised mechanisms and structures becomes increasingly important in order to preserve efficiency and legibility of the code. Such structures, commonly known as design patterns, have long been used in other disciplines such as architecture, but in the young field of informatics they are not yet quite as widely known or understood.

A design pattern’s main function is to provide a reusable and flexible solution to a commonly encountered problem. As such, the proper usage of design patterns, especially in object-oriented software, leads to more robust and flexible systems. For this purpose, and due to the inherent differences in programming languages, it is important to determine which patterns will be most useful in solving a given problem.

1.1 Definition

One of the problems with design patterns is that there is no objective definition that precisely describes what constitutes a design pattern and what does not. However, they must all have the four following attributes: A name, a description of the problem it is meant to solve, an abstract description of the proposed solution in terms of objects and interfaces, and any significant consequences of using that particular pattern in a system.[5, p. 3]

When defining a design pattern, choosing a good name is of critical importance. Names should be one or two words which relate to the solution the pattern provides. A short yet descriptive name makes it significantly easier to discuss the pattern in question, by removing the need to describe its structure every time it is mentioned.

To help developers decide when a given pattern should be used, they must include a description of the problem; the circumstances in which the pattern is applicable, or symptoms of poor design choices which the pattern can alleviate or rectify.

The most important part of the design pattern, the solution, describes how one should go about preventing or overcoming the problem described previously. This is done by providing an abstract overview of the objects
and interfaces used and how they relate to each other. It is important to mind the level of abstraction used, as the solution needs to be abstract enough to be implementable in many different situations, but also needs to be concrete enough for it to actually help solve the problem.

Lastly, listing the consequences of implementing a design pattern can help the developer determine if the trade-offs are worth it, or if they would be better served using another pattern.

1.2 Motivation

In modern object oriented programming, design patterns are a useful tool for many situations, and especially in large enterprise systems they can be a great help in making the code not only more robust and flexible, but also more legible and thus more easily maintainable. Design patterns are not only applicable in large scale systems, though, as they can be useful in even the smallest project. Because of how intuitive some of them are, many programmers even implement certain patterns without knowing that there is a formal name and description for them, simply due to their experience leading them to deduce the most efficient ways to solve certain commonly occurring problems.[11]

Despite this, a program can not necessarily be improved just by adding arbitrary patterns without considering their consequences. To make the most out of a pattern, it is essential to understand when it will be helpful, and when it is not needed, or in the worst case would be detrimental to the system. More than just the problem to be solved, however, it is also important to take into account what language is being used. Different languages employ different mechanisms and are built differently, and as such certain patterns may prove much more useful in one language than in another.

1.3 Goal

The goal of this thesis is to determine the impact languages have on the successful implementation of patterns, which metrics are most affected, and which characteristics of languages have the most effect on how beneficial patterns can be in a given situation.
1.4 Approach

To compare the patterns, several simple programs were written. In most cases one program was written for each pattern for each language, though in some cases two programs were written for the same pattern in the same language in order to better compare the efficiency of the program when using said pattern versus when not using the pattern.

The program would first be written in Java using the pattern, then rewritten in each of the other languages with the same functionality and similar structure. One potential weakness with this approach is that the structure of the program would be influenced by the language it was first written in, such that the Python version might not utilise Python’s strengths to as much of an extent as it could have if it had first been written in Python. This weakness would be difficult to get around, however, as each language’s version of the program needs to be sufficiently similar for a proper comparison to be possible; if the programs were too different, changes in efficiency could easily be attributed to differences in implementation rather than the suitability of the pattern itself.

1.5 Work Done

For the purposes of this thesis, a number of patterns have been investigated, most of which are categorised as creational patterns. The efficiency and ease of implementation of these have been compared across the languages Java, Python, and Javascript (with the possibility of adding C# to this list).

The patterns investigated are as follows:

• Factory Method, a creational pattern that provides a common interface for creating any number of subclasses while leaving the specifics of which subclass to instantiate transparent to the client.

• Abstract Factory, a creational pattern closely related to Factory Method, that allows a client to create a family of related or dependent objects without having to worry about which specific classes to instantiate.

• Prototype, a creational pattern that instantiates classes called prototypes. Whenever a new object of the prototype’s class is requested, a clone of the prototype is returned rather than an entirely new object.

• Strategy, a behavioural pattern that makes methods interchangeable, allowing algorithms to vary independently of the client using them.

• Flyweight, a structural pattern that removes the need to create many identical objects of the same type, instead storing a single instance to be represented in many different parts of the code.

The programs written for each of these patterns were separate, though based on each other and in some cases nearly identical due to similarities
in the patterns. The implementations for Factory Method and Abstract Factory, for instance, only differ in which classes the methods belong to.

1.6 Evaluation

When performing a comparison, determining the metrics to be used to measure the differences between languages is just as important as the measurements themselves. Using the wrong metrics could lead to reaching incorrect conclusions, making the comparison useless or in the worst case misleading. Not using enough metrics can also adversely affect the results, however, potentially leading to equally inaccurate conclusions.

For the purposes of this study, the most important metric to measure was determined to be the complexity of the code, as well as the total number of lines of code. Additionally, where relevant, the runtime of the program was also taken into consideration.

1.6.1 Complexity

In many cases, the complexity and number of lines of code are a deciding factor in determining whether or not a design pattern is fit for implementation in a certain language. One must distinguish between complex code and complicated code, however, as high complexity does not necessarily imply that the code is difficult to read and understand.

The logical complexity of a program is determined by the total number of interacting entities the program contains, as well as how many individual paths it is possible to trace through an execution of the program. When writing highly complex systems, there is a significant risk of inadvertently introducing bugs and logical errors, whose sources can be exceedingly difficult to track down. Ultimately, this can lead to a system that is almost unmaintainable. [14]

One of the most important factors when it comes to the complexity of a system is the built-in features and mechanisms of the language. In some cases, the language may employ features which render the pattern superfluous; so trivial to implement that it effectively cannot be called a pattern, or on the other hand it may prove so cumbersome to implement that it would be better to find a different pattern or to not use one at all.

1.6.2 Runtime

While comparing the runtime of two different programs written in the same language is simple, comparing this across languages can be challenging; determining whether a difference in runtime is due to the implementation of the design pattern or the simple fact that different languages run at different speeds can be almost impossible on a case-by-case basis.

One might be able to circumvent this problem by comparing many different code snippets across languages and using those as a baseline, but for the scope of this study, we will only compare the runtime for certain
patterns where the runtime is noticeably affected by the choice of whether or not to implement the pattern in a given language.

1.7 Results

Upon implementing the various patterns, we found that in general, creational patterns prove simple to implement in Java and C#, and successfully aid in reducing the complexity of the code, as well as increasing the legibility of the code. In Python and JavaScript, however, the results varied from pattern to pattern, though in most cases the pattern was either more cumbersome to implement, or simply irrelevant due to the languages’ built-in features.

- Factory Method and Abstract Factory were simple to implement, but Python’s duck typing as well as its "Easier to Ask for Forgiveness than Permission" philosophy make interfaces superfluous, and forcing the class hierarchy not only resulted in an increased number of lines of code and increased complexity, it also resulted in longer run times due to the heavy isinstance function.

- Similarly to the Factory Method and Abstract Factory patterns, implementing the Strategy pattern in Python only resulted in a higher number of lines of codes, as well as an increased runtime directly proportional to the number of objects whose hierarchy needed to be checked.

- The Prototype pattern, while not strictly detrimental to the program, proved wholly unnecessary in both Python and Javascript. Python provides direct access to any object’s class object, which makes for a far simpler way to copy an object than using the Prototype pattern. Javascript on the other hand already uses prototypes for all its objects inherently, making the explicit implementation of the pattern redundant.

1.8 Conclusion

From these results, we concluded that the main factor in determining how useful a pattern will be in a given language is whether the language is statically or dynamically typed. For the most part, creational patterns are well suited for statically typed languages, but are either redundant or outright detrimental to programs written in dynamically typed languages. This appears to be mainly due to the fact that creational patterns often make use of abstract classes, interfaces, and inheritance, as other patterns which are not creational but require these features also seem best suited for statically typed languages.
1.9 Outline

This thesis is split into four parts. Part one provides an overview of the study and a summary of the results and conclusions. Part two describes the background of the thesis, briefly documenting the history of design patterns and the so-called Gang of Four. Part three details the comparisons made and the programs written. Part four provides the conclusion made based on the work done.
Part II

Background
Chapter 2

Introduction to Design Patterns

In the field of software engineering, design patterns are used to solve commonly occurring problems that would otherwise cause programs to become complex, unwieldy, and inefficient. To do so, they provide standardised methods that define how to create objects, how to link objects to form a cohesive system, and how to make these objects interact efficiently.

There are three different categories of design patterns – creational, structural, and behavioural patterns. Each pattern of a given category solves similar but distinct problems, but in some cases these problems overlap enough that one pattern can successfully be substituted for another. Additionally, several patterns interact in such a way that it can be beneficial to implement two patterns to counteract a single problem. As an example, the Factory Method and Abstract Factory patterns solve the same problem of standardising object creation, and are similar enough that in many cases, they are considered to be the same pattern, simply referred to as the "factory pattern."

This chapter provides a brief look at the history of design patterns, then goes on to introduce the patterns examined in this thesis, grouped by their category. Each pattern is described by defining its name, what problem it solves, how it solves it, and what the possible consequences of implementing it are.

As there are too many design patterns to examine all of them for the purposes of this study, the ones below were selected in order to be able to examine one of the categories, the creational patterns, in-depth, while one pattern from each of the two other categories were also selected in order to provide contrast in case the ones chosen from amongst the creational ones were too uniform in their implementation.
2.1 The History of Design Patterns

The idea of using patterns or predefined samples to aid in the design of buildings is an old one, but the concept was formalised by American architect Christopher Wolfgang Alexander. He proposed that the users of a building should be the ones to design it, and published a large amount of such patterns to make this possible.[7]

Later, this concept has spread to other disciplines. In 1987, software engineers Kent Beck and Ward Cunningham, who had been studying Christopher Alexander’s patterns, adapted the concept to a software project whose design proved difficult to complete. They later reported the results at the OOPSLA conference, but it was not until the OOPSLA workshop of 1992 when Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides got together and started working on developing the concept, publishing their book Design Patterns: Elements of Reusable Object-Oriented Software in 1995.

2.2 Creational Patterns

Creational patterns address how objects are instantiated, and how this process can be made transparent to the client and the system at large, so that the system can work independently of how its objects are created.

2.2.1 Factory Method

![UML class diagram for Factory Method pattern.](image)

Figure 2.1: UML class diagram for Factory Method pattern.
Problem

In cases where a framework is required to provide a standardised method for creating objects, but without access to knowledge about which specific object to create, the Factory Method pattern may be used to define a separate method for creating the objects, letting subclasses be responsible for the specifics of each type of object to be created.

Solution

There are several ways to implement this pattern, but the implementation used in this project involves creating a superclass with a method which returns an object, as demonstrated in figure 2.1. This method is then overridden in the class’ subclasses, so that each extended factory method returns a specific subclass of the object to be created. Note that the superclass may be either abstract or concrete, depending on whether or not an appropriate base case exists for the given object family.[4]

Consequences

By employing the Factory Method pattern, one eliminates the need to hardcode handling of specific classes in the system, instead only requiring that one handle the interface common to all the objects to be created. This results in greater flexibility, as one only needs to create a new subclass if a new variation of the object is introduced, rather than having to change every part of the system where these objects are used.

2.2.2 Abstract Factory

Problem

When a large family of related objects must be created, but without prior knowledge as to the specifics of the objects’ behaviour, the Abstract Factory pattern can be useful. An example of this is a system whose user interface is different depending on which operating system it is run on. The various components will perform largely the same function regardless of operating system, but the specifics of their implementation must be different.

Solution

The Abstract Factory pattern solves this problem by implementing a "factory of factories," shown in figure 2.2. This is a class that defines one factory for each type of object to be created. One can then instantiate subclasses of this factory to create the various objects, only needing to know of each object’s interface rather than all the details of their implementation. [4]
Consequences

This pattern grants a great deal of control over how objects are created, and how they are allowed to interact with each other. The fact that a single factory is responsible for the creation of all the component objects means that one can ensure that each object is compatible with each other. It also allows one to easily switch between different object families without changing more than which specific factory is to be used. However, adding new types of objects to the pattern is difficult, as this involves changing the interface of the factory, and as such requires the extension of both the superclass and all of its subclasses.

2.2.3 Prototype

Problem

If a program needs to create many different objects which perform similar tasks, creating a separate factory for each of them may be expensive to the point of being unfeasible. In this case, the Prototype pattern allows one to only use one factory for all of the objects, greatly reducing the necessary number of lines of code.
Solution

As long as all the objects to be created implement an interface which supports a clone function, a factory can be created which takes such a class as a parameter, and returns a copy of it. This is demonstrated in figure 2.3. This separates the factory from the specifics of the object it creates, allowing a single factory to instantiate any number of different objects so long as they all conform to the same interface. All that is required is to have a prototypical instance of the object, which can then be copied. The attributes of each copy may then be changed to fit its intended purposes, if necessary. [4]

Consequences

Implementing this pattern allows the system to add new concrete objects to its factories at runtime, simply by creating and registering a prototypical instance of it. This gives the client much more flexibility than many other patterns can provide. By using object composition and deep copying, one may even create prototypes of groups of objects, allowing one to easily copy subsets of object groups.
2.3 Structural Patterns

Structural patterns help define the overall shape of a system, by defining how objects interact to form larger, more complex structures.

2.3.1 Flyweight

![UML class diagram for Flyweight pattern.]

**Problem**

In a system that requires a large number of identical objects, creating each object as a separate instance can be quite taxing on the system, requiring much more space than is desired.

**Solution**

The Flyweight pattern lets objects be shared, so that a single instance may be represented in several different locations and contexts. The flyweight contains only the attributes that are shared across all occurrences of the object, and any information specific to a given representation must be inferred from the context.[4] See figure 2.4 for details.

**Consequences**

Implementation of the Flyweight pattern can decrease the space requirements of the system greatly, depending on how many times the objects occur in the system. However, it may cause runtime issues, as calculating the attributes of a given occurrence may be expensive, especially if these attributes were originally stored in the object instance as variables.
2.4 Behavioural Patterns

Behavioural patterns define how classes behave, how they communicate with each other, and what responsibilities each object holds.

2.4.1 Strategy

![UML class diagram for Strategy pattern.](image)

Figure 2.5: UML class diagram for Strategy pattern.

**Problem**

There are often several different algorithms for performing a given task, and different approaches may be useful in different situations. Having to implement each of the potential algorithms as part of the classes that use them would make the system excessively complex, and this is what the Strategy pattern is designed to avoid.

**Solution**

Instead of performing the necessary action itself, a class can delegate this responsibility to a separate object which implements a certain interface, as exemplified in figure 2.5. Any object implementing this interface may then be used interchangeably, the differences in algorithms completely transparent to the class that references it.[4]

**Consequences**

Like the Abstract Factory and Factory Method, this pattern provides a great deal of flexibility, making the system independent of which specific algorithm it implements. This allows the client to freely choose which strategy to use, though one potential drawback is that the client must be
aware of the various strategies and how they differ in order to select the one best suited for the task at hand.

2.5 Summary

Design patterns are widely used in today’s software engineering, but have their origins in architecture. Software design patterns are divided into three categories, of which creational patterns have been most thoroughly examined for the purposes of this study. Behavioural and structural patterns have also been compared, but not to as great an extent as the creational ones. The main components of a design pattern are its name, the problem it endeavours to solve, the solution to this problem, and the potential consequences of implementing the pattern.
Chapter 3

Languages

There exist many programming languages of widely differing types, each equipped to solve different types of problems, and in different ways. As such, there are a myriad ways to classify different programming languages. In the broadest sense, they may be split into how high level they are, or how far removed from machine code they are in terms of abstractions. As all languages investigated in this project are considered high level, this means of classification proves to be irrelevant in this context.

Another means of classifying languages is by their programming paradigm. This is a way to distinguish languages by their coding style. Various features of a programming language help determine which paradigms it falls under, and as such paradigms are generally not mutually exclusive, and each language can fall under several paradigms. Object oriented programming is one such paradigm, and even though there exist design patterns for languages that are not object oriented[16], all of the ones used for this study were developed with object orientation in mind.

Because our project addresses only a specific subset of programming languages, a specific classification is required as well. Object oriented languages can generally be divided into whether their types are static or dynamic, and this distinction suits our purposes well. The following languages have thus been chosen with this in mind. Two languages from each of the two categories were selected, all of them ubiquitous in contemporary software engineering. This to provide more than one reference point, in case there are other more important deciding factors in the implementation of patterns than whether the language in question is statically or dynamically typed.

The following sections describe the languages used in this project and their relevant features and mechanisms. A code snippet is also included for each language to demonstrate how variables and object creation work in each of them.
3.1 Statically Typed Languages

A language that employs static typing requires that when a variable is declared, its type must also be declared. Additionally, these languages often use strong typing, meaning that a program’s type safety is checked at compile time, and type errors will cause the compilation to fail. This ensures that variables are not used incorrectly, and allows any bugs resulting from type errors to be caught early in the development process. This philosophy of always ensuring type safety is known as "Look Before You Leap," and is used in most of the older, well established object oriented languages such as Java[9], C++[2], C#[15], and Emerald[13].

3.1.1 Java

Java is one of the most popular programming languages in the world, released in 1995 by Sun Microsystems[6] which were later acquired by Oracle Corporation. It is designed to be as portable as possible, so that any Java program can be run on any platform which supports Java without needing to be recompiled.

Types

Java utilises two categories of types: Primitive types and reference types. Primitive types are predefined by the language, and their names are reserved keywords. The primitive types are divided into two additional categories: Boolean and numerals. Boolean is a single type which can have the value true or false, whereas numerals are a collection of types such as char, byte, int, and float. Values of the numeral types may be implicitly cast to each other as long as there is no risk of data loss (an int is guaranteed to have enough space to store any value stored in a byte, for instance), or explicitly if data loss may occur (such as casting a floating point number to an integer). The boolean type is incompatible with any other primitive type, and may not be cast to a numeral, explicitly or implicitly.

Reference types, like primitive types, are divided into categories. There are four such categories: Class types, interface types, type variables, and array types. The two categories most relevant to the purposes of this project are the class and interface types, and these are described in more depth in the following two subsections. The name of a reference type is defined in its declaration. These names must not be one of the predefined keywords of the language, and must be unique within their scope.

Classes and Objects

As a high level object oriented language, Java supports polymorphism, single inheritance, and abstraction. Classes are by default concrete, but may be defined as abstract, so that the given class cannot be instantiated directly, and instead must be extended to concrete subclasses. Due to its single inheritance, each class may only be a subclass of one superclass, but
may implement several interfaces. The following example illustrates how a class is declared, including both inheritance from a superclass and the implementation of an interface.

class myClass extends MyParent,  
   implements MyInterface {

   private int myNum;

   public myChild(int number) {
      this.myNum = number;
   }

   public void myInterfaceMethod() {
      // ...
   }
}

In the above example, the class myChild has one integer as a class variable, and one constructor method which takes an integer as an argument, and assigns that number to its class variable. New objects in Java are created by calling one of its constructors. Seing as myChild has only one constructor which takes an int as its sole argument, any attempt to instantiate it by using a different type or number of arguments will cause an error during compilation. The simplest, and by far the most common, way to create an object in Java is by using the new keyword, as demonstrated below:

MyClass m = new MyClass(myInteger);

An object can be stored in a variable whose type is declared as the object’s parent, though only methods and variables present in the parent may be accessed through such a variable unless the object is explicitly typecast to the child class.

Interfaces

An interface is like an abstract class in that it declares methods and cannot be instantiated directly, but two important distinctions exist: An interface may not declare variables, and a class may implement any number of interfaces rather than just one. Like with abstract classes, a variable may be declared to have the type of an interface despite the interface itself not being instantiable. This variable can then hold an object of any class which implements the interface, and the methods defined in this interface are the only ones which are accessible unless the object is explicitly typecast to either its specific class or one of its superclasses. The interface implemented above would be declared as follows:

public interface MyInterface {
   public void myInterfaceMethod();
}


An interface may declare as many methods as desired, both public, private, and protected. Any class implementing an interface must include all methods declared in the interface, or an error will occur during compilation.

3.1.2 C#

C#, like Java, is a widely used object oriented language. It uses strong typing, and is intended to be simple, modern, and general-purpose[1]. It is being developed by Microsoft, and was first released in 2000 as part of the development of their .NET framework.

Types

There are two categories of types: Built-in types and custom types. These categories are equivalent to the primitive types and reference types in Java, respectively. Unlike in Java, however, string and object are built-in types in C# rather than custom ones. In C#, all types inherit directly or indirectly from the object type. As such, any value may be assigned to a variable of type object, including the built-in value types like int and bool.

While C# is strongly and statically typed, unlike Java it does support implicit typing through the use of the keyword var, similar to the implicit types of Python. When this keyword is used, the compiler will infer the type of the variable based on the value assigned to it. However, because its typing is static, an implicitly typed variable may not change types after it has been declared, and as such the following code will not compile.

```csharp
var a = "string";
a = 5;
```

Classes and Objects

Declaring a class in C# is relatively similar to how it is done in Java, the main difference being in how inheritance and implementation of interfaces is declared, as illustrated by the following example:

```csharp
public class MyClass : MyParent, MyInterface
{
    public int myNum { get; set }

    public MyClass(int number)
    {
        this.myNum = number;
    }

    public void MyInterfaceMethod()
    {
        \ \ ...
    }
}
```
Instantiating a class is also very similar to how it is done in Java, by using the `new` keyword and providing the appropriate (if any) argument(s) for the constructor to be used:

```csharp
MyClass myObject = new MyClass(number);
```

Like in Java, a C# class may implement any number of interfaces, but can inherit from only one parent class.

Another difference from Java worth noting is the `get` and `set` keywords. These function as shorthand for the following code, allowing the variable to be read and written by use of the appropriate methods.

```csharp
private int myNum;
public int MyNum
{
    get
    {
        return this.myNum;
    }
    set
    {
        this.myNum = value;
    }
}
```

In addition to classes, C# also includes a type called struct. Structs are like classes, but are generally smaller collections of variables, often without methods of their own. A struct may implement an interface, but cannot inherit from other structs or classes. Otherwise, their declaration is identical to that of a class with regards to variables and constructors, with the exception that defining a default constructor with no parameters will cause an error. It is also possible to instantiate a new struct without the use of the `new` keyword, but all variables will be unassigned, and all of these must be initialised before the struct can be used.

**Interfaces**

Like in Java, interfaces are useful for implementing functionality from several different sources as multiple inheritance is not supported. Additionally, interfaces can be used to simulate inheritance in structs. The declaration of the interface itself is practically identical to its Java equivalent, though interfaces in C# may not implement static members. Below is a demonstration of what the declaration of the Interface used in MyClass would look like.

```csharp
interface MyInterface
{
    public void MyInterfaceMethod();
}
```
3.2 Dynamically Typed Languages

Whereas statically typed languages have rigid rules for checking a program's type safety during compile time, dynamically typed languages take a more interpretive approach and delay type checks until runtime. This strategy is often called "Better to ask Forgiveness than Permission," and in many cases duck typing is used, where an object's suitability for a given task is only checked when the task is to be performed, and the test passes as long as the object can actually perform the function specified, leaving the specific type of the object irrelevant.

3.2.1 Python

Python is a dynamic language designed with readability in mind, and with syntax that allows for programs to express concepts with fewer lines of code than what is possible in many other popular programming languages. It was developed principally by Guido van Rossum, and was first released in 1991.[12]

Types

Being a dynamic language, all objects in Python have a type, which is stored with the object itself and used whenever an operation is to be performed on that object. Built-in types include int, string, boolean, etc. When creating a new object, the object’s type is the name of the class. However, the class itself is also an object, whose type is 'type'. In python, everything is an object, including the built-in types, but also functions and methods.

Classes

Python, unlike the statically typed languages described previously, does not inherently support interfaces or abstract classes. Instead, it supports proper multiple inheritance, allowing each class to be a subclass to more than one superclass. There exist libraries to add functionality for abstract classes, but the multiple inheritance combined with duck typing makes both concepts largely superfluous.

```python
class MyClass(MyParent):
    def __init__(self, number)
        self.myNum = number
```

The above example, which illustrates how to define a class and its constructor in Python, shows that despite its many differences from languages like Java and C#, certain things remain quite similar. The greatest difference is the class variable, which is declared within the constructor rather than outside the methods of the class. This variable, like any other in Python, does not have an explicit type, but is rather given one based on the variable assigned to it through the argument number. This may be an integer, but without type checks in the code, any type of
argument could conceivably be provided to this class, including classes and functions. Though the constructor looks different in that its name is \texttt{\_\_init\_\_} rather than the same as the class name, this is merely a convention to avoid accidentally overriding any of the special methods.

Instantiating a class in Python is very similar to how it is done in Java and C#, with the aforementioned exception that the type of the object is implicit and thus unnecessary in the variable declaration. Despite the constructor in the example above having two parameters, the call to create an object needs only one. This is because the first argument is the object itself, which is supplied by the function call, as demonstrated below.

\texttt{myObject = MyClass(number)}

### 3.2.2 JavaScript

JavaScript is an untyped language, and the most popular language for World Wide Web content. All modern browsers support it without plugins, and most websites employ it to some extent. It was first released in 1995.[8]

**Classes**

Because of how the typelessness of JavaScript works, it does not technically implement classes. The keyword \texttt{class} does exist, but is not strictly necessary, and is considered syntactical sugar. Instead, objects in JavaScript are based on prototyping. This means that each object can be created directly using an object initialiser, and new objects of the same type can then be created simply by calling the \texttt{Create} function with that object as its parameter.

Each object holds a reference to its prototype, which is used similarly to superclasses in other object-oriented languages. As opposed to in other object oriented languages like Java, an object does not itself own the methods it inherits unless explicitly overridden. Instead, a call to an inherited method will traverse up along the prototype chain until the method is found or the null object is reached, at which point the call will simply return "undefined."

There are several ways to instantiate objects in JavaScript. The most common one, and the one used in this project, is by using functions as illustrated in the following example:

\begin{verbatim}
function MyClass(number) {
   MyParent.apply(this, arguments);
   this.myNum = number;
}
MyClass.prototype = new MyParent();

Instances of this "class" are then created much like objects in the other languages:

\texttt{var myObject = new MyClass(number);}
\end{verbatim}
The function MyClass returns an object with a variable named myNum, and whose prototype is an object of type MyParent. This achieves results comparable to those of class declarations in other languages. Inheritance is not quite as straightforward, as it is objects that inherit from other objects, rather than uninstantiated classes. This necessitates the creation of a new object of the parent type, to be used as the prototype for the child object.

If so desired, the keyword class may be used. Though it still does not create an actual class like in other languages, in practice the functionality is similar enough as to make them interchangeable. The following is an example of how to use the class keyword in JavaScript:

```javascript
class MyClass extends MyParent {
    constructor(number) {
        this.myNum = number;
    }
}
```

3.3 Summary

For the purposes of this study, we have elected to distinguish languages by whether their typing system is static or dynamic. Two languages from each category were chosen; the statically typed languages Java and C#, and the dynamically typed languages Python and JavaScript. The patterns (as introduced in chapter 2) were implemented and compared across these four languages.
Part III

Comparisons
Chapter 4

Patterns

This chapter details each comparison, in the order by which they were performed. Each section lists the pattern examined with a short general description of the program written for the comparison, what results were expected from the comparison, the actual results with a description of the program in each language examined, and finally a discussion of these results and their significance.

To compare the patterns, an example program was written for each case. This program was modelled after the maze game example as described in Design Patterns: Elements of Reusable Object-Oriented Software[5], but implemented differently in a few key aspects. Instead of building the maze by having each room and door hold references to each other, rooms and walls are laid out on a 2D array of maze component objects, of which both Room and Wall are subclasses. This way, if the maze game were to be written to allow a player to traverse it, a player’s position would be stored as a set of coordinates rather than a direct reference to a room. Upon moving to an adjacent tile, the program would check the target coordinates and determine whether the tile is solid or not, and only move the player if said tile is not solid. Usually this would only apply to rooms, but this opens for the possibility of having variant types of walls which can be broken down or otherwise destroyed. As variant subclasses are an important feature with regards to many design patterns, particularly the creational ones, this implementation makes the program more flexible, more easily facilitating the inclusion of these patterns.

The full code for each of the programs written may be accessed online at the following address:
https://github.com/vebjorrs/pattern-comparisons

4.1 Factory Method

To implement this pattern, a base class MazeGame was created, containing the factory methods makeMaze, makeRoom, and makeWall, as well as a method for generating the maze. In this case, the createMaze method is used to fill the grid with walls, then rooms are created in an H-shape. This method does not need to be changed in subclasses unless desired, as it
uses the factory methods to create the rooms, walls, and the maze itself rather than instantiating them directly. The walls and rooms created by MazeGame are simple, both implementing the interface MazeComponent which supports the methods isSolid and getSymbol.

The subclass of MazeGame used in this comparison, CoinMaze, is identical to its parent, with the exception that it overrides the makeRoom method, causing it to return a CoinRoom rather than a regular Room. CoinRoom extends Room, adding a 25% chance that the room contains a coin. If the room does contain a coin, its getSymbol method will return a different character than what a regular Room or a CoinRoom without a coin would.

4.1.1 Expectations

The overall complexity of the program was expected to be low in all languages tested, as the focus of the pattern is on inheritance and how the factory method in question is implemented in the subclasses. Runtime was expected to be irrelevant, as the specific method to be executed and objects to be created is determined based on whether it is MazeGame or one of its subclasses that is instantiated. None of the unused subclasses affect the runtime, and as such one can have an arbitrary number of classes without the execution time being affected in any meaningful way.

In Java and C#, this pattern was expected to be simple to implement, and to result in reduced duplicate code and higher readability. Some problems were expected in Python and JavaScript due to their lack of explicit typing.

4.1.2 Results

Java

The pattern proved trivial to implement in Java, supported by the language’s polymorphism and mechanisms such as interfaces. The methods and classes worked as expected, and the resulting code was clean and manageable.

The only change necessary to implement the CoinMaze variant was to create a subclass of MazeGame called CoinMaze, which only overrode the factory method makeRoom(). In the parent class, this method looked like the following:

```java
protected Room makeRoom() {
    return new Room();
}
```

As such, the entirety of the CoinMaze class was as follows:
The program written in C# proved nearly identical to its Java counterpart. The biggest difference was in how methods are overridden, in that the MakeRoom method in the parent class had to be virtual, and the same method in the child had to be explicitly stated to override its parent method.

```csharp
class MazeGame
{
    public virtual Room MakeRoom()
    {
        return new Room();
    }
    {...} \ Remaining methods
}

class CoinMaze : MazeGame
{
    public override Room MakeRoom()
    {
        return new CoinRoom();
    }
}
```

Python

As Python does not inherently support interfaces or abstract classes, it was necessary to import libraries that would support such features. This led to an increased complexity. The reason for this is that the problem meant to be addressed by the Factory Method pattern does not occur in this language to the extent it does in strongly typed languages, and in fact, forcing the use of this pattern by using the isinstance function resulted in not only a higher total line count, but also a longer runtime, as this function is relatively expensive.
The implementation of the method itself proved similar to Java, and with fewer constraints and less focus on abstract classes and interfaces, certain aspects of the pattern may be useful in this language as well. The CoinMaze class itself, for instance, ended up very similar to the one in Java, only requiring a couple lines to override the makeRoom method:

```python
class CoinMaze(MazeGame):
    def makeRoom(self):
        return CoinRoom()
```

As mentioned, the main problem with this implementation was the abstract classes and class hierarchy. In contrast to the Java implementation, in which the interface only required three lines of code for the interface signature plus the declaration of the isSolid and getSymbol methods, the abstract class used in python required twice as many lines to achieve the same results:

```python
class MazeComponent:
    __metaclass__ = ABCMeta

    @abstractmethod
def isSolid(self): pass

    @abstractmethod
def getSymbol(self): pass
```

Additionally, to be able to enforce the constraint that walls and rooms have to implement (or in this case, extend) MazeComponent, it was necessary to use isinstance in the addComponent method:

```python
def addComponent(self, component, x, y):
    if isinstance(component, MazeComponent):
        self.grid[x][y] = component
```

This, being an expensive method, increased the runtime of the program directly proportional to the size of the maze, as every component had to be checked.

A second version of this program was made, forgoing the use of abstract classes and instead relying on Python’s duck typing. In this second iteration, the Room and Wall classes are separate from one another with no common parent class. Additionally, instead of using the isinstance function, try/except blocks were used to ensure that the objects to be used implemented the right methods. This resulted in a significant decrease in runtime, and lowered the overall complexity of the program.

**JavaScript**

JavaScript turned out to be somewhat of a middle ground between Java and Python. Implementation was simple, and the code ended up very similar to the Java implementation. However, as it does not support actual classes, JavaScript does not inherently support abstract classes or interfaces. Similarly to Python, enforcing this leads to unnecessary complexity, and the use of the instanceof function.
Due to the way inheritance works in JavaScript, with objects inheriting from other objects rather than from class to class, the CoinMaze "class" is more complicated than its Java, C#, and Python counterparts:

```javascript
function CoinMaze() {
  MazeGame.apply(this, arguments);

  this.makeRoom = function() {
    return new CoinRoom();
  }
}
CoinMaze.prototype = new MazeGame();
```

In this implementation, like in the second version of the Python program, the MazeComponent interface was not used at all. CoinRoom extends Room, but Room and Wall are separate classes with no common parent. Like Python, JavaScript does not lend itself well to abstract classes, but its duck typing makes such abstract parents superfluous. Attempting to force this approach in JavaScript would, like in Python, result in higher complexity and more lines of code for no noticeable benefit.

### 4.1.3 Discussion

In all of the examined languages, the Factory Method proved to be useful in one way or another. There were some issues with implementation in Python and JavaScript, but this was chiefly due to attempting to implement the pattern in the same way across all languages. When a different approach was taken for the dynamically typed languages which more closely followed their design philosophy, the pattern became simpler to implement and the code became less complex. In conclusion, factory methods as defined by the Gang of Four [5, p. 107] may be more straightforward to implement in statically typed languages, but its general idea can prove useful in other object oriented languages as well.

As a side note, the implementations of the program were more similar than anticipated in Java and C#. The results suggest that the main factor in the applicability and ease of implementation of the pattern was whether the language in question was statically typed or not, but these results might have been different had another statically typed language been chosen instead of C#.
4.2 Abstract Factory

The need to include this pattern was debated, as it is very similar to the Factory Method pattern to the point where modern catalogues of design patterns sometimes list them as one and the same, simply called the Factory pattern. However, it was determined to be prudent to include it for the sake of thoroughness. The programs written for this comparison closely match those from the Factory Method pattern, the main difference being that instead of having the MazeGame class be responsible for the details of instantiating components of the maze, this work is delegated to a separate MazeFactory class. Thus, instead of creating subclasses of the potentially large MazeGame class, the MazeFactory can be extended instead when specialisation is needed.

4.2.1 Expectations

Due to the similarities between Abstract Factory and Factory Method, the results were expected to be similar to those. The same problems with duck typing encountered earlier were also expected to show up again.

4.2.2 Results

Java

Much of the code proved identical to that from the first comparison, the main difference being the inclusion of the MazeFactory class:

```java
class MazeFactory {
    Maze makeMaze(int x, int y) {
        return new Maze(x, y);
    }

    Room makeRoom() {
        return new Room();
    }

    Wall makeWall() {
        return new Wall();
    }
}
```

Moving these three methods to a separate class increased the overall number of lines of code slightly, but resulted in each class being less cluttered, and as such more maintainable. This change also meant that to implement the CoinMaze, only MazeFactory would need to be extended rather than the MazeGame class itself. The CoinFactory looked nearly identical to the CoinMaze class from before, but as its parent class only contained methods relevant to the instantiation of maze components, it did not extend irrelevant functionality such as the createMaze method.
class CoinFactory extends MazeFactory {
    public Room makeRoom() {
        return new CoinRoom();
    }
}

C#

As in the first comparison, C# was too similar to Java to have any noticeable differences in the implementation of the pattern. The results were practically identical as the language mechanisms used are the same across these two languages.

Python

This program ran into the exact same problems as in the previous comparison with regards to inheritance, though as before these problems were mostly avoided by relying on duck typing rather than strictly enforcing class hierarchies. Like in Java, the biggest difference from the Factory Method comparison was the MazeFactory class, which served to separate the methods into more distinct groups.

class MazeFactory:
    def makeMaze(self, x, y):
        return Maze(x, y)

    def makeRoom(self):
        return Room()

    def makeWall(self):
        return Wall()

This decentralisation meant that the only method remaining in the MazeGame class was createMaze, which now took a MazeFactory object as an argument, using it to instantiate the components it used to build the maze itself. All in all, this made for more manageable, more easily maintainable code.

def createMaze(self, mazeFactory):
    m = mazeFactory.makeMaze(width, height)

    for x in range(width):
        for y in range(height):
            m.addComponent(
                mazeFactory.makeWall(),
                x, y
            )

    m.addComponent(mazeFactory.makeRoom(), 1, 2)
    (...
    m.addComponent(mazeFactory.makeRoom(), 3, 3)

    return m
**JavaScript**

Because of the complete lack of typing in JavaScript, the createMaze function could theoretically accept any object as an argument, whether or not it actually supports the functions required of it. One could solve this problem by using try and catch blocks, but calls to the functions of MazeFactory are made throughout the createMaze function, and having an entire function encapsulated by a single try/catch block is generally considered poor design.

```javascript
if (!(mf instanceof MazeFactory)) {
    throw new Error("Invalid MazeFactory used!");
}
```

Instead, an instanceof check at the beginning of the function was used, which as previously noted is an expensive and clumsy way of forcing functionality which goes against the language’s underlying philosophy of "better to ask for forgiveness than permission."

### 4.2.3 Discussion

The results of this comparison were very similar to the ones from the Factory Method pattern, though the Abstract Factory pattern may be more suited for dynamically typed languages than Factory Method is. The patterns very closely resembled each other, but with enough differences to justify writing separate comparisons for Abstract Factory in addition to the Factory Method ones.

It was mentioned in the results from the JavaScript example that typeless languages cannot guarantee that the object provided as an argument is of the right type, but this is not necessarily of great relevance. If an object is used which does not support the necessary methods, an exception will be thrown. This is generally something one wants to avoid in languages like Java and C#, but in the end it is the programmer’s responsibility to avoid them. As long as the code is used properly, and the end user never experiences an exception, there is no issue.
4.3 Strategy

This comparison used a MazeGame program similar to the one used in the comparisons of the Factory Method and Abstract Factory patterns, but instead of letting the MazeGame class be responsible for the generation of the maze, a MazeGenerator class was implemented and added as an instance variable to the MazeGame class. The MazeGenerator is an abstract class which contains the makeWall and makeRoom methods, as well as an abstract generate method which takes the dimensions of the maze to be created as arguments and returns a fully generated Maze object. Two subclasses were implemented for this generator; LineGenerator and BoxGenerator. When the generate method is called, these two subclasses create a maze containing a line of rooms running through the middle of it and a maze with rooms along its outer walls, respectively. The specific generator to be used is determined by MazeGame's constructor, which instantiates and stores the generator corresponding to the string provided as an argument.

4.3.1 Expectations

As was determined in the previous comparisons, runtime may be a significant factor in languages that employ duck typing. As before, complexity and number of lines of code were also expected to differ greatly based on how well the given language supports the mechanisms used. As before, Java and C# were expected to be the most fitting languages for this pattern, with Python and JavaScript less suited due to their typing and lack of inherent support for abstract classes.

4.3.2 Results

Java

As expected, the Strategy pattern was simple to implement and run in Java, with good efficiency, low number of lines of code, and manageable complexity. The abstract MazeGenerator ended up looking similar to the MazeGame class in the Factory Method implementation:

```
abstract class MazeGenerator {
    abstract Maze generate(int x, int y);

    protected Wall makeWall() {
        return new Wall();
    }

    protected Room makeRoom() {
        return new Room();
    }
}
```

This decentralisation of methods causes MazeGame itself to be more concise and thus more easily understandable and maintainable. It still has
the createMaze method, but this only contains a call to the generate method in the MazeGenerator:

```java
public Maze createMaze(int x, int y) {
    if (mazeGen != null) {
        return mazeGen.generate(x, y);
    }
    return null;
}
```

C#

Like in the previous comparisons, C# proved to be so similar to Java as to be almost indistinguishable, with comparable complexity and a number of lines of code that is higher only due to the coding conventions of the language. To illustrate, below is the abstract MazeGenerator class as it appears in C#.

```csharp
abstract class MazeGenerator
{
    public abstract Maze generate(int x, int y);

    protected Wall makeWall()
    {
        return new Wall();
    }

    protected Room makeRoom()
    {
        return new Room();
    }
}
```

Python

Python was again ill suited for implementing the pattern, requiring libraries to be imported and resulting in an increased number of lines of code. Like in the first comparison, a version was first made which conformed to the letter of the design pattern. Like before, this resulted in an unwanted amount of complexity, as well as an increased runtime. Like in Java, the MazeGenerator class ended up looking similar to the MazeGame class from the Factory Method comparison.
class MazeGenerator:
    __metaclass__ = ABCMeta

    @abstractmethod
def generate(self, x, y): pass

def makeWall(self):
    return Wall()

def makeRoom(self):
    return Room()

In this comparison like in the first, a second iteration of the Python program was written which more effectively utilised the language’s duck typing and inherent strengths. This reduced the runtime by a fair amount and the complexity somewhat, though the importing of abstract class functionality was still necessary for the sake of the MazeGenerator.

JavaScript

Like in the comparisons of the Factory Method and Abstract Factory patterns, the lack of support for classes (and by extension abstract classes) made enforcing the use of interfaces problematic. To implement an "abstract class" in JavaScript, the base object was created in such a way that attempting to instantiate it would cause an exception:

MazeGenerator = function() { throw "Abstract class!" }

To ensure that the generate function was also abstract, so that any concrete child object must override it, its implementation in the MazeGenerator object also needed to throw an exception if called:

MazeGenerator.prototype.generate = function(x, y) {
    throw "Abstract function not overridden by child!";
}

This implementation was overly complex, and had the highest number of lines of code of any program so far.

4.3.3 Discussion

This pattern, like the others before it, proved more straightforward in its implementation in the statically typed languages than in the dynamically typed ones. The Strategy pattern, while being behavioural rather than creational, employs many of the same mechanisms as the previously examined patterns. As such, it was to be expected that it would have many of the same results, though this implies that the difference in static vs dynamic typing in languages has an effect on not just creational patterns, but possibly both of the other categories as well.

One notable feature of Python and JavaScript is that their functions are treated as first-class objects. This means that in these languages, functions may be passed as arguments, returned from other functions, and assigned
to variables. This functionality makes the Strategy pattern as defined in Design Patterns: Elements of Reusable Object-Oriented Software[5] effectively superfluous, as the createMaze function could take a strategy function as an argument directly, with no need for any objects to contain them, much less a class hierarchy. However, this does require the program to trust the client to provide correct functions which do what they are supposed to do.

4.4 Prototype

The MazeGame example was used, with the inclusion of a MazePrototype-Factory object which controlled the prototypes for the Maze, Room, and Wall classes. The original objects are supplied to the factory as parameters in its constructor, and copies are then retrieved using the makeMaze, makeRoom, and makeWall methods, all of which call their respective prototype’s clone method. These clone methods, in turn, create new copies of themselves by creating new objects using themselves as the parameter for their constructor. The constructor makes a deep copy of the provided object.

4.4.1 Expectations

The mechanisms and features of the statically typed languages were, as with the other patterns, expected to be well suited for the implementation of prototypes. However, some problems were anticipated in the implementation of deep copying, particularly in Java as in many cases it copies by reference rather than by value. In fact, the javadoc for the default clone method supported by all objects[10] states that the precise meaning of "copy" varies from object to object. The intent is that the copy, though not the exact same object as the original, is of the same class and that x.clone().equals(x) returns true, but none of these are absolute requirements and cannot be assumed to be true for any given object.

As opposed to Java, Python’s libraries include a deepcopy method in the copy module, which simplifies the process of cloning objects. For this reason, the pattern was expected to be simpler to implement in Python than in Java. However, Python gives the client direct access to any object’s class object, which means that creating a new object directly can potentially be done much more efficiently than through cloning an existing object by calling the object’s __class__ method. This allows the client the flexibility of not needing to know the exact class of the object to be created, which is one of the benefits of prototypes.

JavaScript, unlike the other languages, does not support classes. Instead, every object is created from prototypes defined in the code. As such, implementing prototypes in JavaScript would only lead to redundant code.
4.4.2 Results

Java

Most of the implementation proved relatively straightforward. The classes that followed this pattern were the MazeComponent classes (Wall, Room, and their subclasses) and the Maze class, though only Maze benefited from the pattern. The classes Wall and Room had no intrinsic values to be copied over, resulting in a clone method which simply acted as an additional layer of indirection for instantiation of new objects:

```java
class Room implements MazeComponent, Cloneable {
    public Room clone() {
        return new Room();
    }

    (...) // Remaining methods
}
```

The CoinRoom subclass did contain a class variable, but this was the boolean which determined whether the room contained a coin or not, which needed to be random for each room. This necessitated a clone method which was similarly useless.

```java
class CoinRoom extends Room {
    boolean coin;

    public CoinRoom() {
        Random r = new Random();
        if (r.nextInt(100) >= 75) {
            coin = true;
        }
    }

    public CoinRoom clone() {
        return new CoinRoom();
    }

    (...) // Remaining methods
}
```

If desired, two prototypes could be made for the CoinRoom; one with a coin and one without. Then, the prototype to be used could be determined in the prototype factory, but this would have taken control away from the CoinRoom itself, and lead to poor maintainability.

The Maze class was the only one to require two separate constructors. One was used when the prototype was instantiated, and set the original maze up. The other took another Maze object as a parameter, and created a copy of the original's map.
class Maze implements Cloneable {
    private MazeComponent[][] map;
    private int length;
    private int height;

    public Maze(int x, int y) {
        map = new MazeComponent[x][y];
        length = x;
        height = y;
    }

    private Maze(Maze other) {
        length = other.getLength();
        height = other.getHeight();
        map = new MazeComponent[length][height];
        (...); // Loop through each component
        // of other’s map and clone them
    }

    public Maze clone() {
        return new Maze(this);
    }
    (...); // Remaining methods
}

Unfortunately, as each MazeGame only requires one Maze object, the implementation of this pattern only resulted in more objects being instantiated than what was necessary, and each object instantiation having a higher overhead time than it otherwise would.

C#

Like in Java, the clone method of the Maze class was heavy and expensive, whereas the clone methods of Room, CoinRoom, and Wall were practically useless. One difference between C# and Java was in the return type of the clone method. In the interface, the return type is specified as MazeComponent, like in Java. However, in the Java program, the clone method of for instance the Room class could successfully return a Room object, as shown:

public Room clone() {
    return new Room();
}

However, in C#, the methods must have the exact same signature as the interface. As such, the clone method ends up looking like this:

public virtual MazeComponent clone()
{
    return new Room();
}
For this reason, the makeRoom and makeWall methods in the MazePrototypeFactory class must either also be generalised to return MazeComponent objects, or must be cast explicitly to their respective implementing classes.

Python

The inclusion of the deepcopy method from the copy module ensured that implementing this pattern was simple. Even the Maze class, which required an additional constructor and manual copying of each element in Java, was easily cloneable with only a few lines of code:

```python
class Maze:
    def __init__(self, x, y):
        self.length = x
        self.height = y
        self.grid = []

        (...) # loop through the grid,
        # creating an empty 2D array

    def clone(self):
        return copy.deepcopy(self)
```

To make CoinRoom more compatible with the Prototype pattern, the way to determine whether a given room contains a coin or not was changed. Instead of being chosen in the constructor, a separate method was created which was only called from its clone method, so that the prototype would never contain a coin, but any of its clones had the normal 25% chance of containing one.

```python
class CoinRoom(Room):
    def __init__(self):
        self.coin = False

    def determineCoin(self):
        if random.randint(0, 100) >= 75:
            self.coin = True

    def clone(self):
        c = copy.deepcopy(self)
        c.determineCoin()
        return c
```

Due to the deepcopy method, the implementation of this pattern proved much simpler in Python than in Java. Despite this, however, Python’s built-in mechanisms allows one to copy classes much more efficiently than with the Prototype patern. In the first iteration of the Python program for this comparison, the makeMaze, makeRoom, and makeWall methods of the MazePrototypeFactory looked like this:
def makeMaze(self):
    return self.prototypeMaze.clone()

def makeRoom(self):
    return self.prototypeRoom.clone()

def makeWall(self):
    return self.prototypeWall.clone()

However, in the second iteration, it was changed to create new objects
directly based on the prototypes' class objects:

def makeMaze(self, x, y):
    return self.prototypeMaze.__class__(x, y)

def makeRoom(self):
    return self.prototypeRoom.__class__()

def makeWall(self):
    return self.prototypeWall.__class__()

This approach, in addition to using fewer lines of code by virtue of
not requiring a clone method for each class, improved the runtime in this
example by an order of magnitude. For a 5 tiles by 5 tiles maze, the runtime
of the first iteration averaged 18.50 milliseconds, whereas that of the second
iteration averaged only 1.88.

4.4.3 Discussion

This pattern provided an example of a pattern not necessarily being more
difficult to implement in the dynamically typed languages, but which is
nonetheless more useful in the strongly typed ones due to the mechanisms
and features of Python and JavaScript allowing for more efficient means to
achieve similar results.

No programs were written in JavaScript for this comparison, as a
program implementing the Prototype pattern would look identical to a
program not written to implement any particular patterns at all.

function Room() {
    this.isSolid = function() {
        return false;
    }

    this.getSymbol = function() {
        return '.';
    }
}

Using the Room class (as illustrated above) as an example, when a
"class" is defined in JavaScript, it is stored as a function with the same name
as the class. When this function is called, it returns a copy of the object
described in the function. As this is precisely what the Prototype pattern
accomplishes, an explicit implementation of the pattern in this language
would be unnecessary.
As for Python, accessing the object’s class name does not perform the exact same function as the Prototype pattern, but it does accomplish an equivalent amount of flexibility in that one does not need to know the name of the class to be copied, and new objects can be added to the catalogue arbitrarily. As this leads to a lower total runtime than using the Prototype pattern would, in most cases this approach may be preferable to the Prototype pattern.

4.5 Flyweight

The goal of this pattern is to allow for an arbitrary number of instances of a class without requiring numerous identical objects. As such, a MazeGame program was written for each language wherein only one Wall object and one Room object existed, the same object instead having repeat representations across the two-dimensional array representing the maze itself. As the base MazeGame used in previous examples has no extrinsic values, the CoinMaze was used in order to take advantage of this aspect of the pattern. As every representation of the CoinRoom is the same object, the object itself cannot contain information as to whether or not the room has a coin. Instead, this information must be calculated based on the coordinates of the room whenever the value is to be checked.

4.5.1 Expectations

This pattern was not expected to pose a problem for any of the languages tested. Though Python and JavaScript had caused issues with the implementation of other patterns previously, this pattern in particular did not involve interfaces, abstract classes, or class hierarchies to the degree most of the others had. For this reason, all the languages were expected to easily accommodate the implementation of the Flyweight pattern.

4.5.2 Results

Java

Rather than using an interface for the maze components as had been done in previous examples, this version of the program used only a single MazeComponent class, using an enumeration to determine what specific type of component it was.

```java
public enum ComponentType {
    WALL, ROOM, COINROOM
}
```

This ensured that the flyweight had both an intrinsic state and an extrinsic one; Each representation of a room was the same MazeComponent object with its type set to ComponentType.COINROOM, whereas each piece of wall was a single MazeComponent object with ComponentType.WALL set. The isSolid method returned true or false based on this intrinsic state,
and in the case of walls and normal rooms, so did the getSymbol method. However, with coin rooms, context was required in order to know whether a given room would contain a coin or not. The MazeContext object contained the seed which was used in the creation of the Maze, and the current position, flattened to a one-dimensional index. Using these two values, the same result could consistently be given for the same room in the same location:

```java
public boolean hasCoin() {
    Random r = new Random(seed*index);
    if (r.nextInt(100) >= 75) {
        return true;
    } else {
        return false;
    }
}
```

The getSymbol method would then return a symbol based on the return value of this hasCoin method.

The ComponentFactory itself contained an array of components. When the createComponent method was called, it would check the array for the object being asked for. If no such object already existed, it would first create it and place it in the array, before returning the corresponding object from the array.

```java
class ComponentFactory {
    private MazeComponent[] flyweight;

    public ComponentFactory() {
        flyweight = new MazeComponent[
            ComponentType.values().length];
    }

    public MazeComponent createComponent(
        ComponentType ct) {
        if (null == flyweight[ct.ordinal()]) {
            flyweight[ct.ordinal()] =
                new MazeComponent(ct);
        }

        return flyweight[ct.ordinal()]}
}
```

Overall, the implementation of this pattern proved unproblematic. The complexity and number of lines of code were comparatively high, but reducing the number of objects from arbitrarily high to simply two, resulted in a significantly reduced amount of memory taken up by the program.

C#

Like in the other comparisons, Java and C# had nearly identical implementations. The biggest difference in this case was the implementation of the
enumerations. C# allows for easier access to the value of each enumeration, a detail which is more abstract in Java. In C#, one simply needs to explicitly cast the enumeration to an integer in order to access its value, whereas Java requires the use of the ordinal method.

```csharp
public MazeComponent CreateComponent(ComponentType ct)
{
    if (null == flyweights[(int) ct])
    {
        flyweights[(int) ct] = new MazeComponent(ct);
    }
    return flyweights[(int) ct];
}
```

**Python**

This implementation was almost surprisingly straightforward, compared to the other patterns that had been examined so far. The mechanisms of the language supported the pattern well, and no overly complex classes or functions were required. In fact, due to Python’s dictionaries, certain parts of the program ended up being less complex than Java and C# which used enumerations for the same purpose:

```python
class ComponentFactory:
    def __init__(self):
        self.flyweights = {}

    def createComponent(self, ct):
        if ct not in self.flyweights:
            self.flyweights[ct] = MazeComponent(ct)

        return self.flyweights[ct]
```

Aside from this, the program looked and ran similarly to those written in Java and C#, gaining the same advantage in reduced memory load as in the other languages.

**JavaScript**

As JavaScript, like Python, inherently supports dictionaries, implementing flyweights in this language resulted in a program of equal complexity and comparable number of lines of code. The main issue with this implementation was the fact that JavaScript does not support seeds for the random number generator provided with the language. For this reason, it was necessary to implement a custom random number generator[3]:

```javascript
47
```
This, however, is only tangentially related to the implementation of the pattern itself, and in other programs whose objects do not rely on randomness this difference between Python and JavaScript will not be evident. That said, for any object that does employ randomness in one or more forms, it will be necessary to work around this shortcoming.

4.5.3 Discussion

This comparison presented a departure from the previous ones, as the implementation was similarly straightforward and efficient in all four languages, though with lower complexity in the dynamically typed languages due to their dictionaries. As Flyweight is not a creational pattern, this suggests that the difference in ease of implementation between statically and dynamically typed languages is unique to the creational patterns. However, as was determined previously, this difference is apparent in the Strategy pattern as well, which is behavioural. The greatest difference between this pattern and the others examined so far is that this one does not depend upon class hierarchies or polymorphism. Based on this, one may conclude that this is the deciding factor that determines whether a pattern is more suitable in statically or dynamically typed languages. As all the creational patterns examined heavily depend on subclasses, the assumption that this is true for all creational patterns, including the Builder and Singleton patterns which were not examined for this study, cannot be made in confidence.

4.6 Discussion and Conclusion

Based on these comparisons, the main deciding factor when it comes to the suitability of a given pattern is whether the language in question utilises static or dynamic typing. Especially in patterns where class hierarchies played a major part, the dynamically typed languages either did not accommodate the pattern easily, requiring the importing of additional libraries, or they simply circumvented the need for the pattern in the first place. Conversely, in the one pattern examined which did not heavily rely on class hierarchies, the implementation proved simpler and more efficient in the dynamically typed languages.

Table 4.1 on page 49 showcases the ease with which each pattern was implemented across the two categories of languages examined. Due to their similar execution and results, the two factory patterns were merged.
for this table. A plus symbol signifies that the pattern was relatively simple to implement and efficient in its execution. A minus signifies the opposite; that implementing the pattern was problematic, and/or that its execution was inefficient. Lastly, a N/A denotes that the pattern was irrelevant or superfluous in the corresponding language category.

These results may have been influenced negatively by the choice of languages. As Java and C# are very similar in most regards, this may not be an accurate representation of statically typed languages as a whole. Had another statically typed language been used for the comparisons, such as C++, the results may have been different. There is no guarantee for this, however, as Python and JavaScript are relatively distinct languages, yet the results were generally uniform between the two, the same way the results were similar across Java and C#.

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factory</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Strategy</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Prototype</td>
<td>+</td>
<td>N/A</td>
</tr>
<tr>
<td>Flyweight</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table 4.1: Ease of implementation for each pattern
Part IV

Conclusion
Chapter 5

Conclusion

For this study, several simple programs were written in various commonly used programming languages, implementing some of the more common design patterns and comparing them across languages to determine differences in efficiency and ease of implementation.

The main distinguishing feature when deciding what patterns to use, appears to be whether the language is dynamically or statically typed. In the patterns that made heavy use of abstract classes and interfaces, the statically typed languages proved far more straightforward and efficient. By contrast, in the patterns that did not rely as much on these mechanisms, the results were more uniform across languages, though the dynamically typed languages as a whole required fewer lines of code and had higher overall legibility.

In dynamically typed languages, objects do not have to be explicitly typed, which alleviates the need for rigid inheritance trees. Because of the so-called duck typing, also known as EAFP (easier to ask forgiveness than permission) philosophy of dynamic languages, the interface of an object is not checked until each individual method is required. If at that time the method does not exist, it will simply throw an exception. Static languages, however, rely on a predefined interface which is checked at compile time. This makes patterns like Abstract Factory and Prototype more suitable in these languages than in the dynamic ones.
Bibliography


