UiO : Department of Informatics
University of Oslo

How Programming Languages Affect Design Patterns

A comparative study of programming languages and design patterns

Vladislav Georgiev Alfredov

Master's Thesis Autumn 2016

Abstract

This thesis is an empirical study of the affects that Language Features, provided by Programming Languages on Design Pattern implementations. The set of Design Patterns is mostly based on the ones described by first book to document Design Patterns, namely the Gang of Four book [31]. It examines comparatively two Programming Languages, Java and Python. In terms of comparing Language Features and Design Pattern implementations in across the two Programming Languages. The examination is based on the Programming Languages' documentation as well as the literature available with regards to Design Pattern implementations in these Programming Languages. Such cross language study, has not been conducted before and thus it is an approach to the topic of Design Patterns from a novel perspective. The thesis proposes a classification scheme with regards to Language Features in relation to Design Pattern. In addition to that it proposes a categorization scheme of Language Features, that implement Design Patterns in Programming Languages. It also assesses the dependence of Design Patterns on specific Language Features.

This thesis also examines some of the observed effects of Design Patterns on Programming Languages, by comparative examination of two languages. It also catalogs related Language Features' development over time. Leading to the definition of the relationship between Design Patterns and Programming Languages as bidirectional. And the theory, explaining the consequences of changes to either Design Patterns or Programming Languages.

Contents

List of Figures

List of Tables

Preface

I want to thank my supervisor Eric Bartley Jul for valuable guidance, encouraging comments and great advises.

I would also like to thank my family, friends for being there and supporting me.

Part I

Introduction

Chapter 1

Introduction

This thesis is a comparative study over a subset of the design patterns, over a selection of Programming Languages. Here, comparisons between the selected Design Patterns and comparisons between the languages can be found. The Design Patterns discussed in this thesis are chosen for their distinct differences or similarities across the languages and relevance to Design Pattern implementations.

1.1 Motivation

Programming Languages are not all equivalent or equal. If they were then there would be no point of the existence of more than one Programming Language. As such they are usually meant to solve particular set of problems. The idea that Programming Languages are not equal and that some languages are better at writing programs that solve some problems than other languages as described in the essay by Paul Graham [64]. Even though General Purpose Languages aim to be broadly applicable across domains, solving problems using a particular language might be easier than using others. That is due to the combination of features, mechanisms, syntax and programming paradigms that the language provides versus the ones that other languages provide their users. Thus the user of the language could leverage these language features to solve the problem.

The above mentioned idea could be traced to the fact that some languages directly implement some design Patterns, or provide features that are substitutes for certain design Patterns. Some examples of which have been discussed in this thesis.

Given the fact that Design Patterns have existed for around 22 years, since their definition by the GoF book [31], and the fact that Programming Languages continuously evolve over the years to support their users' needs, questioning how the Programming Languages affect Design Pattern and in term interact with each other becomes of interest. The main focus of this master thesis is the effects of Programming Languages on

Design Pattern implementations in these languages, trough the Language Features that the Programming Languages implement. The method of investigation of choice is in the form of a comparative investigation of Programming Languages, the Language Features they provide and the Design Pattern implementations they affect.

1.2 Background and Definitions of Therms

This section clarifies some of the therms used throughout the thesis. Further detailed theoretical background analysis, can be found in Chapter 2.

1.2.1 Programming Language

It is worth taking the time to clarify what exactly is meant, by Programming Language. Some could think of a Programming Language as of the Base Language according to the language specification. In other words as the compiler understands it. Others might understand it as the Full Language or the Base Language plus all the standard libraries that come with the said language interpretation. Throughout this thesis Programming Language refers to the Full Language. The Base Language according to the language specification and the standard library that comes included in the default or the "De facto" language implementation. Also on the topic of compilers and interpreters, only the most commonly used, "De facto" compiler or interpreter is considered. Namely for Python this is the cpython implementation [22] and documentation [50]for Java this is the default Oracle implementation [37] and documentation [49]. Also when writing Java or Python, it should be understood that it is meant, the Full Implementation of the language and not just the Base Language specification.

1.2.2 Language Features

In this thesis, Language Features refer to programming principles, language features and language mechanisms. In other words anything that the programming language provides or facilitates that could be relevant to the implementation of design Patternsin that particular programming language.

1.2.3 Design Pattern

Design Patterns have been a popular programming concept since 1994, when the first book on the subject was published by four authors Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Titled Design

Patterns - Elements of Reusable Object-Oriented Software [31] which initiated the concept of Design Pattern in Software development. The four authors are commonly referred to as the "gang of four" or GoF for short and the book in term as the GoF book [31]. Because, the concept has existed for so long, it has had it's fair share of popularity in many different programming languages. Design Patterns rely on the object oriented programming concepts. Because, they utilize inheritance, object composition, aggregation and delegation, as defined in the GoF book [31]. Thus any discussion and comparison over the Design Patterns defined in the GoF book [31], would have to be over a set of Programming Languages, which support Object Oriented Programming (OOP) paradigm. Otherwise the definition of Design Patterns would have to be widened to include patterns such as object, inheritance and other common programming principles, features, mechanisms and parts of Object Oriented Programming Languages. Thus Design Pattern refers to the design patterns such as defined in the Gang of Four (GoF) book [31]. Mostly the general use Design Patterns are of interest and not the specific ones to concurrent programming or distributed programming or any of the special case Design Pattern variants that stem from the GoF ones.

1.2.4 UML Extension

This thesis contains several UML diagrams, in which the relation of type association is used to depict that two classes are associated, as described in the uml-diagrams website [77]. The relationship in question is demonstrated in Figure 1.1.

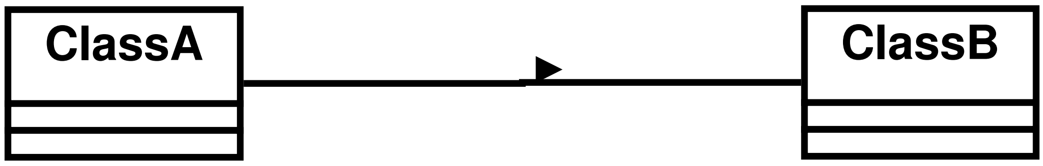

Figure 1.1: UML diagram depicting the Association relationship.

1.3 Approach

The thesis is an empirical study of a selection of Design Patterns, over a selection of Programming Languages, Java and Python. In other words comparative analysis of Design Patterns over different programming languages in order to learn something about the interactions between Design Patterns and Programming Languages.

Examine a set of Languages and a set of Design Patterns, using the documentation and literature on the topics to logically asses the affects of Programming Languages and the Language Features they provide on Design Pattern implementations.

1.4 Goals

The goal of this thesis is to extract useful or meaningful information about the interaction between Design Patterns and Programming Languages. The interaction in terms of the Design Pattern implementations using facilities provided by the language or Language Features, versus Design Patterns being implemented as such Language Features. Thus the result might be better understanding of how the Programming Languages have changed, historically over time, in terms of the Language Features they provide, since the first book on Design Patterns, by GoF [31]. The results might also be used to support or be used against some of the popular theories proposed over the years regarding design patters. Such as the proposal that Design Patterns are actually missing language features [**design_vs_feature**], discussed in Chapter 2.

1.5 Language Choice Criteria

The first book on the topic of Design Patterns, GoF [31], describes Design Patterns in both Smalltalk and C++, which are both Object oriented languages. Thus, much like the the authors of the GoF book [31], the Programming Language selection is restricted to Object Oriented Programming (OOP) languages or programming languages supporting multi paradigm programming in addition to OOP. This choice simplifies the analysis and comparisons of Design Patterns, and distinctly defines a the scope of the Design Patterns selection. As such Java and Python are the two Programming Languages selected for the comparative study and Chapter 3, discusses the Programming Language selection process and criteria in further detail.

1.6 Work Done

The work done is in the form of implementing some Design Patterns in the Programming Languages, based on documentation of the Language Features of these languages and based on the literature, documenting Design Patterns in the Programming Languages of choice. Then comparing the implementations over the Programming Languages, in attempt to assess the impact of Language Features on Design Pattern implementations.

1.7 Relevant Literature

The most relevant literature to the topic, aside from the GoF book [31] and the Programming Languages' documentations, are the "Software Architecture Design Patterns in Java" [42] and "Mastering Python Design Patterns" [40]. Further discussion on the resources related to the topic can be found in Chapter 2.

1.8 Outline

The thesis consists of seven chapters and an appendix, separated in two parts, Introduction and Analysis.

Chapter 1 Provides an overview of the thesis, as well as definitions of the more commonly used terms throughout the thesis.

Chapter 2 Examines the theoretical background related to the topic, resources overview, related work and relevant related theories.

Chapter 3 Describes the Programming Language selection and process.

Chapter 4 Discusses and describes relevant Language Features of the Python Programming Language on their own, from the perspective of the topic and serves as a background for the Evaluation chapter.

Chapter 5 Discusses and describes relevant Language Features of the Java Programming Language on their own, from the perspective of the topic and serves as a background for the Evaluation chapter.

Chapter 6 Evaluates the relevant Design Patterns and Language Features.

Chapter 7 Conclusion Contains the conclusion of this thesis with regards to evaluating the effects of Programming Languages on Design Pattern implementations.

Chapter A Appendix Contains full sample implementations of Design Patterns and Language Features sample uses, relevant to the topic and discussed throughout.

1.9 Results

The result of this thesis is that it confirms that Programming Languages affect Design Pattern implementations trough the Language Features they provide. A Language Feature scoring scheme is established in Section 6.7 with regards to Design Pattern implementation impact. A classification scheme of Language Features that are Design Pattern implementations is proposed in Section 6.9, where two classes of Language Features are established, Invisible and Partial implementations of Design Patterns within the Programming Language. In addition to that as described in Section 6.8, a theory with regards to the adoption mechanism of Design Patterns in Programming Languages is proposed. The relationship between Programming Languages and Design Patterns is defined as bidirectional (Section 6.10), in to defining the "Ripple Effect Theory" (Section 6.11).

1.10 Contributions

A few topics for further study are suggested as a result of writing this thesis, in addition to the classification scheme for Language Features from Section 6.9, the overview of the landscape of Design Pattern implementations of Section 6.6 and the scoring scheme, proposed in Section 6.7. The four topics proposed in the Background Chapter 2: "Examination of effects of frameworks on Design Patterns implementations and use in Programming Languages", "The role of metaprogramming in Design Pattern implementations", "Programming Paradigm effects on Design Patterns", "Modeling versus Typing". In addition to that another interesting topic, related to this thesis is "How Programming Languages affect Anti-patterns".

In addition to the topic suggestions, this thesis contributes to the theoretical knowledge of Design Patterns in relationship to Programming Languages, by providing an overview of the Language Features with relationship to Design Patterns implementations (Section 6.6), proposing a scoring scheme for Language Features (Section 6.7), classifying the Language Features of Java and Python (Section 6.9),determining the relationship between Design Patterns and Programming Languages as bidirectional (Section 6.10) and by proposing two theories with relation to Design Patterns and Programming Languages. One explaining the life cycle of a Design Pattern (Section 6.8) and one explaining the consequence of the bidirectional relationship between Design Patterns and Programming Languages (Section 6.10)

1.11 Conclusion

Within the writing of this thesis, as it has been observed and thus is a suitable to conclude that the relationship between Design Patterns and Programming Languages is in-fact bidirectional. That not only do Programming Languages affect Design Patterns, but the opposite is also true.

Chapter 2

Background

This chapter discusses the background for the thesis. In terms of Design Patterns evaluation scheme and selection, in terms of related to the subject resources. In addition to that it examines theories related to Design Patterns and Programming Languages and specifies scope restrictions, imposed on the topic.

2.1 Resources Overview

There have been many resources, written on the subject of Design Patterns over the years, since the release of the first book on the subject, GoF [31]. Most of the books on the subject are specialized in introducing the topic of Design Patterns to it's audience in a specific Programming Language, such as "Software Architecture Design Patterns in Java" [42] and "Mastering Python Design Patterns" [40]. Other examples include "JavaScript Patterns" [68] and "Mastering JavaScript Design Patterns" [74], introducing Design Patterns in JavaScript, as well as "Design Patterns in C sharp" [51] in C sharp. And these are just a few examples, based on the Programming Languages, considered as candidates for this thesis in chapter 3.

In addition to that many more Design Patterns as well as variations of the original 23 Design Patterns have been documented. As such it could be said that the subject of Design Patterns in nearly all modern Programming Languages has been thoroughly documented. The Language Specific books tend to follow the template of the GoF book [31], by explaining the Design Pattern, by often using UML diagrams and some simplified toy example to demonstrate the need (the problem) and consequently a demonstration of the application of the Design Pattern (the solution to that problem) in the established context. And whilst there are many resources either concentrated on describing the original 23 GoF Design Patterns in a particular language or documenting new Design Pattern, there aren't any recent resources comparing Design Patterns implementations across multiple Programming Languages. Thus the motivation for this thesis.

2.2 Design Patterns and Language Features Scope

The scope with relation to Language Features of the Programming Languages, selected for the evaluation and Design Patterns is as depicted in the Venn diagram in Figure 2.1.

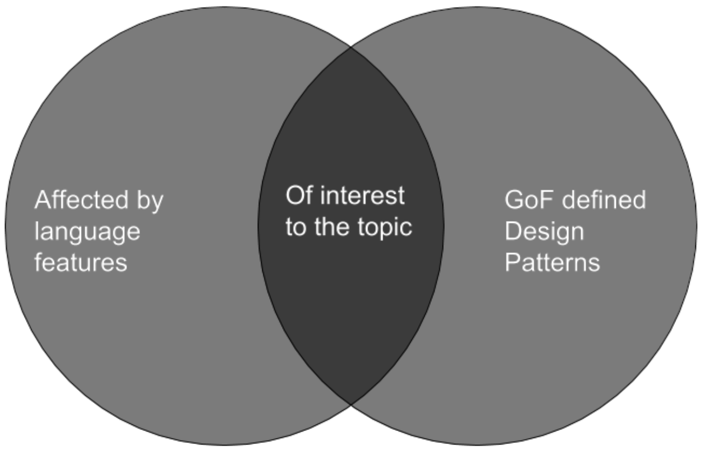

Figure 2.1: Venn diagram of the scope of this master thesis with relation to Design Patterns and Language Features.

As depicted in Figure 2.1, the most relevant Design Patterns are in the intersection of the two logical sets of Design Patterns, namely the Design Patterns affected in some direct way by the Language Features provided by the Programming Languages under evaluation and the Design Patterns defined by the GoF book [31].

2.3 Evaluation Criteria

The evaluation of Language Features is with regards to their impact on Design Pattern implementations in Object Oriented Programming paradigm (OOP) and to the Design Patterns defined in the GoF book

[31]. Thus Language Features and Design Pattern implementations are evaluated with regards to, how the Programming Languages in question influence the Design Pattern implementations and use in these Programming Languages. An evaluation criteria or classification is proposed in Section 6.9 as well as a scoring system in Section 6.7.

2.4 Design Patterns being missing Language Features

An important recurring and related to the topic theory is the one that Design Patterns are missing or unimplemented Language Features. It is relevant to this discussion because if Design Patterns are truly missing Language Features, then this topic will be "How languages affect their missing features", which implies a very short and direct answer: "By ignoring them and not implementing them.".

Strong evidence exists and has been presented that the theory is true. Peter Norvig found that, 16 of the 23 patterns in Design Patterns were "invisible or simpler" in dynamic Programming Languages. Where by "invisible" it is meant that the pattern is integrated in the language to such degree that it is part of it [27]. But there should be evidence against the theory as well. Or some kind of reasoning to explain why Design Patterns still exist even after so many years have passed since the theory that they are simply missing Language Features was presented. It stands to logic that if Design Patterns are truly just missing Language Features, then by now they would be part as standard Language Features of nearly all modern Programming Languages, such as Java [49], Python [50], C sharp [15], JavaScript [14] and C++ [69] and many more. But they are not. In addition to that this theory has reappeared and been documented in multiple sources [13].

It is possible that language designers simply have a choice of which more or less standardized features they should include? Maybe the fact that not all Design Patterns are implemented as features has to do with the fact that not all Programming Languages are the same. Leading back to the argument or fact that specific languages are designed to solve specific set of problems and if not all Design Patterns are required to solve these problems efficiently then there is little purpose for the language designers to implement Design Patterns that nobody or a very small subset of the users will actually use. As implementing Language Features that are not used is effectively wasting time and energy. As opposed to implementing Language Features that the users need and will use. In a sense making the argument that only what is needed is implemented. Because, for example the Bridge Design Pattern is not a Language Feature that is commonly used or popular, then it is not implemented as a standard Language

Feature in Programming Languages, such as Java [49], Python [50], C sharp [15], JavaScript [14] and C++ [69] and others. In other words if there is no demand for the Language Feature implementation by the users of the Programming Language, then the Language Feature probably isn't implemented.

And in addition to that including features that a relatively small subset of users would use in a language could lead to bloating the language interface with irrelevant features, making it more difficult to learn. For example the C++ language reference [69] has consistently grown over the years, up to around 1000 pages.

Thus the argument is that in order for a general programming language to stay a general programming language and be easy to learn, the language should not try to tailor itself to every domain specific Design Pattern and try to provide special tools for it. And rather leave that to the implementations of libraries and frameworks. Such that the language does not try to become a Domain Specific Language (DSL) for all domains, which would inevitably lead to a bloated Programming Language.

As such, it is assumed in this thesis that, while some Design Patterns may be missing Language Features in some cases, this is not necessarily true for all Design Patterns. Otherwise Design Patterns would not exist anymore and this topic wouldn't exist.

2.5 Related Work

This section contains a list of Language Features or topics that are related to the topic of this thesis, but either not related closely enough to be considered in this thesis or that examining the Language Features or topics in question could potentially be a topic on their own.

2.5.1 Annotations

Annotation is a way for the programmer to specify metadata for use by the compiler or other tools that deal with inspecting the code. They are provided by both Java [43] and Python [43]. In Java the annotations are often used by frameworks, some of which implement Design Patterns. Examples of that are Dagger framework [23] and the Spring framework [67], which both provide the Inversion Of Control Design Pattern and dependency injection. However the Annotations as a Language Feature, is not directly responsible for implementations of Design Patterns, but only used to facilitate the implementations. As such any deeper discussions and examinations of the Language Feature would lead to examinations of the frameworks and possible comparisons of frameworks to the Programming Languages. Thus the Language Feature even though somewhat related to the subject is out of scope of this thesis.

Because, doing so would derail the topic and possibly be a topic on it's own. An example heading of which could be "Examination of effects of frameworks on Design Patterns implementations and use in Programming Languages".

2.5.2 Metaprogramming

By "metaprogramming" it is meant the "programming technique in which computer programs have the ability to treat programs as their data." [44]. This includes Reflection in Java [76] and metaprogramming in Python [45]. Metaprogramming as previously mentioned involves the program treating programs or itself as data. Ability to modify itself. As such using it for Design Pattern implementations, especially for example for Structural Design Pattern implementation is unsuitable, because the point of the Design Pattern, imposing a thought of structure on the code in an organizational matter would be encapsulated in the modification of code. Thus resulting in possible obscuring of the said Design Pattern. Whilst some metaprogramming techniques are used and discussed in this thesis, in terms of using Meta Classes in Python (Section 4.3 and Section 6.1) to implement Design Patterns, it is not the case that all metaprogramming is examined or even attempted to be examined or the relationship between Metaprogramming and Design Pattern in Programming Languages evaluated. This examination, partly due to the fact that is not directly related and partly due to the fact that a through examination or evaluation of the topic is a masters thesis topic on it's own, is thus defined as out of scope of this topic. As such it might be an appropriate topic of it's own "The role of metaprogramming in Design Pattern implementations".

2.5.3 Programming Paradigm Affects on Design Pattern Implementations

Programming Paradigms are related to Programming Languages in terms of which paradigm the languages provide the users of these languages support for. Whilst the interaction between Programming Paradigms and Design Patterns implementations in the Programming Languages as a result of the paradigms supported by the languages is relevant and is examined to some degree in this thesis (Section 6.3 and Section 4.5), a fully fledged examination of that interaction is out of scope of the thesis. It could potentially be a good masters thesis topic on it's own in the field. A suggested title could be "Programming Paradigm effects on Design Patterns".

2.5.4 Serialization Techniques

The State and Memento Design Pattern's could potentially be implemented using the Serializable Java Language Feature or in Python the pickle Language Feature [1]. However based on the fact that these Language Features are not straight implementations and the Design Patterns are not strongly related to the the Language Features, it was decided that implementing the Memento and State Design Patterns using these Language Features would be in effect misusing them. It will be misusing them, because the Language Features in question are meant to serialize objects to disk and not simply to make snapshots of their state. Thus not contributing constructively to the topic of discussion. As such these implementations are defined out of scope of this discussion.

2.5.5 Modeling vs Typing As Topic Suggestion

This section found that interfaces in Java affect pretty much every Design Pattern implementation. It also discussed the differences between Interfaces and Abstract Classes in Java in relation to the fact that Python does not have direct replacement for Java Interfaces, but has an implementation of Abstract Classes, called Abstract Base Classes (ABC's). The Interfaces in Java are more geared towards defining Types, whilst inheritance with Abstract Classes augmenting it towards Modeling in an Object Oriented Programming context. Thus a potential discussion on the topic could lead to potentially interesting results. However a full fledged discussion and examination of that topic is out of scope of this thesis, as it could potentially in addition to requiring a thorough investigation, require case studies to back up any claims made in that investigation with real world data. As such, it could potentially be a suitable masters thesis topic on it's own. Maybe called "Modeling versus Typing".

2.6 Summary

This chapter discussed the related to the subject resources that are available. In addition to that it defined a few related topics as out of scope of the thesis and related to the topic theories.

Overall, there are plenty resources available, which describe Design Patterns in a particular Programming Language. However, most restrict the discussion to only one Programming Language (such as in Java [42] or Python [40]) and the ones that describe Design Patterns in more than one Programming Language (such as the Sourcemaking website [26]) do not actually analyze or make efforts to compare the implementations.

Part II

Analysis

This part of the thesis contains the analysis and comparisons of languages, language features, mechanisms, programming concepts and anything else relevant to the implementation of a design pattern.

Chapter 3

Programming Languages and Design Patterns

This chapter explains the Programming Languages selection criteria, This chapter describes the process used to select programming languages for this thesis. Because, the pool of existing Programming Languages is potentially quite large, some selection rules must be established to govern the selection process. As an example of the potential size of selection pool of languages, "The Big List of 256 Programming Languages"[70] lists 256 programming languages, based on TIOBE [75] and GitHub alone. Obviously this is not a definitive list of all existing Programming Languages, but it gives an idea of the landscape and the need of clearly defined selection criteria.

3.1 Criteria

The first requirement for the candidate Programming Languages is that, Object Oriented Programming paradigm (OOP) is supported within the Programming Language of choice. This requirement is necessary, due to the fact that the Design Patterns in question (the 23 described in the Gang of Four book [31]) rely heavily on the use of OOP. In addition to that, without support for OOP the Design Patterns would be more difficult to compare.

The second requirement is that, the Programming Language of choice is among the more popular ones, at least in the recent years. The reasoning behind that requirement is simple. The more popular a Programming Language is the more it is used. Thus the more used Programming Language is potentially better kept to date than a niche Programming Language and it has had more development iterations. Because of that, a more popular Programming Language is more likely to truthfully reflect effects of Programming Languages on Design Patterns, than a no longer used or updated Programming Language.

And third and last requirement is that the language brings something different to the discussion and comparison. This could be by either supporting more than one programming paradigm or maybe by providing different Language Features to the language users. There is no point in examining two very similar or almost the same Programming Languages and ignoring the dissimilar ones from the selection pool. As this would not only lead to comparisons of poor quality and contrast, but it would possibly lead to false conclusions.

3.2 Selection

As mentioned before the selection pool for Programming Languages is potentially large, as such a shorter selection or short-list of at most 10 or so to choose from must be established. And from that short list a selection of two or three Programming Languages must be made. The process of selecting a set of Programming Languages, based on the criteria described in the previous subsection is based on using online rankings of the most popular Programming Languages. Three rankings were used, The Spectrum IEEE magazine website [16], TIOBE Software website [75] and the RedMonk Programming language rankings online [66]. [16], [75], [66]

Thus, based on these rankings and the established from Section 3.1, selection criteria, the selection of Programming Languages was narrowed down to two, Java and Python. Other Programming Languages such as JavaScript and C sharp were considered, but ultimately the selection was narrowed to two, additionally due to time constraints.

3.2.1 Java

Java is an Object Oriented Programming Language and it's syntax is heavily influenced by C++ [73]. It is among the top ten languages by popularity according to the previously mentioned ranking sites [16], [75], [66]. In addition to that it is heavily used in the courses in the University of Oslo. This makes it a perfect choice for the purposes of this thesis.

3.2.2 Python

Python is also among the ten most popular languages in the previously mentioned rankings [16], [75], [66]. It was chosen due to it's contrast to Java. Whilst Java is OOP language, Python supports both OOP and Functional programming (FP) [30]. In addition to that Python is primarily an interpreted Programming Language, which could potentially lead to more differences between the languages based on their typical use case scenario.

3.2.3 Other Candidates

There are three other languages that were considered and part of the including in the comparison. They were also in the top ten most popular languages rankings [75], [16], [66]. They are C sharp [15], JavaScript [14] and C++ [69].

Java, C sharp and C++ are fairly similar and there is little point in including multiple similar languages, thus only one of them had to be selected. As previously mentioned that is Java, so C sharp and C++ are dropped from the selection.

JavaScript is definitely an interesting candidate for the Programming Language comparison. Whilst there are similarities between Java and JavaScript in terms of syntax and libraries, the two languages differ in design and use. JavaScript is a muti-paradigm language, supporting both Object oriented programming and functional programming, whilst Java is Object oriented one. JavaScript is primarily used in web development for client side scripting, though since the introduction of Node.js (the JavaScript run-time-environment [28]) it is possible to use JavaScript for server side scripting as well. Unfortunately due to time constraints it was decided that two languages are more than enough for a comparison. Thus JavaScript had to be dropped from the comparison list as well.

3.3 Summary

Java and Python are the two Programming Languages, selected for examination in this thesis. Both support the Object Oriented Programming paradigm (OOP) and whilst there are many other viable candidates [70], the choice had to be restricted to only two. As such these two Programming Languages are used for the comparative study in this thesis.

Chapter 4

Python

This chapter examines a selection of relevant to the topic Language Features provided by the Python programming language and it's standard library, in the standard cpython implementation [22]. This chapter also briefly describes the process of introducing new Language Features to the language and the history of the relevant Python Language Features.

4.1 Python Enhancement Proposals

Python Enhancement Proposals (PEP) [52] is a mechanism for the Python community to propose and select Language Features to include in future iterations of the Python language. Roughly the process is such: First a PEP is written, then possibly redacted and re-written until a consensus is reached. Then the enhancement is implemented.

The list of Python PEPs can be found at [52] and there are three types of PEPs, "Standard track", which are descriptions of new features, "Informational", which describe particular design issue and "Proces", describing a process surrounding Python. The relevant to the purposes of this topic PEPs will mostly be the "Standard Track" PEPs. More information about the PEPs can be found in pep-001 [53], which serves as a guideline document.

4.2 Python Magic Methods

Magic method is a term that refers to a class of methods provided by Python, that facilitate specialized functionality to the user. Their names are always preceded and followed by two underscores, for example `__init__()`. And they support all object oriented functionality that Python provides. Magic methods are not well documented, by the python documentations, in a sense that one could not browse all magic methods, rather their definitions are spread over the documentation. An attempt to succinctly list and document all magic methods has been made by quite a few websites and blogs. One of them is the "Guide to Python's Magic Methods" by Rafe Kettler [12]. Another example of magic method use is when one decorates a class or a function with an object, that implements the `__call__()`. By having that magic method, the object effectively is considered to be a function by the Python interpreter, due to the dynamic typing.

They are of importance, since many of the examples in this thesis depend on implementing these methods in order to work.

For example by providing `__call__()` an implementation that uses decorators, can use a class to decorate another class. This does qualify as an alternative implementation. And thus, some magic methods affect Design Patterns by providing an alternative implementation route. More details on how the decorators and `__call__()` function interact can be found in Section 4.5.

Magic methods are simply a different mechanism compared to Java, that facilitates the same functionality. Example of that is the `__iter__()` function, which is supposed to return an instance of the iterator. In Java the collection would have to implement the Iterable interface. But, Python doesn't have interfaces and has duck typing, where if the Object implements `__iter__()` it is of type Iterable. Their impact on Design Pattern implementations, however seems to be minimal.

4.3 Metaclasses

According to Guido van Rossum's blog, The History of Python [72], Metaclasses were first available in Python since version 1.5, 1998. This makes them a relatively old Language Feature of the language, since an early iteration of the Language Feature was part of nearly the first version of the programming language. As such there doesn't seem to be a PEP corresponding to the original concept of Metaclasses. However a PEP, describing the changes to Metaclasses in Python version 3 is available, called "PEP 3115 – Metaclasses in Python 3000" [56].

According to the Python documentation, in Python everything is an Object [7]. That means that, classes are objects, even that modules are objects. Because, classes are objects, their class in term is an object as well. In Python one can define these "classes of a class" and they are called Metaclass. It is somewhat self explanatory from the name, since it is called a Metaclass.

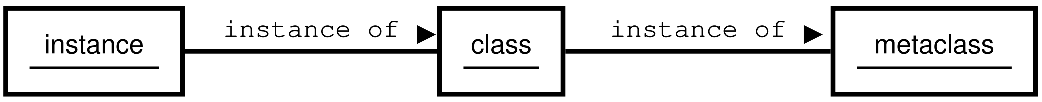

Figure 4.1: Diagram of metaclasses in Python

Thus a Metaclass is the class of the class [79], as depicted in figure 4.1. In addition to that the default Meta Class of classes is `type`, also dubbed as the `type()` Built-in function [2]. Type, according to the documentation is a special function or object and it's own class, tasked with constructing new types, and as stated by the documentation equivalent to using the class statement[2]. Thus it is the function responsible for creating classes in Python, but it can also be used instead of the `class` keyword to dynamically construct classes. This function is also linked to the "Type Objects" in the language, since it is used to access the type of a given object [8]. Figure 4.2 and Figure 4.3 depict use of the `type()` function versus the equivalent Python syntax.

```python
MyClass = type('MyClass', (), {})
# Where, the result is:
# >>> MyClass
# <class '__main__.MyClass'>
```

Figure 4.2: A demonstration using Metaclasses to construct classes in Python

```python
class MyClass():
   pass
# Where, the result is:
# >>> MyClass
# <class '__main__.MyClass'>
```

Figure 4.3: A demonstration of the equivalent Python syntax to using Metaclasses in Python

4.3.1 Use

Figure 4.4, depicts the typical syntax of Metaclasses in Python. In it, a class "MetaClass" is defined and used as a Metaclass by another class "MyClass".

```python
class MetaClass(type):
    pass

class MyClass(metaclass=MetaClass):
    pass
```

Figure 4.4: A demonstration of the Metaclass syntax in Python

4.3.2 Using Metaclasses to Implement Design Patterns

Metaclasses are mostly useful in the Design Pattern context, to control the creation of classes. As such their usefulness in terms of implementing Design Patterns is limited to Creational Design Patterns. In particular, a typical and popular example is that of using meataclasses to implement the Singleton Design Pattern, where the metaclass holds a list of instances and only creates one, amending the list accordingly. This is demonstrated in figure 4.5.

```python
class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        if cls not in Singleton._instances:
            Singleton._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return Singleton._instances[cls]

class MyClass(object, metaclass=Singleton):
    def __init__(self, name):
        self.name = name
```

Figure 4.5: An implementation of the Singleton design pattern using metaclasses in Python

In figure 4.5, a Metaclass called Singleton is defined. This metaclass extends the type metaclass and contains a dictionary `_instances`, in which all instances of the metaclass objects are registered upon creation. Notice that instances are only entered if the `_instances` does not contain an instance of the class. So, normally the `__call__()` method in the Metaclass would be executed, before the `__init__()` method in the class that has that Metaclass as metaclass. Thus in effect intercepting the call to `__init__()` and only completing the call `super(Singleton, cls).__call__(*args, **kwargs)` to the `super.__init__()` if it satisfies the Singleton condition. This is better demonstrated in the more complete example, that can be found in Appendix A.1. The example in Appendix A.1 also demonstrates that the method works when multiple classes have the same Singleton Metaclass, meaning that `cls not in Singleton._instance` evaluates to `True` for different classes with the same Metaclass, inside the Metaclass.

4.3.3 Summary

Overall Metaclasses seem to be a useful feature that has a noticable impact on how design patterns are implemented. In addition to that, the type class is an implementation of a design pattern.

4.4 Abstract Base Classes

Abstract Base Classes in Python, is a Language Feature, provided by the `abc.py` module [20] are, as the name suggests abstract classes intended to be used as base classes for other classes, by using inheritance. They were first introduced in 2007 and available in Python version 3, according to "PEP 3119 – Introducing Abstract Base Classes" [57].

In Python the abc module provides a class and a The important part is that, while inheriting from the appropriate ABC is encouraged, by potentially providing "default implementations for certain functionality to an improved ability to distinguish between mappings and sequences." [57], but not enforced by the Python language. Thus in a way it is a Language Feature that remains backwards compatible.

4.4.1 Use

Abstract Base Classes are used, by either using them as metaclasses for the custom user-defined classes or by extending from them. Both methods yield equivalent result, as demonstrated in Figure 4.6 and Figure 4.7.

```
class C(metaclass=ABCMeta):
    @abstractmethod
    def c():
        pass

class D(C):
    def c():
        pass
```

Figure 4.6: Using the metaclass in Python

```
class C(ABC):
    @abstractmethod
    def c():
        pass

class D(C):
    def c():
        pass
```

Figure 4.7: Using a Metaclass, by extending it in Python

Implementation wise, Abstract Base Classes are soft-implemented in the Python language, as it can be seen from the Cpython implementation [20]. The annotation `@abstractmethod`that is in fact part of the heart or core of the implementation detail, simply sets a flag `__isabstractmethod__` to true. Thus upon object creation of this class, a check is made if all abstract methods are implemented and if not, as Figure 4.8 shows an exception is raised.

```
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#TypeError: Can't instantiate abstract class D with abstract methods c
```

Figure 4.8: Exception thrown, when the subclass has not implemented all abstract methods and tries to create new objects in Python

Abstract Base classes are centered around the `@abstractmethod`, without defining at least one `@abstractmethod`, the Abstract Base Class has nothing else that stops the user from instantiating objects from the class. This has to do with the fact that they are, as previously mentioned, "soft-implemented",

```
class E(metaclass=ABCMeta):
    pass
#>>> e = E() # is completely legal and ok with the interpreter
```

Figure 4.9: Example of instantiating empty Abstract Base Class Python

As Figure 4.9 demonstrates, instantiating a class that has `ABCMeta` as metaclass, but no methods defined as `@abstractmethod` results in no error. It is arguable how useful it would be to have an empty abstract method, but the fact is that substantiation of an abstract method is possible. Which in other languages such as Java is not the case.

4.4.2 Summary

Overall, the Language Feature Abstract Base Classes in Python, is not fully part of the language, but rather implemented as an extension. When used in practice, the Language Feature is useful for and defining Base classes with abstract methods that have to be implemented. In other words it can be used to define something similar to interface or type, in terms a specification that has to be followed. In the Python documentation and design documents it is referred to as a method to check if an object conforms to a "Protocol" [57]. It is loosely related to Design Patterns, in terms of possibly being used as an alternative means of defining a specific interface. This discussion can be found in Section 6.1.

4.5 Decorators

The decorator Language Feature that Python provides is thought to be, by some, an implementation of the Decorator Design Pattern, based on it's name. However further investigation, reveals that this is not exactly the case, as this section would demonstrate and explain. This feature was first introduced in Python version 2.4, according to "PEP 318 - Decorators for Functions and Methods" [59]. And subsequently enhanced to additionally be able to decorate classes in Python version 3.0, according to "PEP 3129 – Class Decorators" [58].

4.5.1 Description and Example Use

As presented in the Python documentation [63], in Python decorators are a "syntax that allows us to more conveniently alter functions and methods". Other sources such as [24] compare python decorators with macros and even claim that the feature could be used to implement the decorator pattern. Though as I will show later I disagree with the practicality of that statement.

One of the best descriptions of decorators available describes decorators as "providing a simple syntax for calling higher-order functions."[61]. It presents the feature in its context, namely oriented towards the use of the language in a functionally oriented programming manner.

That description suits the feature the best because it is primarily meant to be a functional programming feature from the start of it's implementation as described in the Python wiki on decorators[63]. And as it will become evident by the end of this section.

The decorators that python provides use Java like annotation syntax, the decorator is listed before the definition of a function or a class and preceded by an @. They can be used to decorate any object that implements the `__call__` function. Thus both classes and functions can be decorated. Also multiple decorators may be used on the same function or class, in which case the order of decoration goes from the closest defined to the function or class up towards the last. Decorators, behind the scenes serve as syntactic sugar for what can be called function or method reassignment. As mentioned in the decorator's PEP [60], that is according to their design.

Usually decorators contain wrapper functions, because the wrapper functions' arguments are the decorated object's arguments. In other words without defining and returning an inner wrapper function one cannot forward the arguments to the original object and thus limits the use of the decorator to only be used to decorate objects without arguments. The general idea behind Python decorators is the following: Define an object (function or class) to be used as a decorator. Inside that, define a wrapper, that is returned at the end of `__call__`, or the end of the function. The arguments to the wrapper would be the arguments to the original function.Inside the wrapper, before executing the decorated function one could execute other code. One could also execute other code after running the decorated function, by running it after explicitly executing the decorated function with the arguments of the wrapper. Also if the wrapped object is a class one should return the instance.

The following two figures demonstrate a typical use case and syntax of Python decorators. Where, on the left, in figure 4.10 a function is used to decorate another. And on the right, in figure 4.11 side of it the same result accomplished with re-assigning the function reference.

```
@decorator1
def func():
    pass
```

Figure 4.10: Python decorators

```
def func():
    pass
func = f1(arg)(f2(func))
```

Figure 4.11: Function assignment

A more detailed and functioning example of a function being used to decorate another is available in A.2. And the re-assignment equivalent is available in A.3.

As previously mentioned any object that provides an implementation to `__call__()` is legible for both being a decorator and for being decorated. Thus both functions and classes can be used to decorate or can be decorated.Table 4.1 shows the possible permutations.

function decorating a function	class decorating a function
function decorating a class	class decorating a class

Table 4.1: Combinations of possible Python decorators usage

Technically, if an object implements `__call__()` it could be considered a function, due to the duck typing of Python. Thus weather one uses a class or a function, both are treated as the same thing, namely an object that implements `__call__()`. Using a class, from Object Oriented Programming (OOP) perspective, could provide more power and flexibility to the user, as it allows the user to leverage the advantages of OOP, encapsulating subroutines in functions, object variables...etc. Even though one could technically decorate a class, as demonstrated in figure A.4, only the `__init__()` function of that class is actually decorated.

When using a class for a decorator (A.5 and A.6), `__init__()` would be called to initialize it, with the decorated object as an argument. And `__call__()` would be called, when the decorated object (function or classes method) is called, and it's arguments would be the arguments to the function or classes method. The resulting behaviour is the same as if a function is decorating another. A.6

Decorators can also accept arguments, just like functions. As demonstrated in A.7.

4.5.2 Decorators versus Decorator Design Pattern

As it should be obvious by now Python's decorators are are different from the Decorator Design Pattern.

The Decorator Design Pattern (if implemented using composition or inheritance) effectively re-uses the class/object that it decorates. Thus the original decorated object could be reused, decorated by different decorators, on their own or on top of each other. One also could define multiple decorators to the same class or object. And the important part is that the original class is still accessible and usable.

On the other hand Python decorators accomplish the reverse. One defines a decorator that can be reused to decorate multiple classes or functions. However the decorated functions can either be decorated by a single decorator or multiple decorators. Thus the decoration is not in the same sense as the Decorator Design Pattern. One possible solution to that is to un-decorate the function and re-decorate it afterwards, using the undecorate package, [78].

In addition to that as demonstrated in the previous subsection, decorating a class in python involves editing the original definition of the function or class. Which is not what the Decorator Design Pattern does.

Thus the Decorator feature of Python, whilst useful in implementing some Design Patterns is not applicable as to the implementation of the Decorator Design Pattern.

The issue with Python decorators is that they replace the reference to the original class or function with the decorated one. This is possible, because of the first class citizen status of functions in Python. And that is what Python decorators as a Language Feature distill, replacing the original reference with the one of the decorator. Also this is the main difference between decorators and the Decorator Design Pattern. That Python decorators are syntactic sugar of a special use-case implementation of the Design Pattern, where the original reference is not needed and thus replaced. Figure A.8 demonstrates what the Decorator Design Pattern would look like in Python for functions. The important difference is depicted in 4.13.

Thus in conclusion it could be said, based on all of the above examples and the feature's behaviour, that Python decorators cannot be considered an implementation of the Decorator Design Pattern. They only happen to be named the same, but differ in the functionality that they provide.

```
decorated = decorator(original)
decorated(5)
```

Figure 4.12: Decorator Design Pattern using python

```
original = decorator(original)
```

Figure 4.13: Python Decorators function assignment

4.5.3 Use of Decorators to Implement Design Patterns

As previously described, the re-assignment behaviour of Python decorators limits their use for Design Pattern implementations. As the resultant changes are be permanent.

Thus, and as previously described, due to the fact that the original references to the decorated object are replaced by the decorator, implementing the Decorator Design Pattern using Python decorators, is not appropriate. Despite the claim made in Mastering Python Design Patterns that it is the approach to use [40]. The result of using Python decorators to implement the Decorator Design Pattern would also require editing the original class definitions in order to decorate the functions provided by the original class.

Because, decorators replace the reference of the class or function they decorate, implementing the Singleton Design Pattern is probably the most straight forward and direct use of decorators. The permanent changes to the class, in order to make sure that only one instance is ever created are desired. In that sense the undecorate python package becomes a weak point of the implementation. Because the user can undecorate or in other words remove the decorator that makes sure that only one instance is allowed.

The implementation can be found in figure A.21. For simplicity the example implementation uses a function to decorate the classes that will be singletons. Decorating a class, decorates the `__init__()` method of that class, thus in the decorator one could create a dictionary of the instances and maintain only one instance in the dictionary, returning it instead of creating a new instance, when called.

The positive about using Python decorators to implement the Singleton Design Pattern is that the decorator can be reused, once the decorator has been defined. Also it is visually distinct and clear, by the statement "@singleton" above the class definition. x

4.5.4 Summary

Despite the fact that Python decorators are specialized form of the Decorator Design Pattern, with limited application with regards to other Design Pattern implementations, they can still be useful. However their

```python
def singleton(class_):
    instances = {}

    def getinstance(*args, **kwargs):
        if class_ not in instances:
            instances[class_] = class_(*args, **kwargs)
        return instances[class_]
    return getinstance
```

Figure 4.14: Python decorator definition for singleton implementation

use is limited, and they cannot be used effectively to implement the Decorator Design Pattern. But they could be used to implement the Singleton Design Pattern to a satisfactory result.

4.6 Iterator

Python provides the Iterator Design Pattern, as a Language Feature, first released in Python version 2.1, according to "PEP 234 – Iterators" [54]. Python provides implementations of the Iterator Design Pattern in the form of providing a set of types as part of the Builtin types [9]. Thus defining the standard operations according to the Design Pattern. The types in question are Iterator and Iterable. Note, Iterable is actually referred to as Container in the documentation, however, for clarity and simplicity it is referred to as Iterator henceforth.

4.6.1 Implementation Details

By having the Iterator Design Pattern defined as a composition of types, Python clearly defines the interface of iteration that users can implement in their collections or containers. In addition to that all of the default collections and containers implement the Iterator Design Pattern, according to the Python documentation [9].

The above-mentioned types are implemented as Abstract Base Classes (ABC's) behind the scenes [19], where the Iterator type corresponds to the Iterator ABC and the `container.__iter__()` is actually defined in the Iterable ABC. A summary of the methods that a class should implement in order to be considered an Iterator or Iterable by Python is shown in figure 4.2.

Iterator	Iterable
`iterator.__iter__()` `iterator__next__()`	`iterable.__iter__()`

Table 4.2: Iterator and Iterable types summary

Iterators do not, however, implement all of the methods defined by the Iterator Design Pattern. Instead of providing a method `iterator.__hasNext__()` that returns true, if the iterator has more elements to iterate, the `iterator.__next__()` raises an StopIteration exception. This could potentially be the weak point of the implementation, as exceptions as the name suggests are meant to be raised in exceptional situations. And usually there are costs involved with raising an exception, with regards to creating the exception objects, with the necessary information. One would expect that using an exception would be slower than an if-else statements block. However as shown by blog posts [41] [32], that examine the topic of "performance comparisons between exceptions and if statements" in further detail, the decision isn't as simple. In other words, whilst it could be a concern, in practice the difference in performance is probably negligible, since it has been implemented as such since version Python version 2.1. Instead here an exception is raised upon reaching the end of an iteration. The other methods in the figure are `container.__iter__()`, which returns an iterator object for this container, `iterator.__iter__()`, which returns itself.

The Iterator implementation was first proposed in 2001 [55], and is available for Python version 2.1.

4.6.2 Why use Python Iterators

The main reasons for implementing iterator using the provided build-in types are: First, by doing so, the implementation is following a well defined standard, thus the user implementation is easier to understand and use by others, since the interface is pre-defined and well known. The second reason for using the build-in types is that all of the for-loops in Python actually use the iterator types for their iteration. Thus again the collections or containers would be easier to iterate, thus easier to use. Figure 4.15 compares the for-loop use versus using the underlying iterator. It is obvious from the figure that using the for-loops to iterate collections requires many statements less from the user, thus being less error prone and more compact.

Overall the feature is a useful implementation of the Iterator Design Pattern, albeit a bit modified to use exceptions to signal when the iterator is exhausted. In addition to that the for-loops provide excellent integration of the Design Pattern into the language.

```python
for(element in collection):
    #do something with element
    pass
```

```python
iterator = collection.__iter__()
while True:
    try:
        element = iterator.__next__()
        # do something with the element
    except StopIteration:
        break;
```

Figure 4.15: Comparison of iterator use to for-loop in Python

4.7 Summary

Overall, Python Iterators are fairly well integrated in the Programming Language and consistently used throughout the language by the for each loops for iterating over objects. In addition to that as Section 4.6.2, thanks to the way that Iterators are implemented, the user of the language is incentivized to follow the template as defined by the Language Feature for the user-defined collections as well. Thus defining a certain standard, which leads to better consistency throughout the Programming Language and the code written by the users of that language. Which in term could lead to less error prone code and more readability of the code, because of the consistency.

4.8 Object Copying

Python provides object copying out of the box for the users. The feature seems to have first been available in Python version 1.0 (1995-01-10), evident from the language implementation source file history [21].

4.8.1 Description and Example Use

As explained by the Python documentation [10], Python provides two ways of copying or cloning an object. They are by calling either the `copy()` or the `deepcopy()` method of copy in the standard library. The `copy()` returns what is called a shallow copy, whilst the other, `deepcopy()` returns a deep copy of the object. As previously mentioned a shallow copy is one that does not include copies of other objects referenced by that object, whilst the deep copy makes deep copies of the referenced by the current object objects. Providing user defined implementations of these methods is done by defining the `__copy__()` and `__deepcopy__()` methods inside the class that should have a different than standard behaviour on copy and on deepcopy. In principle implementing these methods does not mean that the object that implements them actually overrides the methods in the super, since they are not actually defined in the super-class, but the standard library instead. So in order to actually copy an object, one has to import copy from the standard library. A simple example of the Language Feature in practice can be found in figure 4.16.

```python
import copy

newCopy = copy.copy(oldObject)
```

Figure 4.16: A demonstration of object copying in Python

However, due to the fact that these functions are in a library means that modifying their behaviour could be problematic for the user in the case that the user would like to user the default behaviour, but also build on it and contain it inside either the `__copy__()` or `__deepcopy__()` methods. In this case one could use `getattr()` [3] and `setattr()` [4] to modify the attributes of the object. A demonstration of that can be found in figure 4.17, where the copy from the Python library is used to make a copy of the object inside the overridden copy method, without creating an infinite recursion loop, by removing the `__copy__()` attribute from the object, before calling the copy method.

```python
def __copy__(self):
        # remove method to make a copy, while inside method:
        copyMethod = getattr(type(self), "__copy__", None)
        setattr(type(self), "__copy__", None)

        result = copy.copy(self)

        # reassign the method
        setattr(type(self), "__copy__", copyMethod)

        # make special copy operations
        result.machine = copy.copy(self.machine)

        # return the copy to client
        return result
```

Figure 4.17: A demonstration of overriding the copy method in Python

An example use case, where one would like to create a copy of some referenced by the object of interest objects, but not all, can be found in figure 4.18.

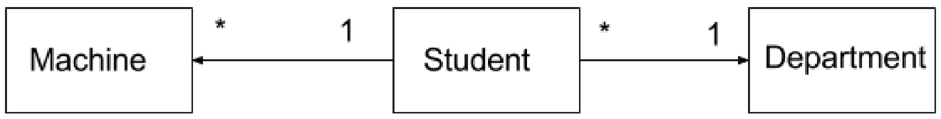

Figure 4.18: Cloneable example overview

Figure 4.18, depicts the object relationships in the example. In short, Assume that a system for tracking students and their personal computers, machines, in a department exists. Thus, the department would exist for many students, but each student should have their own machine. A full implementation of the use-case solution in Python can be found in Appendix A.10. The clone function has already been presented, in figure 4.17, where the machine is copied, whilst the department is left as a reference to the same object as the original of the copy.

4.8.2 Problems With the Language Feature

As discussed in Eric Bartley Jul's phd "Object mobility in a distributed object-oriented system" [39] there exists the so called "Copying the World problem", when doing a deep copy of objects. In a sense, if the `deepcopy()` function is recursively ran on all references inside an object, then it is bound to return a copy of the entire system aka. the world. If on the other hand the function implementation does not copy all objects referenced by the one that is deep copied, even if the objects in question are part of the Python environment, then the function is not a true "deep copy". In other words, "If a deep copy is to be provided by default, then how does the default implementation know where the desired by the user deep copy conceptually stop? Also do the system objects get deep copied, when a deep copy is called on a user defined object ? If so why ? If not then why again? What is a reasonable behaviour in this case could entirely depend on the user use-case and interpretation. Thus, a "reasonable" or "sane" deepcopy implementation, could possibly not exist at all. As such providing a default implementation of deep copy, is bound to be sub optimal for some cases, based on the user's use-case and interpretation. And thus the question of whether it is worth providing such implementation, given the circumstances has to be raised.

In Python, as evident by the documentation [10], it has been decided that providing a default implementation of deep copy, is worth investing time in. That implementation presumably does not copy system objects, based on the documentation pages, "This module does not copy types like module, method, stack trace, stack frame, file, socket, window, array, or any similar types. It does “copy” functions and classes (shallow and deeply), by returning the original object unchanged; this is compatible with the way these are treated by the pickle module." [10] . As such the language provides the functionality in that form as a default implementation, with the ability of the users to override the implementation with their own customized one.

4.8.3 Summary

The copy Language Feature is provided in Python as a module that the user has to import in order to use. This detachment of the Language Feature from the Programming Language is a drawback, with regards to the ease of use from user perspective. And whilst it can be argued that implementing a module is simple and easy to do, it is not the same as having the copy functions available by default. Despite some challenges on a conceptual level, as discussed in this Section, the Language Feature, with it's default implementations are usable and presumably utilized by the Programming Language users.

4.9 Summary

Overall from the Language Features discussed, it was found that, Metaclasses 4.3, Abstract Base Classes 4.4, Decorators 4.5 and Object Copying 4.8 can be used to implement Design Patterns, whilst Iterators 4.6 are an implementation of a Design Pattern within the language. In addition to that Abstract Base Classes 4.4 are actually implementation of the Abstract Parent Class, as found in Section 6.1. Whilst using Object Copying 4.8, to implement the Prototype Design Pattern is discussed in Section 6.5. Magic Methods 4.2 on the other hand, were discussed, due to the functionality that they provide. This chapter discussed some of the more directly related to Design Pattern implementations Language Features provided in the Python Programming Language on their own merits. Showing how they are used and briefly assessing their implementations.

Chapter 5

Java

This chapter examines a selection of relevant to the topic Language Features provided by the Java programming language on their own, evaluating these Language Features from the perspective of the thesis topic.

5.1 Cloneable

Java provides a Language Feature called cloning or Cloneable and this Language Feature has been available in the Java programming language since version 1, according to the documentation [17]. In order for the language user to implement cloning as specified by the language, the classes that would support cloning should override `clone()` method, the class Object provides the which the users are supposed to override. In addition to that Cloneable is an interface, without any method definitions. Which the user is expected to implement in combination to overriding the `Object.clone()` method. If the class doesn't override the `clone()` method, but the user of the class or object calls it, the `CloneNotSupportedException` will be thrown, as per specification [17],[46]. So, in other words the empty interface implementation is to signify that the user provides an implementation of the clone() method and nothing else.

Should have Cloneable contained clone() method or not ? One would expect that if one has to implement an interface, that then this interface would contain some methods definitions. This is simply included in the definition of the word interface as per this context. In fact the mismatch that one has to implement an empty interface and override a method from Object has caused many of the language users to question the decision. I would speculate that `Object.clone()` has to be overridden and the overriding method has to call `super.clone()`, because Object and any other potential hierarchical parents could potentially have special requirements as to how cloning of the object has to be done properly.

```java
@Override
protected Object clone() throws CloneNotSupportedException {
    Object output = super.clone();
    return output;
}
```

Figure 5.1: Java overriding clone method

The Java documentation specifies that, with regards to the `Object.clone()` method there are three requirements. First that the cloned object will have different memory address (reference), second that the cloned class will match the original's class and third that the result of comparing the cloned and original using `.equals()` method returns true. Of these only the first is guaranteed, whilst the other are not. The `Object.equals()` according to documentation compares the value of obj a vs obj b, true if equal, false otherwise. According to the documentation one should also override .hashCode if overriding the .equals, which could lead to the speculative conclusion that in order to compare instances of Object quickly Object.equals() uses it's hashCode comparison. Which in term could be based on discretely building a hash based on the class definition. For example an instance of String could hash the contents of the string and compare the hash instead of each and every element of the string, as an optimization. And thus the reasons for the `Object.clone()` method being part of the super class of virtually any class in Java becomes clear. By doing so one allows calls to be propagated upwards the hierarchical chain of inheritance, by calling `super.clone()` inside the overridden method, so that the parents have a chance to handle cloning of the classes properly, on their own, whilst the extending classes can additively modify the way that clones of the particular class are made. In addition to that, another reason to override the parent method and to only make shallow clones by default is that, this way the class implementer can decide which attributes need to be cloned and which don't. Thus opening the possibility for a hybrid or custom solutions versus being able to choose from deep clone and shallow clone. Thus this solution of having shallow clones by default is better than the one proposed in the criticism of the java implementation, that states "It is the desired behavior in most the cases. We want a clone which is independent of original and making changes in clone should not affect original."[11]. Having deep clones by default, means allocating potentially much more memory than the user potentially needs, by creating deep clones of objects that the user doesn't need, just so the user can throw them away after. Thus having the more basic implementation as default and making it extendable to the more complex one is the better choice, because it works for all out of the box and can be extended for a more specialized use. An example of a hybrid implementation can be found in Appendix A, figure A.12 to A.15. Where, in the example there is a Student, which has a Machine and is part of a Department. Now every student has a Machine that belongs to that Student, but many students belong to a Department or in other words share a Department. Thus when cloning Student, Machine is also cloned, whilst Department is kept as a reference.

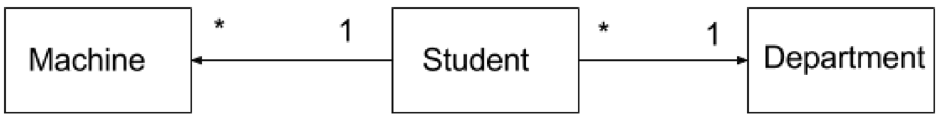

Figure 5.2: Cloneable example overview

But then why have the Cloneable interface ? Most likely to adhere to the established uniform or standardized way of providing design pattern implementations. This becomes more obvious when one examines the Iterator, Iterable, Obeservable and other interfaces that Java provides. They are all provided as interfaces that the user's classes should implement in order to provide this functionality. In other words the interfaces has become a standardized way of communicating that the functionality is implemented or used by the user side code.

5.1.1 Summary

What in reality is provided by this Language Feature in Java, is a default implementation of object copying, making shallow copies of objects. Which the users could override easily. However in order for the users of the Programming Language to make use of even the default implementation, the classes of the objects to be copied have to implement the Cloneable interface. This is a drawback of the implementation, potentially making the Language Feature more difficult to use. On the other hand, the objects that would presumably be copied explicitly state that are "Cloneable", by implementing that interface and references to these objects could be of the type "Cloneable", which could lead to decoupling of potential implementations. So, even though the implementation comes with a drawback it has a positive side to it.

5.2 Iterator

Java provides implementations of the Iterator Design Pattern in the form of providing a set of interfaces. This Language Feature has been available since Java version 1.5, according to the Java documentation pages [34]. And by providing this set of interfaces it effectively is defining the standard operations according to the Design Pattern. The interfaces in question are Iterator, Enumeration and Iterable.

5.2.1 Interfaces

The idea behind providing the interfaces is as follows: The provided by Java collections implement the Collection interface or interfaces that inherit the methods from the Collection interface, which in tern inherits it's methods from the Iterable interface. In other words, the interface that the default collections provided by the language is exposed to the users of the language, such that they could implement that interface. This leads to a consistent naming and use not only inside the implementations provided by the language, but the ones that the users x define. In addition to that

5.2.2 Elements of the Implementation

Technically speaking the Enumeration interface is an implementation of the Iterator pattern. It has been available since Java 1.0 (released 1996) [29] and it does define all methods defined by the Iterator Design Pattern. Albeit named slightly differently, Instead of next() the function is called nextElement(), and instead of hasNext() the function is named hasMoreElements(). However the Enumeration interface, even though available from the very start, is still available for backwards compatibility reasons and new implementations should use the Iterator interface instead, as mentioned in the documentation of the interface [29].

Instead Java provides an interface called Iterator, that is intended for universal iteration and in it's current form, has been available since Java 5 (or version 1.5, released in 2004), according to the Iterable interface documentation [34].

Table 5.1 compares the Iterator interface to the Enumerator interface.

Enumerator	Iterator
boolean hasMoreElements()	boolean hasNext()
E next Element()	E next()
	default void remove()

Table 5.1: Iterator versus Enumerator summary

Just as described by the Design Pattern, in the GoF book[31], there are two entities involved in the Iterator implementation. In the pattern description these are Aggregate and Iterator, in Java they are Iterable[34] and Iterator[36]. Where the aggregate is the collection, whilst the iterator the object responsible for the iteration. In Java the "Aggregate" is the Iterable interface, or one of it's sub-interfaces such as Collection, List, Set, SortedSet, and so forth.

Iterator	Iterable
default void forEachRemaining(...)	default void forEach(...)
boolean hasNext()	Iterator<T> iterator()
E next()	default Spliterator<T> spliterator()
default void remove()	

Table 5.2: Iterator and Iterable interfaces summary

5.2.3 Use

Figure A.11 demonstrates an implementation of the interfaces, by the classes CustomCollection and CustomIterator. CustomItearator doesn't support remove operations and it throws UnsupportedOperationException to communicate that fact to the user. Typically a custom collection would implement either Collection or some of it's sub-interfaces, such as List, Set, Map and others. However for the sake of the simplicity and not to clutter the class interface with unnecessary methods, the example implements the Iterable directly.

The fact that the example class implements the Iterable interface means that, users that would like to iterate over the CustomCollection can use the well defined methods to do so. In addition to that one could use the foreach loop [71]. An example of it's use versus the equivalent code snippet required to iterate over the collection in a while loop is available in figures: 5.3 5.4.

```
Iterator iter = collection.iterator();
  while (iter.hasNext()) {
    Object element = iter.next();
    // do something with the element
  }
```

Figure 5.3: Iterator use

```
for (Object element : collection) {
  // do something with the element
}
```

Figure 5.4: Equivalent for each loop

5.2.4 Summary

As demonstrated, the use of a for-each loop simplifies the iteration process significantly and it is available for any class implementing the Iterable interface and returning an object implementing the Iterator interface. This in combination with the fact that the interfaces are pre-defined and standardized, and unified for use is the benefit of using the provided by Java interfaces. And in combination with the fact that the pre-defined in Java collections all implement these interfaces is the benefit of the Java's implementation of the Iterator Design Pattern.

5.3 Observer

Ever since Java version 1.0 (released in 1994), Java has provided implementation of the Observer Design Pattern. That is according to the documentation of the Language Feature [48] [47].

Java provides implementations of the Observer Design Pattern in the form of providing an interface [48] and a class Observable [47], thus defining the standard operations according to the Design Pattern. The interface in question is Observer and the class, Observable.

By providing a class for Observable, Java provides a default implementation, that in most cases would be sufficient, thus increasing code reuse and reducing code duplication.

Whilst the Observer Design Pattern, as defined in GoF [31], specifies the actors of the Observer Design Pattern to be Subject and Observer, Java renames the Subject to Observable [47][48][42]. In addition to that, Java renames the methods of the design pattern participants, adds a few new ones and defines additional methods to the Observable interface. A comparison between the class interfaces of Observable and Subject can be seen in table 5.3.

Subject	Observable
Attach(Observer)	addObserver(Observer)
Detach(Observer)	deleteObserver(Observer)
Notify()	notifyObservers()
	deleteObservers()
	countObservers()
	hasChanged()
	clearChanged()
	setChanged()

Table 5.3: Subject versus Observable class summary

As table 5.3 demonstrates, Java renames the Attach, Detach and Notify methods to `addObserver()`, `deleteObserver()` and `notifyObservers()`. And also provides additional convenience methods. They are, `deleteObservers()`, which removes all observers from the registered to the Observable Observers, `countObservers()`, which returns the number of registered with the Observable Observers and `hasChanged()`, which returns weather the observable has changed. In addition to that it defines two protected methods, `clearChanged()`, to clear the changed flag of the Observable and `setChanged()`, to flag the Observable as changed.

Observer design pattern	Java Observer
Update()	update(Observable ob, Object o)

Table 5.4: Observer comparison table

And table 5.4 shows the Java Observer interface versus the Observer class, as defined by the Observer design pattern. From the table it can be seen that in Java, when the Observer is updated by the `update(Observer ob, Object o)` function, it receives a reference to the Observable that updated it. This is important, because if an Observer subscribes to or is added to more than one Observables, then it could differentiate on update in terms of which Observable is the cause of the update.

5.3.1 Use

The way the Java implementation of the Observer Design Pattern is used, is by either using the Observable class or extending it and by implementing the Observer interface in the user-defined, custom Observer classes. A full example can be found in Figure A.16 and A.17.

5.4 Summary

Overall from the Language Features discussed, it was found that, Java provides implementations to two GoF [31] Design Patterns. The Iterator Design Pattern is implemented in the form of a Language Feature in Java (Section 5.2) and Observer Design Pattern, implemented as the Observable Language Feature, Section 5.3. Whilst in addition to that Java provides Language Feature related to Object Duplication (Section 5.1), which as discussed in relation to Design Pattern implementations in Section 6.5.

This chapter discussed some of the directly related to Design Pattern implementations Language Features provided in the Java Programming Language on their own merits. Showing how they are used and briefly assessing their implementations. Whilst Chapter 6 contains other Language Features, examined in comparison to Python and in relation to Design Patterns implementations.

Chapter 6

Evaluation

This chapter evaluates the impact of Language Features provided by Programming Languages on Design Pattern implementations in these languages. It does so by comparing equivalent or approximately similar Language Features across the two Programming Languages, Java and Python.

Whilst the previous chapters go more in depth in the Language Features provided by the languages, here the evaluation is in terms of comparison across languages, over Language Features.

6.1 Basic Design Patterns

One of the books on the topic of design patterns in Java, Software Architecture Design Patterns in Java [42], describes and categorizes a few extra design patterns, that are relevant to this topic. They are categorized as Basic Design patterns. Basic, probably due to the fact that in Java, many of the Design Pattern implementations tend to rely on these Design Patterns, as shown in Section 6.6. This Section examines implementations of these Design Patterns in Java and Python and to what degree the said Design Patterns are implementable in the said language, in relation to the provided by the language Language Features.

6.1.1 Overview of Basic Design Patterns Versus Language Features

The Design Patterns versus what Language Feature they either correspond to or rely on in Java, versus in Python is depicted in table 6.1. Note that not all of the Basic Design Patterns described in the book are actually discussed here, only the ones that are relevant to the topic.

Basic Design Pattern	Java Language Feature	Python Language Feature
Interface	interfaces	Abstract Base Class
Abstract Parent Class	abstract classes	Abstract Base Class
Accessor Methods	member access control	property
Private Methods	member access control	naming convention

Table 6.1: Summary of relevant Basic Design Patterns in Java

These patterns are not necessarily unique to Java and can be implemented in any programming language in principle. As such they will be examined in both the context of Java and Python in this Section. Arguably, more in the context of Python, because the Java design patterns book [42] specifies and thoroughly discusses them in Java.

As it can be seen from Figure 6.1, most Java Language Features, have corresponding to them Python Language Features, which the Design Patterns either correspond to or heavily rely on. All these are discussed in the subsequent subsections.

6.1.2 Interface

The Interface Design Pattern is the typical way to use interfaces in Java. Interfaces are used to define a type or interface of a class type, or for the providers. That specific interface is often equated to making a contract between the implementation classes and the user classes, where that contract, in the form of an interface, guarantees that the implementations provide the promised interface and the user classes agree to use that interface. Thus the clients of the providers or interfaces know a type which is common to all providers and only specifies the relevant interface of the type.

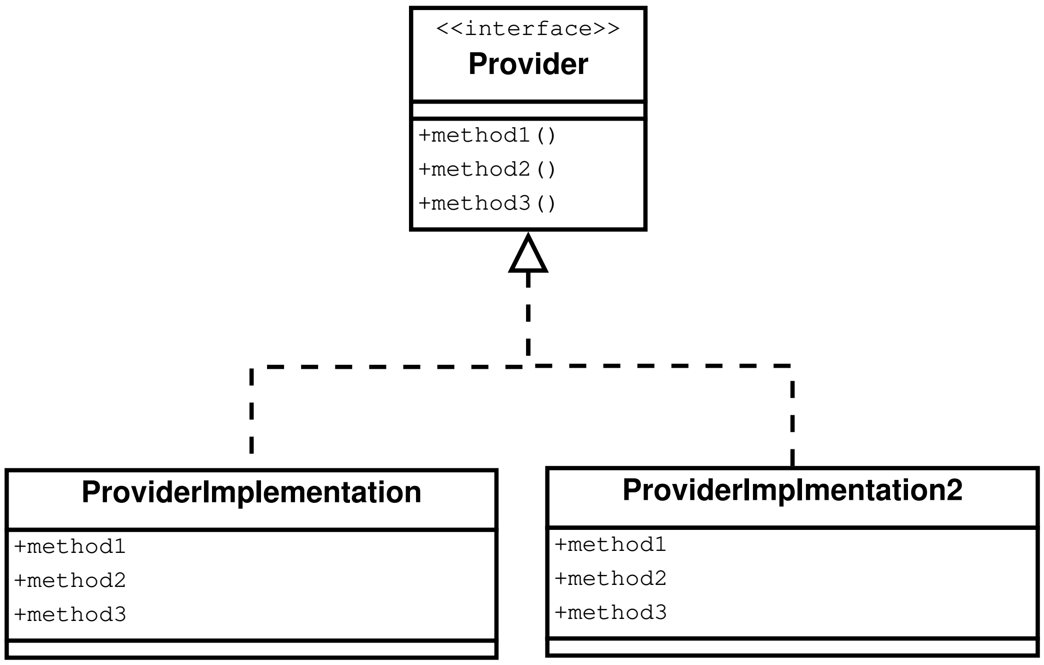

Figure 6.1: UML diagram of the Interface Design Pattern, and the Java implementation

Figure 6.2, demonstrates the definition and implementation of an interface in Java.

```java
public interface Provider {
    void method1();
    void method2();
    void method3();
}

public class ProviderImplmentation1 implements Provider {

    @Override
    public void method1() {
        System.out.println("method1");
    }

    @Override
    public void method2() {
        System.out.println("method2");
    }

    @Override
    public void method3() {
        System.out.println("method3");
    }
```

Figure 6.2: Java Interface definition

The fully fledged Java implementation demonstrated in Appendix A.18, however the essence of the pattern use is, that the client would define the type of the variables referencing the Providers and any potential method arguments that are Providers as the interface and not a concrete class, whilst the implementations could be passed as an argument or set with setter methods. Alternatively the client or user classes could decide which particular implementation to use. An example of that is when using most of the provided Java collections, such as List, Map, Set. The user classes are free to choose, say the ArrayList implementation of the interface List, or HashSet implementation of the Set interface or HashMap implementation of the Map interface. So that in essence the pattern is casting the reference to an object to the interface, as demonstrated in Figure 6.3:

```
Provider provider = ProviderImplementation1();
provider.method1();
provider.method2();
provider.method3();
```

Figure 6.3: Java Interface use

In Python, on the other hand, interfaces do not exist. Thus in principle implementing this pattern explicitly, exactly as specified is impossible. However, the types in Python are governed by the duck-typing mechanism, where if a certain object implements all of the methods of a certain type, then it is deemed to be of that type.

Thus the Interface design pattern could be implemented either implicitly, by implementing all the required methods according to some specifications or documentation. Or by substituting interface implementation with class inheritance. Or even by using Python's Abstract Base Classes and extending (realizing) them in concrete classes instead of implementing an interface.

In principle all of these solutions are valid substitutes for the interface Language Feature in Python, however neither is a exactly matching substitute and in all of the three cases the implementer has to accept a certain sacrifice. In the implicit case the implementer sacrifices the clarity of explicitly defined interfaces in the code and the fact that they would likely be defined elsewhere, either in a specification document or documentation. Figure 6.4 depicts this implementation.

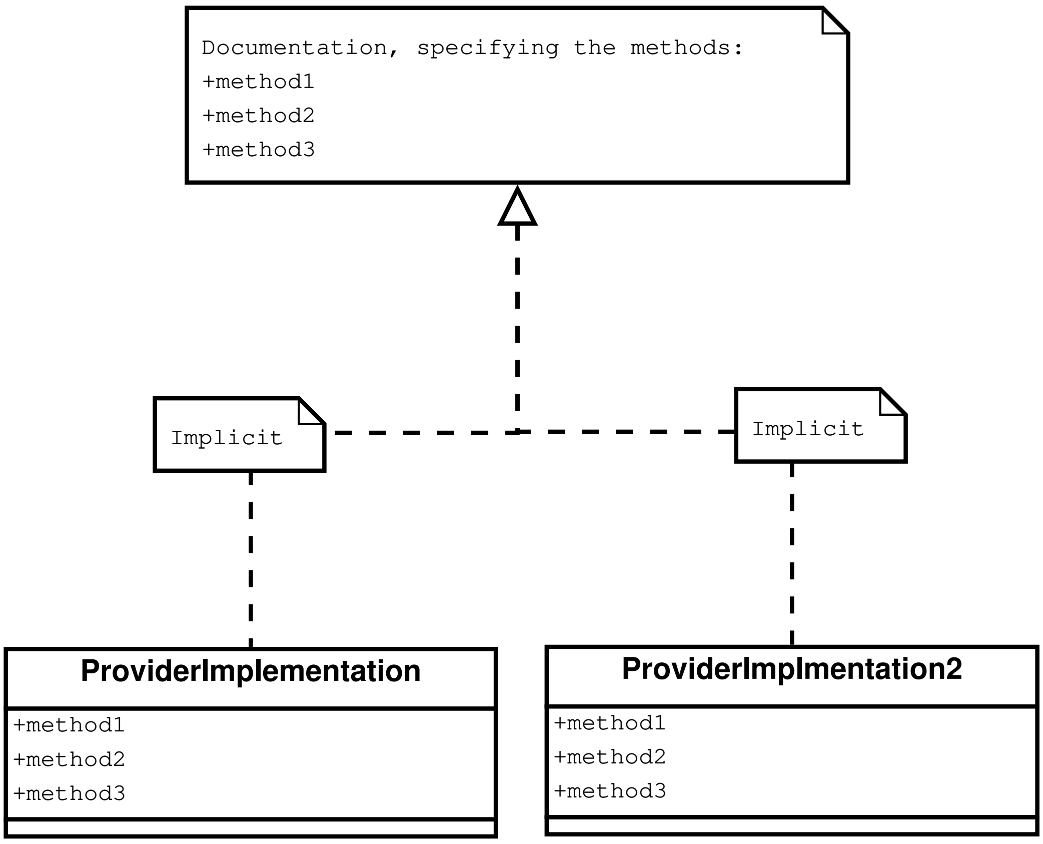

Figure 6.4: UML diagram of the Interface Design Pattern in Python, using duck typing.

The complete implementation in terms of code of the Interface Design Pattern, that uses the duck typing in Python is demonstrated in Appendix A.19. It can easily be seen how this "implicit" specification of the formal interface is an imperfect solution to the problem. First off, the user has to read potentially separate documentation defining the interface. This means that the implementer has to keep up to update one more documentation or design document. In particular, it is a problem that the documentation is very separate from the implementation and the two can easily drift apart.

The second substitute relies on inheritance. Whilst in Java, using up the only super class slot for a class would potentially be ill-advised from extensibility standpoint, Python supports multiple inheritance. So the cost of this implementation is that it will result in an additional super class. Potentially increasing the complexity of the data model.

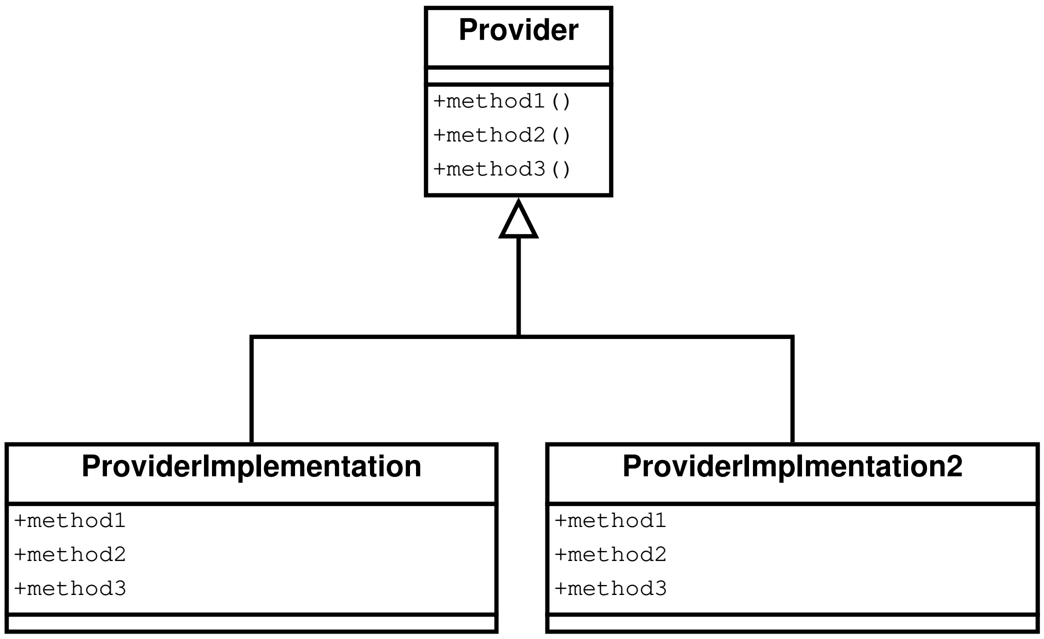

Figure 6.5: UML diagram of the Interface Design Pattern in Python, using inheritance.

In the case of inheritance, the parent class definition could implement the methods to throw an `NotImplementedError` exception. This would communicate to the user that One should extend the class and provide his own implementations.

```python
class  Provider:

    def method1(self):
        raise NotImplementedError

    def method2(self):
        raise NotImplementedError

    def method3(self):
        raise NotImplementedError
```

Figure 6.6: Interface Provider using NotImplementedError in Python

But does not stop the user from instantiating the class. One could also raise an `NotImplementedError` in the super class and override it in the sub class. Raising an exception from the `__init__()` method of the Provider could possibly solve the problem of instantiating the Provider, however, it could potentially cause problems with multi inheritance and other sub-classes. So this method has to be used with care. It could look something like demonstrated in Figure 6.7.

```python
class  Provider:

    def __init__(self):
        raise NotImplementedError
```

Figure 6.7: Interface Provider using NotImplementedError for init in Python

One more problem with raising `NotImplementedError` exceptions is that the sub-classes are not forced to provide implementation of the methods, that raise the exceptions. And unless there is documentation or the implementer sees the definition of the class it is not immediately apparent which methods should be overridden. So again, just as with the previous solution there is an issue of communicating the intent clearly and directly. And the final problem with this solution, in the case that the `__init__()` does not raise an `NotImplementedError`is that, inside the unimplemented methods, raising a `NotImplementedError` is done after the user calls them. Thus it is not a preventative solution, less directly obvious and clear to the user and it could go unnoticed, if, for example, the method is not used.

The third method of substituting Java interfaces in Python is by using the standard Python `abc` module, which as discussed in Section 4.4 provides the Abstract Base Classes like functionality to the Python users. Even though the Abstract Base Classes are meant to be equivalent to abstract classes, they could be used to substitute Java's interfaces in Python. Again it will depend on inheritance and it does add as such it does add complexity to the hierarchy of models in the system.

```python
from abc import ABC, abstractmethod

class  Provider(ABC):

    @abstractmethod
    def method1(self):
        pass

    @abstractmehtod
    def method2(self):
        pass

    @abstractmethod
    def method3(self):
        pass
```

Figure 6.8: Interface implementation using the abc module in Python

The benefits of using the `abc` module are that by decorating the unimplemented methods with the `abstractmethod` decorators, the sub classes are expected to implement these methods. In addition to that an exception is thrown (as demonstrated in Section 4.4), if the sub-class has not overridden the `abstractmethod` methods and the user attempts to create instances of that class. This is an improvement over the previous solution, where the user could create instances of the sub-class that has not provided implementations to the "stub" methods that just raise an exception. So the user is provided feedback that there is something incomplete about the implementation earlier. However, in Java this error is detected and the user is informed as soon as the compiler goes over the class definition. And that happens during compilation. Python, on the other hand, usually is interpreted, so the equivalent behaviour, could have been to raise an exception when the interpreter reads the class definition. However whether this error is severe enough to thwart the interpretation of the script is debatable. As it is Java manages to detect and communicate to the user sooner that the user defined sub-class has not implemented an abstract method during compilation versus Python's on class instantiating.

It is important to point out that all Python implementations share the same Client implementation. Due to the fact that Python is a dynamic programming language, the type of the argument to the Client method, `useProvider(p)` is never checked. An example client implementation is defined in Figure 6.9, where the Provider instance is supplied as an argument to the Client class, using the function `useProvider(provider)`.

```python
class Client:

    def useProvider(self, provider):
        self.provider = provider
        provider.method1()
        provider.method2()
        provider.method3()
```

Figure 6.9: Interface client implementation in Python

There are other ways in which the Provider instance supplied to the Client, such as an argument for the `__init__()` method or alternatively the Clients could know of the ConcreteProviders and pick the preferred implementation. This is all use-case specific.

In conclusion, the Interface Design Pattern is easily implemented in Java, because it provides the Language Features that it relies on. In Python on the other hand, the third proposed substitute implementation is arguably the closest Python equivalent to Java interfaces and consequently implementing the Interface Design Pattern in Python. Whilst not a perfect fit, substituting interface implementation for class extension works. And using the decorators, metaclass and class provided by the `abc` Python module works to communicate the expected interface implementation, though with some oversights as previously discussed.

6.1.3 Abstract Parent Class

Abstract Parent Class is a design pattern similar to Interface design pattern [42], however it is meant to overcome the shortcoming of interfaces that interfaces did not (at the time), in Java, allow the interface writer to specify a default implementation for the methods in the interface. Thus leading to the redundancy of code. Having the same default method implementations repeated over and over again in a project's code-base. However the expense of using abstract classes to define essentially interfaces with default method implementations, is that in Java multi inheritance is not supported. Which means that extending an abstract parent class is taking up the one and only possible parent of that class with interface definition with default methods and variables. The positive side is that the variables do not have to be static and final. And that the abstract class could provide utility methods. Figure 6.10 depicts the UML representation of the Design Pattern, as defined in the Java Design Patterns book [42].

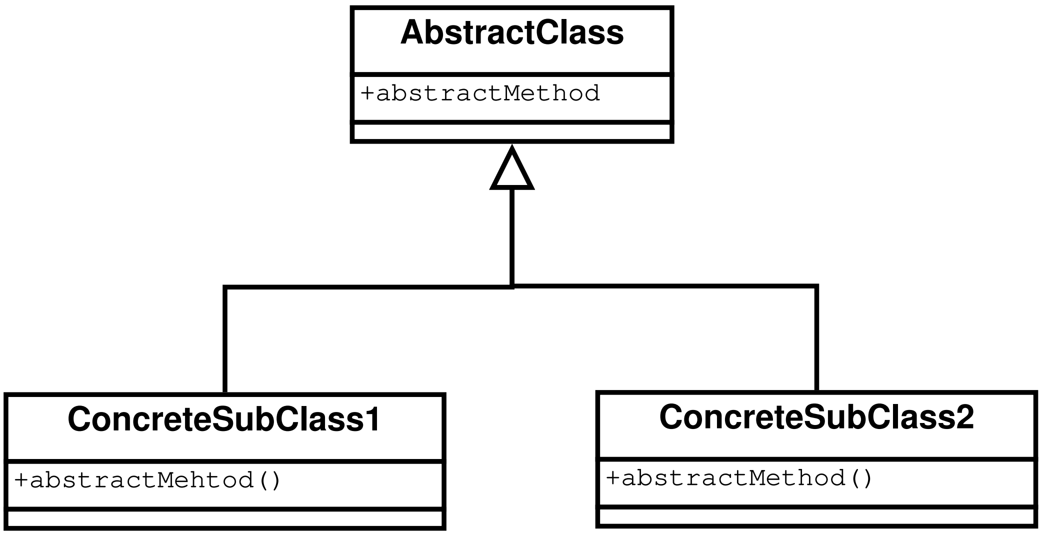

Figure 6.10: UML diagram of the Abstract Parent Class Design Pattern

The Abstract Parent Class Design Pattern in Java is implemented using abstract classes. Figure 6.11 demonstrates the implementation of the Abstract Parent Class in Java:

```java
public abstract class AbstractClass {

    public String variable1 = "AbsClassVar";

    public abstract void abstractMethod();

    public void method2() {
        System.out.println("method2");
    }
}

public class ConcreteSubClass1 extends AbstractClass {

    @Override
    public void abstractMethod() {
        System.out.println("abstractMethod");
    }
}
```

Figure 6.11: An example implementation of the Abstract Parent Class Design Pattern in Java

Abstract classes in Java have the following key characteristics: The abstract parent class cannot be instantiated. Any class that extends the abstract class has to either be an abstract class or implement all of the abstract methods. In addition to that if an abstract class implements an interface it does not necessarily have to implement all methods of that said interface, but it's sub-classes have to provide implementations if the abstract class doesn't.

Python's module `abc`, provides support for abstract methods in the language. It is described in detail Section 4.4 and it is used to implement the Abstract Parent Class in Python, as demonstrated in Figure 6.12.

```python
from abc import ABC, abstractmethod

class AbstractClass(ABC):

    @abstractmethod
    def abstractMethod(self):
        pass

class ConcreteSubCLass1(AbstractClass):

    def abstractMethod(self):
       print("abstractMethod")
```

Figure 6.12: An example implementation of the Abstract Parent Class Design Pattern in Python

Section 6.1.2 already discussed using Abstract Base Classes in Python as substitutes for the interface Language Feature in Java to implement the Interface Design Pattern. The Abstract Base Classes are arguably closer substitute for abstract classes from Java, than the interfaces, as the name suggests. The main question is are they a direct equivalent to the Java abstract classes. And the answer is partially yes, with minor differences, as already described in Section 4.4.

Based on that the Abstract Parent Class is overall quite similarly implemented in both Java and Python, so Figure 6.10, depicts the implementation of both of them, by using the corresponding Language Feature. Abstract classes for Java and Abstract Parent Classes for Python. Arguably, as the name suggests the Python Language Feature has been conceived with the same purpose as the Abstract Parent Class Design Pattern, namely to be used as an abstract parent for OOP modeling.

6.1.4 Interface Compared to Abstract Parent Class

Because the two Design Patterns, Interface and Abstract Parent Class are quite similar, it is worth comparing the Language Features they rely on in the two Programming Languages. This subsection evaluates the two Design Patterns and the Language Features.

Whilst using abstract classes to implement the Interface Design Pattern is indeed silly in Java, given the fact that a specific Language Feature exists, exactly for that purpose, doing the opposite is not inconceivable, due to changes to interfaces introduced in Java 8.

Default methods, introduced in Java 8, help bridge the gap between interfaces and abstract classes in Java. What the Language Feature entails is the ability to provide implementations to the methods within an interface in Java, by using the `default` keyword before the method definition. As such this makes interfaces more flexible and potentially cuts back on code repetition and redundancy. An example of the feature is depicted in Figure 6.13.

```java
interface SomeInterface {
  default public void method1() {
    System.out.println("default implementation of method1");
  }
}
```

Figure 6.13: An example implementation using default methods in Java

This Language Feature became available after the description of the Interface Design Pattern and Abstract Parent Class were published in the "Software Architecture Design Patterns in Java" [42]. As such it possibly provides alternative venue of implementing the Abstract Parent Class, depending on the use-case, by using interfaces and default method implementations. In principle, it puts interfaces closer to abstract classes, due to it's functionality. Table 6.2 depicts the differences between interfaces and abstract classes in Java, in light of the new functionality introduced in Java 8.

Interface	Abstract Class
cannot define constructor	can define constructor
only final static variables	any variables
default methods	non-abstract methods
abstract methods	abstract methods

Table 6.2: Interfaces compared to Abstract Classes in Java

As it can be seen from table 6.2, the major difference between interfaces and abstract classes in Java is that abstract classes can define non-static and non-final variables, as well as defining a constructor. The interesting fact is that, if the user chooses to use interfaces instead of abstract classes for the implementation, then the user is effectively substituting inheritance for interface extension. This is even more important, because of the fact that Java does not support multi inheritance. However the down side of substituting abstract classes with interfaces is the limitations on the implementation. All of the variables have to be static and final and the interface cannot initialize variables during run-time in a constructor as an abstract class could. This in my opinion is a significant limitation, because in my experience a lot of implementations of the Abstract Parent Class utilize the class not only to define the abstract methods and non-abstract ones, but also variables that could potentially be used in the non-abstract methods, or even used by the sub-classes of the Parent.

As such the new interfaces are a nice improvement of the Language Feature in terms of allowing the users more flexibility. In other words the language has adapted to provide better tools or Language Features to it's users, such that implementing a specific design pattern (Interfaces) is preferred over another (Abstract Parent Class) in certain situations. However interfaces are not a complete substitute of the abstract classes. And even though in some cases the Abstract Parent Class Design Pattern theoretically could be implemented using interfaces as substitutes for abstract classes, it would only be a small subset, due to the limitations of the Language Feature. In addition to that, in larger systems with many implementations of the Abstract Parent Class Design Pattern, implementing some, by using interfaces whilst others, using abstract classes would lead to inconsistencies of the implementation and potential confusion for the users of the said implementations. Thus doing so, would potentially degrade the implementation and negatively impact it.

Abstract Base Classes on the other hand, can be used for implementing both the Interface and the Abstract Parent Class. Arguably, the Language Feature is closest to being an implementation of the Abstract Parent Class. And the Interface Design Pattern could be thought of as less applicable due to the duck-typing of Python. On the other hand as already explained in 6.1.2 a definition of type interface in code is beneficial and thus using the Abstract Base Classes of Python to do so is preferred. Thus, whilst Java has a Language Features roughly corresponding to each Design Pattern (Interface and Abstract Base Class), in Python Abstract Base Classes seem like the most appropriate Language Feature that implements both Design Patterns.

The fact is that these two Design Patterns, Interface and Abstract Parent Class solve two different problems. The Abstract Base Class, most often used for use-case modelling, to provide a base, with meaningful default implementations of methods and contain meaningful shared variables for it's sub-classes. Whilst the Interface, used to define abstract specification of the interface of a specific class of classes or simpler said Type. Without specifying any default method implementations or any shared variables. As such the Java 8 extension of it's interfaces in terms of the ability to provide default implementations for the methods of interfaces is substantially less useful, when put into this context. First off, any variable definitions inside interfaces in Java, have to be static and final. Thus, these variables are in reality constants and as such can never change. In addition to that, interfaces cannot have constructors defined. This leads to the fact any default method implementations only have access to constants and reduces the possible spectrum of useful implementations of these default methods. Thus the default methods implementations would be so restricted, that they would be hardly useful, aside from not requiring implementation, when the interface is implemented by a class. Due to these differences in implementation of the Design Pattern in Java in particular, it is common to see the two Design Patterns used in conjunction with each other. Figure 6.14, depicts an uml representation of this particular configuration of use, where an interface is defined and this interface is implemented by an Abstract Base Class, of which actual implementation classes are sub-classes.

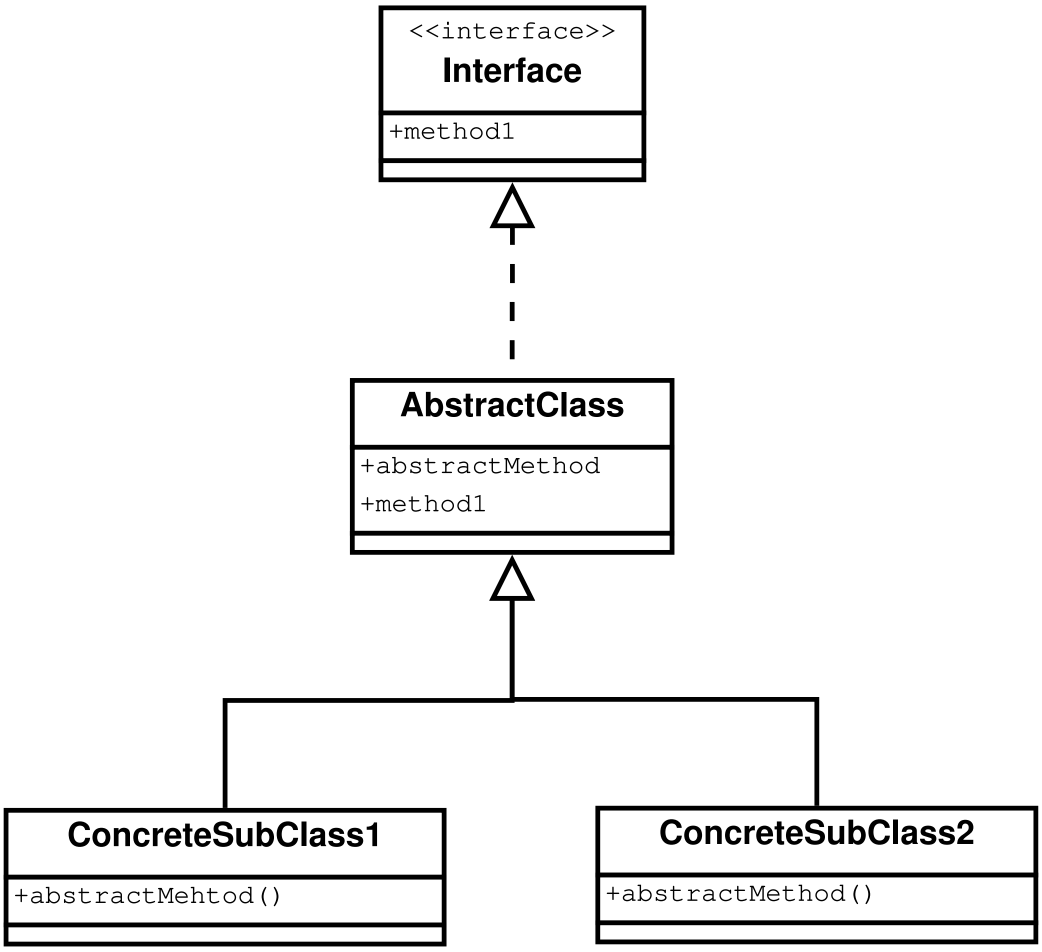

Figure 6.14: UML diagram of Interfaces used in conjunction with Abstract Parent Classes in Java.

As such the user could reference the interface and in effect not know anything about the implementation of that interface, aside from the fact that it is guaranteed. And the sub classes of the abstract class implicitly implement that interface, because their super class does so. And in addition to that the sub-classes of the abstract class are provided with some, where necessary, default method implementations. In addition to which the abstract parent class provides common variable definitions and potentially default values and initialization, depending on the use-case and implementation. In python on the other hand, as described earlier, Language Feature that implements the Interface Design Pattern does not exist. As such the above described collaboration between the two Design Pattern is not possible in the same way it is in Java. At best one Abstract Base Class is used to define the interface, effectively implementing the Interface Design Pattern, whilst another is a sub-class of that, effectively implements the Abstract Parent Class, using Abstract Base Classes. This is a very clear example of Design Patterns implementation and use within the Programming Language, being affected by the Language Features that the Programming Language provide.

6.1.5 Accessor Methods and Private Methods

This Section discusses the rest of the relevant Basic Design Patterns, in both Java and Python, as described [42]. The particular Design Patterns in question are the Accessor Methods and Private Methods Design Patterns. Both the Accessor Methods and the Private Methods rely on member access control in Java. Namely the keywords: public, protected, private and no modifier, as specified in the Java documentation [18].

An implementation of Private Methods can be found below, in Figure 6.15. The implementation, as depicted by Figure 6.15 is trivial, simply using the Language Feature.

```java
public class SomeClass {

    private void privateMethod(){
        System.out.println("PrivateMethod");
    }
}
```

Figure 6.15: An example implementation of Private Methods in Java

On the other hand Accessor Methods requires a bit more of the implementer, but still quite little, as depicted in Figure 6.16.

```java
public class SomeClass {

  private int var1 = 0;

  public void setVar1(int var1) {
    this.var1 = var1;
  }

  public int getVar1() {
    return this.var1;
  }
}
```

Figure 6.16: An example implementation of Accessor Methods in Java

The two requirements of the Accessor Methods in Java, that the implementation usually has to satisfy are: The naming convention, accessor methods are normally named `set` and `get`, followed by variable or attribute name that they set or get. In fact often these methods are called setters and getters. And the variable also should preferably be restricted in terms of access from the outside using the Java member access control [18]. So, either defined as `private` or `protected`. Aside from these two, the pattern implementation is straight forward.

Unfortunately in Python every member of all objects is publicly accessible by design and there is no way of restricting access to object or class members. There is however a convention of appending one or two underscores before and after an internal method or variable name, such as for example: `__init__()`. By following this convention the user of the class, upon seeing the name of the method or variable, is informed that the variables are internal and should not be altered and that the methods are special or internal and should normally not be called directly from the user code. Thus, an equivalent implementation of the Private Methods Design Pattern is arguably impossible in Python. At best the methods that are private could follow the above mentioned convention, but there is no way to actually enforce the member access control and the methods will be visible to all and callable by all, as it is demonstrated in Figure 6.17.

```python
class ContainingClass(object):

    def __privateMethod1__(self):
        print("__privateMethd1__ called")

# Output:
# >>> c =  ContainingClass()
# >>> dir(c)
# ['__class__', '__delattr__', '__dict__', '__dir__',
#'__doc__', '__eq__', '__format__', '__ge__',
#'__getattribute__', '__gt__', '__hash__', '__init__',
#'__le__', '__lt__', '__module__', '__ne__', '__new__',
#'__privateMethod1__', '__reduce__', '__reduce_ex__',
#'__repr__', '__setattr__', '__sizeof__', '__str__',
#'__subclasshook__', '__weakref__']

# >>> getattr(c, '__privateMethod1__')
#<bound method ContainingClass.__privateMethod1__ of
#<__main__.ContainingClass object at 0x7f2c6b7df4e0>>
```

Figure 6.17: An example implementation of the Private Method Design Pattern in Python

As demonstrated in Figure 6.17, the method is accessible from outside of the class and instance of the class and visible, using the built-in function `dir()` [5]. In addition to that one could use the built-in function `getattr(object, name)` [3] to get a specific attribute and even use the built-in function `setattr` [4] to assign another function in it's place, thus altering the object run-time. Thus the Private Methods Design Pattern, whilst not impossible to implement using convention is impossible to enforce, due to the fact that Python does not provide equivalent to the Java member access control Language Feature to it's users.

Accessor Methods Design Pattern [42], if attempted to be implemented in the same manner as in Java, but in Python, has a similar issue to Private Methods, but with the variables that the accessors are supposed to wrap. Because the variable that the Accessor Methods are supposed to expose to the world to modify in a controlled manner is already public, without any real way to enforce access to them other than following a convention. In addition to that, it is more typical and wider accepted in Python to directly access variables, rather than to deal with accessor methods [62]. The property built-in function and decorator as per Python documentation [6] is precisely meant to solve the issue with defining user controlled access to variables. The use of `property` to implement Accessor Methods Design Pattern is demonstrated in Figure 6.18.

```python
class Class(object):

    def __init__(self, var1 = 3):
        self.var1 = var1

    def set_var1(self, value):
        if(value > 0):
            self._var1 = value

    def get_var1(self):
        return self._var1

    var1 = property(get_var1, set_var1)

# Output:
# >>> C = Class(1)
# >>> dir(c)
# ['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
# '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
# '__hash__', '__init__', '__le__', '__lt__', '__module__', '__ne__',
# '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
# '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_var1',
#  'get_var1', 'set_var1', 'var1']
# >>> c.var1 = -5
# >>> c.var1
# 1
# >>> c.var1 = 4
# >>> c.var1
# 4
# >>> property(c.get_var1, c.set_var1)
# <property object at 0x7fca32b07b88>
```

Figure 6.18: An example implementation of the Accessor Methods Design Pattern using property in Python

As it can be seen in Figure 6.18, The `property` built-in function is the best possible way to implement the Accessor Methods Design Pattern in Python. Because, as demonstrated in Figure 6.18, `var1` is not actually the variable that holds the value, rather it is the result of the `property` function call, a property object with a getter, setter, deleter and a document methods. And these methods are automatically called, according to the situation. An example of a more interesting typical use-case of Accessor

Methods in python is depicted in Figure 6.19.

```
1  class Class(object):
2
3      def __init__(self, var1 = 3):
4          self._var1 = var1
5
6      def get_var1(self):
7          return self._var1
8
9      var1 = property(get_var1)
10
11 # Output:
12 #>>> c.var1 = 5
13 #Traceback (most recent call last):
14 #  File "<stdin>", line 1, in <module>
15 #AttributeError: can't set attribute
```

Figure 6.19: An example of how the implementation of the Accessor Methods Design Pattern using property in Python would restrict access to variables

Why does the call `c.var1 = 5` in Figure 6.19 fail ? Because, first of all, the `var1` that the user is assigning a value to isn't actually the variable that holds the value. The variable that actually holds the value of interest is named `_var1`, as seen in line 4 and 7 in Figure 6.19. So, somehow the call to the built-in `property` function, maps the assignment to `var1` to the getter provided in the `property(getter, setter, deleter, doc)` call. And likewise for the setter, deleter and the documentation for the property. However on the other hand, `var1` is clearly listed in the `dir(c)` call result in Figure 6.18. Thus however this is technically implemented the end result for the user is the definition of a `var1`, which acts as an interaction end-point for the user, instead of the actual variable `_var1`. Thus resulting in an adequate implementation of the Accessor Methods Design Pattern, in Python, which does not technically support the same Language Feature for manipulating members access of objects and classes. Yes the original variable, `_var1` is still accessible from the outside of the object and class and there is still no way to enforce no access to it. However, if Pythonic conventions with regards to naming schemes are followed, (as they usually are in most languages and most languages have their own conventions), then the Accessor Methods Design Pattern implementation is adequate to the Design Pattern initial intent, to control access to object variables trough accessor methods. It has been affected though, quite heavily, in a sense that without using the `property` Language Feature and the variable naming convention, the implementation suffers from potential obscurity with regards to which variables should not be modified from outside the object as well as being less typical or accepted implementation (less Pythonic) within the Python programming language.

6.1.6 Conclusion

It could be argued that the Basic Design Patterns are nothing but descriptions of use of the language provided features. However, it also be argued that these design patterns are a consequence of well implemented Language Features that influenced or are one of the main causes of the Design Pattern existence, since they are heavily reliant on the discussed in this chapter Language Features. On the other hand, Python does not provide implementations of the Basic Design Patterns, such as Interface or Private Methods. And the implementation of these Design Patterns is not quite straight forward as discussed in this Section. In-fact, as it was found, the Interfaces and Abstract Parent Class, used in conjunction with each other could lead to a difficult situation, since the same Language Feature could be used to implement both Design Patterns. As such Java, provides the better medium, trough it's Language Features, to use these two Design Patterns in conjunction to each other in a productive manner. Accessor Methods, on the other hand as discussed in Section 6.1.5, whilst adequately applicable to Java, are more invisibly and naturally implemented with regards to ease of use and adaptability in Python.

All of the above-discussed Design Patterns show that the programming language, with the Language Features that they provide, can and do significantly affect the Design Pattern implementations in said languages, due to the fact that these Design Patterns are very basic in nature and often used by other Design Pattern implementations.

6.2 Decorator

The Decorator Design Pattern, as defined by GoF [31] and as implemented in Java, can be found in Figure 6.20. In principle the Decorator Design Pattern is supposed to either using inheritance override the decorated methods or using composition to implement the same interface or abstract class (if either of these is provided by the Programming Language in question) to decorate the original class's methods. However, in Python decorators are a bit different.

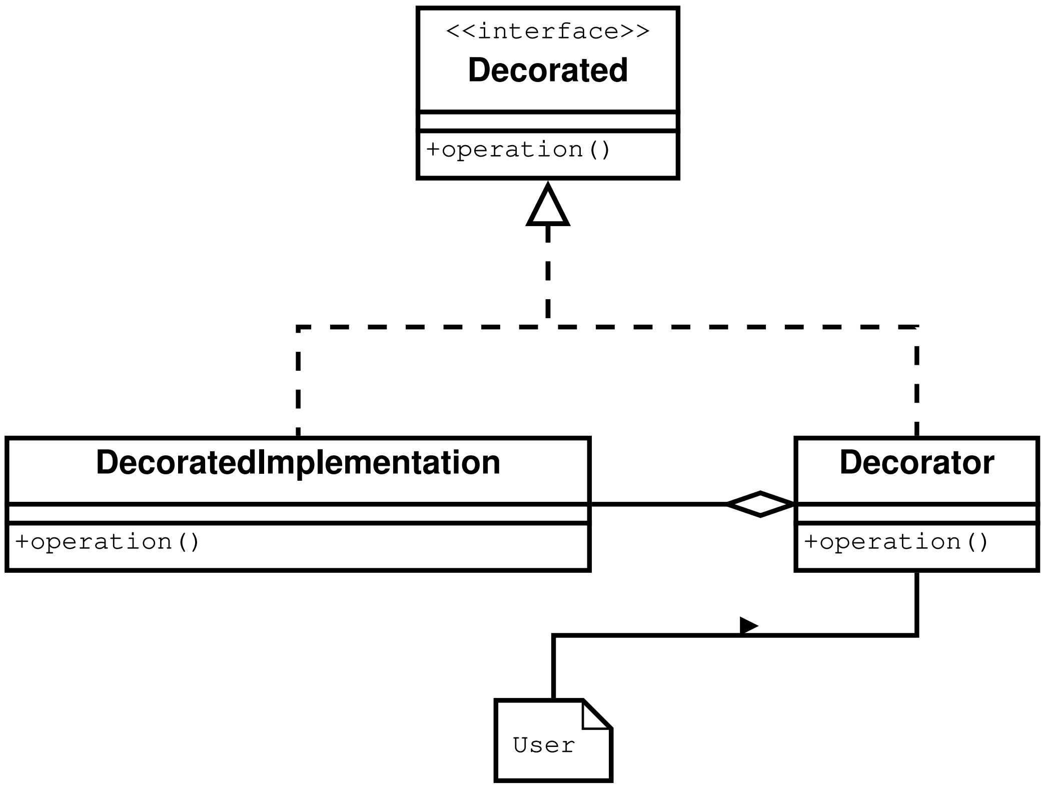

Figure 6.20: Extended UML diagram of the Decorator design pattern

As explained in Section 4.5, the Python's implementation of decorators work primarily on functions. Thus the only relevant to Object Oriented Programming case is when one would like to decorate the constructor of a class, since that is what the result of attempting to decorate a class with a function or another class in Python.

Due to the fact that everything in Python is an object, the Python's implementation of decorators could be represented as it is in Figure 6.21. Where an instance of the decorating class is returned, which is an object, that implements `__call__()` or a function, which as well is an object. Thus the DecoratingClass's instance could be a function object as well as an instance of a class and the instance of the DecoratedClass is always a function object. The decorating object returns another wrapper object (or a function), that in terms is the one executed when normally the constructor of the original (decorated) class would be. And the decorating object returns the result of the original (decorated) class's constructor. This whole process is invisible to the user of the class. As the instance returned is of the decorated class and the user cannot directly discern that the original class's constructor has been decorated, since it executes as normal. However, in addition to the normal behaviour of the original class, the decorator could execute custom operations before and after the original class's constructor is executed.

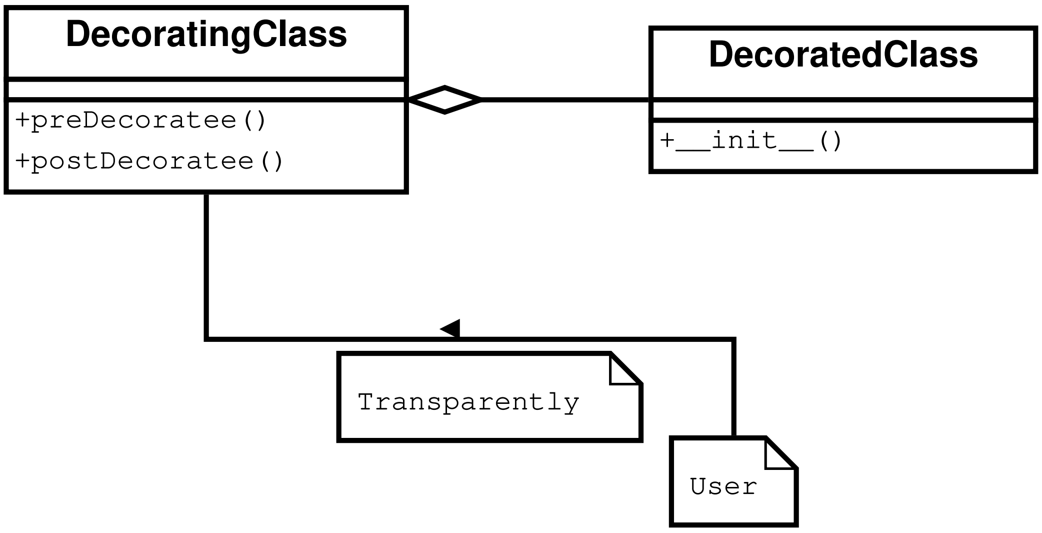

Figure 6.21: UML diagram of Python's implementation of the Decorator design pattern

6.2.1 Conclusion

The Python implementation of the Decorator Design Pattern, could be thought of a special case implementation of the Design Pattern, that is primarily meant to be used in functional programming context. Because using a class to decorate a class or a function does not seem to offer that much more than using a function, aside from the ability to abstract sub-operations in functions and hold data in the object of the class or even as class variables. Also because, in order to use a class as a decorator, the class would have to be of the "type" function, by implementing `__call__()`, such that the result of the call: `callable(obj)` is True. Also because when decorating a class, in reality only the `__init__()` method is decorated. And all this is possible because of the dynamic typing in Python in combination with the Decorator Python Language Feature implementation.

6.3 Iterators

This Section draws comparisons between the languages' implementations and evaluates of how the languages affect the implementations of the Iterator pattern, in terms of what they provide to the language users.

Whilst Iterators are technically provided, the language user still has to implement classes in a specific way in order for iteration with it's language integration to work. Thus the languages facilitate easier Design Patternimplementation at the cost of users following the pattern as described by the languages.

Both Java and Python, provide their own interpretation of the Iterator Design Patternas the interface of Iterators in the language. An UML diagram of the Iterator Design Pattern as defined by the GoF [31] can be found in Figure 6.22.

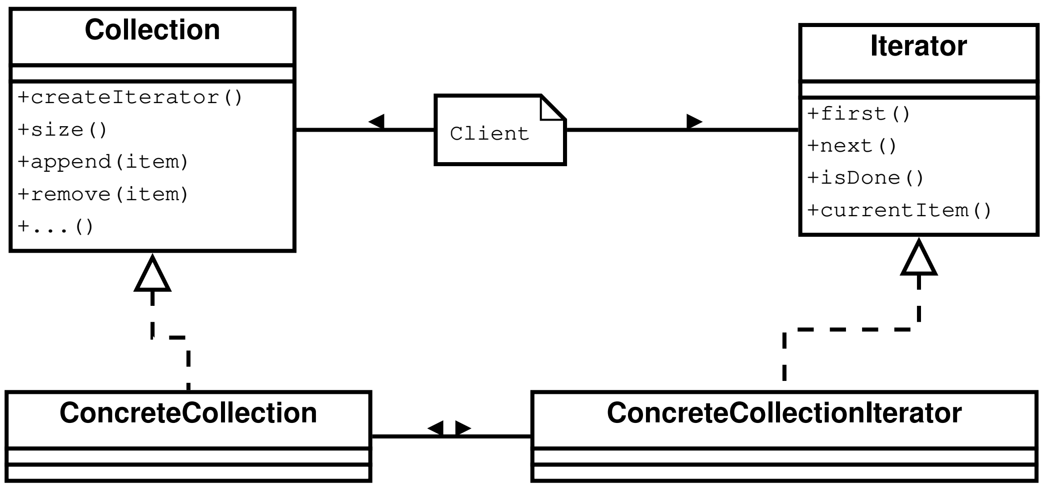

Figure 6.22: UML diagram of the Iterator design pattern

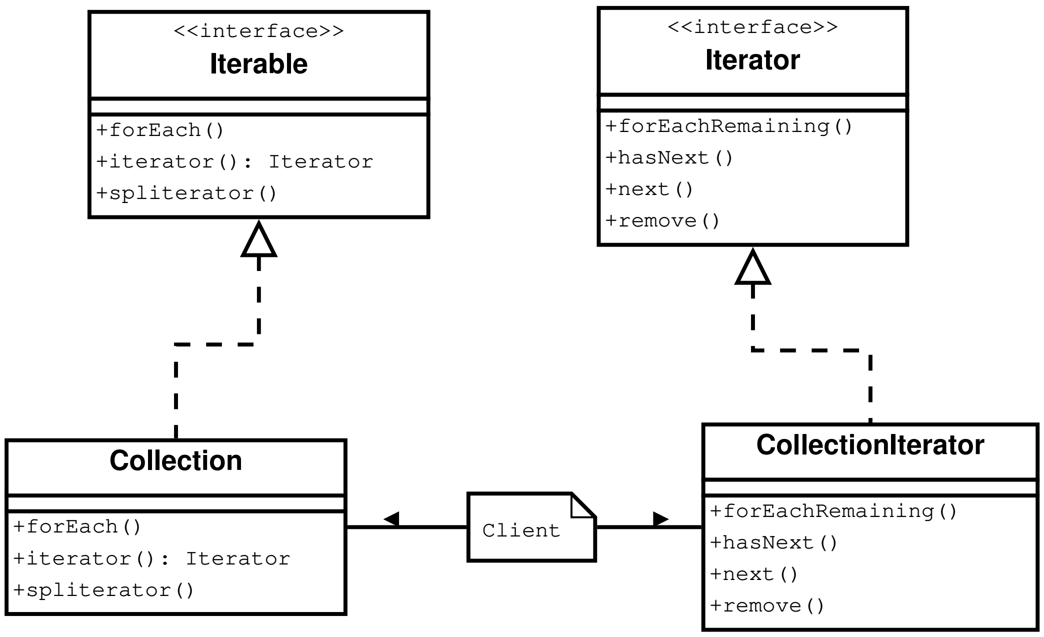

Figure 6.23: UML diagram of Java's implementation of the Iterator design pattern

Both Java and Python have imposed a change on the terminology with regards to the elements of the Iterator Design Pattern. Namely instead of using "Aggregate" they use Iterable. The renaming is a good thing because Iterable caries more meaning and context than Aggregate, one clearly can understand that the object in question can be iterated on and that the interface/object is related to iterators.

Because, the release of Java 8, users can use the so called "passive" also known as "internal" iterators in the form of the `.forEach()` method, described to greater extend in the excellent article in JavaWorld [35]. By providing lambdas and the `.forEach()` method, to some degree Java facilitates functional programming in the Programming Language. As such the functional programming variant of the Iterator Design Pattern, namely internal Iterator, is possible to be utilized. Python on the other hand, does not directly support this in specialized syntax and the user is expected to use the `for` loop syntax or alternatively use the `map` functions.

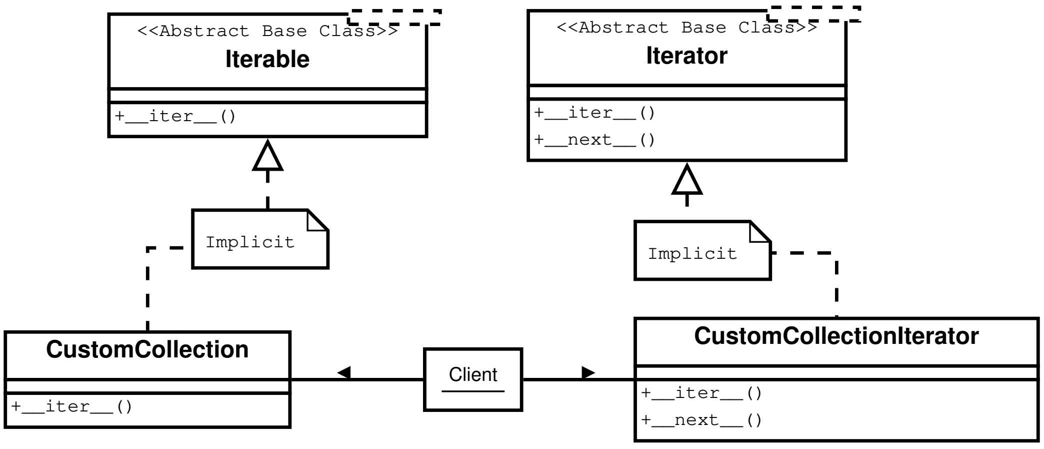

Figure 6.24: UML diagram of Python's implementation of the Iterator design pattern

As previously mentioned the Python's implementation of the Iterator Design Pattern does not include the `__hasNext__()` method and the `__next__()` throws an exception when the iterator has been exhausted. Because, the user's of the language have a good incentive to follow the interface defied by the language, as demonstrated in Section 4.6, then they would typically not implement a hasNext() method either. Thus the language has affected the implementation of the Iterator Design Pattern, by specifying an alternative simplified one and providing an incentive for the users to the users to implement it as such, by providing better language integration of the implementations.

However the choice of using exceptions instead of implementing the `hasNext()` function is problematic. First off all, it is contrary to the intended use of exceptions. The end of an iteration is not an exceptional state at all. Rather more of an inevitability, due to the fact that computer memory is finite.

Another drawback of using exceptions to signify the end of the iteration is that in principle throwing an exception, to signify the end of an iteration, is in theory more computationally expensive [25], when compared to a function call and boolean check, as explained in Section 4.6.

6.3.1 Conclusion

The new lambda functions in Java and the `forEach()` methods are example of the Programming Language enabling the users of the language, by providing a feature that allows the functional programming oriented variant of a Design Pattern to be used. This is another example of the Programming Language evolving, by providing lambda functions and as a result of it's evolution affecting a Design Pattern in terms of enabling the use an alternative variant of that Design Pattern. Both Java and Python provide similar enhanced for loops, which in both Programming Languages depend on the iterators Language Features. This is an example of a Design Pattern implementation, that is deeply ingrained into the Programming Language, such that other related Language Features that the Programming Language provides, such as the enhanced for loops, depend on the Language Feature that is an implementation of the Iterator Design Pattern.

6.4 Observer

As described in chapter 5.3, Java implements the Observer design pattern, since version 1.0, released in 1994. Python on the other hand does not implement the Observer design pattern and has left the users of the language to implement it themselves, from scratch.

Figure 6.25 depicts the Observer design pattern as defined by the GoF book [31].

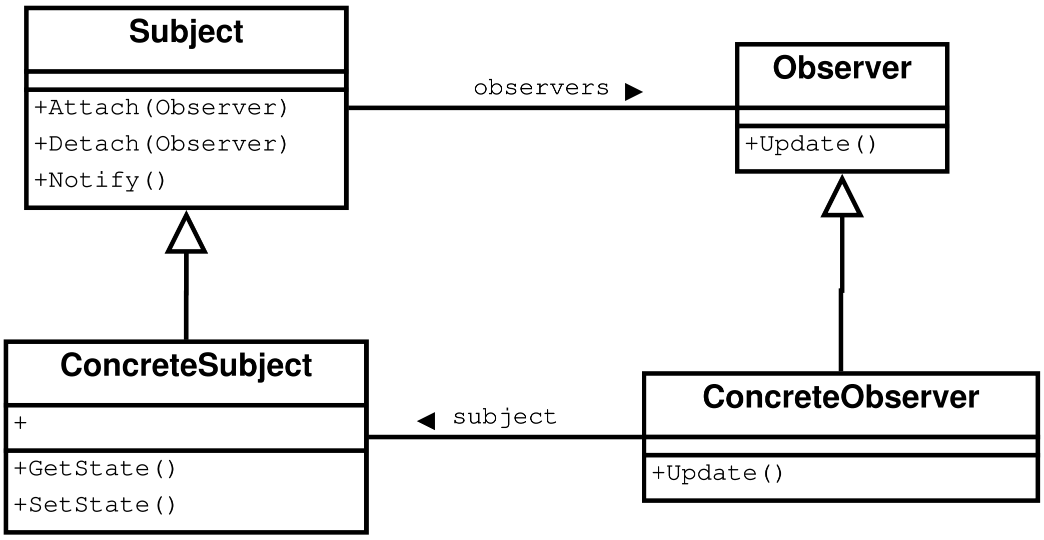

Figure 6.25: UML diagram of the Observer Design Pattern

Figure 6.26 depicts an UML representation the Observer design pattern implementation in Java.

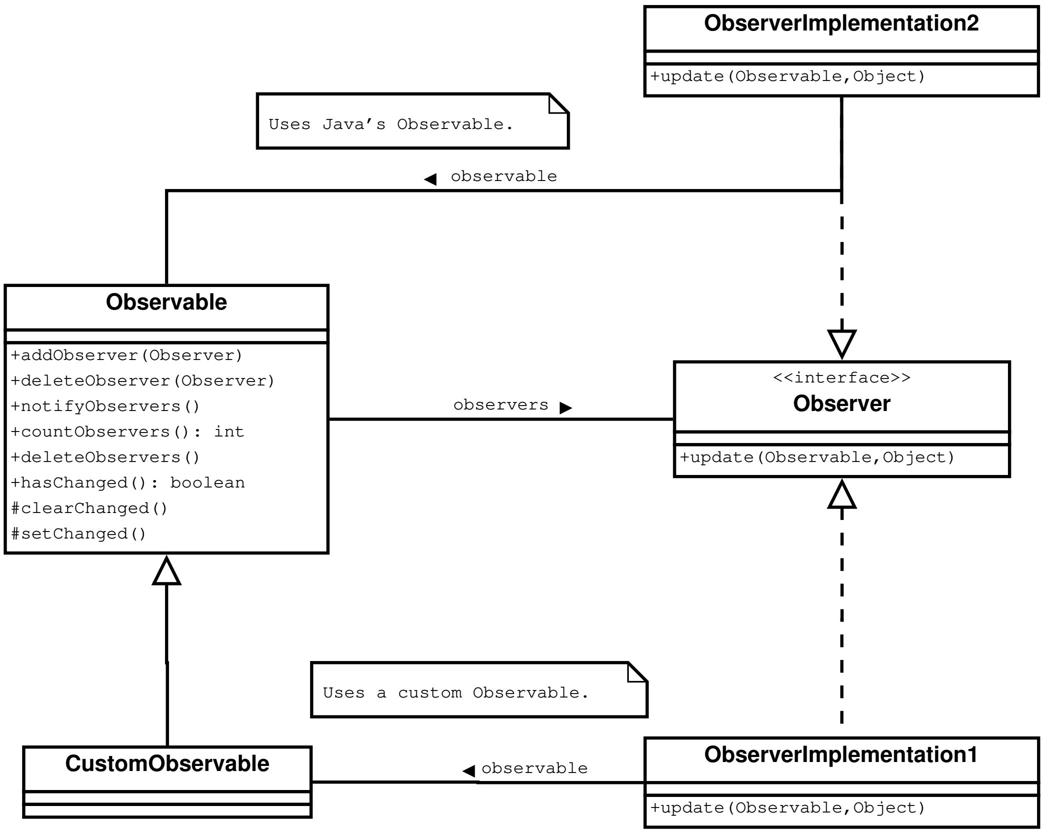

Figure 6.26: UML diagram of the Observer Design Pattern in Java

As it can be seen from Figure 6.25 and 6.26, the Java implementation follows closely the Observer design pattern definition. The differences are as described in Section 5.3 mostly cosmetic renaming of methods and the fact that instead of the ConcreteObserver inheriting from the class Observer, in Java the ConcreteObserver implements the interface Observer. Overall this is the only major affect that Java has on the Observer design pattern implementation in the programming language. Python, on the other hand does not implement the Observer design pattern. However a typical implementation could mirror the GoF [31] definition closely, using inheritance just as the Design Pattern defines. In addition to that the Python implementation could define Abstract Base Class for the Observer implementation, as described in Section 6.1.2. However, that implementation would be inspired by Java and the fact that Abstract Base Classes, as shown in Section 6.1.2 are a possible substitute for interfaces, since Python lacks that Language Feature. Thus such implementation in Python is possible, but unlikely.

6.4.1 Conclusion

Java implements the Observer Design Pattern as a Language Feature, whilst Python does not. In principle, due to the popularity of the Reactive X frameworks in both languages, which provide implementations for Observers and Subjects it is hard to assess whether

6.5 Object Duplication

Object duplication, replication, copying or even cloning are all synonymous words that could be used to refer to the same concept in object oriented programming. Namely the concept of creating a duplicate or a copy of an existing object in memory, resulting in two objects with the same data. Many Programming Languages provide means of duplicating objects in their standard libraries, examples of that are Java and Python. Both Programming Languages implement their own interpretation of object duplication. The implementations differ from one another, as they are in two different programming languages, probably influenced by the standardized ways that these programming languages implement Language Features. Also probably influenced by the community surrounding these programming languages.

6.5.1 Copy Method and Copy Constructor

At a conceptual level, there are two typical ways of implementing object duplication. In the form of a copy method or in the form of a copy constructor, such as in C++ [69]. In principle, a constructor in Object

Oriented Programming is a method, just a special one, that is tasked with initializing or constructing an object of the given class and returning a reference to it. In the first case, a method would be defined, called `clone()` or `copy()` or even `deepcopy()`. This method, when called on a reference of an object would return a separate instance of the class, which is a duplicate of the object. In the second case, the case of a copy constructor, one would call the method class with a reference to an object of the class. Then that constructor would return a separate instance of the class which is a duplicate of the object. Both Python and Java support the first method out of the box and the user is free to implement the copy constructor if desired.

6.5.2 Shallow and Deep Copy

In addition to that as previously discussed there are two basic ways to duplicate an object. The first is to return a shallow copy, where references to other objects in the original are references to the same objects in the duplicate. And the second method is called deep copying, where the references to other object in the original are copies of these objects in the duplicate. In both languages, the user can implement a custom duplication solution by overriding the provided by the Programming Language methods. One reason for implementing a custom solution could be that the particular use case requires a hybrid between deep and shallow copy scheme that must be custom to the models of the system.

6.5.3 Use of Object Duplication for Design Pattern Implementation

Why is Object duplication a relevant concept to the topic? In short, because it can be used to implement the Prototype design pattern. By using Object Duplication, the user does not have to re-implement the copying parts of the code for the most part and in the most general case where only a shallow copy is necessary.

Thus it can be said that it is an implementation of the design pattern in practice, event though in theory the Prototype design pattern and Object Duplication are different, but related concepts. They are related, because they specify similar behaviour, namely using one object as a template to create copies of that object. Figure 6.27 shows the Prototype Design Pattern, as defined in the GoF book [31].

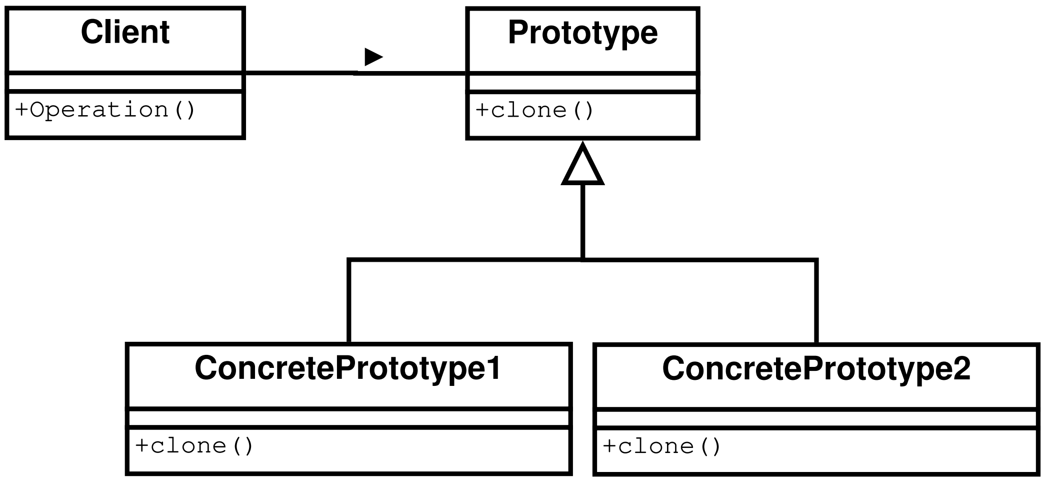

Figure 6.27: UML diagram of the Prototype Design Pattern

Cloneable, in Java, in practice is the implementation or the implementation of the Programming Language's interpretation of the Prototype Design Pattern. As described in Section 5.1, it requires the user to both override the `clone()` method from Object and implement the Cloneable interface.

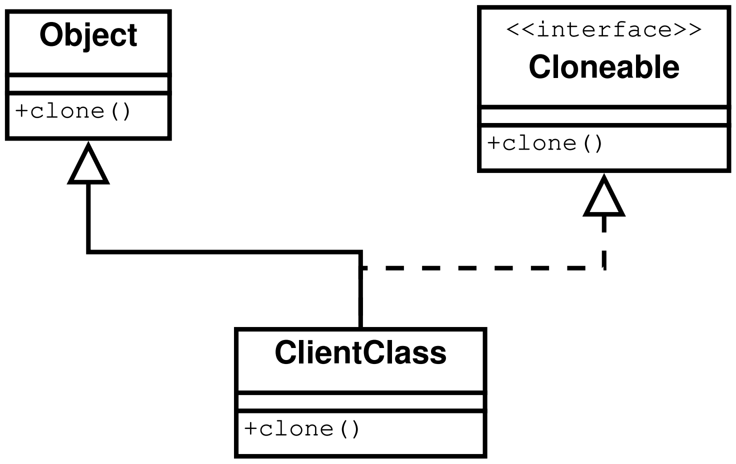

Figure 6.28: UML diagram of object duplication in Java

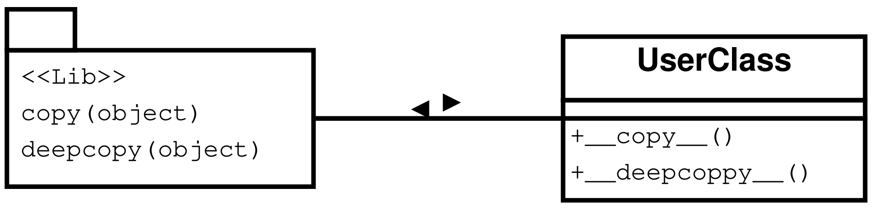

Figure 6.29: UML diagram of the copy methods in Python

The Java implementation of object duplication as depicted in Figure 6.28 differs from the Prototype Design Pattern by only the fact that any class that overrides the `clone()` method has to implement the empty Cloneable interface. Aside from that difference, as it can be seen from the Figures 6.28 and 6.27 it is very similar to the Prototype Design Pattern. In that it uses inheritance to override the method defined in the super-class, just like the pattern defines.

The Python implementation, as depicted in Figure 6.29, differs from the from the Prototype Design Pattern quite a bit more on the other hand. First, the copy methods are defined inside the standard library module copy. Second the methods or functions are stand-alone, meaning they are not part of an object called copy, but part of the module copy. And they receive a reference to the object to copy, rather than have no arguments as the Prototype Design Pattern and the Java implementation define. In addition to that they look into the attributes defined by the object to copy, to determine weather either of the `__copy__()` or `__deepcopy__()` methods are defined. And if the object to be copied implements these methods, the object implementations are used instead. This approach is quite different from the inheritance and method overriding one, that is specified by the Prototype Design Pattern and in the Java implementation. Instead of inheritance it uses composition in a functional programming context. Nevertheless the structural difference in the implementation, it could still be said that

6.5.4 Conclusion

Due to these similarities and differences between both implementations provided by Java and Python, versus the Prototype Design Pattern, as described by the GoF [31]. It could be argued that the Programming Languages implements the Prototype Design Pattern, if not in theory, then in practice. Because, if one were to try to implement the Prototype Design Pattern in these languages, the base for the pattern is already implemented and available. And if parts of the Design Pattern are missing one could argue that the implementation provided by the language is as interpreted or as chosen to be implemented by the Programming Language. And this would be how the Programming Languages affect the Design Pattern implementation in said Programming Languages.

6.6 Language Features Overview

This Section summarizes the Language Features used in the implementations of Design Pattern's.

A somewhat interesting and useful view could be assembling a timeline of related to Design Patterns, Language Features.As such there are plenty of sources stating the version release history of the languages considered in this thesis. For Python for example the Guido van Rossum's blog [65], as well as the Wikipedia article [33] on the subject, provide a table of the Python versions and release dates. For Java on the other hand, the Java Papers blog, by Joe [38], as well as the Wikipedia article on the subject [33], provide a table of the Java versions and release dates.

Based on these and the findings presented so far in this thesis, a timeline of the discussed and related to the Design Patterns Language Features in both Java and Python can be composed as depicted in table 6.3 and table 6.4. Of which, table 6.3 depicts the introduction of the relevant Language Features to Design Pattern implementations over the course of version releases and time in Java and the equivalent Python table is by it's side, table 6.4.

Language Feature	Java Release
Interface	JDK 1.0 1996
Abstract Class	JDK 1.0 1996
Cloneable	JDK 1.0 1996
Observer	JDK 1.0 1996
Iterator	JDK 5 2005
Default Functions	JDK 8 2014

Table 6.3: Overview table of relevant Java Language Feature releases.

Language Feature	Python Release
Object Copying	1.0 1994
Magic Methods	1.0 1994
Metaclass	1.5 1998
Iterator	2.1 2001
Decorator	2.4 2004
Decorator	2.4 2004
ABC	v3 2007

Table 6.4: Overview table of relevant Python Language Feature releases.

Table 6.5, on the other hand, summarizes the Design Pattern implementations and what they rely on as described in the GoF book [31]. By listing this summary information, it is not meant to categorize Composition, Inheritance, Aggregation and Delegation as Language Features. But, rather to make possible the comparison of what Language Features possibly substitute for them, when discussing the Language Features, in addition to having a clearer overview of the Design Patterns.

Design Pattern	Description
Factory Method	composition
Abstract Factory	composition
Builder	composition
Prototype	composition
Singleton	composition
Adapter	inheritance
Bridge	delegation
Decorator	composition / inheritance
Composite	
Facade	aggregation
Flyweight	aggregation
Proxy	aggregation
Chain of Responsibility	delegation
Command	aggregation
Interpreter	
Iterator	
Mediator	delegation
Memento	
Observer	
State	delegation
Strategy	delegation
Template Method	
Visitor	delegation

Table 6.5: Overview of language mechanism versus Design Pattern

Table 6.6 summarizes the Language Features that Design Pattern implementations in the given language depend on. Based on both the findings of this thesis and the documentation and books that are referenced by it. By listing the Language Features and Design Pattern in that way it is not meant that, the implementations have to use the particular Language Feature. It is not an absolute requirement to do so in order to have an adequate Design Pattern implementation. But, rather that since the Language Feature is provided by the said programming language and aids the implementation in that particular language in some way, it might as well be used for the implementation, rather than not.

Design Patterns versus Language Features:

Design Pattern	Java	Python
Interface	Interface	Abstract Base Class
Private Method	Member Access Control	
Accessor Methods	Member Access Control	Property
Factory Method	Interface	
Abstract Factory	Abstract Class, Interface	
Builder	Interface	
Prototype	Clonable Interface	Copy Deepcopy
Singleton	Static Methods	Metaclass Abstract Base Class Decorator
Adapter	Interface	
Bridge	Abstract Class, Interface	
Decorator	Interface	
Composite	Interface	
Facade		
Flyweight	Interface	
Proxy	Interface	
Chain of Resp.	Abstract Class	
Command	Interface	
Interpreter	Interface Abstract Class	
Iterator	Java implementation (Iterable, Iterator Interfaces)	Python implementation (Abstract Base Class)
Mediator	Interface	
Memento		
Observer	Java implementation (Interface)	
State	Interface	
Strategy	Abstract Class, Interface	
Template Method	Abstract Class, Interface	
Visitor	Interface	

Table 6.6: Design patterns versus what Language Features they depend on.

As it can be seen from the table 6.6, most design pattern implementations in Java depend on using the interfaces to define a precise concise contract with the user, an interface definition in other words. The blank Java Design Patterns are not affected by any specific Language Features that the language provides. The Design Patterns implementations in Python on the other hand, where the Java versions use Interfaces or Abstract Classes, could very easily be implemented using Abstract Base Classes as substitutes for the Interface or Abstract Class, as demonstrated in Sections 6.1.2 and 6.1.3. Most of the Java entries in this table are based on the "Software Architecture Design Patterns in Java" [42]. And the Python ones, roughly on "Mastering Python Design Patterns" [40] as well as the findings from the previous chapters.

Another fact of importance is that the provided Design Patterns implementations in Java, which also depend on the interfaces, since they are interface definitions, whilst the internal to the language implementations are classes adhering to these interfaces. And in Python the feature implementations rely on Abstract Base Classes to define their interfaces to the users of the language and Language Features.

6.7 Results and comparisons graphs

Based on the impact of the Language Feature on Design Pattern implementations, one could compose a scoring and consequently categorization scheme. Where the provided by the two programming languages in question (Python and Java) Language Features of interest are evaluated with regards to: First if the feature implements a Design Pattern, the implementation. And second if the feature helps facilitate implementation or affects the implementation of design Patterns.

Table 6.7 depicts an overview of the Language Features and their score. This thesis defines a scoring scheme for Language Features, in relation to Design Pattern implementations as the following scoring criteria: To be part of the table the Language Feature has to be relevant to Design Pattern implementations. A two stars score "**" to Language Features, that Design Pattern and provided to the users, default implementations use it as well. A three star score "***" to Language Features, that affect more than one Design Pattern implementation, including a highly related Design Pattern. A five star score "*****" to Language Features, that impact most design patterns implementations.

Language	Feature	score
Python	Abstract Base Class	*****
	Property	***
	Copy, Deepcopy	****
	Decorators	***
	Iterator	**
	Metaclasses	****
Java	Interface	*****
	Abstract Class	*****
	Member Access Control	***
	Iterator	**
	Cloneable	**
	Observer	**

Table 6.7: Programming language features, examined in relation to Design Patterns.

As depicted in table 6.7, Interfaces in Java as well as Abstract Classes score highest, as they potentially affect most if not all Design Pattern implementations in the language. Also as seen from table 6.6. Similar to that, as the equivalent of Abstract Classes in Java, Abstract Base Classes of Python also achieve the top score. These three Language Features deserve the top score, because of their universal applicability to all Design Pattern implementations. Further down the list in Python are Copy, Deepcopy and Metaclasses, with four stars.

6.8 Life Cycle of a Design Pattern

This Section proposes and describes and proposes a theory of what the life cycle of Design Patterns is in relation to Programming Language Language Features. That theory is based on observations made while writing and compiling information for this thesis. Along with being inspired by the work of Peter Novrig [27] with regard to Design Patterns and Programming Language Language Features analysis in Dynamic Languages. Figure 6.30 graphically depicts the possible ways in witch Design Patterns could potentially over time evolve in relation to Programming Languages.

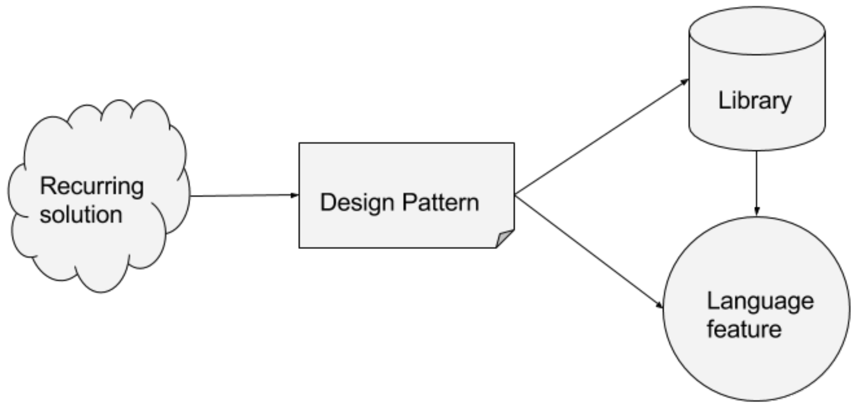

Figure 6.30: Design Pattern adoption path

In short, the theory regarding the life of a Design Pattern attempts to capture the potential evolution of Design Pattern. And it could be described as follows. Naturally a Design Pattern is extracted, abstracted solution to a specific type of problem, distilled after potentially multiple sightings. Thus, this abstraction becomes a Design Pattern. Just as it was done with the first 23 Design Patterns, described in GoF [31]. From there, the Design Pattern could be implemented as a library or a framework or even directly implemented as a Language Feature in a said Programming Language. Examples of that are the previously discussed in this thesis Language Features. The abstractions in libraries or frameworks in term, could be incorporated into a Programming Language. If the framework or library in question is popular enough and easy enough to be incorporated.

6.9 Classification of Language Features

This Section proposes a classification scheme for Language Features in relation to Design Patterns, inspired by the classification, described in Peter Norvig's examination of Design Patterns in Dynamic Programming Languages [27]. In particular page 7 of his presentation, where the Design Pattern implementations are graded in terms of "Level of implementation of a Pattern" one of three categories: "Invisible" - So much part of the programming language that it is indistinguishable , "Informal" - Referred to by name but must be implemented from scratch and "Formal" - have to be implemented by user.

Whilst these tree categories work for the discussion in question, they are not as precise as they could be with relation to this thesis. And their naming scheme, whilst fine for the discussion they belong to, isn't quite suited or applicable to the discussion in this thesis. As such, defining a new classification,still inspired by that classification, but based on the current context is in order.

6.9.1 Classification

Table 6.8 demonstrates the three classes or states of features.

Language	Feature	
Python	Invisible	Decorators
	Invisible	Property
	Invisible	Metaclasses
	Partial	Abstract Base Class
	Partial	Copy, Deepcopy
	Partial	Iterator
Java	Invisible	Interface
	Invisible	Abstract Class
	Invisible	Member Access Control
	Partial	Iterator
	Partial	Cloneable
	Partial	Observer

Table 6.8: Programming Language features, classification in relation to Design Patterns.

There are two classes in the new classification: Invisible and Partial. An Unimplemented classification doesn't make sense, since if a Language Feature isn't implemented then it doesn't exist.

6.9.2 Invisible

Invisible - The similar to as described by Peter Norvig [27]. Language Features that are ready to use out of the box for the user. The following Language Features are classified as Invisible:

Member Access control (section 6.1.5), in Java the keywords, used to control access to members of classes (and packages for that matter) are `private`, `public` and `protected`. To any programmer that is familiar with the language is also familiar with them. The `abstract` keyword as well. It represents a concept that is part of the language. As well as the `interface`, it represents defining an Interface, just like `class` defines a class.

In Python, the Language Features that belong to this category are Metaclass (section 4.3), which are part of the language as well and are specified by using the `metaclass=x` in the class definition. Decorators (section 4.5) and Iterator (section 4.6) as well. Using the Java annotations symbol and placement, but to decorate functions and classes for example `@abstractclass` is consistent with the rest of the language. Property (section 6.1.5), present in the form of both a function and a decorator is also very consistent and ready to use out of the box.

All of these Language Features belong to the classification as Invisible, because they are highly integrated in the language, in other words they are indistinguishable part of the language, often as usable keywords by the user and the user does not have to include or import any modules or packages. They are there.

6.9.3 Partial

Partial - Meaning the Language Feature implementation is complete, but the users have to put in effort in terms of implementation to make custom cases work or in some cases to even to use the Language Feature. Thus potentially requiring knowledge of other concepts outside of the language, such as Design Patterns.

In Java these are the Clonable (section 5.1), Iterator (section 5.2) and Observer (section 5.3). In order for the user to use them in user-defined classes, the appropriate package has to be imported, and the user-defined classes have to implement the appropriate interfaces or extend the appropriate classes.

In Python the belonging to this category Language Features are Abstract Base Class (section 4.4), Copy and Deepcopy (section 4.8) . Again in order for the user to use them in user-defined classes, the appropriate module has to be imported, and thus the user gains access to the appropriate class or function definition, in order to use or extend in the user code.

All of these Language Features belong to the classification as Partial, because, whilst they are part of the language standard libraries or language, they are not part of it by default. The user has to import a module and then be able to use them.

6.9.4 Summary

So in a sense it could be said that this classification to some degree corresponds to evaluating weather the Language Feature in question is part of the Language Specification or the Standard Libraries and the Language Specification and the implementation of the compiler/interpreter.

6.10 Bidirectional Relationship

The fact is that the Design Patterns do not exist in vacuum, neither do Programming Languages and that there are interactions between the two. In addition to that, both change over time (as described in Section 6.6), influenced by factors such as the users, the programmers using the Programming Languages, the projects these languages are used on and problems they solve. Likewise the Design Patterns are adapted to the Programming Languages and problem domains and new Design Patterns or variations of existing ones defined. Such as the Proxy Design Pattern and it's variations, such as Remote Proxy and Virtual Proxy as described in "Software Architecture Design Patterns in Java" [42].

Based on the findings of this chapter, it can be concluded that the interaction or relationship between Programming Languages and Design Patterns is in-fact a bidirectional one. Thus, instead of the relationship between the two, being simply one way, either Programming Languages affecting Design Pattern implementations or Design Patterns inspiring Language Feature implementations in Programming Languages it in-fact could be that both of these cases are true.

6.11 The Ripple Effect Theory

As already established in Section 6.10, the relationship between Programming Languages and Design Patterns is a bidirectional. This leads to the consequence that a change in Language Features implemented by a Programming Language could lead to changes in Design Pattern implementations in this language. This is because, the new or changed Language Feature might affect how the Design Patterns are implemented, by possibly either making it easier to implement certain Design Patterns or making the implementations of certain Design Patterns by the user's obsolete, in case of the Language Feature being a direct implementation. For example Python Abstract Base Classes (Section 4.4) are an implementation of the Abstract Parent Class, as discussed in Section 6.1. In addition to that they could be used to implement the Interface Design Pattern. Thus as shown in Section 6.6, if used to implement the Interface Design Pattern, then nearly all Design Pattern implementations could benefit from using Abstract Base Classes to define their interfaces in a more explicit manner. Thus they are likely to be used by the users of the Python language in Design Pattern implementations.

And thus the ripple effect, where change to one could potentially lead to changes to the other. If a new Language Feature that is related to Design Pattern implementations is implemented in a Programming Language, then it could lead to changes in the way other Design Patterns are implemented in the language.

A more concrete example, from Section 6.1 is if Python suddenly provided equivalent to Java interfaces. Then the interface Design Pattern would have a dedicated implementation in the form of a Language Feature. But that Language Feature would then be available to the implementations of Design Patterns in the language. And as found in Section 6.6 and in particular as detailed in Figure 6.6, interfaces in Java affect many Design Pattern implementations, so the assumption that the implementation of a similar feature in Python would have a similar effect is not far fetched. An example of changes in Design Patterns affecting is not difficult to come up with either. As discussed in Sections 6.6 and in this thesis in general some Design Patterns have been implemented as Language Features in Programming Languages, such as Iterators and Observer. Thus the idea that, if a new highly popular and deemed by the Programming Language users Design Pattern is described, then it might be implemented as a Language Feature, is not far fetched either.

Based on this, the "Ripple Effect Theory", describes an eventual consequence of the interaction between Design Patterns and Programming Languages.

6.12 Summary

This chapter examined comparatively the relevant to Design Patterns implementations Language Features in both Java and Python. Where relevant comparing the Language Features to the Design Patterns, to which they correspond. As well as evaluating the effects of Language Features on the Design Pattern's implementations. In addition to that, with regards to some Design Pattern's, which are highly coupled to Language Features in one Programming Language, substitute Language Features were suggested and the consequent Design Pattern implementations across the two Programming Languages, Java and Python compared. Example of that is Section 6.1

On a more theoretical level, this Chapter provided a theory with regards to the way Design Patterns possibly affect Programming Languages and their Language Features, called "The lifecycle of a Design Pattern" Section 6.8, based on the observations made in this thesis. It also defined the relationship between Programming Languages and Design Patterns as bidirectional (Section 6.10) and proposed a theory to explain the consequences of that relationship, called the "Ripple Effect Theory", in Section 6.11.

And it established a timeline of the relevant Language Features provided by the Programming Languages in Section 6.6, based on the Language Features examined in the thesis.

In addition to that, as a result of the discussions with regards to Language Features provided by the Programming Languages, it proposed a categorization scheme for the Programming Languages' Language Features, with respect to the Design Patterns they relate to.

Chapter 7

Conclusion

Implementing a Design Pattern in a particular language depends on what features that programming language provides to it's user. Thus programming languages' effect on Design Patterns is the programming languages' effect on the resultant implementation. In terms of the direct method of effect, the programming languages' effect is determined by what features the programming languages provide and what programming paradigms they support. That is assuming that the Design Pattern implementer is familiar with the feature in question and uses it to implement the Design Pattern. That is logically so, because the provided features and supported programming paradigms are what the user of the language directly uses.

Asking the question "How do programming languages affect Design Patterns", implies examining the effect of the Programming Languages on Design Pattern implementations in the respective language. Furthermore asking that question also implies examining the affect of the Language Features provided by languages on these Design Patterns implementations, in the context of Object Oriented Programming. Thus the topic "How do programming languages affect Design Patterns" can be expanded in more concrete terms to "How do Language Features provided by Programming Languages affect Design Patterns' implementations in these respective languages?".

The answer to that is that the Language Features in question could directly implement a Design Pattern or provide functionality that helps the implementation of the Design Pattern in the language.

If the Design Pattern is provided as a Language Feature, then the Programming Language affects the implementation of the Design Pattern in question directly, as demonstrated in Chapter 6. Concrete examples of which are the implementations of the Interface, Abstract Parent Class, Iterator and Observer Design Patterns in Java as well as the implementation of the Abstract Parent Class and Iterator in Python. Where as Chapter 6 discusses, the language's in question affect the implementations in terms of what methods they implement, in some cases more than specified by the Design Pattern, such as the

Java implementation of Iterators. In other cases less, such as in the case of Python Iterators. In addition to that both are examples of the Programming Language directly affecting the naming conventions of methods, classes and interfaces.

If the Language Feature on the other hand is not a direct implementation of Design Pattern, it could affect potential implementations of Design Patterns in the language significantly, as again, demonstrated further detail in Chapter 6. Some examples of that are the Python copy and Java Cloneable Language Features, that make implementing the Prototype Design Pattern much easier. And example of negative affect is the fact that since Python does not provide direct implementation of the Interface Design Pattern, thus the user's of the language are not likely to use both Interface and Abstract Parent Class, as described in Section 6.1.4. Another one is the fact that in Python all members of the objects and classes are always public, thus making the implementation of the Accessor Methods and Private Methods Design Patterns difficult to implement, as described in Section 6.1.

Thus the Language Features that Programming Languages provide could directly implement a Design Pattern, affect a Design Pattern implementation by aiding it or by hindering it.

As a conclusion it could be said that not only do not only do Programming Languages affect Design Pattern implementations, trough the set of Language Features they provide, but the Design Patterns in term affect Language Features. Meaning that the relationship is bidirectional, as described in Section 6.10. In addition to that a sort of ripple effect, as described in Section 6.11 possibly could occur when changes to Programming Languages occur onto Design Pattern implementations. And vice versa.

Bibliography

[1] *12.1. pickle — Python object serialization — Python 3.6.0 documentation*. URL: https://docs.python.org/3/library/pickle.html (visited on 23/12/2016).

[2] *2. Built-in Functions — Python 3.6.0 documentation*. URL: https://docs.python.org/3.6/library/functions.html#type (visited on 07/12/2016).

[3] *2. Built-in Functions — Python 3.6.0 documentation*. URL: https://docs.python.org/3.6/library/functions.html#getattr (visited on 12/12/2016).

[4] *2. Built-in Functions — Python 3.6.0 documentation*. URL: https://docs.python.org/3.6/library/functions.html#setattr (visited on 12/12/2016).

[5] *2. Built-in Functions — Python 3.6.0 documentation*. URL: https://docs.python.org/3.6/library/functions.html#dir (visited on 12/12/2016).

[6] *2. Built-in Functions — Python 3.6.0 documentation*. URL: https://docs.python.org/3.6/library/functions.html#property (visited on 12/12/2016).

[7] *3. Data model — Python 3.5.2 documentation*. URL: https://docs.python.org/3/reference/datamodel.html (visited on 05/12/2016).

[8] *4. Built-in Types — Python 3.6.0 documentation*. URL: https://docs.python.org/3.6/library/stdtypes.html#bltin-type-objects (visited on 07/12/2016).

[9] *4. Built-in Types — Python 3.6.0b2 documentation*. URL: https://docs.python.org/dev/library/stdtypes.html#iterator-types (visited on 13/10/2016).

[10] *8.10. copy — Shallow and deep copy operations — Python 3.5.2 documentation*. URL: https://docs.python.org/3.5/library/copy.html (visited on 10/11/2016).

[11] *A Guide to Object Cloning in Java - HowToDoInJava*. URL: http://howtodoinjava.com/core-java/cloning/a-guide-to-object-cloning-in-java/ (visited on 03/11/2016).

[12] *A Guide to Python's Magic Methods « rafekettler.com*. URL: http://www.rafekettler.com/magicmethods.html (visited on 21/10/2016).

[13] *Are Design Patterns Missing Language Features*. URL: http://wiki.c2.com/?AreDesignPatternsMissingLanguageFeatures (visited on 16/12/2016).

[14] Ethan Brown. *Learning JavaScript: JavaScript Essentials for Modern Application Development*. 3 edition. O'Reilly Media, 5th Mar. 2016. 358 pp. ISBN: 978-1-4919-1491-5.

[15] *C# Programming Guide*. URL: https://msdn.microsoft.com/en-us/library/67ef8sbd.aspx (visited on 14/12/2016).

[16] Stephen Cass. *The 2015 Top Ten Programming Languages*. IEEE Spectrum: Technology, Engineering, and Science News. 20th July 2015. URL: http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages (visited on 14/06/2016).

[17] *Cloneable (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Cloneable.html (visited on 03/11/2016).

[18] *Controlling Access to Members of a Class (The Java™ Tutorials > Learning the Java Language > Classes and Objects)*. URL: https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html (visited on 11/12/2016).

[19] *cpython: 7fb90c4ae643 Lib/_collections_abc.py*. URL: https://hg.python.org/cpython/file/3.5/Lib/_collections_abc.py (visited on 14/10/2016).

[20] *cpython: c4f39b6f3176 Lib/abc.py*. URL: https://hg.python.org/cpython/file/3.6/Lib/abc.py (visited on 07/12/2016).

[21] *cpython: fdccc3222d0a*. URL: https://hg.python.org/cpython/rev/fdccc3222d0a (visited on 10/11/2016).

[22] *cpython: log*. URL: https://hg.python.org/cpython/ (visited on 05/12/2016).

[23] *Dagger A fast dependency injector for Android and Java*. URL: https://google.github.io/dagger/ (visited on 15/09/2016).

[24] *Decorators — Python 3 Patterns, Recipes and Idioms*. URL: https://python-3-patterns-idioms-test.readthedocs.io/en/latest/PythonDecorators.html (visited on 05/09/2016).

[25] *Design and History FAQ — Python 2.7.12 documentation*. URL: https://docs.python.org/2/faq/design.html#how-fast-are-exceptions (visited on 24/10/2016).

[26] *Design Patterns and Refactoring*. URL: https://sourcemaking.com (visited on 23/12/2016).

[27] *Design Patterns in Dynamic Languages*. URL: http://www.norvig.com/design-patterns/ (visited on 17/09/2016).

[28] *Docs | Node.js*. URL: https://nodejs.org/en/docs/ (visited on 15/12/2016).

[29] *Enumeration (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Enumeration.html (visited on 08/10/2016).

[30] *Functional Programming HOWTO — Python 3.6.0 documentation*. URL: https://docs.python.org/3.6/howto/functional.html (visited on 15/12/2016).

[31] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. 1 edition. Reading, Mass: Addison-Wesley Professional, 10th Nov. 1994. 395 pp. ISBN: 978-0-201-63361-0.

[32] *Greg Ward » Post: Performance Penalty of Python Exceptions*. Greg Ward. 18th Sept. 2015. URL: http://gerg.ca/blog/post/2015/try-except-speed/ (visited on 17/12/2016).

[33] *History of Python*. In: *Wikipedia*. Page Version ID: 753109662. 5th Dec. 2016. URL: https://en.wikipedia.org/w/index.php?title=History_of_Python&oldid=753109662 (visited on 13/12/2016).

[34] *Iterable (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html (visited on 08/10/2016).

[35] *Iterating over collections in Java 8 | JavaWorld*. URL: http://www.javaworld.com/article/2461744/java-language/java-language-iterating-over-collections-in-java-8.html (visited on 16/12/2016).

[36] *Iterator (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html (visited on 08/10/2016).

[37] *Java SE - Downloads | Oracle Technology Network | Oracle*. URL: http://www.oracle.com/technetwork/java/javase/downloads/index-jsp-138363.html (visited on 16/12/2016).

[38] *Java Versions, Features and History*. Java Tutorial Blog. 1st Jan. 2012. URL: http://javapapers.com/core-java/java-features-and-history/ (visited on 13/12/2016).

[39] E. B. Jul. 'Object Mobility in a Distributed Object-oriented System'. UMI Order No: GAX90-00257. PhD thesis. Seattle, WA, USA: University of Washington, 1989.

[40] Sakis Kasampalis. *Mastering Python Design Patterns*. Packt Publishing, 28th Jan. 2015. 212 pp. ISBN: 978-1-78398-932-4.

[41] Jeff Knupp. *Write Cleaner Python: Use Exceptions*. URL: https://jeffknupp.com/blog/2013/02/06/write-cleaner-python-use-exceptions/ (visited on 17/12/2016).

[42] Partha Kuchana. *Software Architecture Design Patterns in Java*. 1 edition. Boca Raton, FL: Auerbach Publications, 22nd Apr. 2004. 416 pp. ISBN: 978-0-8493-2142-9.

[43] *Lesson: Annotations (The Java™ Tutorials > Learning the Java Language)*. URL: https://docs.oracle.com/javase/tutorial/java/annotations/ (visited on 16/12/2016).

[44] *Metaprogramming*. In: *Wikipedia*. Page Version ID: 754803512. 14th Dec. 2016. URL: https://en.wikipedia.org/w/index.php?title=Metaprogramming&oldid=754803512 (visited on 16/12/2016).

[45] *Metaprogramming — Python 3 Patterns, Recipes and Idioms*. URL: https://python-3-patterns-idioms-test.readthedocs.io/en/latest/Metaprogramming.html (visited on 16/12/2016).

[46] *Object (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#clone-- (visited on 03/11/2016).

[47] *Observable (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Observable.html (visited on 30/11/2016).

[48] *Observer (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/java/util/Observer.html (visited on 30/11/2016).

[49] *Overview (Java Platform SE 8)*. URL: https://docs.oracle.com/javase/8/docs/api/ (visited on 16/12/2016).

[50] *Overview — Python 3.6.0 documentation*. URL: https://docs.python.org/3.6/ (visited on 16/12/2016).

[51] Jean Paul. *Design Patterns in C#*. 2012. 142 pp.

[52] *PEP 0 – Index of Python Enhancement Proposals (PEPs)*. Python.org. URL: https://www.python.org/dev/peps/ (visited on 25/09/2016).

[53] *PEP 1 – PEP Purpose and Guidelines*. Python.org. URL: https://www.python.org/dev/peps/pep-0001/ (visited on 25/09/2016).

[54] *PEP 234 – Iterators*. Python.org. URL: https://www.python.org/dev/peps/pep-0234/ (visited on 13/12/2016).

[55] *PEP 234 – Iterators*. Python.org. URL: https://www.python.org/dev/peps/pep-0234/ (visited on 13/10/2016).

[56] *PEP 3115 – Metaclasses in Python 3000 | Python.org*. URL: https://www.python.org/dev/peps/pep-3115/ (visited on 06/12/2016).

[57] *PEP 3119 – Introducing Abstract Base Classes*. Python.org. URL: https://www.python.org/dev/peps/pep-3119/ (visited on 08/12/2016).

[58] *PEP 3129 – Class Decorators*. Python.org. URL: https://www.python.org/dev/peps/pep-3129/ (visited on 13/12/2016).

[59] *PEP 318 – Decorators for Functions and Methods*. Python.org. URL: https://www.python.org/dev/peps/pep-0318/ (visited on 13/12/2016).

[60] *PEP 318 – Decorators for Functions and Methods*. Python.org. URL: https://www.python.org/dev/peps/pep-0318/ (visited on 29/09/2016).

[61] *Primer on Python Decorators - Real Python*. URL: https://realpython.com/blog/python/primer-on-python-decorators/ (visited on 05/09/2016).

[62] *Python Is Not Java (dirtSimple.org)*. URL: http://dirtsimple.org/2004/12/python-is-not-java.html (visited on 12/12/2016).

[63] *PythonDecorators - Python Wiki*. URL: https://wiki.python.org/moin/PythonDecorators#What_is_a_Decorator (visited on 05/09/2016).

[64] *Revenge of the Nerds*. URL: http://www.paulgraham.com/icad.html (visited on 21/09/2016).

[65] Guido van Rossum. *A Brief Timeline of Python*. URL: http://python-history.blogspot.com/2009/01/brief-timeline-of-python.html (visited on 13/12/2016).

[66] sogrady. *The RedMonk Programming Language Rankings: June 2015*. tecosystems. 1st July 2015. URL: http://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/ (visited on 14/06/2016).

[67] *spring.io*. URL: https://spring.io/ (visited on 15/09/2016).

[68] Stoyan Stefanov. *JavaScript Patterns*. 1 edition. Sebastopol, CA: O'Reilly Media, 1st Oct. 2010. 236 pp. ISBN: 978-0-596-80675-0.

[69] Bjarne Stroustrup. *The C++ Programming Language, 4th Edition*. 4 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 19th May 2013. 1368 pp. ISBN: 978-0-321-56384-2.

[70] *The Big List of 256 Programming Languages - DZone Java*. dzone.com. URL: https://dzone.com/articles/big-list-256-programming (visited on 01/09/2016).

[71] *The For-Each Loop*. URL: https://docs.oracle.com/javase/1.5.0/docs/guide/language/foreach.html (visited on 10/10/2016).

[72] *The History of Python: Origin of metaclasses in Python*. URL: https://python-history.blogspot.no/2013/10/origin-of-metaclasses-in-python.html (visited on 05/12/2016).

[73] *The Java Language Environment*. URL: http://www.oracle.com/technetwork/java/intro-141325.html (visited on 14/09/2016).

[74] Simon Timms. *Mastering JavaScript Design Patterns*. Birmingham: Packt Publishing - ebooks Account, 21st Nov. 2014. 290 pp. ISBN: 978-1-78398-798-6.

[75] *TIOBE Index | Tiobe - The Software Quality Company*. URL: http://www.tiobe.com/tiobe_index (visited on 14/06/2016).

[76] *Trail: The Reflection API (The Java™ Tutorials)*. URL: https://docs.oracle.com/javase/tutorial/reflect/ (visited on 16/12/2016).

[77] *UML association is relationship between classifiers to show that instances of classifiers could be either linked to each other or combined into some aggregation*. URL: http://www.uml-diagrams.org/association.html (visited on 23/12/2016).

[78] *undecorate 0.2.1 : Python Package Index*. URL: https://pypi.python.org/pypi/undecorate/0.2.1 (visited on 05/09/2016).

[79] *Understanding Python metaclasses - ... and Python objects in general*. URL: https://blog.ionelmc.ro/2015/02/09/understanding-python-metaclasses/ (visited on 05/12/2016).

Appendices

Appendix A

Language Features

In this appendix you will find code samples demonstrating Language Features and Design Patterns in the two Programming Languages.

A.0.1 Python Metaclass

Listing A.1: Singleton implementation using Metaclassses in Python

```python
#!/usr/bin/python

# Singleton using metaclass:
# Seems to work just fine.
# simply remember to use __call__ instead of __init__ in meta

class Singleton(type):
    _instances = {}

    def __call__(cls, *args, **kwargs):
        print("call method")
        if cls not in Singleton._instances:
            Singleton._instances[cls] = super(Singleton, cls).__call__(*args, **kwargs)
        return Singleton._instances[cls]

class MyClass(object, metaclass=Singleton):

    def __init__(self, name):
        print("init method")
        self.name = name

class MyClass2(object, metaclass=Singleton):
    def __init__(self, name):
        print("init method")
        self.name = name
```

```

##Simple demo of the singleton:
m = MyClass('aa')
k = MyClass('bb')
a = MyClass2('a2')
b = MyClass2('b2')
print(k == m)
print(m is k)
print(type(k))
print(type(m))
print(k.name)
print(m.name)
print(a.name)
print(b.name)
print(type(a))
print(type(b))
print(a is m)

# Output:
# init method
# call method
# call method
# call method
# call method
# True
# True
# <class '__main__.MyClass'>
# <class '__main__.MyClass'>
# aa
# aa
# a2
# a2
# <class '__main__.MyClass2'>
# <class '__main__.MyClass2'>
# False
```

A.0.2 Python Decorators

Listing A.2: Function decorating another

```
#!/usr/bin/python
# An example use of the Pyhton decorator feature:

def decorator(function):
```

```python
    def wrapper(*args, **kwargs):
        print("inside decorator: pre-decorated")
        retvalue = function(*args, **kwargs)
        print("inside decorator: post-decorated, return value = ", retvalue)
        return retvalue
    return wrapper

@decorator
def original() :
    print("inside original function")

#if __name__ == "__main__":
original()

# Output sample:
# inside decorator: pre-decorated
# inside original function
# inside decorator: post-decorated, return value =  None
```

Listing A.3: Python decorator equivalent

```python
#!/usr/bin/python
# An example of what decorators in Python are equivalent to:

def decorator(object):
    def wrapper():
        print("inside decorator")
        return object()
    return wrapper


def original() :
    print("inside original function")

if __name__ == "__main__":
    original = decorator(original)
    original()

# Output sample:
# inside decorator
# inside original function
```

Listing A.4: Function decorating a class

```python
#!/usr/bin/python
# An example of function being used to decorate a class:
```

```python

def decorator(class_):
    def wrapper(*args, **kwargs):
        print("inside decorator, pre-decorated")
        instance = class_(*args, **kwargs)
        print("inside decorator, post-decorated")
        return instance
    return wrapper

@decorator
class OriginalClass():

    def __init__(self):
        print("inside __init__()")

    def test(self):
        print("a test funcion")

a = OriginalClass()
print(type(a))
a.test()

# Output:
# inside decorator, pre-decorated
# inside __init__()
# inside decorator, post-decorated
# <class '__main__.OriginalClass'>
# a test funcion
```

Listing A.5: Class decorating a function

```python
#!/usr/bin/python

class my_decorator(object):

    def __init__ (self, f):
        print("inside decorator.__init__()")
        self.f = f

    def __call__(self, *args, **kwargs):
        print("inside decorator.__call__(), pre-decorated")
        returnValue = self.f(*args, **kwargs)
        print("post-function")
        return returnValue

```

```python
@my_decorator
def function():
    print("inside function")

function()

# Output:
# inside decorator.__init__()
# inside decorator.__call__(), pre-decorated
# inside function
# post-function
```

Listing A.6: Class decorating a class

```python
#!/usr/bin/python

class decorator(object):

    def __init__(self, other, *args, **kwargs):
        print("inside decorator.__init__()")
        self.other = other


    def __call__(self, *args, **kwargs):
        classattr=self.other
        print("inside decorator.__call__(), pre-decorated")
        instance = self.other(*args, **kwargs)
        print("inside decorator.__call__(), post-decorated")
        return instance

@decorator
class Test():

    def __init__(self):
        print("inside Test.__init__()")

    def test(self):
        print("test func");

t = Test()
print(type(t))
t.test()

# Output:
# inside decorator.__init__()
# inside decorator.__call__(), pre-decorated
# inside Test.__init__()
# inside decorator.__call__(), post-decorated
# <class '__main__.Test'>
# test func
```

Listing A.7: Python decorator with arguments

```
#!/usr/bin/python
# A quick demo of decorators with arguments.

class dec_with_args(object):

    def __init__(self, arg1, arg2, arg3):
        print("Inside __init__()")
        self.arg1 = arg1
        self.arg2 = arg2
        self.arg3 = arg3

    def __call__(self, f):
        print("Inside __call__()")
        def wrapped_f(*args, **kwargs):
            print("Inside wrapped_f(), pre-decorated")
            print("dec args: ", self.arg1, self.arg2, self.arg3)
            f(*args, **kwargs)
            print("Post decorated")
        return wrapped_f

@dec_with_args("hello", "world", 42)
def sayHello(a1, a2, a3, a4):
    print("sayHello: ", a1, a2, a3, a4)

sayHello("say", "hello", "argument", "list")

sayHello("a", "b", "c", "d")

# Output:
# Inside __init__()
# Inside __call__()
# Inside wrapped_f(), pre-decorated
# dec args:  hello world 42
# sayHello:  say hello argument list
# Post decorated
# Inside wrapped_f(), pre-decorated
# dec args:  hello world 42
```

```python
# sayHello:  a b c d
# Post decorated

# Comments:
# The arguments that one passes to the decorator @ are passed to the __init__ method
```

Listing A.8: Implementation of the decorator design pattern on functions in Python

```python
#!/usr/bin/python
# A more realistic implementation of the Decorator design pattern using Python:

def decorator(decoratee):
    def wrapper(*args, **kwargs):
        print("pre-decoratee")
        returnVal = decoratee(*args, **kwargs)
        print("post-decoratee")
        return returnVal
    return wrapper

def original(arg):
    print("inside original func, arg=", arg)

decorated = decorator(original)
decorated(5)

# Output
# pre-decoratee
# inside original func, arg= 5
# post-decoratee
```

Listing A.9: Implementation of the iterator design pattern using the Python iterators

```python
#!/usr/bin/python
# A demonstration of a custom iterable/container/collection implementing the iterator design pattern as defined by Python.

class MyCollection(object):

    def __init__(self, start, stop):
        self.start = start
        self.stop = stop
        self.current = start

    def __iter__(self):
        '''As per the Iterable type definition, returns an iterator object that knows how to iterat
        return MyIterator(self)

    def nextElement(self):
        '''A custom logic next item function.'''
        if self.current < self.stop:
            self.current = self.current  + 1
            return True
        else :
            return False

    def getCurrent(self):
        return self.current

class MyIterator(object):

    def __init__(self, colletion):
        self.collection = collection

    def __iter__(self):
        '''As per the Iterator type definition iterators should return themselves'''
        return self

    def __next__(self):
        '''As per the Iterator type definition, a method to get the next item of the collection'''
        if(collection.nextElement()):
            return collection.getCurrent()
        else :
            raise StopIteration


collection = MyCollection(1,6)

print("Iterating over items:")
for item in collection:
    print(item, " ")

# Output:
# Iterating over items:
# 2
# 3
# 4
# 5
# 6
```

A.0.3 Python Copy

Listing A.10: Example custom implementation of the copy methods in Python

```python
#!/usr/bin/python
# An example use-case of how a custom __copy__() can be implemented:
import copy

class Department(object):

    def __init__(self, name):
        self.name = name

class Machine(object):

    def __init__(self, name):
        self.name = name

class Student(object):

    def __init__(self, name, department, machine):
        self.name = name
        self.department = department
        self.machine = machine

    def __copy__(self):
        # remove method to make a copy, while inside method:
        copyMethod = getattr(type(self), "__copy__", None)
        setattr(type(self), "__copy__", None)

        result = copy.copy(self)

        # reassign the method
        setattr(type(self), "__copy__", copyMethod)

        # make special copy operations
        result.machine = copy.copy(self.machine)

        # return the copy to client
        return result

department = Department("department1")
machine = Machine("machine1")
student = Student("student1", department, machine)
```

```

student2 = copy.copy(student)
student2.name ="student2"
student2.machine.name = "machine2"

print("Student1 : ", student.name, " ", student.department.name, " ", student.machine.name )
print("Student2 : ", student2.name, " ", student2.department.name, " ", student2.machine.name )

# Output:
# Student1 :  student1   department1   machine1
# Student2 :  student2   department1   machine2
```

A.0.4 Java Iterator

Listing A.11: Example use of Java Iterators

```java

import java.util.Iterator;
import java.util.NoSuchElementException;

/**
 * An example collection implementing Iterable and returning a custom iterator that knows how to it
 */
public class CustomCollection implements Iterable<Integer> {
    private int start;
    private int end;

    CustomCollection(int start, int end) {
        this.start = start;
        this.end = end;
    }

    @Override
    public Iterator iterator() {
        return new CustomIterator(this);
    }

    public int getStart() {
        return start;
    }

    /**
     * THe iterator implmentation:
     */
    private class CustomIterator implements Iterator<Integer> {
        private int current;

        CustomIterator(CustomCollection collection) {
            current = collection.getStart();
        }

        @Override
        public boolean hasNext() {
            return this.current < end;
        }

        @Override
        public Integer next() {
            if (this.hasNext()) {
                int curr = current;
                current++;
                return curr;
            }
            throw new NoSuchElementException();
        }

        @Override
        public void remove() {
            throw new UnsupportedOperationException();
        }
    }

    public static void main(String[] args) {
        CustomCollection collection = new CustomCollection(1, 10);

        System.out.println("Using a while statement to iterate:");
        // Without using the syntactic sugar:
        Iterator<Integer> it = collection.iterator();
        while (it.hasNext()) {
            int cur = it.next();
            System.out.println("cur=" + cur);
        }

        System.out.println("Using the enhanced forloop:");
        // Using the enhanced forloop:
        for (Integer cur : collection) {
            System.out.println("cur=" + cur);
        }
    }
```

```
}


// Output sample:
// Using a while statement to iterate:
// cur=1
// cur=2
// cur=3
// cur=4
// cur=5
// cur=6
// cur=7
// cur=8
// cur=9
// Using the enhanced forloop:
// cur=1
// cur=2
// cur=3
// cur=4
// cur=5
// cur=6
// cur=7
// cur=8
// cur=9
```

A.0.5 Java Cloneable

Listing A.12: Java Cloneable example Student class

```java
package sample.design.patterns;

public class Student implements Cloneable {

    private String name;
    private Department department;
    private Machine machine;

    public Student(String name, Department dept, Machine machine) {
        this.name = name;
        this.department = dept;
        this.machine = machine;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        Student output = (Student) super.clone();
        output.setMachine((Machine) machine.clone());
        return output;
    }

    public String getName() {
        return name;
    }

    public Machine getMachine() {
        return machine;
    }

    public void setName(String name) {
        this.name = name;
    }

    public void setMachine(Machine machine) {
        this.machine = machine;
    }
}
```

Listing A.13: Java Cloneable example Machine class

```java
package sample.design.patterns;

public class Machine implements Cloneable {

    private String name;

    public Machine(String name) {
        this.name = name;
    }

    @Override
    protected Object clone() throws CloneNotSupportedException {
        return super.clone();
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Listing A.14: Java Cloneable example Department class

```
package sample.design.patterns;

public class Department implements  Cloneable {

    private String name;

    public Department(String name) {
        this.name = name;
    }

    @Override
    protected Department clone() throws CloneNotSupportedException {
        return (Department) super.clone();
    }
}
```

Listing A.15: Java Cloneable example Main class

```
package sample.design.patterns;

public class Main {
    public static void main(String[] args) throws CloneNotSupportedException {
        Department department = new Department("Human Resources");
        Machine machine = new Machine("Student1's pc");
        Student original = new Student("Student1", department, machine);
        Student cloned = null; //Lets create a clone of original object
        try {
            cloned = (Student) original.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println("Clone isn't supported");
            e.printStackTrace();
        }
        System.out.println("Testing cloning:");
        //Whilst there is only one department, each student should have his own pc:
        cloned.getMachine().setName("A different machine");
        System.out.println(original.getMachine().getName());
        System.out.println(cloned.getMachine().getName());
        /*Output:
        esting cloning:
        Student1's pc
        A different machine
        Id's:
        Original id: 1
        Cloned id: 2*/
    }
}
```

A.0.6 Java Observable

Listing A.16: Java Observable example Observer class

```java
package sample.design.patterns;

import java.util.Observable;

public class ObservableDemo extends Observable {

    @Override
    public void notifyObservers() {
        //For the purpouse of demonstrating observables easier, flag as chenged, before notifying:
        super.setChanged();
        super.notifyObservers();
    }
}
```

Listing A.17: Java Observablee example Observable class

```java
package sample.design.patterns;

import java.util.Observable;
import java.util.Observer;

public class ObserverDemo implements Observer {

    @Override
    public void update(Observable observable, Object o) {
        System.out.println("Observer updated");
    }

    public static void main(String[] args) {
        Observer observer = new ObserverDemo();
        Observable observable = new ObservableDemo();

        System.out.println("adding Observer to observable:");
        //register the obesrver for updates...
        // The observer could register itself as well.
        observable.addObserver(observer);
        System.out.println("Nr of observers : " + observable.countObservers());

        System.out.println("Notifying all observables");
        //assume another thread or object calls this:
        observable.notifyObservers();
        /* Output:
        adding Observer to observable:
        Nr of observers : 1
        Notifying all observables
        Observer updated */
    }
}
```

A.0.7 Interfaces

Listing A.18: Java Interfaces example

```java
package sample.design.patterns;

public interface Provider {
    void method1();

    void method2();

    void method3();
}

class ProviderImplementation1 implements Provider {

    @Override
    public void method1() {
        System.out.println("ProviderImplementation1.method1()");
    }

    @Override
    public void method2() {
        System.out.println("ProviderImplementation1.method2()");
    }

    @Override
    public void method3() {
        System.out.println("ProviderImplementation1.method3()");
    }
}

class Providerimplementation2 implements Provider {

    @Override
    public void method1() {
        System.out.println("ProviderImplementation2.method1()");
    }

    @Override
    public void method2() {
        System.out.println("ProviderImplementation2.method2()");
    }

    @Override
    public void method3() {
        System.out.println("ProviderImplementation2.method3()");
    }
}

class Client {
    Provider provider;

    public void useProvider(Provider provider) {
        this.provider = provider;
        provider.method1();
        provider.method2();
        provider.method3();
    }
}
```

Listing A.19: Python implicit Interfaces example

```python
#!/usr/bin/python
# A demonstration of the Interface Design Pattern implementation that uses duck typing.

# The implicit type definition is here, in the documentation:
# Provider:
#   + method1()
#   + method2()
#   + method3()

class ProviderImplementation1:
    def method1(self):
        print("ProviderImplementation1.method1()")

    def method2(self):
        print("ProviderImplementation1.method2()")

    def method3(self):
        print("ProviderImplementation1.method3()")


class ProviderImplementation2:
    def method1(self):
        print("ProviderImplementation2.method1()")

    def method2(self):
        print("ProviderImplementation2.method2()")

    def method3(self):
        print("ProviderImplementation2.method3()")

class Client:

    def useProvider(self, provider):
        self.provider = provider
        provider.method1()
        provider.method2()
        provider.method3()

# # Just as a proof, the output of :
# p1 = ProviderImplementation1()
# p2 = ProviderImplementation2()

# c = Client()

# c.useProvider(p1)
# c.useProvider(p2)

# results in :
# ProviderImplementation1.method1()
# ProviderImplementation1.method2()
# ProviderImplementation1.method3()
# ProviderImplementation2.method1()
# ProviderImplementation2.method2()
# ProviderImplementation2.method3()
```

Listing A.20: Python explicit Interfaces example

```python
#!/usr/bin/python
# A demonstration of the Interface Design Pattern implementation that uses inheritance.

# The type definition is here, as a class definition, with all methods raising a NotImplementederror :
class  Provider:

    def method1(self):
        raise NotImplementedError

    def method2(self):
        raise NotImplementedError

    def method3(self):
        raise NotImplementedError

# The first implementer:
class ProviderImplementation1(Provider):

    def method1(self):
        print("ProviderImplementation1.method1()")

    def method2(self):
        print("ProviderImplementation1.method2()")

    def method3(self):
        print("ProviderImplementation1.method3()")

# The second implementer:
class ProviderImplementation2(Provider):

    def method1(self):
        print("ProviderImplementation2.method1()")

    def method2(self):
        print("ProviderImplementation2.method2()")

    def method3(self):
        print("ProviderImplementation2.method3()")

class Client:

    def useProvider(self, provider):
        self.provider = provider
        provider.method1()
        provider.method2()
        provider.method3()


# # Just as a proof, the output of :
# p1 = ProviderImplementation1()
# p2 = ProviderImplementation2()

# c = Client()

# c.useProvider(p1)
# c.useProvider(p2)

# # results in :

# ProviderImplementation1.method1()
# ProviderImplementation1.method2()
# ProviderImplementation1.method3()
# ProviderImplementation2.method1()
# ProviderImplementation2.method2()
# ProviderImplementation2.method3()
```

Listing A.21: Implementation of Singleton design pattern in Python using the decorator feature.

```
#!/usr/bin/python

# Singleton implementation using decorators.
# Despite the voiced concerns it seems to be working.
# According to python (when calling the type() funciton) the type of the singleton class is a clas

# The pro's listed are however valid.
# Using decorators to decorate a class into being a singleton seems quite natural.

# It also is a behaviour that cannot be overridden by subclassing.

# Decorator functions and classes could be undecorated using the undecorate package.
# _original(), --> __wrapped__ ? --> undecorated() package. !!
# https://stackoverflow.com/questions/1166118/how-to-strip-decorators-from-a-function-in-python
# However one could utilize the ._original() to call the undecorated funciton.



def singleton(class_):
    instances = {}

    def getinstance(*args, **kwargs):
        if class_ not in instances:
            instances[class_] = class_(*args, **kwargs)
        return instances[class_]
    return getinstance

@singleton
class MyClass:
    def __init__(self, *args, **kwargs):
        self.name = "Decorated singleton object's variable"

    def putMessage(self):
        print("Decorated singleton object's method");


class ObjTest:
    def __init__(self, *args, **kwargs):
        self.name = "Object's variable"

    def putMessage(self):
        print("Object's method");

# Tests:
#if __name__ == "__main__":
m = MyClass()
k = MyClass()
print(type(m))
print(type(k))
print(m == k)
print(m is k)
m.putMessage()

o = ObjTest()
print(type(o))
o.putMessage()

# Output:
# <class '__main__.MyClass'>
# <class '__main__.MyClass'>
# True
# True
# Decorated singleton object's method
# <class '__main__.ObjTest'>
# Object's method
```