NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

GUILHERME VALENTE PEREIRA

BSc in Computer Science

ASSESSMENT OF PYTHON ADVANCED COMPOSITION MECHANISMS BASED ON DESIGN PATTERNS

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
April, 2025

DEPARTMENT OF
COMPUTER SCIENCE

ASSESSMENT OF PYTHON ADVANCED COMPOSITION MECHANISMS BASED ON DESIGN PATTERNS

GUILHERME VALENTE PEREIRA

BSc in Computer Science

Adviser: Miguel Jorge Tavares Pessoa Monteiro
Assistant Professor, NOVA University Lisbon

Dissertation Plan
MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
April, 2025

Abstract

Python continues to increase in popularity consistently evidenced by its position at the top of the TIOBE Index, highlighting its significant role in the programming world.

Despite its widespread use, existing books and resources focusing on implementing design patterns in Python, tend to overlook the language's advanced mechanisms and there remains a notable gap of scientific studies regarding Python's support for modularity and advanced module composition.

This thesis aims to explore these under-researched mechanisms and how they can enhance the implementation of design patterns. By examining these mechanisms, it seeks to demonstrate how they can improve the modularity, reusability, module composition and provide insights that could influence best practices and contribute to the further development of Python in software engineering.

As the thesis progressed, the patterns were grouped into three categories: those with direct support from the language, those enhanced by advanced mechanisms, and those where no improvement was observed.

Keywords: Python, Object-Oriented Programming, Module composition, Advanced Composition Mechanisms, Design Patterns

Resumo

Python continua a aumentar em popularidade, evidenciado consistentemente pela sua posição no topo do Índice TIOBE, destacando o seu papel significativo no mundo da programação.

Apesar de ser utilizado em múltiplos projetos e áreas differentes, os livros e recursos existentes que se concentram na implementação de padrões de design em Python tendem a ignorar os mecanismos avançados da linguagem e ainda existe uma lacuna notável de estudos académicos sobre as capacidades do Python para modularidade e composição avançada.

Esta tese tem como objetivo explorar estes mecanismos pouco explorados e como podem melhorar a implementação de padrões de design. Ao examinar esses mecanismos, procura demonstrar como podem melhorar a modularidade, reutilização, composição de módulos e fornecer conhecimentos que podem influenciar as melhores práticas e contribuir para o desenvolvimento futuro do Python na engenharia de software.

À medida que a tese foi avançando, os padrões foram agrupados em três grupos: os que têm suporte direto da linguagem, os que foram melhorados por mecanismos avançados e os que não apresentaram melhorias.

Palavras-chave: Python, Object-Oriented Programming, Module composition, Advanced Composition Mechanisms, Design Patterns

Contents

List of Figures

List of Tables

Listings

Glossary

Design Pattern A general, reusable solution to a commonly occurring problem in software design *(p. 28)*

Dispatch A mechanism that determines which function or method to execute at runtime based on the type or number of input arguments *(p. 1)*

Metaclass A class of a class that defines how other classes are created *(p. 1)*

Mixin A class providing methods that can be inherited without forming a strict "is-a" relationship *(p. 1)*

Python A high-level programming language for general-purpose programming *(p. 1)*

Python Decorator A function that modifies or extends the behavior of another function or method *(p. 1)*

Reflection The process of inspecting and modifying a program's structure at runtime *(p. 1)*

Syntax Sugar A syntax feature in a programming language that simplifies code readability or usage without adding new functionality, often making the code more concise or expressive *(p. 17)*

Acronyms

AOP Aspect-Oriented Programming *(pp. 23, 72)*
AST Abstract syntax tree *(p. 4)*

GOF Gang Of Four *(pp. 28, 64)*

JIT Just-In-Time Compiler *(p. 4)*

OOP Object-Oriented Programming *(p. 2)*

PEP Python Enhancement Proposal *(p. 17)*
PVM Python Virtual Machine *(p. 4)*

1

Introduction

This chapter introduces the context and motivation for the work presented in this thesis. It also presents the objectives and expected contributions of the work, as well as the structure of the document.

1.1 Context and Description

There have been previous studies and books that evaluated popular programming languages like Java [12] [19], Ruby [33] or Go [40] in terms of their ability to implement design patterns and Python is no exception being the subject of many books [3] [11] [18] [36] that detail its implementations of various patterns . However, while these books provide valuable insights into how Python implements design patterns, they often don't fully explore the advanced mechanisms that Python supports namely Mixin (see section 2.8), Dispatch (see section 2.10), Metaclass (see section 2.7), Reflection (see section 2.13), Python Decorator (see section 2.9), Dynamic Behaviour (see section 2.5) and Modules (see section 2.12).

This thesis explores how Python handles and supports these mechanisms and examines how they can be effectively utilized to improve the modularity, reusability, module composition and how they can enhance the implementation of Design Patterns (see section 4) in Python.

This thesis also analyzes how Python handles object-oriented programming [6] (see section 2.4) , examining how this approach can improve the structure and maintenance of Python programs.

1.2 Motivation

As software projects evolve, the complexity and demand for high-quality code also increase. Given Python's growing popularity, there should be a corresponding increase in studies on Python's module composition mechanisms and how to use them effectively. However, to our knowledge, very few studies have focused on how Python handles complex module composition mechanisms.

Building on the importance of Object-Oriented Programming (OOP), modularity, readability, and reusability, and considering Python's status as one of the most popular programming languages [43], the motivation behind this thesis is to understand how Python supports these advanced mechanisms and how design patterns can be effectively implemented in Python to leverage these principles for better software development outcomes.

1.3 Objectives and Expected Contribution

The objective of this thesis is to understand how Python supports advanced mechanisms and how these can improve module composition, modularity and reusability in the implementation of Design Patterns in Python. By doing so, it seeks to provide practical insights and guidelines for developers working on complex software projects using Python which are likely to use some of the patterns. Additionally, given the lack of studies regarding Python's support for advanced composition mechanisms, hopefully this study can be a valuable contribution to the scientific and professional understanding of Python's capabilities.

A GitHub repository [35] has also been created as part of this work. It includes the full set of design pattern implementations from Cooper [11] and Ayeva & Kasampalis [3], as well as the enhanced versions of the patterns developed in this thesis that were not showcased.

1.4 Approach

To achieve the objectives of this thesis, the first task will be evaluating the implementation of design patterns in Python of 4 books [3], [11], [18], [36] while primarily using *James Cooper's* book [11]. Cooper's book, published most recently among these sources, will serve as the primary reference due to completeness and comprehensive and detailed examples that will serve as a solid basis for this thesis. It consists of 23 scenarios, respective to each Gang of Four (GOF) [17] design patterns. Important to note that the term scenario, as used in Monteiro's work [31], is used to describe the idea or metaphor that sets up the classes for a given pattern example. Each scenario can give rise to multiple examples, which are specific implementations in different languages or different ways within the same language.

Following the first task, Python's advanced composition mechanisms will be explored in depth and how they affect the implementation of design patterns, attempting to produce new versions (new examples) of the same scenarios that take advantage of these more advanced mechanisms. Alongside the implementation, certain criteria like modularity properties and composition properties that have been used in other literature that are discussed in section 6.1 can be useful in the assessment and evaluation of the implementations. A final analysis will be concluded on the implementation of the design patterns regarding their modularity, module composition and reusability.

1.5 Document Structure

The rest of this document is structured as follows:

- **Chapter 2 (Python)** - This chapter introduces Python, focusing on the features most closely related to modularity, module composition and separation of concerns [10]. It presents Python's advanced mechanisms and their functionality.
- **Chapter 3 (Study Setup)** - This chapter explains the difference between classical and advanced design pattern implementations in Python. It reviews key books, preparing for examples in the next chapter.
- **Chapter 4 (Design Patterns Implementation)** - This chapter presents the GoF design patterns and shows which are directly supported by Python. It then showcases selected implementations using Python's advanced features.
- **Chapter 5 (Discussion of Results)** - This chapter serves as a discussion of the enhanced pattern implementations. It reflects on how Python's features impacted each pattern.
- **Chapter 6 (Related Work)** - This chapter provides a detailed review of existing literature and studies relevant to Python's design patterns and modularity mechanisms.
- **Chapter 7 (Conclusion & Future Work)** - This chapter summarizes the findings of the thesis and outlines possible directions for future work.

2

Python

This chapter intends to give a simple introduction to Python, help readers understand the basics of Python and its key features and also how Python supports advanced mechanisms.

2.1 Python Introduction

Python [37] was created by Guido van Rossum and first released in 1991. It was designed to emphasize code readability and simplicity, following the philosophy of *"There should be one and preferably only one obvious way to do it."*. Python's high-level data structures, extensive standard library and straightforward approach to object-oriented programming makes it very popular among various fields like web development, data science, machine learning, scripting, scientific computing, and many more. Although Python is often seen as an interpreted language and can be slower than compiled languages, its high level of abstraction helps developers handle complex tasks easily and to address performance issues, many developers use tools like *Cython* for compiling Python code or *PyPy*, a Just-In-Time (JIT) compiler that can significantly speed up execution which supports the notion that Python can indeed be used for almost anything.

2.2 Python Interpreter

The Python interpreter is called *CPython* [9] and it is written in the C programming language [23]. When a Python script is ran or commands are typed into the Python interpreter, it follows a series of steps. The first step is called `lexical analysis`. This involves breaking down the source code into tokens, which are the smallest units of the language, such as keywords, identifiers, operators, and literals.(e.g. "if" ; "else" ; "+" ; variables). After tokenization, the interpreter parses these tokens, constructing a parse tree or an abstract syntax tree (AST) that represents the hierarchical structure of the code that is then translated into bytecode.

Bytecode is a low-level representation of the code, similar to machine code but designed for Python's virtual machine, known as the Python Virtual Machine (PVM). The PVM then executes this bytecode, which involves manipulating Python objects and performing operations defined by the Python language. Since Python is dynamically typed, the interpreter determines variable types during runtime. During execution, the interpreter dynamically allocates memory for objects, manages their life cycles, and employs a garbage collector to reclaim memory when objects are no longer needed.

2.3 General Concepts

2.3.1 Variables

A *variable* is a name that refers to a value stored in the computer's memory. These values can be of different types, such as numbers, strings, lists, dictionaries, or any other Python data type.
Instance variables are for data unique to each instance and **Class variables** are for attributes and methods shared by all instances of the class. (See Section 2.4.1 for more information on classes.)

```
class MyClass:
    class_var = 10 # Class variable

    def __init__(self, instance_var):
        self.instance_var = instance_var # Instance variable
```

Listing 2.1: Defining and Accessing Class and Instance Variables

In Python, there aren't ***private*** instance variables that can't be accessed from outside an object. However, a common practice is to prefix a name with an underscore (e.g., _spam). This signals that it's intended for internal use only and shouldn't be accessed directly.

2.3.2 Functions

The keyword *def* introduces a function definition. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented. Functions without a return statement do return a value: "None".

```
def square(x):
  x * x
print(square(5)) // None
```

Listing 2.2: Demonstrating a Function's Default Return of None

2.3.3 Default Arguments & Keyword Arguments

Default arguments allow us to specify a default value for a parameter in the function definition. If the caller doesn't provide a value for that parameter, the default value is used.

```python
def greet(name="World"):
  print("Hello," + name)

greet()
greet("Alice")

# Output:
>>> Hello, World
>>> Hello, Alice
```

Listing 2.3: Function with Default Parameter Value

Keyword arguments allow us to specify arguments by their parameter names when calling a function. This allows us to pass arguments in any order and to skip arguments with default values.

```python
def greet(greeting, name):
  print(greeting + "," + name)

greet(name="Alice", greeting="Hi")
# Output:
>>> Hi, Alice
```

Listing 2.4: Function with Keyword Parameter Value

2.3.4 Special parameters

Since arguments can be passed to a function either by their position or explicitly using keywords, it's helpful to limit how arguments are passed. This way, developers only need to check the function definition to see if arguments are passed by position, a mix of position and keyword, or solely by keyword. When defining a function, special symbols like / and * can be used to indicate how arguments are passed.

- / - separates parameters that can only be passed by position from those that can be passed by position or keyword.
- * - marks the end of positional or positional-or-keyword parameters. After *, all parameters must be passed by keyword only.

```python
def example(pos1, pos2, /, pos_or_kwd, *, kwd1, kwd2):
  print(pos1, pos2, pos_or_kwd, kwd1, kwd2)

# Example call to this function
example(1, 2, 3, kwd1='value1', kwd2='value2')

# Output:
>>> 1 2 3 value1 value2
```

Listing 2.5: Function definition using Special Symbols

In Listing 2.5,

- `pos1` and `pos2` are positional-only parameters. They can only be passed by position.
- `pos_or_kwd` can be passed by position or keyword.
- `kwd1` and `kwd2` are keyword-only parameters. They must be passed by keyword.

The least frequently used option is to specify that a function can be called with an arbitrary number of arguments. A common usage of `*args` is to allow the acceptance of any number of positional arguments, while `**kwargs` is commonly employed to accept any number of keyword arguments.

2.3.5 Data Structures

Data structures in Python are fundamental components used for organizing and storing data efficiently. Python's built-in data structures are efficient and offer good performance for most common use cases. These include:

- **Lists** - Ordered collections of items, which can be of any data type. They are mutable, meaning that elements can be added, removed, or modified.

```
my_list = [1, 2, 3, 4, 5]
my_list.append(6) # Adding an element
print(my_list)
# Output:
>>> [1, 2, 3, 4, 5, 6]
```

Listing 2.6: Example of a List

- **Tuples** - Similar to lists, but they are immutable, meaning that their elements cannot be changed after creation.
- **Sets** - Unordered collections of unique elements. They are mutable, and elements can be added or removed.

```
my_set = {1, 2, 3, 4, 5}
my_set.add(6) # Adding an element
print(my_set)
# Output:
>>> {1, 2, 3, 4, 5, 6}
```

Listing 2.7: Example of a Set

- **Dictionaries** - Key-value pairs, where each key is associated with a value. They provide efficient look-up and insertion operations based on keys.

```python
my_dict = {'name': 'Alice', 'age': 30, 'city': 'New York'}
print(my_dict['name'])
# Output:
>>> Alice
my_dict['age'] = 31 # Modifying an element
print(my_dict)
# Output:
>>> {'name': 'Alice', 'age': 31, 'city': 'New York'}
```

Listing 2.8: Example of a Dictionary

In addition to the built-in data structures, Python offers several other data structures through standard library modules or third-party libraries, such as `collections`, `heapq`, and `array`, as well as the possibility to create custom data structures.

```python
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None

    def append(self, value):
        if not self.head:
            self.head = Node(value)
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = Node(value)

# Example usage
ll = LinkedList()
ll.append(1)
ll.append(2)
print(ll.head.value)
# Output:
>>> 1
print(ll.head.next.value)
# Output:
>>> 2
```

Listing 2.9: Custom Data Structure - LinkedList

2.3.6 Namespaces

A *namespace* is a mapping from names to objects. It serves as a container for variables, functions, classes, and other objects defined in a Python program. In essence, it provides a structured organization for the elements of the code, facilitating clarity and preventing confusion or overlap between different components. The Python interpreter uses namespaces to keep track of the names defined in a program and to resolve name conflicts. The main types of namespaces in Python are:

- **Built-in Namespace** - contains names that are pre-defined in Python's built-in modules. These names are accessible throughout the program.
- **Global Namespace** - contains names defined at the top level of a module or script. It is accessible throughout the entire module.
- **Local Namespace** - contains names defined within a function or method. It is created when the function is called and destroyed when the function exits.

The built-in namespace encompasses the global namespace and the global namespace encompasses the local namespace.

2.3.7 Scopes

A *scope* is a region of a program where a namespace is directly accessible. It determines the visibility and lifetime of names.

- **Built-In Scope** - refers to the namespace containing names that are pre-defined in Python's built-in modules (e.g., built-in types (e.g., int, str)).
- **Global Scope** - refers to the namespace associated with the entire module or script. Names defined at the top level of a module belong to the global scope and are accessible throughout the module.
- **Local Scope** - refers to the namespace associated with a specific function or method. Names defined within a function belong to the local scope and are accessible only within that function.

When a name is referenced, the interpreter initially looks for it in the local scope. If it's not found there, it proceeds to search in the global scope. If the name is still not found, the interpreter continues its search in the built-in scope. The **`global`** and **`nonlocal`** keywords are used to declare variables in different scopes. The `global` keyword is used inside a function to declare that a variable refers to the global scope rather than the local scope. The `nonlocal` keyword is used inside a nested function (a function defined inside another function) to declare that a variable refers to the nearest enclosing scope that is not the global scope.

2.4 Python's Object-Oriented Features

2.4.1 Classes

A *class* is a blueprint for creating objects. It defines the properties (attributes) and behaviors (methods) that objects of the class will have. In Python, classes are created dynamically and are objects, meaning they can be assigned to variables, passed as arguments, and returned from functions.

By default, classes are constructed using `type()`. `type()` is a built-in function that returns the type of an object. However, when given three arguments, it returns a new type object. The parameters required to construct a new class are as follows:

Name The name of the new class, usually given as a string.

Bases A tuple defining the base classes from which the new class inherits. This can be empty if no base classes are needed.

Attrs A dictionary containing the class's attributes and methods, defining its behavior and state.

An example usage of the `type()` function to create a class can be as follows:

```
MyClass = type('MyClass', (object,), {'x': 5})
print(MyClass.x)
# Output:
# >>> 5
```

Listing 2.10: Creating a Class Dynamically with the `type()` Function

Most commonly, to define a class, the `class` keyword is used followed by the class name and a colon. Inside the class attributes and methods are defined. (Important to note that this specific example creates the exact same class as the previous example.)

```
class MyClass:
    x = 5
```

Listing 2.11: Defining a Class

An object is an instance of a class. It represents a unique entity with its own state (attributes) and behavior (methods) based on the class blueprint. To create an object (instance) of a class, call the class as if it was a function. The instantiation operation ("calling" a class object) creates an empty object. It is very useful to create objects with instances customized to a specific initial state. Therefore, a class may define a special method named `__init__()`. This is commonly known as the constructor method.

```
myClass = MyClass()
# creates a new instance of the class and assigns it to myClass.
```

Listing 2.12: Instantiation of a Class

2.4.2 Self Variable

The *self* variable is a reference to the current instance of a class and is used within class methods to access and manipulate instance attributes and methods.(Similar to `this` in Java.) When defining methods within a class, the first parameter in Python conventionally represents the instance itself and is typically named `self`. This parameter is automatically passed when calling methods on an instance of the class. The use of `self` allows Python to differentiate between instance attributes and local variables within a class method. Without `self`, Python would treat variables as local to the method and not as attributes of the instance.

2.4.3 Inheritance

Inheritance allows a class (known as the subclass or derived class) to inherit properties and behaviors from another class (known as the superclass or base class). This relationship establishes an "is-a" connection, where the subclass is considered to be a specialized version of the superclass. A subclass can override methods of the superclass by providing a specialized implementation, and an overriding method can extend rather than simply replace the base class method. In Python, inheritance is implemented by specifying the superclass name in parentheses after the subclass name in the class definition and both single and multiple inheritance are supported. In **single inheritance**, a subclass inherits from only one superclass.

```python
class A:
    pass
class B(A): # B inherits from A
    pass
```

Listing 2.13: Single Inheritance

Multiple inheritance allows a subclass to inherit from multiple superclasses.

```python
class A:
    pass

class B:
    pass

class C(A, B):
    pass
# C inherits from both A and B
```

Listing 2.14: Multiple Inheritance

In **multilevel inheritance**, a subclass inherits from a superclass, and then another subclass inherits from the first subclass.

```python
class A:
    pass
```

```

class B(A):
    pass

class C(B):
    pass
# C inherits from B, and B inherits from A
```

Listing 2.15: Multilevel Inheritance

Hierarchical inheritance involves multiple subclasses inheriting from the same superclass.

```
class A:
    pass

class B(A):
    pass

class C(A):
    pass
# Both B and C inherit from A
```

Listing 2.16: Hierarchical Inheritance

Method Resolution Order (MRO) is the sequence in which Python searches for methods and attributes in a class hierarchy. MRO is from bottom to top and left to right. This means that, first, the method is searched in the class of the object. If it's not found, it is searched in the immediate superclass. In the case of multiple superclasses, it is searched left to right, in the order by which they were declared.

2.4.4 Polymorphism

Polymorphism refers to the ability of different objects to respond to the same method or function call in different ways. In Python, *Method overriding* is a form of polymorphism in which a subclass provides a specific implementation of a method that is already defined in its superclass. This allows objects of different subclasses to respond differently to the same method call.

```
class Animal:
    def make_sound(self):
        print("Animal sound")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")
```

```

dog = Dog()
cat = Cat()

dog.make_sound()
# Output:
# >>> Woof!
cat.make_sound()
# Output:
# >>> Meow!
```

Listing 2.17: Method Overriding Example

Duck Typing is a form of polymorphism and a type system used in dynamic languages(like Python). It emphasizes the object's behavior over its type, so as long as an object supports certain methods or behaviors, it can be used interchangeably with other objects that have the same behavior, even if they belong to different classes or types.

```
class Duck:
    def sound(self):
        print("Quack!")

class Car:
    def sound(self):
        print("Vroom!")

def make_sound(entity):
    entity.sound()

duck = Duck()
car = Car()

make_sound(duck)
# Output:
# >>> Quack!
make_sound(car)
# Output:
# >>> Vroom!
```

Listing 2.18: Duck Typing Example

In Listing 2.18, since both `Duck` and `Car` classes have a `sound()` method, they can be used interchangeably with the `make_sound()` function.

Polymorphism is particularly useful in scenarios where different objects need to be treated in a uniform way. It allows for more flexible and modular code, as new classes can be added without changing existing code.

2.5 Dynamic Behaviour

Python's dynamic typing infers variable types at runtime without necessitating explicit declarations during initialization. Additionally, Python's dynamic typing system requires explicit conversions between incompatible types, preventing unintended type coercion. For example, it is not possible to directly add an integer to a string without explicitly converting one of them.

```python
x = 10
y = "20"
z = x + y
# TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Listing 2.19: TypeError Example

```python
x = 10
y = "20"
z = x + int(y)
print(z) # Output: 30
```

Listing 2.20: Correct Approach

In Python, functions are first-class objects. This means they can be assigned to variables, passed as arguments, returned from other functions, and stored in data structures.

```python
def greet(name):
    return f"Hello, {name}!"

say_hello = greet # Assign function to a variable
print(say_hello("Alice")) # Output: Hello, Alice!
```

Listing 2.21: Using Functions as First-Class Objects

Python also allows methods to be added or replaced at runtime, both at the instance and class level.

```python
class Animal:
    pass

def bark(self):
    print("Woof!")

Animal.speak = bark

dog = Animal()
dog.speak() # Output: Woof!
```

Listing 2.22: Assigning Methods Dynamically

2.6 Meta Object Protocol

The Meta Object Protocol (MOP) [24] is a protocol that allows programmers to modify the behavior of the object-oriented programming language itself. It provides a set of mechanisms to customize and extend the language's object model, making it possible to adapt the language to specific application requirements.

MOP allows objects (metaobjects) to manipulate the behavior and structure of other objects within the system. These metaobjects can control various aspects of class behavior, such as object creation, method invocation, inheritance, and attribute access. This reflective capability provides a way to define how basic operations on objects should be performed, thus offering deep customization of the language's behavior. The concept of a MOP was first implemented in the Smalltalk programming language and later refined in the Common Lisp Object System (CLOS) [24].

Python itself does not have a formal Meta Object Protocol like those found in the previous programming languages but it does it have a set of features that allow for similar levels of customization. One of them is Metaclasses.

2.7 Metaclasses

Metaclasses are the classes that define how other classes are created. Since in Python, everything is an object, classes are objects too. Therefore, classes are instances of metaclasses. As seen in section 2.4.1 the `type()` function can be used to create a class thus `type()` is a metaclass.

Metaclasses allow to exert fine-grained control over class instantiation, attribute handling, and method resolution. The process of creating a class can be customized by passing the `metaclass` keyword argument in the class definition or by inheriting from an existing class that included such an argument.

```
class NameConventionMeta(type):
  def __new__(cls, name, bases, dct):
    for attr_name in dct:
       if not attr_name.startswith('_'):
          raise ValueError(Attributes must start with an underscore.")
    return super().__new__(cls, name, bases, dct)

class MyClass(metaclass=NameConventionMeta):
  _valid_attribute = 5 # This is allowed

  invalid_attribute = 10 # This will raise a ValueError

# Output:
>>> ValueError: Attributes must start with an underscore
```

Listing 2.23: Example use of Metaclasses

In Listing 2.23, a custom metaclass `NameConventionMeta` is defined. This metaclass overrides the `__new__` method, which is called when the class `MyClass` is created. Within the `__new__` method, it iterates over the attributes defined in the class dictionary (`dct`). If any attribute does not start with an underscore, it raises a `ValueError` indicating that the naming convention is violated.

Metaclasses are particularly useful in scenarios where it is needed to enforce certain constraints or automatically modify class definitions. For instance, metaclasses can be used to enforce coding standards, like it was done in the example above to ensure that all attributes start with an underscore.

2.8 Mixin Classes

Mixins are a concept used in object-oriented languages to facilitate code reuse across multiple classes. They were first introduced by *Bracha and Cook* in 1990 [7]. Essentially, mixins are compact, reusable classes that offer a collection of methods and fields, which can be integrated into other classes to enhance their functionality.

Mixins are particularly useful for adding common behavior to multiple classes without resorting to inheritance, which can lead to complex hierarchies. They are used in scenarios where multiple classes need to share the same functionality like for example frameworks or libraries where common behaviors need to be applied across various components. For instance, a popular python framework, `Django` [16], uses mixins to provide additional functionality to class-based views, such as authentication, permissions, and caching.

In Python, a mixin is a parent class that provides functionality to one or more subclasses but is not intended to be instantiated itself which means that they do not imply a "is-a" relationship mentioned in 2.4.3, but rather a "has-a" relationship. For example, if a class `Bird` has a `FlyingCapability` mixin, it means that the bird has the ability to fly, rather than being a type of FlyingCapability. An example of a mixin that saves state (in the example below, an array of messages) and can be used by multiple classes is shown below:

```python
class MessageStorageMixin:
    def __init__(self):
        self.messages = []

    def add_message(self, message):
        self.messages.append(message)

class ErrorHandler(MessageStorageMixin):
    def handle_error(self, error):
        self.add_message(f"Error: {error}")

class Logger(MessageStorageMixin):
    def log(self, info):
        self.add_message(f"Log: {info}")

# Usage
error_handler = ErrorHandler()
error_handler.handle_error("File not found")

logger = Logger()
logger.log("User logged in")

```

```
print(error_handler.messages) # ['Error: File not found']
print(logger.messages) # ['Log: User logged in']
```

Listing 2.24: Example use of a Mixin class

In Listing 2.24, a mixin class `MessageStorageMixin` provides the ability to store messages. Two classes, `ErrorHandler` and `Logger` inherit from the mixin class, which means each of the classes has their own seperate instance of the `messages` list. When `error_handler.handle_error("File not found")` is called it uses the `add_message` method from the mixin class to add the error message to the `messages` list. Similarly, when `logger.log("User logged in")` is called it uses the `add_message` method to add the log message to the `messages` list. Each instance keeps track of its messages independently, which is why the outputs are separate.

2.9 Python Decorators

In very old versions of Python (year 2000s), to change the behavior of a function named firstVersion, another function called firstVersionModifier would be created, and the original function would be replaced with it. Example:

```
def firstVersion():
    ...

firstVersion = firstVersionModifier(firstVersion)
```

Listing 2.25: Old way of modifying a function

When the proposal `PEP318` [42] was accepted, Decorators were introduced in Python. With Decorators, the previous example can be written as:

```
@firstVersionModifier
def firstVersion():
    ...
```

Listing 2.26: Modifying a function using Decorators

Essentially, Decorators are just Syntax Sugar *"for calling whatever is after the decorator as a first parameter of the decorator itself, and the result would be whatever the decorator returns."* [2].

Python decorators allow to modify or extend the behavior of functions or methods without changing their actual code. They are applied using the '@' syntax and involve placing the '@decorator_name' above the function definition. Decorators can even be stacked on top of each other, allowing for a composition of behaviors. This is useful for adding multiple functionalities to a function. Decorators can also take arguments, which is helpful when passing configuration or extra data to the decorator.

A simple usage of a decorator:

```
def simple_decorator(func):
    def wrapper():
```

```python
        print("Something is happening before the function is called.")
        func()
        print("Something is happening after the function is called.")
    return wrapper

@simple_decorator
def say_hello():
    print("Hello!")

say_hello()
# Output:
# Something is happening before the function is called.
# Hello!
# Something is happening after the function is called.
```

Listing 2.27: Basic Decorator Example

In Listing 2.27, 'simple_decorator' is a decorator function that wraps the 'say_hello' function, adding behavior before and after its execution.

Python provides several built-in decorators and there are many third-party libraries available that extend their functionality:

- **@staticmethod**: Converts a method to a static method, which does not require an instance or class.
- **@classmethod**: Converts a method to a class method, which takes the class as its first argument.
- **@property**: Converts a method to a property, allowing it to be accessed like an attribute.

```python
class Square:
    def __init__(self, side_length):
        self.side_length = side_length

    @property
    def area(self):
        return self.side_length ** 2

# Creating an instance of Square
square = Square(4)

# Using the property
print(f"Area of the square: {square.area}") # Output: 16
```

Listing 2.28: Using @property Decorator

2.10 Varieties of Method Dispatch

Dispatch refers to the approach by which a program selects which function or method to execute at runtime based on input parameters, typically the types of the objects involved. This concept is crucial in implementing Polymorphism, allowing the same function or method name to behave differently based on different inputs.

Python natively supports single dispatch(a function behaves differently based on the type of the **first** argument) via the `functools.singledispatch` decorator. This facilitates a basic form of polymorphism by not having to explicitly check types within the function body.

```python
def describe(obj):
    if isinstance(obj, str):
        return f"This is a string: {obj}"
    elif isinstance(obj, int):
        return f"This is an integer: {obj}"
    else:
        return f"This is an object: {obj}"

# Usage
print(describe("Python"))
print(describe(123))
print(describe([1, 2, 3]))
```

Listing 2.29: Function that needs to check the incoming Type of argument

```python
from functools import singledispatch

@singledispatch
def describe(obj):
    return f"This is an object: {obj}"

@describe.register(str)
def _(obj):
    return f"This is a string: {obj}"

@describe.register(int)
def _(obj):
    return f"This is an integer: {obj}"

# Usage
print(describe("Python"))
print(describe(123))
print(describe([1, 2, 3]))
```

Listing 2.30: Emulating Method Overloading with Python Decorators

In Listing 2.30, the `@singledispatch` decorator is used in the base function **describe** to define a generic behavior for any object. The decorator `@describe.register` registers the following functions as the handlers for when the `describe` function is called with a string or an integer.

In the examples below, both listings produce the same output. However, the second one defines the behavior for each type in separate functions.

While Python's standard library supports only single dispatch, multiple dispatch (where the function or method to execute is determined by the types of multiple arguments) can be implemented using third-party libraries, such as `multipledispatch`. Multiple dispatch allows for more sophisticated type-based function overloading by considering all arguments' types, rather than just the first one.

2.11 Abstract Classes

Abstract classes [22] in Python are a mechanism for defining interfaces and creating blueprints for other classes. They are implemented using the abc module, which stands for "Abstract Base Classes." An abstract class can include one or more abstract methods that are declared but contain no implementation. Subclasses of the abstract class are expected to provide concrete implementations for these abstract methods.

To define an abstract class, inherit from `abc.ABC` and use the `@abstractmethod` decorator to mark methods that must be overridden:

```python
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def make_sound(self):
        pass
```

Listing 2.31: Abstract class with an abstract method

In Listing 2.31, `Animal` is an abstract class with an abstract method `make_sound`. Any subclass of `Animal` must implement the `make_sound` method:

```python
class Dog(Animal):
    def make_sound(self):
        return "Woof!"

class Cat(Animal):
    def make_sound(self):
        return "Meow!"
```

Listing 2.32: Example Subclasses of Animal Class

Attempting to instantiate an abstract class directly will result in a `TypeError`:

```python
animal = Animal() # Raises TypeError
```

Listing 2.33: TypeError when instantiating an abstract class

Abstract classes are useful for defining a common interface for a set of subclasses. They ensure that certain methods are implemented in subclasses, promoting consistency across different parts of an application.

2.12 Modules

Modules in Python are files containing Python code. They are one of the most important aspects of Python as they are the backbone units of code organization, allowing developers to logically separate code into different files.

A simple Python module can be for instance:

```python
# my_module.py

def greet(name):
    return f"Hello, {name}!"

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

    def greet(self):
        return f"Hi, {self.name}!"
```

Listing 2.34: Definition of `my_module.py`

The previous module can be imported and used in other Python scripts:

```python
# main.py

import my_module

message = my_module.greet("Alice")
print(message) # Output: Hello, Alice!

greeter = my_module.Greeter("Bob")
print(greeter.greet()) # Output: Hi, Bob!
```

Listing 2.35: Using `my_module.py` in `main.py`

Modules can also be imported by selecting specific functions or classes:

```python
from my_module import greet, Greeter

print(greet("Charlie")) # Output: Hello, Charlie!

greeter = Greeter("Dana")
print(greeter.greet()) # Output: Hi, Dana!
```

Listing 2.36: Selective Import from `my_module.py`

Python also supports organizing modules into packages. A package is a directory containing a special file named `__init__.py`, which can be empty or contain initialization code for the package. Packages allow for hierarchical structuring of modules. For example:

```python
# Directory structure:
# my_package/
# __init__.py
# module_a.py
# module_b.py
```

Listing 2.37: Directory Structure of a Python Package

Modules can be imported from a package using dotted notation:

```
from my_package import module_a

result = module_a.some_function()
```

Listing 2.38: Importing from a Package

Modules and packages enhance modularity by allowing developers to break down complex applications into manageable parts. They enable flexible composition, as modules can be imported and combined in different configurations as needed.

They are fundamental to Python and are widely used in its standard library and popular third-party libraries. Libraries like NumPy [32], Pandas [29] and Django [15] organize code into modules that group related functions and classes. For example, NumPy uses modules like `numpy.linalg` for linear algebra operations and `numpy.fft` for Fourier transforms. This modular structure allows developers to import only the necessary components, helping them share and reuse code easily.

Even though certain functionalities are built into the Python installation, they are organized into modules that must be explicitly imported before use. The `copy` module, for example, is part of the standard library and does not require external installation. It still needs to be imported with `import copy` to access its functions. Other features like random number generation (`import random`), mathematical operations (`import math`), and file system interactions (`import os`) also reside in the standard library. Importing these modules ensures that the namespace remains clean and prevents naming conflicts, reinforcing Python's modular design.

2.13 Reflection

Reflection [27] enables a program to inspect, analyze, and modify its own structure and behavior at runtime. In Python, reflection is a `first-class feature`, deeply integrated into the language. Python provides several built-in functions and features that support reflective programming, such as `type()` that was already mentioned in sections 2.4.1 and 2.7.

Python's reflection capabilities include various functions and modules that allow for introspection and dynamic modification:

- **`type()`**: Determines the type of an object.
- **`getattr()`**: Retrieves the value of an attribute of an object.
- **`setattr()`**: Sets the value of an attribute of an object.
- **`hasattr()`**: Checks if an object has a specific attribute.
- **`dir()`**: Lists the attributes and methods of an object.

- `callable()`: Checks if an object appears callable (i.e., like a function).
- **Inspection Module**: The `inspect` module provides several useful functions to get information about live objects, such as modules, classes, methods, functions, tracebacks, frame objects, and code objects.

Advanced reflection involves techniques that allow deeper introspection and dynamic modifications of a program's behavior at runtime. These techniques extend beyond basic type inspection and attribute access.

Dynamic class and function creation is a powerful feature of advanced reflection. It allows for the generation of new types and behaviors at runtime, based on the current state of the program or input data.

Meta-programming is another advanced reflection technique, involving writing programs that can manipulate themselves or other programs as their data. In Python, this is often achieved using Metaclasses.

2.14 Aspect Oriented Programming

Aspect Oriented Programming (AOP) [25] aims to increase modularity by allowing the seperation of crosscutting concerns. `Crosscutting concerns` are functionalities that affect multiple parts of a program.

Lets say an application has several functions that handle various calculations and data processing tasks. As the application grows, the need for debugging and logging grows as well. Logging is a crosscutting concern.

In Python, AOP isn't built into the core of the language, but using decorators and metaprogramming allows implementing something similar to AOP. A common library used for AOP in Python, *aspectlib* [28] allows for aspects to be implemented via decorators and context managers, making it possible to inject and modify behaviors at `runtime`.

AOP is briefly mentioned in this thesis because many of the studies reviewed in Chapter 6 are based on AOP languages, such as AspectJ, Eos and OT/J. At the start of this study, one of the questions was to what extent Python could rival these AOP languages in terms of results, this was also part of the motivation for mentioning AOP.

Although AOP itself was not explored in this thesis, one possible direction for future work (see Section 7.2) would be to investigate how Python compares to AOP-specific languages more formally, possibly using third-party libraries like *aspectlib*.

3

Study Setup

This chapter explains the difference between classical and advanced implementations of design patterns in Python. It analyzes key books on design patterns and evaluates their coverage preparing the reader for the next chapter.

3.1 Defining Classical and Advanced Implementations in Python

Previous works that this thesis takes inspiration from often compare the implementation of design patterns across two different programming languages [31] [39] [19].

This thesis, however, introduces a key distinction. Instead of comparing different languages, it compares the implementation of design patterns in two different styles within the same language **Python**: a *classical* approach versus an *advanced* approach. It is therefore crucial to first define what distinguishes a *classical* implementation from an *advanced* one and to clarify the differences between both.

For the purposes of this thesis, a *classical* implementation of a design pattern is one that follows the established logic found in traditional object-oriented implementations like in Java, C++, or similar languages. For example, inheritance can be used as the primary mechanism for structuring a pattern. However, Python might have an alternative mechanism that can, in certain situations, bypass the need for inheritance while maintaining or even improving the pattern's structure and functionality. In such cases, the *classical* implementation is the one that adheres to inheritance, while the *advanced* approach explores these alternative mechanisms.

The term *advanced*, while being somewhat presumptuous, whether referring to the implementation of a design pattern or a language mechanism, signifies a less conventional but potentially more *Pythonic* way of structuring the pattern, one that takes advantage of Python's dynamic capabilities.

The notion of *enhancing* an implementation is not always straightforward or objectively measurable. Sometimes, merely improving code readability and maintainability can be considered an enhancement. In other cases, both the *classical* and *advanced* approaches may have their respective advantages and trade-offs, making the assessment of improvement more nuanced.

This discussion is very important to this thesis. By evaluating whether the use of a particular Python mechanism enhances the implementation of a pattern, this work contributes not only to the study of design patterns but also to a deeper understanding of Python's capabilities.

3.2 Review of Python Design Pattern Literature

In this thesis, several key books on Python design patterns were consulted to guide the implementation of various patterns. Each source was evaluated for its comprehensiveness and suitability for implementing advanced composition mechanisms in Python. Below is a table summarizing the books used and the design patterns that were not covered in each.

Book	Missing Patterns
AyevaKasampalis'18 - Mastering Python DPs 2Ed [3]	Composite, Mediator, and Visitor
Cooper'21 - Python Programming with DPs [11]	None
Giridhar'16 - Learning Python DPs [18]	Builder, Prototype, Flyweight, Chain of Responsibility, Interpreter, Iterator, Mediator, Memento, Visitor
Phillips'18 - Python 3 OOP 3rd Ed [36]	Builder, Façade, Mediator, Memento, Visitor

Table 3.1: Books and their coverage of design patterns

Among the books analyzed, Cooper's *Python Programming with Design Patterns* [11] served as the primary reference throughout this thesis. Several factors contributed to this choice. First, it is the most recent among the books considered, reflecting up-to-date best practices in Python. Additionally, Cooper is an established author who has written extensively on design patterns, providing a structured and well-organized approach to each pattern. The book includes implementations for all 23 GoF patterns, making it the most complete reference in terms of coverage. Another key advantage is that Cooper's examples often integrate graphical user interfaces (GUIs), making the implementations easier to follow and visualize.

Ayeva and Kasampalis' *Mastering Python Design Patterns* [3] was also thoroughly analyzed. Many implementations are well-structured, easy to understand, and effectively demonstrate fundamental concepts, making this book the second primary reference for this thesis.

The books *Learning Python Design Patterns* by Giridhar [18] and *Python 3 Object-Oriented Programming* by Phillips [36] were not heavily relied upon in this thesis. Although they provide valuable explanations of fundamental design patterns, their coverage is incomplete, omitting several important patterns. As such, they were not suitable as primary references. However, they were occasionally consulted to verify whether any underexplored mechanisms were utilized in their implementations. When relevant, insights from these books were analyzed, but their role in shaping the thesis was secondary compared to Cooper and Ayeva & Kasampalis.

3.3 Analysis of Mechanisms Used in Book Implementations

Each pattern implementation from Cooper and Ayeva & Kasampalis was analyzed in detail. Table 3.3 summarizes the mechanisms each author used in every pattern.

A brief description of the mechanisms used are defined below(with exception of Metaclasses and Dispatch are covered in sections 2.7 and 2.10):

- **N.A.** – No Python mechanism was used.
- **Inheritance** – A class extends another to reuse or override its behavior.
- **Method Overriding** – A subclass replaces a method from its parent class.
- **Reflection (`getattr`)** – Accesses object attributes dynamically.
- **`@property`** – A decorator that manages attribute access.
- **Override `__new__`** – Customizes how objects are created.
- **Modules (Singleton)** – Uses Python's module system, which initializes only once.
- **`copy` Module** – Creates object copies (shallow or deep).
- **`pyparsing`** – An external library for parsing text-based formats.
- **`pickle`** – Built in module that saves and restores object states.
- **Iterator Protocol (`__iter__`, `__next__`)** – Enables objects to be looped over.

The terms in Table 3.3 link back to these definitions.

Pattern	Cooper	Ayeva & Kasampalis
Abstract Factory	Inheritance	N.A.
Adapter	Inheritance	N.A.
Bridge	Inheritance	N.A.
Builder	Inheritance	N.A.
Chain of Responsibility	Inheritance	N.A.
Command	Inheritance, Method overriding	N.A.
Composite	Inheritance	No implementation
Decorator	Inheritance (GUI extension)	N.A.
Facade	N.A.	N.A.
Factory Method	Inheritance	`@property` decorator
Flyweight	N.A.	Override `__new__`
Interpreter	Reflection (`getattr`)	`pyparsing` for parsing
Iterator	N.A	__iter__, __next__
Mediator	Inheritance, Method overriding	No implementation
Memento	Inheritance	`pickle` for saving/restoring
Observer	Inheritance	N.A.
Prototype	Manual cloning	`copy` module
Proxy	Inheritance, Method overriding	N.A.
Singleton	N.A	Metaclasses
State	Inheritance, Method overriding	N.A
Strategy	Inheritance	N.A.
Template Method	Inheritance, Method overriding	N.A.
Visitor	Inheritance, Dispatch	No implementation

Table 3.2: GoF Patterns (Alphabetical): Cooper and Ayeva&Kasampalis Used Mechanisms

Cooper's implementations rely heavily on inheritance and method overriding, following a traditional object-oriented approach. One possible reason for this reliance is that Cooper previously authored a similar book on design patterns in Java [13], a language where inheritance plays a more central role in structuring patterns.

Ayeva & Kasampalis' implementations introduce some Python-specific mechanisms but still use relatively conventional approaches. Some exceptions include the use of `Metaclasses` in the implementation of Singleton, `@property` decorator in the Factory Method pattern , overriding `__new__` to customize how the class is created in Flyweight, reflection (`getattr`) in Interpreter, and the use of `pickle` in Memento for state persistence, making the pattern almost native to Python.

Despite these exceptions, both books largely do not explore Python's more dynamic capabilities. Features such as metaclasses, mixins or decorators are not considered or are under explored. Some of the books point out the possible use of these in some of the patterns, but do not provide the implementation.

In Chapter 4, selected pattern implementations are presented along with explanations of how Python's mechanisms influenced their structure and behavior.

Chapter 5 presentes the discussion of results regarding these patterns.

4

Design Patterns Implementation

This chapter presents the GoF design patterns and identifies those with direct support in Python, such as Singleton, Iterator, Prototype, Decorator, Interpreter, Memento, and Façade. It highlights how Python's features make explicit implementations of these patterns often unnecessary. The chapter also showcases enhanced implementations of selected patterns, improving on the versions by Cooper and Ayeva & Kasampalis using Python's advanced mechanisms.

4.1 Design Patterns

Design Pattern are reusable solutions to common problems that occur in software design.

They represent solutions that have been developed and refined over time, encapsulating best practices that are adaptable to various development scenarios. When a problem frequently recurs within various contexts, the repeated application of a successful solution naturally evolves into a pattern. They are not concrete implementations but rather templates that describe how to solve a problem in different situations

In 1994 a book titled "Design Patterns: Elements of Reusable Object-Oriented Software" [17] was published by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, which is considered a classic in the field of software design, introducing a catalog of simple and succinct solutions to commonly occurring design problems. The term "Gang of Four" GOF refers to the four authors that defined 23 specific patterns, organizing them into three categories: creational, structural, and behavioral patterns.

- **Creational** - By abstracting the instantiation process, these patterns help make a system independent of how its objects are created, composed, and represented.

- **Structural** - These patterns help ensure that if one part of a system changes, the entire system doesn't need to do the same. They also help in making sure the system is easy to understand and easy to maintain.

- **Behavioral** - These patterns help in defining how objects interact in a manner that increases flexibility in carrying out communication.

Category	**Pattern**
Creational	Factory Method
	Abstract Factory
	Builder
	Prototype
	Singleton
Structural	Adapter
	Bridge
	Composite
	Decorator
	Façade
	Flyweight
	Proxy
Behavioral	Interpreter
	Template Method
	Chain of Responsibility
	Command
	Iterator
	Mediator
	Memento
	Observer
	State
	Strategy
	Visitor

Table 4.1: The GoF design patterns by category

While there is general consensus that design patterns can be categorized into categories some believe that the distinctions between these categories are quite nuanced and can overlap depending on the complexity and specific requirements of individual projects.

4.2 Division of Design Patterns according to Results

As the thesis progressed, the design patterns naturally divided themselves based on the enhancement of their implementation. Some, like the *Prototype* or *Singleton*, are embedded into the language through the use of Python mechanisms. (See Section 4.3)

Others had their classical implementations from the books enhanced. Examples include the *Abstract Factory*, *Command*, and *Observer* patterns. These patterns are showcased in section 4.4 instead of others due to their scenarios being easy to follow, their implementations being concise (some classical implementations from the books relied on mock databases for data, which made their showcase and explanation extremely hard), and because they were well explained in the books used. The point of showcasing the implementation of these patterns is to demonstrate how to take advantage of Python's advanced mechanisms to produce simpler, more flexible versions of the patterns. Nonetheless, many other patterns that do not have their implementation showcased provided valuable results, which will be discussed in Chapter 5.

Based on Python's level of support for these patterns, three distinct groups emerged:

- **Built-in Patterns:** These patterns are naturally supported by Python's language features, effectively making the implementations redundant. Python's native mechanisms inherently fulfill their role, eliminating the need for separate design pattern structures. These include:

  Iterator, Decorator, Prototype, Memento, Singleton, Interpreter and Facade.

- **Enhanced Implementations:** These patterns benefited from Python's advanced features, such as decorators, metaclasses, and dynamic typing. Classical implementations relied on more rigid structures, whereas the enhanced versions use Python's flexibility to improve modularity, readability, and maintainability. These include:

  Proxy, Command, Observer, Strategy, State, Chain of Responsibility, Visitor, Builder, Factory Method, Flyweight, Abstract Factory, and Composite.

- **Minimal or No Enhancement:** Some patterns did not significantly benefit from Python's dynamic features. Either the classical implementation was already well-structured in Python, or the potential improvements did not provide substantial gains. These include:

  Adapter, Template Method, Mediator, and Bridge.

4.3 Built in Patterns through Advanced Mechanisms

The following patterns have direct support from the language:

4.3.1 Iterator Pattern

The *Iterator* pattern provides a way to access elements of a collection sequentially without exposing its underlying representation. Python inherently supports iteration through its built-in iteration protocols, making an explicit implementation of the *Iterator* pattern largely redundant.

Python provides two key methods that allow objects to be iterable:

- `__iter__()`: Returns an iterator object.
- `__next__()`: Returns the next item in the sequence and raises `StopIteration` when there are no more elements.

While these methods form the foundation of iteration in Python, they are typically implemented behind the scenes in built-in iterable objects such as lists, tuples, and dictionaries. Instead of having to manually define iterators, Python allows direct iteration using the `for` loop, which automatically invokes these methods when iterating over a collection.

Older versions of languages such as C++, Java (before Java 5), and C required explicit iterators or index-based loops to traverse collections. In these languages, developers had to manually retrieve elements using iterators or pointer arithmetic. Most modern programming languages, including Python, Java (with `for-each`), and C# (with `foreach`), provide built-in iteration mechanisms that make the explicit *Iterator* pattern unnecessary.

4.3.2 Singleton Pattern

The *Singleton* pattern ensures that a class has only one instance and provides a global point of access to it. This pattern is commonly used for scenarios such as configuration managers, connection pools, or logging systems. In Python, multiple approaches can be used to implement this pattern, but some are more idiomatic than others. Unlike languages such as Java or C++, where enforcing a single instance requires private constructors and static accessors, Python provides built-in mechanisms that naturally achieve this behavior.

Implementing Singleton Using `__new__()`

One of the simplest ways to implement the Singleton pattern in Python is by overriding the `__new__()` method. Since `__new__()` is responsible for creating a new instance of a class, it can be modified to return the same instance every time.

```python
class Singleton:
    _instance = None

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

# Example usage
s1 = Singleton()
s2 = Singleton()
print(s1 is s2) # Output: True
```

Listing 4.1: Singleton Implementation Using `__new__()`

This approach ensures that only one instance of `Singleton` is created, as subsequent calls to instantiate the class return the same object.

The Pythonic Approach: Singleton via Modules

While the `__new__()` method provides a valid Singleton implementation, Python's module system naturally enforces the same behavior. When a module is imported, it is initialized only once and stored in `sys.modules`. Any subsequent import retrieves the same module instance, making explicit Singleton logic unnecessary.

A more Pythonic way to achieve the Singleton pattern is to store the shared instance inside a module.

```python
# singleton_module.py
class Singleton:
    def __init__(self, name):
        self.name = name

singleton_instance = Singleton("Alan")
```

Listing 4.2: Using a Module as a Singleton

In another file, the singleton instance can be accessed directly:

```python
from singleton_module import singleton_instance

def main():
    s1 = singleton_instance
    s2 = singleton_instance
    print(s1 is s2) # Output: True
```

Listing 4.3: Accessing the Singleton Module Instance

Since modules are only loaded once per interpreter session, the instance remains persistent throughout the application. This approach eliminates additional complexity, making it the preferred way to implement Singleton in Python.

Implementing Singleton using Metaclasses

Another way to enforce the Singleton pattern is by using metaclasses, as seen in Ayeva & Kasampalis' *Mastering Python Design Patterns* [3]. Instead of modifying `__new__()`, the Singleton behavior is encapsulated in a metaclass, overriding `__call__()` to ensure only one instance is created.

```python
    class SingletonType(type):
    _instances = {}
    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            cls._instances[cls] = super(SingletonType, cls).__call__(*args, **kwargs)
        return cls._instances[cls]


class URLFetcher(metaclass=SingletonType):
```

Listing 4.4: Singleton Implementation by Ayeva & Kampalis

While metaclasses provide an elegant way to enforce Singleton behavior at the class level, they are not necessary for most Python applications. The module-based approach remains the cleanest and most Pythonic solution.

Python inherently supports the Singleton pattern through its module system, making explicit implementations unnecessary in many cases. While approaches such as modifying `__new__()` or using metaclasses can enforce Singleton behavior, they introduce additional complexity. The module-based approach provides a straightforward and efficient way to achieve Singleton functionality with minimal code.

4.3.3 Prototype Pattern

The *Prototype* pattern is a creational design pattern that allows objects to be cloned from existing ones without relying on their concrete classes. This pattern is particularly useful when creating a new instance of a class is resource-intensive or complex, and copying an existing instance is more efficient. By cloning existing objects, the Prototype pattern enables the creation of new objects with the same properties as the originals, without the overhead of re-initializing them.

Python inherently supports the Prototype pattern through its built-in `copy` module, which provides both shallow and deep copy operations. This means the objectives of the Prototype pattern can be achieved without implementing additional code, simply by using Python's copying mechanisms.

Ayeva & Kasampalis [3] also relied on the `copy` module for their Prototype implementation, demonstrating that Python's built-in features already provide the intended behavior of this pattern.

Demonstration of Shallow and Deep Copy in Python

```
import copy
obj_clone = copy.copy(existing_obj) # Shallow copy
deep_clone = copy.deepcopy(existing_obj) # Deep copy
```

Listing 4.5: Demonstration of Shallow and Deep Copy in Python

In this example, both shallow and deep copies of an existing object are created. The `copy()` function creates a shallow copy, while `deepcopy()` creates a deep copy.

- **Shallow Copy**: Creates a new object but inserts references to the items found in the original. Changes to mutable attributes in the shallow copy may affect the original object because they share the same references.

- **Deep Copy**: Creates a new object and recursively adds copies of nested objects found in the original. Changes to mutable attributes in the deep copy do not affect the original object, as the deep copy contains entirely separate copies of the nested objects.

To observe the differences between shallow and deep copies, let's modify the `value` of a child in both copies:

```
from copy import copy, deepcopy

```

```
original = [1, 2, [3, 4]]

# Shallow copy
shallow_copied = copy(original)

# Deep copy
deep_copied = deepcopy(original)

# Modify the nested list in the shallow copy
shallow_copied[2][0] = 99

# Modify the nested list in the deep copy
deep_copied[2][1] = 100

print(original) # Output: [1, 2, [99, 4]]
print(shallow_copied) # Output: [1, 2, [99, 4]]
print(deep_copied) # Output: [1, 2, [3, 100]]
```

Listing 4.6: Modifying Copies and Observing Effects on the Original

As seen in the output, modifying the `value` of a child in the shallow copy also affected the original object because they share references to the same child objects. In contrast, modifying the deep copy did not affect the original object, as the deep copy contains entirely separate copies of the nested objects.

With this, the need for implementing the pattern largely disappears, as the language itself provides the core behavior the pattern is intended to achieve.

4.3.4 Decorator Pattern

The *Decorator* pattern is a structural pattern that allows behavior to be added to individual objects, either statically or dynamically, without affecting the behavior of other objects from the same class.

> *"Suppose that we have a program that uses eight objects, but three of them need an additional feature. You could create a derived class for each of these objects, and in many cases, this would be a perfectly acceptable solution. However, if each of these three objects requires different features, this would mean creating three derived classes. Further (...) create complexity that is both confusing and unnecessary."* [11].

James Cooper explains the benefits of using the *Decorator* pattern in such cases. To showcase how the Decorator pattern is naturally embedded into Python, Cooper's implementation is analyzed and the enhancement of the implementation is presented.

Cooper's Scenario and Implementation

Cooper introduces a scenario of decorating buttons with animations when hovering over them. His implementation follows a classical approach using inheritance, where a base

`Decorator` class is extended by custom button classes.

```python
from tkinter import *

# Base decorator class
class Decorator(Button):
    def __init__(self, master, **kwargs):
        super().__init__(master, **kwargs)
        self.configure(relief=FLAT)
        self.bind("<Enter>", self.on_enter)
        self.bind("<Leave>", self.on_leave)

    def on_enter(self, evt):
        self.configure(relief=RAISED)

    def on_leave(self, evt):
        self.configure(relief=FLAT)

# Custom buttons extending Decorator
class CButton(Decorator):
    def __init__(self, master, **kwargs):
        super().__init__(master, text="C Button")

class DButton(Decorator):
    def __init__(self, master, **kwargs):
        super().__init__(master, text="D Button")

# Builder class to construct the interface
class Builder():
    def build(self):
        root = Tk()
        cbut = CButton(root)
        dbut = DButton(root)
        qbut = Button(root, text="Quit", command=quit)
```

Listing 4.7: Cooper's Implementation of the Decorator Pattern

In Listing 4.7, Cooper uses the `Tkinter` library [38] to create custom buttons with hover effects. The `Decorator` class extends the `Button` widget, changing its appearance when the mouse hovers over it by switching the relief (border style) between FLAT and RAISED.

This code produces the following interface:

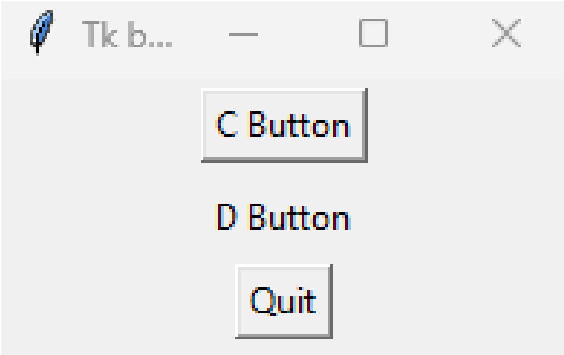

Figure 4.1: Cooper's Implementation: Hover Effects Using Inheritance

This approach has several drawbacks: The hover effect is tightly coupled to the base class, adding new behavior requires modifying the class hierarchy ane because of this the Open/Closed Principle is broken by forcing subclass modifications.

Enhanced Implementation Using Python's Decorators

Rather than relying on inheritance, we use Python decorators. The Decorator pattern and Python's decorators (See Section 2.9) are not the same thing and while they serve similar purposes, adding functionality, the mechanisms differ. However, it is possible to implement the Decorator pattern using Python's decorators. By utilizing Python's @decorator syntax, classes and functions can be modified at definition time, effectively adding functionality in a way that mirrors the intent of the Decorator pattern.

Below is the definition of a decorator function that adds an hover effect:

```
import tkinter as tk

# Decorator to add hover effect
def hoverable(cls):
    original_init = cls.__init__

    def __init__(self, *args, **kwargs):
        original_init(self, *args, **kwargs)
        self.configure(relief=tk.FLAT)
        self.bind("<Enter>", self.on_enter)
        self.bind("<Leave>", self.on_leave)

    def on_enter(self, event):
        self.configure(relief=tk.RAISED)

    def on_leave(self, event):
        self.configure(relief=tk.FLAT)

    cls.__init__ = __init__
    cls.on_enter = on_enter
    cls.on_leave = on_leave
```

```
    return cls
```

Listing 4.8: Decorator to Add Hover Effect

Instead of defining the hover effect within a base class, this `@hoverable` decorator adds the behavior dynamically to any class it is applied to.

Another example is styling:

```
def stylable(font=("Helvetica", 12)):
    def decorator(cls):
        original_init = cls.__init__

        def __init__(self, *args, **kwargs):
            original_init(self, *args, **kwargs)
            self.configure(font=font)

        cls.__init__ = __init__
        return cls
    return decorator
```

Listing 4.9: Decorator to Add Custom Font Styling

Now, applying these decorators to button classes eliminates the need for subclassing:

```
@hoverable
@stylable(font=("Helvetica", 14))
class CButton():
    def __init__(self, master, **kwargs):
        super().__init__(master, text="C Button", **kwargs)

@hoverable
class DButton():
    def __init__(self, master, **kwargs):
        super().__init__(master, text="D Button")

class Builder:
    def build(self):
        root = tk.Tk()
        c_button = CButton(root)
        d_button = DButton(root)
        q_button = tk.Button(root, text="Quit", command=root.quit)
        c_button.pack()
        d_button.pack()
        q_button.pack()
        root.mainloop()
```

Listing 4.10: Applying Decorators to Buttons

This results in the following interface:

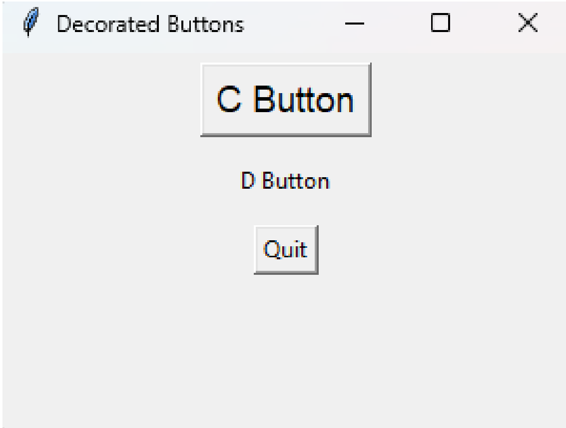

Figure 4.2: Enhanced Implementation: Hover Effects Using Python Decorators

Since we added the @stylable decorator to the C button, it is bigger than the D button, effectively showcasing that adding functionality can be done in one line of code.

Table 4.2 highlights the key differences between Cooper's inheritance-based approach and the enhanced decorator-based implementation.

Cooper (Inheritance)	**Enhanced (Decorators)**
Requires modifying base class or subclassing	New behaviors can be added dynamically via decorators
Tightly coupled to class hierarchy	Easily removable or extendable

Table 4.2: Comparison of Cooper's and Enhanced Implementations

Python's `@decorator` mechanism makes the explicit *Decorator* pattern unnecessary, as the language natively supports its intent in a more flexible way.

4.3.5 Memento Pattern

The *Memento* pattern is a behavioral design pattern that provides the ability to restore an object to a previous state. It is particularly useful when implementing undo functionality, allowing the system to revert to earlier versions without exposing its internal structure. The pattern normally consists of three main components:

- **Originator**: The object whose state needs to be saved and restored.
- **Memento**: A storage mechanism for the state.
- **Caretaker**: Manages the history of stored states.

Python simplifies the implementation of this pattern through its built-in `pickle` module, which allows objects to be serialized (saved) and deserialized (restored). Serialization refers to converting an object into a format that can be stored or transmitted and later reconstructed.

Introduction to the pickle Module

The pickle module is a standard Python library used for serializing and deserializing objects.

Key pickle functions include:

- `pickle.dumps(obj)` – Serializes an object into a byte stream.
- `pickle.loads(data)` – Deserializes a byte stream back into an object.
- `pickle.dump(obj, file)` – Serializes and writes an object to a file.
- `pickle.load(file)` – Reads and deserializes an object from a file.

Cooper's Approach

James Cooper provides an implementation of the Memento pattern in the context of graphical objects. His approach uses a combination of classes to manage states manually. Instead of using Python's built-in serialization capabilities, his implementation maintains a separate `Memento` class that stores object properties explicitly. While this adheres to the classical definition of the pattern, it introduces additional complexity that can be avoided in Python.

Ayeva & Kasampalis' Approach

In contrast, Ayeva & Kasampalis' implementation takes advantage of Python's `pickle` module.

```
import pickle

class Quote:
    def __init__(self, text, author):
        self.text = text
        self.author = author

    def save_state(self):
        return pickle.dumps(self.__dict__)

    def restore_state(self, memento):
        self.__dict__.clear()
        self.__dict__.update(pickle.loads(memento))

    def __str__(self):
        return f'{self.text} - By {self.author}.'

def main():
    print('Quote 1')
    q1 = Quote("A room without books is like a body without a soul.",
               'Unknown author')
```

```
    print(f'\nOriginal version:\n{q1}')
    q1_mem = q1.save_state()

    # Now, we found the author's name
    q1.author = 'Marcus Tullius Cicero'
    print(f'\nWe found the author, and did an update:\n{q1}')

    # Restoring previous state (Undo)
    q1.restore_state(q1_mem)
    print(f'\nWe had to restore the previous version:\n{q1}')

if __name__ == "__main__":
    main()
```

Listing 4.11: Ayeva & Kasampalis' implementation of the Memento pattern

This implementation uses pickle.dumps() to serialize an object's complete state and pickle.loads() to restore it. The `save_state()` method stores the current state, while the `restore_state()` method reverts the object to a previous state, effectively implementing the Memento pattern with minimal additional code.

Unlike languages where manual state management is necessary, Python's built-in serialization mechanisms make the Memento pattern largely redundant. By using `pickle`, the pattern is seamlessly integrated into the language. This eliminates the need for explicit *Caretaker* and *Memento* classes, significantly reducing complexity while maintaining the intended functionality of the pattern.

4.3.6 Façade Pattern

The *Façade* pattern is a structural design pattern that provides a simplified interface to a complex system. Instead of exposing the underlying implementation details of multiple interacting components, the Façade pattern offers a unified, high-level API, making the system easier to use.

Cooper's Implementation

Cooper implements the Façade pattern by creating a interface for interacting with a MySQL database. His implementation involves classes for managing queries, retrieving data, and handling database connections. The interface simplifies database operations through a graphical user interface (GUI) built with Tkinter.

Kasampalis' Implementation

Kasampalis applies the Façade pattern in an operating system simulation, where an `OperatingSystem` class acts as a high-level interface, encapsulating two complex subsystems: the `FileServer` and the `ProcessServer`. This implementation demonstrates a classical use of Façade by providing simple methods (`create_file()` and `create_process()`)

to interact with the system without exposing the underlying complexity.

Python's Module System Achieves the Façade Pattern

Although the book's implementations are completly valid, Python can naturally support the Façade pattern through its module system. Modules act as natural façades by encapsulating functionality within a single namespace and exposing a simple API.

For example, instead of defining an explicit Façade class, one could structure a Python module that internally manages complex subsystems while providing an interface:

```python
# file_management.py (Façade module)
import os

def create_file(name, content=""):
    with open(name, "w") as f:
        f.write(content)

def delete_file(name):
    os.remove(name)
```

Listing 4.12: Python's Module System as a Façade

```python
# main.py
import file_management
file_management.create_file("example.txt", "Hello, Façade!")
file_management.delete_file("example.txt")
```

Listing 4.13: Calling Methods from the Façade Example Module

In Listing 4.12, `file_management.py` acts as a façade by providing high-level functions (`create_file()` and `delete_file()`) that internally handle system-level operations, abstracting the underlying complexity.

Both Cooper and Kasampalis implement the Façade pattern, but Python's module system already achieves its purpose. Modules serve as façades by organizing code and managing interactions with complex systems. Because of this, the Façade pattern is naturally part of Python, making an explicit implementation unnecessary.

4.3.7 Interpreter Pattern

The *Interpreter* pattern is a behavioral design pattern that provides a way to evaluate and execute sentences in a defined language or grammar. It is commonly used for parsing expressions, interpreting commands, or implementing domain-specific languages.

Many programming languages require an explicit implementation of the *Interpreter* pattern to parse and evaluate expressions. However, Python provides built-in mechanisms, such as `eval()` and the `ast` module, which implement the objectives of the pattern, making an explicit implementation largely unnecessary.

Cooper's' Implementation

Cooper provides an implementation of the *Interpreter* pattern that interprets textual commands to manipulate and sort a dataset. His approach relies on **reflection** using `getattr()` to dynamically resolve attributes and execute commands. This approach follows the classical implementation but can be simplified using Python's built-in parsing capabilities.

Ayeva & Kasampalis' Implementation

Kasampalis presents an implementation using `pyparsing`, an external library designed for parsing structured text. While effective, `pyparsing` is not part of Python's standard library, meaning that its use requires additional dependencies. However, Python natively provides parsing capabilities without requiring external libraries.

Python's Built-in Interpreter Implementation

Python eliminates the need for an explicit implementation of the *Interpreter* pattern through built-in evaluation functions. The `eval()` function directly interprets expressions, while the `ast` module provides a safer alternative by evaluating only simple literals.

```
expression = "2 + 3 * 5"
result = eval(expression)
print(result) # Output: 17

user_input = "print('Hacked!')"
eval(user_input) # Output: Hacked!
```

Listing 4.14: Using `eval()` for Expression Evaluation

While `eval()` is powerful, it can execute arbitrary code (as seen in the example above when executing the print function), making it unsafe for untrusted input. The `ast.literal_eval()` function is a safer alternative, restricting evaluation to simple literals.

```
import ast

expression = "{'x': 10, 'y': 20}"
result = ast.literal_eval(expression)
print(result) # Output: {'x': 10, 'y': 20}
```

Listing 4.15: Using `ast.literal_eval()` for Safe Parsing

Both Cooper and Kasampalis provide valid implementations of the *Interpreter* pattern, but Python inherently supports its objectives through built-in parsing mechanisms. By using `eval()` and `ast`, Python allows for the direct evaluation of expressions without requiring a separate interpreter structure. Consequently, the *Interpreter* pattern is effectively built into Python, making an explicit implementation unnecessary in many cases.

4.4 Examples of Patterns Enhanced through Advanced Mechanisms

This section serves to showcase the implementation and enhancement of some patterns. Since Ayeva & Kasampalis implementations are very similar to Cooper's, only Cooper's is showcased. The solutions provided to the issues ahead are valid to both scenarios.

4.4.1 Abstract Factory Pattern

The *Abstract Factory* pattern is a creational design pattern that provides an interface for creating families of related objects without specifying their concrete classes. The primary purpose of this pattern is to encapsulate object creation logic and ensure that a system uses only objects from a specific family.

Cooper's Implementation

Cooper's implementation follows a traditional *Abstract Factory* approach where a `Garden` interface defines a set of required methods, and different concrete gardens (`VeggieGarden`, `AnnualGarden`, and `PerennialGarden`) implement them. Below is a partial implementation showcasing the pattern logic. Some classes and details have been omitted for brevity.

```
class Garden:
    def getShade(self): pass
    def getCenter(self): pass
    def getBorder(self): pass

class VeggieGarden(Garden):
    def getShade(self):
        return Plant("Broccoli")
    def getCenter(self):
        return Plant("Corn")
    def getBorder(self):
        return Plant("Peas")
```

Listing 4.16: Cooper's Implementation of the Abstract Factory Pattern

However, as noted by Cooper [11], this design can introduce some challenges:

> *"Adding new class families takes some effort because you need to define new, unambiguous conditions that cause such a new family of classes to be returned. " [11]*

This issue can be solved in a simplified manner:

Using Metaclasses for Automatic Method Generation

To enforce method presence across all concrete factories,a metaclass can be used instead of hardcoded base class methods. This ensures required methods exist while still allowing custom methods per subclass.

```
class GardenMeta(type):
    REQUIRED_METHODS = ['get_shade', 'get_center', 'get_border']

    def __new__(cls, name, bases, dct):
        for method in cls.REQUIRED_METHODS:
            if method not in dct:
                dct[method] = lambda self: f"{name} has no {method}"
        return super().__new__(cls, name, bases, dct)

class Garden(metaclass=GardenMeta):
    pass

class BonsaiGarden(Garden):
    def get_shade(self):
        return "Bonsai prefers partial shade"

    def watering_frequency(self):
        return "Water every two days"

class VeggieGarden(Garden):
    def get_center(self):
        return "Tomatoes grow in the center"

bonsai = BonsaiGarden()
veggie = VeggieGarden()

print(bonsai.get_shade()) # Defined in subclass
print(bonsai.get_border()) # Auto-generated
print(veggie.get_center()) # Defined in subclass
print(veggie.get_shade()) # Auto-generated
```

Listing 4.17: Metaclass-Enforced Method Creation in Garden Classes

This approach ensures that all necessary methods exist while allowing additional subclass-specific methods without breaking the hierarchy.

Cooper points our another issue:

> *"Although all the classes that the Abstract Factory generates have the same base class, there is nothing to prevent some subclasses from having additional methods that differ from the methods of other classes. This presents the same problem as occurs in any subclasses: you don't know whether you can call a class method unless you know whether that subclass is one that allows those methods".*

And while this may be true, and no solution completely eliminates this problem, Python provides a straightforward way to handle it with minimal code.

Duck Typing and Attribute Checking

Python allows checking for the presence of a method at runtime using duck typing through the `hasattr()` function. It enables method calls only when they are available, eliminating the need for complex type checking or modifying the base class to accommodate methods that do not apply to all subclasses.

```
class BonsaiGarden:
    def watering_frequency(self):
        return "Water every two days"

class VeggieGarden:
    def harvest_time(self):
        return "Harvest in three months"

def garden_info(garden):
    print(f"Garden Type: {garden.__class__.__name__}")

    if hasattr(garden, 'watering_frequency'):
        print("Watering:", garden.watering_frequency())

    if hasattr(garden, 'harvest_time'):
        print("Harvest:", garden.harvest_time())

bonsai = BonsaiGarden()
veggie = VeggieGarden()

garden_info(bonsai) #Output: Garden Type: BonsaiGarden Watering: Water every two days
garden_info(veggie) #Output: Garden Type: VeggieGarden Harvest: Harvest in three months
```

Listing 4.18: Using Duck Typing to Handle Subclass-Specific Methods

In Listing 4.18, the `garden_info()` function dynamically checks whether an instance has a specific method before calling it.

Both solutions provided, Metaclasses and the Duck Typing approach, are suggestions to address the issues highlighted by Cooper. However, other approaches are equally valid. For instance, Python decorators could be used to dynamically add extra methods to subclasses, eliminating the need to modify either the base class or the subclasses directly. These examples demonstrate how Python can solve the challenges Cooper describes with minimal code.

4.4.2 Command Pattern

The Command pattern is a behavioral design pattern that encapsulates a request as an object. This allows parameterization of clients with queues, requests, and operations, and supports undoable actions.

Cooper's Scenario and Implementation

In Cooper's implementation, a simple graphical application lets users draw red or blue lines on a canvas and undo the last action. The key components are:

- `Command` interface with `execute` and `undo` methods.
- `CommandStack` that manages executed commands and supports undo.
- Concrete command classes that implement drawing actions.

When a user clicks the "Red" or "Blue" button, a command object is created, executed, and stored in the stack. The "Undo" button retrieves the last command from the stack and calls its `undo` method.

The `Command` interface defines the `execute` and `undo` methods. Concrete command classes implement these methods to perform drawing actions and reverse them. The `CommandStack` maintains a history of executed commands, enabling the undo functionality.

Command Interface

```
class Command():
    def execute(self): pass
    def undo(self): pass
```

Listing 4.19: Command interface defining execute and undo methods

Each command must implement `execute` to perform an action and `undo` to reverse it.

CommandStack

```
class CommandStack():
    def __init__(self, canvas):
        self.commands = [] # stack of commands
        self.canvas = canvas

    def addDraw(self, command):
        self.commands.append(command)
        command.execute() # perform the action

    def redraw(self):
        self.canvas.delete('all')
        for comd in self.commands:
            comd.execute()

    def undo(self):
        comd = None
```

```
        if len(self.commands) > 0:
            comd = self.commands.pop()
            self.redraw()
        return comd
```

Listing 4.20: CommandStack for managing command history

The CommandStack stores executed commands, allowing to undo them. Adding a command executes it and pushes it onto the stack. Undoing removes the last command and triggers a redraw of the remaining commands.

ButtonCommand and Related Commands

```
class ButtonCommand(Command):
    def __init__(self, button, x1, y1, x2, y2, color):
        self.canvas = button.getCanvas()
        self.button = button
        self.x1, self.x2 = x1, x2
        self.y1, self.y2 = y1, y2
        self.color = color

    def execute(self):
        self.canvas.create_line(
            self.x1, self.y1, self.x2, self.y2, fill=self.color)

    def undo(self):
        self.button.undo()
```

Listing 4.21: ButtonCommand base class

ButtonCommand is a concrete command that draws a line. It retrieves the coordinates and color, and executes by drawing. Its undo delegates to the button's undo logic.

DButton

```
class DButton(Button, Command):
    def __init__(self, master, **kwargs):
        super().__init__(master, command=self.execute, **kwargs)
    def execute(self): pass
```

Listing 4.22: DButton abstract button class

DButton links a button interface to the Command interface. Pressing the button triggers execute. Although it extends a GUI button class, the focus here is its role as a command trigger.

RedButton and BlueButton

```
class RedButton(DButton):
def __init__(self, root, canvas:Canvas,stack:CommandStack):
   super().__init__(root, text="Red")
   self.root = root
   self.canvas=canvas
   self.stack = stack
   self.x = self.y = 0
# create a button command at the stack and draw it
def execute(self):
   bcomd = ButtonCommand(self,
         self.x,self.y,
         self.x+20,self.y+150,'red')
   self.x += 20 # move red to right
   self.stack.addDraw(bcomd) #push it and draw it

#returns canvas
def getCanvas(self):
   return self.canvas

# resets x coord back one line
def undo(self):
   self.x -= 20
```

Listing 4.23: RedButton command logic

`RedButton` creates a `ButtonCommand` for drawing a red line. Its `execute` updates coordinates, creates a command, and uses `addDraw` to execute and store the command. `BlueButton` mirrors the `RedButton` logic with its own coordinates and color.

When the Red button is clicked, the `execute()` method is called. This method creates a new ButtonCommand instance and it is added to stack. Inside the `addDraw()` method. command.execute() is called, which actually draws the line.

Below is the output of Cooper's code after clicking:

$$\text{Red} \rightarrow \text{Red} \rightarrow \text{Red} \rightarrow \text{Red} \rightarrow \text{Blue} \rightarrow \text{Blue} \rightarrow \text{Blue} \rightarrow \text{Blue} \rightarrow \text{Blue}.$$

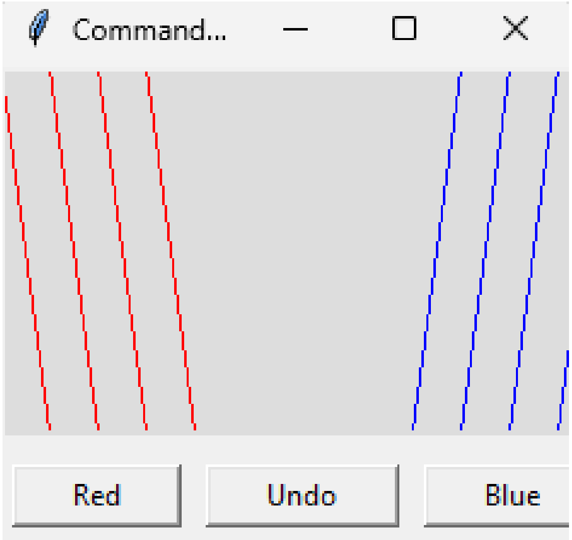

Figure 4.3: Cooper's Graphical Interface for the Command Pattern.

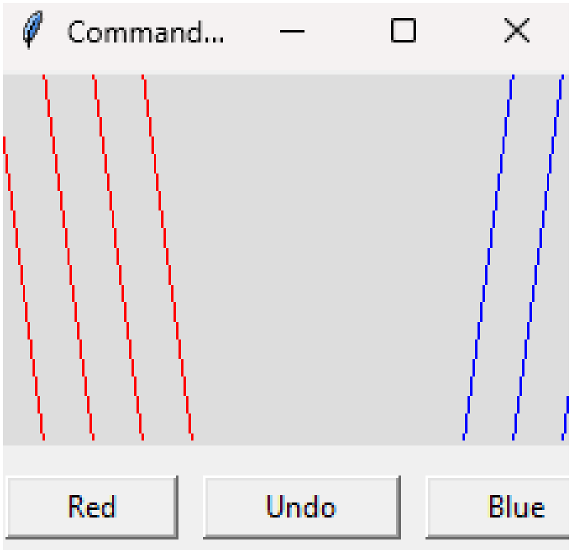

Figure 4.4: Cooper's Graphical Interface for the Command Pattern After one Undo.

In this implementation, each button (RedButton and BlueButton) is responsible for creating a command(ButtonCommand), executing the command and adding it to the stack. Which means buttons are not just UI elements, but also part of the logic, making this implementation tightly coupled. Ideally the buttons should only trigger actions.

Enhanced Implementation of the Command Pattern

Below is an enhanced version of the Command pattern implementation. It uses abstract base classes, dataclasses and python decorators .This approach separates concerns more clearly, makes it easier to add new commands or modify existing ones, and removes dependencies on GUI elements from the core command logic.

Abstract Command Interface

```python
from abc import ABC, abstractmethod
from typing import Optional

class Command(ABC):
    @abstractmethod
    def execute(self) -> None:
        pass

    @abstractmethod
    def undo(self) -> None:
        pass
```

Listing 4.24: Abstract Command interface

This interface defines the required methods. All commands must implement `execute` and `undo`.

CommandStack

```python
class CommandStack:
    def __init__(self):
        self._commands = []

    def execute_command(self, command: Command) -> None:
        command.execute()
        self._commands.append(command)

    def undo(self) -> None:
        if self._commands:
            last_command = self._commands.pop()
            last_command.undo()
```

Listing 4.25: CommandStack with execute and undo operations

The `CommandStack` manages command execution history. Calling `execute_command` runs a command and stores it. Calling `undo` reverts the most recent command. This stack is independent of any user interface.

LineCommand as a Dataclass

```python
from dataclasses import dataclass

@dataclass
class LineCommand(Command):
    canvas: "CanvasInterface" # A simplified interface for drawing
    x1: int
    y1: int
```

```
    x2: int
    y2: int
    color: str
    line_id: Optional[int] = None

    def execute(self) -> None:
        self.line_id= self.canvas.draw_line(self.x1, self.y1, self.x2, self.y2, self.color)

    def undo(self) -> None:
        if self.line_id is not None:
            self.canvas.delete_line(self.line_id)
```

Listing 4.26: A LineCommand using a dataclass

`LineCommand` (previously named `ButtonCommand`, renamed to better reflect its function) showcases the built-in `@dataclass` decorator. By using it, the amount of code needed to define and manage class data is reduced. The dataclass automatically generates methods like `__init__` based on the fields listed, so the focus is on the command logic rather than writing boilerplate code.

@execute_and_store Decorator for Command Creation

```
def execute_and_store(stack: CommandStack):
    def decorator(func):
        def wrapper(*args, **kwargs):
            cmd = func(*args, **kwargs)
            stack.execute_command(cmd)
            return cmd
        return wrapper
    return decorator
```

Listing 4.27: Decorator for executing and storing commands automatically

The `execute_and_store` decorator automates command execution and storage, reducing repetitive code. It ensures that any command created and returned by the decorated function is immediately executed and pushed onto the stack.

CommandFactory (New class)

```
class CommandFactory:
    def __init__(self, canvas: "CanvasInterface", stack: CommandStack):
        self.canvas = canvas
        self.stack = stack
        self.red_x = 0
        self.blue_x = 300

    @execute_and_store(stack)
    def create_red_line_command(self) -> LineCommand:
        cmd = LineCommand(self.canvas, self.red_x, 0, self.red_x + 20, 150, 'red')
```

```
            self.red_x += 20
            return cmd

    @execute_and_store(stack)
    def create_blue_line_command(self) -> LineCommand:
        cmd = LineCommand(self.canvas, self.blue_x, 0, self.blue_x - 20, 150, 'blue')
        self.blue_x -= 20
        return cmd
```

Listing 4.28: CommandFactory for creating commands

When a button is clicked, it simply calls either create_blue_line_command or create_red_line_command. The rest is handled by CommandFactory. The `CommandFactory` centralizes the creation of commands and uses the decorator to ensure that each created command is executed and stored. Adjusting positions or adding new command types can be done here without affecting other parts of the code.

By doing this, the drawing logic is isolated from the buttons that trigger it, making it easier to add or modify commands.

The `execute_and_store` decorator automates command execution and storage, removing repetitive logic and simplifying the addition of new commands.

By using the built-in @dataclass decorator and the custom @execute_and_store decorator, the pattern became simpler, more extensible, and easier to implement.

4.4.3 Observer Pattern

The Observer pattern is a behavioral design pattern that establishes a one-to-many dependency between objects, allowing multiple observers to be notified and updated automatically whenever a subject changes state. The basis of this pattern is loose coupling, as the subject does not need to know the specifics of its observers.

4.4.3.1 Cooper's Scenario and Implementation

In Cooper's implementation, the scenario involves a GUI application where a user selects a color from a set of radio buttons in one window (the subject). This selection is automatically reflected in two separate windows (the observers): one displays the selected color as text in a list, and the other updates its background color.

The three main components in Cooper's code are:

- `ColorRadio`: The *Subject*, which manages the selected color and notifies observers of changes.
- `ColorFrame`: An *Observer* that reacts by updating its background color.
- `ColorList`: Another *Observer* that adds the selected color name to a list.

Observer And Subject interfaces

The `Observer` interface requires implementing a `sendNotify` method, ensuring that all observers respond to notifications. The `Subject` interface requires a `registerInterest` method.

```
# Observer interface
class Observer:
    def sendNotify(self):
        pass

# Subject interface
class Subject:
    def registerInterest(self, obs:Observer):
        pass
```

Listing 4.29: Cooper's Observer and Subject interfaces

Concrete Observer Classes

The `ColorList` observer appends the name of the selected color to a list, while the `ColorFrame` observer updates its background color. Both implement the `Observer` interface and define their own behavior for `sendNotify`.

```
class ColorList(Observer):
    def __init__(self, master=None):
        frame = Toplevel(master)
        frame.geometry("100x100")
        frame.title("Color list")
        self.list = Listbox(frame)
        self.list.pack()

    def sendNotify(self, color: str):
        self.list.insert(END, color.capitalize())

class ColorFrame(Observer):
    def __init__(self, master=None):
        self.frame = Toplevel(master)
        self.frame.geometry("100x100")
        self.frame.title("Color")

    def sendNotify(self, color: str):
        self.frame.config(bg=color)
```

Listing 4.30: Concrete Observer Classes: ColorList and ColorFrame

Concrete Subject Class

The `ColorRadio` class implements the `Subject` interface and maintains a list of observers. Its `colrChange` method retrieves the selected color and notifies all registered observers by calling their `sendNotify` methods.

```
class ColorRadio(Subject):
    def __init__(self, root):
        root.geometry("100x100")
        root.title("Subj")
        self.subjects = []
        self.var = tk.IntVar()
        self.colors = ["red", "blue", "green"]

        tk.Radiobutton(root, text='Red', padx=20, variable=self.var, value=0, command=self.colrChange).pack(anchor=W)
        tk.Radiobutton(root, text='Blue', padx=20, variable=self.var, value=1, command=self.colrChange).pack(anchor=W)
        tk.Radiobutton(root, text='Green', padx=20, variable=self.var, value=2, command=self.colrChange).pack(anchor=W)
        self.var.set(None) # No buttons selected

    def colrChange(self):
        cindex = self.var.get()
        color = self.colors[cindex]
        for subj in self.subjects:
            subj.sendNotify(color)

    def registerInterest(self, subj:Subject):
        self.subjects.append(subj)
```

Listing 4.31: Concrete Subject Class: ColorRadio

Below is a class diagram of the Observer Pattern: (here the Subject is ColorRadio and the concreteObserverA and concreteObserverB are ColorList and ColorFrame).

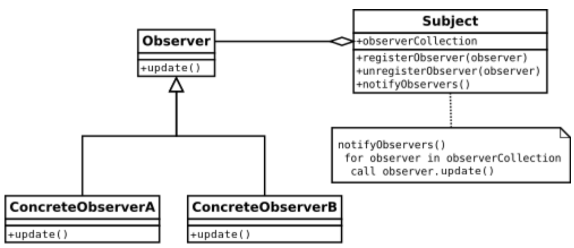

Figure 4.5: Class diagram of the Observer Pattern.

When the application runs, the user is presented with three windows: a window containing radio buttons for selecting a color (`ColorRadio`), a window with a blank background (`ColorFrame`), and a window featuring an empty list box (`ColorList`). Selecting a color updates both observer windows; the background of the `ColorFrame` changes to the selected color, and the `ColorList` window adds the name of the selected color to its list.

Below is the output of Cooper's code after clicking the Red Button.

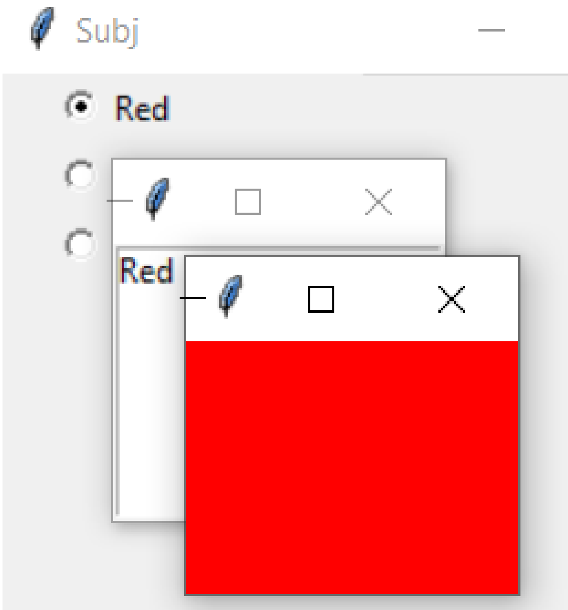

Figure 4.6: Cooper's Graphical Interface for the Observer Pattern.

Cooper's implementation has some drawbacks. First, it includes repetitive boilerplate code, as both the subject and observer classes contain duplicated logic for registering and notifying observers. Second, it suffers from tight coupling, with the subject explicitly managing a list of observers and directly invoking their methods, which makes generalizing the pattern harder. Finally, the implementation has limited extensibility, as adding new observers or subjects requires duplicating code and following the same repetitive structure.

4.4.3.2 Enhanced Implementation of the Observer Pattern

To address these issues, this implementation was enhanced using primarily Python decorators (see section 2.9).

BaseObserver Class

The `BaseObserver` class provides a default `sendNotify` implementation. Instead of requiring each observer to implement `sendNotify`, this method uses a callback function (`update_callback`) to handle notifications. The explanation regarding the use of this callback function is below Listing 4.35.

```
class BaseObserver:
    def __init__(self, update_callback=None):
        self.update_callback = update_callback

    def sendNotify(self, *args, **kwargs):
        if self.update_callback:
            self.update_callback(*args, **kwargs)
```

Listing 4.32: BaseObserver class for default observer behavior

This eliminates repetitive `sendNotify` logic across observers, as each observer can define its unique response to notifications through a callback.

@subject Decorator

The `@subject` decorator adds methods to manage observers, like `registerInterest` for registering and `notifyObservers` for notifying them. This keeps the subject separate from specific observer implementations. This decorator, when applied to a class, makes the class a subject.

```python
def subject(cls):
    cls._observers = [] # Initialize an empty list for observers

    def registerInterest(self, obs, callback=None):
        """Register an observer with an optional callback."""
        if callback:
            obs.update_callback = callback
        self._observers.append(obs)

    def notifyObservers(self, *args, **kwargs):
        """Notify all observers about an update."""
        for obs in self._observers:
            if hasattr(obs, 'update_callback') and obs.update_callback:
                obs.update_callback(*args, **kwargs)
            elif hasattr(obs, 'sendNotify'):
                obs.sendNotify(*args, **kwargs)

    setattr(cls, 'registerInterest', registerInterest)
    setattr(cls, 'notifyObservers', notifyObservers)
    return cls
```

Listing 4.33: @subject decorator for observer management

This eliminates boilerplate in subject classes, as all subjects can use the same notification logic.

@observer Decorator

The `@observer` decorator enforces that any observer either implements `sendNotify` or provides a callback function. This ensures consistency and prevents runtime errors. This decorator, when applied to a class, makes the class an observer.

```python
def observer(cls):
    if not hasattr(cls, 'sendNotify') and not hasattr(cls, 'update_callback'):
        raise TypeError(f"Class {cls.__name__} must implement sendNotify or provide a
            ↪ callback.")
```

```python
    return cls
```

Listing 4.34: @observer decorator for observer validation

@notify_after Decorator

The `notify_after` decorator automates the process of notifying observers after specific subject methods are executed. This reduces manual calls to `notifyObservers`.

```python
def notify_after(method):
    def wrapper(self, *args, **kwargs):
        result = method(self, *args, **kwargs) # Execute the original method
        self.notifyObservers(result) # Notify observers with the result
        return result
    return wrapper
```

Listing 4.35: notify_after decorator for automatic notifications

Rest of the implementation of the Observer Pattern

```python
@observer
class ColorList(BaseObserver):
    def __init__(self, master=None):
        frame = Toplevel(master)
        frame.geometry("100x100")
        frame.title("Color list")
        self.list = Listbox(frame)
        self.list.pack()
        super().__init__(self.update_list)

    def update_list(self, color: str):
        """Update the list box with the selected color."""
        self.list.insert(END, color.capitalize())

@observer
class ColorFrame(BaseObserver):
    def __init__(self, master=None):
        self.frame = Toplevel(master)
        self.frame.geometry("100x100")
        self.frame.title("Color")
        super().__init__(self.update_frame)

    def update_frame(self, color: str):
        """Update the frame's background color."""
        self.frame.config(bg=color)

@subject
class ColorRadio:
    def __init__(self, root):
```

```
        root.geometry("100x100")
        root.title("Subj")
        self.var = tk.IntVar() # Variable to track the selected radio button
        self.colors = ["red", "blue", "green"]

        # Create radio buttons for color selection
        tk.Radiobutton(root, text='Red', padx=20, variable=self.var, value=0,
                       command=self.colrChange).pack(anchor=W)
        tk.Radiobutton(root, text='Blue', padx=20, variable=self.var, value=1,
                       command=self.colrChange).pack(anchor=W)
        tk.Radiobutton(root, text='Green', padx=20, variable=self.var, value=2,
                       command=self.colrChange).pack(anchor=W)
        self.var.set(None) # No buttons selected initially

    @notify_after
    def colrChange(self):
        """Handle color selection and notify observers."""
        cindex = self.var.get() # Get the selected color's index
        return self.colors[cindex] # Return the selected color
```

Listing 4.36: Rest of the implementation of the Observer Pattern

A **callback function** is a function passed as an argument to another function or object, which is executed when a specific event occurs. The `BaseObserver` class provides a default implementation of the `sendNotify` method, which calls a callback function, referred to as `update_callback`.

Both `ColorList` and `ColorFrame` inherit from `BaseObserver` and provide their own custom callbacks during initialization. For `ColorList`, the callback is the `update_list` method, which appends the selected color name to a list box. For `ColorFrame`, the callback is the `update_frame` method, which changes the background color of the frame. These callbacks are passed to the `BaseObserver` class through its constructor and are stored as the `update_callback` attribute.

When the subject, `ColorRadio`, triggers a notification using the `notifyObservers` method, it checks if the observer has a callback function (stored in `update_callback`). If a callback exists, it is executed with the appropriate arguments (in this case, the selected color).

The flow of execution is as follows: when a user selects a color in the `ColorRadio` window, the `colrChange` method returns the selected color and triggers the `notifyObservers` method (because of `@notify_after` decorator). The `notifyObservers` method iterates over all registered observers and calls their respective `update_callback` functions. For `ColorList`, the callback function `update_list` is executed, which adds the color name to the list. For `ColorFrame`, the callback function `update_frame` is executed, which updates the background color of the frame.

This way different observers can provide different callbacks without needing to change the notification mechanism.

The following table summarizes the key differences between Cooper's original Observer pattern implementation and the enhanced version using Python's advanced mechanisms:

Feature	**Cooper's Implementation**	**Enhanced Implementation**
Observer Registration	Explicit in `ColorRadio`	Automated via `@subject`
Notification Logic	Manual `for` loop iterating over observers	Automated via `@notify_after` decorator
Observer Definition	Observers must implement `sendNotify()`	Uses callback functions for flexibility
Extensibility	Requires modifying subclasses to add new observers	Observers can dynamically plug/unplug without modifying the subject

Table 4.3: Comparison Between Cooper's and Enhanced Observer Pattern Implementations

5

Discussion of Results

This chapter analyzes how classical design patterns were enhanced using Python's advanced mechanisms. It introduces subgroups based on the type of enhancement, such as decorators, metaclasses, and built-in support, and reflects on the broader implications of these enhancements for pattern usage in Python.

5.1 Design Patterns Enhancement Conclusion

As mentioned in Section 4.2, the patterns divided themselves based on whether mechanisms could enhance their classical implementation. The enhancement of one pattern can be very different from the enhancement of another and so, naturally, within the *group* of enhanced patterns, narrower subgroups emerged. Some enhancements relied on the same mechanisms as other enhancements, while others primarily simplified the pattern to improve readability. In some cases, the enhancements were so significant that they nearly amount to direct language support.

These *narrower* groups are:

- **Clarity and Readability:** These patterns were enhanced by reducing boilerplate code, making them more readable while preserving their original structure. Python's method chaining, exception handling, and higher-order functions simplified their implementation.

  This group differs from the others because *every* pattern had its clarity and readability improved, as discussed in Section 5.2. However, for these patterns, clarity and readability were the *primary* focus of enhancement, whereas for others, improvements were based on different mechanisms. These include:

  Builder, Chain of Responsibility.

- **Decorators:** Python's decorator mechanism allowed these patterns to be enhanced by dynamically modifying behavior without requiring additional classes or deep inheritance structures. The ability to wrap functions or properties at runtime eliminated much of the complexity found in classical implementations. These include:

Observer, Command.

- **Metaclasses and Reflection:** Patterns that involve dynamic object creation used Python's metaclass and reflection capabilities. The use of `getattr()` and metaclasses eliminated the need for rigid factory logic, making these patterns more flexible and adaptable. These include:

  Factory Method, Abstract Factory.

- **Dynamic Behavior:** These patterns used Python's dynamic method assignment, first-class functions(*functions that can be assigned to variables, passed as arguments, and returned from other functions, just like any other object.*) and dynamic typing. By removing the need for predefined structures, these patterns are more modular. These include:

  Strategy, State, Visitor, Proxy.

- **Nearly Built-in:** These patterns were enhanced to the point where they closely resemble built-in Python features. Python's native mechanisms almost make the explicit design pattern structures and implementations redudant but some implementation is still needed. These include:

  Flyweight, Composite.

Flyweight Pattern

The *Flyweight* design pattern is a structural pattern that optimizes memory usage by sharing a common state among multiple objects. In classical implementations, objects are stored and shared manually through a centralized factory. In Python, `__new__` and object interning already provide this behavior for immutable objects like strings and integers, making an explicit Flyweight implementation often unnecessary.

Python **automatically interns strings, integers, and some immutable objects**, meaning no factory is needed for basic cases:

```python
a = "data"
b = "data"
assert a is b # Interned automatically
```

Listing 5.1: Automatic Interning of Immutable Objects

Additionally, `__new__` enables object caching natively, eliminating the need for a separate Flyweight factory.

Composite Pattern

The *Composite* pattern allows treating *individual objects and collections of objects uniformly*, typically through a common interface. In traditional OOP, explicit `Component`, `Leaf`, and `Composite` classes enforce this structure. In Python, **lists and dictionaries already support hierarchical structures**, and it is important to note that these structures are also *heterogeneous* (able to mix different types). This flexibility makes a strict Composite implementation almost unnecessary, reducing the need for explicit class hierarchies.

```
tree = {
    "root": ["A", "B", {"subtree": ["C", "D"]}]
}
```

Listing 5.2: Tree-Like Structure Using Python Dictionaries

Table 5.1 below presents a conclusion for every enhancement of design pattern.

Pattern	Enhanced	Short Conclusion	Sub-Group
Iterator	Yes (Built-in)	Section 4.3	Built-in
Decorator	Yes (Built-in)	Section 4.3	Built-in
Prototype	Yes (Built-in)	Section 4.3	Built-in
Memento	Yes (Built-in)	Section 4.3	Built-in
Singleton	Yes (Built-in)	Section 4.3	Built-in
Interpreter	Yes (Built-in)	Section 4.3	Built-in
Facade	Yes (Built-in)	Section 4.3	Built-in
Flyweight	Yes	Improved efficiency through Python's `__new__` method enabling object reuse.	Nearly Built-in
Composite	Yes	Naturally represented using Python's built-in structures (lists, dictionaries).	Nearly Built-in
Factory Method	Yes	Enhanced via dynamic instantiation via metaclasses and reflection (`getattr`).	Metaclasses and Reflection
Abstract Factory	Yes	Enhanced using metaclasses, reflection, and duck typing.	Metaclasses and Reflection
Command	Yes	Enhanced using Abstract Classes, Python decorators, and dynamic typing	Decorators
Observer	Yes	Enhanced using Python decorators (e.g (`@property`))	Decorators
Proxy	Yes	Enhanced using Python's dynamic attributes (`__getattr__`, `__setattr__`)	Dynamic Behavior
Strategy	Yes	Enhanced using first-class functions for dynamic strategy selection without separate classes.	Dynamic Behavior
State	Yes	Enhanced using dynamic method assignment and function mapping to simplify state management and transitions.	Dynamic Behavior
Visitor	Yes	Utilized dynamic dispatch (`functools.singledispatch`).	Dynamic Behavior
Chain of Responsibility	Yes	Simplified via a dictionary-based event dispatcher.	Clarity and Readability
Builder	Yes	Simplified via method chaining, dynamic typing, and duck typing.	Clarity and Readability
Adapter	Minimal	Slightly simplified through duck typing; explicit adapter structure remains essential.	Minimal Enhancement
Bridge	Minimal	Python simplifies abstraction but explicit separation between logic and implementation persists.	Minimal Enhancement
Template Method	Minimal	Easily handled with method overriding; explicit subclassing still required.	Minimal Enhancement
Mediator	Minimal	Dynamic typing simplifies component interaction; explicit mediator structure remains necessary.	Minimal Enhancement

Table 5.1: Summary of Python's Enhancement of Design Patterns with Group Classification

5.2 Clarity and readability

In all 23 GOF patterns, regardless of the scenarios or implementations by Cooper [11], Ayeya [3], every classical implementation was made clearer and easier to read. Python's advanced mechanisms helped simplify the code and improve its structure without changing the pattern's behavior. For example, the `@dataclass` decorator, showcased in the enhancement of Cooper's *Command Pattern* in Section 4.4.2, Listing 4.26, automatically creates common methods like `__init__`. This reduces repetitive code and keeps the focus on the logic of the pattern. Below is an example of the difference initializing a class using @dataclass side by side:

```
class Human:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Listing 5.3: Normal Class initialization

```
@dataclass
class Human:
    name: str
    age: int
```

Listing 5.4: Initialization with @dataclass

Python's dynamic features, such as dictionary-based method dispatching can also help the readability. For instance, in the enhancement of Ayeva & Kasampalis's *Chain of Responsibility Pattern*, instead of using several *if-elif* statements, the improved approach directly maps events to handlers using a predefined dictionary.

```
class Widget:
    def handle(self, event):
        handler = f'handle_{event}'
        if hasattr(self, handler):
            method = getattr(self, handler)
            method(event)
        elif self.parent is not None:
            self.parent.handle(event)
        elif hasattr(self, 'handle_default'):
            self.handle_default(event)
```

Listing 5.5: Ayeva & Kasampalis' original event handling

```
class Widget:
    def handle(self, event):
        handlers = {
            "close": self.handle_close,
            "paint": self.handle_paint,
            "down": self.handle_down
        }
        method = handlers.get(event, self.handle_default)
        method(event) if callable(method) else self.parent.handle(event)
```

Listing 5.6: Improved event handling with a dictionary

As mentioned in Section 3, the notion of enhancing or improving code can be nuanced at times. Listings 5.5 and 5.6 demonstrate that both approaches are valid and correct, with the choice depending on the programmer's preference. Some may be more accustomed to a particular syntax or style of coding. In a study conducted by Todd Sedano [41], 21 master's students were presented with various code examples and asked what changes would lead to a better readability. The study found that simplifying if conditions and extracting methods to reduce complexity were commonly associated with improved readability. Python's syntax is known to be simple and readable and Python's mechanisms further enhance readability by replacing verbose constructs with simpler alternatives.

In a previous study conducted by Zhang [44], it is demonstrated that Python's idiomatic constructs significantly enhance the readability and simplicity of code. It shows that refactoring non-idiomatic Python code (*code that does not follow the conventions, style, or "best practices" commonly accepted by the Python community*) not only reduces complexity but also aligns the implementation closer to the language's philosophy of clarity and elegance.

This study aligns with the findings that Python mechanisms simplify classical implementations discussed in the referenced books.

5.3 Modularity and Reusability

Overall, the mechanisms discussed in this thesis improve modularity *(organizing code so that each part handles a specific task and can work independently from the rest)* by making it easier to keep related behavior in the same place. This reduces unnecessary reliance between parts of the system and makes the structure easier to follow.

One example of this is the use of decorators. They make it possible to add new behavior to functions without changing their original code. In the *Observer* pattern, decorators were used to register observers directly at the point of definition. This kept the registration logic close to the relevant function and avoided spreading it elsewhere.

The same approach can be seen with metaclasses and reflection, which allow behavior to be added at runtime without requiring one part of the program to refer directly to another. In the *Abstract Factory* pattern, metaclasses automatically registered classes, while reflection was used to look them up when needed.

Alongside modularity, these mechanisms also contribute to *reusability*. By focusing on behavior rather than rigid structure, components can be applied in different contexts without modification. For instance, in the *Decorator* pattern, extra functionality was added to objects without modifying their core behavior. Each decorator wrapped the original object and introduced new logic, making it easy to reuse the same base while adjusting the behavior for different situations.

High cohesion and low coupling [21] are key goals in software design, and these features help in that direction. Taken together, these features support high cohesion by grouping related behavior and low coupling by allowing different parts of the program to work together without strong dependencies. Python's dynamic mechanisms make it possible to build software that remains organized even as it grows.

5.4 Visitor Pattern & Third-Party Libraries

The *Visitor* pattern enables adding new operations to existing object structures without modifying their classes. A key requirement for an efficient implementation of this pattern is *double dispatch*, which allows method calls to be dynamically resolved based on the runtime types of two objects, typically, the element being visited and the visitor.

Multiple dispatch generalizes this concept by enabling method selection based on the runtime types of all arguments, not just one or two. In the context of the Visitor pattern, multiple dispatch simplifies the expression of complex visitations involving multiple element types and visitors almost making the implementation of the pattern redudant.

Python, unlike some other languages such as CLOS [14] or Julia [5], does not provide built-in multiple dispatch. Instead, it relies on *single dispatch*, which allows dynamic method resolution based only on the first argument's type. While Python's `functools.singledispatch` provides a partial solution, it does not natively support full multiple dispatch. Achieving multiple dispatch in Python requires third-party libraries, such as `multiple dispatch`, which extend the language's dynamic method resolution capabilities. Using this library would make the implementation of the Visitor pattern feel almost natively supported in Python.

However, since this thesis focuses on enhancements using Python's native mechanisms, external libraries are not explored. Thus, while Python's existing dynamic features, such as `functools.singledispatch`, improve Visitor implementations, the lack of native multiple dispatch prevents the pattern from being integrated into the language.

6

Related Work

This section reviews studies that focus on evaluating programming languages through design pattern implementations and exploring concepts related to modularity and composition.

6.1 Language assessment based on Design Patterns

Jan Hannemann and Gregor Kiczales [19] studied the implementation of the Gang of Four (GoF) design patterns in Java and AspectJ. Their research showed that aspect-oriented programming (AOP), through AspectJ, improves modularity in 17 out of the 23 design patterns analyzed. They evaluated this improvement based on criteria such as code locality, reusability, composability, and (un)pluggability. The authors noted that *"AspectJ implementations of 17 of the 23 GoF patterns were localized, with 12 of these enabling a core part of the implementation to be abstracted into reusable code."* This indicates a clear benefit of AspectJ in organizing pattern implementations. The study introduces a predictive model that distinguishes design patterns based on the nature of roles, defining roles versus superimposed roles. This distinction helps to understand how different roles affect modularity in AspectJ implementations. Defining roles have specific responsibilities, while superimposed roles allow for flexibility and reuse. Moreover, the localized implementations in AspectJ make it easier to understand and maintain the code. A single modular unit for a pattern enhances code clarity and acts as documentation for other developers.

Miguel Monteiro and João Gomes [31] discuss how the Object Teams (OT/J) approach enhances the implementation of design patterns by introducing explicit support for collaboration and role-based modularization. Their study compares OT/J to previous implementations by Jan Hannemann and Gregor Kiczales, highlighting improvements in modularity and maintainability. They concluded that OT/J provides superior support for flexible module extensibility, fine-grained composition at the instance level, and the ability to encapsulate multiple pattern participants within a unified and cohesive module, ultimately enabling cleaner separation of concerns and more reusable design abstractions.

Hridesh Rajan [39] discusses how the Eos programming language, a unified aspect-oriented programming model, simplifies the implementation of design patterns compared to traditional approaches. He notes that *"The simplification in these cases is the result of including instantiation and instance-level advising as language features, unifying aspect and class, and unifying method and advice"*. His study concluded that out of 23 design patterns, 7 showed improvement while the remaining 16 stayed intact with no degradation.

Frank Schmager, Nicholas Cameron and James Noble [40] try to evaluate the Go programming language through the implementation of design patterns. The study discusses Go's rejection of traditional inheritance in favor of embedding, which is a unique feature distinguishing Go from other object-oriented languages. They also noted that Go's interface system is notably different because it uses a structural typing system, which means that type compatibility and implementation are determined implicitly by the methods a type provides.

Diogo Escaleira [1] assessed the object-oriented capabilities of Octave by implementing and analyzing all 23 Gang of Four (GoF) design patterns. The study evaluated how well Octave supports modularity and flexible module composition, comparing the resulting implementations against their Java counterparts. Due to Octave's limitation such as incomplete classdef support and lack of interfaces, the author identified recurring obstacles in encapsulation, abstraction, and polymorphism. Nevertheless, the research demonstrates that it is possible to emulate most OO design patterns in Octave using creative workarounds, such as error-throwing base classes to simulate interfaces and adapter-like wrappers to manage type polymorphism in arrays.

Kristian Pedersen [34] investigated how the choice of programming language affects the implementation of Gang of Four (GoF) design patterns. The study focused on four patterns Composite, Prototype, Adapter, and Decorator implemented across five object-oriented languages: Python, JavaScript, C#, Go, and Smalltalk. Pedersen evaluated each implementation using a comprehensive set of metrics, including NCLOC, cyclomatic complexity, class cohesion, inheritance depth, and shallow references. The results showed that languages with flexible typing, minimal boilerplate, and attribute visibility control generally yielded more concise and maintainable implementations. While no single language emerged as universally superior, Python and JavaScript demonstrated strong performance due to their dynamic nature and reduced verbosity.

Gerald Baumgartner, Konstantin Läufer, and Vincent Russo [4] examined how object-oriented design patterns both reveal and depend upon the capabilities of the underlying programming language. Their study identifies that many patterns compensate for missing language constructs, such as first-class interfaces, multimethod dispatch, and closures. They argue that incorporating these constructs into statically typed object-oriented languages can significantly simplify the implementation of patterns and increase modularity. The authors classify certain patterns like Singleton, Iterator, and Visitor as paradigmatic idioms, workarounds necessitated by language limitations rather than inherent design solutions. They propose a language design combining features such as interface-conformance typing, lexically scoped closures, and classless singleton objects, showing how these mechanisms would reduce boilerplate and make pattern implementations more natural, reusable, and accessible across language communities.

6.2 Improvements Through Language Mechanisms

Craig Chambers [8] discusses Cecil, an object-oriented language emphasizing multimethod support. It introduces multimethods, which dispatch based on runtime types of multiple arguments, unlike single dispatch. The paper also covers Cecil's handling of encapsulation, multiple inheritance, and multiple dispatching.

Miguel Monteiro and João Fernandes [30] highlight the benefits of aspect-oriented programming and describe aspect-oriented refactoring, which transforms Java systems by isolating crosscutting concerns into separate modules. They survey tools like AOP-Migrator and SoothSayer, and present case studies demonstrating the practical application of AOP in refactoring complex Java systems.

Robert Hirschfeld [20] discusses AspectS, an aspect-oriented programming extension for the Squeak/Smalltalk environment. It introduces how AspectS is designed to support the modularization of crosscutting concerns through dynamic weaving at runtime. Unlike static AOP systems like AspectJ, AspectS integrates aspects dynamically, on-demand, utilizing Smalltalk's metaobject protocol (MOP)

6.3 Lightweight systematic search on Google scholar

A lightweight systematic search was conducted on Google Scholar to identify relevant studies in Python. The following criteria were used to filter the search results:

- **Time Interval** - The search query starts in 2001. Although the first version of Python was released in 1991, Python 2.2 was released in 2001, which introduced significant enhancements to its object-oriented programming model.

- **Additional settings.** - The settings "include patents" and "include citations" were both deselected.

- **Browser Pages considered** - The search query was limited to the first 3 pages of results.

- **Research Topic** - The topic covered had to be either from the community of Python developers, users or researchers.

Initially, many beginner-oriented Python books were identified, which were deemed out of scope. Subsequent queries focused on design patterns and advanced mechanisms such as mixins and Python decorators.

Each result was assessed by reading the abstract to determine relevance to the research topic. Books like [3], [11], [18], and [36] were found when querying design patterns, indicating that our queries were on the right path as these books were previously recommended by my advisor.

Queries including mixins, decorators, and related mechanisms led to the discovery of books like [9] and [2], which provided in-depth insights into these concepts. Their content was reviewed to evaluate their relevance to the thesis.

The section described here was not intended to serve as a comprehensive survey of related work but rather as a complement, yet it reinforced our conviction that there are very few studies exploring the advanced mechanisms that Python supports and how to enhance design patterns with them.

7 Conclusion & Future Work

7.1 Summing up

This thesis explored how Python's advanced mechanisms impact the implementation of design patterns. By using mechanisms such as metaclasses, decorators, dynamic typing, and reflection, many classical design pattern implementations were enhanced. The results showed that Python's native features eliminate the need for some patterns altogether, while others were enhanced in ways that reduced complexity, more adaptable, readability or modularity.

Out of the 23 design patterns analyzed, 19 were improved through Python's mechanisms. Seven patterns, including *Iterator, Prototype, Memento, Singleton, Decorator, Interpreter, and Facade*, were found to be directly supported by the language, making explicit implementations unnecessary. The remaining 12 patterns benefited from Python's dynamic capabilities, allowing for more concise and flexible implementations. Only four patterns *Bridge, Adapter, Template Method, and Mediator* remained mostly unchanged, as their structure and intent did not significantly benefit from Python's advanced features.

This raises an important question: *Are design patterns necessary in Python?* Unlike languages such as Java or C++, where design patterns often compensate for missing features, Python provides built-in mechanisms that naturally support many of these patterns. For example, decorators eliminate the need for explicit *Decorator* pattern implementations, while metaclasses can simplify factories.

However, while Python reduces the need for boilerplate implementations, design patterns remain relevant. They provide a structured approach to solving common problems and serve as a shared vocabulary among developers. Even when a pattern's explicit implementation is not required, recognizing its underlying principles helps in making informed design decisions. Patterns also improve code readability and maintainability by establishing familiar structures, reducing the learning curve for new contributors in larger projects.

In this sense, Python does not eliminate the need for design patterns but instead reshapes how they are applied.

Why isn't Python used more in large-scale codebases? This thesis focused on modularity and composition but did not analyze runtime performance, which is one of Python's weakest points [26]. Also, while Python's flexibility enhances development speed, dynamic typing can introduce runtime errors that would be caught at compile time in statically typed languages. This can make maintenance and scaling more difficult, which is one reason why large-scale projects often favor statically typed languages.

Ultimately, this study demonstrates that Python's advanced mechanisms can improve design patterns and the core concepts behind these patterns remain useful. Understanding these patterns helps developers make better use of Python's features, ensuring that code remains maintainable, flexible, and modular.

7.2 Future Work

There are several possible directions for future work that can extend the scope of this thesis.

One option is to explore third-party libraries that provide additional mechanisms not available in the standard library. For example, the `multipledispatch` (see Section 5.4). These libraries may offer different trade-offs in terms of readability, flexibility, or integration with existing code.

Another option is to focus on one of the mechanisms mentioned in this thesis and explore it in more depth. For instance, mixins were described in Section 2.8 but not applied extensively in the implementation of design patterns. A dedicated analysis could help clarify when mixins are useful and how they interact with other composition techniques.

A study between Python and other dynamic languages could also be done to contrast their mechanisms or to assess if they have some in common, contributing to the knowledge about how these languages support modularity and composition.

Finally, since aspect-oriented programming (AOP) was not explored in depth in this work, a possible future direction would be to investigate how Python, compares to AOP languages (e.g AspectJ) in modularity and design pattern support.

Bibliography

[1] D. de Almeida Escaleira. "Assessment of Octave's OO Features Based on GOF Patterns". MA thesis. Universidade NOVA de Lisboa (Portugal), 2023 (cit. on p. 68).

[2] M. Anaya. *Clean Code in Python: Develop maintainable and efficient code*. Packt Publishing Ltd, 2021 (cit. on pp. 17, 70).

[3] K. Ayeva and S. Kasampalis. *Mastering Python Design Patterns: A Guide to Creating Smart, Efficient, and Reusable Software*. 2nd ed. Packt Publishing, 2018. ISBN: 9781788837484 (cit. on pp. 1, 2, 25, 32, 33, 64, 69).

[4] G. Baumgartner, K. Läufer, and V. F. Russo. "On the interaction of object-oriented design patterns and programming languages". In: (1996) (cit. on p. 68).

[5] J. Bezanson et al. "Julia: A fresh approach to numerical computing". In: *SIAM review* 59.1 (2017), pp. 65–98. URL: https://doi.org/10.1137/141000671 (cit. on p. 66).

[6] G. Booch. "Object-oriented development". In: *IEEE transactions on Software Engineering* 2 (1986), pp. 211–221 (cit. on p. 1).

[7] G. Bracha and W. Cook. "Mixin-based inheritance". In: *Proceedings of the European Conference on Object-Oriented Programming on Object-Oriented Programming Systems, Languages, and Applications*. ACM Press. Ottawa, Canada, 1990, pp. 303–311 (cit. on p. 16).

[8] C. Chambers. "Object-oriented multi-methods in Cecil". In: *European Conference on Object-Oriented Programming*. Springer. 1992, pp. 33–56 (cit. on p. 69).

[9] W. Chun. *Core python programming*. Vol. 1. Prentice Hall Professional, 2001 (cit. on pp. 4, 70).

[10] W. contributors. *Separation of concerns — Wikipedia, The Free Encyclopedia*. 2024. URL: https://en.wikipedia.org/wiki/Separation_of_concerns (cit. on p. 3).

[11] J. W. Cooper. *Python Programming with Design Patterns*. Pearson, 2021 (cit. on pp. 1, 2, 25, 34, 43, 64, 69).

[12] J. W. Cooper. *The Design Patterns Java Companion*. Addison-Wesley Design Patterns Series. Addison-Wesley, 1998 (cit. on p. 1).

[13] J. W. Cooper. "Java design patterns: a tutorial". In: (2000) (cit. on p. 27).

[14] L. G. DeMichiel and R. P. Gabriel. "The common lisp object system: An overview". In: *European Conference on Object-Oriented Programming*. Springer. 1987, pp. 151–170 (cit. on p. 66).

[15] J. Forcier, P. Bissex, and W. J. Chun. *Python web development with Django*. Addison-Wesley Professional, 2008 (cit. on p. 22).

[16] D. S. Foundation. *Django: The web framework for perfectionists with deadlines*. Website. 2024. URL: https://www.djangoproject.com/ (cit. on p. 16).

[17] E. Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994. ISBN: 0201633612 (cit. on pp. 2, 28).

[18] C. Giridhar. *Learning Python Design Patterns*. 2nd ed. Packt Publishing, 2016. ISBN: 9781785888038 (cit. on pp. 1, 2, 25, 69).

[19] J. Hannemann and G. Kiczales. "Design pattern implementation in Java and AspectJ". In: *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2002, pp. 161–173 (cit. on pp. 1, 24, 67).

[20] R. Hirschfeld. "Aspects-aspect-oriented programming with squeak". In: *Net. ObjectDays: International Conference on Object-Oriented and Internet-Based Technologies, Concepts, and Applications for a Networked World*. Springer. 2002, pp. 216–232 (cit. on p. 69).

[21] M. Hitz and B. Montazeri. *Measuring coupling and cohesion in object-oriented systems*. na, 1995 (cit. on p. 65).

[22] R. E. Johnson and B. Foote. "Designing reusable classes". In: *Journal of object-oriented programming* 1.2 (1988), pp. 22–35 (cit. on p. 20).

[23] B. W. Kernighan and D. M. Ritchie. "The C programming language". In: (2002) (cit. on p. 4).

[24] G. Kiczales, J. Des Rivieres, and D. G. Bobrow. *The art of the metaobject protocol*. MIT press, 1991 (cit. on pp. 14, 15).

[25] G. Kiczales et al. "Aspect-oriented programming". In: *ECOOP'97—Object-Oriented Programming: 11th European Conference Jyväskylä, Finland, June 9–13, 1997 Proceedings 11*. Springer. 1997, pp. 220–242 (cit. on p. 23).

[26] K. Lei, Y. Ma, and Z. Tan. "Performance comparison and evaluation of web development technologies in php, python, and node. js". In: *2014 IEEE 17th international conference on computational science and engineering*. IEEE. 2014, pp. 661–668 (cit. on p. 72).

[27] P. Maes. "Concepts and experiments in computational reflection". In: *ACM Sigplan Notices* 22.12 (1987), pp. 147–155 (cit. on p. 22).

[28] I. C. Mărieș. *Aspectlib: Aspect-oriented programming, monkey-patch and decorators library*. Version 1.4.2. Accessed: 2024-07-02. 2016. URL: https://python-aspectlib.readthedocs.io/en/latest/ (cit. on p. 23).

[29] W. McKinney et al. "pandas: a foundational Python library for data analysis and statistics". In: *Python for high performance and scientific computing* 14.9 (2011), pp. 1–9 (cit. on p. 22).

[30] M. P. Monteiro and J. M. Fernandes. "Aspect-oriented Refactoring of Java Programs". In: *Java in Academia and Research* (2011), pp. 141–158 (cit. on p. 69).

[31] M. P. Monteiro and J. Gomes. "Implementing design patterns in Object Teams". In: *Software: Practice and Experience* 43.12 (2013), pp. 1519–1551 (cit. on pp. 2, 24, 67).

[32] T. E. Oliphant et al. *Guide to numpy*. Vol. 1. Trelgol Publishing USA, 2006 (cit. on p. 22).

[33] R. Olsen. *Design patterns in Ruby*. Pearson Education, 2007 (cit. on p. 1).

[34] K. B. Pedersen. "How Implementation Language Affects Design Patterns: A Comparison of Gang of Four Design Pattern Implementations in Different Languages". MA thesis. 2019 (cit. on p. 68).

[35] G. Pereira. *Python-Patterns*. https://github.com/Guivape/Python-Patterns. 2025 (cit. on p. 2).

[36] D. Phillips. *Python 3 Object-Oriented Programming: Build robust and maintainable software with object-oriented design patterns in Python 3.8*. 3rd. Packt Publishing, 2018. ISBN: 9781789615852 (cit. on pp. 1, 2, 25, 69).

[37] Python Software Foundation. *Python Language Reference, version 3.12*. Accessed: 2024-07-02. 2024. URL: https://www.python.org/ (cit. on p. 4).

[38] Python Software Foundation. *Tkinter — Python interface to Tcl/Tk*. 2024 (cit. on p. 35).

[39] H. Rajan. "Design pattern implementations in Eos". In: *Proceedings of the 14th Conference on Pattern Languages of Programs*. 2007, pp. 1–11 (cit. on pp. 24, 67).

[40] F. Schmager, N. Cameron, and J. Noble. "Gohotdraw: Evaluating the go programming language with design patterns". In: *Evaluation and usability of programming languages and tools*. 2010, pp. 1–6 (cit. on pp. 1, 68).

[41] T. Sedano. "Code readability testing, an empirical study". In: *2016 IEEE 29th International conference on software engineering education and training (CSEET)*. IEEE. 2016, pp. 111–117 (cit. on p. 65).

[42] K. D. Smith et al. *PEP 318 – Decorators for Functions and Methods*. https://peps.python.org/pep-0318/. 2004 (cit. on p. 17).

[43] TIOBE. *TIOBE Index for July 2024: Top 10 Most Popular Programming Languages*. 2024. URL: https://www.tiobe.com/tiobe-index/ (cit. on p. 2).

[44] Z. Zhang et al. *Making Python Code Idiomatic by Automatic Refactoring Non-Idiomatic Python Code with Pythonic Idioms, ESEC/FSE, 2022*. 2022 (cit. on p. 65).