

Second Edition

Mastering Python Design Patterns

A guide to creating smart, efficient
and reusable software

Kamon Ayeva
Sakis Kasampalis



BIRMINGHAM - MUMBAI

Second Edition

Mastering Python Design Patterns

Copyright © 2018 Packt Publishing

First published: January 2015

Second edition: August 2018

Production reference: 1300818

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78883-748-4

www.packtpub.com

Contents

Preface	1
Chapter 1: The Factory Pattern	7
The factory method	8
Real-world examples	9
Use cases	9
Implementing the factory method	10
The abstract factory	18
Real-world examples	18
Use cases	19
Implementing the abstract factory pattern	19
Summary	24
Chapter 2: The Builder Pattern	25
Real-world examples	26
Use cases	27
Implementation	31
Summary	37
Chapter 3: Other Creational Patterns	38
The prototype pattern	39
Real-world examples	39
Use cases	40
Implementation	40
Singleton	44
Real-world examples	44
Use cases	45
Implementation	45
Summary	49
Chapter 4: The Adapter Pattern	51
Real-world examples	51
Use cases	52
Implementation	52
Summary	55
Chapter 5: The Decorator Pattern	56
Real-world examples	57
Use cases	57
Implementation	58
Summary	63

Chapter 6: The Bridge Pattern	64
Real-world examples	64
Use cases	65
Implementation	65
Summary	68
Chapter 7: The Facade Pattern	70
Real-world examples	71
Use cases	71
Implementation	72
Summary	76
Chapter 8: Other Structural Patterns	77
The flyweight pattern	78
Real-world examples	79
Use cases	79
Implementation	80
The model-view-controller pattern	85
Real-world examples	86
Use cases	87
Implementation	88
The proxy pattern	92
Real-world examples	95
Use cases	95
Implementation	96
Summary	100
Chapter 9: The Chain of Responsibility Pattern	101
Real-world examples	102
Use cases	104
Implementation	105
Summary	110
Chapter 10: The Command Pattern	111
Real-world examples	112
Use cases	112
Implementation	113
Summary	121
Chapter 11: The Observer Pattern	122
Real-world examples	122
Use cases	123
Implementation	124
Summary	130
Chapter 12: The State Pattern	131

Real-world examples	132
Use cases	133
Implementation	133
Summary	140
Chapter 13: Other Behavioral Patterns	141
Interpreter pattern	142
Real-world examples	143
Use cases	143
Implementation	143
Strategy pattern	150
Real-world examples	151
Use cases	152
Implementation	152
Memento pattern	157
Real-world examples	157
Use cases	158
Implementation	158
Iterator pattern	161
Real-world examples	162
Use cases	162
Implementation	163
Template pattern	166
Real-world examples	166
Use cases	167
Implementation	168
Summary	170
Chapter 14: The Observer Pattern in Reactive Programming	173
Real-world examples	174
Use cases	174
Implementation	175
A first example	175
A second example	178
A third example	179
A fourth example	182
Summary	188
Chapter 15: Microservices and Patterns for the Cloud	189
The Microservices pattern	190
Real-world examples	190
Use cases	191
Implementation	191
A first example	193
A second example	195
The Retry pattern	198

Real-world examples	198
Use cases	198
Implementation	199
A first example	199
A second example, using a third-party module	201
A third example, using another third-party module	203
The Circuit Breaker pattern	205
Real-world examples	205
Use cases	205
Implementation	206
The Cache-Aside pattern	208
Real-world examples	209
Use cases	209
Implementation	209
Throttling	216
Real-world examples	217
Use cases	217
Implementation	218
Summary	221

Index	226
--------------	-----

Preface

Python is an object-oriented scripting language that is used in a wide range of categories. In software engineering, a design pattern is a solution for solving software design problems. Although they have been around for a while, design patterns remain one of the hot topics in software engineering and are a good source of information for software developers to solve the problems they face on a regular basis.

This book takes you through a variety of design patterns and explains them with real-world examples. You will get to grips with low-level details and concepts that show you how to write Python code, without focusing on common solutions as enabled in Java and C++. You'll also hunt sections on corrections, best practices, system architecture, and its designing aspects.

This book will help you learn the core concepts of design patterns and the way they can be used to resolve software design problems. You'll focus on all the Gang of Four (GoF) design patterns, which are used to solve everyday problems and take your skills to the next level with reactive and functional patterns that help you build resilient, scalable, and robust applications. By the end of the book, you'll be able to efficiently address commonly-faced problems and develop applications, and also be comfortable working on scalable and maintainable projects of any size.

Who this book is for

This book is for intermediate Python developers. Prior knowledge of design patterns is not required to enjoy this book.

What this book covers

Chapter 1, *The Factory Pattern*, will teach you how to use the Factory design pattern (Factory Method and Abstract Factory) to initialize objects, and also covers the benefits of using the Factory design pattern instead of direct object instantiation.

Chapter 2, *The Builder Pattern*, will teach you how to simplify the object creation process for cases typically composed of several related objects.

Chapter 3, *Other Creational Patterns*, will teach you how to handle other object creation situations with techniques such as creating a new object that is a full copy (hence, named clone) of an existing object, a technique offered by the Prototype pattern. You will also learn about the Singleton pattern.

Chapter 4, *The Adapter Pattern*, will teach you how to make your existing code compatible with a foreign interface (for example, an external library) with minimal changes.

Chapter 5, *The Decorator Pattern*, will teach you how to enhance the functionality of an object without using inheritance.

Chapter 6, *The Bridge Pattern*, will teach you how to externalize an object's implementation details from its class hierarchy to another object class hierarchy. This chapter encourages the idea of preferring composition over inheritance.

Chapter 7, *The Facade Pattern*, will teach you how to create a single entry point to hide the complexity of a system.

Chapter 8, *Other Structural Patterns*, will teach you the Flyweight, Model-View-Controller and Proxy patterns. With the Flyweight pattern, you will learn to reuse objects from an object pool to improve the memory usage and possibly the performance of your applications. The Model-View-Controller (MVC) pattern is used in application development (desktop, web) to improve maintainability by avoiding mixing the business logic with the user interface. And with the Proxy pattern, you provide a special object that acts as a surrogate or placeholder for another object to control access to it and reduce complexity and/or improve performance.

Chapter 9, *The Chain of Responsibility Pattern*, will teach another technique to improve the maintainability of your applications by avoiding mixing the business logic with the user interface.

Chapter 10, *The Command Pattern*, will teach you how to encapsulate operations (such as undo, copy, paste) as objects, to improve your application. Among the advantages of this technique, the object that invokes the command is decoupled from the object that performs it.

Chapter 11, *The Observer Pattern*, will teach you how to send a request to multiple receivers.

Chapter 12, *The State Pattern*, will teach you how to create a state machine to model a problem and the benefits of this technique.

Chapter 13, *Other Behavioral Patterns*, will teach you, among several other advanced programming techniques, how to create a simple language on top of Python, which can be used by domain experts without forcing them to learn how to program in Python.

Chapter 14, *The Observer Pattern in Reactive Programming*, will teach you how to send a notification to the registered stakeholders of a stream of data (and events), whenever there is a state change.

Chapter 15, *Microservices and Patterns for the Cloud*, will teach you several system design patterns which are important with today's increasing adoption of Cloud-Native applications and microservices architectures. You will learn that you can split your application into functional and/or technical services that can be maintained and deployed more independently, using microservices-oriented frameworks, containers, and other techniques. Since you rely more and more on remote services as part of an application (for example, an API), retry mechanisms are used in places where a call is possible to fail but, if retried more than once, it is more probable to succeed. As an addition to doing retries for fault-tolerance, you will learn how to use Circuit breakers, a technique to allow one subsystem to fail without destroying the entire system. In applications that rely heavily on accessing data from a data store, using the Cache-Aside pattern can improve the performance while reading data from the data store via caching. This pattern can be used for read and update operations of data to and from the data store. Last but not least, the chapter introduces the Throttling pattern, the concept where, based on a rate-limiting or alternative technique, you can control how users consume your API or your service and make sure the service does not get overwhelmed by one particular tenant.

To get the most out of this book

1. Use a machine with a recent version of Windows, Linux or MacOS.
2. Install Python 3.6. Also, it is useful to know about advanced syntax and new syntax introduced in the Python 3 releases. You might also want to learn about idiomatic Python programming, by checkout out resources on Internet about the topic, if needed.
3. Install and use Docker on your machine, to easily install and run the RabbitMQ server needed to run the microservices examples for Chapter 15, *Microservices and Patterns for the Cloud*. If you choose to use the Docker installation method, which is more and more needed for many server software and services packaged as containers, you will find useful information here https://hub.docker.com/_/rabbitmq/ as well as here <https://docs.nameko.io/en/stable/installation.html>.

Download the example code files

You can download the example code files for this book from your account at www.PacktPub.com. If you purchased this book elsewhere, you can visit www.packtpub.com/support and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at www.packtpub.com.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at <https://github.com/PacktPublishing/Mastering-Python-Design-Patterns-Second-Edition>. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at <https://github.com/PacktPublishing/>. Check them out!

Conventions used

There are a number of text conventions used throughout this book.

CodeInText: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "In the `Musician` class, the main action is performed by the `play()` method."

A block of code is set as follows:

```
class Musician:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return f'the musician {self.name}'
    def play(self):
        return 'plays music'
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "A **remote proxy**, which acts as the local representation of an object that really exists in a different address space (for example, a network server)."



Warnings or important notes appear like this.



Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email feedback@packtpub.com and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at questions@packtpub.com.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/submit-errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

1

The Factory Pattern

Design patterns are reusable programming solutions that have been used in various real-world contexts, and have proved to produce expected results. They are shared among programmers and continue being improved over time. This topic is popular thanks to the book by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, titled *Design Patterns: Elements of Reusable Object-Oriented Software*.



Gang of Four: The book by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides is also called the *Gang of Four* book for short (or *GOF book* for even shorter).

Here is a quote about design patterns from the *Gang of Four* book:

A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context.

There are several categories of design patterns used in **object-oriented programming**, depending on the type of problem they address and/or the types of solutions they help us build. In their book, the *Gang of Four* present 23 design patterns, split into three categories: **creational**, **structural**, and **behavioral**.

Creational design patterns are the first category we will cover throughout this chapter, and Chapters 2, *The Builder Pattern* and Chapter 3, *Other Creational Patterns*. These patterns deal with different aspects of object creation. Their goal is to provide better alternatives for situations where direct object creation, which in Python happens within the `__init__()` function, is not convenient.



See <https://docs.python.org/3/tutorial/classes.html> for a quick overview of object classes and the special `__init__()` method Python uses to initialize a new class instance.

We will start with the first creational design pattern from the *Gang of Four* book: the factory design pattern. In the factory design pattern, a **client** (meaning client code) asks for an object without knowing where the object is coming from (that is, which class is used to generate it). The idea behind a factory is to simplify the object creation process. It is easier to track which objects are created if this is done through a central function, compared to letting a client create objects using a direct class instantiation. A factory reduces the complexity of maintaining an application by decoupling the code that creates an object from the code that uses it.

Factories typically come in two forms—the **factory method**, which is a method (or simply a function for a Python developer) that returns a different object per input parameter, and the **abstract factory**, which is a group of factory methods used to create a family of related objects.

In this chapter, we will discuss:

- The factory method
- The abstract factory

The factory method

The factory method is based on a single function written to handle our object creation task. We execute it, passing a parameter that provides information about what we want, and, as a result, the wanted object is created.

Interestingly, when using the factory method, we are not required to know any details about how the resulting object is implemented and where it is coming from.

Real-world examples

An example of the factory method pattern used in reality is in the context of a plastic toy construction kit. The molding material used to construct plastic toys is the same, but different toys (different figures or shapes) can be produced using the right plastic molds. This is like having a factory method in which the input is the name of the toy that we want (for example, *duck* or *car*) and the output (after the molding) is the plastic toy that was requested.

In the software world, the Django web framework uses the factory method pattern for creating the fields of a web form. The `forms` module, included in Django, supports the creation of different kinds of fields (for example, `CharField`, `EmailField`, and so on). And parts of their behavior can be customized using attributes such as `max_length` or `required` (j.mp/djangofac). As an illustration, there follows a snippet that a developer could write for a form (the `PersonForm` form containing the fields `name` and `birth_date`) as part of a Django application's UI code:

```
from django import forms

class PersonForm(forms.Form):
    name = forms.CharField(max_length=100)
    birth_date = forms.DateField(required=False)
```

Use cases

If you realize that you cannot track the objects created by your application because the code that creates them is in many different places instead of in a single function/method, you should consider using the factory method pattern. The factory method centralizes object creation and tracking your objects becomes much easier. Note that it is absolutely fine to create more than one factory method, and this is how it is typically done in practice. Each factory method logically groups the creation of objects that have similarities. For example, one factory method might be responsible for connecting you to different databases (MySQL, SQLite), another factory method might be responsible for creating the geometrical object that you request (circle, triangle), and so on.

The factory method is also useful when you want to decouple object creation from object usage. We are not coupled/bound to a specific class when creating an object; we just provide partial information about what we want by calling a function. This means that introducing changes to the function is easy and does not require any changes to the code that uses it.

Another use case worth mentioning is related to improving the performance and memory usage of an application. A factory method can improve the performance and memory usage by creating new objects only if it is absolutely necessary. When we create objects using a direct class instantiation, extra memory is allocated every time a new object is created (unless the class uses caching internally, which is usually not the case). We can see that in practice in the following code (file `id.py`), it creates two instances of the same class, `A`, and uses the `id()` function to compare their **memory addresses**. The addresses are also printed in the output so that we can inspect them. The fact that the memory addresses are different means that two distinct objects are created as follows:

```
class A:
    pass

if __name__ == '__main__':
    a = A()
    b = A()
    print(id(a) == id(b))
    print(a, b)
```

Executing the `python id.py` command on my computer results in the following output:

```
False
<__main__.A object at 0x7f5771de8f60> <__main__.A object at 0x7f5771df2208>
```

Note that the addresses that you see if you execute the file are not the same as the ones I see because they depend on the current memory layout and allocation. But the result must be the same—the two addresses should be different. There's one exception that happens if you write and execute the code in the Python **Read-Eval-Print Loop (REPL)**—or simply put, the interactive prompt—but that's a REPL-specific optimization which does not happen normally.

Implementing the factory method

Data comes in many forms. There are two main file categories for storing/retrieving data: human-readable files and binary files. Examples of human-readable files are XML, RSS/Atom, YAML, and JSON. Examples of binary files are the `.sq3` file format used by SQLite and the `.mp3` audio file format used to listen to music.

In this example, we will focus on two popular human-readable formats—XML and JSON. Although human-readable files are generally slower to parse than binary files, they make data exchange, inspection, and modification much easier. For this reason, it is advised that you work with human-readable files, unless there are other restrictions that do not allow it (mainly unacceptable performance and proprietary binary formats).

In this case, we have some input data stored in an XML and a JSON file, and we want to parse them and retrieve some information. At the same time, we want to centralize the client's connection to those (and all future) external services. We will use the factory method to solve this problem. The example focuses only on XML and JSON, but adding support for more services should be straightforward.

First, let's take a look at the data files.

The JSON file, `movies.json`, is an example (found on GitHub) of a dataset containing information about American movies (title, year, director name, genre, and so on). This is actually a big file but here is an extract, simplified for better readability, to show how its content is organized:

```
[
  {
    "title": "After Dark in Central Park",
    "year": 1900,
    "director": null, "cast": null, "genre": null
  },
  {
    "title": "Boarding School Girls' Pajama Parade",
    "year": 1900,
    "director": null, "cast": null, "genre": null
  },
  {
    "title": "Buffalo Bill's Wild West Parad",
    "year": 1900,
    "director": null, "cast": null, "genre": null
  },
  {
    "title": "Caught",
    "year": 1900,
    "director": null, "cast": null, "genre": null
  },
  {
    "title": "Clowns Spinning Hats",
    "year": 1900,
    "director": null, "cast": null, "genre": null
  },
  {
    "title": "Capture of Boer Battery by British",
    "year": 1900,
    "director": "James H. White", "cast": null, "genre": "Short documentary"
  },
  {
    "title": "The Enchanted Drawing",
    "year": 1900,
    "director": "J. Stuart Blackton", "cast": null, "genre": null
  },
  {
    "title": "Family Troubles",
    "year": 1900,
    "director": null, "cast": null, "genre": null
  },
  {
    "title": "Feeding Sea Lions",
    "year": 1900,
```



```
"director":null, "cast":"Paul Boyton", "genre":null}  
]
```

The XML file, `person.xml`, is based on a Wikipedia example (j.mp/wikijson), and contains information about individuals (`firstName`, `lastName`, `gender`, and so on) as follows:

1. We start with the enclosing tag of the persons XML container:

```
<persons>
```

2. Then, an XML element representing a person's data code is presented as follows:

```
<person>  
  <firstName>John</firstName>  
  <lastName>Smith</lastName>  
  <age>25</age>  
  <address>  
    <streetAddress>21 2nd Street</streetAddress>  
    <city>New York</city>  
    <state>NY</state>  
    <postalCode>10021</postalCode>  
  </address>  
  <phoneNumbers>  
    <phoneNumber type="home">212 555-1234</phoneNumber>  
    <phoneNumber type="fax">646 555-4567</phoneNumber>  
  </phoneNumbers>  
  <gender>  
    <type>male</type>  
  </gender>  
</person>
```

3. An XML element representing another person's data is shown by the following code:

```
<person>  
  <firstName>Jimmy</firstName>  
  <lastName>Liar</lastName>  
  <age>19</age>  
  <address>  
    <streetAddress>18 2nd Street</streetAddress>  
    <city>New York</city>  
    <state>NY</state>  
    <postalCode>10021</postalCode>  
  </address>  
  <phoneNumbers>  
    <phoneNumber type="home">212 555-1234</phoneNumber>  
  </phoneNumbers>
```

```

    <gender>
      <type>male</type>
    </gender>
  </person>

```

4. An XML element representing a third person's data is shown by the code:

```

<person>
  <firstName>Patty</firstName>
  <lastName>Liar</lastName>
  <age>20</age>
  <address>
    <streetAddress>18 2nd Street</streetAddress>
    <city>New York</city>
    <state>NY</state>
    <postalCode>10021</postalCode>
  </address>
  <phoneNumbers>
    <phoneNumber type="home">212 555-1234</phoneNumber>
    <phoneNumber type="mobile">001 452-8819</phoneNumber>
  </phoneNumbers>
  <gender>
    <type>female</type>
  </gender>
</person>

```

5. Finally, we close the XML container:

```

</persons>

```

We will use two libraries that are part of the Python distribution for working with JSON and XML, `json` and `xml.etree.ElementTree`, as follows:

```

import json
import xml.etree.ElementTree as etree

```

The `JSONDataExtractor` class parses the JSON file and has a `parsed_data()` method that returns all data as a dictionary (`dict`). The property decorator is used to make `parsed_data()` appear as a normal attribute instead of a method, as follows:

```

class JSONDataExtractor:
    def __init__(self, filepath):
        self.data = dict()
        with open(filepath, mode='r', encoding='utf-8') as f:
            self.data = json.load(f)
        @property
        def parsed_data(self):
            return self.data

```

The `XMLDataExtractor` class parses the XML file and has a `parsed_data()` method that returns all data as a list of `xml.etree.Element` as follows:

```
class XMLDataExtractor:
    def __init__(self, filepath):
        self.tree = etree.parse(filepath)
    @property
    def parsed_data(self):
        return self.tree
```

The `dataextraction_factory()` function is a factory method. It returns an instance of `JSONDataExtractor` or `XMLDataExtractor` depending on the extension of the input file path as follows:

```
def dataextraction_factory(filepath):
    if filepath.endswith('json'):
        extractor = JSONDataExtractor
    elif filepath.endswith('xml'):
        extractor = XMLDataExtractor
    else:
        raise ValueError('Cannot extract data from {}'.format(filepath))
    return extractor(filepath)
```

The `extract_data_from()` function is a wrapper of `dataextraction_factory()`. It adds exception handling as follows:

```
def extract_data_from(filepath):
    factory_obj = None
    try:
        factory_obj = dataextraction_factory(filepath)
    except ValueError as e:
        print(e)
    return factory_obj
```

The `main()` function demonstrates how the factory method design pattern can be used. The first part makes sure that exception handling is effective, as follows:

```
def main():
    sqlite_factory = extract_data_from('data/person.sql3')
    print()
```

The next part shows how to work with the JSON files using the factory method. Based on the parsing, the title, year, director name, and genre of the movie can be shown (when the value is not empty), as follows:

```
json_factory = extract_data_from('data/movies.json')
json_data = json_factory.parsed_data
print(f'Found: {len(json_data)} movies')
for movie in json_data:
    print(f"Title: {movie['title']}")
    year = movie['year']
    if year:
        print(f"Year: {year}")
    director = movie['director']
    if director:
        print(f"Director: {director}")
    genre = movie['genre']
    if genre:
        print(f"Genre: {genre}")
    print()
```

The final part shows you how to work with the XML files using the factory method. XPath is used to find all person elements that have the last name Liar (using `liars = xml_data.findall(f"//person[lastName='Liar']")`). For each matched person, the basic name and phone number information are shown, as follows:

```
xml_factory = extract_data_from('data/person.xml')
xml_data = xml_factory.parsed_data
liars = xml_data.findall(f"//person[lastName='Liar']")
print(f'found: {len(liars)} persons')
for liar in liars:
    firstname = liar.find('firstName').text
    print(f'first name: {firstname}')
    lastname = liar.find('lastName').text
    print(f'last name: {lastname}')
    [print(f"phone number ({p.attrib['type']}):", p.text)
     for p in liar.find('phoneNumbers')]
    print()
```

Here is the summary of the implementation (you can find the code in the `factory_method.py` file):

1. We start by importing the modules we need (`json` and `ElementTree`).
2. We define the JSON data extractor class (`JSONDataExtractor`).
3. We define the XML data extractor class (`XMLDataExtractor`).
4. We add the factory function, `dataextraction_factory()`, for getting the right data extractor class.
5. We also add our wrapper for handling exceptions, the `extract_data_from()` function.
6. Finally, we have the `main()` function, followed by Python's conventional trick for calling it when invoking this file from the command line. The following are the aspects of the `main` function:
 - We try to extract data from an SQL file (`data/person.sql3`), to show how the exception is handled
 - We extract data from a JSON file and parse the result
 - We extract data from an XML file and parse the result

The following is the type of output (for the different cases) you will get by calling the `python factory_method.py` command.

First, there is an exception message when trying to access an SQLite (`.sql3`) file:

```
Cannot extract data from data/person.sql3
```

Then, we get the following result from processing the `movies` file (JSON):

```
Found: 9 movies
Title: After Dark in Central Park
Year: 1900

Title: Boarding School Girls' Pajama Parade
Year: 1900

Title: Buffalo Bill's Wild West Parad
Year: 1900

Title: Caught
Year: 1900

Title: Clowns Spinning Hats
Year: 1900

Title: Capture of Boer Battery by British
Year: 1900
Director: James H. White
Genre: Short documentary

Title: The Enchanted Drawing
Year: 1900
Director: J. Stuart Blackton

Title: Family Troubles
Year: 1900

Title: Feeding Sea Lions
Year: 1900
```

Finally, we get this result from processing the person XML file to find the people whose last name is `Liar`:

```
found: 2 persons
first name: Jimmy
last name: Liar
phone number (home): 212 555-1234

first name: Patty
last name: Liar
phone number (home): 212 555-1234
phone number (mobile): 001 452-8819
```

Notice that although `JSONDataExtractor` and `XMLDataExtractor` have the same interfaces, what is returned by `parsed_data()` is not handled in a uniform way. Different Python code must be used to work with each **data extractor**. Although it would be nice to be able to use the same code for all extractors, this is not realistic for the most part, unless we use some kind of common mapping for the data, which is very often provided by external data providers. Assuming that you can use exactly the same code for handling the XML and JSON files, what changes are required to support a third format, for example, SQLite? Find an SQLite file, or create your own and try it.

The abstract factory

The abstract factory design pattern is a generalization of the factory method. Basically, an abstract factory is a (logical) group of factory methods, where each factory method is responsible for generating a different kind of object.

We are going to discuss some examples, use cases, and a possible implementation.

Real-world examples

The abstract factory is used in car manufacturing. The same machinery is used for stamping the parts (doors, panels, hoods, fenders, and mirrors) of different car models. The model that is assembled by the machinery is configurable and easy to change at any time.

In the software category, the `factory_boy` (https://github.com/FactoryBoy/factory_boy) package provides an abstract factory implementation for creating Django models in tests. It is used for creating instances of models that support **test-specific attributes**. This is important because, this way, your tests become readable and you avoid sharing unnecessary code.



Django models are special classes used by the framework to help store and interact with data in the database (tables). See the Django documentation (<https://docs.djangoproject.com>) for more details.

Use cases

Since the abstract factory pattern is a generalization of the factory method pattern, it offers the same benefits, it **makes tracking an object creation easier**, it **decouples object creation from object usage**, and it **gives us the potential to improve the memory usage and performance of our application**.

But, a question is raised: How do we know when to use the factory method versus using an abstract factory? The answer is that we usually start with the factory method which is simpler. If we find out that our application requires many factory methods, which it makes sense to combine to create a family of objects, we end up with an abstract factory.

A benefit of the abstract factory that is usually not very visible from a user's point of view when using the factory method is that it gives us the ability to modify the behavior of our application dynamically (at runtime) by changing the active factory method. The classic example is the ability to change the look and feel of an application (for example, Apple-like, Windows-like, and so on) for the user while the application is in use, without the need to terminate it and start it again.

Implementing the abstract factory pattern

To demonstrate the abstract factory pattern, I will reuse one of my favorite examples, included in the book, *Python 3 Patterns, Recipes and Idioms*, by Bruce Eckel. Imagine that we are creating a game or we want to include a mini-game as part of our application to entertain our users. We want to include at least two games, one for children and one for adults. We will decide which game to create and launch at runtime, based on user input. An abstract factory takes care of the game creation part.

Let's start with the kid's game. It is called **FrogWorld**. The main hero is a frog who enjoys eating bugs. Every hero needs a good name, and in our case, the name is given by the user at runtime. The `interact_with()` method is used to describe the interaction of the frog with an obstacle (for example, a bug, puzzle, and other frogs) as follows:

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

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
        act = obstacle.action()
        msg = f'{self} the Frog encounters {obstacle} and {act}!'
        print(msg)
```

There can be many different kinds of obstacles but for our example, an obstacle can only be a bug. When the frog encounters a bug, only one action is supported. It eats it:

```
class Bug:
    def __str__(self):
        return 'a bug'

    def action(self):
        return 'eats it'
```

The `FrogWorld` class is an abstract factory. Its main responsibilities are creating the main character and the obstacle(s) in the game. Keeping the creation methods separate and their names generic (for example, `make_character()` and `make_obstacle()`) allows us to change the active factory (and therefore the active game) dynamically without any code changes. In a statically typed language, the abstract factory would be an abstract class/interface with empty methods, but in Python, this is not required because the types are checked at runtime (`j.mp/ginstromdp`). The code is as follows:

```
class FrogWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t----- Frog World -----'

    def make_character(self):
        return Frog(self.player_name)
```

```
def make_obstacle(self):
    return Bug()
```

The **WizardWorld** game is similar. The only difference is that the wizard battles against monsters such as orks instead of eating bugs!

Here is the definition of the Wizard class, which is similar to the Frog one:

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

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
        act = obstacle.action()
        msg = f'{self} the Wizard battles against {obstacle}
        and {act}!'
        print(msg)
```

Then, the definition of the Ork class is as follows:

```
class Ork:
    def __str__(self):
        return 'an evil ork'

    def action(self):
        return 'kills it'
```

We also need to define the WizardWorld class, similar to the FrogWorld one that we have discussed; the obstacle, in this case, is an Ork instance:

```
class WizardWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t----- Wizard World -----'

    def make_character(self):
        return Wizard(self.player_name)

    def make_obstacle(self):
        return Ork()
```

The `GameEnvironment` class is the main entry point of our game. It accepts the factory as an input and uses it to create the world of the game. The `play()` method initiates the interaction between the created hero and the obstacle, as follows:

```
class GameEnvironment:
    def __init__(self, factory):
        self.hero = factory.make_character()
        self.obstacle = factory.make_obstacle()

    def play(self):
        self.hero.interact_with(self.obstacle)
```

The `validate_age()` function prompts the user to give a valid age. If the age is not valid, it returns a tuple with the first element set to `False`. If the age is fine, the first element of the tuple is set to `True` and that's the case where we actually care about the second element of the tuple, which is the age given by the user, as follows:

```
def validate_age(name):
    try:
        age = input(f'Welcome {name}. How old are you? ')
        age = int(age)
    except ValueError as err:
        print(f"Age {age} is invalid, please try again...")
        return (False, age)
    return (True, age)
```

Last but not least comes the `main()` function. It asks for the user's name and age, and decides which game should be played, given the age of the user, as follows:

```
def main():
    name = input("Hello. What's your name? ")
    valid_input = False
    while not valid_input:
        valid_input, age = validate_age(name)
    game = FrogWorld if age < 18 else WizardWorld
    environment = GameEnvironment(game(name))
    environment.play()
```

The summary for the implementation we just discussed (see the complete code in the `abstract_factory.py` file) is as follows:

1. We define the `Frog` and `Bug` classes for the `FrogWorld` game.
2. We add the `FrogWorld` class, where we use our `Frog` and `Bug` classes.

3. We define the `Wizard` and `Ork` classes for the `WizardWorld` game.
4. We add the `WizardWorld` class, where we use our `Wizard` and `Ork` classes.
5. We define the `GameEnvironment` class.
6. We add the `validate_age()` function.
7. Finally, we have the `main()` function, followed by the conventional trick for calling it. The following are the aspects of this function:
 - We get the user's input for name and age
 - We decide which game class to use based on the user's age
 - We instantiate the right game class, and then the `GameEnvironment` class
 - We call `.play()` on the environment object to play the game

Let's call this program using the `python abstract_factory.py` command, and see some sample output.

The sample output for a teenager is as follows:

```
Hello. What's your name? Billy
Welcome Billy. How old are you? 12

----- Frog World -----
Billy the Frog encounters a bug and eats it!
```

The sample output for an adult is as follows:

```
Hello. What's your name? Charles
Welcome Charles. How old are you? 25

----- Wizard World -----
Charles the Wizard battles against an evil ork and kills it!
```

Try extending the game to make it more complete. You can go as far as you want; create many obstacles, many enemies, and whatever else you like.

Summary

In this chapter, we have seen how to use the factory method and the abstract factory design patterns. Both patterns are used when we want to track object creation, decouple object creation from object usage, or even improve the performance and resource usage of an application. Improving the performance was not demonstrated in this chapter. You might consider trying it as a good exercise.

The factory method design pattern is implemented as a single function that doesn't belong to any class and is responsible for the creation of a single kind of object (a shape, a connection point, and so on). We saw how the factory method relates to toy construction, mentioned how it is used by Django for creating different form fields, and discussed other possible use cases for it. As an example, we implemented a factory method that provided access to the XML and JSON files.

The abstract factory design pattern is implemented as a number of factory methods that belong to a single class and are used to create a family of related objects (the parts of a car, the environment of a game, and so forth). We mentioned how the abstract factory is related to car manufacturing, how the `django_factory` package for Django makes use of it to create clean tests, and then we covered its common use cases. Our implementation example of the abstract factory is a mini-game that shows how we can use many related factories in a single class.

In the next chapter, we will discuss the builder pattern, which is another creational pattern that can be used for fine-tuning the creation of complex objects.

2

The Builder Pattern

In the previous chapter, we covered the first two creational patterns, the factory method and abstract factory, which both offer approaches to improve the way we create objects in nontrivial cases.

Now, imagine that we want to create an object that is composed of multiple parts and the composition needs to be done step by step. The object is not complete unless all its parts are fully created. That's where the builder design pattern can help us. The builder pattern separates the construction of a complex object from its representation. By keeping the construction separate from the representation, the same construction can be used to create several different representations (j.mp/builderpat).

A practical example can help us understand what the purpose of the builder pattern is. Suppose that we want to create an HTML page generator. The basic structure (construction part) of an HTML page is always the same: it begins with `<html>` and finishes with `</html>`, inside the HTML section are the `<head>` and `</head>` elements, inside the head section are the `<title>` and `</title>` elements, and so forth. But the representation of the page can differ. Each page has its own title, its own headings, and different `<body>` contents. Moreover, the page is usually built in steps: one function adds the title, another adds the main heading, another the footer, and so on. Only after the whole structure of a page is complete can it be shown to the client using a final render function. We can take it even further and extend the HTML generator so that it can generate totally different HTML pages. One page might contain tables, another page might contain image galleries, yet another page contains the contact form, and so on.

The HTML page generation problem can be solved using the builder pattern. In this pattern, there are two main participants:

- **The builder:** The component responsible for creating the various parts of a complex object. In this example, these parts are the title, heading, body, and the footer of the page.
- **The director:** The component that controls the building process using a `builder` instance. It calls the builder's functions for setting the title, the heading, and so on. And, using a different `builder` instance allows us to create a different HTML page without touching any of the code of the director.

In this chapter, we will discuss:

- Real-world examples
- Use cases
- Implementation

Real-world examples

In our everyday life, the builder design pattern is used in fast-food restaurants. The same procedure is always used to prepare a burger and the packaging (box and paper bag), even if there are many different kinds of burgers (classic, cheeseburger, and more) and different packages (small-sized box, medium-sized box, and so forth). The difference between a classic burger and a cheeseburger is in the representation, and not in the construction procedure. In this case, the **director** is the cashier who gives instructions about what needs to be prepared to the crew, and the **builder** is the person from the crew that takes care of the specific order.

We can also find software examples:

- The HTML example that was mentioned at the beginning of the chapter is actually used by `django-widgy` (<https://wid.gy/>), a third-party tree editor for Django that can be used as a **content management system (CMS)**. The `django-widgy` editor contains a page builder that can be used for creating HTML pages with different layouts.

- The `django-query-builder` library (<https://github.com/ambitioninc/django-query-builder>) is another third-party Django library that relies on the builder pattern. This library can be used for building SQL queries dynamically, allowing you to control all aspects of a query and create a different range of queries, from simple to very complex ones.

Use cases

We use the builder pattern when we know that an object must be created in multiple steps, and different representations of the same construction are required. These requirements exist in many applications such as page generators (for example, the HTML page generator mentioned in this chapter), document converters, and **user interface (UI)** form creators (`j.mp/pipbuild`).

Some online resources mention that the builder pattern can also be used as a solution to the telescopic constructor problem. The telescopic constructor problem occurs when we are forced to create a new constructor for supporting different ways of creating an object. The problem is that we end up with many constructors and long parameter lists, which are hard to manage. An example of the telescopic constructor is listed at the Stack Overflow website (`j.mp/sobuilder`). Fortunately, this problem does not exist in Python, because it can be solved in at least two ways:

- With named parameters (`j.mp/sobuipython`)
- With argument list unpacking (`j.mp/arglistpy`)

At this point, the distinction between the builder pattern and the factory pattern might not be very clear. The main difference is that a factory pattern creates an object in a single step, whereas a builder pattern creates an object in multiple steps, and almost always through the use of a director. Some targeted implementations of the builder pattern, such as Java's `StringBuilder`, bypass the use of a director, but that's the exception to the rule.

Another difference is that while a factory pattern returns a created object immediately, in the builder pattern the client code explicitly asks the director to return the final object when it needs it (`j.mp/builderpat`).

The new computer analogy might help you to distinguish between a builder pattern and a factory pattern. Assume that you want to buy a new computer. If you decide to buy a specific, preconfigured computer model, for example, the latest Apple 1.4 GHz Mac Mini, you use the factory pattern. All the hardware specifications are already predefined by the manufacturer, who knows what to do without consulting you. The manufacturer typically receives just a single instruction. Code-wise, this would look like the following (apple_factory.py):

```
MINI14 = '1.4GHz Mac mini'

class AppleFactory:
    class MacMini14:
        def __init__(self):
            self.memory = 4 # in gigabytes
            self.hdd = 500 # in gigabytes
            self.gpu = 'Intel HD Graphics 5000'

        def __str__(self):
            info = (f'Model: {MINI14}',
                    f'Memory: {self.memory}GB',
                    f'Hard Disk: {self.hdd}GB',
                    f'Graphics Card: {self.gpu}')
            return '\n'.join(info)

    def build_computer(self, model):
        if model == MINI14:
            return self.MacMini14()
        else:
            msg = f"I don't know how to build {model}"
            print(msg)
```

Now, we add the main part of the program, the snippet which uses the AppleFactory class:

```
if __name__ == '__main__':
    afac = AppleFactory()
    mac_mini = afac.build_computer(MINI14)
    print(mac_mini)
```



Notice the nested `MacMini14` class. This is a neat way of forbidding the direct instantiation of a class.

Another option would be to buy a custom PC. In this case, you use the builder pattern. You are the director that gives orders to the manufacturer (`builder`) about your ideal computer specifications. Code-wise, this looks like the following (`computer_builder.py`):

- We define the `Computer` class as follows:

```
class Computer:
    def __init__(self, serial_number):
        self.serial = serial_number
        self.memory = None # in gigabytes
        self.hdd = None # in gigabytes
        self.gpu = None

    def __str__(self):
        info = (f'Memory: {self.memory}GB',
                f'Hard Disk: {self.hdd}GB',
                f'Graphics Card: {self.gpu}')
        return '\n'.join(info)
```

- We define the `ComputerBuilder` class:

```
class ComputerBuilder:
    def __init__(self):
        self.computer = Computer('AG23385193')

    def configure_memory(self, amount):
        self.computer.memory = amount

    def configure_hdd(self, amount):
        self.computer.hdd = amount

    def configure_gpu(self, gpu_model):
        self.computer.gpu = gpu_model
```

- We define the `HardwareEngineer` class as follows:

```
class HardwareEngineer:
    def __init__(self):
        self.builder = None

    def construct_computer(self, memory, hdd, gpu):
        self.builder = ComputerBuilder()
        steps = (self.builder.configure_memory(memory),
                 self.builder.configure_hdd(hdd),
                 self.builder.configure_gpu(gpu))
        [step for step in steps]

    @property
    def computer(self):
        return self.builder.computer
```

- We end our code with the `main()` function, followed by the trick to call it when the file is called from the command line:

```
def main():
    engineer = HardwareEngineer()
    engineer.construct_computer(hdd=500,
                               memory=8,
                               gpu='GeForce GTX 650 Ti')

    computer = engineer.computer
    print(computer)

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

The basic changes are the introduction of a builder, `ComputerBuilder`, a director, `HardwareEngineer`, and the step-by-step construction of a computer, which now supports different configurations (notice that `memory`, `hdd`, and `gpu` are parameters and are not preconfigured). What do we need to do if we want to support the construction of tablets? Implement this as an exercise.

You might also want to change the computer's `serial_number` into something that is different for each computer, because as it is now, it means that all computers will have the same serial number (which is impractical).

Implementation

Let's see how we can use the builder design pattern to make a pizza-ordering application. The pizza example is particularly interesting because a pizza is prepared in steps that should follow a specific order. To add the sauce, you first need to prepare the dough. To add the topping, you first need to add the sauce. And you can't start baking the pizza unless both the sauce and the topping are placed on the dough. Moreover, each pizza usually requires a different baking time, depending on the thickness of its dough and the topping used.

We start by importing the required modules and declaring a few `Enum` parameters (`j.mp/pytenum`) plus a constant that is used many times in the application. The `STEP_DELAY` constant is used to add a time delay between the different steps of preparing a pizza (prepare the dough, add the sauce, and so on) as follows:

```
from enum import Enum
import time
PizzaProgress = Enum('PizzaProgress', 'queued preparation baking ready')
PizzaDough = Enum('PizzaDough', 'thin thick')
PizzaSauce = Enum('PizzaSauce', 'tomato creme_fraiche')
PizzaTopping = Enum('PizzaTopping',
                    'mozzarella double_mozzarella bacon ham mushrooms red_onion
                    oregano')
STEP_DELAY = 3 # in seconds for the sake of the example
```

Our end product is a pizza, which is described by the `Pizza` class. When using the builder pattern, the end product does not have many responsibilities, since it is not supposed to be instantiated directly. A builder creates an instance of the end product and makes sure that it is properly prepared. That's why the `Pizza` class is so minimal. It basically initializes all data to sane default values. An exception is the `prepare_dough()` method.

The `prepare_dough()` method is defined in the `Pizza` class instead of a builder for two reasons. First, to clarify the fact that the end product is typically minimal, which does not mean that you should never assign it any responsibilities. Second, to promote code reuse through composition.

So, we define our `Pizza` class as follows:

```
class Pizza:
    def __init__(self, name):
        self.name = name
        self.dough = None
        self.sauce = None
        self.topping = []
```

```
def __str__(self):
    return self.name

def prepare_dough(self, dough):
    self.dough = dough
    print(f'preparing the {self.dough.name} dough of your {self}...')
    time.sleep(STEP_DELAY)
    print(f'done with the {self.dough.name} dough')
```

There are two builders: one for creating a margarita pizza (`MargaritaBuilder`) and another for creating a creamy bacon pizza (`CreamyBaconBuilder`). Each builder creates a `Pizza` instance and contains methods that follow the pizza-making procedure: `prepare_dough()`, `add_sauce()`, `add_topping()`, and `bake()`. To be precise, `prepare_dough()` is just a wrapper to the `prepare_dough()` method of the `Pizza` class.

Notice how each builder takes care of all the pizza-specific details. For example, the topping of the margarita pizza is double mozzarella and oregano, while the topping of the creamy bacon pizza is mozzarella, bacon, ham, mushrooms, red onion, and oregano.

This part of our code is laid out as follows:

- We define the `MargaritaBuilder` class as follows:

```
class MargaritaBuilder:
    def __init__(self):
        self.pizza = Pizza('margarita')
        self.progress = PizzaProgress.queued
        self.baking_time = 5 # in seconds for the sake of
                               the example

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough.thin)

    def add_sauce(self):
        print('adding the tomato sauce to your margarita...')
        self.pizza.sauce = PizzaSauce.tomato
        time.sleep(STEP_DELAY)
        print('done with the tomato sauce')

    def add_topping(self):
        topping_desc = 'double mozzarella, oregano'
        topping_items = (PizzaTopping.double_mozzarella,
                         PizzaTopping.oregano)
        print(f'adding the topping ({topping_desc}) to your
              margarita')
        self.pizza.topping.append([t for t in topping_items])
```

```
time.sleep(STEP_DELAY)
print(f'done with the topping ({topping_desc})')
```

```
def bake(self):
    self.progress = PizzaProgress.baking
    print(f'baking your margarita for {self.baking_time}
seconds')
    time.sleep(self.baking_time)
    self.progress = PizzaProgress.ready
    print('your margarita is ready')
```

- We define the CreamyBaconBuilder class as follows:

```
class CreamyBaconBuilder:
    def __init__(self):
        self.pizza = Pizza('creamy bacon')
        self.progress = PizzaProgress.queued
        self.baking_time = 7 # in seconds for the sake of
the example

    def prepare_dough(self):
        self.progress = PizzaProgress.preparation
        self.pizza.prepare_dough(PizzaDough.thick)

    def add_sauce(self):
        print('adding the crème fraîche sauce to your creamy
bacon')
        self.pizza.sauce = PizzaSauce.creme_fraiche
        time.sleep(STEP_DELAY)
        print('done with the crème fraîche sauce')

    def add_topping(self):
        topping_desc = 'mozzarella, bacon, ham, mushrooms,
red onion, oregano'
        topping_items = (PizzaTopping.mozzarella,
                          PizzaTopping.bacon,
                          PizzaTopping.ham,
                          PizzaTopping.mushrooms,
                          PizzaTopping.red_onion,
                          PizzaTopping.oregano)
        print(f'adding the topping ({topping_desc}) to your
creamy bacon')
        self.pizza.topping.append([t for t in topping_items])
        time.sleep(STEP_DELAY)
        print(f'done with the topping ({topping_desc})')

    def bake(self):
        self.progress = PizzaProgress.baking
```

```

print(f'baking your creamy bacon for {self.baking_time}
seconds')
time.sleep(self.baking_time)
self.progress = PizzaProgress.ready
print('your creamy bacon is ready')

```

The director in this example is the waiter. The core of the `Waiter` class is the `construct_pizza()` method, which accepts a builder as a parameter and executes all the pizza-preparation steps in the right order. Choosing the appropriate builder, which can even be done at runtime, gives us the ability to create different pizza styles without modifying any of the code of the director (`Waiter`). The `Waiter` class also contains the `pizza()` method, which returns the end product (prepared pizza) as a variable to the caller as follows:

```

class Waiter:
    def __init__(self):
        self.builder = None

    def construct_pizza(self, builder):
        self.builder = builder
        steps = (builder.prepare_dough,
                  builder.add_sauce,
                  builder.add_topping,
                  builder.bake)
        [step() for step in steps]

    @property
    def pizza(self):
        return self.builder.pizza

```

The `validate_style()` function is similar to the `validate_age()` function as described in Chapter 1, *The Factory Pattern*. It is used to make sure that the user gives valid input, which in this case is a character that is mapped to a pizza builder. The `m` character uses the `MargaritaBuilder` class and the `c` character uses the `CreamyBaconBuilder` class. These mappings are in the builder parameter. A tuple is returned, with the first element set to `True` if the input is valid, or `False` if it is invalid as follows:

```

def validate_style(builders):
    try:
        input_msg = 'What pizza would you like, [m]argarita or
[c]reamy bacon? '
        pizza_style = input(input_msg)
        builder = builders[pizza_style]()
        valid_input = True
    except KeyError:
        error_msg = 'Sorry, only margarita (key m) and creamy

```

```

        bacon (key c) are available'
    print(error_msg)
    return (False, None)
return (True, builder)

```

The last part is the `main()` function. The `main()` function contains code for instantiating a pizza builder. The pizza builder is then used by the `Waiter` director for preparing the pizza. The created pizza can be delivered to the client at any later point:

```

def main():
    builders = dict(m=MargaritaBuilder, c=CreamyBaconBuilder)
    valid_input = False
    while not valid_input:
        valid_input, builder = validate_style(builders)
    print()
    waiter = Waiter()
    waiter.construct_pizza(builder)
    pizza = waiter.pizza
    print()
    print(f'Enjoy your {pizza}!')

```

Here is the summary of the implementation (see the complete code in the `builder.py` file):

1. We start with a couple of imports we need, for the standard `Enum` class and `time` module.
2. We declare the variables for a few constants: `PizzaProgress`, `PizzaDough`, `PizzaSauce`, `PizzaTopping`, and `STEP_DELAY`.
3. We define our `Pizza` class.
4. We define the classes for two builders, `MargaritaBuilder` and `CreamyBaconBuilder`.
5. We define our `Waiter` class.
6. We add the `validate_style()` function to improve things regarding exception handling.
7. Finally, we have the `main()` function, followed by the snippet for calling it when the program is run. In the `main` function, the following happens:
 - We make it possible to choose the pizza builder based on the user's input, after validation via the `validate_style()` function.
 - The pizza builder is used by the waiter for preparing the pizza.
 - The created pizza is then delivered.

Here is the output produced by calling the `python builder.py` command to execute this example program:

```
What pizza would you like, [m]argarita or [c]reamy bacon? r
Sorry, only margarita (key m) and creamy bacon (key c) are available
What pizza would you like, [m]argarita or [c]reamy bacon? m

preparing the thin dough of your margarita...
done with the thin dough
adding the tomato sauce to your margarita...
done with the tomato sauce
adding the topping (double mozzarella, oregano) to your margarita
done with the topping (double mozzarella, oregano)
baking your margarita for 5 seconds
your margarita is ready

Enjoy your margarita!
```

But... supporting only two pizza types is a shame. Feel like getting a Hawaiian pizza builder? Consider using inheritance after thinking about the advantages and disadvantages. Check the ingredients of a typical Hawaiian pizza and decide which class you need to extend: `MargaritaBuilder` or `CreamyBaconBuilder`?. Perhaps both (`j.mp/pymulti`)?

In his book, *Effective Java (2nd edition)*, Joshua Bloch describes an interesting variation of the builder pattern where calls to builder methods are chained. This is accomplished by defining the builder itself as an inner class and returning itself from each of the setter-like methods on it. The `build()` method returns the final object. This pattern is called the **fluent builder**. Here's a Python implementation, which was kindly provided by a reviewer of the book:

```
class Pizza:
    def __init__(self, builder):
        self.garlic = builder.garlic
        self.extra_cheese = builder.extra_cheese

    def __str__(self):
        garlic = 'yes' if self.garlic else 'no'
        cheese = 'yes' if self.extra_cheese else 'no'
        info = (f'Garlic: {garlic}', f'Extra cheese: {cheese}')
        return '\n'.join(info)

class PizzaBuilder:
    def __init__(self):
        self.extra_cheese = False
        self.garlic = False
```

```

def add_garlic(self):
    self.garlic = True
    return self

def add_extra_cheese(self):
    self.extra_cheese = True
    return self

def build(self):
    return Pizza(self)

if __name__ == '__main__':
    pizza = Pizza.PizzaBuilder().add_garlic().add_extra_cheese().build()
    print(pizza)

```

Adapt the pizza example to make use of the fluent builder pattern. Which version of the two do you prefer? What are the pros and cons of each version?

Summary

In this chapter, we have seen how to use the builder design pattern. We use the builder pattern for creating an object in situations where using the factory pattern (either a factory method or an abstract factory) is not a good option. A builder pattern is usually a better candidate than a factory pattern when the following applies:

- We want to create a complex object (an object composed of many parts and created in different steps that might need to follow a specific order).
- Different representations of an object are required, and we want to keep the construction of an object decoupled from its representation.
- We want to create an object at one point in time, but access it at a later point.

We saw how the builder pattern is used in fast-food restaurants for preparing meals, and how two third-party Django packages, `django-widgy`, and `django-query-builder`, use it for generating HTML pages and dynamic SQL queries, respectively. We focused on the differences between a builder pattern and a factory pattern, and provided a preconfigured (factory) and customer (builder) computer order analogy to clarify them.

In the *Implementation* section, we looked at how to create a pizza-ordering application with preparation dependencies. There were many recommended and interesting exercises in this chapter, including implementing a fluent builder.

In the next chapter, you will learn about other useful creational patterns.

Other Creational Patterns

In the previous chapter, we covered a third creational pattern, that is, builder, which offers a nice way of creating the various parts of a complex object. Besides the factory method, the abstract factory, and the builder patterns covered so far, there are other creational patterns that are interesting to discuss, such as the **prototype** pattern and the **singleton** pattern.

What is the prototype pattern? The prototype pattern is useful when one needs to create objects based on an existing object by using a **cloning** technique.

As you may have guessed, the idea is to use a copy of that object's complete structure to produce the new object. We will see that this is almost natural in Python because we have a **copy feature** that helps greatly in using this technique.

What is the singleton pattern? The singleton pattern offers a way to implement a class from which you can only create one object, hence the name singleton. As you will understand with our exploration of this pattern, or while doing your own research, there have always been discussions about this pattern, and some even consider it an **anti-pattern**.

Besides that, what is interesting is that it is useful when we need to create one and only one object, for example to store and maintain a global state for our program, and in Python it can be implemented using some special built-in features.

In this chapter, we will discuss:

- The prototype pattern
- The singleton pattern

The prototype pattern

Sometimes, we need to create an exact copy of an object. For instance, assume that you want to create an application for storing, sharing, and editing presentation and marketing content for products promoted by a group of salespeople. Think of the popular distribution model called **direct selling** or **network marketing**, the home-based activity where individuals partner with a company to distribute products within their social network, using promotional tools (brochures, PowerPoint presentations, videos, and so on).

Let's say a user, Bob, leads a team of distributors within a network marketing organization. They use a presentation video on a daily basis to introduce the product to their prospects. At some point, Bob gets his friend, Alice, to join him and she also uses the same video (one of the governing principles is to follow the system or, as they say, *Duplicate what already works*). But Alice soon finds prospects that could join her team and help her business grow, if only the video was in French, or at least subtitled. *What should they do?* The original presentation video cannot be used for the different custom needs that may arise.

To help everyone, the system could allow the distributors with certain rank or trust levels, such as Bob, to create independent copies of the original presentation video, as long as the new version is validated by the **compliance team** of the **backing company** before public use. Each copy is called a clone; it is an exact copy of the original object at a specific point in time.

So Bob, with the validation of the compliance team, which is part of the process, makes a copy of the presentation video to address the new need and hands it over to Alice. She could then adapt that version by adding French subtitles.

With cloning, Bob and Alice can have their own copy of a video, and as such, changes by each one of them will not affect the other person's version of the material. In the alternative situation, which is what actually happens by default, each person would hold a reference to the same (reference) object; changes made by Bob would impact Alice and vice versa.

The prototype design pattern helps us with creating object clones. In its simplest version, this pattern is just a `clone()` function that accepts an object as an input parameter and returns a clone of it. In Python, this can be done using the `copy.deepcopy()` function.

Real-world examples

A non-computer example that comes to mind is the sheep named Dolly that was created by researchers in Scotland by cloning a cell from a mammary gland.

There are many Python applications that make use of the prototype pattern ([j.mp/pythonprot](#)), but it is almost never referred to as "prototype" since cloning objects is a built-in feature of the language.

Use cases

The prototype pattern is useful when we have an existing object that needs to stay untouched, and we want to create an exact copy of it, allowing changes in some parts of the copy.

There is also the frequent need for duplicating an object that is populated from a database and has references to other database-based objects. It is costly (multiple queries to a database) to clone such a complex object, so a prototype is a convenient way to solve the problem.

Implementation

Nowadays, some organizations, even of small size, deal with many websites and apps via their infrastructure/DevOps teams, hosting providers, or cloud service providers.

When you have to manage multiple websites, there is a point where it becomes difficult to follow. You need to access information quickly such as IP addresses that are involved, domain names and their expiration dates, and maybe details about the DNS parameters. So you need a kind of inventory tool.

Let's imagine how these teams deal with this type of data for daily activities, and touch on the implementation of a piece of software that helps consolidate and maintain the data (other than in Excel spreadsheets).

First, we need to import Python's standard `copy` module, as follows:

```
import copy
```

At the heart of this system, we will have a `Website` class for holding all the useful information such as the name, the domain name, the description, the author of a website we are managing, and so on.

In the `__init__()` method of the class, only some parameters are fixed: `name`, `domain`, `description`, and `author`. But we also want flexibility, and client code can pass more parameters in the form of keywords (`name=value`) using the `kwargs` variable-length collection (a Python dictionary).

Note that there is a Python idiom to set an arbitrary attribute named `attr` with a value `val` on an object `obj`, using the `setattr()` built-in function: `setattr(obj, attr, val)`.

So, we are using this technique for the optional attributes of our class, at the end of the initialization method, this way:

```
for key in kwargs:
    setattr(self, key, kwargs[key])
```

So our `Website` class is defined as follows:

```
class Website:
    def __init__(self, name, domain, description, author, **kwargs):
        '''Examples of optional attributes (kwargs):
           category, creation_date, technologies, keywords.
           '''
        self.name = name
        self.domain = domain
        self.description = description
        self.author = author
        for key in kwargs:
            setattr(self, key, kwargs[key])

    def __str__(self):
        summary = [f'Website "{self.name}"\n',]
        infos = vars(self).items()
        ordered_infos = sorted(infos)
        for attr, val in ordered_infos:
            if attr == 'name':
                continue
            summary.append(f'{attr}: {val}\n')
        return ''.join(summary)
```

Next, the `Prototype` class implements the prototype design pattern.

The heart of the `Prototype` class is the `clone()` method, which is in charge of cloning the object using the `copy.deepcopy()` function. Since cloning means we allow setting values for optional attributes, notice how we use the `setattr()` technique here with the `attrs` dictionary.

Also, for more convenience, the `Prototype` class contains the `register()` and `unregister()` methods, which can be used to keep track of the cloned objects in a dictionary:

```
class Prototype:
    def __init__(self):
        self.objects = dict()

    def register(self, identifier, obj):
        self.objects[identifier] = obj

    def unregister(self, identifier):
        del self.objects[identifier]

    def clone(self, identifier, **attrs):
        found = self.objects.get(identifier)
        if not found:
            raise ValueError(f'Incorrect object identifier: {identifier}')
        obj = copy.deepcopy(found)
        for key in attrs:
            setattr(obj, key, attrs[key])

        return obj
```

In the `main()` function, as shown in the following code, we can clone a first `Website` instance, `site1`, to get a second object' `site2`. Basically, we instantiate the `Prototype` class and we use its `.clone()` method. That is what the following code shows:

```
def main():
    keywords = ('python', 'data', 'apis', 'automation')
    site1 = Website('ContentGardening',
                    domain='contentgardening.com',
                    description='Automation and data-driven apps',
                    author='Kamon Ayeva',
                    category='Blog',
                    keywords=keywords)

    prototype = Prototype()
    identifier = 'ka-cg-1'
    prototype.register(identifier, site1)
    site2 = prototype.clone(identifier,
```

```
name='ContentGardeningPlayground',  
domain='play.contentgardening.com',  
description='Experimentation for techniques featured  
on the blog',  
category='Membership site',  
creation_date='2018-08-01')
```

To end that function, we can use the `id()` function which returns the memory address of an object, for comparing both objects' addresses, as follows. When we clone an object using a deep copy, the memory addresses of the clone must be different from the memory addresses of the original object:

```
for site in (site1, site2):  
    print(site)  
print(f'ID site1 : {id(site1)} != ID site2 : {id(site2)}')
```

You will find the program's full code in the `prototype.py` file. Here is a summary of what we do in the code:

1. We start by importing the `copy` module.
2. We define the `Website` class, with its initialization method (`__init__()`) and its string representation method (`__str__()`) as shown earlier.
3. We define our `Prototype` class as shown earlier.
4. Then, we have the `main()` function, where we do the following:
 - We define the `keywords` list we need
 - We create the instance of the `Website` class, called `site1` (we use the `keywords` list here)
 - We create the `Prototype` object and we use its `register()` method to register `site1` with its identifier (this helps us keep track of the cloned objects in a dictionary)
 - We clone the `site1` object to get `site2`
 - We display the result (both `Website` objects)

A sample output when I execute the `python prototype.py` command on my machine is as follows:


```
Website "ContentGardening"  
author: Kamon Ayeva  
category: Blog  
description: Automation and data-driven apps  
domain: contentgardening.com  
keywords: ('python', 'data', 'apis', 'automation')  
  
Website "ContentGardeningPlayground"  
author: Kamon Ayeva  
category: Membership site  
creation_date: 2018-08-01  
description: Experimentation for techniques featured on the blog  
domain: play.contentgardening.com  
keywords: ('python', 'data', 'apis', 'automation')  
  
ID site1 : 2209666079432 != ID site2 : 2209666114000
```

Indeed, `Prototype` works as expected. We can see the information about the original `Website` object and its clone.

Looking at the output of the `id()` function, we can see that the two addresses are different.

Singleton

The singleton pattern restricts the instantiation of a class to *one* object, which is useful when you need one object to coordinate actions for the system.

The basic idea is that only one instance of a particular class, doing a job, is created for the needs of the program. To ensure that this works, we need mechanisms that prevent the instantiation of the class more than once and also prevent cloning.

Real-world examples

In a real-life scenario, we can think of the captain of a ship or a boat. On the ship, he is the one in charge. He is responsible for important decisions, and a number of requests are directed to him because of this responsibility.

In software, the Plone CMS has, at its core, an implementation of the singleton. There are actually several singleton objects available at the root of a Plone site, called **tools**, each in charge of providing a specific set of features for the site. For example, the **Catalog tool** deals with content indexation and search features (built-in search engines for small sites where you don't need to integrate products like ElasticSearch), the **Membership tool** deals with things related to user profiles, and the **Registry tool** provides a configuration registry to store and maintain different kinds of configuration properties for the Plone site. Each tool is global to the site, created from a specific `singleton` class, and you can't create another instance of that `singleton` class in the context of the site.

Use cases

The singleton design pattern is useful when you need to create only one object or you need some sort of object capable of maintaining a global state for your program.

Other possible use cases are:

- Controlling concurrent access to a shared resource. For example, the class managing the connection to a database.
- A service or resource that is transversal in the sense that it can be accessed from different parts of the application or by different users and do its work. For example, the class at the core of the logging system or utility.

Implementation

Let's implement a program to fetch content from web pages, inspired by the tutorial from Michael Ford (<https://docs.python.org/3/howto/urllib2.html>). We have only take the simple part since the focus is to illustrate our pattern more than it is to build a special web-scraping tool.

We will use the `urllib` module to connect to web pages using their URLs; the core of the program would be the `URLFetcher` class that takes care of doing the work via a `fetch()` method.

We want to be able to track the list of web pages that were tracked, hence the use of the singleton pattern: we need a single object to maintain that global state.

First, our naive version, inspired by the tutorial but modified to help us track the list of URLs that were fetched, would be:

```
import urllib.parse
import urllib.request

class URLFetcher:

    def __init__(self):
        self.urls = []
    def fetch(self, url):
        req = urllib.request.Request(url)
        with urllib.request.urlopen(req) as response:
            if response.code == 200:
                the_page = response.read()
                print(the_page)
                urls = self.urls
                urls.append(url)
                self.urls = urls
```

As an exercise, add the usual `if __name__ == '__main__':` block with a few lines of code to call the `.fetch()` method on an instance of `URLFetcher`.

But then, does our class implement a singleton? Here is a clue. To create a singleton, we need to make sure one can only create one instance of it. So, to see if our class implements a singleton or not, we could use a trick which consists of comparing two instances using the `is` operator.

You may have guessed the second exercise. Put the following code in your `if __name__ == '__main__':` block instead of what you previously had:

```
f1 = URLFetcher()
f2 = URLFetcher()
print(f1 is f2)
```

As an alternative, use the concise but still elegant form:

```
print(URLFetcher() is URLFetcher())
```

With this change, when executing the program, you should get `False` as the output.

Okay! This means that the first try does not yet give us a singleton. Remember, we want to manage a global state, using one and only one instance of the class for the program. The current version of the class does not yet implement a singleton.

After checking the literature and the forums on the web, you will find that there are several techniques, each with pros and cons, and some are probably outdated.

Since many use Python 3 nowadays, the recommended technique we will choose is the **metaclass** technique. We first implement a metaclass for the singleton, meaning the class (or type) of the classes that implement the singleton pattern, as follows:

```
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]
```

Now, we will rewrite our `URLFetcher` class to use that metaclass. We also add a `dump_url_registry()` method, which is useful to get the current list of URLs tracked:

```
class URLFetcher(metaclass=SingletonType):

    def fetch(self, url):
        req = urllib.request.Request(url)
        with urllib.request.urlopen(req) as response:
            if response.code == 200:
                the_page = response.read()
                print(the_page)
                urls = self.urls
                urls.append(url)
                self.urls = urls

    def dump_url_registry(self):
        return ', '.join(self.urls)

if __name__ == '__main__':
    print(URLFetcher() is URLFetcher())
```

This time, you get `True` by executing the program.

Let's complete the program to do what we wanted, using a `main()` function that we will call:

```
def main():

    MY_URLS = ['http://www.voidspace.org.uk',
               'http://google.com',
               'http://python.org',
               'https://www.python.org/error',
               ]

    print(URLFetcher() is URLFetcher())

    fetcher = URLFetcher()
    for url in MY_URLS:
        try:
            fetcher.fetch(url)
        except Exception as e:
            print(e)
    print('-----')
    done_urls = fetcher.dump_url_registry()
    print(f'Done URLs: {done_urls}')
```

You will find the program's full code in the `singleton.py` file. Here is a summary of what we do:

1. We start with our needed module imports (`urllib.parse` and `urllib.request`)
2. As shown earlier, we define the `SingletonType` class, with its special `__call__()` method
3. As shown earlier, we define `URLFetcher`, the class implementing the fetcher for the web pages, initializing it with the `urls` attribute; as discussed, we add its `fetch()` and `dump_url_registry()` methods
4. Then, we add our `main()` function
5. Lastly, we add Python's conventional snippet used to call the `main` function

As seen in the implementation example we discussed, using a prototype in Python is natural and based on built-in features, so it is not something even mentioned.

The singleton pattern can be implemented by making the `singleton` class use a metaclass, its type, having previously defined the said metaclass. As required, the metaclass's `__call__()` method holds the code that ensures that only one instance of the class can be created.

The next chapter is about the adapter pattern, a structural design pattern that can be used to make two incompatible software interfaces compatible.

4

The Adapter Pattern

In the previous chapters, we have covered creational patterns, object-oriented programming patterns that help us with object creation procedures. The next category of patterns we want to present is *structural design patterns*.

A structural design pattern proposes a way of composing objects for creating new functionality. The first of these patterns we will cover is the *adapter* pattern.

The *adapter* pattern is a structural design pattern that helps us make two incompatible interfaces compatible. What does that really mean? If we have an old component and we want to use it in a new system, or a new component that we want to use in an old system, the two can rarely communicate without requiring any code changes. But, changing the code is not always possible, either because we don't have access to it, or because it is impractical. In such cases, we can write an extra layer that makes all the required modifications for enabling the communication between the two interfaces. This layer is called an **adapter**.

In general, if you want to use an interface that expects `function_a()`, but you only have `function_b()`, you can use an adapter to convert (adapt) `function_b()` to `function_a()`.

In this chapter, we will discuss the following:

- Real-world examples
- Use cases
- Implementation

Real-world examples

When you are traveling from most European countries to the UK or USA, or the other way around, you need to use a plug adapter for charging your laptop. The same kind of adapter is needed for connecting some devices to your computer: the USB adapter.

In the software category, the Zope application server (<http://www.zope.org>) is known for its **Zope Component Architecture (ZCA)**, which contributed an implementation of interfaces and adapters used by several big Python web projects. Pyramid, built by former Zope developers, is a Python web framework that took good ideas from Zope to provide a more modular approach for developing web apps. Pyramid uses adapters for making it possible for existing objects to conform to specific APIs without the need to modify them. Another project from the Zope ecosystem, *Plone CMS*, uses adapters under the hood.

Use cases

Usually, one of the two incompatible interfaces is either foreign or old/legacy. If the interface is foreign, it means that we have no access to the source code. If it is old, it is usually impractical to refactor it.

Using an adapter for making things work after they have been implemented is a good approach because it does not require access to the source code of the foreign interface. It is also often a pragmatic solution if we have to reuse some legacy code.

Implementation

Let's look at a relatively simple application to illustrate **adaptation**: a club's activities, mainly the need to organize performances and events for the entertainment of its clients, by hiring talented artists.

At the core, we have a `Club` class that represents the club where hired artists perform some evenings. The `organize_performance()` method is the main action that the club can perform. The code is as follows:

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

    def __str__(self):
        return f'the club {self.name}'

    def organize_event(self):
        return 'hires an artist to perform for the people'
```

Most of the time, our club hires a DJ to perform, but our application addresses the need to organize a diversity of performances, by a musician or music band, by a dancer, a one-man or one-woman show, and so on.

Via our research to try and reuse existing code, we find an open source contributed library that brings us two interesting classes: `Musician` and `Dancer`. In the `Musician` class, the main action is performed by the `play()` method. In the `Dancer` class, it is performed by the `dance()` method.

In our example, to indicate that these two classes are external, we place them in a separate module. The code is as follows for the `Musician` class:

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

    def __str__(self):
        return f'the musician {self.name}'

    def play(self):
        return 'plays music'
```

Then, the `Dancer` class is defined as follows:

```
class Dancer:
    def __init__(self, name):
        self.name = name
    def __str__(self):
        return f'the dancer {self.name}'
    def dance(self):
        return 'does a dance performance'
```

The client code, using these classes, only knows how to call the `organize_performance()` method (on the `Club` class); it has no idea about `play()` or `dance()` (on the respective classes from the external library).

How can we make the code work without changing the `Musician` and `Dancer` classes?

Adapters to the rescue! We create a generic `Adapter` class that allows us to adapt a number of objects with different interfaces, into one unified interface. The `obj` argument of the `__init__()` method is the object that we want to adapt, and `adapted_methods` is a dictionary containing key/value pairs matching the method the client calls and the method that should be called.

The code for the `Adapter` class is as follows:

```
class Adapter:
    def __init__(self, obj, adapted_methods):
        self.obj = obj
        self.__dict__.update(adapted_methods)
```

```
def __str__(self):
    return str(self.obj)
```

When dealing with the different instances of the classes, we have two cases:

- The compatible object that belongs to the `Club` class needs no adaptation. We can treat it as is.
- The incompatible objects need to be adapted first, using the `Adapter` class.

The result is that the client code can continue using the known `organize_performance()` method on all objects without the need to be aware of any interface differences between the used classes. Consider the following code:

```
def main():
    objects = [Club('Jazz Cafe'), Musician('Roy Ayers'), Dancer('Shane Sparks')]
    for obj in objects:
        if hasattr(obj, 'play') or hasattr(obj, 'dance'):
            if hasattr(obj, 'play'):
                adapted_methods = dict(organize_event=obj.play)
            elif hasattr(obj, 'dance'):
                adapted_methods = dict(organize_event=obj.dance)
            # referencing the adapted object here
            obj = Adapter(obj, adapted_methods)
        print(f'{obj} {obj.organize_event()}')
```

Let's recapitulate the complete code of our adapter pattern implementation:

1. We define the `Musician` and `Dancer` classes (in `external.py`).
2. Then, we need to import those classes from the external module (in `adapter.py`):

```
from external import Musician, Dance
```

3. We then define the `Adapter` class (in `adapter.py`).
4. We add the `main()` function, as shown earlier, and the usual trick to call it (in `adapter.py`).

Here is the output when executing the `python adapter.py` command, as usual:

```
the club Jazz Cafe hires an artist to perform for the people
the musician Roy Ayers plays music
the dancer Shane Sparks does a dance performance
```

As you can see, we managed to make the `Musician` and `Dancer` classes compatible with the interface expected by the client, without changing their source code.

Summary

This chapter covered the adapter design pattern. We use the adapter pattern for making two (or more) incompatible interfaces compatible. We use adapters every day for interconnecting devices, charging them, and so on.

The adapter makes things work after they have been implemented. The Pyramid web framework, the Plone CMS, and other Zope-based or related frameworks use the adapter pattern for achieving interface compatibility.

In the *Implementation* section, we saw how to achieve interface conformance using the adapter pattern without modifying the source code of the incompatible model. This is achieved through a generic `Adapter` class that does the work for us.

In the next chapter, we will cover the decorator pattern.

5

The Decorator Pattern

As we saw in the previous chapter, using an **adapter**, a first structural design pattern, you can adapt an object implementing a given interface to implement another interface. This is called **interface adaptation** and includes the kinds of patterns that encourage composition over inheritance, and it could bring benefits when you have to maintain a large codebase.

A second interesting structural pattern to learn about is the **decorator** pattern, which allows a programmer to add responsibilities to an object dynamically, and in a transparent manner (without affecting other objects).

There is another reason why this pattern is interesting to us, as you will see in a minute.

As Python developers, we can write decorators in a **Pythonic** way (meaning using the language's features), thanks to the built-in decorator feature (https://docs.python.org/3/reference/compound_stmts.html#function). What exactly is this feature? A Python decorator is a **callable** (function, method, or class) that gets a function object `func_in` as input, and returns another function object `func_out`. It is a commonly used technique for extending the behavior of a function, method, or class.

But, this feature should not be completely new to you. We have already seen how to use the built-in **property** decorator that makes a method appear as a variable in both [Chapter 1, The Factory Pattern](#), and [Chapter 2, The Builder Pattern](#). And there are several other useful built-in decorators in Python. In the *Implementation* section of this chapter, we will learn how to implement and use our own decorators.

Note that there is no one-to-one relationship between the decorator pattern and Python's decorator feature. Python decorators can actually do much more than the decorator pattern. One of the things they can be used for is to implement the decorator pattern ([j.mp/moinpydec](#)).

In this chapter, we will discuss:

- Real-world examples
- Use cases
- Implementation

Real-world examples

The decorator pattern is generally **used for extending the functionality of an object**. In everyday life, examples of such extensions are: adding a silencer to a gun, using different camera lenses, and so on.

In the Django Framework, which uses decorators a lot, we have the `View` decorators which can be used for (j.mp/djangodec) the following:

- Restricting access to views based on the HTTP request
- Controlling the caching behavior on specific views
- Controlling compression on a per-view basis
- Controlling caching based on specific HTTP request headers

Both the Pyramid Framework and the Zope application server also use decorators to achieve various goals:

- Registering a function as an event subscriber
- Protecting a method with a specific permission
- Implementing the adapter pattern

Use cases

The decorator pattern shines when used for implementing cross-cutting concerns (j.mp/wikicrosscut). Examples of cross-cutting concerns are as follows:

- Data validation
- Caching
- Logging

- Monitoring
- Debugging
- Business rules
- Encryption

In general, all parts of an application that are generic and can be applied to many other parts of it are considered to be cross-cutting concerns.

Another popular example of using the decorator pattern is **graphical user interface (GUI)** toolkits. In a GUI toolkit, we want to be able to add features such as borders, shadows, colors, and scrolling to individual components/widgets.

Implementation

Python decorators are generic and very powerful. You can find many examples of how they can be used at the decorator library of `python.org` (`j.mp/pydeclib`). In this section, we will see how we can implement a memoization decorator (`j.mp/memoi`). All recursive functions can benefit from memoization, so let's try a function `number_sum()` that returns the sum of the first n numbers. Note that this function is already available in the `math` module as `fsum()`, but let's pretend it is not.

First, let's look at the naive implementation (the `number_sum_naive.py` file):

```
def number_sum(n):
    '''Returns the sum of the first n numbers'''
    assert(n >= 0), 'n must be >= 0'
    if n == 0:
        return 0
    else:
        return n + number_sum(n-1)

if __name__ == '__main__':
    from timeit import Timer
    t = Timer('number_sum(30)', 'from __main__ import number_sum')
    print('Time: ', t.timeit())
```

A sample execution of this example shows how slow this implementation is. It takes 15 seconds to calculate the sum of the first 30 numbers. We get the following output when executing the `python number_sum_naive.py` command:

```
Time: 15.69870145995352
```

Let's see if using memoization can help us improve the performance number. In the following code, we use a `dict` for caching the already computed sums. We also change the parameter passed to the `number_sum()` function. We want to calculate the sum of the first 300 numbers instead of only the first 30.

Here is the new version of the code, using memoization:

```
sum_cache = {0:0}
def number_sum(n):
    '''Returns the sum of the first n numbers'''
    assert(n >= 0), 'n must be >= 0'
    if n in sum_cache:
        return sum_cache[n]
    res = n + number_sum(n-1)
    # Add the value to the cache
    sum_cache[n] = res
    return res
if __name__ == '__main__':
    from timeit import Timer
    t = Timer('number_sum(300)', 'from __main__ import number_sum')
    print('Time: ', t.timeit())
```

Executing the memoization-based code shows that performance improves dramatically, and is acceptable even for computing large values.

A sample execution, using `python number_sum.py`, is as follows:

```
Time: 0.5695815602065222
```

But there are already a few problems with this approach. While the performance is not an issue any longer, the code is not as clean as it is when not using memoization. And what happens if we decide to extend the code with more math functions and turn it into a module? We can think of several functions that would be useful for our module, for problems such as Pascal's triangle or the Fibonacci numbers suite algorithm.

So, if we wanted a function in the same module as `number_sum()`, for the Fibonacci numbers suite, using the same memoization technique, we would add code, as follows:

```
cache_fib = {0:0, 1:1}

def fibonacci(n):
    '''Returns the suite of Fibonacci numbers'''
    assert(n >= 0), 'n must be >= 0'
    if n in cache_fib:
        return cache_fib[n]
    res = fibonacci(n-1) + fibonacci(n-2)
    cache_fib[n] = res
    return res
```

Do you notice the problem already? We ended up with a new dict called `cache_fib` which acts as our cache for the `fibonacci()` function, and a function that is more complex than it would be without using memoization. Our module is becoming unnecessarily complex. Is it possible to write these functions keeping them as simple as the naive versions, but achieving a performance similar to the performance of the functions that use memoization?

Fortunately, it is, and the solution is to use the decorator pattern.

First, we create a `memoize()` decorator as shown in the following example. Our decorator accepts the function `fn` that needs to be memoized, as an input. It uses a dict named `cache` as the cached data container. The `functools.wraps()` function is used for convenience when creating decorators. It is not mandatory but a good practice to use it, since it makes sure that the documentation, and the signature of the function that is decorated, are preserved ([j.mp/funcwraps](#)). The argument list `*args` is required in this case because the functions that we want to decorate accepts input arguments (such as the `n` argument for our two functions):

```
import functools

def memoize(fn):
    cache = dict()

    @functools.wraps(fn)
    def memoizer(*args):
        if args not in cache:
            cache[args] = fn(*args)
        return cache[args]

    return memoizer
```

Now, we can use our `memoize()` decorator with the naive version of our functions. This has the benefit of readable code without performance impact. We apply a decorator using what is known as decoration (or a decoration line). A decoration uses the `@name` syntax, where `name` is the name of the decorator that we want to use. It is nothing more than syntactic sugar for simplifying the usage of decorators. We can even bypass this syntax and execute our decorator manually, but that is left as an exercise for you.

So the `memoize()` decorator can be used with our recursive functions as follows:

```
@memoize
def number_sum(n):
    '''Returns the sum of the first n numbers'''
    assert(n >= 0), 'n must be >= 0'
    if n == 0:
        return 0
    else:
        return n + number_sum(n-1)

@memoize
def fibonacci(n):
    '''Returns the suite of Fibonacci numbers'''
    assert(n >= 0), 'n must be >= 0'
    if n in (0, 1):
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
```

In the last part of the code, via the `main()` function, we show how to use the decorated functions and measure their performance. The `to_execute` variable is used to hold a list of tuples containing the reference to each function and the corresponding `timeit.Timer()` call (to execute it while measuring the time spent), thus avoiding code repetition. Note how the `__name__` and `__doc__` method attributes show the proper function names and documentation values, respectively. Try removing the `@functools.wraps(fn)` decoration from `memoize()`, and see if this is still the case.

Here is the last part of the code:

```
def main():
    from timeit import Timer

    to_execute = [
        (number_sum,
         Timer('number_sum(300)', 'from __main__ import number_sum')),
        (fibonacci,
         Timer('fibonacci(100)', 'from __main__ import fibonacci'))
    ]
```

```

for item in to_execute:
    fn = item[0]
    print(f'Function "{fn.__name__}": {fn.__doc__}')
    t = item[1]
    print(f'Time: {t.timeit()}')
    print()

if __name__ == '__main__':
    main()

```

Let's recapitulate how we write the complete code of our math module (file `mymath.py`):

1. After the import of Python's `functools` module that we will be using, we define the `memoize()` decorator function
2. Then, we define the `number_sum()` function, decorated using `memoize()`
3. We also define the `fibonacci()` function, as decorated
4. Finally, we add the `main()` function, as shown earlier, and the usual trick to call it

Here is a sample output when executing the `python mymath.py` command:

```

Function "number_sum": Returns the sum of the first n numbers
Time: 0.6511659908041739

Function "fibonacci": Returns the suite of Fibonacci numbers
Time: 0.6524761144050873

```



The execution times might differ in your case.

Nice. We ended up with readable code and acceptable performance. Now, you might argue that this is not the decorator pattern, since we don't apply it at runtime. The truth is that a decorated function cannot be undecorated, but you can still decide at runtime if the decorator will be executed or not. That's an interesting exercise left for you.



Use a decorator that acts as a wrapper, which decides whether or not the real decorator is executed based on some condition.

Another interesting property of decorators that is not covered in this chapter is that you can decorate a function with more than one decorator. So here's another exercise: create a decorator that helps you to debug recursive functions, and apply it on `number_sum()` and `fibonacci()`. In what order are the multiple decorators executed?

Summary

This chapter covered the decorator pattern and its relationship to the Python programming language. We use the decorator pattern as a convenient way of extending the behavior of an object without using inheritance. Python, with its built-in decorator feature, extends the decorator concept even more, by allowing us to extend the behavior of any callable (function, method, or class) without using inheritance or composition.

We have seen a few examples of real-world objects that are decorated, such as cameras. From a software point of view, both Django and Pyramid use decorators to achieve different goals, such as controlling HTTP compression and caching.

The decorator pattern is a great solution for implementing cross-cutting concerns because they are generic and do not fit well into the OOP paradigm. We mentioned several categories of cross-cutting concerns in the *Use cases* section. In fact, in the *Implementation* section, a cross-cutting concern was demonstrated: memoization. We saw how decorators can help us to keep our functions clean, without sacrificing performance.

The next chapter covers the bridge pattern.

6

The Bridge Pattern

In the previous two chapters, we covered our first structural pattern, *adapter*, which is used to make two incompatible interfaces compatible, and *decorator*, which allows us to add responsibilities to an object in a dynamic way. There are more similar patterns. Let's continue with the series!

A third structural pattern to look at is the *bridge* pattern. We can actually compare the *bridge* and the *adapter* patterns, looking at the way both work. While *adapter* is used later to make unrelated classes work together, as we saw in the implementation example we discussed in Chapter 4, *The Adapter Pattern*, the *bridge* pattern is designed up-front to decouple an implementation from its abstraction, as we are going to see.

In this chapter, we will discuss:

- Real-world examples
- Use cases
- Implementation

Real-world examples

In our *modern*, everyday lives, an example of the bridge pattern I can think of is from the *digital economy*: information products. Nowadays, the information product or *infoproduct* is part of the resources one can find online for training, self-improvement, or one's ideas and business development. The purpose of an information product that you find on certain marketplaces, or the website of the provider, is to deliver information on a given topic in such a way that it is easy to access and consume. The provided material can be a PDF document or ebook, an ebook series, a video, a video series, an online course, a subscription-based newsletter, or a combination of all those formats.

In the software realm, *device drivers* are often cited as an example of the bridge pattern, when the developers of an OS defines the interface for device vendors to implement it.

Use cases

Using the bridge pattern is a good idea when you want to share an implementation among multiple objects. Basically, instead of implementing several specialized classes, defining all that is required within each class, you can define the following special components:

- An abstraction that applies to all the classes
- A separate interface for the different objects involved

An implementation example we are about to see will illustrate this approach.

Implementation

Let's assume we are building an application where the user is going to manage and deliver content after fetching it from diverse sources, which could be:

- A web page (based on its URL)
- A resource accessed on an FTP server
- A file on the local file system
- A database server

So, here is the idea: instead of implementing several content classes, each holding the methods responsible for getting the content pieces, assembling them, and showing them inside the application, we can define an abstraction for the *Resource Content* and a separate interface for the objects that are responsible for fetching the content. Let's try it!

We begin with the class for our *Resource Content* abstraction, called `ResourceContent`. Then, we will need to define the interface for implementation classes that help fetch content, that is, the `ResourceContentFetcher` class. This concept is called **the Implementor**.

The first trick we use here is that, via an attribute (`_imp`) on the `ResourceContent` class, we maintain a reference to the object which represents the *Implementor*:

```
class ResourceContent:
    """
    Define the abstraction's interface.
```

```
Maintain a reference to an object which represents the Implementor.  
"""
```

```
def __init__(self, imp):  
    self._imp = imp  
  
def show_content(self, path):  
    self._imp.fetch(path)
```

As you may know now, we define the equivalent of an interface in Python using two features of the language, the metaclass feature (which helps define the *type of a type*), and **abstract base classes (ABC)**:

```
class ResourceContentFetcher(metaclass=abc.ABCMeta):  
    """  
    Define the interface for implementation classes that fetch content.  
    """  
    @abc.abstractmethod  
    def fetch(path):  
        pass
```

Now, we can add an implementation class to fetch content from a web page or resource:

```
class URLFetcher(ResourceContentFetcher):  
    """  
    Implement the Implementor interface and define its concrete  
    implementation.  
    """  
    def fetch(self, path):  
        # path is an URL  
        req = urllib.request.Request(path)  
        with urllib.request.urlopen(req) as response:  
            if response.code == 200:  
                the_page = response.read()  
                print(the_page)
```

We can also add an implementation class to fetch content from a file on the local filesystem:

```
class LocalFileFetcher(ResourceContentFetcher):
    """
    Implement the Implementor interface and define its concrete
    implementation.
    """

    def fetch(self, path):
        # path is the filepath to a text file
        with open(path) as f:
            print(r.read())
```

Based on that, our main function to show content using both *content fetchers* could look like the following:

```
def main():
    url_fetcher = URLFetcher()
    iface = ResourceContent(url_fetcher)
    iface.show_content('http://python.org')

    print('=====')
    localfs_fetcher = LocalFileFetcher()
    iface = ResourceContent(localfs_fetcher)
    iface.show_content('file.txt')
```

Let's see a summary for the complete code of our example (the `bridge.py` file):

1. We import the three modules we need for the program (`abc`, `urllib.parse`, and `urllib.request`).
2. We define the `ResourceContent` class for the interface of the *abstraction*.
3. We define the `ResourceContentFetcher` class for the *Implementator*.
4. We define two implementation classes:
 - `URLFetcher` for fetching content from an URL
 - `LocalFileFetcher` for fetching content from the local filesystem
 - Finally, we add the `main()` function, as shown earlier, and the usual trick to call it

Here is a sample output when executing the `python bridge.py` command:


```

et Python">Community News</a></li>\n      \n      <li class="tier-2 element-3" role="treeitem"><a href="http://pyfound.blog
\n      \n      <li class="tier-2 element-4" role="treeitem"><a href="http://pycon.blogspot.com/" title="PyCon Blog">PyCon
/li>\n      \n      <li class="tier-1 element-7">\n          <a href="/events/" >Events</a>\n          \n          \n<ul class=
-2 element-1" role="treeitem"><a href="/events/python-events" title="">Python Events</a></li>\n      \n      <li class="tie
s/python-user-group/" title="">User Group Events</a></li>\n      \n      <li class="tier-2 element-3" role="treeitem"><a h
thon Events Archive</a></li>\n      \n      <li class="tier-2 element-4" role="treeitem"><a href="/events/python-user-group
">/li>\n      \n      <li class="tier-2 element-5" role="treeitem"><a href="https://wiki.python.org/moin/PythonEventsCalend
ent</a></li>\n      \n</ul>\n      \n      </li>\n      \n      <li class="tier-1 element-8">\n          <a href="/dev/" >Contrib
s="subnav menu">\n      \n      <li class="tier-2 element-1" role="treeitem"><a href="https://devguide.python.org/" title="
<li class="tier-2 element-2" role="treeitem"><a href="https://bugs.python.org/" title="">Issue Tracker</a></li>\n      \n      <li c
eitem"><a href="https://mail.python.org/mailman/listinfo/python-dev" title="">python-dev list</a></li>\n      \n      <li c
ef="dev/core-mentorship/" title="">Core Mentorship</a></li>\n      \n</ul>\n      \n</li>\n      \n</ul>\n      \n</li>
-link" href="#python-network"><span aria-hidden="true" class="icon-arrow-up"></span></span></span> Back to Top</a>\n
<!-- end .container -->\n      </div><!-- end .main-footer-links -->\n      <div class="site-base">\n
      \n      <ul class="footer-links navigation menu do-not-print" role="tree">\n
help/">Help &amp; <span class="say-no-more">General</span> Contact</a></li>\n      <li class="tier-1 elem
ity <span class="say-no-more">Initiatives</span></a></li>\n      <li class="tier-1 element-3"><a href="ht
Submit Website Bug</a></li>\n      <li class="tier-1 element-4">\n          <a href="ht
"python-status-indicator-default" id="python-status-indicator"></span></a>\n      </li>\n      <li class="pre">copyright">\n      <p><small>\n      <span class="pre"><a href="/psf-landing/">Python Software Foundation</a></span>\n      <span class="pre">Copyright &copy;2001-2018.</
n class="pre"><a href=""/>Privacy Policy</a></span>\n      <span class=
s</a></span>\n      &nbsp;<span class="pre"><a href="/privacy/">Privacy Policy</a></span>\n      </small></p>\n
href="/psf/sponsorship/sponsors/">Powered by Rackspace</a></span>\n      </small></p>\n
ontainer -->\n      </div><!-- end .site-base -->\n      \n      </footer>\n      \n      </div><!-- end #touchnav-wrapper -->\n\
/ajax/libs/jquery/1.8.2/jquery.min.js"></script>\n      <script>window.jQuery || document.write('<script src="/static/js/lib
">\n      <script src="/static/js/libs/masonry.pkgd.min.js"></script>\n      <script type="text/javascript" src="/static/js/
n      <!--[if lte IE 7]>\n      <script type="text/javascript" src="/static/js/plugins/IE8-min.js" charset="utf-8"></script>
lte IE 8]>\n      <script type="text/javascript" src="/static/js/plugins/getComputedStyle-min.js" charset="utf-8"></script>\
\n      \n</body></html>\n'

```

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Proin in nibh in enim euismod mattis placerat in velit. Donec mal
 nident porttitor euismod. Etiam non odio sodales, tincidunt elit ac, sodales nisi. Donec massa felis, pharetra ut libero n
 sem non erat ultricies finibus. Donec sed blandit arcu. Aliquam erat volutpat. Donec aliquam ipsum risus, et accumsan nibh
 tae rutrum. Sed ullamcorper leo sed orci efficitur rhoncus.

Duis vitae dolor vestibulum nibh semper faucibus. Vivamus libero quam, ultrices quis sapien vel, blandit ultricies purus. N
 non ligula. Duis ullamcorper, nulla quis luctus commodo, massa lorem tristique orci, quis aliquam diam est semper nisi. Ma
 n convallis tellus aaculis. Fusce quis purus nibh. Nulla tempor est vel metus sodales, in dapibus risus molestie. Donec tri
 ue ut vehicula mauris. Vivamus pellentesque, tellus in dictum vehicula, justo ex volutpat sem, at cursus nisl elit non ex.
 rnare vitae mi a vestibulum. Suspendisse potenti. Donec sed ligula ac enim mattis posuere.

This is a basic illustration of how, using the bridge pattern in your design, you can extract content from different sources and integrate the results in the same data manipulation system or user interface.

Summary

In this chapter, we discussed the bridge pattern. Sharing similarities with the adapter pattern, the bridge pattern is different from it, in the sense that it is used up-front to define an abstraction and its implementation in a decoupled way so that both can vary independently.

The bridge pattern is useful when writing software for problem domains such as operation systems and device drivers, GUIs, and website builders where we have multiple themes and we need to change the theme of a website based on certain properties.

To help you understand this pattern, we discussed an example in the domain of content extraction and management, where we defined an interface for the abstraction, an interface for the implementor, and two implementations.

In the next chapter, we are going to cover the façade pattern.

7

The Facade Pattern

In the previous chapter, we covered a third structural pattern, the bridge pattern, which helps to define an abstraction and its implementation in a decoupled way, so that both can vary independently.

As systems evolve, they can get very complex. It is not unusual to end up with a very large (and sometimes confusing) collection of classes and interactions. In many cases, we don't want to expose this complexity to the client. This is where our next structural pattern comes to the rescue: **façade**.

The façade design pattern helps us to hide the internal complexity of our systems and expose only what is necessary to the client through a simplified interface. In essence, façade is an abstraction layer implemented over an existing complex system.

Let's take the example of the computer to illustrate things. A computer is a complex machine that depends on several parts to be fully functional. To keep things simple, the word "computer", in this case, refers to an IBM derivative that uses a von Neumann architecture. Booting a computer is a particularly complex procedure. The CPU, main memory, and hard disk need to be up and running, the boot loader must be loaded from the hard disk to the main memory, the CPU must boot the operating system kernel, and so forth. Instead of exposing all this complexity to the client, we create a façade that encapsulates the whole procedure, making sure that all steps are executed in the right order.

In terms of object design and programming, we should have several classes, but only the `Computer` class needs to be exposed to the client code. The client will only have to execute the `start()` method of the `Computer` class, for example, and all the other complex parts are taken care of by the façade `Computer` class.

In this chapter, we will discuss:

- Real-world examples
- Use cases
- Implementation

Real-world examples

The façade pattern is quite common in life. When you call a bank or a company, you are usually first connected to the customer service department. The customer service employee acts as a façade between you and the actual department (billing, technical support, general assistance, and so on), and the employee that will help you with your specific problem.

As another example, a key used to turn on a car or motorcycle can also be considered a façade. It is a simple way of activating a system that is very complex internally. And, of course, the same is true for other complex electronic devices that we can activate with a single button, such as computers.

In software, the `django-oscar-datacash` module is a Django third-party module that integrates with the **DataCash** payment gateway. The module has a gateway class that provides fine-grained access to the various DataCash APIs. On top of that, it also offers a façade class that provides a less granular API (for those who don't want to mess with the details), and the ability to save transactions for auditing purposes.

Use cases

The most usual reason to use the façade pattern is for providing a single, simple entry point to a complex system. By introducing façade, the client code can use a system by simply calling a single method/function. At the same time, the internal system does not lose any functionality, it just encapsulates it.

Not exposing the internal functionality of a system to the client code gives us an extra benefit: we can introduce changes to the system, but the client code remains unaware of and unaffected by the changes. No modifications are required to the client code.

Façade is also useful if you have more than one layer in your system. You can introduce one façade entry point per layer, and let all layers communicate with each other through their façades. That promotes **loose coupling** and keeps the layers as independent as possible.

Implementation

Assume that we want to create an operating system using a multi-server approach, similar to how it is done in MINIX 3 (j.mp/minix3) or GNU Hurd (j.mp/gnuhurd). A multiserver operating system has a minimal kernel, called the **microkernel**, which runs in privileged mode. All the other services of the system are following a server architecture (driver server, process server, file server, and so forth). Each server belongs to a different memory address space and runs on top of the microkernel in user mode. The pros of this approach are that the operating system can become more fault-tolerant, reliable, and secure. For example, since all drivers are running in user mode on a driver server, a bug in a driver cannot crash the whole system, nor can it affect the other servers. The cons of this approach are the performance overhead and the complexity of system programming, because the communication between a server and the microkernel, as well as between the independent servers, happens using message passing. Message passing is more complex than the shared memory model used in monolithic kernels such as Linux (j.mp/helenosm).

We begin with a `Server` interface. An `Enum` parameter describes the different possible states of a server. We use the `ABC` module to forbid direct instantiation of the `Server` interface and make the fundamental `boot()` and `kill()` methods mandatory, assuming that different actions are needed to be taken for booting, killing, and restarting each server. If you have not used the `ABC` module before, note the following important things:

- We need to subclass `ABCMeta` using the `metaclass` keyword.
- We use the `@abstractmethod` decorator for stating which methods should be implemented (mandatory) by all subclasses of `server`.

Try removing the `boot()` or `kill()` method of a subclass and see what happens. Do the same after removing the `@abstractmethod` decorator also. Do things work as you expected?

Let's consider the following code:

```
State = Enum('State', 'new running sleeping restart zombie')
class Server(metaclass=ABCMeta):
    @abstractmethod
    def __init__(self):
        pass
    def __str__(self):
        return self.name
    @abstractmethod
    def boot(self):
        pass
    @abstractmethod
```

```
def kill(self, restart=True):  
    pass
```

A modular operating system can have a great number of interesting servers: a file server, a process server, an authentication server, a network server, a graphical/window server, and so forth. The following example includes two stub servers: the `FileServer` and the `ProcessServer`. Apart from the methods required to be implemented by the `Server` interface, each server can have its own specific methods. For instance, the `FileServer` has a `create_file()` method for creating files, and the `ProcessServer` has a `create_process()` method for creating processes.

The `FileServer` class is as follows:

```
class FileServer(Server):  
    def __init__(self):  
        '''actions required for initializing the file server'''  
        self.name = 'FileServer'  
        self.state = State.new  
  
    def boot(self):  
        print(f'booting the {self}')        '''actions required for booting the file server'''  
        self.state = State.running  
  
    def kill(self, restart=True):  
        print(f'Killing {self}')        '''actions required for killing the file server'''  
        self.state = State.restart if restart else State.zombie  
  
    def create_file(self, user, name, permissions):  
        '''check validity of permissions, user rights, etc.'''  
        print(f"trying to create the file '{name}' for user '{user}' with  
permissions {permissions}")
```

The `ProcessServer` class is as follows:

```
class ProcessServer(Server):  
    def __init__(self):  
        '''actions required for initializing the process server'''  
        self.name = 'ProcessServer'  
        self.state = State.new  
  
    def boot(self):  
        print(f'booting the {self}')        '''actions required for booting the process server'''  
        self.state = State.running
```

```

def kill(self, restart=True):
    print(f'Killing {self}')
    '''actions required for killing the process server'''
    self.state = State.restart if restart else State.zombie

def create_process(self, user, name):
    '''check user rights, generate PID, etc.'''
    print(f"trying to create the process '{name}' for user '{user}'")

```

The `OperatingSystem` class is a façade. In its `__init__()`, all the necessary server instances are created. The `start()` method, used by the client code, is the entry point to the system. More wrapper methods can be added, if necessary, as access points to the services of the servers, such as the wrappers, `create_file()` and `create_process()`. From the client's point of view, all those services are provided by the `OperatingSystem` class. The client should not be confused by unnecessary details such as the existence of servers and the responsibility of each server.

The code for the `OperatingSystem` class is as follows:

```

class OperatingSystem:
    '''The Facade'''
    def __init__(self):
        self.fs = FileServer()
        self.ps = ProcessServer()

    def start(self):
        [i.boot() for i in (self.fs, self.ps)]

    def create_file(self, user, name, permissions):
        return self.fs.create_file(user, name, permissions)

    def create_process(self, user, name):
        return self.ps.create_process(user, name)

```

As you are going to see in a minute, when we present a summary of the example, there are many dummy classes and servers. They are there to give you an idea about the required abstractions (`User`, `Process`, `File`, and so forth) and servers (`WindowServer`, `NetworkServer`, and so forth) for making the system functional. A recommended exercise is to implement at least one service of the system (for example, file creation). Feel free to change the interface and the signature of the methods to fit your needs. Make sure that after your changes, the client code does not need to know anything other than the façade `OperatingSystem` class.

We are going to recapitulate the details of our implementation example; the full code is in the `facade.py` file:

1. We start with the imports we need:

```
from enum import Enum
from abc import ABCMeta, abstractmethod
```

2. We define the `State` constant using `Enum`, as shown earlier.
3. We then add the `User`, `Process`, and `File` classes, which do nothing in this minimal but functional example:

```
class User:
    pass

class Process:
    pass

class File:
    pass
```

4. We define the base `Server` class, as shown earlier.
5. We then define the `FileServer` class and the `ProcessServer` class, which are both subclasses of `Server`.
6. We add two other dummy classes, `WindowServer` and `NetworkServer`:

```
class WindowServer:
    pass

class NetworkServer:
    pass
```

7. Then we define our façade class, `OperatingSystem`, as shown earlier.
8. Finally, here is the main part of the code, where we use the façade we have defined:

```
def main():
    os = OperatingSystem()
    os.start()
    os.create_file('foo', 'hello', '-rw-r-r')
    os.create_process('bar', 'ls /tmp')

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


As you can see, executing the `python facade.py` command shows the starting message of our two stub servers:

```
booting the FileServer
booting the ProcessServer
trying to create the file 'hello' for user 'foo' with permissions -rw-r--r
trying to create the process 'ls /tmp' for user 'bar'
```

The `façade OperatingSystem` class does a good job. The client code can create files and processes without needing to know internal details about the operating system, such as the existence of multiple servers. To be precise, the client code can call the methods for creating files and processes, but they are currently dummy. As an interesting exercise, you can implement one of the two methods, or even both.

Summary

In this chapter, we have learned how to use the façade pattern. This pattern is ideal for providing a simple interface to client code that wants to use a complex system but does not need to be aware of the system's complexity. A computer is a façade since all we need to do to use it is press a single button for turning it on. All the rest of the hardware complexity is handled transparently by the BIOS, the boot loader, and the other components of the system software. There are more real-life examples of façade, such as when we are connected to the customer service department of a bank or a company, and the keys that we use to turn a vehicle on.

We discussed a Django third-party module that uses Façade: `django-oscar-datacash`. It uses the façade pattern to provide a simple DataCash API and the ability to save transactions.

We covered the basic use cases of façade and ended the chapter with an implementation of the interface used by a multiserver operating system. A façade is an elegant way of hiding the complexity of a system because, in most cases, the client code should not be aware of such details.

In the next chapter, we will cover other structural design patterns.

Other Structural Patterns

Besides the patterns covered in previous chapters, there are other structural patterns we can cover: **flyweight**, **model-view-controller (MVC)**, and **proxy**.

What is the flyweight pattern? Object-oriented systems can face performance issues due to the overhead of object creation. Performance issues usually appear in embedded systems with limited resources, such as smartphones and tablets. They can also appear in large and complex systems where we need to create a very large number of objects (and possibly users) that need to coexist at the same time. The *flyweight* pattern teaches programmers how to minimize memory usage by sharing resources with similar objects as much as possible.

The MVC pattern is useful mainly in application development and helps developers improve the maintainability of their applications by avoiding mixing the business logic with the user interface.

In some applications, we want to execute one or more important actions before accessing an object, and this is where the proxy pattern comes in. An example is the accessing of sensitive information. Before allowing any user to access sensitive information, we want to make sure that the user has sufficient privileges. The important action is not necessarily related to security issues. Lazy initialization (j.mp/wikilazy) is another case; we want to delay the creation of a computationally expensive object until the first time the user actually needs to use it. The idea of the proxy pattern is to help with performing such an action before accessing the actual object.

In this chapter, we will discuss:

- The flyweight pattern
- The MVC pattern
- The proxy pattern

The flyweight pattern

Whenever we create a new object, extra memory needs to be allocated. Although virtual memory provides us, theoretically, with unlimited memory, the reality is different. If all the physical memory of a system gets exhausted, it will start swapping pages with the secondary storage, usually a **hard disk drive (HDD)**, which, in most cases, is unacceptable due to the performance differences between the main memory and HDD. **Solid-state drives (SSDs)** generally have better performance than HDDs, but not everybody is expected to use SSDs. So, SSDs are not going to totally replace HDDs anytime soon.

Apart from memory usage, performance is also a consideration. Graphics software, including computer games, should be able to render 3-D information (for example, a forest with thousands of trees, a village full of soldiers, or an urban area with a lot of cars) extremely quickly. If each object in a 3-D terrain is created individually and no data sharing is used, the performance will be prohibitive.

As software engineers, we should solve software problems by writing better software, instead of forcing the customer to buy extra or better hardware. The flyweight design pattern is a technique used to minimize memory usage and improve performance by introducing data sharing between similar objects (j.mp/wflyw). A flyweight is a shared object that contains state-independent, immutable (also known as **intrinsic**) data. The state-dependent, mutable (also known as **extrinsic**) data should not be part of flyweight because this is information that cannot be shared, since it differs per object. If flyweight needs extrinsic data, it should be provided explicitly by the client code.

An example might help to clarify how the flyweight pattern can be practically used. Let's assume that we are creating a performance-critical game, for example, a **first-person shooter (FPS)**. In FPS games, the players (soldiers) share some states, such as representation and behavior. In *Counter-Strike*, for instance, all soldiers on the same team (counter-terrorists versus terrorists) look the same (representation). In the same game, all soldiers (on both teams) have some common actions, such as jump, duck, and so forth (behavior). This means that we can create a flyweight that will contain all of the common data. Of course, the soldiers also have a lot of data that is different per soldier and will not be a part of the flyweight, such as weapons, health, location, and so on.

Real-world examples

Flyweight is an optimization design pattern, therefore, it is not easy to find a good noncomputing example of it. We can think of flyweight as caching in real life. For example, many bookstores have dedicated shelves with the newest and most popular publications. This is a cache. First, you can take a look at the dedicated shelves for the book you are looking for, and if you cannot find it, you can ask the bookseller to assist you.

The Exaile music player uses flyweight to reuse objects (in this case, music tracks) that are identified by the same URL. There's no point in creating a new object if it has the same URL as an existing object, so the same object is reused to save resources.

Peppy, a XEmacs-like editor implemented in Python, uses the flyweight pattern to store the state of a major mode status bar. That's because unless modified by the user, all status bars share the same properties.

Use cases

Flyweight is all about improving performance and memory usage. All embedded systems (phones, tablets, games consoles, microcontrollers, and so forth) and performance-critical applications (games, 3-D graphics processing, real-time systems, and so forth) can benefit from it.

The *Gang of Four* (GoF) book lists the following requirements that need to be satisfied to effectively use the flyweight pattern:

- The application needs to use a large number of objects.
- There are so many objects that it's too expensive to store/render them. Once the mutable state is removed (because if it is required, it should be passed explicitly to flyweight by the client code), many groups of distinct objects can be replaced by relatively few shared objects.
- Object identity is not important for the application. We cannot rely on object identity because object sharing causes identity comparisons to fail (objects that appear different to the client code end up having the same identity).

Implementation

Let's see how we can implement the example mentioned previously for cars in an area. We will create a small car park to illustrate the idea, making sure that the whole output is readable in a single terminal page. However, no matter how large you make the car park, the memory allocation stays the same.

Before diving into the code, let's spend a moment noting the differences between the memoization and the flyweight pattern. **Memoization** is an optimization technique that uses a cache to avoid recomputing results that were already computed in an earlier execution step. Memoization does not focus on a specific programming paradigm such as **object-oriented programming (OOP)**. In Python, memoization can be applied to both methods and simple functions. Flyweight is an OOP-specific optimization design pattern that focuses on sharing object data.

First, we need an Enum parameter that describes the three different types of car that are in the car park:

```
CarType = Enum('CarType', 'subcompact compact suv')
```

Then, we will define the class at the core of our implementation: Car. The pool variable is the object pool (in other words, our cache). Notice that pool is a class attribute (a variable shared by all instances).

Using the __new__() special method, which is called before __init__(), we are converting the Car class to a metaclass that supports self-references. This means that cls references the Car class. When the client code creates an instance of Car, they pass the type of the car as car_type. The type of the car is used to check if a car of the same type has already been created. If that's the case, the previously created object is returned; otherwise, the new car type is added to the pool and returned:

```
class Car:
    pool = dict()

    def __new__(cls, car_type):
        obj = cls.pool.get(car_type, None)
        if not obj:
            obj = object.__new__(cls)
            cls.pool[car_type] = obj
            obj.car_type = car_type
        return obj
```

The `render()` method is what will be used to render a car on the screen. Notice how all the mutable information not known by flyweight needs to be explicitly passed by the client code. In this case, a random `color` and the coordinates of a location (of form `x, y`) are used for each car.

Also, note that to make `render()` more useful, it is necessary to ensure that no cars are rendered on top of each other. Consider this as an exercise. If you want to make rendering more fun, you can use a graphics toolkit such as Tkinter, Pygame, or Kivy.

The `render()` method is defined as follows:

```
def render(self, color, x, y):
    type = self.car_type
    msg = f'render a car of type {type} and color {color} at ({x},
{y})'
    print(msg)
```

The `main()` function shows how we can use the flyweight pattern. The color of a car is a random value from a predefined list of colors. The coordinates use random values between 1 and 100. Although 18 cars are rendered, memory is allocated only for three. The last line of the output proves that when using flyweight, we cannot rely on object identity. The `id()` function returns the memory address of an object. This is not the default behavior in Python because by default, `id()` returns a unique ID (actually the memory address of an object as an integer) for each object. In our case, even if two objects appear to be different, they actually have the same identity if they belong to the same **flyweight family** (in this case, the family is defined by `car_type`). Of course, different identity comparisons can still be used for objects of different families, but that is possible only if the client knows the implementation details.

Our example `main()` function's code is as follows:

```
def main():
    rnd = random.Random()
    colors = 'white black silver gray red blue brown beige yellow
green'.split()
    min_point, max_point = 0, 100
    car_counter = 0

    for _ in range(10):
        c1 = Car(CarType.subcompact)
        c1.render(random.choice(colors),
                    rnd.randint(min_point, max_point),
                    rnd.randint(min_point, max_point))
        car_counter += 1
```

```

for _ in range(3):
    c2 = Car(CarType.compact)
    c2.render(random.choice(colors),
               rnd.randint(min_point, max_point),
               rnd.randint(min_point, max_point))
    car_counter += 1

for _ in range(5):
    c3 = Car(CarType.suv)
    c3.render(random.choice(colors),
               rnd.randint(min_point, max_point),
               rnd.randint(min_point, max_point))
    car_counter += 1

print(f'cars rendered: {car_counter}')
print(f'cars actually created: {len(Car.pool)}')

c4 = Car(CarType.subcompact)
c5 = Car(CarType.subcompact)
c6 = Car(CarType.suv)
print(f'{id(c4)} == {id(c5)}? {id(c4) == id(c5)}')
print(f'{id(c5)} == {id(c6)}? {id(c5) == id(c6)}')

```

Here is the full code listing (the `flyweight.py` file) to show you how the flyweight pattern is implemented and used:

1. We need a couple of imports:

```

import random
from enum import Enum

```

2. The Enum for the types of cars is shown here:

```

CarType = Enum('CarType', 'subcompact compact suv')

```

3. Then we have the `Car` class, with its `pool` attribute and the `__new__()` and `render()` methods:

```

class Car:
    pool = dict()

    def __new__(cls, car_type):
        obj = cls.pool.get(car_type, None)
        if not obj:
            obj = object.__new__(cls)
            cls.pool[car_type] = obj
            obj.car_type = car_type
        return obj

```

```

def render(self, color, x, y):
    type = self.car_type
    msg = f'render a car of type {type} and color {color} at
({x}, {y})'
    print(msg)

```

4. In the first part of the main function, we define some variables and render a set of cars of type subcompact:

```

def main():
    rnd = random.Random()
    colors = 'white black silver gray red blue brown beige yellow
green'.split()
    min_point, max_point = 0, 100
    car_counter = 0

    for _ in range(10):
        c1 = Car(CarType.subcompact)
        c1.render(random.choice(colors),
                   rnd.randint(min_point, max_point),
                   rnd.randint(min_point, max_point))
        car_counter += 1

```

5. The second part of the main function is as follows:

```

for _ in range(3):
    c2 = Car(CarType.compact)
    c2.render(random.choice(colors),
               rnd.randint(min_point, max_point),
               rnd.randint(min_point, max_point))
    car_counter += 1

```

6. The third part of the main function is as follows:

```

for _ in range(5):
    c3 = Car(CarType.suv)
    c3.render(random.choice(colors),
               rnd.randint(min_point, max_point),
               rnd.randint(min_point, max_point))
    car_counter += 1

print(f'cars rendered: {car_counter}')
print(f'cars actually created: {len(Car.pool)}')

```


7. Finally, here is the fourth part of the main function:

```
c4 = Car(CarType.subcompact)
c5 = Car(CarType.subcompact)
c6 = Car(CarType.suv)
print(f'{id(c4)} == {id(c5)}? {id(c4) == id(c5)}')
print(f'{id(c5)} == {id(c6)}? {id(c5) == id(c6)}')
```

8. We do not forget our usual `__name__ == '__main__'` trick and good practice, as follows:

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

The execution of the `python flyweight` command shows the type, random color, and coordinates of the rendered objects, as well as the identity comparison results between flyweight objects of the same/different families:

```
render a tree of type TreeType.apple_tree and age 28 at (27, 14)
render a tree of type TreeType.apple_tree and age 10 at (15, 47)
render a tree of type TreeType.apple_tree and age 29 at (41, 29)
render a tree of type TreeType.apple_tree and age 26 at (80, 57)
render a tree of type TreeType.apple_tree and age 11 at (6, 66)
render a tree of type TreeType.apple_tree and age 24 at (0, 1)
render a tree of type TreeType.apple_tree and age 24 at (33, 96)
render a tree of type TreeType.apple_tree and age 15 at (24, 87)
render a tree of type TreeType.apple_tree and age 6 at (0, 87)
render a tree of type TreeType.apple_tree and age 9 at (40, 45)
render a tree of type TreeType.cherry_tree and age 24 at (27, 86)
render a tree of type TreeType.cherry_tree and age 30 at (75, 76)
render a tree of type TreeType.cherry_tree and age 15 at (78, 74)
render a tree of type TreeType.peach_tree and age 30 at (11, 24)
render a tree of type TreeType.peach_tree and age 24 at (21, 7)
render a tree of type TreeType.peach_tree and age 10 at (13, 89)
render a tree of type TreeType.peach_tree and age 2 at (85, 93)
render a tree of type TreeType.peach_tree and age 27 at (81, 79)
trees rendered: 18
trees actually created: 3
3085454594848 == 3085454594848? True
3085454594848 == 3085454560616? False
```

Do not expect to see the same output since the colors and coordinates are random, and the object identities depend on the memory map.

The model-view-controller pattern

One of the design principles related to software engineering is the **separation of concerns (SoC)** principle. The idea behind the SoC principle is to split an application into distinct sections, where each section addresses a separate concern. Examples of such concerns are the layers used in a layered design (data access layer, business logic layer, presentation layer, and so forth). Using the SoC principle simplifies the development and maintenance of software applications.

The MVC pattern is nothing more than the SoC principle applied to OOP. The name of the pattern comes from the three main components used to split a software application: the model, the view, and the controller. **MVC is considered an architectural pattern rather than a design pattern.** The difference between an architectural and a design pattern is that the former has a broader scope than the latter. Nevertheless, MVC is too important to skip just for this reason. Even if we will never have to implement it from scratch, we need to be familiar with it because all common frameworks use MVC or a slightly different version of it (more on this later).

The model is the core component. It represents knowledge. It contains and manages the (business) logic, data, state, and rules of an application. The view is a visual representation of the model. Examples of views are a computer GUI, the text output of a computer terminal, a smartphone's application GUI, a PDF document, a pie chart, a bar chart, and so forth. The view only displays the data; it doesn't handle it. The controller is the link/glue between the model and view. All communication between the model and the view happens through a controller.

A typical use of an application that uses MVC, after the initial screen is rendered to the user is as follows:

1. The user triggers a view by clicking (typing, touching, and so on) a button
2. The view informs the controller of the user's action
3. The controller processes user input and interacts with the model
4. The model performs all the necessary validation and state changes and informs the controller about what should be done
5. The controller instructs the view to update and display the output appropriately, following the instructions that are given by the model

You might be wondering, why the controller part is necessary? Can't we just skip it? We could, but then we would lose a big benefit that MVC provides: the ability to use more than one view (even at the same time, if that's what we want) without modifying the model. To achieve decoupling between the model and its representation, every view typically needs its own controller. If the model communicated directly with a specific view, we wouldn't be able to use multiple views (or at least, not in a clean and modular way).

Real-world examples

MVC is the SoC principle applied to OOP. The SoC principle is used a lot in real life. For example, if you build a new house, you usually assign different professionals to: 1) install the plumbing and electricity; and, 2) paint the house.

Another example is a restaurant. In a restaurant, the waiters receive orders and serve dishes to the customers, but the meals are cooked by the chefs.

In web development, several frameworks use the MVC idea:

- The Web2py Framework (j.mp/web2py) is a lightweight Python Framework that embraces the MVC pattern. If you have never tried Web2py, I encourage you to do it since it is extremely simple to install. There are many examples that demonstrate how MVC can be used in Web2py on the project's web page.
- Django is also an MVC Framework, although it uses different naming conventions. The controller is called **view**, and the view is called **template**. Django uses the name **Model-Template-View (MTV)**. According to the designers of Django, the view describes what data is seen by the user, and therefore, it uses the name view as the Python callback function for a particular URL. The term **template** in Django is used to separate content from representation. It describes how the data is seen by the user, not which data is seen.

Use cases

MVC is a very generic and useful design pattern. In fact, all popular web frameworks (Django, Rails, and Symfony or Yii) and application frameworks (iPhone SDK, Android, and QT) make use of MVC or a variation of it—**model-view-adapter (MVA)**, **model-view-presenter (MVP)**, and so forth. However, even if we don't use any of these frameworks, it makes sense to implement the pattern on our own because of the benefits it provides, which are as follows:

- The separation between the view and model allows graphics designers to focus on the UI part and programmers to focus on development, without interfering with each other.
- Because of the loose coupling between the view and model, each part can be modified/extended without affecting the other. For example, adding a new view is trivial. Just implement a new controller for it.
- Maintaining each part is easier because the responsibilities are clear.

When implementing MVC from scratch, be sure that you create smart models, thin controllers, and dumb views.

A model is considered smart because it does the following:

- Contains all the validation/business rules/logic
- Handles the state of the application
- Has access to application data (database, cloud, and so on)
- Does not depend on the UI

A controller is considered thin because it does the following:

- Updates the model when the user interacts with the view
- Updates the view when the model changes
- Processes the data before delivering it to the model/view, if necessary
- Does not display the data
- Does not access the application data directly
- Does not contain validation/business rules/logic

A view is considered dumb because it does the following:

- Displays the data
- Allows the user to interact with it
- Does only minimal processing, usually provided by a template language (for example, using simple variables and loop controls)
- Does not store any data
- Does not access the application data directly
- Does not contain validation/business rules/logic

If you are implementing MVC from scratch and want to find out if you did it right, you can try answering some key questions:

- If your application has a GUI, is it skinnable? How easily can you change the skin/look and feel of it? Can you give the user the ability to change the skin of your application during runtime? If this is not simple, it means that something is going wrong with your MVC implementation.
- If your application has no GUI (for instance, if it's a terminal application), how hard is it to add GUI support? Or, if adding a GUI is irrelevant, is it easy to add views to display the results in a chart (pie chart, bar chart, and so on) or a document (PDF, spreadsheet, and so on)? If these changes are not trivial (a matter of creating a new controller with a view attached to it, without modifying the model), MVC is not implemented properly.

If you make sure that these conditions are satisfied, your application will be more flexible and maintainable compared to an application that does not use MVC.

Implementation

I could use any of the common frameworks to demonstrate how to use MVC, but I feel that the picture will be incomplete. So, I decided to show you how to implement MVC from scratch, using a very simple example: a quote printer. The idea is extremely simple. The user enters a number and sees the quote related to that number. The quotes are stored in a quotes tuple. This is the data that normally exists in a database, file, and so on, and only the model has direct access to it.

Let's consider the example in the following code:

```
quotes =  
(  
    'A man is not complete until he is married. Then he is finished.',
```

```

    'As I said before, I never repeat myself.',
    'Behind a successful man is an exhausted woman.',
    'Black holes really suck...',
    'Facts are stubborn things.'
)

```

The model is minimalistic; it only has a `get_quote()` method that returns the quote (string) of the quotes tuple based on its index n . Note that n can be less than or equal to zero, due to the way indexing works in Python. Improving this behavior is given as an exercise for you at the end of this section:

```

class QuoteModel:
    def get_quote(self, n):
        try:
            value = quotes[n]
        except IndexError as err:
            value = 'Not found!'
        return value

```

The view has three methods: `show()`, which is used to print a quote (or the message **Not found!**) on the screen, `error()`, which is used to print an error message on the screen, and `select_quote()`, which reads the user's selection. This can be seen in the following code:

```

class QuoteTerminalView:
    def show(self, quote):
        print(f'And the quote is: "{quote}"')
    def error(self, msg):
        print(f'Error: {msg}')
    def select_quote(self):
        return input('Which quote number would you like to see? ')

```

The controller does the coordination. The `__init__()` method initializes the model and view. The `run()` method validates the quoted index given by the user, gets the quote from the model, and passes it back to the view to be displayed as shown in the following code:

```

class QuoteTerminalController:
    def __init__(self):
        self.model = QuoteModel()
        self.view = QuoteTerminalView()
    def run(self):
        valid_input = False
        while not valid_input:
            try:
                n = self.view.select_quote()
                n = int(n)
                valid_input = True
            except ValueError as err:

```

```
        self.view.error(f"Incorrect index '{n}'")
quote = self.model.get_quote(n)
self.view.show(quote)
```

Last but not least, the `main()` function initializes and fires the controller as shown in the following code:

```
def main():
    controller = QuoteTerminalController()
    while True:
        controller.run()
```

The following is the full code of the example (the `mvc.py` file):

- We start by defining a variable for the list of quotes as shown in the following code snippet:

```
quotes =
(
    'A man is not complete until he is married. Then he is finished.',
    'As I said before, I never repeat myself.',
    'Behind a successful man is an exhausted woman.',
    'Black holes really suck...',
    'Facts are stubborn things.'
)
```

- Here is the code for the model class, `QuoteModel`:

```
class QuoteModel:
    def get_quote(self, n):
        try:
            value = quotes[n]
        except IndexError as err:
            value = 'Not found!'
        return value
```

- Here is the code for the view class, QuoteTerminalView:

```
class QuoteTerminalView:
    def show(self, quote):
        print(f'And the quote is: "{quote}"')

    def error(self, msg):
        print(f'Error: {msg}')

    def select_quote(self):
        return input('Which quote number would you like to see?
')
```

- Here is the code for the controller class, QuoteTerminalController:

```
class QuoteTerminalController:
    def __init__(self):
        self.model = QuoteModel()
        self.view = QuoteTerminalView()
    def run(self):
        valid_input = False
        while not valid_input:
            try:
                n = self.view.select_quote()
                n = int(n)
                valid_input = True
            except ValueError as err:
                self.view.error(f"Incorrect index '{n}'")
        quote = self.model.get_quote(n)
        self.view.show(quote)
```

- Here is the end of our example code with the main() function:

```
def main():
    controller = QuoteTerminalController()
    while True:
        controller.run()

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


A sample execution of the `python mvc.py` command shows how the program prints quotes to the user:

```
Which quote number would you like to see? 2
And the quote is: "Behind a successful man is an exhausted woman."
Which quote number would you like to see? 4
And the quote is: "Facts are stubborn things."
Which quote number would you like to see? 1
And the quote is: "As I said before, I never repeat myself."
Which quote number would you like to see? 6
And the quote is: "Not found!"
Which quote number would you like to see? 3
And the quote is: "Black holes really suck..."
Which quote number would you like to see? 0
And the quote is: "A man is not complete until he is married. Then he is finished."
Which quote number would you like to see? _
```

The proxy pattern

The proxy design pattern gets its name from the proxy (also known as **surrogate**) object used to **perform an important action before accessing the actual object**. There are four different well-known proxy types (j.mp/proxypat). They are as follows:

- A **remote proxy**, which acts as the local representation of an object that really exists in a different address space (for example, a network server).
- A **virtual proxy**, which uses lazy initialization to defer the creation of a computationally expensive object until the moment it is actually needed.
- A **protection/protective proxy**, which controls access to a sensitive object.
- A **smart (reference) proxy**, which performs extra actions when an object is accessed. Examples of such actions are reference counting and thread-safety checks.

I find virtual proxies very useful so let's see an example of how we can implement them in Python right now. In the *Implementation* section, you will learn how to create protective proxies.

There are many ways to create a virtual proxy in Python, but I always like focusing on the idiomatic/Pythonic implementations. The code shown here is based on the great answer by Cyclone, a user of the site [stackoverflow.com \(j.mp/solazyinit\)](https://stackoverflow.com/j.mp/solazyinit). To avoid confusion, I should clarify that in this section, the terms *property*, *variable*, and *attribute* are used interchangeably. First, we create a `LazyProperty` class that can be used as a decorator. When it decorates a property, `LazyProperty` loads the property lazily (on the first use), instead of instantly. The `__init__()` method creates two variables that are used as aliases to the method that initializes a property. The `method` variable is an alias to the actual method, and the `method_name` variable is an alias to the method's name. To get a better understanding of how the two aliases are used, print their value to the output (uncomment the two commented lines in the following code):

```
class LazyProperty:
    def __init__(self, method):
        self.method = method
        self.method_name = method.__name__
        # print(f"function overridden: {self.fget}")
        # print(f"function's name: {self.func_name}")
```

The `LazyProperty` class is actually a descriptor (j.mp/pydesc). Descriptors are the recommended mechanisms to use in Python to override the default behavior of its attribute access methods: `__get__()`, `__set__()`, and `__delete__()`. The `LazyProperty` class overrides only `__set__()` because that is the only access method it needs to override. In other words, we don't have to override all access methods. The `__get__()` method accesses the value of the property the underlying method wants to assign, and uses `setattr()` to do the assignment manually. What `__get__()` actually does is very neat: it replaces the method with the value! This means that not only is the property lazily loaded, it can also be set only once. We will see what this means in a moment. Again, uncomment the commented line in the following code to get some extra information:

```
def __get__(self, obj, cls):
    if not obj:
        return None
    value = self.method(obj)
    # print(f'value {value}')
    setattr(obj, self.method_name, value)
    return value
```

The `Test` class shows how we can use the `LazyProperty` class. There are three attributes: `x`, `y`, and `_resource`. We want the `_resource` variable to be loaded lazily; thus, we initialize it to `None` as shown in the following code:

```
class Test:
    def __init__(self):
```

```
self.x = 'foo'  
self.y = 'bar'  
self._resource = None
```

The `resource()` method is decorated with the `LazyProperty` class. For demonstration purposes, the `LazyProperty` class initializes the `_resource` attribute as a tuple, as shown in the following code. Normally, this would be a slow/expensive initialization (database, graphics, and so on):

```
@LazyProperty  
def resource(self):  
    print(f'initializing self._resource which is: {self._resource}')  
    self._resource = tuple(range(5)) # expensive  
    return self._resource
```

The `main()` function, as follows, shows how lazy initialization behaves:

```
def main():  
    t = Test()  
    print(t.x)  
    print(t.y)  
    # do more work...  
    print(t.resource)  
    print(t.resource)
```

Notice how overriding the `__get()` access method makes it possible to treat the `resource()` method as a simple attribute (we can use `t.resource` instead of `t.resource()`).

In the execution output of this example (the `lazy.py` file), we can see that:

- The `_resource` variable is indeed initialized not by the time the `t` instance is created, but the first time that we use `t.resource`.
- The second time `t.resource` is used, the variable is not initialized again. That's why the initialization string initializing `self._resource` is shown only once.

Here is the output we get when executing the `python lazy.py` command:

```
foo  
bar  
initializing self._resource which is: None  
(0, 1, 2, 3, 4)  
(0, 1, 2, 3, 4)
```

There are two basic, different kinds of lazy initialization in OOP. They are as follows:

- **At the instance level:** This means that an object's property is initialized lazily, but the property has an object scope. Each instance (object) of the same class has its own (different) copy of the property.
- **At the class or module level:** In this case, we do not want a different copy per instance, but all the instances share the same property, which is lazily initialized. This case is not covered in this chapter. If you find it interesting, consider it as an exercise.

Real-world examples

Chip (also known as **Chip and PIN**) cards (j.mp/wichpin) offer a good example of how a protective proxy is used in real life. The debit/credit card contains a chip that first needs to be read by the ATM or card reader. After the chip is verified, a password (PIN) is required to complete the transaction. This means that you cannot make any transactions without physically presenting the card and knowing the PIN.

A bank check that is used instead of cash to make purchases and deals is an example of a remote proxy. The check gives access to a bank account.

In software, the `weakref` module of Python contains a `proxy()` method that accepts an input object and returns a smart proxy to it. Weak references are the recommended way to add reference-counting support to an object.

Use cases

Since there are at least four common proxy types, the proxy design pattern has many use cases, as follows:

- It is used when creating a distributed system using either a private network or the cloud. In a distributed system, some objects exist in the local memory and some objects exist in the memory of remote computers. If we don't want the client code to be aware of such differences, we can create a remote proxy that hides/encapsulates them, making the distributed nature of the application transparent.
- It is used when our application is suffering from performance issues due to the early creation of expensive objects. Introducing lazy initialization using a virtual proxy to create the objects only at the moment they are actually required can give us significant performance improvements.

- It is used to check if a user has sufficient privileges to access a piece of information. If our application handles sensitive information (for example, medical data), we want to make sure that the user trying to access/modify it is allowed to do so. A protection/protective proxy can handle all security-related actions.
- It is used when our application (or library, toolkit, framework, and so forth) uses multiple threads and we want to move the burden of thread safety from the client code to the application. In this case, we can create a smart proxy to hide the thread-safety complexities from the client.
- An **object-relational mapping (ORM)** API is also an example of how to use a remote proxy. Many popular web frameworks, including Django, use an ORM to provide OOP-like access to a relational database. An ORM acts as a proxy to a relational database that can be actually located anywhere, either at a local or remote server.

Implementation

To demonstrate the proxy pattern, we will implement a simple protection proxy to view and add users. The service provides two options:

- **Viewing the list of users:** This operation does not require special privileges
- **Adding a new user:** This operation requires the client to provide a special secret message

The `SensitiveInfo` class contains the information that we want to protect. The `users` variable is the list of existing users. The `read()` method prints the list of the users. The `add()` method adds a new user to the list.

Let's consider the following code:

```
class SensitiveInfo:
    def __init__(self):
        self.users = ['nick', 'tom', 'ben', 'mike']
    def read(self):
        nb = len(self.users)
        print(f"There are {nb} users: {' '.join(self.users)}")
    def add(self, user):
        self.users.append(user)
        print(f'Added user {user}')
```

The `Info` class is a protection proxy of `SensitiveInfo`. The `secret` variable is the message required to be known/provided by the client code to add a new user. Note that this is just an example. In reality, you should never do the following:

- Store passwords in the source code
- Store passwords in a clear-text form
- Use a weak (for example, MD5) or custom form of encryption

In the `Info` class, as we can see next, the `read()` method is a wrapper to `SensitiveInfo.read()` and the `add()` method ensures that a new user can be added only if the client code knows the secret message:

```
class Info:
    '''protection proxy to SensitiveInfo'''
    def __init__(self):
        self.protected = SensitiveInfo()
        self.secret = '0xdeadbeef'
    def read(self):
        self.protected.read()
    def add(self, user):
        sec = input('what is the secret? ')
        self.protected.add(user) if sec == self.secret else print("That's
wrong!")
```

The `main()` function shows how the proxy pattern can be used by the client code. The client code creates an instance of the `Info` class and uses the displayed menu to read the list, add a new user, or exit the application. Let's consider the following code:

```
def main():
    info = Info()
    while True:
        print('1. read list |==| 2. add user |==| 3. quit')
        key = input('choose option: ')
        if key == '1':
            info.read()
        elif key == '2':
            name = input('choose username: ')
            info.add(name)
        elif key == '3':
            exit()
        else:
            print(f'unknown option: {key}')
```

Let's recapitulate the full code of the `proxy.py` file:

1. First, we define the `LazyProperty` class:

```
class LazyProperty:
    def __init__(self, method):
        self.method = method
        self.method_name = method.__name__
        # print(f"function overridden: {self.fget}")
        # print(f"function's name: {self.func_name}")
    def __get__(self, obj, cls):
        if not obj:
            return None
        value = self.method(obj)
        # print(f'value {value}')
        setattr(obj, self.method_name, value)
        return value
```

2. Then, we have the code for the `Test` class, as follows:

```
class Test:
    def __init__(self):
        self.x = 'foo'
        self.y = 'bar'
        self._resource = None
    @LazyProperty
    def resource(self):
        print(f'initializing self._resource which is:
{self._resource}')
        self._resource = tuple(range(5)) # expensive
        return self._resource
```

3. Finally, here is the `main()` function and the end of the code:

```
def main():
    t = Test()
    print(t.x)
    print(t.y)
    # do more work...
    print(t.resource)
    print(t.resource)

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

4. We can see here a sample output of the program when executing the `python proxy.py` command:

```
1. read list ==| 2. add user ==| 3. quit
choose option: 1
There are 4 users: nick tom ben mike
1. read list ==| 2. add user ==| 3. quit
choose option: 2
choose username: bill
what is the secret? 12345
That's wrong!
1. read list ==| 2. add user ==| 3. quit
choose option: 2
choose username: bill
what is the secret? 0xdeadbeef
Added user bill
1. read list ==| 2. add user ==| 3. quit
choose option: 1
There are 5 users: nick tom ben mike bill
1. read list ==| 2. add user ==| 3. quit
```

Have you already spotted flaws or missing features that can improve our proxy example? I have a few suggestions. They are as follows:

- This example has a very big security flaw. Nothing prevents the client code from bypassing the security of the application by creating an instance of `SensitiveInfo` directly. Improve the example to prevent this situation. One way is to use the `abc` module to forbid direct instantiation of `SensitiveInfo`. What are other code changes required in this case?
- A basic security rule is that we should never store clear-text passwords. Storing a password safely is not very hard as long as we know which libraries to use (`j.mp/hashsec`). If you have an interest in security, try to implement a secure way to store the secret message externally (for example, in a file or database).
- The application only supports adding new users, but what about removing an existing user? Add a `remove()` method.

Summary

In this chapter, we covered three other structural design patterns: flyweight, MVC, and proxy.

We can use flyweight when we want to improve the memory usage and possibly the performance of our application. This is quite important in all systems with limited resources (think of embedded systems), and systems that focus on performance, such as graphics software and electronic games.

In general, we use flyweight when an application needs to create a large number of computationally expensive objects that share many properties. The important point is to separate the immutable (shared) properties from the mutable. We implemented a tree renderer that supports three different tree families. By providing the mutable `age` and `x, y` properties explicitly to the `render()` method, we managed to create only three different objects instead of eighteen. Although that might not seem like a big win, imagine if the trees were 2,000 instead of 18.

MVC is a very important design pattern used to structure an application in three parts: the model, the view, and the controller. Each part has clear roles and responsibilities. The model has access to the data and manages the state of the application. The view is a representation of the model. The view does not need to be graphical; textual output is also considered a totally fine view. The controller is the link between the model and the view. Proper use of MVC guarantees that we end up with an application that is easy to maintain and extend.

We discussed several use cases of the proxy pattern, including performance, security, and how to offer simple APIs to users. In the first code example, we created a virtual proxy (using decorators and descriptors), allowing us to initialize object properties in a lazy manner. In the second code example, we implemented a protection proxy to handle users. This example can be improved in many ways, especially regarding its security flaws and the fact that the list of users is not persistent.

In the next chapter, we will start exploring behavioral design patterns. Behavioral patterns cope with object interconnection and algorithms. The first behavioral pattern that will be covered is a chain of responsibility.

9 The Chain of Responsibility Pattern

When developing an application, most of the time we know which method should satisfy a particular request in advance. However, this is not always the case. For example, think of any broadcast computer network, such as the original Ethernet implementation (j.mp/wikishared). In broadcast computer networks, all requests are sent to all nodes (broadcast domains are excluded for simplicity), but only the nodes that are interested in a sent request process it.

All computers that participate in a broadcast network are connected to each other using a common medium such as the cable that connects all nodes. If a node is not interested or does not know how to handle a request, it can perform the following actions:

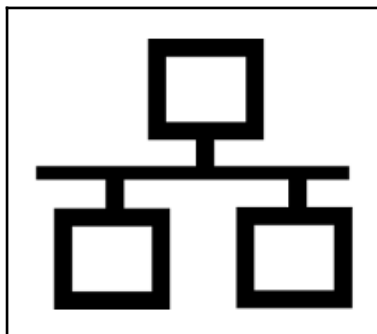
- Ignore the request and do nothing
- Forward the request to the next node

The way in which the node reacts to a request is an implementation detail. However, we can use the analogy of a broadcast computer network to understand what the Chain of Responsibility pattern is all about. The Chain of Responsibility pattern is used when we want to give a chance to multiple objects to satisfy a single request, or when we don't know in advance which object (from a chain of objects) should process a specific request.

To illustrate the principle, imagine a chain (linked list, tree, or any other convenient data structure) of objects, and the following flow:

1. We start by sending a request to the first object in the chain
2. The object decides whether it should satisfy the request or not
3. The object forwards the request to the next object
4. This procedure is repeated until we reach the end of the chain

At the application level, instead of talking about cables and network nodes, we can focus on objects and the flow of a request. The following diagram shows how the client code sends a request to all processing elements of an application:



Note that the client code only knows about the first processing element, instead of having references to all of them, and each processing element only knows about its immediate next neighbor (called the **successor**), not about every other processing element. This is usually a one-way relationship, which in programming terms means a singly linked list in contrast to a doubly linked list; a singly linked list does not allow navigation in both ways, while a doubly linked list allows that. This chain organization is used for a good reason. It achieves decoupling between the sender (client) and the receivers (processing elements).

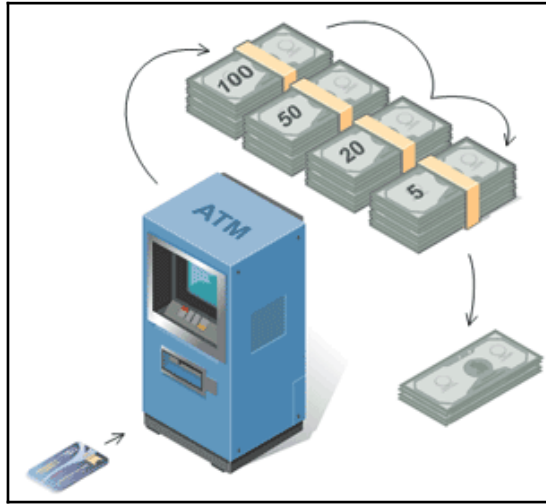
In this chapter, we will discuss:

- Real-world examples
- Use cases
- Implementation

Real-world examples

ATMs and, in general, any kind of machine that accepts/returns banknotes or coins (for example, a snack-vending machine) use the Chain of Responsibility pattern.

There is always a single slot for all banknotes, as shown in the following diagram, courtesy of sourcemaking.com (www.sourcemaking.com):



When a banknote is dropped, it is routed to the appropriate receptacle. When it is returned, it is taken from the appropriate receptacle. We can think of the single slot as the shared communication medium and the different receptacles as the processing elements. The result contains cash from one or more receptacles. For example, in the preceding diagram, we see what happens when we request \$175 from the ATM.

In software, the servlet filters of Java are pieces of code that are executed before an HTTP request arrives at a target. When using servlet filters, there is a chain of filters. Each filter performs a different action (user authentication, logging, data compression, and so forth), and either forwards the request to the next filter until the chain is exhausted, or it breaks the flow if there is an error—for example, the authentication failed three consecutive times (j.mp/soservl).

Another software example, Apple's Cocoa and Cocoa Touch frameworks, use the Chain of Responsibility to handle events. When a view receives an event that it doesn't know how to handle, it forwards the event to its superview. This goes on until a view is capable of handling the event or the chain of views is exhausted (j.mp/chaincocoa).

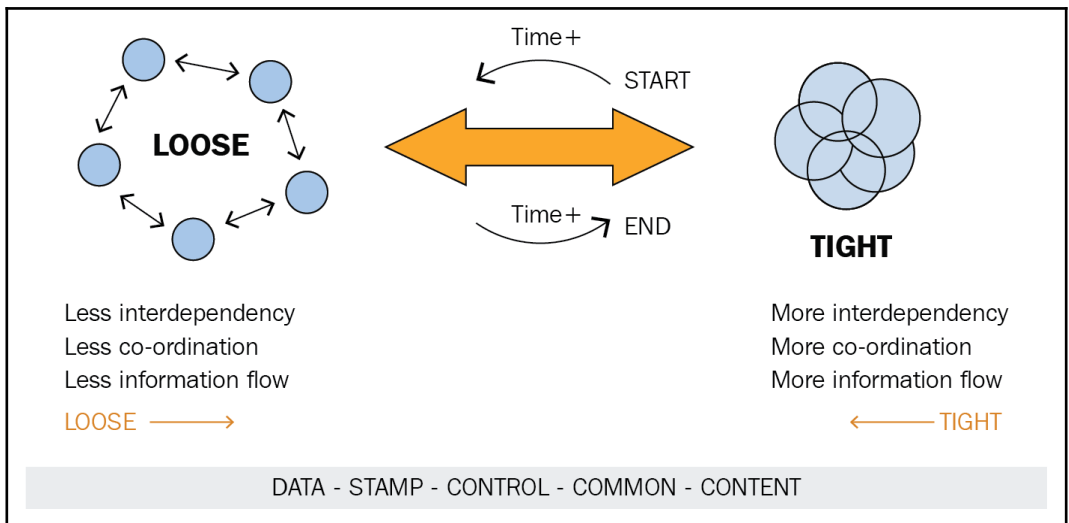
Use cases

By using the Chain of Responsibility pattern, we provide a chance to a number of different objects to satisfy a specific request. This is useful when we don't know which object should satisfy a request in advance. An example is a purchase system. In purchase systems, there are many approval authorities. One approval authority might be able to approve orders up to a certain value, let's say \$100. If the order is for more than \$100, the order is sent to the next approval authority in the chain that can approve orders up to \$200, and so forth.

Another case where the Chain of Responsibility is useful is when we know that more than one object might need to process a single request. This is what happens in event-based programming. A single event, such as a left-mouse click, can be caught by more than one listener.

It is important to note that the Chain of Responsibility pattern is not very useful if all the requests can be taken care of by a single processing element, unless we really don't know which element that is. The value of this pattern is the decoupling that it offers. Instead of having a many-to-many relationship between a client and all processing elements (and the same is true regarding the relationship between a processing element and all other processing elements), a client only needs to know how to communicate with the start (head) of the chain.

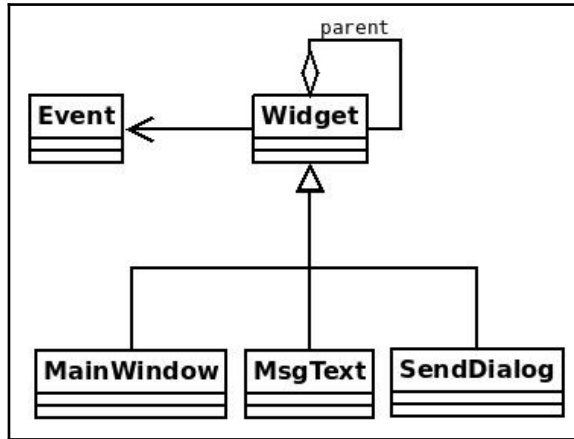
The following diagram illustrates the difference between tight and loose coupling. The idea behind loosely coupled systems is to simplify maintenance and make it easier for us to understand how they function (j.mp/loosecouple):



Implementation

There are many ways to implement a Chain of Responsibility in Python, but my favorite implementation is the one by Vespe Savikko (<https://legacy.python.org/workshops/1997-10/proceedings/savikko.html>). Vespe's implementation uses dynamic dispatching in a Pythonic style to handle requests (<http://j.mp/ddispatch>).

Let's implement a simple, event-based system using Vespe's implementation as a guide. The following is the UML class diagram of the system:



The `Event` class describes an event. We'll keep it simple, so in our case, an event has only a name:

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

    def __str__(self):
        return self.name
```

The `Widget` class is the core class of the application. The **parent** aggregation shown in the UML diagram indicates that each widget can have a reference to a parent object, which by convention, we assume is a `Widget` instance. Note, however, that according to the rules of inheritance, an instance of any of the subclasses of `Widget` (for example, an instance of `MsgText`) is also an instance of `Widget`. The default value of `parent` is `None`:

```
class Widget:
    def __init__(self, parent=None):
        self.parent = parent
```

The `handle()` method uses dynamic dispatching through `hasattr()` and `getattr()` to decide who is the handler of a specific request (event). If the widget that is asked to handle an event does not support it, there are two fallback mechanisms. If the widget has a parent, then the `handle()` method of the parent is executed. If the widget has no parent but a `handle_default()` method, `handle_default()` is executed:

```
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)
```

At this point, you might have realized why the `Widget` and `Event` classes are only associated (no aggregation or composition relationships) in the UML class diagram. The association is used to show that the `Widget` class knows about the `Event` class but does not have any strict references to it, since an event needs to be passed only as a parameter to `handle()`.

`MainWindow`, `MsgText`, and `SendDialog` are all widgets with different behaviors. Not all these three widgets are expected to be able to handle the same events, and even if they can handle the same event, they might behave differently. `MainWindow` can handle only the close and default events:

```
class MainWindow(Widget):
    def handle_close(self, event):
        print(f'MainWindow: {event}')

    def handle_default(self, event):
        print(f'MainWindow Default: {event}')
```

`SendDialog` can handle only the paint event:

```
class SendDialog(Widget):
    def handle_paint(self, event):
        print(f'SendDialog: {event}')
```

Finally, `MsgText` can handle only the down event:

```
class MsgText(Widget):
    def handle_down(self, event):
        print(f'MsgText: {event}')
```

The `main()` function shows how we can create a few widgets and events, and how the widgets react to those events. All events are sent to all the widgets. Note the parent relationship of each widget. The `sd` object (an instance of `SendDialog`) has as its parent the `mw` object (an instance of `MainWindow`). However, not all objects need to have a parent that is an instance of `MainWindow`. For example, the `msg` object (an instance of `MsgText`) has the `sd` object as a parent:

```
def main():
    mw = MainWindow()
    sd = SendDialog(mw)
    msg = MsgText(sd)

    for e in ('down', 'paint', 'unhandled', 'close'):
        evt = Event(e)
        print(f'Sending event {evt} to MainWindow')
        mw.handle(evt)
        print(f'Sending event {evt} to SendDialog')
        sd.handle(evt)
        print(f'Sending event {evt} to MsgText')
        msg.handle(evt)
```

The following is the full code of the example (`chain.py`):

1. We define the `Event` class:

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

    def __str__(self):
        return self.name
```

2. Then, we define the `Widget` class:

```
class Widget:
    def __init__(self, parent=None):
        self.parent = parent

    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)
```


3. We add the specialized widget classes, `MainWindow`, `SendDialog`, and `MsgText`:

```
class MainWindow(Widget):
    def handle_close(self, event):
        print(f'MainWindow: {event}')

    def handle_default(self, event):
        print(f'MainWindow Default: {event}')

class SendDialog(Widget):
    def handle_paint(self, event):
        print(f'SendDialog: {event}')

class MsgText(Widget):
    def handle_down(self, event):
        print(f'MsgText: {event}')
```

4. Finally, we have the code for the `main()` function and the usual snippet where we call it:

```
def main():
    mw = MainWindow()
    sd = SendDialog(mw)
    msg = MsgText(sd)

    for e in ('down', 'paint', 'unhandled', 'close'):
        evt = Event(e)
        print(f'Sending event -{evt}- to MainWindow')
        mw.handle(evt)
        print(f'Sending event -{evt}- to SendDialog')
        sd.handle(evt)
        print(f'Sending event -{evt}- to MsgText')
        msg.handle(evt)

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

5. Executing the `python chain.py` command gives us the following output:

```
Sending event -down- to MainWindow
MainWindow Default: down
Sending event -down- to SendDialog
MainWindow Default: down
Sending event -down- to MsgText
MsgText: down
Sending event -paint- to MainWindow
MainWindow Default: paint
Sending event -paint- to SendDialog
SendDialog: paint
Sending event -paint- to MsgText
SendDialog: paint
Sending event -unhandled- to MainWindow
MainWindow Default: unhandled
Sending event -unhandled- to SendDialog
MainWindow Default: unhandled
Sending event -unhandled- to MsgText
MainWindow Default: unhandled
Sending event -close- to MainWindow
MainWindow: close
Sending event -close- to SendDialog
MainWindow: close
Sending event -close- to MsgText
MainWindow: close
```

There are some interesting things that we can see in the output. For instance, sending a `down` event to `MainWindow` ends up being handled by the default `MainWindow` handler. Another nice case is that although a `close` event cannot be handled directly by `SendDialog` and `MsgText`, all the `close` events end up being handled properly by `MainWindow`. That's the beauty of using the parent relationship as a fallback mechanism.

If you want to spend some more creative time on the event example, you can replace the dumb `print` statements and add some actual behavior to the listed events. Of course, you are not limited to the listed events. Just add your favorite event and make it do something useful!

Another exercise is to add a `MsgText` instance during runtime that has `MainWindow` as the parent. Is this hard? Do the same for an event (add a new event to an existing widget). Which is harder?

Summary

In this chapter, we covered the Chain of Responsibility design pattern. This pattern is useful to model requests and/or handle events when the number and type of handlers aren't known in advance. Examples of systems that fit well with Chain of Responsibility are event-based systems, purchase systems, and shipping systems.

In the Chain of Responsibility pattern, the sender has direct access to the first node of a chain. If the request cannot be satisfied by the first node, it forwards it to the next node. This continues until either the request is satisfied by a node or the whole chain is traversed. This design is used to achieve loose coupling between the sender and the receiver(s).

ATMs are an example of Chain of Responsibility. The single slot that is used for all banknotes can be considered the head of the chain. From here, depending on the transaction, one or more receptacles are used to process the transaction. The receptacles can be considered to be the processing elements of the chain.

Java's servlet filters use the Chain of Responsibility pattern to perform different actions (for example, compression and authentication) on an HTTP request. Apple's Cocoa Frameworks use the same pattern to handle events such as button presses and finger gestures.

10

The Command Pattern

Most applications nowadays have an **undo** operation. It is hard to imagine, but undo did not exist in any software for many years. Undo was introduced in 1974 (j.mp/wiundo), but Fortran and Lisp, two programming languages that are still widely used, were created in 1957 and 1958, respectively (j.mp/proghist)! I wouldn't like to have been an application user during those years. Making a mistake meant that the user had no easy way to fix it.

Enough with the history. We want to know how we can implement the undo functionality in our applications. And since you have read the title of this chapter, you already know which design pattern is recommended to implement undo: the Command pattern.

The Command design pattern helps us encapsulate an operation (undo, redo, copy, paste, and so forth) as an object. What this simply means is that we create a class that contains all the logic and the methods required to implement the operation. The advantages of doing this are as follows (j.mp/cmdpattern):

- We don't have to execute a command directly. It can be executed at will.
- The object that invokes the command is decoupled from the object that knows how to perform it. The invoker does not need to know any implementation details about the command.
- If it makes sense, multiple commands can be grouped to allow the invoker to execute them in order. This is useful, for instance, when implementing a multilevel undo command.

In this chapter, we will discuss:

- Real-world examples
- Use cases
- Implementation

Real-world examples

When we go to a restaurant for dinner, we give the order to the waiter. The check (usually paper) that they use to write the order on is an example of a command. After writing the order, the waiter places it in the check queue that is executed by the cook. Each check is independent and can be used to execute many different commands, for example, one command for each item that will be cooked.

As you would expect, we also have several examples in software. Here are two I can think of:

- PyQt is the Python binding of the QT toolkit. PyQt contains a `QAction` class that models an action as a command. Extra optional information is supported for every action, such as description, tooltip, shortcut, and more (`j.mp/qaction`).
- Git Cola (`j.mp/git-cola`), a Git GUI written in Python, uses the Command pattern to modify the model, amend a commit, apply a different election, check out, and so forth (`j.mp/git-cola-code`).

Use cases

Many developers use the undo example as the only use case of the Command pattern. The truth is that undo is the killer feature of the Command pattern. However, the Command pattern can actually do much more (`j.mp/commandp`):

- **GUI buttons and menu items:** The PyQt example that was already mentioned uses the Command pattern to implement actions on buttons and menu items.
- **Other operations:** Apart from undo, commands can be used to implement any operation. A few examples are cut, copy, paste, redo, and capitalize text.
- **Transactional behavior and logging:** Transactional behavior and logging are important to keep a persistent log of changes. They are used by operating systems to recover from system crashes, relational databases to implement transactions, filesystems to implement snapshots, and installers (wizards) to revert canceled installations.
- **Macros:** By macros, in this case, we mean a sequence of actions that can be recorded and executed on demand at any point in time. Popular editors such as Emacs and Vim support macros.

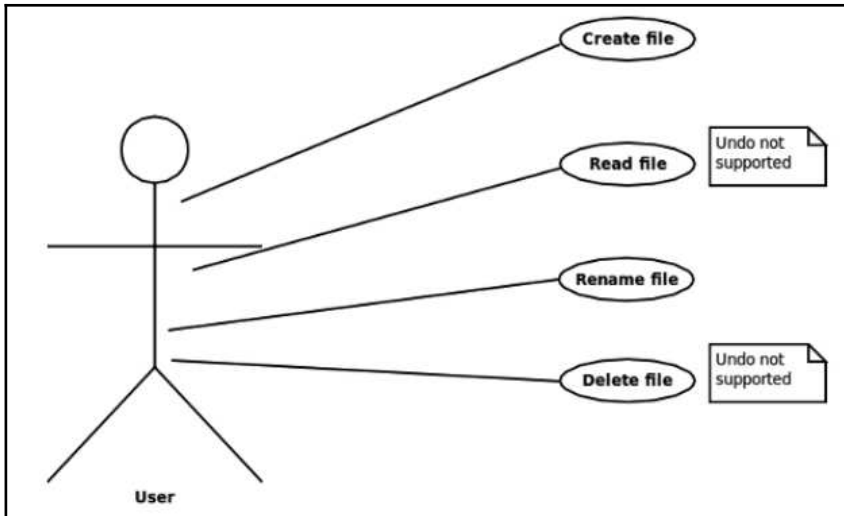
Implementation

In this section, we will use the Command pattern to implement the most basic file utilities:

- Creating a file and optionally writing text (a string) to it
- Reading the contents of a file
- Renaming a file
- Deleting a file

We will not implement these utilities from scratch, since Python already offers good implementations of them in the `os` module. What we want is to add an extra abstraction level on top of them so that they can be treated as commands. By doing this, we get all the advantages offered by commands.

From the operations shown, renaming a file and creating a file support undo. Deleting a file and reading the contents of a file do not support undo. Undo can actually be implemented on delete file operations. One technique is to use a special trash/wastebasket directory that stores all the deleted files, so that they can be restored when the user requests it. This is the default behavior used on all modern desktop environments and is left as an exercise.



Each command has two parts:

- **The initialization part:** It is taken care of by the `__init__()` method and contains all the information required by the command to be able to do something useful (the path of a file, the contents that will be written to the file, and so forth).

- **The execution part:** It is taken care of by the `execute()` method. We call the `execute()` method when we want to actually run a command. This is not necessarily right after initializing it.

Let's start with the `rename` utility, which is implemented using the `RenameFile` class. The `__init__()` method accepts the source (`src`) and destination (`dest`) file paths as parameters (strings). If no path separators are used, the current directory is used to create the file. An example of using a path separator is passing the `/tmp/file1` string as `src` and the `/home/user/file2` string as `dest`. Another example, where we would not use a path, is passing `file1` as `src` and `file2` as `dest`:

```
class RenameFile:
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest
```

We add the `execute()` method to the class. This method does the actual renaming using `os.rename()`. The `verbose` variable corresponds to a global **flag**, which, when activated (by default, it is activated) gives feedback to the user about the operation that is performed. You can deactivate it if you prefer silent commands. Note that although `print()` is good enough for an example, normally something more mature and powerful can be used, for example, the `logging` module (`j.mp/py3log`):

```
def execute(self):
    if verbose:
        print(f"[renaming '{self.src}' to '{self.dest}']")
    os.rename(self.src, self.dest)
```

Our `rename` utility (`RenameFile`) supports the `undo` operation through its `undo()` method. In this case, we use `os.rename()` again to revert the name of the file to its original value:

```
def undo(self):
    if verbose:
        print(f"[renaming '{self.dest}' back to '{self.src}']")
    os.rename(self.dest, self.src)
```

In this example, deleting a file is implemented in a function, instead of a class. That is to show it is not mandatory to create a new class for every command that you want to add (more on that will be covered later). The `delete_file()` function accepts a file path as a string and uses `os.remove()` to delete it:

```
def delete_file(path):
    if verbose:
        print(f"deleting file {path}")
    os.remove(path)
```

Back to using classes again. The `CreateFile` class is used to create a file. The `__init__()` method for that class accepts the familiar `path` parameter and a `txt` parameter for the content (a string) that will be written to the file. If nothing is passed as `txt`, the default `hello world` text is written to the file. Normally, the sane default behavior is to create an empty file, but for the needs of this example, I decided to write a default string in it.

The definition of the `CreateFile` class starts as follows:

```
class CreateFile:

    def __init__(self, path, txt='hello world\n'):
        self.path = path
        self.txt = txt
```

Then we add an `execute()` method, in which we use the `with` statement and Python's `open()` built-in function to open the file (`mode='w'` means write mode), and the `write()` function to write the `txt` string to it, as follows:

```
def execute(self):
    if verbose:
        print(f"[creating file '{self.path}']")
    with open(self.path, mode='w', encoding='utf-8') as out_file:
        out_file.write(self.txt)
```

The undo for the operation of creating a file is to delete that file. So, the `undo()` method, which we add to the class, simply uses the `delete_file()` function to achieve that, as follows:

```
def undo(self):
    delete_file(self.path)
```

The last utility gives us the ability to read the contents of a file. The `execute()` method of the `ReadFile` class uses `open()` again, this time in read mode, and just prints the content of the file using `print()`.

The `ReadFile` class is defined as follows:

```
class ReadFile:

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

    def execute(self):
        if verbose:
            print(f"[reading file '{self.path}']")
```



```
with open(self.path, mode='r', encoding='utf-8') as in_file:
    print(in_file.read(), end='')
```

The `main()` function makes use of the utilities we have defined. The `orig_name` and `new_name` parameters are the original and new name of the file that is created and renamed. A `commands` list is used to add (and configure) all the commands that we want to execute at a later point. Note that the commands are not executed unless we explicitly call `execute()` for each command:

```
def main():
    orig_name, new_name = 'file1', 'file2'
    commands = (
        CreateFile(orig_name),
        ReadFile(orig_name),
        RenameFile(orig_name, new_name)
    )
    [c.execute() for c in commands]
```

The next step is to ask the users if they want to undo the executed commands or not. The user selects whether the commands will be undone or not. If they choose to undo them, `undo()` is executed for all commands in the `commands` list. However, since not all commands support undo, exception handling is used to catch (and ignore) the `AttributeError` exception generated when the `undo()` method is missing. The code would look like the following:

```
answer = input('reverse the executed commands? [y/n] ')
if answer not in 'yY':
    print(f"the result is {new_name}")
    exit()
for c in reversed(commands):
    try:
        c.undo()
    except AttributeError as e:
        print("Error", str(e))
```

Using exception handling for such cases is an acceptable practice, but if you don't like it, you can check explicitly whether a command supports the undo operation by adding a **Boolean** method, for example, `supports_undo()` or `can_be_undone()`. Again, that is not mandatory.

Here's the full code of the example (`command.py`):

1. We import the `os` module and we define the constant we need:

```
import os
verbose = True
```

2. Here, we define the class for the rename file operation:

```
class RenameFile:

    def __init__(self, src, dest):
        self.src = src
        self.dest = dest

    def execute(self):
        if verbose:
            print(f"[renaming '{self.src}' to '{self.dest}']")
            os.rename(self.src, self.dest)

    def undo(self):
        if verbose:
            print(f"[renaming '{self.dest}' back to '{self.src}']")
            os.rename(self.dest, self.src)
```

3. Here, we define the class for the create file operation:

```
class CreateFile:

    def __init__(self, path, txt='hello world\n'):
        self.path = path
        self.txt = txt

    def execute(self):
        if verbose:
            print(f"[creating file '{self.path}']")
            with open(self.path, mode='w', encoding='utf-8') as
out_file:
                out_file.write(self.txt)

    def undo(self):
        delete_file(self.path)
```

4. We also define the class for the read file operation, as follows:

```
class ReadFile:

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

    def execute(self):
        if verbose:
            print(f"[reading file '{self.path}']")
            with open(self.path, mode='r', encoding='utf-8') as
in_file:
                print(in_file.read(), end='')
```

5. And for the delete file operation, we decide to use a function (and not a class), as follows:

```
def delete_file(path):
    if verbose:
        print(f"deleting file {path}")
    os.remove(path)
```

6. Here is the main part of the program now:

```
def main():

    orig_name, new_name = 'file1', 'file2'
    commands = (
        CreateFile(orig_name),
        ReadFile(orig_name),
        RenameFile(orig_name, new_name)
    )
    [c.execute() for c in commands]
    answer = input('reverse the executed commands? [y/n] ')
    if answer not in 'yY':
        print(f"the result is {new_name}")
        exit()
    for c in reversed(commands):
        try:
            c.undo()
        except AttributeError as e:
            print("Error", str(e))
    if __name__ == "__main__":
        main()
```

Let's see two sample executions using the `python command.py` command line.

In the first one, there is no undo of commands:

```
[creating file 'file1']
[reading file 'file1']
hello world
[renaming 'file1' to 'file2']
reverse the executed commands? [y/n] y
[renaming 'file2' back to 'file1']
Error 'ReadFile' object has no attribute 'undo'
deleting file file1
```

In the second one, we have the undo of commands:

```
[creating file 'file1']  
[reading file 'file1']  
hello world  
[renaming 'file1' to 'file2']  
reverse the executed commands? [y/n] n  
the result is file2
```

But wait! Let's see what can be improved in our command implementation example. Among the things to consider are the following:

- What happens if we try to rename a file that doesn't exist?
- What about files that exist but cannot be renamed because we don't have the proper filesystem permissions?

We can try improving the utilities by doing some kind of error handling. Checking the return status of the functions in the `os` module can be useful. We could check if the file exists before trying the delete action, using the `os.path.exists()` function.

Also, the file creation utility creates a file using the default file permissions as decided by the filesystem. For example, in POSIX systems, the permissions are `-rw-rw-r--`. You might want to give the ability to the user to provide their own permissions by passing the appropriate parameter to `CreateFile`. How can you do that? Hint: one way is by using `os.fopen()`.

And now, here's something for you to think about. I mentioned earlier that a command does not necessarily need to be a class. That's how the delete utility was implemented; there is just a `delete_file()` function. What are the advantages and disadvantages of this approach? Here's a hint: Is it possible to put a delete command in the commands list as was done for the rest of the commands? We know that functions are first-class citizens in Python, so we can do something such as the following (see the `first-class.py` file):

```
import os  
verbose = True  
  
class CreateFile:  
  
    def __init__(self, path, txt='hello world\n'):  
        self.path = path  
        self.txt = txt  
  
    def execute(self):
```

```

        if verbose:
            print(f"[creating file '{self.path}']")
        with open(self.path, mode='w', encoding='utf-8') as out_file:
            out_file.write(self.txt)
    def undo(self):
        try:
            delete_file(self.path)
        except:
            print('delete action not successful...')
            print('... file was probably already deleted.')

def delete_file(path):
    if verbose:
        print(f"deleting file {path}...")
    os.remove(path)
def main():

    orig_name = 'file1'
    df=delete_file

    commands = [CreateFile(orig_name),]
    commands.append(df)
    for c in commands:
        try:
            c.execute()
        except AttributeError as e:
            df(orig_name)
    for c in reversed(commands):
        try:
            c.undo()
        except AttributeError as e:
            pass
    if __name__ == "__main__":
        main()

```

Although this variant of the implementation example works, there are still some issues:

- The code is not uniform. We rely too much on exception handling, which is not the normal flow of a program. While all the other commands we implemented have an `execute()` method, in this case, there is no `execute()`.
- Currently, the delete file utility has no undo support. What happens if we eventually decide to add undo support for it? Normally, we add an `undo()` method in the class that represents the command. However, in this case, there is no class. We could create another function to handle undo, but creating a class is a better approach.

Summary

In this chapter, we covered the Command pattern. Using this design pattern, we can encapsulate an operation, such as copy/paste, as an object. This offers many benefits, as follows:

- We can execute a command whenever we want, and not necessarily at creation time
- The client code that executes a command does not need to know any details about how it is implemented
- We can group commands and execute them in a specific order

Executing a command is like ordering at a restaurant. Each customer's order is an independent command that enters many stages and is finally executed by the cook.

Many GUI frameworks, including PyQt, use the Command pattern to model actions that can be triggered by one or more events and can be customized. However, Command is not limited to frameworks; normal applications such as `git-cola` also use it for the benefits it offers.

Although the most advertised feature of command by far is undo, it has more uses. In general, any operation that can be executed at the user's will at runtime is a good candidate to use the Command pattern. The command pattern is also great for grouping multiple commands. It's useful for implementing macros, multilevel undoing, and transactions. A transaction should either succeed, which means that all operations of it should succeed (the commit operation), or it should fail completely if at least one of its operations fails (the rollback operation). If you want to take the Command pattern to the next level, you can work on an example that involves grouping commands as transactions.

To demonstrate command, we implemented some basic file utilities on top of Python's `os` module. Our utilities supported undo and had a uniform interface, which makes grouping commands easy.

The next chapter covers the Observer pattern.

11

The Observer Pattern

When we need to update a group of objects when the state of another object changes, a popular solution is offered by the **Model-View-Controller (MVC)** pattern. Assume that we are using the data of the same *model* in two *views*, for instance in a pie chart and in a spreadsheet. Whenever the model is modified, both the views need to be updated. That's the role of the Observer design pattern.

The Observer pattern describes a publish-subscribe relationship between a single object, the publisher, which is also known as the **subject** or **observable**, and one or more objects, the subscribers, also known as **observers**.

In the case of MVC, the publisher is the model and the subscribers are the views. There are other examples and we will discuss them throughout this chapter.

The ideas behind Observer are the same as those behind the separation of concerns principle, that is, to increase decoupling between the publisher and subscribers, and to make it easy to add/remove subscribers at runtime.

In this chapter, we will discuss:

- Real-world examples
- Use cases
- Implementation

Real-world examples

In reality, an auction resembles the Observer pattern. Every auction bidder has a number paddle that is raised whenever they want to place a bid. Whenever the paddle is raised by a bidder, the auctioneer acts as the subject by updating the price of the bid and broadcasting the new price to all bidders (subscribers).

In software, we can cite at least two examples:

- Kivy, the Python Framework for developing user interfaces, has a module called **Properties**, which implements the Observer pattern. Using this technique, you can specify what should happen when a property's value changes.
- The RabbitMQ library can be used to add asynchronous messaging support to an application. Several messaging protocols are supported, such as HTTP and AMQP. RabbitMQ can be used in a Python application to implement a publish-subscribe pattern, which is nothing more than the Observer design pattern (`j.mp/rabbitmqobs`).

Use cases

We generally use the Observer pattern when we want to inform/update *one or more objects* (observers/subscribers) about a change that happened on *a given object* (subject/publisher/observable). The number of observers, as well as who those observers are may vary and can be changed dynamically.

We can think of many cases where Observer can be useful. One such use case is **news feeds**. With RSS, Atom, or other related formats, you follow a feed, and every time it is updated, you receive a notification about the update.

The same concept exists in social networking. If you are connected to another person using a social networking service, and your connection updates something, you are notified about it. It doesn't matter if the connection is a Twitter user that you follow, a real friend on Facebook, or a business colleague on LinkedIn.

Event-driven systems are another example where Observer is usually used. In such systems, you have *listeners* that *listen* for specific events. The listeners are triggered when an event they are listening to is created. This can be typing a specific key (on the keyboard), moving the mouse, and more. The event plays the role of the publisher and the listeners play the role of the observers. The key point in this case is that multiple listeners (observers) can be attached to a single event (publisher).

Implementation

In this section, we will implement a data formatter. The ideas described here are based on the `ActiveState Python Observer` code recipe (<https://code.activestate.com/>). There is a default formatter that shows a value in the decimal format. However, we can add/register more formatters. In this example, we will add a hex and binary formatter. Every time the value of the default formatter is updated, the registered formatters are notified and take action. In this case, the action is to show the new value in the relevant format.

`Observer` is actually one of the patterns where inheritance makes sense. We can have a base `Publisher` class that contains the common functionality of adding, removing, and notifying observers. Our `DefaultFormatter` class derives from `Publisher` and adds the formatter-specific functionality. And, we can dynamically add and remove observers on demand.

We begin with the `Publisher` class. The observers are kept in the `observers` list. The `add()` method registers a new observer, or throws an error if it already exists. The `remove()` method unregisters an existing observer, or throws an exception if it does not exist. Finally, the `notify()` method informs all observers about a change:

```
class Publisher:
    def __init__(self):
        self.observers = []

    def add(self, observer):
        if observer not in self.observers:
            self.observers.append(observer)
        else:
            print(f'Failed to add: {observer}')

    def remove(self, observer):
        try:
            self.observers.remove(observer)
        except ValueError:
            print(f'Failed to remove: {observer}')

    def notify(self):
        [o.notify(self) for o in self.observers]
```

Let's continue with the `DefaultFormatter` class. The first thing that its `__init__()` does is call the `__init__()` method of the base class, since this is not done automatically in Python.

A `DefaultFormatter` instance has a name to make it easier for us to track its status. We use name mangling in the `_data` variable to state that it should not be accessed directly. Note that this is always possible in Python but fellow developers have no excuse for doing so, since the code already states that they shouldn't. There is a serious reason for using name mangling in this case. Stay tuned. `DefaultFormatter` treats the `_data` variable as an integer, and the default value is zero:

```
class DefaultFormatter(Publisher):
    def __init__(self, name):
        Publisher.__init__(self)
        self.name = name
        self._data = 0
```

The `__str__()` method returns information about the name of the publisher and the value of the `_data` attribute. `type(self).__name__` is a handy trick to get the name of a class without hardcoding it. It is one of those tricks that makes your code easier to maintain:

```
def __str__(self):
    return f"{type(self).__name__}: '{self.name}' has data = {self._data}"
```

There are two `data()` methods. The first one uses the `@property` decorator to give read access to the `_data` variable. Using this, we can just execute `object.data` instead of `object.data()`:

```
@property
def data(self):
    return self._data
```

The second `data()` method is more interesting. It uses the `@setter` decorator, which is called every time the assignment (`=`) operator is used to assign a new value to the `_data` variable. This method also tries to cast a new value to an integer, and does exception handling in case this operation fails:

```
@data.setter
def data(self, new_value):
    try:
        self._data = int(new_value)
    except ValueError as e:
        print(f'Error: {e}')
    else:
        self.notify()
```

The next step is to add the observers. The functionality of `HexFormatter` and `BinaryFormatter` is very similar. The only difference between them is how they format the value of data received by the publisher, that is, in hexadecimal and binary, respectively:

```
class HexFormatterObs:
    def notify(self, publisher):
        value = hex(publisher.data)
        print(f"{type(self).__name__}: '{publisher.name}' has now hex data
= {value}")

class BinaryFormatterObs:
    def notify(self, publisher):
        value = bin(publisher.data)
        print(f"{type(self).__name__}: '{publisher.name}' has now bin data
= {value}")
```

To help us use those classes, the `main()` function initially creates a `DefaultFormatter` instance named `test1` and, afterwards, attaches (and detaches) the two available observers. We also have some exception handling to make sure that the application does not crash when erroneous data is passed by the user.

The code is as follows:

```
def main():
    df = DefaultFormatter('test1')
    print(df)

    print()
    hf = HexFormatterObs()
    df.add(hf)
    df.data = 3
    print(df)

    print()
    bf = BinaryFormatterObs()
    df.add(bf)
    df.data = 21
    print(df)
```

Moreover, tasks such as trying to add the same observer twice or removing an observer that does not exist should cause no crashes:

```
print()
df.remove(hf)
df.data = 40
print(df)
```

```

print()
df.remove(hf)
df.add(bf)

df.data = 'hello'
print(df)

print()
df.data = 15.8
print(df)

```

Let's now recapitulate the full code of the example (the `observer.py` file):

1. We define the Publisher class:

```

class Publisher:
    def __init__(self):
        self.observers = []
    def add(self, observer):
        if observer not in self.observers:
            self.observers.append(observer)
        else:
            print(f'Failed to add: {observer}')
    def remove(self, observer):
        try:
            self.observers.remove(observer)
        except ValueError:
            print(f'Failed to remove: {observer}')
    def notify(self):
        [o.notify(self) for o in self.observers]

```

2. We define the DefaultFormatter class, with its special `__init__` and `__str__` methods:

```

class DefaultFormatter(Publisher):
    def __init__(self, name):
        Publisher.__init__(self)
        self.name = name
        self._data = 0
    def __str__(self):
        return f"{type(self).__name__}: '{self.name}' has data = {self._data}"

```

3. We add the data property getter and setter methods to the DefaultFormatter class:

```

@property
def data(self):

```

```

        return self._data
    @data.setter
    def data(self, new_value):
        try:
            self._data = int(new_value)
        except ValueError as e:
            print(f'Error: {e}')
        else:
            self.notify()

```

4. We define our two observer classes, as follows:

```

class HexFormatterObs:
    def notify(self, publisher):
        value = hex(publisher.data)
        print(f"{type(self).__name__}: '{publisher.name}' has now  
hex data = {value}")
class BinaryFormatterObs:
    def notify(self, publisher):
        value = bin(publisher.data)
        print(f"{type(self).__name__}: '{publisher.name}' has now  
bin data = {value}")

```

5. Now, we take care of the main part of the program; the first part of the `main()` function is as follows:

```

def main():
    df = DefaultFormatter('test1')
    print(df)
    print()
    hf = HexFormatterObs()
    df.add(hf)
    df.data = 3
    print(df)
    print()
    bf = BinaryFormatterObs()
    df.add(bf)
    df.data = 21
    print(df)

```

6. Here is the end of the `main()` function:

```

print()
df.remove(hf)
df.data = 40
print(df)
print()
df.remove(hf)

```

```
df.add(bf)
df.data = 'hello'
print(df)
print()
df.data = 15.8
print(df)
```

7. We do not forget the usual snippet that calls the `main()` function:

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

Executing the `python observer.py` command gives the following output:

```
DefaultFormatter: 'test1' has data = 0
HexFormatterObs: 'test1' has now hex data = 0x3
DefaultFormatter: 'test1' has data = 3
HexFormatterObs: 'test1' has now hex data = 0x15
BinaryFormatterObs: 'test1' has now bin data = 0b10101
DefaultFormatter: 'test1' has data = 21
BinaryFormatterObs: 'test1' has now bin data = 0b101000
DefaultFormatter: 'test1' has data = 40
Failed to remove: <__main__.HexFormatterObs object at 0x0000023707F90D30>
Failed to add: <__main__.BinaryFormatterObs object at 0x0000023707F90D68>
Error: invalid literal for int() with base 10: 'hello'
DefaultFormatter: 'test1' has data = 40
BinaryFormatterObs: 'test1' has now bin data = 0b1111
DefaultFormatter: 'test1' has data = 15
```

What we see in the output is that as the extra observers are added, more (and relevant) output is shown, and when an observer is removed, it is not notified any longer. That's exactly what we want: runtime notifications that we are able to enable/disable on demand.

The defensive programming part of the application also seems to work fine. Trying to do funny things, such as removing an observer that does not exist or adding the same observer twice, is not allowed. The messages shown are not very user-friendly, but I leave that up to you as an exercise. Runtime failures such as trying to pass a string when the API expects a number are also properly handled without causing the application to crash/terminate.

This example would be much more interesting if it were interactive. Even a simple menu that allows the user to attach/detach observers at runtime and to modify the value of `DefaultFormatter` would be nice because the runtime aspect becomes much more visible. Feel free to do it.

Another nice exercise is to add more observers. For example, you can add an octal formatter, a Roman numeral formatter, or any other observer that uses your favorite representation. Be creative!

Summary

In this chapter, we covered the Observer design pattern. We use Observer when we want to be able to inform/notify all stakeholders (an object or a group of objects) when the state of an object changes. An important feature of Observer is that the number of subscribers/observers, as well as who the subscribers are, may vary and can be changed at runtime.

To understand Observer, you can think of an auction, with the bidders being the subscribers and the auctioneer being the publisher. This pattern is used quite a lot in the software world.

As specific examples of software using Observer, we mentioned the following:

- Kivy, the framework for developing innovative user interfaces, with its *Properties* concept and module.
- The Python bindings of RabbitMQ. We referred to a specific example of RabbitMQ used to implement the publish-subscribe (also known as Observer) pattern.

In the implementation example, we saw how to use Observer to create data formatters that can be attached and detached at runtime to enrich the behavior of an object. Hopefully, you will find the recommended exercises interesting.

The next chapter introduces the State design pattern, which can be used to implement a core computer science concept: state machines.

12

The State Pattern

In the previous chapter, we covered the Observer pattern, which is useful in a program to notify other objects when the state of a given object changes. Let's continue discovering those patterns proposed by the *Gang of Four*.

Object-oriented programming (OOP) focuses on maintaining the states of objects that interact with each other. A very handy tool to model **state transitions** when solving many problems is known as a **finite-state machine** (commonly called a **state machine**).

What's a state machine? A state machine is an abstract machine that has two key components, that is, **states** and **transitions**. A state is the current (active) status of a system. For example, if we have a radio receiver, two possible states for it are to be tuned to FM or AM. Another possible state is for it to be switching from one FM/AM radio station to another. A transition is a switch from one state to another. A transition is initiated by a triggering event or condition. Usually, an action or set of actions is executed before or after a transition occurs. Assuming that our radio receiver is tuned to the 107 FM station, an example of a transition is for the button to be pressed by the listener to switch it to 107.5 FM.

A nice feature of state machines is that they can be represented as **graphs** (called **state diagrams**), where each state is a node and each transition is an edge between two nodes.

State machines can be used to solve many kinds of problems, both non-computational and computational. Non-computational examples include vending machines, elevators, traffic lights, combination locks, parking meters, and automated gas pumps. Computational examples include game programming and other categories of computer programming, hardware design, protocol design, and programming language parsing.

Now we have an idea of what state machines are! But, how are state machines related to the State design pattern? It turns out that the State pattern is nothing more than a state machine applied to a particular software engineering problem [Gang of Four-95 book, page 342], [Python 3 Patterns, Recipes and Idioms by Bruce Eckel-08, page 151].

In this chapter, we will discuss:

- Real-world examples
- Use cases
- Implementation

Real-world examples

A snack vending machine is an example of the State pattern in everyday life. Vending machines have different states and react differently depending on the amount of money that we insert. Depending on our selection and the money we insert, the machine can do the following:

- Reject our selection because the product we requested is out of stock
- Reject our selection because the amount of money we inserted was not sufficient
- Deliver the product and give no change because we inserted the exact amount
- Deliver the product and return the change

There are, for sure, more possible states, but you get the point.

In the software category, we can think of the following examples:

- The `django-fsm` package is a third-party package that can be used to simplify the implementation and usage of state machines in the Django Framework (j.mp/django-fsm).
- Python offers more than one third-party package/module to use and implement state machines (j.mp/pyfsm). We will see how to use one of them in the implementation section.
- The **State Machine Compiler (SMC)** project (<http://smc.sourceforge.net>). With SMC, you can describe your state machine in a single-text file using a simple **domain-specific language (DSL)**, and it will generate the state machine's code automatically. The project claims that the DSL is so simple that you can write it as a one-to-one translation of a state diagram. I haven't tried it, but it sounds very interesting. SMC can generate code in a number of programming languages, including Python.

Use cases

The State pattern is applicable to many problems. All the problems that can be solved using state machines are good use cases for using the State pattern. An example we have already seen is the process model for an operating/embedded system.

Programming language compiler implementation is another good example. Lexical and syntactic analysis can use states to build abstract syntax trees.

Event-driven systems are yet another example. In an event-driven system, the transition from one state to another triggers an event/message. Many computer games use this technique. For example, a monster might move from the guard state to the attack state when the main hero approaches it.

To quote Thomas Jaeger:

"The state design pattern allows for full encapsulation of an unlimited number of states on a context for easy maintenance and flexibility."

Implementation

Let's write code that demonstrates how to create a state machine based on the state diagram shown earlier in this chapter. Our state machine should cover the different states of a process and the transitions between them.

The State design pattern is usually implemented using a parent `State` class that contains the common functionality of all the states, and a number of concrete classes derived from `State`, where each derived class contains only the state-specific required functionality. In my opinion, these are implementation details. The State pattern focuses on implementing a state machine. The core parts of a state machine are the states and transitions between the states. It doesn't matter how those parts are implemented.

To avoid reinventing the wheel, we can make use of the existing Python modules that not only help us create state machines, but also do it in a Pythonic way. A module that I find very useful is `state_machine`. Before going any further, if `state_machine` is not already installed on your system, you can install it using the `pip install state_machine` command.

The `state_machine` module is simple enough that no special introduction is required. We will cover most aspects of it while going through the code of the example.

Let's start with the `Process` class. Each created process has its own state machine. The first step to create a state machine using the `state_machine` module is to use the `@acts_as_state_machine` decorator:

```
@acts_as_state_machine
class Process:
```

Then, we define the states of our state machine. This is a one-to-one mapping of what we see in the state diagram. The only difference is that we should give a hint about the initial state of the state machine. We do that by setting the `initial` attribute value to `True`:

```
    created = State(initial=True)
    waiting = State()
    running = State()
    terminated = State()
    blocked = State()
    swapped_out_waiting = State()
    swapped_out_blocked = State()
```

Next, we are going to define the transitions. In the `state_machine` module, a transition is an instance of the `Event` class. We define the possible transitions using the arguments `from_states` and `to_state`:

```
    wait = Event(from_states=(created,
                               running,
                               blocked,
                               swapped_out_waiting),
                 to_state=waiting)
    run = Event(from_states=waiting,
               to_state=running)
    terminate = Event(from_states=running,
                     to_state=terminated)
    block = Event(from_states=(running,
                               swapped_out_blocked),
                 to_state=blocked)
    swap_wait = Event(from_states=waiting,
                    to_state=swapped_out_waiting)
    swap_block = Event(from_states=blocked,
                     to_state=swapped_out_blocked)
```

Also, as you may have noted, `from_states` can be either a single state or a group of states (tuple).

Each process has a name. Officially, a process needs to have much more information to be useful (for example, ID, priority, status, and so forth) but let's keep it simple to focus on the pattern:

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

Transitions are not very useful if nothing happens when they occur. The `state_machine` module provides us with the `@before` and `@after` decorators that can be used to execute actions before or after a transition occurs, respectively. You can imagine updating some object(s) within the system or sending an email or a notification to someone. For the purpose of this example, the actions are limited to printing information about the state change of the process, as follows:

```
@after('wait')
def wait_info(self):
    print(f'{self.name} entered waiting mode')

@after('run')
def run_info(self):
    print(f'{self.name} is running')

@before('terminate')
def terminate_info(self):
    print(f'{self.name} terminated')

@after('block')
def block_info(self):
    print(f'{self.name} is blocked')

@after('swap_wait')
def swap_wait_info(self):
    print(f'{self.name} is swapped out and waiting')

@after('swap_block')
def swap_block_info(self):
    print(f'{self.name} is swapped out and blocked')
```

Next, we need the `transition()` function, which accepts three arguments:

- `process`, which is an instance of `Process`
- `event`, which is an instance of `Event` (`wait`, `run`, `terminate`, and so forth)
- `event_name`, which is the name of the event

And the name of the event is printed if something goes wrong when trying to execute event.

Here is the code for the function:

```
def transition(process, event, event_name):
    try:
        event()
    except InvalidStateTransition as err:
        print(f'Error: transition of {process.name}
              from {process.current_state} to {event_name} failed')
```

The `state_info()` function shows some basic information about the current (active) state of the process:

```
def state_info(process):
    print(f'state of {process.name}: {process.current_state}')
```

At the beginning of the `main()` function, we define some string constants, which are passed as `event_name`:

```
def main():
    RUNNING = 'running'
    WAITING = 'waiting'
    BLOCKED = 'blocked'
    TERMINATED = 'terminated'
```

Next, we create two `Process` instances and display information about their initial state:

```
p1, p2 = Process('process1'), Process('process2')
[state_info(p) for p in (p1, p2)]
```

The rest of the function experiments with different transitions. Recall the state diagram we covered in this chapter. The allowed transitions should be with respect to the state diagram. For example, it should be possible to switch from a running state to a blocked state, but it shouldn't be possible to switch from a blocked state to a running state:

```
print()
transition(p1, p1.wait, WAITING)
transition(p2, p2.terminate, TERMINATED)
[state_info(p) for p in (p1, p2)]
print()
transition(p1, p1.run, RUNNING)
transition(p2, p2.wait, WAITING)
[state_info(p) for p in (p1, p2)]
print()
transition(p2, p2.run, RUNNING)
[state_info(p) for p in (p1, p2)]
print()
[transition(p, p.block, BLOCKED) for p in (p1, p2)]
[state_info(p) for p in (p1, p2)]
```

```

print()
[transition(p, p.terminate, TERMINATED) for p in (p1, p2)]
[state_info(p) for p in (p1, p2)]

```

Here is the full code of the example (the `state.py` file):

1. We begin by importing what we need from `state_machine`:

```

from state_machine import (State, Event, acts_as_state_machine,
                           after, before, InvalidStateTransition)

```

2. We define the `Process` class with its simple attributes:

```

@acts_as_state_machine
class Process:
    created = State(initial=True)
    waiting = State()
    running = State()
    terminated = State()
    blocked = State()
    swapped_out_waiting = State()
    swapped_out_blocked = State()
    wait = Event(from_states=(created,
                               running,
                               blocked,
                               swapped_out_waiting),
                 to_state=waiting)
    run = Event(from_states=waiting,
               to_state=running)
    terminate = Event(from_states=running,
                     to_state=terminated)
    block = Event(from_states=(running,
                               swapped_out_blocked),
                 to_state=blocked)
    swap_wait = Event(from_states=waiting,
                    to_state=swapped_out_waiting)
    swap_block = Event(from_states=blocked,
                     to_state=swapped_out_blocked)

```

3. We add the `Process` class's initialization method:

```

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

```

4. We also need to define, on the `Process` class, the methods to provide its states:

```

@after('wait')
def wait_info(self):

```

```

        print(f'{self.name} entered waiting mode')
    @after('run')
    def run_info(self):
        print(f'{self.name} is running')
    @before('terminate')
    def terminate_info(self):
        print(f'{self.name} terminated')
    @after('block')
    def block_info(self):
        print(f'{self.name} is blocked')
    @after('swap_wait')
    def swap_wait_info(self):
        print(f'{self.name} is swapped out and waiting')
    @after('swap_block')
    def swap_block_info(self):
        print(f'{self.name} is swapped out and blocked')

```

5. We define the transition() function:

```

def transition(process, event, event_name):
    try:
        event()
    except InvalidStateTransition as err:
        print(f'Error: transition of {process.name}
              from {process.current_state} to {event_name} failed')

```

6. Next, we define the state_info() function:

```

def state_info(process):
    print(f'state of {process.name}: {process.current_state}')

```

7. Finally, here is the main part of the program:

```

def main():
    RUNNING = 'running'
    WAITING = 'waiting'
    BLOCKED = 'blocked'
    TERMINATED = 'terminated'
    p1, p2 = Process('process1'), Process('process2')
    [state_info(p) for p in (p1, p2)]
    print()
    transition(p1, p1.wait, WAITING)
    transition(p2, p2.terminate, TERMINATED)
    [state_info(p) for p in (p1, p2)]
    print()
    transition(p1, p1.run, RUNNING)
    transition(p2, p2.wait, WAITING)
    [state_info(p) for p in (p1, p2)]

```

```

    print()
    transition(p2, p2.run, RUNNING)
    [state_info(p) for p in (p1, p2)]
    print()
    [transition(p, p.block, BLOCKED) for p in (p1, p2)]
    [state_info(p) for p in (p1, p2)]
    print()
    [transition(p, p.terminate, TERMINATED) for p in (p1, p2)]
    [state_info(p) for p in (p1, p2)]
if __name__ == '__main__':
    main()

```

Here's what we get when executing `python state.py`:

```

state of process1: created
state of process2: created

process1 entered waiting mode
Error: transition of process2 from created to terminated failed
state of process1: waiting
state of process2: created

process1 is running
process2 entered waiting mode
state of process1: running
state of process2: waiting

process2 is running
state of process1: running
state of process2: running

process1 is blocked
process2 is blocked
state of process1: blocked
state of process2: blocked

Error: transition of process1 from blocked to terminated failed
Error: transition of process2 from blocked to terminated failed
state of process1: blocked
state of process2: blocked

```

Indeed, the output shows that illegal transitions such as `created → terminated` and `blocked → terminated` fail gracefully. We don't want the application to crash when an illegal transition is requested, and this is handled properly by the `except block`.

Notice how using a good module such as `state_machine` eliminates conditional logic. There's no need to use long and error-prone `if...else` statements that check for each and every state transition and react to them.

To get a better feeling for the State pattern and state machines, I strongly recommend you implement your own example. This can be anything: a simple video game (you can use state machines to handle the states of the main hero and the enemies), an elevator, a parser, or any other system that can be modeled using state machines.

Summary

In this chapter, we covered the State design pattern. The State pattern is an implementation of one or more finite-state machines (in short, state machines) used to solve a particular software-engineering problem.

A state machine is an abstract machine with two main components: states and transitions. A state is the current status of a system. A state machine can have only one active state at any point in time. A transition is a switch from the current state to a new state. It is normal to execute one or more actions before or after a transition occurs. State machines can be represented visually using state diagrams.

State machines are used to solve many computational and non-computational problems. Some of them are traffic lights, parking meters, hardware designing, programming language parsing, and so forth. We saw how a snack vending machine relates to the way a state machine works.

Modern software offers libraries/modules to make the implementation and usage of state machines easier. Django offers the third-party `django-fsm` package and Python also has many contributed modules. In fact, one of them (`state_machine`) was used in the implementation section. The State Machine Compiler is yet another promising project, offering many programming language bindings (including Python).

We saw how to implement a state machine for a computer system process using the `state_machine` module. The `state_machine` module simplifies the creation of a state machine and the definition of actions before/after transitions.

In the next chapter, we will discuss other behavioral design patterns: Interpreter, Strategy, Mememto, Iterator, and Template.

13

Other Behavioral Patterns

We have seen in Chapter 12, *The State pattern*, which helps us in making behavior changes when an object's internal state changes, by using state machines. There is a number of behavioral patterns and, in this chapter, we are going to discuss five more of them: Interpreter, Strategy, Memento, Iterator, and Template.

What is the Interpreter pattern? The Interpreter pattern is interesting for the advanced users of an application. The main idea behind this pattern is to give the ability to non-beginner users and domain experts to use a simple language, to get more productive in doing what they need to with the application.

What is the Strategy pattern? The Strategy pattern promotes using multiple algorithms to solve a problem. For example, if you have two algorithms to solve a problem with some difference in performance depending on the input data, you can use strategy to decide which algorithm to use based on the input data at runtime.

What is the Memento pattern? The Memento pattern helps add support for **Undo** and/or **History** in an application. When implemented, for a given object, the user is able to restore a previous state that was created and kept for later possible use.

What is the Iterator pattern? The Iterator pattern offers an efficient way to handle a container of objects and traverse to these members one at a time, using the famous *next* semantic. It is really useful since, in programming, we use sequences and collections of objects a lot, particularly in algorithms.

What is the Template pattern? The Template pattern focuses on eliminating code redundancy. The idea is that we should be able to redefine certain parts of an algorithm without changing its structure.

In this chapter, we will discuss the following:

- The Interpreter pattern
- The Strategy pattern
- The Memento pattern
- The Iterator pattern
- The Template pattern

Interpreter pattern

Usually, what we want to create is a **domain-specific language (DSL)**. A DSL is a computer language of limited expressiveness targeting a particular domain. DSLs are used for different things, such as combat simulation, billing, visualization, configuration, communication protocols, and so on. DSLs are divided into internal DSLs and external DSLs, j.mp/wikidsl, and j.mp/fowlerdsl.

Internal DSLs are built on top of a host programming language. An example of an internal DSL is a language that solves linear equations using Python. The advantages of using an internal DSL are that we don't have to worry about creating, compiling, and parsing grammar because these are already taken care of by the host language. The disadvantage is that we are constrained by the features of the host language. It is very challenging to create an expressive, concise, and fluent internal DSL if the host language does not have these features j.mp/jwodsl.

External DSLs do not depend on host languages. The creator of the DSL can decide all aspects of the language (grammar, syntax, and so forth), but they are also responsible for creating a parser and compiler for it. Creating a parser and compiler for a new language can be a very complex, long, and painful procedure j.mp/jwodsl.

The interpreter pattern is related only to internal DSLs. Therefore, our goal is to create a simple but useful language using the features provided by the host programming language, which in this case is Python. Note that Interpreter does not address parsing at all. It assumes that we already have the parsed data in some convenient form. This can be an **abstract syntax tree (AST)** or any other handy data structure [Gang of Four-95 book, page 276].

Real-world examples

A musician is an example of the Interpreter pattern in reality. Musical notation represents the pitch and duration of a sound graphically. The musician is able to reproduce a sound precisely based on its notation. In a sense, musical notation is the language of music, and the musician is the interpreter of that language.

We can also cite software examples:

- In the C++ world, `boost::spirit` is considered an internal DSL for implementing parsers.
- An example in Python is PyT, an internal DSL to generate (X)HTML. PyT focuses on performance and claims to have comparable speed with Jinja2.jmp/ghpyt. Of course, we should not assume that the Interpreter pattern is necessarily used in PyT. However, since it is an internal DSL, the Interpreter is a very good candidate for it.

Use cases

The Interpreter pattern is used when we want to offer a simple language to domain experts and advanced users to solve their problems. The first thing we should stress is that the interpreter should only be used to implement simple languages. If the language has the requirements of an external DSL, there are better tools to create languages from scratch (Yacc and Lex, Bison, ANTLR, and so on).

Our goal is to offer the right programming abstractions to the specialist, who is often not a programmer, to make them productive. Ideally, they shouldn't know advanced Python to use our DSL, but knowing even a little bit of Python is a plus since that's what we eventually get at the end. Advanced Python concepts should not be a requirement. Moreover, the performance of the DSL is usually not an important concern. The focus is on offering a language that hides the peculiarities of the host language and offers a more human-readable syntax. Admittedly, Python is already a very readable language with far less peculiar syntax than many other programming languages.

Implementation

Let's create an internal DSL to control a smart house. This example fits well into the **Internet of Things (IoT)** era, which is getting more and more attention nowadays. The user is able to control their home using a very simple event notation. An event has the form of `command -> receiver -> arguments`. The `arguments` part is optional.

Not all events require arguments. An example of an event that does not require any arguments is shown here:

```
open -> gate
```

An example of an event that requires arguments is shown here:

```
increase -> boiler temperature -> 3 degrees
```

The `->` symbol is used to mark the end of one part of an event and state the beginning of the next one. There are many ways to implement an internal DSL. We can use plain old regular expressions, string processing, a combination of operator overloading, and metaprogramming, or a library/tool that can do the hard work for us. Although officially, the interpreter does not address parsing, I feel that a practical example needs to cover parsing as well. For this reason, I decided to use a tool to take care of the parsing part. The tool is called **Pyparsing** and, to find out more about it, check the mini-book *Getting Started with Pyparsing* by Paul McGuire. If Pyparsing is not already installed on your system, you can install it using the `pip install pyparsing` command.

Before getting into coding, it is a good practice to define a simple grammar for our language. We can define the grammar using the **Backus-Naur Form (BNF)** notation [`j.mp/bnfgram`]:

```
event ::= command token receiver token arguments
command ::= word+
word ::= a collection of one or more alphanumeric characters
token ::= ->
receiver ::= word+
arguments ::= word+
```

What the grammar basically tells us is that an event has the form of `command -> receiver -> arguments`, and that commands, receivers, and arguments have the same form: a group of one or more alphanumeric characters. If you are wondering about the necessity of the numeric part, it is included to allow us to pass arguments, such as three degrees at the `increase -> boiler temperature -> 3 degrees` command.

Now that we have defined the grammar, we can move on to converting it to actual code. Here's how the code looks:

```
word = Word(alphanums)
command = Group(OneOrMore(word))
token = Suppress("->")
device = Group(OneOrMore(word))
argument = Group(OneOrMore(word))
event = command + token + device + Optional(token + argument)
```

The basic difference between the code and grammar definition is that the code needs to be written in the bottom-up approach. For instance, we cannot use a word without first assigning it a value. `Suppress` is used to state that we want the `->` symbol to be skipped from the parsed results.

The full code of this implementation example (the `interpreter.py` file) uses many placeholder classes, but to keep you focused, I will first show a minimal version featuring only one class. Let's take a look at the `Boiler` class. A boiler has a default temperature of 83° Celsius. There are also two methods to increase and decrease the current temperature:

```
class Boiler:
    def __init__(self):
        self.temperature = 83 # in celsius
    def __str__(self):
        return f'boiler temperature: {self.temperature}'
    def increase_temperature(self, amount):
        print(f"increasing the boiler's temperature by {amount} degrees")
        self.temperature += amount
    def decrease_temperature(self, amount):
        print(f"decreasing the boiler's temperature by {amount} degrees")
        self.temperature -= amount
```

The next step is to add the grammar, which we already covered. We will also create a boiler instance and print its default state:

```
word = Word(alphanums)
command = Group(OneOrMore(word))
token = Suppress("->")
device = Group(OneOrMore(word))
argument = Group(OneOrMore(word))
event = command + token + device + Optional(token + argument)

boiler = Boiler()
print(boiler)
```

The simplest way to retrieve the parsed output of `pyparsing` is by using the `parseString()` method. The result is a `ParseResults` instance, which is actually a parse tree that can be treated as a nested list. For example, executing `print(event.parseString('increase -> boiler temperature -> 3 degrees'))` would give `[['increase'], ['boiler', 'temperature'], ['3', 'degrees']]` as a result.

So, in this case, we know that the first sublist is the command (increase), the second sublist is the receiver (boiler temperature), and the third sublist is the argument (3°). We can actually unpack the `ParseResults` instance, which gives us direct access to these three parts of the event. Having direct access means that we can match patterns to find out which method should be executed:

```
cmd, dev, arg = event.parseString('increase -> boiler temperature -> 3
degrees')
cmd_str = ' '.join(cmd)
dev_str = ' '.join(dev)

if 'increase' in cmd_str and 'boiler' in dev_str:
    boiler.increase_temperature(int(arg[0]))
print(boiler)
```

Executing the preceding code snippet (using `python boiler.py`, as usual) gives the following output:

```
boiler temperature: 83
increasing the boiler's temperature by 3 degrees
boiler temperature: 86
```

The full code (the `interpreter.py` file) is not very different from what I just described. It is just extended to support more events and devices. Let's present it now:

- First, we import all we need from `pyparsing`:

```
from pyparsing import Word, OneOrMore, Optional, Group, Suppress,
alphanums
```

- We define the `Gate` class:

```
class Gate:
    def __init__(self):
        self.is_open = False
    def __str__(self):
        return 'open' if self.is_open else 'closed'
    def open(self):
        print('opening the gate')
        self.is_open = True
    def close(self):
        print('closing the gate')
        self.is_open = False
```

- We define the Garage class:

```
class Garage:
    def __init__(self):
        self.is_open = False
    def __str__(self):
        return 'open' if self.is_open else 'closed'
    def open(self):
        print('opening the garage')
        self.is_open = True
    def close(self):
        print('closing the garage')
        self.is_open = False
```

- We define the Aircondition class:

```
class Aircondition:
    def __init__(self):
        self.is_on = False
    def __str__(self):
        return 'on' if self.is_on else 'off'
    def turn_on(self):
        print('turning on the air condition')
        self.is_on = True
    def turn_off(self):
        print('turning off the air condition')
        self.is_on = False
```

- We also introduce the Heating class:

```
class Heating:
    def __init__(self):
        self.is_on = False
    def __str__(self):
        return 'on' if self.is_on else 'off'
    def turn_on(self):
        print('turning on the heating')
        self.is_on = True
    def turn_off(self):
        print('turning off the heating')
        self.is_on = False
```

- We define the Boiler class, already presented:

```
class Boiler:
    def __init__(self):
        self.temperature = 83 # in celsius
    def __str__(self):
```



```

        return f'boiler temperature: {self.temperature}'
    def increase_temperature(self, amount):
        print(f"increasing the boiler's temperature by {amount}
degrees")
        self.temperature += amount
    def decrease_temperature(self, amount):
        print(f"decreasing the boiler's temperature by {amount}
degrees")
        self.temperature -= amount

```

- Lastly, we define the Fridge class:

```

class Fridge:
    def __init__(self):
        self.temperature = 2 # in celsius
    def __str__(self):
        return f'fridge temperature: {self.temperature}'
    def increase_temperature(self, amount):
        print(f"increasing the fridge's temperature by {amount}
degrees")
        self.temperature += amount
    def decrease_temperature(self, amount):
        print(f"decreasing the fridge's temperature by {amount}
degrees")
        self.temperature -= amount

```

- Next, let's see our main function, starting with its first part:

```

def main():
    word = Word(alphanums)
    command = Group(OneOrMore(word))
    token = Suppress("->")
    device = Group(OneOrMore(word))
    argument = Group(OneOrMore(word))
    event = command + token + device + Optional(token + argument)
    gate = Gate()
    garage = Garage()
    airco = Aircondition()
    heating = Heating()
    boiler = Boiler()
    fridge = Fridge()

```

- We prepare the parameters for tests we will be performing, using the following variables (tests, open_actions, and close_actions):

```

tests = ('open -> gate',
        'close -> garage',
        'turn on -> air condition',

```

```

        'turn off -> heating',
        'increase -> boiler temperature -> 5 degrees',
        'decrease -> fridge temperature -> 2 degrees')

open_actions = {'gate':gate.open,
                'garage':garage.open,
                'air condition':airco.turn_on,
                'heating':heating.turn_on,
                'boiler
temperature':boiler.increase_temperature,
                'fridge
temperature':fridge.increase_temperature}
close_actions = {'gate':gate.close,
                'garage':garage.close,
                'air condition':airco.turn_off,
                'heating':heating.turn_off,
                'boiler
temperature':boiler.decrease_temperature,
                'fridge
temperature':fridge.decrease_temperature}

```

- Now, we execute the test actions, using the following snippet (that ends the main function):

```

for t in tests:
    if len(event.parseString(t)) == 2: # no argument
        cmd, dev = event.parseString(t)
        cmd_str, dev_str = ' '.join(cmd), ' '.join(dev)
        if 'open' in cmd_str or 'turn on' in cmd_str:
            open_actions[dev_str]()
        elif 'close' in cmd_str or 'turn off' in cmd_str:
            close_actions[dev_str]()
    elif len(event.parseString(t)) == 3: # argument
        cmd, dev, arg = event.parseString(t)
        cmd_str, dev_str, arg_str = ' '.join(cmd), ' '.join(dev), ' '.join(arg)
        num_arg = 0
        try:
            num_arg = int(arg_str.split()[0]) # extract the
numeric part
        except ValueError as err:
            print(f"expected number but got: '{arg_str[0]}'")
        if 'increase' in cmd_str and num_arg > 0:
            open_actions[dev_str](num_arg)
        elif 'decrease' in cmd_str and num_arg > 0:
            close_actions[dev_str](num_arg)

```

- We add the snippet to call the `main` function:

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

Executing the `python interpreter.py` command gives the following output:

```
opening the gate  
closing the garage  
turning on the air condition  
turning off the heating  
increasing the boiler's temperature by 5 degrees  
decreasing the fridge's temperature by 2 degrees
```

If you want to experiment more with this example, I have a few suggestions for you. The first change that will make it much more interesting is to make it interactive. Currently, all the events are hardcoded in the tests tuple. However, the user wants to be able to activate events using an interactive prompt. Do not forget to check how sensitive `pyarsing` is regarding spaces, tabs, or unexpected input. For example, what happens if the user types `turn off -> heating 37`?

Strategy pattern

Most problems can be solved in more than one way. Take, for example, the sorting problem, which is related to putting the elements of a list in a specific order.

There are many sorting algorithms, and, in general, none of them is considered the best for all cases j.mp/algocomp.

There are different criteria that help us pick a sorting algorithm on a per-case basis. Some of the things that should be taken into account are listed here:

- **A number of elements that need to be sorted:** This is called the input size. Almost all the sorting algorithms behave fairly well when the input size is small, but only a few of them have good performance with a large input size.
- **The best/average/worst time complexity of the algorithm:** Time complexity is (roughly) the amount of time the algorithm takes to complete, excluding coefficients and lower-order terms. This is often the most usual criterion to pick an algorithm, although it is not always sufficient.

- **The space complexity of the algorithm:** Space complexity is (again roughly) the amount of physical memory needed to fully execute an algorithm. This is very important when we are working with big data or embedded systems, which usually have limited memory.
- **Stability of the algorithm:** An algorithm is considered stable when it maintains the relative order of elements with equal values after it is executed.
- **Code complexity of the algorithm:** If two algorithms have the same time/space complexity and are both stable, it is important to know which algorithm is easier to code and maintain.

There are possibly more criteria that can be taken into account. The important question is are we really forced to use a single sorting algorithm for all cases? The answer is of course not. A better solution is to have all the sorting algorithms available and using the mentioned criteria to pick the best algorithm for the current case. That's what the Strategy pattern is about.

The Strategy pattern promotes using multiple algorithms to solve a problem. Its killer feature is that it makes it possible to switch algorithms at runtime transparently (the client code is unaware of the change). So, if you have two algorithms and you know that one works better with small input sizes, while the other works better with large input sizes, you can use Strategy to decide which algorithm to use based on the input data at runtime.

Real-world examples

Reaching an airport to catch a flight is a good Strategy example used in real life:

- If we want to save money and we leave early, we can go by bus/train
- If we don't mind paying for a parking place and have our own car, we can go by car
- If we don't have a car but we are in a hurry, we can take a taxi

There are trade-offs between cost, time, convenience, and so forth.

In software, Python's `sorted()` and `list.sort()` functions are examples of the Strategy pattern. Both functions accept a named parameter `key`, which is basically the name of the function that implements a sorting strategy [Python 3 Patterns, Recipes, and Idioms, by Bruce Eckel-08, page 202].

Use cases

Strategy is a very generic design pattern with many use cases. In general, whenever we want to be able to apply different algorithms dynamically and transparently, Strategy is the way to go. By different algorithms, I mean different implementations of the same algorithm. This means that the result should be exactly the same, but each implementation has a different performance and code complexity (as an example, think of sequential search versus binary search).

We have already seen how Python and Java use the Strategy pattern to support different sorting algorithms. However, Strategy is not limited to sorting. It can also be used to create all kinds of different resource filters (authentication, logging, data compression, encryption, and so forth) [j.mp/javaxfilter].

Another usage of the Strategy pattern is to create different formatting representations, either to achieve portability (for example, line-breaking differences between platforms) or dynamically change the representation of data.

Implementation

There is not much to be said about implementing the Strategy pattern. In languages where functions are not first-class citizens, each Strategy should be implemented in a different class. Wikipedia demonstrates that at j.mp/stratwiki. In Python, we can treat functions as normal variables and this simplifies the implementation of Strategy.

Assume that we are asked to implement an algorithm to check if all characters in a string are unique. For example, the algorithm should return `true` if we enter the `dream` string because none of the characters are repeated. If we enter the `pizza` string, it should return `false` because the letter `z` exists two times. Note that the repeated characters do not need to be consecutive, and the string does not need to be a valid word. The algorithm should also return `false` for the `1r2a3ae` string, because the letter `a` appears twice.

After thinking about the problem carefully, we come up with an implementation that sorts the string and compares all characters pair by pair. First, we implement the `pairs()` function, which returns all neighbor pairs of a sequence, `seq`:

```
def pairs(seq):
    n = len(seq)
    for i in range(n):
        yield seq[i], seq[(i + 1) % n]
```

Next, we implement the `allUniqueSort()` function, which accepts a string, `s`, and returns `True` if all characters in the string are unique; otherwise, it returns `False`. To demonstrate the Strategy pattern, we will make a simplification by assuming that this algorithm fails to scale. We assume that it works fine for strings that are up to five characters. For longer strings, we simulate a slowdown by inserting a `sleep` statement:

```
SLOW = 3                                # in seconds
LIMIT = 5                               # in characters
WARNING = 'too bad, you picked the slow algorithm :('
def allUniqueSort(s):
    if len(s) > LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    srtStr = sorted(s)
    for (c1, c2) in pairs(srtStr):
        if c1 == c2:
            return False
    return True
```

We are not happy with the performance of `allUniqueSort()`, and we are trying to think of ways to improve it. After some time, we come up with a new algorithm, `allUniqueSet()`, that eliminates the need to sort. In this case, we use a set. If the character in check has already been inserted in the set, it means that not all characters in the string are unique:

```
def allUniqueSet(s):
    if len(s) < LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    return True if len(set(s)) == len(s) else False
```

Unfortunately, while `allUniqueSet()` has no scaling problems, for some strange reason, it has worse performance than `allUniqueSort()` when checking short strings. What can we do in this case? Well, we can keep both algorithms and use the one that fits best, depending on the length of the string that we want to check.

The `allUnique()` function accepts an input string, `s`, and a strategy function, `strategy`, which in this case is one of `allUniqueSort()`, `allUniqueSet()`. The `allUnique()` function executes the input strategy and returns its result to the caller.

Then, the `main()` function lets the user perform the following:

- Enter the word to be checked for character uniqueness
- Choose the pattern that will be used

It also does some basic error handling and gives the ability to the user to quit gracefully:

```
def main():
    while True:
        word = None
        while not word:
            word = input('Insert word (type quit to exit)> ')
            if word == 'quit':
                print('bye')
                return
            strategy_picked = None
            strategies = { '1': allUniqueSet, '2': allUniqueSort }
            while strategy_picked not in strategies.keys():
                strategy_picked = input('Choose strategy: [1] Use a set,
[2] Sort and pair> ')
            try:
                strategy = strategies[strategy_picked]
                print(f'allUnique({word}): {allUnique(word,
strategy)})')
            except KeyError as err:
                print(f'Incorrect option: {strategy_picked}')
```

Here's the complete code of the example (the `strategy.py` file):

- We import the `time` module:

```
import time
```

- We define the `pairs()` function:

```
def pairs(seq):
    n = len(seq)
    for i in range(n):
        yield seq[i], seq[(i + 1) % n]
```

- We define the values for the `SLOW`, `LIMIT`, and `WARNING` constants:

```
SLOW = 3                                # in seconds
LIMIT = 5                               # in characters
WARNING = 'too bad, you picked the slow algorithm :('
```

- We define the function for the first algorithm, `allUniqueSort()`:

```
def allUniqueSort(s):
    if len(s) > LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    srtStr = sorted(s)
    for (c1, c2) in pairs(srtStr):
        if c1 == c2:
            return False
    return True
```

- We define the function for the second algorithm, `allUniqueSet()`:

```
def allUniqueSet(s):
    if len(s) < LIMIT:
        print(WARNING)
        time.sleep(SLOW)
    return True if len(set(s)) == len(s) else False
```

- Next, we define the `allUnique()` function that helps call a chosen algorithm by passing the corresponding strategy function:

```
def allUnique(word, strategy):
    return strategy(word)
```

- Now is the time for the `main()` function, followed by Python's script execution stanza:

```
def main():
    while True:
        word = None
        while not word:
            word = input('Insert word (type quit to exit)> ')
            if word == 'quit':
                print('bye')
                return
            strategy_picked = None
            strategies = { '1': allUniqueSet, '2': allUniqueSort }
            while strategy_picked not in strategies.keys():
                strategy_picked = input('Choose strategy: [1] Use a
set, [2] Sort and pair> ')
            try:
                strategy = strategies[strategy_picked]
                print(f'allUnique({word}): {allUnique(word,
strategy)})')
            except KeyError as err:
                print(f'Incorrect option: {strategy_picked}')
```



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

Let's see the output of a sample execution using the `python strategy.py` command:

```
Insert word (type quit to exit)> balloon  
Choose strategy: [1] Use a set, [2] Sort and pair> 2  
too bad, you picked the slow algorithm :(  
allUnique(balloon): False  
Insert word (type quit to exit)> bye  
Choose strategy: [1] Use a set, [2] Sort and pair> 1  
too bad, you picked the slow algorithm :(  
allUnique(bye): True  
Insert word (type quit to exit)> h  
Choose strategy: [1] Use a set, [2] Sort and pair> 1  
too bad, you picked the slow algorithm :(  
allUnique(h): True  
Insert word (type quit to exit)> h  
Choose strategy: [1] Use a set, [2] Sort and pair> 2  
allUnique(h): False  
Insert word (type quit to exit)>
```

The first word, `balloon`, has more than five characters and not all of them are unique. In this case, both algorithms return the correct result, `False`, but `allUniqueSort()` is slower and the user is warned.

The second word, `bye`, has less than five characters and all characters are unique. Again, both algorithms return the expected result, `True`, but this time, `allUniqueSet()` is slower and the user is warned once more.

Normally, the strategy that we want to use should not be picked by the user. The point of the Strategy pattern is that it makes it possible to use different algorithms transparently. Change the code so that the faster algorithm is always picked.

There are two usual users of our code. One is the end user, who should be unaware of what's happening in the code, and to achieve that we can follow the tips given in the previous paragraph. Another possible category of users is the other developers. Assume that we want to create an API that will be used by the other developers. How can we keep them unaware of the Strategy pattern? A tip is to think of encapsulating the two functions in a common class, for example, `allUnique`. In this case, the other developers will just need to create an instance of `allUnique` and execute a single method, for instance, `test()`. What needs to be done in this method?

Memento pattern

In many situations, we need a way to easily take a snapshot of the internal state of an object, so that we can restore the object with it when needed. Memento is a design pattern that can help us implement a solution for such situations.

The Memento design pattern has three key components:

- *Memento*, a simple object that contains basic state storage and retrieval capabilities
- *Originator*, an object that gets and sets values of Memento instances
- *Caretaker*, an object that can store and retrieve all previously created Memento instances

Memento shares many similarities with the Command pattern.

Real-world examples

The memento pattern can be seen in many situations in real life.

An example could be found in the dictionary we use for a language, such as English or French. The dictionary is regularly updated through the work of the academic experts, with new words being added and other words becoming obsolete. Spoken and written languages evolve and the official dictionary has to reflect that. From time to time, we revisit a previous edition to get an understanding of how the language was used at some point in the past. This could also be needed simply because information can be lost after a long period of time, and to find it, you may need to look into old editions. That can be useful to understand something in a particular field. Someone doing a research could use an old dictionary or go to the archives to find information about some words and expressions.

This example can be extended to other written material, such as books and newspapers.

Zope (<http://www.zope.org>), with its integrated *object database*, called **Zope Object Database (ZODB)**, offers a good software example of the Memento pattern. It is famous for its object, *Undo* support, exposed *Through The Web* for content managers (website administrators). ZODB is an object database for Python and is in heavy use in the Pyramid and Plone communities, and in many other applications.

Use cases

Memento is usually used when you need to provide some sort of *undo* and *redo* capability for your users.

Another usage is the implementation of a UI dialog with **Ok/Cancel** buttons, where we would store the state of the object on load, and if the user chooses to cancel, we would restore the initial state of the object.

Implementation

We will approach the implementation of Memento, in a simplified way, and by doing things in a natural way for the Python language. This means we do not necessarily need several classes.

One thing we will use is Python's *pickle* module. What is *pickle* used for? According to the module's documentation (<https://docs.python.org/3/library/pickle.html>), we can see that the pickle module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure.

Let's take a `Quote` class, with the attributes `text` and `author`. To create the memento, we will use a method on that class, `save_state()`, which as the name suggests will dump the state of the object, using the `pickle.dumps()` function. This creates *the memento*:

```
class Quote:

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

    def save_state(self):
        current_state = pickle.dumps(self.__dict__)
        return current_state
```

That state can be restored later. For that, we add the `restore_state()` method, making use of the `pickle.loads()` function:

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

Let's also add the `__str__` method:

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

Then, in the main function, we can take care of things and test our implementation, as usual:

```
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()

    q1.author = 'Marcus Tullius Cicero'
    print(f'\nWe found the author, and did an updated:\n{q1}')

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

    print()
    print('Quote 2')
    q2 = Quote("To be you in a world that is constantly trying to make you
    be something else is the greatest accomplishment.",
               'Ralph Waldo Emerson')
    print(f'\nOriginal version:\n{q2}')
    q2_mem1 = q2.save_state()

    q2.text = "To be yourself in a world that is constantly trying to make
    you something else is the greatest accomplishment."
    print(f'\nWe fixed the text:\n{q2}')
    q2_mem2 = q2.save_state()
    q2.text = "To be yourself when the world is constantly trying to make
    you something else is the greatest accomplishment."
    print(f'\nWe fixed the text again:\n{q2}')
    q2.restore_state(q2_mem2)
    print(f'\nWe had to restore the 2nd version, the correct one:\n{q2}')
```

Here's the complete code of the example (the `memento.py` file):

- We import the `pickle` module:

```
import pickle
```

- We define the Quote class:

```
class Quote:

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

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

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

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

- Here is our main function:

```
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 updated:\n{q1}')

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

    print()
    print('Quote 2')
    q2 = Quote("To be you in a world that is constantly trying to
make you be something else is the greatest accomplishment.",
                'Ralph Waldo Emerson')
    print(f'\nOriginal version:\n{q2}')
    q2_mem1 = q2.save_state()

    # changes to the text
    q2.text = "To be yourself in a world that is constantly trying
```

```

to make you something else is the greatest accomplishment."
    print(f'\nWe fixed the text:\n{q2}')
    q2_mem2 = q2.save_state()
    q2.text = "To be yourself when the world is constantly trying
to make you something else is the greatest accomplishment."
    print(f'\nWe fixed the text again:\n{q2}')
    # Restoring previous state (Undo)
    q2.restore_state(q2_mem2)
    print(f'\nWe had to restore the 2nd version, the correct
one:\n{q2}')
```

- Let's not forget the script execution stanza:

```

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

Let's view a sample execution using the `python memento.py` command:

```

Quote 1
Original version:
A room without books is like a body without a soul. - By Unknown author.

We found the author, and did an updated:
A room without books is like a body without a soul. - By Marcus Tullius Cicero.

We had to restore the previous version:
A room without books is like a body without a soul. - By Unknown author.

Quote 2
Original version:
To be you in a world that is constantly trying to make you be something else is the greatest accomplishment. - By Ralph Waldo Emerson.

We fixed the text:
To be yourself in a world that is constantly trying to make you something else is the greatest accomplishment. - By Ralph Waldo Emerson.

We fixed the text again:
To be yourself when the world is constantly trying to make you something else is the greatest accomplishment. - By Ralph Waldo Emerson.

We had to restore the 2nd version, the correct one:
To be yourself in a world that is constantly trying to make you something else is the greatest accomplishment. - By Ralph Waldo Emerson.
```

The output shows the program does what we expected: we can restore a previous state for each of our `Quote` objects.

Iterator pattern

In programming, we use sequences or collections of objects a lot, particularly in algorithms and when writing programs that manipulate data. One can think of automation scripts, APIs, data-driven apps, and other domains. In this chapter, we are going to see a pattern that is really useful whenever we have to handle collections of objects: the Iterator pattern.

Note the following, according to the definition given by Wikipedia:

Iterator is a design pattern in which an iterator is used to traverse a container and access the container's elements. The iterator pattern decouples algorithms from containers; in some cases, algorithms are necessarily container-specific and thus cannot be decoupled.

The iterator pattern is extensively used in the Python context. As we will see, this translates into iterator being a language feature. It is so useful that the language developers decided to make it a feature.

Real-world examples

Whenever you have a collection of things, and you have to go through the collection by taking those things one by one, it is an example of iterator pattern.

So, there are many examples in life, for instance as follows:

- A classroom where the teacher is going to each student to give them their textbook
- A waiter in a restaurant attending to people at a table, and taking the order of each person

What about software?

As we said, iteration is already in Python as a feature. We have iterable objects and iterators. *Container* or *sequence* types (*list*, *tuple*, *string*, *dictionary*, *set*, and so on) are *iterable*, meaning we can iterate through them. Iteration is done automatically for you whenever you use the `for` or `while` loop to traverse those objects and access their members.

But, we are not limited to that case. Python also has the built-in `iter()` function, which is a helper to transform any object in an iterator.

Use cases

It is a good idea to use the iterator pattern whenever you want one or several of the following behaviors:

- Make it easy to navigate through a collection
- Get the next object in the collection at any point
- Stop when you are done traversing through the collection

Implementation

Iterator is implemented in Python for us, within `for` loops, list comprehensions, and so on. Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.

We can do our own implementation for special cases, using the Iterator protocol, meaning that our iterator object must implement two special methods: `__iter__()` and `__next__()`.

An object is called *iterable* if we can get an iterator from it. Most of the built-in containers in Python (list, tuple, set, string, and so on) are iterable. The `iter()` function (which in turn calls the `__iter__()` method) returns an iterator from them.

Let's consider a football team we want to implement with the help of the `FootballTeam` class. If we want to make an iterator out of it, we have to implement the iterator protocol, since it is not a built-in container type such as the *list* type. Basically, built-in `iter()` and `next()` functions would not work on it unless they are added to the implementation.

First, we define the class of the iterator (`FootballTeamIterator`) that will be used to iterate through the football team object. The `members` attribute allows us to initialize the iterator object with our container object (which will be a `FootballTeam` instance):

```
class FootballTeamIterator:

    def __init__(self, members):
        self.members = members
        self.index = 0
```

We add a `__iter__()` method to it, which would return the object itself, and a `__next__()` method to return the next person from the team at each call until we reach the last person. These will allow looping over the members of the football team via the iterator:

```
def __iter__(self):
    return self

def __next__(self):
    if self.index < len(self.members):
        val = self.members[self.index]
        self.index += 1
        return val
    else:
        raise StopIteration()
```


So, now for the `FootballTeam` class itself; the new thing is adding a `__iter__()` method to it that will initialize the iterator object that it needs (thus using `FootballTeamIterator(self.members)`) and return it:

```
class FootballTeam:
    def __init__(self, members):
        self.members = members
    def __iter__(self):
        return FootballTeamIterator(self.members)
```

We add a small `main` function to test our implementation. Once we have a `FootballTeam` instance, we call the `iter()` function on it to create the iterator, and we loop through it using `while` loop:

```
def main():
    members = [f'player{str(x)}' for x in range(1, 23)]
    members = members + ['coach1', 'coach2', 'coach3']
    team = FootballTeam(members)
    team_it = iter(team)

    while True:
        print(next(team_it))
```

As a recap, here is the full code for our example (the `iterator.py` file):

- We define the class for the iterator:

```
class FootballTeamIterator:

    def __init__(self, members):
        self.members = members      # the list of players and the
staff
        self.index = 0
    def __iter__(self):
        return self
    def __next__(self):
        if self.index < len(self.members):
            val = self.members[self.index]
            self.index += 1
            return val
        else:
            raise StopIteration()
```

- We define the container class:

```
class FootballTeam:
    def __init__(self, members):
```

```
        self.members = members
    def __iter__(self):
        return FootballTeamIterator(self.members)
```

- We define our main function followed by the snippet to call it:

```
def main():
    members = [f'player{str(x)}' for x in range(1, 23)]
    members = members + ['coach1', 'coach2', 'coach3']
    team = FootballTeam(members)
    team_it = iter(team)

    while True:
        print(next(team_it))
if __name__ == '__main__':
    main()
```

Here is the output we get when executing `python iterator.py`:

```
player1
player2
player3
player4
player5
player6
player7
player8
player9
player10
player11
player12
player13
player14
player15
player16
player17
player18
player19
player20
player21
player22
coach1
coach2
coach3
Traceback (most recent call last):
  File "iterator.py", line 39, in <module>
    main()
  File "iterator.py", line 35, in main
    print(next(team_it))
  File "iterator.py", line 16, in __next__
    raise StopIteration()
StopIteration
```

We got the expected output, and we see the exception when we reach the end of the iteration.

Template pattern

A key ingredient in writing good code is avoiding redundancy. In **object-oriented programming (OOP)**, methods and functions are important tools that we can use to avoid writing redundant code.

Remember the `sorted()` example we saw when discussing the Strategy pattern. The `sorted()` function is generic enough that it can be used to sort more than one data structure (lists, tuples, and named tuples) using arbitrary keys. That's the definition of a good function.

Functions such as `sorted()` demonstrate the ideal case. In reality, we cannot always write one-hundred-percent generic code.

In the process of writing code that handles algorithms in the real world, we often end up writing redundant code. That's the problem solved by the Template design pattern. This pattern focuses on eliminating code redundancy. The idea is that we should be able to redefine certain parts of an algorithm without changing its structure.

Real-world examples

The daily routine of a worker, especially for workers of the same company, is very close to the Template design pattern. All workers follow more or less the same routine, but specific parts of the routine are very different.

In software, Python uses the Template pattern in the `cmd` module, which is used to build line-oriented command interpreters. Specifically, `cmd.Cmd.cmdloop()` implements an algorithm that reads input commands continuously and dispatches them to action methods. What is done before the loop, after the loop, and at the command parsing part are always the same. This is also called the *invariant part* of an algorithm. What changes are the actual action methods (the variant part) [j.mp/templatemart, page 27].

Another software example, the Python module `asyncore`, which is used to implement asynchronous socket service client/servers, also uses Template. Methods such as `asyncore.dispatcher.handle_connect()` and `asyncore.dispatcher.handle_write_event()` contain only generic code. To execute the socket-specific code, they execute the `handle_connect()` method. Note that what is executed is `handle_connect()` of a specific socket, not `asyncore.dispatcher.handle_connect()`, which actually contains only a warning. We can see that using the `inspect` module:

```
>>> python
import inspect
import asyncore
inspect.getsource(asyncore.dispatcher.handle_connect)
"    def handle_connect(self):n        self.log_info('unhandled connect
event', 'warning')n"
```

Use cases

The Template design pattern focuses on eliminating code repetition. If we notice that there is repeatable code in algorithms that have structural similarities, we can keep the invariant (common) parts of the algorithms in a template method/function and move the variant (different) parts in action/hook methods/functions.

Pagination is a good use case to use Template. A pagination algorithm can be split into an abstract (invariant) part and a concrete (variant) part. The invariant part takes care of things such as the maximum number of lines/page. The variant part contains functionality to show the header and footer of a specific page that is paginated [j.mp/templatemart, page 10].

All application frameworks make use of some form of the Template pattern. When we use a framework to create a graphical application, we usually inherit from a class and implement our custom behavior. However, before this, a Template method is usually called that implements the part of the application that is always the same, which is drawing the screen, handling the event loop, resizing and centralizing the window, and so on [Python 3 Patterns, Recipes and Idioms, by Bruce Eckel, page 133].

Implementation

In this example, we will implement a banner generator. The idea is rather simple. We want to send some text to a function, and the function should generate a banner containing the text. Banners have some sort of style, for example, dots or dashes surrounding the text. The banner generator has a default style, but we should be able to provide our own style.

The `generate_banner()` function is our Template function. It accepts, as an input, the text (`msg`) that we want our banner to contain, and the style (`style`) that we want to use. The `generate_banner()` function wraps the styled text with a simple header and footer. In reality, the header and footer can be much more complex, but nothing forbids us from calling functions that can do the header and footer generations instead of just printing simple strings:

```
def generate_banner(msg, style):
    print('-- start of banner --')
    print(style(msg))
    print('-- end of banner --nn')
```

The `dots_style()` function simply capitalizes `msg` and prints 10 dots before and after it:

```
def dots_style(msg):
    msg = msg.capitalize()
    msg = '.' * 10 + msg + '.' * 10
    return msg
```

Another style that is supported by the generator is `admire_style()`. This style shows the text in uppercase and puts an exclamation mark between each character of the text:

```
def admire_style(msg):
    msg = msg.upper()
    return '!'.join(msg)
```

The next style is by far my favorite. The `cow_style()` style executes the `milk_random_cow()` method of `cowpy`, which is used to generate a random ASCII art character every time `cow_style()` is executed. If `cowpy` is not already installed on your system, you can install it using the `pip install cowpy` command.

Here is the `cow_style()` function:

```
def cow_style(msg):
    msg = cow.milk_random_cow(msg)
    return msg
```

The `main()` function sends the text `happy coding` to the banner and prints it to the standard output using all the available styles:

```
def main():
    msg = 'happy coding'
    [generate_banner(msg, style) for style in (dots_style, admire_style,
        cow_style)]
```

The following is the full code of the example (the `template.py` file):

- We import the `cow` function from `cowpy`:

```
from cowpy import cow
```

- We define the `generate_banner()` function:

```
def generate_banner(msg, style):
    print('-- start of banner --')
    print(style(msg))
    print('-- end of banner --nn')
```

- We define the `dots_style()` function:

```
def dots_style(msg):
    msg = msg.capitalize()
    msg = '.' * 10 + msg + '.' * 10
    return msg
```

- Next, we define the `admire_style()` function:

```
def admire_style(msg):
    msg = msg.upper()
    return '!'.join(msg)
```

- Next, we define our last style function, `cow_style()`:

```
def cow_style(msg):
    msg = cow.milk_random_cow(msg)
    return msg
```

- We finish with the `main()` function and the snippet to call it:

```
def main():
    styles = (dots_style, admire_style, cow_style)
    msg = 'happy coding'
    [generate_banner(msg, style) for style in styles]
if __name__ == "__main__":
    main()
```

Let's take a look at a sample output by executing `python template.py`. (Your `cow_style()` output might be different due to the randomness of `cowpy`.):

```
-- start of banner --
.....Happy coding.....
-- end of banner --nn
-- start of banner --
H!A!P!P!Y! !C!O!D!I!N!G
-- end of banner --nn
-- start of banner --

< happy coding >
-----
 \      \  /\
  (      )
  .( o ).
-- end of banner --nn
```

Do you like the art generated by `cowpy`? I certainly do. As an exercise, you can create your own style and add it to the banner generator.

Another good exercise is to try implementing your own Template example. Find some existing redundant code that you wrote and see if the Template pattern is applicable.

Summary

In this chapter, we covered the interpreter, strategy, memento, iterator, and template design patterns.

The interpreter pattern is used to offer a programming-like framework to advanced users and domain experts, but without exposing the complexities of a programming language. This is achieved by implementing a DSL.

A DSL is a computer language that has limited expressiveness and targets a specific domain. There are two categories of DSLs: internal DSLs and external DSLs. While internal DSLs are built on top of a host programming language and rely on it, external DSLs are implemented from scratch and do not depend on an existing programming language. The interpreter is related only to internal DSLs.

Musical notation is an example of a non-software DSL. The musician acts as the Interpreter that uses the notation to produce music. From a software perspective, many Python template engines make use of Internal DSLs. PyT is a high-performance Python DSL to generate (X)HTML.

Although parsing is generally not addressed by the Interpreter pattern, in the implementation section, we used Pyparsing to create a DSL that controls a smart house and saw that using a good parsing tool makes **interpreting** the results using pattern matching simple.

Then, we saw the Strategy design pattern. The strategy pattern is generally used when we want to be able to use multiple solutions for the same problem transparently. There is no perfect algorithm for all input data and all cases, and by using Strategy, we can dynamically decide which algorithm to use in each case.

Sorting, encryption, compression, logging, and other domains that deal with resources use Strategy to provide different ways to filter data. Portability is another domain where Strategy is applicable. Simulations are yet another good candidate.

We saw how Python, with its first-class functions, simplifies the implementation of Strategy by implementing two different algorithms that check if all the characters in a word are unique.

Next, we saw the Memento pattern, used to store the state of an object when needed. Memento provides an efficient solution when implementing some sort of undo capability for your users. Another usage is the implementation of a UI dialog with **Ok/Cancel** buttons, where if the user chooses to cancel, we would restore the initial state of the object.

We used an example to get a feel for how Memento, in a simplified form and using the pickle module from the standard library, can be used in an implementation where we want to be able to restore previous states of data objects.

The Iterator pattern gives a nice and efficient way to iterate through sequences and collections of objects. In real life, whenever you have a collection of things and you are getting to those things one by one, you are using a form of the Iterator pattern.

In Python, Iterator is a language feature. We can use it immediately on built-in containers (iterables) such as lists and dictionaries, and we can define new iterable and iterator classes, to solve our problem, by using the Python iterator protocol. We saw that with an example of implementing a football team.

Lastly, we discussed the Template pattern. We use Template to eliminate redundant code when implementing algorithms with structural similarities.

We saw how the daily routine of a worker resembles the Template pattern. We also mentioned two examples of how Python uses Template in its libraries. General use cases of when to use Template were also mentioned.

We concluded by implementing a banner generator, which uses a Template function to implement custom text styles.

In the next chapter, we will cover the Observer pattern in reactive programming.

14

The Observer Pattern in Reactive Programming

In the previous chapter, we covered the last four in our list of behavioral patterns. That chapter also marked the end of the list of patterns presented by the *Gang of Four* in their book.

Among the patterns we have discussed so far, one is particularly interesting now, for this new chapter: the **Observer** pattern (covered in Chapter 11, *The Observer Pattern*), is useful for notifying an object or a group of objects when the state of a given object changes. This type of traditional Observer applies the publish-subscribe principle, allowing us to react to some object change events. It provides a nice solution for many cases, but in a situation where we have to deal with many events, some of them depending on each other, the traditional way could lead to complicated, difficult-to-maintain code. That is where another paradigm called **reactive programming** gives us an interesting option. In simple terms, the concept of reactive programming is to react to many events, **streams of events**, while keeping our code clean.

In this chapter, we will focus on a framework called **ReactiveX** (<http://reactivex.io>), part of reactive programming. The core entity in ReactiveX is called an **Observable**. And as we can see on the official website, ReactiveX is defined as an API for *asynchronous programming with observable streams*. In addition to that, we also have the Observer.

You can think of an Observable as a stream that can push or emit data to the Observer. And it can also emit events.

Here are two quotes from the documentation available, giving a definition of Observables:

"An Observable is the core type in ReactiveX. It serially pushes items, known as emissions, through a series of operators until it finally arrives at an Observer, where they are consumed."

"Push-based (rather than pull-based) iteration opens up powerful new possibilities to express code and concurrency much more quickly. Because an Observable treats events as data and data as events, composing the two together becomes trivial."

In this chapter, we will discuss:

- Real-world examples
- Use cases
- Implementation

Real-world examples

In real life, a stream of water that accumulates somewhere resembles an Observable.

We have quite a few examples in terms of software:

- A spreadsheet application can be seen as an example of reactive programming, based on its internal behavior. In virtually all spreadsheet applications, interactively changing any one cell in the sheet will result in immediately reevaluating all formulas that directly or indirectly depend on that cell and updating the display to reflect these reevaluations.
- The ReactiveX concept is implemented in a variety of languages, including Java (RxJava), Python (RxPY), and JavaScript (RxJS).
- The Angular Framework uses RxJS to implement the Observable pattern.

Use cases

One use case is the idea of the **Collection Pipeline** discussed by Martin Fowler on his blog (<https://martinfowler.com/articles/collection-pipeline>):

"Collection pipelines are a programming pattern where you organize some computation as a sequence of operations which compose by taking a collection as output of one operation and feeding it into the next."

We can use an Observable to do operations such as **map** and **reduce** or **groupBy** on sequences of objects when processing data.

Observables can be created for diverse functions such as button events, requests, and RSS or Twitter feeds.

Implementation

Instead of building a complete implementation here, we will approach different possibilities and see how you can use them.

To get started, install RxPY in your Python environment using the `pip install rx` command.

A first example

To start, let's take the example from the RxPY documentation and write a more fun variant. We are going to observe a stream built from the **Zen of Python** quotes by Tim Peters (<https://www.python.org/dev/peps/pep-0020/>).

Normally, you can see the quotes by using `import this` in the Python console, as you can see in the following screenshot of the console:

```
>>> import this
```

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

Then, the question is how to get that list of quotes from within a Python program. A quick search gives us an answer on Stack Overflow. Basically, you can redirect the result of the `import this` statement to an `io.StringIO` instance and then access it and print it using `print()`, as follows:

```
import contextlib, io
zen = io.StringIO()
with contextlib.redirect_stdout(zen):
    import this
print(zen.getvalue())
```

Now, to really be able to start using our example code (in the `rx_example1.py` file), we need to import `Observable` and `Observer` classes from the `rx` module, as follows:

```
from rx import Observable, Observer
```

Then, we create a function, `get_quotes()`, using the `contextlib.redirect_stdout()` trick, with the code we previously showed slightly adapted, to get and return the quotes:

```
def get_quotes():
    import contextlib, io
    zen = io.StringIO()
    with contextlib.redirect_stdout(zen):
        import this

    quotes = zen.getvalue().split('\n')[1:]
    return quotes
```

That is done! Now, we want to create an `Observable` from the list of quotes we get. The way we can do things as follows:

1. Define a function that hands data items to the `Observer`
2. Use an `Observable.create()` factory, and pass it that function, to set up the source or stream of data
3. Make the `Observer` subscribe to the source

The `Observer` class itself has three methods used for this type of communication:

- The `on_next()` is used to pass items
- The `on_completed()` will signal that no more items are coming
- The `on_error()` signals an error

So let's create a function, `push_quotes()`, that takes an `Observer` object `obs` as input, and, using the sequence of quotes, sends each quote using the `on_next()` and signals the end (after the last quote has been sent) using `on_completed()`. The function is as follows:

```
def push_quotes(obs):  
  
    quotes = get_quotes()  
    for q in quotes:  
        if q: # skip empty  
            obs.on_next(q)  
    obs.on_completed()
```



You can see, in this piece of code, the parallel with the **Iterator** pattern, where with **Python iterators**, the `next(iterator)` idiom gives us the next element in the iteration.

We implement the `Observer` to be used, using a subclass of the `Observer` base class, as follows:

```
class ZenQuotesObserver(Observer):  
  
    def on_next(self, value):  
        print(f"Received: {value}")  
  
    def on_completed(self):  
        print("Done!")  
  
    def on_error(self, error):  
        print(f"Error Occurred: {error}")
```

Next, we define *the source to be observed*, as follows:

```
source = Observable.create(push_quotes)
```

Finally, we define the subscription to the `Observable`, without which nothing would happen:

```
source.subscribe(ZenQuotesObserver())
```

Now we are ready to see the results of the code. Here's what we get when executing

`python rx_example1.py`:

```
Received: Beautiful is better than ugly.
Received: Explicit is better than implicit.
Received: Simple is better than complex.
Received: Complex is better than complicated.
Received: Flat is better than nested.
Received: Sparse is better than dense.
Received: Readability counts.
Received: Special cases aren't special enough to break the rules.
Received: Although practicality beats purity.
Received: Errors should never pass silently.
Received: Unless explicitly silenced.
Received: In the face of ambiguity, refuse the temptation to guess.
Received: There should be one-- and preferably only one --obvious way to do it.
Received: Although that way may not be obvious at first unless you're Dutch.
Received: Now is better than never.
Received: Although never is often better than *right* now.
Received: If the implementation is hard to explain, it's a bad idea.
Received: If the implementation is easy to explain, it may be a good idea.
Received: Namespaces are one honking great idea -- let's do more of those!
Done!
```

A second example

Let's see (in the `rx_example2.py` file) another way to write the code and obtain a similar result as in the first example.

We adopt the `get_quotes()` function in order to return an enumeration of the sequence, using Python's built-in `enumerate()` function, as follows:

```
def get_quotes():
    import contextlib, io
    zen = io.StringIO()
    with contextlib.redirect_stdout(zen):
        import this

    quotes = zen.getvalue().split('\n')[1:]
    return enumerate(quotes)
```

We can call that function and store its result in a variable, `zen_quotes`:

```
zen_quotes = get_quotes()
```

We create the Observable using the special `Observable.from_()` function and chain operations such as `filter()` on the sequence, and finally use `subscribe()` to subscribe to the Observable.



See <http://reactivex.io/documentation/operators.html> for all possible **operators** for Observables.

That last snippet of code is as follows:

```
Observable.from_(zen_quotes) \
    .filter(lambda q: len(q[1]) > 0) \
    .subscribe(lambda value: print(f"Received: {value[0]} - {value[1]}"))
```

Here's what we get when executing `python rx_example2.py`:

```
Received: 1 - Beautiful is better than ugly.
Received: 2 - Explicit is better than implicit.
Received: 3 - Simple is better than complex.
Received: 4 - Complex is better than complicated.
Received: 5 - Flat is better than nested.
Received: 6 - Sparse is better than dense.
Received: 7 - Readability counts.
Received: 8 - Special cases aren't special enough to break the rules.
Received: 9 - Although practicality beats purity.
Received: 10 - Errors should never pass silently.
Received: 11 - Unless explicitly silenced.
Received: 12 - In the face of ambiguity, refuse the temptation to guess.
Received: 13 - There should be one-- and preferably only one --obvious way to do it.
Received: 14 - Although that way may not be obvious at first unless you're Dutch.
Received: 15 - Now is better than never.
Received: 16 - Although never is often better than *right* now.
Received: 17 - If the implementation is hard to explain, it's a bad idea.
Received: 18 - If the implementation is easy to explain, it may be a good idea.
Received: 19 - Namespaces are one honking great idea -- let's do more of those!
```

A third example

Let's see a similar example (in the `rx_example3.py` file) where we react to the Observable (the stream of quotes created using the same `get_quotes()` function), using a chain of `flat_map()`, `filter()`, and `map()` operations.

The main difference from the previous example is that we schedule the streaming of items so that a new item is emitted every five seconds (**the interval**), using the `Observable.interval()` function. Furthermore, we use the `flat_map()` method to map each emission to an `Observable` (`Observable.from_(zen_quotes)` for example) and merge their emissions together into a single `Observable`.

The main part of the code is as follows:

```
Observable.interval(5000) \
    .flat_map(lambda seq: Observable.from_(zen_quotes)) \
    .flat_map(lambda q: Observable.from_(q[1].split())) \
    .filter(lambda s: len(s) > 2) \
    .map(lambda s: s.replace('.', '').replace(',', '').replace('!',
    '').replace('-', '')) \
    .map(lambda s: s.lower()) \
    .subscribe(lambda value: print(f"Received: {value}"))
```



According to the documentation (<http://reactivex.io/documentation/operators/interval.html>), the `Interval` operator returns an `Observable` that emits an infinite sequence of ascending integers, with a constant interval of time of your choosing between emissions. Note: To schedule the push of items with an interval of one second, you would use `Observable.interval(1000)`.

We also add the following line at the end, using the `input()` function, to make sure we can stop the execution when the user wants:

```
input("Starting... Press any key to quit\n")
```

Let's execute the example using the `python rx_example3.py` command. Every five seconds, we see new items in the console (which in this case are the words from the quotes that are of at least three characters, stripped from the punctuation characters, and converted into lowercase). Here is the output:

```
Starting... Press any key to quit
Received: beautiful
Received: explicit
Received: better
Received: simple
Received: than
Received: better
Received: complex
Received: ugly
Received: than
Received: better
Received: flat
Received: implicit
Received: than
Received: better
Received: sparse
Received: complex
Received: than
Received: better
Received: readability
Received: complicated
Received: than
```

Also, another output screenshot is as follows:

```
Received: guess
Received: obvious
Received: than
Received: hard
Received: one
Received: way
Received: first
Received: *right*
Received: easy
Received: honking
Received: unless
Received: now
Received: explain
Received: great
Received: you're
Received: it's
Received: explain
Received: idea
Received: it
Received: dutch
Received: bad
Received: may
Received: let's
Received: idea
Received: more
Received: good
Received: idea
Received: those
```

Since we may not have a huge number of words, in this case, it is relatively quick, but you can quit the program by typing any character (followed by the *Ctrl* key) and re-run it to get a better sense of what happens.

A fourth example

Let's build a stream of the list of people and an Observable based on it.

Here is another trick: To help you handle the data source, we will use a third-party module called **Faker** (<https://pypi.org/project/Faker>) to generate the names of people.

You can install the `Faker` module using the `pip install faker` command.

In the `peoplelist.py` file, we have the following code, with a `populate()` function that leverages a `Faker` instance to generate the first name and last name of fictitious people:

```
from faker import Faker
fake = Faker()

def populate():

    persons = []
    for _ in range(0, 20):
        p = {'firstname': fake.first_name(), 'lastname': fake.last_name()}
        persons.append(p)

    return iter(persons)
```

For the main part of the program, we write the names of the people in the list that was generated, in the text file `people.txt`:

```
if __name__ == '__main__':
    new_persons = populate()

    new_data = [f"{p['firstname']} {p['lastname']}" for p in new_persons]
    new_data = ", ".join(new_data) + ", "

    with open('people.txt', 'a') as f:
        f.write(new_data)
```

Before going further, let's pause and strategize! It seems a good idea to do things in an incremental way, since there are a lot of new concepts and techniques flying around. So we will work on a first implementation for our Observable, and later extend it.

In the `fx_peoplelist_1.py` file, let's write the first version of our code.

First, we define a function, `firstnames_from_db()`, which returns an Observable from the text file (reading the content of the file) containing the names, with transformations (as we have already seen) using `flat_map()`, `filter()`, and `map()` methods, and a new operation, `group_by()`, to emit items from another sequence—the first name found in the file, with its number of occurrence:

```
from rx import Observable

def firstnames_from_db(file_name):
    file = open(file_name)

    # collect and push stored people firstnames
    return Observable.from_(file) \
        .flat_map(lambda content: content.split(', ')) \
        .filter(lambda name: name!='') \
        .map(lambda name: name.split()[0]) \
        .group_by(lambda firstname: firstname) \
        .flat_map(lambda grp: grp.count().map(lambda ct: (grp.key, ct)))
```

Then we define an Observable, as in the previous example, which emits data every five seconds, merging its emission with what is returned from `firstnames_from_db(db_file)`, after setting the `db_file` to the `people.txt` file, as follows:

```
db_file = "people.txt"

# Emit data every 5 seconds
Observable.interval(5000) \
    .flat_map(lambda i: firstnames_from_db(db_file)) \
    .subscribe(lambda value: print(str(value)))

input("Starting... Press any key to quit\n")
```

Now, let's see what happens when we execute both programs (`peoplelist.py` and `rx_peoplelist_1.py`).

From one command-line window or terminal, you can generate the people's names by executing `python peoplelist.py`. The `people.txt` file is created with the names in it, separated by a comma. Each time you run that command again, a new set of names is added to the file.

From a second command-line window, you can run the first version of the program that implements Observable via the `python rx_peoplelist_1.py` command. You will get an output similar to this:

```
('Jerome', 1)
('Morgan', 1)
('Andrea', 1)
('Dylan', 1)
('Eric', 2)
('Erika', 1)
('Caleb', 1)
('Jerry', 1)
('Kaylee', 1)
('Laura', 1)
('Zachary', 1)
('Jay', 1)
('Jennifer', 1)
('Janet', 1)
('Stacy', 1)
('Dennis', 3)
('Cindy', 1)
('Mark', 1)
('Olivia', 1)
('Rebekah', 1)
('Taylor', 1)
('Samantha', 1)
('Evelyn', 1)
('Teresa', 1)
('Cynthia', 1)
('Lorraine', 1)
('Melissa', 1)
('Eddie', 1)
('Victor', 1)
('Jim', 1)
```

By re-executing the first command several times and monitoring what happens in the second window, we can see that the `people.txt` file is continuously read to extract the names, get the first names from those full names, and make the transformation needed to push the items consisting of the first names, each with their number of occurrences.



The grouping operation for the first names is done using the `group_by()` method of the `Observable` class.

To improve what we just achieved, we will try to get an emission of only the first names that are present at least four times. In the `rx_peoplelelist_2.py` file, we need another function to return an Observable and filter it that way. Let's call that function `frequent_firstnames_from_db()`. Compared to the one we used in the first version, we have to use the `filter()` operator to only keep the first name groups for which the count of occurrences (`ct`) value is bigger than three. If you check the code again, based on the group obtained, we get a tuple containing the group's key as the first element and the count as the second element using the Lambda function `lambda grp: grp.count().map(lambda ct: (grp.key, ct))` which is emitted thanks to the `.flat_map()` operator. So the next thing to do, in the function, is to further filter using `.filter(lambda name_and_ct: name_and_ct[1] > 3)` in order to only get first names that currently appear at least four times.

Here is the code for this new function:

```
def frequent_firstnames_from_db(file_name):
    file = open(file_name)

    # collect and push only the frequent firstnames
    return Observable.from_(file) \
        .flat_map(lambda content: content.split(', ')) \
        .filter(lambda name: name!='') \
        .map(lambda name: name.split()[0]) \
        .group_by(lambda firstname: firstname) \
        .flat_map(lambda grp: grp.count().map(lambda ct: (grp.key, ct))) \
        .filter(lambda name_and_ct: name_and_ct[1] > 3)
```

And we add almost the same code for the interval Observable; we only change the name of the referenced function accordingly. The new code for that last bit (as can be seen in the `rx_peoplelelist_2.py` file) is as follows:

```
# Emit data every 5 seconds
Observable.interval(5000) \
    .flat_map(lambda i: frequent_firstnames_from_db(db_file)) \
    .subscribe(lambda value: print(str(value)))

# Keep alive until user presses any key
input("Starting... Press any key to quit\n")
```

Using the same protocol to execute the example, when you run the `python rx_peoplelist_2.py` command in the second window or terminal, you should get an output similar to this:

```
('Robert', 7)
('Lisa', 6)
('Michael', 7)
('Steven', 4)
('Matthew', 5)
('James', 5)
('Mark', 4)
('Lauren', 4)
('Jessica', 4)
('Rachel', 4)
('Robert', 7)
('Lisa', 6)
('Michael', 7)
('Steven', 4)
('Matthew', 5)
('James', 5)
('Mark', 4)
('Lauren', 4)
('Jessica', 4)
('Rachel', 4)
('Robert', 7)
('Lisa', 6)
('Michael', 7)
('Steven', 4)
('Matthew', 5)
('James', 5)
('Mark', 4)
('Lauren', 4)
('Jessica', 4)
```

We can see the pairs of first names and counts emitted, and there is new emission every five seconds, and that values change whenever we re-run the `peoplelist` program (using `python peoplelist.py`) from the first shell window. Nice result!

Finally, in the `rx_peoplelist_3.py` file, we reuse most of the code to show just a variant of the previous example, with a small change: the use of the `distinct()` operation applied to the interval Observable, just before the Observer's subscription to it. The new version of that code snippet is as follows:

```
# Emit data every 5 seconds, but only if it changed
Observable.interval(5000) \
    .flat_map(lambda i: frequent_firstnames_from_db(db_file)) \
    .distinct() \
    .subscribe(lambda value: print(str(value)))
```

Similar to what we did before, when executing `python rx_people1ist_3.py` you should get an output similar to this:

```
('Rachel', 4)
('Robert', 7)
('Lisa', 6)
('Michael', 7)
('Steven', 4)
('Matthew', 5)
('James', 5)
('Mark', 4)
('Lauren', 4)
('Jessica', 4)
('Nicole', 4)
('Robert', 8)
('Steven', 5)
('Joseph', 4)
('Robert', 10)
('Charles', 4)
('Michael', 8)
('Christina', 5)
('Dennis', 4)
('Jessica', 5)
```

Do you notice the difference?

Again, we can see the emission of pairs of the first name and count. But this time, there is less change happening, and depending on the case, you may even have the impression that the data does not change by more than 10 seconds. To see more change, you have to re-run the `people1ist.py` program, maybe several times, so that the count of a few frequent first names can increment. The explanation to that behavior is that, since we added the `.distinct()` operation to the interval Observable, the value of an item is emitted only if it has changed. That's why we have less data being emitted, and for each first name, we do not see the same count twice.

With this examples series, we discovered how Observables offer a clever way to do things that are difficult to do using traditional patterns, and that is very nice! We just scratched the surface though, and this is an opportunity for the interested reader to pursue the exploration of ReactiveX and reactive programming.

Summary

In this chapter, we introduced the Observer pattern in reactive programming.

The core idea of this type of Observer pattern is to react to a stream of data and events, as with the stream of water we see in nature. We have lots of examples in the computing world of this idea or the ReactiveX technique through its extensions for the programming languages (RxJava, RxJS, RxPY, and so on). Modern JavaScript frameworks such as Angular are other examples we mentioned.

We have discussed examples of RxPY that can be used to build functionality, and which serve as an introduction for the reader to approach this programming paradigm and continue their own research via the RxPY official documentation and examples as well as existing code that one may find on GitHub.

In the next chapter, we will cover the Microservices pattern and other patterns for the Cloud.

Microservices and Patterns for the Cloud

Traditionally, developers working on building a server-side application have been using a single code base and implementing all or most functionalities right there, using common development practices such as functions and classes, and design patterns such as the ones we have covered in this book so far. But, with the evolution of the IT industry, economic factors, and pressure for fast times to market and returns on investment, there is a constant need to improve the practices of engineering teams and ensure more reactivity and scalability with servers, service delivery, and operations. We need to learn about other useful patterns, not only object-oriented programming ones. In this chapter, we will cover the last part of our exploration, focused on *Modern Architecture-style design patterns*.

One of the main additions to the catalog of patterns for engineers in recent years has been the *Microservice Architecture pattern* or *Microservices*. The idea is that we can build an application as a set of loosely coupled, collaborating services. In this architectural style, an application might consist of services such as the order management service, the customer management service, and so on. Services can be developed and deployed independently of one another.

Moreover, increasing numbers of applications are deployed in the cloud (AWS, Azure, Google Cloud, Digital Ocean, Heroku, and so on) and must be designed upfront with this type of environment and its constraints in mind. A number of patterns have emerged to help us deal with these new aspects of our work. Even the engineering teams, ways of working have evolved, hence the DevOps teams and roles we now have in many software development and production organizations. There are many of these cloud architecture related patterns, but we choose to focus on a few ones related to Microservices: *Retry*, *Circuit Breaker*, and *Cache-Aside*. There are many of these cloud architecture related patterns, but we choose to focus on a few ones related to Microservices: *Retry*, *Circuit Breaker*, *Cache-Aside*, and *Throttling*.

In this chapter, we will discuss the following patterns:

- The Retry pattern
- The Circuit Breaker pattern
- The Cache-Aside pattern
- The Throttling pattern

The Microservices pattern

Before Microservices, developers of an application have been using a single code base for implementing all functionalities. For example, the front-end UI that includes forms, buttons, and specialized JavaScript code for interactions and effects, the application logic that handles all the passage of data between the UI and the database, and another application logic that, based on triggers or scheduling, does some asynchronous actions behind the scenes, such as sending email notifications. Even the administration UI, as in a Django-based application, would be inside the same application.

For this way of building an application, we talk about *the Monolith model* or *using a Monolithic architecture*.

There is a cost to the Monolithic model of application development. Here are some disadvantages:

- Since it uses a single code base, the development team has to work on maintaining the whole code base simultaneously
- It is more difficult to organize testing and reproduce and fix bugs
- Tests and deployments become difficult to manage as the application and its user base grows and its constraints increase

With the *Microservice pattern* or *Microservices*, an application might consist of services such as the order management service, the customer management service, and so on. And, services can be developed and deployed independently of one another.

Real-world examples

When looking for examples, I found Eventuate™ (<http://eventuate.io>) which is a platform for developing asynchronous microservices. It solves the distributed data management problems inherent in a microservice architecture, enabling you to focus on your business logic.

eShopOnContainers is one of the Application Architecture reference applications from the .NET Foundation (<https://github.com/dotnet-architecture>). It is described as an easy-to-get-started sample reference microservice and container-based application.

We can also cite Serverless architecture/computing, which shares some of the characteristics of microservices. Its main particularity is that there is no provision of server resources for running and operating things. Basically, you split your application at an even more granular level (using functions), and you use serverless hosting vendors to deploy the services. What makes this model attractive is that, with these providers (the big three players being AWS Lambda, Google Cloud Functions and Azure Functions), in theory, you only pay for the use involved (the execution of the functions).

Use cases

We can think of several use cases where microservices offer a clever answer. We can use a microservices architecture-based design every time we are building an application that has at least one of the following characteristics:

- There is a requirement to support different clients, including desktop and mobile
- There is an API for third parties to consume
- We have to communicate with other applications using messaging
- We serve requests by accessing a database, communicating with other systems, and returning the right type of response (JSON, XML, HTML or even PDF)
- There are logical components corresponding to different functional areas of the application

Implementation

Let's briefly talk about software installation and application deployment in the Microservices world. Switching from deploying a single application to deployments of many small services means that the number of things that need to be handled increases exponentially. While you might have been fine with a single application server and few runtime dependencies, when moving to microservices, the number of dependencies will increase drastically. For example, one service could benefit from the relational database while the other would need Elasticsearch. You may need a service using MySQL and another one using the Redis server. So, using the microservices approach also means you need to use *Containers*.

Thanks to Docker, things are getting easier, since we can run those services as containers. The idea is that your application server, dependencies and runtime libraries, compiled code, configurations, and so on, are inside those containers. And then, all you have to do is run services packed as containers and make sure that they can communicate with each other.

You can implement the Microservices pattern, for a web app or an API, by directly using Django or Flask. However, for our examples, we are going to use a specialized microservices framework called **Nameko** (<https://nameko.readthedocs.io>).

According to their official website, Nameko is described as follows:

A microservices framework for Python that lets service developers concentrate on application logic and encourages testability.

Nameko has RPC over AMQP built in, allowing easy communication between the services. It also has a simple interface for HTTP requests.



Nameko and Flask: For writing Microservices that expose an HTTP endpoint, it is recommended that you use a more appropriate framework, such as Flask. To call Nameko methods over RPC using Flask, you can use `flask_nameko`, a wrapper built just for interoperating Flask with Nameko.

Like in the previous chapter, when exploring possibilities offered by the Observer in reactive programming, we are going to discuss two small examples of *service* implementation:

- A service that can return a list of people names, when called
- A service that calls the first one and appends the names obtained as result to a CSV file on disk

For now, you need to install the Nameko Framework using the `pip install nameko` command. But that's not all! In addition to the Python environment, we need a running RabbitMQ server. The easiest way to have a RabbitMQ in the development environment is running its official Docker container if you have Docker installed. But, it is also possible to install it on Linux using `apt-get`, or on Windows by following the installation instructions from the website.

To start RabbitMQ using Docker, run the following command:

```
docker run -d -p 5672:5672 -p 15672:15672 --name rabbitmq rabbitmq
```

A first example

For our first example, you also need to install the `Faker` module, if it was not yet done, using the `pip install faker` command.

Then, we will use a file called `service_first.py` to contain all our code. We will see in a moment how it is going to be used.

Since the service will be called using the *RPC* protocol, we import the `rpc` function that will be used as a decorator to add this capability to the service definition. We also import the `Faker` class.

Our current service module file starts with these imports:

```
from nameko.rpc import rpc
from faker import Faker
```

In `Nameko`, a service is a simple class with methods decorated with `@rpc` for the *jobs* the service offer. So, our `PeopleListService` class would look like this (well, just the skeleton for now):

```
class PeopleListService:

    name = 'peoplelist'

    @rpc
    def populate(self):
        pass
```

What we need to add is the logic that generates the names using a `Faker` instance. So, here is what we need for the `PeopleListService` class once the code is completed:

```
fake = Faker()

class PeopleListService:

    name = 'peoplelist'

    @rpc
    def populate(self, number=20):

        names = []
        for _ in range(0, number):
            n = fake.name()
            names.append(n)

        return names
```

But, we don't stop there. Let's prepare a second file called `test_service_first.py`, which helps us test services. We will use Nameko's `nameko.testing.services` facility to access and run the service.

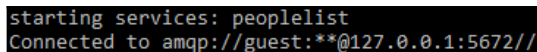
We can use the following code for this test:

```
from nameko.testing.services import worker_factory
from service_first import PeopleListService

def test_people():
    service_worker = worker_factory(PeopleListService)
    result = service_worker.populate()
    for name in result:
        print(name)

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

Before running this, we need to start the service. To do that, open a new terminal or shell window, and use the Nameko command line interface by running `nameko run service_first` command. Note that you are running the Nameko script provided by the installation previously instructed, so you must run that from the Python installation or virtual environment (using the `virtualenv` utility) where Nameko was installed. In my case, I got the following output in the window, showing that the service is started:

A terminal window with a black background and green text. The text shows the service starting and connecting to an AMQP broker.

```
starting services: peoplelist
Connected to amqp://guest:**@127.0.0.1:5672//
```

Next, in another window and, again, from the right Python installation (where **Nameko** is installed, as well as **Faker**), run the testing code via the `python test_service_first.py` command. You will get an output similar to the following:

```
Anita Munoz
Angela Graves
Jacqueline Harper
Ryan Myers
Amy Weeks
Debbie Larson
Andrea Harrington
Brent Lowe
Thomas Neal
Elizabeth Mata
Victoria Lowe
David Hubbard
William Parsons
Steven Mitchell
Harold Black
Nancy Martin
Christine Weber
Tommy Arroyo
Sabrina Simmons
Michael Collins
```

How easy was that? Granted, the logic of the service for this first example was simple. But, let's see how we could build on it for something more interesting in the next experimentation.

A second example

In our second example, let's reuse the idea of a *toy* service that helps produce a list of people. But, now we want the first name, last name, and address of people. We will then have a second service that depends on the first one to call it in order to get a list of people and then save that list on disk. Because our Service B depends on the Service A, we will use the `RpcProxy` class provided by Nameko, as you will see in a minute. The data will be saved using `csv`.

So, as you may guess, we start our `service_second.py` code file with the following imports:

```
from nameko.rpc import rpc, RpcProxy
from faker import Faker
import csv
```


Then, we define the new version of the `PeopleListService` using the following code:

```
fake = Faker()

class PeopleListService:

    name = 'peoplelist'

    @rpc
    def populate(self, number=20):

        persons = []
        for _ in range(0, number):
            p = {'firstname': fake.first_name(),
                'lastname': fake.last_name(),
                'address': fake.address()}
            persons.append(p)

        return persons
```

We add the definition of the `PeopleDataPersistenceService` service class, as follows:

```
class PeopleDataPersistenceService:

    name = 'people_data_persistence'
    peoplelist_rpc = RpcProxy('peoplelist')

    @rpc
    def save(self, filename):
        persons = self.peoplelist_rpc.populate(number=25)

        with open(filename, "a", newline="") as csv_file:
            fieldnames = ["firstname", "lastname", "address"]
            writer = csv.DictWriter(csv_file,
                                   fieldnames=fieldnames,
                                   delimiter=";")

            for p in persons:
                writer.writerow(p)

        return f"Saved data for {len(persons)} new people"
```

We introduce a script, `test_service_second.py`, to test this implementation example. We will import Nameko's `worker_factory` and `ClusterRpcProxy` classes, as well as the class of the service we are going to call to save new people data. The imports part is as follows:

```
from nameko.testing.services import worker_factory
```

```
from nameko.standalone.rpc import ClusterRpcProxy
from service_second import PeopleDataPersistenceService
```

We need to provide the minimum configuration information, using the `config` dictionary, which we are using next in the function:

```
config = {'AMQP_URI': "pyamqp://guest:guest@127.0.0.1"}

def test_peopledata_persist():
    with ClusterRpcProxy(config) as cluster_rpc:
        out =
cluster_rpc.people_data_persistence.save.call_async('people.csv')
    print(out.result())
```

Lastly, we add the following to be able to call the function when running the file as a script:

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

Let's run the example. First, remember you need to start the service. This can be done using the `nameko run service_second` command. (Note you don't have to add the `.py` suffix, if not the command's help will complain and advise you not to.) You will see an output as follows:

```
starting services: people_data_persistence, peoplelist
Connected to amqp://guest:**@127.0.0.1:5672//
Connected to amqp://guest:**@127.0.0.1:5672//
```

Next, in another window, you can run the testing code via the `python test_service_second.py` command. You will get an output similar to the following:

```
Saved data for 25 new people
```

In addition to that, if you check on the file system, you can see the `people.csv` file that was created. You can play with calling the script several times, and see that it keeps adding the new rows of people with their first name, last name, and address to that file.

A possible third example that we could try is sending a mail to an actor of our system; a mail containing a kind of report of the people listing and saving activity handled using the two services we just implemented. That is left as an exercise for the reader.

The Retry pattern

Retrying is an approach that is more and more needed in the context of microservices and cloud-based infrastructure where components collaborate with each other, but are not developed, deployed, and operated by the same teams and parties.

In its daily operation, parts of a cloud-native application may experience what is called *transient faults or failures*, meaning some mini-issues that can look like bugs, but are not due to your application itself but to some constraints outside of your control such as the networking or the external server/service performance. As a result, your application may dysfunction (at least that could be the perception of your users) or even hang at some places. The answer to the risk of such failures is to put in place some retry logic, so that we pass through the issue, by calling the service again, maybe immediately or after some wait time (such as a few seconds).

Real-world examples

There are many implementations or tools to use for your custom case. Here are some examples:

- In Python, the **Retrying** library (<https://github.com/rholder/retrying>) is available to simplify the task of adding retry behavior to our functions
- The **Pester** library (<https://github.com/sethgrid/pester>) for Go developers
- In the Java world, **Spring Retry** helps to use the Retry pattern in Spring applications

Use cases

This pattern is recommended to alleviate the impact of identified *transient failures* while communicating with an external component or service, due to network failure or server overload.

Note that the *retrying* approach is not recommended for handling failures such as internal exceptions caused by errors in the application logic itself.

Also, we have to think about and analyze the way the external service responds. If the application experiences frequent busy faults, it's often a sign that the service being accessed has a scaling issue that should be addressed.

Implementation

Let's see examples that are easy to reproduce to get an idea of how the *Retry* pattern can help improve an application where we communicate with an external service and experience transient failures.

A first example

In a first example, let's imagine we want to write and update a file using two different programs (which could be services). Instead of creating two scripts, we will actually create a single script (in the `retry_write_file.py` file) that can be called by an argument to indicate what we want to do: *create the file* (first step) or *update it*.

We need a few imports, as follows:

```
import time
import sys
import os
```

We define a function to create the file after a certain delay using `time.sleep(after_delay)`, as follows:

```
def create_file(filename, after_delay=5):
    time.sleep(after_delay)

    with open(filename, 'w') as f:
        f.write('A file creation test')
```

Next, we define a function that helps append some text to the file, once it has been created, as follows:

```
def append_data_to_file(filename):
    if os.path.exists(filename):
        with open(filename, 'a') as f:
            f.write(' ...Updating the file')
    else:
        raise OSError
```

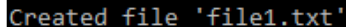
In the main part of the code, we use the right function depending on what is passed in the command line. Also, we call the `append_data_to_file()` function within a `try / except` and inside a `while` loop so that program keeps trying to update the file until this file is created. The code of that part, preceded by the declaration of a global variable for the name of the file (`'file1.txt'`), is as follows:

```
FILENAME = 'file1.txt'

if __name__ == "__main__":
    args = sys.argv

    if args[1] == 'create':
        create_file(FILENAME)
        print(f"Created file '{FILENAME}'")
    elif args[1] == 'update':
        while True:
            try:
                append_data_to_file(FILENAME)
                print("Success! We are done!")
                break
            except OSError as e:
                print("Error... Try again")
```

To test this, open two different terminals (or `cmd` command windows). In one of them, run the `python retry_write_file.py create` command. Immediately after that (*it is important to do it immediately*), in the second window, run the `python retry_write_file.py update` command. You should see output similar to the following for each command:

A terminal window with a black background and green text. The text displayed is "Created file 'file1.txt'".

```
Created file 'file1.txt'
```

[illegible]

It worked! What we have in this script is a simple but working implementation of the Retry pattern.

A second example, using a third-party module

There is actually a library, called `retrying`, that can be used for adding retry behavior to parts of an application. Let's use it for a similar implementation as the one in the first example.

First, make sure you install `retrying` using the `pip install retrying` command.

We need to start our code with the following imports:

```
import time
import sys
import os
from retrying import retry
```

We reuse the same function for creating a file. You could actually externalize it to a common functions module from where you would import it, if you want. For clarity, let's repeat it here:

```
def create_file(filename, after_delay=5):
    time.sleep(after_delay)

    with open(filename, 'w') as f:
        f.write('A file creation test')
```

We also use a slightly different function for the *append data* part, and we decorate it with the *retry decorator* imported from *retrying*, as follows:

```
@retry
def append_data_to_file(filename):

    if os.path.exists(filename):
        print("got the file... let's proceed!")
        with open(filename, 'a') as f:
            f.write(' ...Updating the file')
        return "OK"
    else:
        print("Error: Missing file, so we can't proceed. Retrying...")
        raise OSError
```

The last part of the code is almost identical to the one we had in the first implementation:

```
FILENAME = 'file2.txt'

if __name__ == "__main__":
    args = sys.argv

    if args[1] == 'create':
        create_file(FILENAME)
        print(f"Created file '{FILENAME}'")
    elif args[1] == 'update':
        while True:
            out = append_data_to_file(FILENAME)
            if out == "OK":
                print("Success! We are done!")
                break
```


We replace the import for `retrying` module by the one for `tenacity` module, as follows:

```
import tenacity
```

We replace the decorator as follows:

```
@tenacity.retry
def append_data_to_file(filename):
    # code that could raise an exception
```

With this new version of the code in the `retry_write_file_tenacity_module.py` file, testing it should give the same result as in the second example of this circuit breaker implementation discussion.

But let's not stop there. You could tweak the retrying strategy using parameters such as fixed wait (we wait for a fixed time between retries) or exponential backoff (we increase the time between retries in an incremental way as we go).

A first improvement could be adding a wait, for example, of 2s before retrying. For that, we just have to change our function's decoration to the following:

```
@tenacity.retry(wait=tenacity.wait_fixed(2))
def append_data_to_file(filename):
    # code that could raise an exception
```

When testing the whole thing again after this change, we can confirm that the behavior is as expected: the retries are done, only that each retry happens after a tempo of 2s .

For an exponential backoff, you could do the setup using the following code:

```
@tenacity.retry(wait=tenacity.wait_exponential())
def append_data_to_file(filename):
    # code that could raise an exception
```

Again, a quick test confirms that the implementation works. Exponential backoff is nice and a clever answer to the cases of APIs and remote services we call and where this kind of failures might be occurring. The increase of the wait time gives a chance to get a result from the remote end, and if we do not get it after several attempts, it probably means the service has less chances of responding in time, so we don't waste our resources trying to call it too much.

As you can see, we have many possibilities for building services by adding fault-tolerance behavior to them.

The Circuit Breaker pattern

One approach to fault tolerance involves timeouts and retries, as we have just seen. But, when a failure due to the communication with an external component is likely to be long-lasting, using a retry mechanism can affect the responsiveness of the application. We might be wasting time and resources trying to repeat a request that's likely to fail. This is where another pattern can be useful, the *Circuit Breaker*.

With circuit breaker, you wrap a fragile function call (or an integration point with an external service) in a special (circuit breaker) object, which monitors for failures. Once the failures reach a certain threshold, the circuit breaker *trips*, and all further calls to the circuit breaker return with an error, without the protected call being made at all.

Real-world examples

In life, we can think of a water or electricity distribution circuit.

In software, here are some examples:

- The **pybreaker** Python library
- **Hystrix**, from Netflix, a sophisticated tool for dealing with latency and fault tolerance for distributed systems
- The Java library **Jrugged** (<https://github.com/Comcast/jrugged>)

Use cases

As already said, the Circuit Breaker pattern is recommended when you need a component from your system to be fault-tolerant to *long-lasting failures* when communicating with an external component, service, or resource.

We will understand how it addresses such use cases in the *Implementation* section that follows.

Implementation

We are going to use the `pybreaker` library. You can install it in your Python, via the `pip install pybreaker` command.

Let's say you want to use a circuit breaker on a flaky function (for example, fragile due to the networking environment it depends on).

Our implementation is inspired by a nice script I found in this repository: <https://github.com/veltra/pybreaker-playground>, which we will adapt.

Here is our version of the function to simulate fragile calls:

```
def fragile_function():
    if not random.choice([True, False]):
        print(' / OK', end='')
    else:
        print(' / FAIL', end='')
        raise Exception('This is a sample Exception')
```

Let's define our circuit breaker to automatically open the circuit after five consecutive failures in that function. We need to create an instance of the `CircuitBreaker` class, as follows:

```
breaker = pybreaker.CircuitBreaker(fail_max=2, reset_timeout=5)
```

Then, we use the decorator syntax to protect things, so the new function is as follows:

```
@breaker
def fragile_function():
    if not random.choice([True, False]):
        print(' / OK', end='')
    else:
        print(' / FAIL', end='')
        raise Exception('This is a sample Exception')
```

Let's say you wanted at this point to execute a call to the function (*but you don't need to*), by adding the following:

```
while True:
    fragile_function()
```

When executing the script, the program will stop on an exception, showing the error message *Exception: This is a sample Exception* (and you will keep getting that error if you try again.) What happened is that the exception was raised every time so nothing could work. Indeed, the protection is not yet in place. There is something missing.

We need to catch the exception in the main part of the code for the whole program to work as expected. Here is the rest of the (real) code we need to add, in order to trigger the circuit breaker:

```
if __name__ == "__main__":  
    while True:  
        print(datetime.now().strftime('%Y-%m-%d %H:%M:%S'), end='')  
  
        try:  
            fragile_function()  
        except Exception as e:  
            print(' / {} {}'.format(type(e), e), end='')  
        finally:  
            print('')  
            sleep(1)
```

Let's recap the whole code of this example (in the `circuit_breaker.py` file) in order to understand all that is involved:

- We import everything we need, as follows:

```
import pybreaker  
from datetime import datetime  
import random  
from time import sleep
```

- We set up the circuit breaker, as follows:

```
breaker = pybreaker.CircuitBreaker(fail_max=2, reset_timeout=5)
```

- Next, we have the fragile function, decorated by `@breaker`, as already shown.
- And, finally, we have the main part of the program, where the protected calls to the fragile function happen, as previously shown.

Calling the script by running the `python circuit_breaker.py` command produces the following output:

```
2018-08-20 12:06:01 / OK
2018-08-20 12:06:02 / OK
2018-08-20 12:06:03 / OK
2018-08-20 12:06:04 / FAIL / <class 'Exception'> This is a sample Exception
2018-08-20 12:06:05 / OK
2018-08-20 12:06:06 / FAIL / <class 'Exception'> This is a sample Exception
2018-08-20 12:06:07 / FAIL / <class 'pybreaker.CircuitBreakerError'> Failures threshold reached, circuit breaker opened
2018-08-20 12:06:08 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:09 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:10 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:11 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:12 / FAIL / <class 'pybreaker.CircuitBreakerError'> Trial call failed, circuit breaker opened
2018-08-20 12:06:13 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:14 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:15 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:16 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:17 / OK
2018-08-20 12:06:18 / FAIL / <class 'Exception'> This is a sample Exception
2018-08-20 12:06:19 / OK
2018-08-20 12:06:20 / FAIL / <class 'Exception'> This is a sample Exception
2018-08-20 12:06:21 / OK
2018-08-20 12:06:22 / OK
2018-08-20 12:06:23 / FAIL / <class 'Exception'> This is a sample Exception
2018-08-20 12:06:24 / OK
2018-08-20 12:06:25 / FAIL / <class 'Exception'> This is a sample Exception
2018-08-20 12:06:26 / FAIL / <class 'pybreaker.CircuitBreakerError'> Failures threshold reached, circuit breaker opened
2018-08-20 12:06:27 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:28 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:29 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:30 / <class 'pybreaker.CircuitBreakerError'> Timeout not elapsed yet, circuit breaker still open
2018-08-20 12:06:31 / OK
```

By closely looking at the output, we can see that the circuit breaker does its job as expected:

1. When it is open, all `fragile_function()` calls fail immediately (since they raise the `CircuitBreakerError` exception) without any attempt to execute the intended operation.
2. And, after the timeout of five seconds, the circuit breaker will allow the next call to go through. If that call succeeds, the circuit is closed; if it fails, the circuit is opened again until another timeout elapses.

The Cache-Aside pattern

In situations where data is more frequently read than updated, applications use a cache to optimize repeated access to information stored in a database or data store. In some systems, that type of caching mechanism is built-in and works automatically. When this is not the case, we have to implement it in the application ourselves, using a caching strategy that is suitable for the particular use case.

One such strategy is called *Cache-Aside*, where, to quote the description in Microsoft's documentation about Cloud-Native patterns (see [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn589799\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/dn589799(v=pandp.10))), we perform the following:

Load data on demand into a cache from a data store, as an attempt to improve performance, while maintaining consistency between data held in the cache and the data in the underlying data store.

Real-world examples

Memcached is commonly used as a cache server. It is a popular in-memory *key-value* store for small chunks of data from results of database calls, API calls, or HTML page content. *Redis* is another server solution for the same purpose.

Amazon's *ElastiCache*, according to the documentation site (<https://docs.aws.amazon.com/AmazonElastiCache>), is a web service that makes it easy to set up, manage, and scale a distributed in-memory data store or cache environment in the cloud.

Use cases

The Cache-Aside pattern is useful for data that doesn't change often, and for data storage that doesn't depend on the consistency of a set of entries in the storage (multiple keys). For example, it might work for certain kinds of document stores, where keys are never updated, and occasionally documents are deleted but there is no strong requirement to continue to serve them for some time (until the cache is refreshed).

Also, according to the documentation, we can find (from Microsoft), this pattern might not be suitable in the cases where the cached data set is static, or for caching session state information in a web application hosted in a web farm.

Implementation

We can summarize the steps needed when implementing the Cache-Aside pattern, involving a database and a cache, as follows:

- **Case 1:** When we want to fetch a data item: return the item from cache if found in it. If not found in cache, read the data from the database. Put the read item in the cache and return it.

- **Case 2:** When we want to update a data item: write the item in the database and remove the corresponding entry from the cache.

Let's try a simple implementation with a database of quotes from which the user can ask to retrieve some quotes via an application. Our focus here will be implementing the *Case 1* part.

Here are our choices for the additional software dependencies we need to install on the machine for this implementation:

- We will use an SQLite 3 database, since it is easy to install of most operating systems, and we can query an SQLite database using Python's standard module, `sqlite3`.
- Instead of using a system with real caching capabilities such as Memcached or Redis, we will use a CSV file to emulate how data is loaded to and read from the cache on demand. That way, we will have a working implementation without the need to install an additional software component.

We will use a script (`populate_db.py`) to handle the creation of the database, the quotes table, and add example data to it. We also use Faker again to generate fake quotes that we will use when populating the database (for quick experimentation).

First are the imports we need, followed by the creation of the `Faker` instance:

```
import sys
import sqlite3
import csv
from random import randint
from faker import Faker

fake = Faker()
```

Then, we write a function to take care of the database setup part, as follows:

```
def setup_db():

    try:
        db = sqlite3.connect('data/quotes.sqlite3')

        # Get a cursor object
        cursor = db.cursor()
        cursor.execute('''
            CREATE TABLE quotes(id INTEGER PRIMARY KEY, text TEXT)
        ''')

        db.commit()
```

```

except Exception as e:
    print(e)
finally:
    db.close()

```

Then, we define a central function that takes care of adding a set of new quotes based on a list of sentences or text snippets. Among different things, we associate a quote *identifier* to the quote, for the `id` column in the database table. To make things easier, we just pick a number randomly using `quote_id = randint(1, 100)`. The `add_quotes()` function is defined as follows:

```

def add_quotes(quotes_list):
    quotes = []
    try:
        db = sqlite3.connect('data/quotes.sqlite3')

        cursor = db.cursor()

        quotes = []
        for quote_text in quotes_list:
            quote_id = randint(1, 100)
            quote = (quote_id, quote_text)
            try:
                cursor.execute('''INSERT INTO quotes(id, text) VALUES(?,
?)''', quote)
                quotes.append(quote)
            except Exception as e:
                print(f"Error with quote id {quote_id}: {e}")
        db.commit()
    except Exception as e:
        print(e)
    finally:
        db.close()

    return quotes

```

Next, we add the `main` function, which in fast will have several parts: we want to use command line argument parsing and note the following:

- If we pass the `init` argument, we call the `setup_db()` function
- If we pass the `update_db_and_cache` argument, we inject the quotes in the database, and we add them to the cache
- If we pass the `update_db_only` argument, we only inject the quotes in the database

The code of the `main()` function is as follows:

```
def main():
    args = sys.argv

    if args[1] == 'init':
        setup_db()

    elif args[1] == 'update_db_and_cache':
        quotes_list = [fake.sentence() for _ in range(1, 11)]
        quotes = add_quotes(quotes_list)
        print("New (fake) quotes added to the database:")
        for q in quotes:
            print(f"Added to DB: {q}")

        # Populate the cache with this content
        with open('data/quotes_cache.csv', "a", newline="") as csv_file:
            writer = csv.DictWriter(csv_file,
                                    fieldnames=['id', 'text'],
                                    delimiter=";")

            for q in quotes:
                print(f"Adding '{q[1]}' to cache")
                writer.writerow({'id': str(q[0]), 'text': q[1]})

    elif args[1] == 'update_db_only':
        quotes_list = [fake.sentence() for _ in range(1, 11)]
        quotes = add_quotes(quotes_list)
        print("New (fake) quotes added to the database ONLY:")
        for q in quotes:
            print(f"Added to DB: {q}")
```

We call the `main` function as usual, with the following:

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

That part is done, we will create another module and script for the cache-aside related operations themselves (the `cache_aside.py` file).

We have a few imports needed here too:

```
import sys
import sqlite3
import csv
```

We add a global variable, `cache_key_prefix`, since we use that value at several places, as follows:

```
cache_key_prefix = "quote"
```

Next, we define the cache container object, `QuoteCache`, by providing its `get()` and `set()` methods, as follows:

```
class QuoteCache:

    def __init__(self, filename=""):
        self.filename = filename

    def get(self, key):
        with open(self.filename) as csv_file:
            items = csv.reader(csv_file, delimiter=';')
            for item in items:
                if item[0] == key.split('.')[1]:
                    return item[1]

    def set(self, key, quote):
        existing = []
        with open(self.filename) as csv_file:
            items = csv.reader(csv_file, delimiter=';')
            existing = [cache_key_prefix + "." + item[0] for item in items]

        if key in existing:
            print("This is weird. The key already exists.")
        else:
            # save the new data
            with open(self.filename, "a", newline="") as csv_file:
                writer = csv.DictWriter(csv_file,
                                       fieldnames=['id', 'text'],
                                       delimiter=";")
                writer.writerow({'id': key.split('.')[1], 'text': quote})
```

We can create the cache object:

```
cache = QuoteCache('data/quotes_cache.csv')
```

Next, we define the `get_quote()` function to fetch a quote by its identifier: if we do not find the quote in the cache, we query the database to get it and we put the result in the cache before returning it. The function is defined as follows:

```
def get_quote(quote_id):
    quote = cache.get(f"quote.{quote_id}")
    out = ""
```

```

if quote is None:
    try:
        db = sqlite3.connect('data/quotes.sqlite3')
        cursor = db.cursor()
        cursor.execute(f"SELECT text FROM quotes WHERE id =
{quote_id}")
        for row in cursor:
            quote = row[0]
            print(f"Got '{quote}' FROM DB")
        except Exception as e:
            print(e)
        finally:
            # Close the db connection
            db.close()

        # and add it to the cache
        key = f"{cache_key_prefix}.{quote_id}"
        cache.set(key, quote)
    if quote:
        out = f"{quote} (FROM CACHE, with key 'quote.{quote_id}')"

return out

```

Finally, in the main part of the script, we ask for user input of a quote identifier, and we call `get_quote()` to fetch the quote. The code is as follows:

```

if __name__ == "__main__":
    args = sys.argv

    if args[1] == 'fetch':
        while True:
            quote_id = input('Enter the ID of the quote: ')
            q = get_quote(quote_id)
            if q:
                print(q)

```

Let's test our scripts, with the following steps.

By calling `python populate_db.py init`, we can see that the `quotes.sqlite3` file is created in the `data` folder, so we can conclude the database has been created and the quotes table created in it.

Then, we call `python populate_db.py update_db_and_cache`, and get the following output:

```
Error with quote id 99: UNIQUE constraint failed: quotes.id
New (fake) quotes added to the database:
Added to DB: (76, 'Family card magazine should manage so.')
Added to DB: (48, 'Eight realize third commercial feeling soldier fund.')
Added to DB: (83, 'Establish assume decide myself second increase bar.')
Added to DB: (62, 'Society practice Mrs music admit likely.')
Added to DB: (87, 'Management girl technology summer.')
Added to DB: (99, 'Assume realize fly six.')
Added to DB: (82, 'Account me play figure chance.')
Added to DB: (42, 'Congress cause suffer join either foot.')
Added to DB: (5, 'As for continue collection.')
Adding 'Family card magazine should manage so.' to cache
Adding 'Eight realize third commercial feeling soldier fund.' to cache
Adding 'Establish assume decide myself second increase bar.' to cache
Adding 'Society practice Mrs music admit likely.' to cache
Adding 'Management girl technology summer.' to cache
Adding 'Assume realize fly six.' to cache
Adding 'Account me play figure chance.' to cache
Adding 'Congress cause suffer join either foot.' to cache
Adding 'As for continue collection.' to cache
```

We can also call `python populate_db.py update_db_only`. In that case, we get the following output:

```
Error with quote id 83: UNIQUE constraint failed: quotes.id
Error with quote id 6: UNIQUE constraint failed: quotes.id
New (fake) quotes added to the database ONLY:
Added to DB: (67, 'Lawyer technology about who matter create.')
Added to DB: (91, 'Reveal conference these get.')
Added to DB: (21, 'Land talk similar card service.')
Added to DB: (6, 'Soldier pull see rate industry among lay.')
Added to DB: (11, 'Check new mention break.')
Added to DB: (31, 'Born nearly cultural tax drop probably later.')
Added to DB: (100, 'Try size on change upon.')
Added to DB: (33, 'Street mention religious pretty chair mind.')
```

Next, we call `python cache_aside.py fetch` and we are asked for an input to try and fetch the matching quote. Here are the different outputs I got depending on the values I provided:

```
Enter the ID of the quote: 42
Congress cause suffer join either foot. (FROM CACHE, with key 'quote.42')
Enter the ID of the quote: 87
Management girl technology summer. (FROM CACHE, with key 'quote.87')
Enter the ID of the quote: 21
Got 'Land talk similar card service.' FROM DB
Land talk similar card service. (FROM CACHE, with key 'quote.21')
Enter the ID of the quote: 100
Got 'Try size on change upon.' FROM DB
Try size on change upon. (FROM CACHE, with key 'quote.100')
Enter the ID of the quote: 31
Got 'Born nearly cultural tax drop probably later.' FROM DB
Born nearly cultural tax drop probably later. (FROM CACHE, with key 'quote.31')
Enter the ID of the quote:
```

So, each time I entered an identifier number that matches a quote stored only in the database (as shown by the previous output), the specific output shows that the data is got from the database first, before being returned from the cache (where it was immediately added). And, we can confirm that by looking at the content of the `quotes_cache.csv` file.

We can see that things work as expected. The *update* part of the cache-aside implementation (*write the item in the database and remove the corresponding entry from cache*) is left to you to try. You could add an `update_quote()` function used to update a quote when you pass the `quote_id` to it, and call it when we use the `python cache_aside.py update` command.

Throttling

Throttling is another pattern we may need to use in today's applications, services, and APIs. In this context, throttling is based on limiting the number of requests a user can send to a given web service in a given amount of time, in order to protect the resources of the service from being overused by some users.

For example, we may want to limit the number of user requests for an API to 1000/day. Once that limit is reached, the next request is handled by sending an error message with the **429** HTTP status code to the user with a message such as too many requests.

There are many things to understand about Throttling, including which limiting strategy and algorithm one may use and measuring how the service is used.

You can find technical details about the throttling pattern in the catalog of Cloud design patterns by Microsoft (<https://docs.microsoft.com/en-/azure/architecture/patterns/throttling>).

Real-world examples

Since this is an important feature for Restful APIs, the frameworks have built-in support or third-party modules to implement throttling. The following are some of the examples:

- The built-in support in Django-Rest-Framework (see <https://www.django-rest-framework.org/api-guide/throttling/>)
- Django-throttle-requests (<https://github.com/sobotklp/django-throttle-requests>) is a framework for implementing application-specific rate-limiting middleware for Django projects
- The flask-limiter (<https://flask-limiter.readthedocs.io/en/stable/>) provides rate limiting features to Flask routes

And we can find throttling implemented in the big Cloud players' services themselves, such as AWS API Gateway

Use cases

This pattern is recommended when you need to ensure your system continuously delivers the service as expected, or when you need to optimize the cost of usage of the service, or when you need to handle bursts in activity.

In practice, you may implement the following rules:

- Limit the number of total requests to an API as N/day (for example, N=1000)
- Limit the number of requests to an API as N/day from a given IP address, or from a given country or region
- Limit the number of reads or writes for authenticated users

Implementation

Before diving into an implementation example, you need to know that there are actually different types of throttling, among which Rate-Limit, IP-level Limit (based on a list of whitelisted IP addresses for example), and Concurrent Connections Limit, to only cite those three. The first two are relatively easy to experiment with. We will focus on the first one here.

Let's see an example of rate-limit type throttling using a Flask application. To get a minimal Flask application running on your development machine, you have to add Flask to the Python environment, using the `pip install flask` command. Then add the Flask-Limiter extension using the `pip install flask-limiter` command.

As usual, we setup the Flask application with the following two lines:

```
from flask import Flask
app = Flask(__name__)
```

We then define the Limiter instance; we create it by passing the application object, a key function, which is `get_remote_address` (imported from `flask_limiter.util`), and the default limits values, as follows:

```
limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["100 per day", "10 per hour"]
)
```

Based on that, we can define a route `/limited`, which will be rate-limited using the default limits, as follows:

```
@app.route("/limited")
def limited_api():
    return "Welcome to our API!"
```

Now, for the example to work, let's not forget the imports we need to place at the beginning of the file:

```
from flask import Flask
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
```

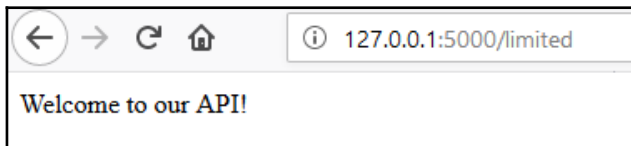
Also, we add the following snippet at the end of the file to ensure the execution when calling the file using the Python executable:

```
if __name__ == "__main__":  
    app.run(debug=True)
```

To run the example Flask app, let's use the `python throttling_in_flaskapp.py` command. We will get an output similar to the following:

```
* Serving Flask app "throttling_in_flaskapp" (lazy loading)  
* Environment: production  
  WARNING: Do not use the development server in a production environment.  
  Use a production WSGI server instead.  
* Debug mode: on  
* Restarting with stat  
* Debugger is active!  
* Debugger PIN: 244-225-855  
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```

Then if you point your browser to `http://127.0.0.1:5000/limited`, you will see the welcome content displayed in the page, as follows:



It gets interesting if you keep hitting the Refresh button. The 10th time, the page content would change and show you a **Too Many Requests** error message, as shown in the following screenshot:



Let's not stop here. To complete our example, we could add another route `/more_limited` with a specific limit of two requests per minute, as follows:

```
@app.route("/more_limited")
@limiter.limit("2/minute")
def more_limited_api():
    return "Welcome to our expensive, thus very limited, API!"
```

By the way, you may want to see the full code of our Flask app example (file `throttling_in_flaskapp.py`) as follows:

```
from flask import Flask
from flask_limiter import Limiter
from flask_limiter.util import get_remote_address
from flask import Flask
app = Flask(__name__)
limiter = Limiter(
    app,
    key_func=get_remote_address,
    default_limits=["100 per day", "10 per hour"]
)
@app.route("/limited")
def limited_api():
    return "Welcome to our API!"
@app.route("/more_limited")
@limiter.limit("2/minute")
def more_limited_api():
    return "Welcome to our expensive, thus very limited, API!"
if __name__ == "__main__":
    app.run(debug=True)
```

Now, let's come back to testing the second route we added. Since the Flask application is run in debug mode (thanks to the `app.run(debug=True)` call), there is nothing else to do. The file is automatically re-loaded as we update the code. So the new version that includes the second route is already live.

To test it, point your browser to `http://127.0.0.1:5000/more_limited`. You will see a new welcome content displayed on the page, as follows:



And if we hit the Refresh button and do it more than twice in a window of 1mn, we get another **Too many requests** message, as shown in the following screenshot :



There are many possibilities for rate-limit type throttling in a Flask application using the Flask-Limiter extension as you can see on the documentation page (<https://flask-limiter.readthedocs.io>) of the module. Flask-Limiter is actually based on a specialized library called limits (<https://limits.readthedocs.io/en/latest/index.html>). The reader can find more information on the library's documentation page on how to use different strategies and storage backends such as Redis or Memcached for a specific implementation.

Summary

In this chapter, we have introduced the Microservices architecture pattern, the idea being to split an application in a set of loosely coupled, collaborating services, and some practices and frameworks that help use it in a project context.

Among the advantages of using Microservices, the development team can more easily collaborate on the implementation parts, the software components, and the deployments. Services can be developed and deployed independently of one another.

There are more and more examples of this category of patterns used in software or application development and deployment today, coming from technology vendors, cloud service providers, as well as in-house DevOps specialists.

We have played with very small but instructive examples using the Nameko microservices framework, including one of those examples of the framework that has gained traction these last three or four years, which emphasizes testing the services we build and provides the tools to help in doing so.

Then, we covered Retrying mechanisms, which are one strategy used for fault-tolerance, in situations where a call is possible to fail, but if we try more times the call might succeed. These techniques are more and more needed now that we are in the Cloud-Native and Microservices architectures era. There are several open source libraries in languages such as Java, Python, and Go that implement retrying mechanisms, and we can use them by following their APIs. We saw an example where we tried to do this ourselves, and an example where we used the retrying library.

Circuit breakers, another approach to fault tolerance, allow a system to keep running even if one subsystem fails. This is done by wrapping fragile operations with a component that can circumvent calls that would otherwise pose a problem, when the system is not healthy. In Python, we can use the PyBreaker library to adding circuit breakers in our application. We used an example to show how a circuit breaker based on Pybreaker can help protect a fragile function (written for demonstration purposes) as part of a typical application.

We also discussed the Cache-Aside pattern. In applications that rely heavily on accessing data from a data store, using the Cache-Aside pattern can improve the performance while reading data from the data store via caching. We used an example to show how one could use the Cache-Aside pattern for the getting data from the data store via the cache part of the use cases, while leaving the updating data to the data store part ready, as an exercise.

We ended our chapter with the Throttling pattern. We approached the rate-limit type throttling technique used to control how users consume a web service and make sure the service does not get overwhelmed by one particular tenant. This was demonstrated using Flask and its extension Flask-Limiter.

This is the end of this book. I hope you enjoyed it. Before I leave you, I want to remind you about something by quoting Alex Martelli, an important Python contributor, who says, "Design patterns are discovered, not invented" [j.mp/templatemart, page 25].

Index

A

Abstract Base Classes (ABC) 66

abstract factory pattern

about 18

implementing 19, 20, 21, 23

real-world examples 18

use cases 19

abstract syntax tree (AST) 142

adapter pattern

about 51

implementing 52, 53, 54

real-world examples 51

use cases 52

B

Backus-Naur Form (BNF) notation 144

behavioral patterns

interpreter pattern 142

iterator pattern 161

Memento pattern 157

strategy pattern 150

template pattern 166

bridge pattern

implementing 65, 66, 67, 68

real-world examples 64

use cases 65

builder 26

builder design pattern

builder 26

director 26

implementing 31, 32, 33, 34, 35, 36

real-world examples 26

use cases 27, 28, 30

C

cache-aside pattern

about 208

implementing 209, 210, 211, 213, 215, 216

real-world examples 209

reference 209

use cases 209

Chain of Responsibility pattern

about 101

implementing 105, 106, 107, 108, 109

real-world examples 102, 103

use cases 104

Chip 95

circuit breaker pattern

about 205

implementing 206, 207

real-world examples 205

use cases 205

Command pattern

implementing 113, 114, 115, 116, 117, 118,
119, 120

real-world examples 112

use cases 112

content management system (CMS) 26

creational design patterns 7

D

decorator pattern

implementing 58, 59, 60, 61, 62

real-world examples 57

use cases 57

design patterns 7

Direct Selling 39

director 26

django-fsm package 132

django-query-builder library

- reference 27
- django-widgy
 - reference 26
- domain-specific language (DSL) 132, 142

E

- ElastiCache
 - reference 209
- event-driven systems 123
- Eventuate™
 - reference 190

F

- factory method
 - about 8
 - implementing 10, 11, 12, 13, 14, 15, 17
 - real-world examples 9
 - use cases 9, 10
- factory_boy package
 - reference 18
- Faker
 - reference 182
- façade pattern
 - about 70
 - implementing 72, 73, 74, 75, 76
 - real-world examples 71
 - use cases 71

- first-person shooter (FPS) game 78

- Flask-Limiter
 - reference 221

- fluent builder 36

- flyweight pattern
 - about 78
 - implementing 80, 81, 82, 83, 84
 - real-world examples 79
 - use cases 79

- FrogWorld game 20

G

- Gang of Four book 7
- Git Cola 112
- GNU Hurd
 - reference 72
- graphical user interface (GUI) 58
- graphs 131

H

- hard disk drive (HDD) 78

- Hystrix 205

I

- interface adaptation 56
- internal DSL 142
- Internet of Things (IoT) 143
- interpreter pattern
 - about 142
 - implementing 143, 144, 145, 147, 148, 150
 - real-world examples 143
 - use cases 143
- iterator pattern
 - about 161, 162
 - implementing 163, 165
 - real-world examples 162
 - use cases 162

J

- Jrugged
 - reference 205

K

- Kivy 123

L

- limits
 - reference 221
- loose coupling 71

M

- Memento pattern
 - about 157
 - components 157
 - implementing 158, 159, 160
 - real-world examples 157
 - use cases 158
- memoization 80
- microservices pattern
 - about 190
 - examples 193, 194, 195, 196, 197
 - implementing 191, 192

- real-world examples 190

- use cases 191

MINIX 3

- reference 72

Model-Template-View (MTV) 86

model-view-adapter (MVA) 87

Model-View-Controller pattern

- about 85

- implementing 88, 89, 90, 91, 92

- real-world examples 86

- use cases 87, 88

model-view-presenter (MVP) 87

N

Nameko

- reference 192

Network Marketing 39

news feeds 123

O

object classes

- reference 7

Object-oriented programming (OOP) 131

object-oriented programming (OOP) 80, 166

Object-relational mapping (ORM) 96

Observable 174

Observable pattern

- examples 175, 176, 178, 179, 180, 182, 183, 184, 186, 187

- implementing 175

- real-world examples 174

- use cases 174

Observer pattern

- implementing 124, 125, 126, 127, 128, 129

- real-world examples 122

- use cases 123

observers 122

operators, for Observables

- reference 179

P

Pester library

- reference 198

pickle module

- reference 158

Properties module 123

protective proxy 92

prototype pattern

- about 39

- implementing 40, 41, 42, 43

- real-world examples 39

- use cases 40

proxy pattern

- about 92, 93, 94

- implementing 96, 97, 98, 99

- real-world examples 95

- use cases 95

Pyparsing 144

PyQt 112

R

reactive programming 173

ReactiveX

- reference 173

Read-Eval-Print Loop (REPL) 10

remote proxy 92

retry pattern

- about 198

- examples 199, 200, 201, 202, 203, 204

- implementing 199

- real-world examples 198

- use cases 198

retrying library

- reference 198

S

Separation of Concerns (SoC) principle 85

singleton pattern

- about 44

- implementing 46, 47, 48, 49

- real-world examples 45

- use cases 45

smart (reference) proxy 92

Solid-state drives (SSD) 78

Spring Retry 198

state machine 131

State Machine Compiler (SMC) 132

State pattern

- implementing 133, 134, 135, 136, 137, 138, 139, 140

- real-world examples 132
- use cases 133
- state transitions 131
- states 131
- strategy pattern
 - about 150
 - implementing 152, 153, 155, 156
 - real-world examples 151
 - use cases 152

T

- template 86
- template pattern
 - about 166
 - implementing 168, 169, 170
 - real-world examples 166, 167
 - use cases 167
- Tenacity
 - reference 203
- test-specific attributes 18
- throttling pattern

- about 216
- implementing 218, 219, 221
- real-world examples 217
- reference 217
- use cases 217
- transitions 131

V

- virtual proxy 92

W

- Web2py Framework
 - reference 86

Z

- Zope application server
 - reference 52
- Zope Component Architecture (ZCA) 52
- Zope Object Database (ZODB) 157
- Zope
 - reference 157