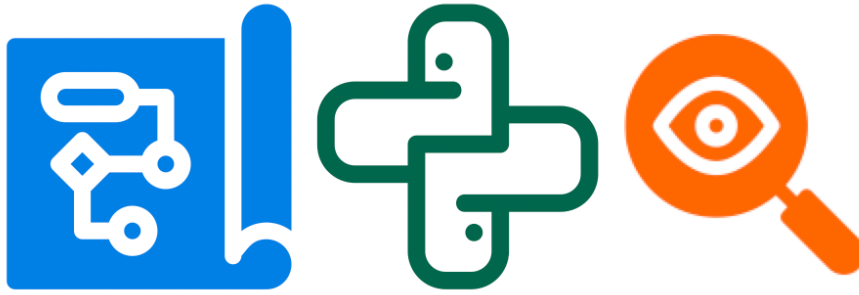


Design Patterns in Python for the Untrained Eye

Ariel Ortizariel.ortiz.ramirez@gmail.com Version 1.0.0, May 1, 2019.

Table of Contents

1. Introduction
 - 1.1. What is a Pattern?
 - 1.2. Pattern Categories
 - 1.3. Relevance of Design Patterns
 - 1.4. Limitations of Design Patterns
 - 1.5. A Bit of History
 - 1.6. Design Principles
 - 1.7. Anatomy of a Design Pattern
 2. Singleton Pattern
 - 2.1. Intent
 - 2.2. Motivation
 - 2.3. Structure
 - 2.4. Implementation
 - 2.4.1. Exercise A ★
 3. Template Method Pattern
 - 3.1. Intent
 - 3.2. Motivation
 - 3.3. Structure
 - 3.4. Implementation
 - 3.4.1. Example: An Average Calculator
 - 3.4.2. Exercise B ★★
 4. Adapter Pattern
 - 4.1. Intent
 - 4.2. Motivation
 - 4.3. Structure
 - 4.4. Implementation
 - 4.4.1. Example: A File Average Calculator Adapter
 - 4.4.2. Exercise C ★
 5. Observer Pattern
 - 5.1. Intent
 - 5.2. Motivation
 - 5.3. Structure
 - 5.4. Implementation
 - 5.4.1. Example: Observing Employee Salary Changes
 - 5.4.2. Exercise D ★★★
 6. Recommended Readings
 7. License and Credits
-



Tutorial @ [PyCon USA 2019](#). May 1, 2019. Cleveland, OH. U.S.A.

Check out the [slides](#) for this tutorial.

1. Introduction

1.1. What is a Pattern?

A pattern is something that you did in the past, was successful, and can be applied to multiple situations. Patterns capture experiences in software development that have been proven to work again and again, and thus provide a solution to specific problems. They are not invented. Instead, they are discovered from practical experience.

When many programmers are trying to solve similar problems they arrive again and again at a solution that works best. Such a solution is later distilled into a solution template, something that we programmers then use to approach similar problems in the future. Such solution templates are often called patterns.

It is important to note that patterns provide only a template for a solution and not a detailed recipe. You will still have to take care of the code and make sure that the pattern implementation makes sense and works well with the rest of the program.

1.2. Pattern Categories

Programming patterns can be split into three groups that cover different abstraction levels:

Idioms

Idioms represent the lowest-level patterns. They address aspects of both design and implementation. Most idioms are language-specific — they capture existing programming experience. Often the same idiom looks different for different languages, and sometimes an idiom that is useful for one programming language does not make sense in another.

Some examples of useful Python idioms are:

- [Chained comparison operators](#):

```
# Don't do this:
if 0 < x and x < 10:
    print('x is greater than 0 but less than 10')

# Instead, do this:
if 0 < x < 10:
    print('x is greater than 0 but less than 10')
```

- Top-level script environment:

```
# Execute only if run as a script and not as a module
if __name__ == '__main__':
    print('Hello from script!')
```

- Conditional expressions:

```
# This statement:
if x < 5:
    return 10
else:
    return 20

# Can be reduced to this one:
return 10 if x < 5 else 20
```

- Indexing during iteration:

```
# Don't do this:
index = 0
for value in collection:
    print(index, value)
    index += 1

# Nor this:
for index in range(len(collection)):
    value = collection[index]
    print(index, value)

# Definitely don't do this:
index = 0
while index < len(collection):
    value = collection[index]
    print(index, value)
    index += 1

# Instead, do this:
for index, value in enumerate(collection):
    print(index, value)
```

Architectural Patterns

Architectural patterns are templates for concrete software architectures. They specify the system-wide structural properties of an application, and have an impact on the architecture of its subsystems. The selection of an architectural pattern is therefore a fundamental design decision when developing a software system.

A few examples of architectural patterns are:

- Pipes and Filters

- Model-View-Controller (MVC)
- Microservices

Design Patterns

Design patterns are medium-scale patterns. They are smaller in scale than architectural patterns, programming language independent, but usually specific to a particular programming paradigm. The application of a design pattern has no effect on the fundamental structure of a software system, but may have a strong influence on the architecture of a subsystem.

In this tutorial we will be looking at the following object-oriented design patterns:

- Singleton
- Template Method
- Adapter
- Observer

1.3. Relevance of Design Patterns

- Design patterns are a toolkit of **tried and tested solutions** to common problems in software design. Even if you never encounter these problems, knowing patterns is still useful because it teaches you how to solve all sorts of problems using principles of object-oriented design.
- Design patterns define a **common language** that you and your teammates can use to communicate more efficiently.

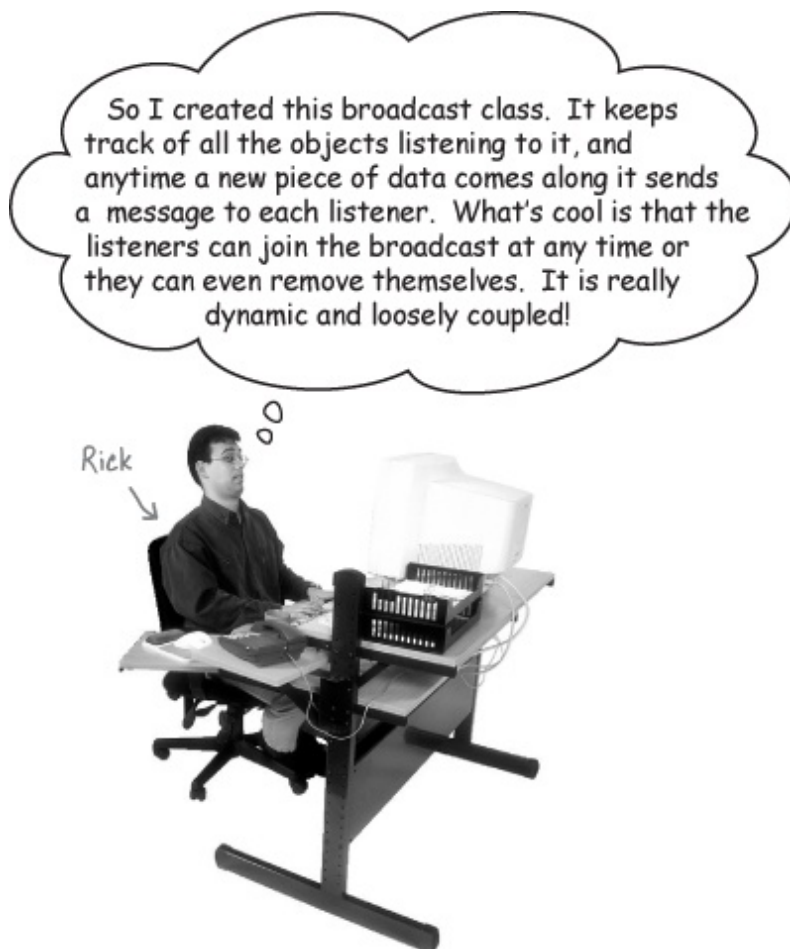


Figure 1. Source: [FREEMAN]

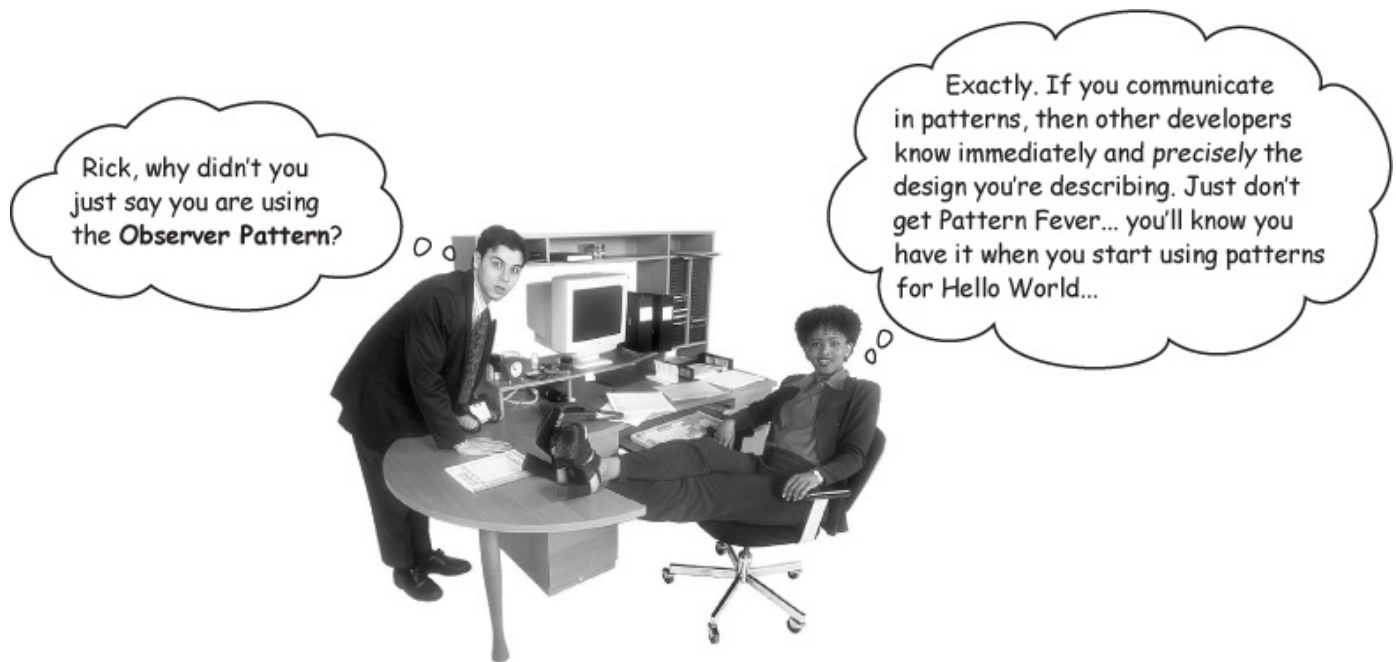


Figure 2. Source: [FREEMAN]

1.4. Limitations of Design Patterns

Here are the most common criticisms against the use of design patterns:

Unjustified Use

This is the problem that haunts many novices who have just acquainted themselves with patterns. Having learned about patterns, they try to apply them everywhere, even in situations where simpler code would do just fine.

Kludges for a Weak Programming Language

Usually the need for patterns arises when people choose a programming language or a technology that lacks the necessary level of abstraction. In this case, patterns become a kludge that gives the language much-needed abilities.

Inefficient Solutions

Design patterns may add memory and processing overhead, so sometimes they might not be appropriate for applications running in resource constrained systems.

1.5. A Bit of History

Wheel reinvention is a constant problem for software engineers. In 1995, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides set out to redirect all the effort going into building redundant software wheels into something more useful. That year, building on the work of Christopher Alexander, Kent Beck, and others, they published *Design Patterns: Elements of Reusable Object-Oriented Software*. The book was an instant hit, with the authors rapidly becoming famous (at least in software engineering circles) as the Gang of Four (GoF).

The GoF did two things for us. First, they introduced most of the software engineering world to the idea of design patterns, where each pattern is a prepackaged solution to a common design problem. We should look around, they wrote, and identify common solutions to common problems. We should give each solution a name, and we should talk about what that solution is good for, when to use it, and when to reach for something else. And we should write all of this information down, so that over time our palette of design solutions will grow.

Second, the GoF identified, named, and described 23 patterns. The original 23 solutions were the recurring patterns that the GoF saw as key to building clean, well-designed object-oriented programs. In the years since *Design Patterns* was published, people have described patterns in everything from real-time micro-controllers to enterprise architectures. But the original 23 GoF patterns stuck to the middle ground of object-

oriented design and focused on some key questions: How do objects like the ones you tend to find in most systems relate to one another? How should they be coupled together? What should they know about each other? How can we swap out parts that are likely to change frequently?

The original 23 design patterns were classified as follows based on their use:

- **Creational Patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
- **Structural Patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
- **Behavioral Patterns** take care of effective communication and the assignment of responsibilities between objects.



Figure 3. The Gang of Four, from left to right: Ralph Johnson, Erich Gamma, Richard Helm, and John Vlissides.

1.6. Design Principles

The GoF used the following principles as guidelines when they developed their catalog of design patterns:

Separate out the things that change from those that stay the same

Take the parts that vary in your program and encapsulate them, so that later you can alter or extend these parts without affecting those that don't.

Program to an interface, not an implementation

When you are presented with some programming interface (be it a class library, a set of functions, a network protocol or anything else) you should only use the things guaranteed by that interface. You may have knowledge about the underlying implementation (you may have written it), but you should not use that knowledge.

Prefer composition over inheritance

Composition is much more versatile than inheritance because it allows you to change behavior at runtime. Inheritance is easy to abuse, and beginners could avoid many naive mistakes by trying to use composition first.

Delegation

An object expresses certain behavior to the outside but in reality delegates responsibility for implementing that behaviour to an associated object. Delegation is like inheritance done manually through object composition.

1.7. Anatomy of a Design Pattern

For this tutorial each design pattern will be elaborated using a *template* that consists of the following sections:

- **Intent:** This section briefly describes both the problem and the solution.
- **Motivation:** This sections further explains the problem and the solution the pattern makes possible.

- **Structure:** This section shows each part of the pattern and how they are related.
- **Implementation:** This section contains Python examples and/or exercises to make it easier to grasp the idea behind the pattern.

2. Singleton Pattern

2.1. Intent

Singleton is a **creational design pattern** that lets you ensure that a class has only one instance, while providing a global access point to this instance.

2.2. Motivation

It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created, and it can provide a way to access the instance.

2.3. Structure

The Singleton pattern UML diagram looks like this:

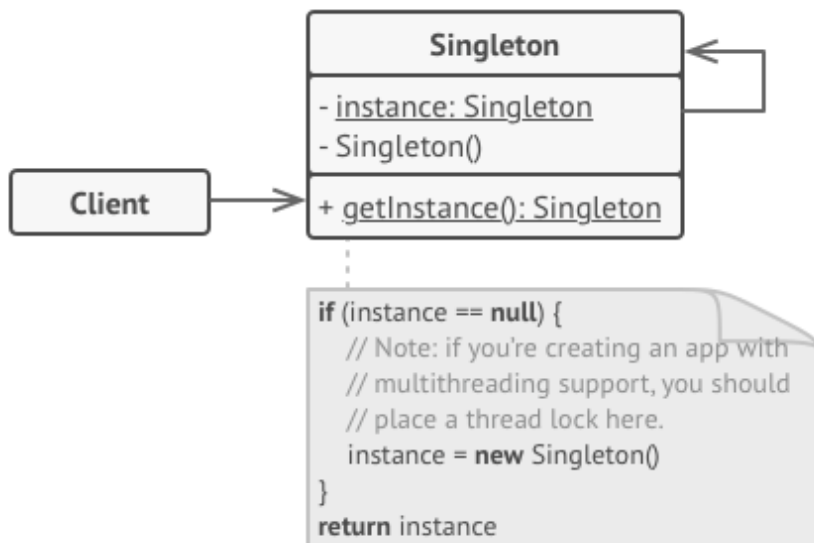


Figure 4. Source [Refactoring.Guru](#)

2.4. Implementation

2.4.1. Exercise A ★

You have the following class definition that is meant to model the famous Tigger character from A. A. Milne's "Winnie The Pooh" books:

File: *tigger.py*

```
class Tigger:

    def __str__(self):
        return "I'm the only one!"

    def roar(self):
        return 'Grrr!'
```

It seems reasonable to expect that there should be only one Tigger object (after all, he is the only one!), but the current implementation allows having multiple distinct Tigger objects:

File: singleton.py

```
from tigger import Tigger

a = Tigger()
b = Tigger()

print(f'ID(a) = {id(a)}')
print(f'ID(b) = {id(b)}')
print(f'Are they the same object? {a is b}')
```

The previous code prints something like this:

```
ID(a) = 139744614641560
ID(b) = 139744614096968
Are they the same object? False
```

Refactor the `Tigger` class into a Singleton following these steps in the `tigger` module:

1. Make the `Tigger` class private by appending to its name a single leading underscore.
2. Create a private module-scoped variable called `_instance` and initialize it with `None`.
3. Define a function called `Tigger` that takes no arguments. This function is responsible for creating an instance of the `_Tigger` class only once and returning a reference to this unique instance every time it gets called. To accomplish this you'll need to use the `_instance` variable from the previous point.

Running `singleton.py` again the new output should now look like this:

```
ID(a) = 139744614097640
ID(b) = 139744614097640
Are they the same object? True
```

This implementation of the Singleton pattern works fine in most cases, yet it might seem a bit like a hack. An alternative cleaner but more advanced Python implementation using metaclasses is available at [Refactoring.Guru](#).

3. Template Method Pattern

3.1. Intent

Template Method is a **behavioral design pattern** that defines the skeleton of an algorithm in the base class but lets derived classes override specific steps of the algorithm without changing its structure.

3.2. Motivation

The Template Method pattern suggests that you break down an algorithm into a series of steps, turn these steps into methods, and put a series of calls to these methods inside a single “template method”. The steps may either be abstract, or have some default implementation. To use the algorithm, the client is supposed to provide its own subclass, implement all abstract steps, and override some of the optional ones if needed (but not the template method itself).

3.3. Structure

This is the Template Method pattern UML diagram:

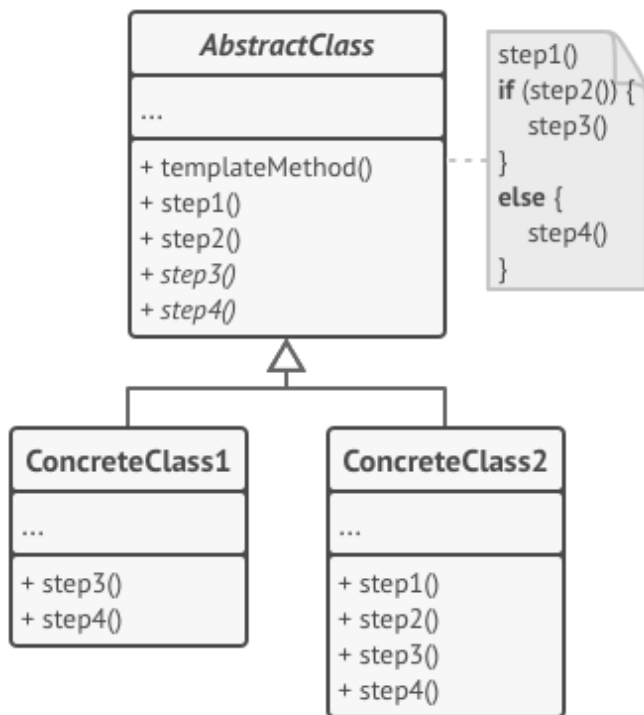


Figure 5. Source [Refactoring.Guru](#)

3.4. Implementation

3.4.1. Example: An Average Calculator

The following code shows a class with a template method that allows computing the average of a series of numbers provided by some sort of sequential data source.

File: calculator.py

```

from abc import ABC, abstractmethod

class AverageCalculator(ABC):

    def average(self):
        try:
            num_items = 0
            total_sum = 0
            while self.has_next():
                total_sum += self.next_item()
                num_items += 1
            if num_items == 0:
                raise RuntimeError("Can't compute the average of zero items.")
            return total_sum / num_items
        finally:
            self.dispose()

    @abstractmethod
    def has_next(self):
        pass

    @abstractmethod
    def next_item(self):
        pass

    def dispose(self):
        pass

```

The `AverageCalculator` class is an abstract base class (`ABC`).

A central feature of an abstract base class is that it cannot be instantiated directly. Instead, an abstract base class is meant to be used as a base class for other classes that are expected to implement the required methods (those that have the `@abstractmethod` decorator). Abstract base classes are used in code that wants to enforce an expected programming interface.

- The `average` method is the *template method*. Observe that it calls at some point all the declared abstract methods, thus, any concrete class that inherits from `AverageCalculator` must implement them.
- The `has_next` abstract method has the following contract: it returns `True` if the current data source object is able to produce at least one more item, otherwise returns `False`.
- The `next_item` abstract method has the following contract: it returns the next available item from the current data source object. The result is undefined if there are no more available items.
- The `dispose` method has the following contract: it will be called in order to free any resources when all the items from the current data source object have been consumed. The default implementation does nothing and doesn't have to be overridden (because is not decorated with `@abstractmethod`).

The `FileAverageCalculator` class extends `AverageCalculator` and represents a source of sequential numerical data obtained from a text file provided during the construction of the object.

File: calculator.py (continued)

```
class FileAverageCalculator(AverageCalculator):

    def __init__(self, file):
        self.file = file
        self.last_line = self.file.readline()

    def has_next(self):
        return self.last_line != ''

    def next_item(self):
        result = float(self.last_line)
        self.last_line = self.file.readline()
        return result

    def dispose(self):
        self.file.close()
```

- Note that you need to send an already opened file when instantiating the class. The file will be automatically closed (when calling the `dispose` method) at the end of the `average` template method.
- We need to call the `readline` method anticipatedly because that's how we know if we have reached the end of file.
- The `readline` method returns an empty string when the end of file is found.

We'll use the following text file to test our program:

File: data.txt

```
4
8
15
16
23
42
```

This is how we would use the `FileAverageCalculator` class:

File: calculator.py (continued)

```
fac = FileAverageCalculator(open('data.txt'))
print(fac.average()) # Call the template method
```

The output should be:

```
18.0
```

3.4.2. Exercise B ★★

In the `calculator.py` file, write a new class called `MemoryAverageCalculator`, which must inherit from the `AverageCalculator` class. This new class should be a source of sequential numerical data obtained from a list provided during the construction of the object. You should be able to use the class like this:

```
mac = MemoryAverageCalculator([3, 1, 4, 1, 5, 9, 2, 6, 5, 3])
print(mac.average()) # Call the template method
```

The expected output should be:

```
3.9
```

4. Adapter Pattern

4.1. Intent

Adapter is a **structural design pattern** that converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

4.2. Motivation

Reusing existing components (objects, subsystems) is a common part of software development. Usually, it is better to reuse an existing solution than rewriting it from scratch, as the latter will inevitably take longer and introduce new bugs.

Using old components in new code, however, brings its own set of problems. Quite frequently, the newer code works against an interface that does not exactly match the existing functionality. We have to write an intermediate object, a kind of translator from the new interface to the old one. This object is called an adapter.

4.3. Structure

The UML diagram for the Adapter pattern is as follows:

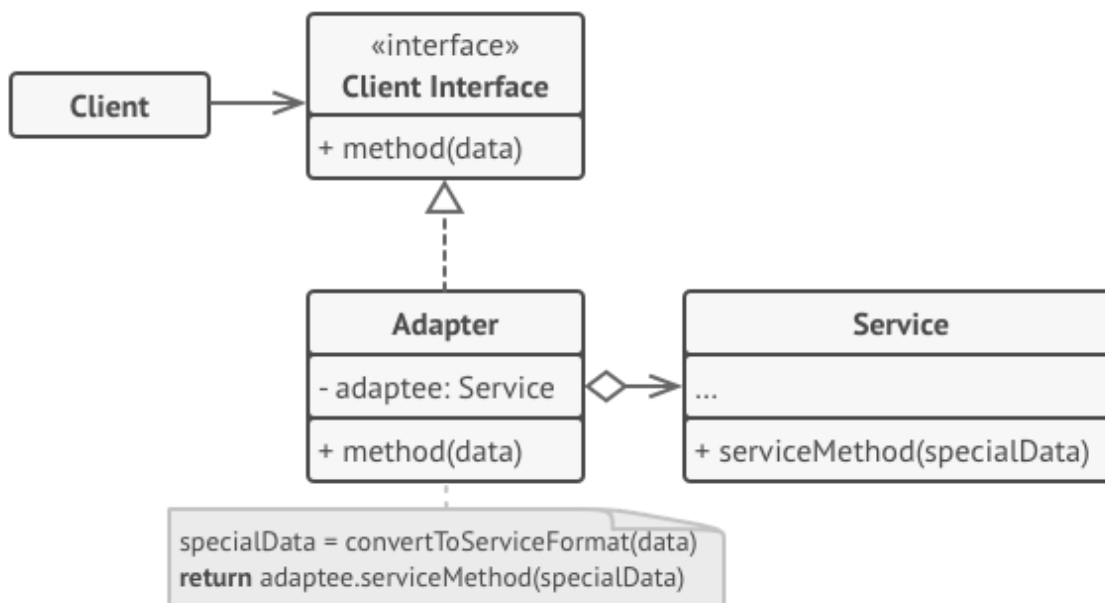


Figure 6. Source [Refactoring.Guru](#)

4.4. Implementation

4.4.1. Example: A File Average Calculator Adapter

In this example we will use generator expressions. These are somewhat similar to list comprehensions, but they construct a list object. Instead of creating a list and keeping the whole sequence in the memory, the generator generates the next element on demand. Syntactically, generator expressions use parentheses in place of square brackets. The following code uses a generator expression to create a generator `g` that is able to produce the first ten powers of 2.

```
>>> g = (2 ** i for i in range(10))
>>> next(g) # Get first power of 2.
1
>>> next(g) # Get second power of 2.
2
>>> next(g) # Get third power of 2.
4
>>> list(g) # Get, as a list, the remaining seven powers of 2.
[8, 16, 32, 64, 128, 256, 512]
```

We want to use the `AverageCalculator` code defined previously to compute the average of a sequence of numbers produced by a generator expression. The most obvious option would be to keep using the Template Method design pattern: create a new class that extends the `AverageCalculator` abstract base class and then implement all its abstract methods. But let's take a different approach in order to demonstrate how to use the Adapter pattern.

We already have the `FileAverageCalculator` class that knows how to process sequential numerical data obtained from a text file. Note that both a text file and a generator work practically the same way: you access their elements sequentially, but with a catch. For the text file you call the `readline` method, yet for the generator you call the `next` function. So, they have the same behavior, but with a different interface. This is a job for the Adapter pattern.

Reviewing the code of the `FileAverageCalculator` class we can see that it only uses two methods specific to a text file object: `readline` and `close`. So an adapter class would only need to provide these two methods, plus the `__init__` method to do any required initialization. With this information we can now define the `GeneratorAdapter` class:

File: calculator.py (continued)

```
class GeneratorAdapter:

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

    def readline(self):
        try:
            return next(self.adaptee)
        except StopIteration:
            return ''

    def close(self):
        pass
```

- The `__init__` method receives the generator it will be adapting (a.k.a. the *adaptee*) and stores it in an instance variable.
- The `readline` method delegates its job to the adaptee by calling the `next` function.

- The `readline` contract establishes that when the end of the file has been reached it should return an empty string. For a generator, the equivalent of an “end of file” is when it can no longer generate more elements. The `next` function raises a `StopIteration` exception when called with an exhausted generator.
- A generator has no equivalent “closing” operation, yet we do need to provide a `close` method even if it does nothing because the `FileAverageCalculator.dispose` method calls it.

This is how the new adapter class should be used:

File: calculator.py (continued)

```
from random import randint

g = (randint(1, 100) for i in range(1000000))
fac = FileAverageCalculator(GeneratorAdapter(g))
print(fac.average()) # Call the template method
```

- We create a generator that generates one million random numbers between 1 and 100.
- We create a `FileAverageCalculator` object, but instead of using a file we actually initialize it with the previously created generator wrapped inside the corresponding adapter.

The expected output should look something like:

```
50.486323
```

The result may vary slightly given that we are working with random numbers.

4.4.2. Exercise C ★

The following problem was adapted from [FREEMAN] pp. 238-240. Assume you have the following two Python classes:

File: poultry.py

```
class Duck:

    def quack(self):
        print('Quack')

    def fly(self):
        print("I'm flying")

class Turkey:

    def gobble(self):
        print('Gobble gobble')

    def fly(self):
        print("I'm flying a short distance")
```

You want your turkeys to behave like ducks, so you need to apply the Adapter pattern. In the same file, write a class called `TurkeyAdapter` and make sure it takes into account the following:

- The adapter's `__init__` method should take its adaptee as an argument.
- The `quack` translation between classes is easy: just call the `gobble` method when appropriate.
- Even though both classes have a `fly` method, turkeys can only fly in short spurts — they can't do long-distance flying like ducks. To map between a duck's `fly` method and the turkey's method, you need to call the turkey's `fly` method five times to make up for it.

Test your class with the following code:

File: poultry.py (continued)

```
def duck_interaction(duck):
    duck.quack()
    duck.fly()

duck = Duck()
turkey = Turkey()
turkey_adapter = TurkeyAdapter(turkey)

print('The Turkey says...')
turkey.gobble()
turkey.fly()

print('\nThe Duck says...')
duck_interaction(duck)

print('\nThe TurkeyAdapter says...')
duck_interaction(turkey_adapter)
```

The expected output is as follows:

```
The Turkey says...
Gobble gobble
I'm flying a short distance

The Duck says...
Quack
I'm flying

The TurkeyAdapter says...
Gobble gobble
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
I'm flying a short distance
```

5. Observer Pattern

5.1. Intent

Observer is a **behavioral design pattern** that defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

5.2. Motivation

The cases when certain objects need to be informed about the changes occurred in other objects are frequent. To have a good design means to decouple as much as possible and to reduce the dependencies. The Observer design pattern can be used whenever a subject (a *publisher* object) has to be observed by one or more observers (the *subscriber* objects).

For example, if you subscribe to a newspaper or magazine, you no longer need to go to the store to check if the next issue is available. Instead, the publisher sends new issues directly to your mailbox right after publication or even in advance.

The publisher maintains a list of subscribers and knows which magazines they're interested in. Subscribers can leave the list at any time when they wish to stop the publisher sending new magazine issues to them.

5.3. Structure

This is the Observer pattern UML class diagram:

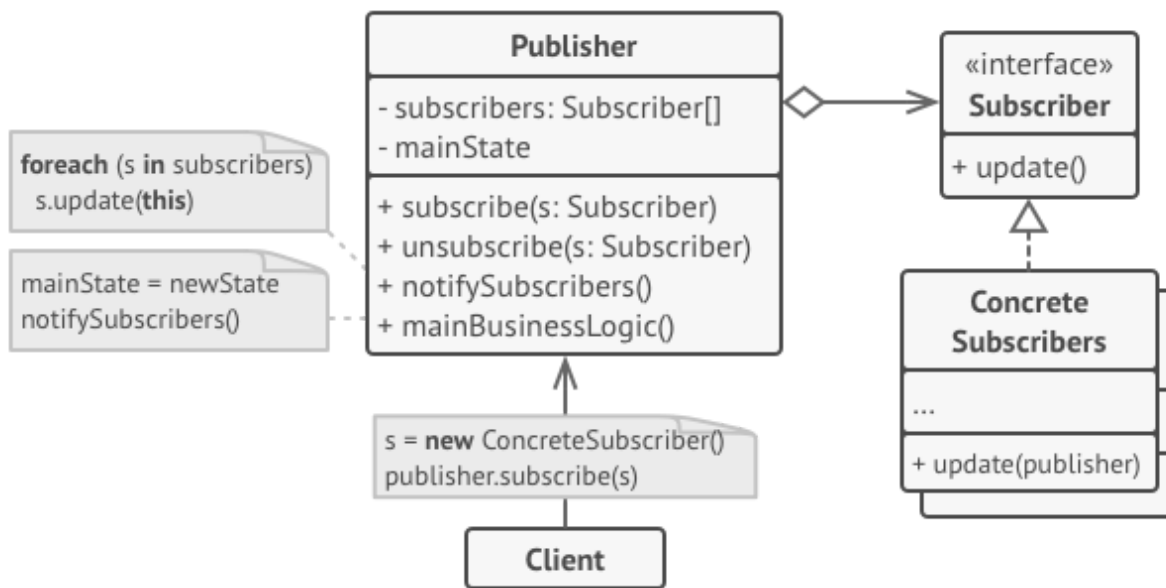


Figure 7. Source [Refactoring.Guru](#)

5.4. Implementation

5.4.1. Example: Observing Employee Salary Changes

First, we'll define the `Observer` and `Observable` classes. These classes provide us the support we require to implement the pattern in most typical cases:

File: `observer.py`


```
from abc import ABC, abstractmethod

class Observer(ABC):

    @abstractmethod
    def update(self, observable, *args):
        pass

class Observable:

    def __init__(self):
        self.__observers = []

    def add_observer(self, observer):
        self.__observers.append(observer)

    def delete_observer(self, observer):
        self.__observers.remove(observer)

    def notify_observers(self, *args):
        for observer in self.__observers:
            observer.update(self, *args)
```

The following code shows how to use these classes. An `Employee` instance is an observable object (publisher). Every time its salary is modified all its registered observer objects (subscribers) get notified. We provide two concrete observer classes for our demo:

- `Payroll`: A class responsible for paying the salary to an employee.
- `TaxMan`: A class responsible for collecting taxes from the employee.

File: observer.py (continued)

```

class Employee(Observable):
    def __init__(self, name, salary):
        super().__init__()
        self._name = name
        self._salary = salary

    @property
    def name(self):
        return self._name

    @property
    def salary(self):
        return self._salary

    @salary.setter
    def salary(self, new_salary):
        self._salary = new_salary
        self.notify_observers(new_salary)

class Payroll(Observer):
    def update(self, changed_employee, new_salary):
        print(f'Cut a new check for {changed_employee.name}! '
              f'Her/his salary is now {new_salary}!')

class TaxMan(Observer):
    def update(self, changed_employee, new_salary):
        print(f'Send {changed_employee.name} a new tax bill!')

```

- A publisher class needs to extend the `Observable` class.
- It's essential that we call the `__init__` method of the base class (`Observable`) using the `super` function. Otherwise, the private attribute `_observers` (the list that will hold this object's observers) will not be created.
- All the subscribers get notified when the salary is updated through its property setter function.
- The `Payroll` and `TaxMan` classes extend the `Observer` class because they are subscribers. These classes need to provide the implementation of the `update` method, which acts as a callback.

Putting everything together:

File: observer.py (continued)

```

e = Employee('Amy Fowler Fawcett', 50000)
p = Payroll()
t = TaxMan()

e.add_observer(p)
e.add_observer(t)

print('Update 1')
e.salary = 60000

e.delete_observer(t)

print('\nUpdate 2')
e.salary = 65000

```

When running this code the expected output is:

```

Update 1
Cut a new check for Amy Fowler Fawcett! Her/his salary is now 60000!
Send Amy Fowler Fawcett a new tax bill!

Update 2
Cut a new check for Amy Fowler Fawcett! Her/his salary is now 65000!

```

5.4.2. Exercise D ★★★

In the `observer.py` file write a class called `Twitter` that inherits from the `Observable` and `Observer` classes so that it effectively behaves simultaneously as a publisher and as a subscriber.

Remember that inheriting from two classes is possible in Python because it supports multiple inheritance.

Review carefully the following code demo so that you may know which methods you need to implement and their corresponding behavior:

File: observer.py (continued)

```

a = Twitter('Alice')
k = Twitter('King')
q = Twitter('Queen')
h = Twitter('Mad Hatter')
c = Twitter('Cheshire Cat')

a.follow(c).follow(h).follow(q)
k.follow(q)
q.follow(q).follow(h)
h.follow(a).follow(q).follow(c)

print(f'==== {q.name} tweets ====')
q.tweet('Off with their heads!')
print(f'\n==== {a.name} tweets ====')
a.tweet('What a strange world we live in.')
print(f'\n==== {k.name} tweets ====')
k.tweet('Begin at the beginning, and go on till you come to the end: then stop.')
print(f'\n==== {c.name} tweets ====')
c.tweet("We're all mad here.")
print(f'\n==== {h.name} tweets ====')
h.tweet('Why is a raven like a writing-desk?')

```



The `follow` method allows cascading (multiple method calls on the same object).

The trick here is to have the method return `self`.

The expected output is:

```
==== Queen tweets ====
Alice received a tweet from Queen: Off with their heads!
King received a tweet from Queen: Off with their heads!
Queen received a tweet from Queen: Off with their heads!
Mad Hatter received a tweet from Queen: Off with their heads!



==== Alice tweets ====
Mad Hatter received a tweet from Alice: What a strange world we live in.




==== King tweets ====

==== Cheshire Cat tweets ====
Alice received a tweet from Cheshire Cat: We're all mad here.
Mad Hatter received a tweet from Cheshire Cat: We're all mad here.

==== Mad Hatter tweets ====
Alice received a tweet from Mad Hatter: Why is a raven like a writing-desk?
Queen received a tweet from Mad Hatter: Why is a raven like a writing-desk?
```

6. Recommended Readings

Author		Title	Notes
		Design Patterns at the Refactoring.Guru web site	<i>One of the best online resources containing a comprehensive catalog with the original 23 GoF design patterns. Contains code examples in Java, C#, PHP, Python, Ruby, Swift, and TypeScript.</i>
[SHVETS]		Alexander Shvets. <i>Dive Into Design Patterns.</i> Refactoring.Guru, 2018.	<i>This is an e-book (PDF, EPUB, MOBI) available from sourcemaking.com. Code samples are written in pseudocode that doesn't constrain the material to a particular programming language. The book also comes with an archive that includes code examples in Java, C#, PHP, Python, Swift, and TypeScript.</i>

Author		Title	Notes
[FREEMAN]		<p>Elisabeth Freeman, Eric Freeman, Bert Bates, & Kathy Sierra.</p> <p><i>Head First Design Patterns.</i></p> <p>O'Reilly, 2004. ISBN: 0596007124</p>	<p><i>This book is an unorthodox, visually intensive, reader-involving combination of puzzles, jokes, nonstandard layout, and an engaging, conversational style devised to immerse the reader in the topic of design patterns. All the code examples are in Java.</i></p>
[OLSEN]		<p>Russ Olsen.</p> <p><i>Design Patterns in Ruby.</i></p> <p>Addison-Wesley, 2007. ISBN: 0321490452</p>	<p><i>A fine, detailed and pragmatic treatment of core design patterns translated into Ruby.</i></p>
[GAMMA]		<p>Erich Gamma, Richard Helm, Ralph Johnson, & John M. Vlissides.</p> <p><i>Design Patterns: Elements of Reusable Object-Oriented Software.</i></p> <p>Addison-Wesley, 1994. ISBN: 0201633612</p>	<p><i>This is the classic software design patterns book by the Gang of Four (GoF). Source code examples are in C++ and Smalltalk.</i></p>

7. License and Credits

- Free use of the source code presented here is granted under the terms of the [GPL version 3 License](#).
- This document was prepared using the [AsciiDoctor](#) text processor.
- Icons made by [Freepik](#) from www.flaticon.com is licensed by [CC BY 3.0](#).

Version 1.0.0

Last updated 2019-05-01 15:18:01 UTC