Design Patterns In Python

Common GoF (Gang of Four) Design Patterns

Implemented In Python

Sean Bradley

Table of contents

1. Design Patterns In Python

Hello, I'm Sean Bradley, and welcome to my book on Design Patterns in Python.

For over 20 years I have been an IT engineer building and managing real time, low latency, high availability, asynchronous, multi threaded, remotely managed, fully automated, monitored solutions in the education, aeronautical, banking, drone, gaming and telecommunications industries.

I have also created and written hundreds of Open Source GitHub Repositories, Medium Articles and video tutorials on YouTube, Udemy and Skillshare.

This book focuses on the 23 famous GoF (Gang of Four) Design Patterns implemented in Python.

A Design Pattern is a description or template that can be repeatedly applied to a commonly recurring problem in software design.

A familiarity of Design Patterns will be very useful when planning, discussing, managing and documenting your applications from now on and into the future.

Also, throughout the book, as each design pattern is discussed and demonstrated using example code, I also introduce new python coding concepts with each new design pattern. So that as you progress through the book and try out the examples, you will also get experience and familiarity with some of the finer details of programming with python.

So, in this book, you will learn about these 23 Design Patterns,

- Creational
  - Factory
  - Abstract Factory
  - Builder
  - Prototype
  - Singleton
- Structural
  - Decorator
  - Adapter
  - Facade
  - Bridge
  - Composite
  - Flyweight
  - Proxy

- Behavioral
    - Command
    - Chain of Responsibility
    - Observer Pattern
    - Interpreter
    - Iterator
    - Mediator
    - Memento
    - State
    - Strategy
    - Template
    - Visitor

1.1 Pattern Types

In the list of patterns above, there are Creational, Structural and Behavioral patterns.

- **Creational** : Abstracts the instantiation process so that there is a logical separation between how objects are composed and finally represented.
- **Structural** : Focuses more on how classes and objects are composed using the different structural techniques, and to form structures with more or altered flexibility.
- **Behavioral** : Are concerned with the inner algorithms, process flow, the assignment of responsibilities and the intercommunication between objects.

1.2 Class Scope and Object Scope Patterns

Each pattern can be further specified whether it relates more specifically to classes or instantiated objects.

Class scope patterns deal more with relationships between classes and their subclasses.

Object scope patterns deal more with relationships that can be altered at runtime

Pattern	Description	Scope	Type
Factory, Abstract Factory	Defers object creation to subclasses	Class	Creational

Pattern	Description	Scope	Type
Builder, Prototype, Singleton	Defers object creation to objects	Object	Creational
Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Proxy	Describes a way to assemble objects	Object	Structural
Interpreter, Template	Describes algorithms and flow control	Class	Behavioral
Chain of Responsibility, Command, Iterator, Mediator, Memento, Observer, State, Strategy, Visitor	Describes how groups of objects co-operate	Object	Behavioral

2. Development Environment Setup

SBCODE Video ID #d29be7

The most universal approach to installing Python is to visit the official Python download page at,

https://www.python.org/downloads/

Normally this page will detect your operating system from the `useragent` in your browser and select which install is appropriate for you.

There will be 64 and 32 bit versions for your operating system. At the time of writing this book, the option of downloading the 64bit version was the most common, and the version was 3.9.x.

The code in this book will be using a simplified generic style of Python that should work in all versions since Python version 3.

To test if you already have python on your system, depending on your operating system, whether Windows, Linux or Mac OSX, open a terminal/bash/cmd or PowerShell prompt.

and type

```
python -V
```

Note the capital `V` in the above command.

Sometimes python is named as `python3`

So you can also try

```
python3 -V
```

You are looking for a response that indicates you have Python 3 or above installed. Not an error, or `Python 2.x`

On my windows workstation, if I use PowerShell, the response is

```
PS> python -V
Python 3.9.2
```

I have **Python3** already installed using the official python link from above.

If you are using a recent version of Linux or Mac OSX, then the command to check for the Python version on your system is most likely to be,

```
python3 -V
```

Remember to follow the official install instructions for your operating system at https://www.python.org/downloads/

2.1 Example Code

All the code examples in this book can be viewed from my GitHub repository at https://github.com/Sean-Bradley/Design-Patterns-In-Python

If you have Git installed, you can download a copy locally using the command

```
git clone https://github.com/Sean-Bradley/Design-Patterns-In-Python.git
```

or you can download a zip of all the code, using the link

https://sbcode.net/python/zips/Design-Patterns-In-Python.zip

or using `wget` on Linux

```
wget https://sbcode.net/python/zips/Design-Patterns-In-Python.zip
sudo apt install unzip
unzip Design-Patterns-In-Python.zip
cd Design-Patterns-In-Python/
```

You can then experiment with the code at your own pace and try out different variations.

If you would rather type the code from the book, then follow these further recommendations.

On my system, I have created a working folder on one of my spare drives, `E:\`, and then created a new folder in it named `python_design_patterns`, and then `cd` into it. You can use a different folder name if you prefer.

```
PS C:\> e:
PS E:\> mkdir python_design_patterns
PS E:\> cd .\python_design_patterns
```

Each section will be in a new folder named after the design pattern section.

I.e,. The code that I write for the **Factory** pattern will be in its own subfolder named `factory`

```
PS E:\python_design_patterns> mkdir factory
PS E:\python_design_patterns> cd .\factory
PS E:\python_design_patterns\factory>
```

2.2 Course Videos

As part of the purchase of this book, I have also provided you with the ability to view for free, all of the videos which are part of my official **Design Patterns in Python** courses on Udemy, YouTube and Skillshare.

To view the videos, in each section of this book, you will find several **SBCODE Video IDs**. Visit https://sbcode.net/python/ and at the beginning of each instructional page on the website, there are options to enter the code and view the related video. Press the **SBCODE** button in the **Video Links** section on the website and then enter the code that you found in each related section of this book.

2.3 VSCode

If you are working on Windows, then I recommend to also install VSCode to use as your IDE when learning Python.

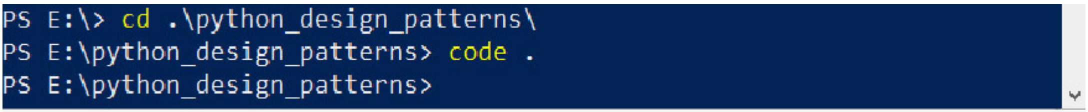
This is optional and you may prefer to use Notepad or any other popular IDE that you can download or use online that will also assist you when writing Python.

You can download VSCode from https://code.visualstudio.com/download

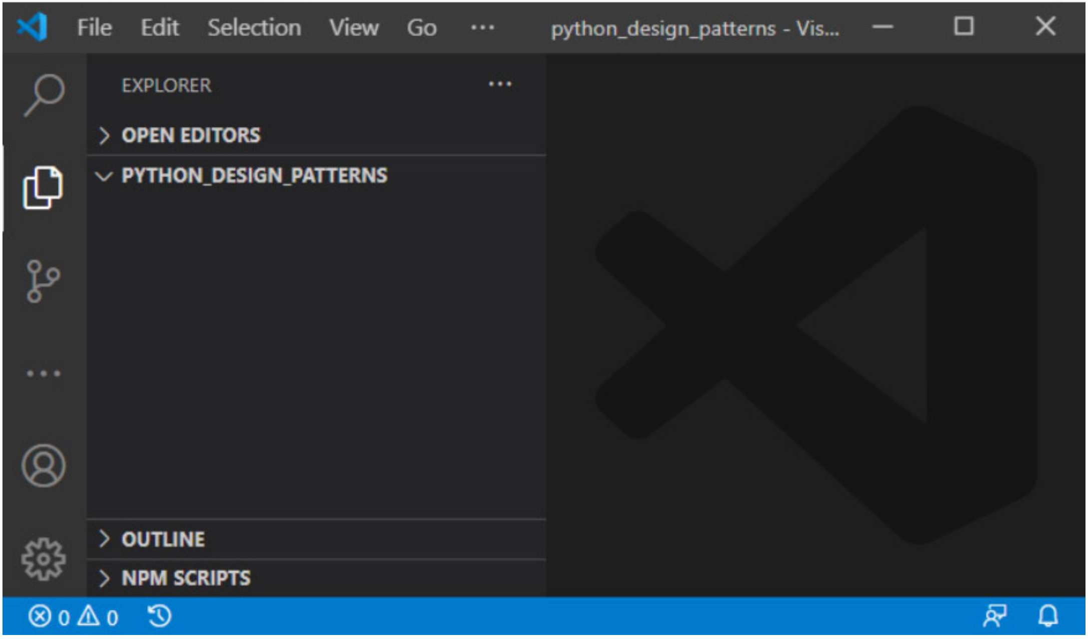
You can then open VSCode from your current working folder by typing `code .`

```
PS E:\python_design_patterns> code .
```

And VSCode will open ready for you in your working folder where you can use it to add new files or folders as needed.



```
PS E:\> cd .\python_design_patterns\
PS E:\python_design_patterns> code .
PS E:\python_design_patterns>
```



3. Coding Conventions

SBCODE Video ID #29949d

3.1 Python Interactive Console Versus *.py Scripts

You can execute Python code by writing your scripts into text files and commonly using the `.py` extension. Text files on most operating systems will be UTF-8 encoded by default. Python also reads UTF-8 encoded text files by default.

Create a new text file called `example.py` and add the following text.

```
print("Hello World!")
```

and then you can execute it using `python` or `python3` depending on your operating system and Python version.

```
PS> python ./example.py
Hello World!
```

You can also enter Python code directly into the Python Interactive Console by typing just `python` or `python3` from the command line and then press `Enter` . You then get a prompt like below.

```
PS> python
Python 3.9.2 (tags/v3.9.2:1a79785, Feb 19 2021, 13:44:55) [MSC ...
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can now enter python commands directly.

```
>>> print("Hello World!")
Hello World!
>>>
```

To exit the Python Interactive Console, you can usually type `quit()` or press `Ctrl-Z` then press `Enter`

This book will show examples of using both `*.py` scripts and the interactive console to execute Python. Look out for the `>>>` characters in the code blocks to indicate if I was using the Python Interactive Console or a `*.py` script.

3.2 PEP8

The code styling in this book is formatted using mostly PEP8 styling recommendations.

- **UPPER_CASE** : Constants will be defined using UPPER_CASE naming style.
- **PascalCase** : Class names will use PascalCase naming style
- **snake_case** : For variables names, method names and method arguments.
- **Docstrings** : Classes and methods contain extra documentation that is descriptive text enclosed in " or """ for multiline strings.
- **_leading_underscore** : Use a leading underscore to indicate variables that should be considered as private class variables.

See PEP-0008 : https://www.python.org/dev/peps/pep-0008/

3.3 Pylint

I use the Pylint tool to check for code styling recommendations.

On most operating systems you would generally install Pylint by using the `PIP` or `PIP3` installer.

```
pip install pylint
```

If using VSCode, open the Command Palette (Ctrl+Shift+P), then set the

`Python: Enable Linting` to `on`

and

`Python: Select Linter` to `Pylint`

3.4 Common Pylint Warning and Error Messages

ID	Message	Description
R0201	Method could be a function (no-self-use)	Your method has an attribute that refers to `self` or `cls`, but it is not necessary since your are **NOT** using `self` or `cls` within the method body. You have the option of using the `@staticmethod` decorator on your methods instead, and to remove the `self` or `cls` from the method attributes.

ID	Message	Description
R0903	Too few public methods (1/2) (too-few-public-methods)	The error assumes that your class may be used for just storing data. You could use a dictionary instead. However the assumption is not always correct. You may be extending a class, or often in my case, I am trying to keep examples very small, readable and to the point. So you have the option to insert a pylint declaration at the top of the file, or at a particular method declaration to ignore this pylint error. `# pylint: disable=too-few-public-methods`
E0110	Abstract class '*ClassName*' with abstract methods instantiated (abstract-class-instantiated)	The Class that implements the abstract interface, or is inheriting from another abstract class, is not implementing all of the abstract methods as described in the interfaces signature; or if extending, then all of the signatures of the abstract class that is being extended.
W0221	Parameters differ from overridden '*method*' method (arguments-differ)	The arguments in your abstract class don't match the arguments in your implementing class. Check spelling of arguments.
C0304	Final newline missing (missing-final-newline)	Pylint preferes a file to end with a new line. When copying code from a webpage into a `.py` file, the copied code may not finish with a new line character. You can add one manually by pressing the `enter` key on your keyboard at the end of your code, or if you use VSCode, pressing the key combination of **SHIFT-ALT-F** will auto format your `*.py` file with a final newline when you have the Pylint linter, or other linter, enabled.
W0612	Unused variable	You can remove the unused variable from your code. If you cannot remove the unused variable then use a `_` as the variable name. See the section the-underscore-only-_variable in the Mediator pattern for more information.

3.5 Command Line Interfaces

Command Line Interfaces (CLI) on different operating systems (Windows, Linux, MacOSX, RaspberryPI) vary in appearance quite a lot.

You can use CMD, PowerShell or Git BASH on Windows, Bash on Linux or Terminal on MacOSX.

```
-- Windows PowerShell --
PS> python example.py
PS E:\python_design_patterns> python example.py

-- Windows CMD --
C:\> python example.py

-- Git BASH
Username@hostname MINGW64 /e/python_design_patterns
$ python example.py

-- Linux --
user@domain:~# python3 example.py
user@domain:$/ python3 example.py
user@domain:/python-design-patterns# python3 example.py
$ python3 example.py
# python3 example.py

-- MacOSX--
hostname:~ username$ python3 example.py
```

Wikipedia - Command-line interface : https://en.wikipedia.org/wiki/Command-line_interface

4. UML Diagrams

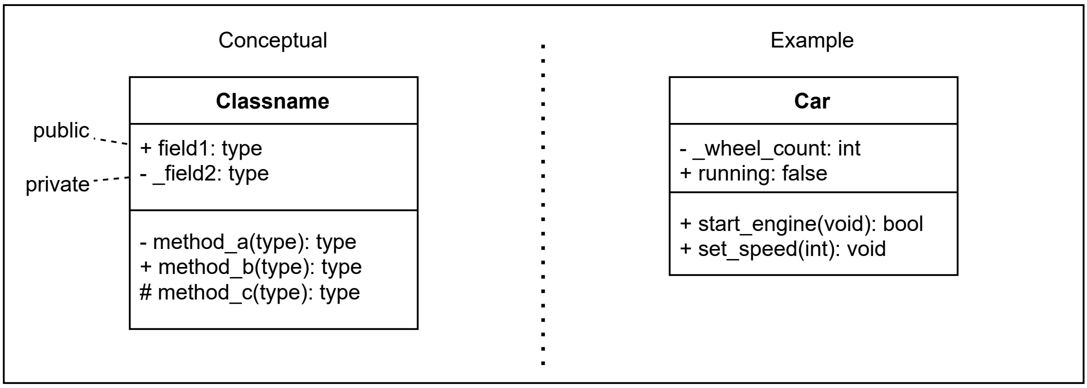
SBCODE Video ID #735229

Unified Modeling Language (UML) Diagrams are used throughout this book to help describe the patterns.

Below are some example self describing UML diagrams.

The left part of the diagram shows the basic concept, and the right side shows a potential example usage.

4.1 A Basic Class

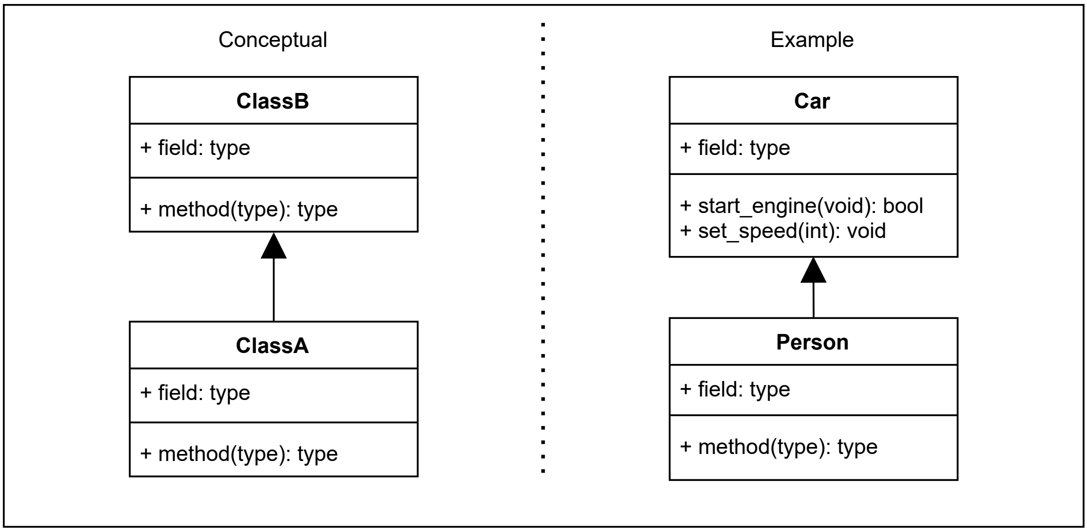


4.2 Directed Association

A filled arrow with a line.

ClassA uses **ClassB** or an object of **ClassB**.

ClassA calls a static class method, a static abstract method or a method/property/field from an object of type ClassB. eg, The **Person** starts the **Car** engine.

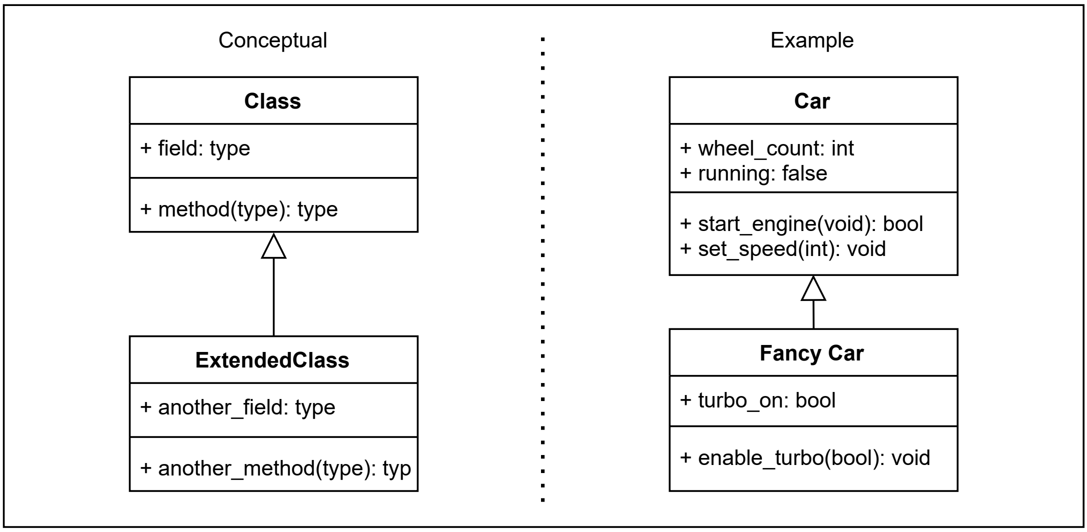


4.3 A Class That Extends/Inherits A Class

An unfilled arrow, with a line pointing to the class that is being extended/inherited.

ClassA extends **ClassB**.

The extended class contains all of the attributes/fields and methods of the inherited class, including its own extra methods, attributes/fields, overrides and overloads.

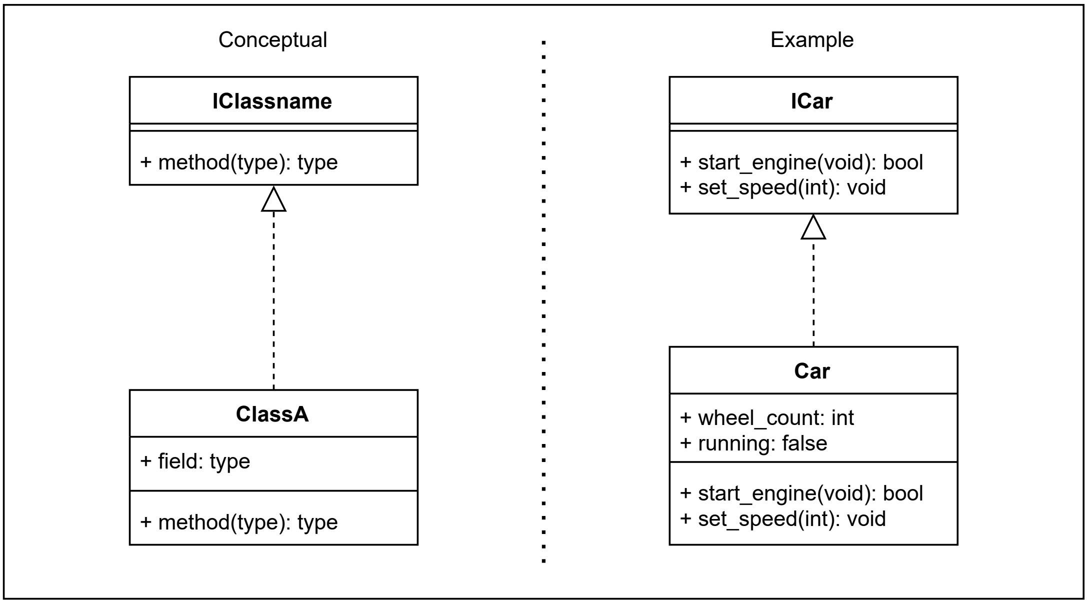


4.4 A Class That Implements An Interface

An unfilled arrow, with a dashed line pointing to the interface that is being implemented.

ClassA implements **ClassB**.

A class that implements an interface must implement all of the methods declared in the interface.

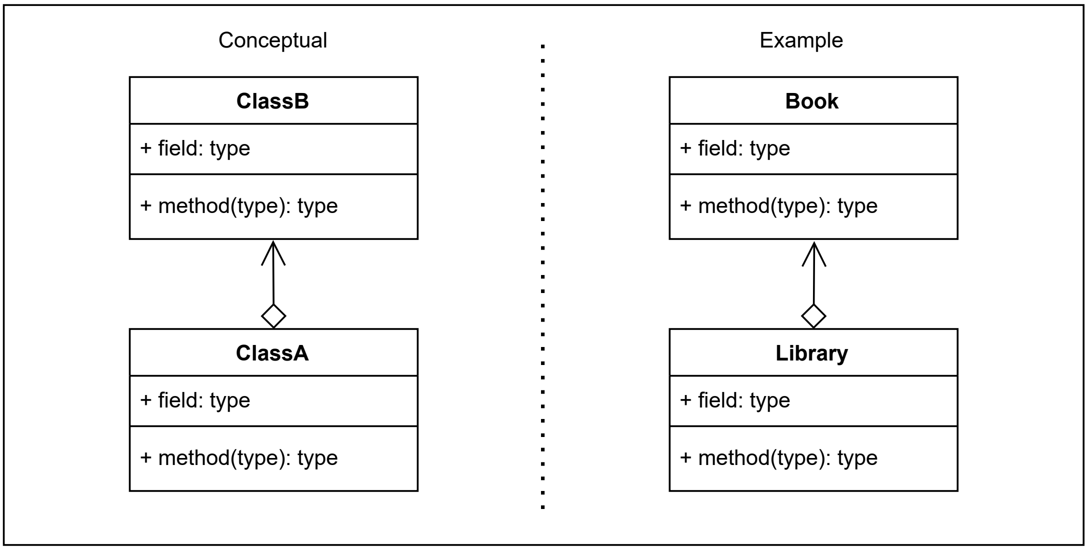


4.5 Aggregates

An unfilled diamond with a line and arrow head.

ClassA aggregates **ClassB**.

Library aggregates **Books**. Books and Library can exist independently of each other. Books can exist without the Library.

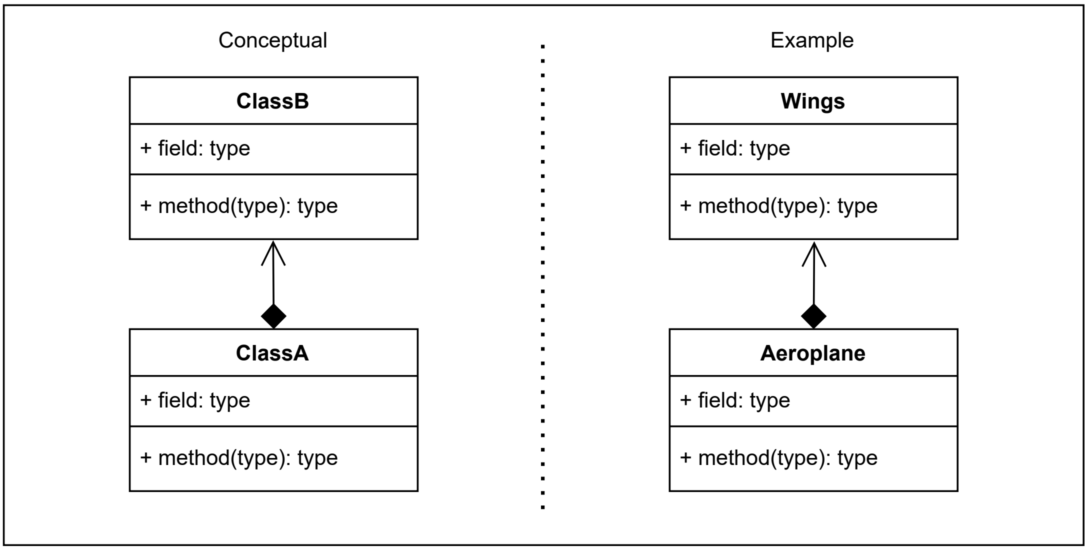


4.6 Composition

A filled diamond with a line and arrow head.

ClassA is composed of **ClassB**

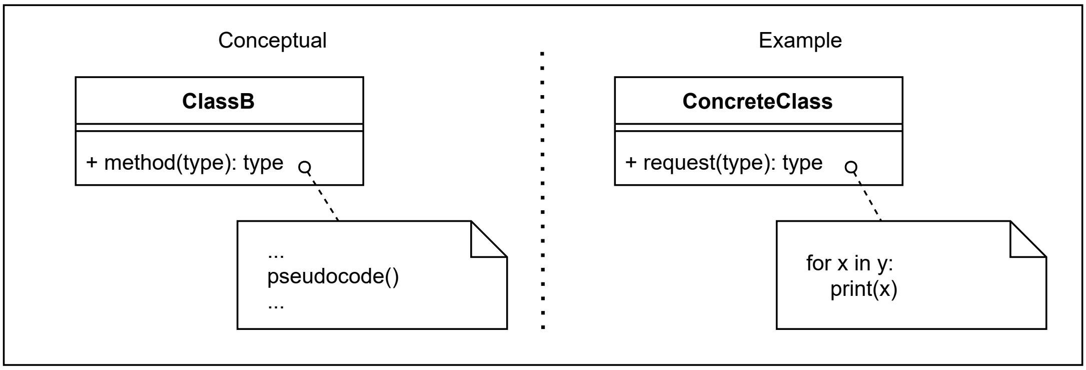
Aeroplane can be composed of **Wings** and other parts. But an aeroplane is no longer really an aeroplane without its wings.



4.7 Pseudocode Annotation

A box with a dashed line and a circle placed near a class method.

Pseudocode is a plain language description of the steps in an algorithm and used to portray a concept without needing to write long lines of code.



5. Creational

5.1 Factory Design Pattern

5.1.1 Overview

SBCODE Video ID #85a1c6

When developing code, you may instantiate objects directly in methods or in classes. While this is quite normal, you may want to add an extra abstraction between the creation of the object and where it is used in your project.

You can use the **Factory** pattern to add that extra abstraction. The Factory pattern is one of the easiest patterns to understand and implement.

Adding an extra abstraction will also allow you to dynamically choose classes to instantiate based on some kind of logic.

Before the abstraction, your client, class or method would directly instantiate an object of a class. After adding the factory abstraction, the concrete product (object) is now created outside of the current class/method, and now in a subclass instead.

Imagine an application for designing houses and the house has a chair already added on the floor by default. By adding the factory pattern, you could give the option to the user to choose different chairs, and how many at runtime. Instead of the chair being hard coded into the project when it started, the user now has the option to choose.

Adding this extra abstraction also means that the complications of instantiating extra objects can now be hidden from the class or method that is using it.

This separation also makes your code easier to read and document.

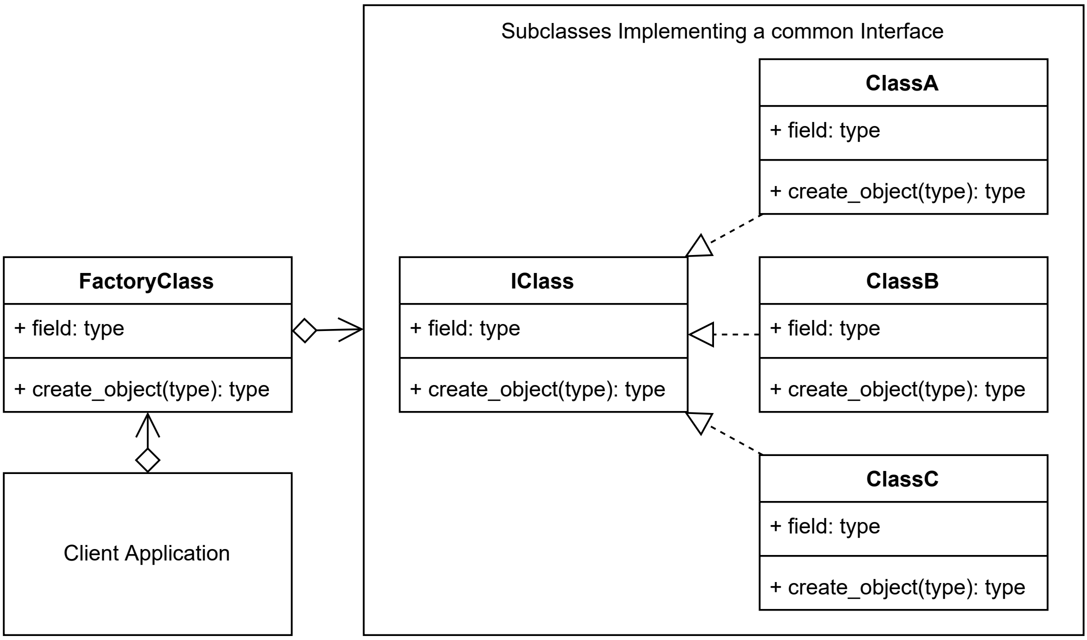
The Factory pattern is really about adding that extra abstraction between the object creation and where it is used. This gives you extra options that you can more easily extend in the future.

5.1.2 Terminology

- **Concrete Creator**: The client application, class or method that calls the Creator (Factory method).
- **Product Interface**: The interface describing the attributes and methods that the Factory will require in order to create the final product/object.
- **Creator**: The Factory class. Declares the Factory method that will return the object requested from it.

- **Concrete Product**: The object returned from the Factory. The object implements the Product interface.

5.1.3 Factory UML Diagram


Subclasses Implementing a common Interface
ClassA
+ field: type
+ create_object(type): type
FactoryClass
+ field: type
+ create_object(type): type
IClass
+ field: type
+ create_object(type): type
ClassB
+ field: type
+ create_object(type): type
Client Application
ClassC
+ field: type
+ create_object(type): type

5.1.4 Source Code

In this concept example, the client wants an object named `b`

Rather than creating `b` directly in the client, it asks the creator (factory) for the object instead.

The factory finds the relevant class using some kind of logic from the attributes of the request. It then asks the subclass to instantiate the new object that it then returns as a reference back to the client asking for it.

./factory/factory_concept.py

```
# pylint: disable=too-few-public-methods
"The Factory Concept"
from abc import ABCMeta, abstractmethod

class IProduct(metaclass=ABCMeta):
    "A Hypothetical Class Interface (Product)"

    @staticmethod
    @abstractmethod
```

```python
    def create_object():
        "An abstract interface method"

class ConcreteProductA(IProduct):
    "A Concrete Class that implements the IProduct interface"

    def __init__(self):
        self.name = "ConcreteProductA"

    def create_object(self):
        return self

class ConcreteProductB(IProduct):
    "A Concrete Class that implements the IProduct interface"

    def __init__(self):
        self.name = "ConcreteProductB"

    def create_object(self):
        return self

class ConcreteProductC(IProduct):
    "A Concrete Class that implements the IProduct interface"

    def __init__(self):
        self.name = "ConcreteProductC"

    def create_object(self):
        return self

class Creator:
    "The Factory Class"

    @staticmethod
    def create_object(some_property):
        "A static method to get a concrete product"
        if some_property == 'a':
            return ConcreteProductA()
        if some_property == 'b':
            return ConcreteProductB()
        if some_property == 'c':
            return ConcreteProductC()
        return None

# The Client
PRODUCT = Creator().create_object('b')
print(PRODUCT.name)
```

5.1.5 Output

```
python ./factory/factory_concept.py
ConcreteProductB
```

5.1.6 Factory Use Case

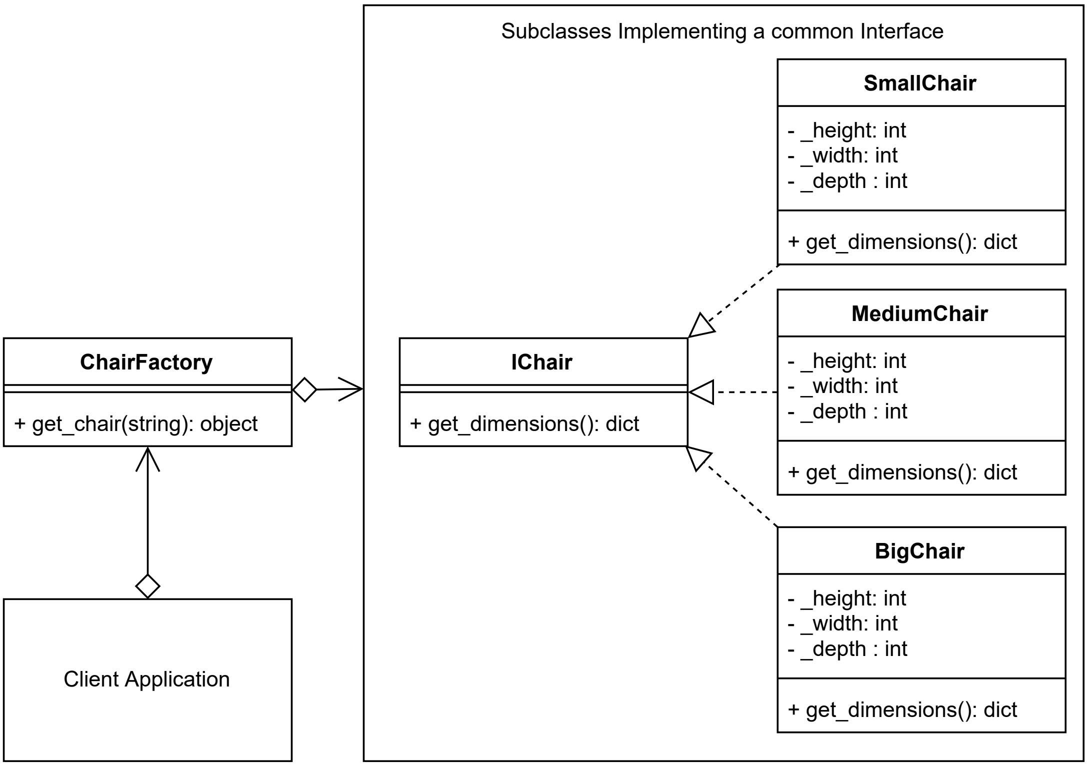
SBCODE Video ID #5d7340

An example use case is a user interface where the user can select from a menu of items, such as chairs.

The user has been given a choice using some kind of navigation interface, and it is unknown what choice, or how many the user will make until the application is actually running and the user starts using it.

So, when the user selected the chair, the factory then takes some property involved with that selection, such as an ID, Type or other attribute and then decides which relevant subclass to instantiate in order to return the appropriate object.

5.1.7 Factory Example UML Diagram



5.1.8 Source Code

./factory/client.py

```python
"Factory Use Case Example Code"

from chair_factory import ChairFactory

# The Client
CHAIR = ChairFactory().get_chair("SmallChair")
print(CHAIR.get_dimensions())
```

./factory/interface_chair.py

```python
# pylint: disable=too-few-public-methods
"The Chair Interface"
from abc import ABCMeta, abstractmethod

class IChair(metaclass=ABCMeta):
    "The Chair Interface (Product)"

    @staticmethod
    @abstractmethod
    def get_dimensions():
        "A static interface method"
```

./factory/chair_factory.py

```python
"The Factory Class"

from small_chair import SmallChair
from medium_chair import MediumChair
from big_chair import BigChair

class ChairFactory:  # pylint: disable=too-few-public-methods
    "The Factory Class"

    @staticmethod
    def get_chair(chair):
        "A static method to get a chair"
        if chair == 'BigChair':
            return BigChair()
        if chair == 'MediumChair':
            return MediumChair()
        if chair == 'SmallChair':
            return SmallChair()
        return None
```

./factory/small_chair.py

```
# pylint: disable=too-few-public-methods
"A Class of Chair"
from interface_chair import IChair

class SmallChair(IChair):
    "The Small Chair Concrete Class implements the IChair interface"

    def __init__(self):
        self._height = 40
        self._width = 40
        self._depth = 40

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

./factory/medium_chair.py

```
# pylint: disable=too-few-public-methods
"A Class of Chair"
from interface_chair import IChair

class MediumChair(IChair):
    "The Medium Chair Concrete Class implements the IChair interface"

    def __init__(self):
        self._height = 60
        self._width = 60
        self._depth = 60

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

./factory/big_chair.py

```
# pylint: disable=too-few-public-methods
"A Class of Chair"
from interface_chair import IChair

class BigChair(IChair):
    "The Big Chair Concrete Class implements the IChair interface"

    def __init__(self):
        self._height = 80
        self._width = 80
        self._depth = 80

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

5.1.9 Output

```
python ./factory/client.py
{'width': 40, 'depth': 40, 'height': 40}
```

5.1.10 New Coding Concepts

ABCMeta

SBCODE Video ID #e6bc73

ABCMeta classes are a development tool that help you to write classes that conform to a specified interface that you've designed.

ABCMeta refers to **A**bstract **B**ase **C**lasses.

The benefits of using ABCMeta classes to create abstract classes is that your IDE and Pylint will indicate to you at development time whether your inheriting classes conform to the class definition that you've asked them to.

Abstract interfaces are not instantiated directly in your scripts, but instead implemented by subclasses that will provide the implementation code for the abstract interface methods. E.g., you don't create `IChair` , but you create `SmallChair` that implements the methods described in the `IChair` interface.

An abstract interface method is a method that is declared, but contains no implementation. The implementation happens at the class that inherits the abstract class.

You don't need to use ABCMeta classes and interfaces that you have created in your final python code. You code will still work without them.

You can try it by removing the interfaces from all of the chair classes above, and you will see that your python program will still run.

eg, change

```
class BigChair(IChair):
```

to

```
class BigChair():
```

and it will still work.

While it is possible to ensure your classes are correct without using abstract classes, it is often easier to use abstract classes as a backup method of checking correctness, especially if your projects become very large and involve many developers.

Note that in all my code examples, the abstract classes are prefixed with a capital **I**, to indicate that they are abstract interfaces. They have no code in their methods. They do not require a `self` or `cls` argument due to the use of `@staticmethod` . The inheriting class will implement the code in each of the methods that the abstract class is describing. If subclasses are inheriting an abstract base class, and they do not implement the methods as described, there will be Pylint error or warning message (E0110).

See PEP 3119 : https://www.python.org/dev/peps/pep-3119/

5.1.11 Summary

- The Factory Pattern is an Interface that defers the creation of the final object to a subclass.
- The Factory pattern is about inserting another layer/abstraction between instantiating an object and where in your code it is actually used.
- It is unknown what or how many objects will need to be created until runtime.
- You want to localize knowledge of the specifics of instantiating a particular object to the subclass so that the client doesn't need to be concerned about the details.
- You want to create an external framework, that an application can import/reference, and hide the details of the specifics involved in creating the final object/product.

- The unique factor that defines the Factory pattern, is that your project now defers the creation of objects to the subclass that the factory had delegated it to.

5.2 Abstract Factory Design Pattern

5.2.1 Overview

SBCODE Video ID #62bde8

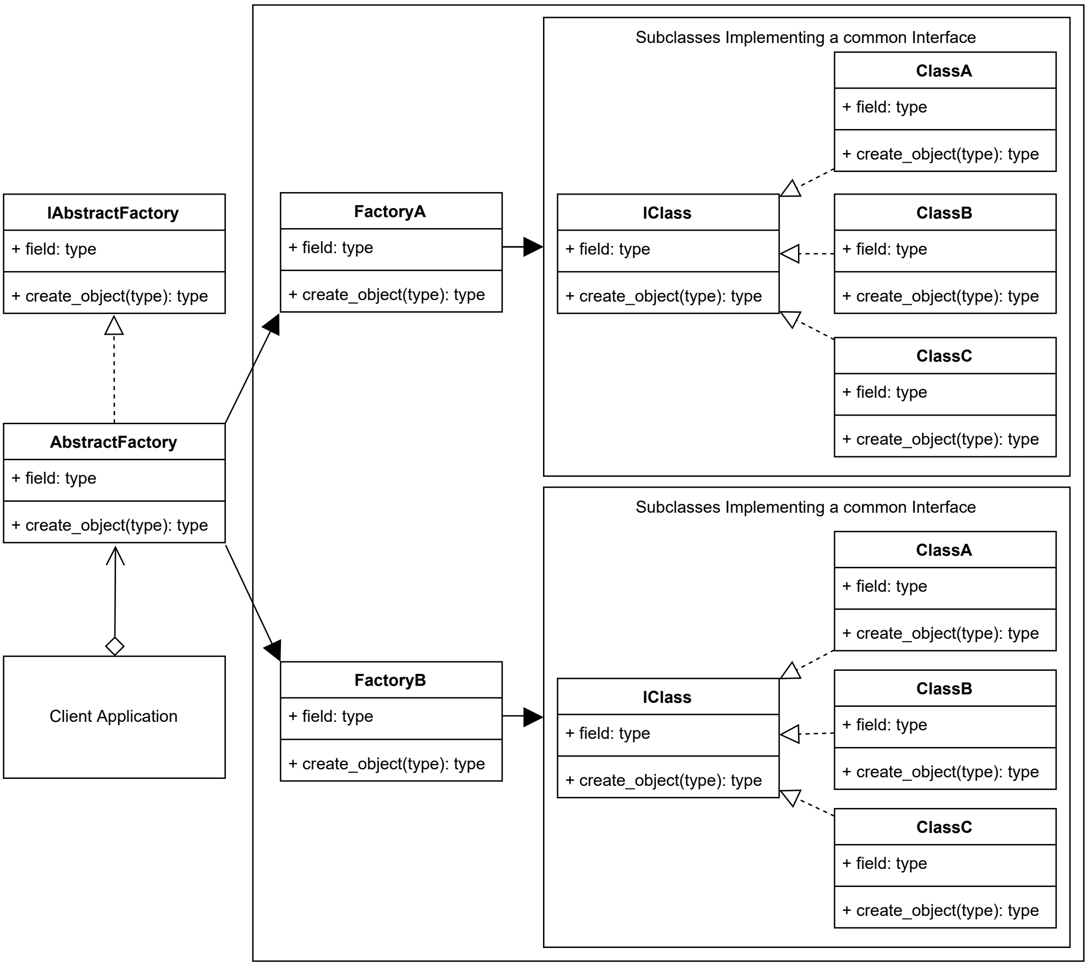
The Abstract Factory Pattern adds an abstraction layer over multiple other creational pattern implementations.

To begin with, in simple terms, think if it as a Factory that can return Factories. Although you will find examples of it also begin used to return Builder, Prototypes, Singletons or other design pattern implementations.

5.2.2 Terminology

- **Client**: The client application that calls the **Abstract Factory**. It's the same process as the **Concrete Creator** in the Factory design pattern.
- **Abstract Factory**: A common interface over all of the sub factories.
- **Concrete Factory**: The sub factory of the **Abstract Factory** and contains method(s) to allow creating the **Concrete Product**.
- **Abstract Product**: The interface for the product that the sub factory returns.
- **Concrete Product**: The object that is finally returned.

5.2.3 Abstract Factory UML Diagram


IAbstractFactory
+ field: type
+ create_object(type): type
AbstractFactory
+ field: type
+ create_object(type): type
Client Application
FactoryA
+ field: type
+ create_object(type): type
FactoryB
+ field: type
+ create_object(type): type
Subclasses Implementing a common Interface
IClass
+ field: type
+ create_object(type): type
ClassA
+ field: type
+ create_object(type): type
ClassB
+ field: type
+ create_object(type): type
ClassC
+ field: type
+ create_object(type): type
Subclasses Implementing a common Interface
IClass
+ field: type
+ create_object(type): type
ClassA
+ field: type
+ create_object(type): type
ClassB
+ field: type
+ create_object(type): type
ClassC
+ field: type
+ create_object(type): type

5.2.4 Source Code

./abstract_factory/abstract_factory_concept.py

```
# pylint: disable=too-few-public-methods
"Abstract Factory Concept Sample Code"
from abc import ABCMeta, abstractmethod
from factory_a import FactoryA
from factory_b import FactoryB

class IAbstractFactory(metaclass=ABCMeta):
    "Abstract Factory Interface"

    @staticmethod
    @abstractmethod
    def create_object(factory):
        "The static Abstract factory interface method"

class AbstractFactory(IAbstractFactory):
    "The Abstract Factory Concrete Class"

    @staticmethod
    def create_object(factory):
        "Static get_factory method"
        try:
            if factory in ['aa', 'ab', 'ac']:
                return FactoryA().create_object(factory[1])
            if factory in ['ba', 'bb', 'bc']:
                return FactoryB().create_object(factory[1])
            raise Exception('No Factory Found')
        except Exception as _e:
            print(_e)
        return None

# The Client
PRODUCT = AbstractFactory.create_object('ab')
print(f"{PRODUCT.__class__}")

PRODUCT = AbstractFactory.create_object('bc')
print(f"{PRODUCT.__class__}")
```

./abstract_factory/factory_a.py

```
# pylint: disable=too-few-public-methods
"FactoryA Sample Code"
from abc import ABCMeta, abstractmethod

class IProduct(metaclass=ABCMeta):
    "A Hypothetical Class Interface (Product)"

    @staticmethod
    @abstractmethod
    def create_object():
        "An abstract interface method"

class ConcreteProductA(IProduct):
    "A Concrete Class that implements the IProduct interface"

    def __init__(self):
        self.name = "ConcreteProductA"

    def create_object(self):
        return self
```

```
class ConcreteProductB(IProduct):
    "A Concrete Class that implements the IProduct interface"

    def __init__(self):
        self.name = "ConcreteProductB"

    def create_object(self):
        return self

class ConcreteProductC(IProduct):
    "A Concrete Class that implements the IProduct interface"

    def __init__(self):
        self.name = "ConcreteProductC"

    def create_object(self):
        return self

class FactoryA:
    "The FactoryA Class"

    @staticmethod
    def create_object(some_property):
        "A static method to get a concrete product"
        try:
            if some_property == 'a':
                return ConcreteProductA()
            if some_property == 'b':
                return ConcreteProductB()
            if some_property == 'c':
                return ConcreteProductC()
            raise Exception('Class Not Found')
        except Exception as _e:
            print(_e)
        return None
```

./abstract_factory/factory_b.py

```
# pylint: disable=too-few-public-methods
"FactoryB Sample Code"
from abc import ABCMeta, abstractmethod

class IProduct(metaclass=ABCMeta):
    "A Hypothetical Class Interface (Product)"

    @staticmethod
    @abstractmethod
    def create_object():
        "An abstract interface method"

class ConcreteProductA(IProduct):
    "A Concrete Class that implements the IProduct interface"

    def __init__(self):
        self.name = "ConcreteProductA"

    def create_object(self):
        return self

class ConcreteProductB(IProduct):
    "A Concrete Class that implements the IProduct interface"

    def __init__(self):
        self.name = "ConcreteProductB"

    def create_object(self):
        return self

class ConcreteProductC(IProduct):
    "A Concrete Class that implements the IProduct interface"

    def __init__(self):
        self.name = "ConcreteProductC"

    def create_object(self):
        return self

class FactoryB:
    "The FactoryB Class"

    @staticmethod
    def create_object(some_property):
        "A static method to get a concrete product"
        try:
            if some_property == 'a':
                return ConcreteProductA()
            if some_property == 'b':
                return ConcreteProductB()
            if some_property == 'c':
                return ConcreteProductC()
            raise Exception('Class Not Found')
        except Exception as _e:
            print(_e)
        return None
```

5.2.5 Output

```
python ./abstract_factory/abstract_factory_concept.py
<class 'factory_a.ConcreteProductB'>
<class 'factory_b.ConcreteProductC'>
```

5.2.6 Abstract Factory Example Use Case

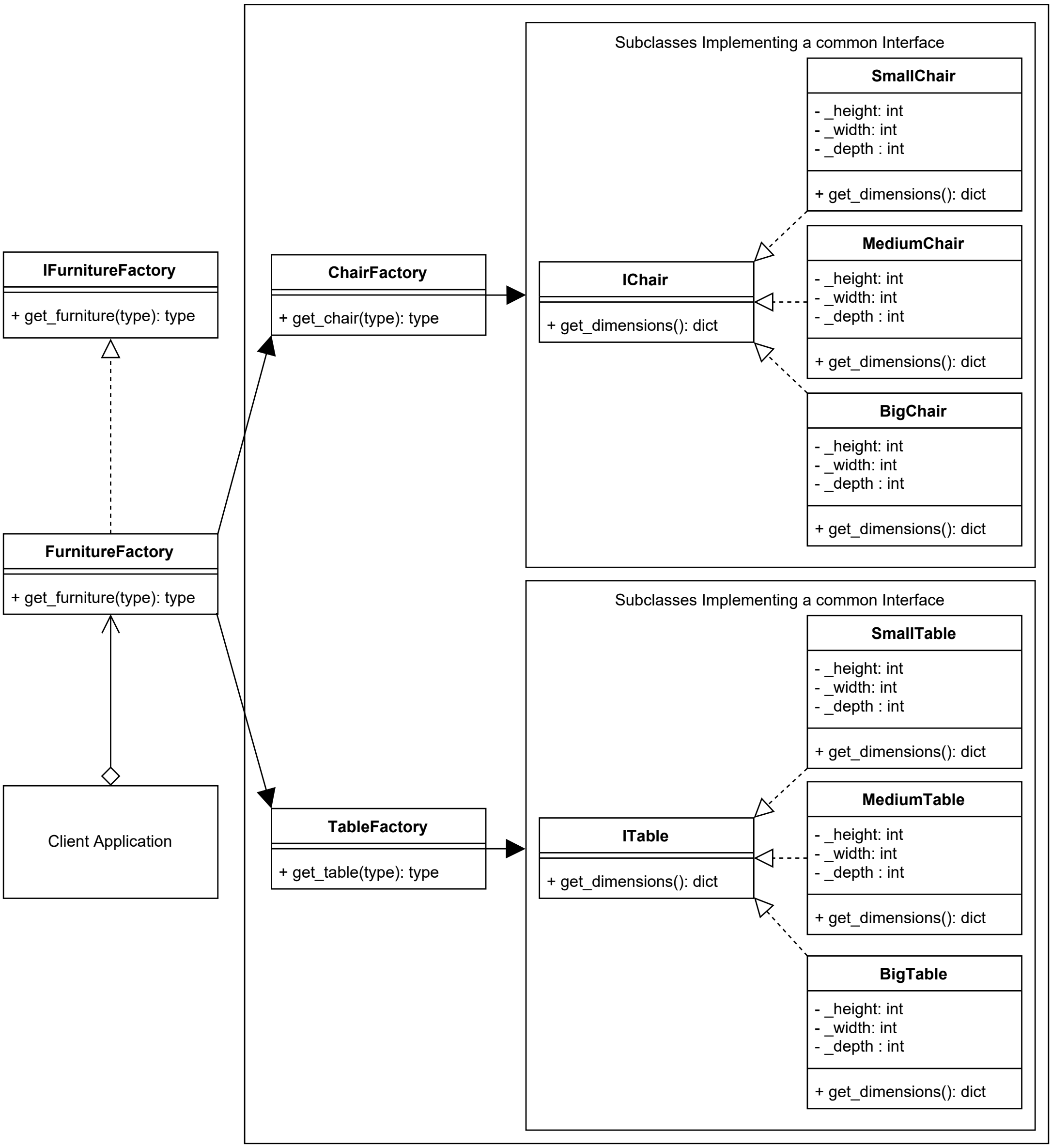
SBCODE Video ID #13899e

An example use case may be that you have a furniture shop front. You sell many different kinds of furniture. You sell chairs and tables. And they are manufactured at different factories using different unrelated processes that are not important for your concern. You only need the factory to deliver.

You can create an extra module called `FurnitureFactory` , to handle the chair and table factories, thus removing the implementation details from the client.

5.2.7 Abstract Factory Example UML Diagram

See this UML diagram of an Abstract Furniture Factory implementation that returns chairs and tables.



5.2.8 Source Code

./abstract_factory/client.py

```
"Abstract Factory Use Case Example Code"

from furniture_factory import FurnitureFactory

FURNITURE = FurnitureFactory.get_furniture("SmallChair")
print(f"{FURNITURE.__class__} : {FURNITURE.get_dimensions()}")
```

```
FURNITURE = FurnitureFactory.get_furniture("MediumTable")
print(f"{FURNITURE.__class__} : {FURNITURE.get_dimensions()}")
```

./abstract_factory/furniture_factory.py

```
# pylint: disable=too-few-public-methods
"Abstract Furniture Factory"
from interface_furniture_factory import IFurnitureFactory
from chair_factory import ChairFactory
from table_factory import TableFactory

class FurnitureFactory(IFurnitureFactory):
    "The Abstract Factory Concrete Class"

    @staticmethod
    def get_furniture(furniture):
        "Static get_factory method"
        try:
            if furniture in ['SmallChair', 'MediumChair', 'BigChair']:
                return ChairFactory().get_chair(furniture)
            if furniture in ['SmallTable', 'MediumTable', 'BigTable']:
                return TableFactory().get_table(furniture)
            raise Exception('No Factory Found')
        except Exception as _e:
            print(_e)
        return None
```

./abstract_factory/interface_furniture_factory.py

```
# pylint: disable=too-few-public-methods
"The Abstract Factory Interface"

from abc import ABCMeta, abstractmethod

class IFurnitureFactory(metaclass=ABCMeta):
    "Abstract Furniture Factory Interface"

    @staticmethod
    @abstractmethod
    def get_furniture(furniture):
        "The static Abstract factory interface method"
```

./abstract_factory/chair_factory.py

```
"The Factory Class"

from small_chair import SmallChair
from medium_chair import MediumChair
from big_chair import BigChair

class ChairFactory:  # pylint: disable=too-few-public-methods
    "The Factory Class"

    @staticmethod
    def get_chair(chair):
        "A static method to get a chair"
        try:
            if chair == 'BigChair':
                return BigChair()
            if chair == 'MediumChair':
                return MediumChair()
            if chair == 'SmallChair':
                return SmallChair()
            raise Exception('Chair Not Found')
        except Exception as _e:
            print(_e)
        return None
```

./abstract_factory/interface_chair.py

```
# pylint: disable=too-few-public-methods
"The Chair Interface"
from abc import ABCMeta, abstractmethod

class IChair(metaclass=ABCMeta):
    "The Chair Interface (Product)"

    @staticmethod
    @abstractmethod
    def get_dimensions():
        "A static interface method"
```

./abstract_factory/small_chair.py

```
"A Class of Chair"
from interface_chair import IChair

class SmallChair(IChair):  # pylint: disable=too-few-public-methods
    "The Small Chair Concrete Class implements the IChair interface"
    def __init__(self):
        self._height = 40
        self._width = 40
        self._depth = 40

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

./abstract_factory/medium_chair.py

```
"A Class of Chair"
from interface_chair import IChair

class MediumChair(IChair):  # pylint: disable=too-few-public-methods
    """The Medium Chair Concrete Class implements the IChair interface"""

    def __init__(self):
        self._height = 60
        self._width = 60
        self._depth = 60

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

./abstract_factory/big_chair.py

```
"A Class of Chair"
from interface_chair import IChair

class BigChair(IChair):  # pylint: disable=too-few-public-methods
    "The Big Chair Concrete Class that implements the IChair interface"

    def __init__(self):
        self._height = 80
        self._width = 80
        self._depth = 80

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

./abstract_factory/table_factory.py

```
"The Factory Class"
from small_table import SmallTable
from medium_table import MediumTable
from big_table import BigTable

class TableFactory:  # pylint: disable=too-few-public-methods
    "The Factory Class"

    @staticmethod
    def get_table(table):
        "A static method to get a table"
        try:
            if table == 'BigTable':
                return BigTable()
            if table == 'MediumTable':
                return MediumTable()
            if table == 'SmallTable':
                return SmallTable()
            raise Exception('Table Not Found')
        except Exception as _e:
            print(_e)
        return None
```

./abstract_factory/interface_table.py

```
# pylint: disable=too-few-public-methods
"The Table Interface"
from abc import ABCMeta, abstractmethod

class ITable(metaclass=ABCMeta):
    "The Table Interface (Product)"

    @staticmethod
    @abstractmethod
    def get_dimensions():
        "A static interface method"
```

./abstract_factory/small_table.py

```
"A Class of Table"
from interface_table import ITable

class SmallTable(ITable):  # pylint: disable=too-few-public-methods
    "The Small Table Concrete Class implements the ITable interface"

    def __init__(self):
        self._height = 60
        self._width = 100
        self._depth = 60

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

./abstract_factory/medium_table.py

```
"A Class of Table"
from interface_table import ITable

class MediumTable(ITable):  # pylint: disable=too-few-public-methods
    "The Medium Table Concrete Class implements the ITable interface"

    def __init__(self):
        self._height = 60
        self._width = 110
        self._depth = 70

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

./abstract_factory/big_table.py

```
"A Class of Table"
from interface_table import ITable

class BigTable(ITable):  # pylint: disable=too-few-public-methods
```

```
    "The Big Chair Concrete Class implements the ITable interface"

    def __init__(self):
        self._height = 60
        self._width = 120
        self._depth = 80

    def get_dimensions(self):
        return {
            "width": self._width,
            "depth": self._depth,
            "height": self._height
        }
```

5.2.9 Output

```
python ./abstract_factory/client.py
<class 'small_chair.SmallChair'> : {'width': 40, 'depth': 40, 'height':
40}
<class 'medium_table.MediumTable'> : {'width': 110, 'depth': 70,
'height': 60}
```

5.2.10 New Coding Concepts

Exception Handling

SBCODE Video ID #9d07c3

Your Python code may produce errors. It happens to everybody. It is hard to foresee all possible errors, but you can try to handle them in case anyway.

Use the `Try` , `Except` and optional `finally` keywords to manage error handling.

In the example code, if no chair or table is returned, an `Exception` error is raised and it includes a text string that can be read and written to the console.

Within your code you can use the `raise` keyword to trigger Python built in exceptions or even create your own.

```
def get_furniture(furniture):
    "Static get_factory method"
    try:
        if furniture in ['SmallChair', 'MediumChair', 'BigChair']:
            return ChairFactory().get_chair(furniture)
        if furniture in ['SmallTable', 'MediumTable', 'BigTable']:
            return TableFactory().get_table(furniture)
        raise Exception('No Factory Found')
    except Exception as _e:
        print(_e)
    return None
```

If `WoodenTable` is requested from the factory, it will print `No Factory Found`

You don't need to always raise an exception to make one happen. In that case you can handle the possibility of any type of error using just `try` and `except`, with the optional `finally` if you need it.

```
try:
  print(my_var)
except:
  print("An unknown error Occurred")
finally:
  print("This is optional and will get called even if there is no error")
```

The above code produces the message `An Error Occurred` because `my_var` is not defined.

The `try/except` allows the program to continue running, as can be verified by the line printed in the `finally` statement. So, this has given you the opportunity to manage any unforeseen errors any way you wish.

Alternatively, if your code didn't include the `try/except` and optional `finally` statements, the Python interpreter would return the error `NameError: name 'my_var' is not defined` and the program will crash at that line.

Also note how the default Python inbuilt error starts with `NameError`. You can handle this specific error explicitly using an extra `except` keyword.

```
try:
    print(my_var)
except NameError:
    print("There was a `NameError`")
except:
    print("An unknown error Occurred")
finally:
    print("This is optional and will get called even if there is no
error")
```

You can add exception handling for as many types of errors as you wish.

Python Errors and Exceptions : https://docs.python.org/3/tutorial/errors.html

5.2.11 Summary

- Use when you want to provide a library of relatively similar products from multiple different factories.
- You want the system to be independent of how the products are created.
- It fulfills all of the same use cases as the Factory method, but is a factory for creational pattern type methods.
- The client implements the abstract factory interface, rather than all the internal logic and Factories. This allows the possibility of creating a library that can be imported for using the Abstract Factory.
- The Abstract Factory defers the creation of the final products/objects to its concrete factory subclasses.
- You want to enforce consistent interfaces across products.
- You want the possibility to exchange product families.

5.3 Builder Design Pattern

5.3.1 Overview

SBCODE Video ID #6fa98c

The Builder Pattern is a creational pattern whose intent is to separate the construction of a complex object from its representation so that you can use the same construction process to create different representations.

The Builder Pattern tries to solve,

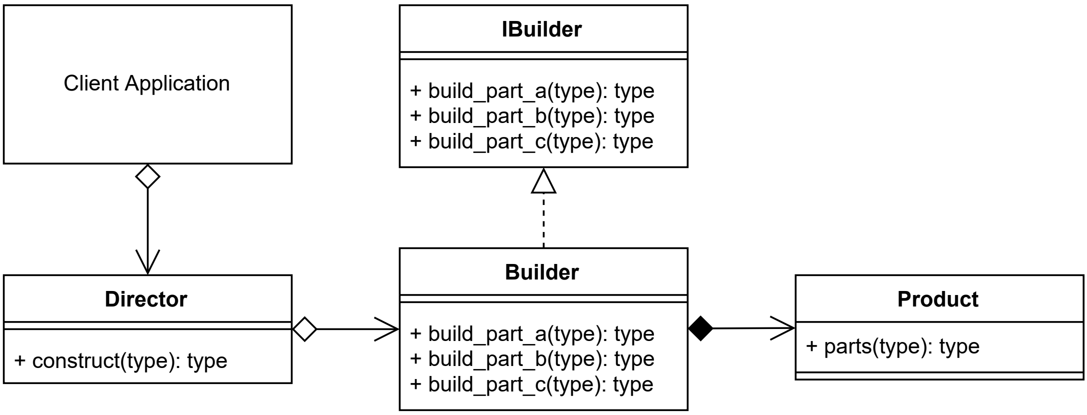
- How can a class create different representations of a complex object?
- How can a class that includes creating a complex object be simplified?

The Builder and Factory patterns are very similar in the fact they both instantiate new objects at runtime. The difference is when the process of creating the object is more complex, so rather than the Factory returning a new instance of `ObjectA` , it calls the builders director constructor method `ObjectA.construct()` that goes through a more complex construction process involving several steps. Both return an Object/Product.

5.3.2 Terminology

- **Product**: The Product being built.
- **Builder Interface**: The Interface that the Concrete builder should implement.
- **Builder**: Provides methods to build and retrieve the concrete product. Implements the **Builder Interface**.
- **Director**: Has a `construct()` method that when called creates a customized product using the methods of the **Builder**.

5.3.3 Builder UML Diagram


Client Application
IBuilder
+ build_part_a(type): type
+ build_part_b(type): type
+ build_part_c(type): type
Director
+ construct(type): type
Builder
+ build_part_a(type): type
+ build_part_b(type): type
+ build_part_c(type): type
Product
+ parts(type): type

5.3.4 Source Code

1. Client creates the **Director**.
2. The Client calls the Directors `construct()` method that manages each step of the build process.
3. The Director returns the product to the client or alternatively could also provide a method for the client to retrieve it later.

./builder/builder_concept.py

```
# pylint: disable=too-few-public-methods
"Builder Concept Sample Code"
from abc import ABCMeta, abstractmethod

class IBuilder(metaclass=ABCMeta):
    "The Builder Interface"

    @staticmethod
    @abstractmethod
    def build_part_a():
        "Build part a"

    @staticmethod
    @abstractmethod
    def build_part_b():
        "Build part b"

    @staticmethod
    @abstractmethod
    def build_part_c():
        "Build part c"

    @staticmethod
    @abstractmethod
    def get_result():
        "Return the final product"

class Builder(IBuilder):
    "The Concrete Builder."

    def __init__(self):
        self.product = Product()

    def build_part_a(self):
        self.product.parts.append('a')
        return self

    def build_part_b(self):
        self.product.parts.append('b')
        return self

    def build_part_c(self):
        self.product.parts.append('c')
        return self

    def get_result(self):
        return self.product

class Product():
    "The Product"

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

class Director:
    "The Director, building a complex representation."

    @staticmethod
    def construct():
        "Constructs and returns the final product"
        return Builder()\
            .build_part_a()\
            .build_part_b()\
            .build_part_c()\
            .get_result()

# The Client
PRODUCT = Director.construct()
print(PRODUCT.parts)
```

5.3.5 Output

```
python ./builder/builder_concept.py
['a', 'b', 'c']
```

5.3.6 Builder Use Case

SBCODE Video ID #81867f

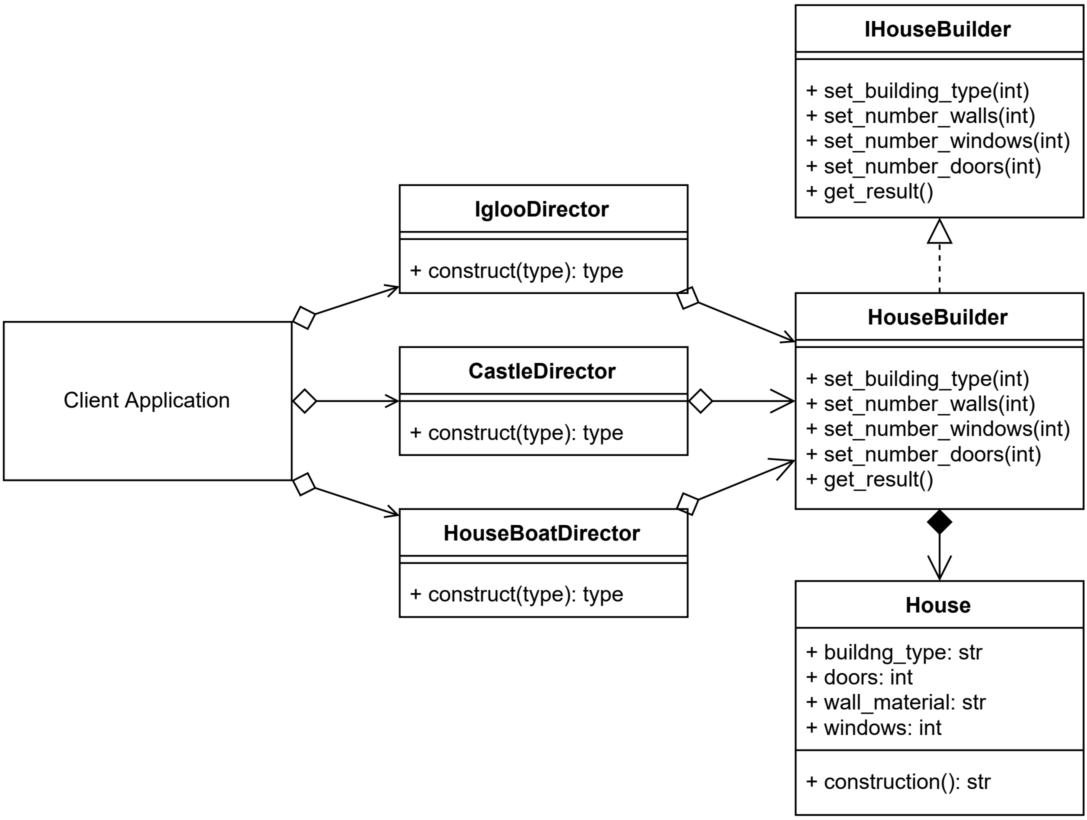
Using the Builder Pattern in the context of a House Builder.

There are multiple directors that can create their own complex objects.

Note that in the `IglooDirector` class, not all of the methods of the `HouseBuilder` were called.

The builder can construct complex objects in any order and include/exclude whichever parts it likes.

5.3.7 Example UML Diagram



5.3.8 Source Code

./builder/client.py

```
"House Builder Example Code"

from igloo_director import IglooDirector
from castle_director import CastleDirector
from houseboat_director import HouseBoatDirector

IGLOO = IglooDirector.construct()
CASTLE = CastleDirector.construct()
HOUSEBOAT = HouseBoatDirector.construct()

print(IGLOO.construction())
print(CASTLE.construction())
print(HOUSEBOAT.construction())
```

./builder/igloo_director.py

```
"A Director Class"
from house_builder import HouseBuilder

class IglooDirector:  # pylint: disable=too-few-public-methods
    "One of the Directors, that can build a complex representation."

    @staticmethod
    def construct():
        """Constructs and returns the final product
        Note that in this IglooDirector, it has omitted the set_number_of
        windows call since this Igloo will have no windows.
        """
        return HouseBuilder()\
            .set_building_type("Igloo")\
            .set_wall_material("Ice")\
            .set_number_doors(1)\
            .get_result()
```

./builder/castle_director.py

```
"A Director Class"
from house_builder import HouseBuilder

class CastleDirector:  # pylint: disable=too-few-public-methods
    "One of the Directors, that can build a complex representation."
    @staticmethod
    def construct():
        "Constructs and returns the final product"
        return HouseBuilder()\
            .set_building_type("Castle")\
            .set_wall_material("Sandstone")\
            .set_number_doors(100)\
            .set_number_windows(200)\
            .get_result()
```

./builder/houseboat_director.py

```
"A Director Class"
from house_builder import HouseBuilder

class HouseBoatDirector:  # pylint: disable=too-few-public-methods
    "One of the Directors, that can build a complex representation."

    @staticmethod
    def construct():
        "Constructs and returns the final product"
        return HouseBuilder()\
            .set_building_type("House Boat")\
            .set_wall_material("Wood")\
            .set_number_doors(6)\
            .set_number_windows(8)\
            .get_result()
```

./builder/interface_house_builder.py

```
"The Builder Interface"
from abc import ABCMeta, abstractmethod

class IHouseBuilder(metaclass=ABCMeta):
    "The House Builder Interface"

    @staticmethod
    @abstractmethod
    def set_building_type(building_type):
        "Build type"

    @staticmethod
    @abstractmethod
    def set_wall_material(wall_material):
        "Build material"
    @staticmethod
    @abstractmethod
    def set_number_doors(number):
        "Number of doors"

    @staticmethod
    @abstractmethod
    def set_number_windows(number):
        "Number of windows"

    @staticmethod
    @abstractmethod
    def get_result():
        "Return the final product"
```

./builder/house_builder.py

```
"The Builder Class"
from interface_house_builder import IHouseBuilder
from house import House

class HouseBuilder(IHouseBuilder):
    "The House Builder."

    def __init__(self):
        self.house = House()

    def set_building_type(self, building_type):
        self.house.building_type = building_type
        return self

    def set_wall_material(self, wall_material):
        self.house.wall_material = wall_material
        return self

    def set_number_doors(self, number):
        self.house.doors = number
        return self

    def set_number_windows(self, number):
        self.house.windows = number
        return self

    def get_result(self):
        return self.house
```

./builder/house.py

```
"The Product"

class House():  # pylint: disable=too-few-public-methods
    "The Product"

    def __init__(self, building_type="Apartment", doors=0,
                 windows=0, wall_material="Brick"):
        # brick, wood, straw, ice
        self.wall_material = wall_material
        # Apartment, Bungalow, Caravan, Hut, Castle, Duplex,
        # HouseBoat, Igloo
        self.building_type = building_type
        self.doors = doors
        self.windows = windows

    def construction(self):
        "Returns a string describing the construction"
        return f"This is a {self.wall_material} "\
            f"{self.building_type} with {self.doors} "\
            f"door(s) and {self.windows} window(s)."
```

5.3.9 Output

```
python ./builder/client.py
This is a Ice Igloo with 1 door(s) and 0 window(s).
This is a Sandstone Castle with 100 door(s) and 200 window(s).
This is a Wood House Boat with 6 door(s) and 8 window(s).
```

5.3.10 New Coding Concepts

Python List

SBCODE Video ID #a2766f

In the file ./builder/builder_concept.py

```
    def __init__(self):
        self.parts = []
```

The `[]` is indicating a Python **List**.

The list can store multiple items, they can be changed, they can have items added and removed, can be re-ordered, can be pre-filled with items when instantiated and is also very flexible.

```
PS> python
>>> items = []
>>> items.append("shouldn't've")
>>> items.append("y'aint")
>>> items.extend(["whomst", "superfluity"])
>>> items
["shouldn't've", "y'aint", 'whomst', 'superfluity']
>>> items.reverse()
>>> items
['superfluity', 'whomst', "y'aint", "shouldn't've"]
>>> items.remove("y'aint")
>>> items
['superfluity', 'whomst', "shouldn't've"]
>>> items.insert(1, "phoque")
>>> items
['superfluity', 'phoque', 'whomst', "shouldn't've"]
>>> items.append("whomst")
>>> items.count("whomst")
2
>>> len(items)
5
>>> items[2] = "bagnose"
>>> items
['superfluity', 'phoque', 'bagnose', "shouldn't've", 'whomst']
>>> items[-2]
"shouldn't've"
```

Lists are used in almost every code example in this book. You will see all the many ways they can be used.

In fact, a list was used in the Abstract Factory example,

```
if furniture in ['SmallChair', 'MediumChair', 'BigChair']:
    ...
```

This line, creates a list at runtime including the strings 'SmallChair', 'MediumChair' and 'BigChair'. If the value in `furniture` equals the same string as one of those items in the list, then the condition is true and the code within the if statement block will execute.

5.3.11 Summary

- The Builder pattern is a creational pattern that is used to create more complex objects than you'd expect from a factory.
- The Builder pattern should be able to construct complex objects in any order and include/ exclude whichever available components it likes.

- For different combinations of products than can be returned from a Builder, use a specific Director to create the bespoke combination.
- You can use an Abstract Factory to add an abstraction between the client and Director.

5.4 Prototype Design Pattern

5.4.1 Overview

SBCODE Video ID #7d8d9f

The **Prototype** design pattern is good for when creating new objects requires more resources than you want to use or have available. You can save resources by just creating a copy of any existing object that is already in memory.

E.g., A file you've downloaded from a server may be large, but since it is already in memory, you could just clone it, and work on the new copy independently of the original.

In the Prototype patterns interface, you create a static clone method that should be implemented by all classes that use the interface. How the clone method is implemented in the concrete class is up to you. You will need to decide whether a shallow or deep copy is required.

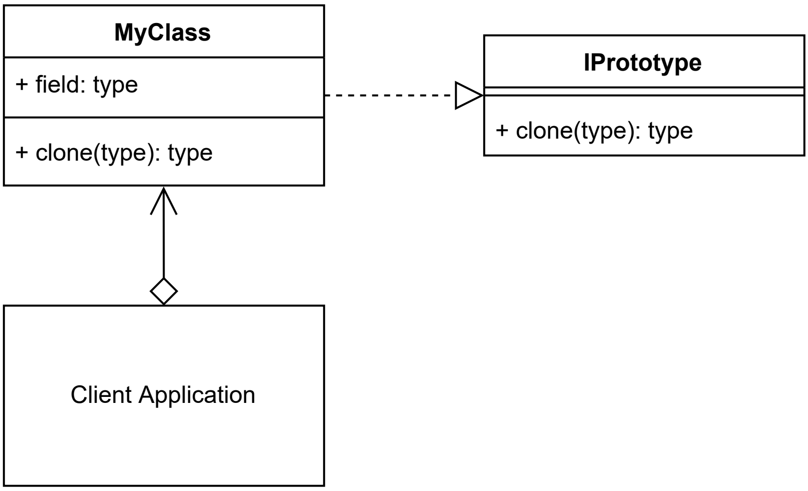
- A shallow copy, copies and creates new references one level deep,
- A deep copy, copies and creates new references for all levels.

In Python you have mutable objects such as Lists, Dictionaries, Sets and any custom Objects you may have created. A shallow copy, will create new copies of the objects with new references in memory, but the underlying data, e.g., the actual elements in a list, will point to the same memory location as the original list/object being copied. You will now have two lists, but the elements within the lists will point to the same memory location. So, changing any elements of a copied list will also affect the original list. Be sure to test your implementation that the copy method you use works as expected. Shallow copies are much faster to process than deep copies and deep copies are not always necessary if you are not going to benefit from using it.

5.4.2 Terminology

- **Prototype Interface**: The interface that describes the `clone()` method.
- **Prototype**: The Object/Product that implements the Prototype interface.
- **Client**: The client application that uses and creates the ProtoType.

5.4.3 Prototype UML Diagram



5.4.4 Source Code

Experiment with the concept code.

By default, it will shallow copy the object you've asked to be cloned. The object can be any type from number to string to dictionary to anything custom that you've created.

In my example, I have created a list of numbers. At first impressions, when this list is copied, it will appear that the list was fully cloned. But the inner items of the list were not. They will point to the same memory location as the original list; however, the memory identifier of the new list is new and different from the original.

In the `MyClass.clone()` method, there is a line `self.field.copy()` that is commented out. Uncomment out this line, and comment out the line before it to now be `# self.field` . Re execute the file, and now the list items will be copied as well. This however is still not a full deep copy. If the list items were actually other lists, dictionaries or other collections, then only the 1st level of that copy would have been cloned to new memory identifiers. I call this a 2-level copy.

For a full recursive copy, use the `copy.deepcopy()` method that is part of an extra dedicated `copy` import included with Python. I demonstrate this in the example use case further down.

Remember that full deep copies can potentially be much slower for very complicated object hierarchies.

./prototype/prototype_concept.py

```
# pylint: disable=too-few-public-methods
"Prototype Concept Sample Code"
```

```
from abc import ABCMeta, abstractmethod

class IProtoType(metaclass=ABCMeta):
    "interface with clone method"
    @staticmethod
    @abstractmethod
    def clone():
        """The clone, deep or shallow.
        It is up to you how you want to implement
        the details in your concrete class"""

class MyClass(IProtoType):
    "A Concrete Class"

    def __init__(self, field):
        self.field = field  # any value of any type

    def clone(self):
        " This clone method uses a shallow copy technique "
        return type(self)(
            self.field  # a shallow copy is returned
            # self.field.copy() # this is also a shallow copy, but has
            # also shallow copied the first level of the field. So it
            # is essentially a shallow copy but 2 levels deep. To
            # recursively deep copy collections containing inner
            # collections,
            # eg lists of lists,
            # Use https://docs.python.org/3/library/copy.html instead.
            # See example below.
        )

    def __str__(self):
        return f"{id(self)}\tfield={self.field}\ttype={type(self.field)}"

# The Client
OBJECT1 = MyClass([1, 2, 3, 4])  # Create the object containing a list
print(f"OBJECT1 {OBJECT1}")

OBJECT2 = OBJECT1.clone()  # Clone

# Change the value of one of the list elements in OBJECT2,
# to see if it also modifies the list element in OBJECT1.
# If it changed OBJECT1s copy also, then the clone was done
# using a 1 level shallow copy process.
# Modify the clone method above to try a 2 level shallow copy instead
# and compare the output
OBJECT2.field[1] = 101

# Comparing OBJECT1 and OBJECT2
```

```
print(f"OBJECT2 {OBJECT2}")
print(f"OBJECT1 {OBJECT1}")
```

5.4.5 Output

When using the shallow copy approach. Changing the inner item of OBJECT2s list, also affected OBJECT1s list.

```
python ./prototype/prototype_concept.py
OBJECT1 1808814538656   field=[1, 2, 3, 4]      type=<class 'list'>
OBJECT2 1808814538464   field=[1, 101, 3, 4]    type=<class 'list'>
OBJECT1 1808814538656   field=[1, 101, 3, 4]    type=<class 'list'>
```

When using the 2-level shallow, or deep copy approach. Changing the inner item of OBJECT2s list, does not affect OBJECT1s list. Read notes below for caveat.

```
python .\prototype\prototype_concept.py
OBJECT1 1808814538656   field=[1, 2, 3, 4]      type=<class 'list'>
OBJECT2 1808814538464   field=[1, 101, 3, 4]    type=<class 'list'>
OBJECT1 1808814538656   field=[1, 2, 3, 4]      type=<class 'list'>
```

Notes

The 2-level shallow copy was used in the above sample code. This only copies collections (list, dictionary, set) one level deep.

E.g., It won't deep copy collections containing inner collections, such as lists of lists, or dictionaries of lists, sets and tuples of any combination, etc.

For full recursive deep copying, use the library at https://docs.python.org/3/library/copy.html

5.4.6 Prototype Use Case

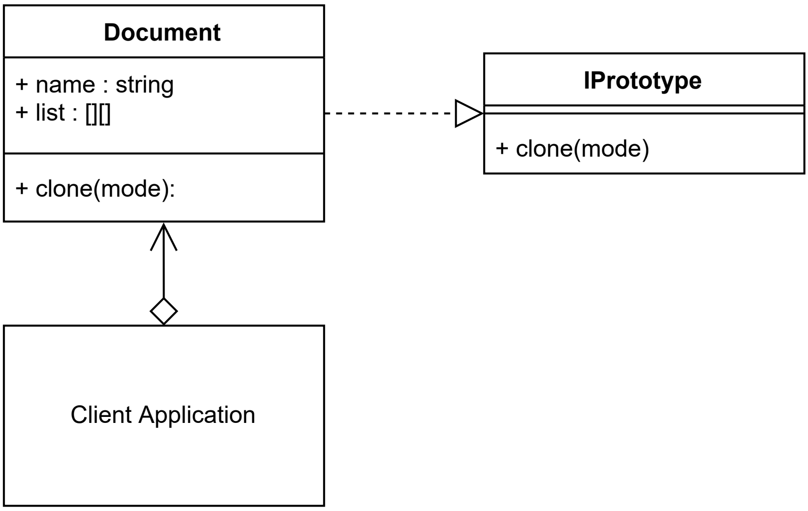
SBCODE Video ID #ca14c2

In this example, an object called document is cloned using shallow, 2 level shallow, and full recursive deep methods.

The object contains a list of two lists. Four copies are created, and each time some part of the list is changed on the clone, and depending on the method used, it can affect the original object.

When cloning an object, it is good to understand the deep versus shallow concept of copying.

5.4.7 Example UML Diagram



5.4.8 Source Code

./prototype/client.py

```
"Prototype Use Case Example Code"
from document import Document

# Creating a document containing a list of two lists
ORIGINAL_DOCUMENT = Document("Original", [[1, 2, 3, 4], [5, 6, 7, 8]])
print(ORIGINAL_DOCUMENT)
print()

DOCUMENT_COPY_1 = ORIGINAL_DOCUMENT.clone(1)  # shallow copy
DOCUMENT_COPY_1.name = "Copy 1"
# This also modified ORIGINAL_DOCUMENT because of the shallow copy
# when using mode 1
DOCUMENT_COPY_1.list[1][2] = 200
print(DOCUMENT_COPY_1)
print(ORIGINAL_DOCUMENT)
print()

DOCUMENT_COPY_2 = ORIGINAL_DOCUMENT.clone(2)  # 2 level shallow copy
DOCUMENT_COPY_2.name = "Copy 2"
# This does NOT modify ORIGINAL_DOCUMENT because it changes the
# list[1] reference that was deep copied when using mode 2
DOCUMENT_COPY_2.list[1] = [9, 10, 11, 12]
print(DOCUMENT_COPY_2)
print(ORIGINAL_DOCUMENT)
print()
```

```
DOCUMENT_COPY_3 = ORIGINAL_DOCUMENT.clone(2)  # 2 level shallow copy
DOCUMENT_COPY_3.name = "Copy 3"
# This does modify ORIGINAL_DOCUMENT because it changes the element of
# list[1][0] that was NOT deep copied recursively when using mode 2
DOCUMENT_COPY_3.list[1][0] = "1234"
print(DOCUMENT_COPY_3)
print(ORIGINAL_DOCUMENT)
print()

DOCUMENT_COPY_4 = ORIGINAL_DOCUMENT.clone(3)  # deep copy (recursive)
DOCUMENT_COPY_4.name = "Copy 4"
# This does NOT modify ORIGINAL_DOCUMENT because it
# deep copies all levels recursively when using mode 3
DOCUMENT_COPY_4.list[1][0] = "5678"
print(DOCUMENT_COPY_4)
print(ORIGINAL_DOCUMENT)
print()
```

./prototype/document.py

```
"A sample document to be used in the Prototype example"
import copy  # a python library useful for deep copying
from interface_prototype import IProtoType

class Document(IProtoType):
    "A Concrete Class"

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

    def clone(self, mode):
        " This clone method uses different copy techniques "
        if mode == 1:
            # results in a 1 level shallow copy of the Document
            doc_list = self.list
        if mode == 2:
            # results in a 2 level shallow copy of the Document
            # since it also create new references for the 1st level list
            # elements aswell
            doc_list = self.list.copy()
        if mode == 3:
            # recursive deep copy. Slower but results in a new copy
            # where no sub elements are shared by reference
            doc_list = copy.deepcopy(self.list)

        return type(self)(
            self.name,  # a shallow copy is returned of the name property
            doc_list  # copy method decided by mode argument
        )

    def __str__(self):
        " Overriding the default __str__ method for our object."
        return f"{id(self)}\tname={self.name}\tlist={self.list}"
```

./prototype/interface_prototype.py

```
# pylint: disable=too-few-public-methods
"Prototype Concept Sample Code"
from abc import ABCMeta, abstractmethod

class IProtoType(metaclass=ABCMeta):
    "interface with clone method"
    @staticmethod
    @abstractmethod
    def clone(mode):
        """The clone, deep or shallow.
        It is up to you how you  want to implement
        the details in your concrete class"""
```

5.4.9 Output

```
python ./prototype/client.py
2520526585808   name=Original   list=[[1, 2, 3, 4], [5, 6, 7, 8]]

2520526585712   name=Copy 1     list=[[1, 2, 3, 4], [5, 6, 200, 8]]
2520526585808   name=Original   list=[[1, 2, 3, 4], [5, 6, 200, 8]]

2520526585664   name=Copy 2     list=[[1, 2, 3, 4], [9, 10, 11, 12]]
2520526585808   name=Original   list=[[1, 2, 3, 4], [5, 6, 200, 8]]

2520526585520   name=Copy 3     list=[[1, 2, 3, 4], ['1234', 6, 200, 8]]
2520526585808   name=Original   list=[[1, 2, 3, 4], ['1234', 6, 200, 8]]

2520526585088   name=Copy 4     list=[[1, 2, 3, 4], ['5678', 6, 200, 8]]
2520526585808   name=Original   list=[[1, 2, 3, 4], ['1234', 6, 200, 8]]
```

5.4.10 New Coding Concepts

Python `id()` Function

SBCODE Video ID #08b4a7

The Python `id()` function returns the memory address of an object.

All objects in Python will have a memory address.

You can test if an object is unique in Python by comparing its ID.

In the examples above, I can tell how deep the copies of the dictionaries and lists were, because the IDs of the inner items will be different. I.e., they point to different memory identifiers.

Note that every time you start a Python process, the IDs assigned at runtime will likely be different.

Also note that integers in Python also have their own IDs.

```
print(id(0))
print(id(1))
print(id(2))
```

Outputs

```
2032436013328
2032436013360
2032436013392
```

5.4.11 Summary

- Just like the other creational patterns, a Prototype is used to create an object at runtime.
- A Prototype is created from an object that is already instantiated. Imagine using the existing object as the class template to create a new object, rather than calling a specific class.
- The ability to create a Prototype means that you don't need to create many classes for specific combinations of objects. You can create one object, that has a specific configuration, and then clone this version many times, rather than creating a new object from a predefined class definition.
- New Prototypes can be created at runtime, without knowing what kind of attributes the prototype may eventually have. E.g., You have a sophisticated object that was randomly created from many factors, and you want to clone it rather than re applying all the same functions over and over again until the new object matches the original.
- A prototype is also useful for when you want to create a copy of an object, but creating that copy may be very resource intensive. E.g., you can either create a new houseboat from the builder example, or clone an existing houseboat from one already in memory.
- When designing your `clone()` method, you should consider which elements will be shallow copied, how deep, and whether or not full recursive deep copy is necessary.
- For recursive deep copying, use the library at https://docs.python.org/3/library/copy.html

5.5 Singleton Design Pattern

5.5.1 Overview

SBCODE Video ID #f4a24d

Sometimes you need an object in an application where there is only one instance.

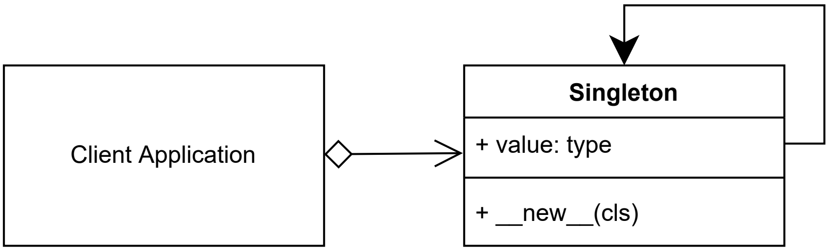
You don't want there to be many versions, for example, you have a game with a score and you want to adjust it. You may have accidentally created several instances of the class holding the score object. Or, you may be opening a database connection, there is no need to create many, when you can use the existing one that is already in memory. You may want a logging component, and you want to ensure all classes use the same instance. So, every class could declare their own logger component, but behind the scenes, they all point to the same memory address (id).

By creating a class and following the **Singleton** pattern, you can enforce that even if any number of instances were created, they will still refer to the original class.

The Singleton can be accessible globally, but it is not a global variable. It is a class that can be instanced at any time, but after it is first instanced, any new instances will point to the same instance as the first.

For a class to behave as a Singleton, it should not contain any references to `self` but use static variables, static methods and/or class methods.

5.5.2 Singleton UML Diagram



5.5.3 Source Code

In the source code, I override the classes `__new__` method to return a reference to itself. This then makes the `__init__` method irrelevant.

When running the example, experiment with commenting out the `__new__` method and you will see that the ids of the instances no longer point to the same memory location of the class, but new memory identifiers instead. The class is no longer a Singleton.

./singleton/singleton_concept.py

```python
# pylint: disable=too-few-public-methods
"Singleton Concept Sample Code"
import copy

class Singleton():
    "The Singleton Class"
    value = []

    def __new__(cls):
        return cls

    # def __init__(self):
    #     print("in init")

    @staticmethod
    def static_method():
        "Use @staticmethod if no inner variables required"

    @classmethod
    def class_method(cls):
        "Use @classmethod to access class level variables"
        print(cls.value)

# The Client
# All uses of singleton point to the same memory address (id)
print(f"id(Singleton)\t= {id(Singleton)}")

OBJECT1 = Singleton()
print(f"id(OBJECT1)\t= {id(OBJECT1)}")

OBJECT2 = copy.deepcopy(OBJECT1)
print(f"id(OBJECT2)\t= {id(OBJECT2)}")

OBJECT3 = Singleton()
print(f"id(OBJECT1)\t= {id(OBJECT3)}")
```

5.5.4 Output

```
python ./singleton/singleton_concept.py
id(Singleton)    = 2164775087968
id(OBJECT1)      = 2164775087968
id(OBJECT2)      = 2164775087968
id(OBJECT3)      = 2164775087968
```

Notes

Variables declared at class level are static variables that can be accessed directly using the class name without the class needing to be instantiated first.

cls is a reference to the class

self is a reference to the instance of the class

***new*** gets called before _***init***_,

***new*** has access to class level variables

***init*** references self that is created when the class is instantiated

By using _***new***_, and returning a reference to **cls**, we can force the class to act as a singleton. For a class to act as a singleton, it should not contain any references to **self**.

5.5.5 Singleton Use Case

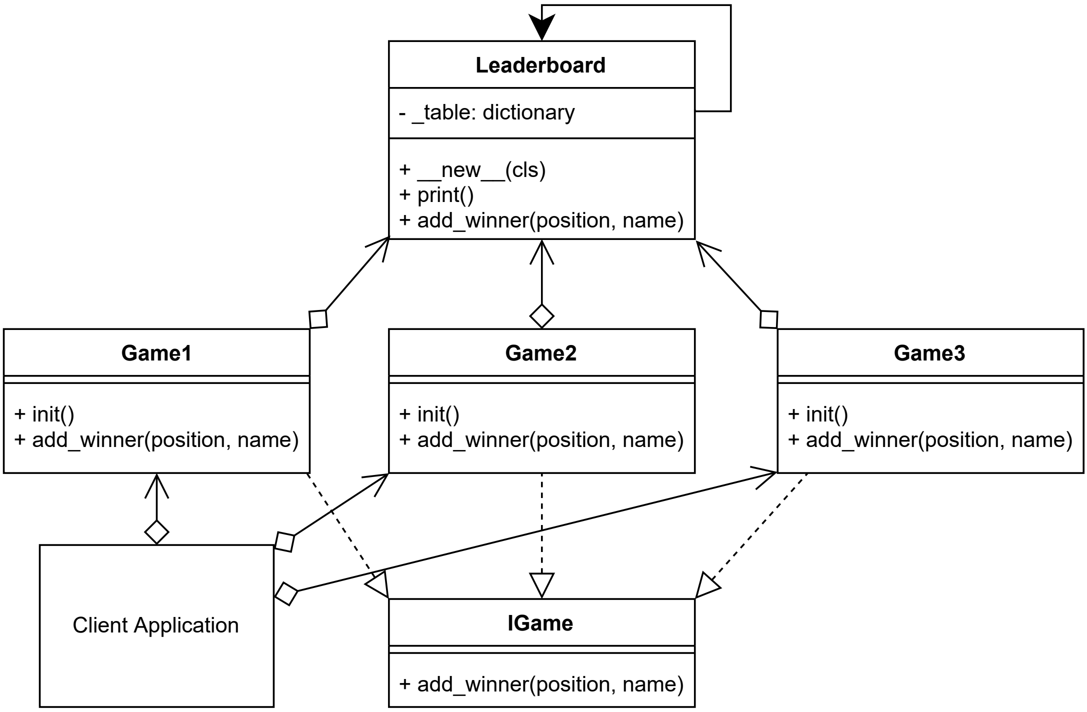
SBCODE Video ID #746648

In the example, there are three games created. They are all independent instances created from their own class, but they all share the same leaderboard. The leaderboard is a singleton.

It doesn't matter how the Games where created, or how they reference the leaderboard, it is always a singleton.

Each game independently adds a winner, and all games can read the altered leaderboard regardless of which game updated it.

5.5.6 Example UML Diagram


Leaderboard
- _table: dictionary
+ __new__(cls)
+ print()
+ add_winner(position, name)
Game1
+ init()
+ add_winner(position, name)
Game2
+ init()
+ add_winner(position, name)
Game3
+ init()
+ add_winner(position, name)
Client Application
IGame
+ add_winner(position, name)

5.5.7 Source Code

./singleton/client.py

```
# pylint: disable=too-few-public-methods

"Singleton Use Case Example Code."

from game1 import Game1
from game2 import Game2
from game3 import Game3

# The Client
# All games share and manage the same leaderboard because it is a
singleton.
GAME1 = Game1()
GAME1.add_winner(2, "Cosmo")

GAME2 = Game2()
GAME2.add_winner(3, "Sean")

GAME3 = Game3()
GAME3.add_winner(1, "Emmy")
```

```
GAME1.leaderboard.print()
GAME2.leaderboard.print()
GAME3.leaderboard.print()
```

./singleton/game1.py

```
"A Game Class that uses the Leaderboard Singleton"

from leaderboard import Leaderboard
from interface_game import IGame

class Game1(IGame):  # pylint: disable=too-few-public-methods
    "Game1 implements IGame"

    def __init__(self):
        self.leaderboard = Leaderboard()

    def add_winner(self, position, name):
        self.leaderboard.add_winner(position, name)
```

./singleton/game2.py

```
"A Game Class that uses the Leaderboard Singleton"

from leaderboard import Leaderboard
from interface_game import IGame

class Game2(IGame):  # pylint: disable=too-few-public-methods
    "Game2 implements IGame"

    def __init__(self):
        self.leaderboard = Leaderboard()

    def add_winner(self, position, name):
        self.leaderboard.add_winner(position, name)
```

./singleton/game3.py

```
"A Game Class that uses the Leaderboard Singleton"

from game2 import Game2

class Game3(Game2):  # pylint: disable=too-few-public-methods
    """Game 3 Inherits from Game 2 instead of implementing IGame"""
```

./singleton/leaderboard.py

```
"A Leaderboard Singleton Class"

class Leaderboard():
    "The Leaderboard as a Singleton"
    _table = {}

    def __new__(cls):
        return cls

    @classmethod
    def print(cls):
        "A class level method"
        print("-----------Leaderboard-----------")
        for key, value in sorted(cls._table.items()):
            print(f"|\t{key}\t|\t{value}\t|")
        print()

    @classmethod
    def add_winner(cls, position, name):
        "A class level method"
        cls._table[position] = name
```

./singleton/interface_game.py

```
# pylint: disable=too-few-public-methods
"A Game Interface"

from abc import ABCMeta, abstractmethod

class IGame(metaclass=ABCMeta):
    "A Game Interface"
    @staticmethod
    @abstractmethod
    def add_winner(position, name):
        "Must implement add_winner"
```

5.5.8 Output

```
python ./singleton/client.py
-----------Leaderboard-----------
|       1       |       Emmy    |
|       2       |       Cosmo   |
|       3       |       Sean    |

-----------Leaderboard-----------
|       1       |       Emmy    |
|       2       |       Cosmo   |
|       3       |       Sean    |

-----------Leaderboard-----------
|       1       |       Emmy    |
|       2       |       Cosmo   |
|       3       |       Sean    |
```

5.5.9 New Coding Concepts

Python Dictionary

SBCODE Video ID #5e8e70

In the file ./singleton/leaderboard.py,

```
    "The Leaderboard as a Singleton"
    _table = {}
```

The `{}` is indicating a Python **Dictionary**.

A Dictionary can be instantiated using the curly braces `{}` or `dict()`

The Dictionary is similar to a List, except that the items are `key:value` pairs.

The Dictionary can store multiple `key:value` pairs, they can be changed, can be added and removed, can be re-ordered, can be pre-filled with `key:value` pairs when instantiated and is very flexible.

Since Python 3.7, dictionaries are ordered in the same way that they are created.

The keys of the dictionary are unique.

You can refer to the dictionary items by key, which will return the value.

```
PS> python
>>> items = {"abc": 123, "def": 456, "ghi": 789}
>>> items["abc"]
123
```

You can change the value at a key,

```
PS> python
>>> items = {"abc": 123, "def": 456, "ghi": 789}
>>> items["def"] = 101112
>>> items["def"]
101112
```

You can add new `key:value` pairs, and remove them by using the key.

```
PS> python
>>> items = {"abc": 123, "def": 456, "ghi": 789}
>>> items["jkl"] = 101112
>>> items["jkl"]
101112
>>> items.pop('def')
456
>>> items
{'abc': 123, 'ghi': 789, 'jkl': 101112}
```

You can order a dictionary alphabetically by key

```
PS> python
>>> items = {"abc": 123, "ghi": 789, "def": 456}
>>> items
{'abc': 123, 'ghi': 789, 'def': 456}
>>> dict(sorted(items.items()))
{'abc': 123, 'def': 456, 'ghi': 789}
```

5.5.10 Summary

- To be a Singleton, there must only be one copy of the Singleton, no matter how many times, or in which class it was instantiated.
- You want the attributes or methods to be globally accessible across your application, so that other classes may be able to use the Singleton.
- You can use Singletons in other classes, as I did with the leaderboard, and they will all use the same Singleton regardless.
- You want controlled access to a sole instance.
- For a class to act as a singleton, it should not contain any references to `self`.

6. Structural

6.1 Decorator Design Pattern

6.1.1 Overview

SBCODE Video ID #ab01cd

The **decorator pattern** is a structural pattern, that allows you to attach additional responsibilities to an object at runtime.

The decorator pattern is used in both the Object Oriented and Functional paradigms.

The decorator pattern is different than the Python language feature of Python Decorators in its syntax and complete purpose. It is a similar concept in the way that it is a wrapper, but it also can be applied at runtime dynamically.

The decorator pattern adds extensibility without modifying the original object.

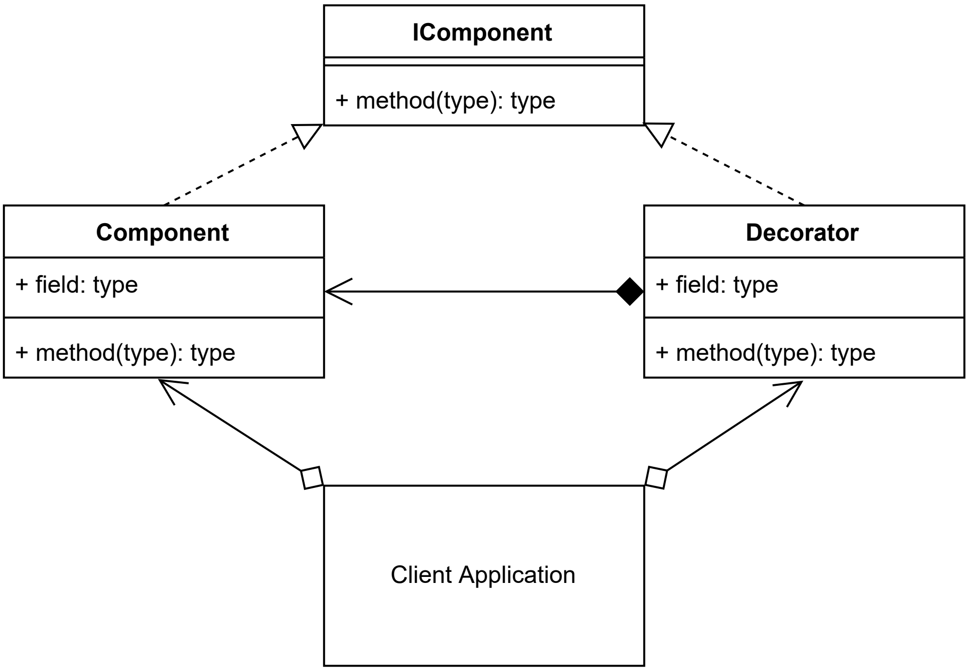
The decorator forwards requests to the enclosed object and can perform extra actions.

You can nest decorators recursively.

6.1.2 Terminology

- **Component Interface**: An interface for objects.
- **Component**: The object that may be decorated.
- **Decorator**: The class that applies the extra responsibilities to the component being decorated. It also implements the same component interface.

6.1.3 Decorator UML Diagram



6.1.4 Source Code

./decorator/decorator_concept.py

```
# pylint: disable=too-few-public-methods
"Decorator Concept Sample Code"
from abc import ABCMeta, abstractmethod

class IComponent(metaclass=ABCMeta):
    "Methods the component must implement"
    @staticmethod
    @abstractmethod
    def method():
        "A method to implement"

class Component(IComponent):
    "A component that can be decorated or not"

    def method(self):
        "An example method"
        return "Component Method"

class Decorator(IComponent):
    "The Decorator also implements the IComponent"
    def __init__(self, obj):
        "Set a reference to the decorated object"
        self.object = obj

    def method(self):
        "A method to implement"
        return f"Decorator Method({self.object.method()})"

# The Client
COMPONENT = Component()
print(COMPONENT.method())
print(Decorator(COMPONENT).method())
```

6.1.5 Output

```
python ./decorator/decorator_concept.py
Component Method
Decorator Method(Component Method)
```

6.1.6 Decorator Use Case

SBCODE Video ID #eb9f25

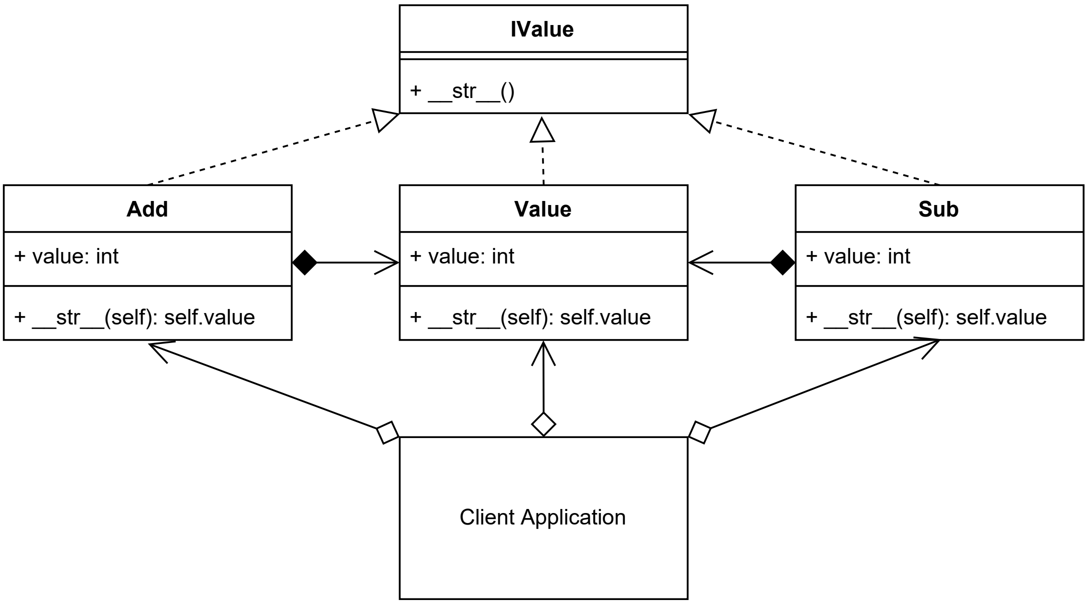
Let's create a custom class called `Value` that will hold a number.

Then add decorators that allow addition (`Add`) and subtraction (`Sub`) to a number (`Value`).

The `Add` and `Sub` decorators can accept integers directly, a custom `Value` object or other `Add` and `Sub` decorators.

`Add` , `Sub` and `Value` all implement the `IValue` interface and can be used recursively.

6.1.7 Example UML Diagram



6.1.8 Source Code

./decorator/client.py

```
"Decorator Use Case Example Code"
from value import Value
from add import Add
from sub import Sub

A = Value(1)
B = Value(2)
C = Value(5)

print(Add(A, B))
print(Add(A, 100))
print(Sub(C, A))
print(Sub(Add(C, B), A))
print(Sub(100, 101))
print(Add(Sub(Add(C, B), A), 100))
print(Sub(123, Add(C, C)))
print(Add(Sub(Add(C, 10), A), 100))
print(A)
print(B)
print(C)
```

./decorator/interface_value.py

```
# pylint: disable=too-few-public-methods
"The Interface that Value should implement"
from abc import ABCMeta, abstractmethod

class IValue(metaclass=ABCMeta):
    "Methods the component must implement"
    @staticmethod
    @abstractmethod
    def __str__():
        "Override the object to return the value attribute by default"
```

./decorator/value.py

```
# pylint: disable=too-few-public-methods
"The Custom Value class"
from interface_value import IValue

class Value(IValue):
    "A component that can be decorated or not"

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

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

./decorator/add.py

```
# pylint: disable=too-few-public-methods
"The Add Decorator"
from interface_value import IValue

class Add(IValue):
    "A Decorator that Adds a number to a number"

    def __init__(self, val1, val2):
        # val1 and val2 can be int or the custom Value
        # object that contains the `value` attribute
        val1 = getattr(val1, 'value', val1)
        val2 = getattr(val2, 'value', val2)
        self.value = val1 + val2

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

./decorator/sub.py

```
# pylint: disable=too-few-public-methods
"The Subtract Decorator"
from interface_value import IValue

class Sub(IValue):
    "A Decorator that subtracts a number from a number"

    def __init__(self, val1, val2):
        # val1 and val2 can be int or the custom Value
        # object that contains the `value` attribute
        val1 = getattr(val1, 'value', val1)
        val2 = getattr(val2, 'value', val2)
        self.value = val1 - val2

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

6.1.9 Output

```
python ./decorator/client.py
3
101
4
6
-1
106
```

```
113
114
1
2
5
```

6.1.10 New Coding Concepts

Python `getattr()` Function

SBCODE Video ID #6a4d04

Syntax: `getattr(object, attribute, default)`

In the `Sub` and `Add` classes, I use the `getattr()` method like a ternary operator.

When initializing the `Add` or `Sub` classes, you have the option of providing an integer or an existing instance of the `Value` , `Sub` or `Add` classes.

So, for example, the line in the `Sub` class,

```
val1 = getattr(val1, 'value', val1)
```

is saying, if the `val1` just passed into the function already has an attribute `value` , then `val1` must be an object of `Value` , `Sub` or `Add` . Otherwise, the `val1` that was passed in is a new integer and it will use that instead to calculate the final value of the instance on the next few lines of code. This behavior allows the `Sub` and `Add` classes to be used recursively.

E.g.,

```
A = Value(2)
Add(Sub(Add(200, 15), A), 100)
```

Dunder `__str__` method

SBCODE Video ID #496ac4

When you `print()` an object, it will print out the objects type and memory location in hex.

```
class ExampleClass:
    abc = 123

print(ExampleClass())
```

Outputs

```
<__main__.ExampleClass object at 0x00000283038B1D00>
```

You can change this default output by implementing the `__str__` dunder method in your class. Dunder is short for saying double underscore.

Dunder methods are predefined methods in python that you can override with your own implementations.

```
class ExampleClass:
    abc = 123

    def __str__(self):
        return "Something different"

print(ExampleClass())
```

Now outputs

```
Something different
```

In all the classes in the above use case example that implement the `IValue` interface, the `__str__` method is overridden to return a string version of the integer value. This allows to print the numerical value of any object that implements the `IValue` interface rather than printing a string that resembles something like below.

```
<__main__.ValueClass object at 0x00000283038B1D00>
```

The `__str__` dunder was also overridden in the Protoype concept code.

6.1.11 Summary

- Use the decorator when you want to add responsibilities to objects dynamically without affecting the inner object.
- You want the option to later remove the decorator from an object in case you no longer need it.
- It is an alternative method to creating multiple combinations of subclasses. I.e., Instead of creating a subclass with all combinations of objects A, B, C in any order, and including/excluding objects, you could create 3 objects that can decorate each other in any order you want. E.g., (D(A(C))) or (B(C)) or (A(B(A(C)))
- The decorator, compared to using static inheritance to extend, is more flexible since you can easily add/remove the decorators at runtime. E.g., use in a recursive function.

- A decorator supports recursive composition. E.g., halve(halve(number))
- A decorator shouldn't modify the internal objects data or references. This allows the original object to stay intact if the decorator is later removed.

6.2 Adapter Design Pattern

6.2.1 Overview

SBCODE Video ID #8b5434

Sometimes classes have been written and you don't have the option of modifying their interface to suit your needs. This happens if the method you are calling is on a different system across a network, a library that you may import or generally something that is not viable to modify directly for your particular needs.

The **Adapter** design pattern solves these problems:

- How can a class be reused that does not have an interface that a client requires?
- How can classes that have incompatible interfaces work together?
- How can an alternative interface be provided for a class?

You may have two classes that are similar, but they have different method signatures, so you create an Adapter over top of one of the method signatures so that it is easier to implement and extend in the client.

An adapter is similar to the Decorator in the way that it also acts like a wrapper to an object. It is also used at runtime; however, it is not designed to be used recursively.

It is an alternative interface over an existing interface. It can also provide extra functionality that the interface being adapted may not already provide.

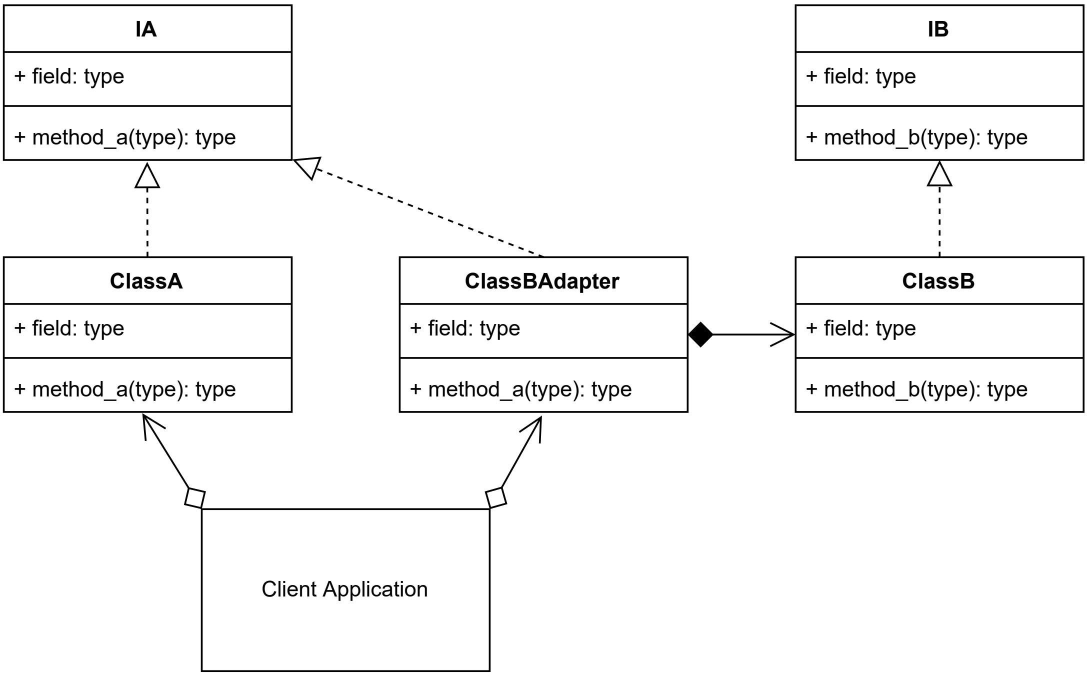
The adapter is similar to the Facade, but you are modifying the method signature, combining other methods and/or transforming data that is exchanged between the existing interface and the client.

The Adapter is used when you have an existing interface that doesn't directly map to an interface that the client requires. So, then you create the Adapter that has a similar functional role, but with a new compatible interface.

6.2.2 Terminology

- **Target**: The domain specific interface or class that needs to be adapted.
- **Adapter Interface**: The interface of the target that the adapter will need to implement.
- **Adapter**: The concrete adapter class containing the adaption process.
- **Client**: The client application that will use the Adapter.

6.2.3 Adapter UML Diagram



6.2.4 Source Code

In this concept source code, there are two classes, `ClassA` and `ClassB` , with different method signatures. Let's consider that `ClassA` provides the most compatible and preferred interface for the client.

I can create objects of both classes in the client and it works. But before using each objects method, I need to do a conditional check to see which type of class it is that I am calling since the method signatures are different.

It means that the client is doing extra work. Instead, I can create an Adapter interface for the incompatible `ClassB` , that reduces the need for the extra conditional logic.

./adapter/adapter_concept.py

```
# pylint: disable=too-few-public-methods
"Adapter Concept Sample Code"
from abc import ABCMeta, abstractmethod

class IA(metaclass=ABCMeta):
    "An interface for an object"
    @staticmethod
    @abstractmethod
    def method_a():
        "An abstract method A"

class ClassA(IA):
    "A Sample Class the implements IA"

    def method_a(self):
        print("method A")

class IB(metaclass=ABCMeta):
    "An interface for an object"
    @staticmethod
    @abstractmethod
    def method_b():
        "An abstract method B"

class ClassB(IB):
    "A Sample Class the implements IB"

    def method_b(self):
        print("method B")

class ClassBAdapter(IA):
    "ClassB does not have a method_a, so we can create an adapter"

    def __init__(self):
        self.class_b = ClassB()

    def method_a(self):
        "calls the class b method_b instead"
        self.class_b.method_b()

# The Client
# Before the adapter I need to test the objects class to know which
# method to call.
ITEMS = [ClassA(), ClassB()]
for item in ITEMS:
    if isinstance(item, ClassB):
        item.method_b()
    else:
        item.method_a()

# After creating an adapter for ClassB I can reuse the same method
# signature as ClassA (preferred)
ITEMS = [ClassA(), ClassBAdapter()]
for item in ITEMS:
    item.method_a()
```

6.2.5 Output

```
python ./adapter/adapter_concept.py
method A
method B
method A
method B
```

6.2.6 Adapter Use Case

SBCODE Video ID #ae7042

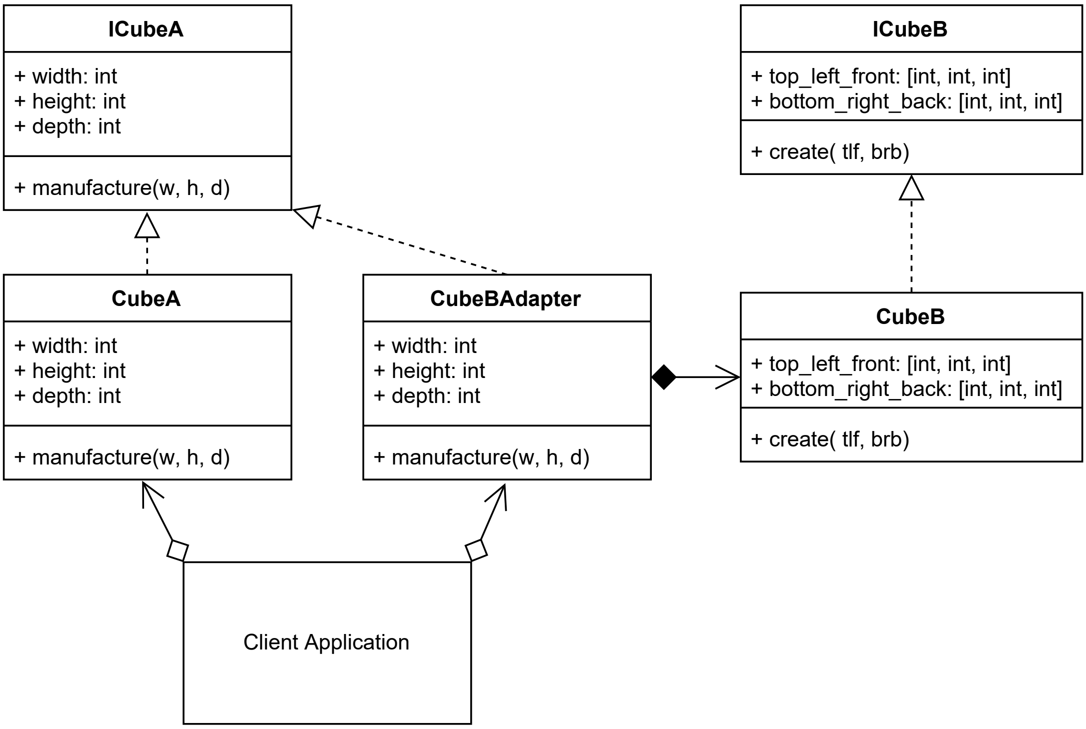
The example client can manufacture a **Cube** using different tools. Each solution is invented by a different company. The client user interface manages the Cube product by indicating the **width**, **height** and **depth**. This is compatible with the company A that produces the Cube tool, but not the company B that produces their own version of the Cube tool that uses a different interface with different parameters.

In this example, the client will re-use the interface for company A's Cube and create a compatible Cube from company B.

An adapter will be needed so that the same method signature can be used by the client without the need to ask company B to modify their Cube tool for our specific domains use case.

My imaginary company needs to use both cube suppliers since there is a large demand for cubes and when one supplier is busy, I can then ask the other supplier.

6.2.7 Example UML Diagram



6.2.8 Source Code

./adapter/client.py

```
"Adapter Example Use Case"

import time
import random
from cube_a import CubeA
from cube_b_adapter import CubeBAdapter

# client
TOTALCUBES = 5
COUNTER = 0
while COUNTER < TOTALCUBES:
    # produce 5 cubes from which ever supplier can manufacture it first
    WIDTH = random.randint(1, 10)
    HEIGHT = random.randint(1, 10)
    DEPTH = random.randint(1, 10)
    CUBE = CubeA()
    SUCCESS = CUBE.manufacture(WIDTH, HEIGHT, DEPTH)
    if SUCCESS:
        print(
            f"Company A building Cube id:{id(CUBE)}, "
            f"{CUBE.width}x{CUBE.height}x{CUBE.depth}")
        COUNTER = COUNTER + 1
    else:  # try other manufacturer
        print("Company A is busy, trying company B")
        CUBE = CubeBAdapter()
        SUCCESS = CUBE.manufacture(WIDTH, HEIGHT, DEPTH)
        if SUCCESS:
            print(
                f"Company B building Cube id:{id(CUBE)}, "
                f"{CUBE.width}x{CUBE.height}x{CUBE.depth}")
            COUNTER = COUNTER + 1
        else:
            print("Company B is busy, trying company A")
    # wait some time before manufacturing a new cube
    time.sleep(1)

print(f"{TOTALCUBES} cubes have been manufactured")
```

./adapter/cube_a.py

```
# pylint: disable=too-few-public-methods
"A Class of Cube from Company A"
import time
from interface_cube_a import ICubeA

class CubeA(ICubeA):
    "A hypothetical Cube tool from company A"
    # a static variable indicating the last time a cube was manufactured
    last_time = int(time.time())

    def __init__(self):
        self.width = self.height = self.depth = 0

    def manufacture(self, width, height, depth):
        self.width = width
        self.height = height
        self.depth = depth
        # if not busy, then manufacture a cube with dimensions
        now = int(time.time())
        if now > int(CubeA.last_time + 1):
            CubeA.last_time = now
            return True
        return False  # busy
```

./adapter/cube_b.py

```
# pylint: disable=too-few-public-methods
"A Class of Cube from Company B"
import time
from interface_cube_b import ICubeB

class CubeB(ICubeB):
    "A hypothetical Cube tool from company B"
    # a static variable indicating the last time a cube was manufactured
    last_time = int(time.time())

    def create(self, top_left_front, bottom_right_back):
        now = int(time.time())
        if now > int(CubeB.last_time + 2):
            CubeB.last_time = now
            return True
        return False  # busy
```

./adapter/cube_b_adapter.py

```
# pylint: disable=too-few-public-methods
"An adapter for CubeB so that it can be used like Cube A"
from interface_cube_a import ICubeA
from cube_b import CubeB

class CubeBAdapter(ICubeA):
    "Adapter for CubeB that implements ICubeA"

    def __init__(self):
        self.cube = CubeB()
        self.width = self.height = self.depth = 0

    def manufacture(self, width, height, depth):
        self.width = width
        self.height = height
        self.depth = depth

        success = self.cube.create(
            [0-width/2, 0-height/2, 0-depth/2],
            [0+width/2, 0+height/2, 0+depth/2]
        )
        return success
```

./adapter/interface_cube_a.py

```
# pylint: disable=too-few-public-methods
"An interface to implement"
```

```
from abc import ABCMeta, abstractmethod

class ICubeA(metaclass=ABCMeta):
    "An interface for an object"
    @staticmethod
    @abstractmethod
    def manufacture(width, height, depth):
        "manufactures a cube"
```

./adapter/interface_cube_b.py

```
# pylint: disable=too-few-public-methods
"An interface to implement"
from abc import ABCMeta, abstractmethod

class ICubeB(metaclass=ABCMeta):
    "An interface for an object"
    @staticmethod
    @abstractmethod
    def create(top_left_front, bottom_right_back):
        "Manufactures a Cube with coords offset [0, 0, 0]"
```

6.2.9 Output

```
python ./adapter/client.py
Company A is busy, trying company B
Company B is busy, trying company A
Company A is busy, trying company B
Company B is busy, trying company A
Company A building Cube id:2968196317136, 2x3x7
Company A is busy, trying company B
Company B building Cube id:2968196317136, 8x2x8
Company A building Cube id:2968196317040, 4x6x4
Company A is busy, trying company B
Company B is busy, trying company A
Company A building Cube id:2968196317136, 5x4x8
Company A is busy, trying company B
Company B building Cube id:2968196317136, 2x2x9
5 cubes have been manufactured
```

6.2.10 New Coding Concepts

Python `isinstance()` Function

SBCODE Video ID #aa3328

Syntax: `isinstance(object, type)`

Returns: `True` or `False`

You can use the inbuilt function `isinstance()` to conditionally check the `type` of an object.

```
>>> isinstance(1,int)
True
>>> isinstance(1,bool)
False
>>> isinstance(True,bool)
True
>>> isinstance("abc",str)
True
>>> isinstance("abc",(int,list,dict,tuple,set))
False
>>> isinstance("abc",(int,list,dict,tuple,set,str))
True
```

You can also test your custom classes.

```
class my_class:
    "nothing to see here"

CLASS_A = my_class()
print(type(CLASS_A))
print(isinstance(CLASS_A, bool))
print(isinstance(CLASS_A, my_class))
```

Outputs

```
<class '__main__.my_class'>
False
True
```

You can use it in logical statements as I do in adapter_concept.py above.

Python `time` Module

SBCODE Video ID #8557c1

The time module provides time related functions, most notably in my case, the current epoch (ticks) since `January 1, 1970, 00:00:00 (UTC)` .

The `time` module provides many options that are outlined in more detail at https://docs.python.org/3/library/time.html

In ./adapter/cube_a.py, I check the `time.time()` at various intervals to compare how long a task took.

```
now = int(time.time())
if now > int(CubeA.last_time + 1):
    CubeA.last_time = now
    return True
```

I also use the `time` module to sleep for a second between loops to simulate a 1 second delay. See ./adapter/client.py

```
# wait some time before manufacturing a new cube
time.sleep(1)
```

When executing ./adapter/cube_a.py you will notice that the process will run for about 10 seconds outputting the gradual progress of the construction of each cube.

6.2.11 Summary

- Use the Adapter when you want to use an existing class, but its interface does not match what you need.
- The adapter adapts to the interface of its parent class for those situations when it is not viable to modify the parent class to be domain-specific for your use case.
- Adapters will most likely provide an alternative interface over an existing object, class or interface, but it can also provide extra functionality that the object being adapted may not already provide.
- An adapter is similar to a Decorator except that it changes the interface to the object, whereas the decorator adds responsibility without changing the interface. This also allows the Decorator to be used recursively.
- An adapter is similar to the Bridge pattern and may look identical after the refactoring has been completed. However, the intent of creating the Adapter is different. The Bridge is a result of refactoring existing interfaces, whereas the Adapter is about adapting over existing interfaces that are not viable to modify due to many existing constraints. E.g., you don't have access to the original code or it may have dependencies that already use it and modifying it would affect those dependencies negatively.

6.3 Facade Design Pattern

6.3.1 Overview

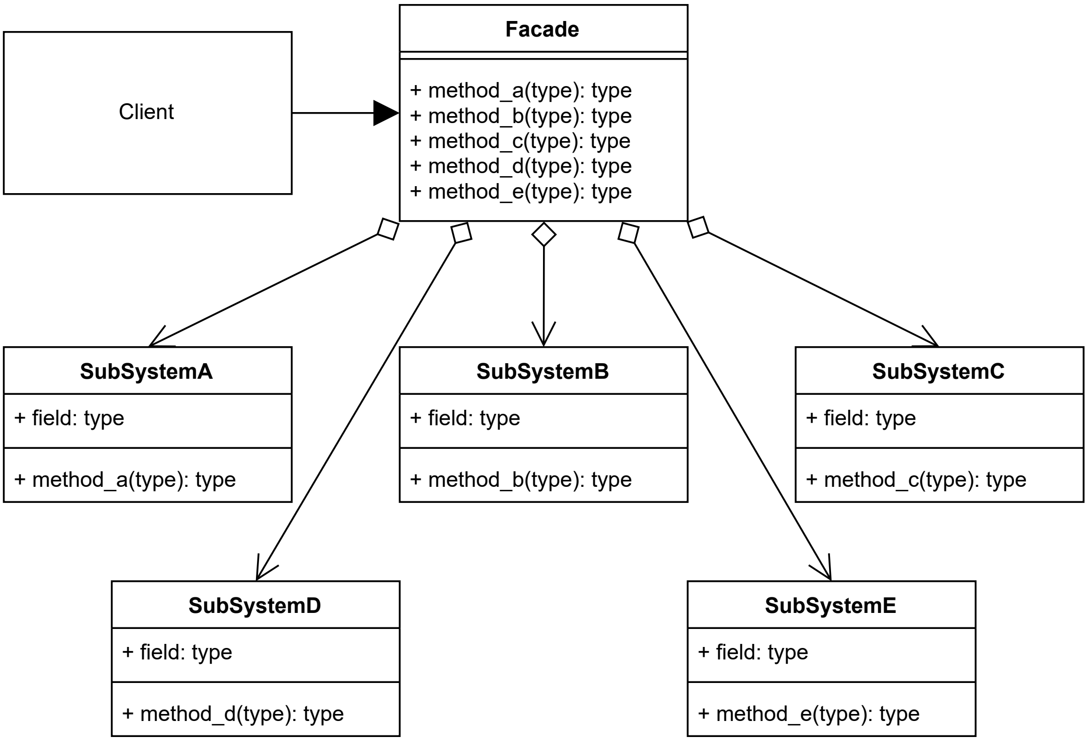
SBCODE Video ID #46770c

Sometimes you have a system that becomes quite complex over time as more features are added or modified. It may be useful to provide a simplified API over it. This is the **Facade** pattern.

The Facade pattern essentially is an alternative, reduced or simplified interface to a set of other interfaces, abstractions and implementations within a system that may be full of complexity and/or tightly coupled.

It can also be considered as a higher-level interface that shields the consumer from the unnecessary low-level complications of integrating into many subsystems.

6.3.2 Facade UML Diagram



6.3.3 Source Code

./facade/facade_concept.py

```python
# pylint: disable=too-few-public-methods
"The Facade pattern concept"

class SubSystemClassA:
    "A hypothetically complicated class"
    @staticmethod
    def method():
        "A hypothetically complicated method"
        return "A"

class SubSystemClassB:
    "A hypothetically complicated class"
    @staticmethod
    def method(value):
        "A hypothetically complicated method"
        return value

class SubSystemClassC:
    "A hypothetically complicated class"
    @staticmethod
    def method(value):
        "A hypothetically complicated method"
        return value

class Facade():
    "A simplified facade offering the services of subsystems"
    @staticmethod
    def sub_system_class_a():
        "Use the subsystems method"
        return SubSystemClassA().method()

    @staticmethod
    def sub_system_class_b(value):
        "Use the subsystems method"
        return SubSystemClassB().method(value)

    @staticmethod
    def sub_system_class_c(value):
        "Use the subsystems method"
        return SubSystemClassC().method(value)

# The Client
# call potentially complicated subsystems directly
print(SubSystemClassA.method())
print(SubSystemClassB.method("B"))
print(SubSystemClassC.method({"C": [1, 2, 3]}))

# or use the simplified facade
print(Facade().sub_system_class_a())
print(Facade().sub_system_class_b("B"))
print(Facade().sub_system_class_c({"C": [1, 2, 3]}))
```

6.3.4 Output

```
python ./facade/facade_concept.py
A
B
{'C': [1, 2, 3]}
A
B
{'C': [1, 2, 3]}
```

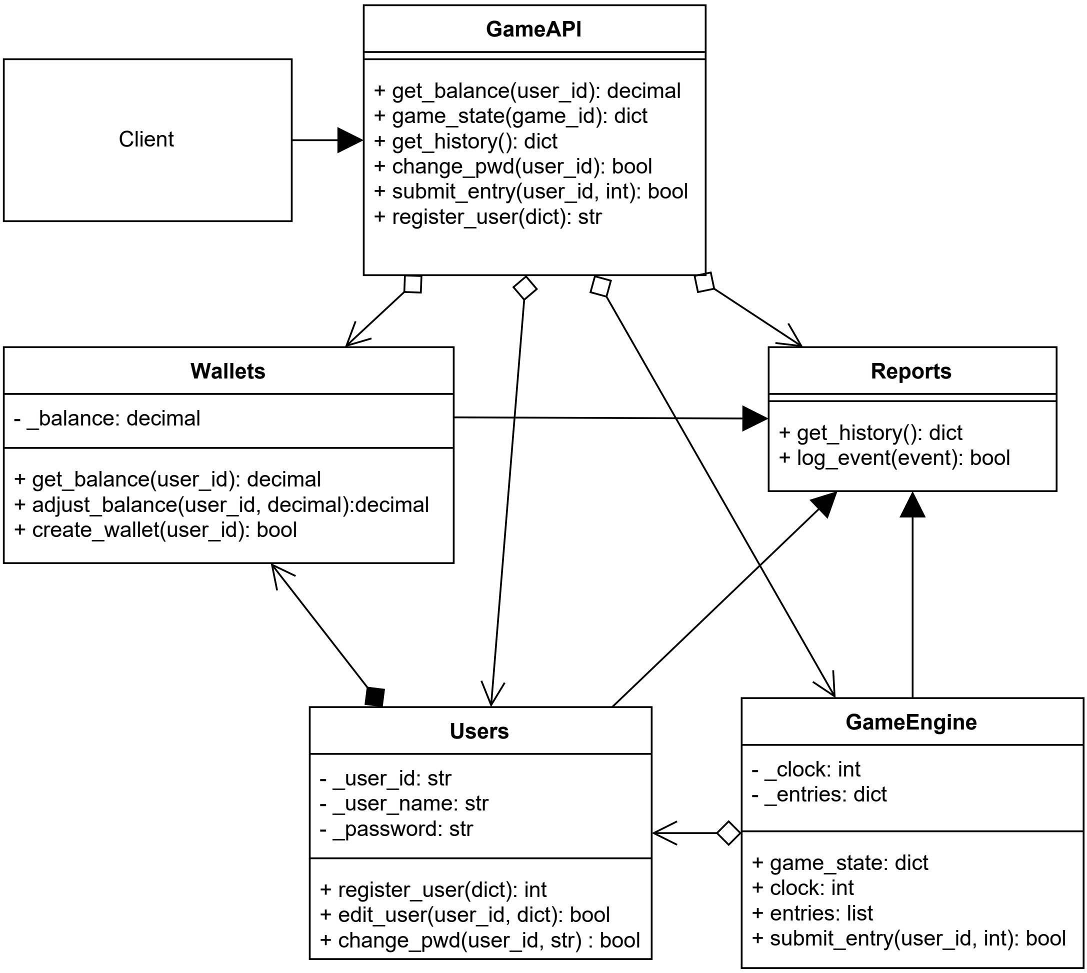
6.3.5 Facade Use Case

SBCODE Video ID #e86c30

This is an example of a game engine API. The facade layer is creating one streamlined interface consisting of several methods from several larger API backend systems.

The client could connect directly to each subsystems API and implement its authentication protocols, specific methods, etc. While it is possible, it would be quite a lot of consideration for each of the development teams, so the facade API unifies the common methods that becomes much less overwhelming for each new client developer to integrate into.

6.3.6 Example UML Diagram


GameAPI
+ get_balance(user_id): decimal
+ game_state(game_id): dict
+ get_history(): dict
+ change_pwd(user_id): bool
+ submit_entry(user_id, int): bool
+ register_user(dict): str
Client
Wallets
- _balance: decimal
+ get_balance(user_id): decimal
+ adjust_balance(user_id, decimal):decimal
+ create_wallet(user_id): bool
Reports
+ get_history(): dict
+ log_event(event): bool
Users
- _user_id: str
- _user_name: str
- _password: str
+ register_user(dict): int
+ edit_user(user_id, dict): bool
+ change_pwd(user_id, str) : bool
GameEngine
- _clock: int
- _entries: dict
+ game_state: dict
+ clock: int
+ entries: list
+ submit_entry(user_id, int): bool

6.3.7 Source Code

./facade/client.py

```
"The Facade Example Use Case"
import time
from decimal import Decimal
from game_api import GameAPI

USER = {"user_name": "sean"}
USER_ID = GameAPI.register_user(USER)

time.sleep(1)

GameAPI.submit_entry(USER_ID, Decimal('5'))
```

```
time.sleep(1)

print()
print("---- Gamestate Snapshot ----")
print(GameAPI.game_state())

time.sleep(1)

HISTORY = GameAPI.get_history()

print()
print("---- Reports History ----")
for row in HISTORY:
    print(f"{row} : {HISTORY[row][0]} : {HISTORY[row][1]}")

print()
print("---- Gamestate Snapshot ----")
print(GameAPI.game_state())
```

./facade/game_api.py

```
"The Game API facade"
from decimal import Decimal
from users import Users
from wallets import Wallets
from game_engine import GameEngine
from reports import Reports

class GameAPI():
    "The Game API facade"
    @staticmethod
    def get_balance(user_id: str) -> Decimal:
        "Get a players balance"
        return Wallets.get_balance(user_id)

    @staticmethod
    def game_state() -> dict:
        "Get the current game state"
        return GameEngine().get_game_state()

    @staticmethod
    def get_history() -> dict:
        "get the game history"
        return Reports.get_history()

    @staticmethod
    def change_pwd(user_id: str, password: str) -> bool:
        "change users password"
        return Users.change_pwd(user_id, password)

    @staticmethod
    def submit_entry(user_id: str, entry: Decimal) -> bool:
        "submit a bet"
        return GameEngine().submit_entry(user_id, entry)

    @staticmethod
    def register_user(value: dict[str, str]) -> str:
        "register a new user and returns the new id"
        return Users.register_user(value)
```

./facade/users.py

```
"A Singleton Dictionary of Users"
from decimal import Decimal
from wallets import Wallets
from reports import Reports

class Users():
    "A Singleton Dictionary of Users"
    _users: dict[str, dict[str, str]] = {}

    def __new__(cls):
        return cls

    @classmethod
    def register_user(cls, new_user: dict[str, str]) -> str:
        "register a user"
        if not new_user["user_name"] in cls._users:
            # generate really complicated unique user_id.
            # Using the existing user_name as the id for simplicity
            user_id = new_user["user_name"]
            cls._users[user_id] = new_user
            Reports.log_event(f"new user `{user_id}` created")
            # create a wallet for the new user
            Wallets().create_wallet(user_id)
            # give the user a sign up bonus
            Reports.log_event(
                f"Give new user `{user_id}` sign up bonus of 10")
            Wallets().adjust_balance(user_id, Decimal(10))
            return user_id
        return ""

    @classmethod
    def edit_user(cls, user_id: str, user: dict):
        "do nothing"
        print(user_id)
        print(user)
        return False

    @classmethod
    def change_pwd(cls, user_id: str, password: str):
        "do nothing"
        print(user_id)
        print(password)
        return False
```

./facade/wallets.py

```
"A Singleton Dictionary of User Wallets"
from decimal import Decimal
from reports import Reports

class Wallets():
    "A Singleton Dictionary of User Wallets"
    _wallets: dict[str, Decimal] = {}

    def __new__(cls):
        return cls

    @classmethod
    def create_wallet(cls, user_id: str) -> bool:
        "A method to initialize a users wallet"
        if not user_id in cls._wallets:
            cls._wallets[user_id] = Decimal('0')
            Reports.log_event(
                f"wallet for `{user_id}` created and set to 0")
            return True
        return False

    @classmethod
    def get_balance(cls, user_id: str) -> Decimal:
        "A method to check a users balance"
        Reports.log_event(
            f"Balance check for `{user_id}` = {cls._wallets[user_id]}")
        return cls._wallets[user_id]

    @classmethod
    def adjust_balance(cls, user_id: str, amount: Decimal) -> Decimal:
        "A method to adjust a user balance up or down"
        cls._wallets[user_id] = cls._wallets[user_id] + Decimal(amount)
        Reports.log_event(
            f"Balance adjustment for `{user_id}`. "
            f"New balance = {cls._wallets[user_id]}"
        )
        return cls._wallets[user_id]
```

./facade/reports.py

```
"A Singleton Dictionary of Reported Events"
import time

class Reports():
    "A Singleton Dictionary of Reported Events"
    _reports: dict[int, tuple[float, str]] = {}
    _row_id = 0

    def __new__(cls):
        return cls

    @classmethod
    def get_history(cls) -> dict:
        "A method to retrieve all historic events"
        return cls._reports

    @classmethod
    def log_event(cls, event: str) -> bool:
        "A method to add a new event to the record"
        cls._reports[cls._row_id] = (time.time(), event)
        cls._row_id = cls._row_id + 1
        return True
```

./facade/game_engine.py

```
"The Game Engine"
import time
from decimal import Decimal
from wallets import Wallets
from reports import Reports

class GameEngine():
    "The Game Engine"
    _instance = None
    _start_time: int = 0
    _clock: int = 0
    _entries: list[tuple[str, Decimal]] = []
    _game_open = True

    def __new__(cls):
        if cls._instance is None:
            cls._instance = GameEngine
            cls._start_time = int(time.time())
            cls._clock = 60
        return cls._instance

    @classmethod
    def get_game_state(cls) -> dict:
        "Get a snapshot of the current game state"
        now = int(time.time())
        time_remaining = cls._start_time - now + cls._clock
        if time_remaining < 0:
            time_remaining = 0
            cls._game_open = False
        return {
            "clock": time_remaining,
            "game_open": cls._game_open,
            "entries": cls._entries
        }

    @classmethod
    def submit_entry(cls, user_id: str, entry: Decimal) -> bool:
        "Submit a new entry for the user in this game"
        now = int(time.time())
        time_remaining = cls._start_time - now + cls._clock
        if time_remaining > 0:
            if Wallets.get_balance(user_id) > Decimal('1'):
                if Wallets.adjust_balance(user_id, Decimal('-1')):
                    cls._entries.append((user_id, entry))
                    Reports.log_event(
                        f"New entry `{entry}` submitted by `{user_id}`")
                    return True
                Reports.log_event(
                    f"Problem adjusting balance for `{user_id}`")
                return False
            Reports.log_event(f"User Balance for `{user_id}` to low")
            return False
        Reports.log_event("Game Closed")
        return False
```

6.3.8 Output

```
python ./facade/client.py

---- Gamestate Snapshot ----
{'clock': 59, 'game_open': True, 'entries': [('sean', Decimal('5'))]}

---- Reports History ----
0 : 1614087127.327007 : new user `sean` created
```

```
1 : 1614087127.327007 : wallet for `sean` created and set to 0
2 : 1614087127.327007 : Give new user `sean` sign up bonus of 10
3 : 1614087127.327007 : Balance adjustment for `sean`. New balance = 10
4 : 1614087128.3278701 : Balance check for `sean` = 10
5 : 1614087128.3278701 : Balance adjustment for `sean`. New balance = 9
6 : 1614087128.3278701 : New entry `5` submitted by `sean`

---- Gamestate Snapshot ----
{'clock': 58, 'game_open': True, 'entries': [('sean', Decimal('5'))]}
```

6.3.9 New Coding Concepts

Python `decimal` Module

SBCODE Video ID #f46fdd

The `decimal` module provides support for correctly rounded decimal floating-point arithmetic.

If representing money values in python, it is better to use the `decimal` type rather than `float` .

Floats will have rounding errors versus decimal.

```
from decimal import Decimal

print(1.1 + 2.2)  # adding floats
print(Decimal('1.1') + Decimal('2.2')) # adding decimals
```

Outputs

```
3.3000000000000003
3.3
```

Note how the float addition results in `3.3000000000000003` whereas the decimal addition result equals `3.3` .

Be aware though that when creating decimals, be sure to pass in a string representation, otherwise it will create a decimal from a float.

```
from decimal import *

print(Decimal(1.1))  # decimal from float
print(Decimal('1.1'))  # decimal from string
```

Outputs

```
1.100000000000000088817841970012523233890533447265625
1.1
```

Python Decimal: https://docs.python.org/3/library/decimal.html

Type Hints

SBCODE Video ID #bc6f00

In the Facade use case example, I have added type hints to the method signatures and class attributes.

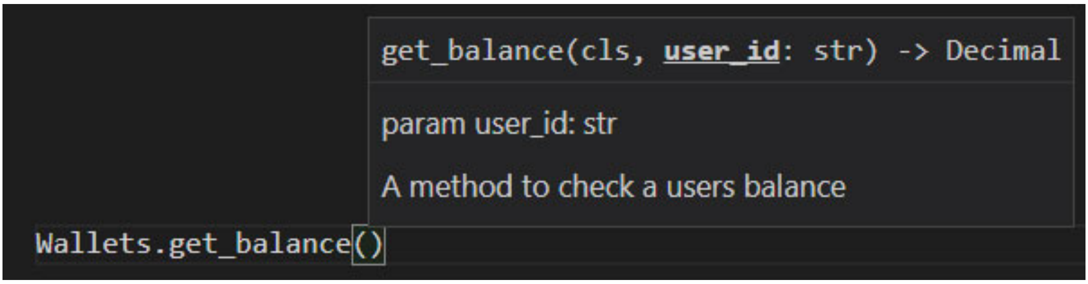
```
    _clock: int = 0
    _entries: list[tuple[str, Decimal]] = []

    ...

    def get_balance(user_id: str) -> Decimal:
        "Get a players balance"
        ...

    ...

    def register_user(cls, new_user: dict[str, str]) -> str:
        "register a user"
        ...
```

See the extra `: str` after the `user_id` attribute, and the `-> Decimal` before the final colon in the `get_balance()` snippet.

This is indicating that if you use the `get_balance()` method, that the `user_id` should be a type of `string`, and that the method will return a `Decimal`.

Note that the Python runtime does not enforce the type hints and that they are optional. However, where they are beneficial is in the IDE of your choice or other third party tools such type checkers.

In VSCode, when typing code, it will show the types that the method needs.



```
get_balance(cls, user_id: str) -> Decimal
```

param user_id: str

A method to check a users balance

```
Wallets.get_balance()
```

For type checking, you can install an extra module called `mypy`

```
pip install mypy
```

and then run it against your code,

```
mypy ./facade/client.py
Success: no issues found in 1 source file
```

Mypy will also check any imported modules at the same time.

If working with money, then it is advisable to add extra checks to your code. Checking that type usage is consistent throughout your code, especially when using Decimals, is a good idea that will make your code more robust.

For example, if I wasn't consistent in using the Decimal throughout my code, then I would see a warning highlighted.

```
mypy ./facade/client.py
facade/game_engine.py:45: error: Argument 1 to "append" of "list" has
incompatible type "Tuple[str, int]"; expected "Tuple[str, Decimal]"
facade/game_api.py:34: error: Argument 2 to "submit_entry" of
"GameEngine" has incompatible type "Decimal"; expected "int"
Found 2 errors in 2 files (checked 1 source file)
```

6.3.10 Summary

- Use when you want to provide a simple interface to a complex subsystem.
- You want to layer your subsystems into an abstraction that is easier to understand.
- Abstract Factory and Facade can be considered very similar. An Abstract Factory is about creating in interface over several creational classes of similar objects, whereas the Facade is more like an API layer over many creational, structural and/or behavioral patterns.
- The Mediator is similar to the Facade in the way that it abstracts existing classes. The Facade is not intended to modify, load balance or apply any extra logic. A subsystem does not need to consider that existence of the facade, it would still work without it.
- A Facade is a minimal interface that could also be implemented as a Singleton.
- A Facade is an optional layer that does not alter the subsystem. The subsystem does not need to know about the Facade, and could even be used by many other facades created for different audiences.

6.4 Bridge Design Pattern

6.4.1 Overview

SBCODE Video ID #83202d

The **Bridge pattern** is similar to the Adapter pattern except in the intent that you developed it.

The Bridge is an approach to refactor already existing code, whereas the Adapter creates an interface on top of existing code through existing available means without refactoring any existing code or interfaces.

The motivation for converting your code to the Bridge pattern is that it may be tightly coupled. There is logic and abstraction close together that is limiting your choices in how you can extend your solution in the way that you need.

E.g., you may have one Car class, that produces a very nice car. But you would like the option of varying the design a little, or outsourcing responsibility of creating the different components.

The Bridge pattern is a process about separating abstraction and implementation, so this will give you plenty of new ways of using your classes.

```
CAR = Car()
print(CAR)
> Car has wheels and engine and windows and everything else.
```

But you would like to delegate the engine dynamically from a separate set of classes or solutions.

```
ENGINE = EngineA()
CAR = Car(EngineA)
```

A Bridge didn't exist before, but since after the separation of interface and logic, each side can be extended independently of each other.

Also, the application of a Bridge in your code should use composition instead of inheritance. This means that you assign the relationship at runtime, rather than hard coded in the class definition.

I.e., `CAR = Car(EngineA)` rather than `class Car(EngineA):`

A Bridge implementation will generally be cleaner than an Adapter solution that was bolted on. Since it involved refactoring existing code, rather than layering on top of legacy or third-party solutions that may not have been intended for your particular use case.

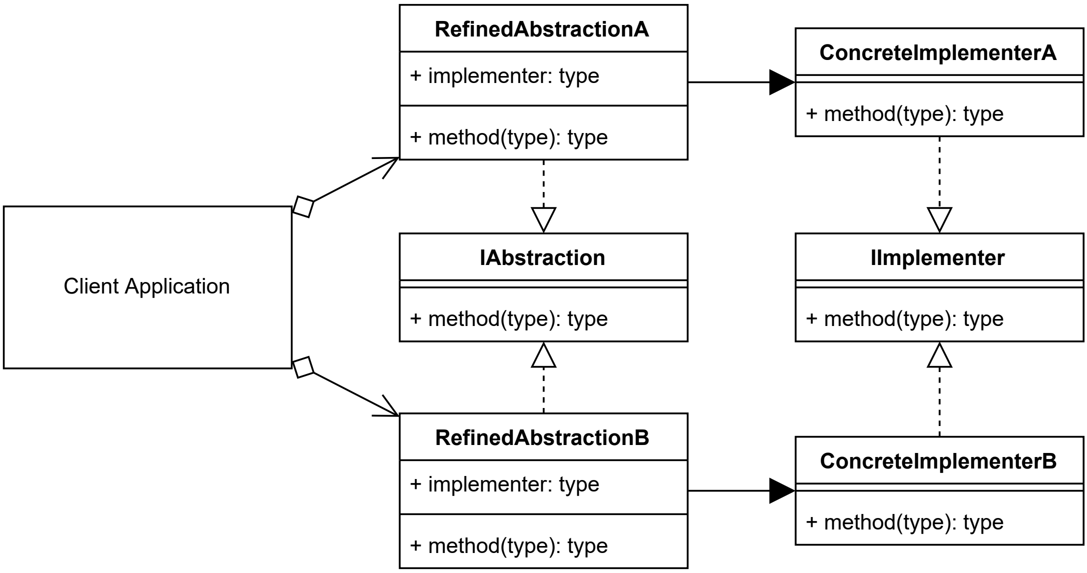
You are the designer of the Bridge, but both approaches to the problem may work regardless.

The implementer part of a Bridge, can have one or more possible implementations for each refined abstraction. E.g., The implementor can print to paper, or screen, or format for a web browser. And the abstraction side could allow for many permutations of every shape that you can imagine.

6.4.2 Terminology

- **Abstraction Interface**: An interface implemented by the refined abstraction describing the common methods to implement.
- **Refined Abstraction**: A refinement of an idea into another class or two. The classes should implement the Abstraction Interface and assign which concrete implementer.
- **Implementer Interface**: The implementer interface that concrete implementers implement.
- **Concrete Implementer**: The implementation logic that the refined abstraction will use.

6.4.3 Bridge UML Diagram



6.4.4 Source Code

In the concept demonstration code, imagine that the classes were tightly coupled. The concrete class would print out some text to the console.

After abstracting the class along a common ground, it is now more versatile. The implementation and has been separated from the abstraction and now it can print out the same text in two different ways.

The befit now is that each refined abstraction and implementer can now be worked on independently without affecting the other implementations.

./bridge/bridge_concept.py

```python
# pylint: disable=too-few-public-methods
"Bridge Pattern Concept Sample Code"
from abc import ABCMeta, abstractmethod

class IAbstraction(metaclass=ABCMeta):
    "The Abstraction Interface"

    @staticmethod
    @abstractmethod
    def method(*args):
        "The method handle"

class RefinedAbstractionA(IAbstraction):
    "A Refined Abstraction"

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

    def method(self, *args):
        self.implementer.method(*args)

class RefinedAbstractionB(IAbstraction):
    "A Refined Abstraction"

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

    def method(self, *args):
        self.implementer.method(*args)

class IImplementer(metaclass=ABCMeta):
    "The Implementer Interface"

    @staticmethod
    @abstractmethod
    def method(*args: tuple) -> None:
        "The method implementation"

class ConcreteImplementerA(IImplementer):
    "A Concrete Implementer"

    @staticmethod
    def method(*args: tuple) -> None:
        print(args)

class ConcreteImplementerB(IImplementer):
    "A Concrete Implementer"
    @staticmethod
    def method(*args: tuple) -> None:
        for arg in args:
            print(arg)

# The Client
REFINED_ABSTRACTION_A = RefinedAbstractionA(ConcreteImplementerA)
REFINED_ABSTRACTION_A.method('a', 'b', 'c')

REFINED_ABSTRACTION_B = RefinedAbstractionB(ConcreteImplementerB)
REFINED_ABSTRACTION_B.method('a', 'b', 'c')
```

6.4.5 Output

```
python ./bridge/bridge_concept.py
('a', 'b', 'c')
a
b
c
```

6.4.6 Bridge Use Case

SBCODE Video ID #96a335

In this example, I draw a square and a circle. Both of these can be categorized as shapes.

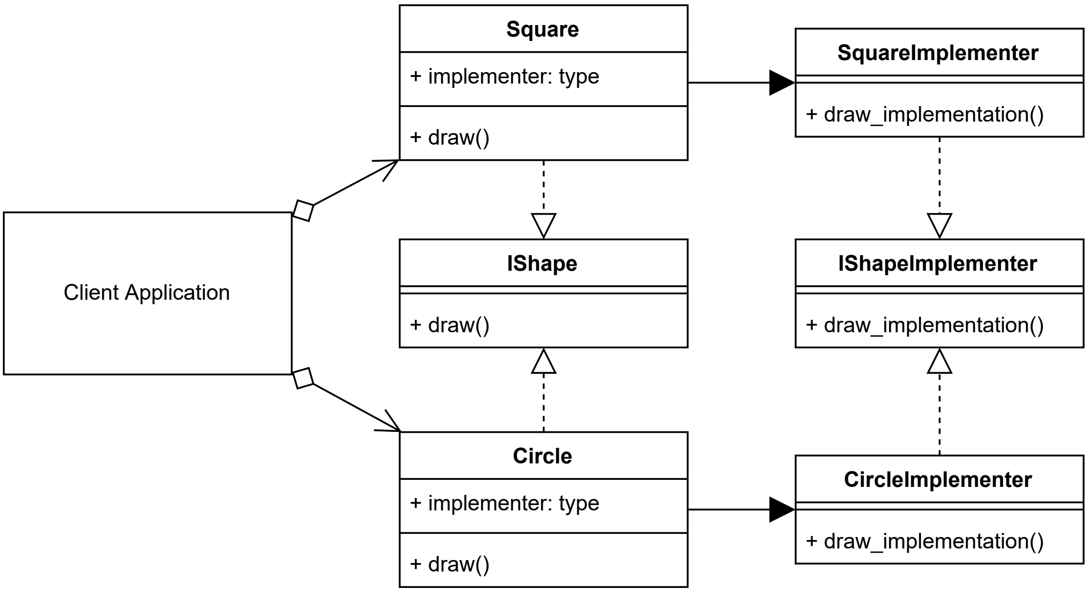
The shape is set up as the abstraction interface. The refined abstractions, `Square` and `Circle` , implement the `IShape` interface.

When the Square and Circle objects are created, they are also assigned their appropriate implementers being `SquareImplementer` and `CircleImplementer` .

When each shape's `draw` method is called, the equivalent method within their implementer is called.

The Square and Circle are bridged and each implementer and abstraction can be worked on independently.

6.4.7 Example UML Diagram



6.4.8 Source Code

./bridge/client.py

```
"Bridge Pattern Concept Sample Code"

from circle_implementer import CircleImplementer
from square_implementer import SquareImplementer
from circle import Circle
from square import Square

CIRCLE = Circle(CircleImplementer)
CIRCLE.draw()

SQUARE = Square(SquareImplementer)
SQUARE.draw()
```

./bridge/circle_implementer.py

```
# pylint: disable=too-few-public-methods
"A Circle Implementer"
from interface_shape_implementer import IShapeImplementer

class CircleImplementer(IShapeImplementer):
    "A Circle Implementer"
    def draw_implementation(self):
        print("    ******")
        print("  **      **")
        print(" *          *")
        print("*            *")
        print("*            *")
        print(" *          *")
        print("  **      **")
        print("    ******")
```

./bridge/square_implementer.py

```
# pylint: disable=too-few-public-methods
"A Square Implementer"
from interface_shape_implementer import IShapeImplementer

class SquareImplementer(IShapeImplementer):
    "A Square Implementer"

    def draw_implementation(self):
        print("**************")
        print("*            *")
        print("*            *")
        print("*            *")
        print("*            *")
        print("*            *")
        print("*            *")
        print("**************")
```

./bridge/circle.py

```
# pylint: disable=too-few-public-methods
"A Circle Abstraction"
from interface_shape import IShape

class Circle(IShape):
    "The Circle is a Refined Abstraction"

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

    def draw(self):
        self.implementer.draw_implementation()
```

./bridge/square.py

```python
# pylint: disable=too-few-public-methods
"A Square Abstraction"
from interface_shape import IShape

class Square(IShape):
    "The Square is a Refined Abstraction"

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

    def draw(self):
        self.implementer.draw_implementation()
```

./bridge/interface_shape_implementer.py

```python
# pylint: disable=too-few-public-methods
"A Shape Implementor Interface"
from abc import ABCMeta, abstractmethod

class IShapeImplementer(metaclass=ABCMeta):
    "Shape Implementer"

    @staticmethod
    @abstractmethod
    def draw_implementation():
        "The method that the refined abstractions will implement"
```

./bridge/interface_shape.py

```python
# pylint: disable=too-few-public-methods
"The Shape Abstraction Interface"
from abc import ABCMeta, abstractmethod

class IShape(metaclass=ABCMeta):
    "The Shape Abstraction Interface"

    @staticmethod
    @abstractmethod
    def draw():
        "The method that will be handled at the shapes implementer"
```

6.4.9 Output

```
python ./bridge/client.py
    ******
  **      **
 *          *
*            *
*            *
 *          *
  **      **
    ******
**************
*            *
*            *
*            *
*            *
*            *
*            *
**************
```

6.4.10 New Coding Concepts

The `*args` Argument.

SBCODE Video ID #c979fc

The `*args` argument takes all arguments that were sent to this method, and packs them into a Tuple.

It is useful when you don't know how many arguments, or what types, will be sent to a method, and you want the method to support any number of arguments or types being sent to it.

If you want your method to be strict about the types that it can accept, the set it specifically to accept List, Dictionary, Set or Tuple, and treat the argument as such within the method body, but the `*args` argument is another common option that you will see in source code throughout the internet.

E.g., when using the `*args` in your method signature, you can call it with any number of arguments of any type.

```
def my_method(*args):
    for arg in args:
        print(arg)

my_method(1, 22, [3], {4})
```

Outputs

```
1
22
[3]
{4}
```

Python Tuple

SBCODE Video ID #cf5cc0

A Python **Tuple** is similar to a List. Except that the items in the Tuple are ordered, unchangeable and allow duplicates.

A Tuple can be instantiated using the round brackets `()` or `tuple()` , verses `[]` for a List and `{}` for a Set or Dictionary.

```
PS> python
>>> items = ("alpha", "bravo", "charlie", "alpha")
>>> print(items)
('alpha', 'bravo', 'charlie', 'alpha')
>>> print(len(items))
4
```

6.4.11 Summary

- Use when you want to separate a solution where the abstraction and implementation may be tightly coupled and you want to break it up into smaller conceptual parts.
- Once you have added the bridge abstraction, you should be able to extend each side of it separately without breaking the other.
- Also, once the bridge abstraction exists, you can more easily create extra concrete implementations for other similar products that may also happen to be split across similar conceptual lines.
- The Bridge pattern is similar to the adapter pattern except in the intent that you developed it. The bridge is an approach to refactor already existing code, whereas the adapter adapts to the existing code through its existing interfaces and methods without changing the internals.

6.5 Composite Design Pattern

6.5.1 Overview

SBCODE Video ID #a8068a

The **Composite** design pattern is a structural pattern useful for hierarchal management.

The Composite design pattern,

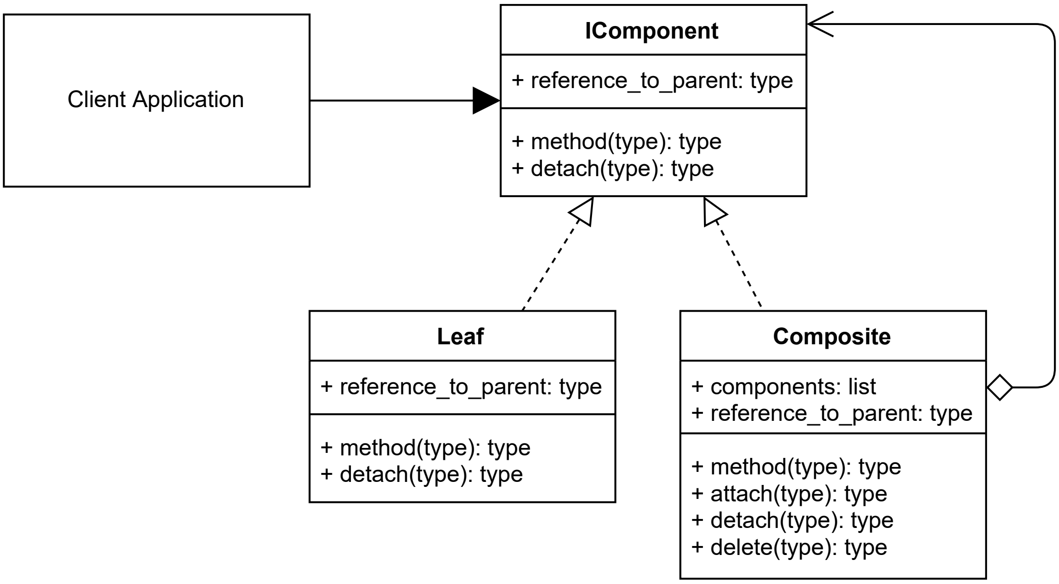
- allows you to represent individual entities(leaves) and groups of leaves at the same.
- is a structural design pattern that lets you compose objects into a changeable tree structure.
- is great if you need the option of swapping hierarchal relationships around.
- allows you to add/remove components to the hierarchy.
- provides flexibility of structure

Examples of using the Composite Design Pattern can be seen in a filesystem directory structure where you can swap the hierarchy of files and folders, and also in a drawing program where you can group, un-group, transform objects and change multiple objects at the same time.

6.5.2 Terminology

- **Component Interface**: The interface that all leaves and composites should implement.
- **Leaf**: A single object that can exist inside or outside of a composite.
- **Composite**: A collections of leaves and/or other composites.

6.5.3 Composite UML Diagram



6.5.4 Source Code

In this concept code, two leaves are created, `LEAF_A` and `LEAF_B` , and two composites are created, `COMPOSITE_1` and `COMPOSITE_2` .

`LEAF_A` is attached to `COMPOSITE_1` .

Then I change my mind and attach `LEAF_A` to `COMPOSITE_2` .

I then attach `COMPOSITE_1` to `COMPOSITE_2` .

`LEAF_B` is not attached to composites.

./composite/composite_concept.py

```
"The Composite pattern concept"
from abc import ABCMeta, abstractmethod

class IComponent(metaclass=ABCMeta):
    """
    A component interface describing the common
    fields and methods of leaves and composites
    """

    reference_to_parent = None
    @staticmethod
    @abstractmethod
    def method():
        "A method each Leaf and composite container should implement"

    @staticmethod
    @abstractmethod
    def detach():
        "Called before a leaf is attached to a composite"

class Leaf(IComponent):
    "A Leaf can be added to a composite, but not a leaf"

    def method(self):
        parent_id = (id(self.reference_to_parent)
                     if self.reference_to_parent is not None else None)
        print(
            f"<Leaf>\t\tid:{id(self)}\tParent:\t{parent_id}"
        )

    def detach(self):
        "Detaching this leaf from its parent composite"
        if self.reference_to_parent is not None:
            self.reference_to_parent.delete(self)

class Composite(IComponent):
    "A composite can contain leaves and composites"

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

    def method(self):
        parent_id = (id(self.reference_to_parent)
                     if self.reference_to_parent is not None else None)
        print(
            f"<Composite>\tid:{id(self)}\tParent:\t{parent_id}\t"
            f"Components:{len(self.components)}")

        for component in self.components:
            component.method()

    def attach(self, component):
        """
        Detach leaf/composite from any current parent reference and
        then set the parent reference to this composite (self)
        """
        component.detach()
        component.reference_to_parent = self
        self.components.append(component)
    def delete(self, component):
        "Removes leaf/composite from this composite self.components"
        self.components.remove(component)

    def detach(self):
        "Detaching this composite from its parent composite"
        if self.reference_to_parent is not None:
            self.reference_to_parent.delete(self)
            self.reference_to_parent = None

# The Client
LEAF_A = Leaf()
LEAF_B = Leaf()
COMPOSITE_1 = Composite()
COMPOSITE_2 = Composite()

print(f"LEAF_A\t\tid:{id(LEAF_A)}")
print(f"LEAF_B\t\tid:{id(LEAF_B)}")
print(f"COMPOSITE_1\tid:{id(COMPOSITE_1)}")
print(f"COMPOSITE_2\tid:{id(COMPOSITE_2)}")

# Attach LEAF_A to COMPOSITE_1
COMPOSITE_1.attach(LEAF_A)

# Instead, attach LEAF_A to COMPOSITE_2
COMPOSITE_2.attach(LEAF_A)

# Attach COMPOSITE1 to COMPOSITE_2
COMPOSITE_2.attach(COMPOSITE_1)

print()
LEAF_B.method()  # not in any composites
COMPOSITE_2.method()  # COMPOSITE_2 contains both COMPOSITE_1 and LEAF_A
```

6.5.5 Output

```
python ./composite/composite_concept.py

LEAF_A          id:2050574298848
LEAF_B          id:2050574298656
COMPOSITE_1     id:2050574298272
COMPOSITE_2     id:2050574298128

<Leaf>          id:2050574298656        Parent: None
<Composite>     id:2050574298128        Parent: None    Components:2
<Leaf>          id:2050574298848        Parent: 2050574298128
```

```
<Composite>     id:2050574298272        Parent: 2050574298128
Components:0
```

6.5.6 Composite Use Case

SBCODE Video ID #a0767c

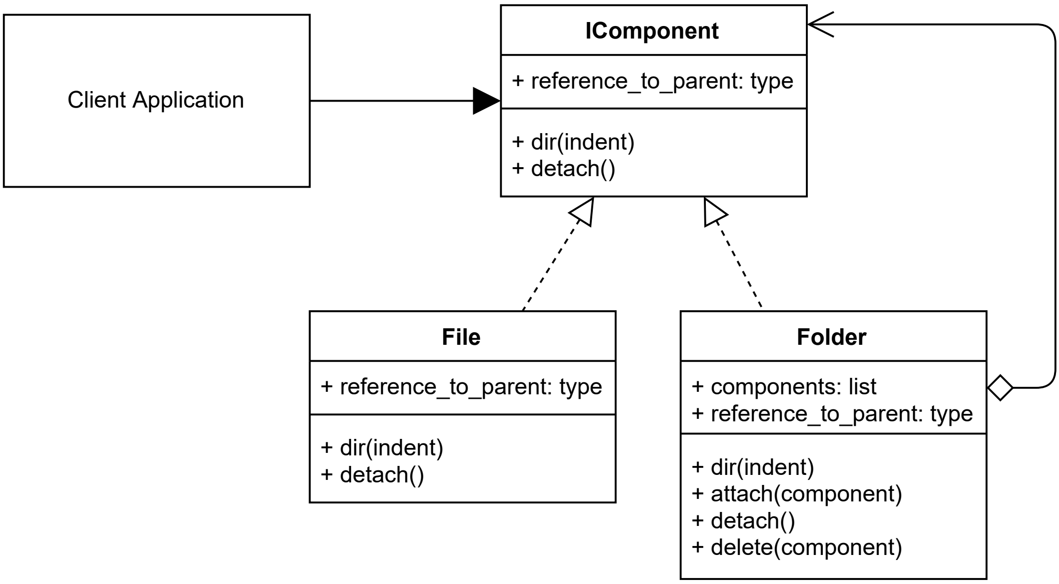
Demonstration of a simple in memory hierarchal file system.

A root object is created that is a composite.

Several files (leaves) are created and added to the root folder.

More folders (composites) are created, and more files are added, and then the hierarchy is reordered.

6.5.7 Composite Example UML Diagram



6.5.8 Source Code

./composite/client.py

```
"A use case of the composite pattern."

from folder import Folder
from file import File

FILESYSTEM = Folder("root")
```

```
FILE_1 = File("abc.txt")
FILE_2 = File("123.txt")
FILESYSTEM.attach(FILE_1)
FILESYSTEM.attach(FILE_2)
FOLDER_A = Folder("folder_a")
FILESYSTEM.attach(FOLDER_A)
FILE_3 = File("xyz.txt")
FOLDER_A.attach(FILE_3)
FOLDER_B = Folder("folder_b")
FILE_4 = File("456.txt")
FOLDER_B.attach(FILE_4)
FILESYSTEM.attach(FOLDER_B)
FILESYSTEM.dir()

# now move FOLDER_A and its contents to FOLDER_B
print()
FOLDER_B.attach(FOLDER_A)
FILESYSTEM.dir()
```

./composite/file.py

```
"A File class"
from interface_component import IComponent


class File(IComponent):
    "The File Class. The files are leaves"

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

    def dir(self, indent):
        parent_id = (id(self.reference_to_parent)
                     if self.reference_to_parent is not None else None)
        print(
            f"{indent}<FILE> {self.name}\t\t"
            f"id:{id(self)}\tParent:\t{parent_id}"
        )

    def detach(self):
        "Detaching this file (leaf) from its parent composite"
        if self.reference_to_parent is not None:
            self.reference_to_parent.delete(self)
```

./composite/folder.py

```
"A Folder, that acts as a composite."

from interface_component import IComponent


class Folder(IComponent):
    "The Folder class can contain other folders and files"

    def __init__(self, name):
        self.name = name
        self.components = []

    def dir(self, indent=""):
        print(
            f"{indent}<DIR>  {self.name}\t\tid:{id(self)}\t"
            f"Components: {len(self.components)}")
        for component in self.components:
            component.dir(indent + "..")

    def attach(self, component):
        """
        Detach file/folder from any current parent reference
        and then set the parent reference to this folder
        """
        component.detach()
        component.reference_to_parent = self
        self.components.append(component)

    def delete(self, component):
        """
        Removes file/folder from this folder so that self.components"
        is cleaned
        """
        self.components.remove(component)

    def detach(self):
        "Detaching this folder from its parent folder"
        if self.reference_to_parent is not None:
            self.reference_to_parent.delete(self)
            self.reference_to_parent = None
```

./composite/interface_component.py

```
"""
A component interface describing the common
fields and methods of leaves and composites
"""
```

```
from abc import ABCMeta, abstractmethod


class IComponent(metaclass=ABCMeta):
    "The Component Interface"

    reference_to_parent = None

    @staticmethod
    @abstractmethod
    def dir(indent):
        "A method each Leaf and composite container should implement"

    @staticmethod
    @abstractmethod
    def detach():
        """
        Called before a leaf is attached to a composite
        so that it can clean any parent references
        """
```

6.5.9 Output

```
python ./composite/client.py
<DIR>  root            id:2028913323984        Components: 4
..<FILE> abc.txt       id:2028913323888        Parent: 2028913323984
..<FILE> 123.txt       id:2028913323792        Parent: 2028913323984
..<DIR>  folder_a      id:2028913432848        Components: 1
....<FILE> xyz.txt     id:2028913433088        Parent: 2028913432848
..<DIR>  folder_b      id:2028913433184        Components: 1
....<FILE> 456.txt     id:2028913434432        Parent: 2028913433184

<DIR>  root            id:2028913323984        Components: 3
..<FILE> abc.txt       id:2028913323888        Parent: 2028913323984
..<FILE> 123.txt       id:2028913323792        Parent: 2028913323984
..<DIR>  folder_b      id:2028913433184        Components: 2
....<FILE> 456.txt     id:2028913434432        Parent: 2028913433184
....<DIR>  folder_a    id:2028913432848        Components: 1
......<FILE> xyz.txt   id:2028913433088        Parent: 2028913432848
```

6.5.10 New Coding Concepts

Conditional Expressions (Ternary Operators).

SBCODE Video ID #12b2b0

In ./composite/composite_concept.py, there are two conditional expressions.

Conditional expressions an alternate form of `if/else` statement.

```
id(self.reference_to_parent) if self.reference_to_parent is not None else
None
```

If the `self.reference_to_parent` is not `None`, it will return the memory address (id) of `self.reference_to_parent`, otherwise it returns `None`.

This conditional expression follows the format

```
value_if_true if condition else value_if_false
```

eg,

```
SUN = "bright"
SUN_IS_BRIGHT = True if SUN == "bright" else False
print(SUN_IS_BRIGHT)
```

or

```
ICE_IS_COLD = True
ICE_TEMPERATURE = "cold" if ICE_IS_COLD == True else "hot"
print(ICE_TEMPERATURE)
```

or

```
CURRENT_VALUE = 99
DANGER = 100
ALERTING = True if CURRENT_VALUE >= DANGER else False
print(ALERTING)
```

Visit https://docs.python.org/3/reference/expressions.html#conditional-expressions for more examples of conditional expressions.

6.5.11 Summary

- The Composite design pattern allows you to structure components in a manageable hierarchal order.
- It provides flexibility of structure since you can add/remove and reorder components.
- File explorer on windows is a very good example of the composite design pattern in use.
- Any system where you need to offer at runtime the ability to group, un-group, modify multiple objects at the same time, would benefit from the composite design pattern structure. Programs that allow you to draw shapes and graphics will often also use this structure as well.

6.6 Flyweight Design Pattern

6.6.1 Overview

SBCODE Video ID #98a1c6

Fly in the term **Flyweight** means light/not heavy.

Instead of creating thousands of objects that share common attributes, and result in a situation where a large amount of memory or other resources are used, you can modify your classes to share multiple instances simultaneously by using some kind of reference to the shared object instead.

The best example to describe this is a document containing many words and sentences and made up of many letters. Rather than storing a new object for each individual letter describing its font, position, color, padding and many other potential things. You can store just a lookup id of a character in a collection of some sort and then dynamically create the object with its proper formatting etc., only as you need to.

This approach saves a lot of memory at the expense of using some extra CPU instead to create the object at presentation time.

The Flyweight pattern, describes how you can share objects rather than creating thousands of almost repeated objects unnecessarily.

A Flyweight acts as an independent object in any number of contexts. A context can be a cell in a table, or a div on a html page. A context is using the Flyweight.

You can have many contexts, and when they ask for a Flyweight, they will get an object that may already be shared amongst other contexts, or already within it self somewhere else.

When describing flyweights, it is useful to describe it in terms of intrinsic and extrinsic attributes.

Intrinsic (in or including) are the attributes of a flyweight that are internal and unique from the other flyweights. E.g., a new flyweight for every letter of the alphabet. Each letter is intrinsic to the flyweight.

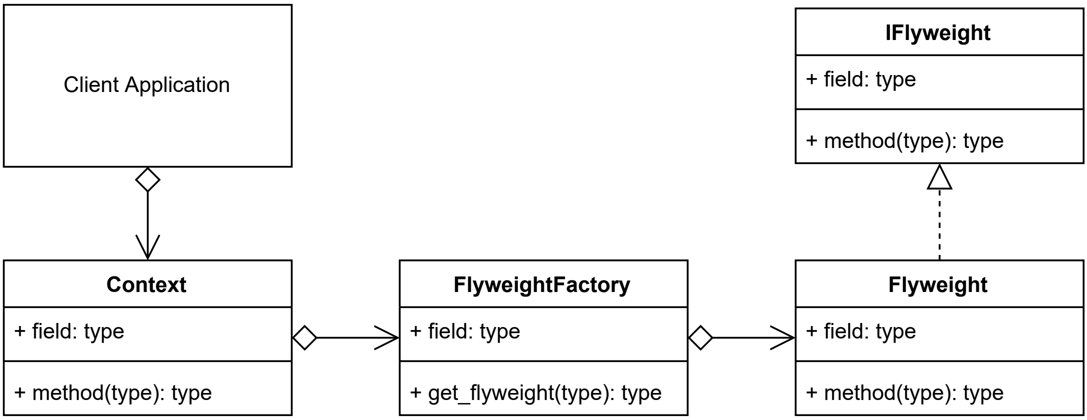
Extrinsic (outside or external) are the attributes that are used to present the flyweight in terms of the context where it will be used. E.g., many letters in a string can be right aligned with each other. The extrinsic property of each letter is the new positioning of its X and Y on a grid.

6.6.2 Terminology

- **Flyweight Interface**: An interface where a flyweight receives its extrinsic attributes.
- **Concrete Flyweight**: The flyweight object that stores the intrinsic attributes and implements the interface to apply extrinsic attributes.

- **Unshared Flyweights**: Not all flyweights will be shared, the flyweight enables sharing, not enforcing it. It also possible that flyweights can share other flyweights but still not yet be used in any contexts anywhere.
- **Flyweight Factory**: Creates and manages flyweights at runtime. It reuses flyweights in memory, or creates a new one in demand.
- **Client**: The client application that uses and creates the Flyweight.

6.6.3 Flyweight UML Diagram



6.6.4 Source Code

A context is created using the string `abracadabra` .

As it is output, it asks the Flyweight factory for the next character. The Flyweight factory will either return an existing Flyweight, or create a new one before returning it.

`abracadabra` has many re-used characters, so only 5 flyweights needed to be created.

./flyweight/flyweight_concept.py

```
# pylint: disable=too-few-public-methods
"The Flyweight Concept"

class IFlyweight():
    "Nothing to implement"

class Flyweight(IFlyweight):
    "The Concrete Flyweight"

    def __init__(self, code: int) -> None:
        self.code = code
```

```
class FlyweightFactory():
    "Creating the FlyweightFactory as a singleton"

    _flyweights: dict[int, Flyweight]= {}

    def __new__(cls):
        return cls

    @classmethod
    def get_flyweight(cls, code: int) -> Flyweight:
        "A static method to get a flyweight based on a code"
        if not code in cls._flyweights:
            cls._flyweights[code] = Flyweight(code)
        return cls._flyweights[code]

    @classmethod
    def get_count(cls) -> int:
        "Return the number of flyweights in the cache"
        return len(cls._flyweights)

class Context():
    """
    An example context that holds references to the flyweights in a
    particular order and converts the code to an ascii letter
    """

    def __init__(self, codes: str) -> None:
        self.codes = list(codes)

    def output(self):
        "The context specific output that uses flyweights"
        ret = ""
        for code in self.codes:
            ret = ret + FlyweightFactory.get_flyweight(code).code
        return ret

# The Client
CONTEXT = Context("abracadabra")

# use flyweights in a context
print(CONTEXT.output())

print(f"abracadabra has {len('abracadabra')} letters")
print(f"FlyweightFactory has {FlyweightFactory.get_count()} flyweights")
```

6.6.5 Output

```
python ./flyweight/flyweight_concept.py
abracadabra
abracadabra has 11 letters
FlyweightFactory has 5 flyweights
```

6.6.6 Flyweight Use Case

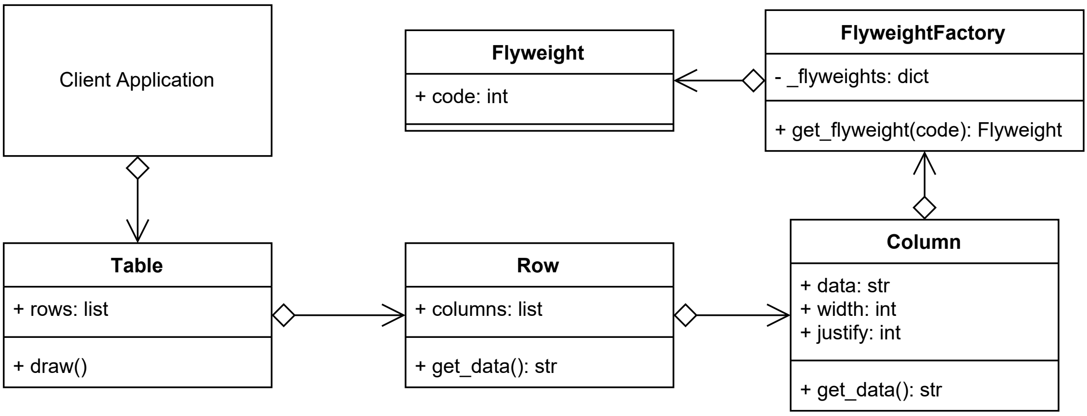
SBCODE Video ID #d9ffbd

In this example, I create a dynamic table with 3 rows and 3 columns each. The columns are then filled with some kind of text, and also chosen to be left, right or center aligned.

The letters are the flyweights and only a code indicating the letter is stored. The letters and numbers are shared many times.

The columns are the contexts and they pass the extrinsic vales describing the combination of letters, the justification left, right or center, and the width of the table column that is then used for the space padding.

6.6.7 Example UML Diagram



6.6.8 Source Code

./flyweight/client.py

```
"The Flyweight Use Case Example"

from table import Table
from flyweight_factory import FlyweightFactory

TABLE = Table(3, 3)
```

```
TABLE.rows[0].columns[0].data = "abra"
TABLE.rows[0].columns[1].data = "112233"
TABLE.rows[0].columns[2].data = "cadabra"
TABLE.rows[1].columns[0].data = "racadab"
TABLE.rows[1].columns[1].data = "12345"
TABLE.rows[1].columns[2].data = "332211"
TABLE.rows[2].columns[0].data = "cadabra"
TABLE.rows[2].columns[1].data = "445566"
TABLE.rows[2].columns[2].data = "aa 22 bb"

TABLE.rows[0].columns[0].justify = 1
TABLE.rows[1].columns[0].justify = 1
TABLE.rows[2].columns[0].justify = 1
TABLE.rows[0].columns[2].justify = 2
TABLE.rows[1].columns[2].justify = 2
TABLE.rows[2].columns[2].justify = 2
TABLE.rows[0].columns[1].width = 15
TABLE.rows[1].columns[1].width = 15
TABLE.rows[2].columns[1].width = 15

TABLE.draw()

print(f"FlyweightFactory has {FlyweightFactory.get_count()} flyweights")
```

./flyweight/flyweight.py

```
"The Flyweight that contains an intrinsic value called code"

class Flyweight():  # pylint: disable=too-few-public-methods
    "The Flyweight that contains an intrinsic value called code"

    def __init__(self, code: int) -> None:
        self.code = code
```

./flyweight/flyweight_factory.py

```
"Creating the FlyweightFactory as a singleton"
from flyweight import Flyweight

class FlyweightFactory():
    "Creating the FlyweightFactory as a singleton"

    _flyweights: dict[int, Flyweight] = {}

    def __new__(cls):
        return cls
    @classmethod
    def get_flyweight(cls, code: int) -> Flyweight:
        "A static method to get a flyweight based on a code"
        if not code in cls._flyweights:
            cls._flyweights[code] = Flyweight(code)
        return cls._flyweights[code]

    @classmethod
    def get_count(cls) -> int:
        "Return the number of flyweights in the cache"
        return len(cls._flyweights)
```

./flyweight/column.py

```
"A Column that is used in a Row"

from flyweight_factory import FlyweightFactory

class Column():  # pylint: disable=too-few-public-methods
    """
    The columns are the contexts.
    They will share the Flyweights via the FlyweightsFactory.
    `data`, `width` and `justify` are extrinsic values. They are outside
    of the flyweights.
    """

    def __init__(self, data="", width=11, justify=0) -> None:
        self.data = data
        self.width = width
        self.justify = justify  # 0:center, 1:left, 2:right

    def get_data(self):
        "Get the flyweight value from the factory, and apply the
extrinsic values"
        ret = ""
        for data in self.data:
            ret = ret + FlyweightFactory.get_flyweight(data).code
        ret = f"{ret.center(self.width)}" if self.justify == 0 else ret
        ret = f"{ret.ljust(self.width)}" if self.justify == 1 else ret
        ret = f"{ret.rjust(self.width)}" if self.justify == 2 else ret
        return ret
```

./flyweight/row.py

```
"A Row in the Table"
from column import Column

class Row():  # pylint: disable=too-few-public-methods
    "A Row in the Table"

    def __init__(self, column_count: int) -> None:
        self.columns = []
        for _ in range(column_count):
            self.columns.append(Column())

    def get_data(self):
        "Format the row before returning it to the table"
        ret = ""
        for column in self.columns:
            ret = f"{ret}{column.get_data()}|"
        return ret
```

./flyweight/table.py

```
"A Formatted Table that includes rows and columns"

from row import Row

class Table():  # pylint: disable=too-few-public-methods
    "A Formatted Table"

    def __init__(self, row_count: int, column_count: int) -> None:
        self.rows = []
        for _ in range(row_count):
            self.rows.append(Row(column_count))

    def draw(self):
        "Draws the table formatted in the console"
        max_row_length = 0
        rows = []
        for row in self.rows:
            row_data = row.get_data()
            rows.append(f"|{row_data}")
            row_length = len(row_data) + 1
            if max_row_length < row_length:
                max_row_length = row_length
        print("-" * max_row_length)
        for row in rows:
            print(row)
        print("-" * max_row_length)
```

6.6.9 Output

```
python ./flyweight/client.py
-----------------------------------------
|abra       |     112233    |    cadabra|
|racadab    |     12345     |     332211|
|cadabra    |     445566    |   aa 22 bb|
-----------------------------------------
FlyweightFactory has 12 flyweights
```

6.6.10 New Coding Concepts

String Justification

SBCODE Video ID #dd45e8

In ./flyweight/column.py, there are commands `center()` , `ljust()` and `rjust()` .

These are special commands on strings that allow you to pad strings and align them left, right, center depending on total string length.

eg,

```
>>> "abcd".center(10)
'   abcd   '
```

```
>>> "abcd".rjust(10)
'      abcd'
```

```
>>> "abcd".ljust(10)
'abcd      '
```

6.6.11 Summary

- Clients should access Flyweight objects only the through a `FlyweightFactory` object to ensure that they are shared.
- Intrinsic values are stored internally in the Flyweight.
- Extrinsic values are passed to the Flyweight and customise it depending on the context.
- Implementing the flyweight is a balance between storing all objects in memory, versus storing small unique parts in memory, and potentially calculating extrinsic values in the context objects.

- Use the flyweight to save memory when it is beneficial. The offset is that extra CPU may be required during calculating and passing extrinsic values to the flyweights.
- The flyweight reduces memory footprint because it shares objects and allows the possibility of dynamically creating extrinsic attributes.
- The contexts will generally calculate the extrinsic values used by the flyweights, but it is not necessary. Values can be stored or referenced from other objects if necessary.
- When architecting the flyweight, start with considering which parts of a common object may be able to be split and applied using extrinsic attributes.

6.7 Proxy Design Pattern

6.7.1 Overview

SBCODE Video ID #c0f2d0

The **Proxy** design pattern is a class functioning as an interface to another class or object.

A Proxy could be for anything, such as a network connection, an object in memory, a file, or anything else you need to provide an abstraction between.

Types of proxies,

- **Virtual Proxy**: An object that can cache parts of the real object, and then complete loading the full object when necessary.
- **Remote Proxy**: Can relay messages to a real object that exists in a different address space.
- **Protection Proxy**: Apply an authentication layer in front of the real object.
- **Smart Reference**: An object whose internal attributes can be overridden or replaced.

Additional functionality can be provided at the proxy abstraction if required. E.g., caching, authorization, validation, lazy initialization, logging.

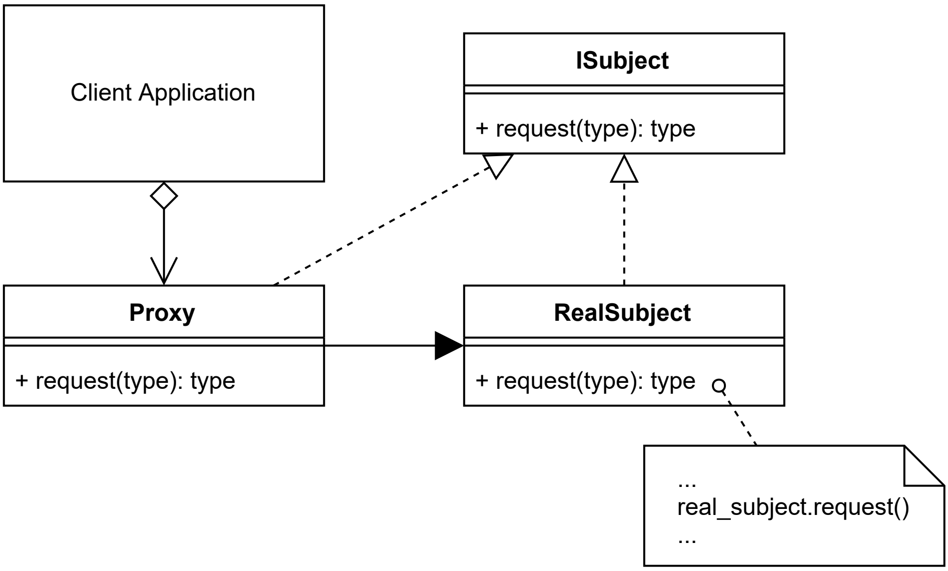
The proxy should implement the subject interface as much as practicable so that the proxy and subject appear identical to the client.

The Proxy Pattern can also be called **Monkey Patching** or **Object Augmentation**

6.7.2 Terminology

- **Proxy**: An object with an interface identical to the real subject. Can act as a placeholder until the real subject is loaded or as gatekeeper applying extra functionality.
- **Subject Interface**: An interface implemented by both the Proxy and Real Subject.
- **Real Subject**: The actual real object that the proxy is representing.
- **Client**: The client application that uses and creates the Proxy.

6.7.3 Proxy UML Diagram



6.7.4 Source Code

./proxy/proxy_concept.py

```
# pylint: disable=too-few-public-methods
"A Proxy Concept Example"

from abc import ABCMeta, abstractmethod

class ISubject(metaclass=ABCMeta):
    "An interface implemented by both the Proxy and Real Subject"
    @staticmethod
    @abstractmethod
    def request():
        "A method to implement"

class RealSubject(ISubject):
    "The actual real object that the proxy is representing"

    def __init__(self):
        # hypothetically enormous amounts of data
        self.enormous_data = [1, 2, 3]

    def request(self):
        return self.enormous_data

class Proxy(ISubject):
    """
    The proxy. In this case the proxy will act as a cache for
    `enormous_data` and only populate the enormous_data when it
    is actually necessary
    """

    def __init__(self):
        self.enormous_data = []
        self.real_subject = RealSubject()

    def request(self):
        """
        Using the proxy as a cache, and loading data into it only if
        it is needed
        """
        if self.enormous_data == []:
            print("pulling data from RealSubject")
            self.enormous_data = self.real_subject.request()
            return self.enormous_data
        print("pulling data from Proxy cache")
        return self.enormous_data

# The Client
SUBJECT = Proxy()
# use SUBJECT
print(id(SUBJECT))
# load the enormous amounts of data because now we want to show it.
print(SUBJECT.request())
# show the data again, but this time it retrieves it from the local cache
print(SUBJECT.request())
```

6.7.5 Output

```
python ./proxy/proxy_concept.py
1848118706080
pulling data from RealSubject
[1, 2, 3]
pulling data from Proxy cache
[1, 2, 3]
```

6.7.6 Proxy Use Case

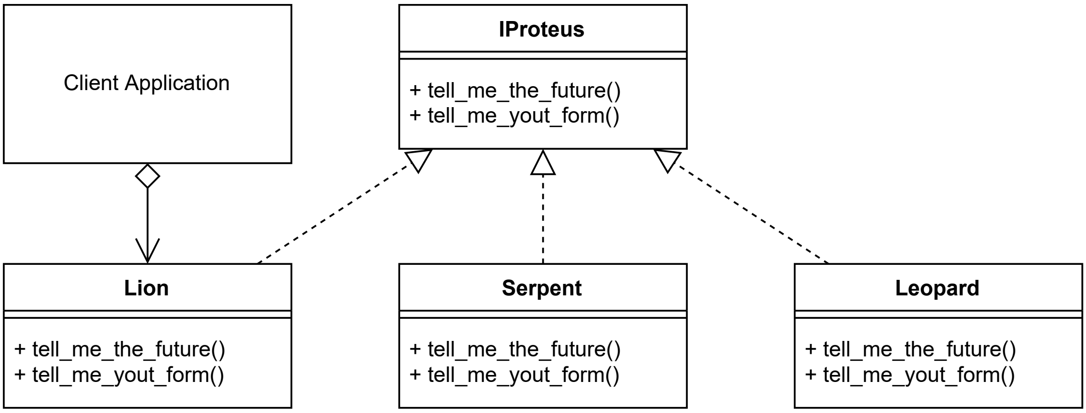
SBCODE Video ID #883f9a

In this example, I dynamically change the class of an object. So, I am essentially using an object as a proxy to other classes.

Every time the `tell_me_the_future()` method is called; it will randomly change the object to use a different class.

The object `PROTEUS` will then use the same static attributes and class methods of the new class instead.

6.7.7 Example UML Diagram



6.7.8 Source Code

./proxy/client.py

```
"The Proxy Example Use Case"

from lion import Lion

PROTEUS = Lion()
PROTEUS.tell_me_your_form()
PROTEUS.tell_me_the_future()
PROTEUS.tell_me_your_form()
PROTEUS.tell_me_the_future()
PROTEUS.tell_me_your_form()
PROTEUS.tell_me_the_future()
PROTEUS.tell_me_your_form()
PROTEUS.tell_me_the_future()
PROTEUS.tell_me_your_form()
PROTEUS.tell_me_the_future()
```

./proxy/interface_proteus.py

```
"The Proteus Interface"
```

```
from abc import ABCMeta, abstractmethod

class IProteus(metaclass=ABCMeta):  # pylint: disable=too-few-public-
methods
    "A Greek mythological character that can change to many forms"

    @staticmethod
    @abstractmethod
    def tell_me_the_future():
        "Proteus will change form rather than tell you the future"

    @staticmethod
    @abstractmethod
    def tell_me_your_form():
        "The form of Proteus is elusive like the sea"
```

./proxy/lion.py

```
"A Lion Class"
import random
from interface_proteus import IProteus
import leopard
import serpent

class Lion(IProteus):  # pylint: disable=too-few-public-methods
    "Proteus in the form of a Lion"

    name = "Lion"

    def tell_me_the_future(self):
        "Proteus will change to something random"
        self.__class__ = leopard.Leopard if random.randint(
            0, 1) else serpent.Serpent

    @classmethod
    def tell_me_your_form(cls):
        print("I am the form of a " + cls.name)
```

./proxy/serpent.py

```
"A Serpent Class"
import random
from interface_proteus import IProteus
import lion
import leopard
```

```
class Serpent(IProteus):  # pylint: disable=too-few-public-methods
    "Proteus in the form of a Serpent"

    name = "Serpent"

    def tell_me_the_future(self):
        "Proteus will change to something random"
        self.__class__ = leopard.Leopard if random.randint(0, 1) else
lion.Lion

    @classmethod
    def tell_me_your_form(cls):
        print("I am the form of a " + cls.name)
```

./proxy/leopard.py

```
"A Leopard Class"
import random
from interface_proteus import IProteus
import lion
import serpent

class Leopard(IProteus):  # pylint: disable=too-few-public-methods
    "Proteus in the form of a Leopard"

    name = "Leopard"

    def tell_me_the_future(self):
        "Proteus will change to something random"
        self.__class__ = serpent.Serpent if random.randint(0, 1) else
lion.Lion

    @classmethod
    def tell_me_your_form(cls):
        print("I am the form of a " + cls.name)
```

6.7.9 Output

```
python ./proxy/client.py
I am the form of a Lion
I am the form of a Leopard
I am the form of a Serpent
I am the form of a Leopard
I am the form of a Lion
```

6.7.10 New Coding Concepts

Changing An Objects Class At Runtime.

SBCODE Video ID #e600e5

You change the class of an object by running `self.__class__ = SomeOtherClass`

Note that doing this does not affect any variables created during initialisation, eg `self.variable_name = 'abc'` , since the object itself hasn't changed. Only its class methods and static attributes have been replaced with the class methods and static attributes of the other class.

This explains how calling `tell_me_the_future()` and `tell_me_your_form()` produced different results after changing `self.__class__`

Avoiding Circular Imports.

SBCODE Video ID #bcf58f

Normally in all the examples so far, I have been importing using the form

> ```
> from module import Class
> ```

In `./proxy/client.py` I import the `Lion` module. The `Lion` module itself imports the `Leopard` and `Serpent` modules, that in turn also re import the `Lion` module again. This is a circular import and occurs in some situations when you separate your modules into individual files.

Circular imports will prevent the python interpreter from compiling your `.py` file into byte code.

The error will appear like,

> ```
> cannot import name 'Lion' from partially initialized module 'lion' (most
> likely due to a circular import)
> ```

To avoid circular import errors, you can import modules using the form.

> ```
> import module
> ```

and when the import is actually needed in some method

> ```
> OBJECT = module.ClassName
> ```

See the Lion, Serpent and Leopard classes for examples.

6.7.11 Summary

- Proxy forwards requests onto the Real Subject when applicable, depending on the kind of proxy.
- A virtual proxy can cache elements of a real subject before loading the full object into memory.
- A protection proxy can provide an authentication layer. For example, an NGINX proxy can add Basic Authentication restriction to a HTTP request.
- A proxy can perform multiple tasks if necessary.
- A proxy is different than an Adapter. The Adapter will try to adapt two existing interfaces together. The Proxy will use the same interface as the subject.
- It is also very similar to the Facade, except you can add extra responsibilities, just like the Decorator. The Decorator however can be used recursively.
- The intent of the Proxy is to provide a stand in for when it is inconvenient to access a real subject directly.
- The Proxy design pattern may also be called the Surrogate design pattern.

7. Behavioral

7.1 Command Design Pattern

7.1.1 Overview

SBCODE Video ID #8c8ea3

The **Command** pattern is a behavioral design pattern, in which an abstraction exists between an object that invokes a command, and the object that performs it.

E.g., a button will call the **Invoker**, that will call a pre-registered **Command**, that the **Receiver** will perform.

A Concrete Class will delegate a request to a command object, instead of implementing the request directly.

Using a command design pattern allows you to separate concerns and to solve problems of the concerns independently of each other.

E.g., logging the execution of a command and its outcome.

The command pattern is a good solution for implementing UNDO/REDO functionality into your application.

Uses:

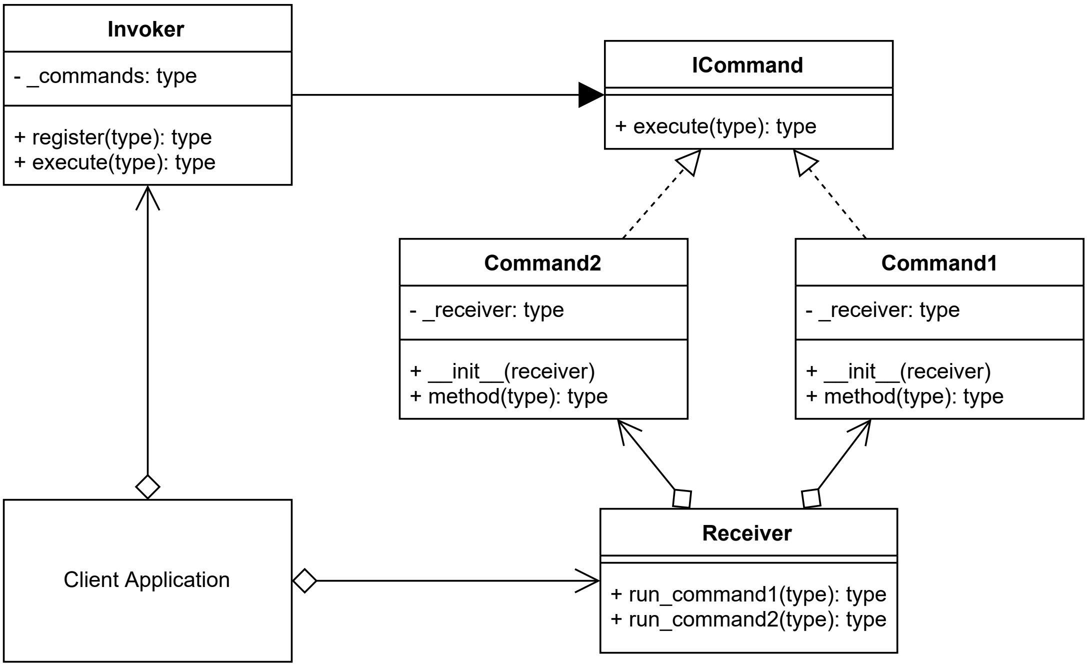
- GUI Buttons, menus
- Macro recording
- Multi-level undo/redo
- Networking - send whole command objects across a network, even as a batch
- Parallel processing or thread pools
- Transactional behavior
- Wizards

7.1.2 Terminology

- **Receiver**: The object that will receive and execute the command.
- **Invoker**: The object that sends the command to the receiver. E.g., A button.
- **Command Object**: Itself, an object, that implements an execute, or action method, and contains all required information to execute it.

- **Client**: The application or component that is aware of the Receiver, Invoker and Commands.

7.1.3 Command Pattern UML Diagram



7.1.4 Source Code

The Client instantiates a Receiver that accepts certain commands that do things.

The Client then creates two Command objects that will call one of the specific commands on the Receiver.

The Client then creates an Invoker, E.g., a user interface with buttons, and registers both Commands into the Invokers dictionary of commands.

The Client doesn't call the receivers commands directly, but the via the Invoker, that then calls the registered Command objects `execute()` method.

This abstraction between the invoker, command and receiver, allows the Invoker to add extra functionality such as history, replay, UNDO/REDO, logging, alerting and any other useful things that may be required.

./command/command_concept.py

```
"The Command Pattern Concept"
from abc import ABCMeta, abstractmethod
```

```
class ICommand(metaclass=ABCMeta):  # pylint: disable=too-few-public-methods
    "The command interface, that all commands will implement"
    @staticmethod
    @abstractmethod
    def execute():
        "The required execute method that all command objects will use"

class Invoker:
    "The Invoker Class"

    def __init__(self):
        self._commands = {}

    def register(self, command_name, command):
        "Register commands in the Invoker"
        self._commands[command_name] = command

    def execute(self, command_name):
        "Execute any registered commands"
        if command_name in self._commands.keys():
            self._commands[command_name].execute()
        else:
            print(f"Command [{command_name}] not recognised")

class Receiver:
    "The Receiver"

    @staticmethod
    def run_command_1():
        "A set of instructions to run"
        print("Executing Command 1")

    @staticmethod
    def run_command_2():
        "A set of instructions to run"
        print("Executing Command 2")

class Command1(ICommand):  # pylint: disable=too-few-public-methods
    """A Command object, that implements the ICommand interface and
    runs the command on the designated receiver"""

    def __init__(self, receiver):
        self._receiver = receiver

    def execute(self):
        self._receiver.run_command_1()

class Command2(ICommand):  # pylint: disable=too-few-public-methods
```

```
    """A Command object, that implements the ICommand interface and
    runs the command on the designated receiver"""

    def __init__(self, receiver):
        self._receiver = receiver

    def execute(self):
        self._receiver.run_command_2()

# The CLient
# Create a receiver
RECEIVER = Receiver()

# Create Commands
COMMAND1 = Command1(RECEIVER)
COMMAND2 = Command2(RECEIVER)

# Register the commands with the invoker
INVOKER = Invoker()
INVOKER.register("1", COMMAND1)
INVOKER.register("2", COMMAND2)

# Execute the commands that are registered on the Invoker
INVOKER.execute("1")
INVOKER.execute("2")
INVOKER.execute("1")
INVOKER.execute("2")
```

7.1.5 Output

```
python ./command/command_concept.py
Executing Command 1
Executing Command 2
Executing Command 1
Executing Command 2
```

7.1.6 Command Use Case

SBCODE Video ID #30566d

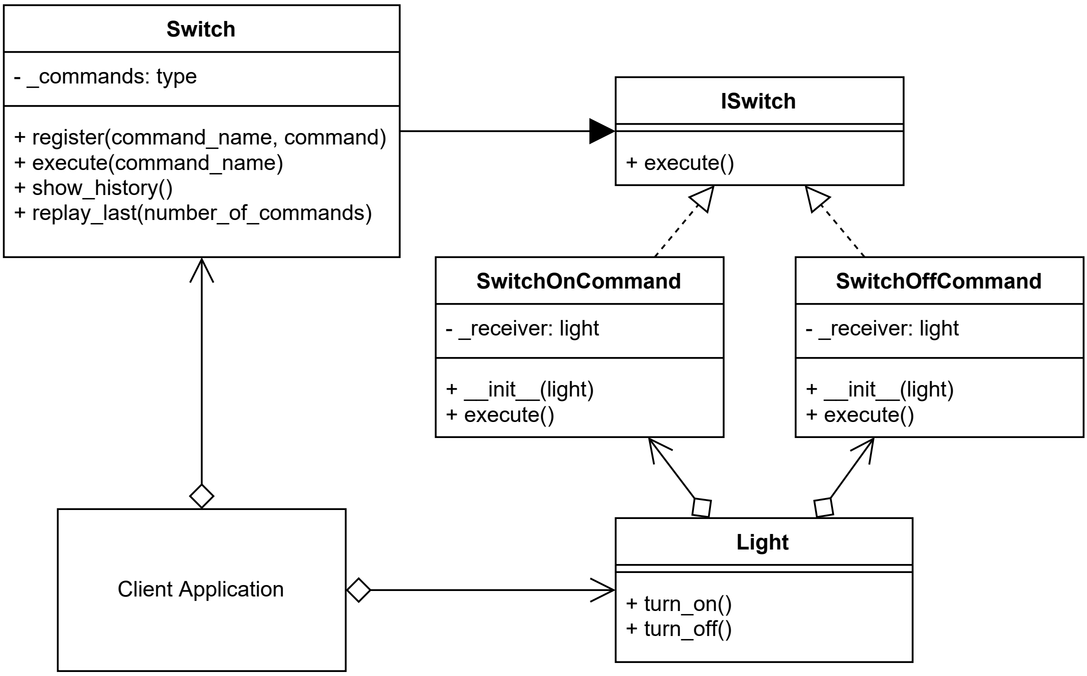
This will be a smart light switch.

This light switch will keep a history of each time one of its commands was called.

And it can replay its commands.

A smart light switch could be extended in the future to be called remotely or automated depending on sensors.

7.1.7 Example UML Diagram



7.1.8 Source Code

./command/client.py

```
"The Command Pattern Use Case Example. A smart light Switch"
from light import Light
from switch import Switch
from switch_on_command import SwitchOnCommand
from switch_off_command import SwitchOffCommand

# Create a receiver
LIGHT = Light()

# Create Commands
SWITCH_ON = SwitchOnCommand(LIGHT)
SWITCH_OFF = SwitchOffCommand(LIGHT)

# Register the commands with the invoker
SWITCH = Switch()
SWITCH.register("ON", SWITCH_ON)
SWITCH.register("OFF", SWITCH_OFF)
```

```
# Execute the commands that are registered on the Invoker
SWITCH.execute("ON")
SWITCH.execute("OFF")
SWITCH.execute("ON")
SWITCH.execute("OFF")

# show history
SWITCH.show_history()

# replay last two executed commands
SWITCH.replay_last(2)
```

./command/light.py

```
"The Light. The Receiver"

class Light:
    "The Receiver"

    @staticmethod
    def turn_on():
        "A set of instructions to run"
        print("Light turned ON")

    @staticmethod
    def turn_off():
        "A set of instructions to run"
        print("Light turned OFF")
```

./command/switch.py

```
"""
The Switch (Invoker) Class.
You can flick the switch and it then invokes a registered command
"""
from datetime import datetime
import time

class Switch:
    "The Invoker Class."

    def __init__(self):
        self._commands = {}
        self._history = []
    def show_history(self):
        "Print the history of each time a command was invoked"
        for row in self._history:
            print(
                f"{datetime.fromtimestamp(row[0]).strftime('%H:%M:%S')}"
                f" : {row[1]}"
            )

    def register(self, command_name, command):
        "Register commands in the Invoker"
        self._commands[command_name] = command

    def execute(self, command_name):
        "Execute any registered commands"
        if command_name in self._commands.keys():
            self._commands[command_name].execute()
            self._history.append((time.time(), command_name))
        else:
            print(f"Command [{command_name}] not recognised")

    def replay_last(self, number_of_commands):
        "Replay the last N commands"
        commands = self._history[-number_of_commands:]
        for command in commands:
            self._commands[command[1]].execute()
            #or if you want to record these replays in history
            #self.execute(command[1])
```

./command/switch_on_command.py

```
"""
A Command object, that implements the ISwitch interface and runs the
command on the designated receiver
"""
from interface_switch import ISwitch

class SwitchOnCommand(ISwitch):  # pylint: disable=too-few-public-methods
    "Switch On Command"

    def __init__(self, light):
        self._light = light

    def execute(self):
        self._light.turn_on()
```

./command/switch_off_command.py

```
"""
A Command object, that implements the ISwitch interface and runs the
command on the designated receiver
"""
from interface_switch import ISwitch

class SwitchOffCommand(ISwitch):  # pylint: disable=too-few-public-methods
    "Switch Off Command"

    def __init__(self, light):
        self._light = light

    def execute(self):
        self._light.turn_off()
```

./command/interface_switch.py

```
"The switch interface, that all commands will implement"
from abc import ABCMeta, abstractmethod

class ISwitch(metaclass=ABCMeta):  # pylint: disable=too-few-public-methods
    "The switch interface, that all commands will implement"

    @staticmethod
    @abstractmethod
    def execute():
        "The required execute method that all command objects will use"
```

7.1.9 Output

```
python ./command/client.py
Light turned ON
Light turned OFF
Light turned ON
Light turned OFF
11:23:35 : ON
11:23:35 : OFF
11:23:35 : ON
11:23:35 : OFF
Light turned ON
Light turned OFF
```

7.1.10 New Coding Concepts

_Single Leading Underscore

SBCODE Video ID #37437a

The single leading underscore `_variable` , on your class variables is a useful indicator to other developers that this property should be considered private.

Private, in C style languages, means that the variable/field/property is hidden and cannot be accessed outside of the class. It can only be used internally by its own class methods.

Python does not have a public/private accessor concept so the variable is not actually private and can still be used outside of the class in other modules.

It is just a useful construct that you will see developers use as a recommendation not to reference this variable directly outside of this class, but use a dedicated method or property instead.

7.1.11 Summary

- State should not be managed in the Command object itself.
- There can be one or more Invokers that can execute the Command at a later time.
- The Command object is especially useful if you want to UNDO/REDO commands at later time.
- The Command pattern is similar to the Memento pattern in the way that it can also be used for UNDO/REDO purposes. However, the Memento pattern is about recording and replacing the state of an object, whereas the Command pattern executes a predefined command. E.g., Draw, Turn, Resize, Save, etc.

7.2 Chain of Responsibility Design Pattern

7.2.1 Overview

SBCODE Video ID #e4659e

Chain of Responsibility pattern is a behavioral pattern used to achieve loose coupling in software design.

In this pattern, an object is passed to a **Successor**, and depending on some kind of logic, will or won't be passed onto another successor and processed. There can be any number of different successors and successors can be re-processed recursively.

This process of passing objects through multiple successors is called a chain.

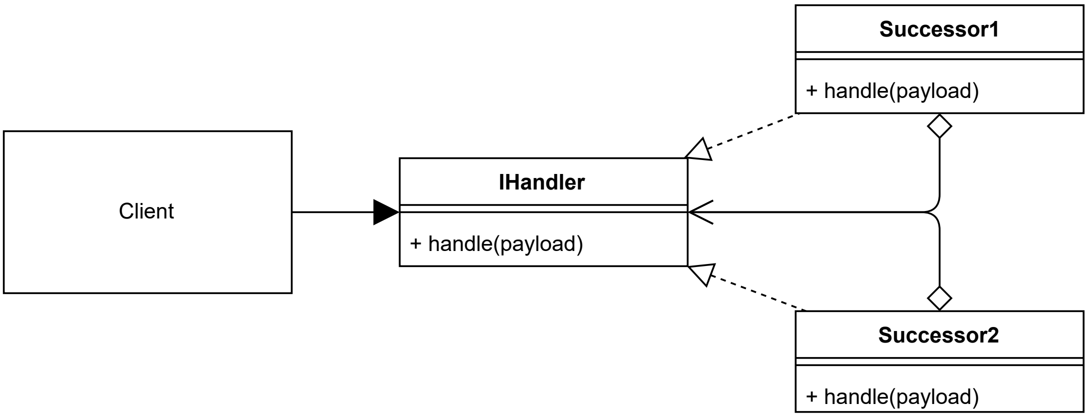
The object that is passed between each successor does not know about which successor will handle it. It is an independent object that may or may not be processed by a particular successor before being passed onto the next.

The chain that the object will pass through is normally dynamic at runtime, although you can hard code the order or start of the chain, so each successor will need to comply with a common interface that allows the object to be received and passed onto the next successor.

7.2.2 Terminology

- **Handler Interface**: A common interface for handling and passing objects through each successor.
- **Concrete Handler**: The class acting as the **Successor** handling the requests and passing onto the next.
- **Client**: The application or class that initiates the call to the first concrete handler (successor) in the chain.

7.2.3 Chain of Responsibility UML Diagram



7.2.4 Source Code

In this concept code, a chain is created with a default first successor. A number is passed to a successor, that then does a random test, and depending on the result will modify the number and then pass it onto the next successor. The process is randomized and will end at some point when there are no more successors designated.

./chain_of_responsibility/chain_of_responsibility_concept.py

```
# pylint: disable=too-few-public-methods
"The Chain Of Responsibility Pattern Concept"
import random
from abc import ABCMeta, abstractmethod

class IHandler(metaclass=ABCMeta):
    "The Handler Interface that the Successors should implement"
    @staticmethod
    @abstractmethod
    def handle(payload):
        "A method to implement"

class Successor1(IHandler):
    "A Concrete Handler"
    @staticmethod
    def handle(payload):
        print(f"Successor1 payload = {payload}")
        test = random.randint(1, 2)
        if test == 1:
            payload = payload + 1
            payload = Successor1().handle(payload)
        if test == 2:
            payload = payload - 1
            payload = Successor2().handle(payload)
        return payload

class Successor2(IHandler):
    "A Concrete Handler"
    @staticmethod
    def handle(payload):
        print(f"Successor2 payload = {payload}")
        test = random.randint(1, 3)
        if test == 1:
            payload = payload * 2
            payload = Successor1().handle(payload)
        if test == 2:
            payload = payload / 2
            payload = Successor2().handle(payload)
        return payload

class Chain():
    "A chain with a default first successor"
    @staticmethod
    def start(payload):
        "Setting the first successor that will modify the payload"
        return Successor1().handle(payload)

# The Client
CHAIN = Chain()
PAYLOAD = 1
OUT = CHAIN.start(PAYLOAD)
print(f"Finished result = {OUT}")
```

7.2.5 Output

```
python ./chain_of_responsibility/chain_of_responsibility_concept.py
Successor1 payload = 1
Successor2 payload = -1
Successor2 payload = -0.5
Successor2 payload = -0.25
Successor1 payload = -0.5
Successor1 payload = 0.5
Successor2 payload = -1.5
Finished result = -1.5
```

7.2.6 Chain of Responsibility Use Case

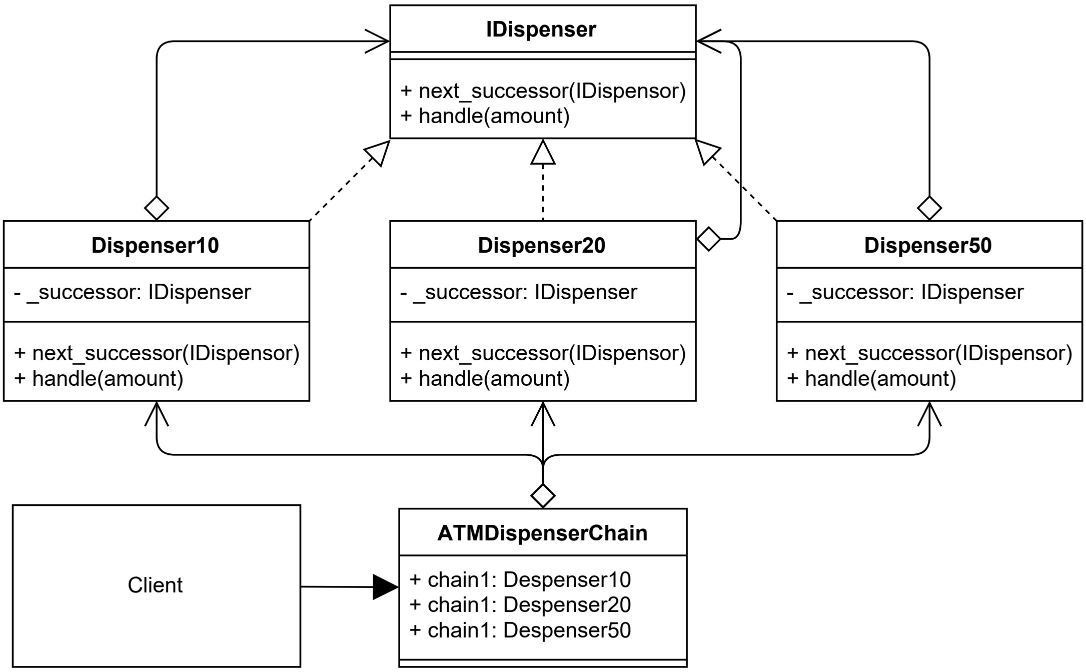
SBCODE Video ID #d89543

In the ATM example below, the chain is hard coded in the client first to dispense amounts of £50s, then £20s and then £10s in order.

This default chain order helps to ensure that the minimum number of notes will be dispensed. Otherwise, it might dispense 5 x £10 when it would have been better to dispense 1 x £50.

Each successor may be re-called recursively for each denomination depending on the value that was requested for withdrawal.

7.2.7 Example UML Diagram



7.2.8 Source Code

./chain_of_responsibility/client.py

```
"An ATM Dispenser that dispenses denominations of notes"
import sys
from atm_dispenser_chain import ATMDispenserChain

ATM = ATMDispenserChain()
AMOUNT = int(input("Enter amount to withdrawal : "))
if AMOUNT < 10 or AMOUNT % 10 != 0:
    print("Amount should be positive and in multiple of 10s.")
    sys.exit()
# process the request
```

```
ATM.chain1.handle(AMOUNT)
print("Now go spoil yourself")
```

./chain_of_responsibility/atm_dispenser_chain.py

```
"The ATM Dispenser Chain"
from dispenser10 import Dispenser10
from dispenser20 import Dispenser20
from dispenser50 import Dispenser50

class ATMDispenserChain:  # pylint: disable=too-few-public-methods
    "The Chain Client"

    def __init__(self):
        # initializing the successors chain
        self.chain1 = Dispenser50()
        self.chain2 = Dispenser20()
        self.chain3 = Dispenser10()
        # Setting a default successor chain that will process the 50s
        # first, the 20s second and the 10s last. The successor chain
        # will be recalculated dynamically at runtime.
        self.chain1.next_successor(self.chain2)
        self.chain2.next_successor(self.chain3)
```

./chain_of_responsibility/interface_dispenser.py

```
"The ATM Notes Dispenser Interface"
from abc import ABCMeta, abstractmethod

class IDispenser(metaclass=ABCMeta):
    "Methods to implement"
    @staticmethod
    @abstractmethod
    def next_successor(successor):
        """Set the next handler in the chain"""

    @staticmethod
    @abstractmethod
    def handle(amount):
        """Handle the event"""
```

./chain_of_responsibility/dispenser10.py

```
"A dispenser of £10 notes"
from interface_dispenser import IDispenser
```

```python
class Dispenser10(IDispenser):
    "Dispenses £10s if applicable, otherwise continues to next successor"

    def __init__(self):
        self._successor = None

    def next_successor(self, successor):
        "Set the next successor"
        self._successor = successor

    def handle(self, amount):
        "Handle the dispensing of notes"
        if amount >= 10:
            num = amount // 10
            remainder = amount % 10
            print(f"Dispensing {num} £10 note")
            if remainder != 0:
                self._successor.handle(remainder)
        else:
            self._successor.handle(amount)
```

./chain_of_responsibility/dispenser20.py

```python
"A dispenser of £20 notes"
from interface_dispenser import IDispenser

class Dispenser20(IDispenser):
    "Dispenses £20s if applicable, otherwise continues to next successor"

    def __init__(self):
        self._successor = None

    def next_successor(self, successor):
        "Set the next successor"
        self._successor = successor

    def handle(self, amount):
        "Handle the dispensing of notes"
        if amount >= 20:
            num = amount // 20
            remainder = amount % 20
            print(f"Dispensing {num} £20 note(s)")
            if remainder != 0:
                self._successor.handle(remainder)
        else:
            self._successor.handle(amount)
```

./chain_of_responsibility/dispenser50.py

```
"A dispenser of £50 notes"
from interface_dispenser import IDispenser

class Dispenser50(IDispenser):
    "Dispenses £50s if applicable, otherwise continues to next successor"

    def __init__(self):
        self._successor = None

    def next_successor(self, successor):
        "Set the next successor"
        self._successor = successor

    def handle(self, amount):
        "Handle the dispensing of notes"
        if amount >= 50:
            num = amount // 50
            remainder = amount % 50
            print(f"Dispensing {num} £50 note(s)")
            if remainder != 0:
                self._successor.handle(remainder)
        else:
            self._successor.handle(amount)
```

7.2.9 Output

```
python ./chain_of_responsibility/client.py
Enter amount to withdrawal : 180
Dispensing 3 £50 note(s)
Dispensing 1 £20 note(s)
Dispensing 1 £10 note
Now go spoil yourself
```

7.2.10 New Coding Concepts

Floor Division

SBCODE Video ID #56c97d

Normally division uses a single **/** character and will return a float even if the numbers are integers or exactly divisible with no remainder,

E.g.,

```
PS> python
>>> 9 / 3
3.0
```

Python Version 3 also has an option to return an integer version (floor) of the number by using the double // characters instead.

```
PS> python
>>> 9 // 3
3
```

See PEP-0238 : https://www.python.org/dev/peps/pep-0238/

Accepting User Input

SBCODE Video ID #675635

In the file ./chain_of_responsibility/client.py above, there is a command `input` .

The `input` command allows your script to accept user input from the command prompt.

In the ATM example, when you start it, it will ask the user to enter a number.

Then when the user presses the `enter` key, the input is converted to an integer and the value tested if valid.

```
AMOUNT = int(input("Enter amount to withdrawal : "))
if AMOUNT < 10 or AMOUNT % 10 != 0:
    ...continue
```

Note that in Python 2.x, use the `raw_input()` command instead of `input()` .

See PEP-3111 : https://www.python.org/dev/peps/pep-3111/

7.2.11 Summary

- The object will propagate through the chain until fully processed.
- The object does not know which successor or how many will process it.
- The next successor in the chain is chosen dynamically at runtime depending on logic from the current successor.
- Successors implement a common interface that makes them work independently of each other, so that they can be used recursively or possibly in a different order.

- A user wizard, or dynamic questionnaire are other common use cases for the chain of responsibility pattern.
- The chain of responsibility and Composite patterns are often used together because of their similar approach to hierarchy and possible re-ordering. The Composites parent/child relationship is set in an object's property by a process outside of the class and can be changed at runtime. While with the Chain of Responsibility, each successor runs a dynamic algorithm internally, to decide which successor is next in line.
- The chain can be fully dynamically created, or it can be set as a default with the possibility of changing at runtime.

7.3 Observer Pattern

7.3.1 Overview

SBCODE Video ID #f5a0d3

The **Observer** pattern is a software design pattern in which an object, called the **Subject** (**Observable**), manages a list of dependents, called **Observers**, and notifies them automatically of any internal state changes by calling one of their methods.

The Observer pattern follows the publish/subscribe concept. A subscriber, subscribes to a publisher. The publisher then notifies the subscribers when necessary.

The observer stores state that should be consistent with the subject. The observer only needs to store what is necessary for its own purposes.

A typical place to use the observer pattern is between your application and presentation layers. Your application is the manager of the data and is the single source of truth, and when the data changes, it can update all of the subscribers, that could be part of multiple presentation layers. For example, the score was changed in a televised cricket game, so all the web browser clients, mobile phone applications, leaderboard display on the ground and television graphics overlay, can all now have the updated information synchronized.

Most applications that involve a separation of data into a presentation layer can be broken further down into the Model-View-Controller (MVC) concept.

- **Controller** : The single source of truth.
- **Model** : The link or relay between a controller and a view. It may use any of the structural patterns (adapter, bridge, facade, proxy, etc.) at some point.
- **View** : The presentation layer of the of the data from the model.

The observer pattern can be used to manage the transfer of data across any layer and even internally to itself to add a further abstraction. In the MVC structure, the View can be a subscriber to the Model, that in turn can also be a subscriber to the controller. It can also happen the other way around if the use case warrants.

The Observer pattern allows you to vary subjects and observers independently. You can reuse subjects without reusing their observers, and vice versa. It lets you add observers without modifying the subject or any of the other observers.

The observer pattern is commonly described as a push model, where the subject pushes updates to all observers. But observers can pull for updates and also only if it decides it is necessary.

Whether you decide to use a push or pull concept to move data, then there are pros and cons to each. You may decide to use a combination of both to manage reliability.

E.g., When sending messages across a network, the receiving client, can be slow to receive the full message that was sent, or even timeout. This pushing from the sender's side can increase the amount of network hooks or threads if there are many messages still waiting to be fully delivered. The subject is taking responsibility for the delivery.

On the other hand, if the observer requests for an update from the subscriber, then the subject (observable) can return the information as part of the requests response. The observer could also indicate as part of the request, to only return data applicable to X, that would then make the response message smaller to transfer at the expense of making the observable more coupled to the observer.

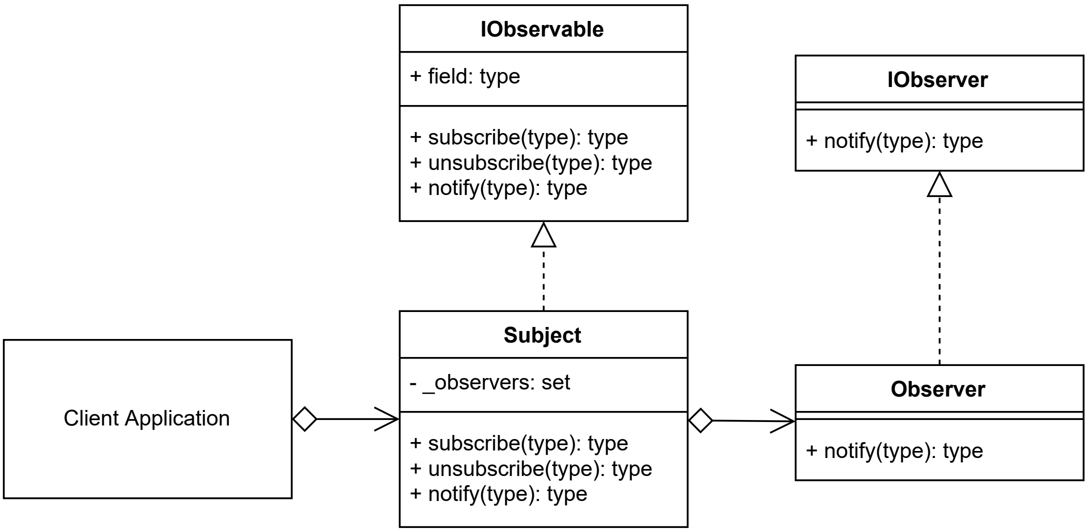
Use a push mechanism from the subject when updates are absolutely required in as close to real time from the perspective of the observer, noting that you may need to manage the potential of extra unresolved resources queueing up at the sender.

If updates on the observer end are allowed to suffer from some delay, then a pull mechanism is most reliable and easiest to manage since it is the responsibly of the observer to synchronize its state.

7.3.2 Terminology

- **Subject Interface**: (Observable Interface) The interface that the subject should implement.
- **Concrete Subject**: (Observable) The object that is the subject.
- **Observer Interface**: The interface that the observer should implement.
- **Concrete Observer**: The object that is the observer. There can be a variable number of observers that can subscribe/unsubscribe during runtime.

7.3.3 Observer UML Diagram



7.3.4 Source Code

A Subject (Observable) is created.

Two Observers are created. They could be across a network, but for demonstration purposes are within the same client.

The Subject notifies the Observers.

One of the Observers unsubscribes,

The Subject notifies the remaining Observer again.

./observer/observer_concept.py

```
# pylint: disable=too-few-public-methods
"Observer Design Pattern Concept"

from abc import ABCMeta, abstractmethod

class IObservable(metaclass=ABCMeta):
    "The Subject Interface"

    @staticmethod
    @abstractmethod
    def subscribe(observer):
        "The subscribe method"

    @staticmethod
    @abstractmethod
    def unsubscribe(observer):
        "The unsubscribe method"

    @staticmethod
    @abstractmethod
    def notify(observer):
        "The notify method"

class Subject(IObservable):
    "The Subject (Observable)"

    def __init__(self):
        self._observers = set()

    def subscribe(self, observer):
        self._observers.add(observer)

    def unsubscribe(self, observer):
        self._observers.remove(observer)

    def notify(self, *args):
        for observer in self._observers:
            observer.notify(self, *args)

class IObserver(metaclass=ABCMeta):
    "A method for the Observer to implement"

    @staticmethod
    @abstractmethod
    def notify(observable, *args):
        "Receive notifications"

class Observer(IObserver):
    "The concrete observer"

    def __init__(self, observable):
        observable.subscribe(self)

    def notify(self, observable, *args):
        print(f"Observer id:{id(self)} received {args}")

# The Client
SUBJECT = Subject()
OBSERVER_A = Observer(SUBJECT)
OBSERVER_B = Observer(SUBJECT)

SUBJECT.notify("First Notification", [1, 2, 3])

SUBJECT.unsubscribe(OBSERVER_B)
SUBJECT.notify("Second Notification", {"A": 1, "B": 2, "C": 3})
```

7.3.5 Output

```
python ./observer/observer_concept.py
Observer id:2084220160272 received ('First Notification', [1, 2, 3])
Observer id:2084220160224 received ('First Notification', [1, 2, 3])
Observer id:2084220160272 received ('Second Notification', {'A': 1, 'B':
2, 'C': 3})
```

7.3.6 Observer Use Case

SBCODE Video ID #0f7fc5

This example mimics the **MVC** approach described earlier.

There is an external process called a `DataController` , and a client process that holds a `DataModel` and multiple `DataViews` that are a Pie graph, Bar graph and Table view.

Note that this example runs in a single process, but imagine that the `DataController` is actually an external process running on a different server.

The `DataModel` subscribes to the `DataController` and the `DataViews` subscribe to the `DataModel` .

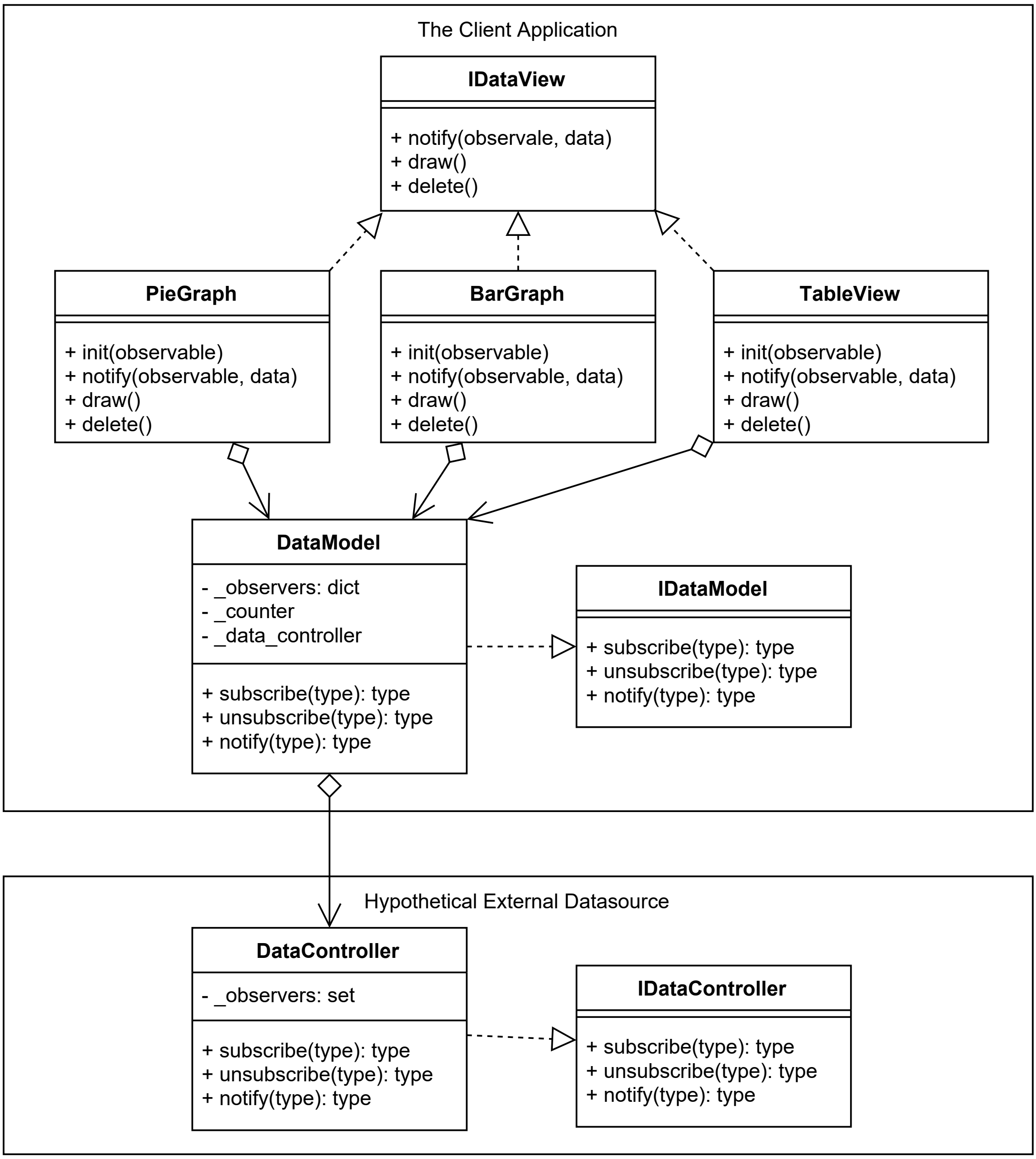
The client sets up the various views with a subscription to the `DataModel` .

The hypothetical external `DataController` then updates the external data, and the data then propagates through the layers to the views.

Note that in reality this example would be much more complex if multiple servers are involved. I am keeping it brief to demonstrate one possible use case of the observer pattern.

Also note that in the `DataController` , the references to the observers are contained in a Set, while in the `DataModel` I have used a Dictionary instead, so that you can see an alternate approach.

7.3.7 Example UML Diagram



7.3.8 Source Code

./observer/client.py

```
"Observer Design Pattern Concept"

from data_model import DataModel
from data_controller import DataController
from pie_graph_view import PieGraphView
from bar_graph_view import BarGraphView
from table_view import TableView

# A local data view that the hypothetical external controller updates
DATA_MODEL = DataModel()

# Add some visualisation that use the dataview
PIE_GRAPH_VIEW = PieGraphView(DATA_MODEL)
BAR_GRAPH_VIEW = BarGraphView(DATA_MODEL)
TABLE_VIEW = TableView(DATA_MODEL)

# A hypothetical data controller running in a different process
DATA_CONTROLLER = DataController()

# The hypothetical external data controller updates some data
DATA_CONTROLLER.notify([1, 2, 3])

# Client now removes a local BAR_GRAPH
BAR_GRAPH_VIEW.delete()

# The hypothetical external data controller updates the data again
DATA_CONTROLLER.notify([4, 5, 6])
```

./observer/table_view.py

```
"An observer"
from interface_data_view import IDataView

class TableView(IDataView):
    "The concrete observer"

    def __init__(self, observable):
        self._observable = observable
        self._id = self._observable.subscribe(self)

    def notify(self, data):
        print(f"TableView, id:{self._id}")
        self.draw(data)

    def draw(self, data):
        print(f"Drawing a Table view using data:{data}")

    def delete(self):
        self._observable.unsubscribe(self._id)
```

./observer/bar_graph_view.py

```
"An observer"
from interface_data_view import IDataView

class BarGraphView(IDataView):
    "The concrete observer"

    def __init__(self, observable):
        self._observable = observable
        self._id = self._observable.subscribe(self)

    def notify(self, data):
        print(f"BarGraph, id:{self._id}")
        self.draw(data)

    def draw(self, data):
        print(f"Drawing a Bar graph using data:{data}")

    def delete(self):
        self._observable.unsubscribe(self._id)
```

./observer/pie_graph_view.py

```
"An observer"
from interface_data_view import IDataView

class PieGraphView(IDataView):
    "The concrete observer"

    def __init__(self, observable):
        self._observable = observable
        self._id = self._observable.subscribe(self)

    def notify(self, data):
        print(f"PieGraph, id:{self._id}")
        self.draw(data)

    def draw(self, data):
        print(f"Drawing a Pie graph using data:{data}")

    def delete(self):
        self._observable.unsubscribe(self._id)
```

./observer/interface_data_view.py

```
"The Data View interface"
from abc import ABCMeta, abstractmethod
```

```
class IDataView(metaclass=ABCMeta):
    "A method for the Observer to implement"

    @staticmethod
    @abstractmethod
    def notify(data):
        "Receive notifications"

    @staticmethod
    @abstractmethod
    def draw(data):
        "Draw the view"

    @staticmethod
    @abstractmethod
    def delete():
        "a delete method to remove observer specific resources"
```

./observer/data_model.py

```
"A Data Model that observes the Data Controller"
from interface_data_model import IDataModel
from data_controller import DataController

class DataModel(IDataModel):
    "A Subject (Observable)"

    def __init__(self):
        self._observers = {}
        self._counter = 0
        # subscribing to an external hypothetical data controller
        self._data_controller = DataController()
        self._data_controller.subscribe(self)

    def subscribe(self, observer):
        self._counter = self._counter + 1
        self._observers[self._counter] = observer
        return self._counter

    def unsubscribe(self, observer_id):
        self._observers.pop(observer_id)

    def notify(self, data):
        for observer in self._observers:
            self._observers[observer].notify(data)
```

./observer/interface_data_model.py

```
"A Data Model Interface"
from abc import ABCMeta, abstractmethod

class IDataModel(metaclass=ABCMeta):
    "A Subject Interface"

    @staticmethod
    @abstractmethod
    def subscribe(observer):
        "The subscribe method"

    @staticmethod
    @abstractmethod
    def unsubscribe(observer_id):
        "The unsubscribe method"

    @staticmethod
    @abstractmethod
    def notify(data):
        "The notify method"
```

./observer/data_controller.py

```
"A Data Controller that is a Subject"
from interface_data_controller import IDataController

class DataController(IDataController):
    "A Subject (Observable)"

    _observers = set()

    def __new__(cls):
        return cls

    @classmethod
    def subscribe(cls, observer):
        cls._observers.add(observer)

    @classmethod
    def unsubscribe(cls, observer):
        cls._observers.remove(observer)

    @classmethod
    def notify(cls, *args):
        for observer in cls._observers:
            observer.notify(*args)
```

./observer/interface_data_controller.py

```
"A Data Controller Interface"
from abc import ABCMeta, abstractmethod

class IDataController(metaclass=ABCMeta):
    "A Subject Interface"
    @staticmethod
    @abstractmethod
    def subscribe(observer):
        "The subscribe method"

    @staticmethod
    @abstractmethod
    def unsubscribe(observer):
        "The unsubscribe method"

    @staticmethod
    @abstractmethod
    def notify(observer):
        "The notify method"
```

7.3.9 Output

```
python ./observer/client.py
PieGraph, id:1
Drawing a Pie graph using data:[1, 2, 3]
BarGraph, id:2
Drawing a Bar graph using data:[1, 2, 3]
TableView, id:3
Drawing a Table view using data:[1, 2, 3]
PieGraph, id:1
Drawing a Pie graph using data:[4, 5, 6]
TableView, id:3
Drawing a Table view using data:[4, 5, 6]
```

7.3.10 New Coding Concepts

Python Set

SBCODE Video ID #c81244

A Python **Set** is similar to a List. Except that the items in the Set are guaranteed to be unique, even if you try to add a duplicate. A set is a good choice for keeping a collection of observables, since the problem of duplicate observables is automatically handled.

A Set can be instantiated using the curly braces `{}` or `set()` , verses `[]` for a List and `()` for a Tuple. It is not the same as a Dictionary, that also uses `{}` , since the dictionary items are created as `key:value` pairs. ie `{"a": 1, "b": 2, "c": 3}`

```
PS> python
>>> items = {"yankee", "doodle", "dandy", "doodle"}
>>> items
{'yankee', 'doodle', 'dandy'}
>>> items.add("grandy")
>>> items
{'grandy', 'yankee', 'doodle', 'dandy'}
>>> items.remove("doodle")
>>> items
{'grandy', 'yankee', 'dandy'}
```

> **Note**
>
> If instantiating an empty **Set** then use `my_object = Set()` rather than `my_object = {}` to reduce ambiguity with creating an empty Dictionary.

7.3.11 Summary

- Use when a change to one object requires changing others and you don't know how many other objects need to be changed.
- A subject has a list of observers, each conforming to the observer interface. The subject doesn't need to know about the concrete class of any observer. It will notify the observer using the method described in the interface.
- Subjects and Observers can belong to any layer of a system whether extremely large or small.
- Using a Push or Pull mechanism for the Observer will depend on how you want your system to manage redundancy for particular data transfers. These things become more of a consideration when the Observer is separated further away from a subject and the message needs to traverse many layers, processes and systems.

7.4 Interpreter Design Pattern

7.4.1 Overview

SBCODE Video ID #5b415b

The **Interpreter** pattern helps to convert information from one language into another.

The language can be anything such as words in a sentence, numerical formulas or even software code.

The process is to convert the source information, into an **Abstract Syntax Tree (AST)** of **Terminal** and **Non-Terminal** expressions that all implement an `interpret()` method.

A Non-Terminal expression is a combination of other Non-Terminal and/or Terminal expressions.

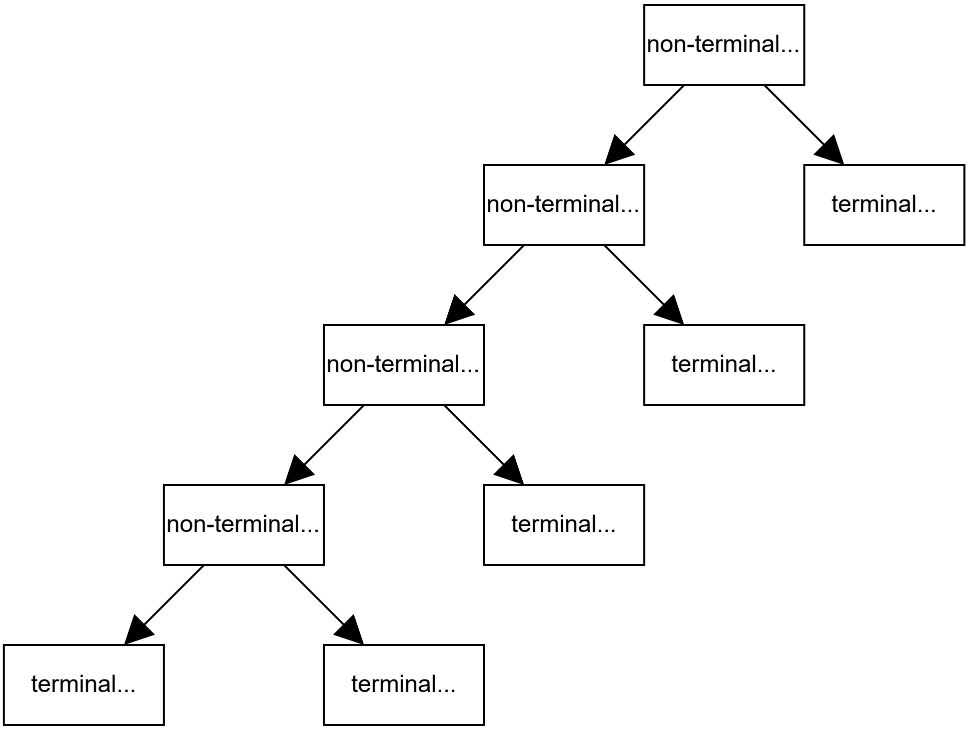
Terminal means terminated, i.e., there is no further processing involved.

An AST root starts with a Non-Terminal expression and then resolves down each branch until all expressions terminate.

An example expression is `A + B`.

The `A` and `B` are Terminal expressions and the `+` is Non-Terminal because it depends on the two other Terminal expressions.

The Image below, is an AST for the expression `5 + 4 - 3 + 7 - 2`



The official Interpreter pattern described in the original GoF Design Patterns book does not state how to construct an Abstract Syntax Tree. How your tree is constructed will depend on the grammatical constructs of your sentence that you want to interpret.

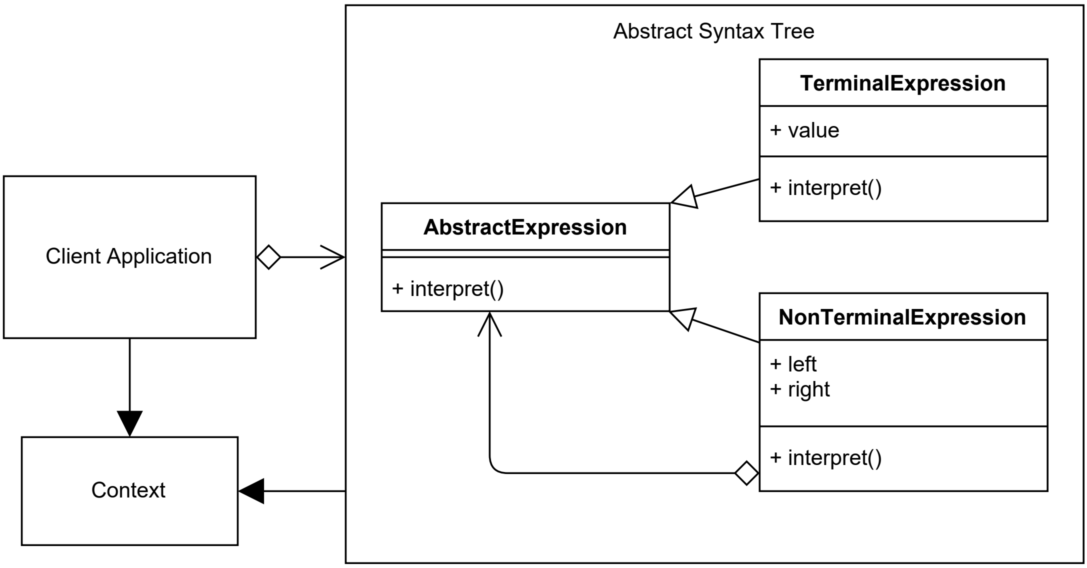
Abstract Syntax Trees can be created manually or dynamically from a custom parser script. In the first example code below, I construct the AST manually.

Once the AST is created, you can then choose the root node and then run the Interpret operation on that and it should interpret the whole tree recursively.

7.4.2 Terminology

- **Abstract Expression**: Describe the method(s) that Terminal and Non-Terminal expressions should implement.
- **Non-Terminal Expression**: A composite of Terminal and/or Non-Terminal expressions.
- **Terminal Expression**: A leaf node Expression.
- **Context**: Context is state that can be passed through interpret operations if necessary.
- **Client**: Builds or is given an Abstract Syntax Tree to interpret.

7.4.3 Interpreter UML Diagram



7.4.4 Source Code

In this example, I interpret the string `5 + 4 - 3 + 7 - 2` and then calculate the result.

The grammar of the string follows a pattern of Number -> Operator -> Number -> etc.

I convert the string into a list of tokens that I can refer to by index in the list.

I then construct the AST manually, by adding a

1. Non-Terminal `Add` row containing two Terminals for the `5` and `4` ,
2. Non-Terminal `Subtract` row containing the previous Non-Terminal row and the `3`
3. Non-Terminal `Add` row containing the previous Non-Terminal row and the `7`
4. Non-Terminal `Subtract` row containing the previous Non-Terminal row and the `2`

The AST root becomes the final row that was added, and then I can run the `interpret()` method on that, which will interpret the full AST recursively because each AST row references the row above it.

./interpreter/interpreter_concept.py

```
# pylint: disable=too-few-public-methods
"The Interpreter Pattern Concept"

class AbstractExpression():
    "All Terminal and Non-Terminal expressions will implement an
```

```
`interpret` method"
    @staticmethod
    def interpret():
        """
        The `interpret` method gets called recursively for each
        AbstractExpression
        """

class Number(AbstractExpression):
    "Terminal Expression"

    def __init__(self, value):
        self.value = int(value)

    def interpret(self):
        return self.value

    def __repr__(self):
        return str(self.value)

class Add(AbstractExpression):
    "Non-Terminal Expression."

    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() + self.right.interpret()

    def __repr__(self):
        return f"({self.left} Add {self.right})"

class Subtract(AbstractExpression):
    "Non-Terminal Expression"

    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() - self.right.interpret()

    def __repr__(self):
        return f"({self.left} Subtract {self.right})"

# The Client
# The sentence complies with a simple grammar of
# Number -> Operator -> Number -> etc,
SENTENCE = "5 + 4 - 3 + 7 - 2"
print(SENTENCE)

# Split the sentence into individual expressions that will be added to
# an Abstract Syntax Tree (AST) as Terminal and Non-Terminal expressions
TOKENS = SENTENCE.split(" ")
print(TOKENS)

# Manually Creating an Abstract Syntax Tree from the tokens
AST: list[AbstractExpression] = []  # A list of AbstractExpressions
AST.append(Add(Number(TOKENS[0]), Number(TOKENS[2])))  # 5 + 4
AST.append(Subtract(AST[0], Number(TOKENS[4])))        # ^ - 3
AST.append(Add(AST[1], Number(TOKENS[6])))             # ^ + 7
AST.append(Subtract(AST[2], Number(TOKENS[8])))        # ^ - 2

# Use the final AST row as the root node.
AST_ROOT = AST.pop()

# Interpret recursively through the full AST starting from the root.
print(AST_ROOT.interpret())

# Print out a representation of the AST_ROOT
print(AST_ROOT)
```

7.4.5 Output

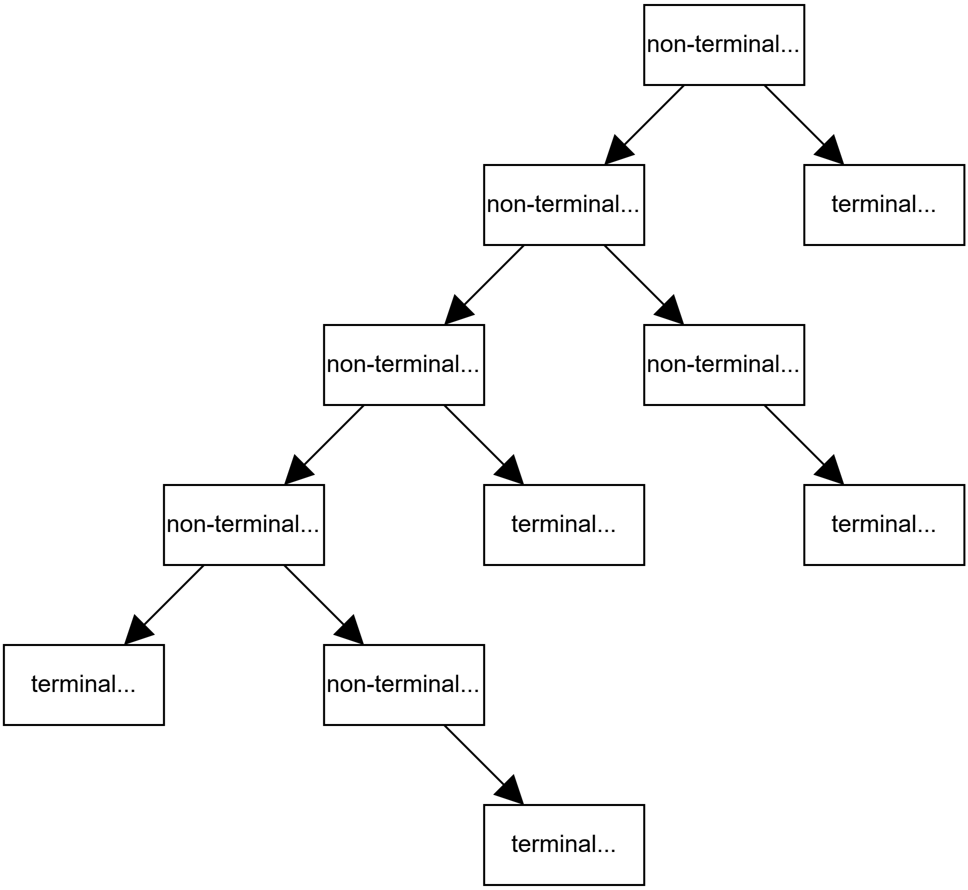
```
python ./interpreter/interpreter_concept.py
5 + 4 - 3 + 7 - 2
['5', '+', '4', '-', '3', '+', '7', '-', '2']
11
((((5 Add 4) Subtract 3) Add 7) Subtract 2)
```

7.4.6 Interpreter Use Case

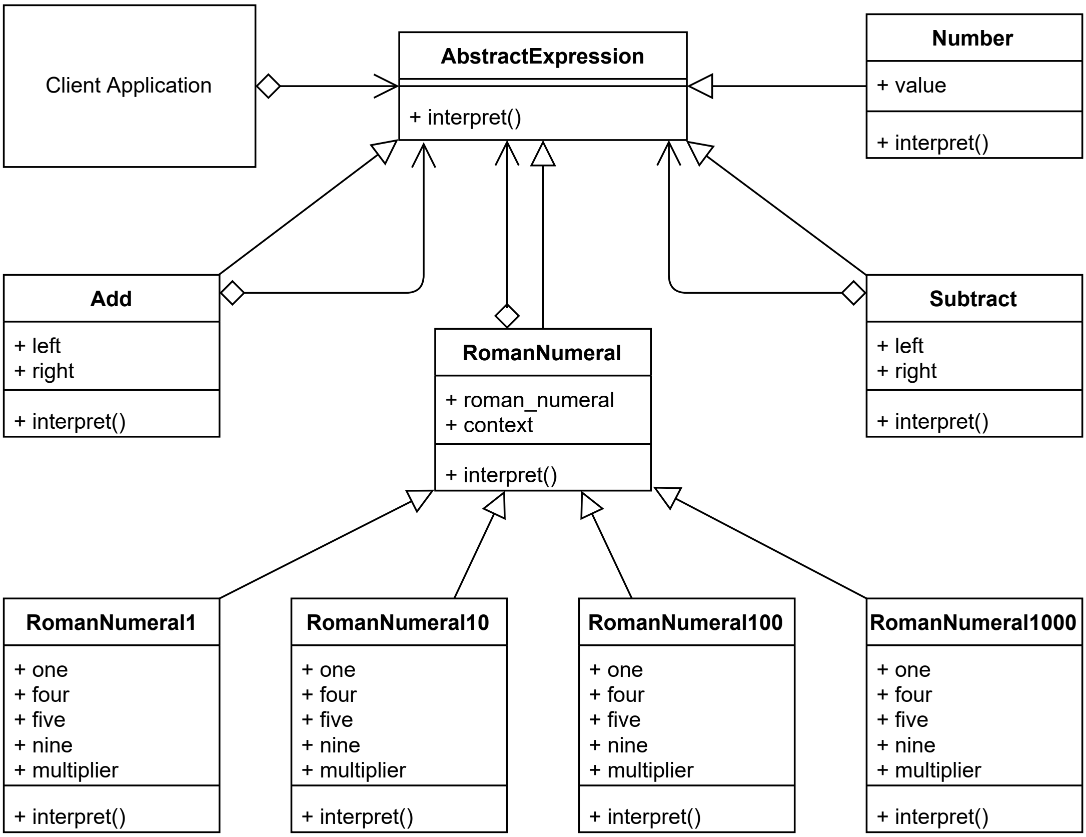
SBCODE Video ID #eb7859

The example use case will expand on the concept example by dynamically creating the AST and converting roman numerals to integers as well as calculating the final result.

The Image below, is an AST for the expression `5 + IV - 3 + VII - 2`


non-terminal...
non-terminal...
terminal...
non-terminal...
non-terminal...
non-terminal...
terminal...
terminal...
terminal...
non-terminal...
terminal...

7.4.7 Example UML Diagram



7.4.8 Source Code

./interpreter/client.py

```
"The Interpreter Pattern Use Case Example"

from sentence_parser import Parser

# The sentence complies with a simple grammar of
# Number -> Operator -> Number -> etc,
SENTENCE = "5 + IV - 3 + VII - 2"
# SENTENCE = "V + IV - III + 7 - II"
# SENTENCE= "CIX + V"
# SENTENCE = "CIX + V - 3 + VII - 2"
# SENTENCE = "MMMCMXCIX - CXIX + MCXXII - MMMCDXII - XVIII - CCXXXV"
print(SENTENCE)

AST_ROOT = Parser.parse(SENTENCE)

# Interpret recursively through the full AST starting from the root.
```

```
print(AST_ROOT.interpret())

# Print out a representation of the AST_ROOT
print(AST_ROOT)
```

./interpreter/abstract_expression.py

```
"An Abstract Expression"
# pylint: disable=too-few-public-methods
class AbstractExpression():
    """
    All Terminal and Non-Terminal expressions will implement an
    `interpret` method
    """
    @staticmethod
    def interpret():
        """
        The `interpret` method gets called recursively for
        each AbstractExpression
        """
```

./interpreter/number.py

```
"A Number. This is a leaf node Expression"
from abstract_expression import AbstractExpression

class Number(AbstractExpression):
    "Terminal Expression"

    def __init__(self, value):
        self.value = int(value)

    def interpret(self):
        return self.value

    def __repr__(self):
        return str(self.value)
```

./interpreter/add.py

```
"Add Expression. This is a Non-Terminal Expression"
from abstract_expression import AbstractExpression

class Add(AbstractExpression):
    "Non-Terminal Expression."
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() + self.right.interpret()

    def __repr__(self):
        return f"({self.left} Add {self.right})"
```

./interpreter/subtract.py

```
"Subtract Expression. This is a Non-Terminal Expression"
from abstract_expression import AbstractExpression

class Subtract(AbstractExpression):
    "Non-Terminal Expression"

    def __init__(self, left, right):
        self.left = left
        self.right = right

    def interpret(self):
        return self.left.interpret() - self.right.interpret()

    def __repr__(self):
        return f"({self.left} Subtract {self.right})"
```

./interpreter/roman_numeral.py

```
# pylint: disable=too-few-public-methods
"Roman Numeral Expression. This is a Non-Terminal Expression"
from abstract_expression import AbstractExpression
from number import Number

class RomanNumeral(AbstractExpression):
    "Non Terminal expression"

    def __init__(self, roman_numeral):
        self.roman_numeral = roman_numeral
        self.context = [roman_numeral, 0]

    def interpret(self):
        RomanNumeral1000.interpret(self.context)
        RomanNumeral100.interpret(self.context)
        RomanNumeral10.interpret(self.context)
        RomanNumeral1.interpret(self.context)
        return Number(self.context[1]).interpret()

    def __repr__(self):
        return f"{self.roman_numeral}({self.context[1]})"

class RomanNumeral1(RomanNumeral):
    "Roman Numerals 1 - 9"
    one = "I"
    four = "IV"
    five = "V"
    nine = "IX"
    multiplier = 1

    @classmethod
    def interpret(cls, *args):

        context = args[0]

        if not context[0]:
            return Number(context[1]).interpret()

        if context[0][0: 2] == cls.nine:
            context[1] += (9 * cls.multiplier)
            context[0] = context[0][2:]
        elif context[0][0] == cls.five:
            context[1] += (5 * cls.multiplier)
            context[0] = context[0][1:]
        elif context[0][0: 2] == cls.four:
            context[1] += + (4 * cls.multiplier)
            context[0] = context[0][2:]

        while context[0] and context[0][0] == cls.one:
            context[1] += (1 * cls.multiplier)
            context[0] = context[0][1:]

        return Number(context[1]).interpret()

class RomanNumeral10(RomanNumeral1):
    "Roman Numerals 10 - 99"
    one = "X"
    four = "XL"
    five = "L"
    nine = "XC"
    multiplier = 10

class RomanNumeral100(RomanNumeral1):
    "Roman Numerals 100 - 999"
    one = "C"
    four = "CD"
    five = "D"
    nine = "CM"
    multiplier = 100

class RomanNumeral1000(RomanNumeral1):
    "Roman Numerals 1000 - 3999"
    one = "M"
    four = ""
    five = ""
    nine = ""
    multiplier = 1000
```

./interpreter/sentence_parser.py

```
"A Custom Parser for creating an Abstract Syntax Tree"

from number import Number
from add import Add
from subtract import Subtract
from roman_numeral import RomanNumeral

class Parser:
    "Dynamically create the Abstract Syntax Tree"

    @classmethod
    def parse(cls, sentence):
        "Create the AST from the sentence"

        tokens = sentence.split(" ")
        print(tokens)

        tree = []  # Abstract Syntax Tree
        while len(tokens) > 1:

            left_expression = cls.decide_left_expression(tree, tokens)

            # get the operator, make the token list shorter
            operator = tokens.pop(0)

            right = tokens[0]

            if not right.isdigit():
                tree.append(RomanNumeral(tokens[0]))
                if operator == '-':
                    tree.append(Subtract(left_expression, tree[-1]))
                if operator == '+':
                    tree.append(Add(left_expression, tree[-1]))
            else:
                right_expression = Number(right)
                if not tree:
                    # Empty Data Structures return False by default
                    if operator == '-':
                        tree.append(
                            Subtract(left_expression, right_expression))
                    if operator == '+':
                        tree.append(
                            Add(left_expression, right_expression))
                else:
                    if operator == '-':
                        tree.append(Subtract(tree[-1], right_expression))
                    if operator == '+':
                        tree.append(Add(tree[-1], right_expression))

        return tree.pop()

    @staticmethod
    def decide_left_expression(tree, tokens):
        """
        On the First iteration, the left expression can be either a
        number or roman numeral. Every consecutive expression is
        reference to an existing AST row
        """
        left = tokens.pop(0)
        left_expression = None
        if not tree:  # only applicable if first round
            if not left.isdigit():  # if 1st token a roman numeral
                tree.append(RomanNumeral(left))
                left_expression = tree[-1]
            else:
                left_expression = Number(left)
        else:
            left_expression = tree[-1]
        return left_expression
```

7.4.9 Output

```
python ./interpreter/client.py
5 + IV - 3 + VII - 2
['5', '+', 'IV', '-', '3', '+', 'VII', '-', '2']
11
((((5 Add IV(4)) Subtract 3) Add VII(7)) Subtract 2)
```

7.4.10 New Coding Concepts

String Slicing

SBCODE Video ID #e13190

Sometimes you want part of a string. In the example code, when I am interpreting the roman numerals, I am comparing the first one or two characters in the context with `IV` or `CM` or many other roman numeral combinations. If the match is true then I continue with further commands.

The format is

```
string[start: end: step]
```

E.g., the string may be

```
MMMCMXCIX
```

and I want the first three characters,

```
test = "MMMCMXCIX"
print(test[0: 3])
```

Outputs

```
MMM
```

or I want the last 4 characters

```
test = "MMMCMXCIX"
print(test[-4:])
```

Outputs

```
XCIX
```

or I want a section in the middle

```
test = "MMMCMXCIX"
print(test[2:5])
```

Outputs

```
MCM
```

or stepped

```
test = "MMMCMXCIX"
print(test[2:9:2])
```

Outputs

```
MMCX
```

or even reversed

```
test = "MMMCMXCIX"
print(test[::-1])
```

Outputs

```
XICXMCMMM
```

The technique is very common in examples of Python source code throughout the internet. So, when you see the `[]` with numbers and colons inside, eg, `[:-1:]` , it is likely to do with extracting a portion of a data structure.

Note that the technique also works on Lists and Tuples.

```
test = [1,2,3,4,5,6,7,8,9]
print(test[0: 3])
print(test[-4:])
print(test[2:5])
print(test[2:9:2])
print(test[::-1])
print(test[:-1:])
```

Outputs

```
[1, 2, 3]
[6, 7, 8, 9]
[3, 4, 5]
[3, 5, 7, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1]
[1, 2, 3, 4, 5, 6, 7, 8]
```

7.4.11 Summary

- ASTs are hard to create and are an enormous subject in themselves. My recommended approach is to create them manually first using a sample sentence to help understand all the steps individually, and then progress the conversion to be fully dynamic one step at a time ensuring that the grammatical constructs still work as you continue to progress.
- The Interpreter pattern uses a class to represent each grammatical rule.
- ASTs consist of multiple Non-Terminal and Terminal Expressions, that all implement an `interpret()` method.
- Note that in the sample code above, the `interpret()` methods in the Non-Terminal expressions, all call further `interpret()` recursively. Only the Terminal expressions `interpret()` method returns an explicit value. See the Number class in the above code.

7.5 Iterator Design Pattern

7.5.1 Overview

SBCODE Video ID #d072b5

The Iterator will commonly contain two methods that perform the following concepts.

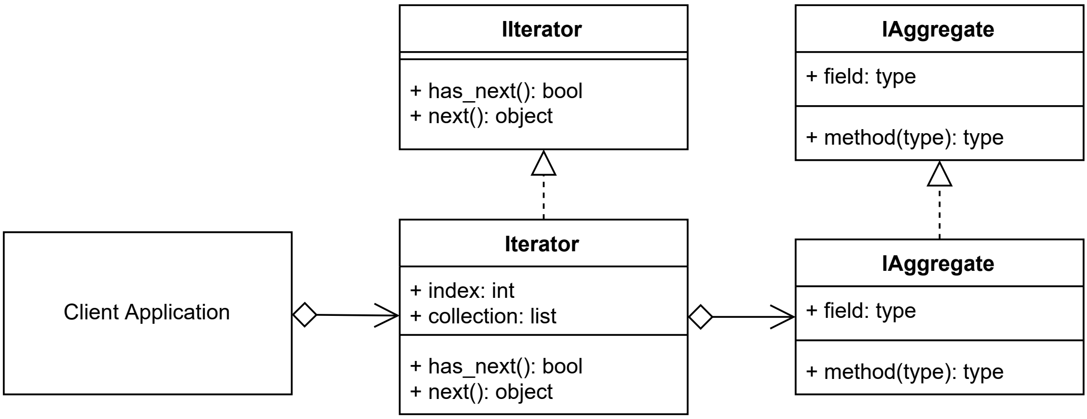
- **next**: returns the next object in the aggregate (collection, object).
- **has_next**: returns a Boolean indicating if the Iterable is at the end of the iteration or not.

The benefits of using the Iterator pattern are that the client can traverse a collection of aggregates(objects) without needing to understand their internal representations and/or data structures.

7.5.2 Terminology

- **Iterator Interface**: The Interface for an object to implement.
- **Concrete Iterator**: (Iterable) The instantiated object that implements the iterator and contains a collection of aggregates.
- **Aggregate Interface**: An interface for defining an aggregate (object).
- **Concrete Aggregate**: The object that implements the Aggregate interface.

7.5.3 Iterator UML Diagram



7.5.4 Source Code

In this concept example, I create 4 objects called Aggregate and group them into a collection.

They are very minimal objects that implement one method that prints a line.

I then create an Iterable and pass in the collection of Aggregates.

I can now traverse the aggregates through the Iterable interface.

./iterator/iterator_concept.py

```python
# pylint: disable=too-few-public-methods
"The Iterator Pattern Concept"
from abc import ABCMeta, abstractmethod

class IIterator(metaclass=ABCMeta):
    "An Iterator Interface"
    @staticmethod
    @abstractmethod
    def has_next():
        "Returns Boolean whether at end of collection or not"

    @staticmethod
    @abstractmethod
    def next():
        "Return the object in collection"

class Iterable(IIterator):
    "The concrete iterator (iterable)"

    def __init__(self, aggregates):
        self.index = 0
        self.aggregates = aggregates

    def next(self):
        if self.index < len(self.aggregates):
            aggregate = self.aggregates[self.index]
            self.index += 1
            return aggregate
        raise Exception("AtEndOfIteratorException", "At End of Iterator")

    def has_next(self):
        return self.index < len(self.aggregates)

class IAggregate(metaclass=ABCMeta):
    "An interface that the aggregates should implement"
    @staticmethod
    @abstractmethod
    def method():
        "a method to implement"
```

```
class Aggregate(IAggregate):
    "A concrete object"
    @staticmethod
    def method():
        print("This method has been invoked")

# The Client
AGGREGATES = [Aggregate(), Aggregate(), Aggregate(), Aggregate()]
# AGGREGATES is a python list that is already iterable by default.

# but we can create own own iterator on top anyway.
ITERABLE = Iterable(AGGREGATES)

while ITERABLE.has_next():
    ITERABLE.next().method()
```

7.5.5 Output

```
python ./iterator/iterator_concept.py
This method has been invoked
This method has been invoked
This method has been invoked
This method has been invoked
```

7.5.6 Iterator Use Case

SBCODE Video ID #e39a6b

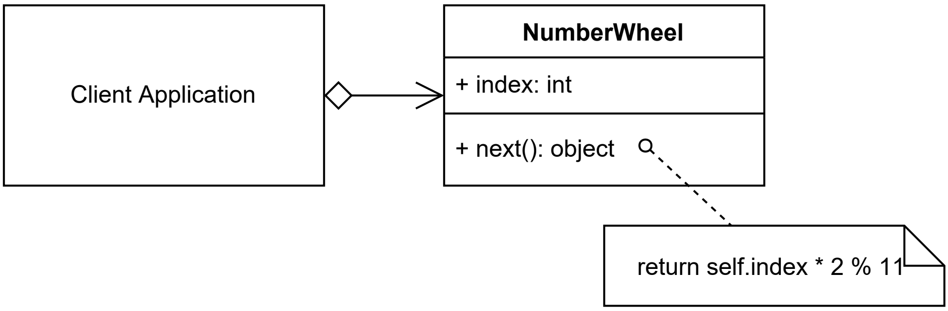
One reason for not using the inbuilt Python data structures that implement iterators already, or using the iter function directly over an existing collection, is in the case when you want to create an object that can dynamically create iterated objects, you want a custom order of objects or an infinite iterator.

The iterator in this brief example will return the next number in the iterator multiplied by 2 modulus 11. It dynamically creates the returned object (number) at runtime.

It has no `has_next()` method since the result is modulated by 11, that will loop the results no matter how large the iterator index is. It will also appear to alternate between a series of even numbers and odd numbers.

Also, just to demonstrate that implementing abstract classes and interfaces is not always necessary, this example uses no abstract base classes or interfaces.

7.5.7 Example UML Diagram



7.5.8 Source Code

./builder/client.py

```
"The Iterator Pattern Concept"


class NumberWheel():  # pylint: disable=too-few-public-methods
    "The concrete iterator (iterable)"

    def __init__(self):
        self.index = 0

    def next(self):
        """Return a new number next in the wheel"""
        self.index = self.index + 1
        return self.index * 2 % 11


# The Client
NUMBERWHEEL = NumberWheel()

for i in range(22):
    print(NUMBERWHEEL.next(), end=", ")
```

7.5.9 Output

```
python ./iterator/client.py
2, 4, 6, 8, 10, 1, 3, 5, 7, 9, 0, 2, 4, 6, 8, 10, 1, 3, 5, 7, 9, 0,
```

7.5.10 New Coding Concepts

Python iter()

SBCODE Video ID #86db0f

Python Lists, Dictionaries, Sets and Tuples are already iterable, so if you want basic iteration for use in a for loop, then you only need to add your objects into one of those and it can be used right away.

```
NAMES = ['SEAN','COSMO','EMMY']
for name in NAMES:
    print(name, end=", ")
#SEAN, COSMO, EMMY,
```

also, you can instantiate an iterable from the List, Dictionary, Tuple or Set by using the Python iter() method, or its own `__iter__()` dunder method, and then iterate over that using the `__next__()` method.

```
NAMES = ['SEAN','COSMO','EMMY']
ITERATOR = iter(NAMES)
print(ITERATOR.__next__())
print(ITERATOR.__next__())
print(ITERATOR.__next__())
```

or

```
NAMES = ['SEAN','COSMO','EMMY']
ITERATOR = NAMES.__iter__()
print(ITERATOR.__next__())
print(ITERATOR.__next__())
print(ITERATOR.__next__())
```

The Python `iter()` method also can accept a `sentinel` parameter.

The `sentinel` parameter is useful for dynamically created objects that are returned from an iterator and indicates where the last item is in the iterator by raising a `StopIteration` exception.

Usage : `iter(object, sentinel)`

When using the `sentinel` , the object passed as the first argument in the `iter()` method will need to be callable.

```
class Doubler():
    count = 1
    @classmethod
    def next(cls):
        cls.count *= 2
        return cls.count

    __call__ = next

ITERATOR = iter(Doubler(), 32)
print(list(ITERATOR))
# Outputs [2, 4, 8, 16]
```

The `__call__ = next` line in the example above is setting the default method of the class to be `next` and that makes the class callable. See Dunder **call** Method for more information.

Also note that the list being printed at the end is automatically filled from the iterator. The list constructor utilizes the default callable method and the `StopIteration` exception automatically during its creation without needing to write this in the code.

7.5.11 Summary

- There are many ways to create Iterators. They are already built into Python and used instead of creating custom classes.
- Use an iterator when you need to traverse over a collection, or you want an object that can output a series of dynamically created objects.
- At minimum, an iterator needs a `next` equivalent method that returns an object.
- Optionally you can also create a helper function that indicates whether an iterator is at the end or not. This is useful if you use your iterator in a `while` loop.
- Alternatively, use the `sentinel` option of the Python `iter()` method to indicate the last iteration. Note that the Iterator object needs to be callable. Set the `__call__` reference to its `next` method.

7.6 Mediator Design Pattern

7.6.1 Overview

SBCODE Video ID #d0089f

Objects communicate through the **Mediator** rather than directly with each other.

As a system evolves and becomes larger and supports more complex functionality and business rules, the problem of communicating between these components becomes more complicated to understand and manage. It may be beneficial to refactor your system to centralize some or all of its functionality via some kind of mediation process.

The mediator pattern is similar to creating a Facade pattern between your classes and processes. Except the Mediator is expected to transact data both ways between two or more other classes or processes that would normally interact directly with each other.

The resulting Mediator interface will be very custom to the use cases that it is now supporting.

The Mediator will generally look like an object that is managing one of more Observer patterns perhaps between the other classes or processes (colleagues). Whether you use an Observer pattern to manage a particular piece of functionality or not depends on whether it is the best use of the resources you have available.

When refactoring your code, you may decide to approach your refactoring from the perspective of implementing an Observer pattern first. This means that all colleagues (Observers) will receive the notification whether it was intended for them or not. If you want to avoid redundant updates in the colleagues then you can write specific cases in your code, or create specific methods as I have done in mediator_concept.py example below.

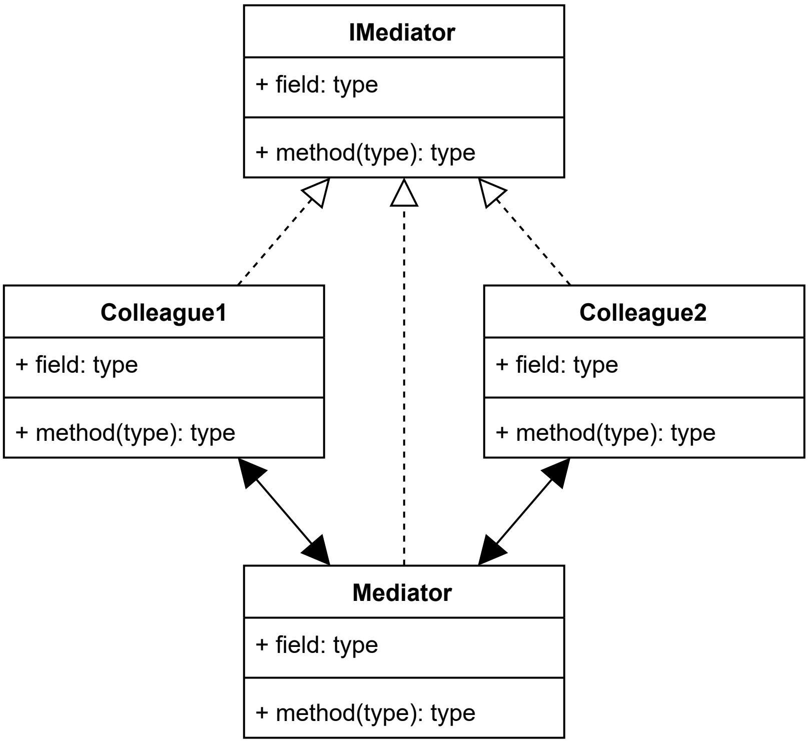
Colleagues now will send and receive requests via a Mediator object rather than directly between each other. The Mediator is like a router in this case, but allows you to add extra programmatic functionality and also give the option of creating other kinds of colleagues that could utilize the communications in new ways.

7.6.2 Terminology

- **Mediator Interface**: An interface that the Mediator and Colleagues implement. Note that different Colleagues will have varied use cases and won't need to implement all the methods described in the Mediator interface.
- **Concrete Mediator**: The single source of truth and coordinator of communications between the components (colleagues).

- **Colleague Classes**: One of the many types of concrete components that use the mediator for its own particular use case.

7.6.3 Mediator UML Diagram



7.6.4 Source Code

In the example concept, there are two colleague classes that use each other's methods. Instead of the Colleagues calling each other's methods directly, they implement the Mediator interface and call each other via the Mediator. Each colleague class or process is designed for a different purpose, but they utilize some related functionality from each other.

The system would work without the Mediator, but adding the Mediator will allow extending functionality to a potential third colleague that provides a different service, such as AI analysis or monitoring, without needing to add specific support or knowledge into the two original colleagues.

./mediator/mediator_concept.py

```
"Mediator Concept Sample Code"
from abc import ABCMeta, abstractmethod

class IMediator(metaclass=ABCMeta):
    "The Mediator interface indicating all the methods to implement"
    @staticmethod
    @abstractmethod
    def colleague1_method():
        "A method to implement"

    @staticmethod
    @abstractmethod
    def colleague2_method():
        "A method to implement"

class Mediator(IMediator):
    "The Mediator Concrete Class"

    def __init__(self):
        self.colleague1 = Colleague1()
        self.colleague2 = Colleague2()

    def colleague1_method(self):
        return self.colleague1.colleague1_method()

    def colleague2_method(self):
        return self.colleague2.colleague2_method()

class Colleague1(IMediator):
    "This Colleague calls the other Colleague via the Mediator"

    def colleague1_method(self):
        return "Here is the Colleague1 specific data you asked for"

    def colleague2_method(self):
        "not implemented"

class Colleague2(IMediator):
    "This Colleague calls the other Colleague via the Mediator"

    def colleague1_method(self):
        "not implemented"

    def colleague2_method(self):
        return "Here is the Colleague2 specific data you asked for"

# This Client is either Colleague1 or Colleague2
# This Colleague will instantiate a Mediator, rather than calling
# the other Colleague directly.
MEDIATOR = Mediator()

# If I am Colleague1, I want some data from Colleague2
DATA = MEDIATOR.colleague2_method()
print(f"COLLEAGUE1 <--> {DATA}")
```

```
# If I am Colleague2, I want some data from Colleague1
DATA = MEDIATOR.colleague1_method()
print(f"COLLEAGUE2 <--> {DATA}")
```

7.6.5 Output

```
python ./mediator/mediator_concept.py
COLLEAGUE1 <--> Here is the Colleague2 specific data you asked for
COLLEAGUE2 <--> Here is the Colleague1 specific data you asked for
```

7.6.6 Mediator Use Case

SBCODE Video ID #4429bf

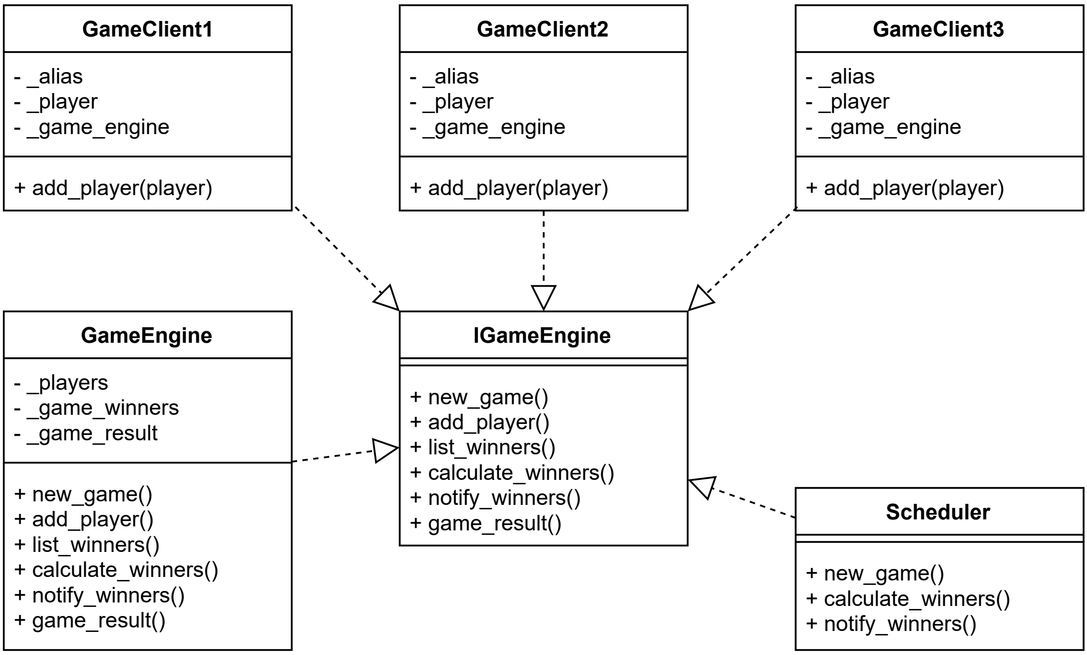
This is a simplified game engine. There is the main game engine component, a scheduler that manages game events and there are game clients that act as separate game players submitting bets into a game round.

All of the components implement the Mediators interface. They all implement one or some of the methods differently depending on their purpose. While they all perform different types of functionality, they all require a single source of truth being the Game Engine that acts as the Mediator.

There is mixture of this Mediator example using the Observer pattern to notify the game clients, as well as specific methods not necessarily shared between the scheduler, game engine and clients but benefits from being managed via the Mediator.

Normally the processes being mediated will be running from different servers or programs, but in this example they are all part of the same client in order to demonstrate the concept easier.

7.6.7 Example UML Diagram



7.6.8 Source Code

./mediator/client.py

```
"Mediator Pattern Example Code"

from game_engine import GameEngine
from game_client import GameClient
from player import Player
from scheduler import Scheduler

# The concrete GameEngine process that would run on a dedicated server
GAMEENGINE = GameEngine()

# 3 Hypothetical game clients, all running externally on mobile phones
# calling the GAMEENGINE mediator across a network proxy
MOBILECLIENT1 = GameClient(GAMEENGINE)
PLAYER1 = Player("Sean", 100)
MOBILECLIENT1.add_player(PLAYER1)

MOBILECLIENT2 = GameClient(GAMEENGINE)
PLAYER2 = Player("Cosmo", 200)
MOBILECLIENT2.add_player(PLAYER2)

MOBILECLIENT3 = GameClient(GAMEENGINE)
```

```python
PLAYER3 = Player("Emmy", 300, )
MOBILECLIENT3.add_player(PLAYER3)

# A scheduler is a separate process that manages game rounds and
# triggers game events at time intervals
SCHEDULER = Scheduler(GAMEENGINE)
SCHEDULER.new_game()

# 3 Hypothetical game clients have received notifications of new game,
# they now place there bets
PLAYER1.place_bets([1, 2, 3])
PLAYER2.place_bets([5, 6, 7, 8])
PLAYER3.place_bets([10, 11])

# The scheduler closes the round, and triggers the result and
# winnings notifications
SCHEDULER.calculate_winners()
SCHEDULER.notify_winners()
```

./mediator/interface_game_engine.py

```python
"The Game Client Mediator Interface"
from abc import ABCMeta, abstractmethod

class IGameEngine(metaclass=ABCMeta):
    "The Game Client Interface"
    @staticmethod
    @abstractmethod
    def new_game():
        "A method to implement"

    @staticmethod
    @abstractmethod
    def add_player(player):
        "A method to implement"

    @staticmethod
    @abstractmethod
    def list_winners():
        "A method to implement"

    @staticmethod
    @abstractmethod
    def calculate_winners():
        "A method to implement"

    @staticmethod
    @abstractmethod
    def notify_winners():
        "A method to implement"

    @staticmethod
    @abstractmethod
    def game_result():
        "A method to implement"
```

./mediator/game_engine.py

```
"A Game Engine which is the mediator"
import random
from interface_game_engine import IGameEngine

class GameEngine(IGameEngine):
    "The Game Engine"

    def __init__(self):
        self._players = {}
        self._game_winners = {}
        self._game_result = -1

    def new_game(self):
        for alias in self._players:
            self._players[alias].notify(
                f"{alias} -> New Game, Place Bets")

    def add_player(self, player):
        self._players[player.alias] = player

    def list_winners(self):
        ret = []
        for key in self._players:
            ret.append([key, self._players[key].last_winnings])
        return ret

    def calculate_winners(self):
        self._game_result = random.randint(0, 12)
        for alias in self._players:
            player = self._players[alias]
            if self._game_result in player.bets:
                player.balance = player.balance + 12
                player.last_winnings = 12
                self._game_winners[alias] = 12

    def notify_winners(self):
        for alias in self._players:
            if alias in self._game_winners:
                self._players[alias].notify(
                    f"{alias} -> You Are The Winner with result "
                    f"`{self._game_result}`."
                    f" You Won {self._players[alias].last_winnings}."
                    f" Your balance = {self._players[alias].balance}"
                )

    def game_result(self):
        return self._game_result
```

./mediator/game_client.py

```
"A Game Client"
from interface_game_engine import IGameEngine

class GameClient(IGameEngine):
    "A Game Client that implements some of the mediator methods"

    def __init__(self, game_engine):
        self._alias = ""
        self._player = {}
        self._game_engine = game_engine

    def new_game(self):
        "not implemented in the game client"

    def add_player(self, player):
        self._player = player
        self._game_engine.add_player(player)

    def list_winners(self):
        "not implemented in the game client"

    def calculate_winners(self):
        "not implemented in the game client"

    def notify_winners(self):
        "not implemented in the game client"

    def game_result(self):
        "not implemented in the game client"
```

./mediator/scheduler.py

```
"A Scheduler"
from interface_game_engine import IGameEngine
```

```
class Scheduler(IGameEngine):
    "The scheduler implements some of the Mediator methods"

    def __init__(self, game_engine):
        self._game_engine = game_engine

    def new_game(self):
        self._game_engine.new_game()

    def add_player(self, player):
        "not implemented in the scheduler"

    def list_winners(self):
        "not implemented in the scheduler"

    def calculate_winners(self):
        self._game_engine.calculate_winners()

    def notify_winners(self):
        self._game_engine.notify_winners()

    def game_result(self):
        "not implemented in the game client"
```

./mediator/player.py

```
"A Player Object"
class Player():
    "A Player Object"

    def __init__(self, alias, balance):
        self.alias = alias
        self.balance = balance
        self.bets = []
        self.last_winnings = 0

    def place_bets(self, bets):
        "When a player places bets, its balance is deducted"
        for _ in bets:
            self.balance -= 1
        self.bets = bets

    @staticmethod
    def notify(message):
        "The player is an observer of the game"
        print(message)
```

7.6.9 Output

```
python.exe ./mediator/client.py
Sean -> New Game, Place Bets
Cosmo -> New Game, Place Bets
Emmy -> New Game, Place Bets
Emmy -> You Are The Winner with result `10`. You Won 12. Your balance =
310
```

7.6.10 New Coding Concepts

The Underscore Only _ Variable

SBCODE Video ID #76bf69

In the Player class, there is a for loop that iterates the bets list.

```
for _ in bets:
    self.balance -= 1
```

The `_` is used instead of a more tradition `i`, `x`, `y` or another variable. If using a letter in the for loop, and not actually using it, like in the above example, then Pylint would indicate a warning of unused variable. Note that the Python interpreter would still run this code ok if using a letter as the variable name, but reducing Pylint warnings helps makes code look neater.

E.g.,

```
for i in bets:
    self.balance -= 1
```

The Pylint warning would be

```
W0612: Unused variable 'i' (unused-variable)
```

So, using the `_` prevents this warning.

7.6.11 Summary

- A mediator replaces a structure with many-to-many interactions between its classes and processes, with a one-to-many centralized structure where the interface supports all of the methods of the many-to-many structure, but via the mediator component instead.
- The mediator pattern encourages usage of shared objects that can now be centrally managed and synchronized.

- The mediator pattern creates an abstraction between two or more components that then makes a system easier to understand and manage.
- The mediator pattern is similar to the Facade pattern, except the Mediator is expected to transact data both ways between two or more other classes or processes that would normally interact directly with each other.

7.7 Memento Design Pattern

7.7.1 Overview

SBCODE Video ID #a37ae3

Throughout the lifecycle of an application, an objects state may change. You might want to store a copy of the current state in case of later retrieval. E.g., when writing a document, you may want to auto save the current state every 10 minutes. Or you have a game, and you want to save the current position of your player in the level, with its score and current inventory.

You can use the **Memento** pattern for saving a copy of state and for later retrieval if necessary.

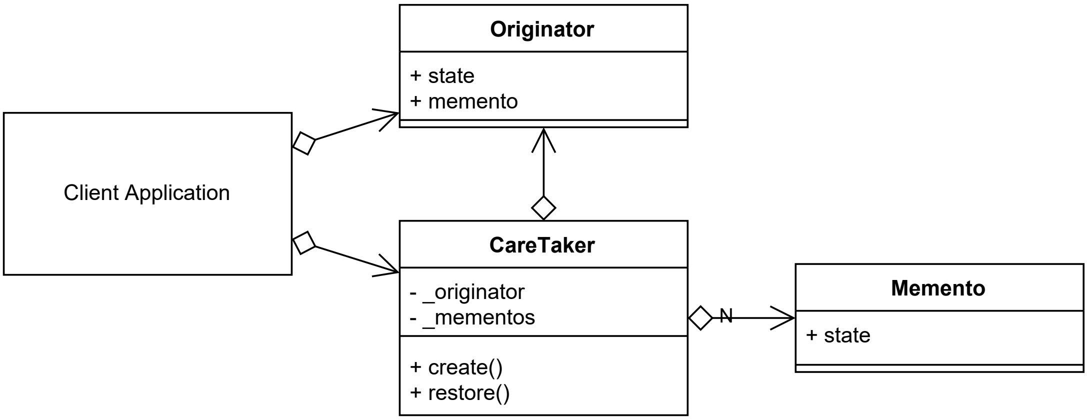
The Memento pattern, like the Command pattern, is also commonly used for implementing UNDO/REDO functionality within your application.

The difference between the Command and the Memento patterns for UNDO/REDO, is that in the Command pattern, you re-execute commands in the same order that changed attributes of a state, and with the Memento, you completely replace the state by retrieving from a cache/store.

7.7.2 Terminology

- **Originator**: The originator is an object with an internal state that changes on occasion.
- **Caretaker**: (Guardian) A Class that asks the Originator to create or restore Mementos. The Caretaker than saves them into a cache or store of mementos.
- **Memento**: A copy of the internal state of the Originator that can later be restored back into the Originator to replace its current state.

7.7.3 Memento UML Diagram



7.7.4 Source Code

In the concept code, the client creates an object whose state will be periodically recorded. The object will be the Originator.

A Caretaker is also created with a reference to the Originator.

The Originators internal state is changed several times. It is then decided that the Caretaker should make a backup.

More changes are made to the Originator, and then another backup is made.

More changes are made to the Originator, and then it is decided that the first backup should be restored instead.

And then the second backup is restored.

./memento/memento_concept.py

```
"Memento pattern concept"


class Memento():  # pylint: disable=too-few-public-methods
    "A container of state"

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


class Originator():
    "The Object in the application whose state changes"

    def __init__(self):
        self._state = ""

    @property
    def state(self):
        "A `getter` for the objects state"
        return self._state

    @state.setter
    def state(self, state):
        print(f"Originator: Setting state to `{state}`")
        self._state = state

    @property
    def memento(self):
        "A `getter` for the objects state but packaged as a Memento"
        print("Originator: Providing Memento of state to caretaker.")
        return Memento(self._state)

    @memento.setter
    def memento(self, memento):
        self._state = memento.state
        print(
            f"Originator: State after restoring from Memento: "
            f"`{self._state}`")


class CareTaker():
    "Guardian. Provides a narrow interface to the mementos"

    def __init__(self, originator):
        self._originator = originator
        self._mementos = []

    def create(self):
        "Store a new Memento of the Originators current state"
        print("CareTaker: Getting a copy of Originators current state")
        memento = self._originator.memento
        self._mementos.append(memento)

    def restore(self, index):
        """
        Replace the Originators current state with the state
        stored in the saved Memento
        """
        print("CareTaker: Restoring Originators state from Memento")
        memento = self._mementos[index]
        self._originator.memento = memento


# The Client
ORIGINATOR = Originator()
CARETAKER = CareTaker(ORIGINATOR)

# originators state can change periodically due to application events
ORIGINATOR.state = "State #1"
ORIGINATOR.state = "State #2"

# lets backup the originators
CARETAKER.create()

# more changes, and then another backup
ORIGINATOR.state = "State #3"
CARETAKER.create()
```

```
# more changes
ORIGINATOR.state = "State #4"
print(ORIGINATOR.state)

# restore from first backup
CARETAKER.restore(0)
print(ORIGINATOR.state)

# restore from second backup
CARETAKER.restore(1)
print(ORIGINATOR.state)
```

Output

```
python ./memento/memento_concept.py
Originator: Setting state to `State #1`
Originator: Setting state to `State #2`
CareTaker: Getting a copy of Originators current state
Originator: Providing Memento of state to caretaker.
Originator: Setting state to `State #3`
CareTaker: Getting a copy of Originators current state
Originator: Providing Memento of state to caretaker.
Originator: Setting state to `State #4`
State #4
CareTaker: Restoring Originators state from Memento
Originator: State after restoring from Memento: `State #2`
State #2
CareTaker: Restoring Originators state from Memento
Originator: State after restoring from Memento: `State #3`
State #3
```

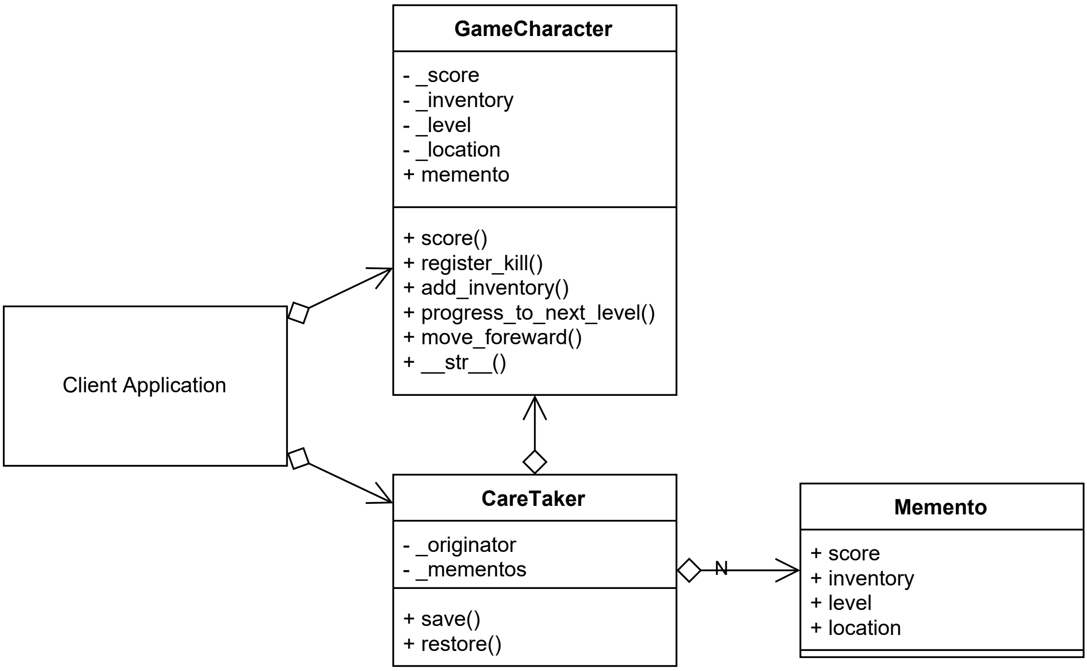
7.7.5 Memento Use Case

SBCODE Video ID #0a7255

There is a game, and the character is progressing through the levels. It has acquired several new items in its inventory, the score is very good and you want to save your progress and continue later.

You then decide you made a mistake and need to go back to a previous save because you took a wrong turn.

7.7.6 Example UML Diagram



7.7.7 Source Code

./memento/client.py

```
"Memento example Use Case"

from game_character import GameCharacter
from caretaker import CareTaker

GAME_CHARACTER = GameCharacter()
CARETAKER = CareTaker(GAME_CHARACTER)

# start the game
GAME_CHARACTER.register_kill()
GAME_CHARACTER.move_forward(1)
GAME_CHARACTER.add_inventory("sword")
GAME_CHARACTER.register_kill()
GAME_CHARACTER.add_inventory("rifle")
GAME_CHARACTER.move_forward(1)
print(GAME_CHARACTER)

# save progress
CARETAKER.save()

GAME_CHARACTER.register_kill()
GAME_CHARACTER.move_forward(1)
GAME_CHARACTER.progress_to_next_level()
GAME_CHARACTER.register_kill()
GAME_CHARACTER.add_inventory("motorbike")
GAME_CHARACTER.move_forward(10)
GAME_CHARACTER.register_kill()
print(GAME_CHARACTER)

# save progress
CARETAKER.save()
GAME_CHARACTER.move_forward(1)
GAME_CHARACTER.progress_to_next_level()
GAME_CHARACTER.register_kill()
print(GAME_CHARACTER)

# decide you made a mistake, go back to first save
CARETAKER.restore(0)
print(GAME_CHARACTER)

# continue
GAME_CHARACTER.register_kill()
```

./memento/game_character.py

```
"The Game Character whose state changes"
from memento import Memento

class GameCharacter():
    "The Game Character whose state changes"

    def __init__(self):
        self._score = 0
        self._inventory = set()
        self._level = 0
        self._location = {"x": 0, "y": 0, "z": 0}

    @property
    def score(self):
        "A `getter` for the objects score"
        return self._score

    def register_kill(self):
        "The character kills its enemies as it progesses"
        self._score += 100

    def add_inventory(self, item):
        "The character finds objects in the game"
        self._inventory.add(item)
    def progress_to_next_level(self):
        "The characer progresses to the next level"
        self._level += 1

    def move_forward(self, amount):
        "The character moves around the environment"
        self._location["z"] += amount

    def __str__(self):
        return(
            f"Score: {self._score}, "
            f"Level: {self._level}, "
            f"Location: {self._location}\n"
            f"Inventory: {self._inventory}\n"
        )

    @ property
    def memento(self):
        "A `getter` for the characters attributes as a Memento"
        return Memento(
            self._score,
            self._inventory.copy(),
            self._level,
            self._location.copy())

    @ memento.setter
    def memento(self, memento):
        self._score = memento.score
        self._inventory = memento.inventory
        self._level = memento.level
        self._location = memento.location
```

./memento/caretaker.py

```
"The Save/Restore Game functionality"

class CareTaker():
    "Guardian. Provides a narrow interface to the mementos"

    def __init__(self, originator):
        self._originator = originator
        self._mementos = []

    def save(self):
        "Store a new Memento of the Characters current state"
        print("CareTaker: Game Save")
        memento = self._originator.memento
        self._mementos.append(memento)

    def restore(self, index):
        """
        Replace the Characters current attributes with the state
        stored in the saved Memento
        """
        print("CareTaker: Restoring Characters attributes from Memento")
        memento = self._mementos[index]
        self._originator.memento = memento
```

./memento/memento.py

```
"A Memento to store character attributes"

class Memento():  # pylint: disable=too-few-public-methods
    "A container of characters attributes"

    def __init__(self, score, inventory, level, location):
        self.score = score
        self.inventory = inventory
        self.level = level
        self.location = location
```

7.7.8 Output

```
python ./memento/client.py
Score: 200, Level: 0, Location: {'x': 0, 'y': 0, 'z': 2}
Inventory: {'rifle', 'sword'}

CareTaker: Game Save
Score: 500, Level: 1, Location: {'x': 0, 'y': 0, 'z': 13}
Inventory: {'motorbike', 'rifle', 'sword'}

CareTaker: Game Save
Score: 600, Level: 2, Location: {'x': 0, 'y': 0, 'z': 14}
Inventory: {'motorbike', 'rifle', 'sword'}

CareTaker: Restoring Characters attributes from Memento
Score: 200, Level: 0, Location: {'x': 0, 'y': 0, 'z': 2}
Inventory: {'rifle', 'sword'}
```

7.7.9 New Coding Concepts

Python Getter/Setters

SBCODE Video ID #a55e85

Often when coding attributes in classes, you may want to provide methods to allow external functions to read or modify a classes internal attributes.

A common approach would be to add two methods prefixed with `get_` and `set_` ,

```
class ExampleClass:
    def __init__(self):
        self._value = 123

    def get_value(self):
        return self._value

    def set_value(self, value):
        self._value = value

example = ExampleClass()
print(example.get_value())
```

This makes perfect sense what the intentions are, but there is a more pythonic way of doing this and that is by using the inbuilt Python `@property` decorator.

```
class ExampleClass:
    def __init__(self):
        self._value = 123

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, value):
        self._value = value

example = ExampleClass()
print(example.value)
```

Note that in the above example, there is an extra decorator named `@value.setter` . This is used for setting the `_value` attribute.

Along with the above two new getter/setter methods, there is also another method for deleting an attribute called `deleter` .

```
class ExampleClass:
    def __init__(self):
        self._value = 123

    @property
    def value(self):
        return self._value

    @value.setter
    def value(self, value):
        self._value = value

    @value.deleter
    def value(self):
        print('Deleting _value')
        del self._value

example = ExampleClass()
print(example.value)
del example.value
print(example.value) # now raises an AttributeError
```

7.7.10 Summary

- You don't need to create a new Memento each time an Originators state changes. You can do it only when considered necessary. E.g., an occasional backup to a file.
- Mementos can be stored in memory or saved/cached externally. The Caretaker will abstract the complications of storing and retrieving Mementos from the Originator.
- Consider the Command pattern for fine grained changes to an objects state to manage UNDO/ REDO between memento saves. Or even save command history into a Memento that can be later replayed.
- In my examples, the whole state is recorded and changed with the Memento. You can use the Memento to record and change partial states instead if required.
- When copying state, be aware of shallow/deep copying. In complicated projects, your restore functionality will probably contain a combination of both the Command and Memento patterns.

7.8 State Design Pattern

7.8.1 Overview

SBCODE Video ID #11b6da

Not to be confused with object state, i.e., one of more attributes that can be copied as a snapshot, the **State Pattern** is more concerned about changing the handle of an object's method dynamically. This makes an object itself more dynamic and may reduce the need of many conditional statements.

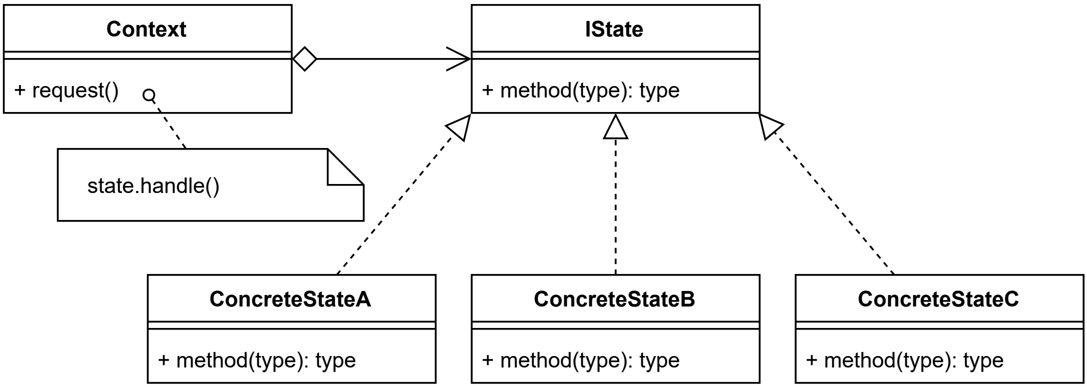
Instead of storing a value in an attribute, and then then using conditional statements within an objects method to produce different output, a subclass is assigned as a handle instead. The object/ context doesn't need to know about the inner working of the assigned subclass that the task was delegated to.

In the state pattern, the behavior of an objects state is encapsulated within the subclasses that are dynamically assigned to handle it.

7.8.2 Terminology

- **State Interface**: An interface for encapsulating the behavior associated with a particular state of the Context.
- **Concrete Subclasses**: Each subclass implements a behavior associated with the particular state.
- **Context**: This is the object where the state is defined, but the execution of the state behavior is redirected to the concrete subclass.

7.8.3 State UML Diagram



7.8.4 Source Code

In the concept example, there are three possible states. Every time the `request()` method is called, the concrete state subclass is randomly selected by the context.

./state/state_concept.py

```python
# pylint: disable=too-few-public-methods
"The State Pattern Concept"
from abc import ABCMeta, abstractmethod
import random

class Context():
    "This is the object whose behavior will change"

    def __init__(self):
        self.state_handles = [ConcreteStateA(),
                              ConcreteStateB(),
                              ConcreteStateC()]
        self.handle = None

    def request(self):
        """A method of the state that dynamically changes which
        class it uses depending on the value of self.handle"""
        self.handle = self.state_handles[random.randint(0, 2)]
        return self.handle

class IState(metaclass=ABCMeta):
    "A State Interface"

    @staticmethod
    @abstractmethod
    def __str__():
        "Set the default method"

class ConcreteStateA(IState):
    "A ConcreteState Subclass"

    def __str__(self):
        return "I am ConcreteStateA"

class ConcreteStateB(IState):
    "A ConcreteState Subclass"

    def __str__(self):
        return "I am ConcreteStateB"

class ConcreteStateC(IState):
    "A ConcreteState Subclass"

    def __str__(self):
        return "I am ConcreteStateC"

# The Client
CONTEXT = Context()
print(CONTEXT.request())
print(CONTEXT.request())
print(CONTEXT.request())
print(CONTEXT.request())
print(CONTEXT.request())
```

Output

```
python.exe ./state/state_concept.py
I am ConcreteStateB
I am ConcreteStateA
I am ConcreteStateB
I am ConcreteStateA
I am ConcreteStateC
```

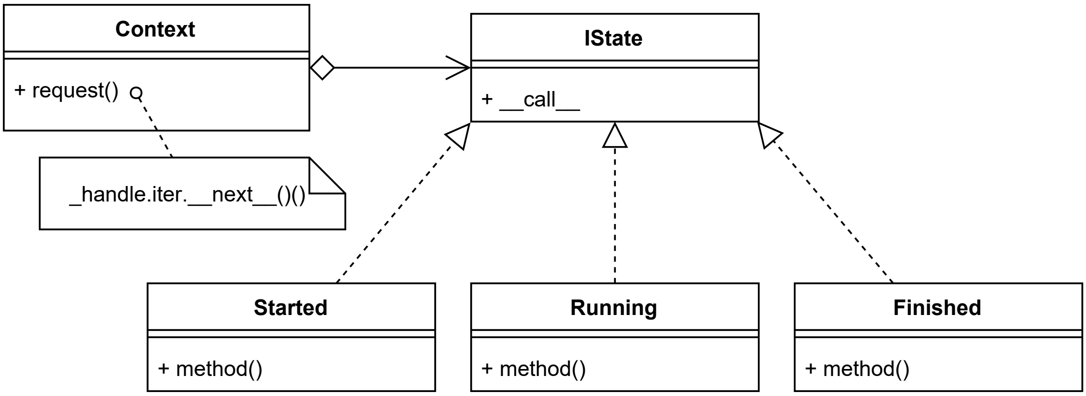
7.8.5 State Use Case

SBCODE Video ID #f5b4b7

This example takes the concept example further and uses an iterator rather than choosing the states subclasses randomly.

When the iterator gets to the end, it raises a `StopIteration` error and recreates the iterator so that the process can loop again.

7.8.6 State Example Use Case UML Diagram



7.8.7 Source Code

./state/client.py

```
# pylint: disable=too-few-public-methods
"The State Use Case Example"
from abc import ABCMeta, abstractmethod


class Context():
    "This is the object whose behavior will change"

    def __init__(self):

        self.state_handles = [
            Started(),
            Running(),
            Finished()
        ]
        self._handle = iter(self.state_handles)

    def request(self):
        "Each time the request is called, a new class will handle it"
        try:
            self._handle.__next__()()
        except StopIteration:
            # resetting so it loops
            self._handle = iter(self.state_handles)


class IState(metaclass=ABCMeta):
    "A State Interface"

    @staticmethod
    @abstractmethod
    def __call__():
        "Set the default method"


class Started(IState):
    "A ConcreteState Subclass"

    @staticmethod
    def method():
        "A task of this class"
        print("Task Started")

    __call__ = method
```

```
class Running(IState):
    "A ConcreteState Subclass"

    @staticmethod
    def method():
        "A task of this class"
        print("Task Running")

    __call__ = method


class Finished(IState):
    "A ConcreteState Subclass"

    @staticmethod
    def method():
        "A task of this class"
        print("Task Finished")

    __call__ = method


# The Client
CONTEXT = Context()
CONTEXT.request()
CONTEXT.request()
CONTEXT.request()
CONTEXT.request()
CONTEXT.request()
CONTEXT.request()
```

7.8.8 Output

```
python.exe ./state/client.py
Task Started
Task Running
Task Finished
Task Started
Task Running
```

7.8.9 New Coding Concepts

Dunder `__call__` Method

SBCODE Video ID #e0b1f0

Overloading the `__call__` method makes an instance of a class callable like a function when by default it isn't. You need to call a method within the class directly.

```
class ExampleClass:
    @staticmethod
    def do_this_by_default():
        print("doing this")

EXAMPLE = ExampleClass()
EXAMPLE.do_this_by_default() # needs to be explicitly called to execute
```

If you want a default method in your class, you can point to it using by the `__call__` method.

```
class ExampleClass:
    @staticmethod
    def do_this_by_default():
        print("doing this")

    __call__ = do_this_by_default

EXAMPLE = ExampleClass()
EXAMPLE() # function now gets called by default
```

7.8.10 Summary

- Makes an object change its behavior when its internal state changes.
- The client and the context are not concerned about the details of how the state is created/assembled/calculated. The client will call a method of the context and it will be handled by a subclass.
- The State pattern appears very similar to the Strategy pattern, except in the State pattern, the object/context has changed to a different state and will run a different subclass depending on that state.

7.9 Strategy Design Pattern

7.9.1 Overview

SBCODE Video ID #545946

The **Strategy** Pattern is similar to the State Pattern, except that the client passes in the algorithm that the context should run and the execution of the algorithm does not affect the state of the context.

The algorithm should be contained within a class that implements the particular strategies interface.

An application that sorts data is a good example of where you can incorporate the Strategy pattern.

There are many methods of sorting a set of data. E.g., Quicksort, Mergesort, Introsort, Heapsort, Bubblesort. See https://en.wikipedia.org/wiki/Sorting_algorithm for more examples.

The user interface of the client application can provide a drop-down menu to allow the user to try the different sorting algorithms.

Upon user selection, a reference to the algorithm will be passed to the context and processed using this new algorithm instead.

The Strategy and State appear very similar, a good way to differentiate them is to consider whether the context is considered to be in a new state or not at various times in the lifecycle.

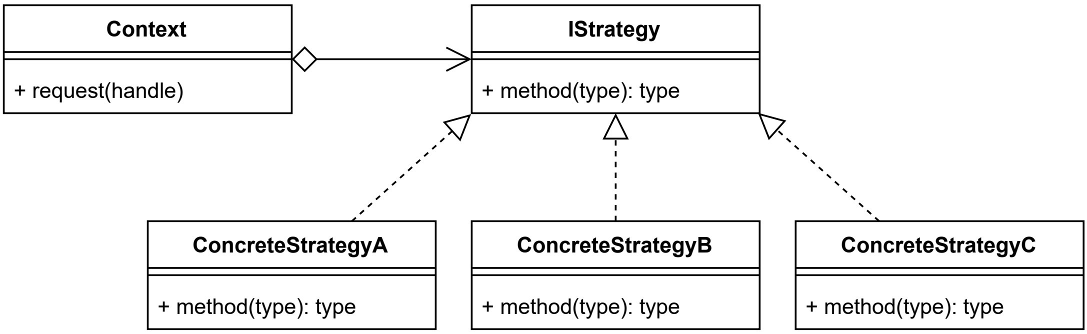
In the Strategy, an object/context runs a chosen algorithm, but the state of the object/context doesn't change in case you want to try a different algorithm.

Software Plugins can be implemented using the Strategy pattern.

7.9.2 Terminology

- **Strategy Interface**: An interface that all Strategy subclasses/algorithms must implement.
- **Concrete Strategy**: The subclass that implements an alternative algorithm.
- **Context**: This is the object that receives the concrete strategy in order to execute it.

7.9.3 Strategy UML Diagram



7.9.4 Source Code

There is a Context, and three different strategies to choose from.

Each Strategy is executed in turn by the context.

./strategy/strategy_concept.py

```
# pylint: disable=too-few-public-methods
"The Strategy Pattern Concept"
from abc import ABCMeta, abstractmethod

class Context():
    "This is the object whose behavior will change"

    @staticmethod
    def request(strategy):
        """The request is handled by the class passed in"""
        return strategy()

class IStrategy(metaclass=ABCMeta):
    "A strategy Interface"

    @staticmethod
    @abstractmethod
    def __str__():
        "Implement the __str__ dunder"

class ConcreteStrategyA(IStrategy):
    "A Concrete Strategy Subclass"

    def __str__(self):
        return "I am ConcreteStrategyA"
```

```
class ConcreteStrategyB(IStrategy):
    "A Concrete Strategy Subclass"

    def __str__(self):
        return "I am ConcreteStrategyB"

class ConcreteStrategyC(IStrategy):
    "A Concrete Strategy Subclass"

    def __str__(self):
        return "I am ConcreteStrategyC"

# The Client
CONTEXT = Context()

print(CONTEXT.request(ConcreteStrategyA))
print(CONTEXT.request(ConcreteStrategyB))
print(CONTEXT.request(ConcreteStrategyC))
```

7.9.5 Output

```
python.exe ./strategy/strategy_concept.py
I am ConcreteStrategyA
I am ConcreteStrategyB
I am ConcreteStrategyC
```

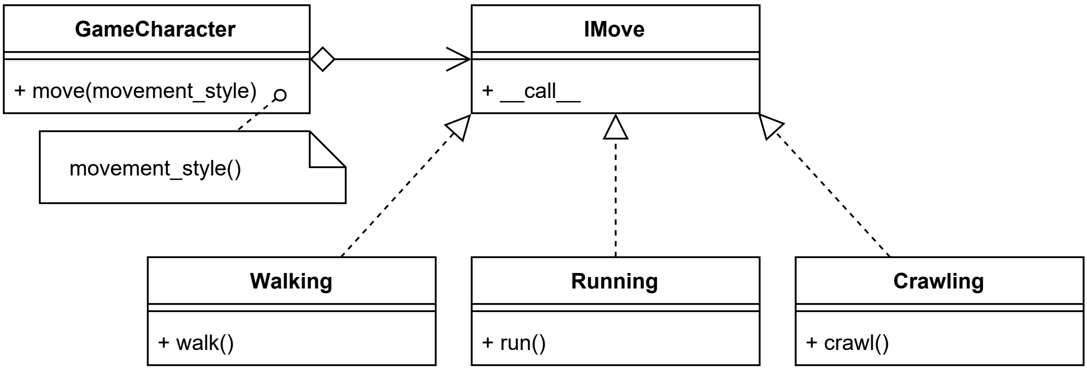
7.9.6 Strategy Use Case

SBCODE Video ID #89d36a

A game character is moving through an environment. Depending on the situation within the current environment, the user decides to use a different movement algorithm. From the perspective of the object/context, it is still a move, but the implementation is encapsulated in the subclass at the handle.

In a real game, the types of things that a particular move could affect is which animation is looped, whether physics attributes changed, the speed of movement, the camera follow mode and more.

7.9.7 Strategy Example Use Case UML Diagram



7.9.8 Source Code

./strategy/client.py

```
# pylint: disable=too-few-public-methods
"The Strategy Pattern Example Use Case"
from abc import ABCMeta, abstractmethod

class GameCharacter():
    "This is the context whose strategy will change"

    @staticmethod
    def move(movement_style):
        "The movement algorithm has been decided by the client"
        movement_style()

class IMove(metaclass=ABCMeta):
    "A Concrete Strategy Interface"

    @staticmethod
    @abstractmethod
    def __call__():
        "Implementors must select the default method"

class Walking(IMove):
    "A Concrete Strategy Subclass"

    @staticmethod
    def walk():
        "A walk algorithm"
        print("I am Walking")

    __call__ = walk
```

```
class Running(IMove):
    "A Concrete Strategy Subclass"

    @staticmethod
    def run():
        "A run algorithm"
        print("I am Running")

    __call__ = run

class Crawling(IMove):
    "A Concrete Strategy Subclass"

    @staticmethod
    def crawl():
        "A crawl algorithm"
        print("I am Crawling")

    __call__ = crawl

# The Client
GAME_CHARACTER = GameCharacter()
GAME_CHARACTER.move(Walking())
# Character sees the enemy
GAME_CHARACTER.move(Running())
# Character finds a small cave to hide in
GAME_CHARACTER.move(Crawling())
```

7.9.9 Output

```
python.exe ./strategy/client.py
I am Walking
I am Running
I am Crawling
```

7.9.10 Summary

- While the Strategy pattern looks very similar to the State pattern, the assigned strategy sub class/algorithm is not changing any state of the context. The class/algorithm can be re-executed or replaced with a different class/algorithm with no effect to the state of the context.
- The Strategy pattern is about having a choice of implementations that accomplish the same relative task.
- The particular strategies algorithm is encapsulated in order to keep the implementation of it de coupled from the context.

- As soon as the state of the context decides which subclass will be executed, then that is the State pattern, otherwise it is the Strategy pattern because the decision was made externally to the context and can be modified again without affecting the context.

7.10 Template Method Design Pattern

7.10.1 Overview

SBCODE Video ID #217370

In the **Template Method** pattern, you create an abstract class (template) that contains a **Template Method** that is a series of instructions that are a combination of abstract and hook methods.

Abstract methods need to be overridden in the subclasses that extend the abstract (template) class.

Hook methods normally have empty bodies in the abstract class. Subclasses can optionally override the hook methods to create custom implementations.

So, what you have, is an abstract class, with several types of methods, being the main template method, and a combination of abstract and/or hooks, that can be extended by different subclasses that all have the option of customizing the behavior of the template class without changing its underlying algorithm structure.

Template methods are useful to help you factor out common behavior within your library classes.

Note that this pattern describes the behavior of a **method** and how its inner method calls behave.

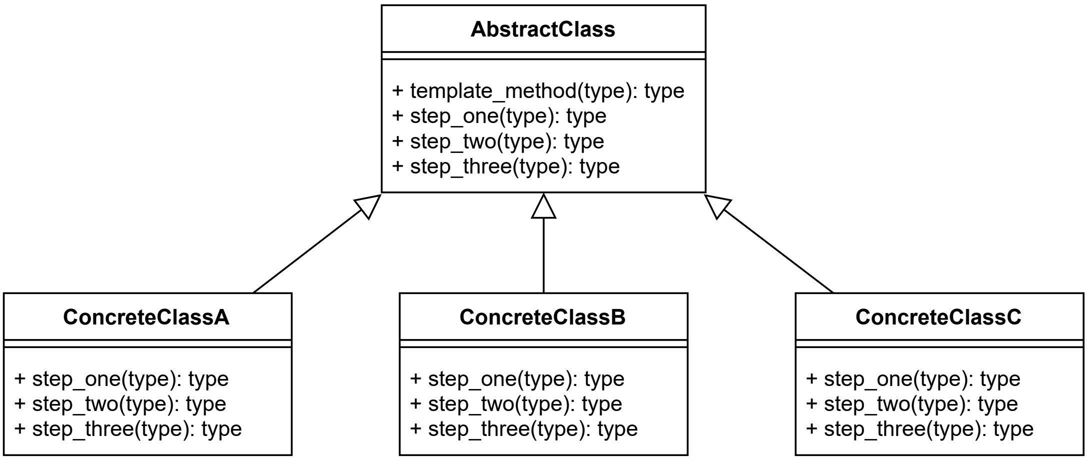
Hooks are default behavior and can be overridden. They are normally empty by default.

Abstract methods, must be overridden in the concrete class that extends the template class.

7.10.2 Terminology

- **Abstract Class**: Defines the template method and the primitive steps as abstract and/or hook methods.
- **Concrete Class**: A subclass that extends some or all of the abstract class primitive methods.

7.10.3 Template Method UML Diagram



7.10.4 Source Code

Note that in both the concrete classes in this concept example, the `template_method()` was not overridden since it was already inherited. Only the primitives (abstract or hooks) were optionally overridden.

To create an empty abstract method in your abstract class, that must be overridden in a subclass, then use the ABCMeta `@abstractmethod` decorator.

./template/template_concept.py

```
# pylint: disable=too-few-public-methods
"The Template Method Pattern Concept"
from abc import ABCMeta, abstractmethod

class AbstractClass(metaclass=ABCMeta):
    "A template class containing a template method and primitive methods"

    @staticmethod
    def step_one():
        """
        Hooks are normally empty in the abstract class. The
        implementing class can optionally override providing a custom
        implementation
        """

    @staticmethod
    @abstractmethod
    def step_two():
        """
        An abstract method that must be overridden in the implementing
        class. It has been given `@abstractmethod` decorator so that
        pylint shows the error.
        """

    @staticmethod
    def step_three():
        """
        Hooks can also contain default behavior and can be optionally
        overridden
        """
        print("Step Three is a hook that prints this line by default.")

    @classmethod
    def template_method(cls):
        """
        This is the template method that the subclass will call.
        The subclass (implementing class) doesn't need to override this
        method since it has would have already optionally overridden
        the following methods with its own implementations
        """
        cls.step_one()
        cls.step_two()
        cls.step_three()

class ConcreteClassA(AbstractClass):
    "A concrete class that only overrides step two"
    @staticmethod
    def step_two():
        print("Class_A : Step Two (overridden)")

class ConcreteClassB(AbstractClass):
    "A concrete class that only overrides steps one, two and three"
    @staticmethod
    def step_one():
        print("Class_B : Step One (overridden)")

    @staticmethod
    def step_two():
        print("Class_B : Step Two. (overridden)")

    @staticmethod
    def step_three():
        print("Class_B : Step Three. (overridden)")

# The Client
CLASS_A = ConcreteClassA()
CLASS_A.template_method()
```

```
CLASS_B = ConcreteClassB()
CLASS_B.template_method()
```

7.10.5 Output

```
python ./template/template_concept.py
Class_A : Step Two (overridden)
Step Three is a hook that prints this line by default.
Class_B : Step One (overridden)
Class_B : Step Two. (overridden)
Class_B : Step Three. (overridden)
```

7.10.6 Template Method Use Case

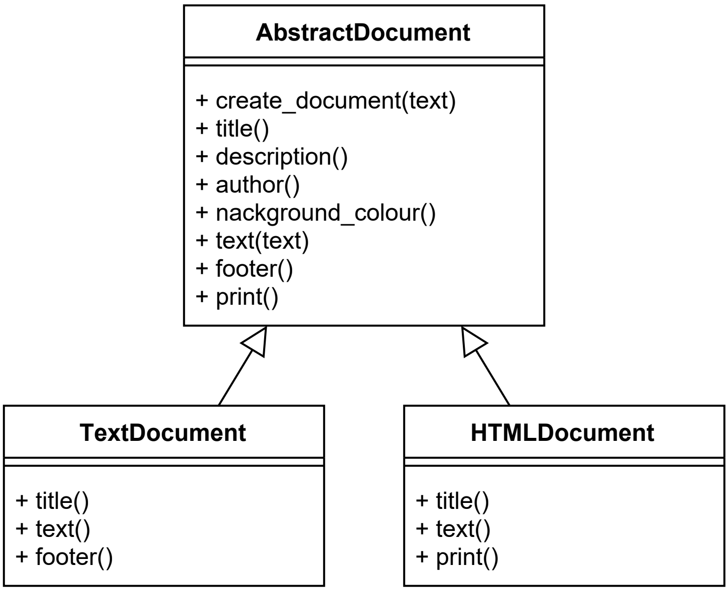
SBCODE Video ID #313583

In the example use case, there is an `AbstractDocument` with several methods, some are optional and others must be overridden.

The document will be written out in two different formats.

Depending on the concrete class used, the `text()` method will wrap new lines with `<p>` tags and the `print()` method will format text with tabs, or include html tags.

7.10.7 Template Method Use Case UML Diagram



7.10.8 Source Code

./template/client.py

```
"The Template Pattern Use Case Example"
from text_document import TextDocument
from html_document import HTMLDocument

TEXT_DOCUMENT = TextDocument()
TEXT_DOCUMENT.create_document("Some Text")

HTML_DOCUMENT = HTMLDocument()
HTML_DOCUMENT.create_document("Line 1\nLine 2")
```

./template/abstract_document.py

```
"An abstract document containing a combination of hooks and abstract
methods"
from abc import ABCMeta, abstractmethod

class AbstractDocument(metaclass=ABCMeta):
    "A template class containing a template method and primitive methods"

    @staticmethod
    @abstractmethod
    def title(document):
        "must implement"

    @staticmethod
    def description(document):
        "optional"

    @staticmethod
    def author(document):
        "optional"

    @staticmethod
    def background_colour(document):
        "optional with a default behavior"
        document["background_colour"] = "white"

    @staticmethod
    @abstractmethod
    def text(document, text):
        "must implement"

    @staticmethod
    def footer(document):
        "optional"

    @staticmethod
    def print(document):
        "optional with a default behavior"
        print("----------------------")
        for attribute in document:
            print(f"{attribute}\t: {document[attribute]}")
        print()

    @classmethod
    def create_document(cls, text):
        "The template method"
        _document = {}
        cls.title(_document)
        cls.description(_document)
        cls.author(_document)
        cls.background_colour(_document)
        cls.text(_document, text)
        cls.footer(_document)
        cls.print(_document)
```

./template/text_document.py

```
"A text document concrete class of AbstractDocument"
from abstract_document import AbstractDocument

class TextDocument(AbstractDocument):
    "Prints out a text document"
    @staticmethod
    def title(document):
        document["title"] = "New Text Document"

    @staticmethod
    def text(document, text):
        document["text"] = text

    @staticmethod
    def footer(document):
        document["footer"] = "-- Page 1 --"
```

./template/html_document.py

```
"A HTML document concrete class of AbstractDocument"
from abstract_document import AbstractDocument
```

```
class HTMLDocument(AbstractDocument):
    "Prints out a HTML formatted document"
    @staticmethod
    def title(document):
        document["title"] = "New HTML Document"

    @staticmethod
    def text(document, text):
        "Putting multiple lines into there own p tags"
        lines = text.splitlines()
        markup = ""
        for line in lines:
            markup = markup + "    <p>" + f"{line}</p>\n"
        document["text"] = markup[:-1]

    @staticmethod
    def print(document):
        "overriding print to output with html tags"
        print("<html>")
        print("  <head>")
        for attribute in document:
            if attribute in ["title", "description", "author"]:
                print(
                    f"    <{attribute}>{document[attribute]}"
                    f"</{attribute}>"
                )
            if attribute == "background_colour":
                print("    <style>")
                print("      body {")
                print(
                    f"        background-color: "
                    f"{document[attribute]};")
                print("      }")
                print("    </style>")
        print("  </head>")
        print("  <body>")
        print(f"{document['text']}")
        print("  </body>")
        print("</html>")
```

7.10.9 Output

```
python ./template/client.py
-----------------------
title   : New Text Document
background_colour       : white
text    : Some Text
```

```
footer  : -- Page 1 --

<html>
  <head>
    <title>New HTML Document</title>
    <style>
      body {
        background-color: white;
      }
    </style>
  </head>
  <body>
    <p>Line 1</p>
    <p>Line 2</p>
  </body>
</html>
```

7.10.10 Summary

- The Template method defines an algorithm in terms of abstract operations and subclasses override some or all of the methods to create concrete behaviors.
- Abstract methods must be overridden in the subclasses that extend the abstract class.
- Hook Methods usually have empty bodies in the superclass but can be optionally overridden in the subclass.
- If a class contains many conditional statements, consider converting it to use the Template Method pattern.

7.11 Visitor Design Pattern

7.11.1 Overview

SBCODE Video ID #4dffa4

Your object structure inside an application may be complicated and varied. A good example is what could be created using the Composite structure.

The objects that make up the hierarchy of objects, can be anything and most likely complicated to modify as your application grows.

Instead, when designing the objects in your application that may be structured in a hierarchical fashion, you can allow them to implement a **Visitor** interface.

The Visitor interface describes an `accept()` method that a different object, called a Visitor, will use in order to traverse through the existing object hierarchy and read the internal attributes of an object.

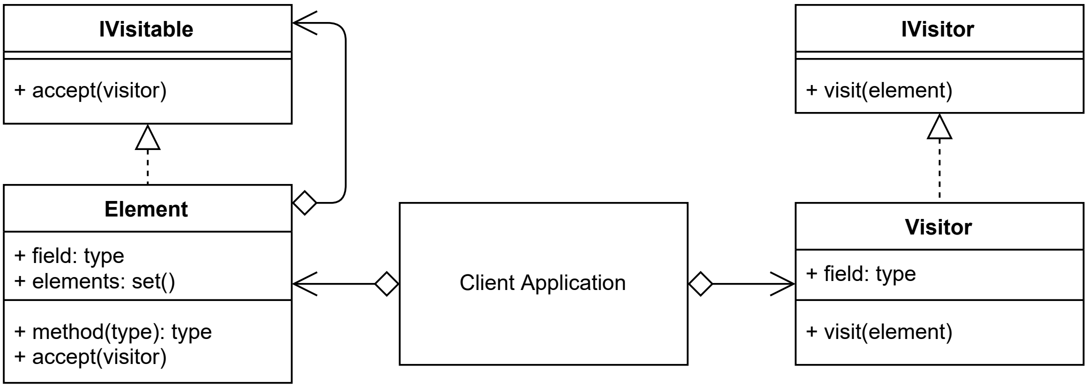
The Visitor pattern is useful when you want to analyze, or reproduce an alternative object hierarchy without implementing extra code in the object classes, except for the original requirements set by implementing the Visitor interface.

Similar to the template pattern it could be used to output different versions of a document but more suited to objects that may be members of a hierarchy.

7.11.2 Terminology

- **Visitor Interface**: An interface for the Concrete Visitors.
- **Concrete Visitor**: The Concrete Visitor will traverse the hierarchy of elements.
- **Visitable Interface**: The interface that elements should implement, that describes the `accept()` method that will allow them to be visited (traversed).
- **Concrete Element**: An object that will be visited. An application will contain a variable number of Elements than can be structured in any particular hierarchy.

7.11.3 Visitor UML Diagram



7.11.4 Source Code

In the concept code below, a hierarchy of any object is created. It is similar to a simplified composite. The objects of `Element` can also contain a hierarchy of sub elements.

The `Element` class could also consist of many variations, but this example uses only one.

Rather than writing specific code inside all these elements every time I wanted to handle a new custom operation, I can implement the `IVisitable` interface and create the `accept()` method that allows the Visitor to pass through it and access the Elements internal attributes.

Two different Visitor classes are created, `PrintElementNamesVisitor` and `CalculateElementTotalsVisitor` . They are instantiated and passed through the existing Object hierarchy using the same `IVisitable` interface.

./visitor/visitor_concept.py

```
# pylint: disable=too-few-public-methods
"The Visitor Pattern Concept"
from abc import ABCMeta, abstractmethod

class IVisitor(metaclass=ABCMeta):
    "An interface that custom Visitors should implement"
    @staticmethod
    @abstractmethod
    def visit(element):
        "Visitors visit Elements/Objects within the application"

class IVisitable(metaclass=ABCMeta):
    """
    An interface the concrete objects should implement that allows
    the visitor to traverse a hierarchical structure of objects
    """
    @staticmethod
    @abstractmethod
    def accept(visitor):
        """
        The Visitor traverses and accesses each object through this
        method
        """

class Element(IVisitable):
    "An Object that can be part of any hierarchy"

    def __init__(self, name, value, parent=None):
        self.name = name
        self.value = value
        self.elements = set()
        if parent:
            parent.elements.add(self)

    def accept(self, visitor):
        "required by the Visitor that will traverse"
        for element in self.elements:
            element.accept(visitor)
        visitor.visit(self)

# The Client
# Creating an example object hierarchy.
Element_A = Element("A", 101)
Element_B = Element("B", 305, Element_A)
Element_C = Element("C", 185, Element_A)
Element_D = Element("D", -30, Element_B)

# Now Rather than changing the Element class to support custom
# operations, we can utilise the accept method that was
# implemented in the Element class because of the addition of
# the IVisitable interface

class PrintElementNamesVisitor(IVisitor):
    "Create a visitor that prints the Element names"
    @staticmethod
    def visit(element):
        print(element.name)

# Using the PrintElementNamesVisitor to traverse the object hierarchy
Element_A.accept(PrintElementNamesVisitor)

class CalculateElementTotalsVisitor(IVisitor):
    "Create a visitor that totals the Element values"
    total_value = 0

    @classmethod
    def visit(cls, element):
        cls.total_value += element.value
        return cls.total_value

# Using the CalculateElementTotalsVisitor to traverse the
# object hierarchy
TOTAL = CalculateElementTotalsVisitor()
Element_A.accept(CalculateElementTotalsVisitor)
print(TOTAL.total_value)
```

7.11.5 Output

```
python ./visitor/visitor_concept.py
D
B
C
A
561
```

7.11.6 Visitor Use Case

SBCODE Video ID #f5f97b

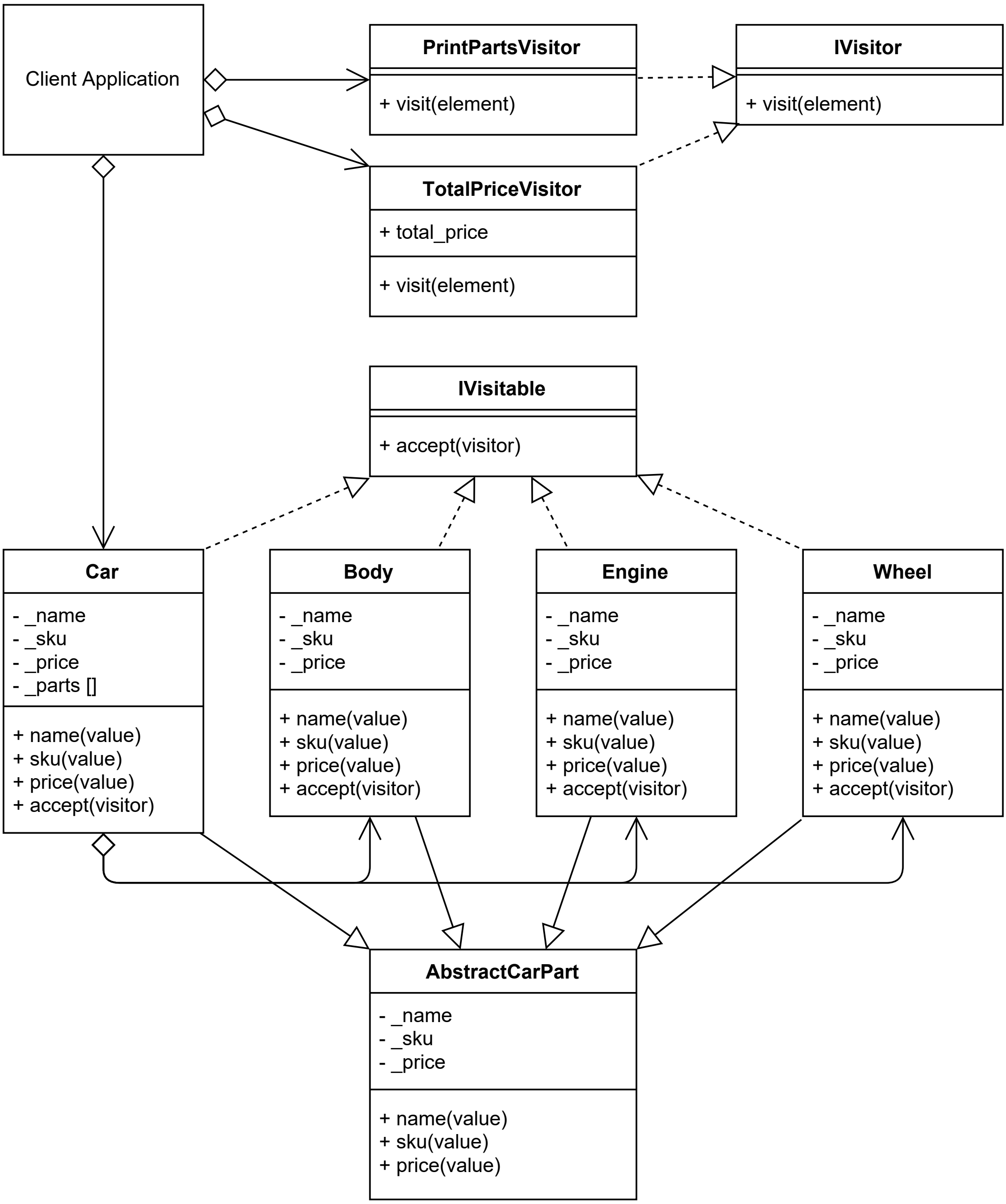
In the example, the client creates a car with parts.

The car and parts inherit an abstract car parts class with predefined property getters and setters.

Instead of creating methods in the car parts classes and abstract class that run bespoke methods, the car parts can all implement the `IVisitor` interface.

This allows for the later creation of Visitor objects to run specific tasks on the existing hierarchy of objects.

7.11.7 Visitor Example UML Diagram



7.11.8 Source Code

./visitor/client.py

```
# pylint: disable=too-few-public-methods
"The Visitor Pattern Use Case Example"
```

```
from abc import ABCMeta, abstractmethod

class IVisitor(metaclass=ABCMeta):
    "An interface that custom Visitors should implement"
    @staticmethod
    @abstractmethod
    def visit(element):
        "Visitors visit Elements/Objects within the application"

class IVisitable(metaclass=ABCMeta):
    """
    An interface that concrete objects should implement that allows
    the visitor to traverse a hierarchical structure of objects
    """
    @staticmethod
    @abstractmethod
    def accept(visitor):
        """
        The Visitor traverses and accesses each object through this
        method
        """

class AbstractCarPart():
    "The Abstract Car Part"
    @property
    def name(self):
        "a name for the part"
        return self._name

    @name.setter
    def name(self, value):
        self._name = value

    @property
    def sku(self):
        "The Stock Keeping Unit (sku)"
        return self._sku

    @sku.setter
    def sku(self, value):
        self._sku = value

    @property
    def price(self):
        "The price per unit"
        return self._price

    @price.setter
    def price(self, value):
        self._price = value

class Body(AbstractCarPart, IVisitable):
    "A part of the car"

    def __init__(self, name, sku, price):
        self._name = name
        self._sku = sku
        self._price = price

    def accept(self, visitor):
        visitor.visit(self)

class Engine(AbstractCarPart, IVisitable):
    "A part of the car"

    def __init__(self, name, sku, price):
        self._name = name
        self._sku = sku
        self._price = price

    def accept(self, visitor):
        visitor.visit(self)

class Wheel(AbstractCarPart, IVisitable):
    "A part of the car"

    def __init__(self, name, sku, price):
        self._name = name
        self._sku = sku
        self._price = price

    def accept(self, visitor):
        visitor.visit(self)

class Car(AbstractCarPart, IVisitable):
    "A Car with parts"

    def __init__(self, name):
        self._name = name
        self._parts = [
            Body("Utility", "ABC-123-21", 1001),
            Engine("V8 engine", "DEF-456-21", 2555),
            Wheel("FrontLeft", "GHI-789FL-21", 136),
            Wheel("FrontRight", "GHI-789FR-21", 136),
            Wheel("BackLeft", "GHI-789BL-21", 152),
            Wheel("BackRight", "GHI-789BR-21", 152),
        ]
    def accept(self, visitor):
        for parts in self._parts:
            parts.accept(visitor)
        visitor.visit(self)

class PrintPartsVisitor(IVisitor):
    "Print out the part name and sku"
    @staticmethod
    def visit(element):
        if hasattr(element, 'sku'):
            print(f"{element.name}\t:{element.sku}".expandtabs(6))

class TotalPriceVisitor(IVisitor):
    "Print out the total cost of the parts in the car"
    total_price = 0

    @classmethod
    def visit(cls, element):
        if hasattr(element, 'price'):
            cls.total_price += element.price
        return cls.total_price

# The Client
CAR = Car("DeLorean")

# Print out the part name and sku using the PrintPartsVisitor
CAR.accept(PrintPartsVisitor())

# Calculate the total prince of the parts using the TotalPriceVisitor
TOTAL_PRICE_VISITOR = TotalPriceVisitor()
CAR.accept(TOTAL_PRICE_VISITOR)
print(f"Total Price = {TOTAL_PRICE_VISITOR.total_price}")
```

7.11.9 Output

```
python ./visitor/client.py
Utility     :ABC-123-21
V8 engine   :DEF-456-21
FrontLeft   :GHI-789FL-21
FrontRight  :GHI-789FR-21
BackLeft    :GHI-789BL-21
BackRight   :GHI-789BR-21
Total Price = 4132
```

7.11.10 New Coding Concepts

Instance `hasattr()`

SBCODE Video ID #87a91c

In the Visitor objects in the example use case above, I test if the elements have a certain attribute during the visit operation.

```
def visit(cls, element):
    if hasattr(element, 'price'):
        ...
```

The `hasattr()` method can be used to test if an instantiated object has an attribute of a particular name.

```
class ClassA():
    name = "abc"
    value = 123

CLASS_A = ClassA()
print(hasattr(CLASS_A, "name"))
print(hasattr(CLASS_A, "value"))
print(hasattr(CLASS_A, "date"))
```

Outputs

```
True
True
False
```

String `expandtabs()`

SBCODE Video ID #f8c834

When printing strings to the console, you can include special characters `\t` that print a series of extra spaces called tabs. The tabs help present multiline text in a more tabular form that appears to be neater to look at.

```
abc     123
defg    456
hi      78910
```

The number of spaces added depends on the size of the word before the `\t` character in the string. By default, a tab makes up 8 spaces.

Now, not all words separated by a tab will line up the same on the next line.

```
abcdef  123
cdefghij        4563
ghi     789
jklmn   1011
```

The problem occurs usually when a word is already 8 or more characters long.

To help solve the spacing issue, you can use the `expandtabs()` method on a string to set how many characters a tab will use.

```
print("abcdef\t123".expandtabs(10))
print("cdefghij\t4563".expandtabs(10))
print("ghi\t789".expandtabs(10))
print("jklmn\t1011".expandtabs(10))
```

Now outputs

```
abcdef    123
cdefghij  4563
ghi       789
jklmn     1011
```

7.11.11 Summary

- Use the Visitor pattern to define an operation to be performed on the elements of a hierarchal object structure.
- Use the Visitor pattern to define the new operation without needing to change the classes of the elements on that it operates.
- When designing your application, you can provision for the future possibility of needing to run custom operations on an element hierarchy, by implementing the Visitor interface in anticipation.
- Usage of the Visitor pattern helps to ensure that your classes conform to the single responsibility principle due to them implementing the custom visitor behavior in a separate class.

8. Summary

A table of one-liners to help summarize the design patterns in this book.

Pattern	Description
Abstract Factory	Adds an abstraction over many other related objects that are created using other creational patterns.
Adapter	An alternative interface over an existing interface.
Bridge	The Bridge pattern is similar to the Adapter pattern except in the intent that you developed it.
Builder	A creational pattern whose intent is to separate the construction of a complex object from its representation so that you can use the same construction process to create different representations.
Chain of Responsibility	Pass an object through a chain of successor handlers.
Command	An abstraction between an object that invokes a command, and the object that performs it. Useful for UNDO/REDO/REPLAY.
Composite	A structural pattern useful for hierarchal management.
Decorator	Attach additional responsibilities to an object at runtime.
Facade	An alternative or simplified interface over other interfaces.
Factory	Abstraction between the creation of an object and where it is used.
Flyweight	Share objects rather than creating thousands of near identical copies.
Interpreter	Convert information from one language to another.
Iterator	Traverse a collection of aggregates.
Mediator	Objects communicate through a Mediator rather than directly with each other.
Memento	Save a copy of state and for later retrieval. Useful for UNDO/REDO/LOAD/SAVE.
Observer Pattern	Manage a list of dependents and notifies them of any internal state changes.
Prototype	Good for when creating new objects requires more resources than you need of have available.

Pattern	Description
Proxy	A class functioning as an interface to another class or object.
Singleton	A class that can be instanced at any time, but after it is first instanced, any new instances will point to the original instance.
State	Alter an objects behavior by changing the handle of one of its methods to one of its subclasses dynamically to reflect its new internal state.
Strategy	Similar to the State Pattern, except that the client passes in the algorithm that the context should then run.
Template Method	An abstract class (template) that contains a method that is a series of instructions that are a combination of methods that can be overridden.
Visitor	Pass an object called a visitor to a hierarchy of objects and execute a method on them.

Remember that design patterns will give you a useful and common vocabulary for when designing, documenting, analyzing, restructuring new and existing software development projects now and into the future.

Good luck and I hope that your projects become very successful.

Sean Bradley