Занятие 27

Паттерны

Давайте пройдем тест
https://itproger.com/test/python

Задача 26-1

Напишите функцию, которая сравнивает две строки и выдает True, если между ними есть не более, чем 1 разница в буквах, и False во всех остальных случаях. Если две строки равны, то True.

Например:

‘abc’ и ‘abc’ – True, ‘abc’ и ‘abcd’ – True,

‘bc’ и ‘abc’ – True, ‘axc’ и ‘abc’ – True

‘abc’ и ‘acb’ – False, ‘abc’ и ‘a’ – False, ‘’ и ‘ ‘ - False

Задача 26-2

Создайте класс Par, используя функцию type и с методом dis, определенным заранее и печатающим все атрибуты класса Par.

Функцию dis для метода Par.dis определите заранее.

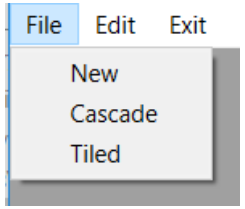
Задача 26-3

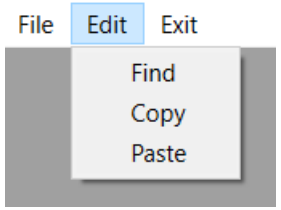
Создайте графическое приложение с пунктами меню: Описание, Инструкция, Помощь.

Меню

Чтобы создать меню, нужно прописать в QMainWindow строку меню .menuBar() и добавить в неё меню, вызвав .addMenu() и передав название создаваемого меню, например &File.

```
bar = self.menuBar()

file = bar.addMenu("File")
file.addAction("New")
file.addAction("Cascade")
file.addAction("Tiled")

edit = bar.addMenu("Edit")
edit.addAction("Find")
edit.addAction("Copy")
edit.addAction("Paste")

fileMenu = bar.addMenu('&Exit')
fileMenu.addAction(exitAction)
```



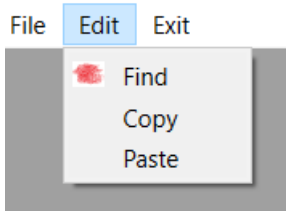


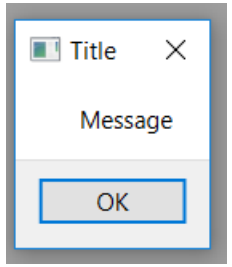
MDI – Multiple Document Interface

```
from PyQt6.QtWidgets import QApplication, QMainWindow, QMdiArea, QMdiSubWindow, QTextEdit
from PyQt6.QtGui import QAction
import sys


class MDIWindow(QMainWindow):
    count = 0

    def __init__(self):
        super().__init__()

        self.mdi = QMdiArea()
        self.setCentralWidget(self.mdi)
        bar = self.menuBar()
        file = bar.addMenu("File")
        file.addAction("New")
        file.addAction("Cascade")
        file.addAction("Tiled")
        file.triggered[QAction].connect(self.WindowTrig)
        self.setWindowTitle("MDI Application")

    def WindowTrig(self, p):
        if p.text() == "New":
            MDIWindow.count = MDIWindow.count + 1
            sub = QMdiSubWindow()
            sub.setWidget(QTextEdit())
            sub.setWindowTitle("Sub" + str(MDIWindow.count))
            self.mdi.addSubWindow(sub)
            sub.show()
        if p.text() == "Cascade":
            self.mdi.cascadeSubWindows()
        if p.text() == "Tiled":
            self.mdi.tileSubWindows()


app = QApplication(sys.argv)
mdiwindow = MDIWindow()
mdiwindow.show()
app.exec()
```

Message Box

```
    editFindAction = QAction(QIcon('t2.png'), 'Find', self)
    editFindAction.triggered.connect(self.clickFind)

    edit = bar.addMenu("Edit")
    edit.addAction(editFindAction)
    edit.addAction("Copy")
    edit.addAction("Paste")

    exitAction = QAction(QIcon('t1.png'), 'Exit', self)
    exitAction.setShortcut('Ctrl+Q')
    exitAction.setStatusTip('Exit application')
    exitAction.triggered.connect(self.close)

    fileMenu = bar.addMenu('&Exit')
    fileMenu.addAction(exitAction)

    file.triggered[QAction].connect(self.WindowTrig)
    self.setWindowTitle("MDI Application")

def clickFind(self):
    QMessageBox.about(self, "Title", "Message")
```

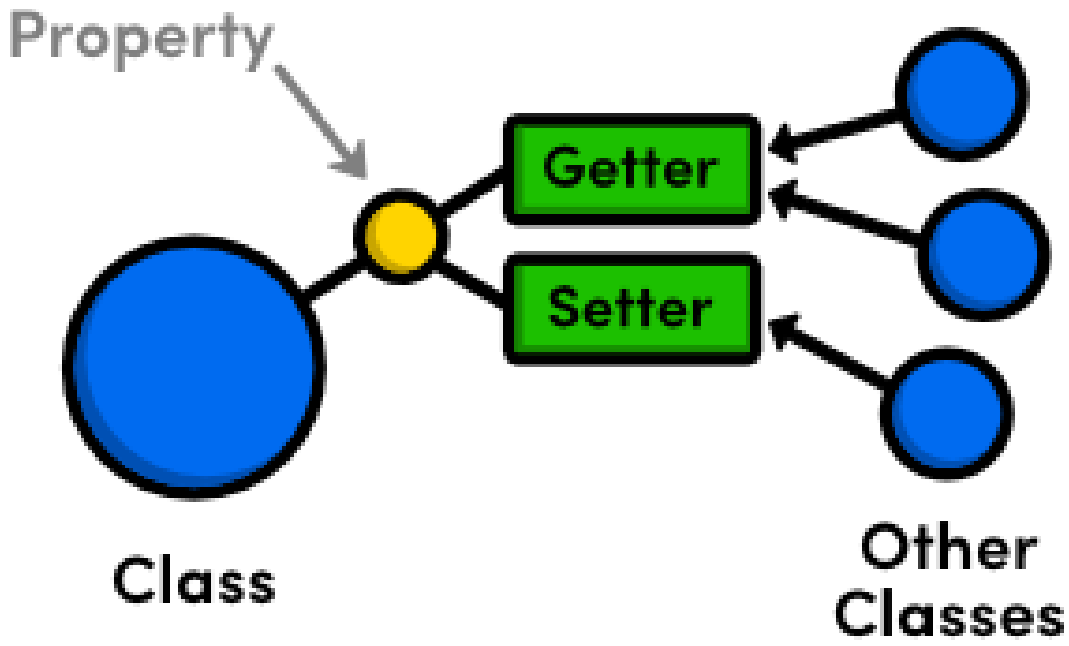




Задание

Добавьте в вашу программу вызов MessageBox в одну из строчек меню

Методы доступа к свойствам



Функции для работы с атрибутами

__getattribute__(self, name) — вызывается при обращении к любому атрибуту

__getattr__(self, name) — вызывается при обращении к несуществующему атрибуту

Разница между ними в том, что метод __getattribute__() вызывается первым и вызывается всегда, а метод __getattr__() вызывается только в том случае, если атрибута, к которому происходит обращение, не существует. Если атрибут существует, метод __getattribute__() возвращает его значение, в противном случае вызывается метод __getattr__().

__setattr__(self, name, value) — вызывается при установке атрибута или изменении его значения

__delattr__(self, name) — вызывается при удалении любого атрибута

Вспомним __str__ и __repr__

Реализуйте класс Vector, описывающий вектор на плоскости, конструктор которого принимает два аргумента:

x — координата вектора по оси x

y — координата вектора по оси y

(вектор начинается в точке (0, 0) и заканчивается в точке (x, y)).

Экземпляр класса Vector должен иметь следующее формальное (__repr__) строковое представление:

Vector(<координата x>, <координата y>)

И следующее неформальное (__str__) строковое представление:

Вектор на плоскости с координатами (<координата x>, <координата y>)

Задание на использование __setattr__(self, name, value)

Реализуйте класс AnyClass, конструктор которого принимает произвольное количество именованных аргументов и устанавливает их в качестве атрибутов создаваемому экземпляру класса.

Экземпляр класса AnyClass должен иметь следующее формальное строковое представление:

AnyClass(<имя 1-го атрибута>=<значение 1-го атрибута>, <имя 2-го атрибута>=<значение 2-го атрибута>, ...)

И следующее неформальное строковое представление:

AnyClass: <имя 1-го атрибута>: <значение 1-го атрибута>, <имя 2-го атрибута>: <значение 2-го атрибута>, ...

Метаклассы

Мы можем добавлять методы и атрибуты из программы

Мы можем создавать экземпляры классов из программы

Как создавать классы из программы?

Для этого используется функция:

type(Имя класса, Родительские классы, Словарь атрибутов и методов)

Метаклассы – type()

```
xyz = type('ClassA', (), {})
x = xyz()
print(type(x)) # <class '__main__.ClassA'>

strb = 'ClassB'
zyx = type(strb, (), {})  # В этой строчке нет слова ClassB !
b = zyx()                 # В этой тоже нет, но b – экземпляр класса ClassB
print(type(b)) # <class '__main__.ClassB’>

# Выполните этот код
```

type() - Метаклассы

```
class Foo(object):
    bar = True
Foo = type('Foo', (), {'bar':True}) # То же самое!

f = Foo()
class FooChild(Foo):
   pass
FooChild = type('Foochild', (Foo,), {})

def echo_bar(self):
    print(self.bar)
FooChild = type('FooChild', (Foo,), {'echo_bar': echo_bar})

hasattr(Foo, 'echo_bar')
my_foo = FooChild()
my_foo.echo_bar()
```

Паттерны проектирования

Шаблоны проектирования

- Шаблоны проектирования – это стандартные(обобщенные) решения для определенных задач.

- Предложены хорошими специалистами
- Проверены временем
- Повсеместно используемые

- В области ПО использование шаблонов проектирования было предложено и развито Gamma, Helms, Johnson и Vlissades – в книге «Design Patterns: Elements of Reusable Object-Oriented Software» 1995

Что общего? Что различного?
Какой паттерн стоит рассмотреть или создать?

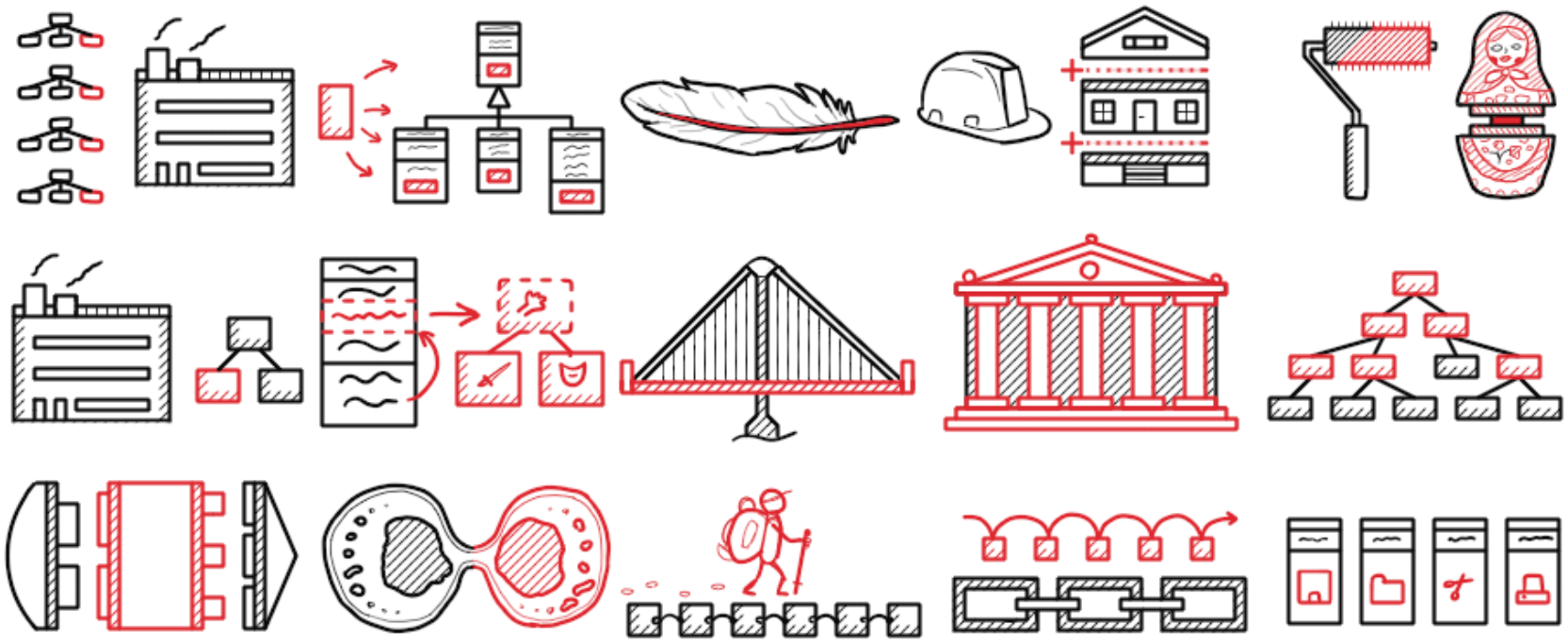
Задача 1.

Преподаватель и ученики. Преподаватель дает задания и проверяет выполнение.

Задача 2.

Менеджер и команда программистов. Менеджер руководит проектом, раздает задания и проверяет их выполнение в срок.

Шаблоны проектирования "банды четырёх (Gang of Four)"



Определение

- Под паттерном проектирования понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.

Для чего применяются паттерны ?

- Для создания “красивого дизайна”
- Инкапсуляция
- Полиморфизм
- Наследование
- Абстракция
- Слабая связанность
- И др.

Классификация паттернов GoF

- Порождающие (Creational)
- Структурные (Structural)
- Поведенческие (Behavioral)

1. Порождающие шаблоны	2. Структурные шаблоны	3. Шаблоны поведения
1. Abstract Factory	1. Adapter	1. Chain of Responsibility
2. Builder	2. Bridge	2. Command
3. Factory Method	3. Composite	3. Interpreter
4. Prototype	4. Decorator	4. Iterator
5. Singleton	5. Façade	5. Mediator
	6. Flyweight	6. Momento
	7. Proxy	7. Observer
		8. State
		9. Strategy
		10. Template Method
		11. Visitor

Порождающие шаблоны **(Creational)** — шаблоны проектирования, которые абстрагируют процесс инстанцирования.
Они позволяют сделать систему независимой от способа создания, композиции и представления объектов.
Шаблон, порождающий классы, использует наследование, чтобы изменять инстанцируемый класс, а шаблон, порождающий объекты, делегирует инстанцирование другому объекту.

Абстрактная фабрика	Abstract factory	Класс, который представляет собой интерфейс для создания компонентов системы.
Строитель	Builder	Класс, который представляет собой интерфейс для создания сложного объекта.
Фабричный метод	Factory method	Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.
Прототип	Prototype	Определяет интерфейс создания объекта через клонирование другого объекта вместо создания через конструктор.
Одиночка	Singleton	Класс, который может иметь только один экземпляр.

Структурные шаблоны (Structural) определяют различные сложные структуры, которые изменяют интерфейс уже существующих объектов или его реализацию, позволяя облегчить разработку и оптимизировать программу.

Адаптер	Adapter / Wrapper	Объект, обеспечивающий взаимодействие двух других объектов, один из которых использует, а другой предоставляет несовместимый с первым интерфейс.
Мост	Bridge	Структура, позволяющая изменять интерфейс обращения и интерфейс реализации класса независимо.
Компоновщик	Composite	Объект, который объединяет в себе объекты, подобные ему самому.
Декоратор или Wrapper/Обёртка	Decorator	Класс, расширяющий функциональность другого класса без использования наследования.
Фасад	Facade	Объект, который абстрагирует работу с несколькими классами, объединяя их в единое целое.
Приспособленец	Flyweight	Это объект, представляющий себя как уникальный экземпляр в разных местах программы, но фактически не являющийся таковым.
Заместитель	Proxy	Объект, который является посредником между двумя другими объектами, и который реализует/ограничивает доступ к объекту, к которому обращаются через него.

Поведенческие шаблоны (Behavioral) определяют взаимодействие между объектами, увеличивая таким образом его гибкость.

Цепочка обязанностей	Chain of responsibility	Предназначен для организации в системе уровней ответственности.
Команда, Action, Transaction	Command	Представляет действие. Объект команды заключает в себе само действие и его параметры.
Интерпретатор	Interpreter	Решает часто встречающуюся, но подверженную изменениям, задачу.
Итератор, Cursor	Iterator	Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из объектов, входящих в состав агрегации.
Посредник	Mediator	Обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.
Хранитель	Memento	Позволяет не нарушая инкапсуляцию зафиксировать и сохранить внутренние состояния объекта так, чтобы позднее восстановить его в этих состояниях.
Наблюдатель или Издатель-подписчик	Observer	Определяет зависимость типа «один ко многим» между объектами таким образом, что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.
Состояние	State	Используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния.
Стратегия	Strategy	Предназначен для определения семейства алгоритмов, инкапсуляции каждого из них и обеспечения их взаимозаменяемости.
Шаблонный метод	Template method	Определяет основу алгоритма и позволяет наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
Посетитель	Visitor	Описывает операцию, которая выполняется над объектами других классов. При изменении класса Visitor нет необходимости изменять обслуживаемые классы.

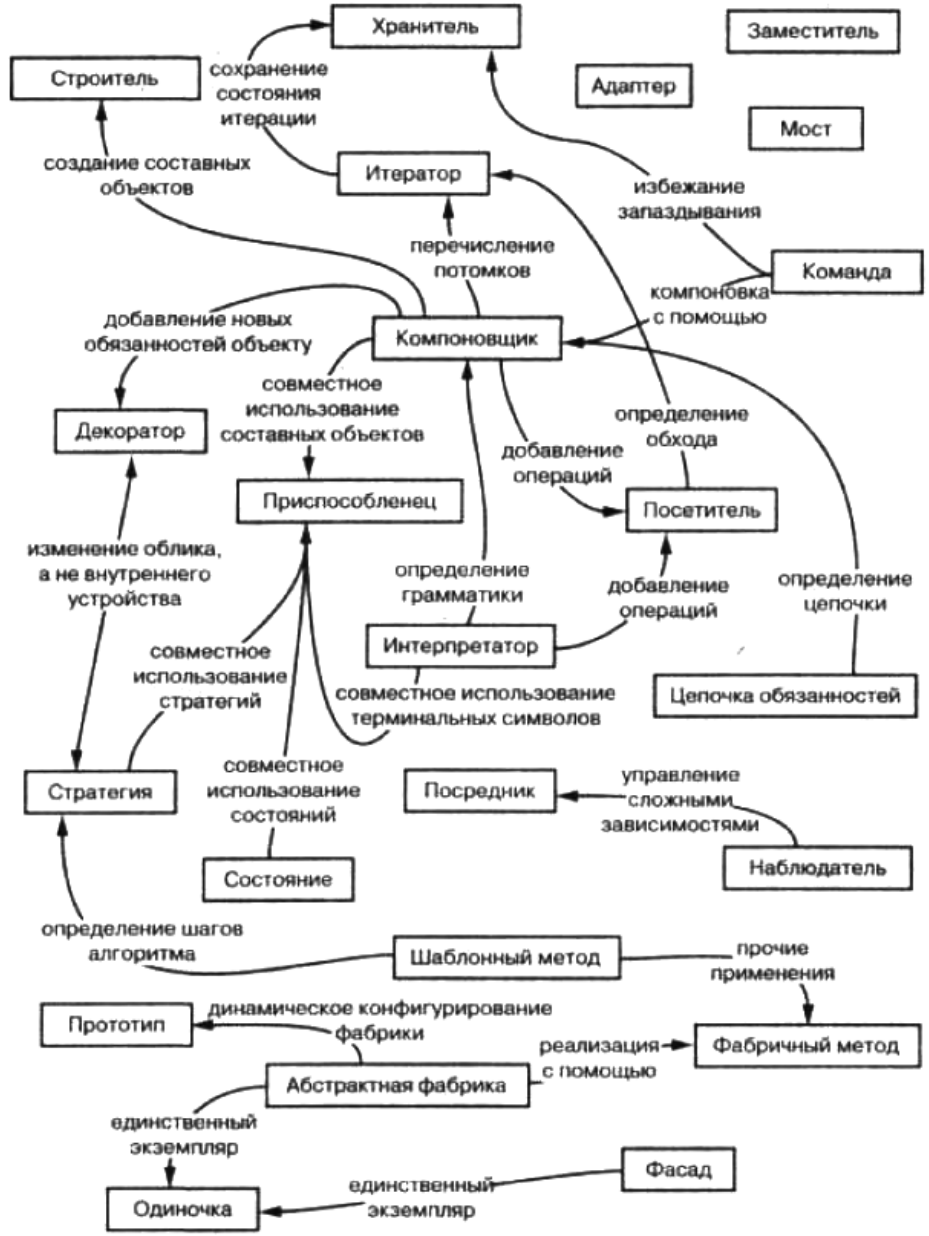


Рис. 1.1. Отношения между паттернами проектирования

Как выбрать паттерн проектирования

- ✔ Подумайте, как паттерны решают проблемы проектирования
- ✔ Пролистайте разделы каталога, описывающие назначение паттернов
- ✔ Изучите взаимосвязи паттернов
- ✔ Проанализируйте паттерны со сходными целями
- ✔ Разберитесь в причинах, вызывающих перепроектирование
- ✔ Посмотрите, что в вашем дизайне должно быть изменяющимся

Когда использовать паттерны ?

- Необоснованное использование паттернов может существенно усложнить программу, поэтому применять паттерны нужно, или когда проблема уже возникает или когда ее появление можно предсказать
- Существует мнение, что применять паттерны надо начинать только тогда, когда программист сам начинает выделять паттерны в своих проектах; до этого момента в большинстве случаев будет иметь место необоснованное их использование
- Но в любом случае познакомиться с ними необходимо хотя бы для того, чтобы как можно раньше появился материал для размышлений

Как пользоваться шаблоном ?

- Прочитайте описание паттерна, чтобы получить о нем общее представление
- Изучите разделы «Структура», «Участники» и «Отношения»
- Посмотрите на раздел «Пример кода», где приведен конкретный пример использования паттерна в программе
- Выберите для участников паттерна подходящие имена

–Обычно имена участников паттерна слишком абстрактны, но иногда бывает удобно включить имена участников паттерна в имена элементов программы (SimpleLayoutStrategy, TeXLayoutStrategy)

- Определите классы

–Объявите их интерфейсы, установите отношения наследования и определите переменные экземпляра, которыми будут представлены данные объекты и ссылки на другие объекты.

- Определите имена операций, встречающихся в паттерне

–Будьте последовательны при выборе имен. Например, для обозначения фабричного метода можно было бы всюду использовать префикс Create-.

- Реализуйте операции, которые выполняют обязанности и отвечают за отношения, определенные в паттерне

Singleton

Синглтон (одиночка) – это паттерн проектирования, цель которого ограничить возможность создания объектов данного класса одним экземпляром.

Он обеспечивает глобальность до одного экземпляра и глобальный доступ к созданному объекту.

Примеры использования

Класс в вашей программе имеет только один экземпляр, доступный всем клиентам.

Например, один объект базы данных, разделяемый различными частями программы.

В случае если вам необходим более строгий контроль над глобальными переменными.

Простой способ

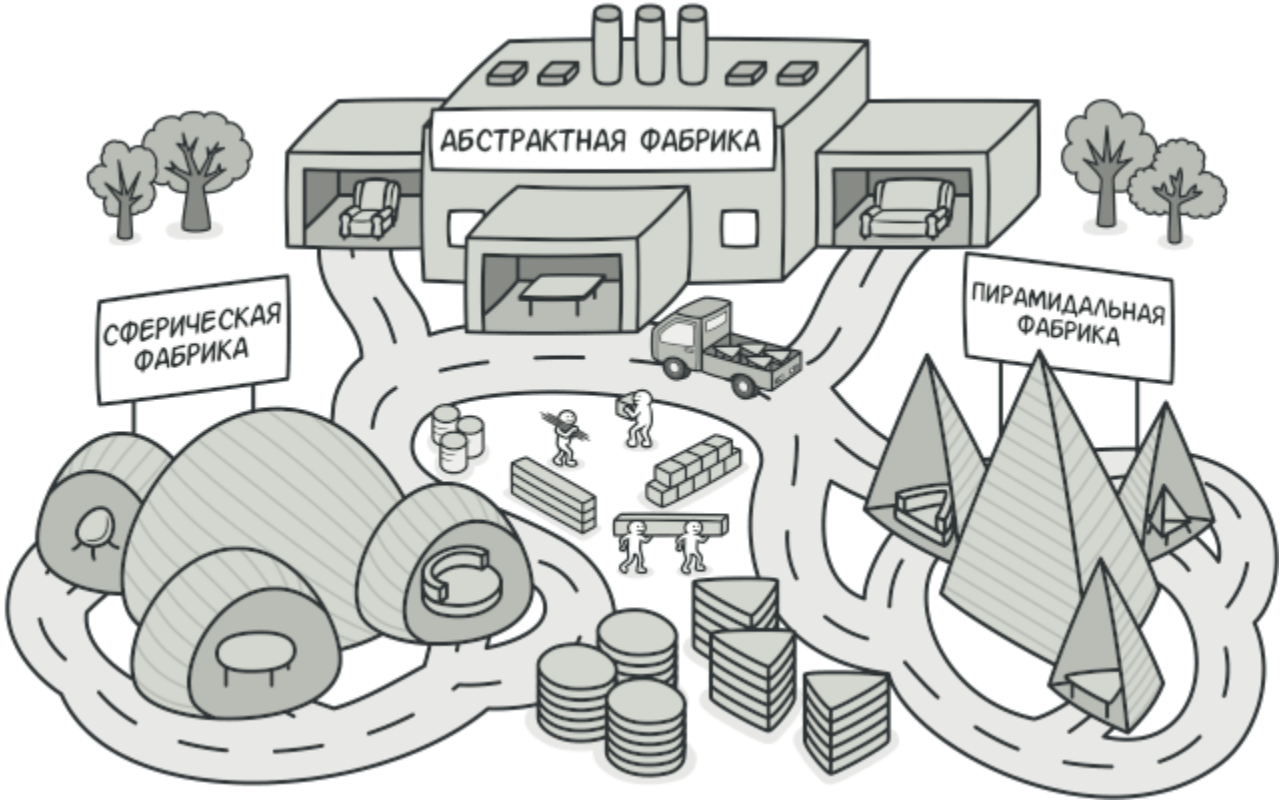
```
class Singleton:
    def __new__(cls):
        if not hasattr(cls, 'instance'):
            cls.instance = super(Singleton, cls).__new__(cls)
        return cls.instance


s1 = Singleton()
s2 = Singleton()
print(s1 is s2) # True
s1.x = 123
print(s2.x)
# Выполните этот код, убедитесь, что это один объект
```

Singleton как метакласс

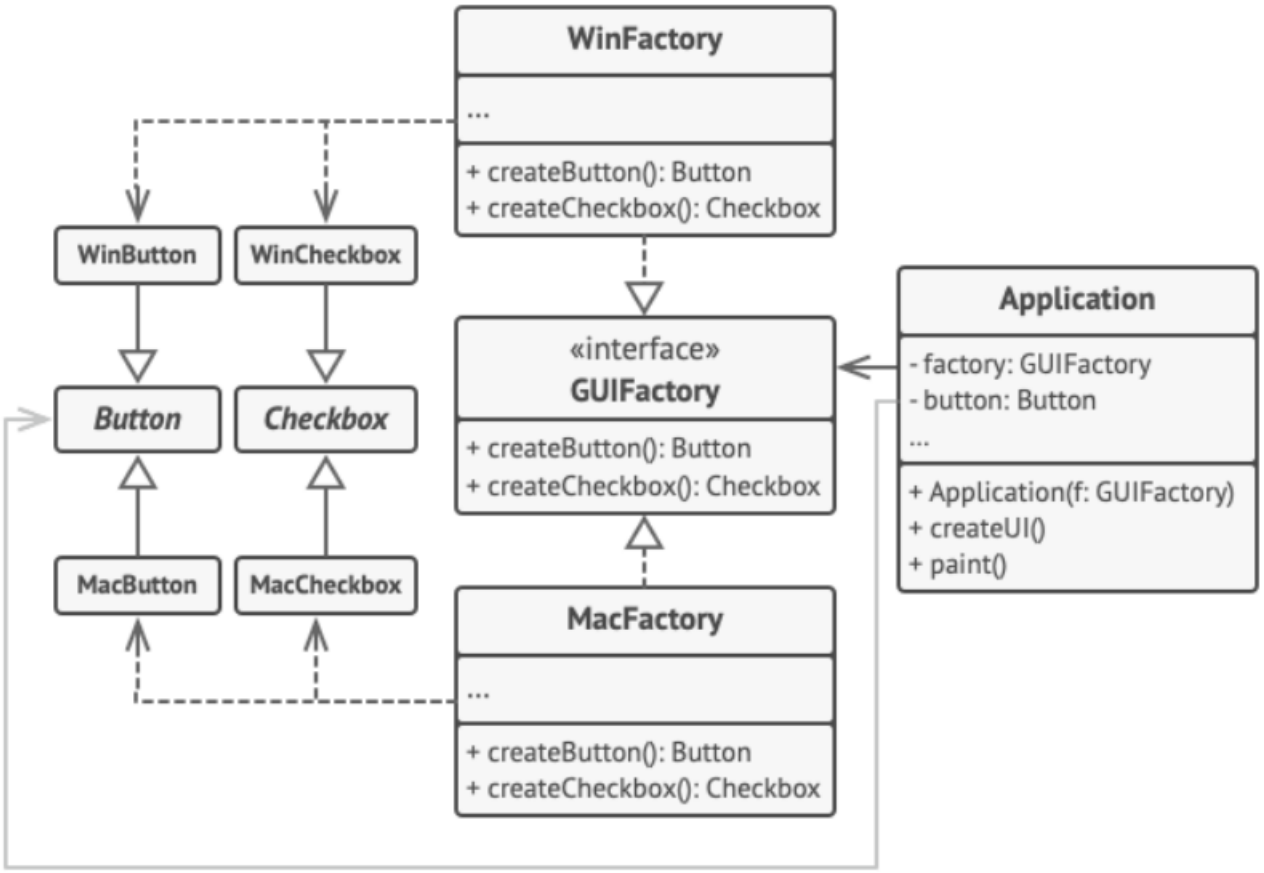
```
class Singleton(type):
        _instances = {}
        def __call__(cls, *args, **kwargs):
                if cls not in cls._instances:
                        instance = super().__call__(*args, **kwargs)
                        cls._instances[cls] = instance
                return cls._instances[cls]


class Logger(metaclass=Singleton):
        def write_log(self, path):
                pass
if __name__ == "__main__":
        logger1 = Logger()
        logger2 = Logger()
```

Абстрактная фабрика

Абстрактная фабрика — это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.



На примере графического интерфейса



Фасад

Представьте, что у вас есть система со значительным количеством объектов.

Каждый объект предлагает богатый набор методов API.

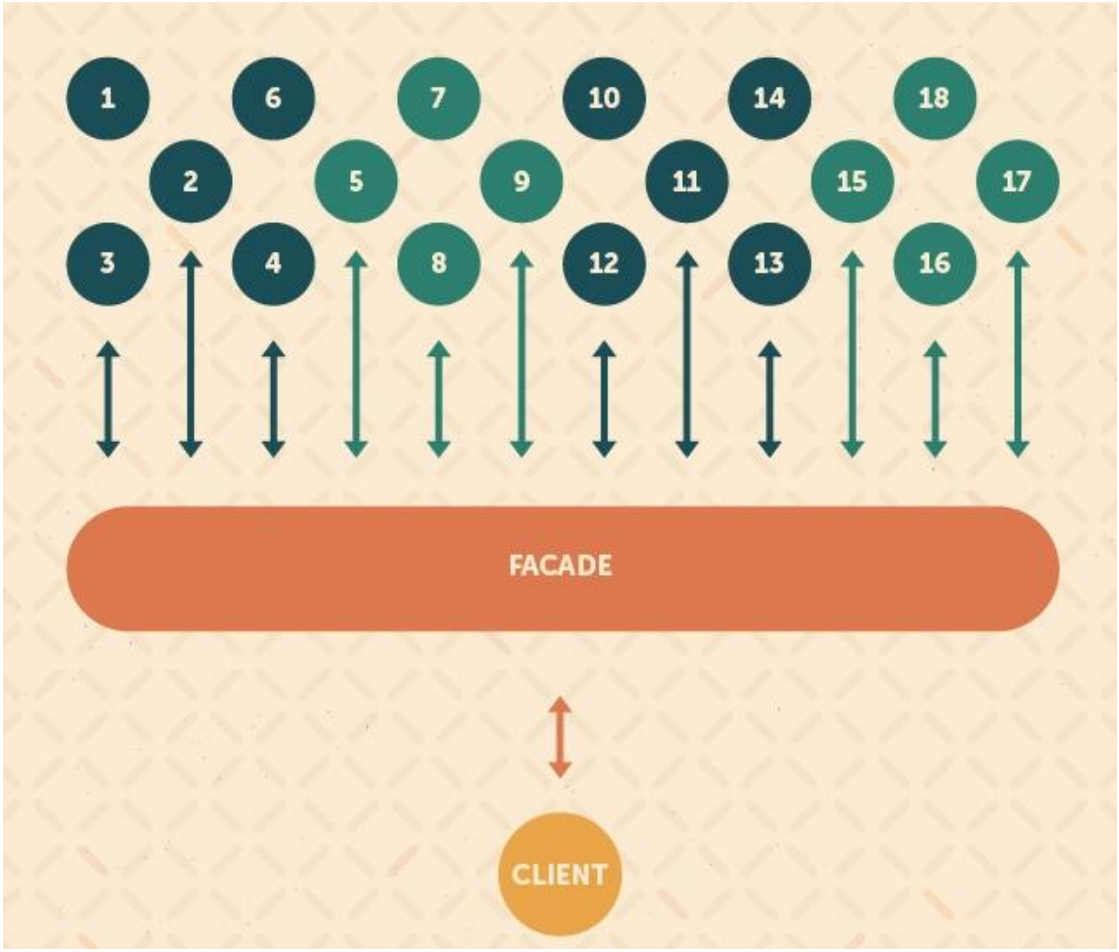
Возможности этой системы велики, но ее интерфейс слишком сложный.

Для удобства можно добавить новый объект, представляющий хорошо продуманные комбинации методов. Это и есть Фасад.

Python предлагает очень элегантную реализацию шаблона.

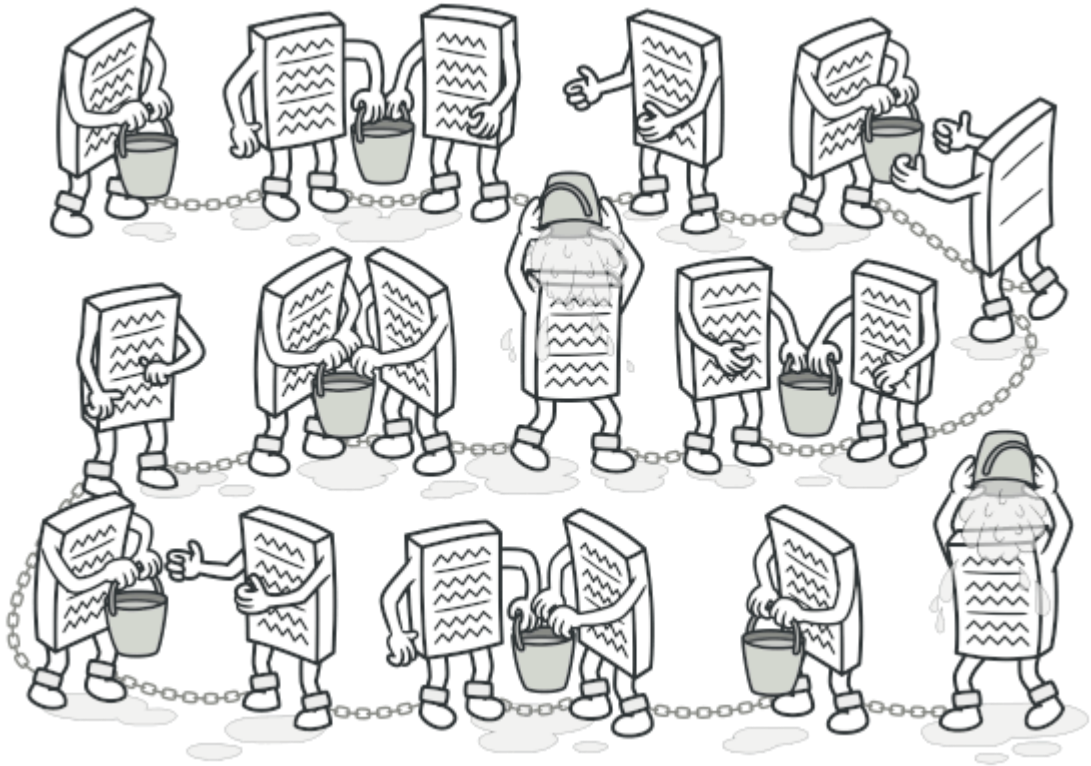
```
class Car(object):
    def __init__(self):
        self._tyres = [Tyre('front_left'),
                       Tyre('front_right'),
                       Tyre('rear_left'),
                       Tyre('rear_right'), ]
        self._tank = Tank(70)

    def tyres_pressure(self):
        return [tyre.pressure for tyre in self._tyres]

    def fuel_level(self):
        return self._tank.level
```

Без всяких трюков и фокусов класс Car стал Фасадом.



Chain of Responsibility

Цепочка обязанностей — это поведенческий паттерн проектирования, который позволяет передавать запросы последовательно по цепочке обработчиков. Каждый последующий обработчик решает, может ли он обработать запрос сам и стоит ли передавать запрос дальше по цепи.



Абстрактный класс

```
class Handler(ABC):
    """
    Интерфейс Обработчика объявляет метод построения цепочки обработчиков.
    Он также объявляет метод для выполнения запроса.
    """


    @abstractmethod
    def set_next(self, handler: Handler) -> Handler:
        pass

    @abstractmethod
    def handle(self, request) -> Optional[str]:
        pass
```

Базовый класс

```
class AbstractHandler(Handler):
    """
    Поведение цепочки по умолчанию может быть реализовано внутри базового класса  обработчика.
    """
    _next_handler: Handler = None

    def set_next(self, handler: Handler) -> Handler:
        self._next_handler = handler
        # Возврат обработчика отсюда позволит связать обработчики простым
        # способом, вот так:
        # monkey.set_next(squirrel).set_next(dog)
        return handler

    @abstractmethod
    def handle(self, request: Any) -> str:
        if self._next_handler:
            return self._next_handler.handle(request)

        return None
```

Конкретные реализации

#Все Конкретные Обработчики либо обрабатывают запрос, либо передают его следующему обработчику в цепочке.

```
class MonkeyHandler(AbstractHandler):
    def handle(self, request: Any) -> str:
        if request == "Banana":
            return f"Monkey: I'll eat the {request}"
        else:
            return super().handle(request)
class SquirrelHandler(AbstractHandler):
    def handle(self, request: Any) -> str:
        if request == "Nut":
            return f"Squirrel: I'll eat the {request}"
        else:
            return super().handle(request)
class DogHandler(AbstractHandler):
    def handle(self, request: Any) -> str:
        if request == "MeatBall":
            return f"Dog: I'll eat the {request}"
        else:
            return super().handle(request)
```

Что общего? Какой паттерн стоит рассмотреть?

Задача 1.

Преподаватель и ученики. Преподаватель дает задания и проверяет выполнение.

Задача 2.

Менеджер и команда программистов. Менеджер руководит проектом, раздает задания и проверяет их выполнение в срок.

Задача 27-1

Введите число n от 1 до 18.

Напечатайте квадратную матрицу, имитирующую Дартс.

Например для n = 5.

1 1 1 1 1

1 2 2 2 1

1 2 3 2 1

1 2 2 2 1

1 1 1 1 1

Задача 27-2

Необходимо реализовать класс Item, описывающий предмет, конструктор которого принимает три аргумента:

name — название предмета

price — цена предмета в рублях

quantity — количество предметов

При обращении к атрибуту name экземпляра класса Item будет возвращаться его название с заглавной буквы, а при обращении к атрибуту total — произведение цены предмета на его количество.

Задача 27-3

Дан список.

Посчитайте сколько в нем элементов, включая вложенные списки.

Например:

```
[]                  -->  0
[1, 2, 3]           -->  3
["x", "y", ["z"]]   -->  4
[1, 2, [3, 4, [5]]] -->  7
```