

# 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, 'ahc' 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 метод `Person`, определите в нем атрибут `self._age`

Используйте декоратор `@property` для определения методов `getter`, `setter`, `deleter`.

В методе `setter` определите проверку, что возраст не может быть меньше 1 и больше 100, при попытке установить такой возраст программа должна печатать «Недопустимый возраст».

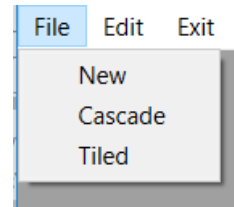
# Задача 26-4

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

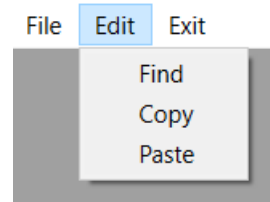
# Меню

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')
#exitAction = QAction(QIcon('t1.png'), 'Exit', self)
exitAction = QAction('Exit', self)
exitAction.triggered.connect(self.close)
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)
```

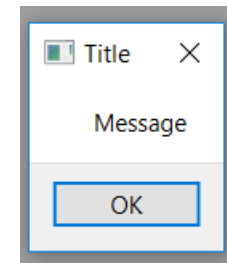
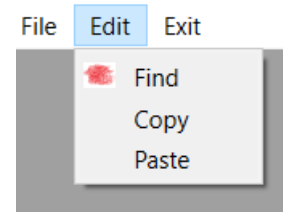
```
edit = bar.addMenu("Edit")
edit.addAction(editFindAction)
edit.addAction("Copy")
edit.addAction("Paste")
```

```
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")
```

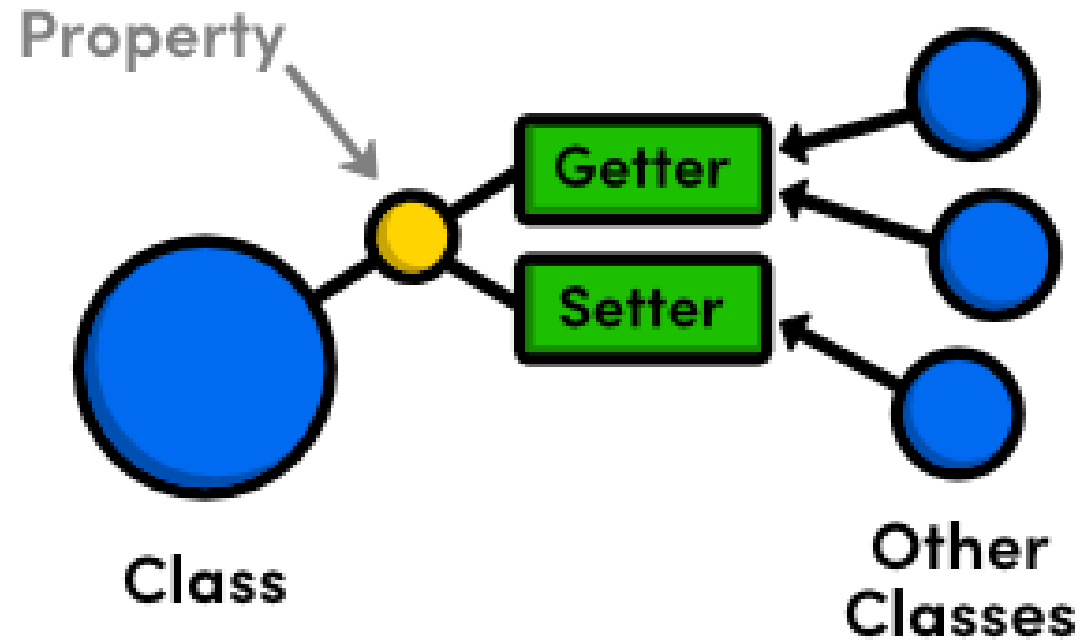
```
def clickFind(self):
    QMessageBox.about(self, "Title", "Message")
```



# 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

# \_\_getattr\_\_

```
class Cat:
```

```
    def __init__(self, name):
```

```
        self.name = name                # ИМЯ КОШКИ
```

```
    def __getattr__(self, attr):
```

```
        print(f'Возвращаю значение атрибута {attr}')
```

```
        return object.__getattr__(self, attr)
```

# обращение к словарю `__dict__` приведет к бесконечным рекурсивным вызовам метода `__getattr__()`, так как словарь `__dict__` сам является атрибутом.

```
cat = Cat('Мурка')
```

```
print(cat.name)
```

# \_\_getattr\_\_

Метод `__getattr__()` вызывается только в двух случаях:

- если в теле метода `__getattribute__()` было возбуждено исключение `AttributeError`
- если метод `__getattr__()` был явно вызван в теле метода `__getattribute__()`

```
class Cat:
```

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

```
    def __getattr__(self, attr):  
        print(f'Возвращаю значение по умолчанию')  
        return None
```

```
cat = Cat('Мурка')
```

```
print(cat.name)    # обращение к существующему атрибуту
```

```
print(cat.age)     # обращение к несуществующему атрибуту
```

```
print(cat.colour)  # обращение к несуществующему атрибуту
```

\_\_getattribute\_\_ и \_\_getattr\_\_

```
class Cat:
```

```
    def __init__(self, name):
```

```
        self.name = name
```

```
    def __getattribute__(self, attr):
```

```
        if attr in self.__dict__:
```

```
            return object.__getattribute__(self, attr)
```

```
        else:
```

```
            raise AttributeError
```

```
    def __getattr__(self, attr):
```

```
        if attr == 'age':
```

```
            return 123
```

```
        else:
```

```
            return None
```

```
c = Cat('Мурка')
```

```
print(c.name)
```

```
print(c.age)
```

```
print(c.color)
```

# 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

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

Подсказка:

```
def __init__(self, **kwargs):
```

```
# используйте self.__dict__ для определения атрибутов и их значений (из kwargs)
```

Экземпляр класса `AnyClass` должен иметь следующее неформальное строковое представление (метод `__str__`):

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

# Метаклассы

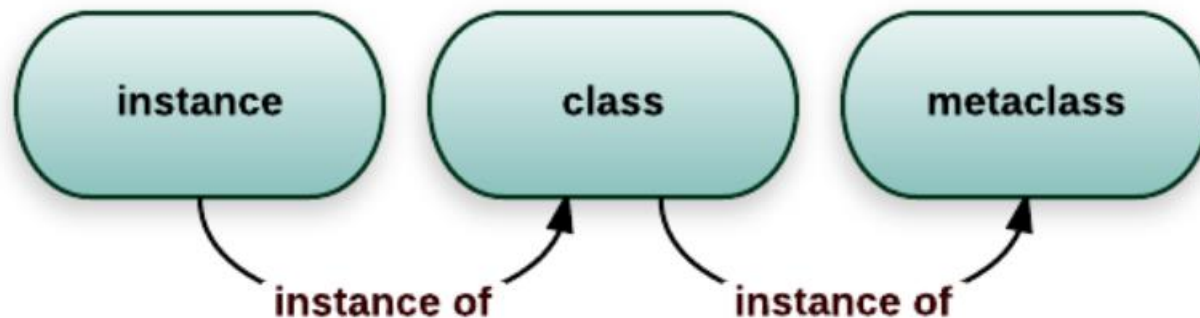
Мы можем добавлять методы и атрибуты из программы

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()

```
class Foo(object):
```

```
    bar = True
```

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

```
f = Foo()
```

```
def echo_bar(self):
```

```
    print(self.bar)
```

```
FooChild = type('FooChild', (Foo,), {'echo_bar': echo_bar})
```

```
print(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

Таким образом, кроме повторного использования кода, можно использовать структуры удачных программ.

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.

- 1.Порождающие** – предоставляют возможность создания контролируемым образом, инициализации и конфигурации объектов, классов и типов данных на основе требуемых критериев.
- 2.Структурные** – помогают организовать структуры связанных объектов и классов, предоставляя новые функциональные возможности.
- 3.Поведенческие** – направлены на выявление общих моделей взаимодействия между объектами.

# Для чего применяются паттерны ?

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

1. Порождающие шаблоны	2. Структурные шаблоны	3. Шаблоны поведения
<ul style="list-style-type: none"><li>1. Abstract Factory</li><li>2. Builder</li><li>3. Factory Method</li><li>4. Prototype</li><li>5. Singleton</li></ul>	<ul style="list-style-type: none"><li>1. Adapter</li><li>2. Bridge</li><li>3. Composite</li><li>4. Decorator</li><li>5. Façade</li><li>6. Flyweight</li><li>7. Proxy</li></ul>	<ul style="list-style-type: none"><li>1. Chain of Responsibility</li><li>2. Command</li><li>3. Interpreter</li><li>4. Iterator</li><li>5. Mediator</li><li>6. Memento</li><li>7. Observer</li><li>8. State</li><li>9. Strategy</li><li>10. Template Method</li><li>11. Visitor</li></ul>

**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.

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

# Singleton (Одиночка)

- Singleton — порождающий шаблон проектирования, гарантирующий, что в однопоточном приложении будет единственный экземпляр некоторого класса, и предоставляющий глобальную точку доступа к этому экземпляру.
- **Проще говоря:** Singleton гарантирует, что созданный объект будет единственным в приложении и позволяет получать к нему доступ из любой части приложения.
- **Еще проще:** хороший пример этого паттерна из жизни — это классный журнал в школе. У каждого класса он только один, и если учитель спросит журнал, то всегда получит один и тот же экземпляр.
- **Когда нужен:** пригодится, если в приложении есть управляющий объект, в котором хранится весь контекст приложения. Например, в сервисах хранения данных.

# Builder (Строитель)

- Builder — порождающий шаблон проектирования, который предоставляет способ создания составного объекта. Отделяет конструирование сложного объекта от его представления так, что в результате одного и того же процесса конструирования могут получаться разные представления.
- **Проще говоря**, Builder позволяет создавать разные объекты с заданным состоянием, используя один и тот же код.
- **Еще проще**: пример этого паттерна в жизни — покупка компьютера в магазине. При выборе мы указываем, какими характеристиками должна обладать техника (например, память 16 ГБ, процессор Intel core i7 и так далее). Таким образом, мы можем создавать различные виды одного объекта.
- **Когда нужен**: пригодится при составлении SQL-запроса, а также в юнит-тестах.
- **Ключевая идея паттерна** в том, чтобы вынести сложную логику создания объекта в отдельный класс «Строитель». Этот класс позволяет создавать сложные объекты поэтапно, на выходе получая готовый объект с нужными нам характеристиками и предоставляя классу упрощенный интерфейс для работы.

# Factory Method (Фабричный метод)

- Фабричный метод — порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать.
- **Проще говоря**, паттерн позволяет использовать один класс для создания объектов разной реализации интерфейса и делегировать логику дочерним классам.
- **Еще проще**: при заказе авиабилета мы указываем только информацию из паспорта, номер рейса и кресла. Такие данные, как номер терминала, время вылета, модель самолета инициализируются без нашего участия. Это экономит время пассажиров и сокращает количество ошибок.
- **Когда нужен**: паттерн подходит для ситуаций, когда нам необходимо выполнить действие (например, отправить запрос), но заранее неизвестны все данные, так как они зависят от входных параметров (например, от протокола передачи данных — rest, soap или socket).

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

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



# Facade (Фасад)

- Facade — структурный шаблон проектирования, позволяющий скрыть сложность системы путём сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы.
- **Проще говоря:** Facade предоставляет упрощенный интерфейс для сложной системы.
- **Еще проще:** При оплате покупки через Apple Pay вы подносите телефон к устройству и оплачиваете покупку. Кажется, что все просто. Но на самом деле внутри этого процесса происходит гораздо больше вещей. Этот упрощенный интерфейс называется фасадом.
- **Когда нужен:** Используется в библиотеках и позволяет описать их так, чтобы пользователю не нужно было вникать в их реализацию.

# 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.

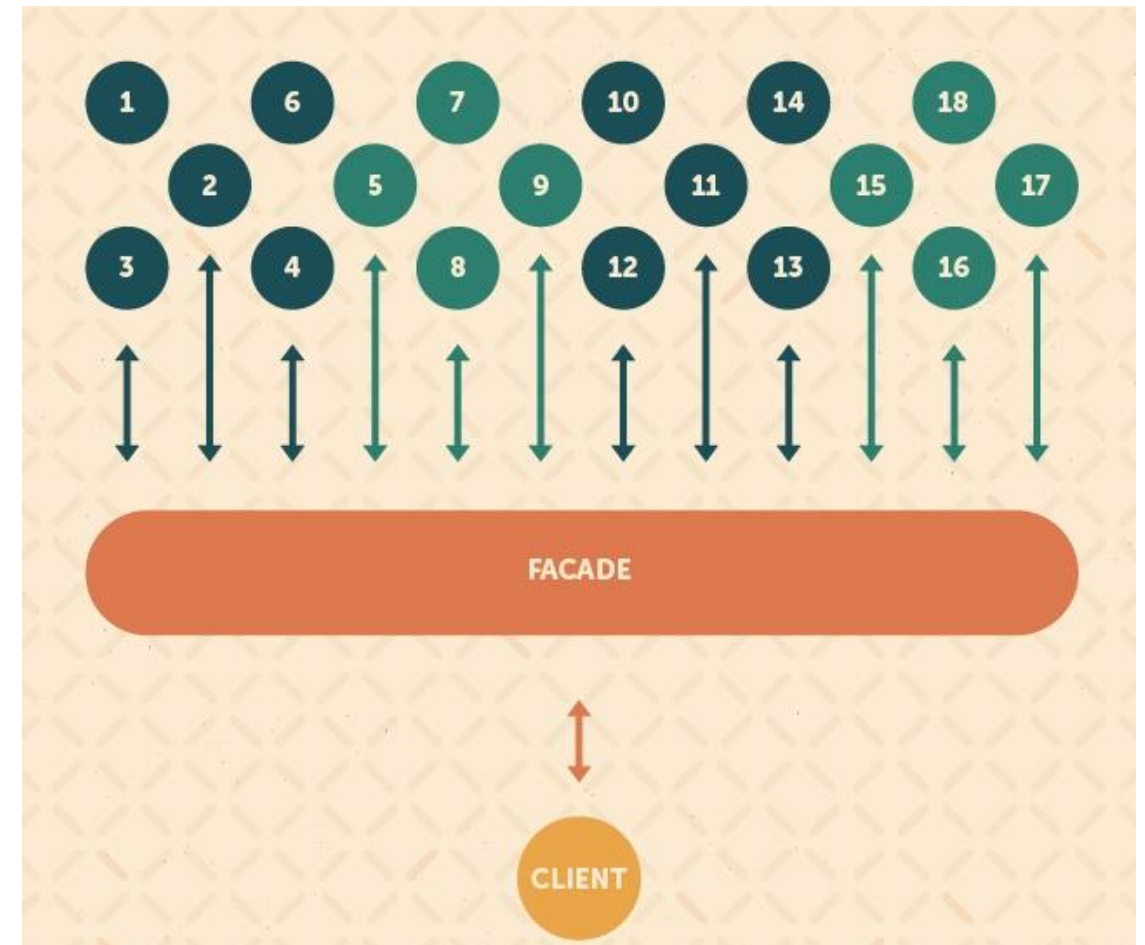
Python предлагает очень элегантную реализацию шаблона.

```
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.



# Adapter (Адаптер)

- Adapter — структурный шаблон проектирования, предназначенный для организации использования функций объекта, недоступного для модификации, через специально созданный интерфейс.
- **Проще говоря**, Adapter позволяет объектам с несовместимыми интерфейсами работать вместе.
- **Еще проще**: европейские розетки отличаются от английских, поэтому, приезжая в Лондон, туристы обязательно берут переходник, или адаптер.
- **Когда нужен**: часто используется, если мы работаем со сторонней библиотекой или компонентом, доступ к изменениям методов которого у нас отсутствует. Или когда есть несколько сторонних систем, доступ к которым должен осуществляться единообразно через одинаковый интерфейс.

Существует два варианта этого паттерна:

- предоставляет возможность работать при помощи интерфейса/протокола, который наши объекты должны реализовать;
- дает доступ при помощи отдельного класса-адаптера, если нам необходимо внедрить более сложную логику преобразований.

# Decorator (Декоратор)

- Decorator — структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту.
- **Проще говоря**, паттерн позволяет добавлять объектам новые функции с помощью обертки без создания отдельного класса.
- **Еще проще**: пример декоратора из жизни — это подключение мышки к ноутбуку, то есть вы добавляете новую функцию устройству, не меняя его.
- **Когда нужен**: у этого паттерна широкая область применения. Им пользуются каждый раз, когда нужно добавить логику в уже созданные объекты и библиотеки, менять которые нельзя.

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

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

# Template Method (Шаблонный метод)

- Template Method — поведенческий шаблон проектирования, определяющий основу алгоритма и позволяющий наследникам переопределять некоторые шаги алгоритма, не изменяя его структуру в целом.
- **Проще говоря**, Template Method или шаблонный метод описывает скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.
- **Еще проще:** Строители используют подход, похожий на шаблонный метод, при строительстве типовых домов. У них есть основной архитектурный проект, в котором расписаны шаги строительства: заливка фундамента, постройка стен, перекрытие крыши, установка окон и так далее. Но, несмотря на стандартизацию каждого этапа, строители могут вносить небольшие изменения на любом из этапов, чтобы сделать дом чуточку непохожим на другие.
- **Когда нужен:** используется для избежания дублирования кода. Например, при описании шаблонов.

# Iterator (Итератор)

- Iterator — поведенческий шаблон проектирования. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов.
- **Проще говоря**, Iterator нужен для последовательного перебора составных элементов коллекции, не раскрывая их внутреннего представления.
- **Еще проще**: представим, что мы пошли в лес с множеством красивых мест. Мы хотим посетить конкретные места, например, озеро и поляну цветов, но в лесу очень просто заблудиться. У нас есть несколько вариантов, как дойти до мест: можем воспользоваться бумажной картой или навигатором, попробовать найти все места самим или попросить лесника провести нас. В данном примере, лес — это коллекция живописных мест, а телефон, карта, лесник и наш мозг — итераторы по коллекции «лес».
- **Когда нужен**: Iterator стоит использовать для перебора сложных составных объектов коллекции, если необходимо скрыть детали реализации от клиента.



# Observer (Наблюдатель)

- Observer — поведенческий шаблон проектирования. Реализует у класса механизм, который позволяет объекту этого класса получать оповещения об изменении состояния других объектов и тем самым наблюдать за ними.
- **Проще говоря**, Observer позволяет одному объекту подписываться на другие объекты и отслеживать их изменения
- **Еще проще**: после того как вы оформили подписку на газету или журнал, вам больше не нужно ездить в супермаркет и проверять, вышел ли очередной номер. Издательство будет присылать новые номера по почте прямо к вам домой сразу после их выхода.
- **Когда нужен**: часто используется при проектировании библиотек, которые управляют состоянием приложения. Также является основой для различных классов, способных реагировать на изменения и уведомлять своих подписчиков об этих изменениях: например, [NotificationCenter](#) в iOS или [LiveData](#) в Android.



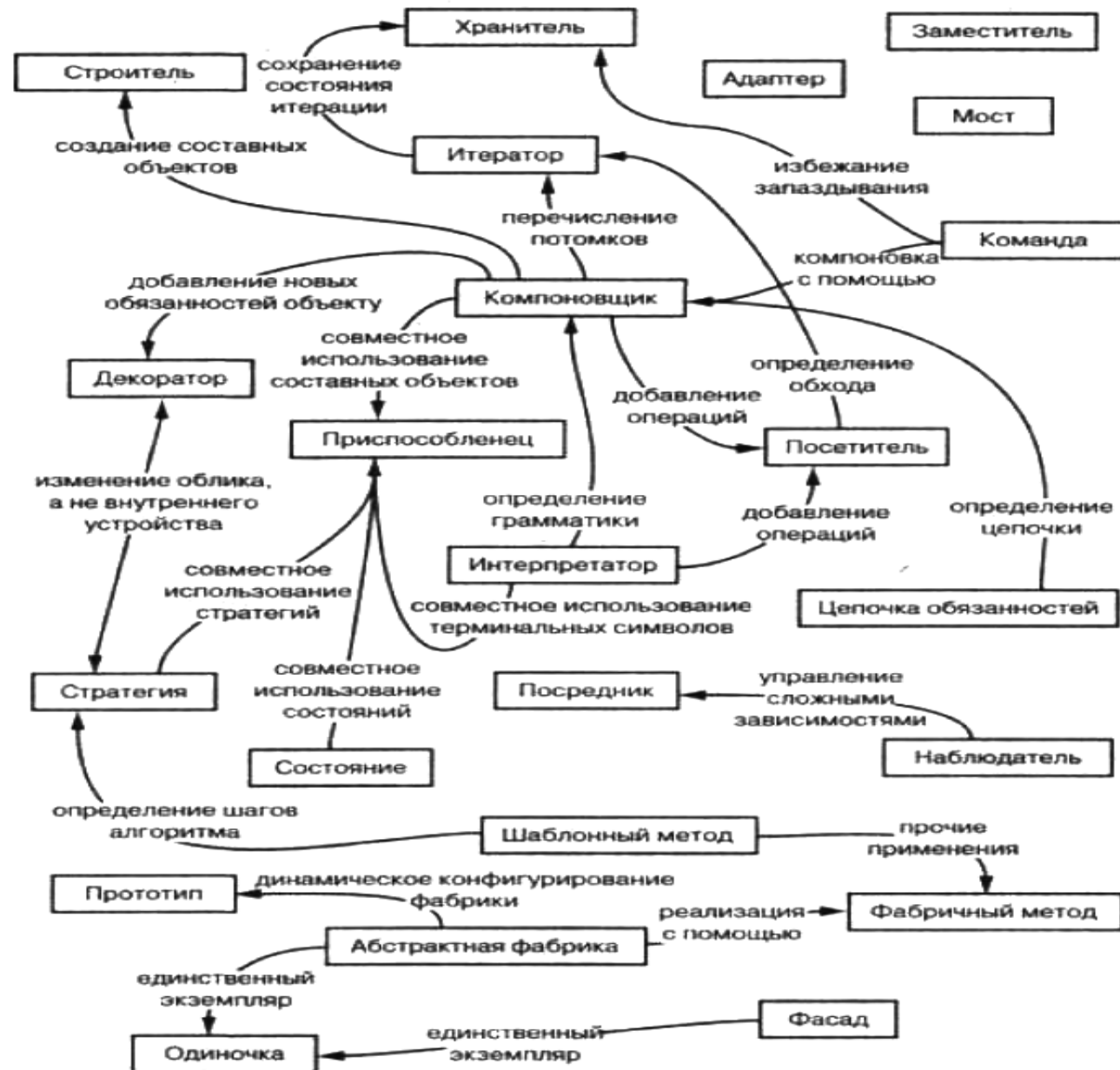


Рис. 1.1. Отношения между паттернами проектирования

# 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 (
- 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

# Прототип

Шаблон «Прототип» используется, когда типы создаваемых объектов определяются экземпляром-прототипом, клонированным для создания новых объектов.

То есть шаблон позволяет дублировать существующий объект и модифицировать копию в соответствии с потребностями. Без заморочек с созданием объекта с нуля и его настройкой.

```
class Person:
    def __init__(self, name, address):
        self.name = name
        self.address = address
    def __str__(self): return f"Person {self.name} from {self.address}"

john = Person('John', ['Nevsky', 25] )
print(john)
jane = john
print(jane)
jane.name = 'Jane'
print(jane)
jane.address = ['Liteiny', 44]
print(jane)
print(john)
```

# Хорошо ли это?

# Что напечатает?

# Прототип

```
import copy
```

```
class Person:
```

```
    def __init__(self, name, address):
```

```
        self.name = name
```

```
        self.address = address
```

```
    def __str__(self): return f"Person {self.name} from {self.address}"
```

```
john = Person('John', ['Nevsky', 25] )
```

```
print(john)
```

```
jane = copy.deepcopy(john) # Глубокая копия
```

```
print(jane)
```

```
jane.name = 'Jane'
```

```
print(jane)
```

```
jane.address = ['Liteiny', 44]
```

```
print(jane)
```

```
print(john)
```

# 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()
```

```
print(Singleton.instance)
```

```
s2 = Singleton()
```

```
print(s1 is s2) # True
```

```
s1.x = 123
```

```
print(s2.x) # 123
```

```
# Выполните этот код, убедитесь, что это один объект
```



# ABC

```
import abc
```

```
class Parent(abc.ABC):
```

```
    @abc.abstractmethod
```

```
    def qq(self):
```

```
        pass
```

```
class Child(Parent):
```

```
    def qq(self):
```

```
        return "күкү"
```

```
c = Child()
```

```
print(c.qq())
```

# ABC

```
from abc import ABC, abstractmethod
```

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

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

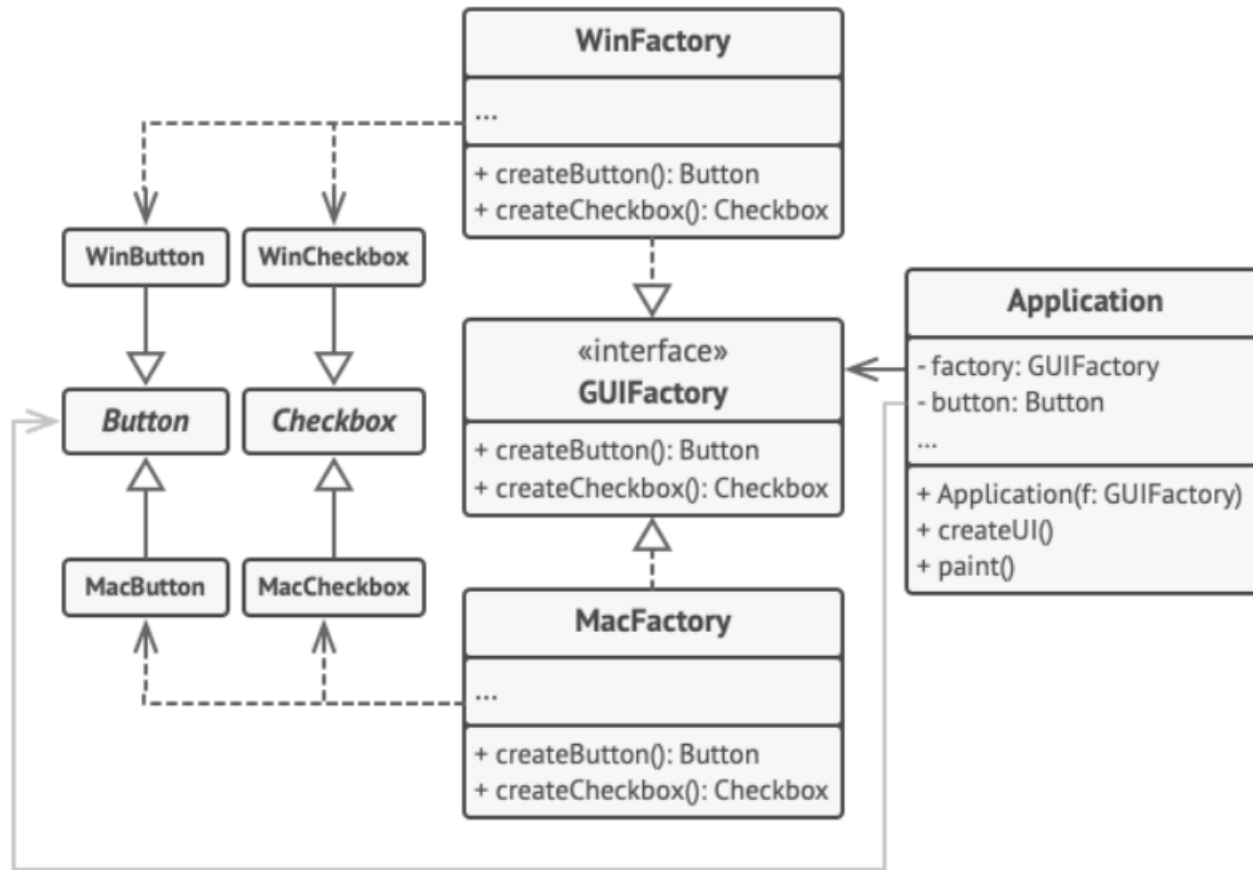
```
class Cat(Animal):  
    def speak(self):  
        return "Meow!"
```

# Абстрактная фабрика

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

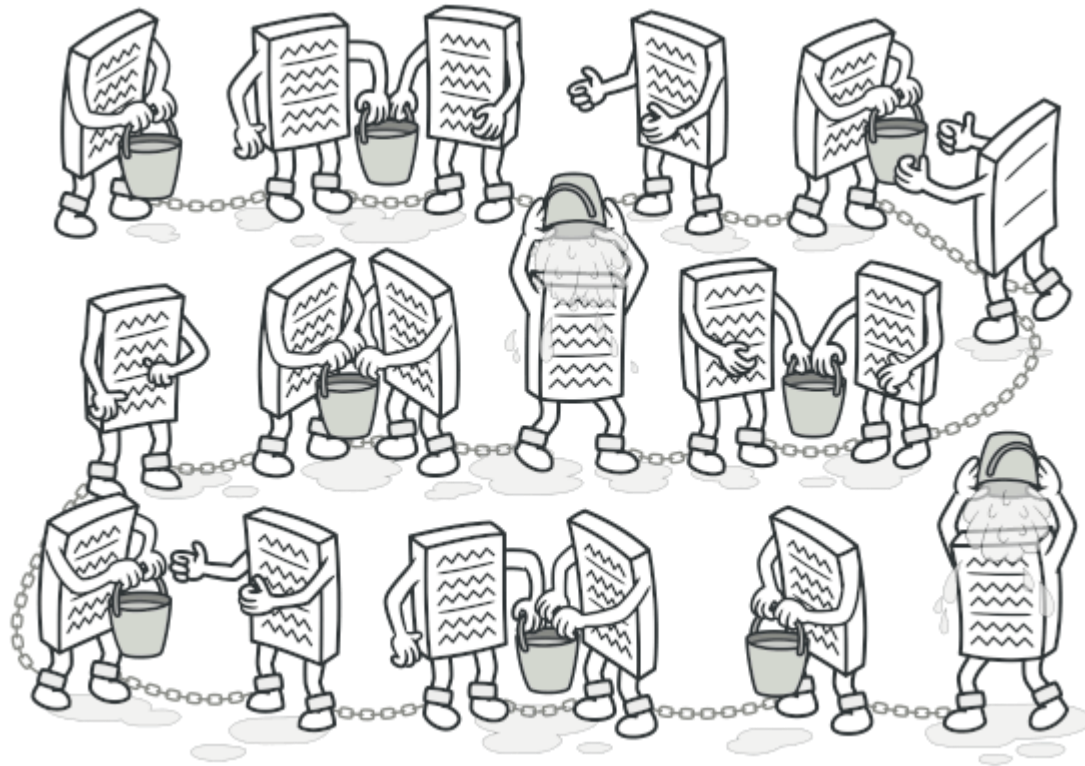


# На примере графического интерфейса



# 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