Патерни проектирования

План

0. Источники

1. Патерни Проектирования

2. Группи патернов

3. [Порождающие (Creational)](http://pythonicway.com/education/basics/36-design-patterns-python" \l "Creational)

4. [Структурные (Structural)](http://pythonicway.com/education/basics/36-design-patterns-python" \l "Structural)

5. [Поведенческие (Behavioral)](http://pythonicway.com/education/basics/36-design-patterns-python" \l "Behavioral)

0. Источники

1. <http://pythonicway.com/education/basics/36-design-patterns-python>

2. https://refactoring.guru/ru/design-patterns

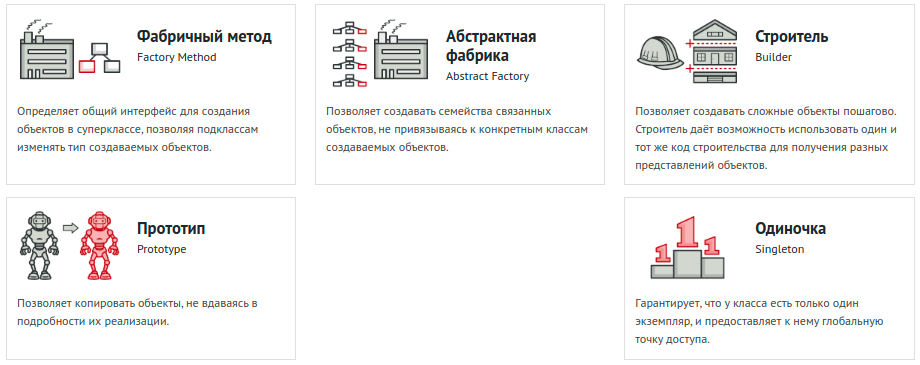
1. Патерни Проектирования

Паттерны проектирования это готовые эффективные решения для задач, которые часто встречаются в практике программиста. Это шаблон решения, по которому вы сможете решить возникшую проблему, внеся небольшие частные изменения в существующий паттерн.

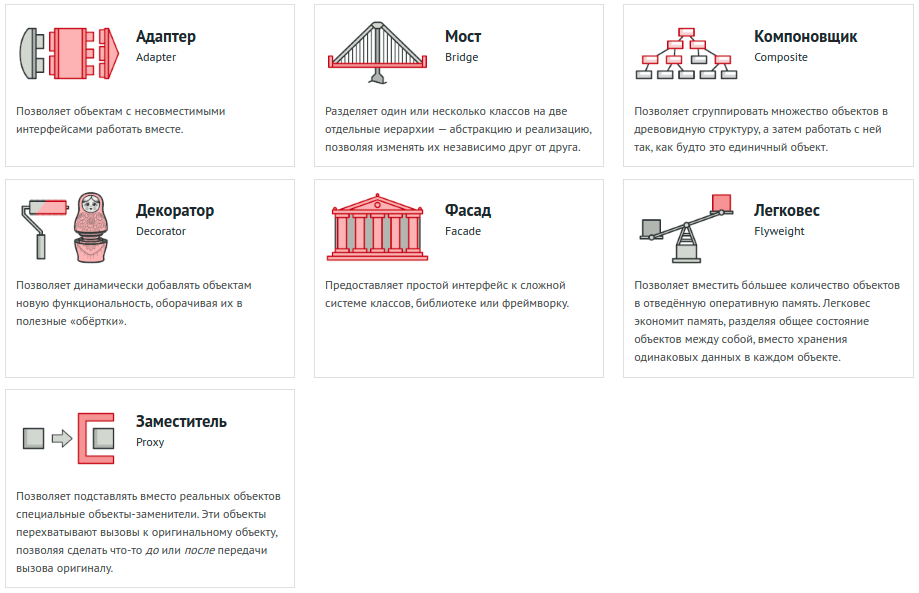
Например, если мы производим автомобили, нам незачем каждый раз изобретать колесо или топливную систему, достаточно посмотреть, как они реализованы в других автомобилях и адаптивать их под свои нужды. Шаблоны проектирования не зависят от языка программирования, соответственно любой из них может быть реализован не только в Python, но и в любом другом языке программирования.

Правильное применение шаблонов проектирования может в значительной мере упростить разработку программы и ее дальнейшую поддержку и расширение. В то же время, бездумное использование паттернов проектирования лишь без надобности усложнит и перегрузит созданную систему.

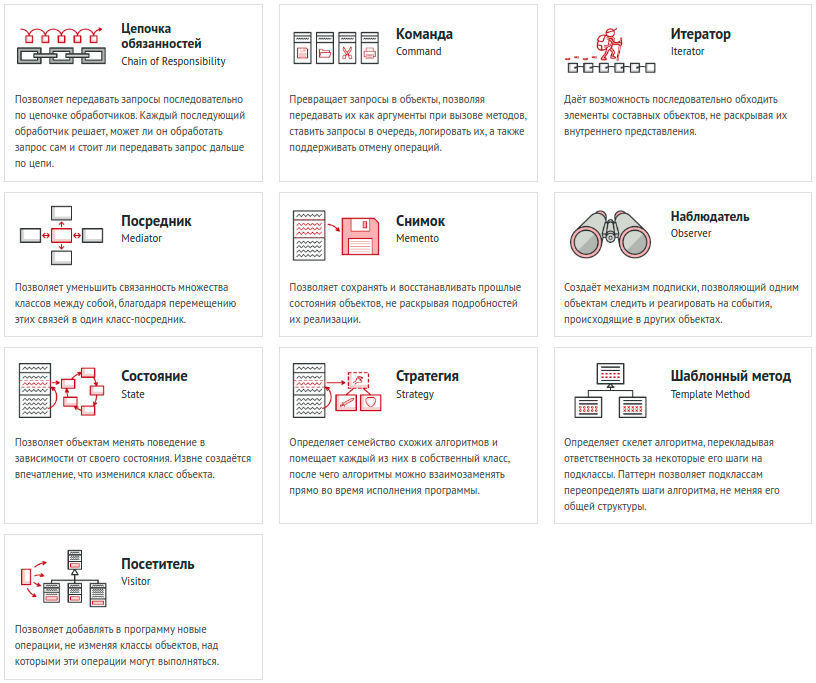
2. Группи патернов

Паттерны отличаются предназначением. Существует три основные группы паттернов:

* [Порождающие (Creational)](http://pythonicway.com/education/basics/36-design-patterns-python" \l "Creational)
* [Структурные (Structural)](http://pythonicway.com/education/basics/36-design-patterns-python" \l "Structural)



* [Поведенческие (Behavioral)](http://pythonicway.com/education/basics/36-design-patterns-python" \l "Behavioral)



3. [Порождающие (Creational)](http://pythonicway.com/education/basics/36-design-patterns-python" \l "Creational)

**Порождающие паттерны** беспокоятся о гибком создании объектов без внесения в программу лишних зависимостей. Они призваны обеспечить ситуативную гибкость при создании объекта. К примеру с машинами, сделать так, чтобы нам не нужно было строить новый завод каждый раз, когда мы хотим выпустить новую модель, вместо этого мы слегка видоизменим уже существующие процессы.

|  |  |  |
| --- | --- | --- |
| Абстрактная фабрика | Abstract Factory | Позволяет создавать семейства взаимосвязанных или взаимозависимых объектов, без указания их конкретных классов. |
| Строитель | Builder | Интерфейс для пошагового создания сложных объектов. |
| Фабричный метод | Factory Method | Общий интерфейс для создания объектов в суперклассе, позволяющий подклассам определять тип создаваемого объекта. |
| Объектный пул | Object Pool | Позволяет использовать уже созданный объект вместо создания нового, в ситуации, когда создание нового объекта требует большого количества ресурсов. |
| Прототип | Prototype | Позволяет копировать объекты без необходимости учитывать особенности их реализации. |
| Одиночка | Singleton | Гарантирует, что у класса есть только один экземпляр и предоставляет глобальную точку доступа к нему. |
| Отложенная инициализация | Lazy initialization | Создание объекта, непосредственно перед его использованием. |
| Мультитон | Multiton | Шаблон позволяющий создавать несколько одиночек (Singleton), доступ и управление которыми производится через ассоциативную таблицу, например словарь. |

4. [Структурные (Structural)](http://pythonicway.com/education/basics/36-design-patterns-python" \l "Structural)

**Структурные паттерны** показывают различные способы построения связей между объектами. Структурные шаблоны проектирования определяют метод сборки объектов и классов в более сложные структуры, сохраняя при этом гибкость и эффективность. Для машины это будут перечень деталей, схемы и порядок сборки автомобиля.

|  |  |  |
| --- | --- | --- |
| Адаптер | Adapter | Создание объекта-посредника, который позволит взаимодействовать двум несовместимым объектам. |
| Мост | Bridge | Разделяет класс на отдельные части: внешнюю абстракцию и внутреннюю реализацию. |
| Компоновщик | Composite | Идея состоит в том, что группа объектов (контейнер) и сам объект (содержимое контейнера) обладают тем же набором свойств, что позволяет работать с группой как с целым объектом. |
| Декоратор | Decorator | Добавляет, убирает или изменяет поведение декорированного объекта. |
| Фасад | Facade | Обертка сложной системы, модуля, пакета в простой интерфейс. |
| Приспособленец | Flyweight | Использование совместных ресурсов для похожих объектов, вместо выделения ресурсов для каждого объекта по отдельности. |
| Прокси | Proxy | Создание объекта-подложки для реального объекта, чтобы контролировать обращения к нему, изменять или перенаправлять их. |

5. [Поведенческие (Behavioral)](http://pythonicway.com/education/basics/36-design-patterns-python" \l "Behavioral)

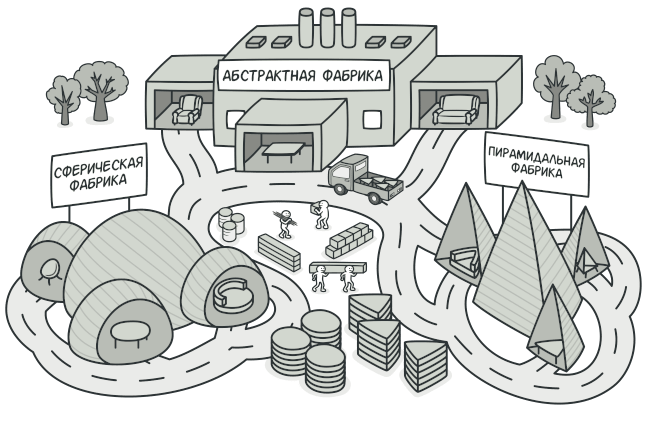
**Поведенческие паттерны** заботятся об эффективной коммуникации между объектами. Поведенческие шаблоны определяют взаимодействие между классами и объектами, их обязанностями и алгоритмы поведения. На автозаводе это будут логистика и распределение обязанностей, построение коммуникации между разными департаментами и отделами.

|  |  |  |
| --- | --- | --- |
| Цепочка обязанностей | Chain of responsibility | Последовательная передача запросов по списку объектов, которые эти запросы обрабатывают и/или передают дальше по цепочке. |
| Итератор | Iterator | Позволяет последовательно получать объекты из контейнера, не раскрывая особенности реализации контейнера. В Python доступен на встроенном уровне. |
| Команда | Command | Добавляет слой абстракции между действием и объектом, который это действие вызывает, например, кнопка и действие, которое выполняется при нажатии на эту кнопку. |
| Посредник | Mediator | Создание такой структуры, в которой объекты не общаются друг с другом, а используют для этого объект-посредник. |
| Хранитель | Memento | Сохраняет состояние объекта на определенный момент для того, чтобы при необходимости к нему можно было вернуться. |
| Null Object | Null Object | Объект который может использоваться в случае отсутствия нужного объекта или объект по умолчанию. |
| Наблюдатель | Observer | Объект "наблюдающий" за состоянием других объектов, информирующий систему / пользователя про изменения состояния наблюдаемого объекта, например пуш-извещения. |
| Состояние | State | Позволяет изменять поведение объекта в зависимости от его состояния. |
| Стратегия | Strategy | Позволяет объединить несколько алгоритмов в группу. Порядок применения алгоритмов может изменяться, благодаря чему достигается гибкость всей системы. |
| Шаблонный метод | Template method | Создание базовых методов и алгоритма их применения в абстрактном родительском классе с тем, чтобы определить конкретные методы в дочерних классах. |
| Посетитель | Visitor | Шаблон, позволяющий выполнять операции над другими объектами, без необходимости изменять эти объекты. |

Параждающий Патерн "Абстрактная фабрика"

<https://refactoring.guru/ru/design-patterns/abstract-factory>

**Абстрактная фабрика** — это порождающий паттерн проектирования, который решает проблему создания целых семейств связанных продуктов, без указания конкретных классов продуктов.



## **Применимость**

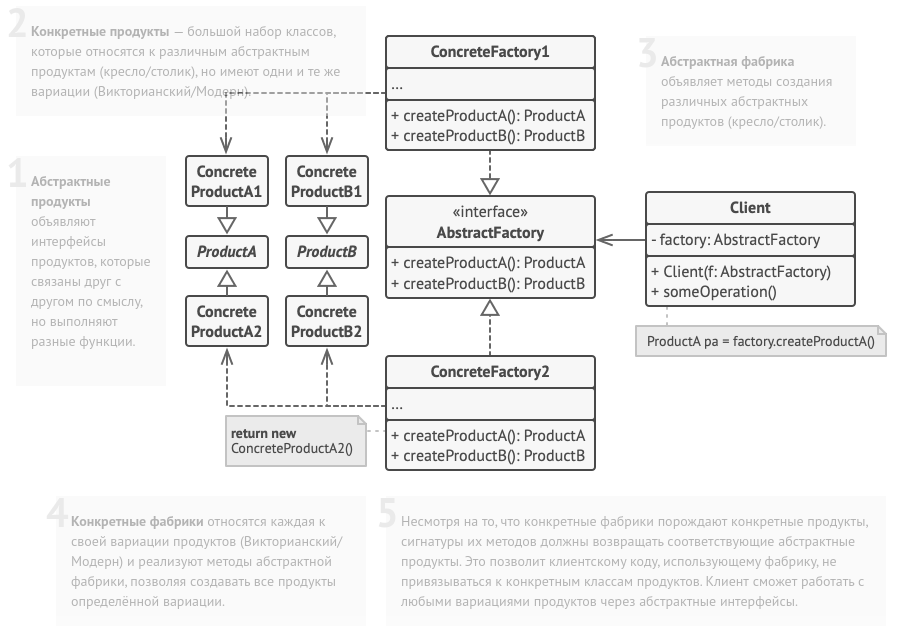
**Когда бизнес-логика программы должна работать с разными видами связанных друг с другом продуктов, не завися от конкретных классов продуктов.**

Абстрактная фабрика скрывает от клиентского кода подробности того, как и какие конкретно объекты будут созданы. Но при этом клиентский код может работать со всеми типами создаваемых продуктов, поскольку их общий интерфейс был заранее определён.

**Когда в программе уже используется** [**Фабричный метод**](https://refactoring.guru/ru/design-patterns/factory-method)**, но очередные изменения предполагают введение новых типов продуктов.**

В хорошей программе каждый класс отвечает только за одну вещь. Если класс имеет слишком много фабричных методов, они способны затуманить его основную функцию. Поэтому имеет смысл вынести всю логику создания продуктов в отдельную иерархию классов, применив абстрактную фабрику.

**Структура**



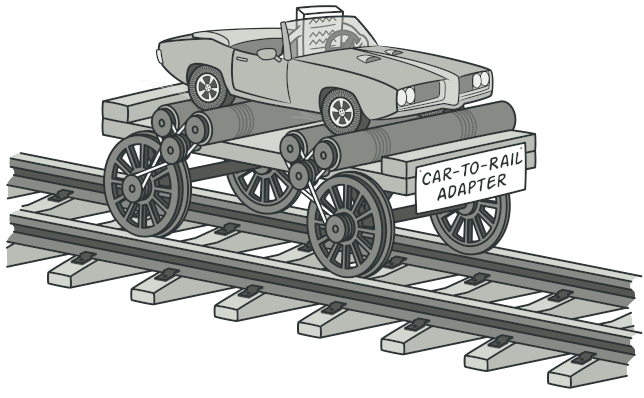
## **Преимущества и недостатки**

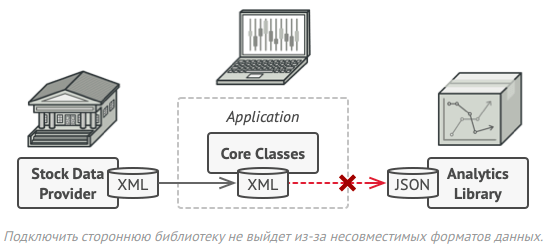
|  |  |
| --- | --- |
| **Преимущества** | **Hедостатки** |
| Гарантирует сочетаемость создаваемых продуктов | Усложняет код программы из-за введения множества дополнительных классов. |
| Избавляет клиентский код от привязки к конкретным классам продуктов. | Требует наличия всех типов продуктов в каждой вариации. |
| Выделяет код производства продуктов в одно место, упрощая поддержку кода. |  |
| Упрощает добавление новых продуктов в программу. |  |
| Реализует принцип открытости/закрытости. |  |

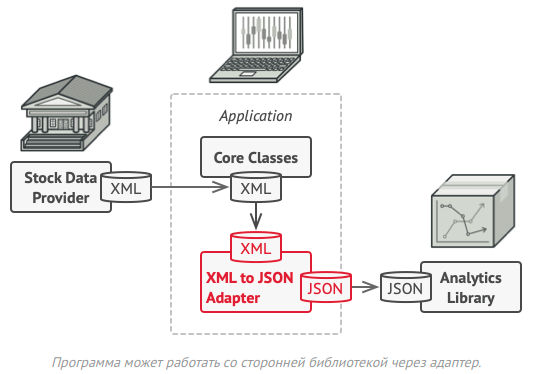
Структурний патерн "Адаптер"

https://refactoring.guru/ru/design-patterns/adapter

**Адаптер** — это структурный паттерн проектирования, который позволяет объектам с несовместимыми интерфейсами работать вместе.







## **Структура**

## **Применимость**

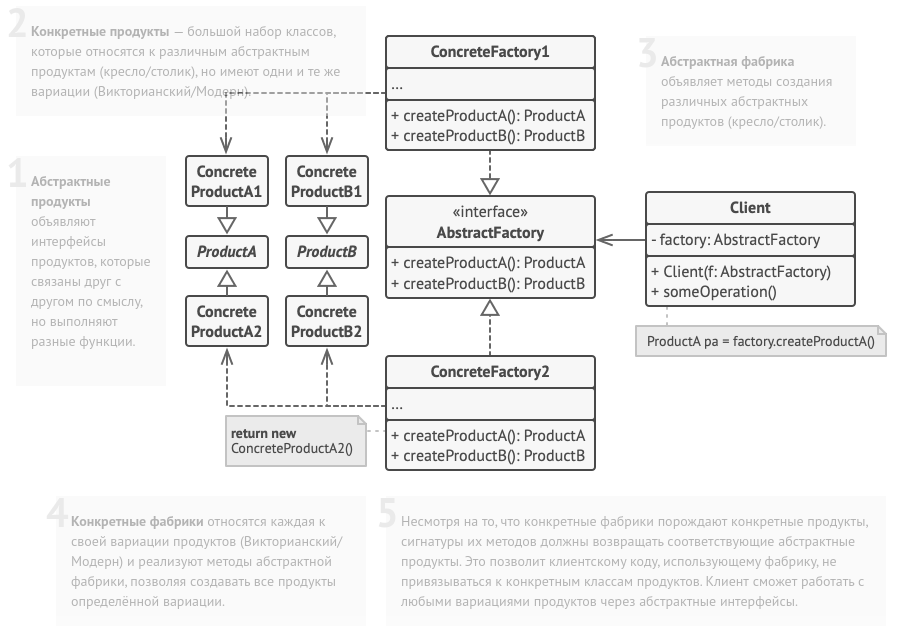
**Когда вы хотите использовать сторонний класс, но его интерфейс не соответствует остальному коду приложения.**

Адаптер позволяет создать объект-прокладку, который будет превращать вызовы приложения в формат, понятный стороннему классу.

**Когда вам нужно использовать несколько существующих подклассов, но в них не хватает какой-то общей функциональности, причём расширить суперкласс вы не можете.**

Вы могли бы создать ещё один уровень подклассов и добавить в них недостающую функциональность. Но при этом придётся дублировать один и тот же код в обеих ветках подклассов.

Более элегантным решением было бы поместить недостающую функциональность в адаптер и приспособить его для работы с суперклассом. Такой адаптер сможет работать со всеми подклассами иерархии. Это решение будет сильно напоминать паттерн [**Декоратор**](https://refactoring.guru/ru/design-patterns/decorator).



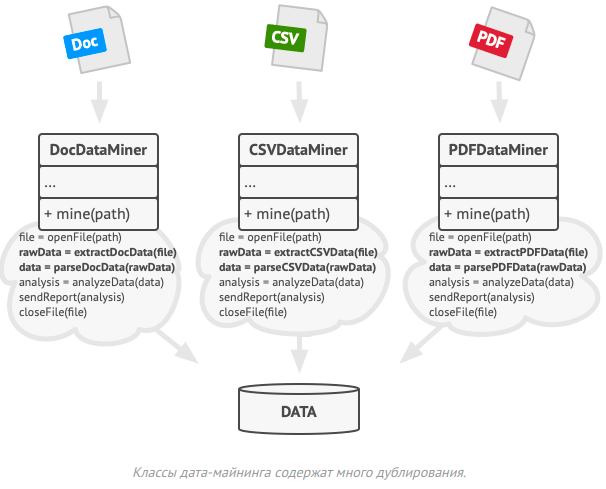
## **Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества** | **Hедостатки** |
| Отделяет и скрывает от клиента подробности преобразования различных интерфейсов. | Усложняет код программы из-за введения дополнительных классов. |

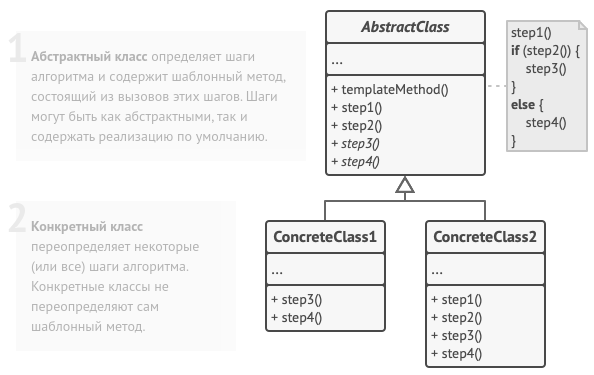
Поведенческий метод "Шаблонный метод"

<https://refactoring.guru/ru/design-patterns/template-method>

**Шаблонный метод** — это поведенческий паттерн проектирования, который определяет скелет алгоритма, перекладывая ответственность за некоторые его шаги на подклассы. Паттерн позволяет подклассам переопределять шаги алгоритма, не меняя его общей структуры.



Структура



## **Применимость**

**Когда подклассы должны расширять базовый алгоритм, не меняя его структуры.**

Шаблонный метод позволяет подклассам расширять определённые шаги алгоритма через наследование, не меняя при этом структуру алгоритмов, объявленную в базовом классе.

**Когда у вас есть несколько классов, делающих одно и то же с незначительными отличиями. Если вы редактируете один класс, то приходится вносить такие же правки и в остальные классы.**

Паттерн шаблонный метод предлагает создать для похожих классов общий суперкласс и оформить в нём главный алгоритм в виде шагов. Отличающиеся шаги можно переопределить в подклассах. Это позволит убрать дублирование кода в нескольких классах с похожим поведением, но отличающихся в деталях.

## **Преимущества и недостатки**

|  |  |
| --- | --- |
| **Преимущества** | **Hедостатки** |
| Облегчает повторное использование кода. | Вы жёстко ограничены скелетом существующего алгоритма. |
|  | Вы можете нарушить принцип подстановки Барбары Лисков, изменяя базовое поведение одного из шагов алгоритма через подкласс. |
|  | С ростом количества шагов шаблонный метод становится слишком сложно поддерживать. |