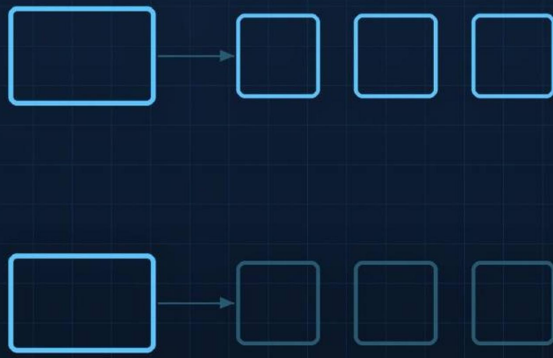
GoF · 01/23

ПОРОЖДАЮЩИЙ

Python C++23 Rust Java



Абстрактная фабрика

Abstract Factory

Алексей Примак

Алексей Примак

Абстрактная фабрика. Паттерн GoF на Python, C++23, Rust и Java

https://litres.ru/74810679
SelfPub; 2026

Аннотация

Как создавать согласованные семейства объектов и не связывать клиентский код с конкретными классами? «Абстрактная фабрика» разбирает паттерн GoF на понятных задачах и схемах, а затем показывает его реализацию на Python, C++23, Rust и Java. В книге рассмотрены классическая структура, практические варианты, модификации, гибридные решения и самостоятельные задания. Подойдёт как для систематизации знаний, так и для практической работы с паттернами проектирования.

Содержание

Глава 1. Реализация на Python 4
Об этой книге 4
1.1. Зачем нужна абстрактная фабрика 5
1.2. Участники и схема 6
1.3. Когда применять и какие последствия 8
1.4. Классическая реализация: светлая и тёмная темы 9
Конец ознакомительного фрагмента. 14

Алексей Примак
Абстрактная фабрика. Паттерн GoF на Python, C++23, Rust и Java

Глава 1. Реализация на Python

Об этой книге

Эта книга посвящена одному паттерну проектирования — **Абстрактная фабрика**(Abstract Factory). Он относится к каталогу «Банды четырёх» (Gang of Four, GoF). Здесь паттерн разобран сразу в четырёх реализациях: на Python, C++23, Rust и Java. Каждая реализация — отдельная глава: сначала задача и общая схема, затем код, модификации, гибридные варианты и самостоятельные работы.

Основы каждого языка здесь не повторяются: они собраны в книгах серий по соответствующему языку. Разделы каждой главы нумеруются заново, поэтому ссылка «раздел 1.2» указывает на раздел той же главы, где она сделана.

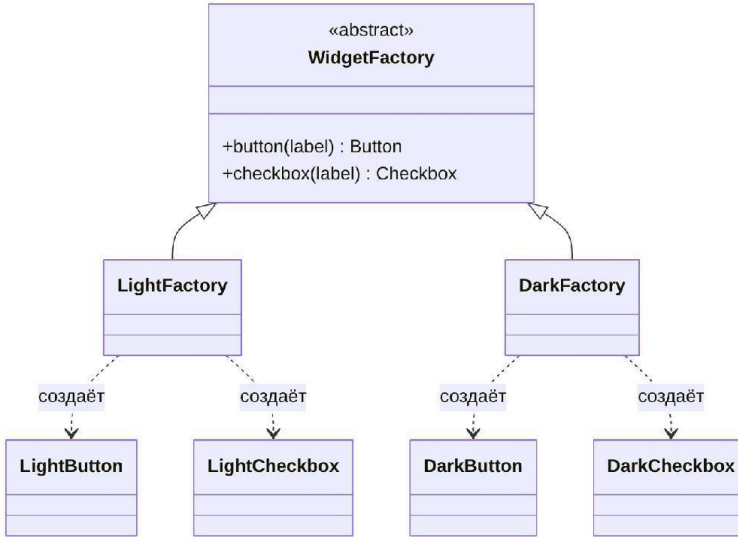
1.1. Зачем нужна абстрактная фабрика

Приложение с графическим интерфейсом работает в светлой и тёмной теме. В каждой теме свои кнопки, поля ввода и флажки. Если код окна сам создаёт LightButton и DarkCheckbox, рано или поздно получится смесь: светлая кнопка рядом с тёмным флажком. А добавление третьей темы потребует найти каждое место, где создаётся элемент.

Абстрактная фабрика собирает создание всех элементов одной темы в одном объекте. Клиентский код получает фабрику и просит у неё «кнопку» и «флажок», не зная, какие именно классы вернутся. Подменить тему — значит подменить одну фабрику, а совместимость элементов гарантирована: одна фабрика создаёт только элементы своего семейства.

Ключевое слово — **семейство**. Если нужно создавать один вид продукта, хватит фабричного метода или простой функции. Абстрактная фабрика нужна, когда продуктов несколько и они должны сочетаться.

1.2. Участники и схема



AbstractFactory(WidgetFactory) — интерфейс с методом создания для каждого вида продукта.

ConcreteFactory— реализация для одного семейства.

AbstractProduct(Button, Checkbox) — интерфейсы продуктов.

ConcreteProduct— конкретные классы.

Client— код, который работает только с интерфейсами и получает фабрику снаружи.

В Python интерфейсы можно описать абстрактными базовыми классами (abc.ABC) — тогда фабрика без нужного метода не создастся — или протоколами (typing.Protocol) — тогда подходит любой объект с нужными методами. А поскольку классы в Python сами являются объектами, фабрикой может быть даже словарь классов.

1.3. Когда применять и какие последствия

Паттерн уместен, когда система работает с несколькими семействами связанных продуктов, выбор семейства делается в одном месте (при запуске, по настройке, в тестах), а клиентский код не должен зависеть от конкретных классов.

Плюсы: семейство меняется подменой одной фабрики, совместимость продуктов обеспечена конструкцией. Минус: добавить **новый вид продукта** трудно — нужно расширить интерфейс фабрики и все реализации. Добавить **новое семейство** легко.

1.4. Классическая реализация: светлая и тёмная темы

Файл 01_classic.py:

01_classic.py Python

```python
from abc import ABC, abstractmethod


class Button(ABC):
    def __init__(self, label: str) -> None:
        self.label = label

    @abstractmethod
    def render(self) -> str: ...


class Checkbox(ABC):
    def __init__(self, label: str) -> None:
        self.label = label

    @abstractmethod
    def render(self) -> str: ...


class LightButton(Button):
    def render(self) -> str:
        return f"[ {self.label} ] (светлая)"


class LightCheckbox(Checkbox):
    def render(self) -> str:
        return f"[x] {self.label} (светлый)"


class DarkButton(Button):
    def render(self) -> str:
        return f"< {self.label} > (тёмная)"


```

Листинг 1.1 *— Классическая реализация: светлая и тём-*

ная темы

01_classic.py (продолжение) Python

```python
class DarkCheckbox(Checkbox):
    def render(self) -> str:
        return f"(*) {self.label} (тёмный)"


class WidgetFactory(ABC):
    @abstractmethod
    def button(self, label: str) -> Button: ...

    @abstractmethod
    def checkbox(self, label: str) -> Checkbox: ...


class LightFactory(WidgetFactory):
    def button(self, label: str) -> Button:
        return LightButton(label)

    def checkbox(self, label: str) -> Checkbox:
        return LightCheckbox(label)


class DarkFactory(WidgetFactory):
    def button(self, label: str) -> Button:
        return DarkButton(label)

    def checkbox(self, label: str) -> Checkbox:
        return DarkCheckbox(label)


def build_dialog(factory: WidgetFactory) -> list[str]:
    ok = factory.button("OK")
    remember = factory.checkbox("Запомнить")
    return [ok.render(), remember.render()]

```

Листинг 1.1 (продолжение) *— Классическая реализация: светлая и тёмная темы*

01_classic.py (продолжение) Python

```
69
70 class BrokenFactory(WidgetFactory):
71     def button(self, label: str) -> Button:
72         return LightButton(label)
73
74
75 if __name__ == "__main__":
76     dark_mode = True  # в реальной программе — из настроек
77     factory = DarkFactory() if dark_mode else LightFactory()
78     for line in build_dialog(factory):
79         print(line)
80     try:
81         BrokenFactory()
82     except TypeError as e:
83         print("Ошибка:", e)
```

```
$ python 01_classic.py
< OK > (тёмная)
(*) Запомнить (тёмный)
Ошибка: Can't instantiate abstract class BrokenFactory without an
    implementation for abstract method 'checkbox'
```

Листинг 1.1 (продолжение) *— Классическая реализация: светлая и тёмная темы*

Разбор. build_dialog получает любую фабрику и вызывает два метода — ей не важно, какие классы стоят за ними. Выбор семейства сделан ровно в одном месте. Чтобы добавить

третью тему, достаточно двух продуктов и одной фабрики.

Абстрактные базовые классы дают проверку, которой нет при утиной типизации: класс BrokenFactory забыл метод checkbox, и Python отказывается создать его экземпляр. Ошибка появляется при создании фабрики, а не в момент, когда пользователь откроет диалог с флажком.

Конструкция if __name__ == «__main__»: выполняет демонстрацию только при запуске файла как программы; при импорте модуля она не сработает.

Конец ознакомительного фрагмента.

Текст предоставлен ООО «Литрес».

Прочитайте эту книгу целиком, купив полную легальную версию на Литрес.

Безопасно оплатить книгу можно банковской картой Visa, MasterCard, Maestro, со счета мобильного телефона, с платежного терминала, в салоне МТС или Связной, через PayPal, WebMoney, Яндекс.Деньги, QIWI Кошелек, бонусными картами или другим удобным Вам способом.