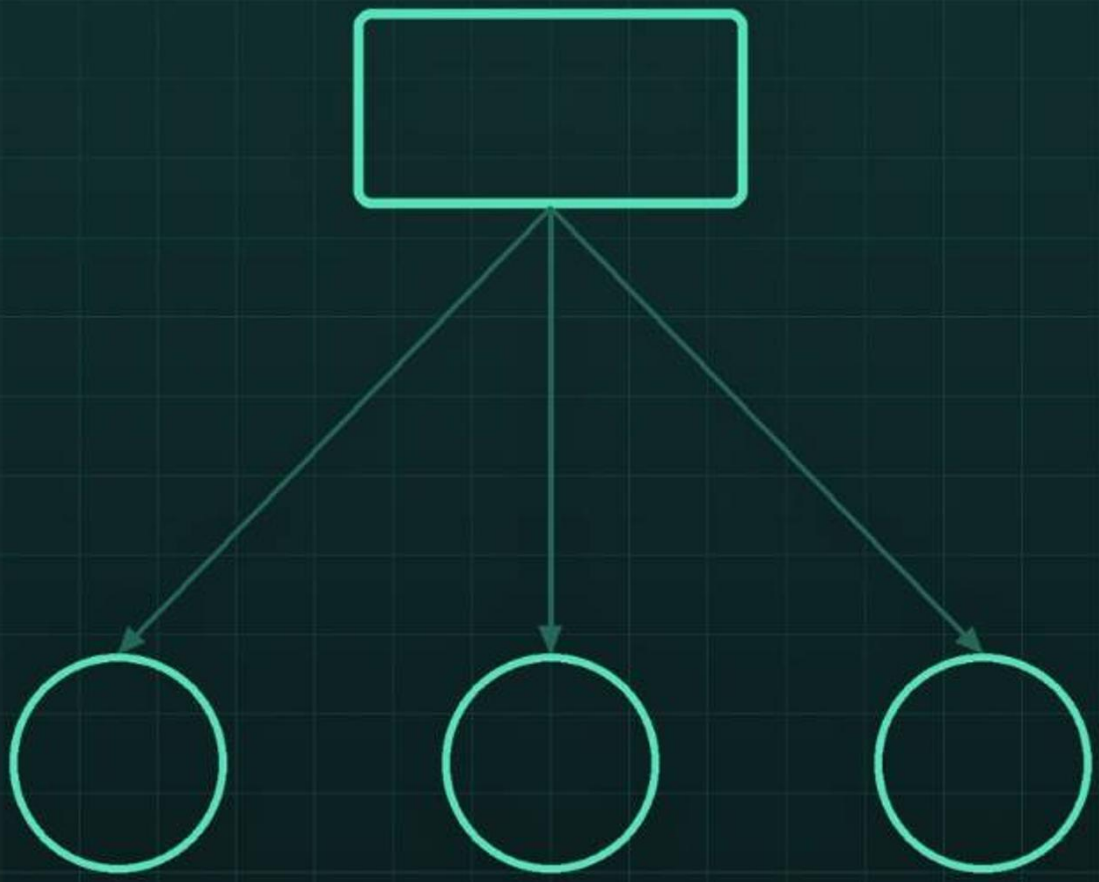
GoF · 03/23

ПОРОЖДАЮЩИЙ

Python C++23 Rust Java



Фабричный метод

Factory Method

Алексей Примак

Алексей Примак

Фабричный метод. Паттерн GoF на Python, C++23, Rust и Java

«Автор»
2026

Примак А.

Фабричный метод. Паттерн GoF на Python, C++23, Rust и Java / А. Примак — «Автор», 2026

Как расширять систему новыми типами объектов, не переписывая общий алгоритм? «Фабричный метод» показывает, как вынести создание продукта в переопределяемую операцию и сохранить общий сценарий работы. Паттерн GoF разобран на понятном примере логистики и реализован на Python, C++23, Rust и Java. В книге также рассматриваются практические альтернативы, модификации и задания для самостоятельной работы.

© Примак А., 2026
© Автор, 2026

Содержание

Введение 5
Как читать 6
Глава 3. Реализация на Python 7
3.1. Зачем нужен фабричный метод 7
3.2. Участники и схема 8
3.3. Когда применять и какие последствия 10
3.4. Классическая реализация: логистика 11
Конец ознакомительного фрагмента. 13

Фабричный метод. Паттерн GoF на Python, C++23, Rust и Java

Введение

Эта книга посвящена одному паттерну проектирования — **Фабричный метод**(Factory Method). 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 часть книги: сначала задача и общая схема, затем код, модификации, гибридные варианты и самостоятельные работы.

Как читать

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

Глава 3. Реализация на Python

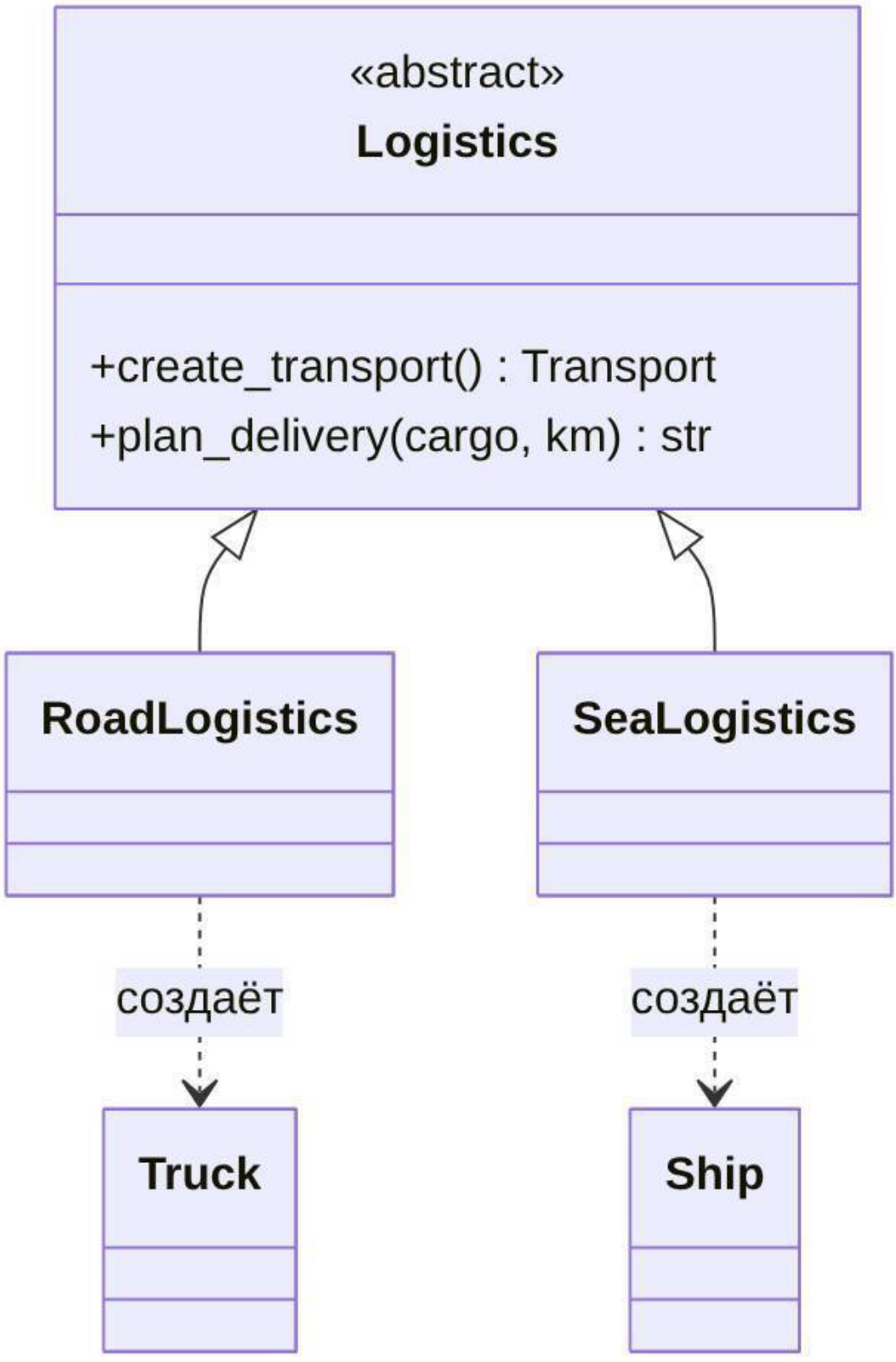
3.1. Зачем нужен фабричный метод

Служба доставки сначала возит грузы только грузовиками, и общий алгоритм — принять заказ, подготовить транспорт, доставить, выписать счёт — прямо внутри создаёт Truck(). Потом появляется морская доставка. Если алгоритм сам создаёт конкретный транспорт, его придётся менять: добавлять ветвления и флаги.

Фабричный метод выделяет **место создания**в отдельную операцию. Общий алгоритм работает с абстрактным «транспортом» и в нужный момент вызывает метод create_transport(). Какой транспорт будет создан, решает подкласс. Сам алгоритм остаётся общим.

Суть паттерна — в связке: **общий алгоритм + переопределяемый шаг создания, которым этот алгоритм пользуется**. Любая функция с create в имени ещё не фабричный метод.

3.2. Участники и схема



Product— интерфейс продукта.
ConcreteProduct— конкретные продукты.
Creator— класс с фабричным методом и общим алгоритмом.
ConcreteCreator— переопределяет фабричный метод.

В Python у паттерна есть короткие альтернативы: передать класс продукта как параметр (классы — объекты), использовать classmethod как именованный конструктор или словарь-реестр. Все они разобраны ниже.

3.3. Когда применять и какие последствия

Паттерн уместен, когда общий алгоритм должен работать с продуктами, тип которых заранее неизвестен, а расширять систему нужно новыми подклассами, не трогая общий код. Особенно он полезен в библиотеках и каркасах: каркас вызывает фабричный метод, а пользователь подставляет свои продукты.

Минус — на каждый продукт часто нужен свой создатель. Если продукт выбирается по значению из конфигурации, удобнее простая фабрика или реестр.

3.4. Классическая реализация: логистика

Файл 01_classic.py:

01_classic.py — Python

```python
from abc import ABC, abstractmethod


class Transport(ABC):
    @abstractmethod
    def deliver(self, cargo: str) -> str: ...

    @property
    @abstractmethod
    def cost_per_km(self) -> int: ...


class Truck(Transport):
    cost_per_km = 40

    def deliver(self, cargo: str) -> str:
        return f"грузовик везёт «{cargo}» по дороге"


class Ship(Transport):
    cost_per_km = 15

    def deliver(self, cargo: str) -> str:
        return f"корабль везёт «{cargo}» по морю"


class Logistics(ABC):
    @abstractmethod
    def create_transport(self) -> Transport:
        """Фабричный метод."""

    def plan_delivery(self, cargo: str, km: int) -> str:
        """Общий алгоритм, использующий продукт."""
        transport = self.create_transport()
```

Листинг 3.1 *— Классическая реализация: логистика*

01_classic.py (продолжение) Python

```python
        price = transport.cost_per_km * km
        return f"{transport.deliver(cargo)}; {km} км, стоимость
            {price} ₽"


class RoadLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Truck()


class SeaLogistics(Logistics):
    def create_transport(self) -> Transport:
        return Ship()


if __name__ == "__main__":
    for planner in (RoadLogistics(), SeaLogistics()):
        print(planner.plan_delivery("мебель", 800))
```

```
$ python 01_classic.py
грузовик везёт «мебель» по дороге; 800 км, стоимость 32000 ₽
корабль везёт «мебель» по морю; 800 км, стоимость 12000 ₽
```

Листинг 3.1 (продолжение) *— Классическая реализация: логистика*

Разбор.plan_delivery написан один раз в базовом классе. Подклассы переопределяют только create_transport. Абстрактное свойство cost_per_km подклассы реализуют простым атрибутом класса — Python считает это реализацией, потому что имя определено.

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.