Python C++23 Rust Java



# Прототип

Prototype

Алексей Примак

Алексей Примак

# Прототип. Паттерн GoF на Python, C++23, Rust и Java

«Автор»
2026

**Примак А.**

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

Иногда новый объект проще получить копированием уже настроенного экземпляра. «Прототип» разбирает этот подход на практических задачах и показывает, как организовать создание копий без жёсткой привязки клиента к конкретному классу. Паттерн GoF последовательно реализован на Python, C++23, Rust и Java. В книге рассматриваются классическая схема, варианты клонирования, модификации, гибридные решения и самостоятельные задания.

© Примак А., 2026
© Автор, 2026

# Содержание

Введение 5
Как читать 6
Глава 4. Реализация на Python 7
4.1. Зачем нужен прототип 7
4.2. Участники и схема 8
4.3. Поверхностная и глубокая копия 11
4.4. Классическая реализация: метод clone 13
Конец ознакомительного фрагмента. 15

# Прототип. Паттерн GoF на Python, C++23, Rust и Java

## Введение

Эта книга посвящена одному паттерну проектирования — **Прототип**(Prototype). 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.

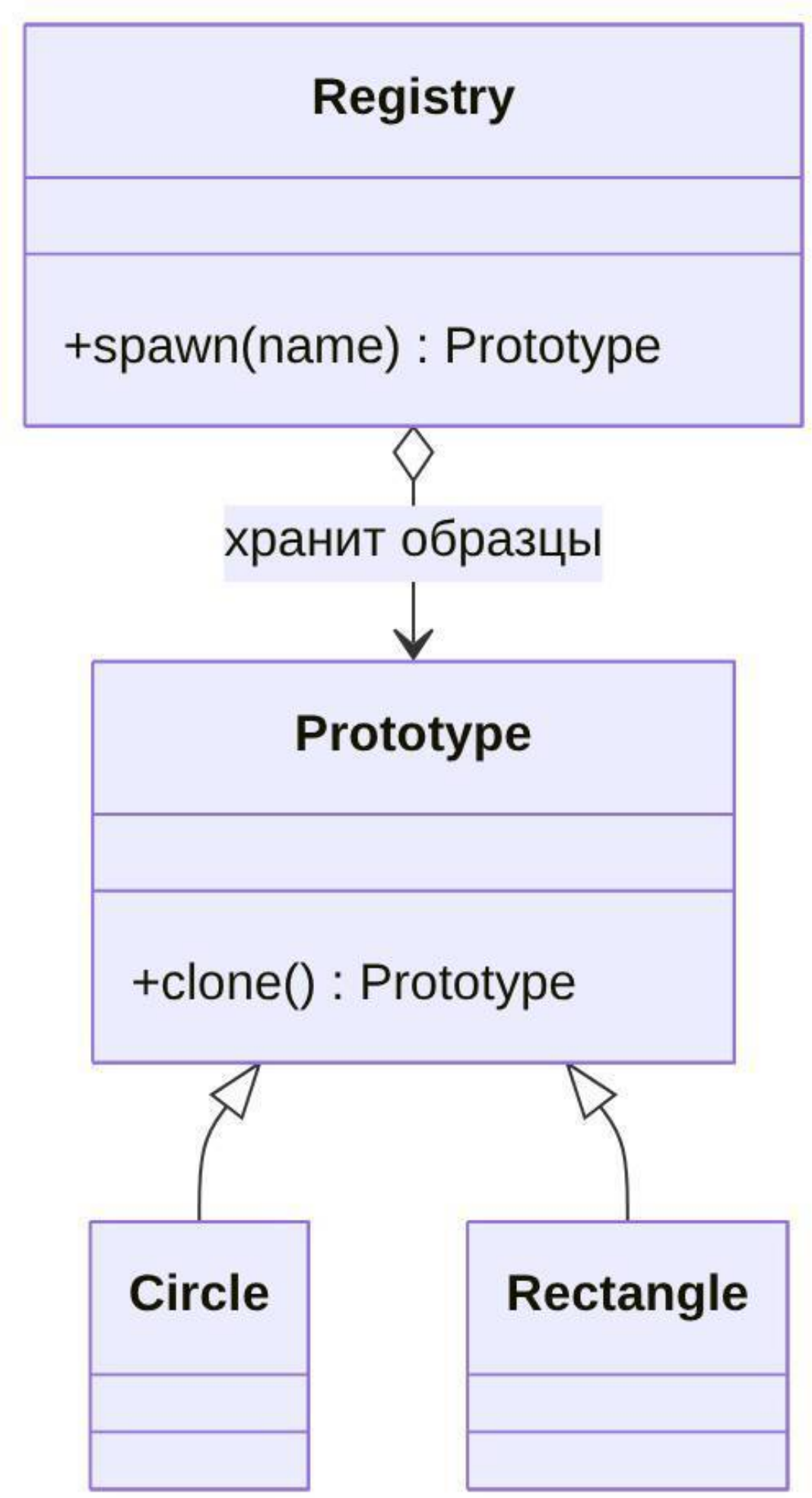
# Глава 4. Реализация на Python

## 4.1. Зачем нужен прототип

Иногда новый объект проще получить копированием готового, чем собирать с нуля. Пользователь редактора настроил фигуру — цвет, толщину, тень — и хочет ещё десять таких же. В игре нужна сотня врагов с одинаковыми характеристиками и небольшими отличиями. Код при этом не обязан знать конкретный класс объекта: достаточно попросить объект создать свою копию.

Прототип описывает именно это: объект умеет клонировать себя, а клиент создаёт новые объекты копированием образца. В Python копирование встроено в стандартную библиотеку — модуль copy с функциями copy.copy (поверхностная копия) и copy.deepcopy (глубокая), а для dataclass — dataclasses.replace. Поэтому главный вопрос паттерна в Python — не «как скопировать», а **что именно копировать**: данные целиком или разделяемые ссылки.

## 4.2. Участники и схема


Registry
+spawn(name) : Prototype
хранит образцы
Prototype
+clone() : Prototype
Circle
Rectangle

**Prototype**— интерфейс клонирования.
**ConcretePrototype**— классы, умеющие себя копировать.
**Client**— создаёт объекты, запрашивая копии.
**Registry**— необязательное хранилище именованных образцов.

## 4.3. Поверхностная и глубокая копия

Это главная ловушка паттерна в Python, поэтому начнём с неё.
**Файл 01_shallow_deep.py:**

01_shallow_deep.py — Python

```python
import copy
from dataclasses import dataclass, field


@dataclass
class Enemy:
    kind: str
    hp: int
    position: tuple[int, int]
    loot: list[str] = field(default_factory=list)


if __name__ == "__main__":
    orc = Enemy("орк", 120, (0, 0), ["топор", "монеты"])

    shallow = copy.copy(orc)
    shallow.hp -= 30
    shallow.loot.append("ключ")
    print("После поверхностной копии прототип:", orc.hp, orc.
        loot)

    orc.loot.remove("ключ")
    deep = copy.deepcopy(orc)
    deep.loot.append("амулет")
    print("После глубокой копии прототип:", orc.hp, orc.loot)
    print("Списки — один объект?", shallow.loot is orc.loot,
        deep.loot is orc.loot)
```

```
$ python 01_shallow_deep.py
После поверхностной копии прототип: 120 ['топор', 'монеты', 'ключ']
После глубокой копии прототип: 120 ['топор', 'монеты']
Списки — один объект? True False
```

***Листинг 4.1** — Поверхностная и глубокая копия*

**Разбор.**copy.copy создаёт новый объект, но его атрибуты ссылаются на те же объекты, что и у оригинала. Число hp неизменяемо, поэтому shallow.hp -= 30 просто привязывает к атрибуту копии новое число — прототип не затронут. А список loot общий: добавление ключа в «копию» испортило прототип. copy.deepcopy рекурсивно копирует вложенные объекты, и копия становится полностью независимой.

Правило: если у объекта есть изменяемые вложенные данные (списки, словари, другие объекты), которые копия будет менять, нужна глубокая копия — или явное копирование именно этих полей.

## 4.4. Классическая реализация: метод clone

Иногда ни поверхностная, ни глубокая копия не подходят целиком: часть данных нужно копировать, часть — разделять, часть — сбрасывать. Тогда объект определяет собственный метод clone или специальные методы __copy__ и __deepcopy__.

**Файл 02_clone.py:**

02_clone.py — Python

```python
import copy
import itertools
from abc import ABC, abstractmethod

_ids = itertools.count(1)


class Shape(ABC):
    def __init__(self, x: int, y: int, color: str) -> None:
        self.id = next(_ids)
        self.x, self.y, self.color = x, y, color
        self.tags: set[str] = set()

    def clone(self) -> "Shape":
        new = copy.copy(self)
        new.id = next(_ids)          # у копии свой идентификатор
        new.tags = set(self.tags)    # свой набор тегов
        return new

    def moved(self, dx: int, dy: int) -> "Shape":
        c = self.clone()
        c.x += dx
        c.y += dy
        return c

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


class Circle(Shape):
    def __init__(self, x: int, y: int, color: str, r: int) -> None:
        super().__init__(x, y, color)
```

***Листинг 4.2**— Классическая реализация: метод clone*

02_clone.py (продолжение) Python

```python
        self.r = r

    def draw(self) -> str:
        return f"#{self.id} {self.color} круг r={self.r} в ({
            self.x}, {self.y}) {sorted(self.tags)}"


class Rect(Shape):
    def __init__(self, x: int, y: int, color: str, w: int, h:
        int) -> None:
        super().__init__(x, y, color)
        self.w, self.h = w, h

    def draw(self) -> str:
        return f"#{self.id} {self.color} прямоугольник {self.w}x
            {self.h} в ({self.x}, {self.y}) {sorted(self.tags)}"


if __name__ == "__main__":
    shapes: list[Shape] = [Circle(0, 0, "красный", 5), Rect(10,
        0, "синий", 4, 2)]
    shapes[0].tags.add("логотип")
    copies = [s.moved(100, 0) for s in shapes]
    copies[0].tags.add("копия")
    for s in shapes + copies:
        print(s.draw())
```

```
$ python 02_clone.py
#1 красный круг r=5 в (0, 0) ['логотип']
#2 синий прямоугольник 4x2 в (10, 0) []
#3 красный круг r=5 в (100, 0) ['копия', 'логотип']
#4 синий прямоугольник 4x2 в (110, 0) []
```

***Листинг 4.2 (продолжение)*** *— Классическая реализация: метод clone*

**Разбор.** Метод clone определён в базовом классе и работает для всех фигур: copy.copy(self) создаёт объект того же класса со всеми атрибутами, включая r, w и h, о которых базовый класс не знает. Затем копия получает собственный идентификатор и собственное множество тегов. Добавление тега «копия» не затронуло оригинал. Клиенту, который вызывает moved, не нужно знать, круг перед ним или прямоугольник.

# 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.