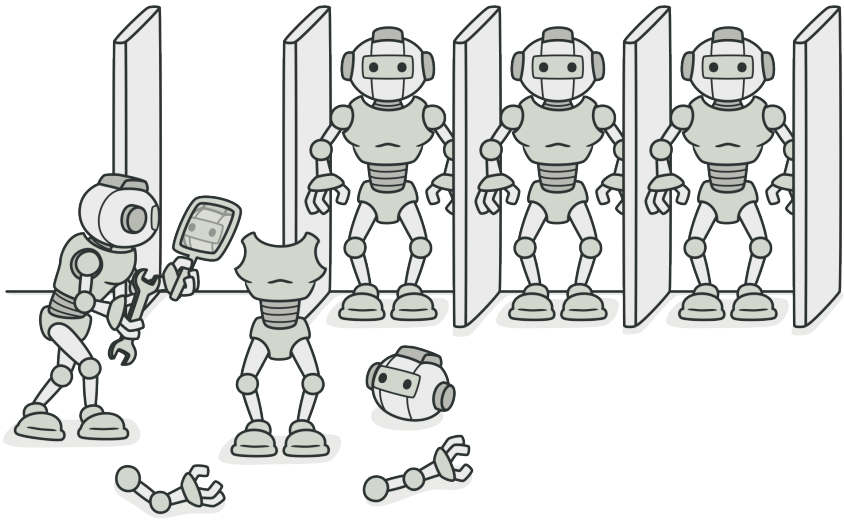


ПРОТОТИП

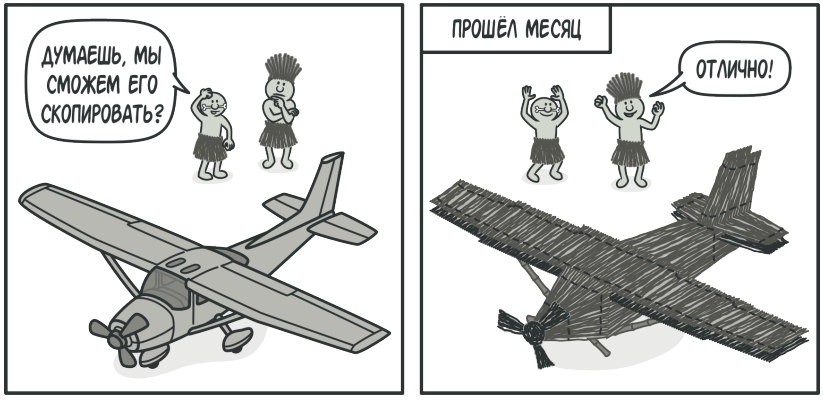
Также известен как: Клон, Prototype

Прототип — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации.

☹ Проблема

У вас есть объект, который нужно скопировать. Как это сделать? Нужно создать пустой объект такого же класса, а затем поочерёдно скопировать значения всех полей из старого объекта в новый.

Прекрасно! Но есть нюанс. Не каждый объект удастся скопировать таким образом, ведь часть его состояния может быть приватной и недоступна для остального кода программы.



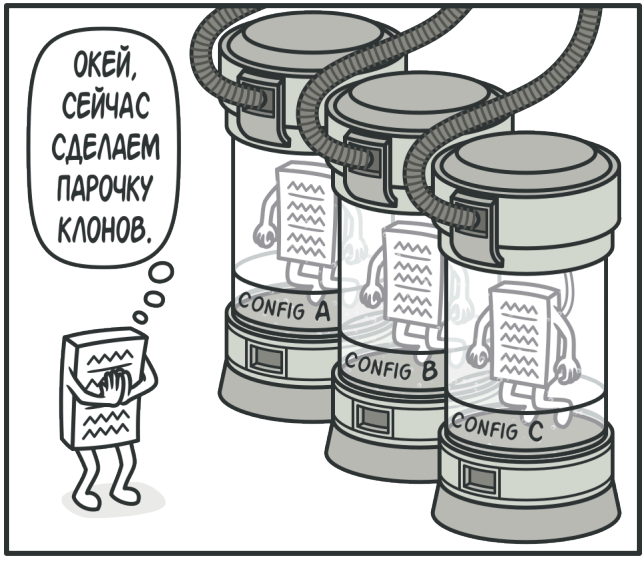
Копирование «извне» не всегда возможно в реальности.

Но есть и другая проблема. Копирующий код станет зависим от классов копируемых объектов. Ведь чтобы перебрать *все* поля объекта, нужно привязаться к его классу. Из-за этого вы не сможете копировать объекты, зная только их интерфейсы, но не конкретные классы.

☺ Решение

Паттерн Прототип поручает создание копий самим копируемым объектам. Он вводит общий интерфейс для всех объектов, поддерживающих клонирование. Это позволяет копировать объекты, не привязываясь к их конкретным классам. Обычно такой интерфейс имеет всего один метод `clone`.

Реализация этого метода в разных классах очень схожа. Метод создаёт новый объект текущего класса и копирует в него значения всех полей объекта. Так получится скопировать даже приватные поля, так как большинство языков программирования разрешает доступ к приватным полям отдельного объекта текущего класса.



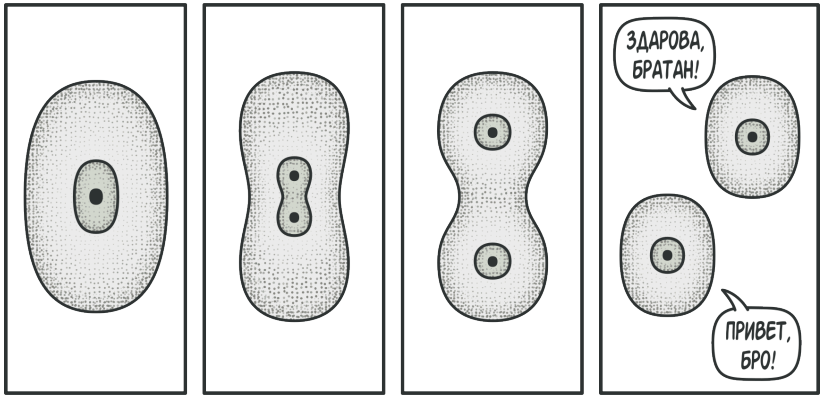
Предварительно заготовленные прототипы могут стать заменой подклассам.

Объект, который копируют, называется *прототипом* (откуда и название паттерна). Когда объекты программы содержат сотни полей и тысячи возможных конфигураций, прототипы могут служить своеобразной альтернативой созданию подклассов.

В этом случае, все возможные прототипы заготавливаются и настраиваются на этапе инициализации программы. Потом, когда программе нужен новый объект, она создаёт копию из приготовленного прототипа.

Аналогия из жизни

В промышленном производстве прототипы создаются перед основной партией продуктов для проведения всевозможных испытаний. При этом прототип не участвует в последующем производстве, отыгрывая пассивную роль.

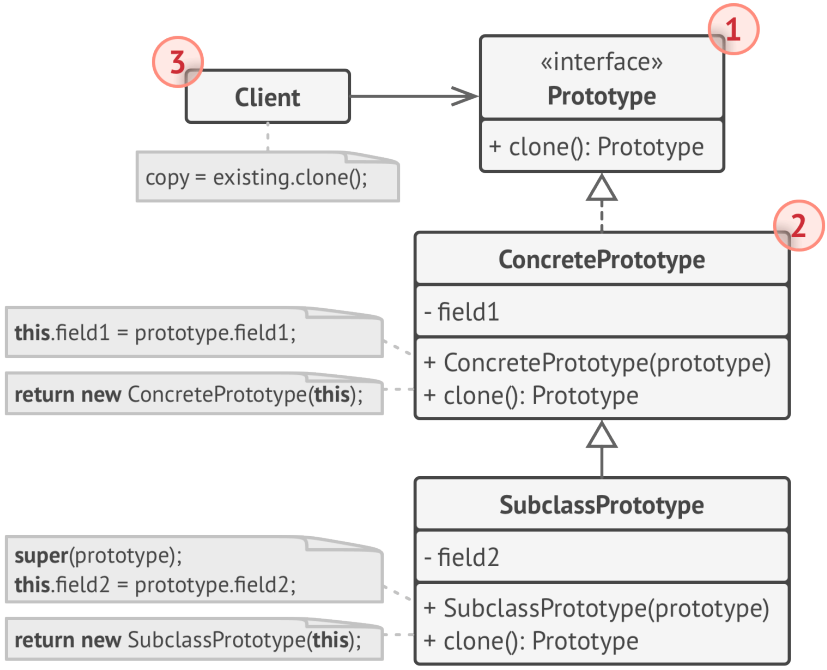


Пример деления клетки.

Прототип на производстве не делает копию самого себя, поэтому более близкий пример паттерна — деление клеток. После митозного деления клеток образуются две совершенно идентичные клетки. Оригинальная клетка отыгрывает роль прототипа, принимает активную роль в создании нового объекта.

Структура

Базовая реализация

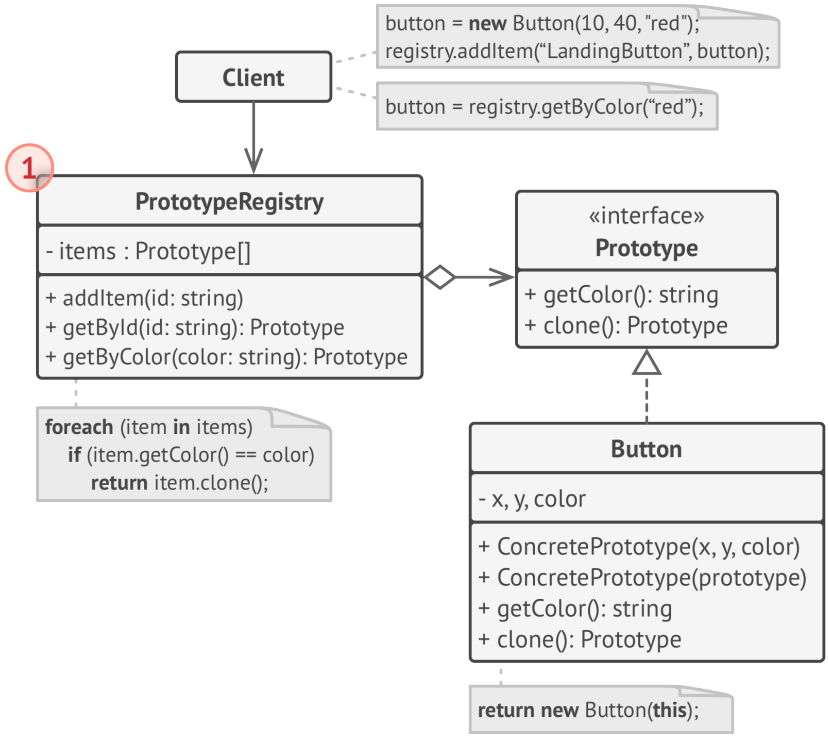


1. **Интерфейс прототипов** описывает операции клонирования. В большинстве случаев — это единственный метод `clone`.

2. **Конкретный прототип** реализует операцию клонирования самого себя. Помимо банального копирования значений всех полей, здесь могут быть спрятаны различные сложности, о которых не нужно знать клиенту. Например, клонирование связанных объектов, распутывание рекурсивных зависимостей и прочее.

3. **Клиент** создаёт копию объекта, обращаясь к нему через общий интерфейс прототипов.

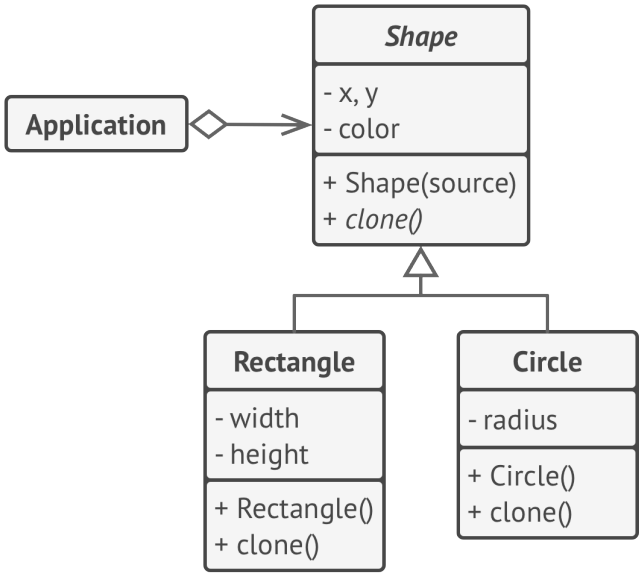
Реализация с общим хранилищем прототипов



1. **Хранилище прототипов** облегчает доступ к часто используемым прототипам, храня предсозданный набор эталонных, готовых к копированию объектов. Простейшее хранилище может быть построено с помощью хеш-таблицы вида `имя–прототипа → прототип`. Но для удобства поиска, прототипы можно маркировать и другими критериями, а не только условным именем.

Псевдокод

В этом примере **Прототип** позволяет производить точные копии объектов геометрических фигур, не привязываясь к их классам.



Пример клонирования иерархии геометрических фигур.

Все фигуры реализуют интерфейс клонирования и предоставляют метод для воспроизводства самой себя. Подклассы используют метод клонирования родителя, а затем копируют собственные поля в получившийся объект.

```
// Базовый прототип.
abstract class Shape is
  field X: int
  field Y: int
  field color: string

  // Копирование всех полей объекта происходит
  // в конструкторе.
  constructor Shape(source: Shape) is
    if (source != null) then
      this.X = source.X
      this.Y = source.Y
      this.color = source.color

  // Результатом операции клонирования всегда будет объект
  // из иерархии классов Shape.
  abstract method clone(): Shape


// Конкретный прототип. Метод клонирования создаёт новый
// объект и передаёт в его конструктор для копирования
// собственный объект. Этим мы пытаемся получить атомарность
// операции клонирования. В данной реализации, пока не
// выполнится конструктор, нового объекта ещё не существует.
// Но как только конструктор завершён, мы получаем полностью
// готовый объект-клон, а не пустой объект, который нужно
// ещё заполнить.
class Rectangle extends Shape is
```

```
  field width: int
  field height: int

  constructor Rectangle(source: Rectangle) is
    // Вызов родительского конструктора нужен, чтобы
    // скопировать потенциальные приватные поля,
    // объявленные в родительском классе.
    super(source)
    if (source != null) then
      this.width = source.width
      this.height = source.height

  method clone(): Shape is
    return new Rectangle(this)


class Circle extends Shape is
  field radius: int

  constructor Circle(source: Circle) is
    super(source)
    if (source != null) then
      this.radius = source.radius

  method clone(): Shape is
    return new Circle(this)


// Где-то в клиентском коде.
class Application is
  field shapes: array of Shape

  constructor Application() is
    Circle circle = new Circle()
```

```
    circle.X = 10
    circle.Y = 20
    circle.radius = 15
    shapes.add(circle)

    Circle anotherCircle = circle.clone()
    shapes.add(anotherCircle)
    // anotherCircle будет содержать точную
    // копию circle.

    Rectangle rectangle = new Rectangle()
    rectangle.width = 10
    rectangle.height = 20
    shapes.add(rectangle)

  method businessLogic() is
    // Неочевидный плюс Прототипа в том, что вы можете
    // клонировать набор объектов, не зная их
    // конкретных классов.
    Array shapesCopy = new Array of Shapes.

    // Например, мы не знаем какие конкретно объекты
    // находятся внутри массива shapes, так как он
    // объявлен с типом Shape. Но благодаря
    // полиморфизму, мы можем клонировать все объекты
    // «вслепую». Будет выполнен метод `clone` того
    // класса, которым является этот объект.
    foreach (s in shapes) do
      shapesCopy.add(s.clone())

    // Переменная shapesCopy будет содержать точные
    // копии элементов массива shapes.
```

Применимость

Когда ваш код не должен зависеть от классов копируемых объектов.

Такое часто бывает, если ваш код работает с объектами, поданными извне через какой-то общий интерфейс. Вы не можете привязаться к их классам, даже если бы хотели, так как их конкретные классы неизвестны.

Паттерн прототип предоставляет клиенту общий интерфейс для работы со всеми прототипами. Клиенту не нужно зависеть от всех классов копируемых объектов, а только от интерфейса клонирования.

Когда вы имеете уйму подклассов, которые отличаются начальными значениями полей. Кто-то создал эти классы, чтобы быстро создавать объекты с определённой конфигурацией.

Паттерн прототип предлагает использовать набор прототипов, вместо создания подклассов для популярных конфигураций объектов.

Таким образом, вместо порождения объектов из подклассов, вы будете копировать существующие объекты-прототипы, в которых уже настроено внутреннее состояние. Это позволит избежать взрывного роста количества классов в программе и уменьшить её сложность.

☑ Шаги реализации

1. Создайте интерфейс прототипов с единственным методом `clone`. Если у вас уже есть иерархия продуктов, метод клонирования можно объявить непосредственно в каждом из её классов.

2. Добавьте в классы будущих прототипов альтернативный конструктор, принимающий в качестве аргумента объект текущего класса. Этот конструктор должен скопировать из поданного объекта значения всех полей, объявленных в рамках текущего класса, а затем передать выполнение родительскому конструктору, чтобы тот позаботился об остальных полях.

   Если ваш язык программирования не поддерживает перегрузку методов, то копирование значений можно проводить и в другом методе, специально созданном для этих целей. Конструктор удобнее тем, что позволяет клонировать объект за один вызов.

3. Метод клонирования обычно состоит всего из одной строки: вызова оператора `new` с конструктором прототипа. Все классы, поддерживающие клонирование, должны явно определить метод `clone`, чтобы подать собственный класс в оператор `new`. В обратном случае, результатом клонирования окажется объект родительского класса.

4. Опционально, создайте центральное хранилище прототипов. В нём можно хранить вариации объектов, возможно даже одного класса, но по-разному настроенных.

   Вы можете разместить это хранилище либо в новом фабричном классе, либо в фабричном методе базового класса прототипов. Такой фабричный метод должен на основании входящих аргументов искать в каталоге прототипов подходящий экземпляр, а затем вызывать его метод клонирования и возвращать полученный объект.

   Наконец, нужно будет избавиться от прямых вызовов конструкторов объектов, заменив их вызовами фабричного метода хранилища прототипов.

⚖ Преимущества и недостатки

- ✔ Позволяет клонировать объекты, не привязываясь к их конкретным классам.
- ✔ Меньше повторяющегося кода инициализации объектов.
- ✔ Ускоряет создание объектов.
- ✔ Альтернатива созданию подклассов для конструирования сложных объектов.

- ✖ Сложно клонировать составные объекты, имеющие ссылки на другие объекты.

⇄ Отношения с другими паттернами

- Многие архитектуры начинаются с применения **Фабричного метода** (более простого и расширяемого через подклассы) и эволюционируют в сторону **Абстрактной фабрики**, **Прототипа** или **Строителя** (более гибких, но и более сложных).

- Классы **Абстрактной фабрики** чаще всего реализуются с помощью **Фабричного метода**, хотя они могут быть построены и на основе **Прототипа**.

- Если **Команду** нужно копировать перед вставкой в историю выполненных команд, вам может помочь **Прототип**.

- Архитектура, построенная на **Компоновщиках** и **Декораторах**, часто может быть улучшена за счёт внедрения **Прототипа**. Он позволяет клонировать сложные структуры объектов, а не собирать их заново.

- **Прототип** не опирается на наследование, но ему нужна сложная операция инициализации. **Фабричный метод** наоборот, построен на наследовании, но не требует сложной инициализации.

- **Снимок** иногда можно заменить **Прототипом**, если объект, чьё состояние требуется сохранять в истории, довольно

простой, не имеет активных ссылок на внешние ресурсы, либо их можно легко восстановить.

- **Абстрактная фабрика**, **Строитель** и **Прототип** могут быть реализованы при помощи **Одиночки**.