Лекция 7: Структурные и порождающие шаблоны

Юрий Литвинов
yurii.litvinov@gmail.com

02.11.2017г

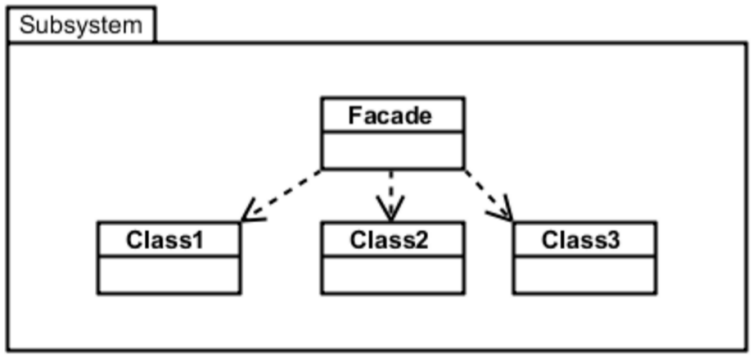
Паттерн “Фасад”

Facade

- Простой интерфейс к сложной системе
- Отделение подсистем от клиента и друг от друга
- Многоуровневая архитектура

“Фасад” (Facade), детали реализации

- Абстрактный Facade
  - Существенно снижает связность клиента с подсистемой



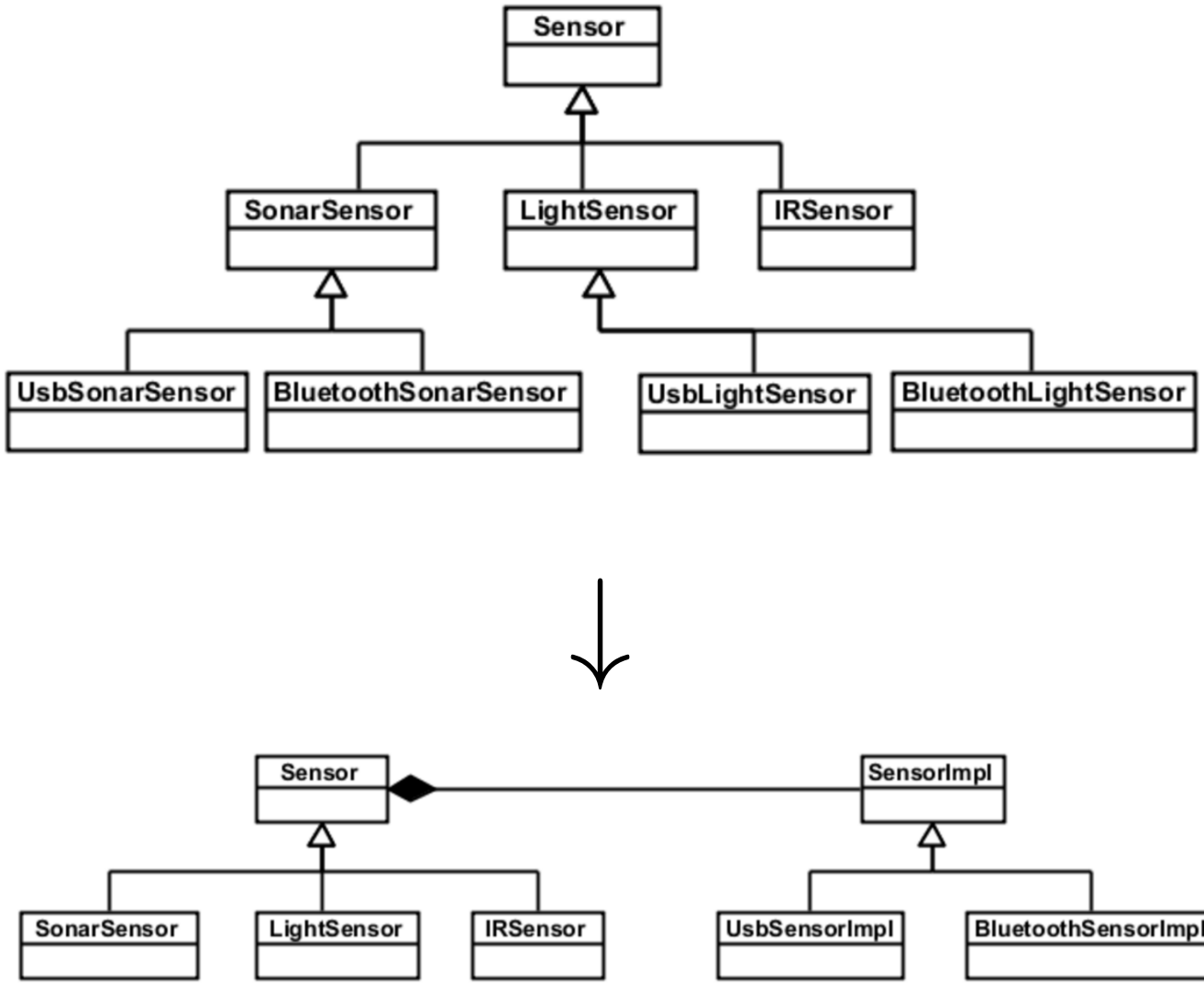
- Открытые и закрытые классы подсистемы
  - Пространства имён и пакеты помогают, но требуют дополнительных соглашений
    - Пространство имён details
  - Инкапсуляция целой подсистемы — это хорошо

Паттерн “Мост” (Bridge)

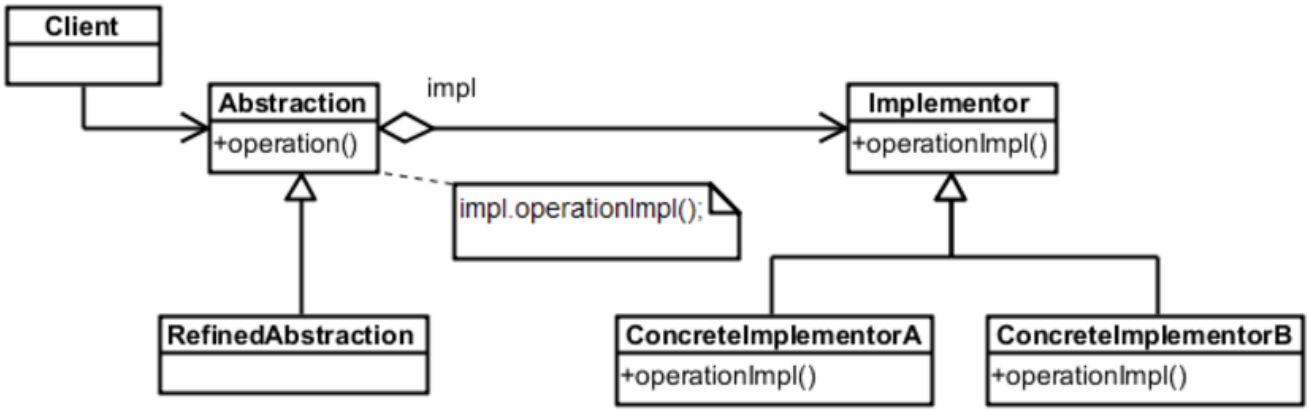
Отделяет абстракцию от реализации
Пример:

- Есть система, интерпретирующая программы для роботов
- Есть класс *Sensor*, от которого наследуются *SonarSensor*, *LightSensor*, ...
- Связь с роботом может выполняться по USB или Bluetooth, а может быть, программа и вовсе исполняется на симуляторе
- Интерпретатор хочет работать с сенсорами, не заморачиваясь реализацией механизма связи
- Рабоче-крестьянская реализация — *USBLightSensor*, *BluetoothLightSensor*, *USBSonarSensor*, *BluetoothSonarSensor*, ...
- Число классов — произведение количества сенсоров и типов связи

“Мост”, пример


Sensor
SonarSensor
LightSensor
IRSensor
UsbSonarSensor
BluetoothSonarSensor
UsbLightSensor
BluetoothLightSensor
Sensor
SensorImpl
SonarSensor
LightSensor
IRSensor
UsbSensorImpl
BluetoothSensorImpl

“Мост”, общая схема



- *Abstraction* — определяет интерфейс абстракции, хранит ссылку на реализацию
- *RefinedAbstraction* — расширяет интерфейс абстракции, делает полезную работу, используя реализацию
- *Implementor* — определяет интерфейс реализации, в котором абстракции предоставляются низкоуровневые операции
- *ConcreteImplementor* — предоставляет конкретную реализацию Implementor

Когда применять

- Когда хочется разделить абстракцию и реализацию, например, когда реализацию можно выбирать во время компиляции или во время выполнения
  - “Стратегия”, “Прокси”
- Когда абстракция и реализация должны расширяться новыми подклассами
- Когда хочется разделить одну реализацию между несколькими объектами
  - Как copy-on-write в строках

Тонкости реализации

Создание правильного Implementor-а

- Самой абстракцией в конструкторе, в зависимости от переданных параметров
  - Как вариант — выбор реализации по умолчанию и замена её по ходу работы
- Принимать реализацию извне (как параметр конструктора, или, реже, как значение в сеттер)
- Фабрика/фабричный метод
  - Позволяет спрятать платформозависимые реализации, чтобы не зависеть от них всех при сборке

Pointer To Implementation (PImpl)

Вырожденный мост для C++, когда “абстракция” имеет ровно одну реализацию, часто полностью дублирующую её интерфейс
Зачем: чтобы клиенты класса не зависели при сборке от его реализации

- Позитивно сказывается на времени компиляции программ на C++
- Позволяет менять реализацию независимо
  - Сохраняя бинарную совместимость

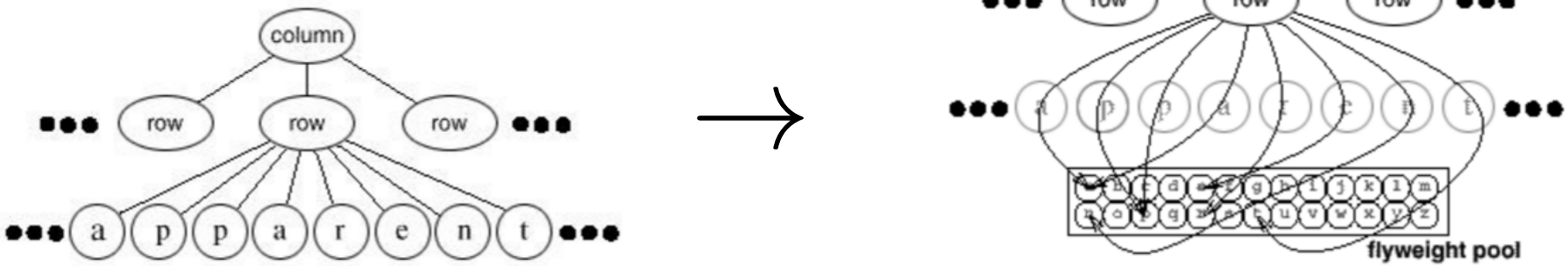
Как: предварительное объявление класса-реализации, полное определение — в .cpp-файле вместе с методами абстракции
Часто используется в реализации библиотек (например, Qt)

Паттерн “Приспособленец” (Flyweight)

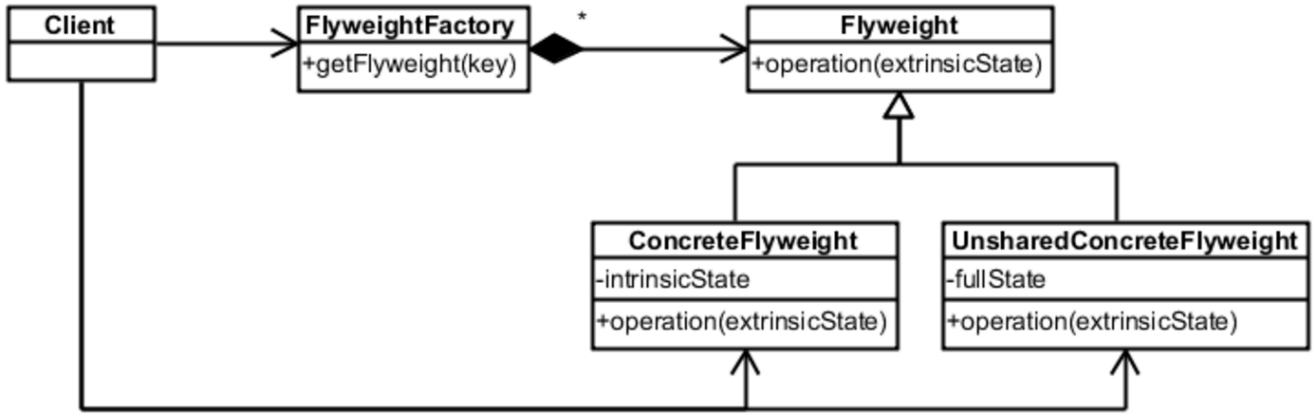
Предназначается для эффективной поддержки множества мелких объектов
Пример:

- Есть текстовый редактор
- Хочется работать с каждым символом как с объектом
  - Единообразие алгоритмов форматирования и внутренней структуры документа
  - Более красивая и ООПшная реализация
    - Паттерн “Компоновщик”, структура “Символ” $\rightarrow$ “Строка” $\rightarrow$ “Страница”
- Наивная реализация привела бы к чрезмерной расточительности по времени работы и по памяти, потому что документы с миллионами символов не редкость

“Приспособленец”, пример

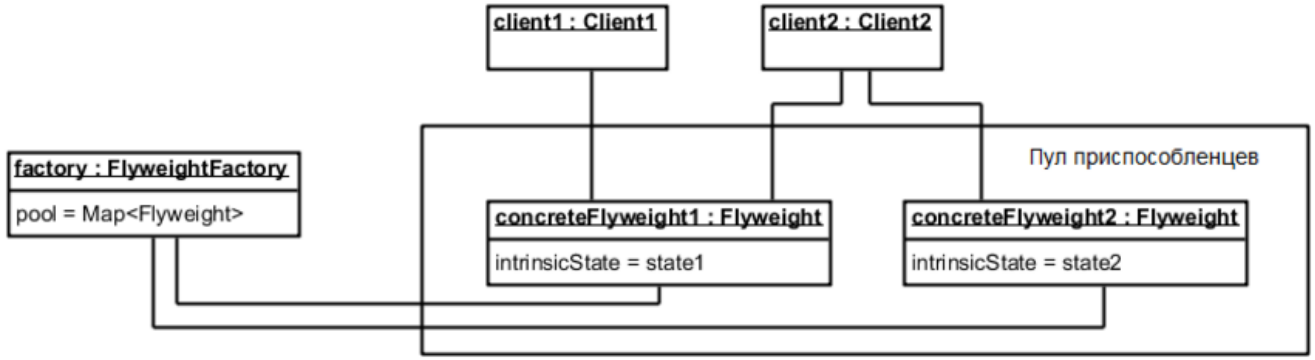


“Приспособленец”, общая схема



- *Flyweight* — определяет интерфейс, через который приспособленцы могут получать внешнее состояние
- *ConcreteFlyweight* — реализует интерфейс Flyweight и может иметь внутреннее состояние, не зависит от контекста
- *UnsharedConcreteFlyweight* — неразделяемый “приспособленец”, хранящий всё состояние в себе, бывает нужен, чтобы собирать иерархические структуры из Flyweight-ов (“Компоновщик”)
- *FlyweightFactory* — содержит пул приспособленцев, создаёт их и управляет их жизнью

“Приспособленец”, диаграмма объектов



- Клиенты могут быть разных типов
- Клиенты могут разделять приспособленцев
  - Один клиент может иметь несколько ссылок на одного приспособленца
- Во время выполнения клиенты имеют право не знать про фабрику

Когда применять

- Когда в приложении используется много мелких объектов
- Они допускают разделение состояния на внутреннее и внешнее
  - Желательно, чтобы внешнее состояние было вычислимо
- Идентичность объектов не важна
  - Используется семантика Value Type
- Главное, когда от такого разделения можно получить ощутимый выигрыш

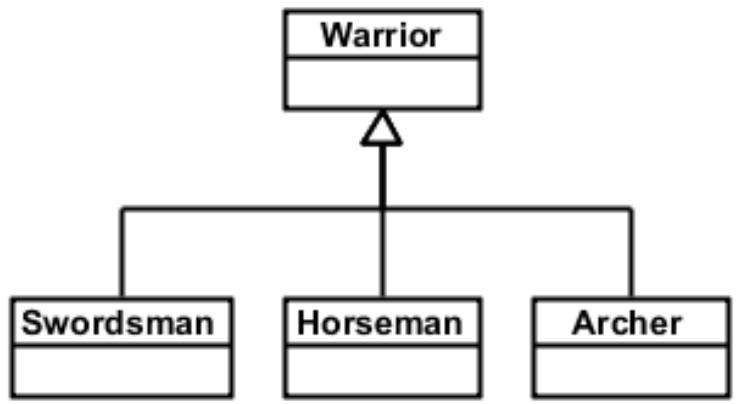
Тонкости реализации

- Внешнее состояние — по сути, отдельный объект, поэтому если различных внешних состояний столько же, сколько приспособленцев, смысла нет
  - Один объект-состояние покрывает сразу несколько приспособленцев
    - Например, объект “Range” может хранить параметры форматирования для всех букв внутри фрагмента
- Клиенты не должны инстанцировать приспособленцев сами, иначе трудно обеспечить разделение
  - Имеет смысл иметь механизм для удаления неиспользуемых приспособленцев
    - Если их может быть много
- Приспособленцы немутабельны и Value Objects (с правильно переопределённой операцией сравнения)
  - Про hashCode() тоже надо не забыть

“Фабричный метод” мотивация

Игра-стратегия

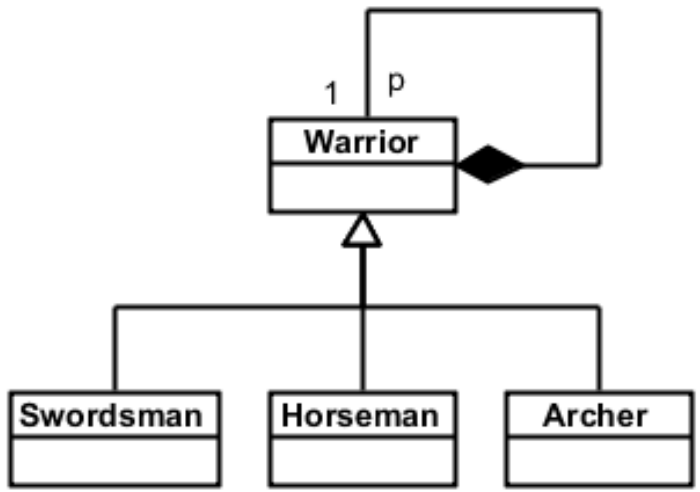
- Воины
  - Мечники
  - Конница
  - Лучники
- Общее поведение
- Общие характеристики



Виртуальный конструктор

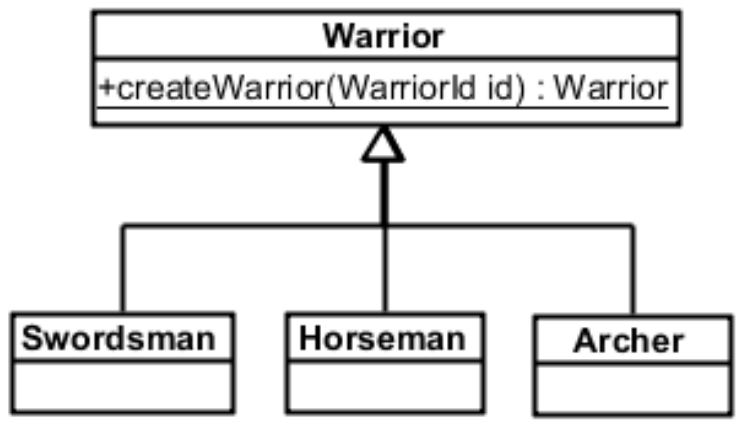
```
enum WarriorId { SwordsmanId, ArcherId, HorsemanId };

class Warrior
{
public:
   Warrior(WarriorId id)
   {
      if (id == SwordsmanId) p = new Swordsman;
      else if (id == ArcherId) p = new Archer;
      else if (id == HorsemanId) p = new Horseman;
      else assert( false);
   }
   virtual void info() { p->info(); }
   virtual ~Warrior() { delete p;  p = 0; }
private:
   Warrior* p;
};
```



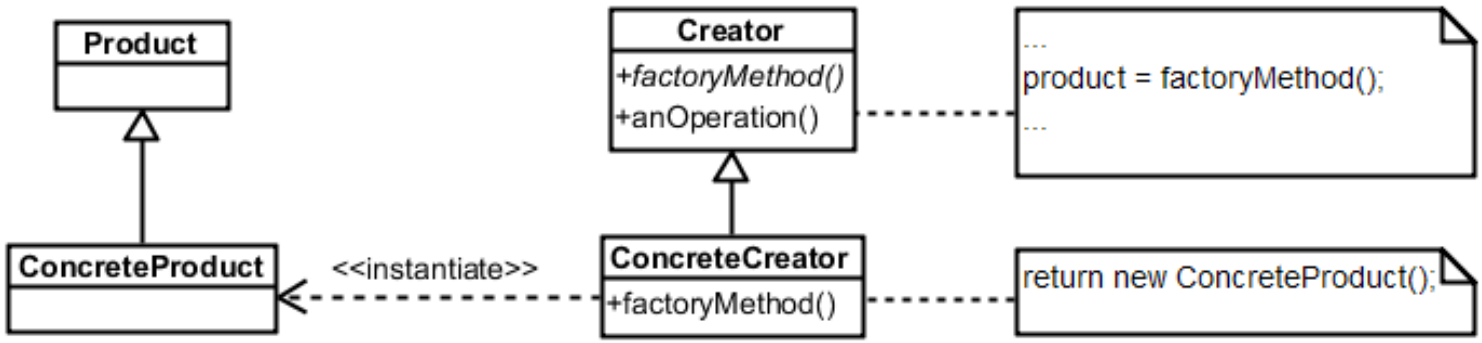
Фабричный метод

- Базовый класс знает про остальные
- switch в createWarrior()



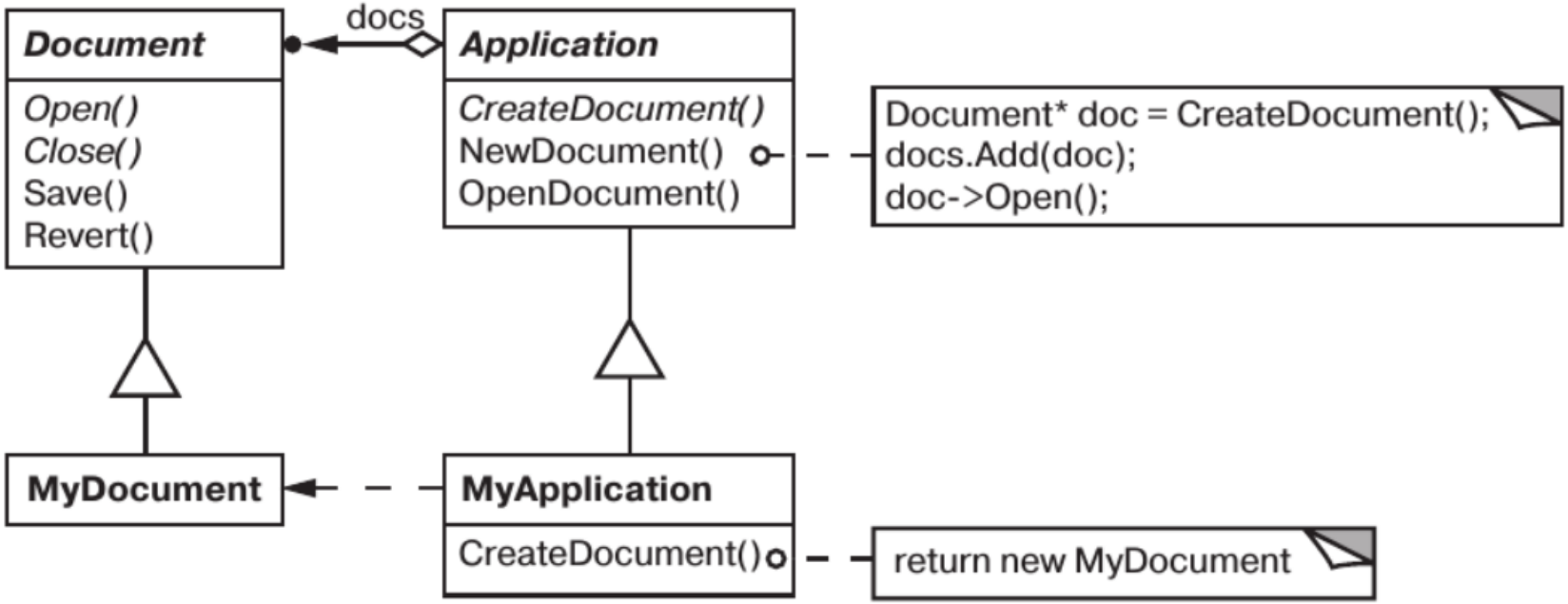
Паттерн “Factory Method”

Factory Method

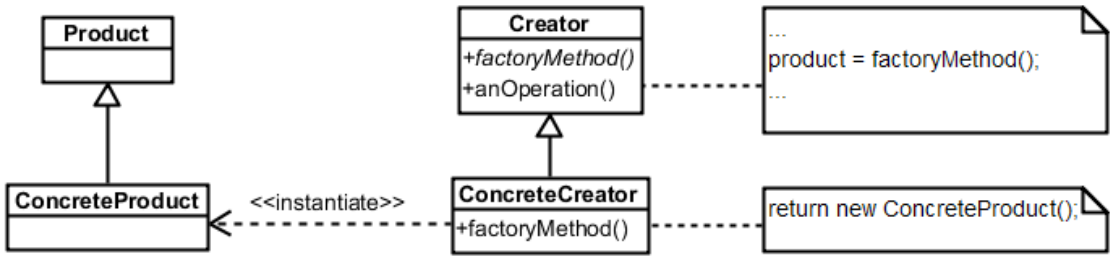


- Применимость:
  - классу заранее неизвестно, объекты каких классов ему нужно создавать
  - объекты, которые создает класс, специфицируются подклассами
  - класс делегирует свои обязанности одному из нескольких вспомогательных подклассов

Пример, текстовый редактор



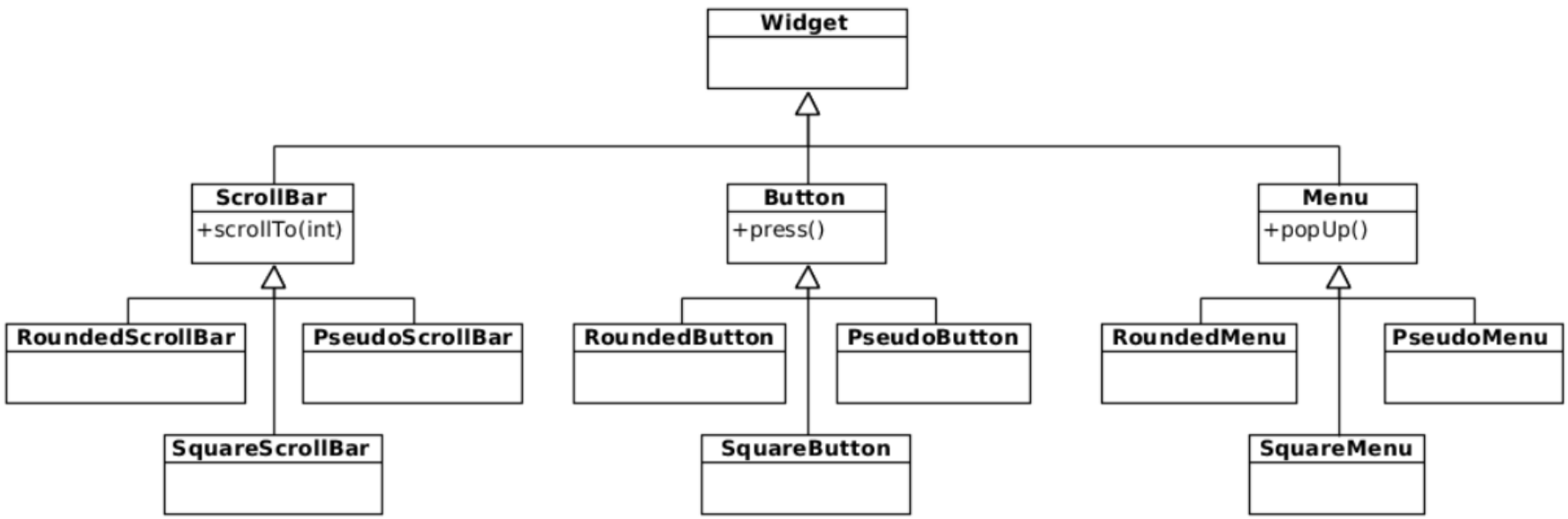
“Фабричный метод”, детали реализации



- Абстрактный Creator или реализация по умолчанию
  - Второй вариант может быть полезен для расширяемости
- Параметризованные фабричные методы
- Если язык поддерживает инстанциацию по прототипу (JavaScript, Smalltalk), можно хранить порождаемый объект
- Creator не может вызывать фабричный метод в конструкторе
- Можно сделать шаблонный Creator

“Абстрактная фабрика”, мотивация

- Хотим поддержать разные стили UI
  - Гибкая поддержка в архитектуре
  - Удобное добавление новых стилей



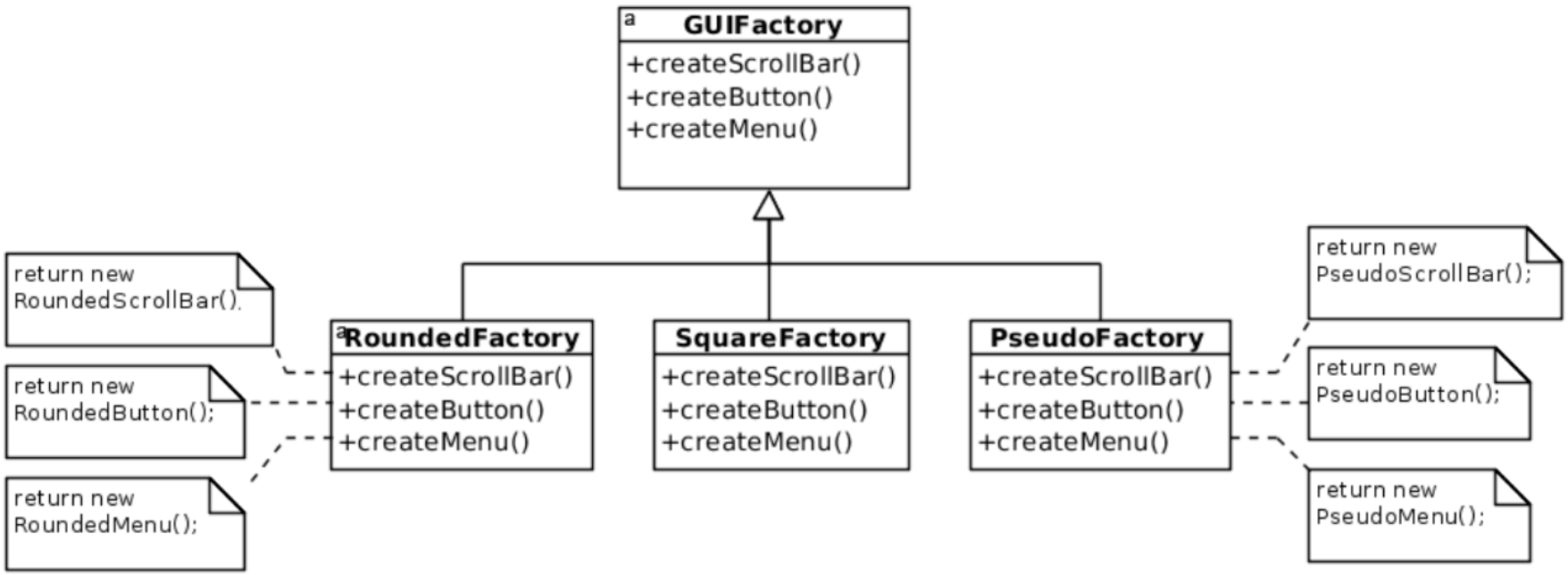
Создание виджетов

```
ScrollBar* bar = new RoundedScrollBar;
```

vs

```
ScrollBar* bar = guiFactory->createScrollBar();
```

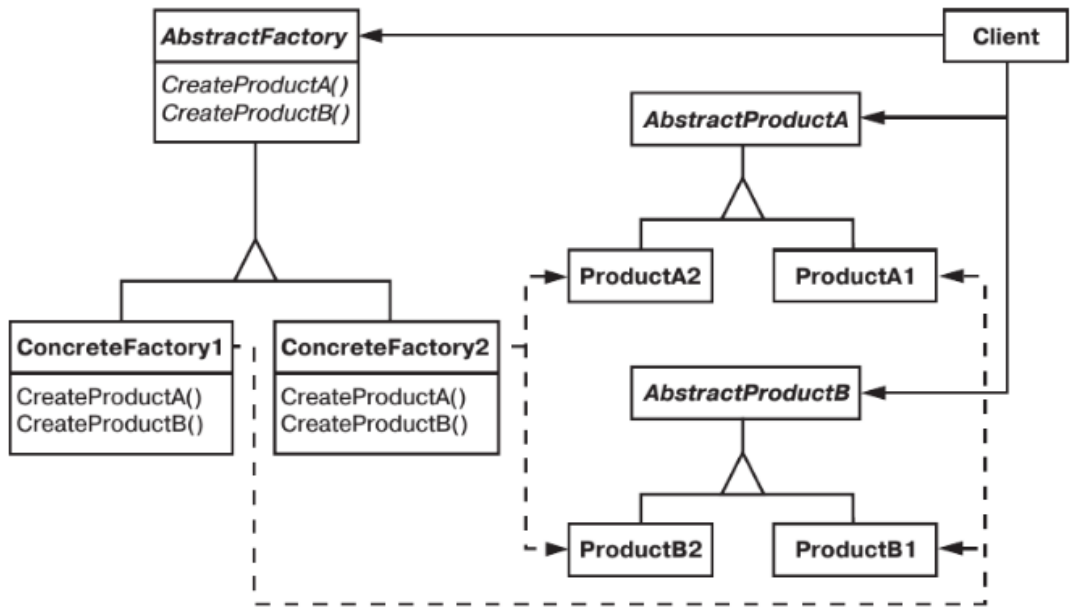
Фабрика виджетов



Паттерн “Абстрактная фабрика”

Abstract Factory

- Изолирует конкретные классы
- Упрощает замену семейств продуктов
- Гарантирует сочетаемость продуктов
- Поддержать новый вид продуктов непросто

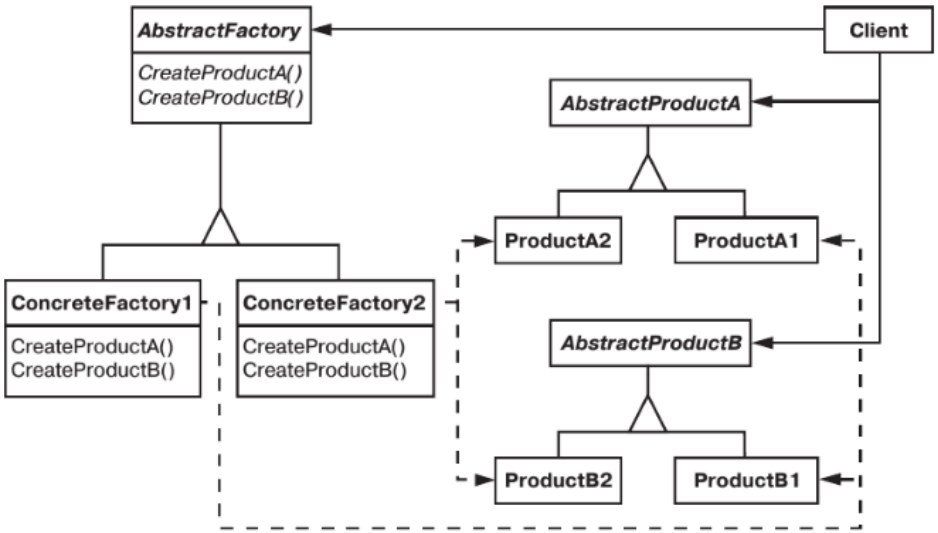


“Абстрактная фабрика”, применимость

- Система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты
- Система должна конфигурироваться одним из семейств составляющих ее объектов
- Взаимосвязанные объекты должны использоваться вместе
- Хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию

“Абстрактная фабрика”, детали реализации

- Хорошо комбинируются с паттерном “Одиночка”
- Если семейств продуктов много, то фабрика может инициализироваться *прототипами*, тогда не надо создавать сотню подклассов

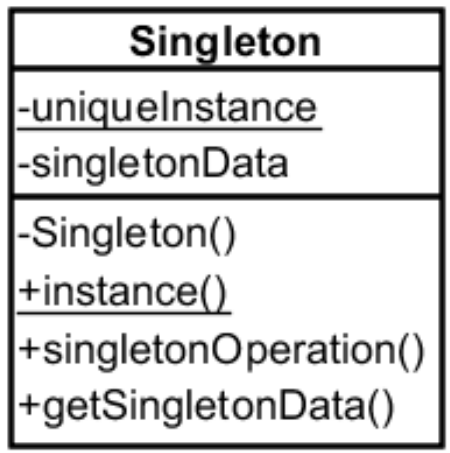


- Прототип на самом деле может быть классом (например, Class в Java)
- Если виды объектов часто меняются, может помочь параметризация метода создания
  - Может пострадать типобезопасность

Паттерн “Одиночка”

Singleton

- Гарантирует, что у класса есть только один экземпляр
- Предоставляет глобальный доступ к этому экземпляру
- Позволяет использовать подклассы без модификации клиентского кода



“Одиночка”, наивная реализация

```
public class Singleton {
    private static Singleton instance;

    private Singleton () {}

    public static Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

“Одиночка”, простая многопоточная реализация

```
public class Singleton {
    private static Singleton instance = new Singleton();

    private Singleton () {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

“Одиночка”, плохая многопоточная реализация

```
public class Singleton {
    private static Singleton instance;

    public static synchronized Singleton getInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```

Double-checked locking

Не делайте так

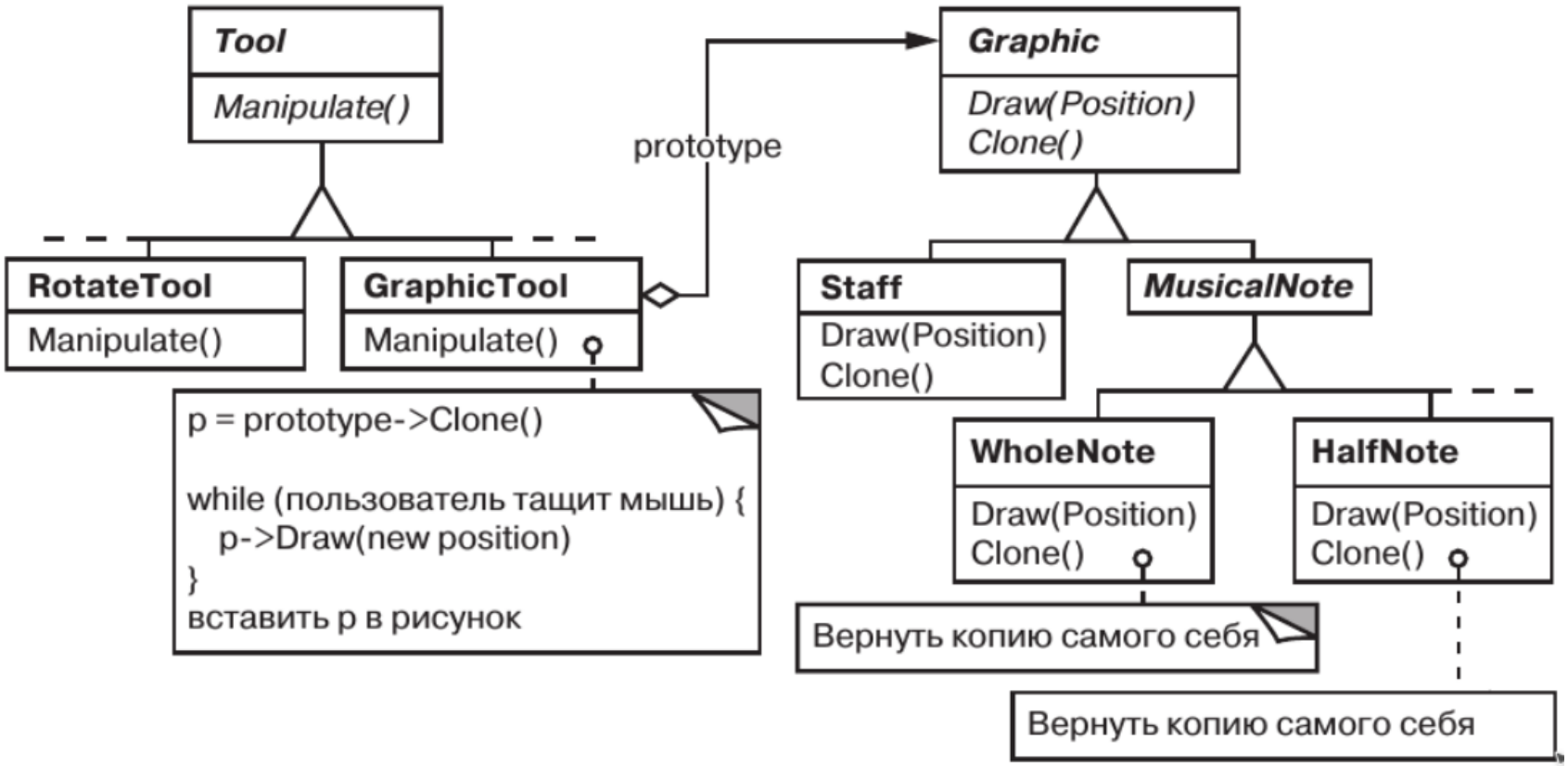
```
public class Singleton {
    private static volatile Singleton instance;

    public static Singleton getInstance() {
        Singleton localInstance = instance;
        if (localInstance == null) {
            synchronized (Singleton.class) {
                localInstance = instance;
                if (localInstance == null) {
                    instance = localInstance = new Singleton();
                }
            }
        }
        return localInstance;
    }
}
```

“Одиночка”, критика

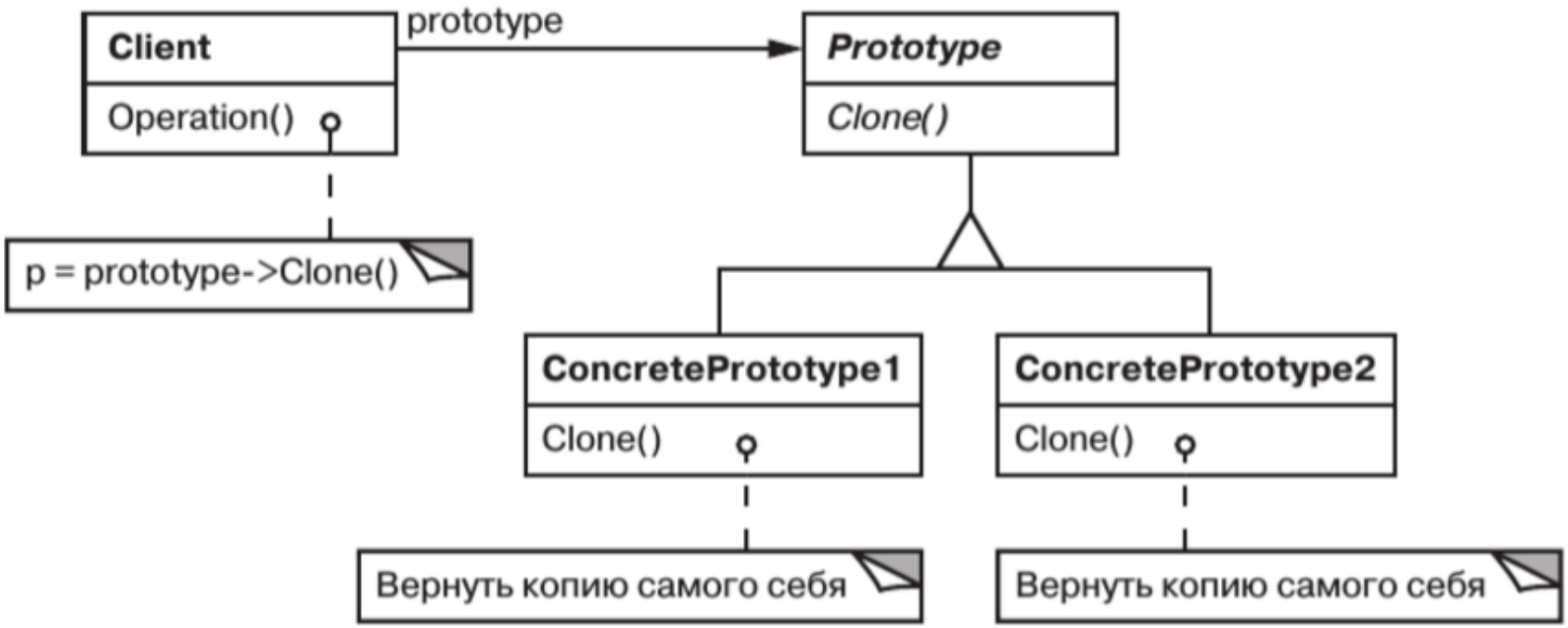
- Добавляет неочевидные зависимости по данным
  - По сути, хитрая глобальная переменная
- Усложняет тестирование
- Нарушает принцип единственности ответственности
- Сложно рефакторить, если потребуется несколько экземпляров

“Прототип”, мотивация



Патерн “Прототип”

Prototype

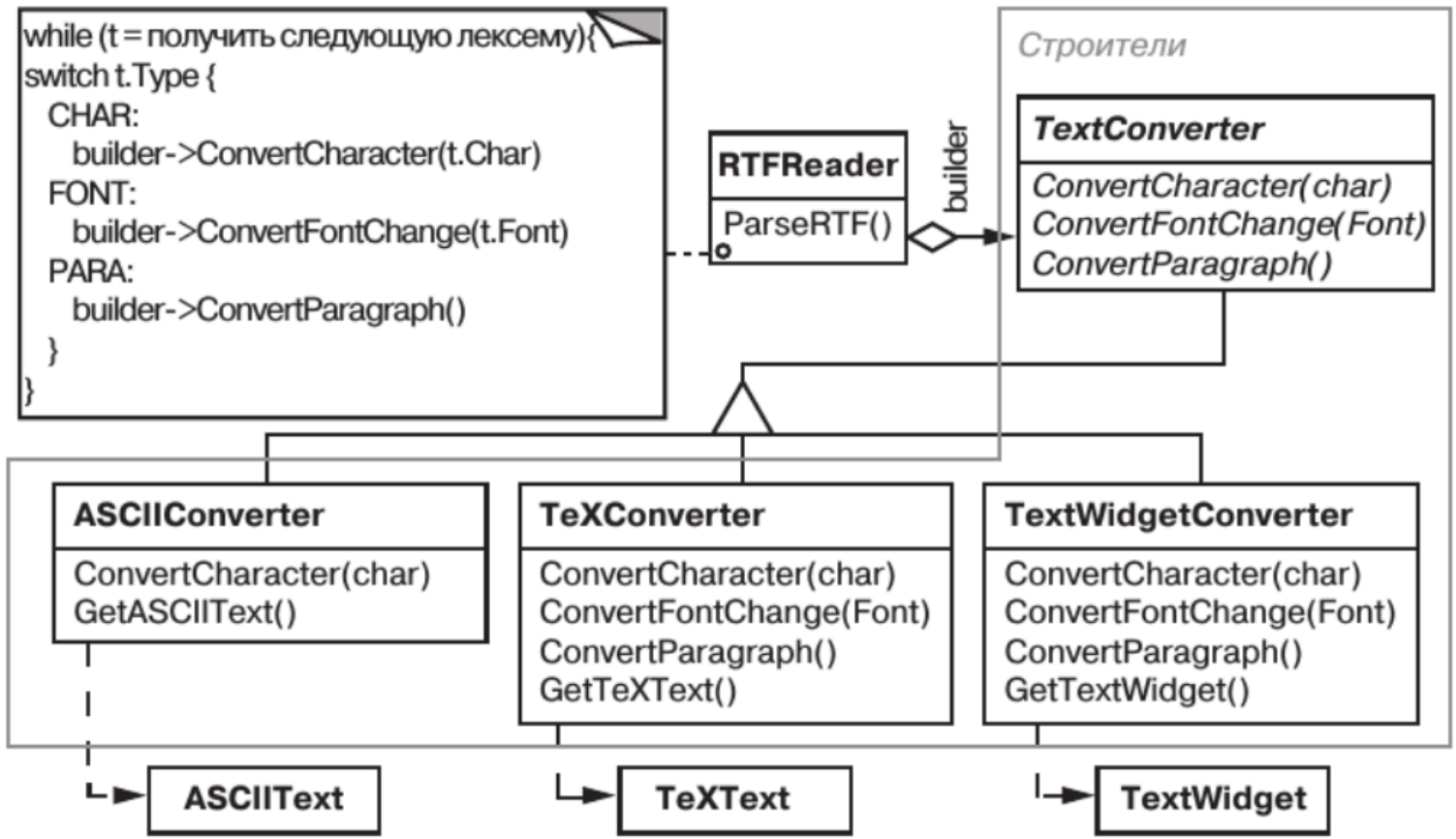


“Прототип”, детали реализации

- Паттерн интересен только для языков, где мало runtime-информации о типе (C++)
- Реестр прототипов, обычно ассоциативное хранилище
- Операция Clone
  - Глубокое и мелкое копирование
  - В случае, если могут быть круговые ссылки
  - Сериализовать/десериализовать объект (но помнить про идентичность)
- Инициализация клона
  - Передавать параметры в Clone — плохая идея

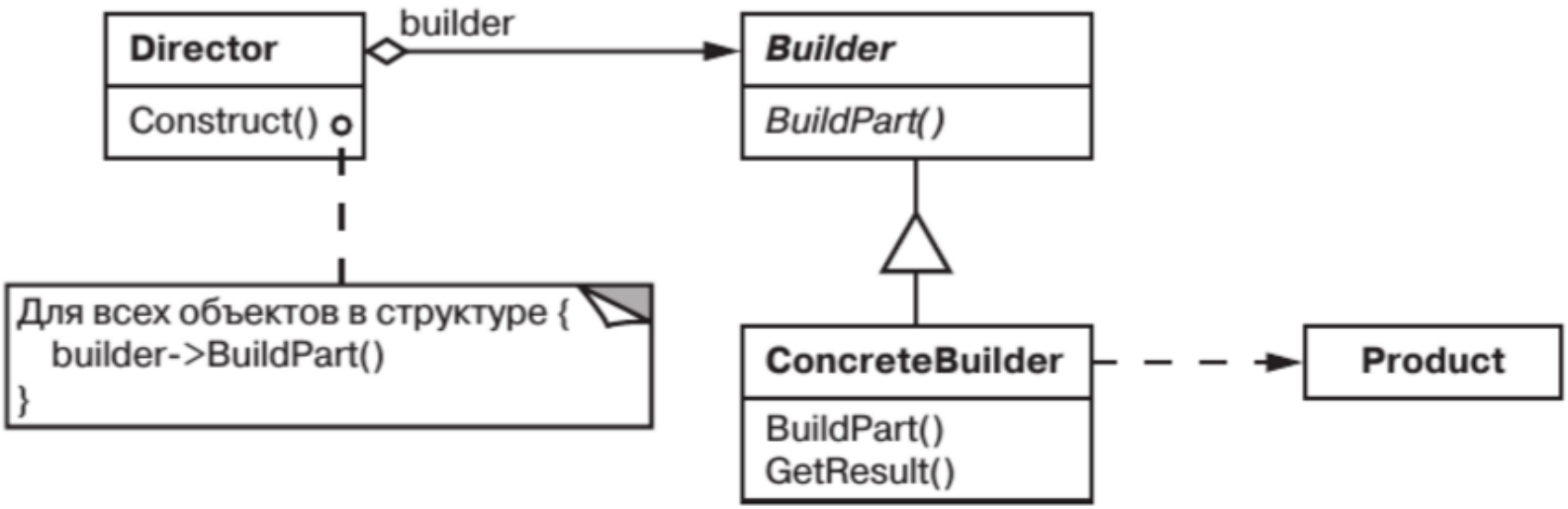
“Строитель”, мотивация

Конвертер текста

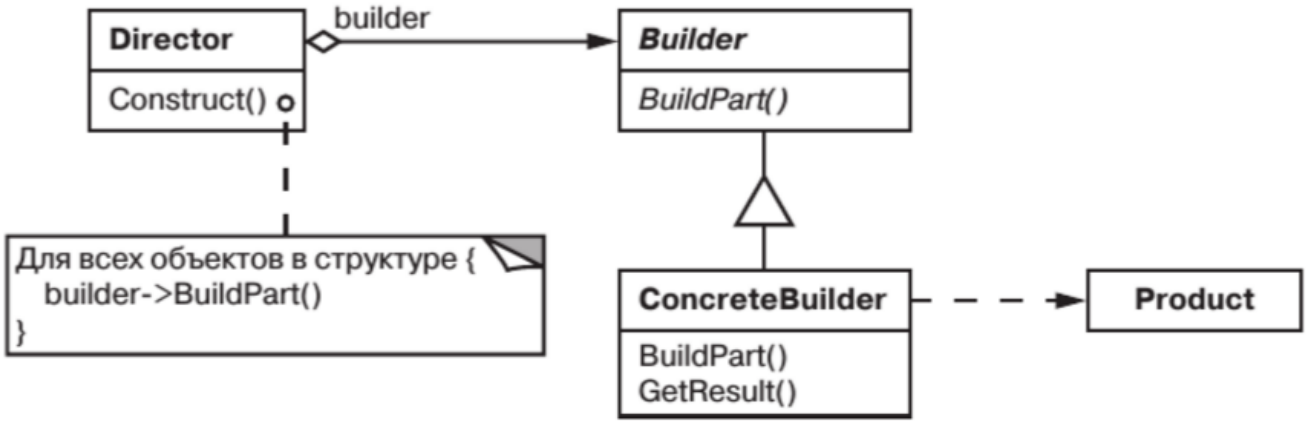


Патерн “Строитель”

Builder



“Строитель” (Builder), детали реализации



- Абстрактные и конкретные строители
  - Достаточно общий интерфейс
- Общий интерфейс для продуктов не требуется
  - Клиент конфигурирует распорядителя конкретным строителем, он же и забирает результат
- Пустые методы по умолчанию

“Строитель”, примеры

- StringBuilder
- Guava, подсистема работы с графами

```
MutableNetwork<Webpage, Link> webSnapshot =
        NetworkBuilder.directed()
    .allowsParallelEdges(true)
    .nodeOrder(ElementOrder.natural())
    .expectedNodeCount(100000)
    .expectedEdgeCount(1000000)
    .build();
```