

# Лекция 7/Практика 5: Порождающие шаблоны

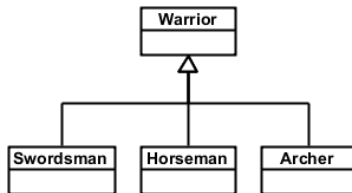
Юрий Литвинов  
y.litvinov@spbu.ru

21.04.2022

# “Фабричный метод” мотивация

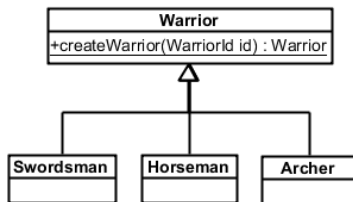
## Игра-стратегия

- ▶ Воины
  - ▶ Мечники
  - ▶ Конница
  - ▶ Лучники
- ▶ Хотим сделать здания, генерирующие юнитов



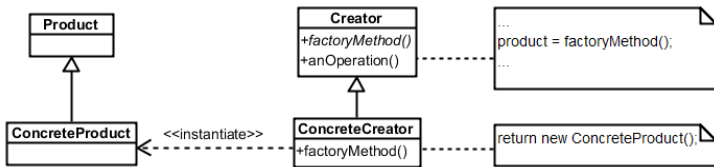
# Фабричный метод

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



# Паттерн “Factory Method”

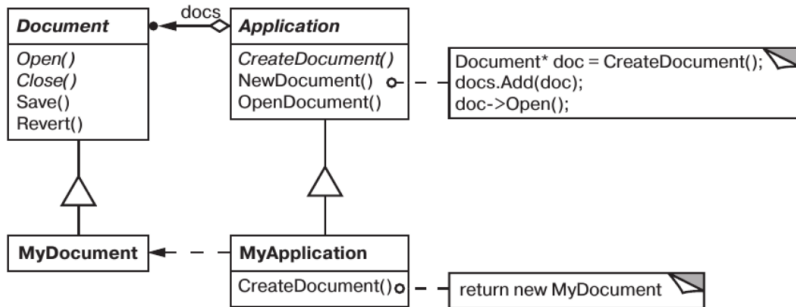
## Factory Method



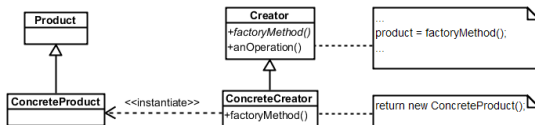
### ► Применимость:

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

# Пример, текстовый редактор



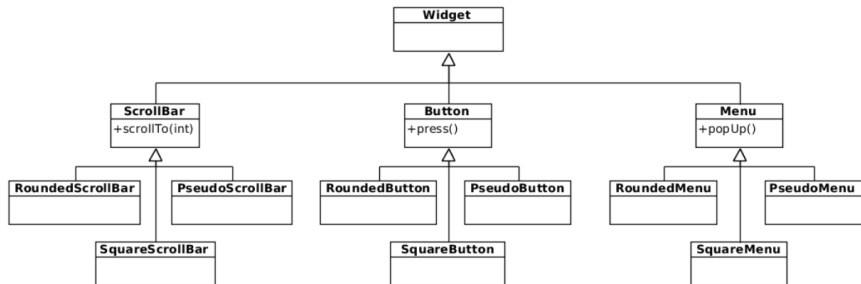
## “Фабричный метод”, детали реализации



- ▶ 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

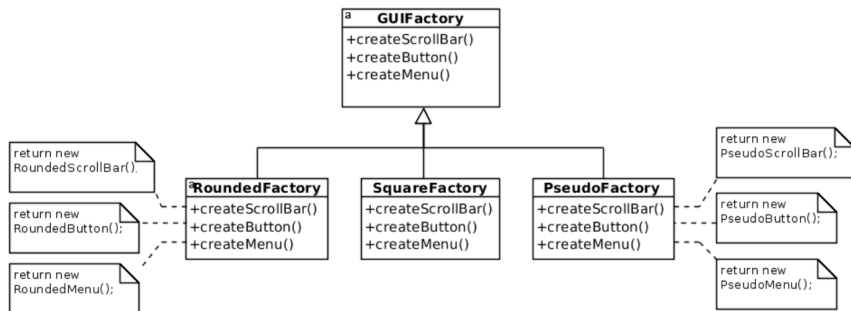
```
ScrollBar* bar = new RoundedScrollBar;
```

vs

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



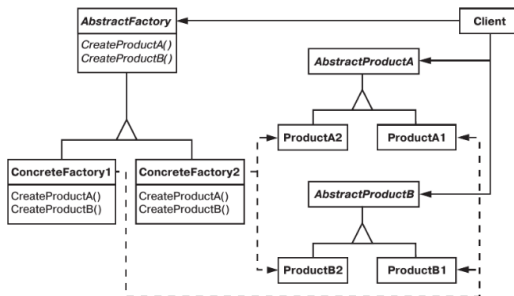
# Фабрика виджетов



# Паттерн “Абстрактная фабрика”

## Abstract Factory

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

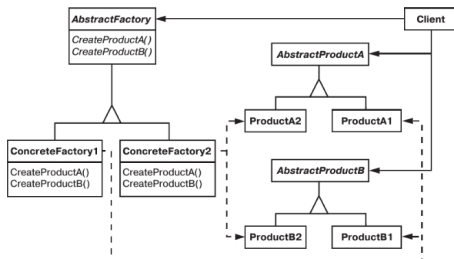


## “Абстрактная фабрика”, применимость

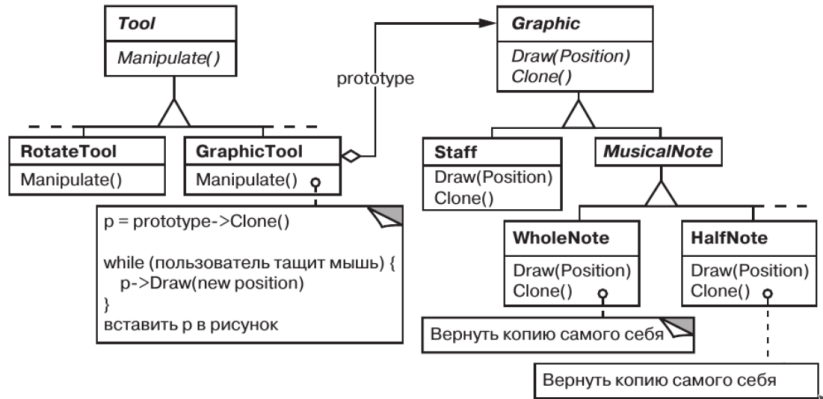
- ▶ Система не должна зависеть от того, как создаются, компонуются и представляются входящие в неё объекты
- ▶ Система должна конфигурироваться одним из семейств составляющих её объектов
- ▶ Взаимосвязанные объекты должны использоваться вместе
- ▶ Хотите предоставить библиотеку объектов, раскрывая только 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

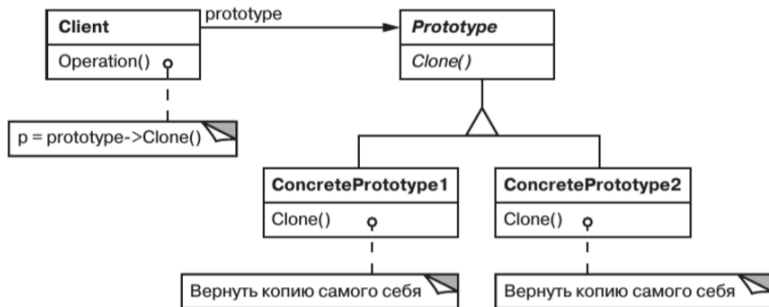


# “Прототип”, мотивация



# Паттерн “Прототип”

Prototype

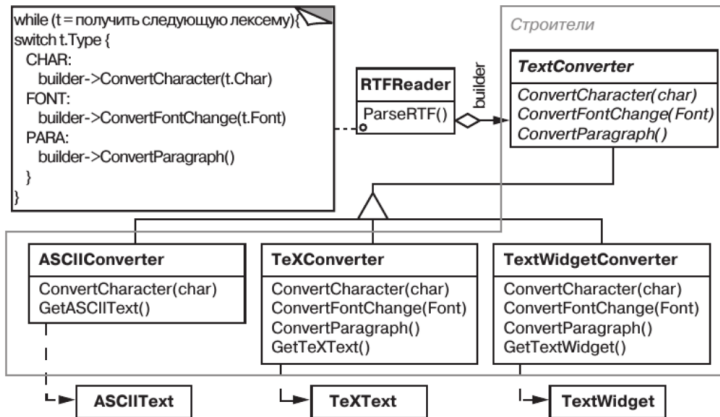


## “Прототип”, детали реализации

- ▶ 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

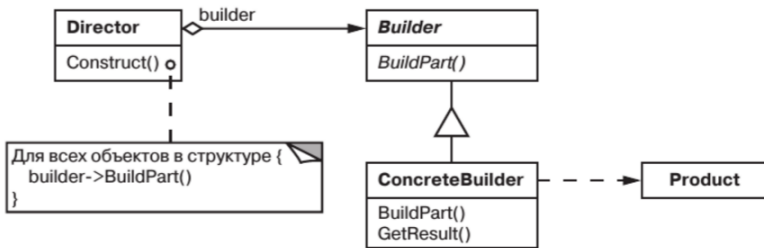
## Конвертер текста



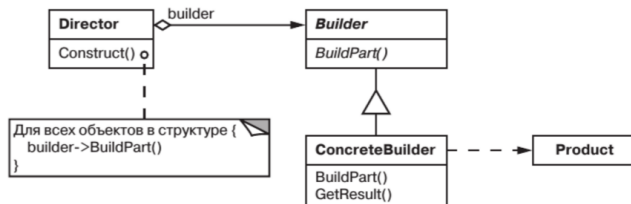


# Паттерн “Строитель”

## 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();
```

## Задание на остаток пары

Уточнить модель компьютерной игры Roguelike:

1. Используя шаблон “Строитель” для инициализации карты
2. Используя шаблон “Абстрактная фабрика” для создания мобов и предметов на карте
3. Используя шаблон “Прототип” для поддержки клонирования персонажей и предметов

Выложить модифицированные диаграммы классов в Teams