

# Лекция 7: Структурные и порождающие шаблоны

Юрий Литвинов  
yurii.litvinov@gmail.com

13.10.2020г

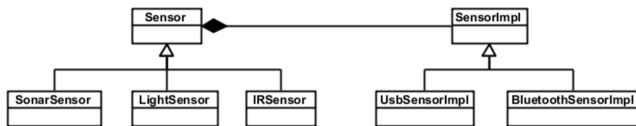
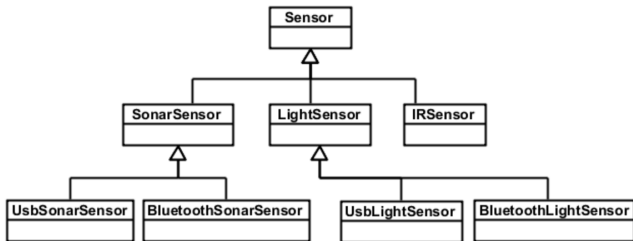
# Паттерн “Мост” (Bridge)

Отделяет абстракцию от реализации

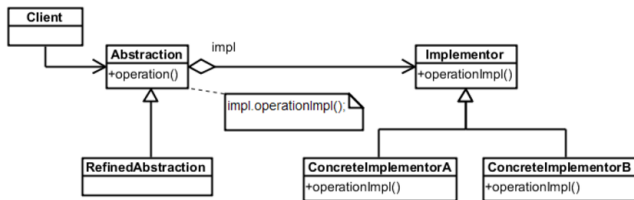
Пример:

- ▶ 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



# “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-a

- ▶ 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.

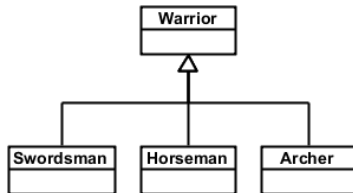
- ▶ Позитивно сказывается на времени компиляции программ на C++
- ▶ Позволяет менять реализацию независимо
  - ▶ Сохраняя бинарную совместимость

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

# “Фабричный метод” мотивация

## Игра-стратегия

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



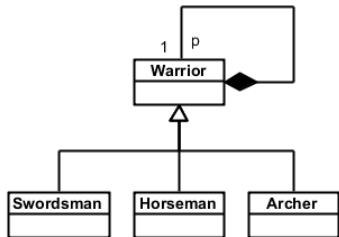


# Виртуальный конструктор

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

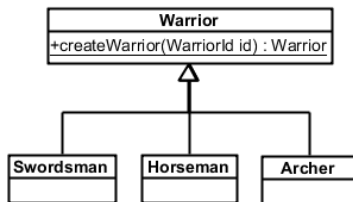
```
class Warrior
```

```
{
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;
};
```



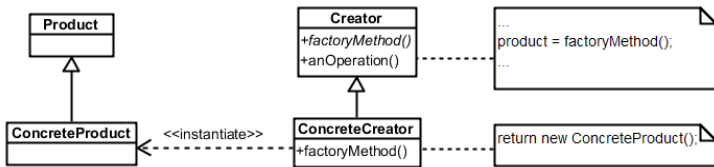
# Фабричный метод

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



# Паттерн “Factory Method”

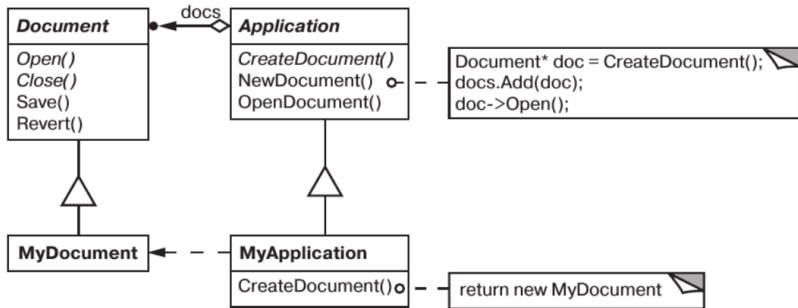
## Factory Method



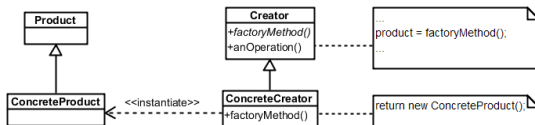
### ► Применимость:

- 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



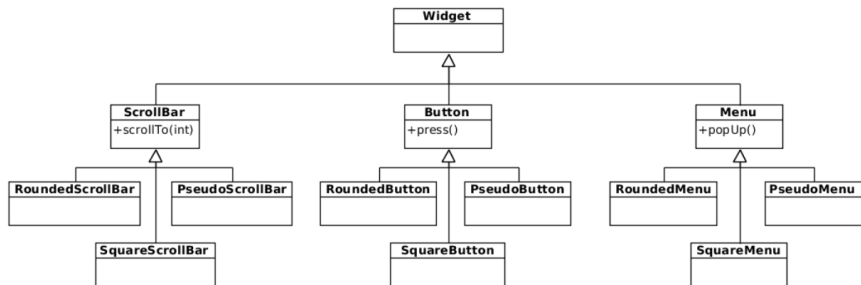
## “Фабричный метод”, детали реализации



- ▶ 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



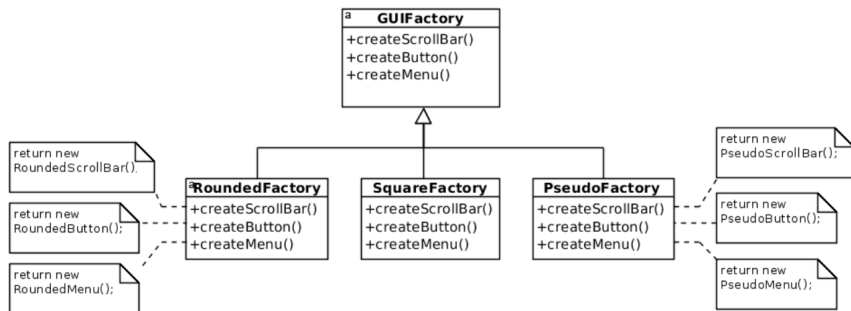
# Создание виджетов

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

vs

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

# Фабрика виджетов

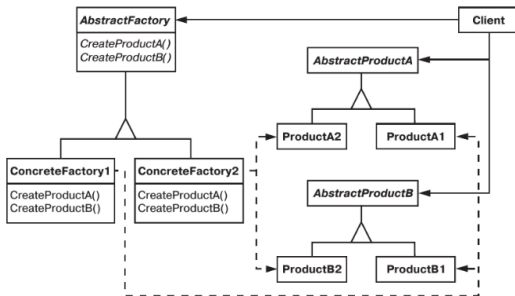




# Паттерн “Абстрактная фабрика”

## Abstract Factory

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

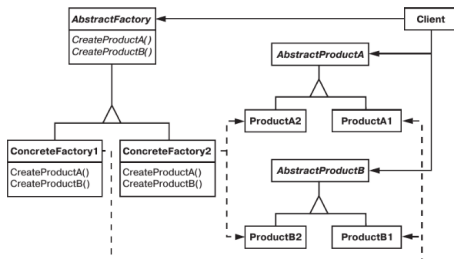


## “Абстрактная фабрика”, применимость

- ▶ 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

Singleton
<u>-uniqueInstance</u>
<u>-singletonData</u>
<u>-Singleton()</u>
<u>+instance()</u>
<u>+singletonOperation()</u>
<u>+getSingletonData()</u>

# “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

Более-менее хорошая многопоточная реализация

```
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;  
    }  
}
```



# “Multiton”

- ▶ Реестр одиночек, обеспечивает уникальность объекта по ключу
  - ▶ Сам создаёт объекты
  - ▶ Не даёт возможности зарегистрировать объект извне

Multiton
-instances : Map<Key, Multiton>
-Multiton() +instance() : Multiton

## “Одиночка”, критика

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

# Паттерн “Ленивая инициализация”

- ▶ Некоторое упрощение одиночки
- ▶ Действие не выполняется до тех пор, пока не нужен его результат
- ▶ Используется повсеместно, для ускорения запуска и экономии на редких вычислениях
  - ▶ Just-In-Time-компиляция
  - ▶ Ленивые структуры данных (списки в Haskell, seq в F#)
  - ▶ Ленивые вычисления (Haskell, Lazy<T> в .NET)
  - ▶ ...
- ▶ Имеет те же проблемы с многопоточностью, что и одиночка

# Паттерн “Пул объектов”, мотивация

## Потоки в .NET

- ▶ Класс Thread, конструктор создаёт поток и запускает в нём переданную операцию
- ▶ Поток уничтожается, когда операция завершилась
- ▶ Создание и остановка потоков — долгие операции
- ▶ Каждый поток требует системных ресурсов
- ▶ Нет смысла иметь больше потоков, чем ядер процессора

# Паттерн "Пул объектов"

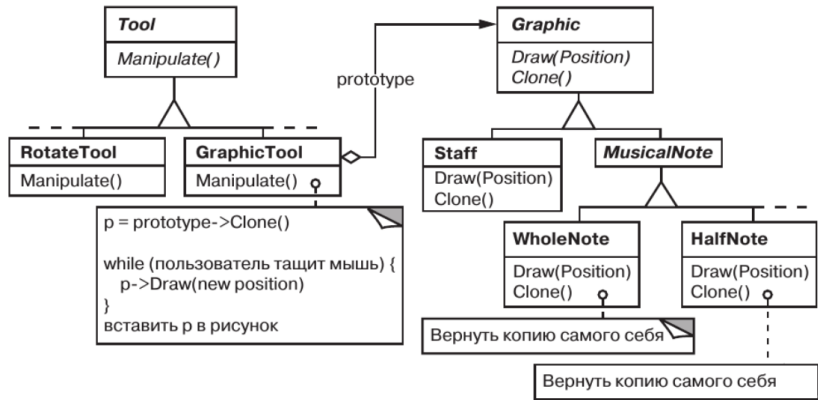
Решение: пул потоков в .NET

- ▶ Класс `ThreadPool`, синглтон
- ▶ Создаёт заранее  $N$  потоков, которые никогда не заканчиваются и ждут задач
- ▶ `QueueUserWorkItem` принимает задачу на исполнение
  - ▶ Например,  
`ThreadPool.QueueUserWorkItem(() => Console.WriteLine("Goodbye, world!"))`
- ▶ Если есть свободный поток, он начинает исполнять задачу
- ▶ Если свободных потоков нет, а задач много, создаётся новый поток
- ▶ Лишние потоки удаляются, если задач нет и число потоков больше  $N$

# Паттерн “Пул объектов”

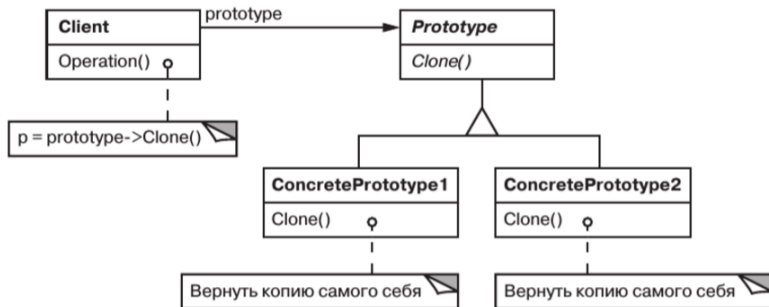
- ▶ Применяется, когда объекты создавать сложно, но каждый объект нужен лишь ненадолго
- ▶ Желательно, чтобы на поддержание объектов в пуле не требовалось много ресурсов, либо объектов в пуле было мало
  - ▶ Например, создать 50000 сетевых соединений “заранее” может быть плохой идеей
- ▶ Следует применять с осторожностью в языках со сборкой мусора — пул держит ссылки на объекты
  - ▶ К тому же, в таких языках new обрабатывает мгновенно
- ▶ Следует помнить про многопоточность
  - ▶ Как правило, методы пула требуют синхронизации

# “Прототип”, мотивация



# Паттерн “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