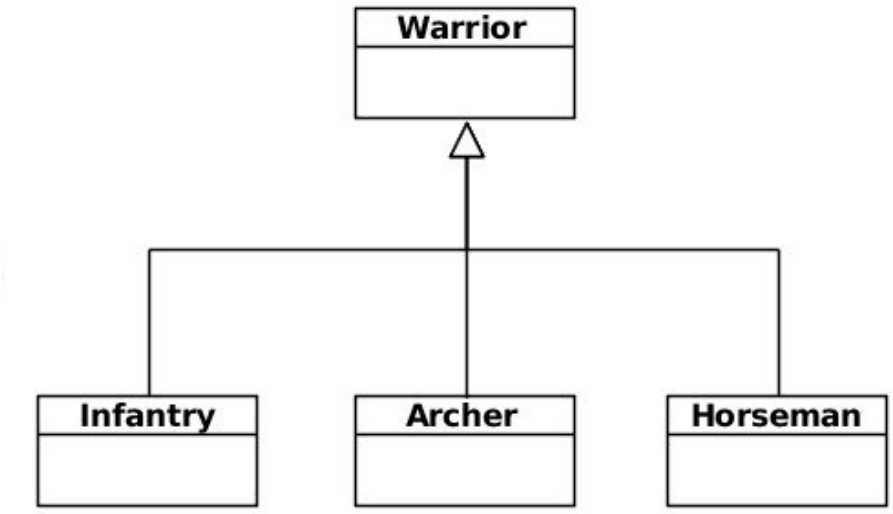
Проектирование ПО

Лекция 10: Порождающие шаблоны

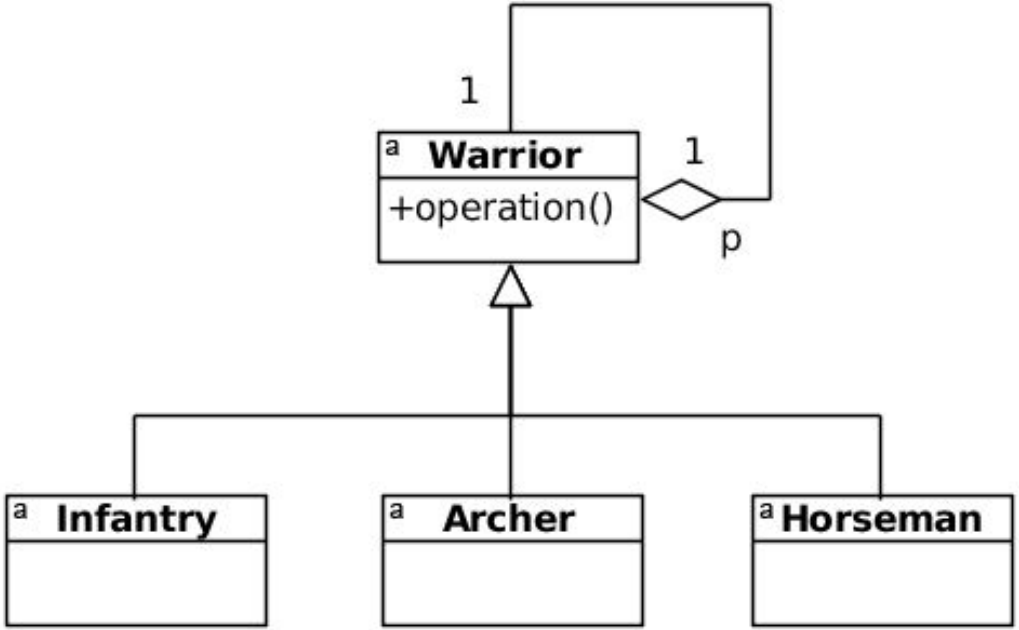
Тимофей Брыксин
timofey.bryksin@gmail.com

Пример: игра-стратегия

- воины
  - пехота
  - лучники
  - конница
- передвижение по полю
- характеристики
  - общие
  - особые



Виртуальный конструктор



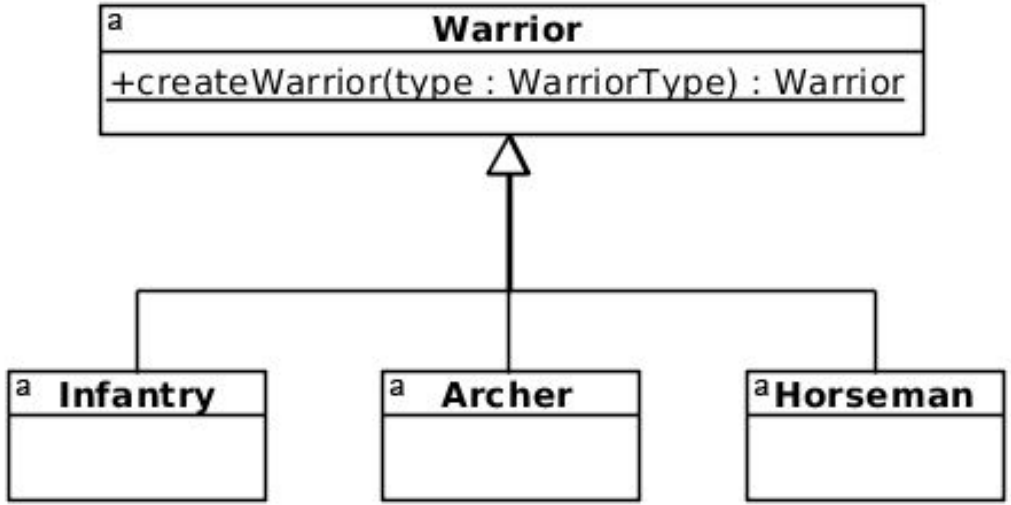
Виртуальный конструктор: реализация

```
enum Warrior_ID { Infantryman_ID, Archer_ID, Horseman_ID };

class Warrior
{
 public:
   Warrior( Warrior_ID id )
   {
      if (id == Infantryman_ID) p = new Infantryman;
      else if (id == Archer_ID) p = new Archer;
      else if (id == Horseman_ID) p = new Horseman;
      else assert( false);
   }
   virtual void info() { p->info(); }
   virtual ~Warrior() { delete p;  p=0; }
 private:
   Warrior* p;
};
```

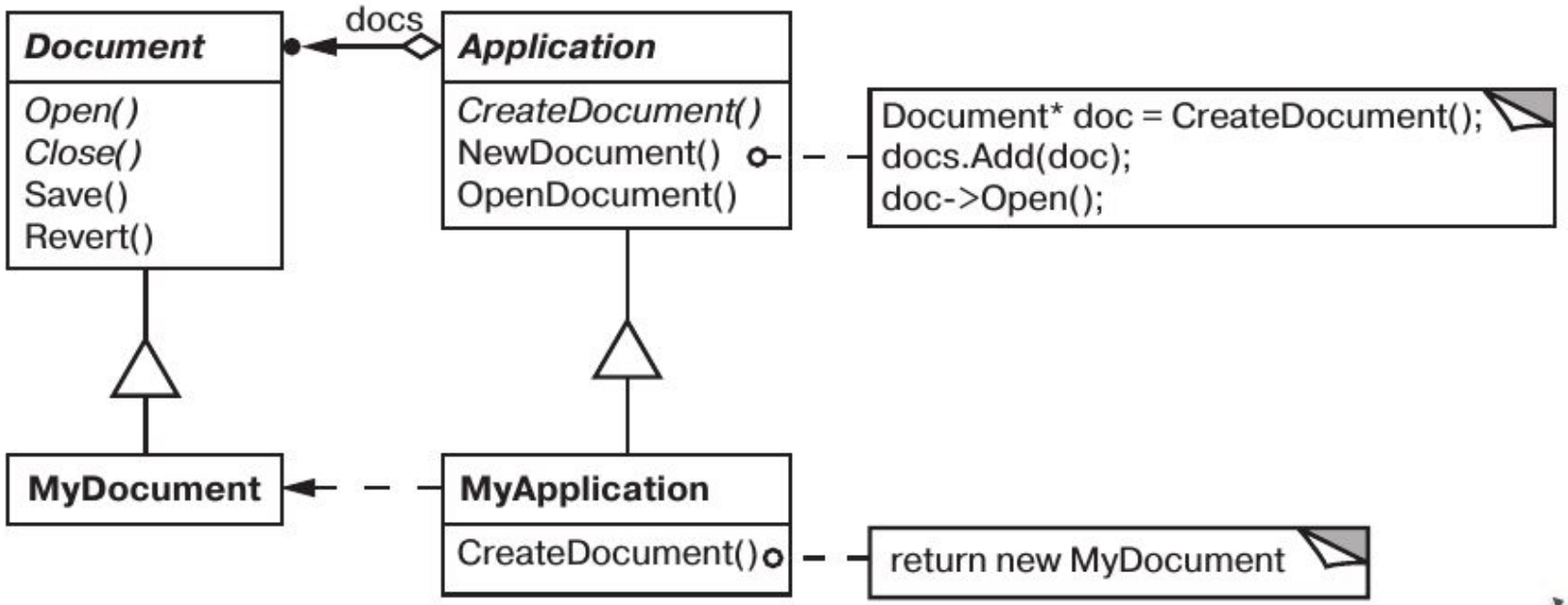
```
class Infantry: public Warrior
{
 public:
   void info() { cout << "Infantry" << endl; }
 private:
   Infantryman(): Warrior() {}
   Infantryman(Infantryman&);
   friend class Warrior;
};

int main()
{
   vector<Warrior*> v;
   v.push_back( new Warrior( Infantryman_ID));
   v.push_back( new Warrior( Archer_ID));
   v.push_back( new Warrior( Horseman_ID));

   for(int i=0; i<v.size(); i++)
      v[i]->info();
}
```

Паттерн Factory Method

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



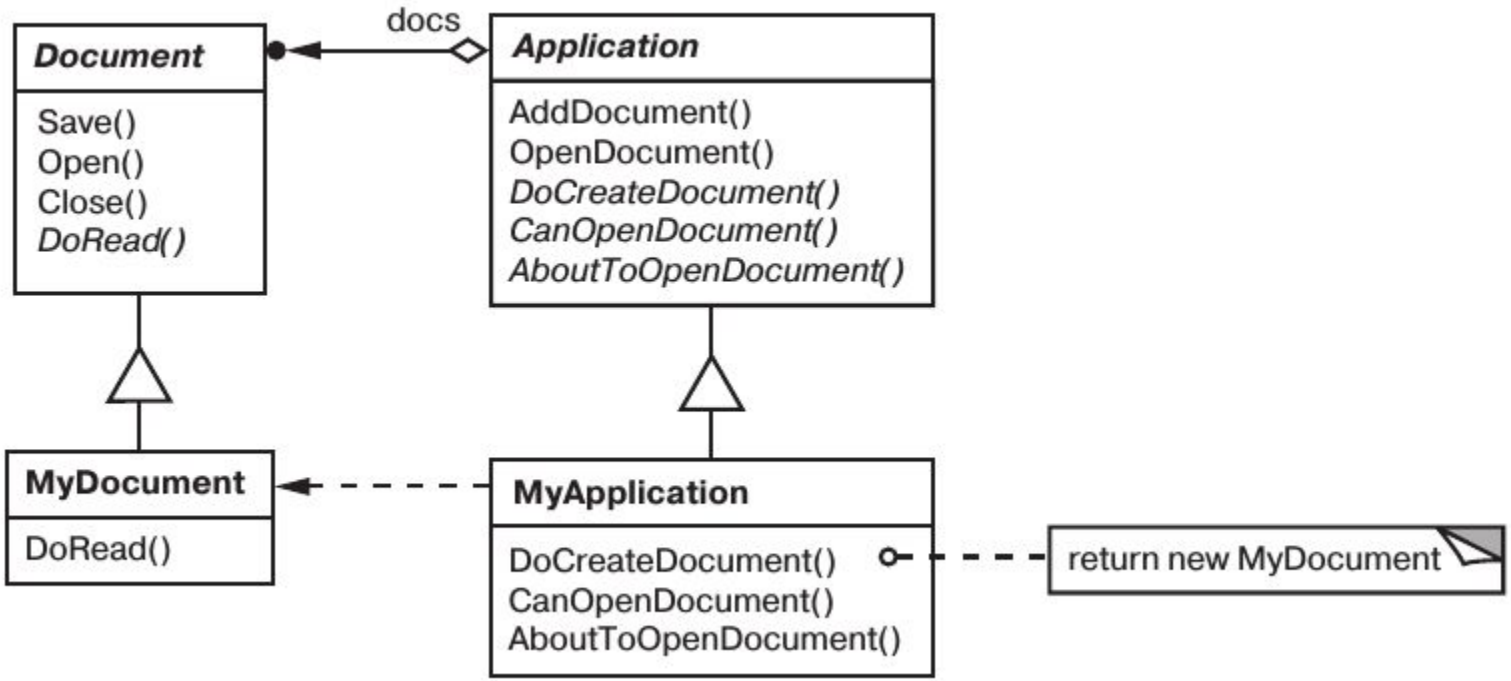
MDI-приложение



Фабричный метод: 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

Паттерн Шаблонный метод

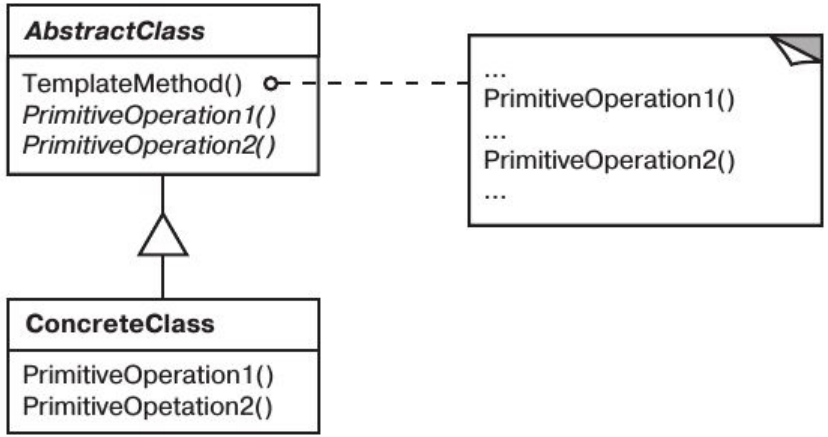


Реализация OpenDocument

```
void Application::OpenDocument (const char* name) {
    if (!CanOpenDocument(name)) {
        return;
    }
    Document* doc = DoCreateDocument();

    if (doc) {
        _docs->AddDocument(doc);
        AboutToOpenDocument(doc);
        doc->Open();
        doc->DoRead();
    }
}
```

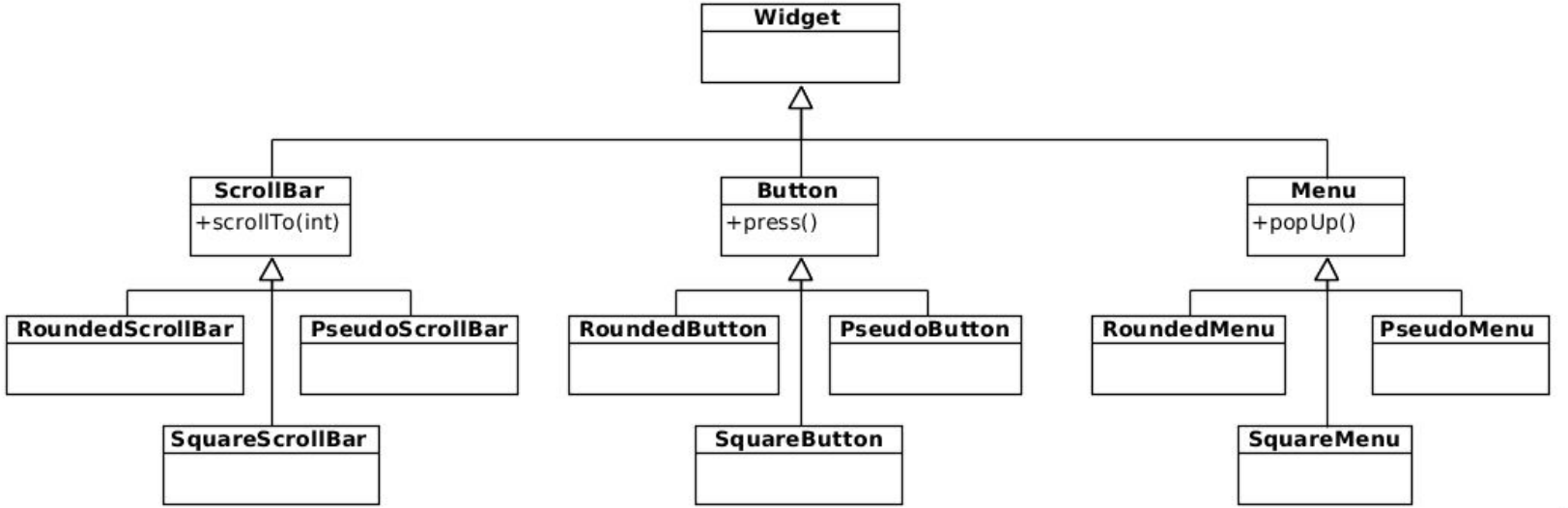
Шаблонный метод: применимость

- задание инварианта алгоритма, поведение в потомках
- вынесение общего кода в базовый класс
- управление точками расширения класса



Настройка внешнего вида редактора

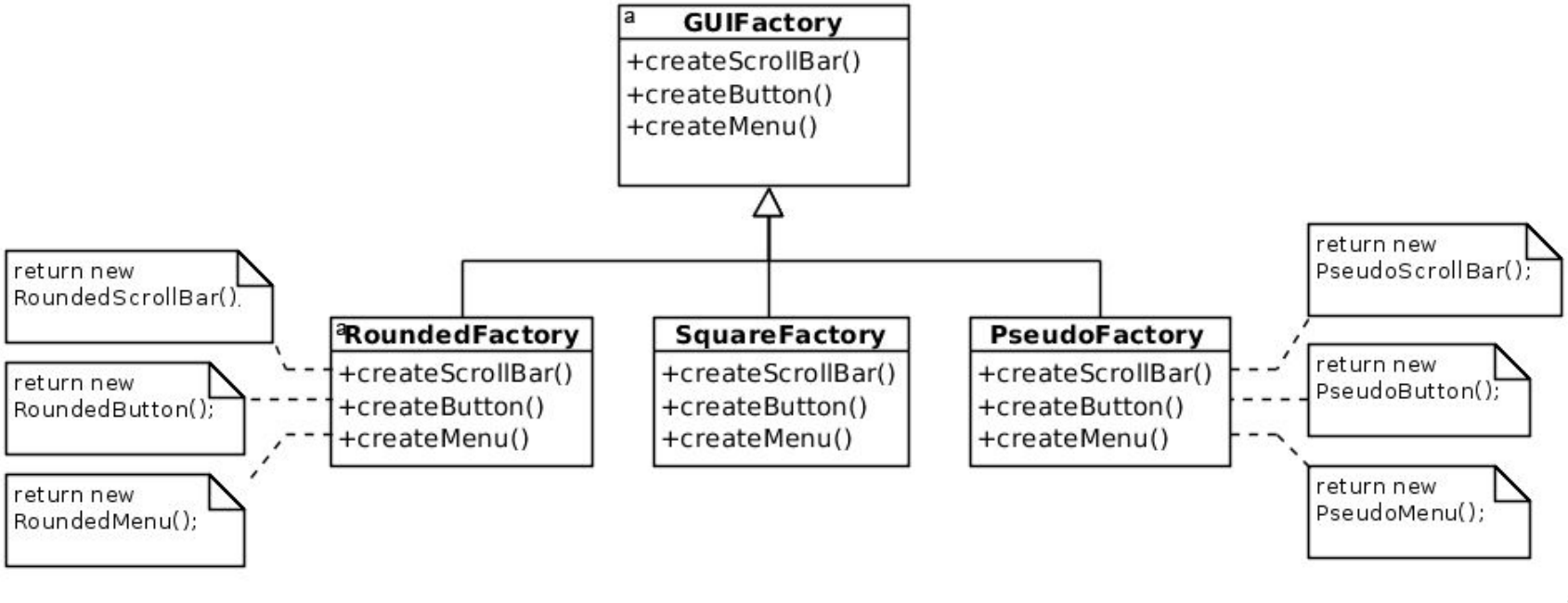
- хотим поддержать разные стили UI
  - гибкая поддержка в архитектуре
  - удобное добавление новых стилей



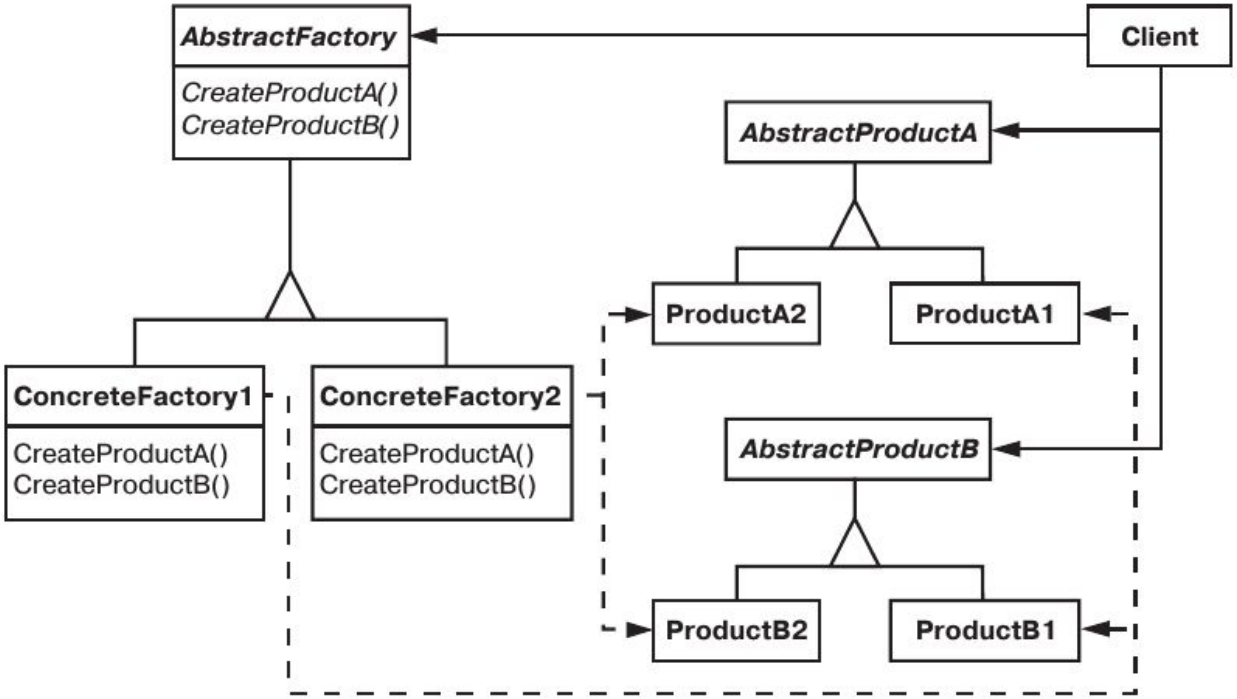
Создание виджетов

- ScrollBar* bar = new RoundedScrollBar;
- ScrollBar* bar = guiFactory->createScrollBar();

Фабрика виджетов



Абстрактная фабрика



Абстрактная фабрика

- изолирует конкретные классы
- упрощает замену семейств продуктов
- гарантирует сочетаемость продуктов

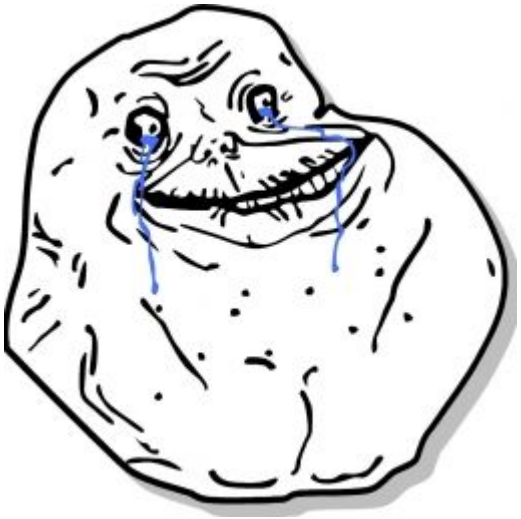
- поддержать новый вид продуктов непросто

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

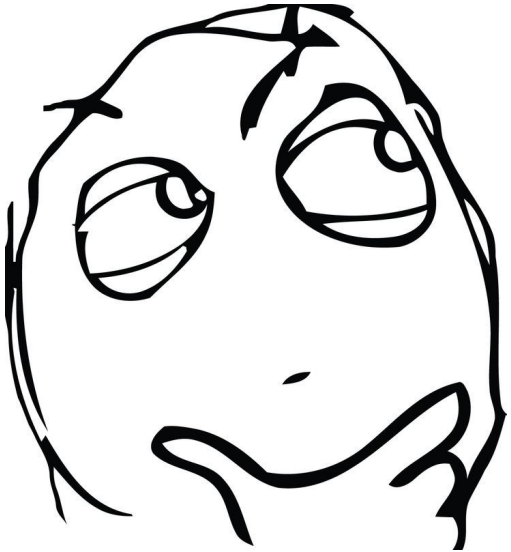
Паттерн Одиночка

- экземпляр класса в единственном экземпляре



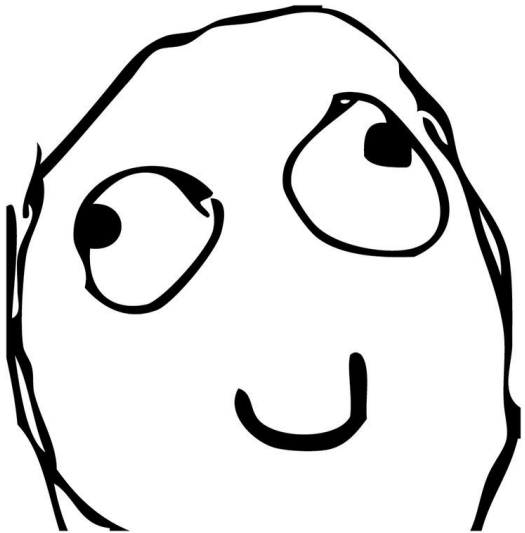
Наивная реализация

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

    private Singleton (){}

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



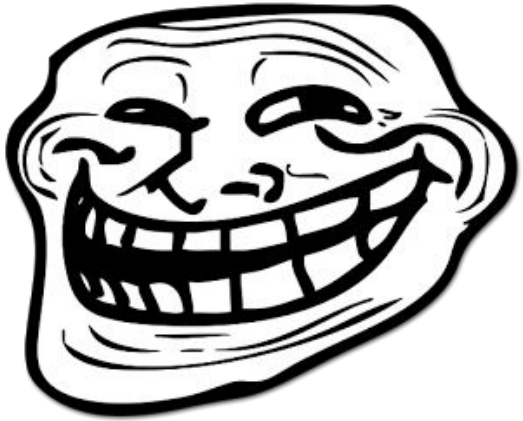
Многопоточный вариант

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

    private Singleton (){}

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



И это тоже синглтон

```
public enum Singleton {
    INSTANCE;
}
```

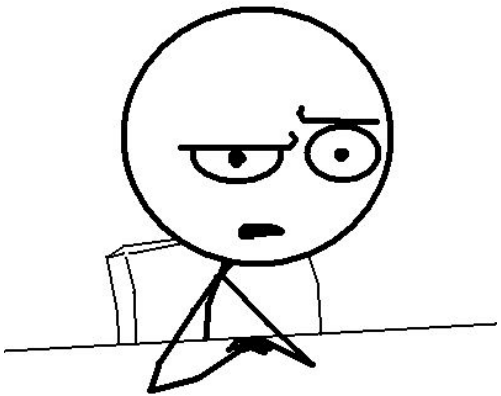


Пробуем ленивость и потокобезопасность

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

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

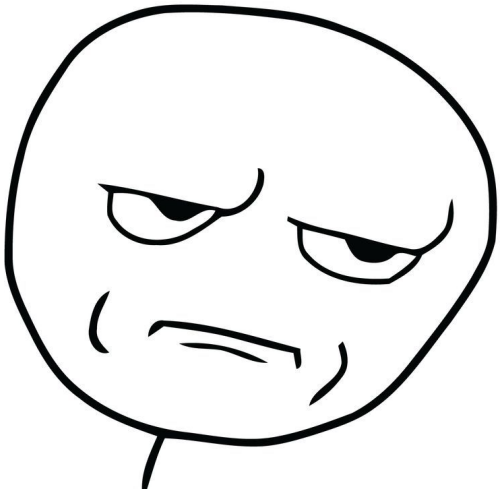
Double-checked locking

```
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;
   }
}
```

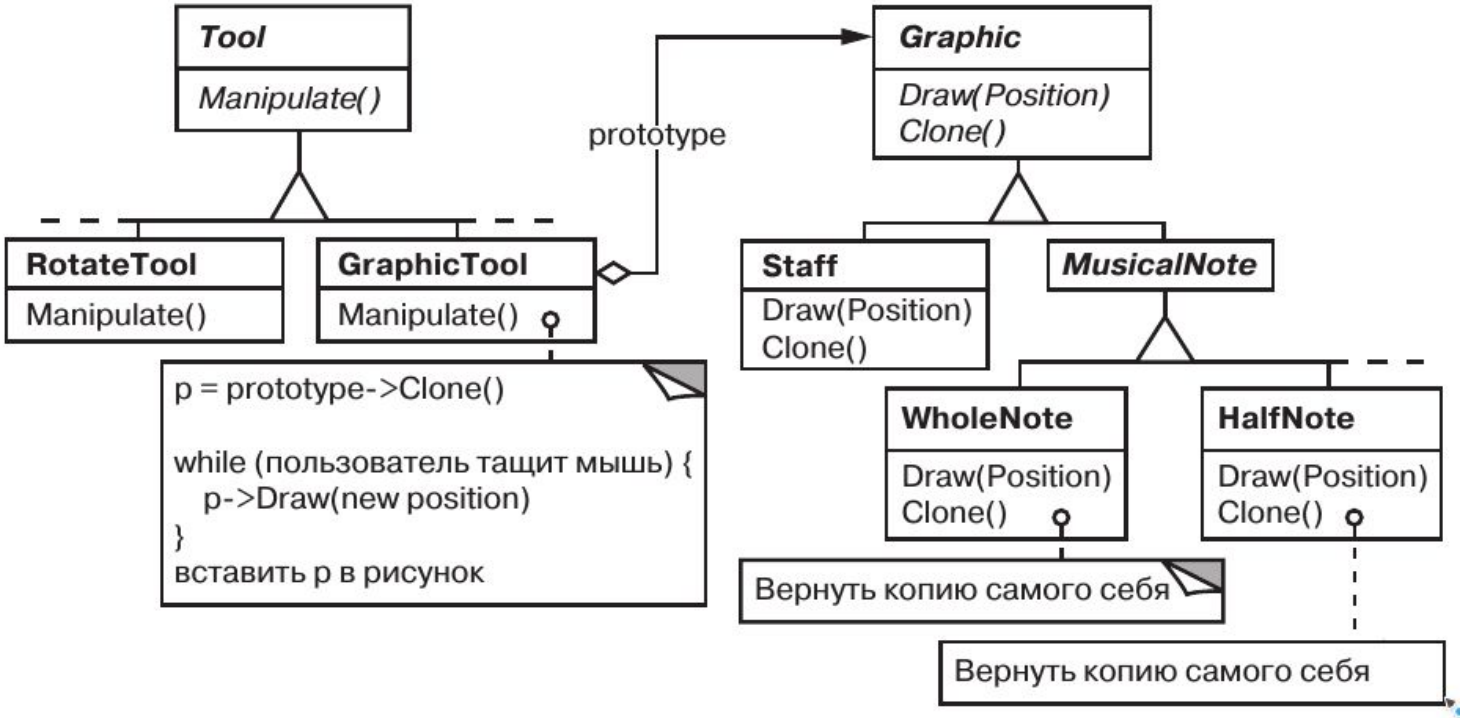


Критика

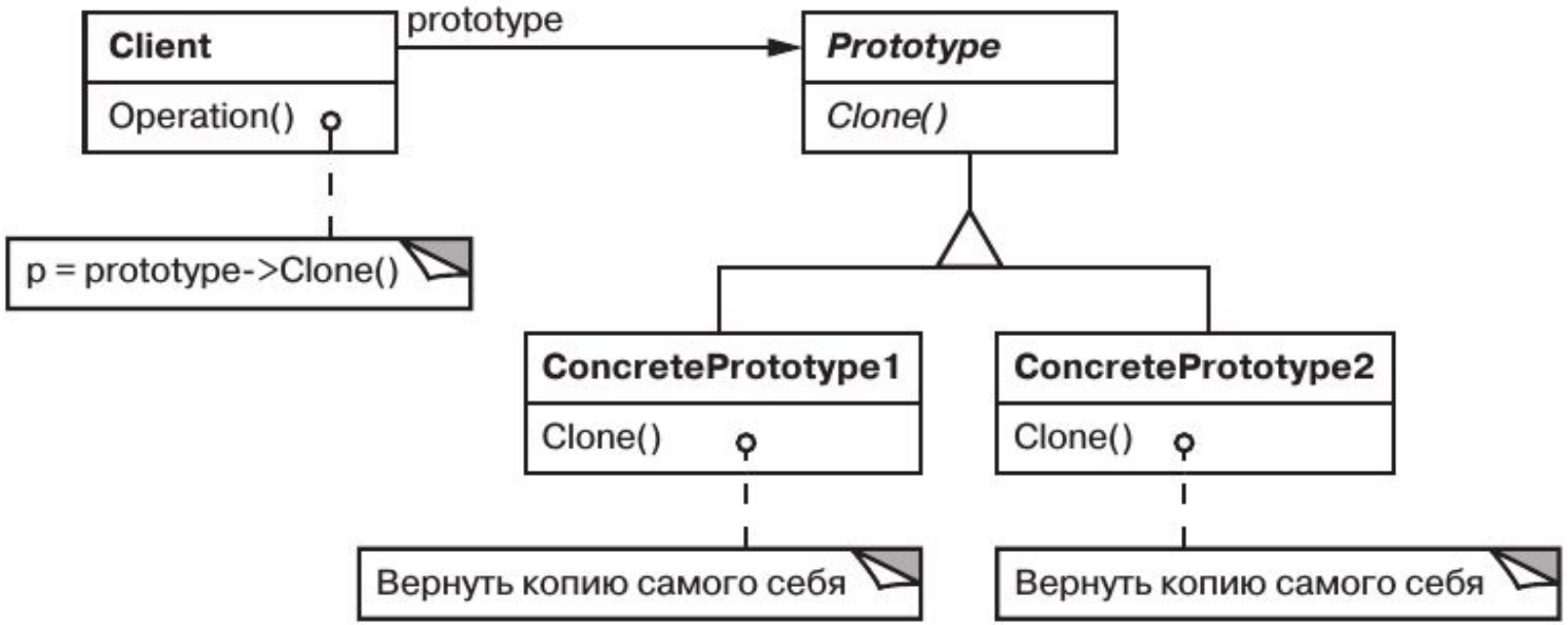
- нарушает SPR
- зависимость не видно в контрактах классов
- глобальное состояние
- снижает тестируемость



Музыкальный редактор



Паттерн Prototype

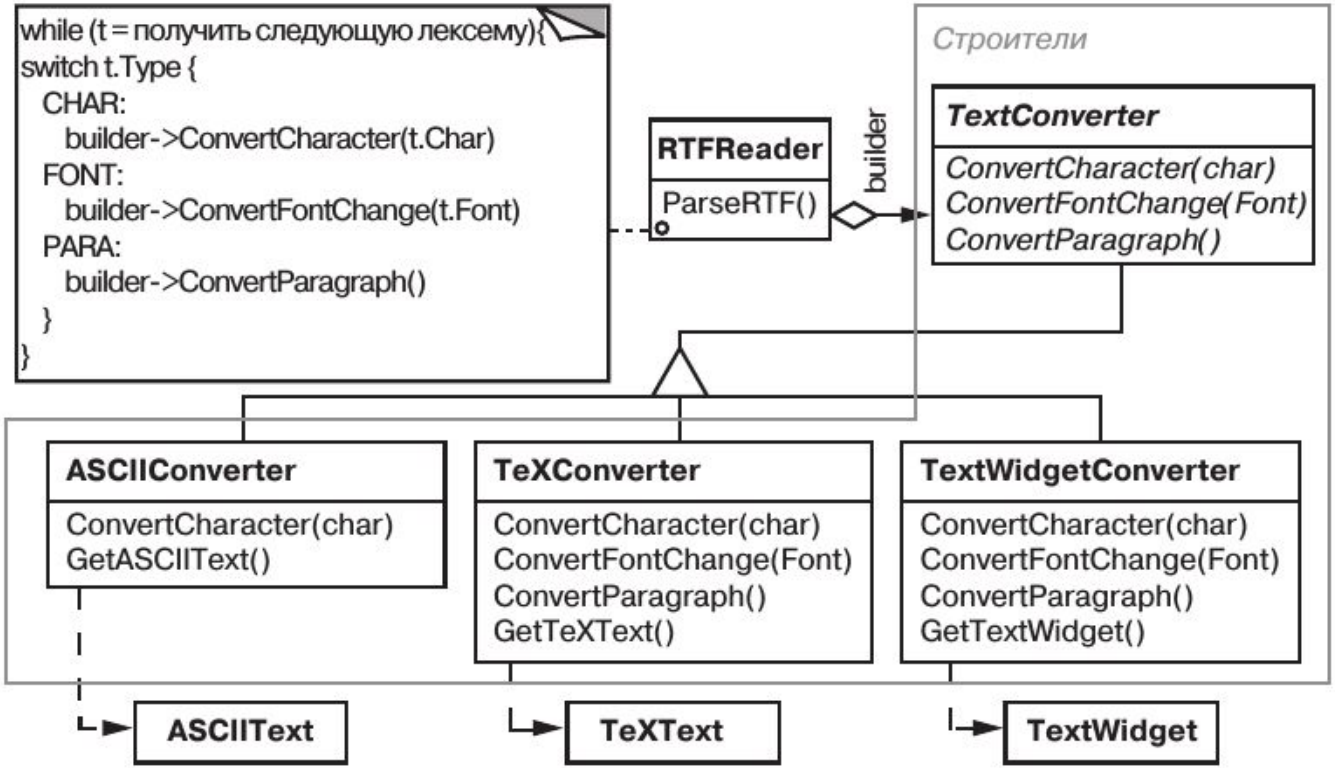


Прототип: применимость

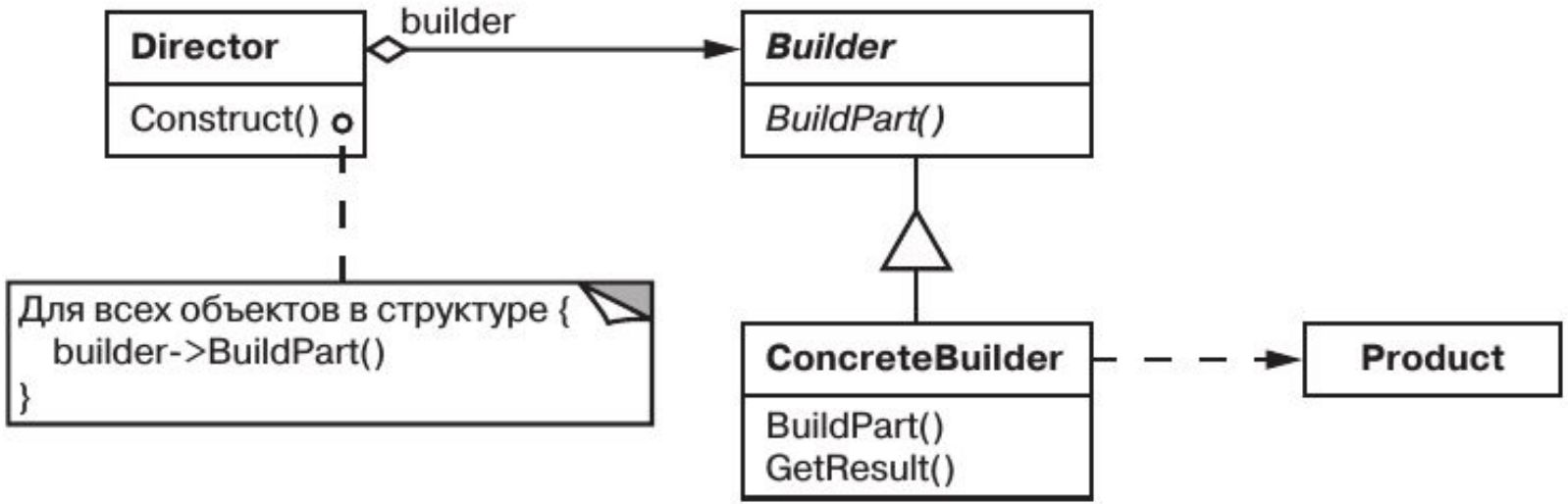
- инстанцируемые классы определяются во время выполнения
- хотим избежать построения иерархий классов или фабрик
- экземпляры класса могут находиться в одном из не очень большого числа различных состояний

- нужно реализовывать операцию clone()

Конвертер текста



Паттерн Builder



Телескопические конструкторы

```
public class NutritionFacts {
   private final int servingSize; // (ml_) required
   private final int servings: // (per container) required
   private final int calories; // optional
   private final int fat; // (g) optional
   private final int sodium; // (mg) optional
   private final int carbohydrate; // (g) optional

   public NutritionFacts(int servingSize, int servings) {
       this(servingSize, servings, 0);
   }

   public NutritionFacts(int servingSize, int servings,
                         int calories) {
       this(servingSize, servings, calories, 0);
   }

   public NutritionFacts(int servingSize, int servings,
                         int calories, int fat) {
       this(servingSize, servings, calories, fat, 0);
   }

   public NutritionFacts(int servingSize, int servings,
                         int calories, int fat, int sodium) {
       this(servingSize, servings, calories, fat, sodium, 0
   }

   public NutritionFacts(int servingSize, int servings,
                         int calories, int fat, int sodium,
                        int carbohydrate) {
       this.servingSize = servingSize;
       this.servings = servings;
       this.calories = calories;
       this.fat = fat;
       this.sodium = sodium;
       this.carbohydrate = carbohydrate;
   }
}
```

Вариант 1: куча сеттеров

```
NutritionFacts cocaCola = new NutritionFacts();
cocaCola.setServingSize(240);
cocaCola.setServings(8);
cocaCola.setCalories(100);
cocaCola.setSodium(35);
cocaCola.setCarbohydrate(27);
```

Вариант 2: builder!

```
public class NutritionFacts {
   private final int servingSize;
   private final int servings;
   private final int calories;
   private final int fat;
   private final int sodium;
   private final int carbohydrate;

   public static class Builder {
       // Required parameters
       private final int servingSize;
       private final int servings;
       // Optional parameters
       private int calories = 0;
       private int fat = 0;
       private int carbohydrate = 0;
       private int sodium = 0;
       public Builder(int servingSize, int servings) {
           this.servingSize = servingSize;
           this.servings = servings;
       }
       public Builder calories(int val)
       { calories = val; return this; }
       public Builder fat(int val)
       { fat = val; return this; }
       public Builder carbohydrate(int val)
       { carbohydrate = val; return this; }
       public Builder sodium(int val)
       { sodium = val; return this; }
       public NutritionFacts build() {
           return new NutritionFacts(this);
       }
   }

   private NutritionFacts(Builder builder) {
       servingSize = builder.servingSize;
       servings = builder.servings;
       calories = builder.calories;
       fat = builder.fat;
       sodium = builder.sodium;
       carbohydrate = builder.carbohydrate;
   }
}
```

Builder rocks! \m/

```
NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).
        calories(100).sodium(35).carbohydrate(27).build();
```

Строитель: применимость

- алгоритм создания не должен зависеть от того, из каких частей объект состоит
- процесс конструирования должен обеспечивать различные представления конструируемого объекта
- хочется конструировать сложный объект по частям, сохраняя его целостность

Обсуждение

- абстрагирование создания объектов
  - порождение подклассов
  - композиция объектов (фабричный объект)