DESIGN BY CONTRACT

Что такое design by contract?

- Операции представляют собой черный ящик
- Клиент и сервер
- Взаимная договоренность или контракт
- Пример: турникет в метро

История

Бертран Мейер, 1986, создатель языка Eiffel.

Компоненты должны быть снабжены точными спецификациями своих интерфейсов (контрактами использования).

Большинство современных языков программирования не имеют поддержки на уровне синтаксиса

Пред- и пост- условия

- Клиент и сервер заключают контракт.
- Клиент обязуется обеспечить выполнение предусловия.
- Сервер обязуется обеспечить выполнения постусловия.

Пример контракта – 1

list.add(o: Object), list: List

- Pre: true
- Post: list.size() == @old list.size() + 1

Пример контракта - 2

list.get(i: int), list: List

- Pre: 0 <= i < list.size()
- Post: true

Пример контракта - 3

s: Socket, new Socket(host, port)

- Pre: 0 <= port <= 2(16), valid_host_name(host), host_exists(host), host_can_accept_connection_on_port(host, port)
- Post: s.isConnected() == true

Предусловие

Предусловие зависит от

- значений аргументов
- состояния объекта
- среды (файловой системы, сети и т.д.)

Что будет, если не выполняется предусловие?

Возможные Exception

- NullPointerException – значения аргументов
- AlreadyConnectedException – состояние объекта
- FileNotFoundException – среда

Проверка постусловий

- Проверяются модульными тестами (unit tests)
- Тестируют отдельные небольшие модули (классы, методы)
- Не используют окружение, базы данных итп
- Выполняются быстро
- Позволяют находить ошибки на ранних стадиях

Пример теста

```
@Test
public void putObjectInMap() {
Map<String, String> map = new HashMap<String, String>();
    map.put("abc", "xyz");
    Assert.assertEquals(map.size(), 1);
    Assert.assertEquals(map.get("abc"), "xyz");
    Assert.assertEquals(map.get("edf"), null);
}
```

Ошибка прохождения теста

```
java.lang.AssertionError:
Expected :null
Actual   :x
        at org.junit.Assert.fail(Assert.java:93)
        at org.junit.Assert.failNotEquals(Assert.java:647)
```

Паттерны проектирования

Лекция 1

Введение

- Повторное использование
- Редко встречаются совершенно новые задачи
- Ценность опыта

Что такое паттерн проектирования?

- Любой паттерн описывает задачу, которая снова и снова возникает в нашей работе, а также принцип ее решения, причем таким образом, что это решение можно потом использовать миллион раз, ничего не изобретая заново

Кристофер Александер

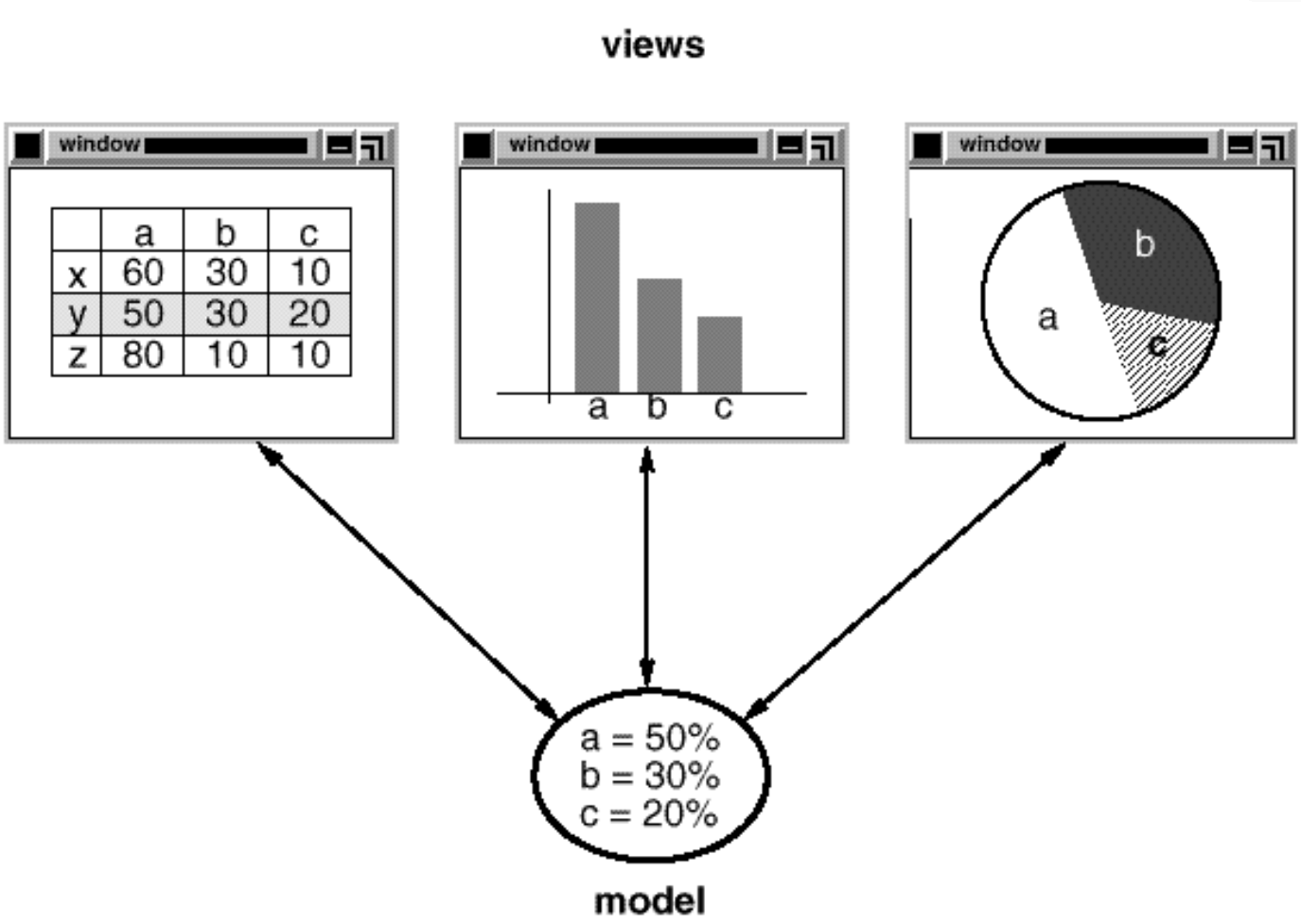
Из чего состоит паттерн

- Имя
- Задача
- Решение
- Результаты

Паттерн

- Паттерн проектирования именует, абстрагирует и идентифицирует ключевые аспекты структуры общего решения, которые и позволяют применить его для создания повторно используемого дизайна.

- Зависит от языка программирования


views
window
window
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
a b c
b
a
c
a = 50%
b = 30%
c = 20%
model

MVC

Model-View-Controller

- Модель (Model) – данные приложения и методы работы с ними
- Вид (View) отвечает за отображение информации
- Контроллер (Controller) –обеспечивает связь между контроллером и моделью

MVC: Архитектурные решения и паттерны

Подписка/оповещение

- **Observer** (наблюдатель) : Определяет между объектами зависимость типа один-ко-многим, так что при изменении состоянии одного объекта все зависящие от него получают извещение и автоматически обновляются.

Представления (View) могут быть вложенными

- **Composite** (компоновщик): Группирует объекты в древовидные структуры для представления иерархий типа «часть-целое». Позволяет клиентам работать с единичными объектами так же, как с группами объектов.

Обработка действий пользователя может меняться независимо от вида

- **Strategy** (стратегия): Определяет семейство алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. Можно менять алгоритм независимо от клиента, который им пользуется.

Классификация паттернов

		Цель		
		Порождающие паттерны	Структурные паттерны	Паттерны поведения
Уровень	Класс	Фабричный метод	Адаптер (класса)	Интерпретатор Шаблонный метод
	Объект	Абстрактная фабрика Одиночка Прототип Строитель	Адаптер (объекта) Декоратор Заместитель Компоновщик Мост Приспособленец Фасад	Итератор Команда Наблюдатель Посетитель Посредник Состояние Стратегия Хранитель Цепочка обязанностей

Зачем нужны паттерны?

- Использование удачных решений
- Коммуникация
- Возможность выбора из альтернатив
- Документирование и поддержка

Patterns vs frameworks

- Паттерны более абстрактны
- Паттерны «меньше» чем frameworks
- Паттерны менее специализированы

Проектирование с учетом будущих изменений

- Явное задание класса при создании объекта

  Паттерны: абстрактная фабрика, прототип, фабричный метод

- Расширение функциональности за счет подклассов

  Паттерны: компоновщик, декоратор

- Зависимость от алгоритмов

  Паттерны: итератор, стратегия, шаблонный метод

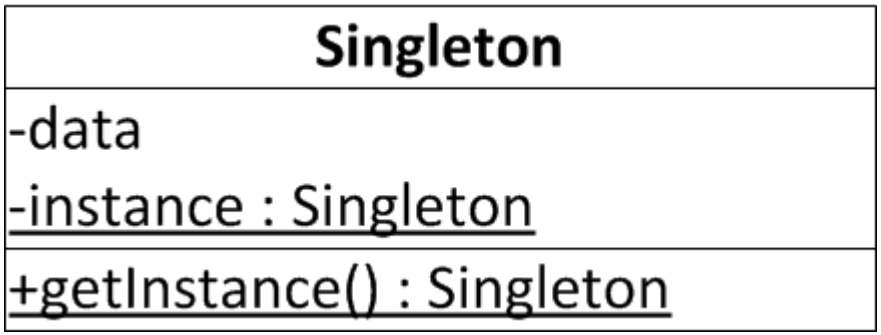
Порождающие паттерны

- Порождающие паттерны - абстрагируют процесс создания объектов

- Паттерн, порождающие классы
  - Factory method (Фабричный метод)
- Паттерн, порождающий объекты
  - Singleton (одиночка)
  - Factory
  - Abstract Factory (Абстрактная фабрика)
  - Builder (Строитель)
  - Prototype (Прототип)

Singleton (Одиночка)

- Описание
  - Singleton (одиночка) - паттерн, порождающий объекты.
- Назначение
  - Гарантирует, что у класса один экземпляр
  - Обеспечивает глобальную точку доступа
- Примеры: логгирование, доступ к оборудованию, фабрики



Результаты

- Контролирует доступ к собственному экземпляру
- Уменьшение числа имен
- Имеет возможность расширяем до нескольких экземпляров

Реализация

```
class Singleton   {
    private static Singleton instance;
    private Singleton()  {
        ...
    }

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

        return instance;
      }
    ...
     public void doSomething() {
       ...
      }
}
```

Ранняя инициализация

```
//Early instantiation using implementation with static field.
class NonLazySingleton  {
private static Singleton instance = new Singleton();

private Singleton() {
        System.out.println("Singleton(): Initializing Instance");
}

public static Singleton getInstance()  {
    return instance;
}

public void doSomething()   {
     System.out.println("doSomething(): Singleton does something!");
}
```

Двойная синхронизация

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

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

Двойная синхронизация – исправленная версия

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

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

Сериализация

```
public class Singleton implements Serializable {
...

    // This method is called immediately after an object of this
           class is deserialized.
    // This method returns the singleton instance.
    protected Object readResolve() {
           return getInstance();
    }
}
```

Factory (фабрика)

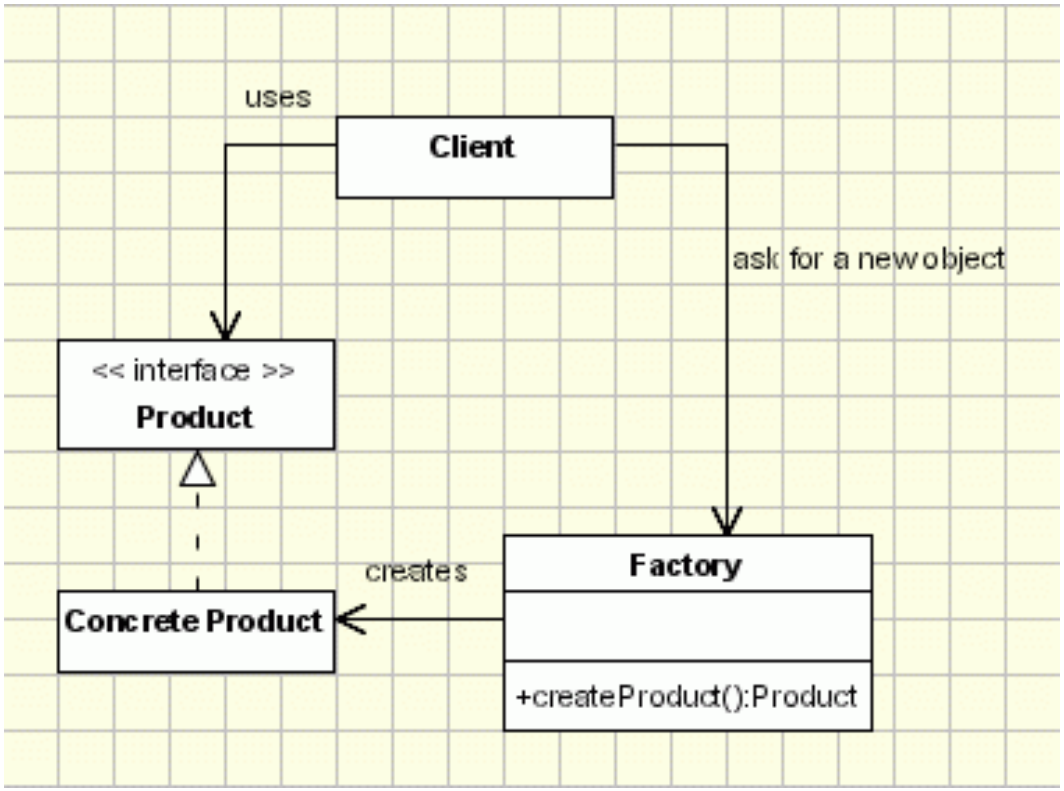
Назначение

- Предоставляет интерфейс для создания объектов без явного указания их классов

Мотивация

- Отсутствие необходимости изменять клиентский код при изменении\добавлении новых объектов

Диаграмма


uses
Client
ask for a new object
<< interface >>
Product
creates
Factory
Concrete Product
+createProduct():Product

Способы реализации

- «Примитивная» реализация
- С помощью reflection
- С помощью factory method

Factory method (фабричный метод)

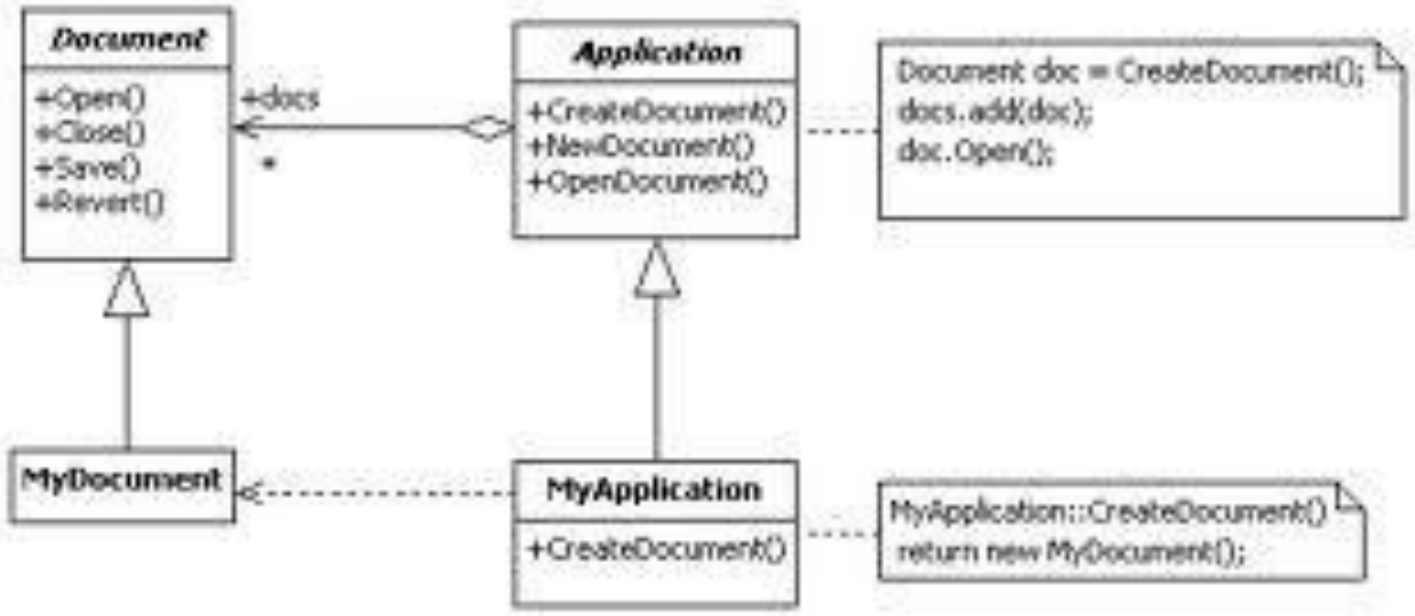
Описание

- Фабричный метод – паттерн, порождающий классы

Назначение

- Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать.

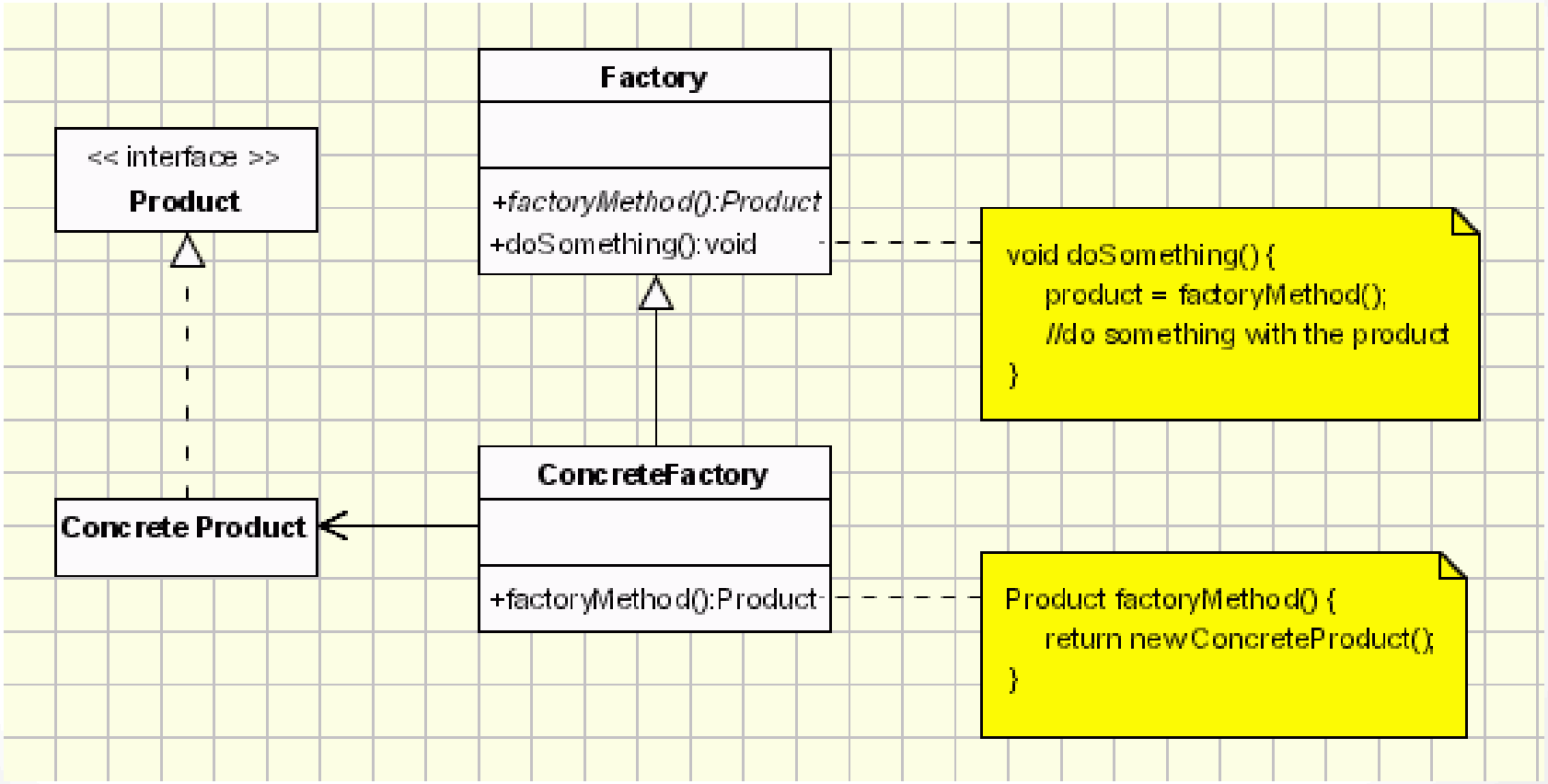
Пример


Document
+Open()
+Close()
+Save()
+Revert()
+docs
*
Application
+CreateDocument()
+NewDocument()
+OpenDocument()
Document doc = CreateDocument();
docs.add(doc);
doc.Open();
MyDocument
MyApplication
+CreateDocument()
MyApplication::CreateDocument()
return new MyDocument();

Применимость

- Для класса заранее неизвестно, объекты каких классов ему нужно создавать;
- класс спроектирован так, чтобы объекты, которые он создает, специфицировались подклассами;
- класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете локализовать знание о том, какой класс принимает эти обязанности на себя.

Диаграмма


Factory
+factoryMethod():Product
+doSomething():void
<< interface >>
Product
Concrete Product
ConcreteFactory
+factoryMethod():Product
void doSomething() {
product = factoryMethod();
//do something with the product
}
Product factoryMethod() {
return new ConcreteProduct();
}

Результаты

- Код приложения изолирован от знания конкретных классов
- Недостаток: необходимость создания производных классов
- Обеспечение точек расширения

Реализация

- Требуется общий класс Product
- Абстрактный и неабстрактный фабричный метод
- Параметризованный фабричный метод

Abstract Factory

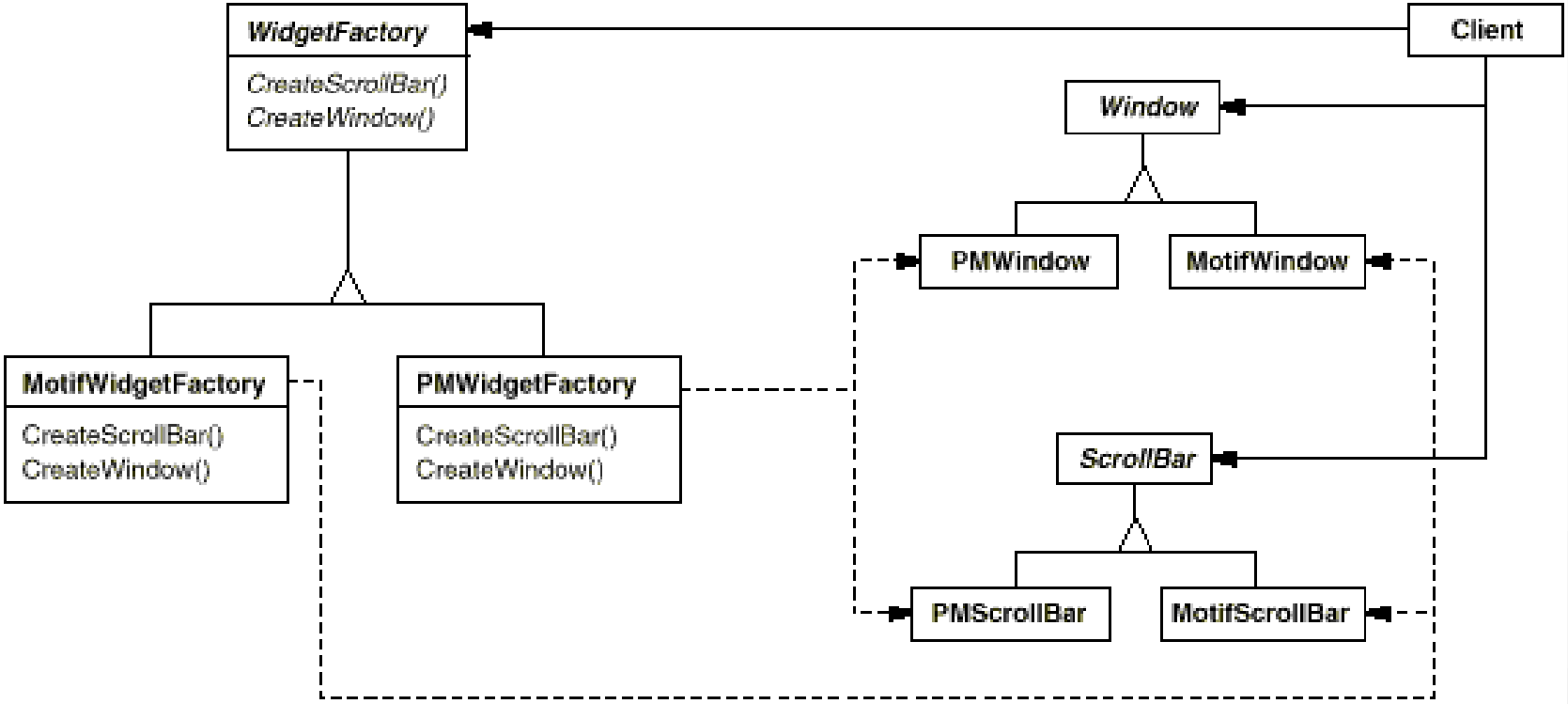
Описание

- Абстрактная фабрика (Abstract Factory) – паттерн, порождающий объекты

Назначение

- Абстрактная фабрика предоставляет интерфейс для создания семейства связанных объектов без явного указания

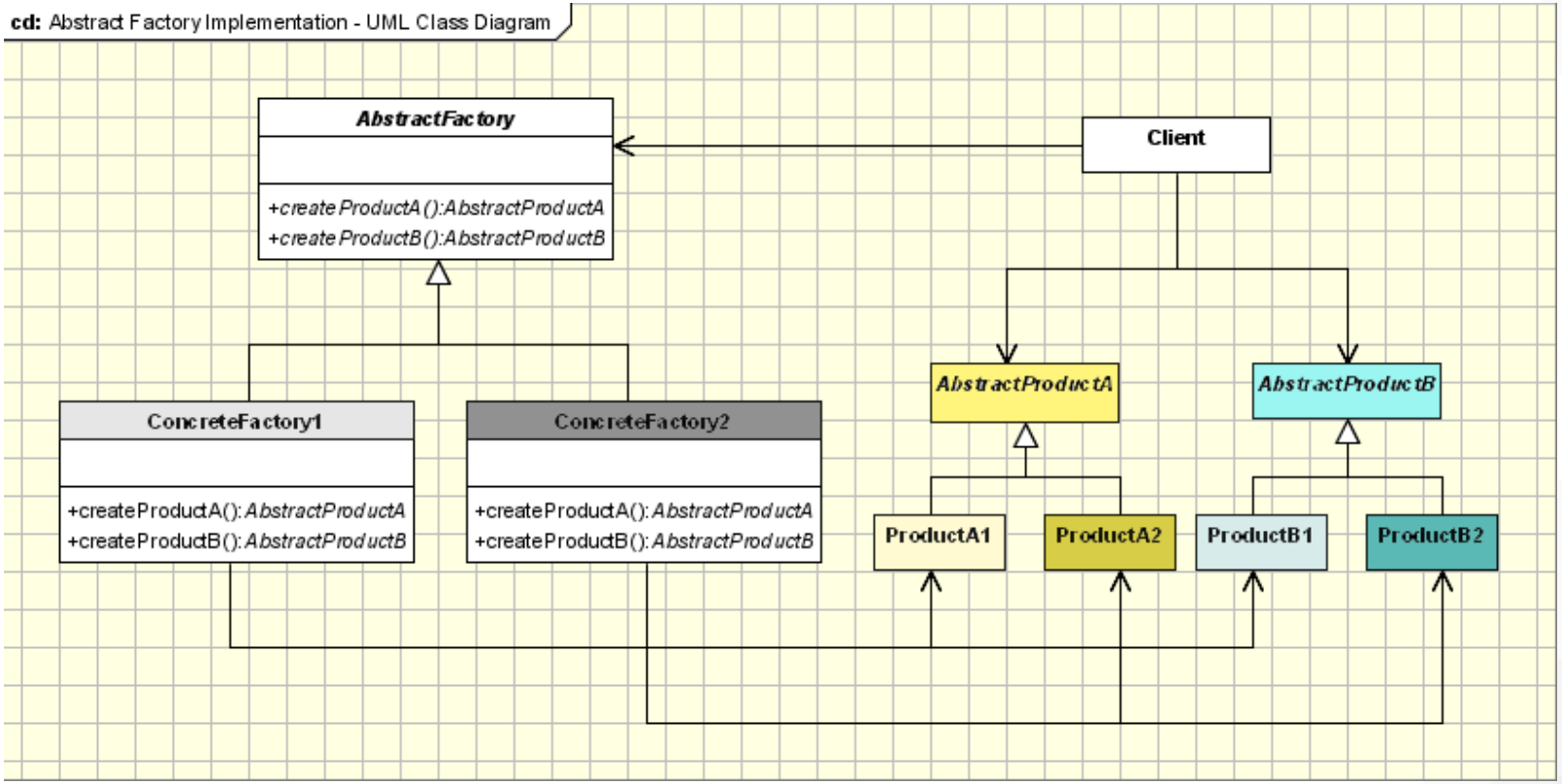
Пример


WidgetFactory
CreateScrollBar()
CreateWindow()
Client
Window
PMWindow
MotifWindow
MotifWidgetFactory
CreateScrollBar()
CreateWindow()
PMWidgetFactory
CreateScrollBar()
CreateWindow()
ScrollBar
PMScrollBar
MotifScrollBar

Применимость

- Система должна быть независима от того, как создаются объекты
- входящие в семейство взаимосвязанные объекты должны использоваться вместе
- система должна конфигурироваться одним из семейств составляющих ее объектов;
- вы хотите предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Диаграмма


cd: Abstract Factory Implementation - UML Class Diagram
AbstractFactory
+createProductA():AbstractProductA
+createProductB():AbstractProductB
Client
AbstractProductA
AbstractProductB
ConcreteFactory1
+createProductA():AbstractProductA
+createProductB():AbstractProductB
ConcreteFactory2
+createProductA():AbstractProductA
+createProductB():AbstractProductB
ProductA1
ProductA2
ProductB1
ProductB2

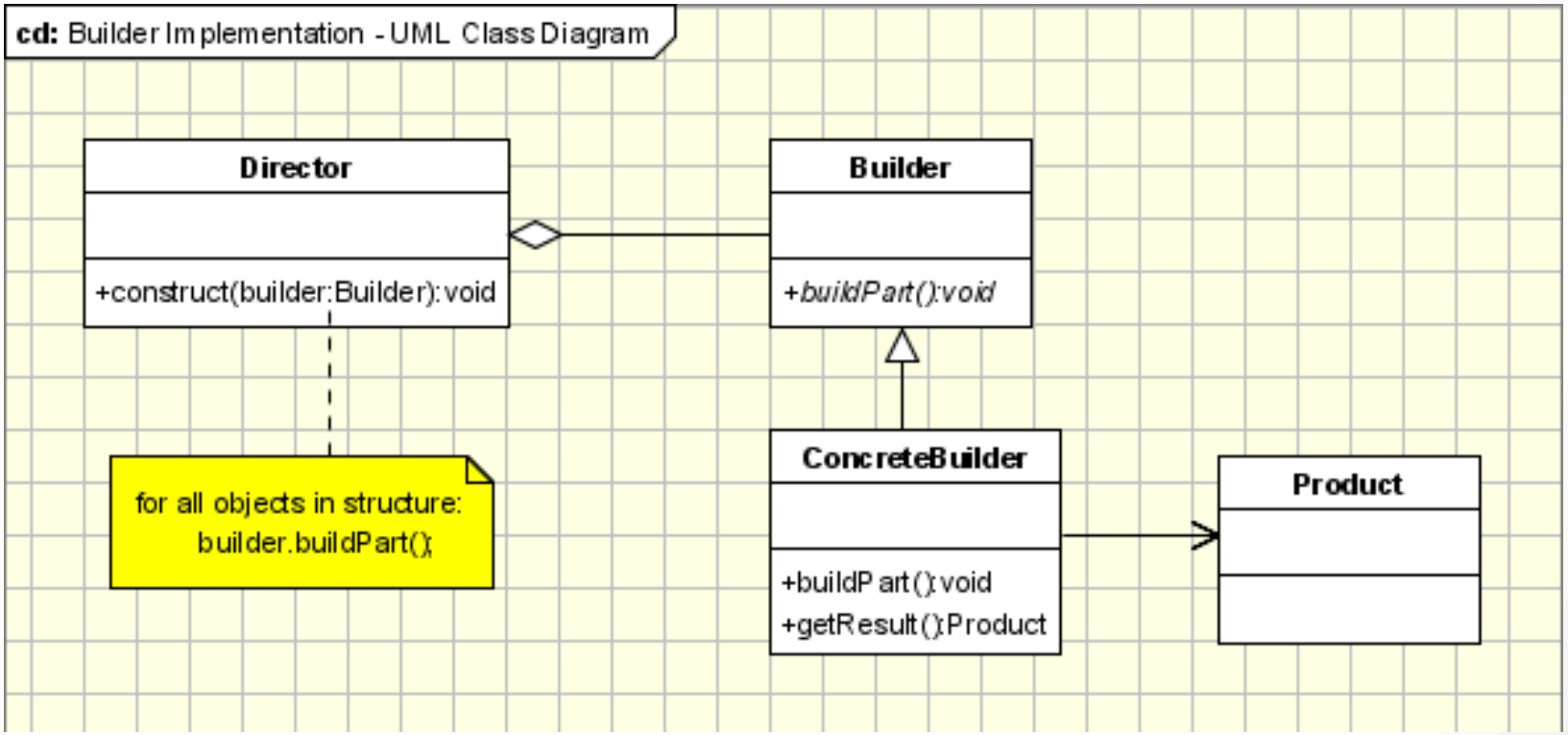
Результаты

- Код приложения изолирован от классов
- Легко заменить семейство продуктов целиком
- Гарантирует правильное сочетание продуктов
- Трудоемко добавлять новый продукт

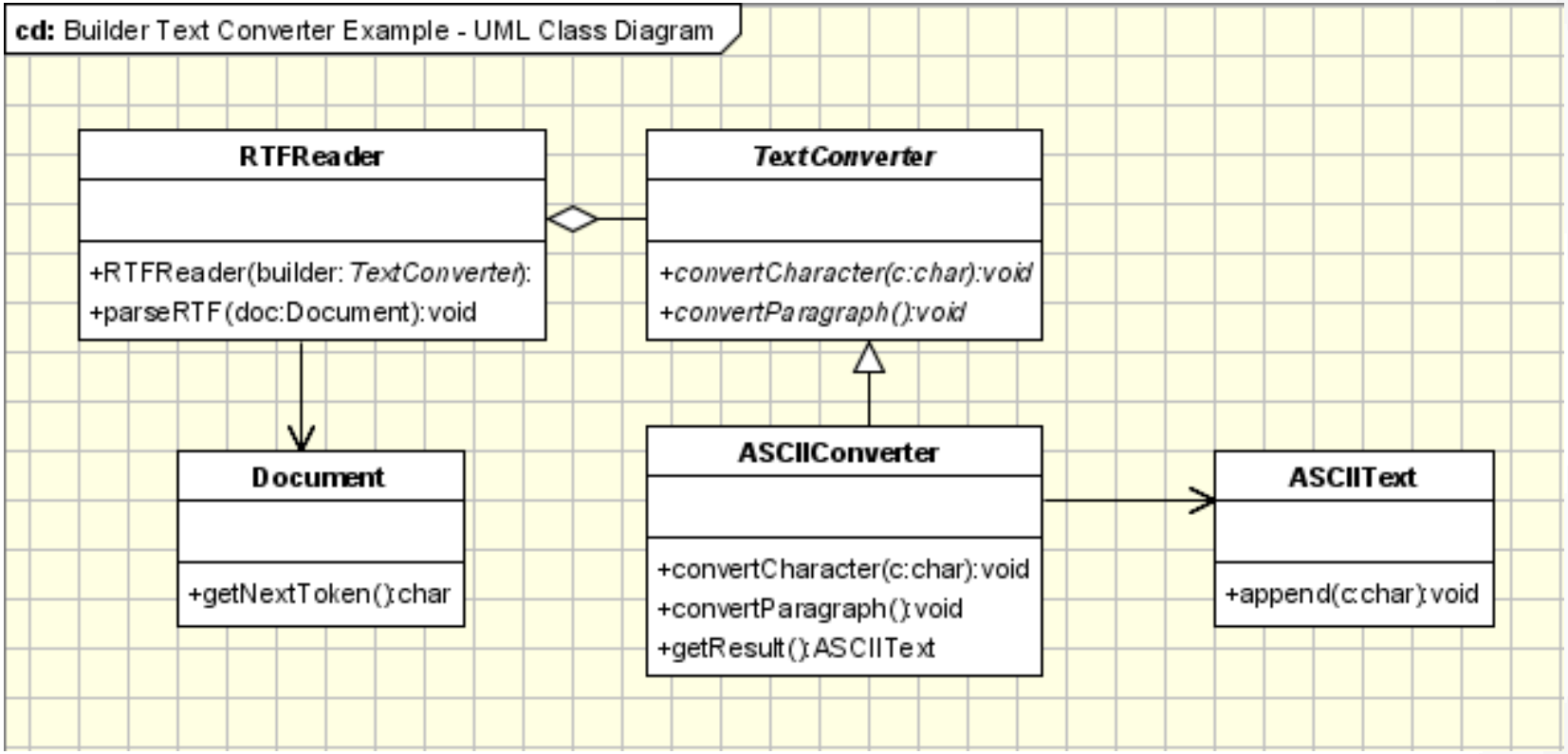
Builder (строитель)

- Описание
  - Builder (строитель) – паттерн, порождающий объекты
- Назначение
  - Отделяет конструирование сложного объекта от его представления
  - Обеспечивает гибкое конструирование сложного объекта

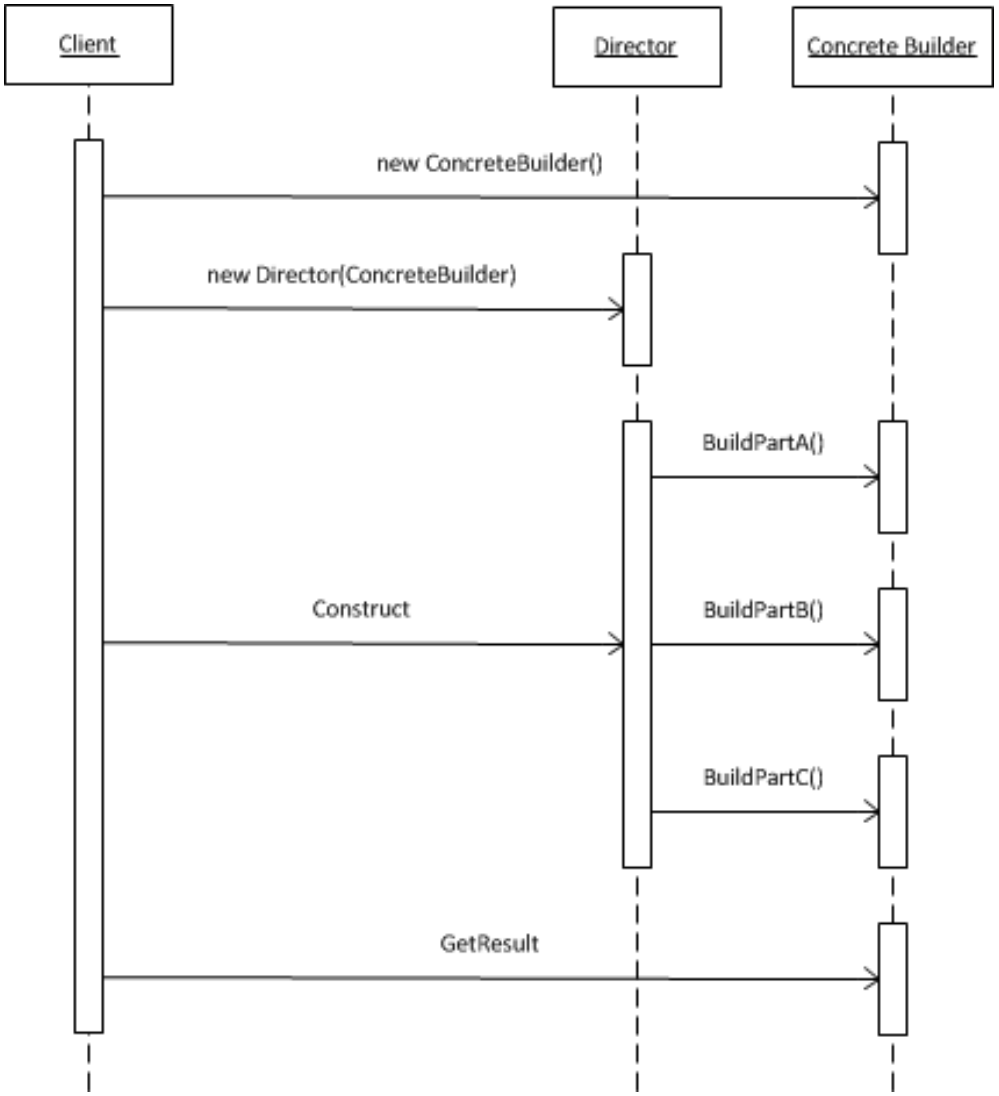
Диаграмма


cd: Builder Implementation - UML Class Diagram
Director
+construct(builder:Builder):void
Builder
+buildPart():void
for all objects in structure:
builder.buildPart();
ConcreteBuilder
+buildPart():void
+getResult():Product
Product

Пример


cd: Builder Text Converter Example - UML Class Diagram
RTFReader
+RTFReader(builder: TextConverter):
+parseRTF(doc:Document):void
TextConverter
+convertCharacter(c:char):void
+convertParagraph():void
Document
+getNextToken():char
ASCIIConverter
+convertCharacter(c:char):void
+convertParagraph():void
+getResult():ASCIIText
ASCIIText
+append(c:char):void

Взаимодействие



Результаты

- Инкапсулирует конструирование и представление объекта
- Более тонкий подход к конструированию

Реализация

- Часто объект Builder содержит экземпляр продукта
- Нет абстрактного класса для продукта
- Методы Builder могут возвращать самого себя, чтобы делать цепочки вызовов

```
ImmutableMap<String, Integer> WORD_TO_INT =
        new ImmutableMap. Builder  <String, Integer>() .
        .put("one", 1)
        .put("two", 2)
        .put("three", 3)
         .build();
```

Книги и ресурсы

- **Приемы объектно-ориентированного проектирования. Паттерны проектирования (***Elements of Reusable Object-Oriented Software* **)-** Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес, Питер, 2007
- Паттерны проектирования (Head first design patterns) - Бейтс Б., Сьерра К., Фримен Э., Фримен Э., Питер, 2013
- http://www.oodesign.com