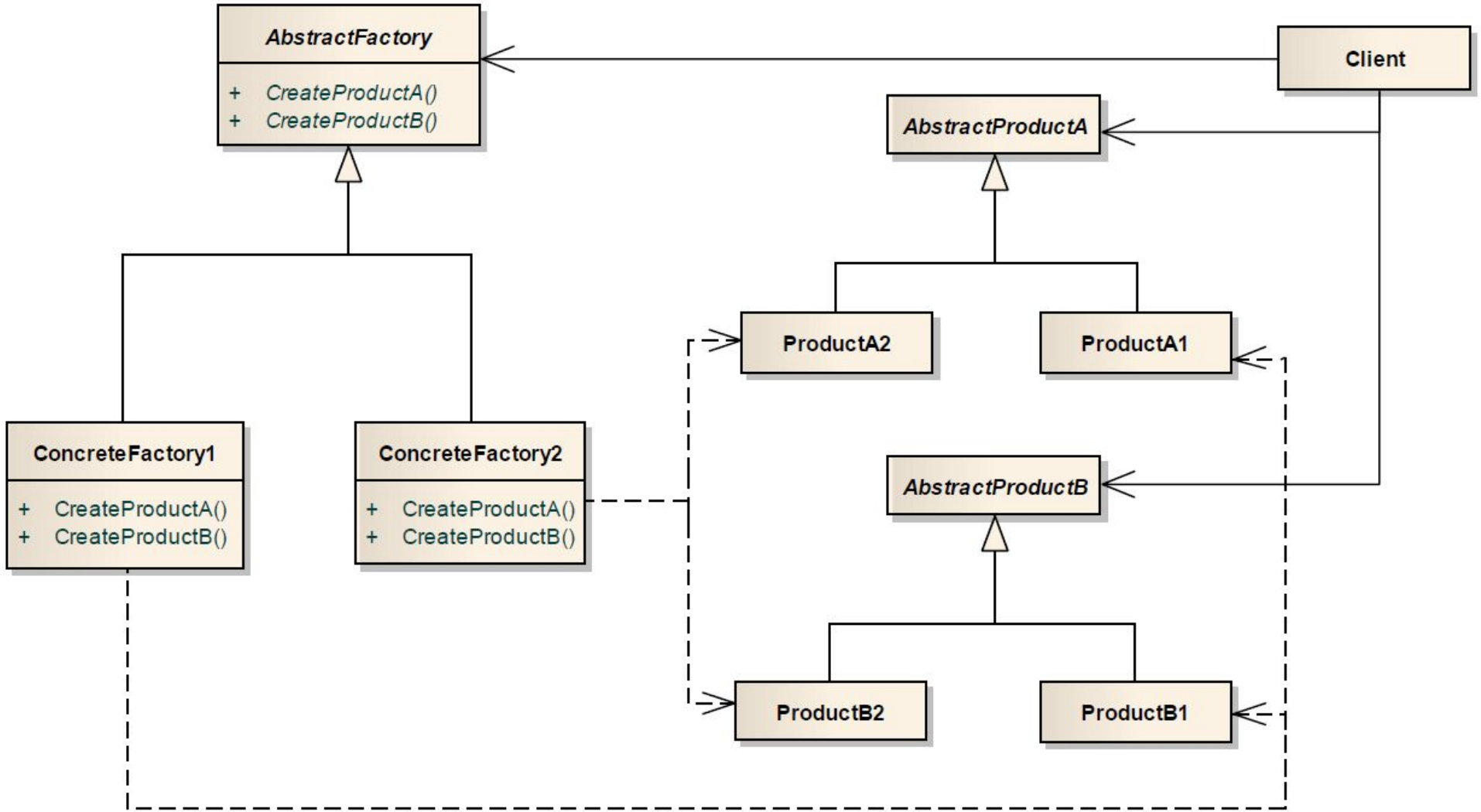
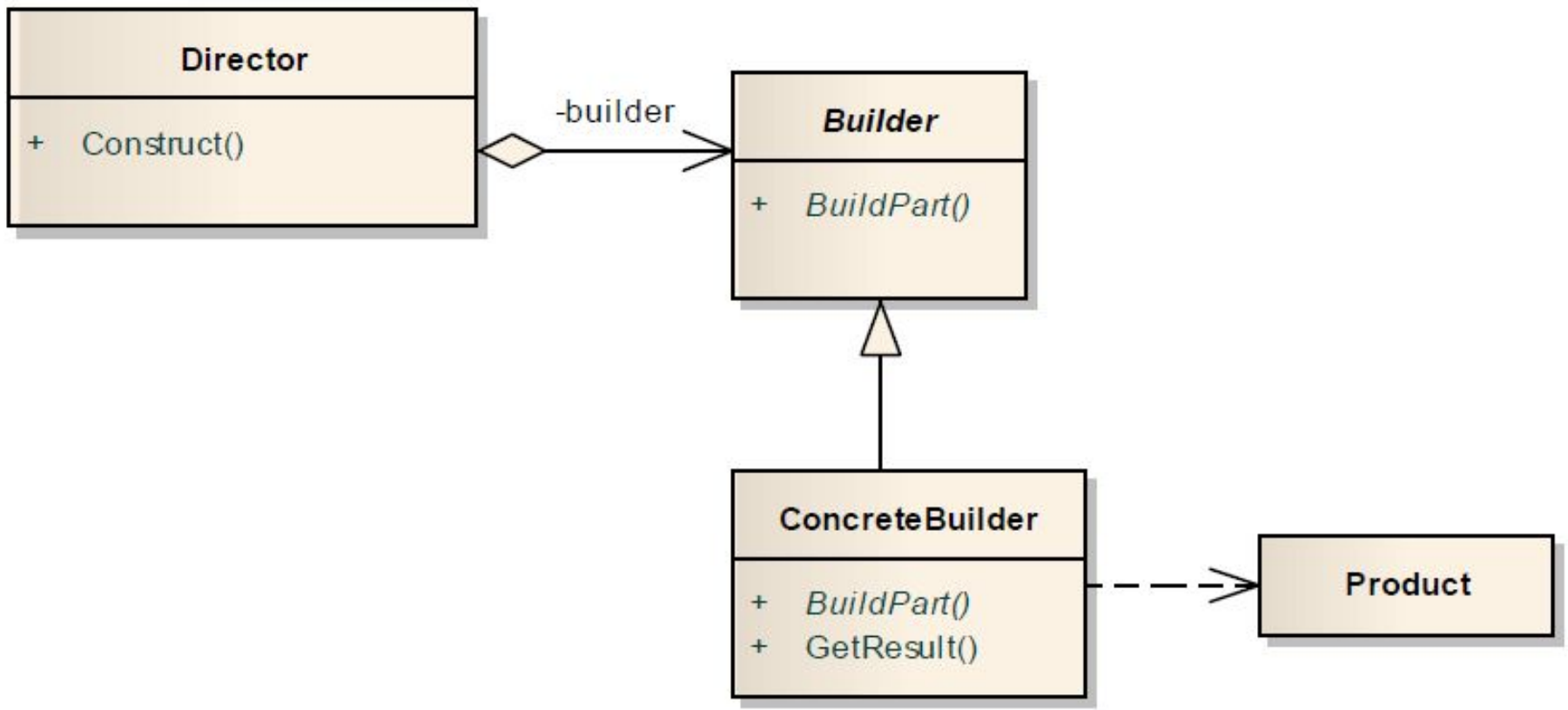
Паттерн фабричный метод (шаблон)

Идрисова И. А.

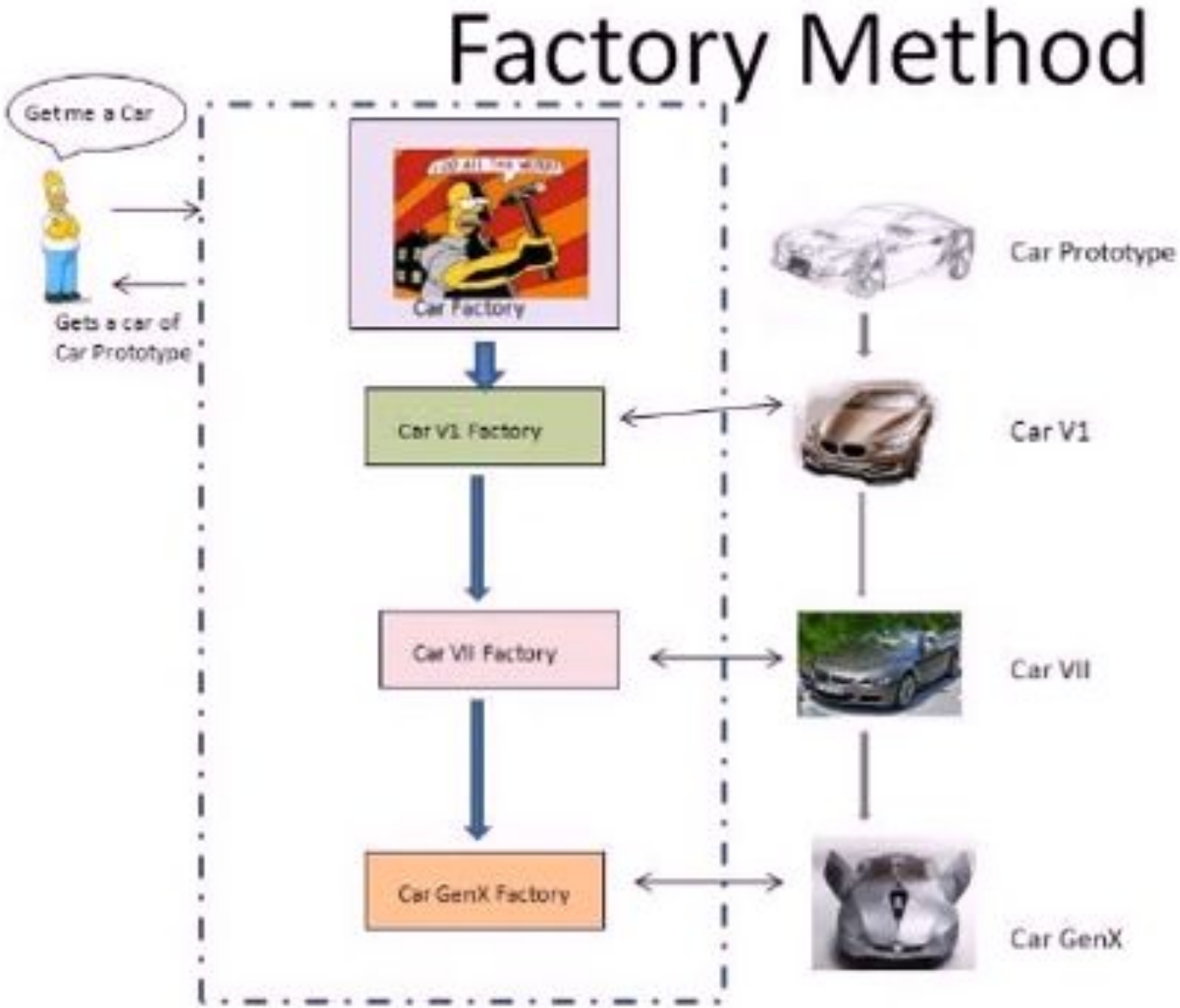
... Абстрактная фабрика



… Builder


Director
+ Construct()
-builder
Builder
+ BuildPart()
ConcreteBuilder
+ BuildPart()
+ GetResult()
Product

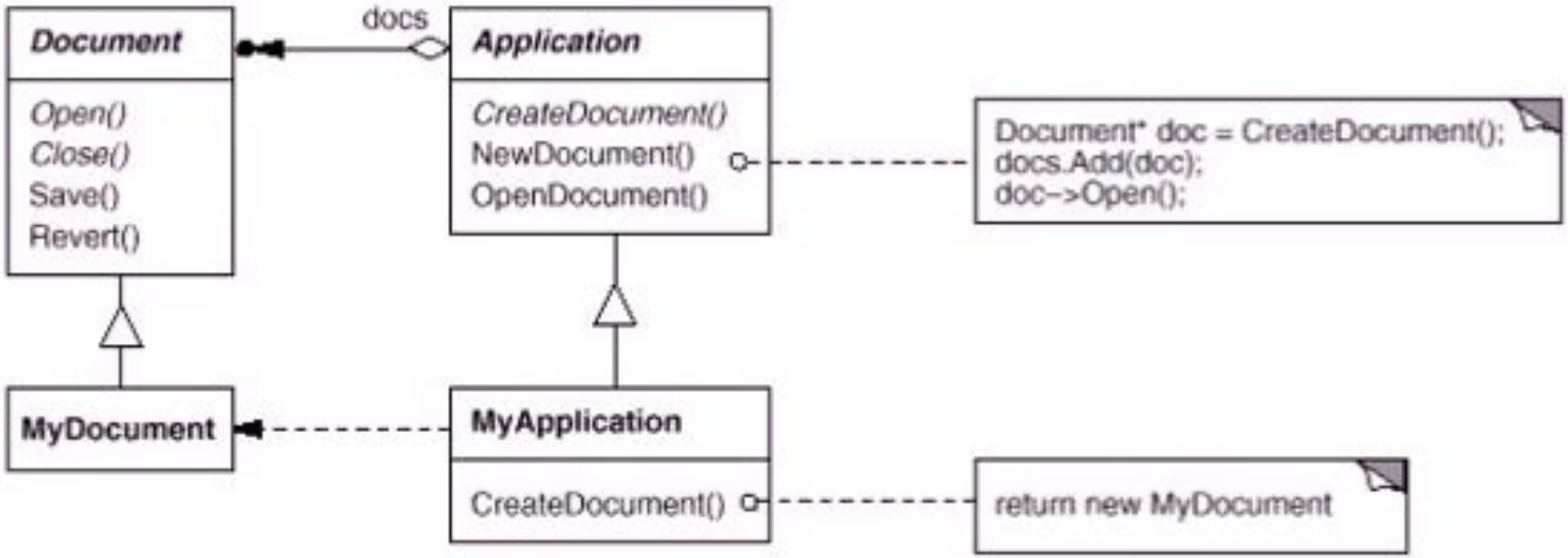
Зачем


Factory Method
Get me a Car
Gets a car of
Car Prototype
Car Factory
Car V1 Factory
Car VII Factory
Car GenX Factory
Car Prototype
Car V1
Car VII
Car GenX

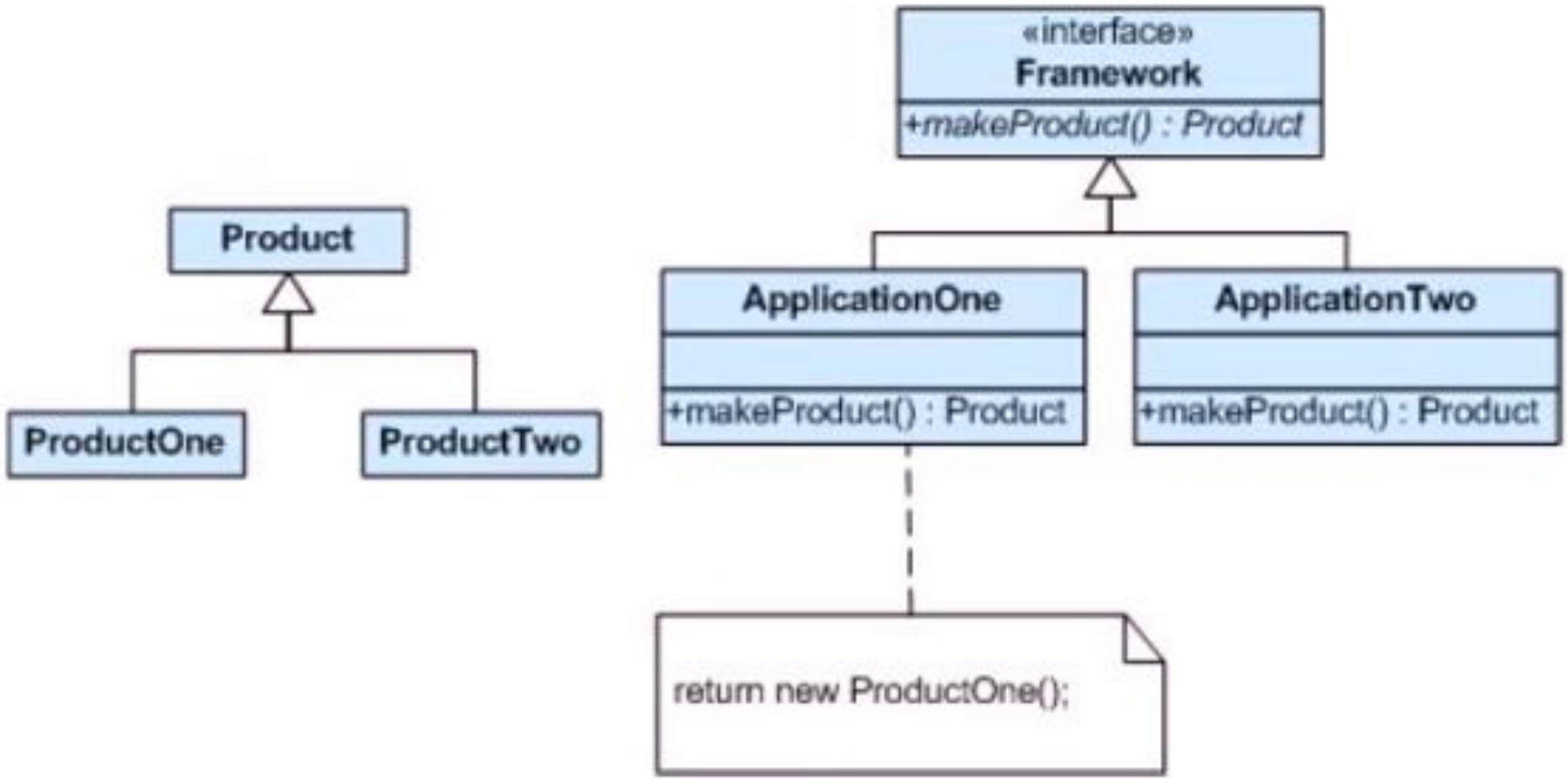
Описание паттернов проектирования

1. Название и классификация
2. Назначение
3. Псевдоним
4. Мотивация
5. Применимость
6. Структура
7. Участники
8. Отношения
9. Результаты
10. Реализация
11. Пример кода
12. Известные применения
13. Родственные паттерны

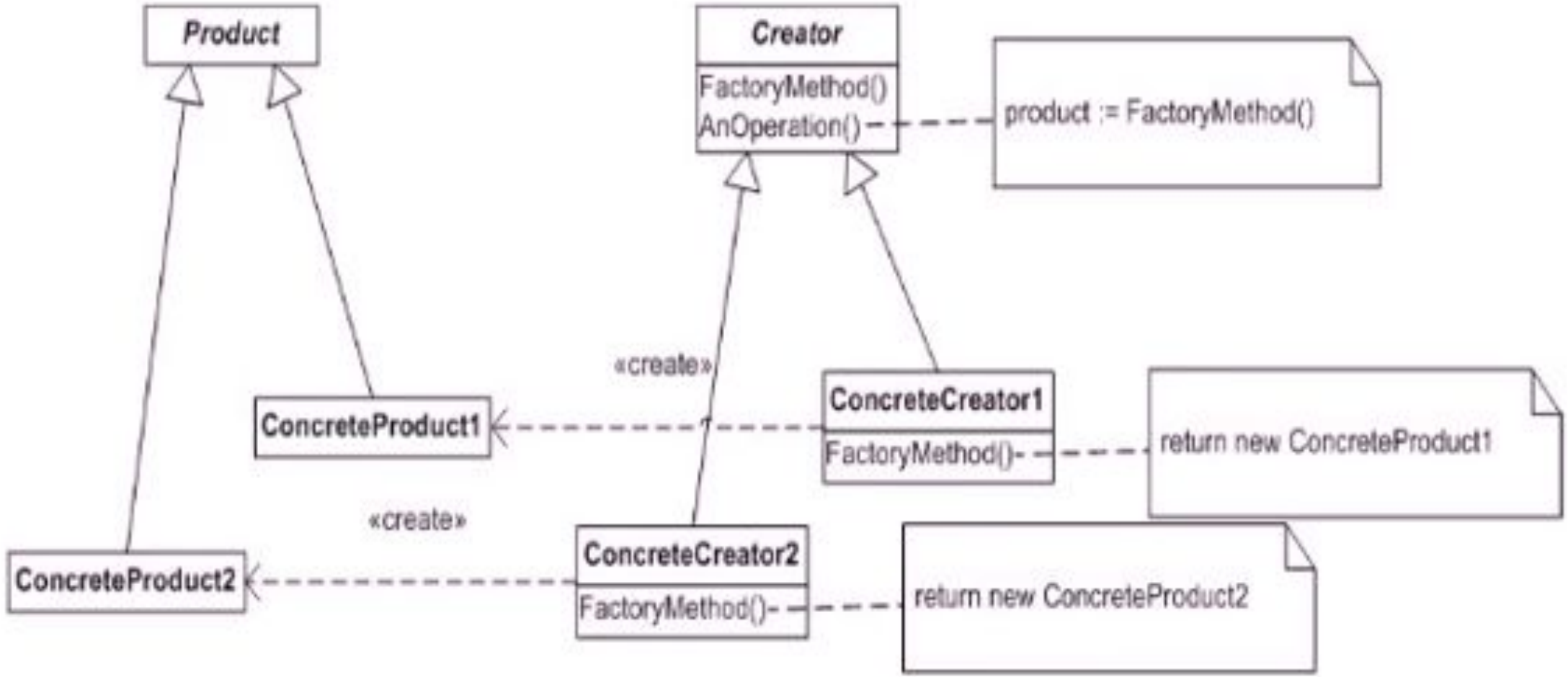
Мотивация


Document
Open()
Close()
Save()
Revert()
docs
Application
CreateDocument()
NewDocument()
OpenDocument()
Document* doc = CreateDocument();
docs.Add(doc);
doc->Open();
MyDocument
MyApplication
CreateDocument()
return new MyDocument

Мотивация 2


«interface»
Framework
+makeProduct() : Product
Product
ApplicationOne
ApplicationTwo
+makeProduct() : Product
+makeProduct() : Product
ProductOne
ProductTwo
return new ProductOne();

Мотивация 3


Product
Creator
FactoryMethod()
AnOperation()
product := FactoryMethod()
«create»
ConcreteProduct1
ConcreteCreator1
FactoryMethod()
return new ConcreteProduct1
«create»
ConcreteProduct2
ConcreteCreator2
FactoryMethod()
return new ConcreteProduct2

...

...

Применимость

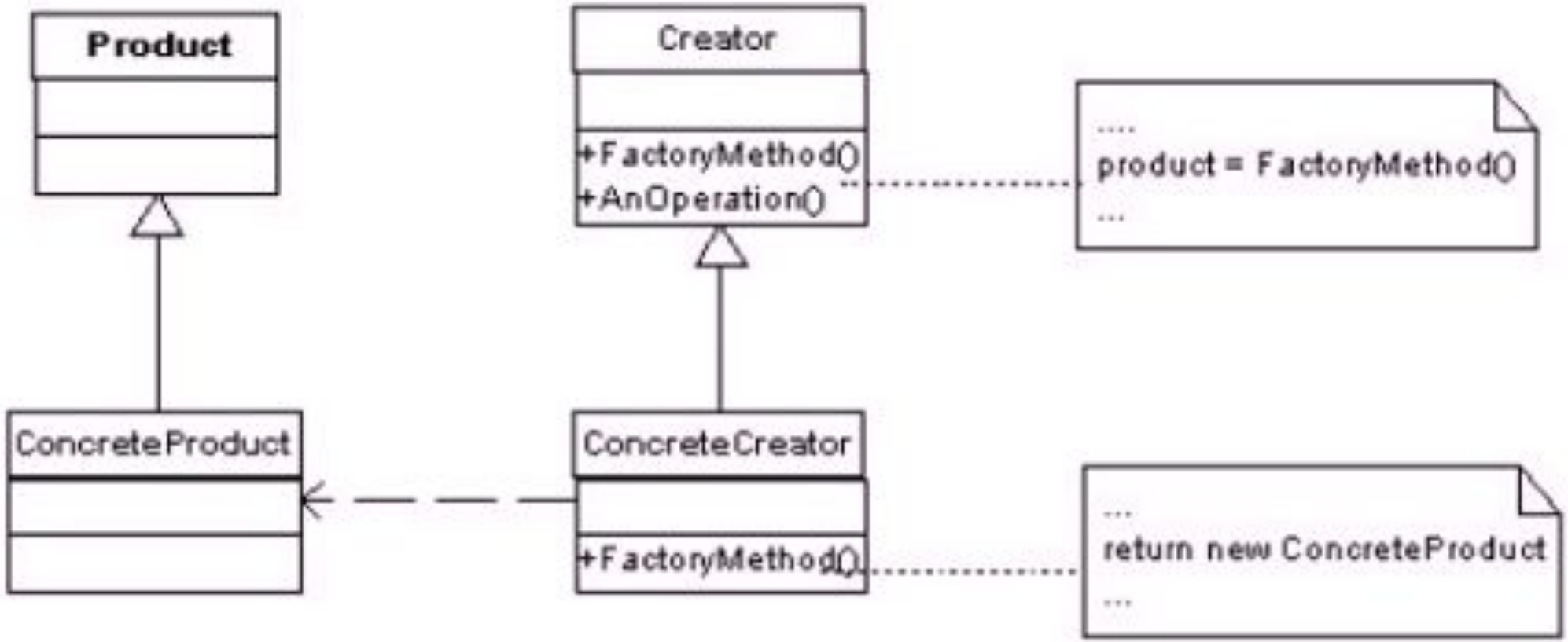
Используйте паттерн фабричный метод, когда:

1. классу **заранее неизвестно, объекты каких классов ему нужно создавать;**

2. класс спроектирован так, чтобы **объекты, которые он создает, специфицировались подклассами**;

3. класс делегирует свои обязанности одному из нескольких вспомогательных подклассов, и вы планируете **локализовать знание о том, какой класс принимает эти обязанности на себя.**

Структура


Product
Creator
+FactoryMethod()
+AnOperation()
....
product = FactoryMethod()
...
ConcreteProduct
ConcreteCreator
+FactoryMethod()
...
return new ConcreteProduct
...

Участники

1. **Product**(Document) - продукт: определяет интерфейс объектов, создаваемых фабричным методом;

2. **ConcreteProduct**(MyDocument) конкретный продукт: реализует интерфейсProduct;

3. **Creator**(Application) = создатель: объявляет фабричный метод, возвращающий объект типаProduct. Creator **может также определять реализацию по умолчанию фабричного метода,** который возвращает объект ConcreteProduct; может вызывать фабричный метод для создания объекта Product.

4. **ConcreteCreator**(MyApplication) = конкретный создатель: замещает фабричный метод, возвращающий объект ConcreteProduct.

Отношения

Класс-создатель **«полагается» на свои подклассы** (конкретные реализации Creator) в определении фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта.

Результаты

1. Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы

2. Можно создавать *(при желании)* расширенные объекты - предоставив конкретным реализациям абстрактного Creator возможно переопределить - в случае необходимости ряд методов (как в примере с файловым диалогом)

3. Соединения параллельных иерархий

Реализация

Выделяют два принципиальных “случая” для реализации:

1. *параметризованные фабричные методы* - данная особенность подразумевает, что вообще говоря - с помощью фабричного метода можно создавать разные виды продуктов в зависимости от переданных параметров

2. различные особенности связанные с конкретным языком реализации

Известные применения.

Фабричные методы где только не встречаются –

большинство библиотек и каркасов так или иначе используют паттерн Фабричный метод

Родственные паттерны

1. **Абстрактная фабрика** часто реализуется с помощью фабричных методов..

2. Прототип в отличии от ФМ не использует наследование от «конкретной фабрики»

- **class Product**{
- **public**:
- **virtual** string getName() = 0;
- **virtual** ~Product() {}
- };

```
class ConcreteProductA: public Product{
public:
  string getName() {return "ConcreteProductA";}
};

class ConcreteProductB: public Product{
public:
  string getName() {return "ConcreteProductB";}
};
```

```
class Creator{
public:
  virtual Product* factoryMethod() = 0;
};

class ConcreteCreatorA: public Creator{
public:
  Product* factoryMethod() {return new ConcreteProductA();}
};

class ConcreteCreatorB: public Creator{
public:
  Product* factoryMethod() {return new ConcreteProductB();}
};
```

```
int main()
{
  static const size_t count = 2;
  ConcreteCreatorA CreatorA;
  ConcreteCreatorB CreatorB;
  // Массив создателей
  Creator*creators[count] = {&CreatorA, &CreatorB};
  // Перебора создателей и создавать продукты

  for(size_t i = 0; i<count; i++){
    Product* product=creators[i]->factoryMethod();
    cout << product->getName() << endl;
    delete product;
  }
  return 0;
}
```

Схожие шаблоны и их отличия

Строитель	**Фабричный метод**	**Абстрактная фабрика**
Создает в несколько шагов один сложный (составной) объект.	Порождает один объект с определенным интерфейсом.	Порождает семейство объектов с определенными интерфейсами.
Интерфейс строителя, реализуемый классами, и класс для управления процессом.	Метод класса, который переопределяется потомками.	Интерфейс, реализуемый классами.
Скрывает процесс создания объекта, порождает требуемую реализацию.	Скрывает реализацию объекта.	Скрывает реализацию семейства объектов.

Лабораторная работа №2 (дедлайн **04.03**)

1. Программа для сферы строительства. Вначале мы захотим построить многоэтажный **панельный дом**. И для этого выбирается соответствующий подрядчик, который возводит каменные дома. Затем нам захочется построить **деревянный дом** и для этого также надо будет выбрать нужного подрядчика.

Подсказки:

a) В качестве абстрактного класса Product здесь выступает класс House. Его две конкретные реализации - PanelHouse и WoodHouse представляют типы домов, которые будут строить подрядчики.

b) В качестве абстрактного класса создателя выступает Developer, определяющий абстрактный метод Create(). Этот метод реализуется в классах-наследниках WoodDeveloper и PanelDeveloper. И если в будущем нам потребуется построить дома какого-то другого типа, например, кирпичные, то мы можем с легкостью создать новый класс кирпичных домов, унаследованный от House, и определить класс соответствующего подрядчика. Таким образом, система получится легко расширяемой.