Паттерны проектирования (Design patterns)

Что такое паттерны (шаблоны) проектирования?

- Эффективные способы решения характерных задач проектирования
- Обобщенное описание решения задачи, которое можно использовать в различных ситуациях
- ОО паттерны проектирования часто показывают отношения и взаимодействия между классами и объектами
  - Алгоритмы не являются паттернами, т.к. решают задачу вычисления, а не программирования

Польза

- Каждый паттерн описывает решение целого класса проблем
- Каждый паттерн имеет известное имя
  - облегчается взаимодействие между разработчиками
  - Правильно сформулированный паттерн проектирования позволяет, отыскав удачное решение, пользоваться им снова и снова
- Шаблоны проектирования не зависят от языка программирования, в отличие от идиом

Порождающие паттерны

Порождающие паттерны проектирования

- Абстрагируют процесс инстанцирования объектов
- Список паттернов
  - Абстрактная фабрика (Abstract Factory)
  - Строитель (Builder)
  - Фабричный метод (Factory method)
  - Прототип (Prototype)
  - Одиночка (Singleton)

Abstract Factory (Абстрактная фабрика)

Назначение паттерна «Абстрактная фабрика»

- Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов

Паттерн «Абстрактная фабрика»

- Шаблон проектирования, позволяющий изменять поведение системы, варьируя создаваемые объекты, при этом сохраняя интерфейсы
- Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов

Реализация паттерна

- Паттерн «Абстрактная фабрика» реализуется созданием абстрактного класса **Factory**, который представляет собой интерфейс для создания абстрактных объектов-продуктов
- На основе данного класса создается один или несколько классов конкретных фабрик, создающих конкретные объекты-продукты

Применение паттерна

- Используйте паттерн, когда
  - Система не должна зависеть от того, как создаются, компонуются и представляются входящие в нее объекты
  - Входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения
  - Система должна конфигурироваться одним из семейств составляющих ее объектов
  - Требуется предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

Структура

Пользуется исключительно интерфейсами, которые объявлены в классах AbstractFactory и AbstractProduct

Объявляет интерфейс для операций, создающих абстрактные объекты-продукты

Объявляет интерфейс для типа объекта-продукта

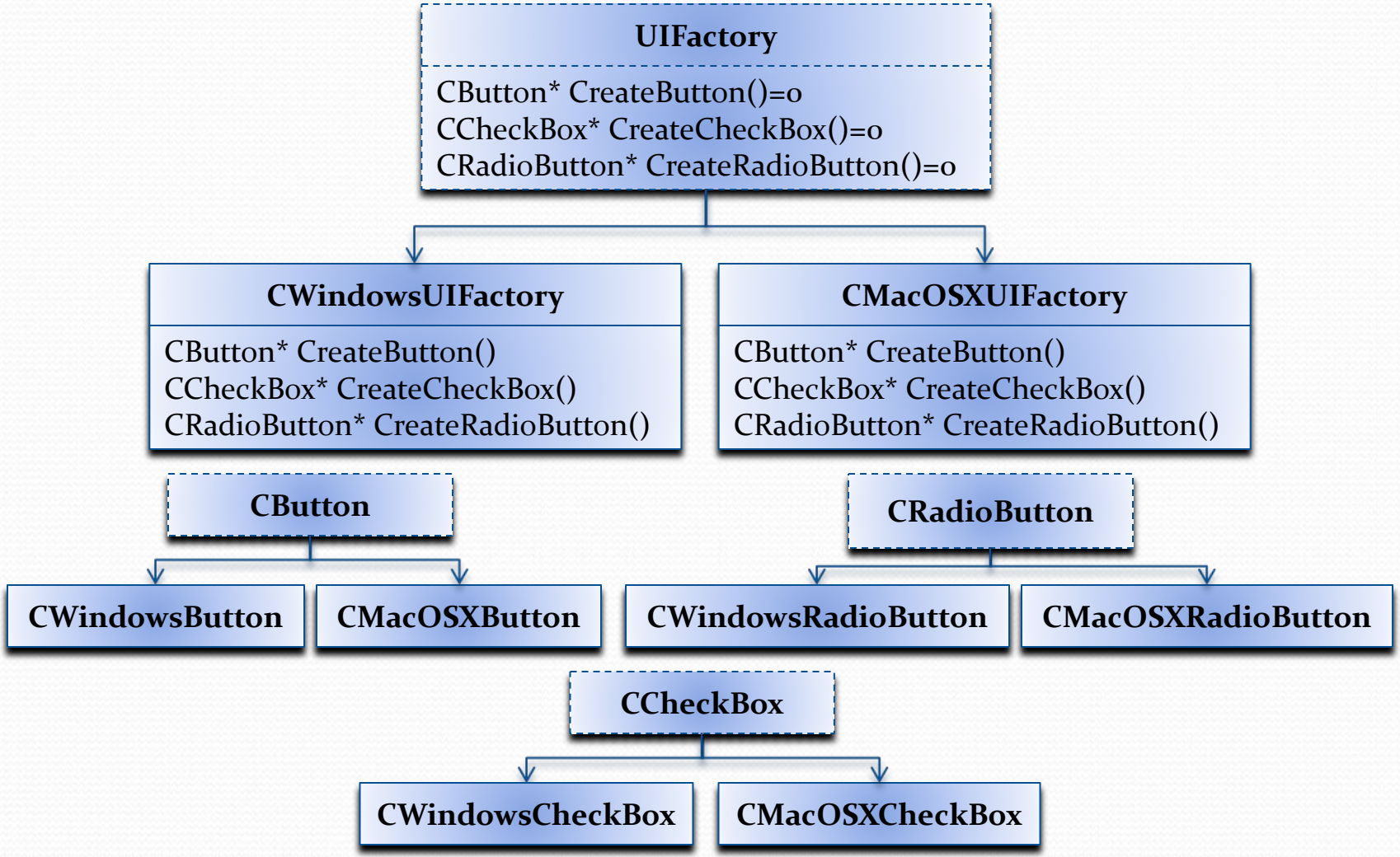
Реализует операции, создающие конкретные объекты-продукты;

Определяет объект-продукт, создаваемый соответствующей конкретной фабрикой

Пример использования

- При разработке приложений с графическим интерфейсом пользователя необходимо создавать различные элементы управления
  - Кнопки, текстовые поля, радио-кнопки, выпадающие списки и т.п.
- Их создание и работа с ними в различных ОС осуществляется по-разному
  - Чтобы приложение можно было легче перенести в другую ОС в нем не должно быть жесткой привязки к типам конкретных классов элементов управления
  - Паттерн «Абстрактная фабрика» облегчает решение данной задачи

Иерархия классов



Абстрактные и конкретные элементы управления

```
class CButton
{
public:
	virtual ~CButton(){}
	//...
};

class CWindowsButton : public CButton
{
	//...
};

class CMacOSXButton : public CButton
{
	//...
};
```

```
class CCheckBox
{
public:
	virtual ~CCheckBox(){}
	// ...
};

class CWindowsCheckBox : public CCheckBox
{
	// ...
};

class CMacOSXCheckBox : public CCheckBox
{
	// ...
};
```

Абстрактная и конкретные фабрики

```
class CUIFactory
{
public:
	virtual CButton* CreateButton()const=0;
	virtual CCheckBox*
CreateCheckBox()const=0;
};

class CWindowsUIFactory : public CUIFactory
{
public:
	virtual CButton* CreateButton()const
	{
		return new CWindowsButton();
	}
	virtual CCheckBox* CreateCheckCBox()const
	{
		return new CWindowsCheckBox();
	}
};
```

```
class CMacOSXUIFactory : public CUIFactory
{
public:
	virtual CButton* CreateButton()const
	{
		return new CMacOSXButton();
	}
	virtual CCheckBox* CreateCheckBox()const
	{
		return new CMacOSXCheckBox();
	}
};
```

```
void BuildUI(CUIFactory const& uiFactory)
{
	CButton * pOKButton = uiFactory.CreateButton();
	CCheckBox * pCheckBox=
uiFactory.CreateCheckBox();
}
int main(int argc, char* argv[])
{
	CWindowsUIFactory uiFactory;
	BuildUI(uiFactory);
	return 0;
}
```

Преимущества использования паттерна «Абстрактная фабрика»

- Изоляция конкретных классов продуктов
  - Фабрика изолирует клиента от деталей реализации классов продуктов
    - Имена изготавливаемых классов известны только конкретной фабрике, в коде клиента они не упоминаются
    - Клиенты манипулируют экземплярами продуктов только через их абстрактные интерфейсы
- Упрощение замены семейств продуктов
  - Приложение может изменить семейство продуктов просто подставив новую конкретную фабрику
- Гарантия сочетаемости продуктов
  - Если продукты спроектированы для совместного использования, важно чтобы в каждый момент времени приложение работало с продуктами единственного семейства

Недостатки паттерна «Абстрактная фабрика»

- Трудность поддержания новых типов продуктов
  - Интерфейс абстрактной фабрики фиксирует набор продуктов, которые можно создать
  - Для поддержки новых продуктов необходимо расширить интерфейс фабрики, внеся изменения в класс AbstractFactory, а также во все его подклассы
- Обход этого ограничения – передача идентификатора типа создаваемого продукта в методы фабрики, создающие продукты
  - Ограничение: создаваемые продукты должны поддерживать общий абстрактный интерфейс
    - Клиент не может различать типы продуктов и делать какие-либо предположения о них

Пример – фабрика, создающая элементы игрового поля

```
enum ItemType
{
	WALL,
	WATER,
	FOREST,
	SAND,
};

class IMapItem
{
public:
	virtual ~IMapItem(){}
	// ...
};
class CWall : public IMapItem {};
class CWater : public IMapItem {};
class CForest : public IMapItem {};
class CSand : public IMapItem {};
```

```
class CMapItemsFactory
{
public:
	IMapItem* CreateItem(ItemType type)const
	{
		switch (type)
		{
		case WALL: return new CWall();
		case WATER: return new CWall();
		case FOREST: return new CForest();
		case SAND: return new CSand();
		default:
			throw new std::invalid_argument
				("Unknown item type");
		}
	}
};
```

Builder (Строитель)

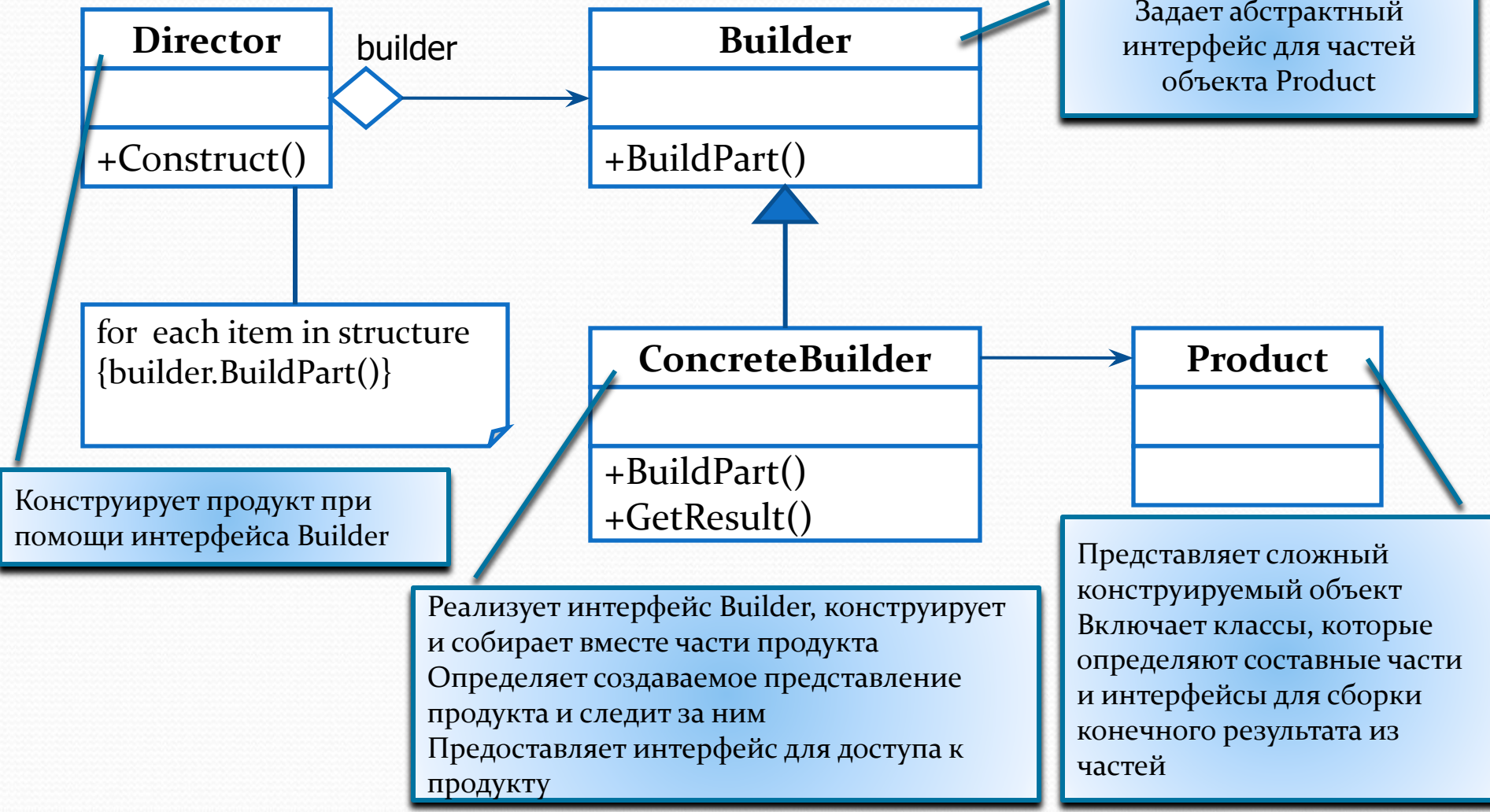
Назначение паттерна «Строитель»

- Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления

Область применения паттерна «Строитель»

- Алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой
- Процесс конструирования должен обеспечивать различные представления конструируемого объекта

Структура


Director
builder
Builder
+Construct()
+BuildPart()
Задает абстрактный интерфейс для частей объекта Product
for each item in structure
{builder.BuildPart()}
ConcreteBuilder
+BuildPart()
+GetResult()
Product
Конструирует продукт при помощи интерфейса Builder
Реализует интерфейс Builder, конструирует и собирает вместе части продукта
Определяет создаваемое представление продукта и следит за ним
Предоставляет интерфейс для доступа к продукту
Представляет сложный конструируемый объект
Включает классы, которые определяют составные части и интерфейсы для сборки конечного результата из частей

Отношения между участниками паттерна

- Клиент создает новый объект «Распорядитель» (Director) и конфигурирует его новым объектом-строителем Builder
- Распорядитель уведомляет строителя о необходимости построения очередной части продукта
- Строитель обрабатывает запросы распорядителя и добавляет новые части к продукту
- Клиент забирает продукт у строителя

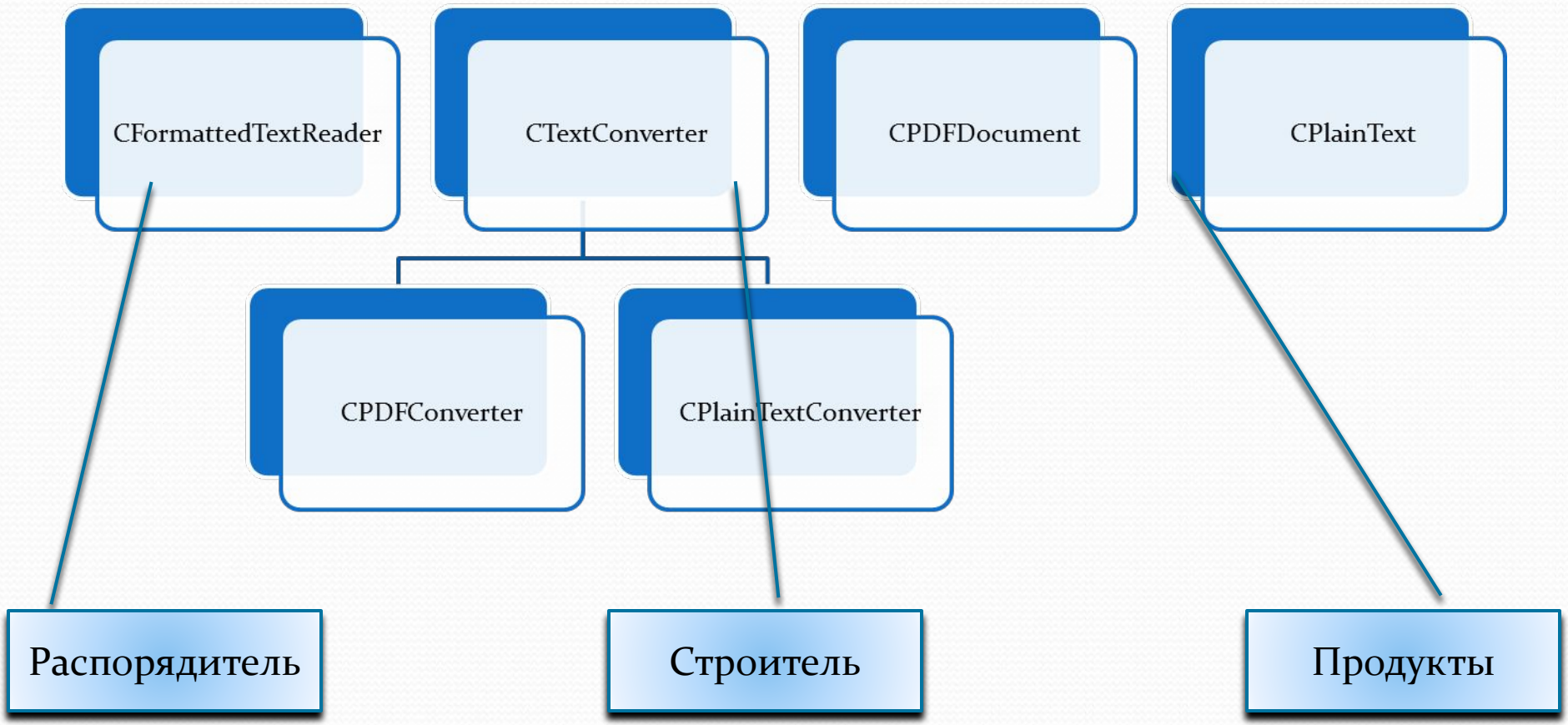
Достоинства паттерна «Строитель»

- Позволяет изменять внутреннее представление продукта
  - Распорядителю предоставляется абстрактный интерфейс Строителя, скрывающего структуру продукта и процесс сборки
    - Для изменения внутреннего представления достаточно определить новую реализацию Строителя
- Изолирует код, реализующий конструирование и представление
  - Клиенту не нужно знать о классах, задающих внутреннюю структуру продукта (в интерфейсе строителя они отсутствуют)
- Дает более тонкий контроль над процессом конструирования
  - Процесс построения продукта происходит не сразу, как в других порождающих паттернах, а шаг за шагом

Пример использования

- В редакторе форматированного текстового документа необходимо реализовать возможность преобразования его в различные форматы
  - Plain text, HTML, RTF, PDF, DOC, DOCX
    - Список можно продолжить
- Задача решается путем введения сущностей
  - Распорядитель - CFormattedTextReader
  - Строитель – CTextConverter
  - Конкретный Строитель – CHtmlConverter, CRTFConverter, CPlainTextConverter, CPDFConverter, …
  - Продукт – CPlainTextDocument, CHtmlDocument, CRTFDocument, CPDFDocument, …

Иерархия классов



Реализация сущности «Строитель»

```
class CTextConverter
{
public:
     virtual void ConvertText
               (std::string const& s){}
     virtual void ConvertFontChange
               (CFont const& fnt){}
     // …
};

class CPlainTextConverter
     : public CTextConverter
{
public:
     void ConvertText(std::string const&
s)
     {
          // …
     }
     std::string const &
GetPlainText()const
     {
          return m_plainText;
     }
private:
```

```
class CPDFConverter
     : public CTextConverter
{
public:
     void ConvertText(std::string const& w)
     {
          // ...
     }
     void ConvertFontChange(CFont const&
fnt)
     {
          // ...
     }

     CPDFDocument const& GetPDFDocument()
const
     {
          return m_pdfDocument;
     }
private:
     CPDFDocument m_pdfDocument;
};
```

Реализация сущности «Распорядитель»

```
class CFormattedTextReader
{
public:
    void Read(CFormattedText const& text, CTextConverter & converter)
    {
        for (size_t index = 0; index < text.GetItems(); ++index)
        {
            CTextItem const & item = text.GetItem(index);
            CTextRangeItem const* pTextRange = NULL;
            CFontChangeCommand const& * pFontCommand = NULL;
            if (pTextRange = dynamic_cast<CTextRangeItem const*>(&item))
            {
                converter.ConvertText(pTextRange->GetText());
            }
            else if (pFontCommand = dynamic_cast<CFontChangeCommand
const*>(&item))
            {
                converter.ConvertFontChange(pFontCommand->GetFont());
            }
            else if (…) { … } …
        }
    }
};
```

Реализация клиента

```
void ConvertToPDF(CFormattedText const& text, std::string const& outputFile)
{
    // создаем строителя
    CPDFConverter converter;

    // создаем распорядителя
    CFormattedTextReader reader;
    // и инициируем процесс построения продукта
    reader.Read(text, converter);

    // получаем конечный продукт
    CPDFDocument const& pdfDoc = converter.GetPDFDocument();
    pdfDoc.SaveToFile(outputFile);
}
```

Factory Method (Фабричный метод)

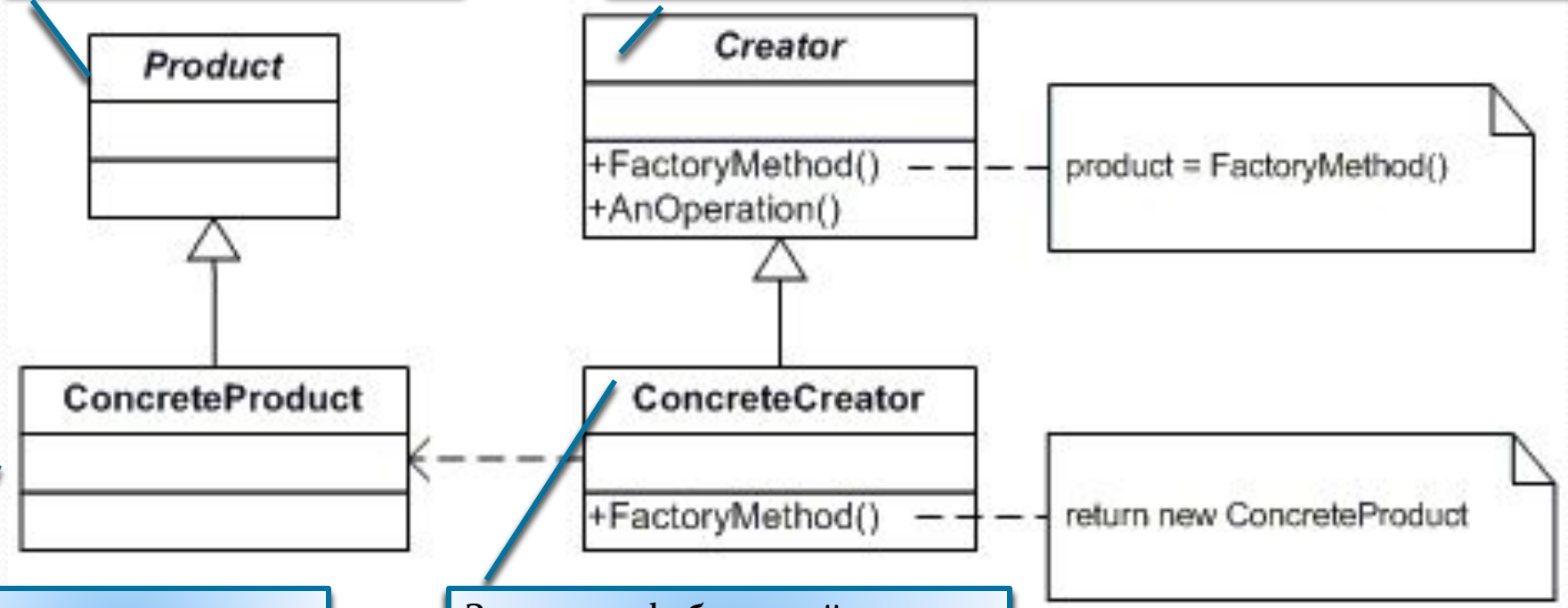
Назначение паттерна «Фабричный метод»

- Определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать
  - Фабричный метод позволяет классу делегировать инстанцирование подклассам
- Альтернативное название паттерна – «Виртуальный конструктор»

Структура

Определяет интерфейс объектов, создаваемых фабричным методом

- Объявляет фабричный метод, возвращающий объект типа Product, может также определять реализацию по умолчанию данного фабричного метода
- Может вызывать фабричный метод для создания объекта Product.



Реализует интерфейс Product

Замещает фабричный метод, возвращающий объект ConcreteProduct

Отношения между участниками паттерна

- **Создатель** «полагается» на свои **подклассы** в определении фабричного метода, который будет возвращать экземпляр подходящего конкретного продукта

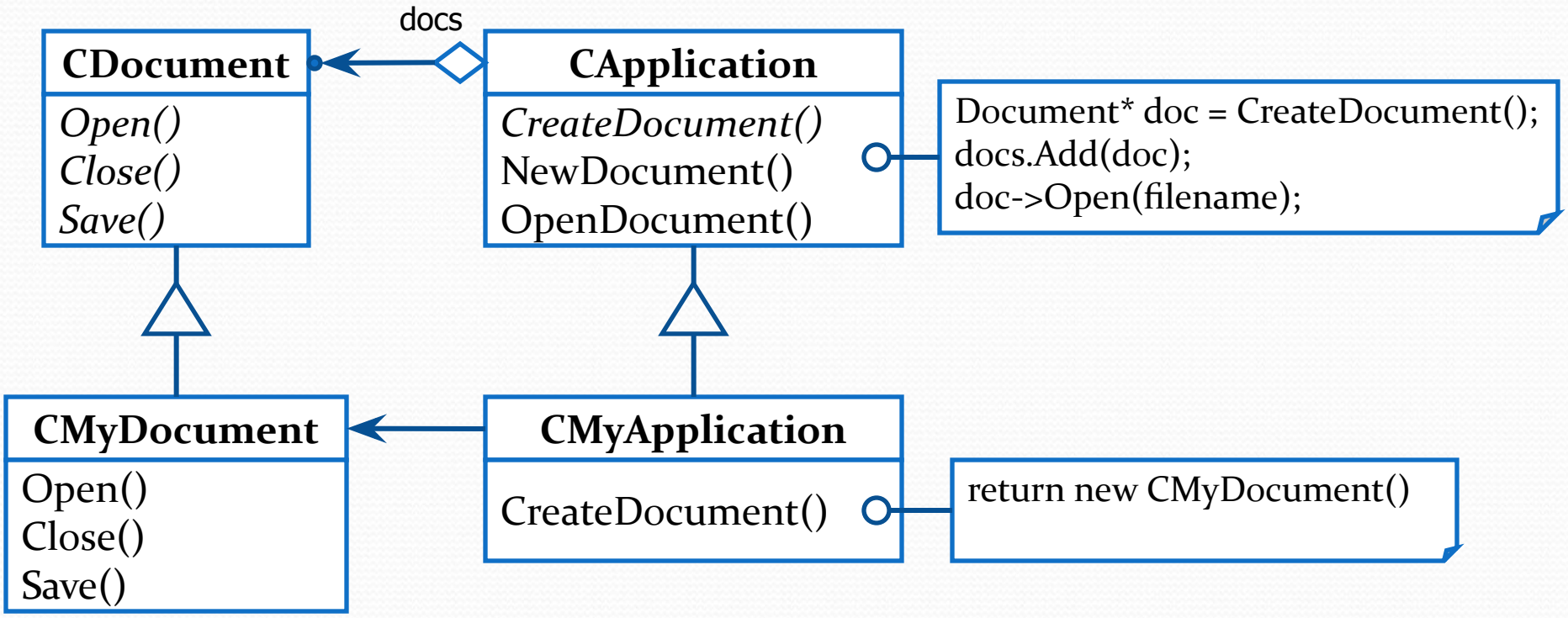
Применимость

- Классу 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
  - тип конкретного вспомогательного подкласса выбирается в ходе выполнения программы

Пример использования

- Каркас приложения, позволяющего пользователю работать с различными типами документов
  - Основные абстракции в таком каркасе – классы CApplication и CDocument
    - CApplication – управляет созданием и открытием документов
    - CDocument – базовый абстрактный класс CDocument, основа для создания конкретных документов
- **Проблема** – класс CApplication знает, в какой момент документ должен быть создан, но ему известно лишь об абстрактном классе CDocument, который инстанцировать нельзя
  - Паттерн «Фабричный метод» предлагает решение данной проблемы

Иерархия классов



Абстрактные реализации сущностей «Product» и «Creator»

```
// Абстрактный Продукт
class CDocument
{
public:
        virtual ~CDocument(){}
        virtual void Open()=0;
        virtual void Save()=0;
        virtual void Close()=0;
};

typedef boost::shared_ptr<CDocument>
CDocumentPtr;
```

```
// Абстракный Создатель
class CApplication
{
public:
        ~ CApplication(){}
        void NewDocument()
        {
                CDocumentPtr pDocument(CreateDocument());
                m_docs.push_back(pDocument);
        }
        void OpenDocument()
        {
                CDocumentPtr pDocument(CreateDocument());
                m_docs.push_back(pDocument);
                pDocument->Open();
        }
protected:
        // фабричный метод
        virtual CDocument * CreateDocument()const = 0;
private:
        std::vector<CDocumentPtr> m_docs;
};
```

Конкретные реализации сущностей «Product» и «Creator»

```
// конкретный продукт
class CMyDocument : public CDocument
{
public:
	virtual void Open()
	{
		// ...
	}
	virtual void Close()
	{
		// ...
	}
	virtual void Save()
	{
		// ...
	}
};
```

```
// Конкретный создатель
class CMyApplication : public CApplication
{
protected:
	// реализация фабричного метода
	virtual CDocument * CreateDocument()const
	{
		return new CMyDocument();
	}
};
```

Достоинства и недостатки паттерна «Фабричный метод»

- Достоинства
  - Фабричные методы избавляют проектировщика от необходимости встраивать в код зависящие от приложения классы
  - Код имеет дело только с интерфейсом класса Product, поэтому он может работать с любыми определенными пользователем классами конкретных продуктов
- Недостатки
  - Клиентам, возможно, придется создавать подкласс класса Creator для создания лишь одного объекта ConcreteProduct