

Технология разработки программного обеспечения (вторая часть)

Порождающие шаблоны проектирования ПО

проф. каф. ОСУ Тузовский А.Ф.

Лекция 6

1. Порождающие паттерны

- Порождающие паттерны проектирования абстрагируют процесс создания объектов класса.
 - Они помогают сделать систему независимой от способа создания, композиции, и представления объектов.
 - Позволяют ответить на вопрос: кто, когда и как создает объекты в системе.

1. **Abstract Factory** (Абстрактная фабрика)

2. **Builder** (Строитель)

3. **Factory Method** (Фабричный метод)

4. **Prototype** (Прототип)

5. **Singleton** (Одиночка)

1.1. Паттерн **Abstract Factory** (Абстрактная фабрика)

- **Название паттерна**
 - **Abstract Factory** / Абстрактная фабрика
 - другие названия:
 - Toolkit / Инструментарий
 - Factory/Фабрика
- **Цель паттерна**
 - предоставить интерфейс для проектирования и реализации семейства, взаимосвязанных и взаимозависимых объектов, не указывая конкретных классов, объекты которых будут создаваться.

Пояснение

- В соответствии с принцип инверсии зависимости (DIP) следует
 - использовать зависимости от абстрактных классов и
 - избегать зависимостей от конкретных классов, особенно когда последние изменчивы.
- Нарушение принципа DIP:

```
Circle c = new Circle(origin, 1);
```

- Здесь `Circle` – конкретный класс, значит, модули, которые создают экземпляры `Circle`, обязательно нарушают DIP.
- ***Любой код, в котором используется ключевое слово `new`, не согласуется с принципом DIP.***

- Часто нарушение принципа DIP практически безвредно.
- Чем выше вероятность того, что конкретный класс будет изменяться, тем вероятнее, что зависимость от него приведет к неприятностям.
 - если конкретный класс не склонен к изменениям, то ничего страшного в зависимости от него нет.
- Например: создание объектов типа `string` не вызывает у меня беспокойства.
 - Зависимость от класса `string` вполне безопасна, потому что в обозримом будущем этот класс не изменится.

- При активной разработки приложения многие конкретные классы часто изменяются – зависимость от них может стать источником проблем.
- Лучше зависеть от абстрактного интерфейса, тогда программа будет изолирована от большинства изменений.
- Паттерн **Abstract Factory** позволяет создавать конкретные объекты, не выходя за рамки зависимости от абстрактного интерфейса.
- Он очень полезен на тех этапах разработки приложения когда конкретные классы еще очень изменчивы.

Когда следует использовать паттерн **Abstract Factory**

- система не должна зависеть от того, как в ней создаются и компонуются объекты;
- объекты, входящие в семейство, должны использоваться вместе;
- система должна конфигурироваться одним из семейств объектов;
- надо предоставить интерфейс библиотеки, не раскрывая её внутреннюю реализацию.

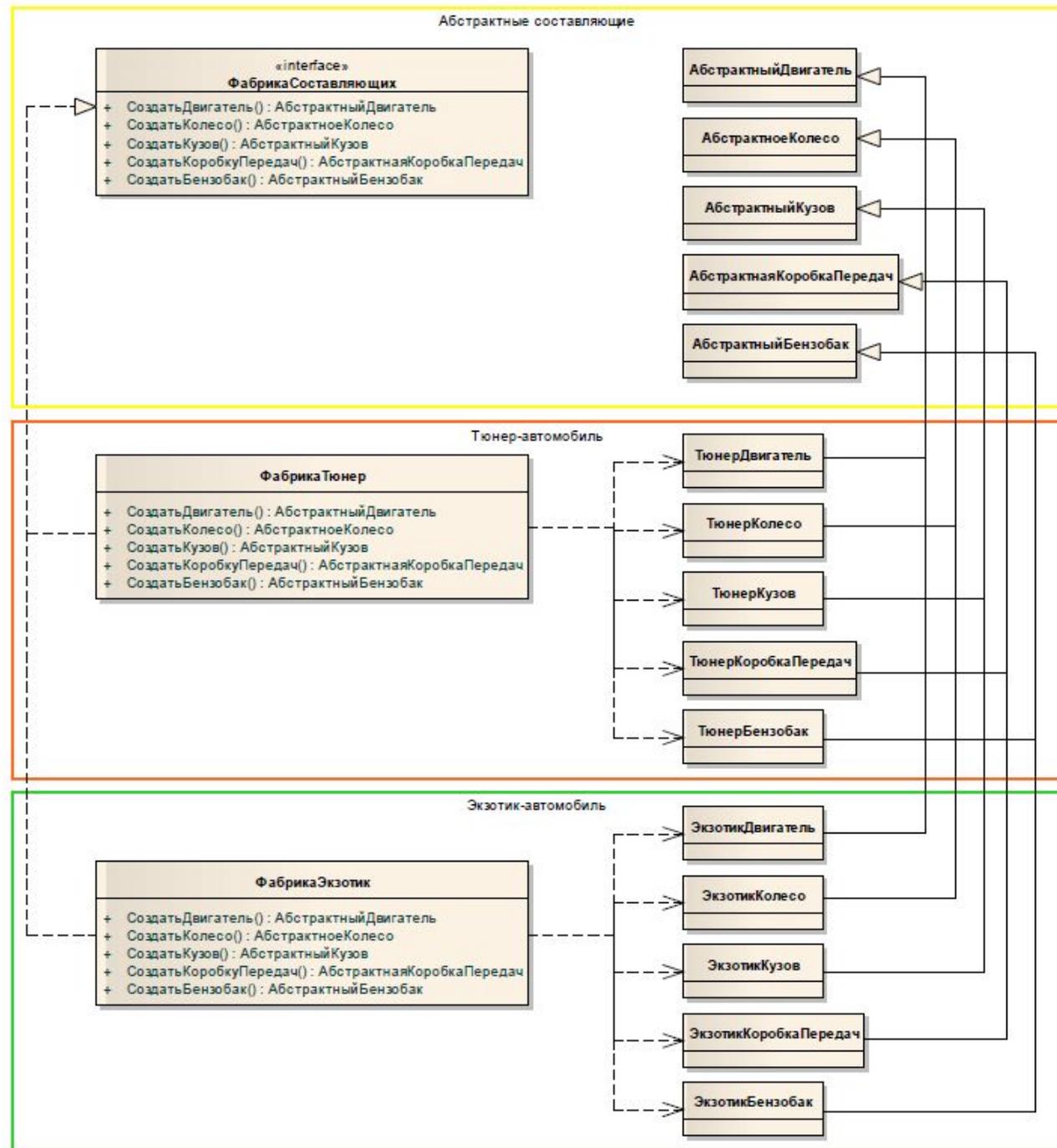
Пример – игра «Супер Ралли» (гонка на автомобилях)

- Одно из требований: игрок должен иметь возможность **выбирать себе автомобиль для участия в гонках.**
- Каждый из автомобилей состоит из специфического набора составляющих:
 - двигатель, колес, кузов, коробка передач, бензобак
 - определяют возможности автомобиля (скорость, маневренность, устойчивость к повреждениям, длительность непрерывной гонки без дозаправки и д.р.).
- Может быть много разных типов автомобилей.
- Их количество может изменяться динамически (например, в зависимости от опыта игрока).
- Клиентский код, выполняющий конфигурирование автомобиля специфичным семейством составляющих, **не должен зависеть от типа**

Предлагаемая реализации

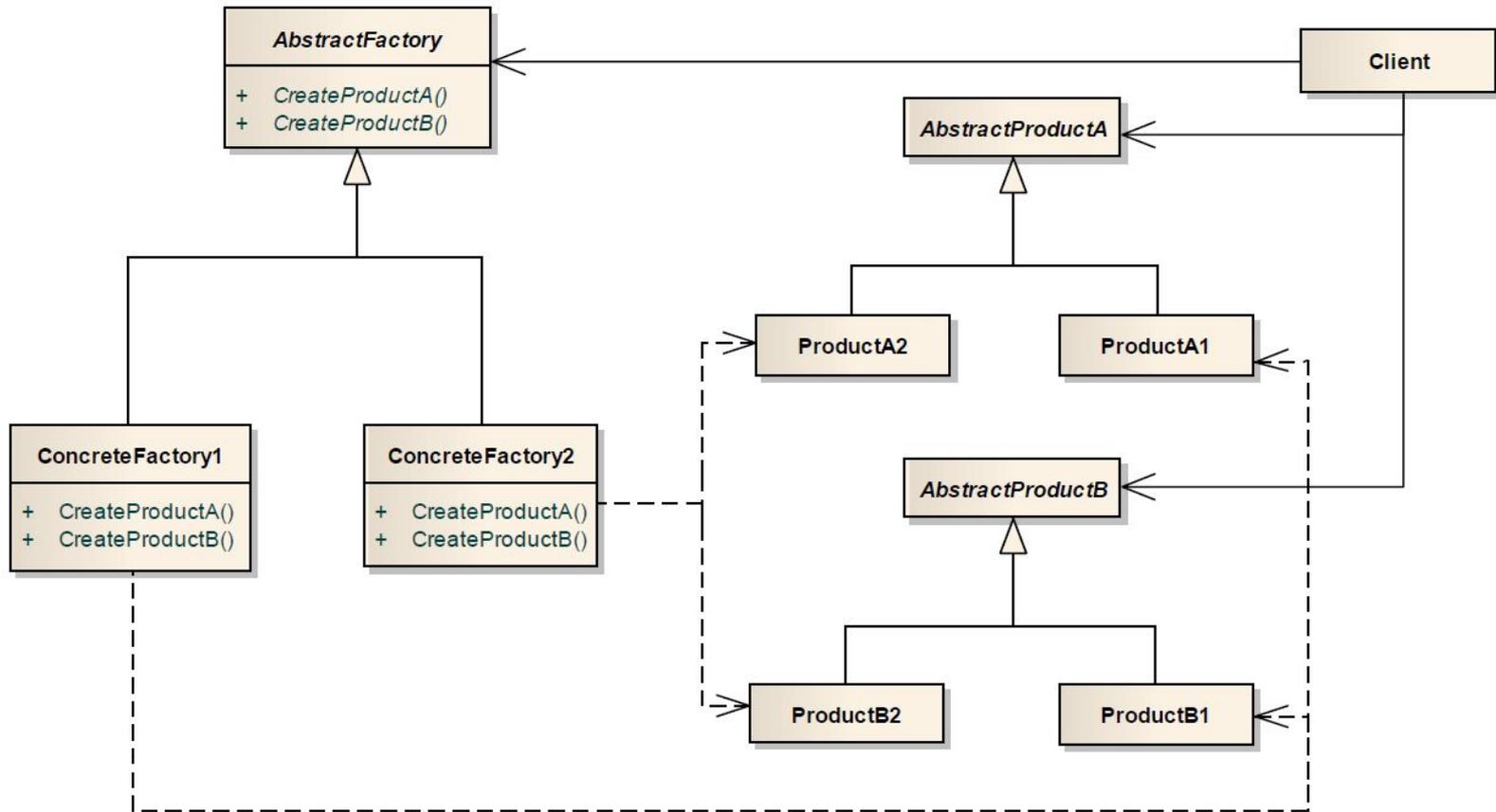
- Создается интерфейс **ФабрикаСоставляющих** – предназначен для создания конкретных классов (фабрик), которые будут создавать семейства составляющих для каждого конкретного типа автомобиля.
- Методы этого класса должны возвращать ссылки на абстрактные составляющие, что позволит в конкретных классах-фабриках, создавать конкретные составляющие (подклассы абстрактных составляющих)

Диаграмма классов



- Клиентский код, который «собирает» автомобиль из деталей, использует интерфейсную ссылку **ФабрикаСоставляющих**,
 - методы данного интерфейса возвращают ссылки на абстрактные составляющие.
- Можно передавать клиенту объект конкретной фабрики, которая создает семейство объектов конкретных составляющих.

Общая структура паттерна Abstract Factory



Участники паттерна

Abstract Factory

- Интерфейс *AbstractFactory* — абстрактная фабрика
 - Предоставляет общий интерфейс для создания семейства продуктов.
- Класс *ConcreteFactory* — конкретная фабрика
 - Реализует интерфейс *AbstractFactory* и создает семейство конкретных продуктов.
- Метод интерфейса *AbstractProduct* — абстрактный продукт
 - Предоставляет интерфейс абстрактного продукта, ссылку на который возвращают методы фабрик.
- Метод класса *ConcreteProduct* — конкретный продукт
 - Реализует конкретный тип продукта, который создается конкретной фабрикой.

Отношения между участниками

- Клиент знает только о существовании абстрактной фабрики и абстрактных продуктов.
- Для создания семейства конкретных продуктов клиент конфигурируется соответствующим экземпляром конкретной фабрики.
- Методы конкретной фабрики создают экземпляры конкретных продуктов, возвращая их в виде ссылок на соответствующие абстрактные продукты.

Достоинства использования паттерна

- Позволяет изолировать конкретные классы продуктов.
- Клиент знает о существовании только абстрактных продуктов, что ведет к упрощению его архитектуры.
- Упрощает замену семейств продуктов.
- Для использования другого семейства продуктов достаточно конфигурировать клиентский код соответствующий конкретной фабрикой.
- Дает гарантию сочетаемости продуктов.
 - Так как каждая конкретная фабрика создает группу продуктов, то она и следит за обеспечением их сочетаемости.

Недостаток использования паттерна

- Трудно поддерживать новые виды продуктов, которые содержат, другой состав компонент.
- Для добавления нового продукта необходимо изменять всю иерархию фабрик, а также клиентский код.

Практический пример использования паттерна

- Задача – разработать ПО для магазина компьютерной техники.
- Одно из требований – быстрое создание конфигурации системного блока.
- Предположим, что в состав конфигурации системного блока входят:
 1. бокс (Box);
 2. процессор (Processor);
 3. системная плата (MainBoard);
 4. жесткий диск (Hdd);
 5. оперативная память (Memory).

- Допустим, что программа должна создавать шаблоны типичных конфигураций двух типов:
 - домашняя конфигурация;
 - офисная конфигурация.
- Для всех этих конфигураций определим абстрактный класс.
- Конкретные модели составляющих будем определять путем наследования от абстрактного базового класса,

Класс персонального компьютера

Рс

- Класс, представляющий конфигурацию системного блока:

```
public class Pc
{
    public Box Box { get; set; }
    public Processor Processor { get; set; }
    public MainBoard MainBoard { get; set; }
    public Hdd Hdd { get; set; }
    public Memory Memory { get; set; }
}
```

Интерфейс фабрики создания конфигурации системного блока

- Ответственность за их создание заданной конфигурации надо возложить на один класс-фабрику.
- Эта фабрика должна реализовать интерфейс `IPcFactory` .
- Методы этого интерфейса возвращают ссылки на классы абстрактных продуктов.

```
public interface IPcFactory
{
    Box CreateBox ( ) ;
    Processor CreateProcessor ( ) ;
    MainBoard CreateMainBoard ( ) ;
    Hdd CreateHddO ;
    Memory CreateMemory ( ) ;
}
```

- Для создания компонентов конфигураций определяем классы конкретных фабрик
 - `HomePcFactory`
 - `OfficePcFactory`.
- В каждом из `create`-методов этих классов создается объект конкретного класса продукта, соответствующего типу конфигурации.

Класс HomePcFactory

- Фабрика для создания "домашней" конфигурации системного блока ПК

```
public class HomePcFactory : IPcFactory
{
    public Box CreateBox()
    { return new SilverBox(); }
    public Processor CreateProcessor()
    {return new IntelProcessor(); }
    public MainBoard CreateMainBoard()
    { return new MSIMainBord(); }
    public Hdd CreateHddO { return new SamsungHDD(); }
    public Memory CreateMemory()
    { return new Ddr2Memory();}
}
```

Класс OfficePcFactory

- Фабрика для создания "офисной" конфигурации системного блока ПК

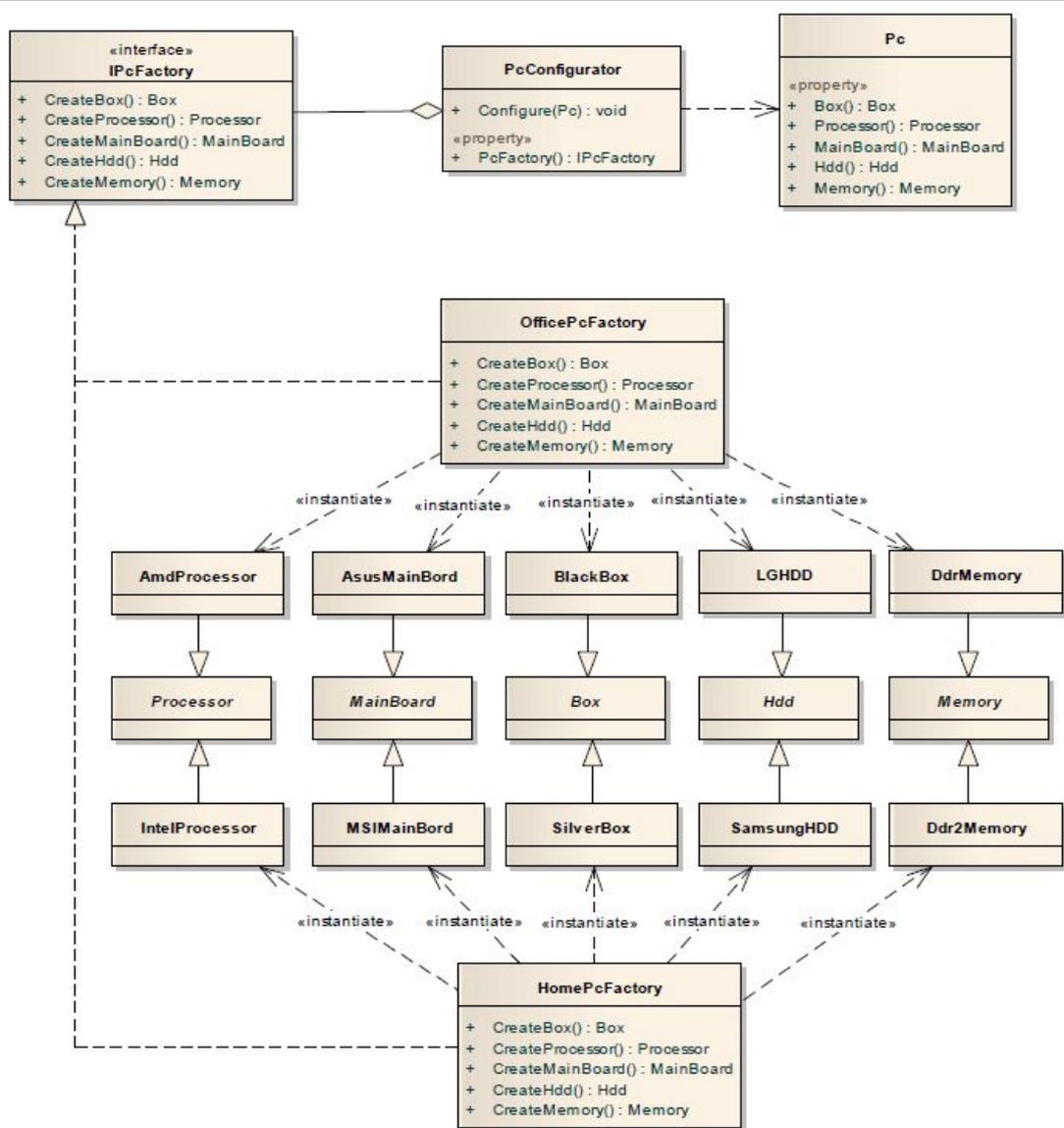
```
public class OfficePcFactory : IPcFactory
{
    public Box CreateBox()
    {return new BlackBoxf); }
    public Processor CreateProcessor()
    { return new AmdProcessor();}
    public MainBoard CreateMainBoard()
    {return new AsusMainBord(); }
    public Hdd CreateHdd{} {return new LGHDD ();}
    public Memory CreateMemory()
    { return new DdrMemory(); }
}
```

Класс PcConfigurator

- Ответственен за создание объекта типа Pc выбранного типа

```
public class PcConfigurator {  
    public IPcFactory PcFactory { get; set; }  
    public void Configure(Pc pc) {  
        pc.Box = PcFactory.CreateBox();  
        pc.MainBoard = PcFactory.CreateMainBoard();  
        pc.Hdd = PcFactory.CreateHdd() ;  
        pc.Memory = PcFactory.CreateMemory();  
        pc.Processor = PcFactory.CreateProcessor();  
    }  
}
```

Полная диаграмма классов



- Класс `PcConfigurator` принимает экземпляр конкретной фабрики и с помощью её методов создает составляющие персонального компьютера.
- `PcConfigurator` работает с интерфейсной ссылкой `IPcFactory` и ничего не знает о конкретных фабриках конфигураций и конкретных составляющих.
- Сила абстрактной фабрики
 - конкретную фабрику можно определять на этапе выполнения программы
 - клиентский код не зависит от конкретных фабрик или конкретных продуктов.

Паттерн Factory (Фабрика)

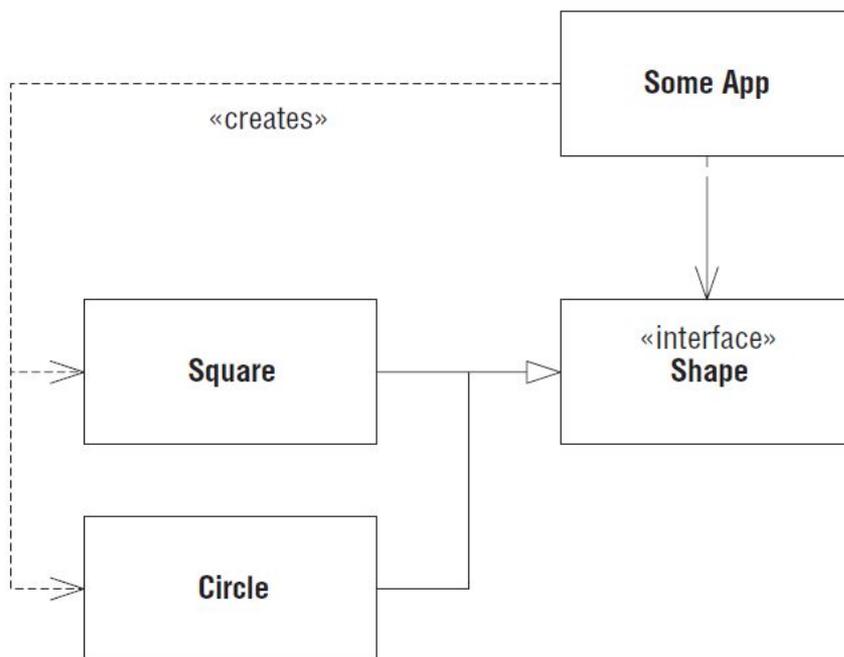
- В соответствии с принцип инверсии зависимости следует
 - использовать зависимости от абстрактных классов и
 - избегать зависимостей от конкретных классов, особенно когда последние изменчивы.
- Следующий фрагмент кода нарушает принцип DIP:
`Circle c = new Circle(origin, 1);`
- Здесь `Circle` – конкретный класс, значит, модули, которые создают экземпляры `Circle`, обязательно нарушают DIP.
- ***Любой код, в котором используется ключевое слово `new`, не согласуется с принципом DIP.***

- Достаточно часто нарушение принципа DIP практически безвредно.
- Чем выше вероятность того, что конкретный класс будет изменяться, тем вероятнее, что зависимость от него приведет к неприятностям.
- Но если конкретный класс не склонен к изменениям, то ничего страшного в зависимости от него нет.
- Так, создание объектов типа `string` не вызывает у меня беспокойства.
- Зависимость от класса `string` вполне безопасна, потому что в обозримом будущем этот класс не изменится.

- Однако, во время активной разработки приложения многие конкретные классы часто изменяются, поэтому зависимость от них может стать источником проблем.
- Лучше зависеть от абстрактного интерфейса, тогда программа будет изолирована от большинства изменений.
- Паттерн **Factory** (Фабрика) позволяет создавать конкретные объекты, не выходя за рамки зависимости от абстрактного интерфейса.
- Он очень полезен на тех этапах разработки приложения когда конкретные классы еще очень изменчивы.

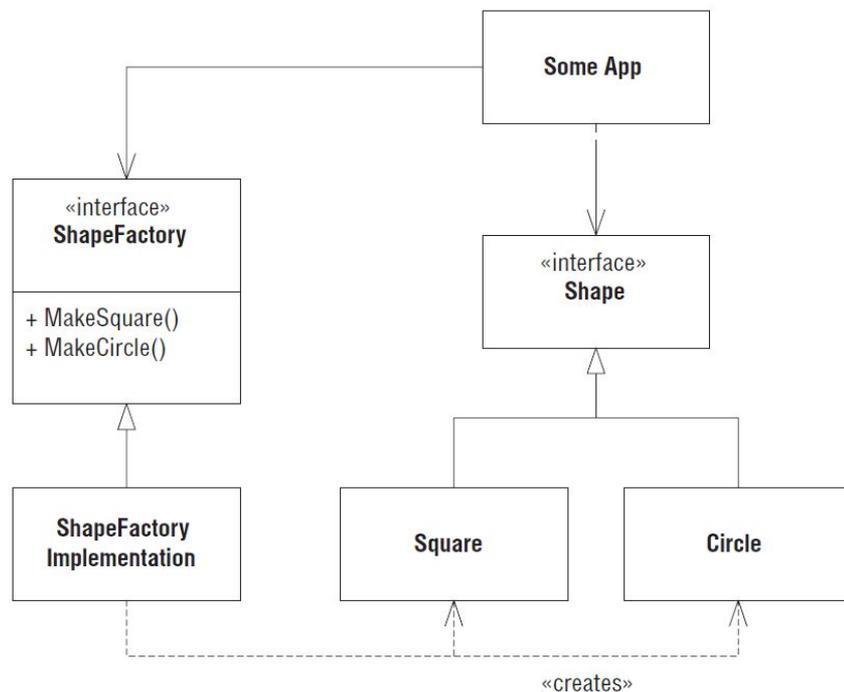
Пример программы

- Имеется класс **SomeApp**, зависящий от интерфейса **Shape**.
- **SomeApp** обращается к экземплярам **Shape** исключительно через интерфейс **Shape** и не пользуется методами, специфичными для классов **Square** или **Circle**.
- К сожалению, **SomeApp** также создает экземпляры **Square** и **Circle** и, следовательно, зависит от ЭТИХ КОНКРЕТНЫХ КЛАССОВ.



Применение паттерна Фабрика к классу SomeApp

- Интерфейс **ShapeFactory**, в котором объявлены два метода:
 - **MakeSquare()** возвращает экземпляр Square,
 - **MakeCircle()** возвращает экземпляр Circle.
- В обоих случаях возвращаемое значение имеет тип **Shape**.



```
public interface ShapeFactory {  
    Shape MakeCircle();  
    Shape MakeSquare();  
}
```

```
public class ShapeFactoryImplementation :  
    ShapeFactory {  
    public Shape MakeCircle()  
    { return new Circle(); }  
    public Shape MakeSquare()  
    { return new Square(); }  
}
```

- Этот прием полностью решает проблему зависимости от конкретных классов.
- Код приложения не зависит от `Circle` и `Square`, но может успешно создавать объекты обоих классов.
- Он манипулирует этими объектами через интерфейс `Shape` и никогда не вызывает методов, специфичных только для `Square` или `Circle`.
- Проблема зависимости от конкретного класса перенесена в другое место.

- Экземпляр `ShapeFactoryImplementation` где-то должен создаваться.
- Но больше ни в одном месте объекты `Square` и `Circle` напрямую не создаются.
- Экземпляр `ShapeFactoryImplementation`, обычно создается в головной программе
 - или в специальном методе инициализации, который вызывается из `Main`.

Проблема зависимости

- В описанном варианте паттерна Фабрика имеется проблема.
- В интерфейсе `ShapeFactory` объявлены методы для всех классов, производных от `Shape`.
- Это приводит к *зависимости по именам*, усложняющей добавление новых подклассов `Shape`.
- При добавлении каждого нового подкласса нужно также добавить новый метод в интерфейс `ShapeFactory`.
 - придется заново откомпилировать и развернуть все клиенты `ShapeFactory`.

- Можно избавиться от этой проблемы, частично пожертвовав безопасностью типов.
- Вместо того чтобы вводить в `ShapeFactory` по одному методу для каждого подкласса `Shape`, можно оставить всего один метод, принимающий строку.

[Test]

```
public void TestCreateCircle() {  
    Shape s = factory.Make("Circle");  
    Assert.IsTrue(s is Circle);  
}
```

- В этом случае в классе `ShapeFactoryImplementation` нужно будет использовать цепочку предложений `if/else`, в которых входной аргумент анализируется на предмет выбора подходящего подкласса `Shape`.

```
public interface ShapeFactory {  
    Shape Make(string name);  
}  
  
public class ShapeFactoryImplementation : ShapeFactory  
{  
    public Shape Make(string name) {  
        if(name.Equals("Circle")) return new Circle();  
        else if(name.Equals("Square")) return new Square();  
        else  
            throw new Exception("ShapeFactory не может создать: {0}", name);  
    }  
}
```

Опасность

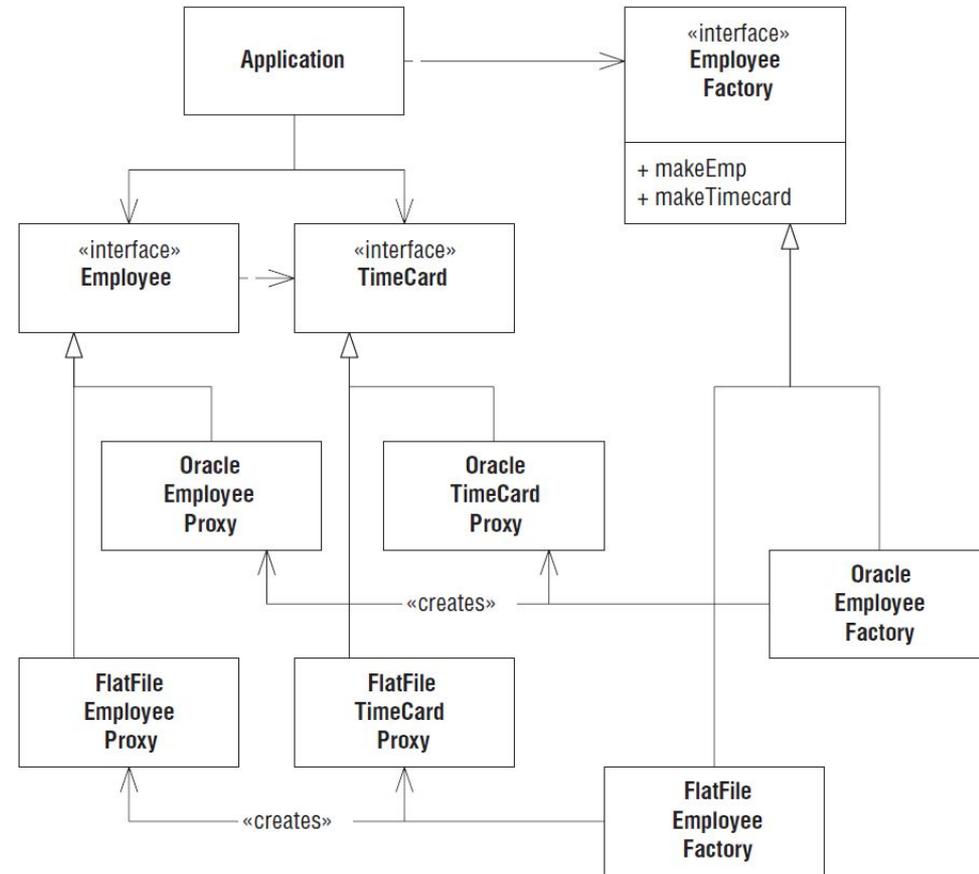
- При неправильном написания названия фигуры клиент получит ошибку на этапе выполнения, а не компиляции.
- Такие ошибки времени выполнения будут обнаружены заранее если
 - пишутся автономные тесты и
 - применяете методика разработки через тестирование.

Взаимозаменяемые фабрики

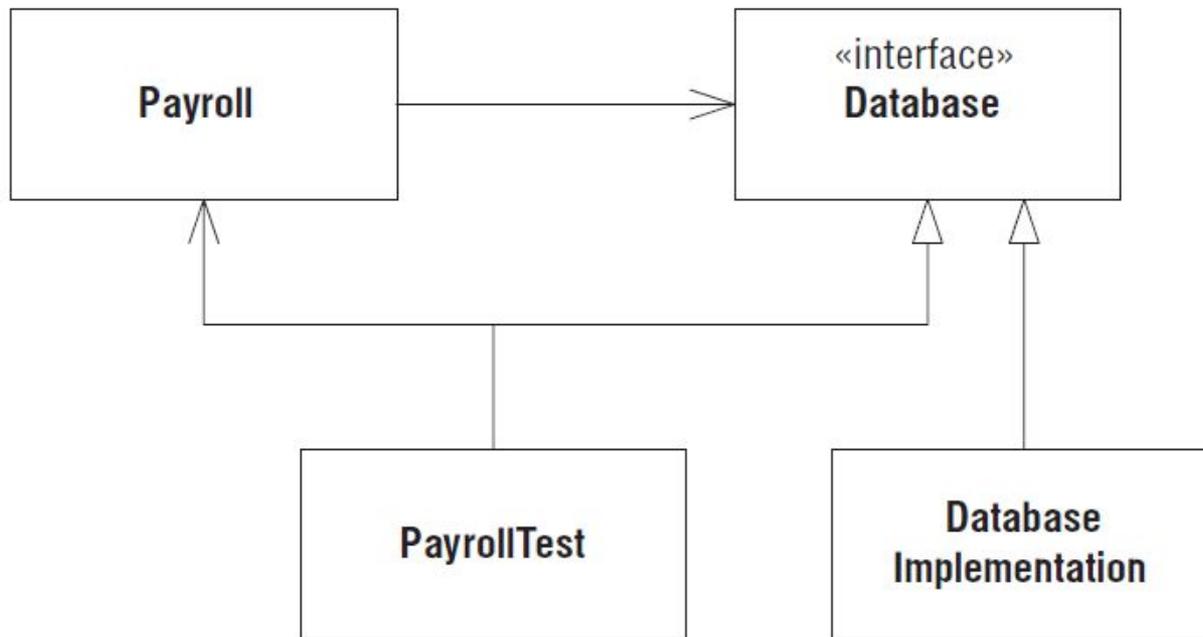
- Большое достоинство фабрик – возможность подменять одну реализацию фабрики другой.
- Это позволяет подставлять в приложение различные семейства объектов.
- Например, пишется приложение, которое должно адаптироваться к нескольким реализациям базы данных.
 - Допустим, что можно либо работать с плоскими файлами, либо купить адаптер к СУБД Oracle.
- Чтобы изолировать приложение от реализации базы данных, можно воспользоваться паттерном **Proxy** (Заместитель).
- А для создания экземпляров заместителей можно применить фабрики.

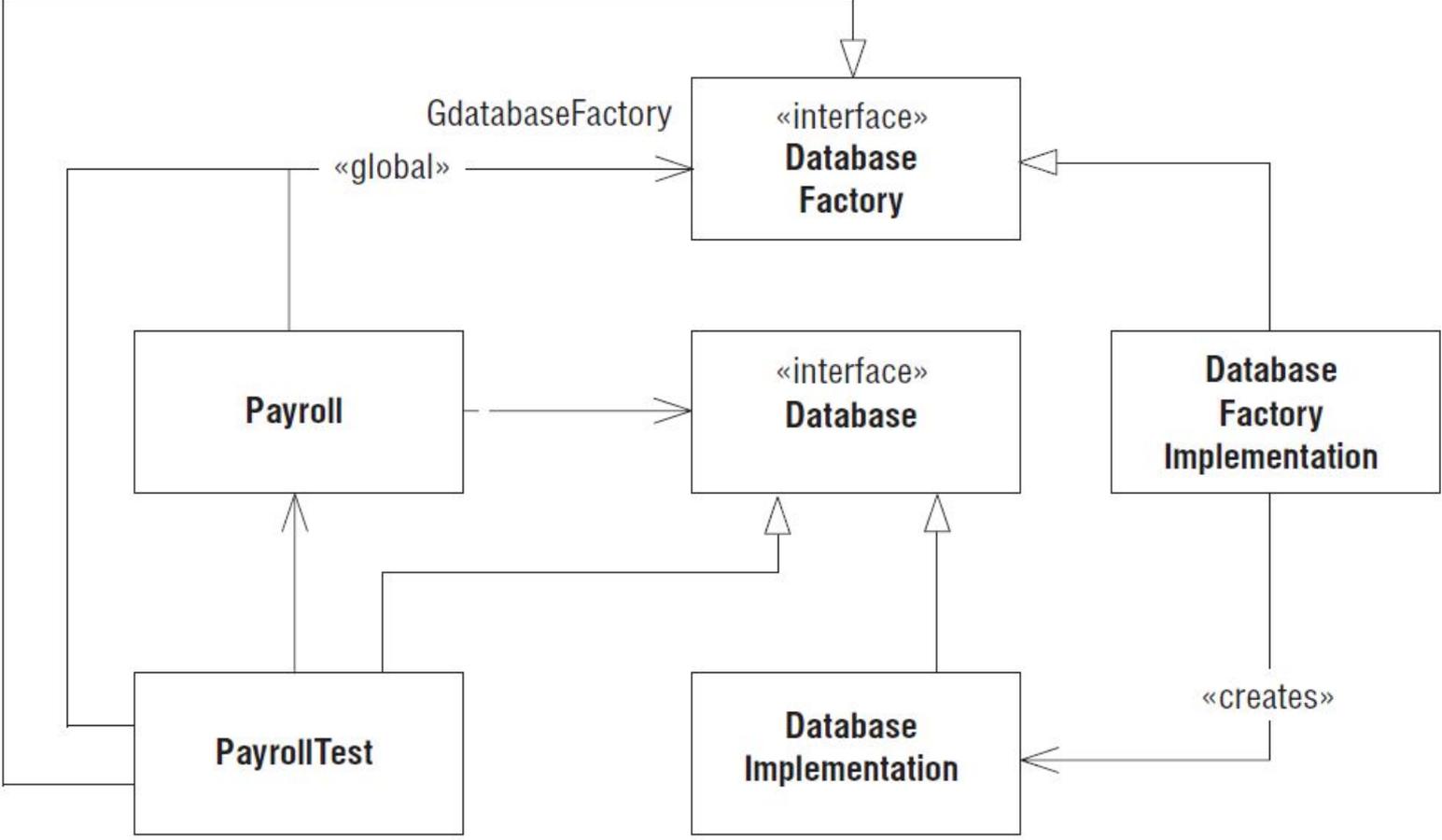
Диаграмма классов

- Есть две реализации интерфейса **EmployeeFactory**.
 - одна создает объекты-заместители для работы с плоскими файлами,
 - другая – для работы с Oracle.
- Приложение не знает о том, какая реализация используется.









Важность фабрик

- Строгая интерпретация принципа DIP означала бы, что нужно создавать фабрику для любого изменчивого класса.
- При этом паттерн Фабрика выглядит очень соблазнительно.
- Поэтому разработчики могут поддаваться искушению использовать фабрики по умолчанию.
- Но этой крайность, которая не рекомендуется.
- Рекомендуется их использовать только при возникновении настоящей необходимости.

Когда использовать Фабрику

- Если нужно применить паттерн Заместитель, возможно, имеет смысл завести и фабрику для создания сохраняемых объектов.
- Если в процессе автономного тестирования возникает ситуация, когда нужно подменить создателя объектов.
- Но пока полезность фабрики не стала очевидной следует обходиться без нее.

- Фабрики приносят сложность, которой часто можно избежать, особенно на ранних стадиях проектирования.
- Если использовать их к часто, то развитие дизайна может сильно осложниться.
- Для создания всего одного содержательного класса придется ввести целых четыре сущности:
 - два интерфейса, представляющих сам новый класс и его фабрику, и
 - два конкретных класса, реализующих эти интерфейсы.

Выводы

- Шаблон Фабрика – это мощный инструмент.
- Она может оказаться ценным инструментом,
 - обеспечивающим согласование с принципом DIP, (позволяет модулям верхнего уровня создавать экземпляры классов, не становясь зависимыми от конкретных реализаций этих классов).
 - дают возможность подменять целые семейства реализаций групп классов.
- Но Фабрики вносят сложность, без которой часто можно обойтись.
- Повсеместное их применение редко бывает оптимальным курсом.

1.2. Паттерн **Builder** (Строитель)

- **Название паттерна**
 - **Builder** / Строитель.
- **Цель паттерна**
 - отделяет процесс построения сложного объекта от его представления – в результате одного и того же процесса создания получаются разные представления объекта.
 - клиентский код может породить сложный объект, определит для него не только тип, но и внутреннее содержимое.
 - клиент не обязан знать о деталях конструирования объекта.

Паттерн **Builder** следует ИСПОЛЬЗОВАТЬ КОГДА...

- Общий алгоритм построения сложного объекта не должен зависеть от специфики каждого из его шагов.
- В результате одного и того же алгоритма конструирования надо получить различные продукты.

Пояснение причины возникновения паттерна

- В качестве примера, рассмотрим ***конвейер выпуска автомобилей.***
- Смысл конвейера – пошаговое построение сложного продукта (например, автомобиля).
- При этом:
 - конвейер определяет общую последовательность шагов (т.е. алгоритм) построения.
 - специфика каждого шага определяется моделью собираемого автомобиля.

- Разделение построения на
 - общий алгоритм построения и
 - специфические операции на каждом шаге

позволяет значительно экономить:

- на одном и том же конвейере могут выпускаться автомобили разных моделей с различными характеристиками.

- С общей технологической точки зрения, конвейер по сборке автомобилей включает следующие этапы:
 1. Сборка кузова.
 2. Установка двигателя.
 3. Установка колес.
 4. Покраска.
 5. Подготовка салона.
- Технические детали процессов, происходящих на каждом шаге, известны уже конкретной технологии производства модели автомобиля

- Пусть завод может производить автомобили следующих моделей:
 1. автомобили класса «мини»,
 2. спортивные автомобили,
 3. внедорожники.
- В таком случае для компании не составит проблем дополнить набор выпускаемых моделей новыми образцами без изменений общего конвейерного цикла.

Рассмотрим ОО программирование

- Определим класс **Конвейер**, который будет прототипом реального конвейера
 - будет определять общую последовательность шагов создания.
- Метод **Собрать()** этого класса будет исполнять процесс построения посредством выполнения этих шагов без зависимости от технических деталей реализации каждого шага.
- Ответственность за реализацию шагов построения возложена на абстрактный класс, который назовем

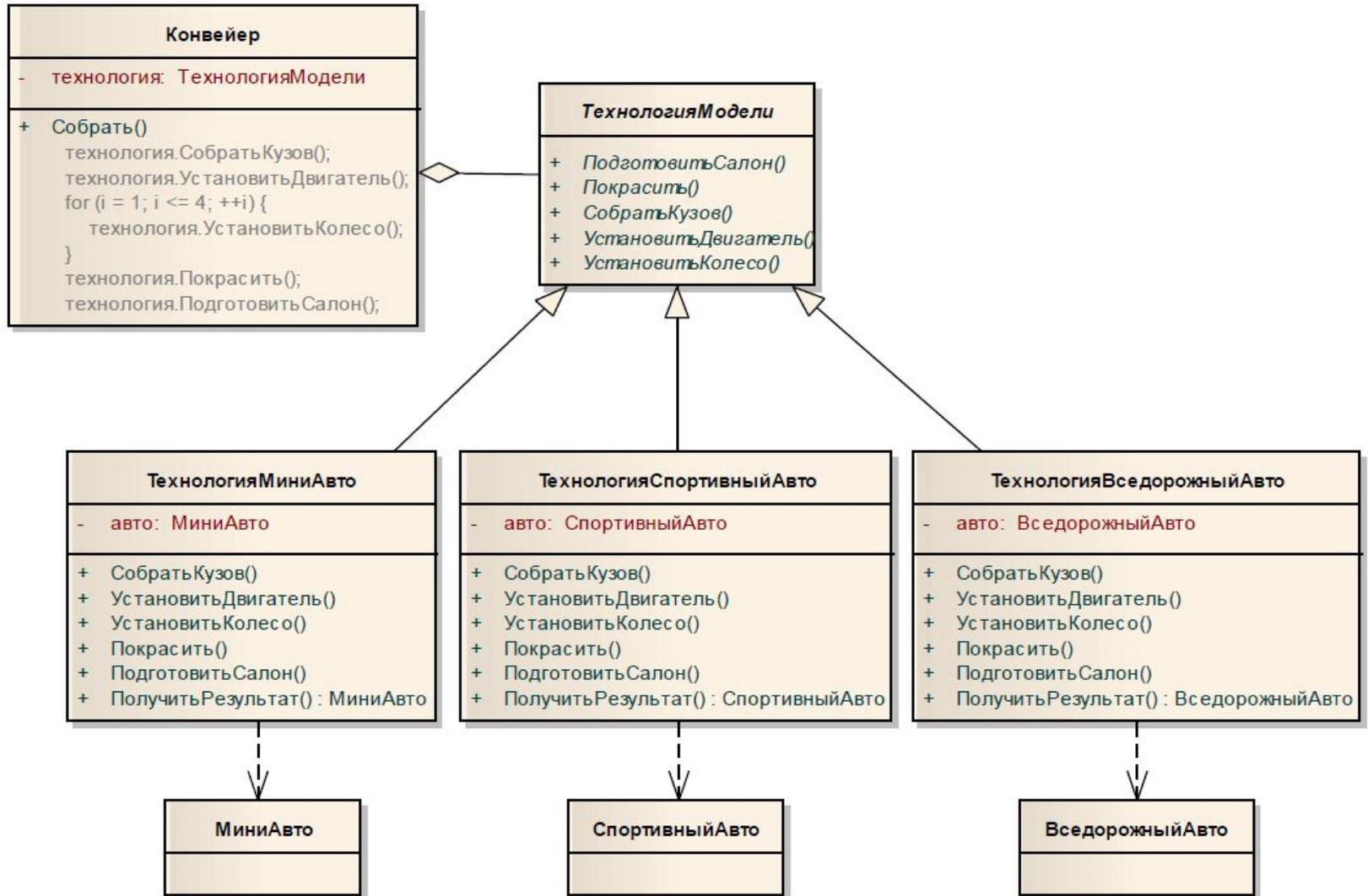
- В результате применения конкретных подклассов класса **ТехнологияМодели** можно получать на выходе разные модели автомобилей, т.е. экземпляры разных классов автомобилей.
- В данном случае определим такие подклассы класса **ТехнологияМодели**:
 - **ТехнологияМиниАвто**,
 - **ТехнологияСпортивныйАвто**,
 - **ТехнологияВнедорожныйАвто**.

- Каждая из этих технологий соответственно предусматривает выпуск таких моделей автомобилей:

1. МиниАвто,
2. СпортивныйАвто,
3. ВседорожныйАвто.

- Для начала производства автомобиля необходимо задать конкретную технологию для конвейера и вызвать метод **Собрать()**.
- После завершения процесса сборки готовый автомобиль можно получить у объекта технологии с помощью метода **ПолучитьРезультат()**.

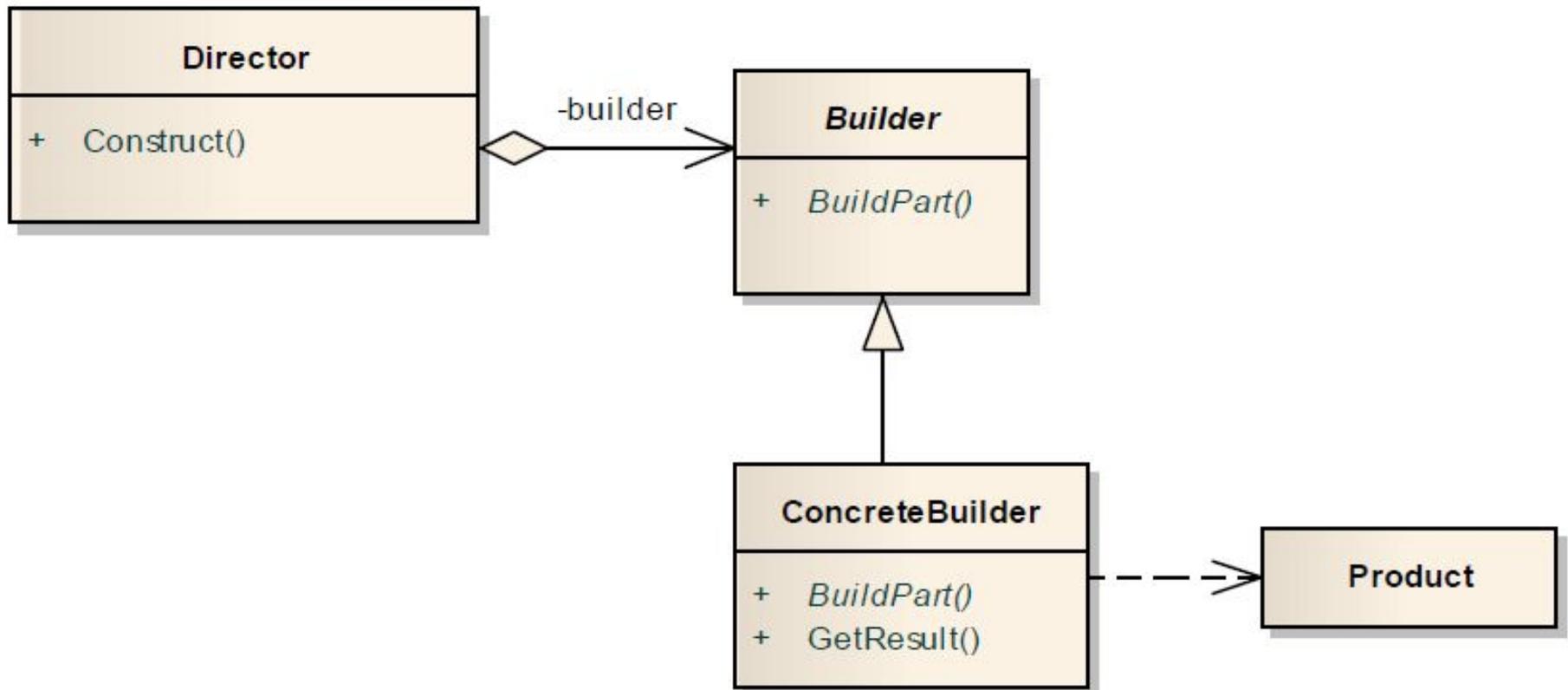
Диаграмма классов модели конвейера по производству автомобилей



Преимущества паттерна

- 1) конкретная технология построения создается по общему шаблону, реализуя действия, которые он определяет;
- 2) общий алгоритм процесса построения не зависит от деталей, специфических для конкретной технологии;
- 3) есть возможность без опасности увеличения сложности структуры модели реализовать под общий алгоритм большое количество конкретных технологий.

Структура паттерна Builder



Участники паттерна

- **Director** (Конвейер) – распорядитель
 - Определяет общий алгоритм конструирования, используя для реализации отдельных шагов возможности класса **Builder**.
- **Builder** (Технология Модели) – строитель
 - Обеспечивает интерфейс для пошагового конструирования сложного объекта (продукта) из частей.
- **ConcreteBuilder** (Технология МиниАвто и др.) – конкретный строитель
 - Реализует шаги построения сложного объекта, определенные в базовом классе **Builder**.
 - Создает результат построения (**Product**) и следит за пошаговым конструированием.
 - Определяют интерфейс для доступа к результату конструирования
- **Product** (МиниАвто и др.) – продукт
 - Сложный объект, который получается в результате конструирования

Отношения между участниками

- Клиент конфигурирует распорядителя (**Director**) экземпляром конкретного строителя.
- Распорядитель вызывает методы строителя для конструирования частей продукта.
- Конкретный строитель создает продукт и следит за его конструированием.
- Конкретный строитель представляет интерфейс для доступа к продукту.

Результаты использования паттерна

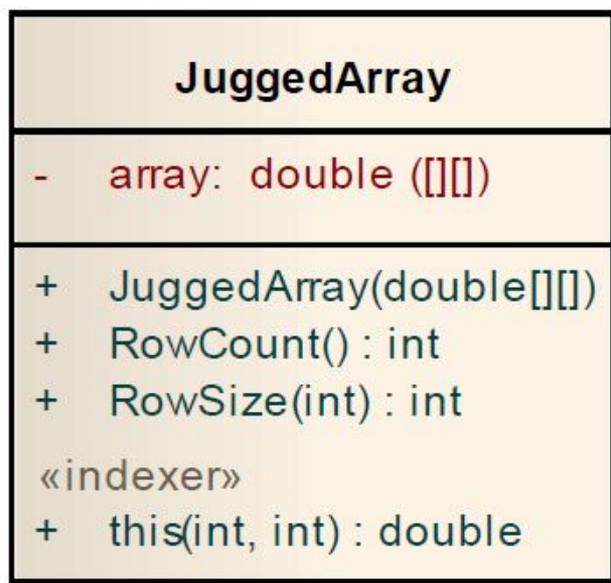
- Есть возможность изменять внутреннюю структуру создаваемого продукта (или создать новый продукт).
 - продукт конструируется через абстрактный интерфейс класса **Builder**, для добавления нового продукта достаточно определить новый вид строителя (т.е. реализовать новый подкласс класса **Builder**).
- Повышение модульности за счет разделения распорядителя и строителя.
 - Каждый строитель имеет весь необходимый код для пошагового построения продукта.
 - Поэтому он может использоваться разными распорядителями для построения вариантов продукта из одних и тех же частей.
- Пошаговое построение продукта позволяет обеспечить более пристальный контроль над процессом конструирования
 - в отличие от других порождающих паттернов, которые создают продукт мгновенно.

Пример использования паттерна

- Требуется разработать программный модуль для работы с двумерными массивами действительных чисел.
 - Строки массивов могут иметь различную длину.
- Цель – преобразование массивов в различные форматы.
- Например:
 - текстовый файл;
 - XML-файл.

Класс `JuggedArray`

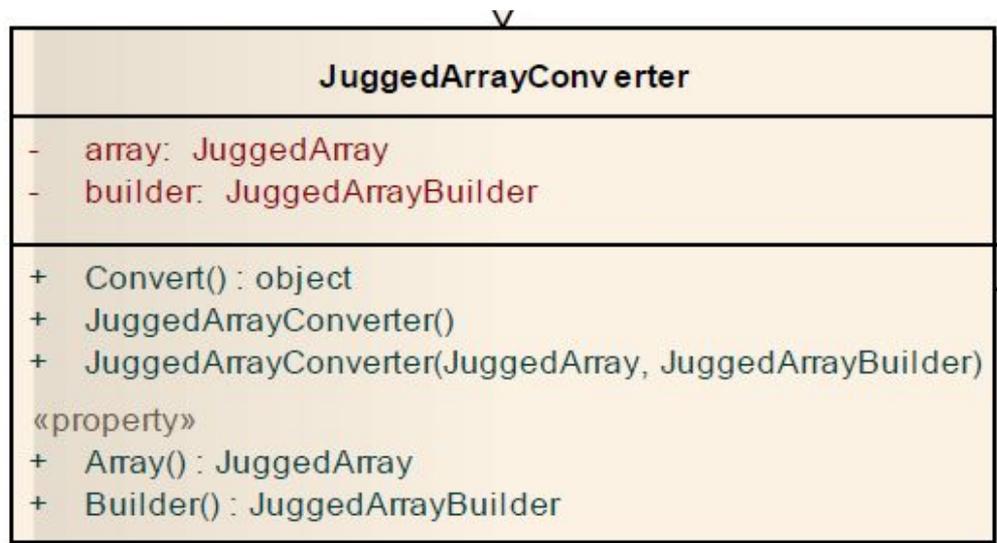
- Определим класс `JuggedArray` (зубчатый массив) с минимальными возможностями нужными для решения поставленной задачи
 - при необходимости интерфейс можно сделать более дружественным.



- Количество различных форматов заранее не известно
 - может изменяться в процессе работы над проектом.
- Клиентский код, использующий данный модуль, должен иметь возможность динамически (на этапе выполнения) выбирать формат, в который будет конвертирован массив.

Класса-конвейер

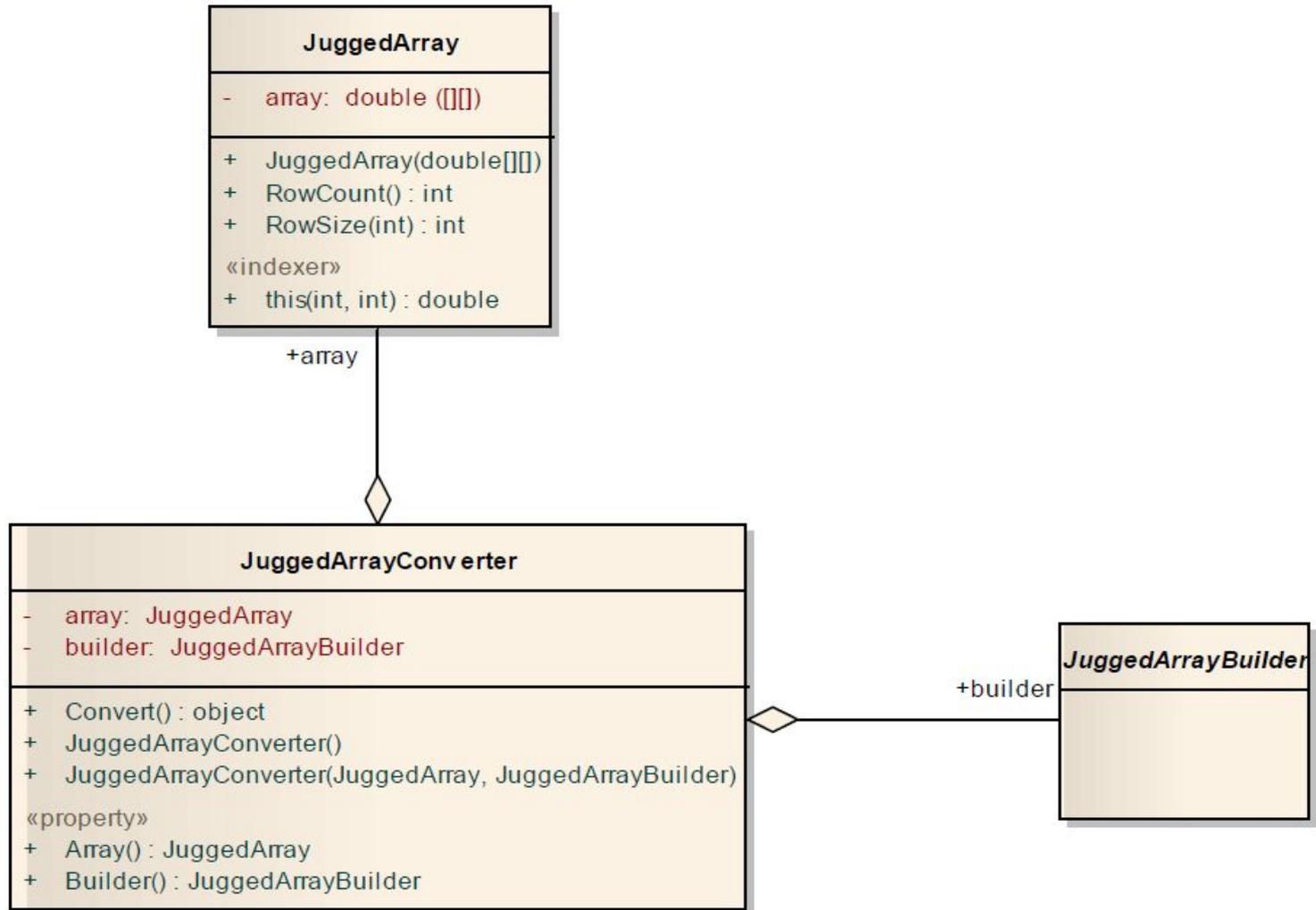
JuggedArrayConverter



Класс массива JuggedArray

```
public class JuggedArray {  
    private double [] [] array;  
    public JuggedArray (double [] [] array) // конструктор  
    { this.array = array; }  
    public double this [int row, int col] { // индексатор  
        get {return  array[row] [col];}  
        set {array[row][col] = value;}  
    }  
    public int RowCount() { return  array.Length; }  
    public int RowSize (int row) { return  array[row].Length; }  
}
```

Диаграмма классов предложенной реализации



Алгоритм пошагового конвертирования массива

- Конвертирование массива определим в классе `JuggedArrayConverter` (класса-строитель)
- В методе `Convert()` задается общее описание алгоритма пошагового конвертирования массива
 - выполнение конкретных шагов построения обеспечивает экземпляр класса-строителя.
- Все методы класса-строителя можно определить как абстрактные
 - нужно учитывать, что при реализации конкретных строителей некоторые из этих методов могут быть не реализованы.

Методы класса `JuggedArrayBuilder`

- Почти все методы сделает конкретными, но с пустой реализацией.
- Определим их как виртуальные (`virtual`).
- Абстрактным (без реализации) сделаем только метод `Result()`
– в связи с необходимостью его

Абстрактный класс-строитель

```
public abstract class JuggedArrayBuilder
{
    public virtual void Initialize () {}
    public virtual void Start() {}
    public virtual void StartRow() {}
    public virtual void AddItem(double item) {}
    public virtual void FinishRow() {}
    public virtual void Finish() {}
    public abstract object Result ();
}
```

Реализация алгоритма конвертирования массива класса

```
public object Convert()
{
    JuggedArrayBuilder
    builder.Initialize(); // инициализировать построение
    builder.Start();     // начать построение
    for (int r = 0; r < this.array.RowCount (); ++r) {
        builder.StartRow(); // начать новую строку массив
        for (int c = 0; c < this.array.RowSize(r); ++c) {
            builder.AddItem(array[r, c]); // добавить элемент массива
        }
        builder.FinishRow(); // завершить строку массива
    }
    builder.Finish();     // завершить построение
    return builder.Result(); // получить результат построения
}
```

- Создадим строитель для конвертирования массива в текст
 - Текст потом, например, можно сохранить в текстовый файл.
- Формат текстового представления
 1. отдельные строки массива должны разделяться символом «\n»;
 2. элементы строки отделяются между собой пробелом.

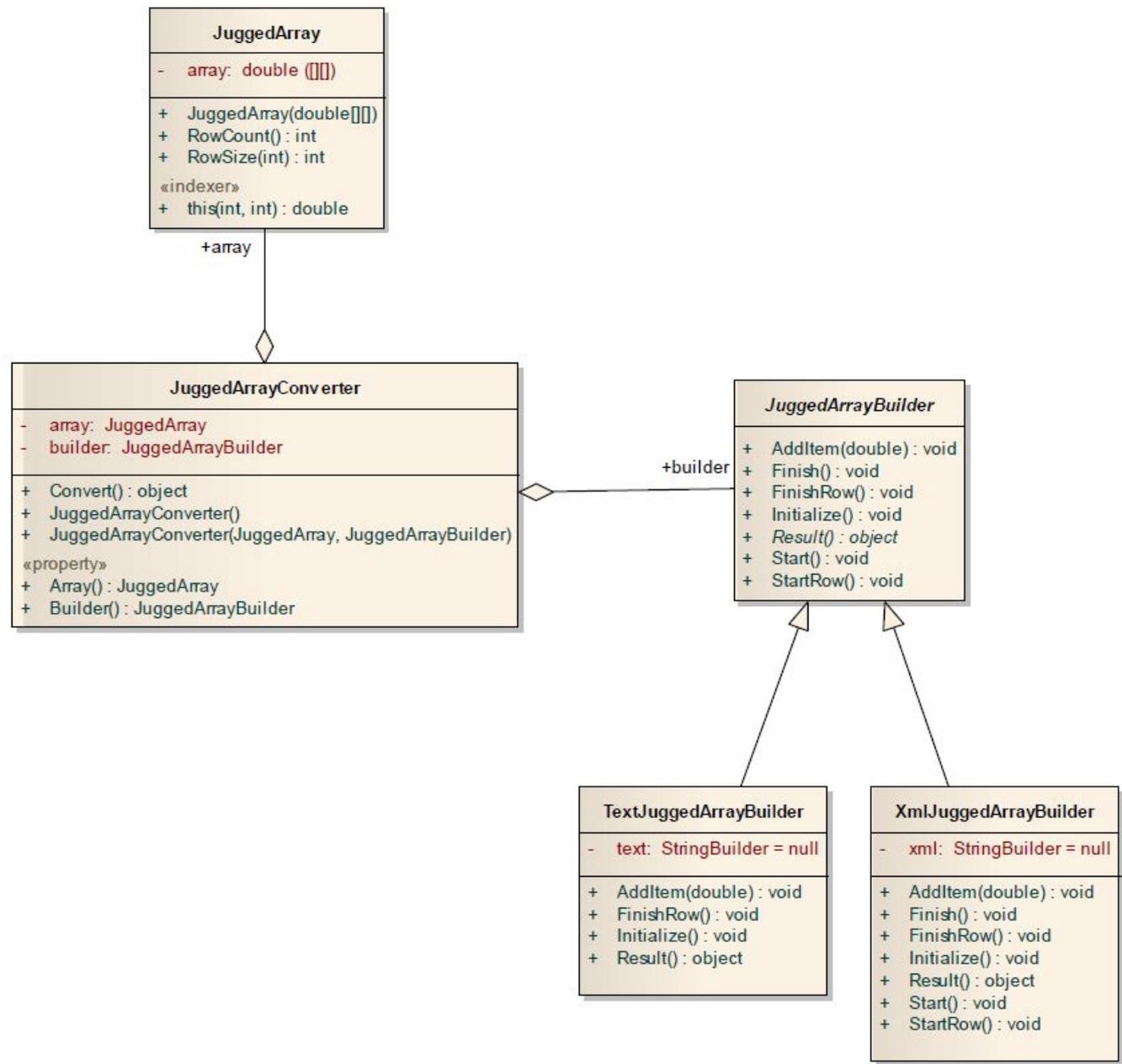
Класс TextJuggedArrayBuilder

```
public class TextJuggedArrayBuilder : JuggedArrayBuilder
{
    private StringBuilder text = null;
    public override void Initialize () { text = new StringBuilder(); }
    public override void AddItem(double item) {
        text.Append(item);
        text.Append(" ");
    }
    public override void FinishRow() { text.AppendLine(); }
    public override object Result() { return text; }
}
// методы Start() и StartRow() не реализованы
```

Класс XmlJuggedArrayBuilder

```
public class XmlJuggedArrayBuilder : JuggedArrayBuilder {
    private StringBuilder xml = null;
    public override void Initialize() { xml = new StringBuilder (); }
    public override void Start() {
        xml.AppendLine("<JuggedArray>"); xml.AppendLine("<rows>"); }
    public override void StartRow() {xml.AppendLine("<row>");}
    public override void AddItem(double item) {
        xml.Append("<item>"); xml.Append(item);
        xml.Append("</item>"); xml.AppendLine();
    }
    public override void FinishRow() { xml.AppendLine("</row>"); }
    public override void Finish() {
        xml.AppendLine("</rows>"); xml.AppendLine("</JuggedArray>"); }
    public override object Result() { return xml;}
}
```

Полный вариант диаграммы классов конвертера массивов



Программа - клиент

```
static void Main(string[] args) {
    JuggedArray array = new JuggedArray (// определяем массив
    new double [ ] [ ] { new double[] {11, 12, 13, 14, 15}, new double[] {21, 22, 23},
    new double[] {31, 32, 33, 34} });
    // создаем конвертер
    JuggedArrayConverter converter = new JuggedArrayConverter ();
    converter.Array = array; // задаем массив
    Console.WriteLine ( "Текстовое представление массива:" );
    // задаем конвертер строителем текстового представления
    converter.Builder = new TextJuggedArrayBuilder ( ) ;
    object textArray = converter.Convert ( ); // проводим конвертацию
    Console.WriteLine (textArray) ; // выводим результат на консоль
    Console.WriteLine("Xml-представление массива:" );
    // задаем конвертер строителем XML-представления
    converter.Builder = new XmlJuggedArrayBuilder();
    object xmlArray = converter.Convert(); // проводим конвертацию
    Console.WriteLine(xmlArray); // выводим результат на
    КОНСОЛЬ
}
```

C:\WINDOWS\system32\cmd.exe

Тестовое представление массива:

```
11 12 13 14 15
21 22 23
31 32 33 34
```

Xml-представление массива:

```
<JuggedArray>
<rows>
<row>
<item>11</item>
<item>12</item>
<item>13</item>
<item>14</item>
<item>15</item>
</row>
<row>
<item>21</item>
<item>22</item>
<item>23</item>
</row>
<row>
<item>31</item>
<item>32</item>
<item>33</item>
<item>34</item>
</row>
</rows>
</JuggedArray>
```

Преимущества шаблона Builder

1. клиентский код не зависит от конкретного строителя;
2. есть возможность определить произвольное число строителей без модификации классов `JuggedArray` и `JuggedArrayConverter`.

1.3. Паттерны Singleton (Одиночка) и Monostate

(Моносостояние)

- Обычно между классами и их объектами существует отношение один-ко-многим,
 - можно создавать много экземпляров одного класса.
- Объекты создаются, когда они требуются, и уничтожаются, когда перестают быть необходимыми.
- Однако у некоторых классов должен быть только один объект.
 - создается в начале работы программы
 - уничтожен при ее завершении.
- Такие объекты могут быть:
 - **корневыми объектами** приложения – из них можно добраться до других объектов системы.
 - **фабриками**, порождающими другие объекты.
 - **менеджерами**, которые следят за другими объектами.

- Во всех случаях требуется только один такой объект
 - наличие нескольких их объектов – серьезная логическая ошибка.
- Если есть более одного корня, то доступ к объектам приложения может зависеть от выбранного корня.
 - Программисты, не знающие о наличии нескольких корней, могут, не сознавая того, видеть лишь подмножество всех объектов приложения.
- Если существует несколько фабрик, то может быть потерян контроль над созданными объектами.
- При наличии нескольких менеджеров операции, предполагавшиеся последовательными, могут

- Может показаться, что вводить специальные механизмы обеспечения единственности таких объектов – излишество.
- В конце концов, на этапе инициализации приложения можно просто создать по одному экземпляру каждого, и дело с концом.
 - Такой шаблон иногда называется «Просто создай одного».

Паттерн Одиночка (Singleton)

- Одиночка (Singleton) – очень простой паттерн.

```
public class Singleton {  
    private static Singleton theInstance = null;  
    private Singleton() {}  
    public static Singleton Instance {  
        get {  
            if (theInstance == null)  
                theInstance = new Singleton();  
            theInstance;  
        }  
    }  
}
```

Реализация класса Singleton

```
public class Singleton {  
    private static Singleton theInstance = null;  
    private Singleton() {}  
    public static Singleton Instance {  
        get {  
            if (theInstance == null)  
                theInstance = new Singleton();  
            theInstance;  
        }  
    }  
}
```

Пример класса

- Описание класса

```
public class Singleton{  
    private static Singleton instance;  
    private Singleton() { }  
    public static Singleton Instance {  
        get { if (instance == null) {instance = new Singleton(); }  
            return instance;  
        }  
    }  
    public String DataItem { get; set; }  
}
```

- Использование класса

```
Singleton single = Singleton.Instance;  
single.DataItem = "value";
```

Достоинства

- Применимость к любому классу.
 - Любой класс можно преобразовать в Одиночку, если сделать его конструкторы закрытыми и добавить соответствующие статические методы и переменную-член.
- Может быть создан путем наследования.
 - Имея некоторый класс, можно создать его подкласс, который будет Одиночкой.
- Отложенное вычисление.
 - Если Одиночка не используется, то он и не создается.

Недостатки

- Уничтожение не определено.
 - Не существует приемлемого способа уничтожить или «списать» Одиночку.
 - Даже если добавить метод, обнуляющий переменную `theInstance`, другие модули могут хранить у себя ссылку на Одиночку.
 - При последующих обращениях к `Instance` будет создан новый экземпляр, что приведет к образованию двух одновременно существующих экземпляров.
 - Эта проблема особенно остро стоит в языке C++, где экземпляр может быть уничтожен, что приведет к разыменованию уже не существующего объекта.
- Не наследуется.
 - Класс, производный от Одиночки, сам не является Одиночкой.
 - Если необходимо, чтобы он был Одиночкой, придется добавить статический метод и переменную-член.
- Эффективность.
 - Каждое обращение к свойству `Instance` приводит к выполнению предложения `if`.
 - Для большинства обращений это предложение бесполезно.
- Непрозрачность.
 - Пользователи Одиночки знают, с чем имеют дело, потому что вынуждены обращаться к свойству `Instance`.

Пример использования шаблона Одиночка

- Предположим, что имеется веб-приложение, позволяющее пользователям входить в защищенные области сервера.
- В таком приложении будет какая-то БД, которая содержит описания пользователей:
 - имена,
 - пароли
 - другие атрибуты.
- Доступ к базе данных выполняется с помощью специального API.

Варианты реализации

- Можно было бы в каждом модуле, которому необходимо читать и изменять данные о пользователях, обращаться к базе напрямую.
- Недостаток:
 - вызовы стороннего API будут разбросаны по всему коду,
 - это затрудняет следованию соглашения о доступе и структуре программы.

- Лучше воспользоваться паттерном **Facade** (Фасад) и создать класс **UserDatabase**, предоставляющий методы для чтения и изменения объектов **User**.
 - Это частный случай шаблона Фасад, называемый шаблоном Шлюз (Gateway).
- Методы обращаются к стороннему API доступа к базе данных, осуществляя отображение между объектами **User** и таблицами базы.
- Внутри класса **UserDatabase** можно обеспечить соглашения о структуре и порядке доступа.
 - Например, можно гарантировать, что не будет добавлена запись **User**, в которой поле **username** пусто.
 - Или упорядочить обращения к записи **User**, так, чтобы никакие два модуля не могли одновременно читать и изменять ее.

Решение на основе паттерна `Singleton` (Одиночка)

- Создается класс с именем `UserDatabaseSource`, который реализует интерфейс `UserDatabase`.
- В коде свойства `Instance` нет традиционного предложения `if`, защищающего от многократного создания.
- Вместо этого используется механизм статической инициализации, имеющийся в .NET.

Интерфейс `UserDatabase` и класс `UserDatabaseSource`

```
public interface UserDatabase {
    User ReadUser(string userName);
    void WriteUser(User user);
}

public class UserDatabaseSource : UserDatabase {
    private static UserDatabase theInstance = new
        UserDatabaseSource();
    public static UserDatabase Instance { get { return theInstance; } }
    private UserDatabaseSource() { }
    public User ReadUser(string userName) { // Реализация }
    public void WriteUser(User user) { // Реализация }
}
```

- Такое использование паттерна Одиночка распространено чрезвычайно широко.
- Гарантируется, что весь доступ к базе данных производится через единственный экземпляр `UserDatabaseSource`.
- При этом в `UserDatabaseSource` очень легко вставлять
 - различные проверки,
 - счетчики
 - блокировки,которые обеспечивают выполнение требуемых соглашений о порядке доступа и структуре кода.

Паттерн **Monostate** (Моносостояние)

- Паттерн Monostate (Моносостояние) предлагает другой способ обеспечения единственности.

```
public class Monostate {  
    private static int itsX;  
    public int X {  
        get { return itsX; }  
        set { itsX = value; }  
    }  
}
```

Тесты для проверки

```
using NUnit.Framework;
[TestFixture]
public class TestMonostate {
[Test]
public void TestInstance() {
    Monostate m =
        new Monostate();
    for (int x = 0; x < 10; x++) {
        m.X = x;
        Assert.AreEqual(x, m.X);
    }
}
```

```
[Test]
public void
    TestInstancesBehaveAsOne()
    {
        Monostate m1 =
            new Monostate();
        Monostate m2 =
            new Monostate();
        for (int x = 0; x < 10; x++) {
            m1.X = x;
            Assert.AreEqual(x, m2.X);
        }
    }
}
```

- Если заменить в этих тестах все предложения `new Monostate` на вызовами `Singleton.Instance`, то тесты все равно прошли бы успешно.
- Таким образом, тесты описывают поведение Одиночки, не налагая ограничения на единственность экземпляра!
- Каким образом два экземпляра могут вести себя так, будто это единственный объект?
- Да просто это означает, что у них одни и те же переменные-члены.
- А добиться этого можно, сделав все переменные-члены статическими.

Реализация класса Monostate

```
public class Monostate {  
    private static int itsX;  
    public int X {  
        get { return itsX; }  
        set { itsX = value; }  
    }  
}
```

Пример класса

- Описание класса

```
public class Singleton{  
    private static Singleton instance;  
    private Singleton() { }  
    public static Singleton Instance {  
        get { if (instance == null) {instance = new Singleton(); }  
            return instance;  
        }  
    }  
    public String DataItem { get; set; }  
}
```

- Использование класса

```
Singleton single = Singleton.Instance;  
single.DataItem = "value";
```

- Описание класса

```
public class Monostate {  
    private static string dataItem;  
    public string DataItem {  
        get { return dataItem; }  
        set { dataItem = value; }  
    }  
    public Monostate() { }  
}
```

- Использование класса

```
var single = new Monostate();  
single.DataItem = "value";
```

- Сколько бы экземпляров класса **Monostate** ни создать, все они ведут себя так, как будто являются одним и тем же объектом.
- Можно даже уничтожить все текущие экземпляры, не потеряв при этом данных.
- Различие между паттернами **Singleton** и **Monostate** – это различие между поведением и структурой.

- Паттерн **Singleton** навязывает структуру единственности, не позволяя создать более одного экземпляра.
- Паттерн **Monostate**, напротив, навязывает поведение единственности, не налагая структурных ограничений.
- Это различие станет понятным, если заметить,
 - тесты для паттерна **Monostate** проходят и для класса **Singleton**,
 - однако у класса **Monostate** нет ни малейшей надежды пройти тесты для **Singleton**.

Достоинства

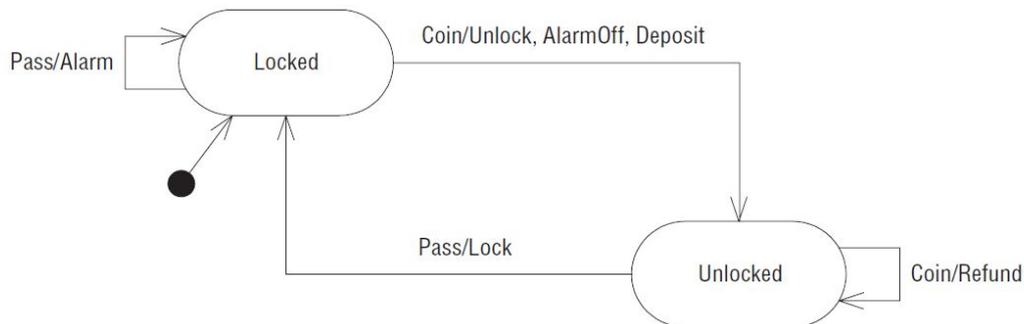
- **Прозрачность** – пользователь работает точно так же, как с обычным объектом, ничего не зная о том, что это «моносостояние».
- **Допускает наследование.**
 - Подклассы моносостояния также обладают этим свойством.
 - Более того, все его подклассы являются частями *одного и того же* моносостояния, так как разделяют одни и те же статические переменные-члены.
- **Полиморфизм**
 - Т.к. методы моносостояния не статические, их можно переопределять в производных классах.
 - Т.е. подклассы могут реализовывать различное поведение при одном и том же наборе статических переменных-членов.
- **Точно определенные моменты создания и уничтожения.**
 - Т.к. переменные-члены моносостояния статические, то моменты их создания и уничтожения точно определены.

Недостатки

- **Невозможность преобразования.**
 - Класс, не являющийся моносостоянием, невозможно превратить в моносостояние с помощью наследования.
- **Эффективность.**
 - Будучи настоящим объектом, моносостояние может многократно создаваться и уничтожаться.
 - Иногда это обходится дорого.
- **Постоянное присутствие.**
 - Переменные-члены моносостояния занимают место в памяти, даже если объект никогда не используется.
- **Локальность.**
 - Паттерн Моносостояние не может гарантировать единственность в нескольких экземплярах CLR или на нескольких компьютерах.

Пример использования паттерна Monostate

- Рассмотрим реализацию простого конечного автомата (КА), описывающего работу турникета в метро .



- Первоначально турникет находится в состоянии **Locked** (Закрыт).
- Если опустить монету, турникет перейдет в состояние **Unlocked**, откроет дверцы, сбросит сигнал тревоги (если он был включен) и поместит монету в монетоприемник.
- Если в этот момент пользователь пройдет через турникет, тот вернется в состояние **Locked** и закроет дверцы.

- Существуют два аномальных условия.
- Если пользователь опускает несколько монет, прежде чем пройти, то лишние монеты возвращаются, а дверцы остаются открытыми.
- Если пользователь пытается пройти, не заплатив, то раздается сигнал тревоги и дверцы остаются закрытыми.

Реализация моносостояния

Turnstile

```
public class Turnstile {
    private static bool isLocked = true;
    private static bool isAlarming = false;
    private static int itsCoins = 0;
    private static int itsRefunds = 0;
    protected static readonly
        Turnstile LOCKED = new Locked();
    protected static readonly
        Turnstile UNLOCKED = new Unlocked();
    protected static Turnstile itsState = LOCKED;
    public void reset() {
        Lock(true);
        Alarm(false);
        itsCoins = 0;
        itsRefunds = 0;
        itsState = LOCKED;
    }
    public bool Locked() { return isLocked; }
    public bool Alarm() { return isAlarming; }
    public virtual void Coin() { itsState.Coin(); }
    public virtual void Pass() { itsState.Pass(); }
    protected void Lock(bool shouldLock) {
        isLocked = shouldLock; }
    protected void Alarm(bool shouldAlarm) {
        isAlarming = shouldAlarm; }
    public int Coins { get { return itsCoins; } }
```

```
        public int Refunds { get { return itsRefunds; } }
        public void Deposit() { itsCoins++; }
        public void Refund() { itsRefunds++; }
    }
```

```
internal class Locked : Turnstile {
    public override void Coin() {
        itsState = UNLOCKED;
        Lock(false);
        Alarm(false);
        Deposit();
    }
    public override void Pass() { Alarm(true); }
}
```

```
internal class Unlocked : Turnstile {
    public override void Coin() { Refund(); }
    public override void Pass() {
        Lock(true);
        itsState = LOCKED;
    }
}
```


Полезные особенности паттерна **Monostate** (Моносостояние)

- Используются
 - возможность создавать полиморфные подклассы и
 - то, что подклассы сами являются моносостояниями.
- Кроме того, видно, насколько трудно бывает превратить объект-моносостояние в объект, таковым не являющийся.
- Структура решения существенно опирается на то, что **Turnstile** – моносостояние.
- Если бы потребовалось применить этот КА к управлению несколькими турникетами, код придется сильно переработать.

Полезные особенности паттерна **Monostate** (Моносостояние)

- Может возникнуть вопрос в связи с необычным использованием наследования в этом примере.
- То, что классы **Unlocked** и **Locked** сделаны производными от **Turnstile**, представляется нарушением принципов ООП.
- Но поскольку **Turnstile** – моносостояние, то не существует его отдельных экземпляров.
- Поэтому **Unlocked** и **Locked** – это не самостоятельные классы, а части абстракции **Turnstile**.
- Они имеют доступ к тем же переменным и методам, что и **Turnstile**.

Выводы по шаблонам Singleton и Monostate

- Часто бывает необходимо обеспечить единственность объекта некоторого класса.
- Шаблоны Singleton и Monostate принципиально различным способом решают эту задачу.
- Паттерн Singleton опирается на использование
 - закрытых конструкторов,
 - статической переменной-члена и статического метода,которые совместно ограничивают количество создаваемых экземпляров.
- В паттерне Monostate все переменные-члены просто сделаны статическими.

Выводы по шаблонам Singleton и Monostate (2)

- Шаблон **Singleton** лучше применять, когда уже есть некоторый класс и тогда обеспечить единственность экземпляра можно, создав его подкласс,
 - если не против обращения к свойству **Instance** для получения доступа к этому экземпляру.
- Шаблон **Monostate** удобнее
 - когда единичный объект класса желательно сделать незаметным для пользователей или
 - когда необходимо полиморфное поведение единственного объекта.

Паттерн Null-объект

