

# Паттерны проектирования (Design patterns)

# Предлагается следующая общая классификация паттернов по категориям их применения:

- *Архитектурные паттерны*
- ***Паттерны проектирования***
- *Паттерны анализа*
- *Паттерны тестирования*
- *Паттерны реализации*

# Признаки плохого кода

- дублирование кода;
- длинный метод;
- большой класс;
- длинный список параметров;
- классы данных;
- несгруппированные данные.

# Причины возникновения плохого кода

- Частые изменения в требованиях, противоречащие исходной архитектуре;
- недостаточно времени сделать работу качественно;
- глупый менеджер/начальник/заказчик и т.д.
- Ваш вариант.

# **Настоящие причины возникновения плохого кода**

- **Непрофессионализм**
- **Лень**

# **Закон Леблана**

**«Потом равносильно никогда»**

# ЧИСТЫЙ КОД

**«Я люблю, чтобы мой код был элегантным и эффективным. Логика должна быть достаточно прямолинейной, чтобы ошибкам было трудно спрятаться; зависимости – минимальными, чтобы упростить сопровождение; обработка ошибок – полной, в соответствии с выбранной стратегией; а производительность – близкой к оптимальной, чтобы не искушать людей загрязнять код беспринципными оптимизациями.»**

**Бьерн Страуструп  
автор языка C++**

# Приемы чистого кода

- Именованние переменных;
- правильная работа с функциями;
- комментирование кода;
- форматирование;
- обработка ошибок;
- тестирование.

# **Объектно-ориентированное проектирование**

**Проектирование объектно-ориентированных программ – нелегкое дело, а если их нужно использовать повторно, то все становится еще сложнее.**

**Эрих Гамма  
Автор книги Design Patterns**

# Правильный дизайн

Правильный дизайн – это гибкий и пригодный для повторного использования дизайн. Он должен, с одной стороны, соответствовать решаемой задаче, с другой — быть общим, чтобы учесть все требования, которые могут возникнуть в будущем.

# Паттерны проектирования

Паттерн проектирования — повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования в рамках некоторого часто возникающего контекста.

# Что такое паттерны (шаблоны) проектирования?

- Эффективные способы решения характерных задач проектирования
- Обобщенное описание решения задачи, которое можно использовать в различных ситуациях
- OO 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.

# Шаблоны проектирования

- В проектировании ПО часто встречаются проблемы, которые уже решались ранее в других проектах.
- В связи с тем, что контексты, в которых данная проблема решалась, могут различаться
  - (другой тип приложения, другая платформа или другой язык программирования),
  - все обычно заканчивается повторением проектирования и реализации данного решения,
  - тем самым возникает ситуация «повторного изобретения колеса».

- В этом случае разработчикам могут помочь, программные шаблоны, включая
  - архитектурные шаблоны и
  - шаблоны проектирования.
- Они позволяют избежать ненужного повторного проектирования и реализации.

- Понятие шаблона (pattern) впервые было предложено Christopher Alexander для разработки архитектуры зданий и описаны в его книге
  - «*The Timeless Way of Building*» (Alexander 1979).
- «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» (Александр Кристофер, архитектор).
- Такое определение верно и в отношении паттернов объектно-ориентированного проектирования.

# Понятие паттерна (шаблона)

- При ООП решения описываются в терминах объектов и интерфейсов, а не стен и дверей, но в обоих случаях смысл паттерна - предложить решение определенной задачи в конкретном контексте.
- Под **паттернами ОО проектирования** понимается описание взаимодействия объектов и классов, адаптированных для решения общей задачи проектирования в конкретном контексте.
- В области ПО использование шаблонов проектирования было предложено и развито Gamma, Helms, Johnson и Vlissides в их книге «*Design Patterns (1995)*», в которой они описали 23 шаблона проектирования.

# Классификация паттернов

- Порождающие (Creational)
- Структурные (Structural)
- Поведенческие (Behavioral)

## **1. Порождающие шаблоны**

- 1. Abstract Factory**
- 2. Builder**
- 3. Factory Method**
- 4. Prototype**
- 5. Singleton**

## **2. Структурные шаблоны**

- 1. Adapter**
- 2. Bridge**
- 3. Composite**
- 4. Decorator**
- 5. Façade**
- 6. Flyweight**
- 7. Proxy**

## **3. Шаблоны поведения**

- 1. Chain of Responsibility**
- 2. Command**
- 3. Interpreter**
- 4. Iterator**
- 5. Mediator**
- 6. Momento**
- 7. Observer**
- 8. State**
- 9. Strategy**
- 10. Template Method**
- 11. Visitor**

# 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.

# 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.

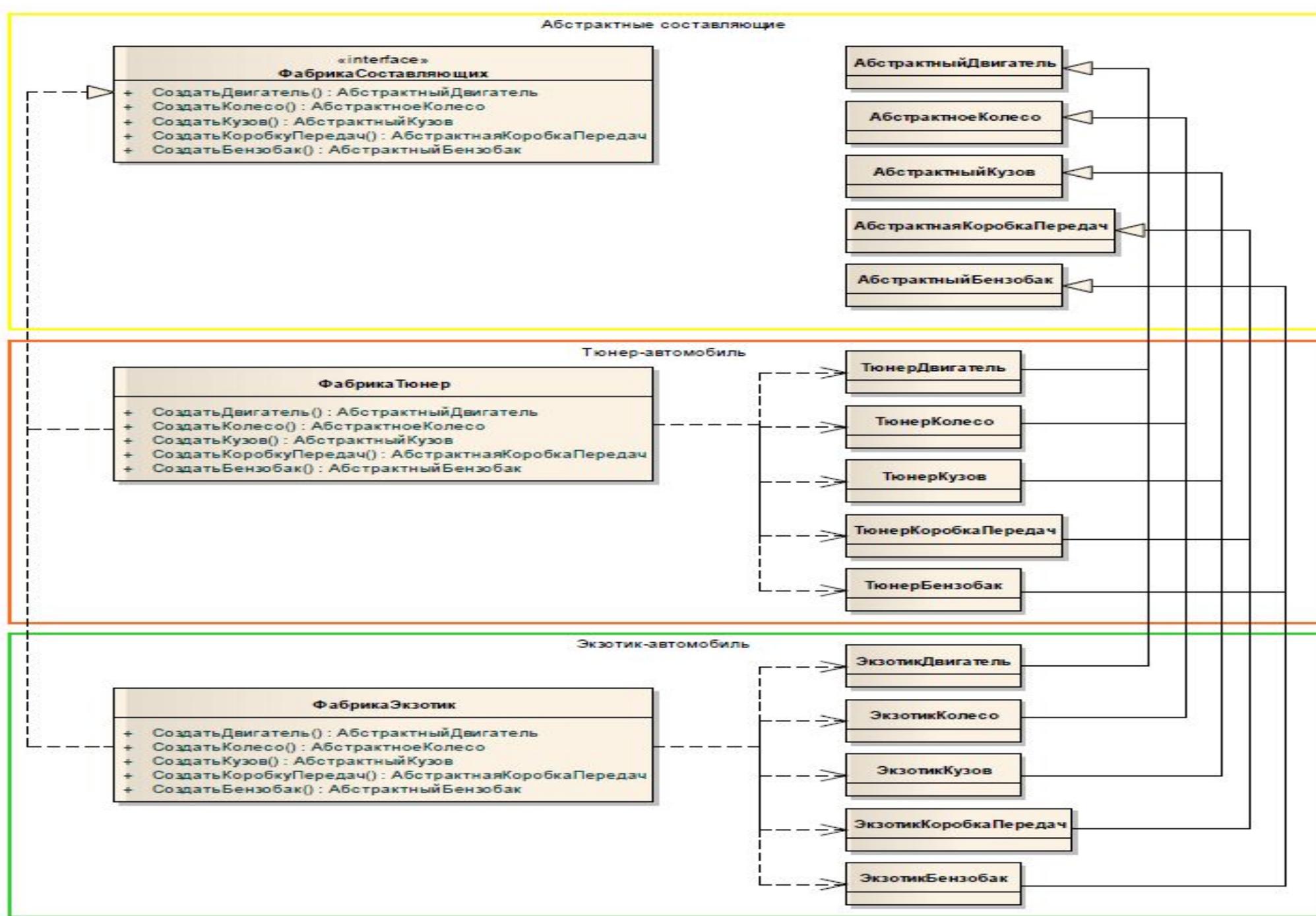
# Пример – игра «Супер Ралли» (гонка на автомобилях)

- 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 выбранного автомобиля.**

# Предлагаемая реализации

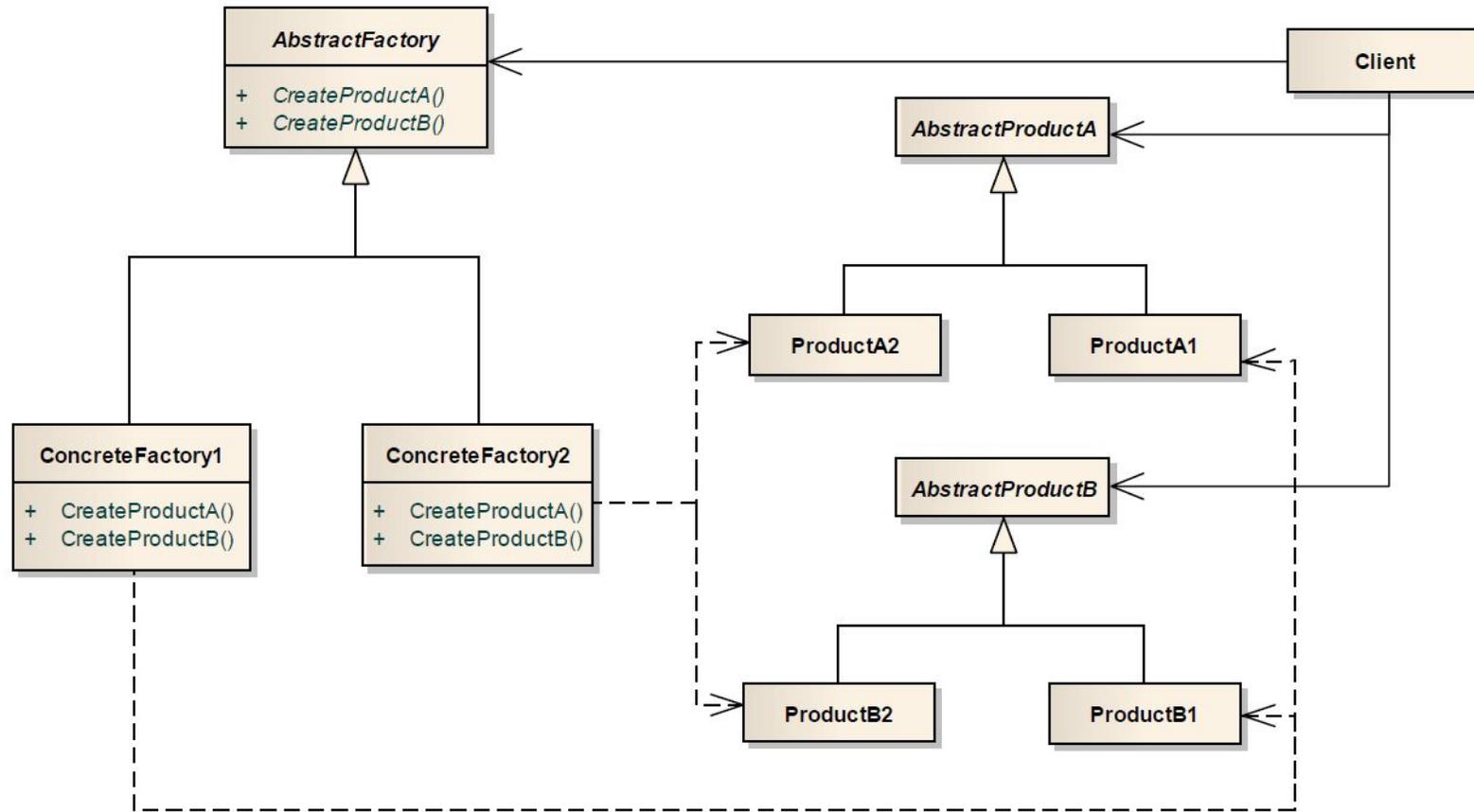
- Создается интерфейс **ФабрикаСоставляющих** – предназначен для создания конкретных классов (фабрик), которые будут создавать семейства составляющих для каждого конкретного типа автомобиля.
- Методы этого класса должны возвращать ссылки на абстрактные составляющие, что позволит в конкретных классах-фабриках,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

- 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 это 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

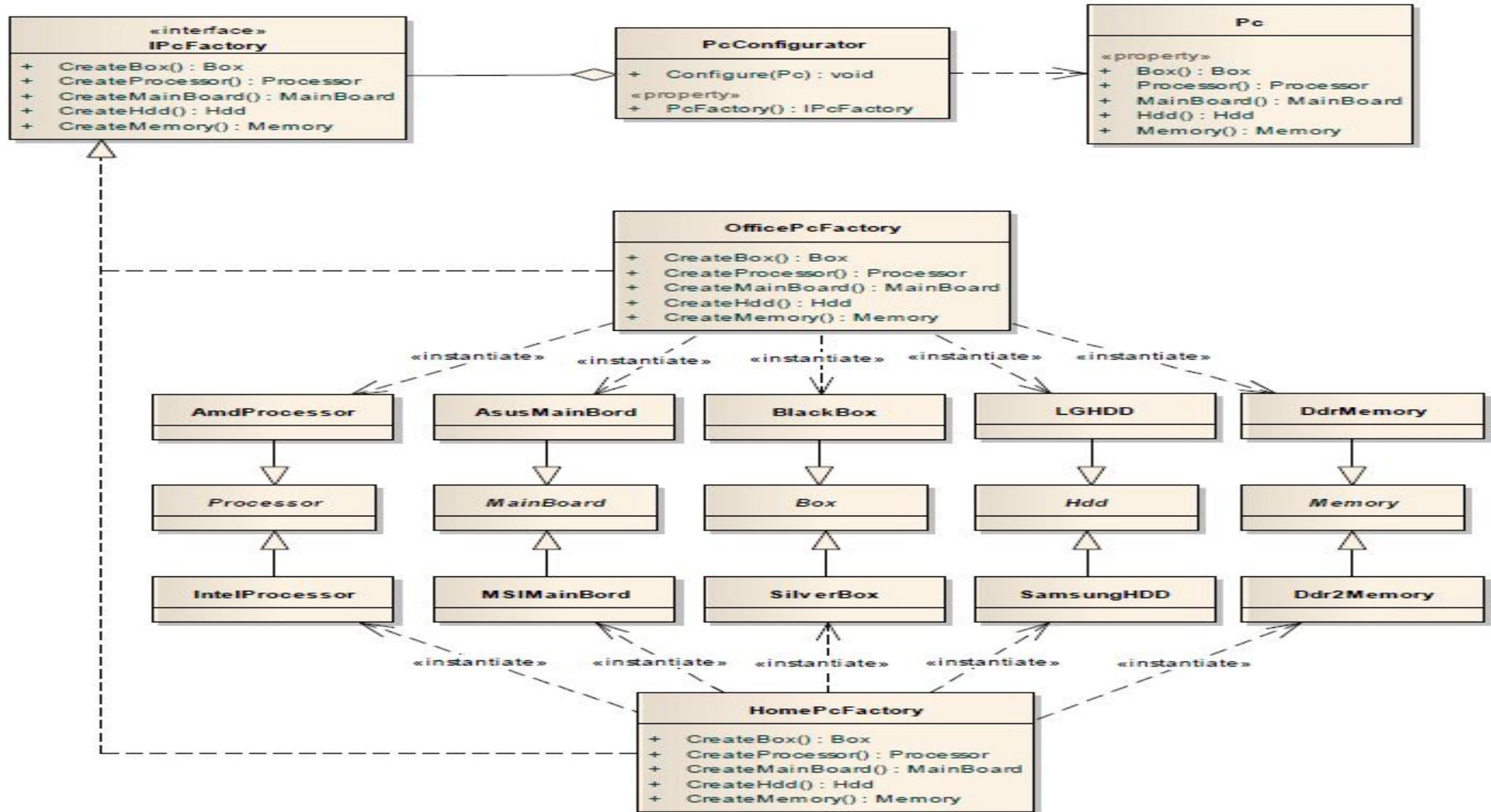
```
public class OfficePcFactory : IPcFactory
{
    public Box CreateBox()
    {return new BlackBoxf); }
    public Processor CreateProcessor()
    { return new AmdProcessor();}
    public MainBoard CreateMainBoard()
    {return new AsusMainBord(); }
    public Hdd CreateHdd() {return new LGHDD ();}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

# 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.

# **Вывод**

**Применяйте паттерны  
проектирования**

# **Заблуждение №1**

**Паттерны гарантируют возможность  
повторного использования  
программного обеспечения, повышение  
производительности, отсутствие  
разногласий и т.д.**

# **Заблуждение №2**

**Паттерны предоставляют  
готовые архитектурные  
решения**

# **Преимущества паттернов проектирования**

- **Использование предыдущего опыта экспертов.**
- **Улучшение взаимопонимания разработчиков.**
- **Альтернатива документации приложений.**
- **Упрощение реструктуризации системы.**