NET.C#.04
Design Patterns.

What is a Design Pattern ?

Christopher Alexander says,

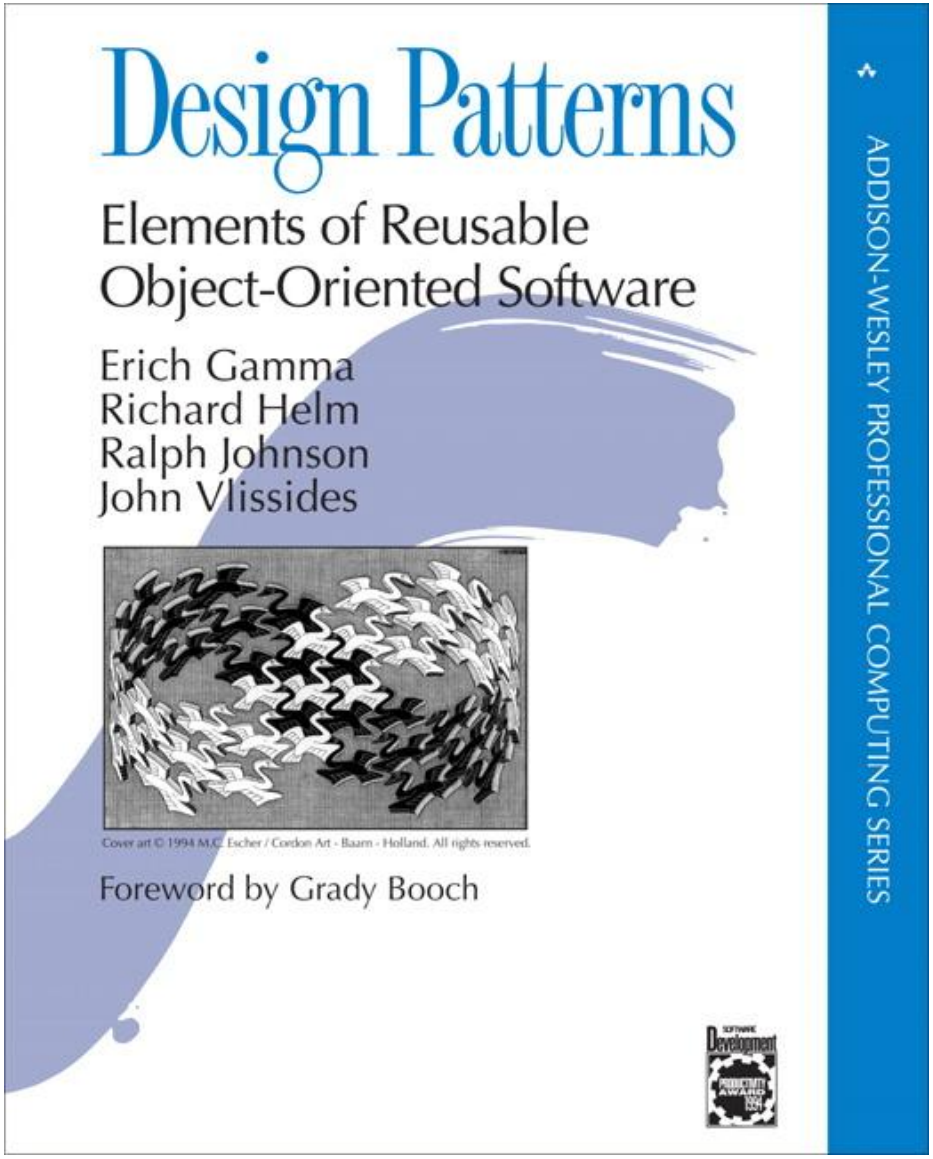
"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

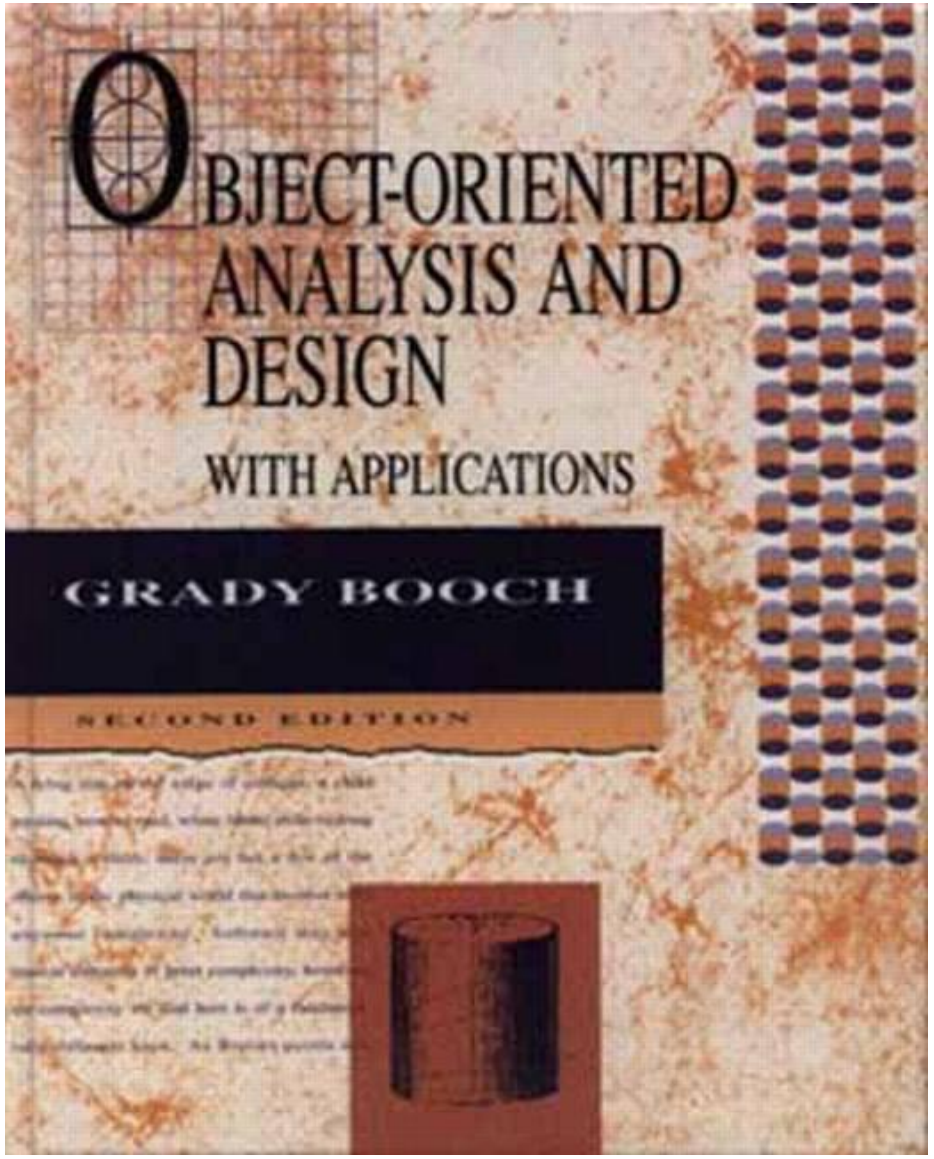
In software engineering,

a design pattern is a general reusable solution to a commonly occurring problem. A design pattern is not a finished design that can be transformed directly into code. It is a description or template for how to solve a problem that can be used in many different situations.

Object-oriented design patterns show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved.

Must have… and must read…





Desing patterns classification

Creational patterns

Structural patterns

Behavioral patterns

Creational patterns concern the process of object creation.

Structural patterns deal with the composition of classes or objects.

Behavioral patterns characterize the ways in which classes or objects interact and distribute responsobility

Abstract Factory

Creational pattern

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Participants

The classes and/or objects participating in this pattern are:

- **Abstract Factory**
- **Concrete Factory**
- **Abstract Product**
- **Product**
- **Client**

Abstract Factory: participants

AbstractFactory - declares an interface for operations that create abstract products

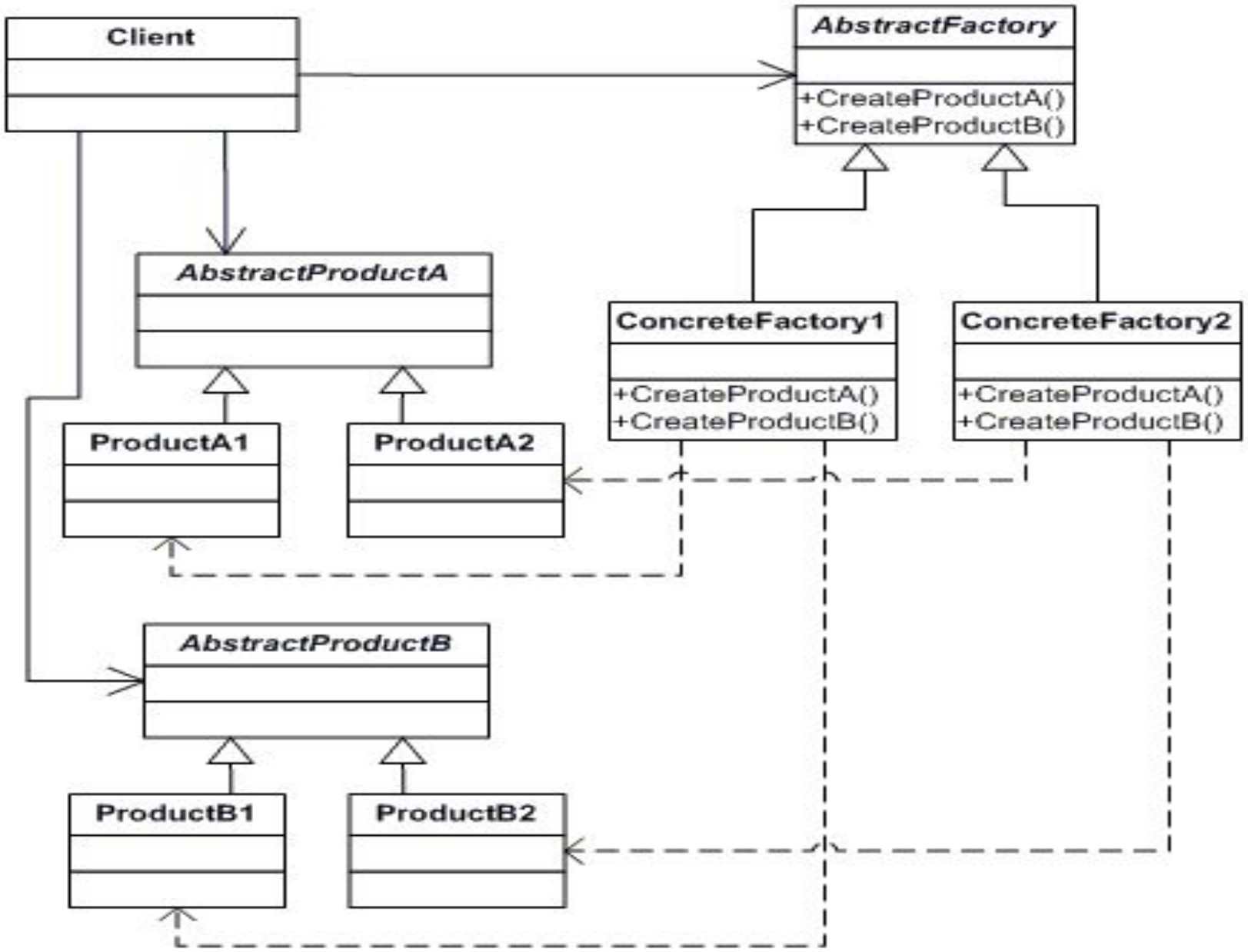
ConcreteFactory - implements the operations to create concrete product objects

AbstractProduct - declares an interface for a type of product object

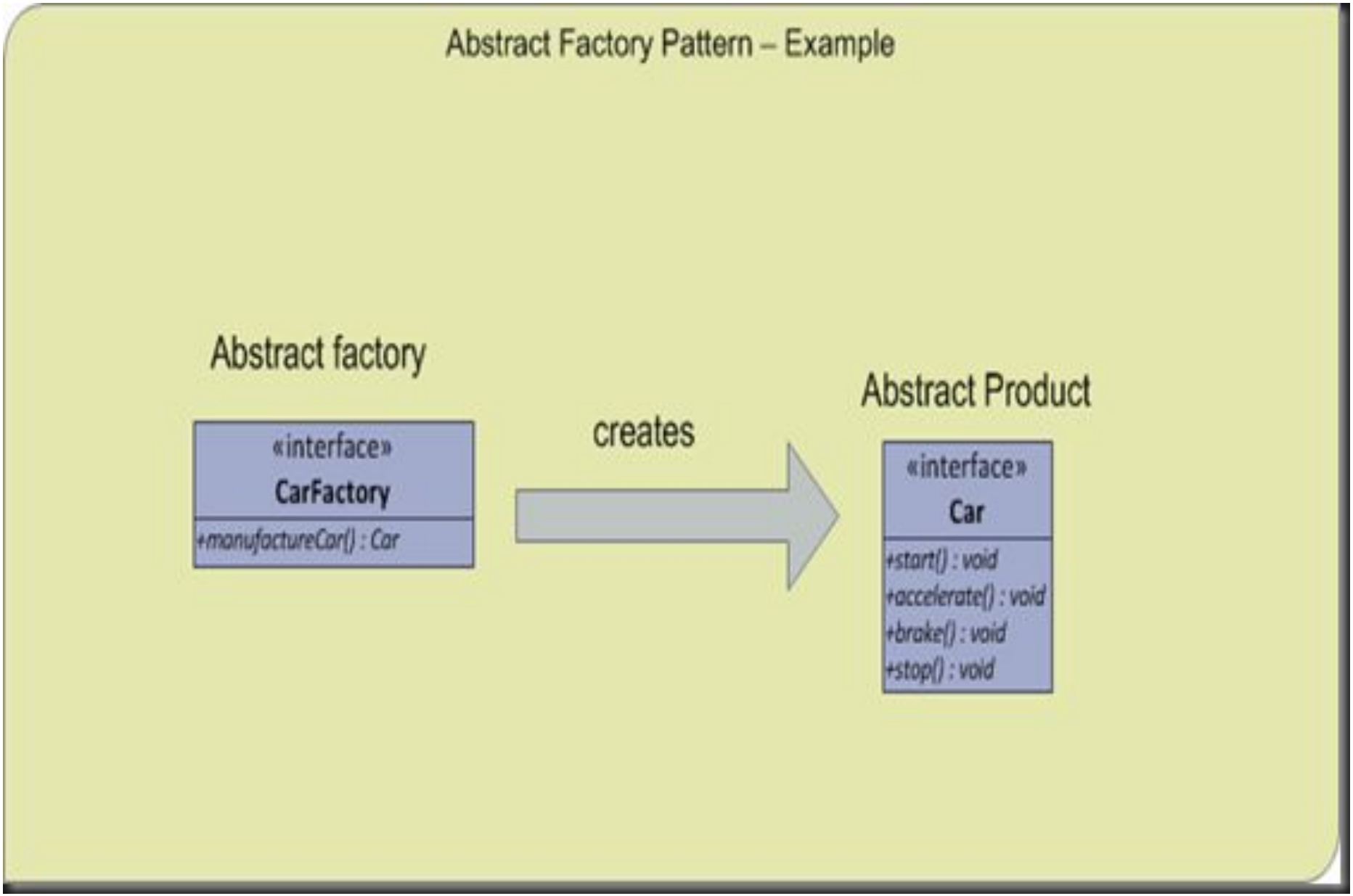
Product - defines a product object to be created by the corresponding concrete factory implements the AbstractProduct interface

Client uses interfaces declared by AbstractFactory and AbstractProduct classes

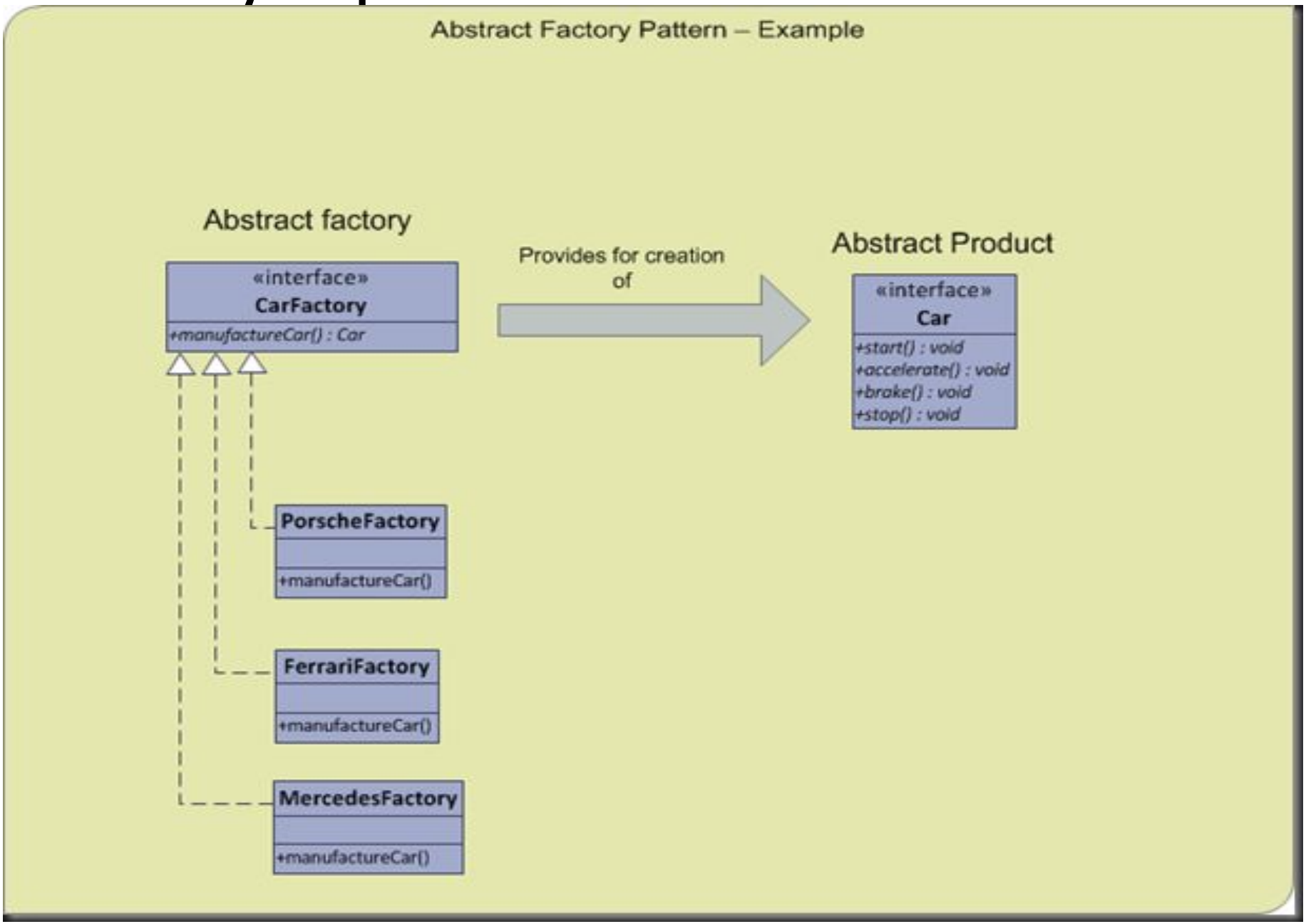
Abstract Factory: UML class diagram



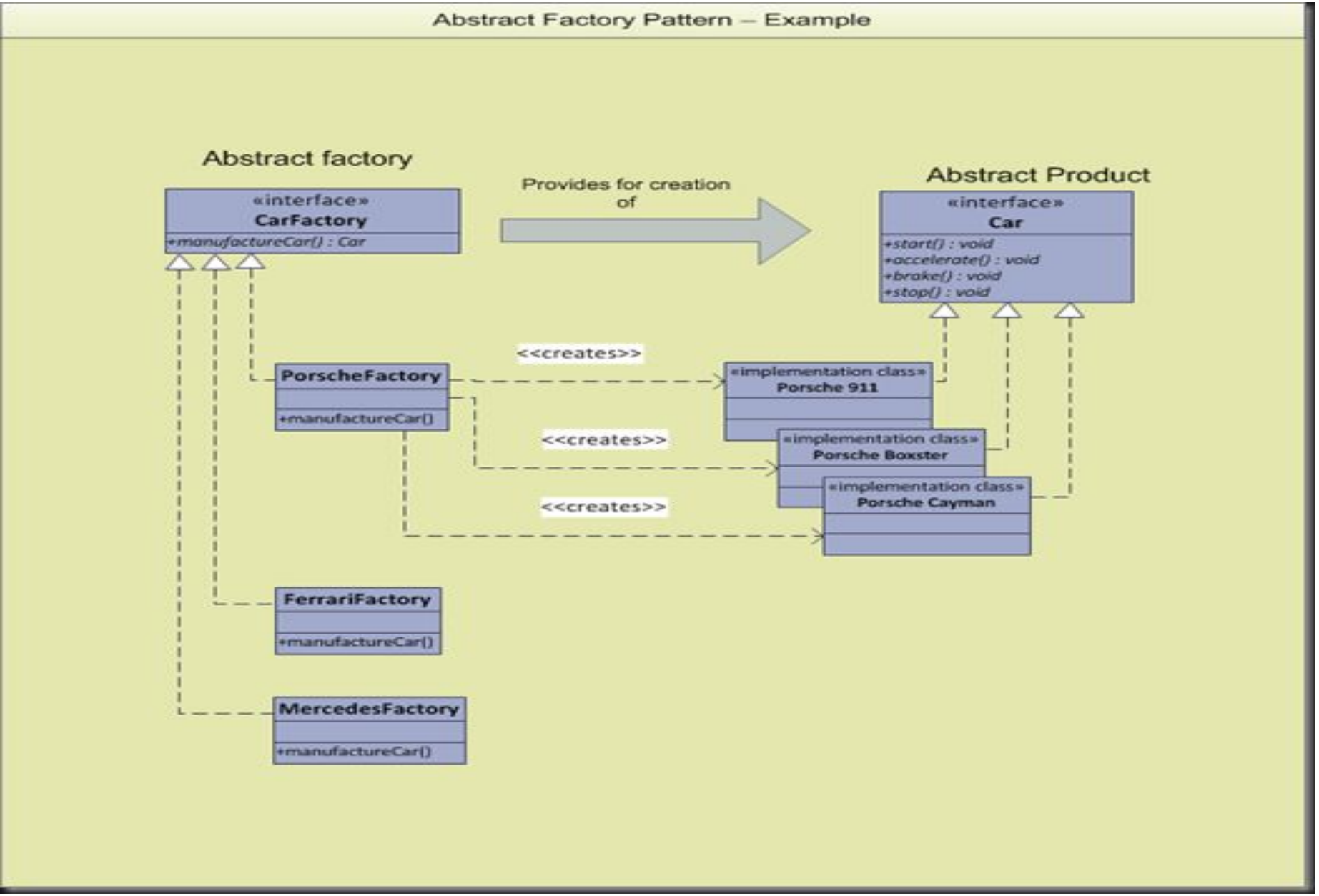
CarFactory. Step 1



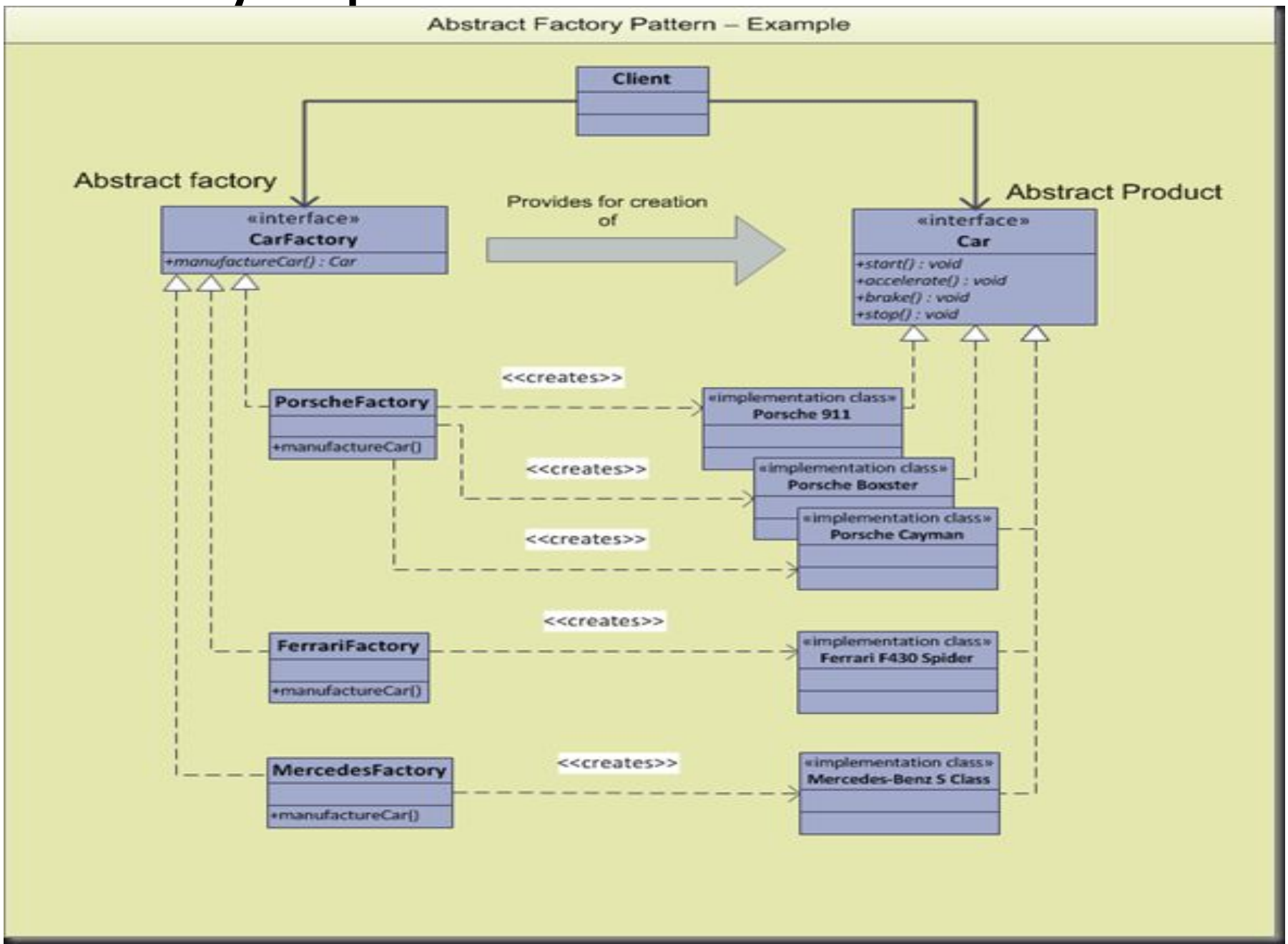
Car Factory. Step 2



Car Factory. Step 3



Car Factory. Step 4



Abstract Factory: example.

Abstract Factory - CarFactory

```
abstract class CarFactory
  {
    public abstract AbstractCar CreateCar();
    public abstract AbstractEngine CreateEngine();
  }
```

Abstract Factory->Concrete Factory

Concrete Factory - BMWFactory

```
class BMWFactory : CarFactory
  {
    public override AbstractCar CreateCar()
    {
      return new BMWCar();
    }
public override AbstractEngine CreateEngine()
    {
     return new BMWEngine();
    }
  }
```

Abstract Factory->Concrete Factory

Concrete Factory - AudiFactory

```
  class AudiFactory : CarFactory
  {
    public override AbstractCar CreateCar()
    {
     return new AudiCar();
    }
public override AbstractEngine CreateEngine()
    {
     return new AudiEngine();
    }
  }
```

Abstract Factory->Abstract Product

Abstract Product - AbsrtactCar

```
abstract class AbstractCar
{
  public abstract void MaxSpeed(AbstractEngine engine);
}
```

Abstract Product - AbsrtactEngine

```
abstract class AbstractEngine
 {
    public int max_speed;
 }
```

Abstract Product->Concrete Product

Class implementation - BMWCar

```
class BMWCar : AbstractCar
{
 public override void MaxSpeed(AbstractEngine engine)
  {
    Console.WriteLine(«Max speed:« +
engine.max_speed.ToString());
  }
}
```

Class implementation - BMWEngine

```
class BMWEngine : AbstractEngine
{
  public BMWEngine()
{
   max_speed = 200;
}
}
```

Abstract Product->Concrete Product

Class implementation - AudiCar

```
  class AudiCar : AbstractCar
  {
     public override void MaxSpeed(AbstractEngine engine)
      {
     Console.WriteLine(«Максимальная скорость: « +
engine.max_speed.ToString());
      }
  }
//Concrete Engine
class AudiEngine : AbstractEngine
  {
  public AudiEngine()
  {
    max_speed = 180;
  }
  }
```

Abstract Factory. Пример

Class implementation – Client, works with abstract factory

```
  class Client
{
  private AbstractCar abstractCar;
  private AbstractEngine abstractEngine;
  public Client(CarFactory car_factory)
  {
    abstractCar = car_factory.CreateCar();
    abstractEngine = car_factory.CreateEngine ();
  }
  public void Run()
  {
     abstractCar.MaxSpeed(abstractEngine);
   }
 }
```

Abstract Factory.

```
public static void Main()
 {
   // Abstract Factory № 1
   CarFactory bmw_car = new BMWFactory ();
   Client c1 = new Client(bmw_car);
   c1.Run();
  // Abstract Factory № 2
   CarFactory audi_factory = new AudiFactory();
   Client c2 = new Client(audi_factory);
   c2.Run();
   Console.Read();
  }
```

Builder Pattern

Creational pattern

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

Participants

The classes and/or objects participating in this pattern are:

- **Builder**
- **Concrete Builder**
- **Director**
- **Product**

Builder: participants

Builder - specifies an abstract interface for creating parts of a Product object

ConcreteBuilder -

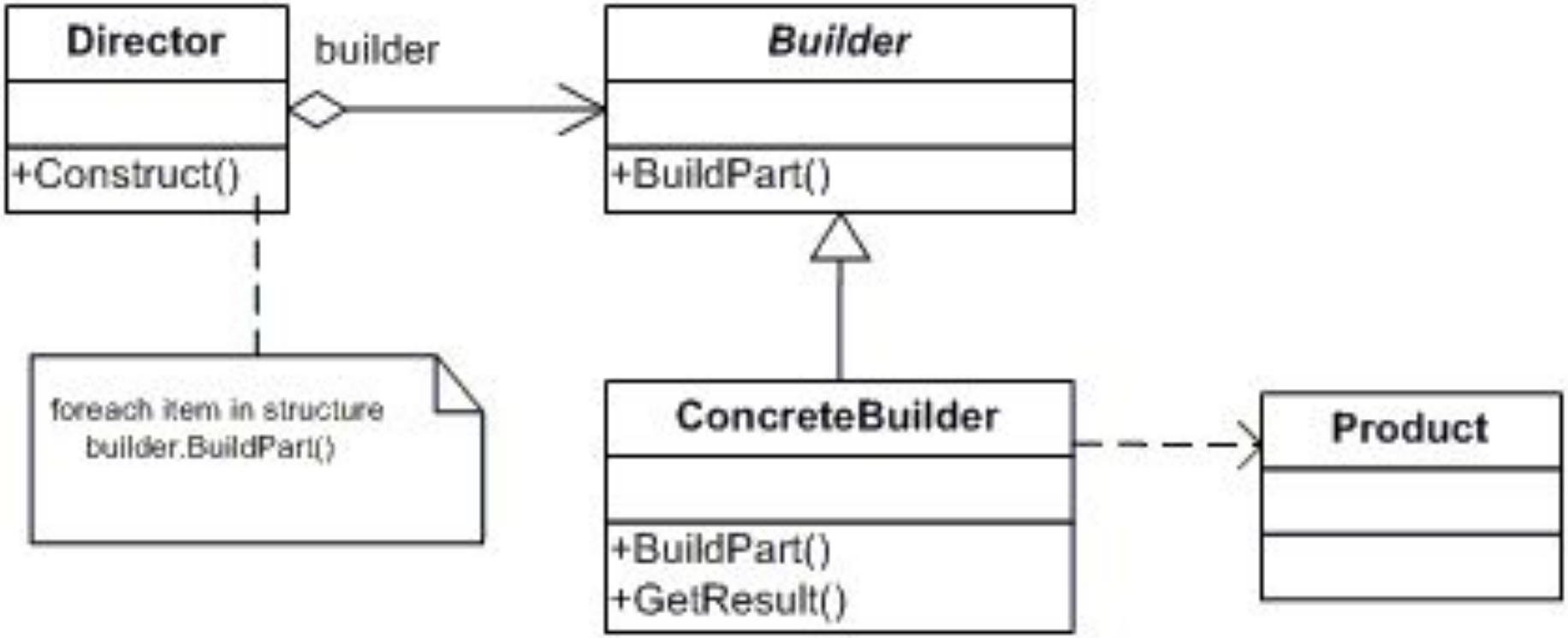
- constructs and assembles parts of the product by implementing the Builder interface
- defines and keeps track of the representation it creates
- provides an interface for retrieving the product

Director - constructs an object using the Builder interface

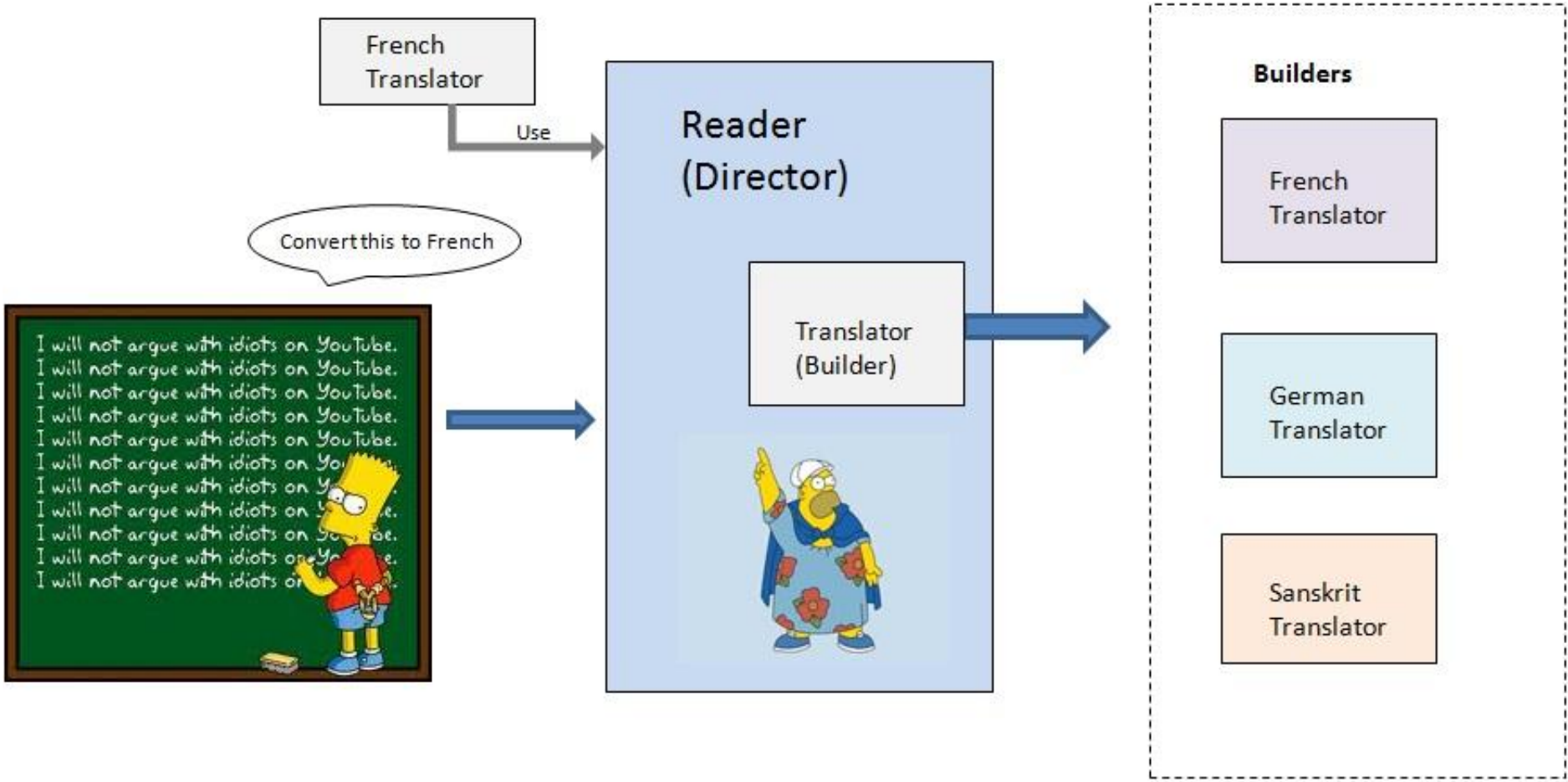
Product

- represents the complex object under construction ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled
- includes classes that define the constituent parts, including interfaces for assembling the parts into the final result

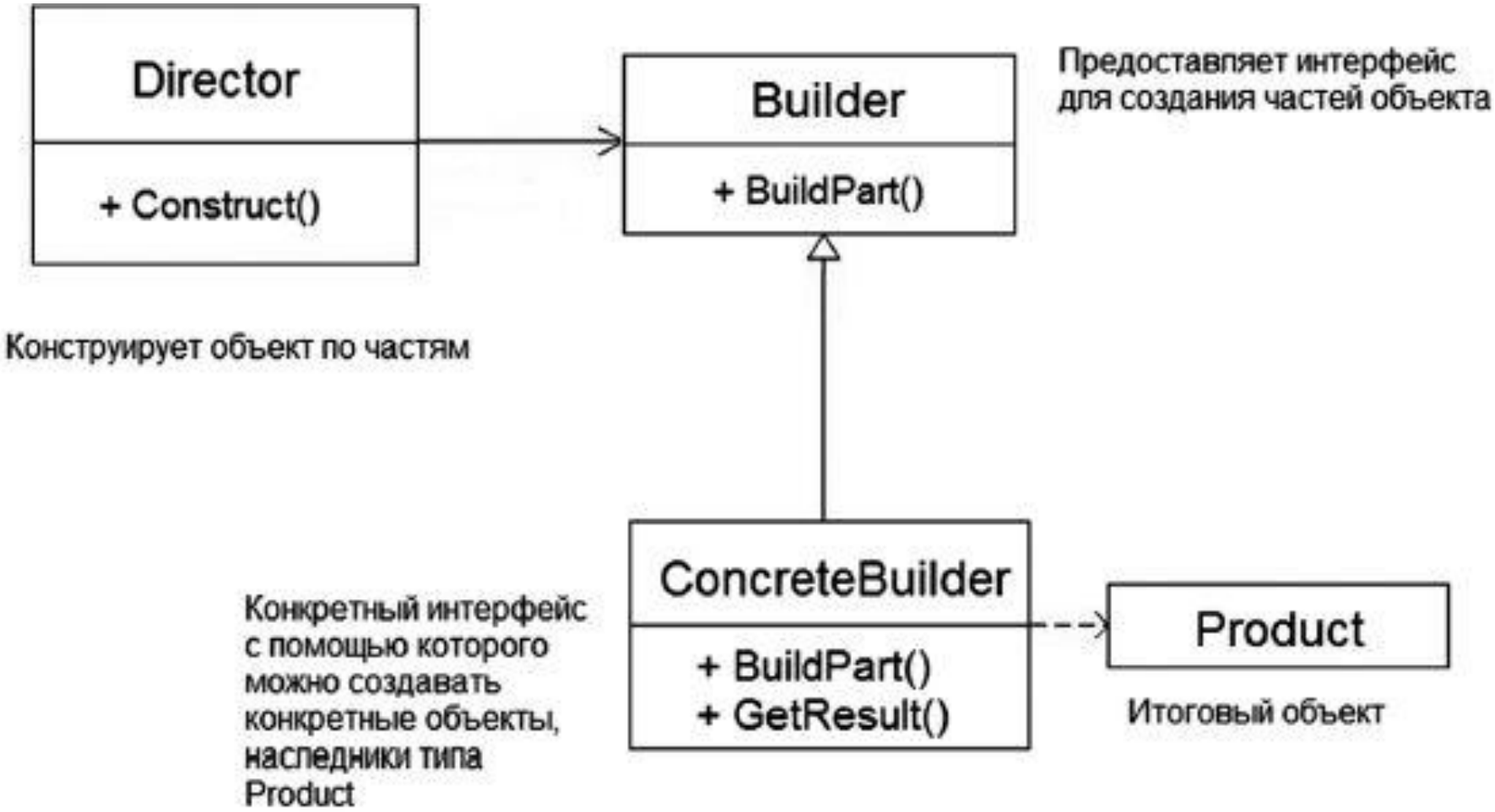
Builder: UML class diagram



Builder


French Translator
Use
Reader (Director)
Builders
French Translator
Convert this to French
Translator (Builder)
German Translator
Sanskrit Translator
I will not argue with idiots on YouTube.

Builder. UML


Director
+ Construct()
Builder
+ BuildPart()
Предоставляет интерфейс для создания частей объекта
Конструирует объект по частям
ConcreteBuilder
+ BuildPart()
+ GetResult()
Конкретный интерфейс с помощью которого можно создавать конкретные объекты, наследники типа Product
Product
Итоговый объект

Builder: Example

HappyMeal, BigHappyMeal

```
class HappyMeal
 {
   // contains information about parts of HappyMeal
   ArrayList parts = new ArrayList();
   // adds a new part
   public void Add(string part)
   {
     parts.Add(part);
   }
   // Shows information about HappyMeal
   public void Show()
   {
     Console.WriteLine(" Happy Meal Parts ——-");
     foreach (string part in parts)
     Console.WriteLine (part);
   }
 }
```

Builder. Example

Declare a builder - an abstract interface for creating an object in parts

```
abstract class HappyMealBuilder
 {
   public abstract void BuildBurger();
   public abstract void BuildPepsi();
   public abstract void BuildFries();
   public abstract void BuildToy();
   public abstract HappyMeal GetProduct();
 }
```

Builder

Declare a concrete builder BigHappyMeal

```
{  class BigHappyMealBuilder : HappyMealBuilder
  {
    private HappyMeal happy_meal = new HappyMeal();
    public override void BuildBurger()
    { happy_meal.Add("BigMac"); }
    public override void BuildPepsi()
    { happy_meal.Add("Pepsi 0.7"); }
    public override void BuildFries()
    { happy_meal.Add("BigFries");  }
    public override void BuildToy()
    { happy_meal.Add("Two toys");     }
    public override HappyMeal GetProduct()
    {
      return happy_meal;
    }
  }
```

Builder

Declare a concrete builder HappyMeal

```
class SmallHappyMealBuilder : HappyMealBuilder
{
   private HappyMeal happy_meal = new HappyMeal();
   public override void BuildBurger()
  { happy_meal.Add("Hamburger");  }
   public override void BuildPepsi()
  { happy_meal.Add("Pepsi 0.3");  }
   public override void BuildFries()
  { happy_meal.Add("SmallFries"); }
   public override void BuildToy()
  { happy_meal.Add("One toy");  }
   public override HappyMeal GetProduct()
  {
    return happy_meal;
   }
 }
```

Builder

Class Director – will construcr the object

```
class Director
{
  // Constructing the object in parts
  public void Construct(HappyMealBuilder builder)
 {
    builder.BuildBurger(); //Call Build-methods
    builder.BuildPepsi();
    builder.BuildFries();
    builder.BuildToy();
  }
}
```

Builder

```
public static void Main()
 {
  // Create a Director and builders
  Director director = new Director();
  HappyMealBuilder big_hmbuilder = new BigHappyMealBuilder();
  HappyMealBuilder small_hmbuilder = new SmallHappyMealBuilder();
  // Construct two products
  director.Construct(big_hmbuilder);
  HappyMeal hm1 = big_hmbuilder.GetProduct();
  hm1.Show();
  director.Construct(small_hmbuilder);
  HappyMeal hm2 = small_hmbuilder.GetProduct();
  hm2.Show();
 }
```