Factory and Facade

Advanced Java Programming 2012-13

OO Design Patterns and Principles

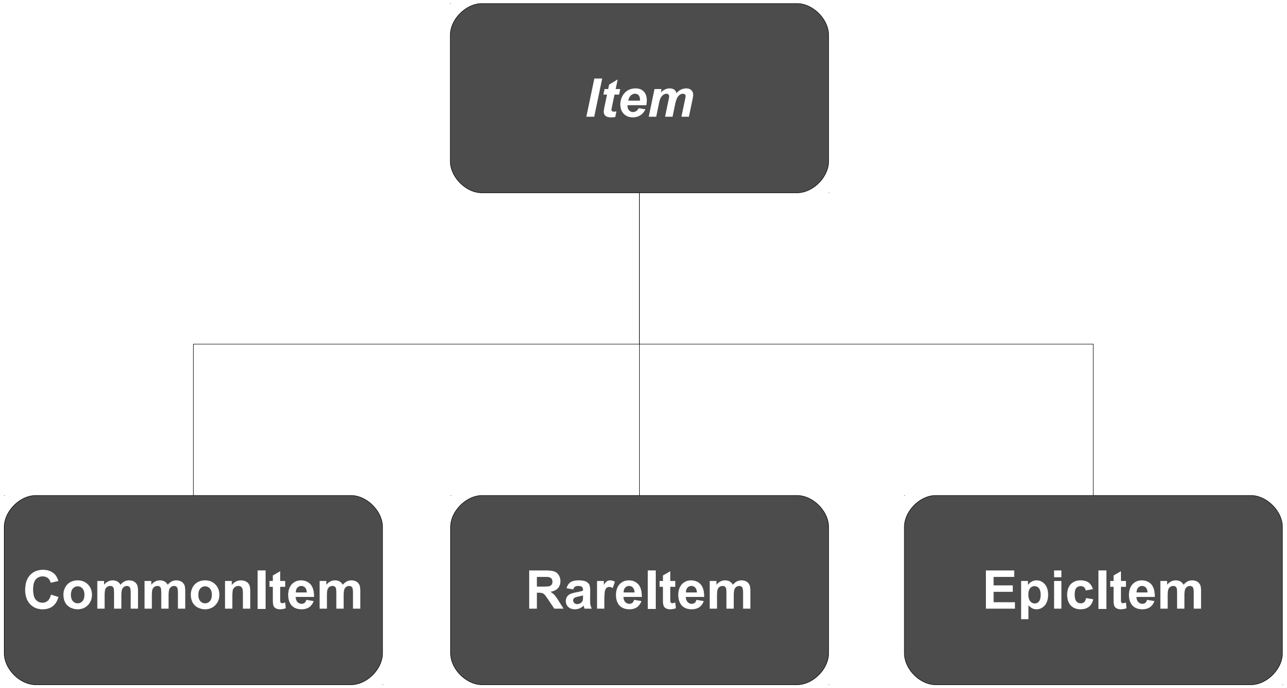
Lecture Outline

- The simple factory
- The *Factory Method* pattern
- The *Facade* pattern

The simple factory

- Imagine you are writing code for a MMORPG which works out the type of item which is dropped when an enemy dies
- There are three types of item, all sub-classes of *Item*
  - *CommonItem*
  - *RareItem*
  - *EpicItem*
- The type of item the user actually gets is related to several factors, which are only known at run time e.g.
  - The level of the user's character
  - The number and level of party members
  - The level of the enemy enemy character
  - Environmental factors e.g. game time

The simple factory



The type of object we instantiate is determined by an algorithm that uses run time factors. For example, a solo low level character who defeats a high level enemy might receive an epic item. A high level character that defeats a minor creature might get a common item. For the purposes of this example, the algorithm that does this is called *rateDifficulty*()

The simple factory

```
// We end up with code that looks a bit like this
Difficulty difficulty = rateDifficulty();
Item i;

switch(difficulty)
{
   case EASY:          i = new CommonItem();
                       break;
   case AVERAGE:       i = new RareItem();
                       break;
   case HARD:          i = new EpicItem();
                        break;
}

// Add i to inventory
```

Algorithm rates difficulty of kill expressed as enum.

This switch statement creates an appropriate reward

The simple factory

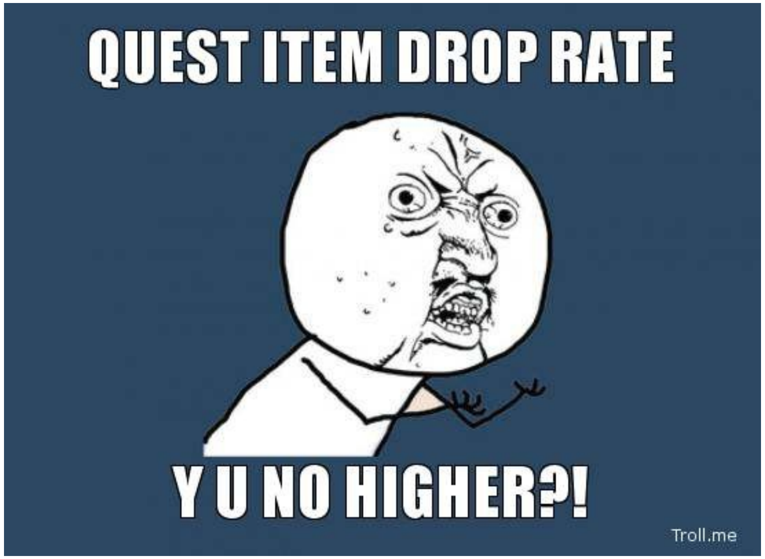
```
...
public Difficulty rateDifficulty()
{
     // work out difficulty of kill
}
```

This method calculates the difficulty of a particular kill given a range of run time factors, returning an element in an enum

```
public enum Difficulty
{
   EASY, AVERAGE, HARD;
}
```

The simple factory

- This seems OK, so far, but then you realise that this bit of code is popping up all over the class
  - A lot of things get killed in this game!
- So, you do the sensible thing and re-factor the code into a method called *createItem*()



The simple factory

```
public Item createItem(Difficulty d)
{
   Item i = null;
   switch(d)
   {
      case EASY:       i = new CommonItem();
                       break;
      case AVERAGE:    i = new RareItem();
                       break;
      case HARD:       i = new EpicItem();
                       break;
   }
   return i;
}
```

You re-factor the *Item* creation code into a method that can be re-used

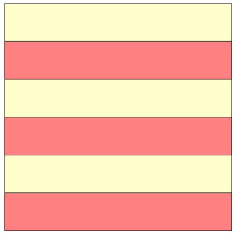
DRY – Don't repeat yourself

The simple factory

- Then your Boss tells you that we wants to make regular changes to your code
  - For example, he wants to add to the range of items returned by the method e.g. *VeryCommonItem*
- You realise that the *createItem*() method is going to change a lot now, and a lot in the future
- As with the *Strategy* pattern, you need some way of *encapsulating* this change and removing it from your core code
  - The *Strategy* pattern encapsulated a family of algorithms
  - What you need is a way to encapsulate the *instantiation* of objects

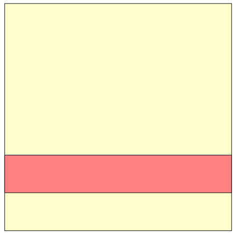
The simple factory

STEP 1



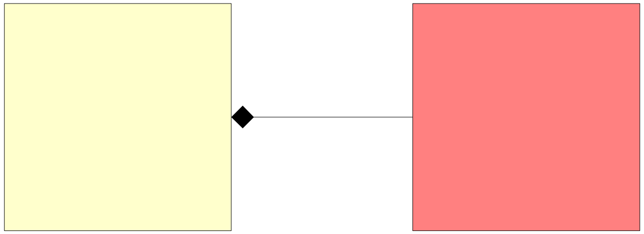
Object creation code occurs throughout class. This code creates one from a range of concrete classes dependent on run time factors

STEP 2



The object creation code is re-factored into a method

STEP 3



The object creation code is encapsulated in a separate class.

🟨 STABLE CODE

🟥 UNSTABLE CODE

Encapsulate what varies. In other words, separate the parts of your code that will change the most from the rest of your application, and re-use where possible.

The simple factory

```
class ItemFactory
{
    public Item createItem(Difficulty d)
    {
        Item i = null;
        switch(d)
        {
              case EASY:       i = new CommonItem();
                               break;
              case AVERAGE:    i = new RareItem();
                               break;
              case HARD:       i = new EpicItem();
                               break;
        }
        return i;
    }
}
```

OUR ITEM FACTORY

This class encapsulates the creation of *Items*. Some people call this sort of thing a *virtual constructor*. We have centralised the process of object creation.

The simple factory

```
public class TestFactory
{
   public static void main(String[] args)
   {
      Difficulty difficulty = rateDifficulty();
      ItemFactory factory = new ItemFactory();
      Item i = factory.createItem(difficulty);
      // add item to inventory
   }
}
```

Using the factory class. Note that we are are creating the correct item inside the client without using the *new* keyword. This can be good from a design point of view.

The simple factory

- What is wrong with using *new*?
  - There is nothing wrong with *new*
  - The real problem is *change* and the way change impacts our use of *new*
  - When you have code that chooses between a range of objects based on run time conditions you are heading for trouble because
    - That code may have to be changed as new concrete classes are added to the range
    - The criteria for object selection may change
  - The simple factory *decouples the client* from this business of object creation

The simple factory

- What's the advantage of this? It looks like we are just pushing the problem off to another object
  - One thing to remember is that the *ItemFactory* class may have many clients
  - Calculating the correct drop is a very common operation in a game, so many classes might be able to re-use the code now it has been factored into a class
  - Plus, we have isolated the volatile code and separated it from the stable code, as per the open-closed principle
- One minor twist on the simple factory involves making the factory method *static*
  - The only difference here is that you do not have to create an instance of the factory class

The simple factory

```
class ItemFactory
{
    public static Item createItem(Difficulty d)
    {
        Item i = null;
        switch(d)
        {
              case EASY:         i = new CommonItem();
                                   break;
              case AVERAGE:      i = new RareItem();
                                   break;
              case HARD:         i = new EpicItem();
                                   break;
        }
        return i;
    }
}
```

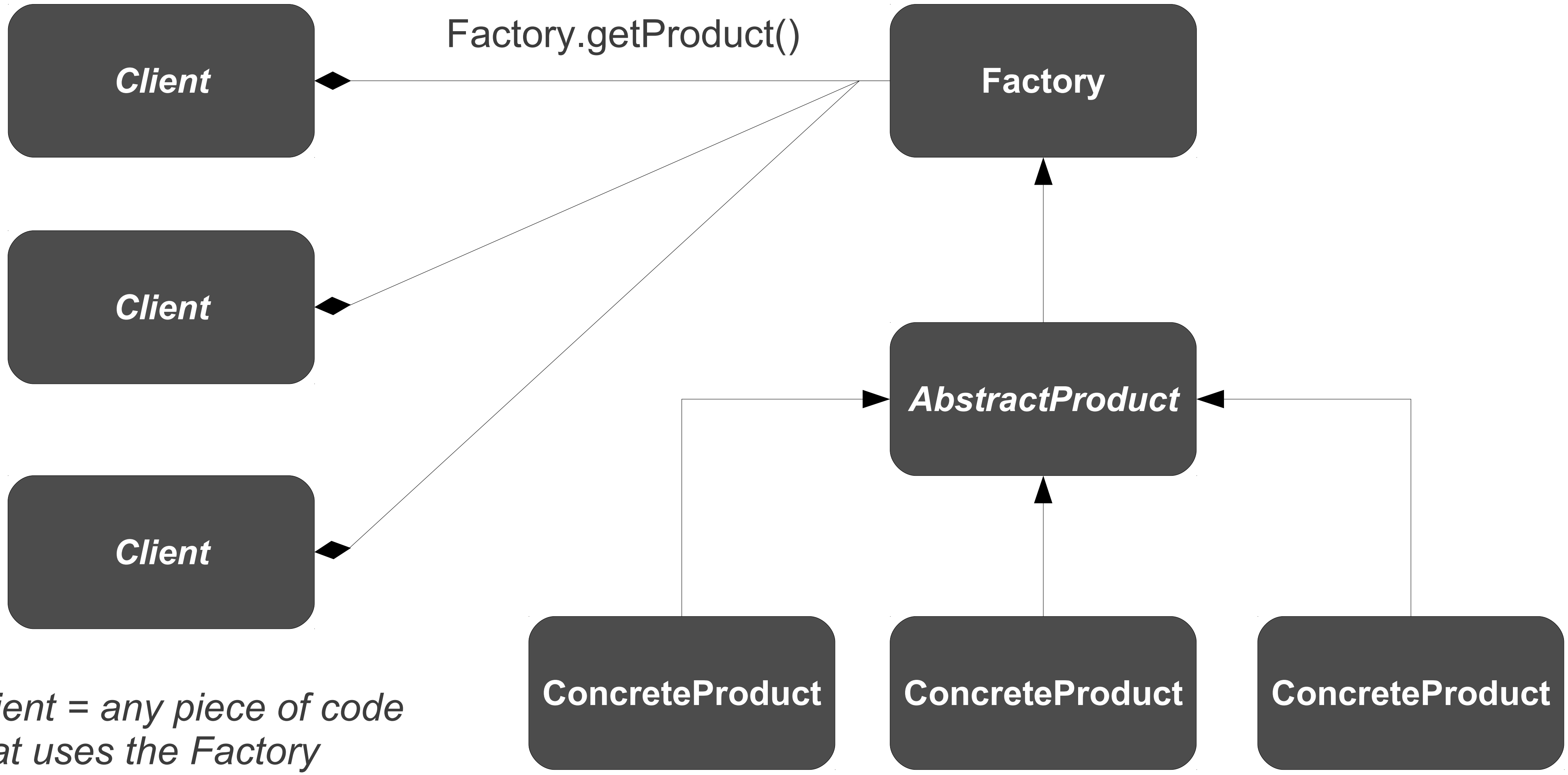
A STATIC FACTORY

The simple factory

```
public class TestFactory
{
   public static void main(String[] args)
   {
      Difficulty difficulty = rateDifficulty();
      Item i = ItemFactory.createItem(difficulty);
      // add item to inventory
   }
....
```

Using a static simple factory in the client. No need to instance the factory class

The simple factory



Client = any piece of code that uses the Factory

The simple factory takes care of the business of creating objects for its clients.
And business is good.

The Factory Method Pattern

The Factory Method pattern

- The simple factory is used extensively in SE
- If you ask most programmers to define the factory pattern, they will start talking about simple factories
  - But there is an important difference between the simple factory and the GoF factory method pattern
- As defined by the gang of four (GoF), the factory method pattern is something more flexible
  - According to their definition, the factory method "defines an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses"
  - Wait...What? WHAT?

The Factory Method pattern

- The key here is in the line 'let the subclasses decide'
- So far, we have not even considered extending the factory class
- But what if we encounter a situation where a simple factory class is no longer appropriate for all the clients
  - For example, the player of our MMORPG enters a realm where the item rewards are different e.g. *ElvishCommonItem*
  - Or the player passes some level threshold that changes the type of items that should be awarded e.g *UltimateItem*

The Factory Method pattern

- In this case, we need to modify the factory class, which we *can* do by simply extending it
  - *ElvenItemFactory* extends *ItemFactory*
- But it would be much better to define an interface, which can be implemented as necessary
  - This gives us more flexibility

```
interface ItemFactory
{
    Item createItem(Difficulty d);
}
```

***ItemFactory* as an interface**

The Factory Method pattern

```
class LowItemFactory implements ItemFactory
{
    @Override
    public Item createItem(Difficulty d)
    {
        Item i = null;
        switch(d)
        {
            case EASY:        i = new CommonItem();
                              break;
            case AVERAGE:     i = new RareItem();
                              break;
            case HARD:        i = new EpicItem();
                              break;
        }
        return i;
    }
}
```

An implementation of the *ItemFactory* interface, used for low level characters

The Factory Method pattern

```
class HighItemFactory implements ItemFactory
{
    @Override
    public Item createItem(Difficulty d)
    {
        Item i = null;
        switch(d)
        {
            case EASY:      i = new RareItem();
                            break;
            case AVERAGE:   i = new EpicItem();
                            break;
            case HARD:      i = new UltimateItem();
                            break;
        }
        return i;
    }
}
```

An implementation of the *ItemFactory* interface, used for high level characters

The Factory Method pattern

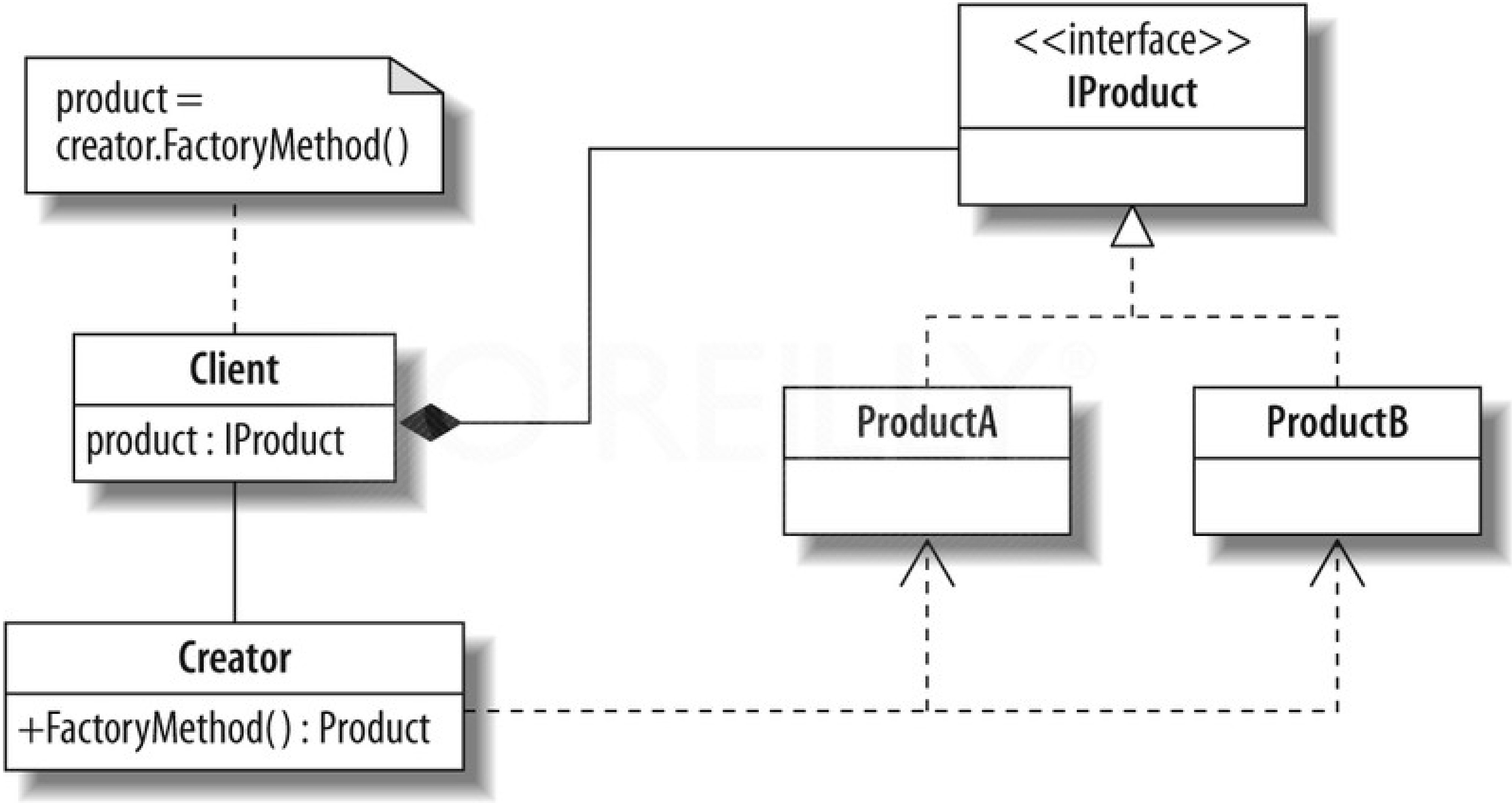
```
// Declare reference to ItemFactory interface
ItemFactory if;

// Character is low level, use the basic item factory
if = new LowItemFactory();
Item i = if.createItem(rateDifficulty());

// Character reaches level 40, time for different drops
if = new HighItemFactory();
Item j = if.createItem(rateDifficulty());
```

In the client....

The Factory Method pattern



Where *Creator* is in fact *ICreator. Note that the open arrow joining Creator to the concrete products simply indicates a navigable relationship, not inheritance. More.*

Composite pattern

The Facade pattern

- Façade is one of the easier design patterns
  - The Façade pattern provides a unified interface to set of interfaces in a subsystem
- A problem scenario – a car assembly line
  - In the 1st stage of the assembly process, the car passes through the *fabrication* bay where the bare metal body of the car is assembled
  - In the 2nd stage of the assembly process, the exterior of the car is sprayed in the *paint* bay
  - In the 3rd stage of the process, the car passes through the *trim* bay where the interior of the car is finished

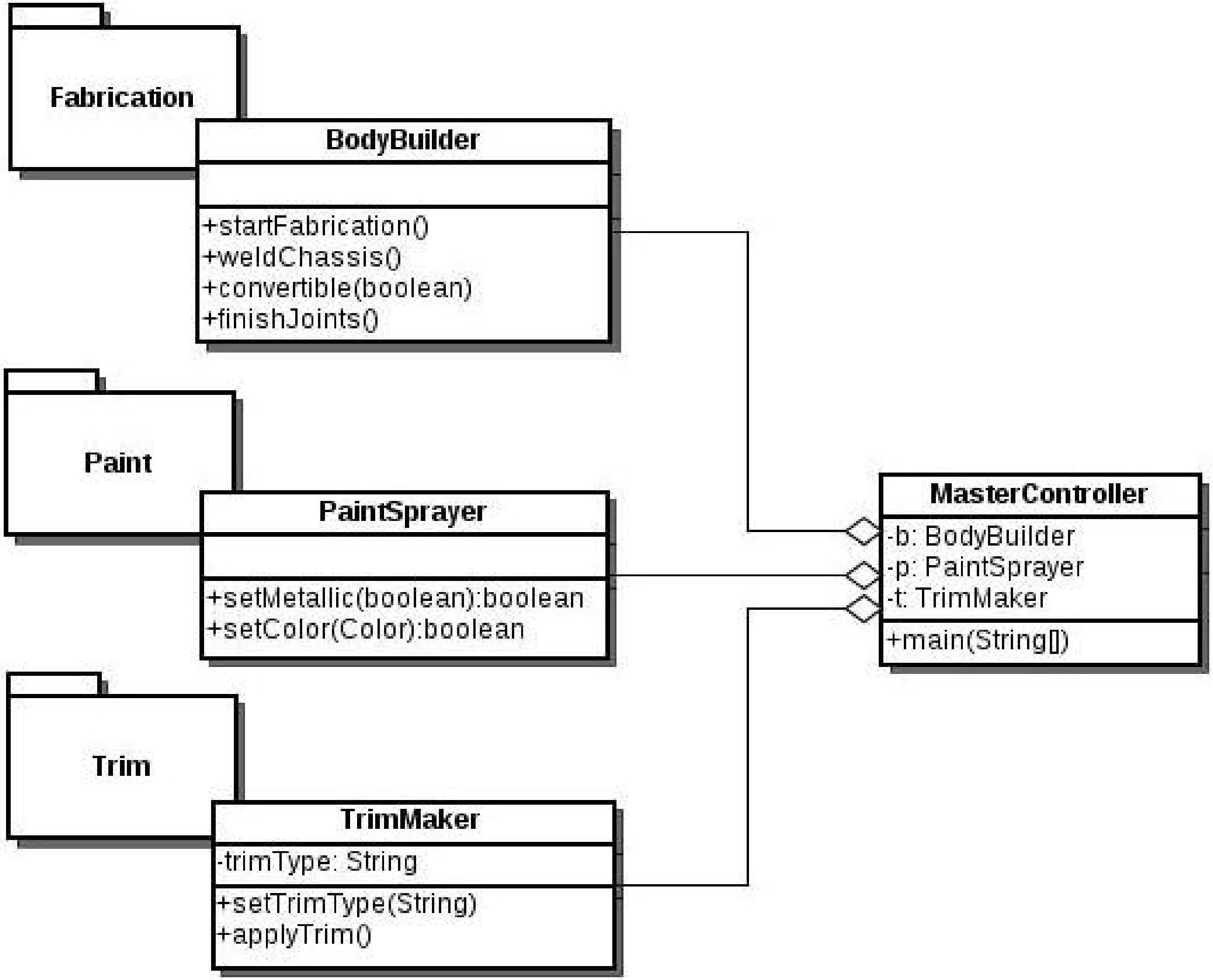
The Facade pattern

Automated car construction

The Facade pattern

- All of these stages are automated
  - Each action is carried out by a different robotic arm
  - All these arms run a Java Virtual Machine (VM)
- At present the code that builds the cars is spread across three separate APIs
- This makes the process of building a car unnecessarily complex
- *Facade* is a structural design pattern that will help you fix this problem

The Facade pattern



The existing code library

The Facade pattern

```
package Fabrication;
public class BodyBuilder
{
    public void startFabrication()
    { // (…) }

    public void weldChassis()
    { // (…) }

    public void convertible(boolean b)
    { // (…) }

    public void finishJoints()
    { // (…) }
}
```

The *BodyBuilder* class – part of the *Fabrication* package

The Facade pattern

```
package Paint;
import java.awt.Color;
public class PaintSprayer
{
    public boolean setMetallic(boolean b)
    {
        // (...)
        return true;
    }

    public boolean setColour(Color c)
    {
        // (...)
        return true;
    }
}
```

The *PaintSprayer* class – part of the *Paint* package

The Facade pattern

```
package Trim;
public class TrimMaker
{
   String trimType;

   public void setTrimType(String trimType)
   {
      this.trimType = trimType;
   }

   public void applyTrim()
   {
      // (...)
   }
}
```

The *TrimMaker* class – part of the *Trim* package

The Facade pattern

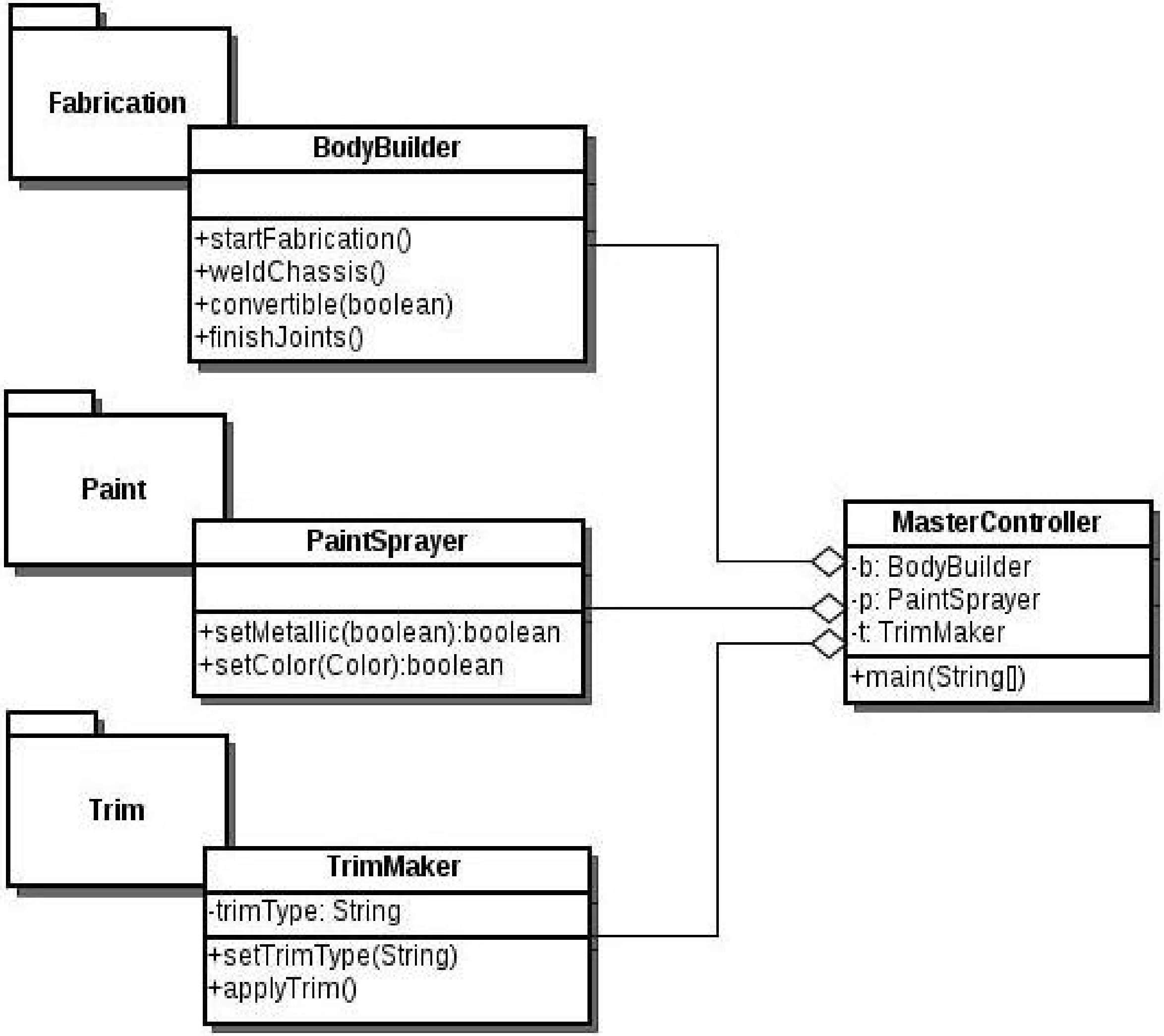
```
public class MasterControl
{
   public static void main(String[] args)
   {
      BodyBuilder b = new BodyBuilder();
      b.startFabrication(); b.weldChassis();
      b.convertible(true); b.finishJoints();

      PaintSprayer ps = new PaintSprayer();
      ps.setMetallic(true); ps.setColour(Color.red);

      TrimMaker tm = new TrimMaker();
      tm.setTrimType("oak"); tm.applyTrim();
   }
}
```

The client makes a number of calls into several different packages.
We need to simplify the interface

The Facade pattern

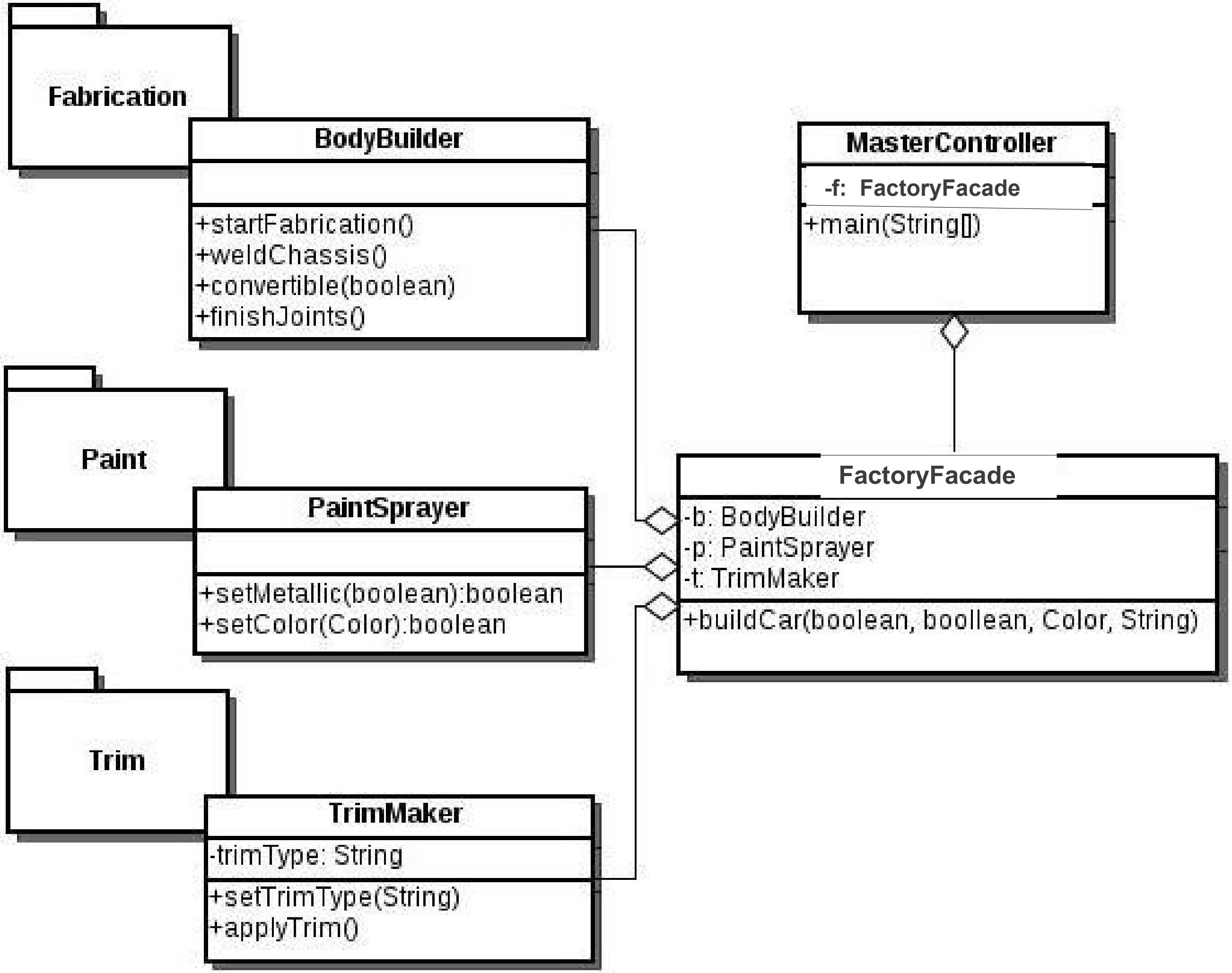


We have a client with too much knowledge of the classes it uses, and an overly complex sub-system. We need the *Facade*

The Facade pattern

- To solve this problem, we need to wrap the complex interface of the car building subsystem in a simplified interface
- We create a new class (called *FactoryFacade*)
  - This class is composed of all the components of the subsystem we want to use
  - This class contains one simplified method for building a car
  - This method wraps up all of the messy method calls seen in the 'bad' implementation
  - The client interacts with the simplified interface

The Facade pattern



The Facade pattern

```
public class FactoryFacade
{
    private BodyBuilder bb; private PaintSprayer ps; private TrimMaker tm;

    public void buildCar(boolean conv, boolean metallic, Color c, String trim)
    {
        bb.startFabrication();
        bb.weldChassis();
        bb.convertible(conv);
        bb.finishJoints();
        ps.setMetallic(metallic);
        ps.setColour(c);
        tm.setTrimType(trim);
        tm.applyTrim();
     }
}
```

A *Facade* class, exposing a simplified interface to our car building subsystem

The Facade pattern

```
public class MasterController
{
   public static void main(String[] args)
   {
      FactoryFacade ff = new FactoryFacade();
      ff.buildCar(true, true, Color.red, null);
   }
}
```

The client just needs to 'know' about the *Facade* class

The Facade pattern

- A facade can
  - Make a software library easier to use and understand
  - Make code that uses the library more readable
- What is the difference between Facade and Adapter?
  - When you need to use an existing class and the interface is not the one your client expects, use an adapter
    - The adapter changes an interface into the one your client expects
  - When you want to wrap a poorly-designed collection of APIs with a single well-designed API, use the Facade

Summary

- The simple factory moves object instantiation code out of the client
  - This is very useful if you anticipate frequent change in the range of objects created
- The GoF factory pattern is more powerful formulation of the same basic idea that defines an abstract factory class
  - Use this pattern when you anticipate several factories
- The Facade pattern is a structural pattern that provides a simplified/unified interface to set of interfaces in a subsystem
  - The client interacts with the simplified interface, decoupling it from changes in the subsystem
- Pre-reading: Composite and Flyweight in HFDP and DPFD