Abstract Factory Pattern

CSCI-4448 - Boese

University of Colorado **Boulder**

Objectives

- Problem
- Definition
- Why
- Examples
- How
- Comparisons

Problem

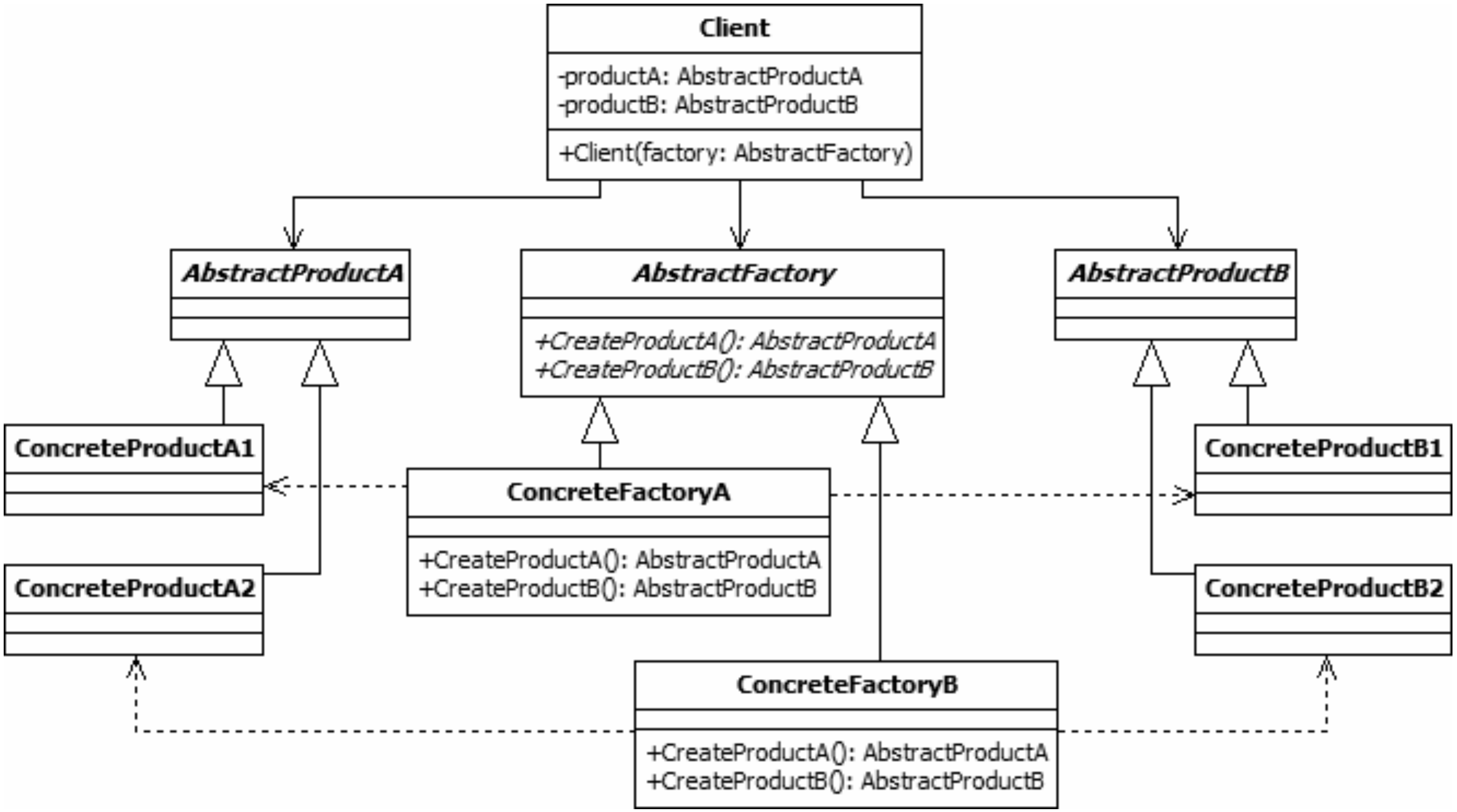
Abstract Factory Design Pattern

Problem:

- If an application is to be portable, it needs to encapsulate platform dependencies.
  These "platforms" might include:
  - windowing system,
  - operating system,
  - database, etc.
- Too often, this encapsulation is not engineered in advance, and lots of **`#ifdef`** case statements with options for all currently supported platforms begin to procreate like rabbits throughout the code.

https://sourcemaking.com/design_patterns/abstract_factory

Abstract Factory Class Diagram


Client
-productA: AbstractProductA
-productB: AbstractProductB
+Client(factory: AbstractFactory)
AbstractProductA
AbstractFactory
+CreateProductA(): AbstractProductA
+CreateProductB(): AbstractProductB
AbstractProductB
ConcreteProductA1
ConcreteProductA2
ConcreteFactoryA
+CreateProductA(): AbstractProductA
+CreateProductB(): AbstractProductB
ConcreteProductB1
ConcreteProductB2
ConcreteFactoryB
+CreateProductA(): AbstractProductA
+CreateProductB(): AbstractProductB

Definition

Definition

The abstract factory pattern is used to provide a client with a set of related or dependent objects.

The "family" of objects created by the factory are determined at run-time.

Definition

Name “Abstract Factory”

- This pattern is one level of abstraction higher than Factory Pattern.
- Abstract factory returns the factory of classes.
  Like Factory pattern returned one of the several sub-classes, this returns such factory which later will return one of the sub-classes.

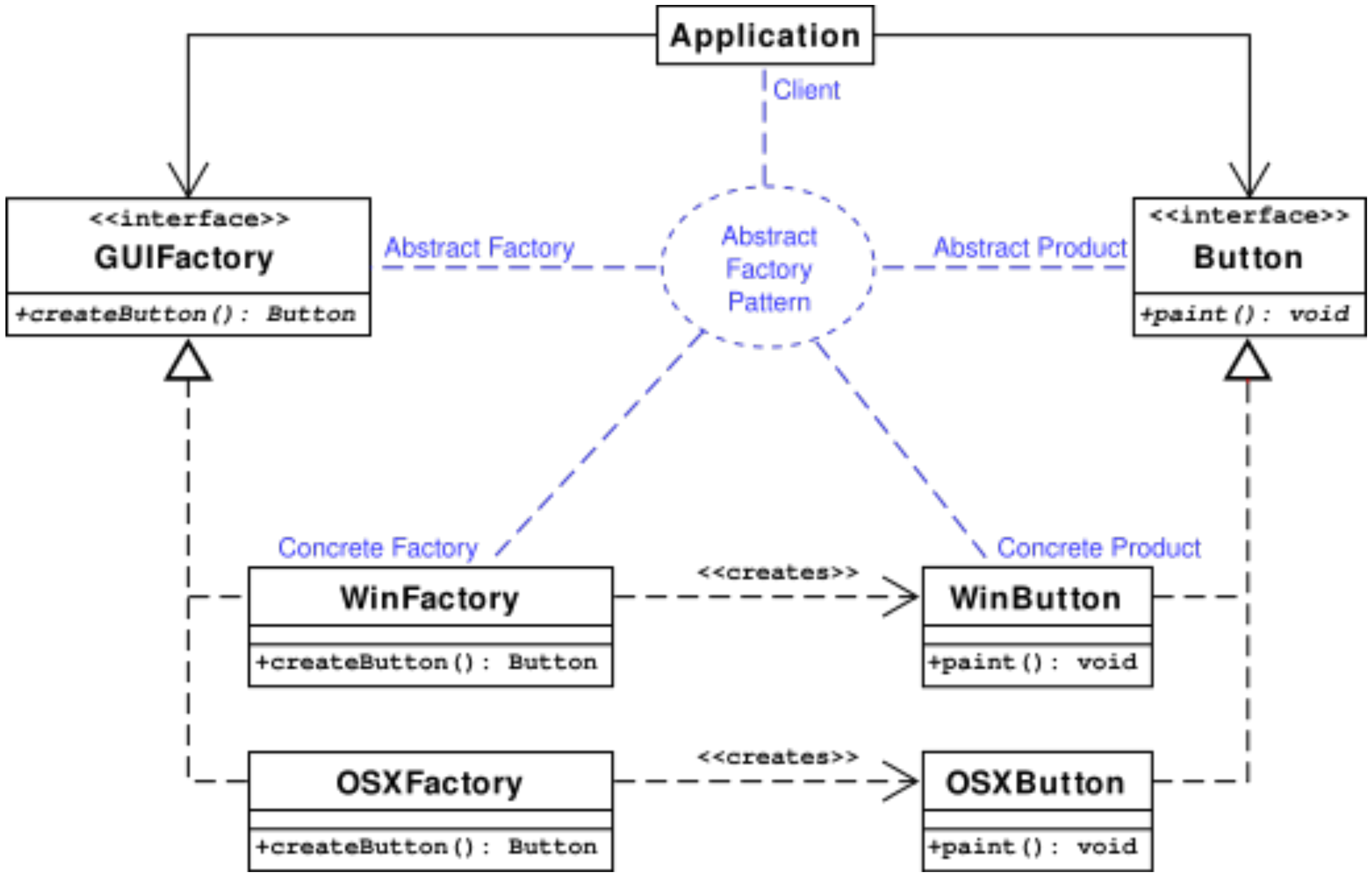
Intent

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- A hierarchy that encapsulates: many possible "platforms", and the construction of a suite of "products".
- When the new operator considered harmful.

Type

- Creational

Example Structure: Platforms



Examples

System

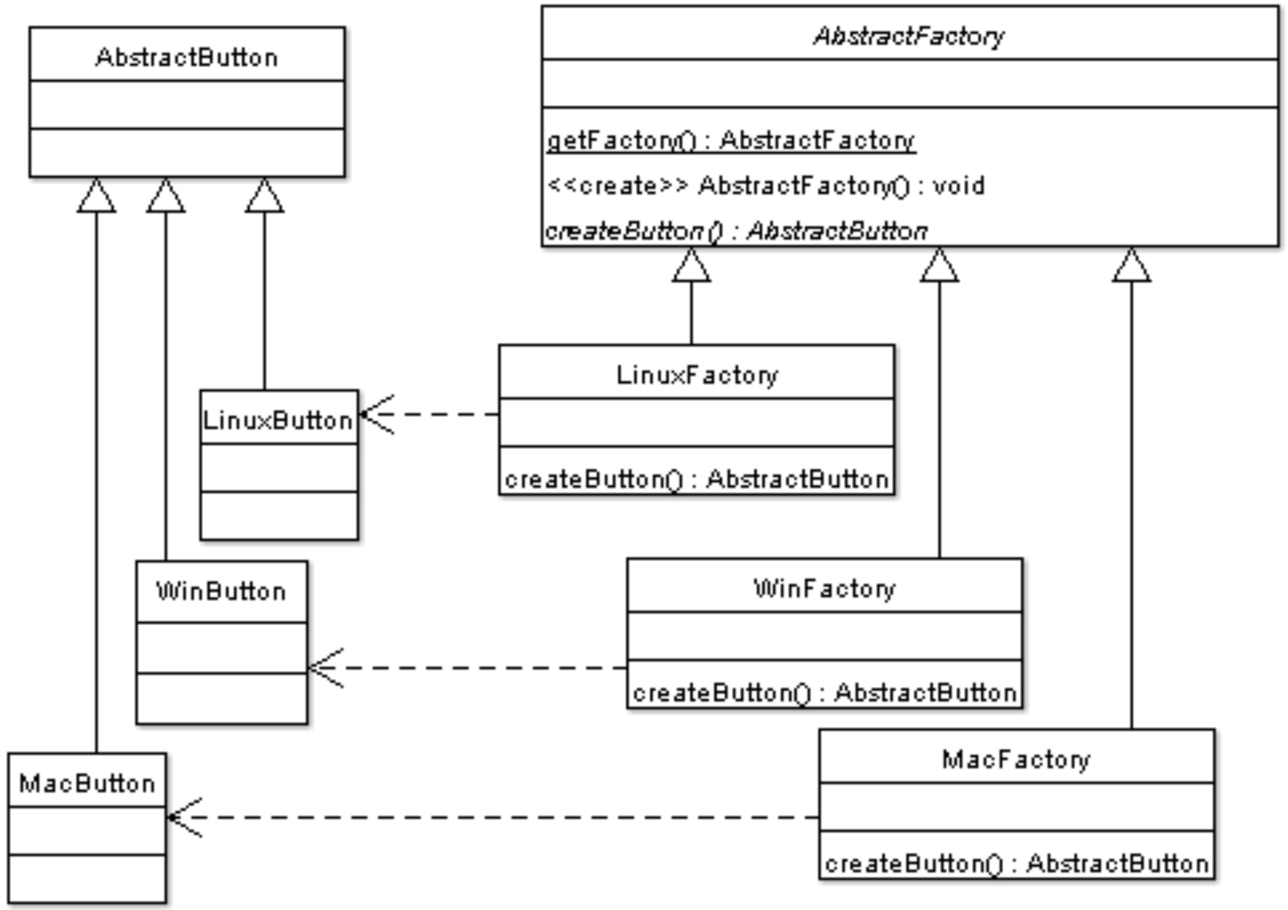
Example: System

An example of an Abstract Factory in use could be UI toolkits.

- Across Windows, Mac and Linux, UI composites such as
  - Windows
  - Buttons
  - Textfields

  are all provided in a widget API like SWT.
- However, the implementation of these widgets vary across platforms. You could write a platform independent client thanks to the Abstract Factory implementation.

Example: Structure


AbstractButton
AbstractFactory
getFactory() : AbstractFactory
<<create>> AbstractFactory() : void
createButton () : AbstractButton
LinuxButton
LinuxFactory
createButton() : AbstractButton
WinButton
WinFactory
createButton() : AbstractButton
MacButton
MacFactory
createButton() : AbstractButton

Example

```
// ConcreteProductA1
public class MSWindow implements Window
{
   public void setTitle( )   {
                                        //MS Windows specific behavior
  }
   public void repaint( )  {
                                        //MS Windows specific behavior
  }
}
```

```
// ConcreteProductA2
public class MacOSXWindow implements Window
{
   public void setTitle( )  {
                                        //Mac OSX specific behavior
  }
   public void repaint( )  {
                                        //Mac OSX specific behavior
  }
}
```

Example

```
// AbstractFactory
public interface AbstractWidgetFactory
{
   public Window createWindow();
}
```

Example

```
// ConcreteFactory1
public class MsWindowsWidgetFactory
{
                                                    //create an MSWindow
  public Window createWindow()
  {
    MSWindow window = new MSWindow();
    return window;
  }
}
```

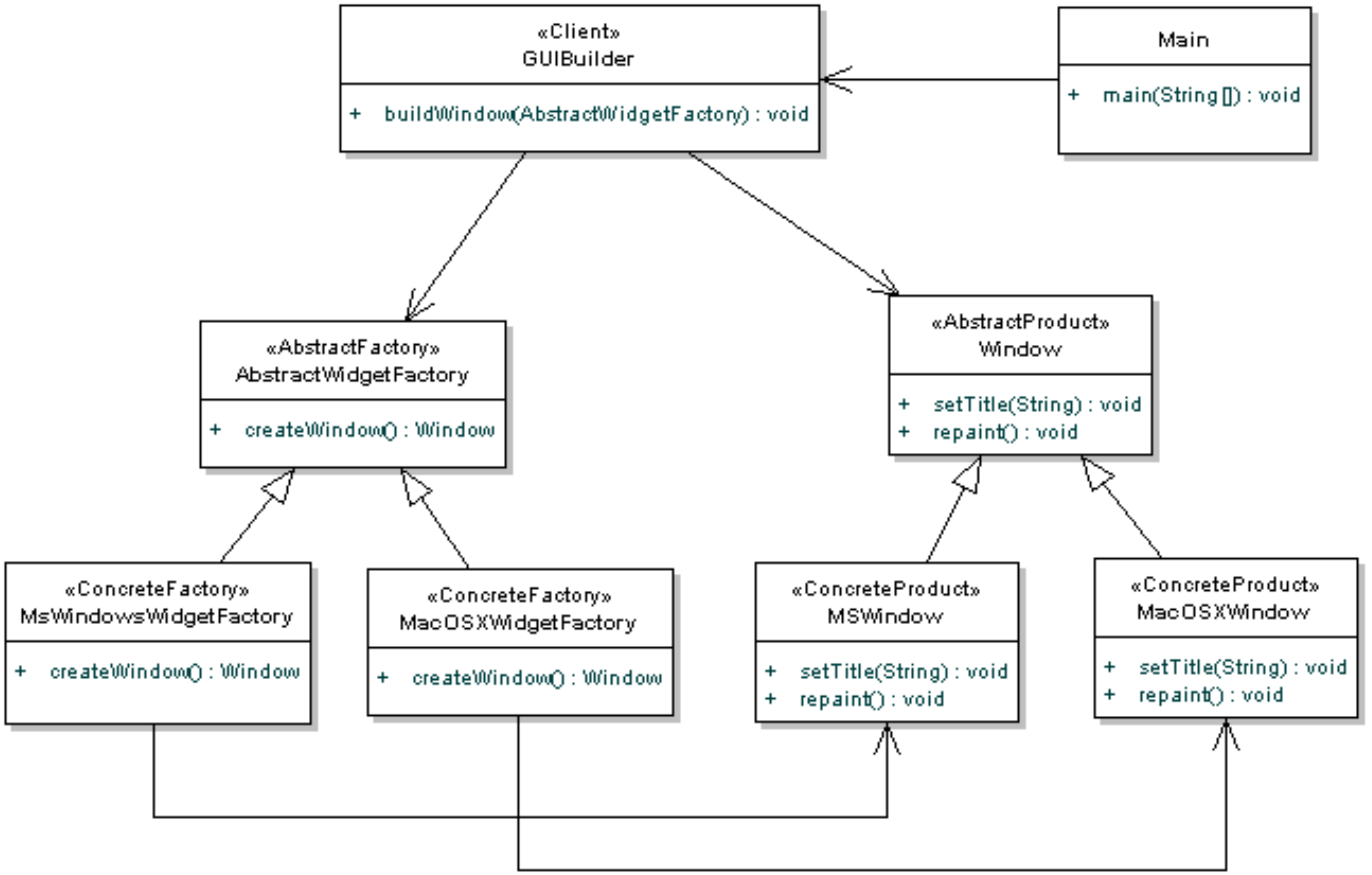
```
// ConcreteFactory2
public class MacOSXWidgetFactory
{
                                                    //create a MacOSXWindow
  public Window createWindow()
  {
    MacOSXWindow window = new MacOSXWindow();
    return window;
  }
}
```

Example

```
// Client
public class GUIBuilder
{
   public void buildWindow(AbstractWidgetFactory widgetFactory)
   {
      Window window = widgetFactory.createWindow();
      window.setTitle("New Window");
   }
}
```

Example

```
public class Main {
   public static void main(String[ ] args)
   {
     GUIBuilder builder = new GUIBuilder( );
     AbstractWidgetFactory widgetFactory = null;
                                                   //check what platform we're on
     if(Platform.currentPlatform( )=="MACOSX")
     {
           widgetFactory  = new MacOSXWidgetFactory( );
     }
     else
     {
           widgetFactory  = new MsWindowsWidgetFactory( );
     }
     builder.buildWindow(widgetFactory);
    }
}
```


«Client»
GUIBuilder
+ buildWindow(AbstractWidgetFactory) : void
Main
+ main(String []) : void
«AbstractFactory»
AbstractWidgetFactory
+ createWindow() : Window
«AbstractProduct»
Window
+ setTitle(String) : void
+ repaint() : void
«ConcreteFactory»
MsWindowsWidgetFactory
+ createWindow() : Window
«ConcreteFactory»
MacOSXWidgetFactory
+ createWindow() : Window
«ConcreteProduct»
MSWindow
+ setTitle(String) : void
+ repaint() : void
«ConcreteProduct»
MacOSXWindow
+ setTitle(String) : void
+ repaint() : void

Examples

Java

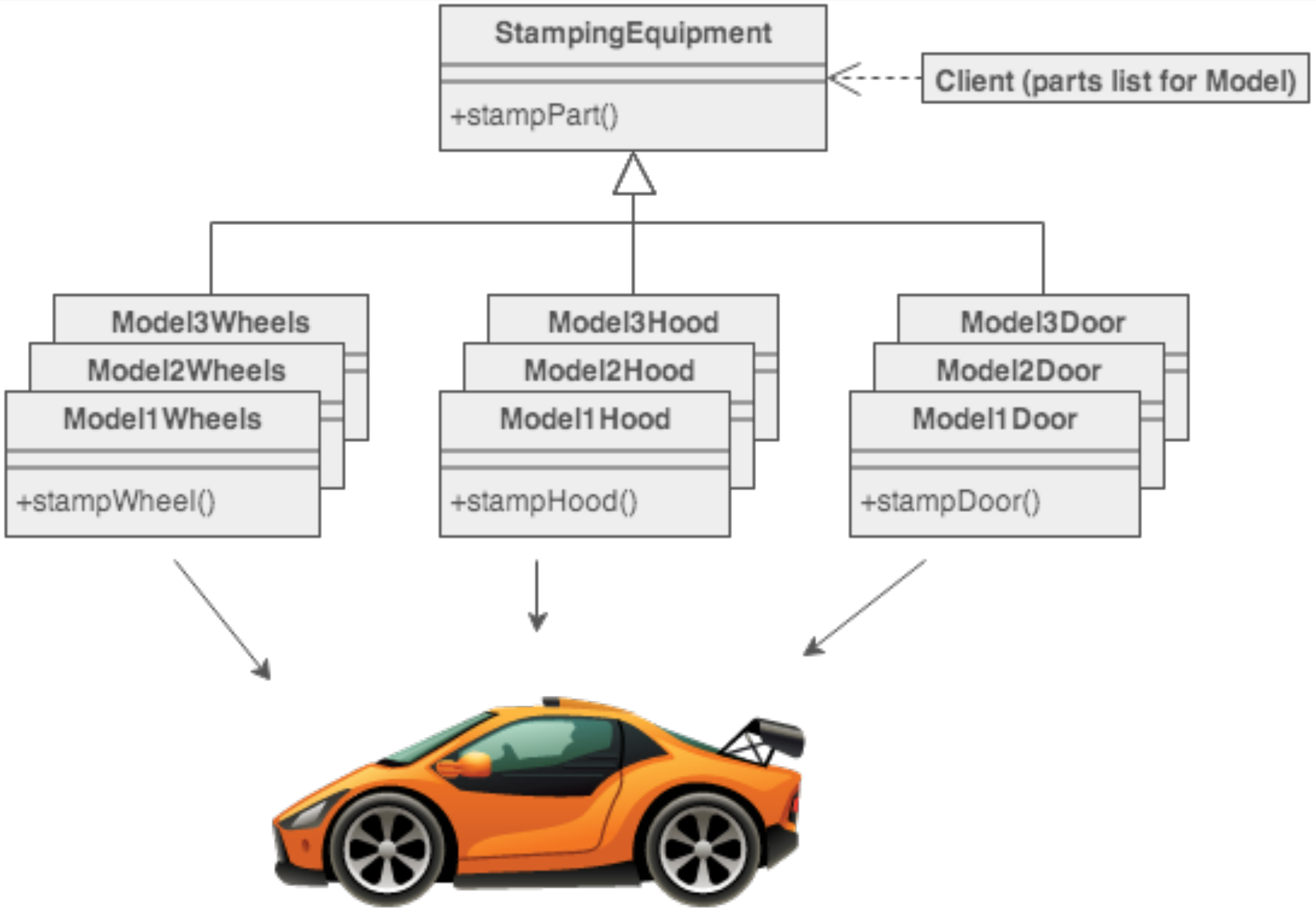
Example: Java

- javax.xml.parsers.DocumentBuilderFactory#newInstance()
- javax.xml.transform.TransformerFactory#newInstance()
- javax.xml.xpath.XPathFactory#newInstance()

Examples

Cars

Abstract Factory Example



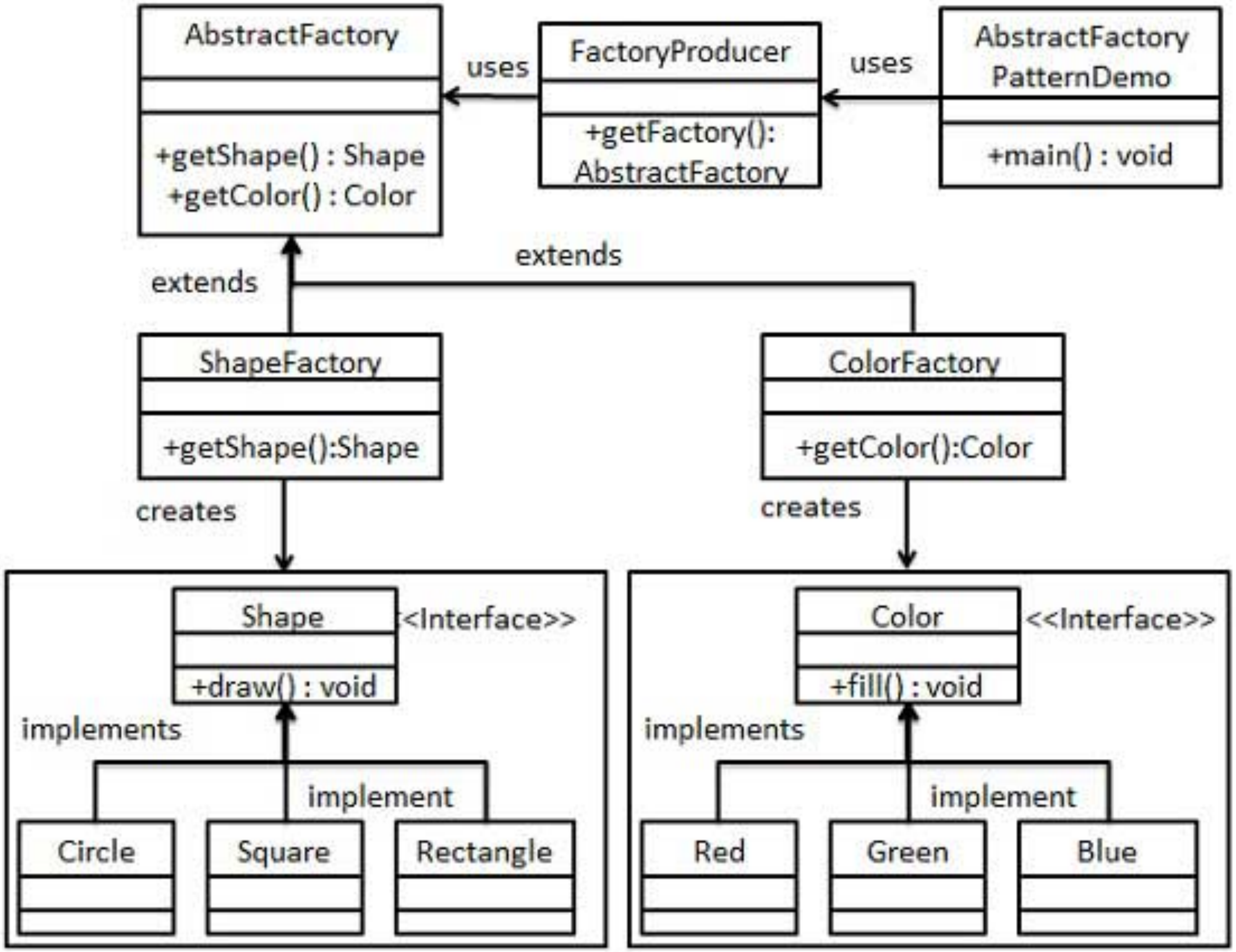
Abstract Factory Example

The purpose of the **Abstract Factory** is to provide an interface for creating families of related objects, *without specifying concrete classes.*

This pattern is found in the sheet metal stamping equipment used in the manufacture of Japanese automobiles.

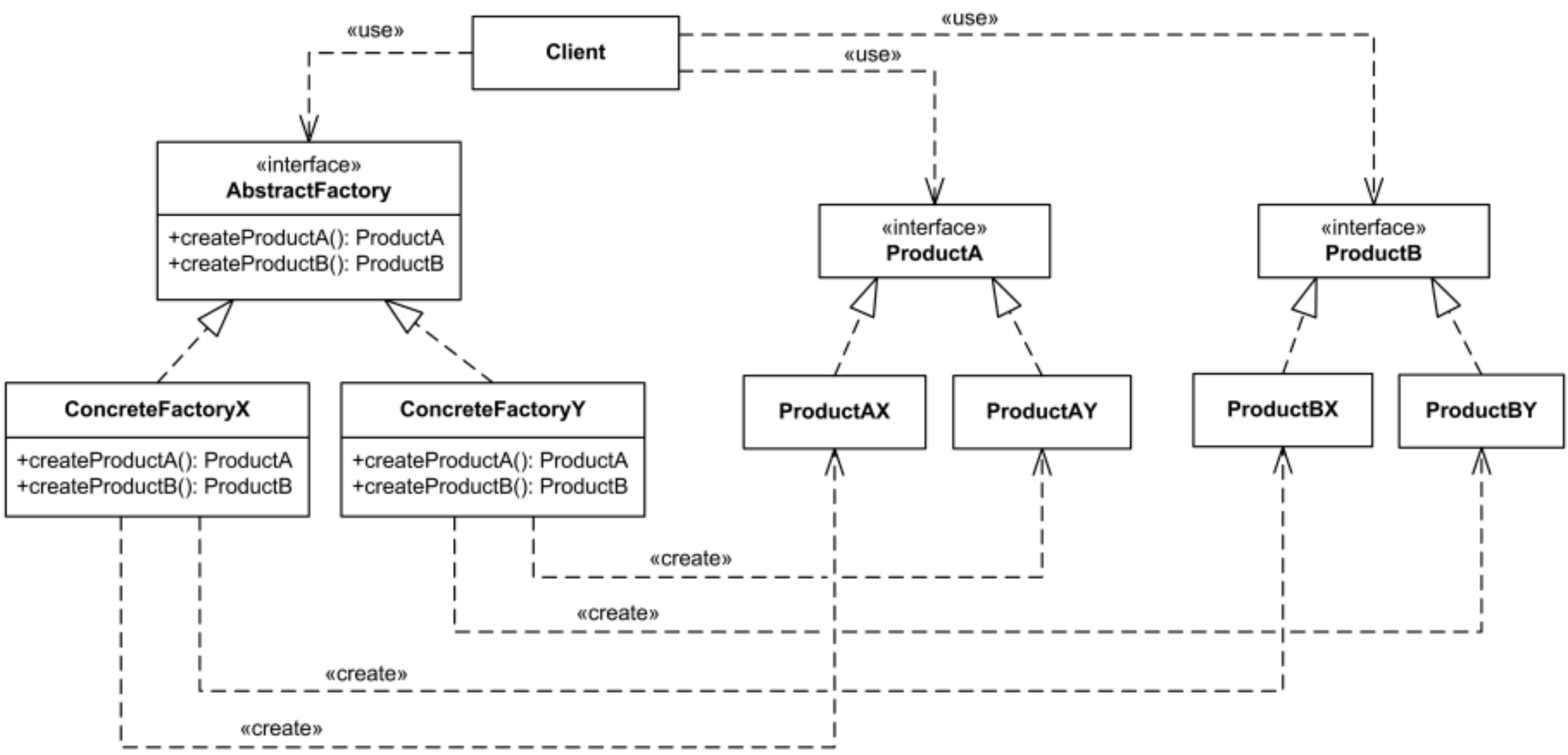
- The stamping equipment is an Abstract Factory which creates auto body parts.
- The same machinery is used to stamp right hand doors, left hand doors, right front fenders, left front fenders, hoods, etc. for different models of cars.
- Through the use of rollers to change the stamping dies, the concrete classes produced by the machinery can be changed within three minutes.

• A


AbstractFactory
+getShape() : Shape
+getColor() : Color
uses
FactoryProducer
+getFactory(): AbstractFactory
uses
AbstractFactory PatternDemo
+main() : void
extends
extends
ShapeFactory
+getShape():Shape
ColorFactory
+getColor():Color
creates
creates
Shape
<<Interface>>
+draw() : void
implements
implement
Circle
Square
Rectangle
Color
<<Interface>>
+fill() : void
implements
implement
Red
Green
Blue

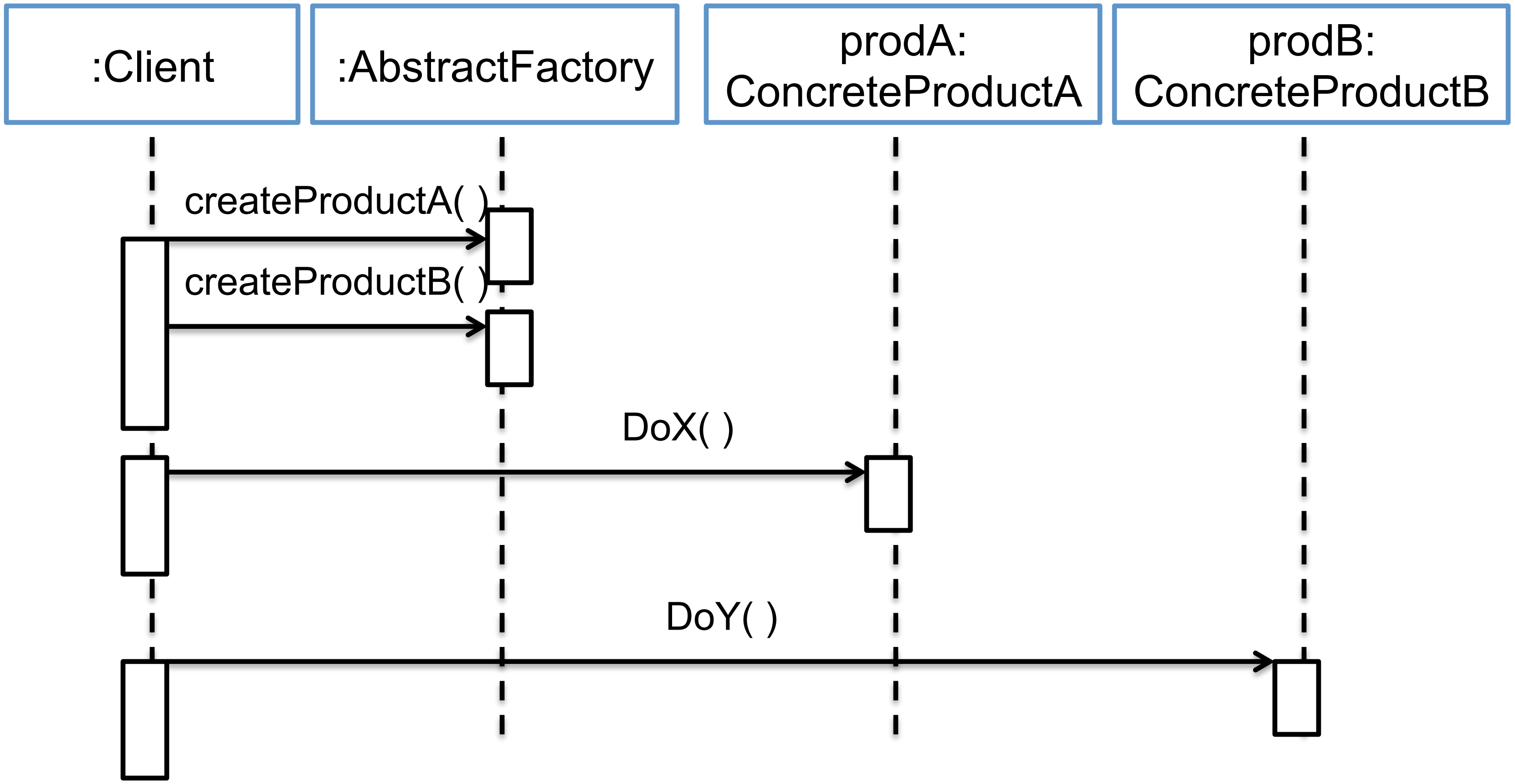
How

Structure



http://www.uml-diagrams.org/design-pattern-abstract-factory-uml-class-diagram-example.html

Abstract Factory Sequence Diagram



How

1. Decide if "platform independence" and creation services are the current source of pain.
2. Map out a matrix of "platforms" versus "products".
3. Define a factory interface that consists of a factory method per product.
4. Define a factory derived class for each platform that encapsulates all references to the new operator.
5. The client should retire all references to **`new`**, and use the factory methods to create the product objects.

Comparisons

Comparisons

- Abstract Factory, Builder, and Prototype can use **Singleton** in their implementation.
- Abstract Factory classes are often implemented with **Factory Methods**, but they can be implemented using **Prototype**.
  - Factory Method: creation through inheritance.
  - Prototype: creation through delegation.
- Often, designs start out using Factory Method (less complicated, more customizable, subclasses proliferate) and evolve toward Abstract Factory, Prototype, or Builder (more flexible, more complex) as the designer discovers where more flexibility is needed.
- Prototype doesn't require subclassing, but it does require an Initialize operation. Factory Method requires subclassing, but doesn't require Initialize.

Comparisons

- Abstract Factory can be used as an alternative to Facade to hide platform-specific classes.
- The advantage of a Factory Method is that it can return the same instance multiple times, or can return a subclass rather than an object of that exact type.
- Some Factory Method advocates recommend that as a matter of language design (or failing that, as a matter of style) absolutely all constructors should be private or protected. It's no one else's business whether a class manufactures a new object or recycles an old one.
- The `new` operator considered harmful. There is a difference between requesting an object and creating one. The new operator always creates an object, and fails to encapsulate object creation. A Factory Method enforces that encapsulation, and allows an object to be requested without inextricable coupling to the act of creation.

Summary

Cons

- Watch out for the downsides
  - While the pattern does a great job of hiding implementation details from the client, there is always a chance that the underlying system will need to change.
    We may have to add new attributes to our **AbstractProduct**, or **AbstractFactory**, which would mean a <u>change to the interface that the client was relying on</u>, thus breaking the API.
  - With both factories - someone has to determine what type of factory the client is dealing with at runtime, usually done with some type of switch statement.

Abstract Factory

- Implementation Issues
  - How many instances of a particular concrete factory should there be?
  - An application typically only needs a single instance of a particular concrete factory
  - Use the Singleton pattern for this purpose
  - How can the factories create the products?
    - *Factory Methods*
    - *Factories*
  - How can new products be added to the AbstractFactory interface?
    - *AbstractFactory defines a different method for the creation of each product it can produce*
    - *We could change the interface to support only a make(String kindOfProduct) method*