# Outline

- **Factory Method Pattern**
- **Abstract Factory**
- **Dependency Injection**

# Creational Patterns

- **Creational Patterns**
  - Factory Method
  - Abstract Factory
  - Dependency Injection
  - Singleton
  - Prototype
  - Builder
  - Creational patterns abstract the object instantiation process, i.e. they hide how objects are created
  - Make the system independent of how its objects are created

# Factory Patterns

- **Intent**
  - Goal: make program code independent of concrete classes

```
IX ref = new X();
```

    - Instantiation with the new operator implies a dependency to the concrete class
    - Exchanging the created class (e.g. with a decorator or proxy) is demanding and error-prone

  => Factories centralize the creation of objects and can easily be adjusted

  - Factory method
    - Class creational pattern which delegate the instantiation to a subclass
  - Abstract factory
    - Object creational pattern delegate the instantiation to another object

# Factory Method Pattern

- **Motivation: JUnit Tests**

```
public class RectangleInformationHidingTest {
   private Figure f;

   @Before
   public void setUp() {
      f = new jdraw.figures.Rect(0, 0, 20, 10);
   }

   @Test
   public void getBoundsTest() {
      Rectangle r = f.getBounds();
      r.translate(10, 10);
      assertTrue("result of getBounds must be cloned",
                                        f.getBounds().x == 0);
   }
}
```

# Factory Method Pattern

- **Motivation: JUnit Tests**

```
public class RectangleNotificationTest {
   private Figure f; private int cnt;

   @Before
   public void setUp() {
      f = new jdraw.figures.Rect(0, 0, 20, 10);
      cnt = 0;
   }

   @Test
   public void notificationTest() {
      f.addFigureListener(e -> cnt++);
      f.move(1, 1);
      assertTrue("figureChanged must be called", cnt == 1);
   }
}
```

# Factory Method Pattern

- **Motivation: JUnit Tests**
  - Problem: I do not want to create a new Test-Class for each figure type
    - Would lead to code duplication (for each figure)
  - Solutions:
    - Parameterize Class with a Figure instance passed with the constructor
      - Not an option for JUnit, classes must have a default constructor
    - Parameterized JUnit Tests
      - An annotated method provides the constructor parameters
    - Factory method
      - Define an abstract method in a test base class which creates the figure to be tested
      - The abstract base class contains the complete test code (=> final) except the instantiation part
      - Allows to use several instances of the figure under test in the test methods

# Factory Method Pattern

- **Parameterized JUnit Tests**

```
@RunWith(Parameterized.class)
public class FigureTest {
  private Figure f;
  private int cnt;

  public FigureTest(Figure f) { this.f = f; }

  @Parameters
  public static List<Object[]> getParams() {
    Object[][] figs = new Object[][] {
      { new Line(new Point(0, 0)) },
      { new Oval(new Point(0, 0)) },
      { new Rect(0, 0, 20, 10, java.awt.Color.RED) } };
    return Arrays.asList(figs);
  }
  …
```

# Factory Method Pattern

- **Parameterized JUnit Tests**

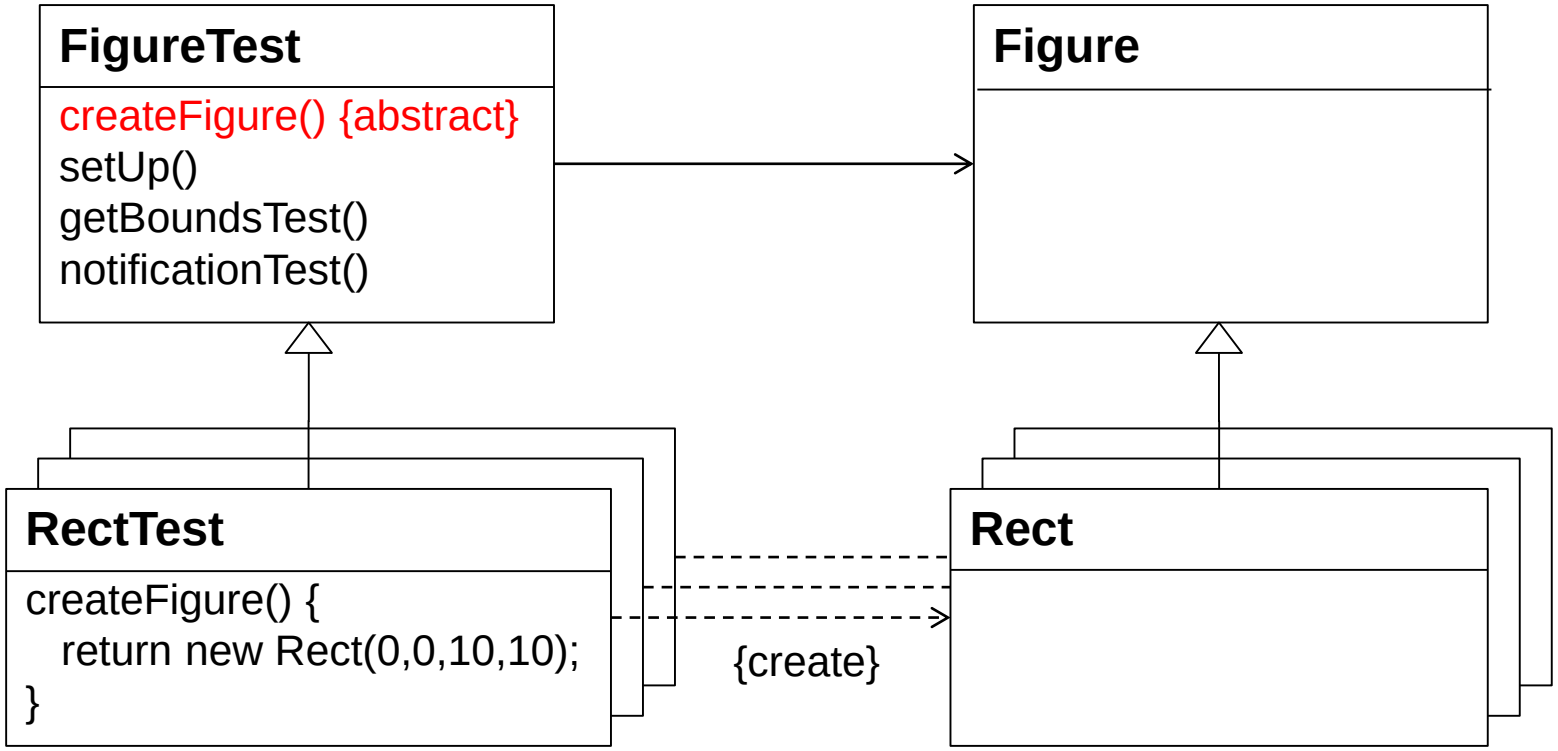
```
  …

  @Before
  public void setUp() { cnt = 0; }

  @Test
  public void notificationTest() {
    f.addFigureListener(new FigureListener() {
      public void figureChanged(FigureEvent e) { cnt++; }
    });
    f.move(1, 1);
    assertTrue("figureChanged must be called", cnt == 1);
  }
}
```

# Factory Method Pattern

- **Structure**



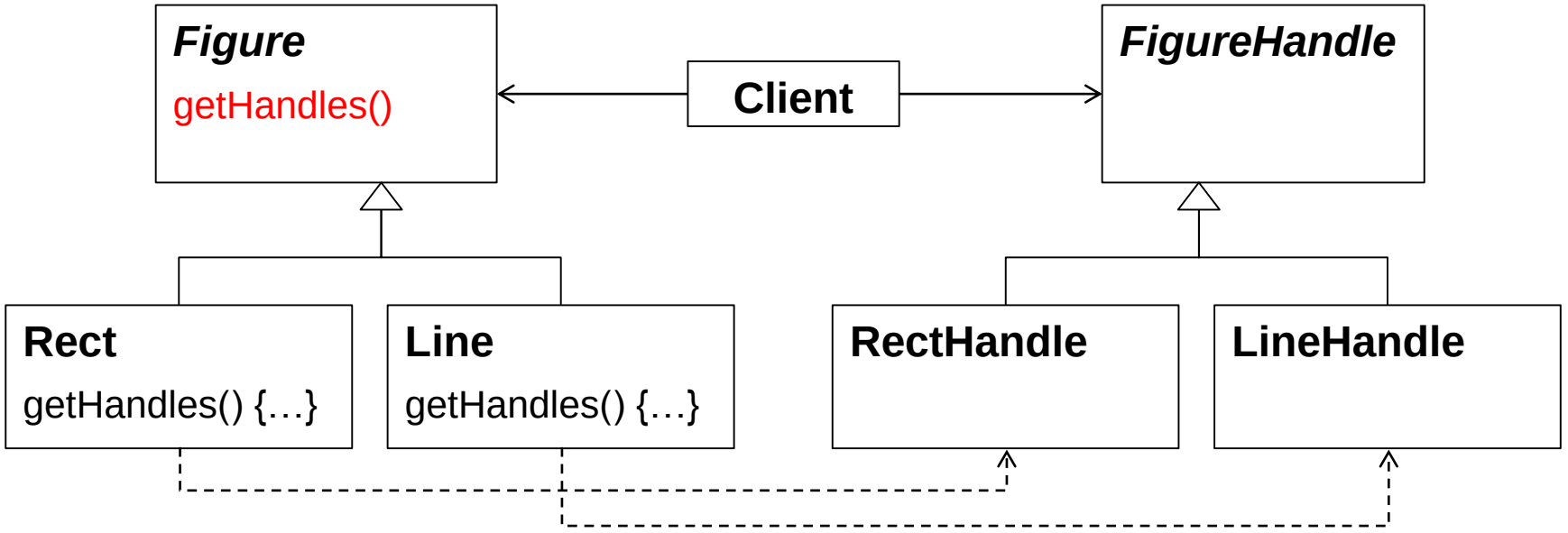
# Factory Method Pattern

- **Intent**
  - Define a method for creating new instances, but let concrete implementations of this method decide which class to instantiate
    - No dependency on concrete classes (i.e. no `new ConcreteClass(…)`)
  - Factory method may be implemented in
    - Subclasses
    - In a static method
    - In an separate object => Abstract Factory Pattern

- **Benefits**
  - Class which is creating instances is independent of concrete classes => flexibility / extendibility
  - Code only uses the product class over its interface, it can thus work with any concrete product that supports this interface

# Factory Method Pattern

- **Example: JDraw FigureHandles**
  - Method `getHandles` is a factory method
  - Typically used in the context of parallel class hierarchies



# Factory Method Pattern

- **Implementation Issues**
  - Default factory
    - Factory method in base abstraction may be abstract, or it may provide a default behavior which can be overridden
    - Alternative is a static factory method

```
class Factory {
   public static Product createProduct() { … }
}
```

      - Cannot be overridden in subclasses, comparable to a final factory method
  - Parameterized factory method
    - The factory method may be parameterized to describe the product it creates

```
Product createProduct(ProductID id)
```

      - This typically leads to case statement inside the code

# Outline

- Factory Method Pattern
- Abstract Factory
- Dependency Injection

# Abstract Factory Pattern

- **Motivation**
  - Let us assume, that we want to generate different types of objects, e.g. GUI controls from different windowing technologies or different look & feel

```
static Label newLabel(String version, String text) {
  switch (version) {
    case "AWT":   return new ComponentsAWT.LabelAWT(text);
    case "Swing": return new ComponentsSwing.LabelSwing(text);
    case "SWT":   return new ComponentsSWT.LabelSWT(text);
    case "FX":    return new ComponentsFX.LabelFX(text);
    default:      throw new IllegalStateException();
  }
}
```

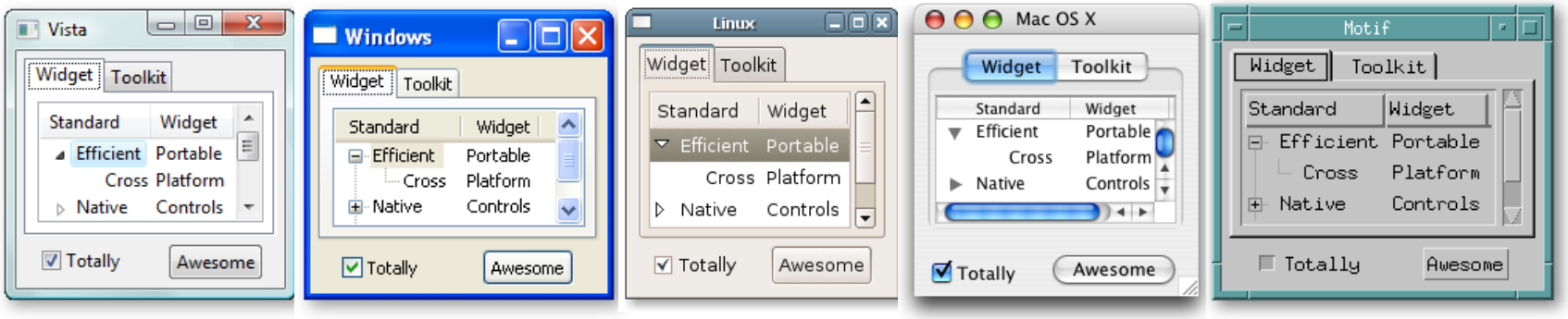
  - The type of objects has to be exchangeable at runtime
  - There might be several factory methods with such a switch statement
    - Which known pattern can be used to eliminate that code smell?

# Abstract Factory Pattern

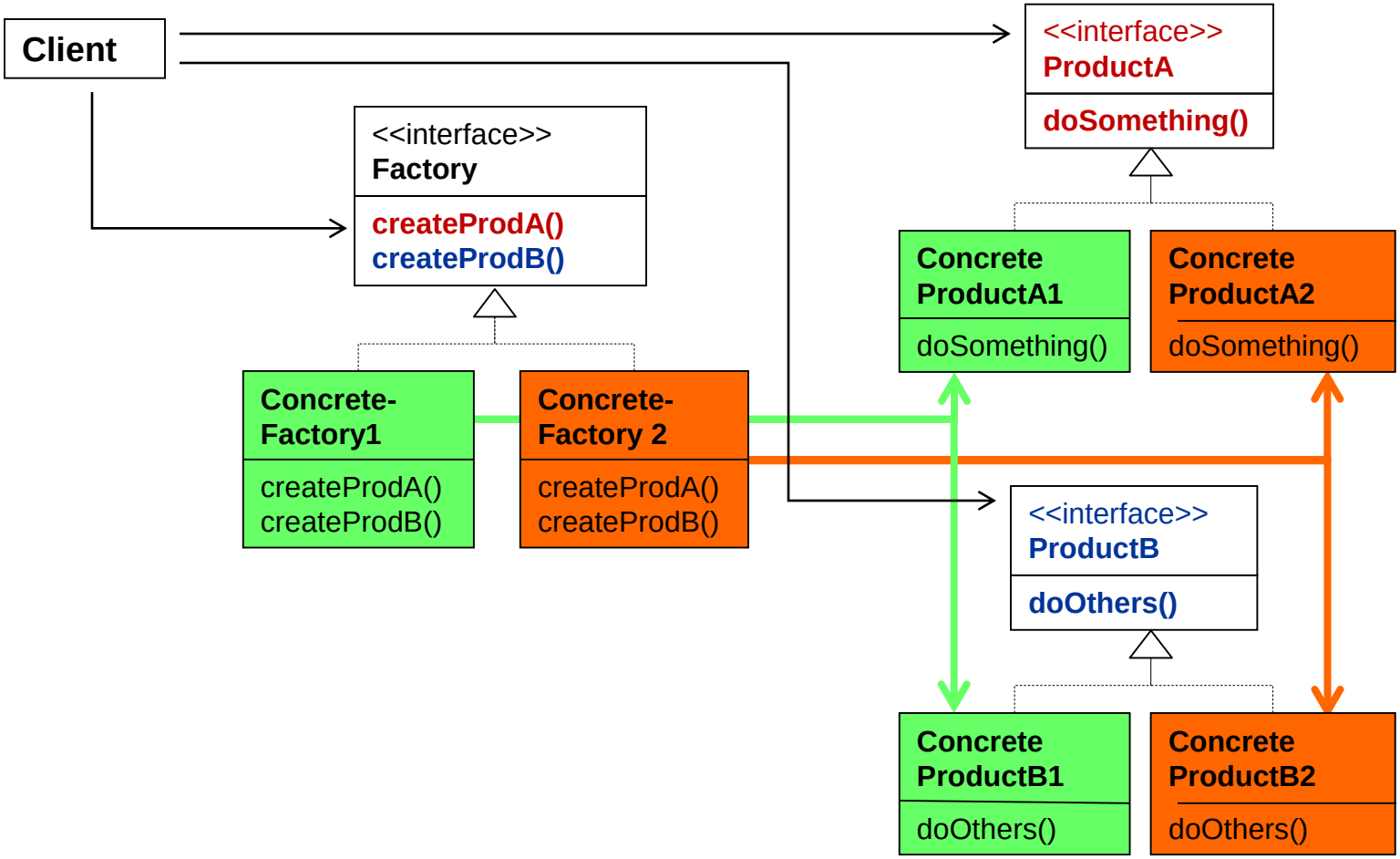
- **Intent**
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes

Abstract Method + Strategy Pattern => Abstract Factory

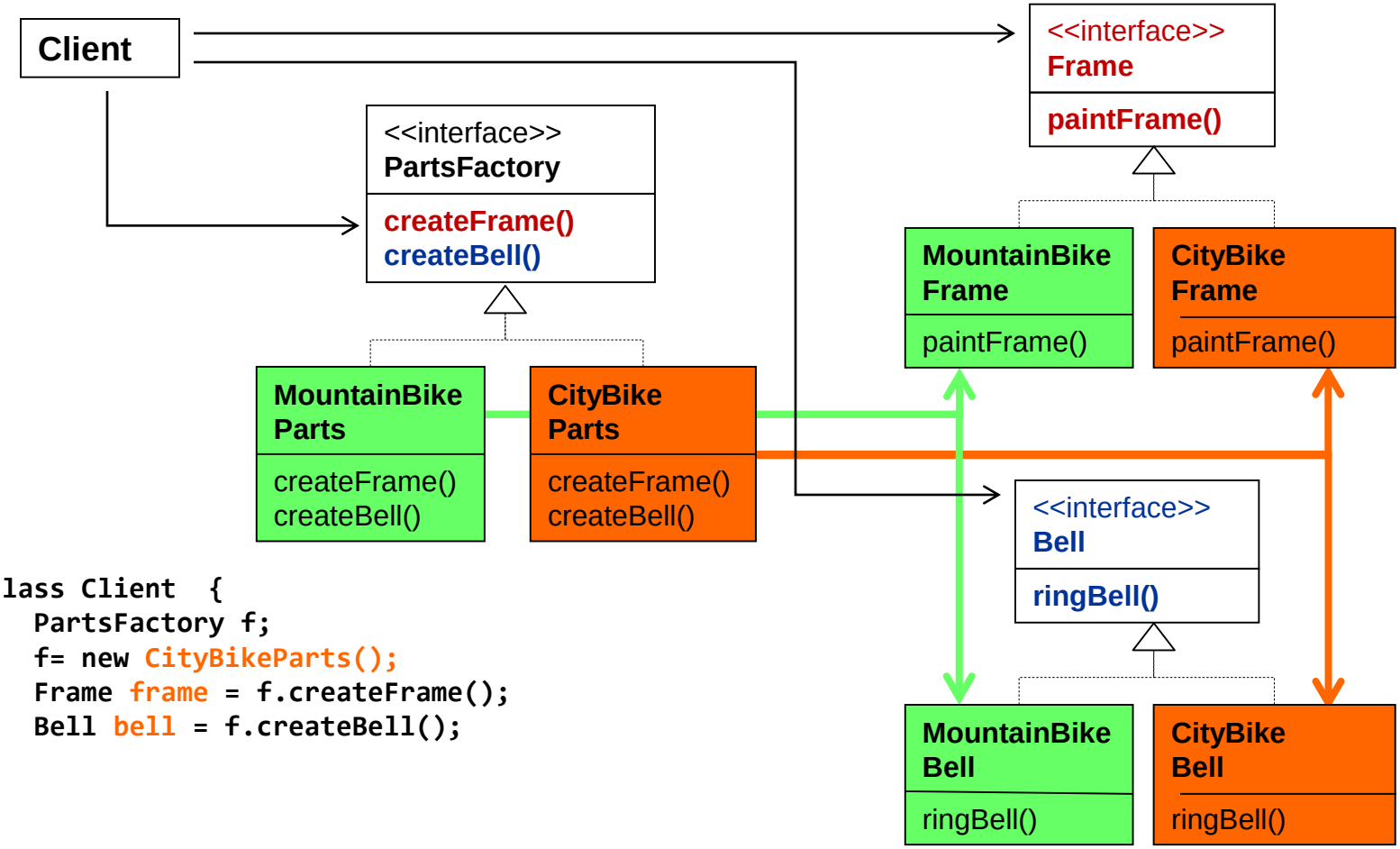
- **Example**
  - Creation of UI controls (Abstract Factory = Creation Strategy)



# Abstract Factory: Structure



# Abstract Factory: Bike Factory




```
class Client  {
   PartsFactory f;
   f= new CityBikeParts();
   Frame frame = f.createFrame();
   Bell bell = f.createBell();
}
```

# Abstract Factory: Implementation

- **Factory Interface**

```
interface Factory {                    interface A { … }
   A createA();                        interface B { … }
   B createB();
}
```

- **Factory Class**

```
abstract class Factory {               interface A { … }
   public abstract A createA();        interface B { … }
   public abstract B createB();
}
```

# Abstract Factory: Implementation

- **Where is the concrete Factory?**
  - Factory may be created by the client
  - If current Factory class has to be accessible by all => static field
    - Either in the abstract Factory class itself
    - In a special class

```
class CurrentFactory {
   private CurrentFactory() { }
   private static Factory current = null;
   public static Factory getFactory() { return current; }
   public static void setFactory(Factory f) {
      if(f==null) throw new IllegalArgumentException();
      current = f;
   }
}
```

- optionally not changeable {frozen}
- optionally with default implementation
- optionally default implementation separately accessible

# Abstract Factory: Implementation

- **How is a concrete Factory registered?**
  **Someone has to call `setFactory` with a concrete Factory object**
  - Externally:

```
CurrentFactory.setFactory(new Factory1());
```

  - Internally in each Factory (static register method):

```
Factory1.register();
```

  - Automatically upon loading of the factory

```
class Factory1 implements Factory {
   public A createA() { … }
   public B createB() { … }
   static { CurrentFactory.setFactory(new Factory1()); }
   private Factory1() { };
}
```

# Abstract Factory: Motivation / Examples

- **Motivation**
  - Client code should be independent of creation of new instances, i.e. independent of concrete classes
  - Configurability of a system with a set of classes (i.e. a concrete factory)
  - Extensibility with new implementations (additional product families)

- **Examples**
  - Pluggable look and feel of Swing
  - JDBC driver (=> several factories are available simultaneously)
  - Test- and Productive Version Factories
  - Local and Remote Factories
  - Different Version Factories (probably constructors have changed)

# Abstract Factory: Remarks

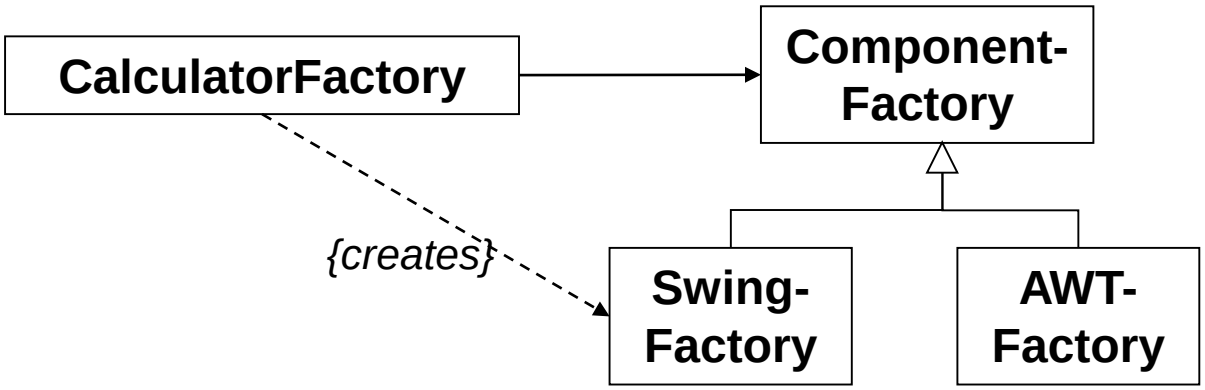
- **Remarks**
  - Configuration may be exchangeable at run-time
  - Default implementations may be provided
  - Factory contains typically a set of create methods (family of objects)
  - Factory may contain state, e.g. attributes which are used to create new instances

  - Adaptations for *new* products very expensive!

# Outline

- **Factory Method Pattern**
- **Abstract Factory**
- **Dependency Injection**

# Problem of Coupling

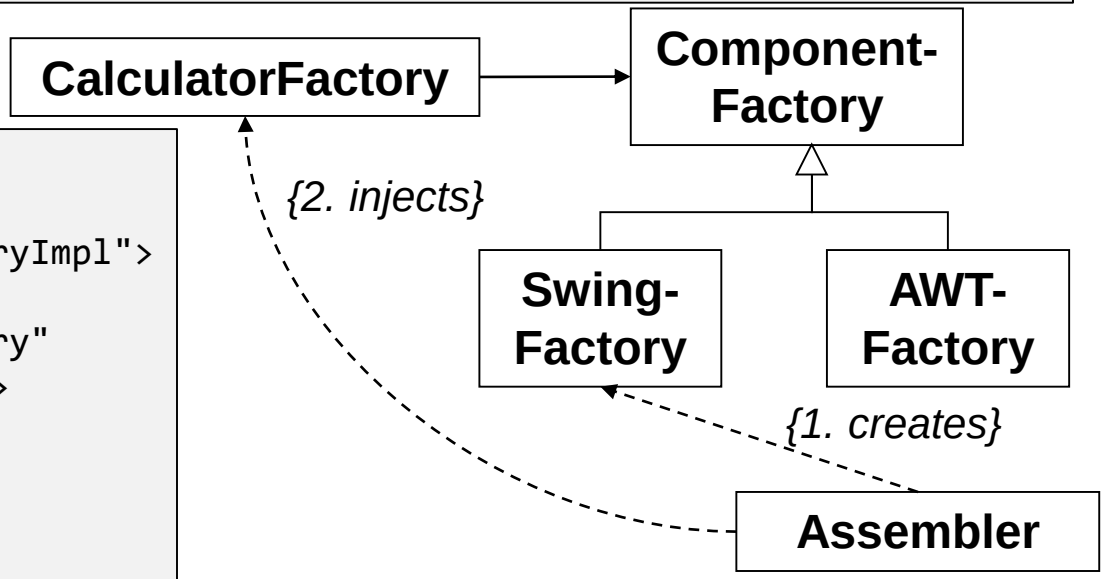



- **Coupled Code**
  - `CalculatorFactory` class has references to concrete factory instances
  - May be mitigated with runtime arguments

# The Solution – Dependency Injection

```
class CalculatorFactoryImpl {
   private ComponentFactory fact;
   public void setComponentFactory(ComponentFactory fact) {
      this.fact = fact;
   }
```

```
<beans>
   <bean id="calcFactory"
         class="CalculatorFactoryImpl">
      <property
           name="componentFactory"
           ref="componentFact"/>
      </property>
   </bean>
   <bean id="componentFact"
      class="SwingFactory">
   </bean>
</beans>
```



# Property values

- **Strings and Numbers**

```
<property name="title"><value>Foo</value></property>      String
<property name="size"><value>7</value></property>         int
<property name="perc"><value>0.25</value></property>      double
<property name="visible"><value>true</value></property>   boolean
```

- **Null values**

```
<property name="tool"><null/></property>
```

- **Other beans**

```
<property name="tool">
   <ref bean = "oval-factory"/>
</property>
```

# Property values

- **<list>**
  - java.util.List / array

```
<property name="listProperty">
 <list>
   <value>string value</value>
   <ref bean="foo"/>
   <ref bean="bar"/>
 </list>
</property>
```

- **<set>**
  - java.util.Set

```
<property name="setProperty">
 <set>
   <value>string value</value>
   <ref bean="foo"/>
   <ref bean="bar"/>
 </set>
</property>
```

- **<map>**
  - java.util.Map

```
<property name="mapProperty">
 <map>
   <entry key="key">
     <value><ref bean="foo"/></value>
   </entry>
 </map>
</property>
```

- **<props>**
  - java.util.Properties

```
<property name="propsProperty">
 <props>
   <prop key="key1">bar1</prop>
   <prop key="key2">bar2</prop>
 </props>
</property>
```

# Spring in Action

```
<beans>
  <bean id="pojo-1">
    <property
        name="p2"
        ref="pojo-2"/>
  </bean>
  <bean id="pojo-2">
    <property
        name="p3"
        ref="pojo-3"/>
  </bean>
  <bean id="pojo-3"/>
<beans>
```