Factory Patterns

The general purpose of the Factory Patterns is to make the program code independent of concrete classes. An instantiation with the `new` operator also causes an explicit dependency to the class of the generated object. Since `new` operators can be spread throughout the entire code, exchanging a class implies a big effort.

Factories help to create objects centrally. In addition, factories can be directed with relatively little effort to create objects from other classes.

Factory Method

Goal

Provide an interface or an abstract method for creating objects. Concrete subclasses implementing the interface or the abstract method then decide which concrete objects are created. *Factory Method* allows to outsource instantiation to specific classes.

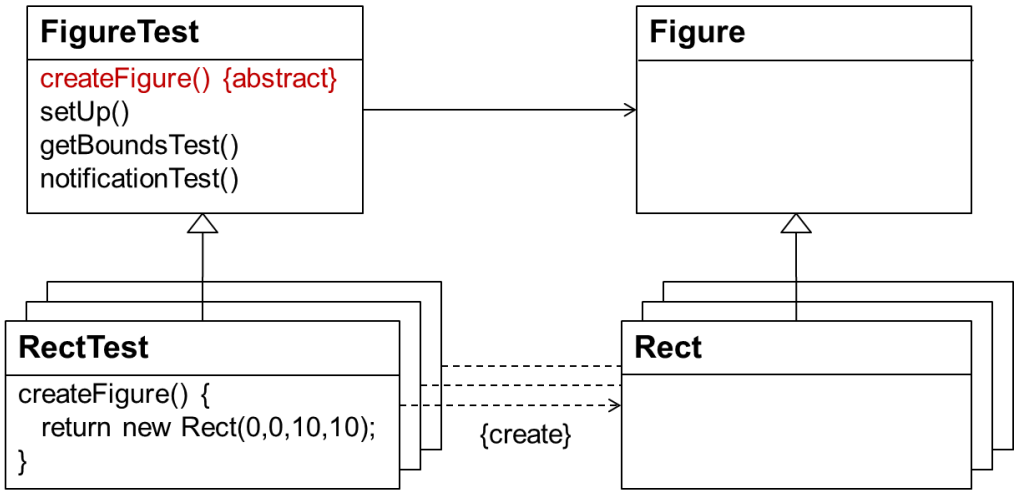
Motivation

JUnit test for figures (e.g., Information Hiding Test). A test class should be written, which checks for any figure whether the figure is sufficiently encapsulated, i.e. if the rectangle that was returned by method `getBounds` is changed, this should have no effect on the position and/or size of the figure.

Solutions

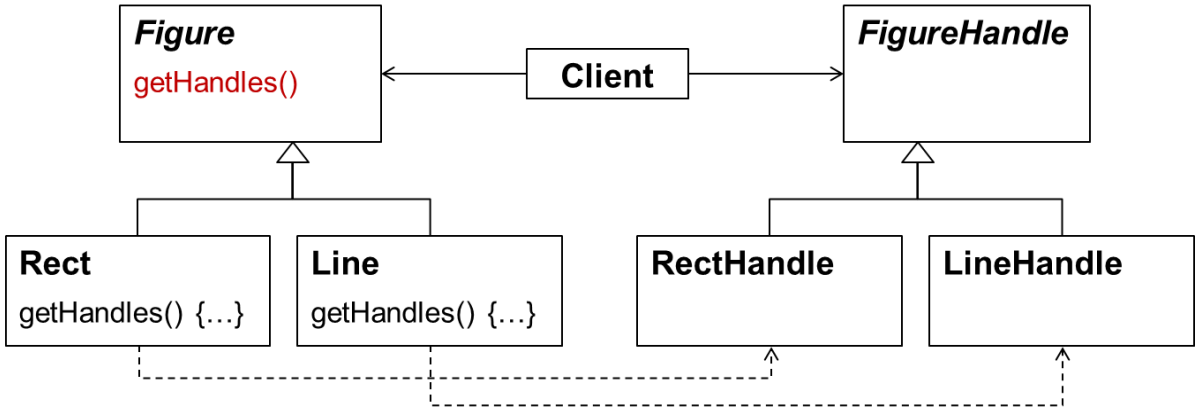
- Add a constructor to the test class which can be used to pass the figure to be tested
  - ⇨ JUnit classes are created using a default constructor
  - ⇨ There exists the possibility of parameterized tests in JUnit. This works for JUnit-Tests but it is not a general solution as only one instance can be passed this way.

- Factory method



Consequences

- The test class is independent of concrete classes! The factory method is an extension point into which concrete subclasses can hook in.
- This approach is also used in parallel class hierarchies to generate instances of corresponding classes.



Implementation Variants

- create-method abstract or with a default behavior
  - Abstract factory methods force subclasses to provide their own implementation. They are used where there is no common default object that could be created in the base class. Example: `AbstractTool`, on this abstraction level no default object can be created.
  - Default implementations are usually implemented in concrete base classes. They offer a convenience implementation that can be overridden by subclasses on demand
    Example: `getHandles`

- Parameterized factory method
  Example: `AbstractTool: createFigure(int figureType)`

```
public Figure createFigure(int figureType) {
  if (figureType == RECTANGLE) return new Rectangle();
  if (figureType == CIRCLE) return new Circle();
  //... add other class instatiations
  return null;
}
```

Do you recognize the code smell?

Abstract Factory

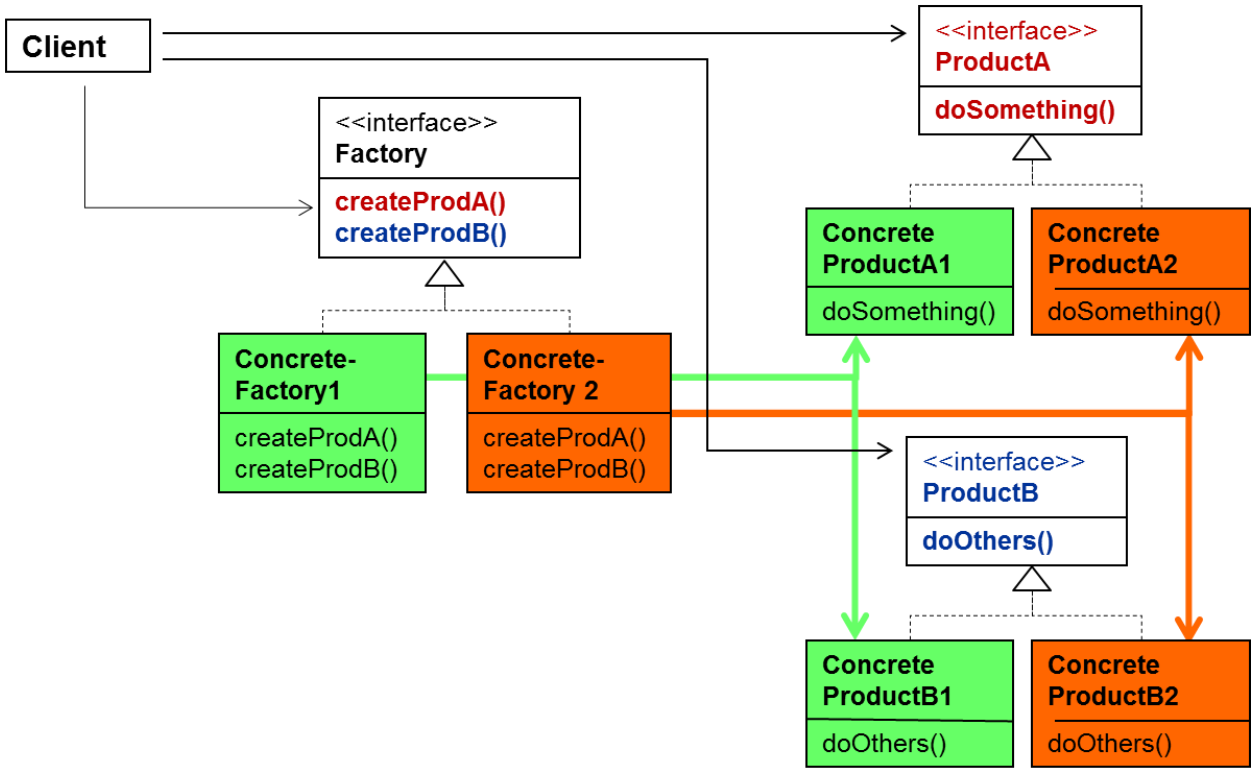
Goal

Provide an interface to create families of related objects.

Motivation

- System should be independent of the generation of objects, i.e. no specific implementations should be referenced.
- System will be configured with a set of concrete classes.
- Extendibility by new implementations.

Structure



Implementation

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

```
interface A {
    // …
}
```

```
interface B {
    // …
}
```

1) Where is the concrete factory stored?
The concrete factory must be accessible to all, i.e. it must be stored in a static class variable.

- If the factory is defined as an abstract class, then this variable can be declared directly in the abstract factory class;
- otherwise a separate class (e.g. `CurrentFactory`) must be defined:

```
public class CurrentFactory {
   private CurrentFactory() { }; // prevents instantiation
   private static Factory fac = null;
   public static Factory getFactory() { return fac; }
   public static void setFactory(Factory f) {
      if (f == null) throw new NullPointerException();
      fac = f;
   }
}
```

Variants:
- Current Factory might provide a default implementation
- Frozen: If the factory can only be set once and not changed afterwards

```
public static void setFactory(Factory f) {
   if (f == null) throw new NullPointerException();
   if (fac != null && f != fac) throw new IllegalStateException();
   fac = f;
}
```

2) How is the concrete factory stored?
Somewhere method `setFactory` must be invoked with a concrete factory!

- Externally: `CurrentFactory.setFactory(new Factory1());`
- Internally: `Factory1.register();`
  Registration method inside the factory which registers a new instance of itself

```
class Factory1 implements Factory {
   public A createA() { /*...*/ }
   public B createB() { /*...*/ }
   static void register() { CurrentFactory.setFactory(new Factory1()); }
   private Factory1() { } // prevents instantiation
}
```

- Automatic registration when the factory is loaded (with the help of a static constructor).

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

With `Class.forName("Factory1")` this class can be loaded and initialized.