Factory Patterns: Factory Method and Abstract Factory

Factory Patterns

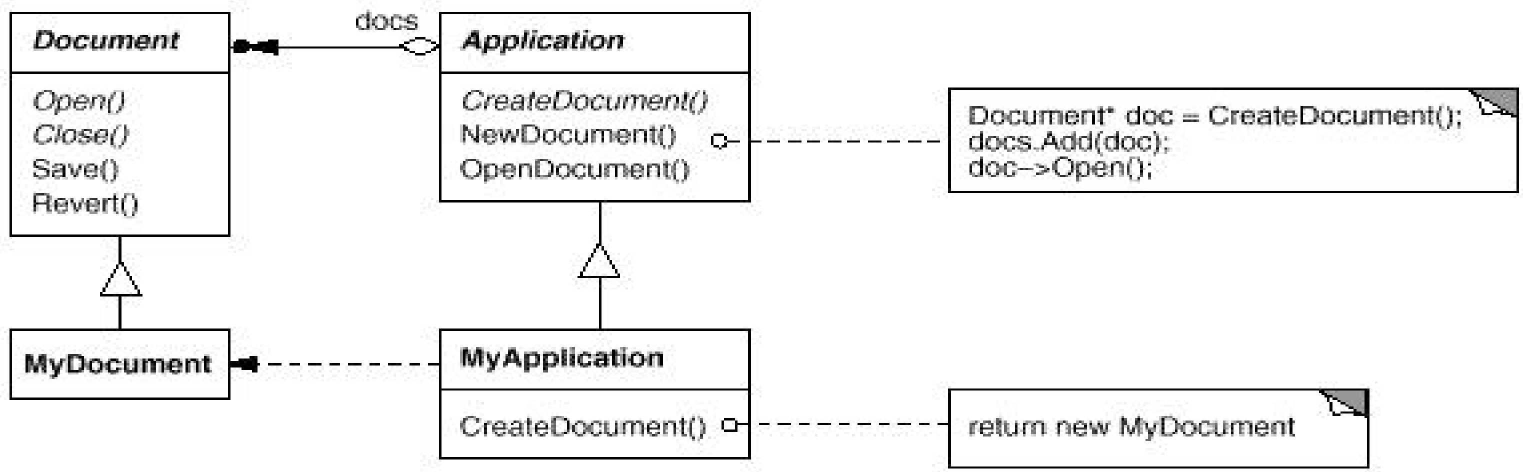
- Factory patterns are examples of creational patterns
- *Creational patterns* abstract the object instantiation process. They hide how objects are created and help make the overall system independent of how its objects are created and composed.
- *Class creational patterns* focus on the use of inheritance to decide the object to be instantiated
  - ⇨ Factory Method
- *Object creational patterns* focus on the delegation of the instantiation to another object
  - ⇨ Abstract Factory

Factory Patterns

- All OO languages have an idiom for object creation. In Java this idiom is the *new* operator. Creational patterns allow us to write methods that create new objects without explicitly using the new operator. This allows us to write methods that can instantiate different objects and that can be extended to instantiate other newly-developed objects, all without modifying the method's code! (Quick! Name the principle involved here!)

The Factory Method Pattern

- Intent
  - ➩ Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
- Motivation
  - ➩ Consider the following framework:



  - ➩ The createDocument() method is a factory method.

The Factory Method Pattern

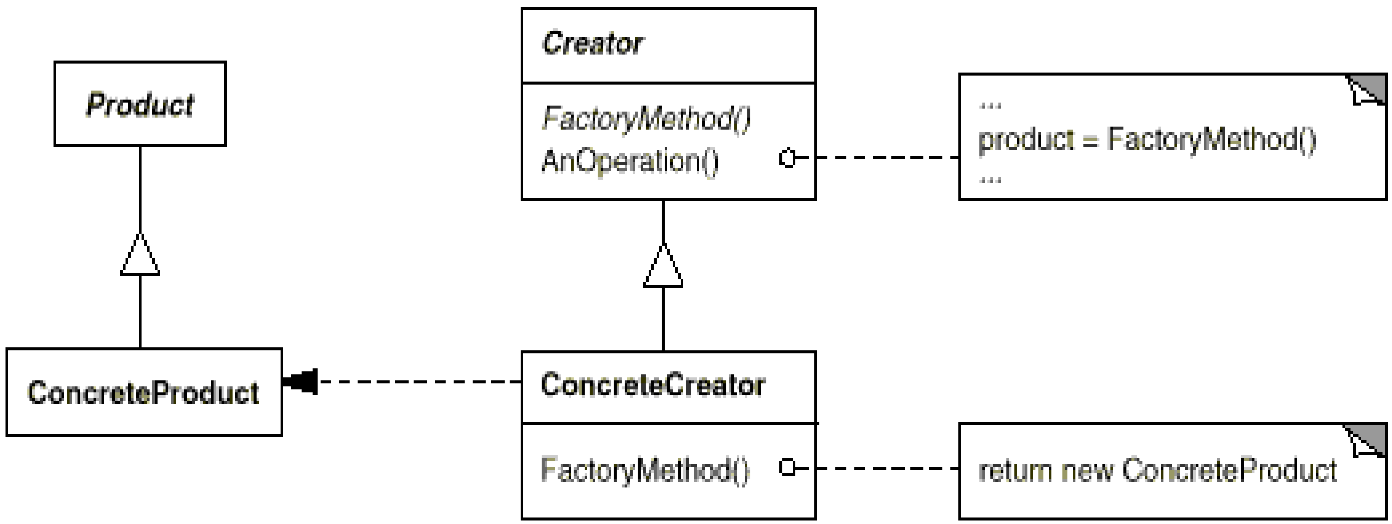
- Applicability

  Use the Factory Method pattern in any of the following situations:

  ➩ A class can't anticipate the class of objects it must create

  ➩ A class wants its subclasses to specify the objects it creates

- Structure



The Factory Method Pattern

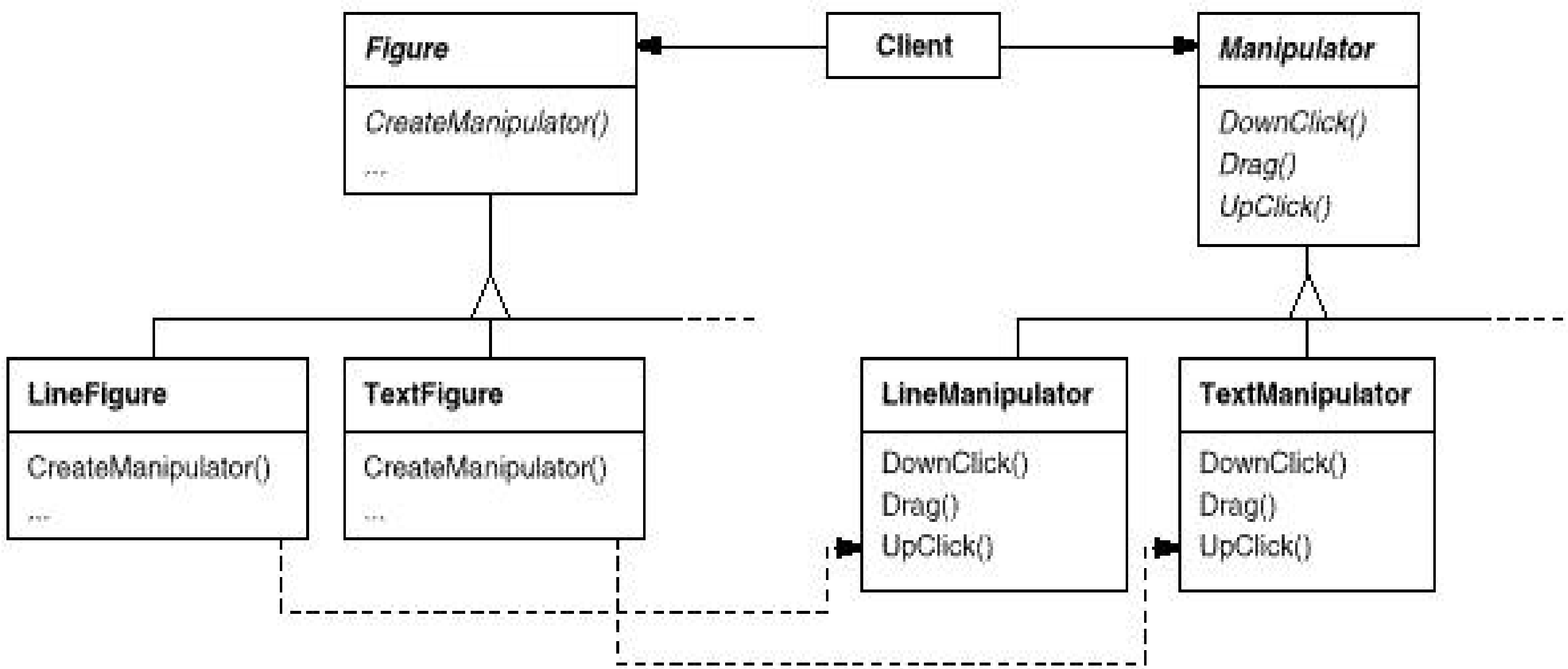
- Participants
  - Product
    - Defines the interface for the type of objects the factory method creates
  - ConcreteProduct
    - Implements the Product interface
  - Creator
    - Declares the factory method, which returns an object of type Product
  - ConcreteCreator
    - Overrides the factory method to return an instance of a ConcreteProduct
- Collaborations
  - Creator relies on its subclasses to implement the factory method so that it returns an instance of the appropriate ConcreteProduct

The Factory Method Pattern

- So what exactly does it mean when we say that "the Factory Method Pattern lets subclasses decide which class to instantiate?"
  - ⇨ It means that Creator class is written without knowing what actual ConcreteProduct class will be instantiated. The ConcreteProduct class which is instantiated is determined solely by which ConcreteCreator subclass is instantiated and used by the application.
  - ⇨ It does *not* mean that somehow the subclass decides at runtime which ConreteProduct class to create

Factory Method Example 1

- Clients can also use factory methods:



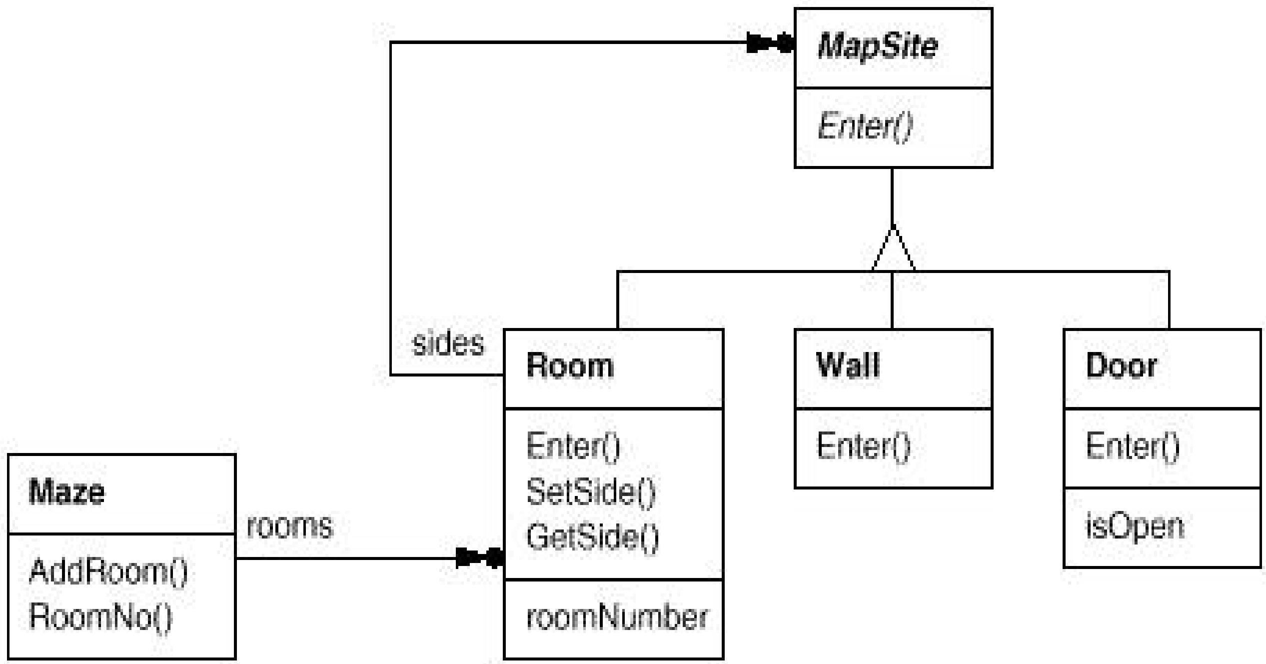
- The factory method in this case is createManipulator()

Factory Method Example 1 (Continued)

- Note that although the Client delegates the creation of a ConcreteManipulator to a ConcreteFigure object, the focus of the pattern is still on how a specific subclass of Figure creates one particular type of Manipulator. So this is still the Factory Method Pattern (a class creational pattern).

Factory Method Example 2

- Consider this maze game:



Factory Method Example 2 (Continued)

- Here's a MazeGame class with a createMaze() method:

```
/**
 * MazeGame.
 */
public class MazeGame {

  // Create the maze.
  public Maze createMaze() {
    Maze maze = new Maze();
    Room r1 = new Room(1);
    Room r2 = new Room(2);
    Door door = new Door(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
```

Factory Method Example 2 (Continued)

```
    r1.setSide(MazeGame.North, new Wall());
    r1.setSide(MazeGame.East, door);
    r1.setSide(MazeGame.South, new Wall());
    r1.setSide(MazeGame.West, new Wall());
    r2.setSide(MazeGame.North, new Wall());
    r2.setSide(MazeGame.East, new Wall());
    r2.setSide(MazeGame.South, new Wall());
    r2.setSide(MazeGame.West, door);
    return maze;
  }

}
```

Factory Method Example 2 (Continued)

- The problem with this createMaze() method is its *inflexibility*.
- What if we wanted to have enchanted mazes with EnchantedRooms and EnchantedDoors? Or a secret agent maze with DoorWithLock and WallWithHiddenDoor?
- What would we have to do with the createMaze() method? As it stands now, we would have to make significant changes to it because of the explicit instantiations using the *new* operator of the objects that make up the maze. How can we redesign things to make it easier for createMaze() to be able to create mazes with new types of objects?

Factory Method Example 2 (Continued)

- Let's add factory methods to the MazeGame class:

```
/**
 * MazeGame with a factory methods.
 */
public class MazeGame {

   public Maze makeMaze() {return new Maze();}

   public Room makeRoom(int n) {return new Room(n);}

   public Wall makeWall() {return new Wall();}

   public Door makeDoor(Room r1, Room r2)
     {return new Door(r1, r2);}
```

Factory Method Example 2 (Continued)

```
    public Maze createMaze() {
      Maze maze = makeMaze();
      Room r1 = makeRoom(1);
      Room r2 = makeRoom(2);
      Door door = makeDoor(r1, r2);
      maze.addRoom(r1);
      maze.addRoom(r2);
      r1.setSide(MazeGame.North, makeWall());
      r1.setSide(MazeGame.East, door);
      r1.setSide(MazeGame.South, makeWall());
      r1.setSide(MazeGame.West, makeWall());
      r2.setSide(MazeGame.North, makeWall());
      r2.setSide(MazeGame.East, makeWall());
      r2.setSide(MazeGame.South, makeWall());
      r2.setSide(MazeGame.West, door);
      return maze;
    }
  }
```

Factory Method Example 2 (Continued)

- We made createMaze() just slightly more complex, but a lot more flexible!
- Consider this EnchantedMazeGame class:

```
public class EnchantedMazeGame extends MazeGame {
  public Room makeRoom(int n) {return new EnchantedRoom(n);}
  public Wall makeWall() {return new EnchantedWall();}
  public Door makeDoor(Room r1, Room r2)
    {return new EnchantedDoor(r1, r2);}
}
```

- The createMaze() method of MazeGame is inherited by EnchantedMazeGame and can be used to create regular mazes or enchanted mazes *without modification*!

Factory Method Example 2 (Continued)

- The reason this works is that the createMaze() method of MazeGame defers the creation of maze objects to its subclasses. That's the Factory Method pattern at work!
- In this example, the correlations are:
  - ➪ Creator => MazeGame
  - ➪ ConcreteCreator => EnchantedMazeGame (MazeGame is also a ConcreteCreator)
  - ➪ Product => MapSite
  - ➪ ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor

The Factory Method Pattern

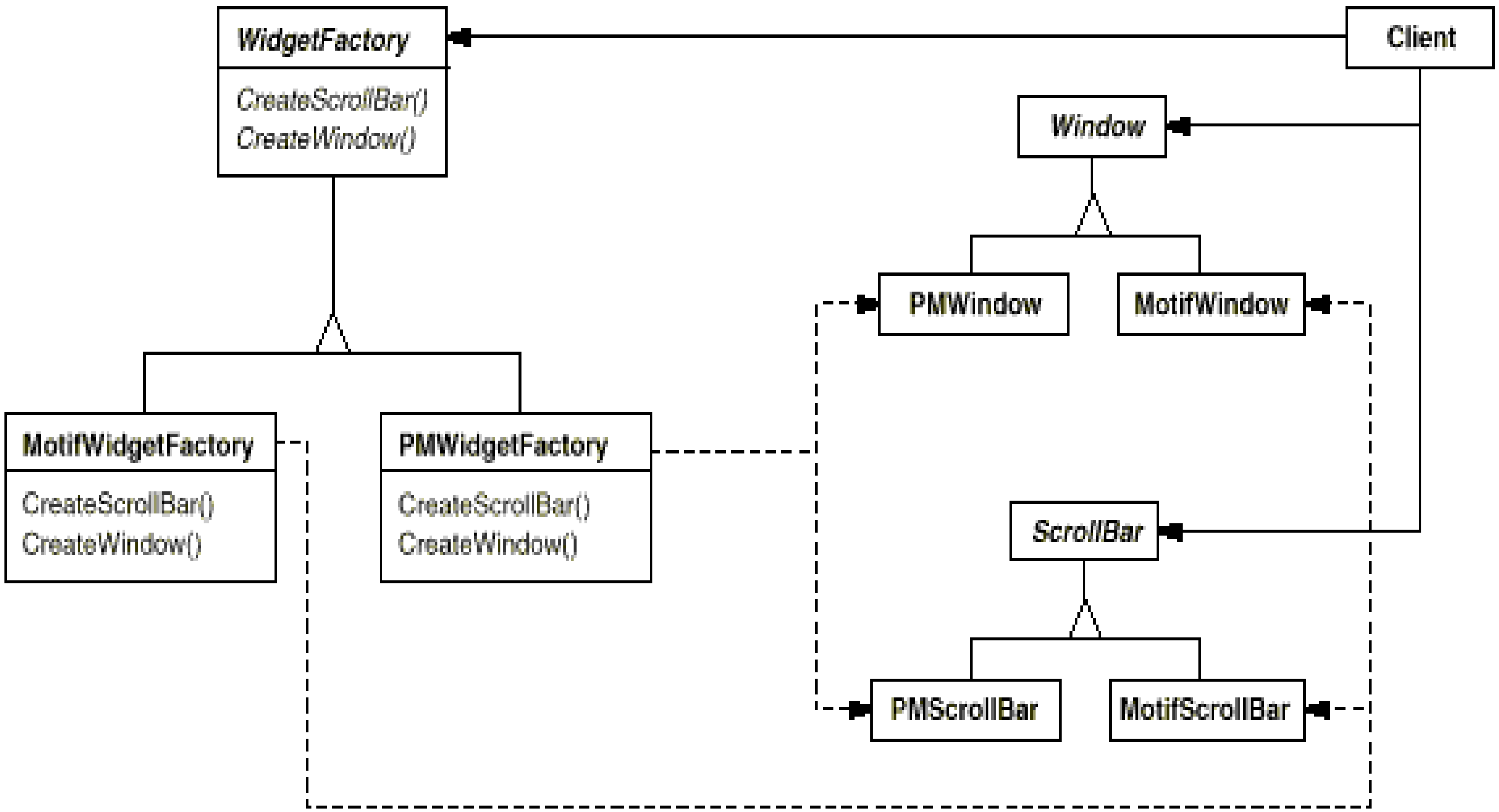
- Consequences
  - Benefits
    - Code is made more flexible and reusable by the elimination of instantiation of application-specific classes
    - Code deals only with the interface of the Product class and can work with any ConcreteProduct class that supports this interface
  - Liabilities
    - Clients might have to subclass the Creator class just to instantiate a particular ConcreteProduct
- Implementation Issues
  - Creator can be abstract or concrete
  - Should the factory method be able to create multiple kinds of products? If so, then the factory method has a parameter (possibly used in an if-else!) to decide what object to create.

The Abstract Factory Pattern

- Intent
  - ⇨ Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - ⇨ The Abstract Factory pattern is very similar to the Factory Method pattern. One difference between the two is that with the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object via composition whereas the Factory Method pattern uses inheritance and relies on a subclass to handle the desired object instantiation.
  - ⇨ Actually, the delegated object frequently uses factory methods to perform the instantiation!

The Abstract Factory Pattern

- Motivation
  - ⇨ A GUI toolkit that supports multiple look-and-feels:



The Abstract Factory Pattern

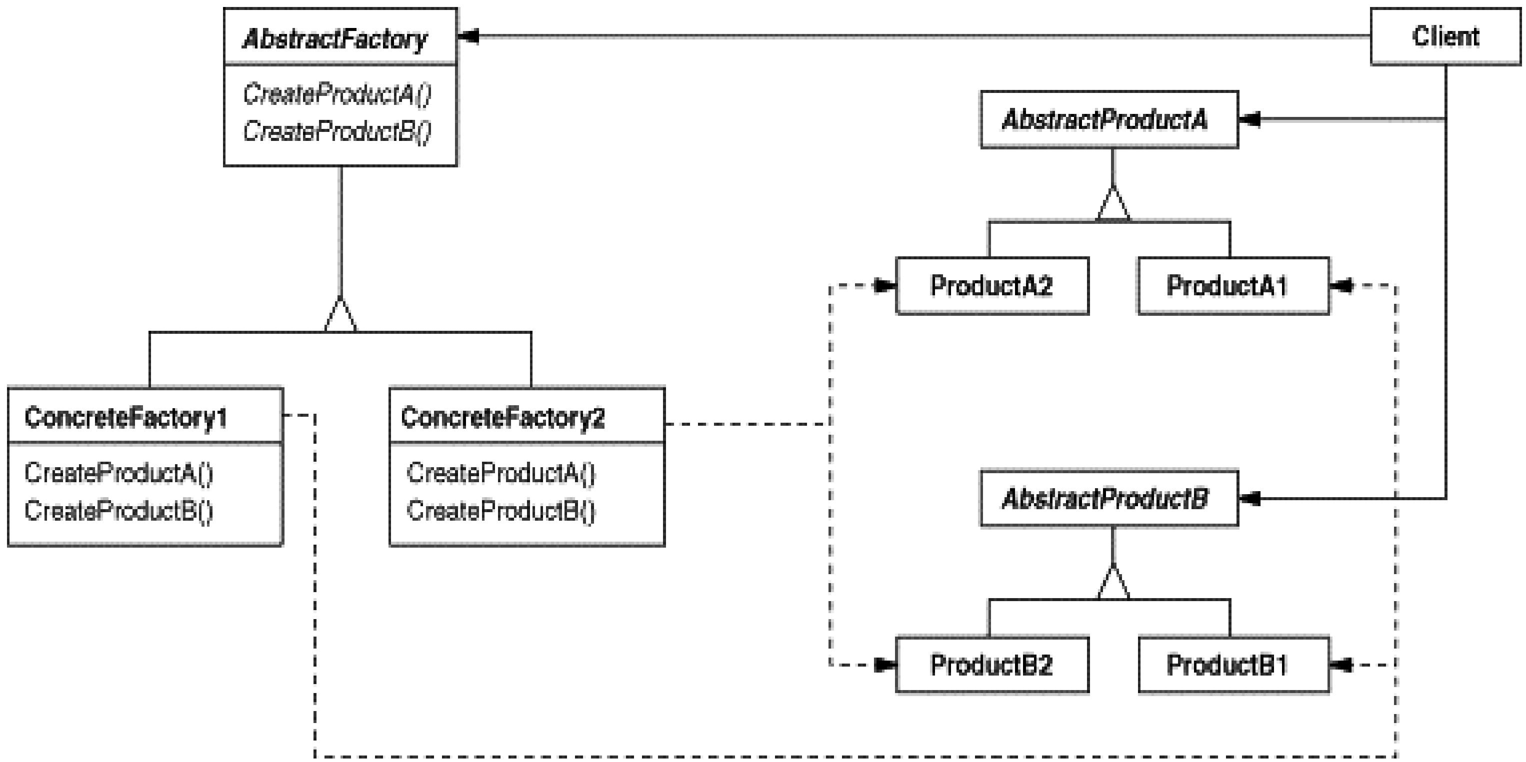
- Applicability

  Use the Abstract Factory pattern in any of the following situations:

  - ⇨ A system should be independent of how its products are created, composed, and represented
  - ⇨ A class can't anticipate the class of objects it must create
  - ⇨ A system must use just one of a set of families of products
  - ⇨ A family of related product objects is designed to be used together, and you need to enforce this constraint

The Abstract Factory Pattern

- Structure



The Abstract Factory Pattern

- Participants
  - AbstractFactory
    - Declares an interface for operations that create abstract product objects
  - ConcreteFactory
    - Implements the operations to create concrete product objects
  - AbstractProduct
    - Declares an interface for a type of product object
  - ConcreteProduct
    - Defines a product object to be created by the corresponding concrete factory
    - Implements the AbstractProduct interface
  - Client
    - Uses only interfaces declared by AbstractFactory and AbstractProduct classes

The Abstract Factory Pattern

- Collaborations
  - ⇨ Normally a single instance of a ConcreteFactory class is created at run-time. (This is an example of the Singleton Pattern.) This concrete factory creates product objects having a particular implementation. To create different product objects, clients should use a different concrete factory.
  - ⇨ AbstractFactory defers creation of product objects to its ConcreteFactory

Abstract Factory Example 1

- Let's see how an Abstract Factory can be applied to the MazeGame
- First, we'll write a MazeFactory class as follows:

```
// MazeFactory.
public class MazeFactory {
  public Maze makeMaze() {return new Maze();}
  public Room makeRoom(int n) {return new Room(n);}
  public Wall makeWall() {return new Wall();}
  public Door makeDoor(Room r1, Room r2) {
    return new Door(r1, r2);}
}
```

- Note that the MazeFactory class is just a collection of factory methods!
- Also, note that MazeFactory acts as both an AbstractFactory and a ConcreteFactory.

Abstract Factory Example 1 (Continued)

- Now the createMaze() method of the MazeGame class takes a MazeFactory reference as a parameter:

```
public class MazeGame {

  public Maze createMaze(MazeFactory factory) {
    Maze maze = factory.makeMaze();
    Room r1 = factory.makeRoom(1);
    Room r2 = factory.makeRoom(2);
    Door door = factory.makeDoor(r1, r2);
    maze.addRoom(r1);
    maze.addRoom(r2);
    r1.setSide(MazeGame.North, factory.makeWall());
    r1.setSide(MazeGame.East, door);
```

Abstract Factory Example 1 (Continued)

```
    r1.setSide(MazeGame.South, factory.makeWall());
    r1.setSide(MazeGame.West, factory.makeWall());
    r2.setSide(MazeGame.North, factory.makeWall());
    r2.setSide(MazeGame.East, factory.makeWall());
    r2.setSide(MazeGame.South, factory.makeWall());
    r2.setSide(MazeGame.West, door);
    return maze;
  }

}
```

- Note how createMaze() delegates the responsibility for creating maze objects to the MazeFactory object

Abstract Factory Example 1 (Continued)

- We can easily extend MazeFactory to create other factories:

```
public class EnchantedMazeFactory extends MazeFactory {
  public Room makeRoom(int n) {return new EnchantedRoom(n);}
  public Wall makeWall() {return new EnchantedWall();}
  public Door makeDoor(Room r1, Room r2)
    {return new EnchantedDoor(r1, r2);}
}
```

- In this example, the correlations are:
  - AbstractFactory => MazeFactory
  - ConcreteFactory => EnchantedMazeFactory (MazeFactory is also a ConcreteFactory)
  - AbstractProduct => MapSite
  - ConcreteProduct => Wall, Room, Door, EnchantedWall, EnchantedRoom, EnchantedDoor
  - Client => MazeGame

Abstract Factory Example 2

- The Java 1.1 Abstract Window Toolkit is designed to provide a GUI interface in a heterogeneous environment
- The AWT uses an Abstract Factory to generate all of the required peer components for the specific platform being used
- For example, here's part of the List class:

```
public class List extends Component implements ItemSelectable {
...
  peer = getToolkit().createList(this);
...
}
```

- The getToolkit() method is inherited from Component and returns a reference to the factory object used to create all AWT widgets

Abstract Factory Example 2 (Continued)

- Here's the getToolkit() method in Component:

```
public Toolkit getToolkit() {
  // If we already have a peer, return its Toolkit.
  ComponentPeer peer = this.peer;
  if ((peer != null) && ! (peer instanceof
     java.awt.peer.LightweightPeer)){
       return peer.getToolkit();
  }
  // If we are already in a container, return its Toolkit.
  Container parent = this.parent;
  if (parent != null) {
    return parent.getToolkit();
  }
  // Else return the default Toolkit.
  return Toolkit.getDefaultToolkit();
}
```

Abstract Factory Example 2 (Continued)

- And here's the getDefaultToolkit() method in Toolkit:

```
public static synchronized Toolkit getDefaultToolkit() {
  if (toolkit == null)
    String nm = System.getProperty("awt.toolkit",
                                   "sun.awt.motif.MToolkit");
    toolkit = (Toolkit)Class.forName(nm).newInstance();
  }
  return toolkit;
}
```

Abstract Factory Example 3

- Sockets are a very useful abstraction for communication over a network
- The socket abstraction was originally developed at UC Berkeley and is now in widespread use
- Java provides some very nice implementations of Berkeley sockets in the Socket and ServerSocket classes in the java.net package
- The Socket class actually delegates all the real socket functionality to a contained SocketImpl object
- And the SocketImpl object is created by a SocketImplFactory object contained in the Socket class
- Sounds like Abstract Factory with just one createProduct() method

Abstract Factory Example 3 (Continued)

- Here’s some code from the Socket class:

```
/**
 * A socket is an endpoint for communication between two
 * machines.  The actual work of the socket is performed by an
 * instance of the SocketImpl class. An application, by changing
 * the socket factory that creates the socket implementation,
 * can configure itself to create sockets appropriate to the
 * local firewall.
 */
public class Socket {

  // The implementation of this Socket.
  SocketImpl impl;

  // The factory for all client sockets.
  private static SocketImplFactory factory;
```

Abstract Factory Example 3 (Continued)

```
/**
 * Sets the client socket implementation factory for the
 * application. The factory can be specified only once.
 * When an application creates a new client socket, the
 * socket implementation factory's createSocketImpl method
 * is called to create the actual socket implementation.
 */
public static synchronized void
  setSocketImplFactory(SocketImplFactory fac)
  throws IOException {
  if (factory != null) {
    throw new SocketException("factory already defined");
  }
  factory = fac;
}
```

Abstract Factory Example 3 (Continued)

```
  /**
   * Creates an unconnected socket, with the
   * system-default type of SocketImpl.
   */
  protected Socket() {
    impl = (factory != null) ? factory.createSocketImpl() :
      new PlainSocketImpl();
  }

  /**
   * Returns the address to which the socket is connected.
   */
  public InetAddress getInetAddress() {
    return impl.getInetAddress();
  }
  // Other sockets methods are delegated to the SocketImpl
  // object!
}
```

Abstract Factory Example 3 (Continued)

- SocketImplFactory is just an interface:

```
public interface SocketImplFactory {
  SocketImpl createSocketImpl();
}
```

- SocketImpl is an abstract class:

```
/**
 * The abstract class SocketImpl is a common superclass
 * of all classes that actually implement sockets.
 * A "plain" socket implements these methods exactly as
 * described, without attempting to go through a firewall or
 * proxy.
 */
public abstract class SocketImpl implements SocketOptions {
  // Details omitted.
}
```

The Abstract Factory Pattern

- Consequences
  - Benefits
    - Isolates clients from concrete implementation classes
    - Makes exchanging product families easy, since a particular concrete factory can support a complete family of products
    - Enforces the use of products only from one family
  - Liabilities
    - Supporting new kinds of products requires changing the AbstractFactory interface
- 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

The Abstract Factory Pattern

- Implementation Issues
  - 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

How Do Factories Create Products?

- Method 1: Use Factory Methods

```
/**
 * WidgetFactory.
 * This WidgetFactory is an abstract class.
 * Concrete Products are created using the factory methods
 *   implemented by sublcasses.
 */
public abstract class WidgetFactory {
  public abstract Window createWindow();
  public abstract Menu createScrollBar();
  public abstract Button createButton();
}
```

How Do Factories Create Products? (Continued)

```
/**
 * MotifWidgetFactory.
 * Implements the factory methods of its abstract superclass.
 */
public class MotifWidgetFactory
  extends WidgetFactory {

  public Window createWindow() {return new MotifWindow();}
  public ScrollBar createScrollBar() {
    return new MotifScrollBar();}
  public Button createButton() {return new MotifButton();}

}
```

How Do Factories Create Products? (Continued)

- Typical client code: (Note: the client code is the same no matter how the factory creates the product!)

```
...
// Create new factory.
WidgetFactory wf = new MotifWidgetFactory();

// Create a button.
Button b = wf.createButton();

// Create a window.
Window w = wf.createWindow();
...
```

How Do Factories Create Products? (Continued)

- Method 2: Use Factories

```
/**
 * WidgetFactory.
 * This WidgetFactory contains references to factories
 *   (composition!) used to create the Concrete Products.
 *   But it relies on a subclass constructor to create the
 *   appropriate factories.
*/
public abstract class WidgetFactory {
  protected WindowFactory windowFactory;
  protected ScrollBarFactory scrollBarFactory;
  protected ButtonFactory buttonFactory;

  public Window createWindow() {return
    windowFactory.createWindow();}
```

How Do Factories Create Products? (Continued)

```
  public ScrollBar createScrollBar() {return
    scrollBarFactory.createScrollBar();}
  public Button createButton() {return
    buttonFactory.createButton();}
}

/**
 * MotifWidgetFactory.
 * Instantiates the factories used by its superclass.
 */
public class MotifWidgetFactory
  extends WidgetFactory {
  public MotifWidgetFactory() {
    windowFactory = new MotifWindowFactory();
    scrollBarFactory = new MotifScrollBarFactory();
    buttonFactory = new MotifButtonFactory();
  }
}
```

How Do Factories Create Products? (Continued)

- Method 3: Use Factories With No Required Subclasses (Pure Composition)

```
/**
 * WidgetFactory.
 * This WidgetFactory contains reference to factories used
 *   to create Concrete Products. But it does not need to be
 *   subclassed.  It has an appropriate constructor to set
 *   these factories at creation time and mutators to change
 *   them during execution.
 */
public class WidgetFactory {

  private WindowFactory windowFactory;
  private ScrollBarFactory scrollBarFactory;
  private ButtonFactory buttonFactory;
```

How Do Factories Create Products? (Continued)

```
 public WidgetFactory(WindowFactory wf,
                      ScrollBarFactory sbf,
                      ButtonFactory bf) {
   windowFactory = wf;
   scrollBarFactory = sbf;
   buttonFactory = bf;
 }
public void setWindowFactory(WindowFactory wf) {
   windowFactory = wf;
 }

 public void setScrollBarFactory(ScrollBarFactory sbf) {
   scrollBarFactory =sbf;
 }

 public void setButtonFactory(ButtonFactory bf) {
   buttonFactory = bf;
 }
```

How Do Factories Create Products? (Continued)

```
  public Window createWindow() {return
    windowFactory.createWindow();}
  public ScrollBar createScrollBar() {return
    scrollBarFactory.createScrollBar();}
  public Button createButton() {return
    buttonFactory.createButton();}
}
```