CptS 487
Software Design and Architecture

Lesson 3

Design Patterns 2:

Abstract Factory

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

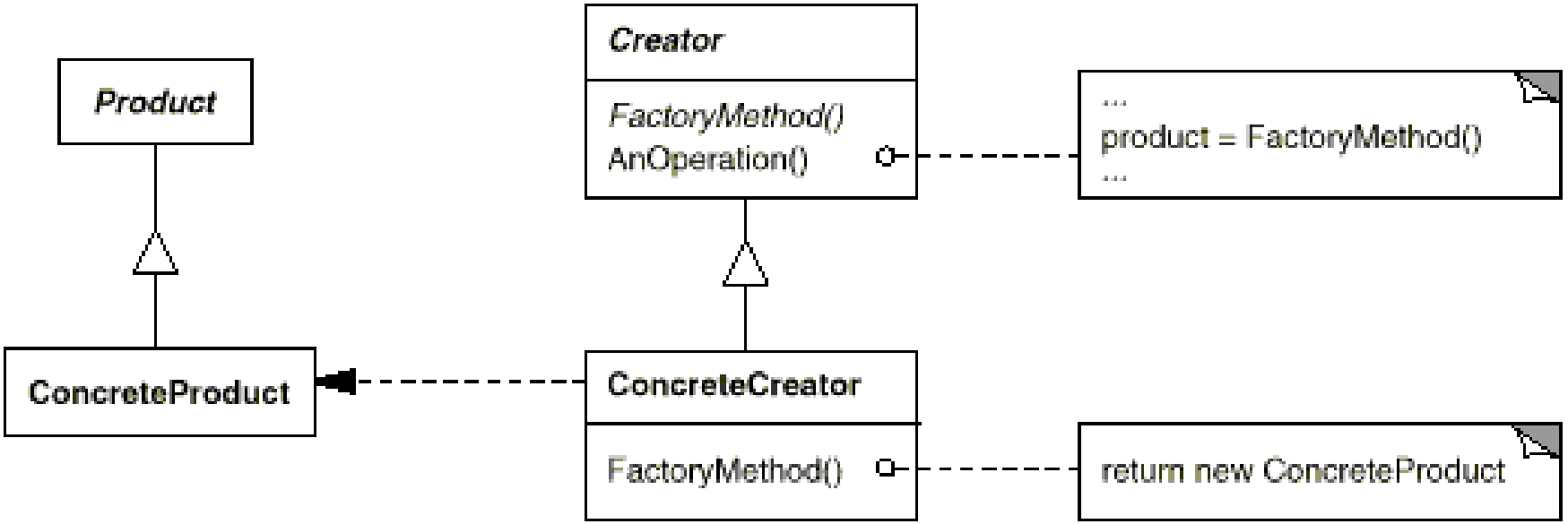
Instructor: Bolong Zeng

Before you start…

- Read the AbstractFactoryPattern_handout.pdf file before proceeding with these slides.

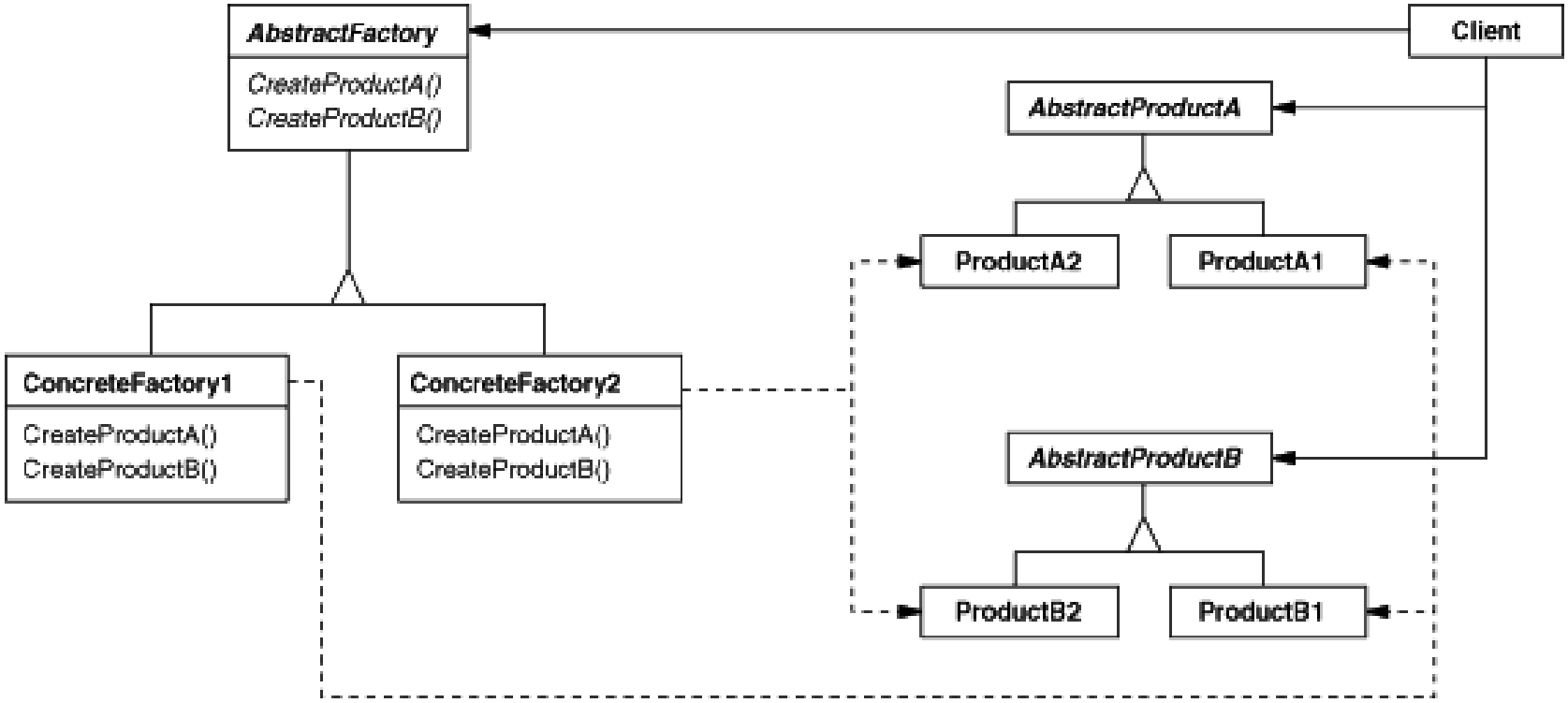
1. Factory Method

- Structure



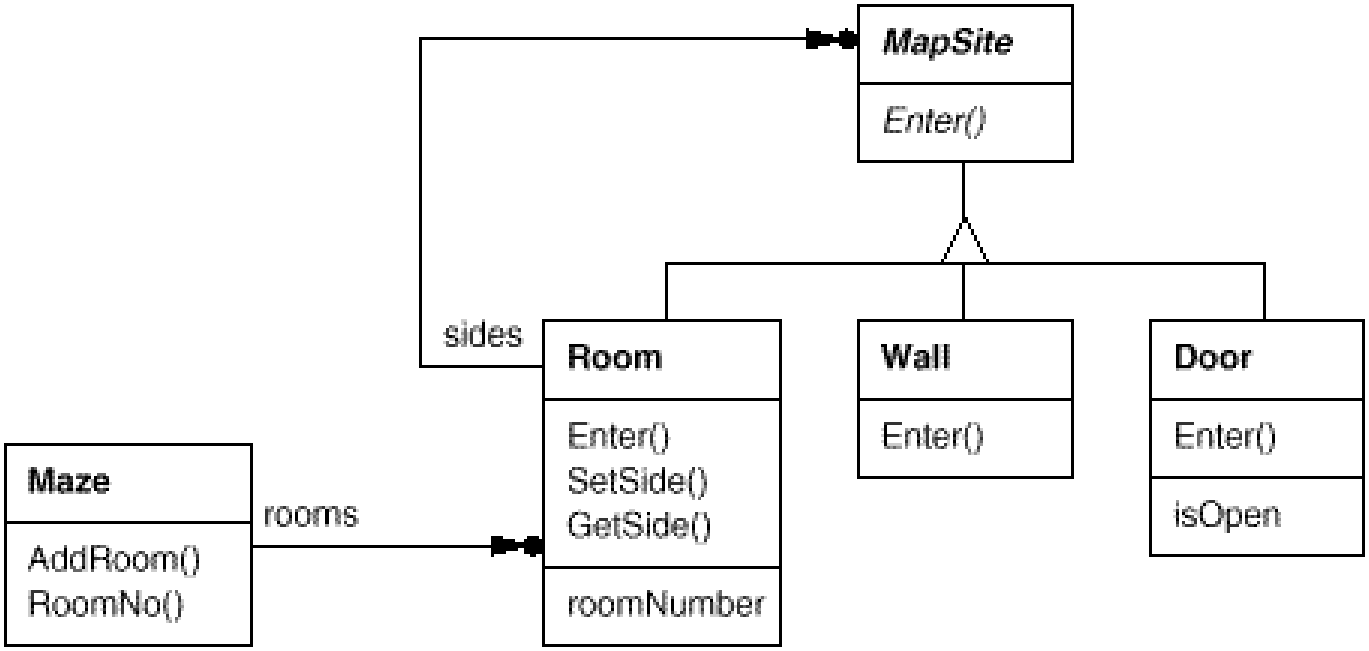
2. Abstract Factory

- Structure



Abstract Factory Example 1 - Maze

- The maze game:



Abstract Factory Example 1 - Maze

- The goal is to insulate client code from object creation (creation of maze, walls, doors, rooms) by having clients ask a factory object to create an object of the desired abstract type and to return an abstract pointer to the object.

Abstract Factory Method Example - 1

- 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);
    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;
  }
}
```

Create a new Maze

Create new Rooms

Create new Door

Create new Walls

Abstract Factory Example 1 - Maze

```
/*** MazeGame.*/
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);}

  // Create the maze with the Factory methods
  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 methods

Abstract Factory Example 1

- First, we'll write a `MazeFactory` class as follows:

- MazeFactory acts as both an AbstractFactory and a ConcreteFactory

```
// 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!

Abstract Factory Example 1

- 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);}
}

public class BombedMazeFactory extends MazeFactory {
    public Room makeRoom(int n)
              {return new RoomWithABomb(n);}
    public Wall makeWall()
              {return new BombedWall();}
}
```

Abstract Factory Example 1 - Maze

```
/*** MazeGame.*/
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);}

  // Create the maze with the Factory methods
  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 methods

Abstract Factory Example 1

- 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);
     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**

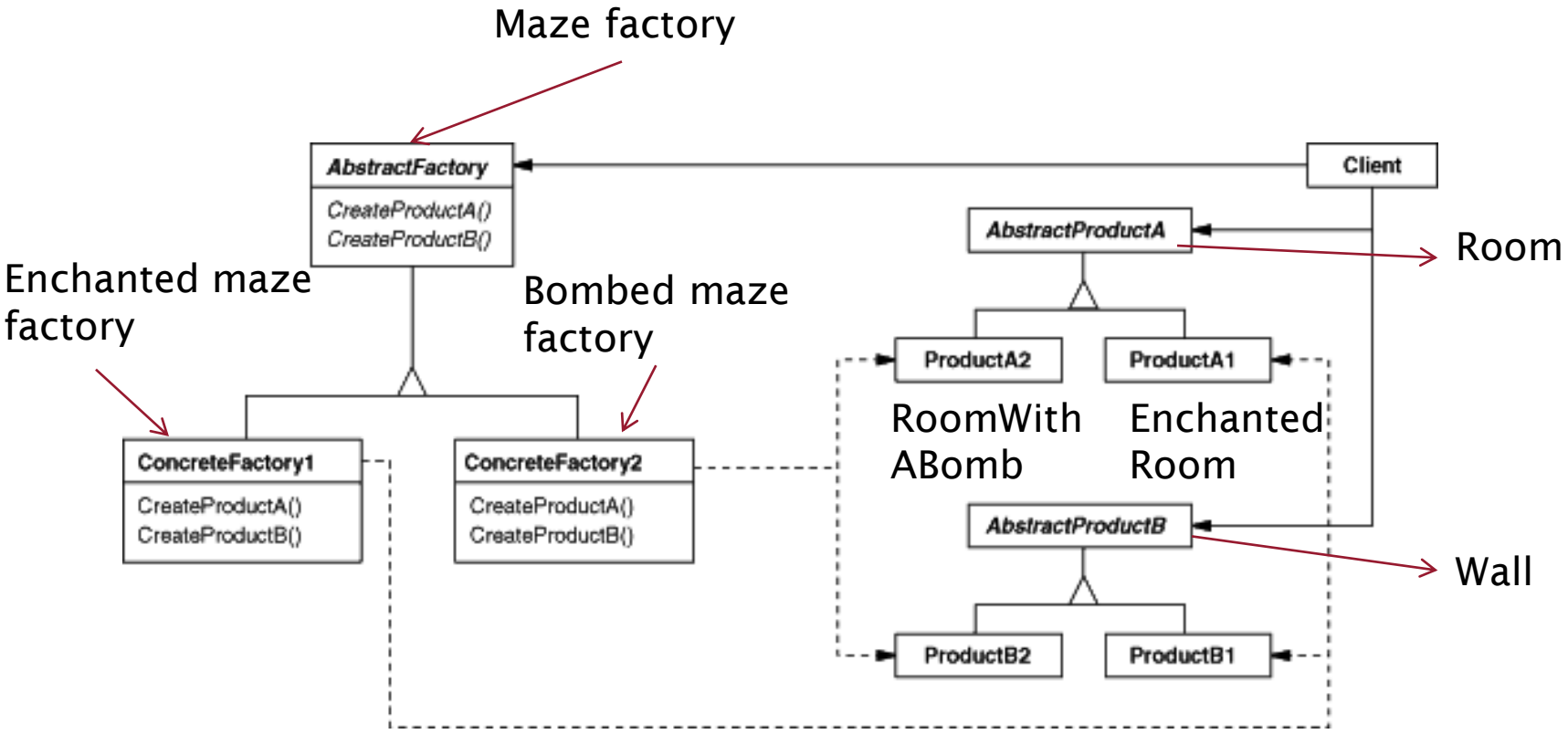
Abstract Factory Example 1

- To build a simple maze that can contain bombs, we simply call `CreateMaze` with a `BombedMazeFactory`

```
MazeGame game;

BombedMazeFactory bFactory;
game.CreateMaze(bFactory);

EnchantedMazeFactory eFactory;
game.CreateMaze(eFactory);
```

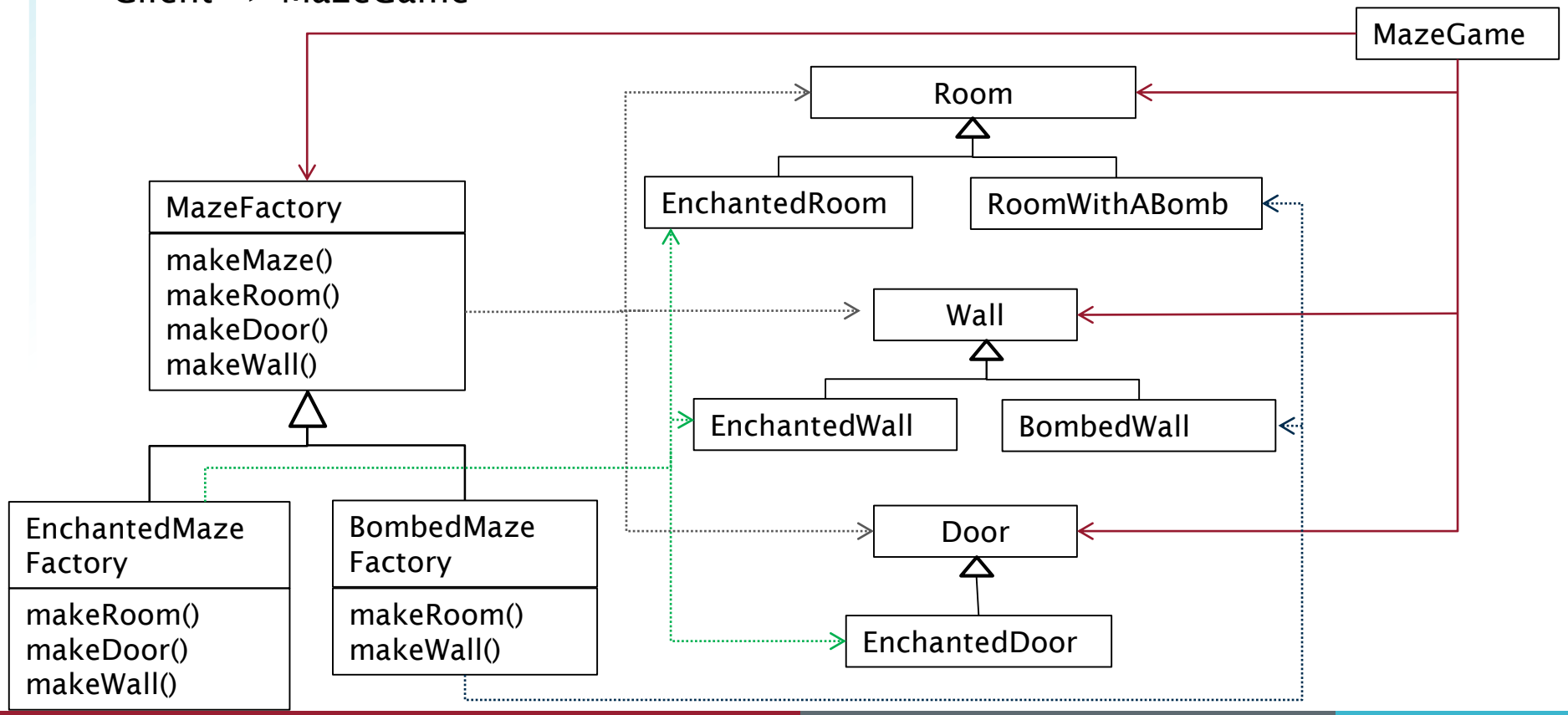
Abstract Factory Example 1 - Maze

- The goal is to insulate client code from object creation (creation of maze, walls, doors, rooms) by having clients ask a factory object to create an object of the desired abstract type and to return an abstract pointer to the object.



Abstract Factory Example 1

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



Questions to think about

- While using the Factory Method, what is the “Product”?

- While using the Abstract Factory, what is(are) the “Product(s)”?

Questions to think about

- MapSites vs. Room/Wall/Door/…
  - Has to be uniformed vs. More flexibility

Questions to think about

- Which class initiates the “creation” of the “products”?
  - In Factory Method?
  - In Abstract Factory?

- In Factory Method
  - It’s the factory/creator itself! (*Creator*)
  - Which means it’s harder to change your mind later.

- In Abstract Factory
  - It’s the client

Questions to think about

- Final question, in Abstract Factory, does the client know about the “exact type” of product it receives?

- And does it need to know?

Detailed Explanation

- http://www.oodesign.com/abstract-factory-pattern.html
  - "Factory of factories"
  - This is important information for your last project deliverable.

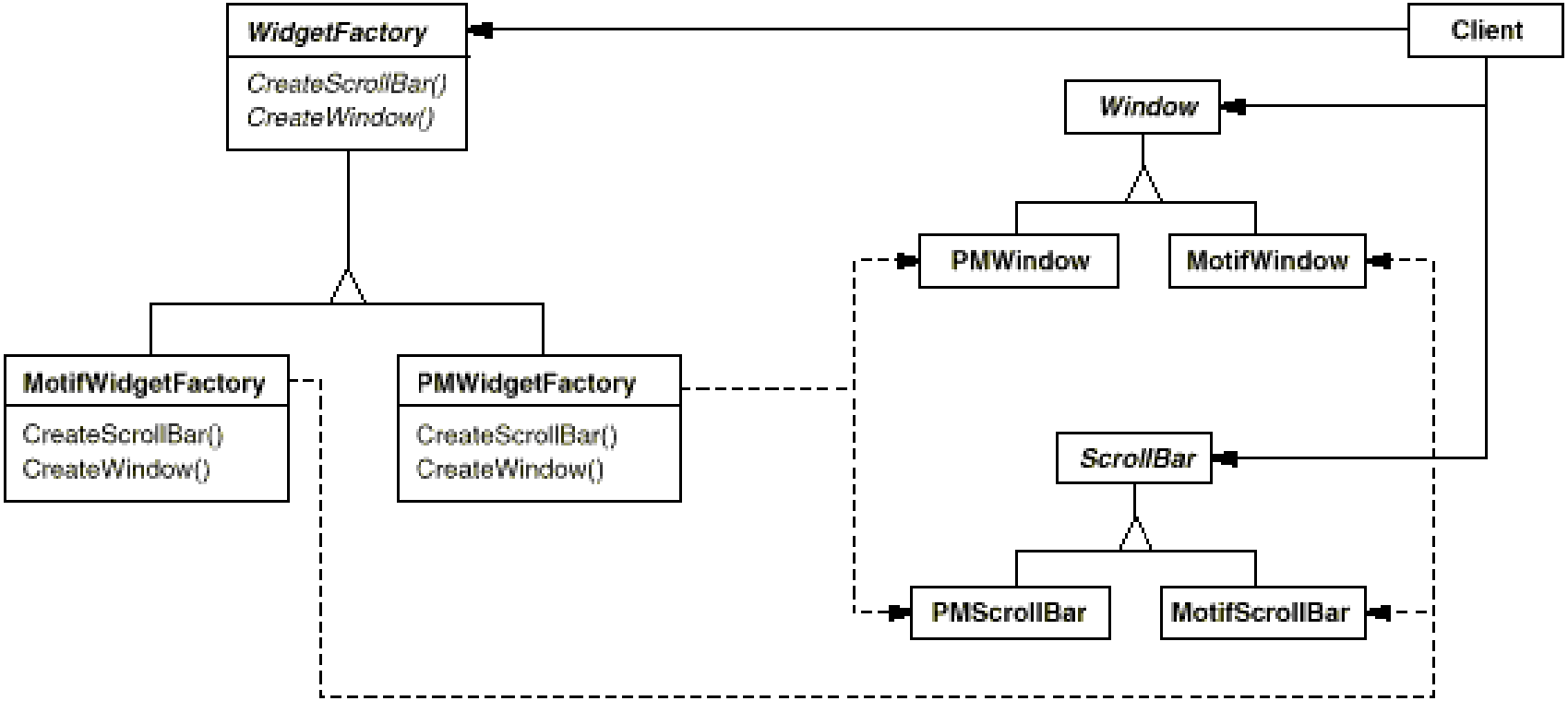
- Feel free to discuss!

2. Abstract Factory (Object creational 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 Factory Method pattern, subclass handle the desired object instantiation by overwriting the factory method (via inheritance)
    - with the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object (via composition)
  - In the Abstract Factory, the delegated object frequently uses factory methods to perform the instantiation!

2. Abstract Factory

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

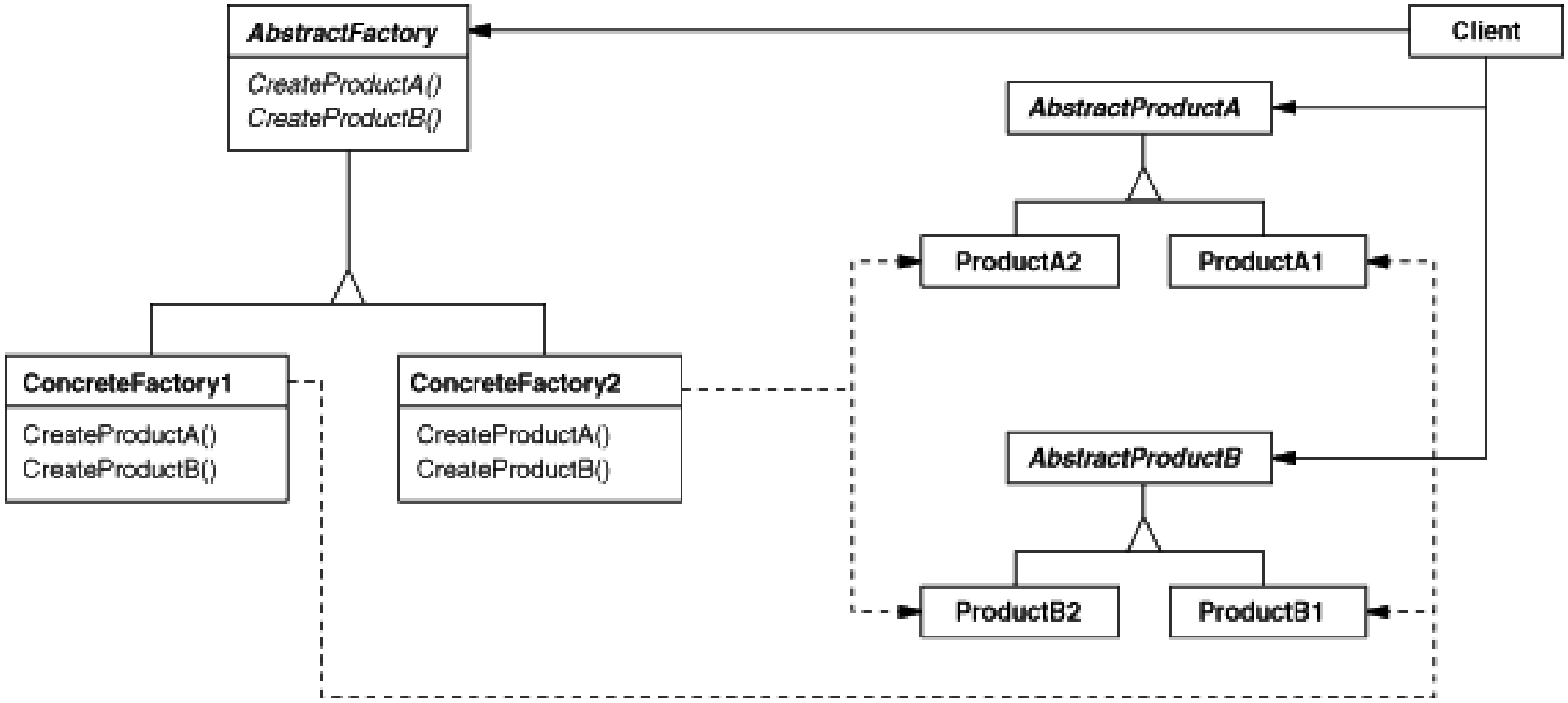


2. Abstract Factory

- 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

2. Abstract Factory

- Structure



2. Abstract Factory

- 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

Abstract Factory

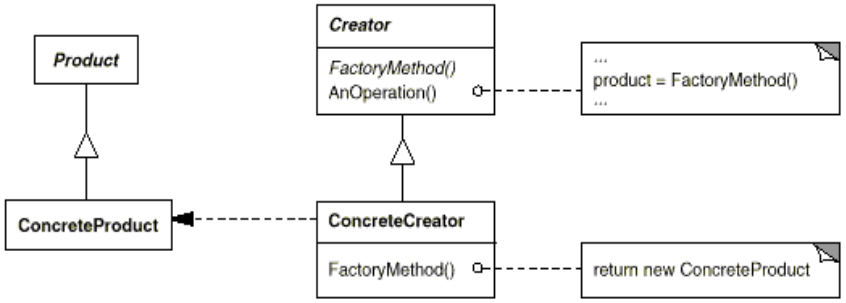
- Consequences
  - Benefits
    - Isolates clients from concrete implementation classes
    - Makes exchanging product families easy, since a particular concrete factory can support a complete family of products
    - Enforces the use of products only from one family
  - Liabilities
    - Supporting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

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

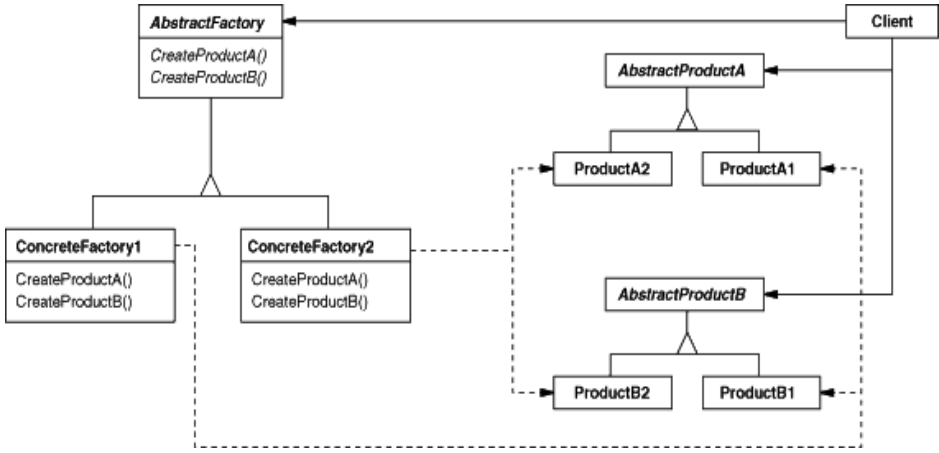
Factory

- Factory creates one product
- The Factory pattern uses inheritance and relies on a subclass to handle the desired object instantiation.



Abstract Factory

- Abstract Factory is used to create a family of related products
- With the Abstract Factory pattern, a class delegates the responsibility of object instantiation to another object (via composition)

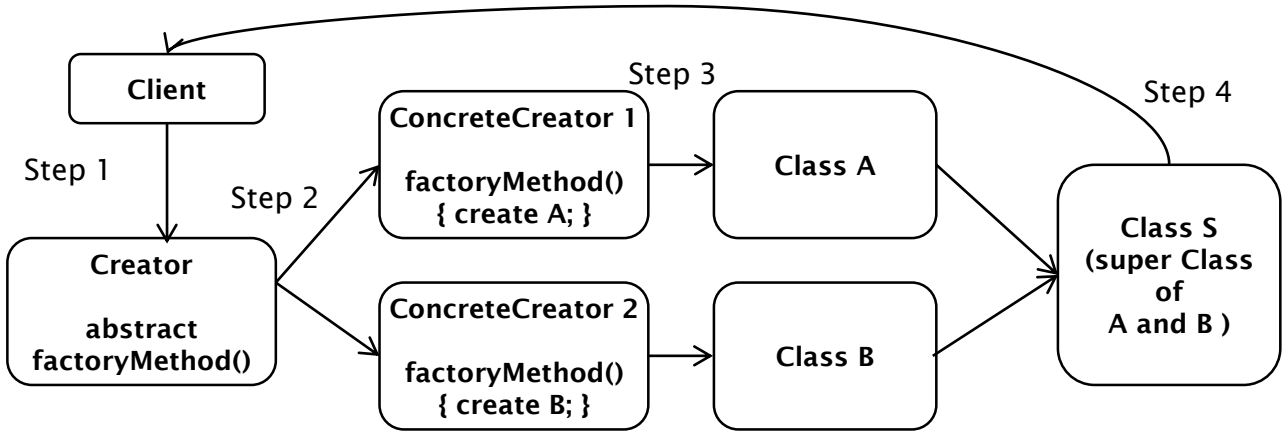


Factory

- Step 1: The client maintains a reference to the abstract Creator, and instantiates it with one of the subclasses, i.e.,

```
Creator c = new ConcreteCreator1();
```

- Step 2: The Creator has an abstract method for creation of an object (factoryMethod()) . It's an abstract method which all subclasses must implement.
- Step 3: The concrete creator creates the concrete object ”Class A".
- Step 4: The concrete object is returned to the client. Note that the client doesn't really know what the type of the object is, just that it is a subclass of the super abstract class S.

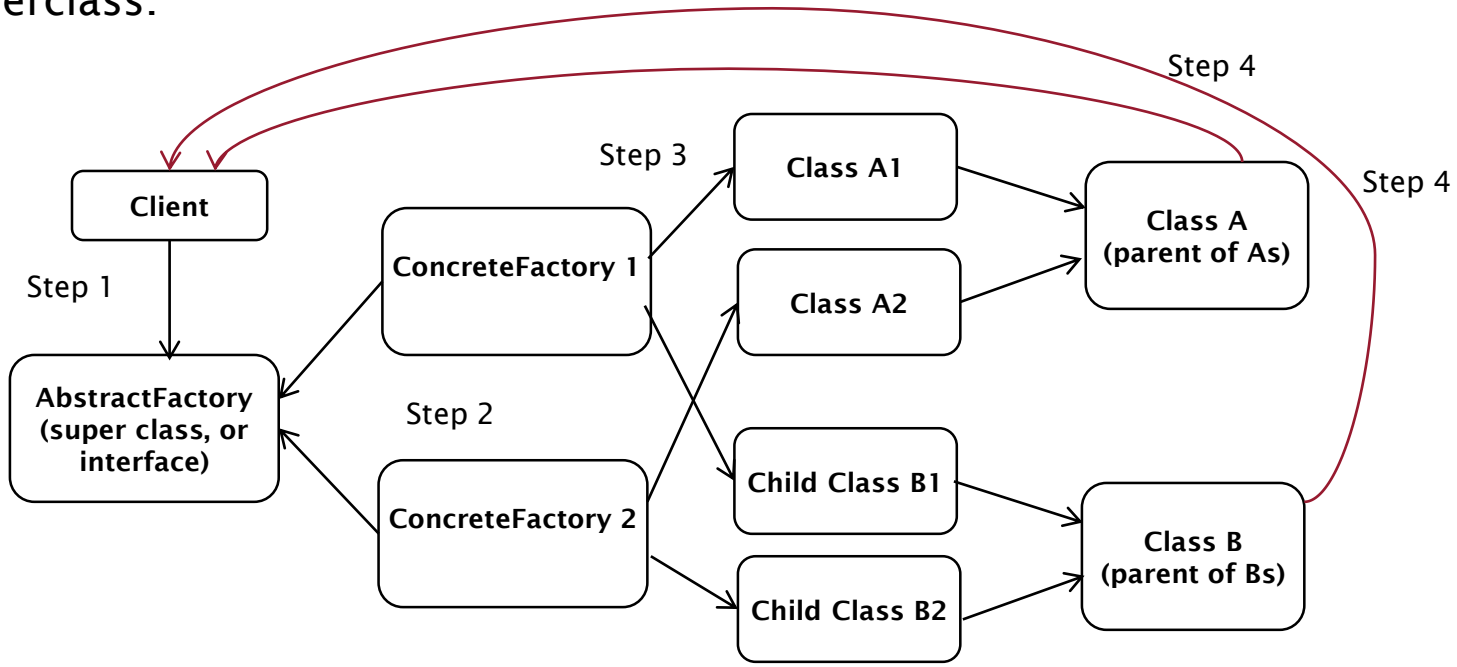


Abstract Factory

- Step 1: The client maintains a reference to an Abstract Factory class, which all Concrete Factories must implement. The abstract Factory is instantiated with a concrete factory, i.e.,

```
ConcreteFactory1 c1;
Client.createAll(c1);
```

- Step 2: The factories are capable of producing multiple object types. This is where the "family of related products" comes into play.
- Step 3: The concrete factory creates the related concrete objects.
- Step 4: The concrete objects are returned to the client. Again, the client doesn't really know what the type of the objects are, just that are a subclasses of the superclass.

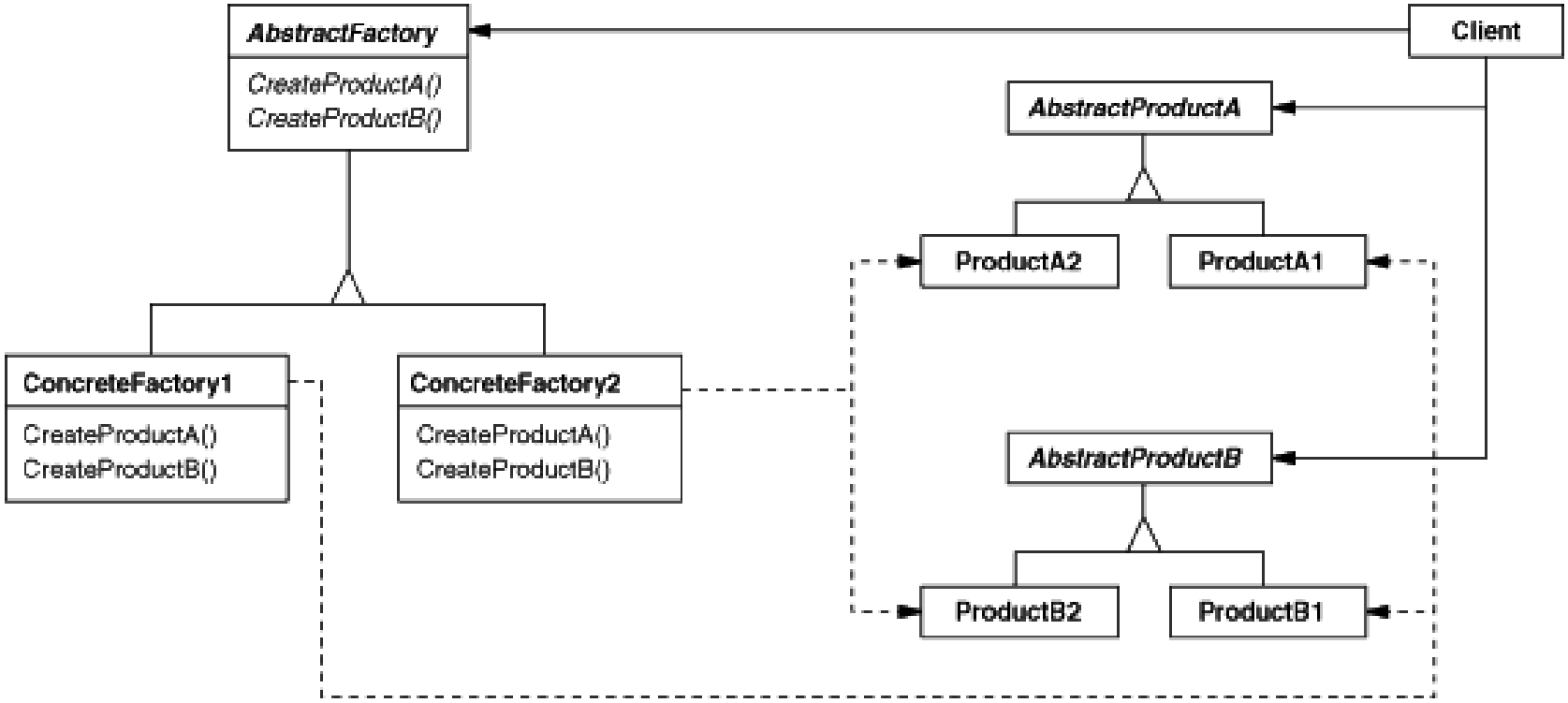


Abstract Factory Example 2 - Sockets

- 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 2 - Sockets

SocketImplFactory



Abstract Factory Example 2

—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 2

- 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 2

— 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 2

- `SocketImplFactory` is just an interface:

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

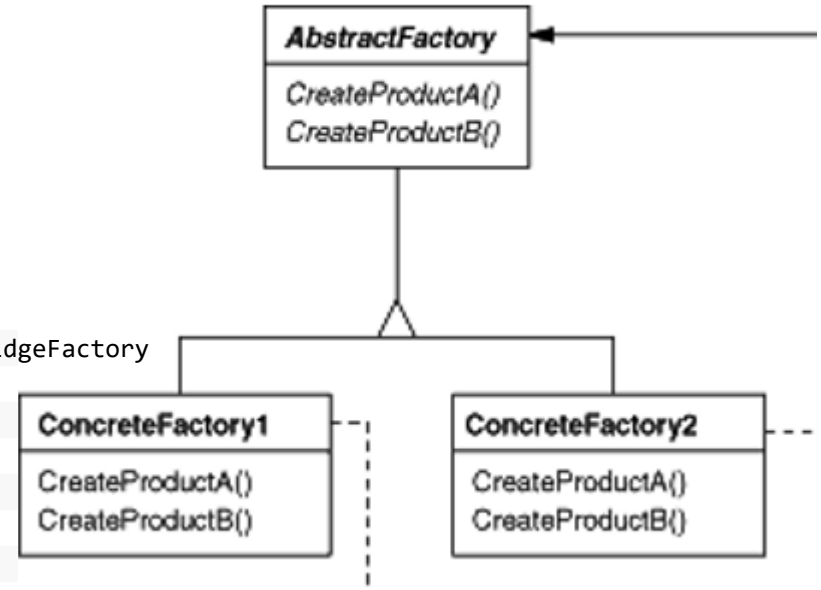
- `SocketImpl` is an abstract class:
  - The abstract class `SocketImpl` is a common superclass of all classes that actually implement sockets. A "plain" socket implements these methods exactly as described.

```
public abstract class SocketImpl implements SocketOptions
  {
 // Details omitted.
 }
```

```java
//Our AbstractProduct
public interface Window
{

   public void setTitle(String text);

   public void repaint();
}
```

```java
//ConcreteProductA1
public class MSWindow implements Window
{




}
```

```java
//ConcreteProductA2
public class MacOSXWindow implements Window
{




}
```




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

```java
//ConcreteFactory1
public class MsWindowsWidgetFactory implements AbstractWidgeFactory
{



}
```

```java
//ConcreteFactory2
public class MacOSXWidgetFactory    implements AbstractWidgeFactory
{



}
```

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

```java
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")
     {

     }
     else
     {

     }

    }
}
```