Abstract Factory Pattern

The **Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Toni Sellarès
Universitat de Girona

Abstract Factory Method Motivation (1)

In the context of an abstract factory there exist:

- Suites or families of related, dependent classes.
- A group of concrete factory classes that implements the interface provided by the abstract factory class.
- Each of these factories controls or provides access to a particular suite of related, dependent objects and implements the abstract factory interface in a manner that is specific to the family of classes it controls.

The Abstract Factory pattern is useful when a client object wants to create an instance of one of a suite of related, dependent classes without having to know which specific concrete class is to be instantiated.

Abstract Factory Method Motivation (2)

An abstract factory provides the necessary interface for creating an appropiate class instance.

For each group or family, a concrete factory is implemented that manages the creation of the objects and the interdependencies and consistency requirements between them.

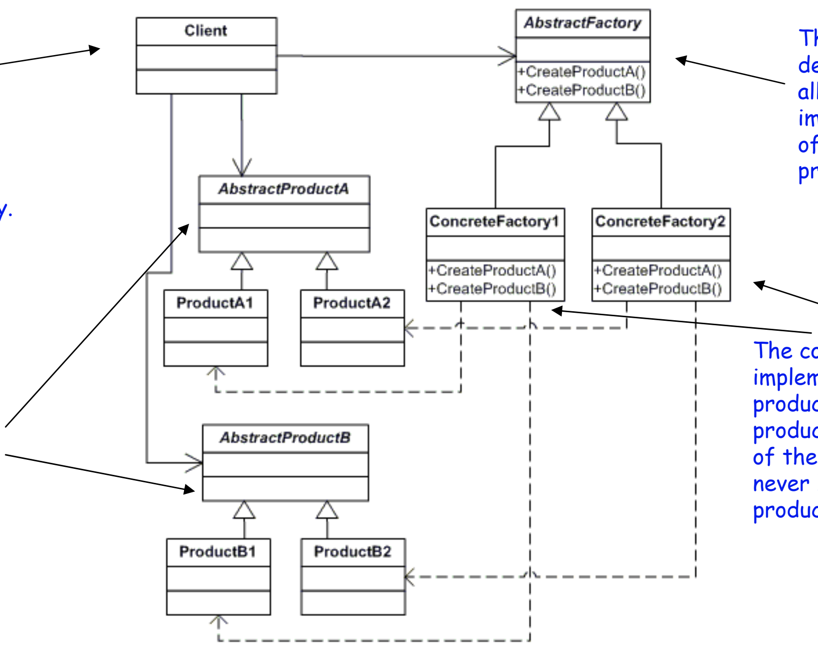
Each concrete factory implements the interface of the abstract factory.

Client objects make use of these concrete factories to create objects and, therefore, do not need to know which concrete class is actually instantiated.

Abstract Factories are usually implemented using the Factory Method pattern.

The Abstract Factory Pattern

The **Abstract Factory Pattern** provides an interface for creating families of related or dependent objects without specifying their concrete classes.



The client is written against the abstract factory and then composed at runtime with an actual factory.

The AbstractFactory defines the interface that all Concrete factories must implement, which consists of a set of methods for producing products.

This is the product family. Each concrete factory can produce an entire set of objects.

The concrete factories implement the different product families. To create a product, the client uses one of these factories, so it never has to instantiate a product object.

Abstract Factory Method Structure

AbstractFactory: declares an interface for operations that create abstract products.

ConcreteFactory: implements the operations to create concrete product objects.

AbstractProduct: declares an interface for a type of product object.

Product: defines a product object to be created by the corresponding concrete factory implements the AbstractProduct interface.

Client: Classes in the Client role use various Product classes to request or receive services from the product that the client is working with. Client classes only know about the abstract Procduct classes. The client has no knowledge of any concrete Product classes.

Abstract Factory Method: Structural Example

```
/**
 * Test class for the abstract factory pattern.
 */
public class Test{
   public static void main( String arg[] ){
         AbstractFactory factory = new ConcreteFactory1();
         AbstractProductA productA = factory.createProductA();
         AbstractProductB productB = factory.createProductB();
   }

}

/**
 * Defines a product object to be created by the corresponsing
 * concrete factory. Implements the AbstractProduct interface.
 */
public class ProductA1 implements AbstractProductA
{
}

/**
 * Defines a product object to be created by the corresponsing
 * concrete factory. Implements the AbstractProduct interface.
 */
public class ProductA2 implements AbstractProductA
{
}
```

```
/**
 * Defines a product object to be created by the corresponsing
 * concrete factory. Implements the AbstractProduct interface.
 */
public class ProductB1 implements AbstractProductB
{
}

/**
 * Defines a product object to be created by the corresponsing
 * concrete factory. Implements the AbstractProduct interface.
 */
public class ProductB2 implements AbstractProductB
{
}
```

```
/**
 * Implements the operations to create concrete product objects.
 */
public class ConcreteFactory1 implements AbstractFactory
{
          public AbstractProductA createProductA() { return new ProductA1(); };
          public AbstractProductB createProductB() { return new ProductB1(); };
}

/**
 * Implements the operations to create concrete product objects.
 */
public class ConcreteFactory2 implements AbstractFactory
{
          public AbstractProductA createProductA() { return new ProductA2(); };
          public AbstractProductB createProductB() { return new ProductB2(); };
}
```

```
/**
 * Declares an interface for a type of product object.
 */
public interface AbstractProductA
{
}

/**
 * Declares an interface for a type of product object.
 */
public interface AbstractProductB
{
}

/**
 * Declares an interface for operations that create
 * abstract product objects.
 */
public interface AbstractFactory
{
          AbstractProductA createProductA();
          AbstractProductB createProductB();
}
```

Example: general purpose gaming environment

Suppose you are creating a general-purpose gaming environment
And you want to be able to support different types of games.

Player objects interact with **Obstacle** objects, but there are different types of players and obstacles depending on what kind of game you're playing.

You determine the kind of game by choosing a particular **GameElementFactory**, and then the **GameEnvironment** controls the setup and play of the game.

```
interface Obstacle {
void action();
}

interface Player {
void interactWith(Obstacle o);
}

class Kitty implements Player {
public void interactWith(Obstacle ob) {
System.out.print("Kitty has encountered a ");
ob.action();
}
}

class KungFuGuy implements Player {
public void interactWith(Obstacle ob) {
System.out.print("KungFuGuy now battles a ");
ob.action();
}
}
```

```
class Puzzle implements Obstacle {
public void action() {
System.out.println("Puzzle");
}
}

class NastyWeapon implements Obstacle {
public void action() {
System.out.println("NastyWeapon");
}
}

// The Abstract Factory:
interface GameElementFactory {
Player makePlayer();
Obstacle makeObstacle();
}
```

```
// Concrete factories:
class KittiesAndPuzzles
implements GameElementFactory {
public Player makePlayer() {
return new Kitty();
}
public Obstacle makeObstacle() {
return new Puzzle();
}
}

class KillAndDismember
implements GameElementFactory {
public Player makePlayer() {
return new KungFuGuy();
}
public Obstacle makeObstacle() {
return new NastyWeapon();
}
}
```

```
class GameEnvironment {
private GameElementFactory gef;
private Player p;
private Obstacle ob;
public GameEnvironment(
GameElementFactory factory) {
gef = factory;
p = factory.makePlayer();
ob = factory.makeObstacle();
}
public void play() { p.interactWith(ob); }
}

public class Games {
GameElementFactory
kp = new KittiesAndPuzzles(),
kd = new KillAndDismember();
GameEnvironment
g1 = new GameEnvironment(kp),
g2 = new GameEnvironment(kd);

public static void main(String args[]) {
Games g = new Games();
g1.play();
g2.play();
}
}
```

Abstract Factory Method Applicability

Use Abstract Factory if:

- A system must be independent of how its products are created.
- A system should be configured with one of multiple families of products.
- A family of related objects must be used together.
- You want to reveal only interfaces of a family of products and not their implementations.