Software Architecture

Design Pattern

Content

- What’s a design pattern?
- Creational Patterns
- Structural Patterns
- Behavioral Patterns

History about design patterns

- Patterns originated as an architectural concept by Christopher Alexander (1977/79).
- In 1987, Kent Beck and Ward Cunningham began to apply patterns to programming and presented their results at the OOPSLA conference that year.
- Design patterns gained popularity in computer science after the book *Design Patterns: Elements of Reusable Object-Oriented Software* was published in 1994 (Gamma et al.).

What is a design pattern?

- In software engineering, a **design pattern** is a general reusable solution to a commonly occurring problem in software design.
  - It is not a finished design that can be transformed directly into code.
  - focuse on a particular object-oriented design problem or issue
  - a description or template for how to solve a problem.

Classification of Design Pattern

- At least 250 existing patterns are used in OO world. The 23 design patterns are well known.
- be divided into three groups:
  - *Creational patterns*
    - create objects, rather than instantiating objects directly.
    - more flexible in deciding which objects need to be created for a given case.

Classification of Design Pattern

- *Structural patterns*
  - compose groups of objects into larger structures, such as complex user interfaces.
- *Behavioral patterns*
  - define the communication between objects in the system and how the flow is controlled in a complex program.

Creational Patterns I

- Provide the best way to create instances of objects.
  - **The Factory Method**
    - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
  - **The Abstract Factory Method**
    - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  - **The Singleton Pattern**
    - Ensure a class has only one instance, and provide a global point of access to it.

Creational Patterns II

- **The Builder Pattern**
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations
- **The Prototype Pattern**
  - starts with an initialized and instantiated class and copies or clones it to make new instances rather than creating new instances.

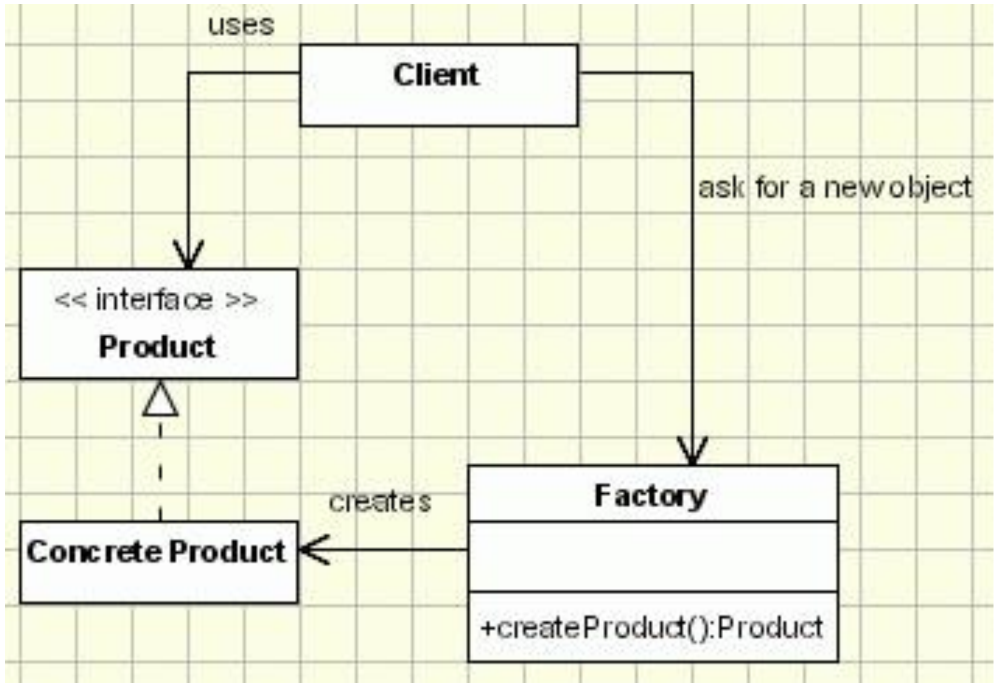
The Factory Pattern

THE FACTORY PATTERN

- The Factory Design Pattern is probably the most used design pattern in modern programming languages like Java and C#.
- It comes in different variants and implementations.
  - Factory Method
  - Abstract Factory.
- **Problem**
  - A framework needs to standardize the architectural model for a range of applications, but allow for individual applications to define their own domain objects and provide for their instantiation.
- **Intent**
  - creates objects without exposing the instantiation logic to the client.
  - refers to the newly created object through a common interface

THE FACTORY PATTERN

- a flavor of factory pattern commonly used



FACTORY METHOD PATTERN

- **Intent**
  - Defines an interface for creating objects, but let subclasses to decide which class to instantiate
  - Refers to the newly created object through a common interface

FACTORY METHOD PATTERN

- **participants**

  The classes and/or objects participating in this pattern are:

  - **Product**
    - defines the interface of objects the factory method creates
  - **ConcreteProduct**
    - implements the Product interface

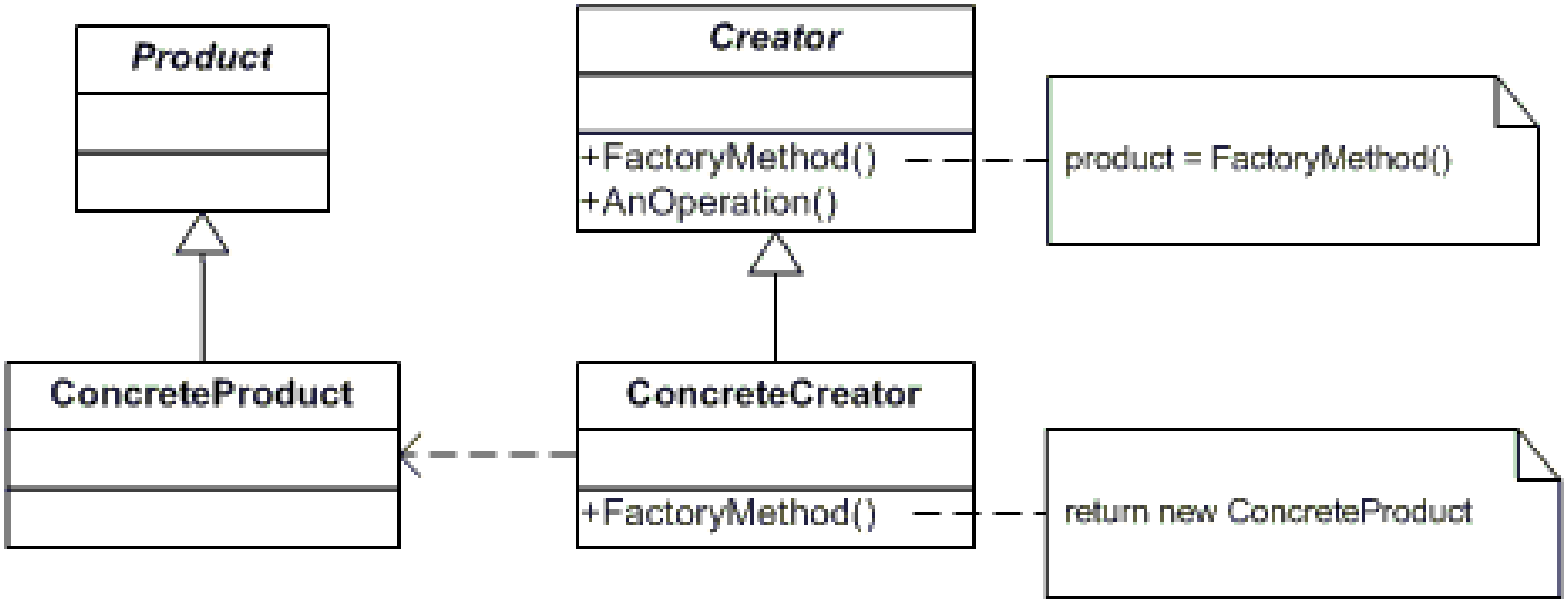
FACTORY METHOD PATTERN

- **Creator**
  - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
  - may call the factory method to create a Product object.

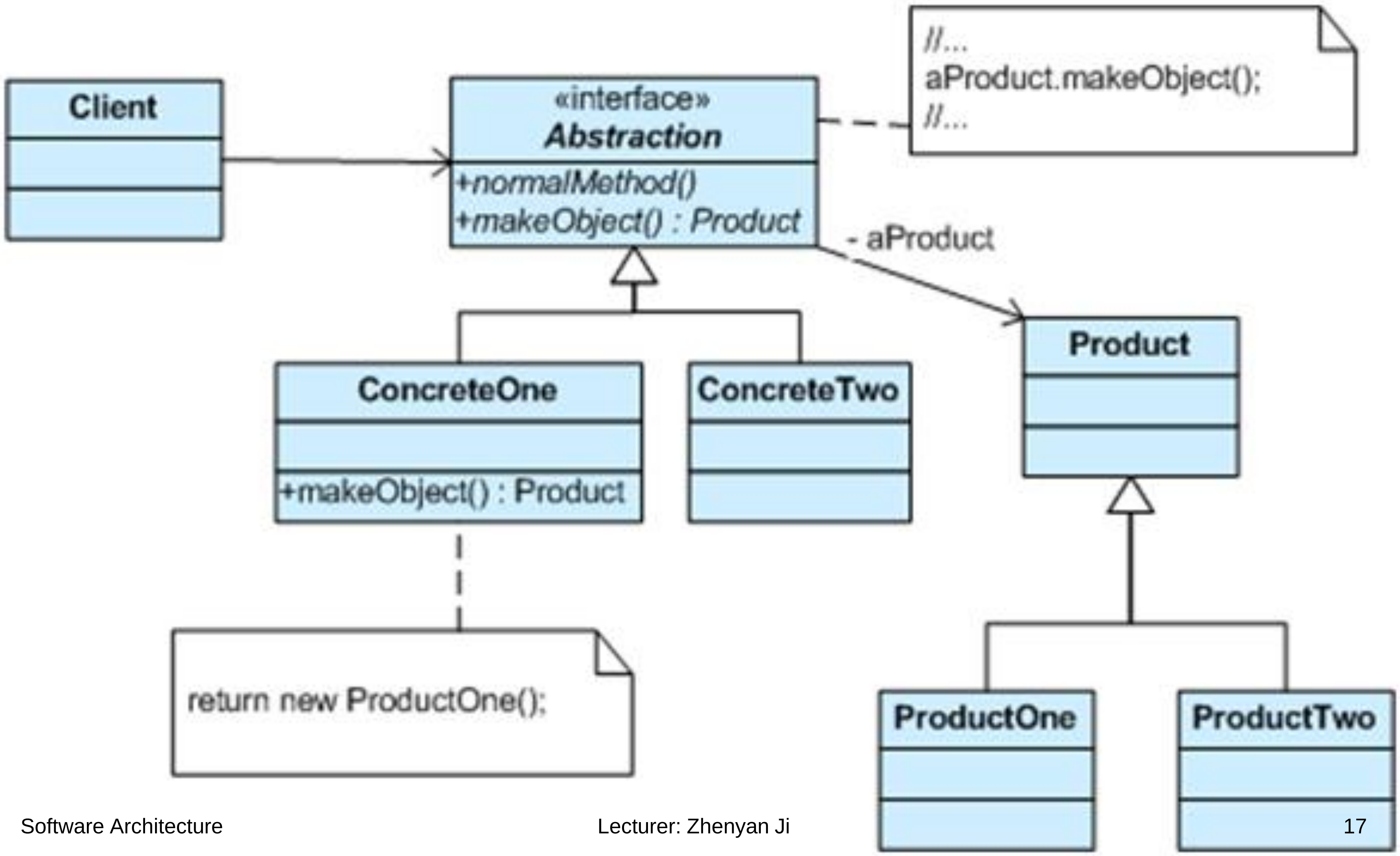
FACTORY METHOD PATTERN

- **ConcreteCreator**
  - overrides the factory method to return an instance of a ConcreteProduct.

Factory Method Pattern

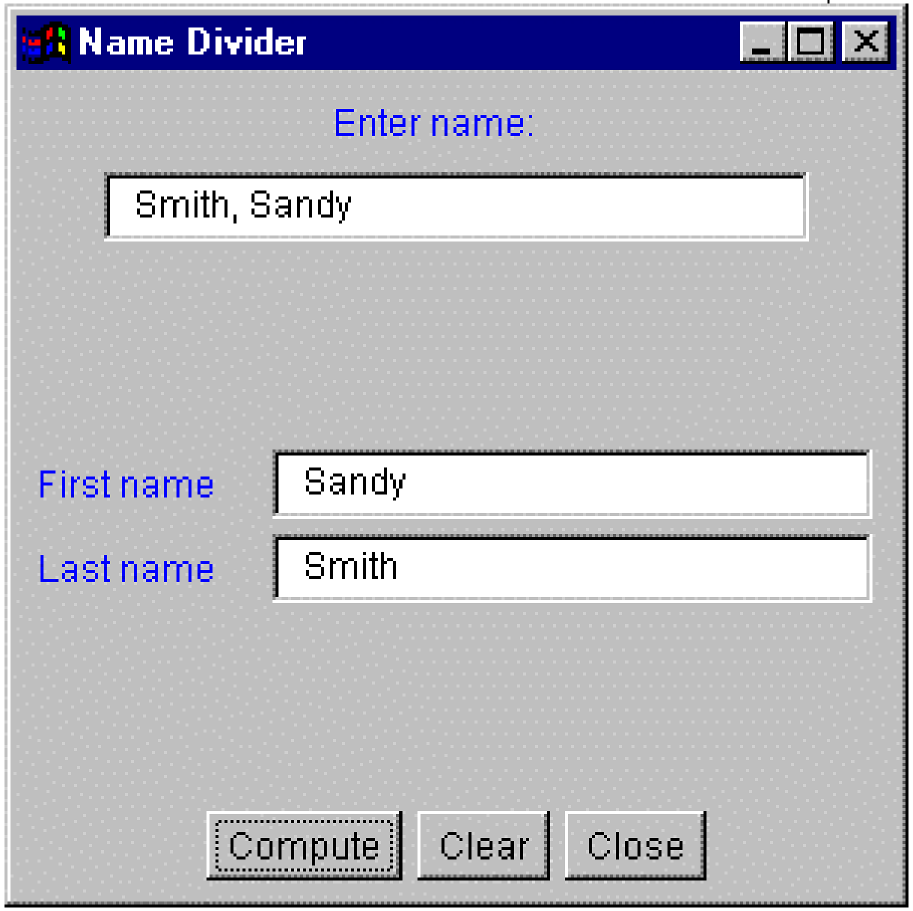


Factory Method Pattern



Example

- An entry form allows the user to enter his name either as “firstname lastname” or as “lastname, firstname”.



Name Divider

Enter name: Smith, Sandy

First name: Sandy

Last name: Smith

Example: base class

```
class Namer {
//a simple class to take a string apart into two
   names
   protected String last; //store last name here
   protected String first; //store first name here

   public String getFirst() {
         return first; //return first name   }
   public String getLast() {
         return last; //return last name   }
}
```

Example: subclass

```
class FirstFirst extends Namer { //split first last
    public FirstFirst(String s) {
        int i = s.lastIndexOf(" "); //find sep space
        if (i > 0) {
            //left is first name
            first = s.substring(0, i).trim();
            //right is last name
            last =s.substring(i+1).trim();
            }
        else {
            first = “”; // put all in last name
            last = s; // if no space }
    } }
```

Example: subclass

```
class LastFirst extends Namer { //split last, first
   public LastFirst(String s) {
        int i = s.indexOf(","); //find comma
        if (i > 0) {
                //left is last name
                last = s.substring(0, i).trim();
                //right is first name
                first = s.substring(i + 1).trim();  }
        else {
                last = s; // put all in last name
                first = ""; // if no comma         }
   } }
```

Building the Factory

```
class NameFactory {
//returns an instance of LastFirst or FirstFirst
//depending on whether a comma is found
    public Namer getNamer(String entry) {
        int i = entry.indexOf(","); //comma determines name
    order
        if (i>0)
            return new LastFirst(entry); //return one class
        else
            return new FirstFirst(entry); //or the other
        }
}
```

Example: using factory

- In our constructor for the program, we initialize an instance of the factory class with

```
NameFactory nfactory = new NameFactory();
private void computeName() {
   //send the text to the factory and get a class back
  namer = nfactory.getNamer(entryField.getText());
   //compute the first and last names
   //using the returned class
   txFirstName.setText(namer.getFirst());
   txLastName.setText(namer.getLast());
}
```

When to Use a Factory Pattern

- Consider using a Factory pattern when
  - A class can’t anticipate which kind of class of objects it must create.
  - A class uses its subclasses to specify which objects it creates.
  - You want to localize the knowledge of which class gets created.

When to Use a Factory Pattern

- There are several similar variations on the factory pattern to recognize.
  - The base class is abstract and the pattern must return a complete working class.
  - The base class contains default methods and is only subclassed for cases where the default methods are insufficient.
  - Parameters are passed to the factory telling it which of several class types to return. In this case the classes may share the same method names but may do something quite different.

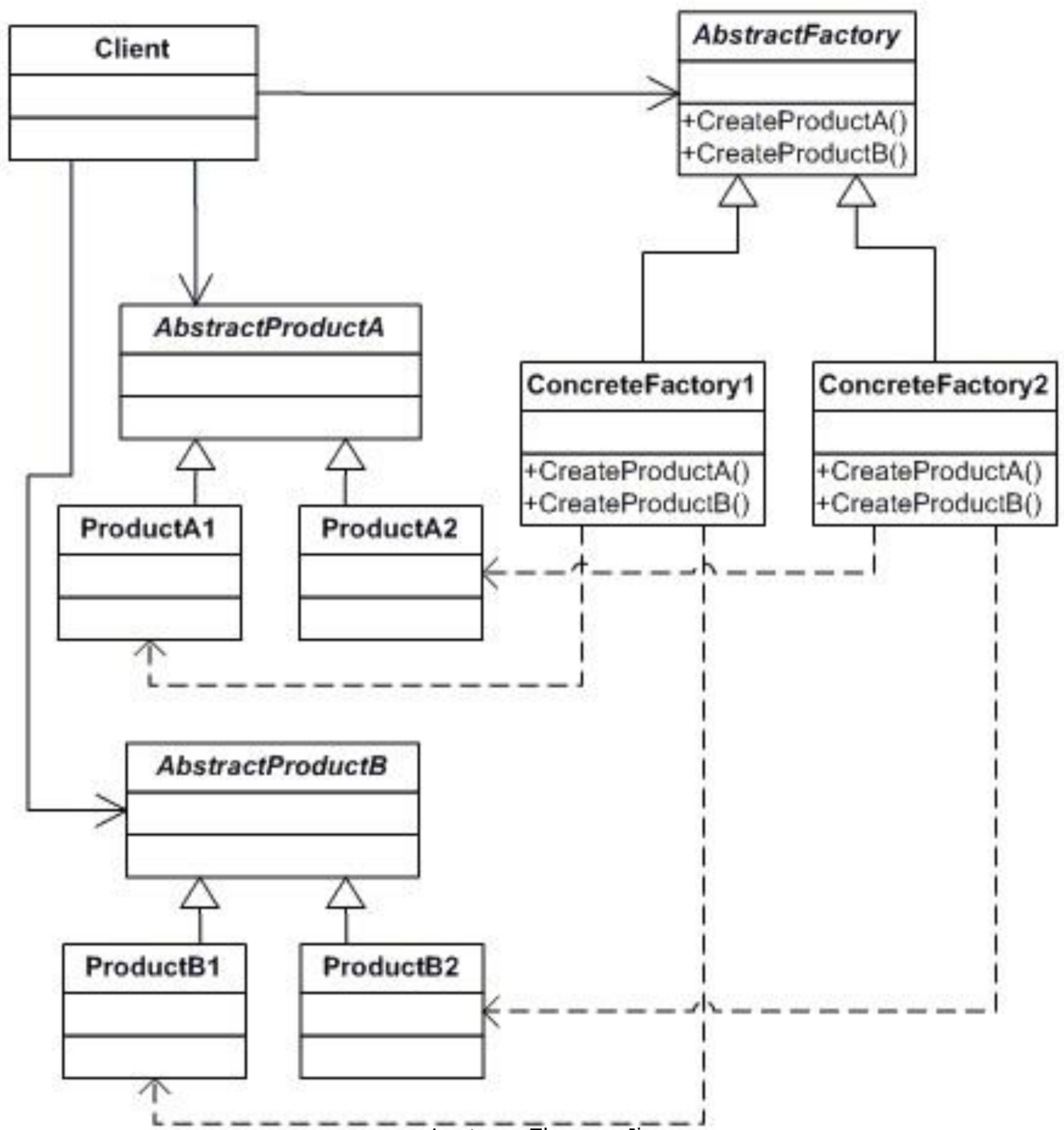
Abstract Factory Pattern

Abstract Factory Pattern

- Definition:
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
- the Abstract Factory is a super-factory which creates other factories (Factory of factories).

Abstract Factory Pattern

- One classic application of the abstract factory is the case where your system needs to support multiple “look-and-feel” user interfaces, such as Windows, Unix or Macintosh.


Client
AbstractFactory
+CreateProductA()
+CreateProductB()
AbstractProductA
ConcreteFactory1
+CreateProductA()
+CreateProductB()
ConcreteFactory2
+CreateProductA()
+CreateProductB()
ProductA1
ProductA2
AbstractProductB
ProductB1
ProductB2

Abstract Factory Pattern

- **participants**

  The classes and/or objects participating in this pattern are:

  - **AbstractFactory**
    - declares an interface for operations that create abstract products

Abstract Factory Pattern

- **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

Abstract Factory Pattern

- **Client**
  - uses interfaces declared by AbstractFactory and AbstractProduct classes

Example

- a program plans the layout of gardens. These could be annual gardens, vegetable gardens or perennial gardens. Questions when planning any kind of garden:
  - What are good border plants?
  - What are good center plants?
  - What plants do well in partial shade?
- …and probably many other plant questions that are omitted in this simple example.

Example: base class

- A base garden class

```
public abstract class Garden {
        public abstract Plant getCenter();
        public abstract Plant getBorder();
        public abstract Plant getShade();
}
```

Example

```
public class Plant {
   String name;
   public Plant(String pname) {
         name = pname; //save name
   }
   public String getName() {
   return name;
   }
}
```

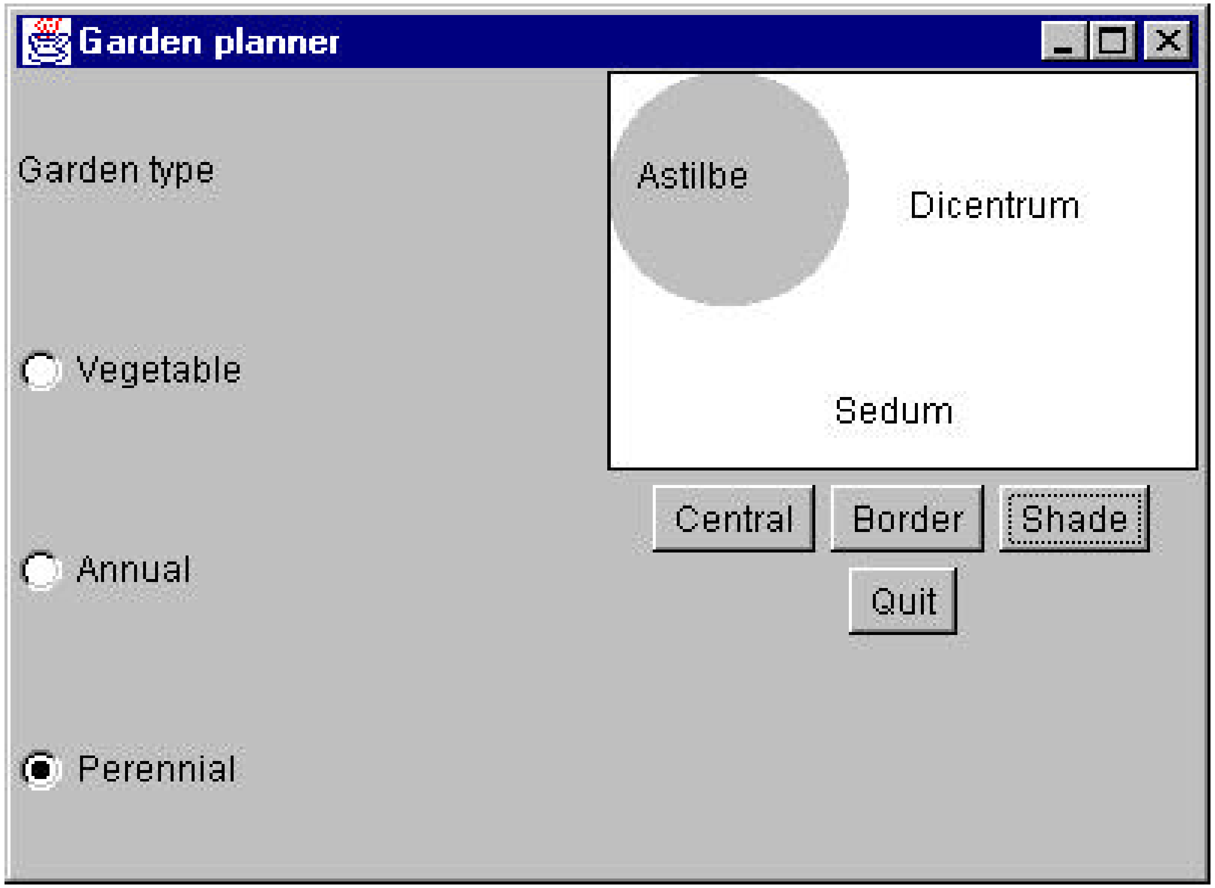
Example: subclass

```
public class VegieGarden extends Garden {
   public Plant getShade() {
        return new Plant("Broccoli");
   }
   public Plant getCenter() {
        return new Plant("Corn");
   }
   public Plant getBorder() {
        return new Plant("Peas");
   } }
```

Example

```
class GardenMaker
{
    //Abstract Factory which returns one of three gardens
    private Garden gd;
    public Garden getGarden(String gtype)
        {
        gd = new VegieGarden(); //default
        if(gtype.equals("Perennial"))
                gd = new PerennialGarden();
        if(gtype.equals("Annual"))
                gd = new AnnualGarden();
        return gd;
 }}
```

Example


Garden planner
Garden type
Vegetable
Annual
Perennial
Astilbe
Dicentrum
Sedum
Central
Border
Shade
Quit

Example

```
public void itemStateChanged(ItemEvent e)
  {
        Checkbox ck = (Checkbox)e.getSource();
        //get a garden type based on label of radio button
        garden = new
              GardenMaker().getGarden(ck.getLabel());
        // Clear names of plants in display
        shadePlant="";
        centerPlant="";
        borderPlant = "";
        gardenPlot.repaint(); //display empty garden
  }
```

```
public void actionPerformed(ActionEvent e) {
   Object obj = e.getSource(); //get button type
   if(obj == Center)                setCenter();
   if(obj == Border)                setBorder();
   if(obj == Shade)                 setShade();
   if(obj == Quit)                  System.exit(0);    }
//-----------------------------------
private void setCenter() {
   if (garden != null)
        centerPlant = garden.getCenter().getName();
   gardenPlot.repaint();
}
```

```
private void setBorder() {
   if (garden != null)
        borderPlant =
   garden.getBorder().getName();
   gardenPlot.repaint();
}
private void setShade() {
   if (garden != null)
        shadePlant =
   garden.getShade().getName();
   gardenPlot.repaint();
}
```

```
class GardenPanel extends Panel {
  public void paint (Graphics g) {
      //get panel size
      Dimension sz = getSize();
      //draw tree shadow
      g.setColor(Color.lightGray);
      g.fillArc( 0, 0, 80, 80,0, 360);
      //draw plant names, some may be blank
  strings
      g.setColor(Color.black);
      g.drawRect(0,0, sz.width-1, sz.height-1);
      g.drawString(centerPlant, 100, 50);
      g.drawString( borderPlant, 75, 120);
      g.drawString(shadePlant, 10, 40);    }}
```

Abstract Factory

- pros
  - it isolates the creation of objects from the client that needs them.
  - The exchanging of product families is easier.
- cons
  - adding new products to the existing factories is difficult.

Abstract Factory

- Some derived classes have additional methods that differ from the methods of other classes.
  - For example a BonsaiGarden class might have a Height or WateringFrequency method that is not present in other classes.
  - A problem: you don't know whether you can call a class method unless you know whether the derived class is one that allows those methods.

Abstract Factory

- Two solutions to the problem:
  - Define all of the methods in the base class, even if they don’t always have an actual function
  - Test to see which kind of class you have:

```
if (gard instanceof BonsaiGarden)
int h = gard.Height();
```

The Singleton Pattern

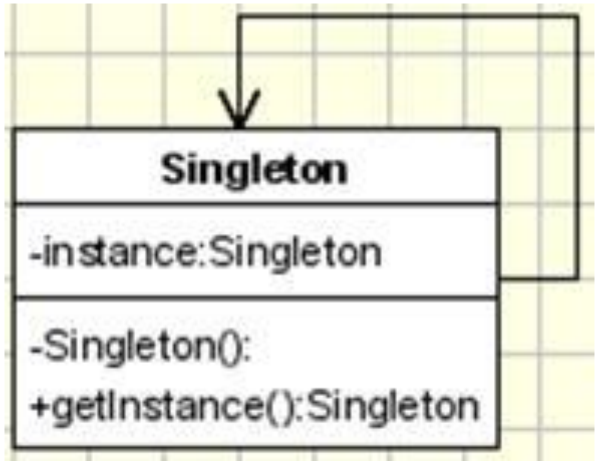
Singleton Pattern

- The Singleton pattern is to some extent a "non-creational" pattern, but it is grouped with the other Creational patterns.
- Definition/Intent:
  - Ensure that only one instance of a class is created.
  - Provide a global point of access to the object.

Singleton Pattern

- Motivation:
  - Some classes should have exactly one instance
    (one print spooler, one file system, one window manager)
  - A global variable makes an object accessible but doesn’t prohibit instantiation of multiple objects
  - Class should be responsible for keeping track of its sole interface

THE SINGLETON PATTERN



THE SINGLETON PATTERN

- **participants**

  The classes and/or objects participating in this pattern are:

- **Singleton**
  - defines an Instance operation that lets clients access its unique instance. Instance is a class operation.
  - responsible for creating and maintaining its own unique instance.

SingletonException

- Create a class that throws an Exception when it is instantiated more than once.

```
class SingletonException extends
   RuntimeException {
        //new exception type for singleton classes
        public SingletonException() {
                super();
        }
        public SingletonException(String s) {
                super(s);
        }
   }
```

Create the Singleton Pattern

- Implement the singleton pattern

```
class PrintSpooler{
        //this is a prototype for a printer-spooler class
        //such that only one instance can ever exist
    static boolean instance_flag=false;
        //true if 1 instance
public PrintSpooler() throws SingletonException {
        if (instance_flag)
                throw new SingletonException("Only one
    spooler allowed");
        else {
                instance_flag = true; //set flag for 1 instance
                System.out.println("spooler opened"); }   }
    //----------------------------------------
    public void finalize() {
                instance_flag = false;
                //clear if destroyed  }
}
```

Creating an Instance of the Class

```
public class singleSpooler {
  static public void main(String argv[]){
        PrintSpooler pr1, pr2;
        //open one spooler--this should always work
        System.out.println("Opening one spooler");
        try{
                pr1 = new PrintSpooler();
        } catch (SingletonException e)
                {System.out.println(e.getMessage());}
//try to open another spooler --should fail
        System.out.println("Opening two
   spoolers");
        try{
                pr2 = new PrintSpooler();
        } catch (SingletonException e)

        {System.out.println(e.getMessage());}
} }
```

Result

```
Opening one spooler
printer opened
Opening two spoolers
Only one spooler allowed
```

Creating Singleton Using a Static Method

- Another approach to create Singletons:
  - using a static method to issue and keep track of instances.
  - making the constructor private so an instance can only be created from within the static method of the class.

Creating Singleton Using a Static Method

```
class iSpooler {
        static boolean instance_flag = false;
        private iSpooler() { }
        static public iSpooler Instance() {
                if (! instance_flag) {
                        instance_flag = true;
                        return new iSpooler();
                } else
                        return null;
         }
   public void finalize() {
        instance_flag = false;    } }
```

Creating Singleton Using a Static Method

- Advantage: don't have to worry about exception handling if the singleton already exists-- you simply get a null return from the Instance method:

```
  iSpooler pr1, pr2;
  System.out.println("Opening one spooler");
  pr1 = iSpooler.Instance();
if(pr1 != null)
   System.out.println("got 1 spooler");
  System.out.println("Opening two spoolers");
  pr2 = iSpooler.Instance();
if(pr2 == null)
   System.out.println("no instance available");
```

To be continued…