Design Patterns I

Ergude Bao
Beijing Jiaotong University

Contents

- Design Patterns
- Creational Patterns I
  - Factory Method Pattern
  - Abstract Factory Pattern
  - Singleton Pattern

Design Patterns

History about Design Patterns

- Patterns originated as an architectural concept by Christopher Alexander (1977/79)

History about Design Patterns

- 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.)

Design Patterns

- In software engineering, each design pattern is a general reusable solution to commonly occurring problem in software design
  – Focuses on a particular object-oriented design problem or issue
  – Description or template for how to solve a problem
  – Not a finished design that can be transformed directly into code

Classification of Design Patterns

- At least 250 existing patterns are used in OOP world. 23 design patterns are well known 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
  - 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

Creational Patterns I

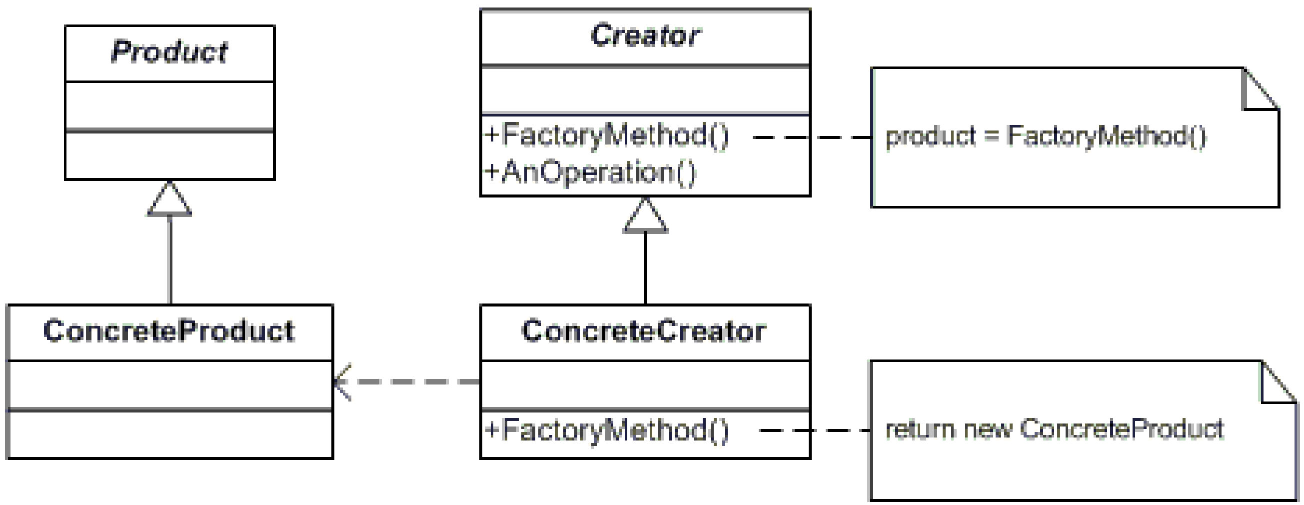
- Factory Method Pattern
  - Defines an interface for creating objects without specifying their concrete class
- Abstract Factory Pattern
  - Provides an interface for creating families of related or dependent objects without specifying their concrete classes
- Singleton Pattern
  - Ensures a class has only one object, and provides a global point of access to it

Factory Method Pattern

Factory Method Pattern

- Motivation
  - It improves cohesion and coupling to separate creation of objects from use of them
- Solution
  - Create objects without exposing the instantiation logic to the client
  - Usually refer to the newly created objects through a common interface
- Factory method pattern is probably the most used design pattern in modern programming languages like Java and C#

Factory Method Pattern

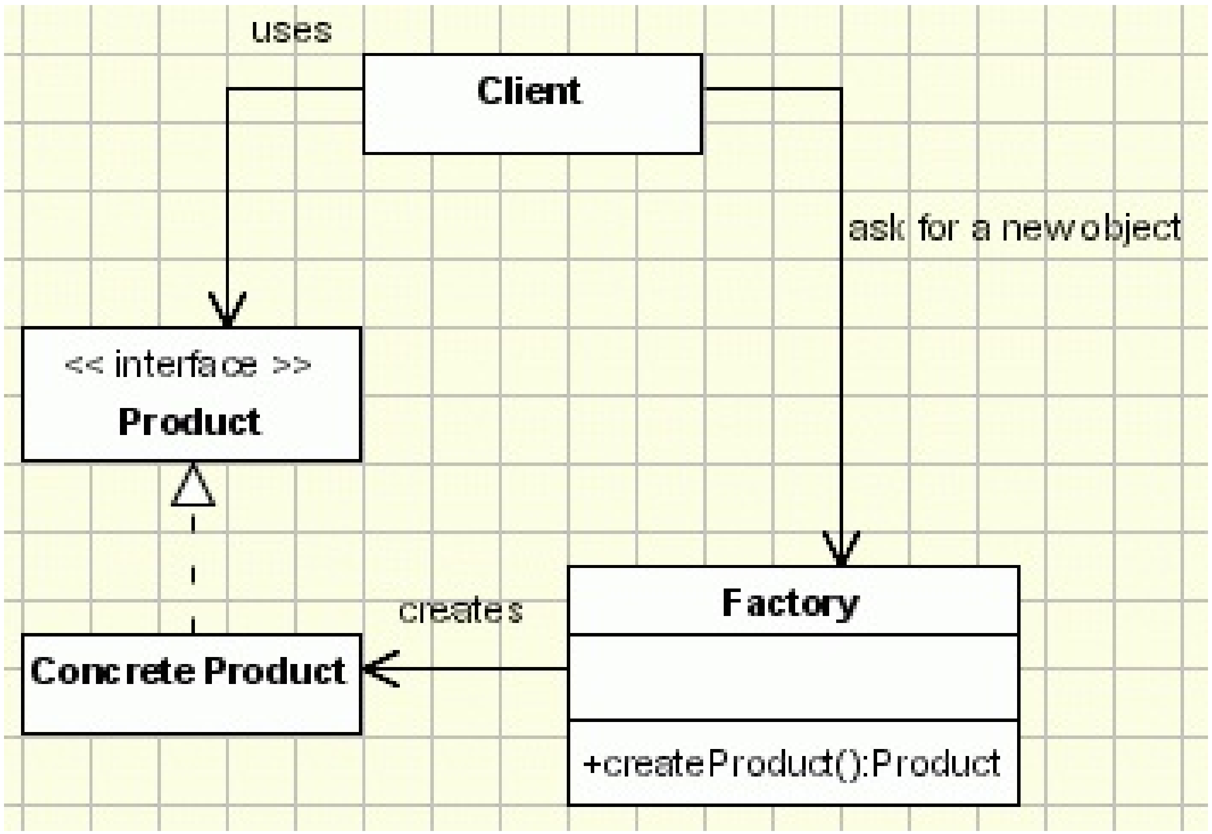


Participants

- Product
  - Declares the interface of objects the factory method creates
- ConcreteProduct
  - Implements the Product interface
- Creator
  - Declares the factory method to return an object of type Product
- ConcreteCreator
  - Overrides the factory method to return an instance of a ConcreteProduct
- Client
  - Uses interfaces declared by Creator and Product classes

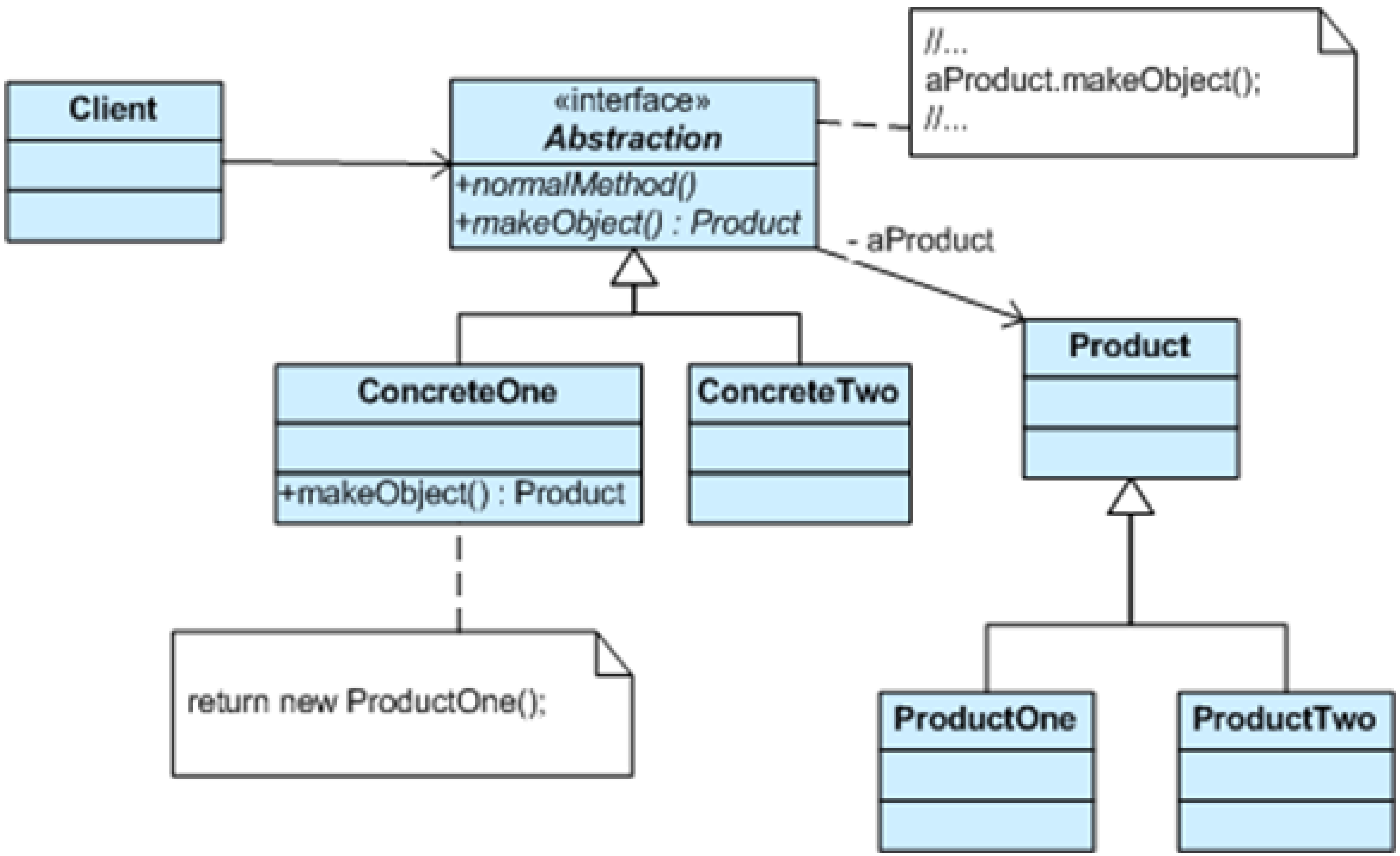
Factory Method Pattern

- Simplified version

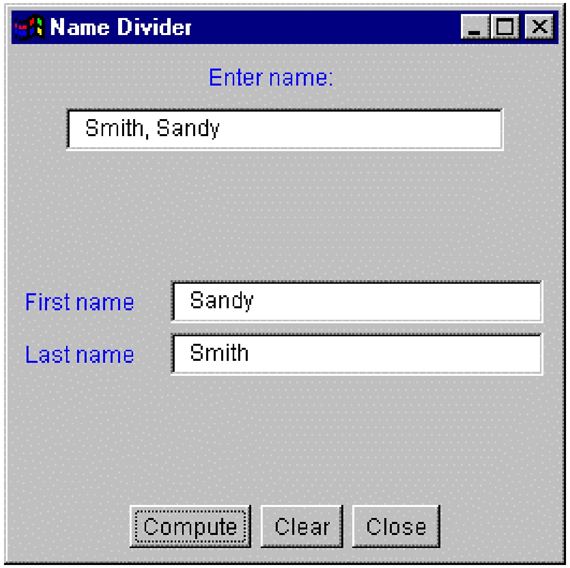


Factory Method Pattern

- Complex version



Example



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

Example

```
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

```
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

```
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
        }
    }
}
```

Example

```
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

- 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

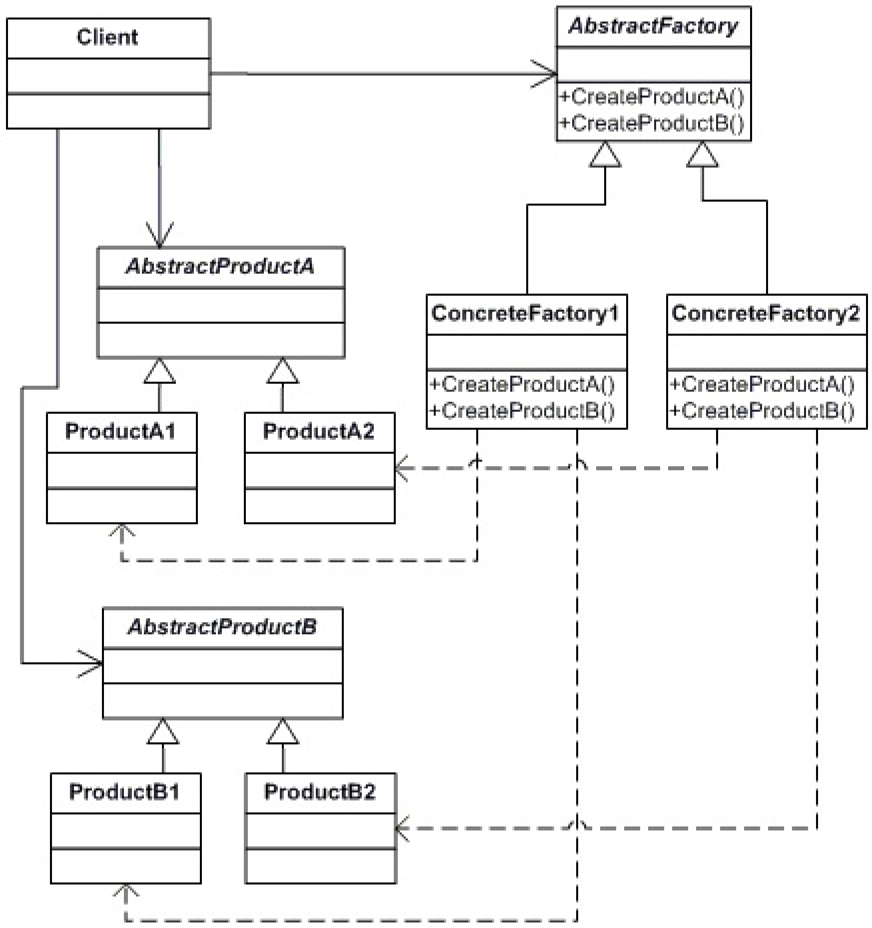
- Cannot anticipate objects of which classes to create and want to localize the judgment logic
  - Parameters are passed to the factory telling it objects of which classes to create and return
- Want to use different subclasses in the same manner
  - Returned objects may share the same method names but may do something quite different

Abstract Factory Pattern

Abstract Factory Pattern

- Motivation
  - It improves cohesion and coupling to separate creation of families of objects from use of them
- Solution
  - Provides an interface for creating families of related or dependent objects without specifying their concrete classes
- 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, Linux or Mac

Abstract Factory Pattern



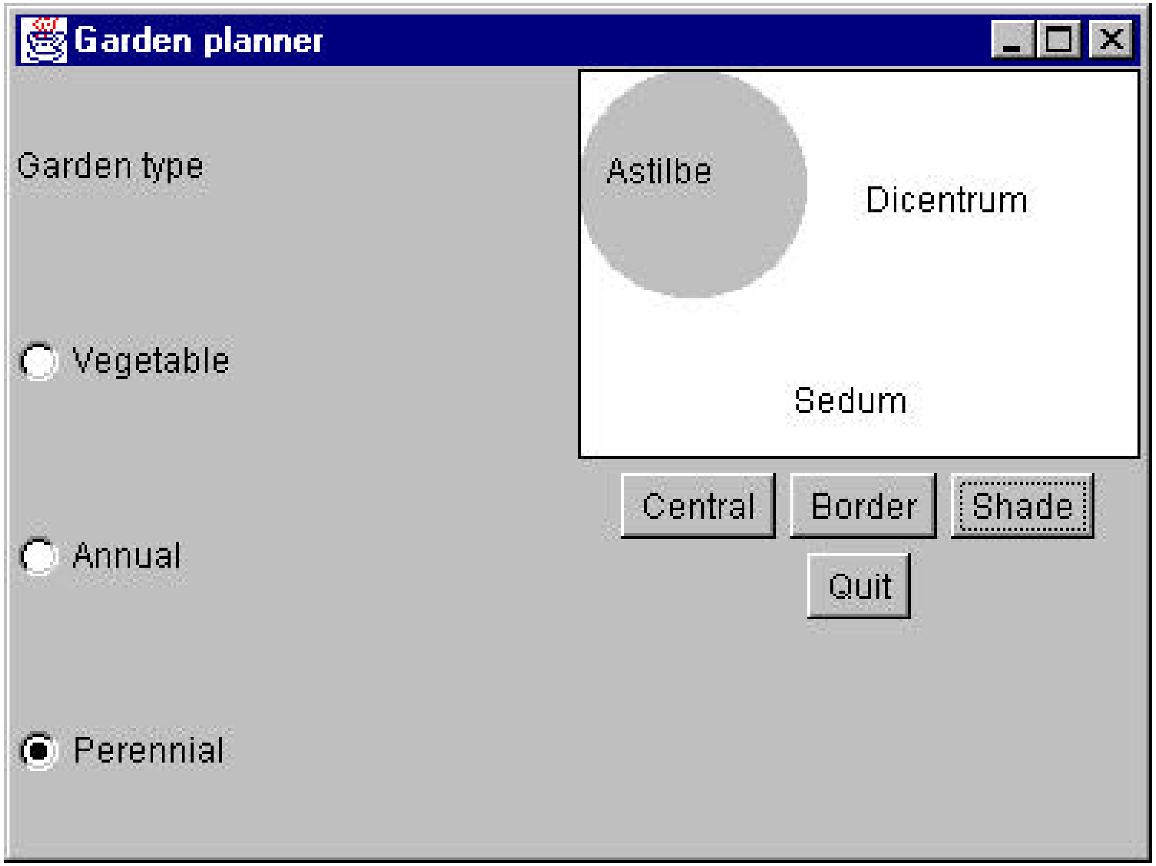
Participants

- AbstractProduct
  – Declares an interface for a type of product object
- Product
  – Implements the AbstractProduct interface
- AbstractFactory
  – Declares an interface for operations that create AbstractProducts
- ConcreteFactory
  – Implements the operations to create concrete Product objects
- 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 the gardens:
  - What are good border plants?
  - What are good center plants?
  - What is the shading area for a plant?
  - Many other plant questions that are omitted in this simple example…

Example


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

Example

- 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

```
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 returning 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

```
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
}
```

Example

```
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();
}
```

Example

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

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

Example

```
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);
	}
}
```

Advantages

- It isolates the creation of objects from the client that needs them
- Switching between different families is easier

Disadvantage

- Adding new objects to the existing families is difficult

Problem & Solutions

- Some subclasses have additional methods that differ from the methods of other classes, so you do not know whether you can call a method unless you know whether the derived class is one that allows those methods
  - For example, a BonsaiGarden class might have a Height or WateringFrequency method that is not present in other classes
- Two solutions to the problem:
  - Define all of the methods in the base class, even if they do not always have an actual function
  - Test to see which kind of class you have:

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

Factory Method vs. Abstract Factory

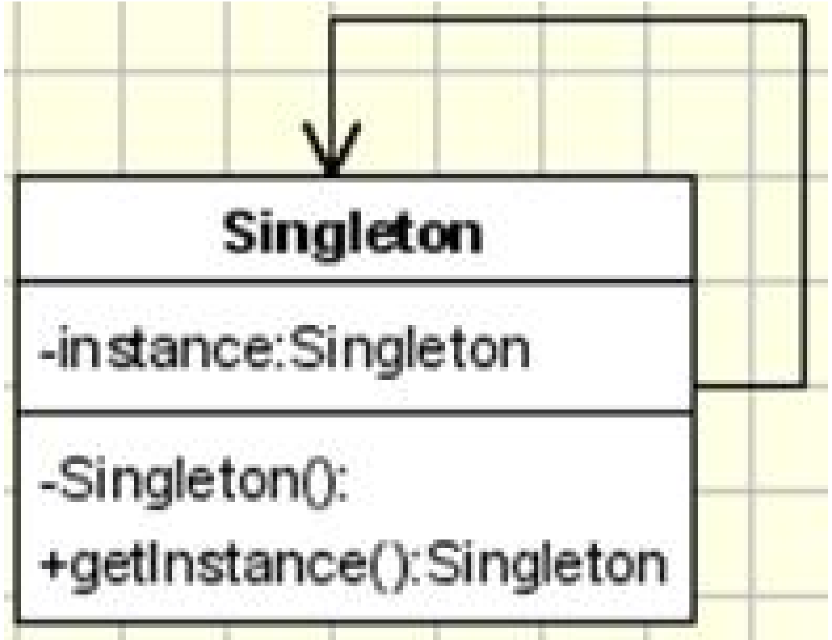
- Similarity: both use factory to create objects
- Difference: former creates one kind of objects in each factory, while latter creates a family of objects in each factory

Singleton Pattern

Singleton Pattern

- Motivation
  - Some classes should have exactly one object (one print spooler, one file system, one window manager)
  - A global variable makes an object accessible but does not prohibit instantiation of multiple objects
- Solution
  - Ensure that only one instance of a class is created
  - Provide a global point of access to the object
- Pattern not based on any design principle

Singleton Pattern



Participant

- Singleton
  - Responsible for creating and maintaining its own unique object
  - Defines an instance operation that lets clients access the unique object

Example I

- Open one spooler
- Open printer
- Open two spoolers
- Allow only one spooler

Example I

- Implement the singleton pattern

```
class PrintSpooler{
        //this is a prototype for a printer-spooler class
        //such that only one object can ever exist
        static boolean instance_flag=false;
        //true if 1 object

        public PrintSpooler() throws SingletonException {
                if (instance_flag)
                        throw new SingletonException("Only one spooler allowed");
                else {
                        instance_flag = true; //set flag for 1 object
                        System.out.println("spooler opened");
                }
        }

        public void finalize() {
                instance_flag = false;
                //clear if destroyed
        }
}
```

Example I

```
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());}
	}
}
```

Example II

```
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;
	}
}
```

Example II

- Advantage: do not 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");
```