Two Creational OO Design Patterns

CSC 335: Object-Oriented Programming and Design

Outline

- Two Creational Design Patterns
  - Singleton
  - Factory

To use new or to not use new? That is the question

- Most OO languages provide object instantiation with new and initialization with constructors
- There may be a tendency to simply use these facilities directly without forethought to future consequences
- The overuse of this functionality can introduces inflexibility in a system

Creational Patterns

- Creational patterns describe object-creation mechanisms that enable greater levels of reuse in evolving systems: Builder, Singleton, Prototype
- Singleton Design Pattern
  - Ensure there is only one instance (and think of race conditions avoided with a synchronized method)
- The most widely used is Factory Design Pattern
  - This pattern calls for the use of a specialized object solely to create other objects

OO Design Pattern
Singleton

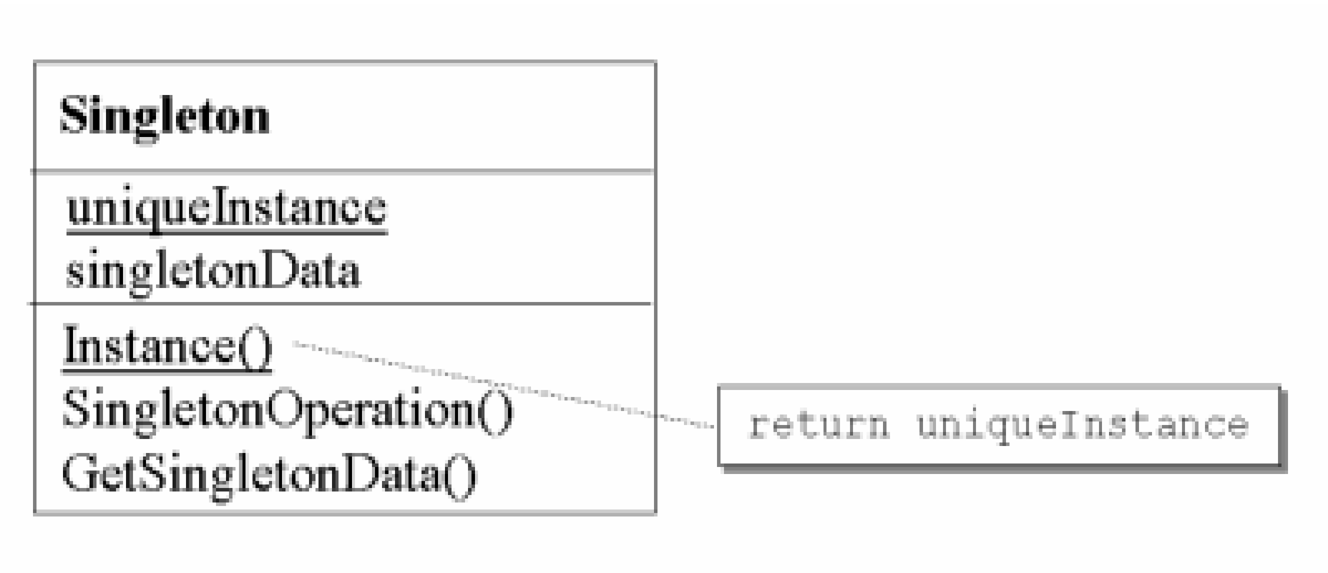
Recurring Problem

- Some classes have only one instance. For example, there may be many printers in a system, but there should be only one printer spooler
- How do we ensure that a class has only one instance and that instance is easily accessible?

Solution

- Have constructor return the same instance when called multiple times
- Takes responsibility of managing that instance away from the programmer
  - It is simply impossible to construct more instances

UML General form as UML



Example Used in a 335 final project

```
/** This class is a DECORATOR of ArrayList. Its purpose is to make
  * sure there are no duplicate names anywhere in the universe.
  * That's why it's SINGLETON; because many classes use it but
  * there should be only one list of names.  */
public class NamesList implements Serializable {
  private ArrayList<String> npcNames;
  private static NamesList self;

  private NamesList() {
    npcNames = new ArrayList<String>();
  }

  public static synchronized NamesList getInstance() {
    if (self == null) {
      self = new NamesList();
    }
    return self;
  }
```

OO Design Pattern
Factory Method

- **Name**: Factory Method
- **Problem**: A Client needs an object and it doesn't know which of several objects to instantiate
- **Solution**: Let an object instantiate the correct object from several choices. The return type is an abstract class or an interface type.

Characteristics

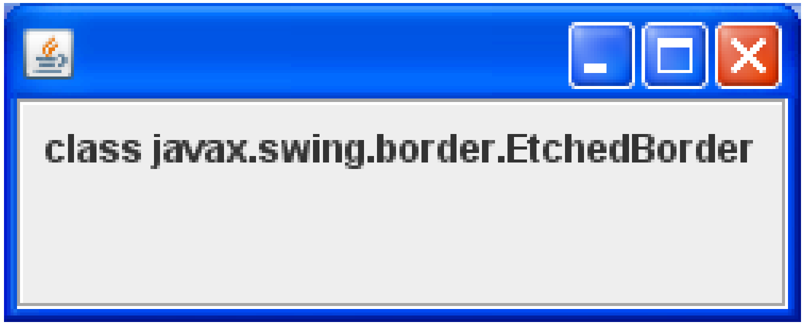
- A method returns an object
- The return type is an abstract class or interface
- The interface is implemented by two or more classes or the class is extended by two or more classes

Example from Java

- Border is an interface
- AbstractBorder is an abstract class that implements Border
- BorderFactory has a series of static methods returning different types that implement Border
  - This hides the implementation details of the subclasses
- Factory methods such as `createMatteBorder` `createEtchedBorder` `createTitleBorder` directly call constructors of AbstractBorder's subclasses

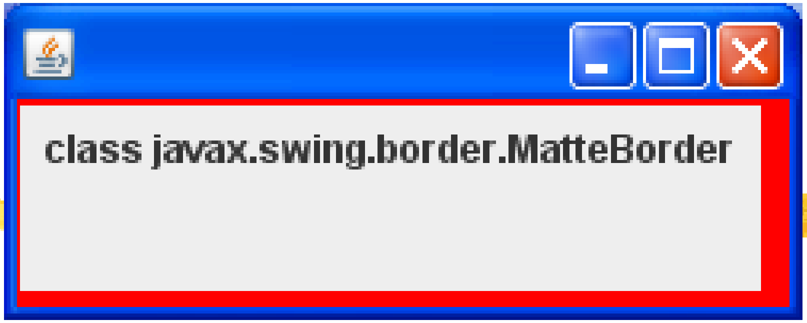
One type

```
JFrame f = new JFrame();
f.setSize(250, 100);
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

JPanel toBeBordered = new JPanel();
Border border = BorderFactory.createEtchedBorder();
toBeBordered.add(new JLabel("" + border.getClass()));
toBeBordered.setBorder(border);

f.add(toBeBordered);
f.setVisible(true);
```



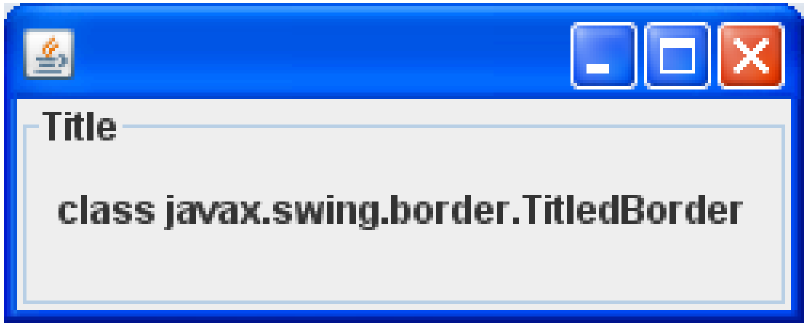
class javax.swing.border.EtchedBorder

```
border = BorderFactory.createMatteBorder(2, 1, 5, 9,Color.RED);
```



class javax.swing.border.MatteBorder

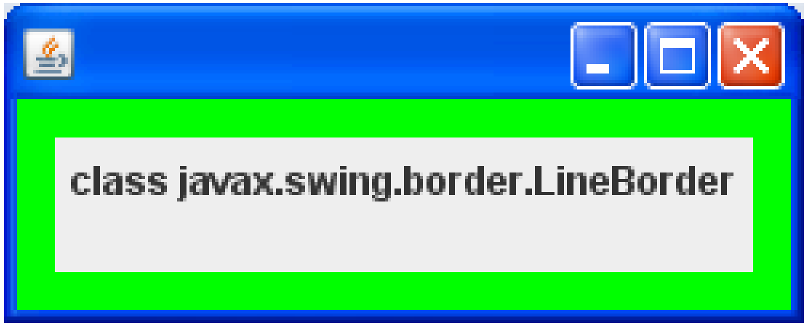
```
border = BorderFactory.createTitledBorder("Title");
```



Title

class javax.swing.border.TitledBorder

```
border = BorderFactory.createLineBorder(Color.GREEN, 12);
```



class javax.swing.border.LineBorder

Lots of Subclasses

javax.swing.border.AbstractBorder

java.lang.Object

javax.swing.border.AbstractBorder

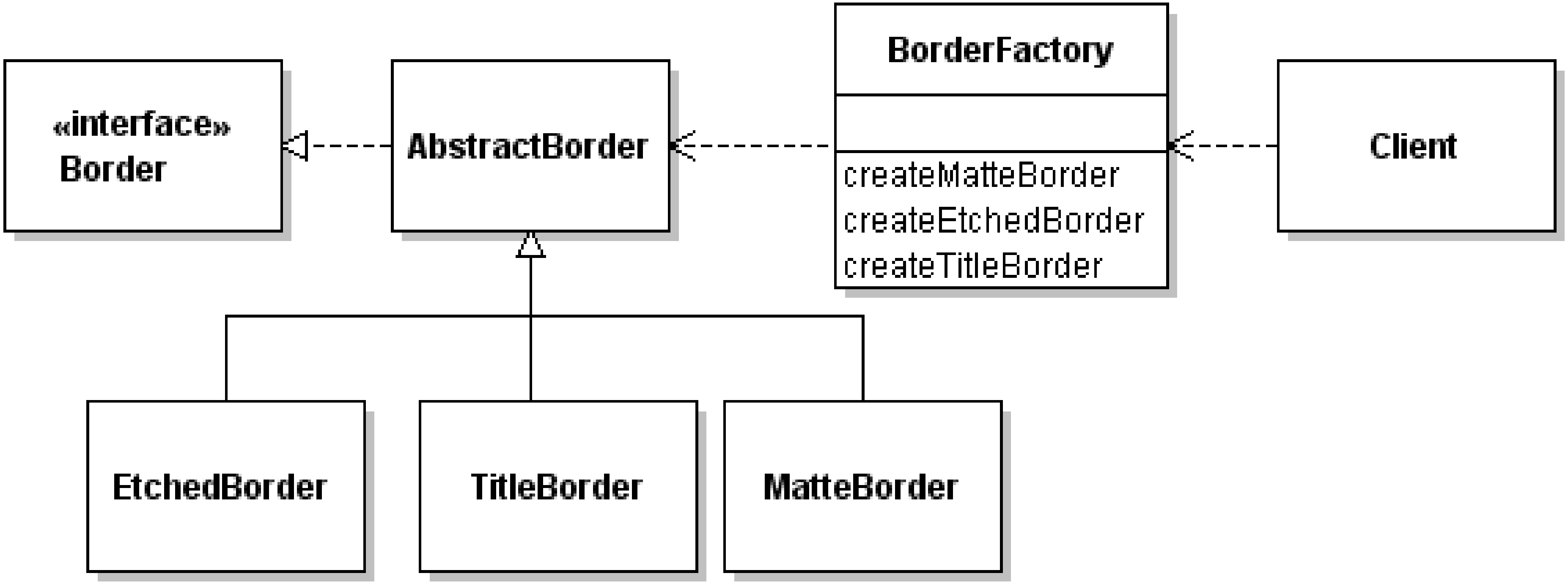
All Implemented Interfaces:

Serializable, Border

Direct Known Subclasses:

BasicBorders.ButtonBorder, BasicBorders.FieldBorder, BasicBorders.MarginBorder, BasicBorders.MenuBarBorder, BevelBorder, CompoundBorder, EmptyBorder, EtchedBorder, LineBorder, MetalBorders.ButtonBorder, MetalBorders.Flush3DBorder, MetalBorders.InternalFrameBorder, MetalBorders.MenuBarBorder, MetalBorders.MenuItemBorder, MetalBorders.OptionDialogBorder, MetalBorders.PaletteBorder, MetalBorders.PopupMenuBorder, MetalBorders.ScrollPaneBorder, MetalBorders.TableHeaderBorder, MetalBorders.ToolBarBorder, TitledBorder

Factory Design Pattern Example



NumberFormat, a factory

- Objects can be returned without directly using new

```
double amount = 12345.1234656789457;
NumberFormat formatter =
           NumberFormat.getCurrencyInstance();
System.out.println(formatter.format(amount));
```

Output if the computer is set to US Locale

```
$12,345.12
```

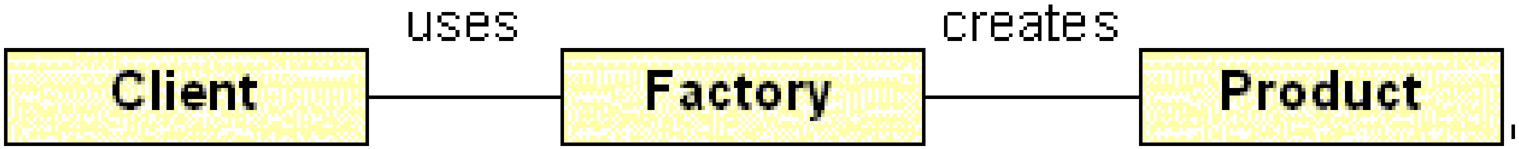
Use computer setting to Germany Locale and we get this:

```
NumberFormat.getCurrencyInstance(Locale.GERMANY);
 12.345,12 €
```

What Happened?

- `getCurrencyInstance` returns an instance of `DecimalFormat` where methods like `setCurrency` help build the appropriate object
  - It encapsulates the creation of objects
- Can be useful if the creation process is complex, for example if it depends on settings in configuration files or the jre or the OS
- Factory used in a research project with Tree Ring Lab
- Inspired by perhaps having a MapFactory

Behind the scenes



- Client: `main` method
- Factory Method: `getCurrencyInstance`
- Product: a properly configured instance of `DecimalFormat`
- Another example of Factory used in Java’s API

```java
import java.awt.Color;
import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.Border;

public class A {
  public static void main(String[] args) {
    JFrame f = new JFrame();
    f.setSize(250, 100);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JPanel toBeBordered = new JPanel();
    Border border = null;

    border = BorderFactory.createEtchedBorder();
    //    border = BorderFactory.createMatteBorder(2, 1, 5, 9, Color.RED);
    //    border = BorderFactory.createTitledBorder("Title");
    //    border = BorderFactory.createLineBorder(Color.GREEN, 12);

    // Show the name of the class that BorderFactory instantiated for us
    toBeBordered.add(new JLabel("" + border.getClass()));
    toBeBordered.setBorder(border);

    f.add(toBeBordered);
    f.setVisible(true);
  }
}
```

```java
import java.awt.Color;
import java.text.NumberFormat;
import java.util.Locale;

import javax.swing.BorderFactory;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.border.Border;

public class TwoFactories {

  public static void main(String[] args) {
    JFrame f = new JFrame();
    f.setSize(250, 100);
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

    JPanel toBeBordered = new JPanel();
    Border border = BorderFactory.createMatteBorder(2, 1, 5, 9, Color.RED);

    // Border border = BorderFactory.createEtchedBorder();
    // Border border = BorderFactory.createTitledBorder("Title");
    // Border border = BorderFactory.createLineBorder(Color.GREEN, 12);

    toBeBordered.add(new JLabel("" + border.getClass()));
    toBeBordered.setBorder(border);

    f.getContentPane().add(toBeBordered);
    f.setVisible(true);

    double amount = 12345.1234656789457;
    NumberFormat formatter = NumberFormat.getCurrencyInstance();
    System.out.println(formatter.format(amount));
    formatter = NumberFormat.getCurrencyInstance(Locale.GERMANY);
    System.out.println(formatter.format(amount));
    formatter = NumberFormat.getCurrencyInstance(Locale.UK);
    System.out.println(formatter.format(amount));
  }
}
```