Object Oriented Software Design

From design to realization

Giuseppe Lipari
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

November 11, 2011

Outline

Outline

Motivation

- Good Object Oriented programming is not easy
  - Emphasis on design
- Errors may be expensive
  - Especially design errors!
- Need a lot of experience to improve the ability in OO design and programming

Motivation

- Good Object Oriented programming is not easy
  - Emphasis on design
- Errors may be expensive
  - Especially design errors!
- Need a lot of experience to improve the ability in OO design and programming
- Reuse experts' design
- Patterns $=$ documented experience

The source

- The design patterns idea was first proposed to the software community by the “Gang of four” [2]
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  - Design patterns: elements of reusable object-oriented software
- They were inspired by a book on architecture design by Christopher Alexander [1]

  Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Expected Benefits

- The idea of patterns has a general meaning and a general application: from architecture to software design

Expected Benefits

- The idea of patterns has a general meaning and a general application: from architecture to software design
  - One of the few examples in which software development has been inspired by other areas of engineering

Expected Benefits

- The idea of patterns has a general meaning and a general application: from architecture to software design
  - One of the few examples in which software development has been inspired by other areas of engineering
- The expected benefits of applying well-know design structures

Expected Benefits

- The idea of patterns has a general meaning and a general application: from architecture to software design
  - One of the few examples in which software development has been inspired by other areas of engineering
- The expected benefits of applying well-know design structures
  - Finding the right code structure (which classes, their relationship)

Expected Benefits

- The idea of patterns has a general meaning and a general application: from architecture to software design
  - One of the few examples in which software development has been inspired by other areas of engineering
- The expected benefits of applying well-know design structures
  - Finding the right code structure (which classes, their relationship)
  - Coded infrastructures

Expected Benefits

- The idea of patterns has a general meaning and a general application: from architecture to software design
  - One of the few examples in which software development has been inspired by other areas of engineering
- The expected benefits of applying well-know design structures
  - Finding the right code structure (which classes, their relationship)
  - Coded infrastructures
  - A Common design jargon (factory, delegation, composite, etc.)

Expected Benefits

- The idea of patterns has a general meaning and a general application: from architecture to software design
  - One of the few examples in which software development has been inspired by other areas of engineering
- The expected benefits of applying well-know design structures
  - Finding the right code structure (which classes, their relationship)
  - Coded infrastructures
  - A Common design jargon (factory, delegation, composite, etc.)
  - Consistent format

Patterns and principles

- Patterns can be seen as extensive applications of the OO principles mentioned above
- For every patter we will try to highlight the benefits in terms of hiding, reuse, decoupling, substitution, etc.

Pattern Categories

- **Creational:** Replace explicit creation problems, prevent platform dependencies
- **Structural:** Handle unchangeable classes, lower coupling and offer alternatives to inheritance
- **Behavioral:** Hide implementation, hides algorithms, allows easy and dynamic configuration of objects

Outline

Intent

Ensure that a class only has one instance, and provide a global point of access to it

- For some classes it is important to have exactly one instance
  - The should be only one window manager in the system
- Of course, the same can be achieved with a global variable
- However, for complex system we could run in some problems
  - the initialization order
  - the object is created many times by mistake, etc.
- A better solution is to make the class itself responsible for creating and maintaining the instance

Code

```
class SysParams {
    static private SysParams inst = new SysParams();
    private SysParams() {
        ...
    }
    // other private non static members
    int nHandles = 0;
    String confFile = "";
    static public SysParams getInstance() {
        return inst;
    }
    // other non static members
    public int getHandles() {
         return nHandles;
    }
    public String getConfFileName() {
         return confFile;
    }
    public void setConfFileName(String name) {
         confFile = name;
    }
}
```

Comment

- The constructor is private:
  - It is not possible to create new instances of this class
- There is only one instance
  - The instance is referred by a static member, which is a reference to an object of the same class
  - The instance is initialised when the class is used for the first time
  - The only way to access the class instance is to invoke the static method `getInstance()`
- All other members are non static
  - Data is private
  - Functions are public

Usage

```
...
int n = SysParams.getInstance().getHandles();

SysParams.getInstance().setConfFileName("file.cfg");
...

SysParams p = new SysParams();
```

- To access any method of the instance, we must use the `SysParams.getInstance()` followed by the method to be called
- It is not possible to create new instances

Alternative

An alternative is to use only static members:

```
class SysParams {
    private SysParams() { }
    static int nHandles = 0;
    static String confFile = "";
    // other non static members
    public static int getHandles() {
         return nHandles;
    }
    public static String getConfFileName() {
         return confFile;
    }
    public static void setConfFileName(String name) {
         confFile = name;
    }
}
```

Usage:

```
int n = SysParams.getHandles();
SysParams.setConfFileName("file.cfg");
```

Alternative?

- The alternative seems simpler
  - Less typing is required!
- However it is less flexible! The Singleton has many advantages
  1. Easier to extend (subclassing is not possible)
  2. Better implementation hiding
  3. Can be easily and more flexibly configured for concurrent programs
  4. Can be extended to provide either two or a limited set of instances

Subclassing and registry

- Sometimes it may be useful to have different subclasses of the class, but only one instance of one of them
- For example, we could chose one of several windows managers
- We can do that at compile/link time by using conditional compilation;
  - In this case, every subclass has its implementation of the `getInstance()` that returns the correct pointer, and the one to compile/link is decided though compilation switches
- We can also do it at run-time (during instantiation), using for example an environment variable
  - In this case, it is necessary to implement the creation code in the `getInstance()` method of the base class

Creation in getInstance

```
class SysParams {
    static private SysParams inst = 0
    private SysParams() {
        ...
    }
    // other private non static members
    ...
    static public SysParams getInstance() {
        if (inst == 0)
            inst = new SysParams();  // or something else
        return inst;
    }
    // other non static members
    ...
}
```

Inheritance

```
class WinManager {
    static private inst = 0;
    ...
    static public getInstance() {
         if (inst == 0) {
             String wm = SysParams.getWinMan();
             if (wm == "Motif")
                 inst = new MotifWinManager();
             else if (wm == "GTK")
                 inst = new GTKWinManager();
             else inst = new DefWinManager();
         }
         return inst;
    }
}

class GTKWinManager extends WinManager { ... }
class MotifWinManager extends WinManager { ... }
class DefWinManager extends DefWinManager { ... }
```

When to use singletons

- A Singleton is useful to implement global variables in a safe way
  - For example, it provides a global point of access and an interface to a set of global objects (e.g. system parameters, a window manager, a configuration manager, etc.)
- It may be useful to control the order of initialisation
- The object is not created if not used
- Sometimes this pattern is overused
  - Singletons everywhere!
  - It is not worth to make it for a few primitive global variables that are local to a module

Outline

Abstract factory

- A program must be able to choose one of several families of classes
- Example,
  - a program's GUI should run on several platforms
  - Each platform comes with its own set of GUI classes:
    - WinButton, WinScrollBar, WinWindow MotifButton, MotifScrollBar, MotifWindow, pmButton, pmScrollBar, pmWindow

Abstract factory

- A program must be able to choose one of several families of classes
- Example,
  - a program's GUI should run on several platforms
  - Each platform comes with its own set of GUI classes:
    - WinButton, WinScrollBar, WinWindow MotifButton, MotifScrollBar, MotifWindow, pmButton, pmScrollBar, pmWindow
  - Inheritance:
    - Clearly, we can make all "button" classes derive from an abstract button that implements a virtual "draw" function
    - Then, we hold a pointer to button, and assign a specific button object, so that the correct draw() function is invoked each time

Abstract factory

- A program must be able to choose one of several families of classes
- Example,
  - a program's GUI should run on several platforms
  - Each platform comes with its own set of GUI classes:
    - WinButton, WinScrollBar, WinWindow MotifButton, MotifScrollBar, MotifWindow, pmButton, pmScrollBar, pmWindow
  - Inheritance:
    - Clearly, we can make all "button" classes derive from an abstract button that implements a virtual "draw" function
    - Then, we hold a pointer to button, and assign a specific button object, so that the correct draw() function is invoked each time
  - We probably need to dynamically create a lot of this objects

Abstract factory

- A program must be able to choose one of several families of classes
- Example,
  - a program's GUI should run on several platforms
  - Each platform comes with its own set of GUI classes:
    - WinButton, WinScrollBar, WinWindow MotifButton, MotifScrollBar, MotifWindow, pmButton, pmScrollBar, pmWindow
  - Inheritance:
    - Clearly, we can make all "button" classes derive from an abstract button that implements a virtual "draw" function
    - Then, we hold a pointer to button, and assign a specific button object, so that the correct draw() function is invoked each time
  - We probably need to dynamically create a lot of this objects
  - Problem: how can we simplify the creation of these objects?

Naive approach

- We keep a global variable (or object) that represents the current window manager and "look-and-feel" for all the objects
- Every time we create an object, we execute a switch/case on the global variable to see which object we must create

```
lf = getWinManagerTypeString();
// need to create a button
switch(lf) {
case "WIN":   button = new WinButton(...);
              break:
case "MOTIF": button = new MotifButton(...);
              break;
case "PM":    button = new PmButton(...);
              ...
}
```

Problems with the naive approach

- What happens if we need to add a new look-and-feel?
  - We must change lot of code (for every creation, we must add a new case)

Problems with the naive approach

- What happens if we need to add a new look-and-feel?
  - We must change lot of code (for every creation, we must add a new case)
- How much code must we use?
  - Assuming that each look and feel is part of a different library, all libraries must be linked together
  - Large amount of code

Problems with the naive approach

- What happens if we need to add a new look-and-feel?
  - We must change lot of code (for every creation, we must add a new case)
- How much code must we use?
  - Assuming that each look and feel is part of a different library, all libraries must be linked together
  - Large amount of code
- This solution is not compliant with the open/closed principle
  - Every time we add a new look and feel, we must change the code of existing functions/classes

Problems with the naive approach

- What happens if we need to add a new look-and-feel?
  - We must change lot of code (for every creation, we must add a new case)
- How much code must we use?
  - Assuming that each look and feel is part of a different library, all libraries must be linked together
  - Large amount of code
- This solution is not compliant with the open/closed principle
  - Every time we add a new look and feel, we must change the code of existing functions/classes
- This solution *does not scale*

Requirements

- Uniform treatment of every button, window, etc.
  - Once you define the interface, you can easily use inheritance
- Uniform object creation
- Easy to switch between families
- Easy to add a family

Solution: Abstract factory

- Define a *factory* (i.e. a class whose sole responsibility is to create objects)

```
interface WidgetFactory {
    Button makeButton(args);
    Window makeWindow(args);
    // other widgets...
}
```

Solution: Abstract factory

- Define a *factory* (i.e. a class whose sole responsibility is to create objects)

```
interface WidgetFactory {
    Button makeButton(args);
    Window makeWindow(args);
    // other widgets...
}
```

- Define a concrete factory for each of the families

```
class WinWidgetFactory implements WidgetFactory {
    Button makeButton(args) {
        return new WinButton(args);
    }
    Window makeWindow(args) {
        return new WinWindow(args);
    }
}
```

Solution - cont.

- Select once which family to use:

```
WidgetFactory wf;
lf = getWinManagerTypeString();
switch (lf) {
case "WIN":     wf = new WinWidgetFactory();
                break;
case "MOTIF":   wf = new MotifWidgetFactory();
                break;
...
}
```

- When creating objects in the code, don't use "new" but call:

```
Button b = wf.makeButton(args);
```

Solution - cont.

- Select once which family to use:

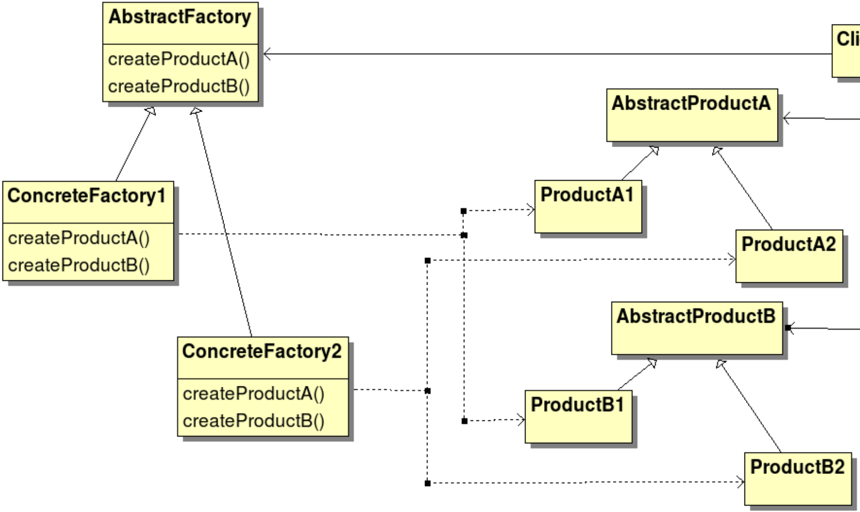
```
WidgetFactory wf;
lf = getWinManagerTypeString();
switch (lf) {
case "WIN":     wf = new WinWidgetFactory();
                break;
case "MOTIF":   wf = new MotifWidgetFactory();
                break;
...
}
```

- When creating objects in the code, don't use "new" but call:

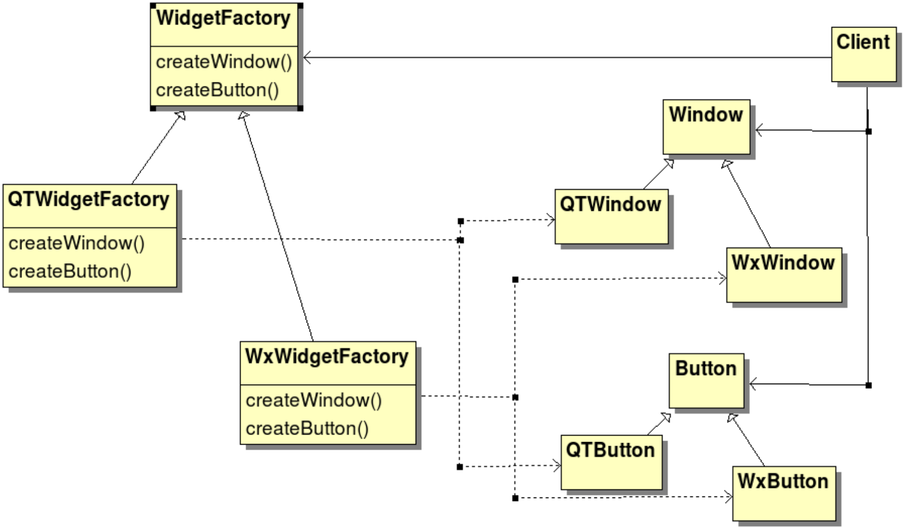
```
Button b = wf.makeButton(args);
```

- Switch families – once in the code
- Add a family – one new factory, no effect on existing code

UML diagram


AbstractFactory
createProductA()
createProductB()
Client
AbstractProductA
ConcreteFactory1
createProductA()
createProductB()
ProductA1
ProductA2
AbstractProductB
ConcreteFactory2
createProductA()
createProductB()
ProductB1
ProductB2

UML diagram, applied



Participants

- **AbstractFactory** (WidgetFactory)
  - declares an interface for operations that create abstract product objects.
- **ConcreteFactory** (MotifWidgetFactory, PMWidgetFactory)
  - implements the operations to create concrete product objects.
- **AbstractProduct** (Window, ScrollBar)
  - declares an interface for a type of product object.
- **ConcreteProduct** (MotifWindow, MotifScrollBar)
  - 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.

Comments

- Pros:
  - *It makes exchanging product families easy*. It is easy to change the concrete factory that an application uses. It can use different product configurations simply by changing the concrete factory.
  - *It promotes consistency among products*. When product objects in a family are designed to work together, it's important that an application uses objects from only one family at a time time.
  - AbstractFactory makes this easy to enforce.
- Cons:
  - Not easy to extend the abstract factory's interface
- Other patterns:
  - Usually one factory per application, a perfect example of a singleton

Known uses

- Different operating systems (could be Button, could be File)
- Different look-and-feel standards
- Different communication protocols

Outline

Intent

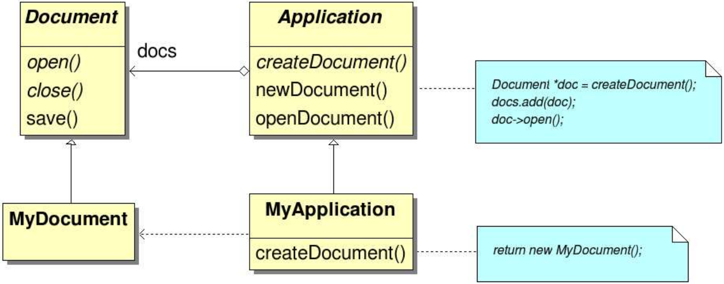
Define an interface for creating an object, but let subclasses decide which class to instantiate

- Also known as *Virtual Constructor*
- The idea is to provide a virtual function to create objects of a class hierarchy
- each function will then know which class to instantiate

Example

- Consider a framework for an office suite
  - Typical classes will be `Document` and `Application`
  - there will be different types of documents, and different types of applications
  - for example: Excel and PowerPoint are applications, excel sheet and presentation are documents
  - all applications derive from the same abstract class *Application*
  - all documents derive from the same abstract class *Document*
  - we have *parallel hierarchies* of classes
  - every application must be able to create its own document object

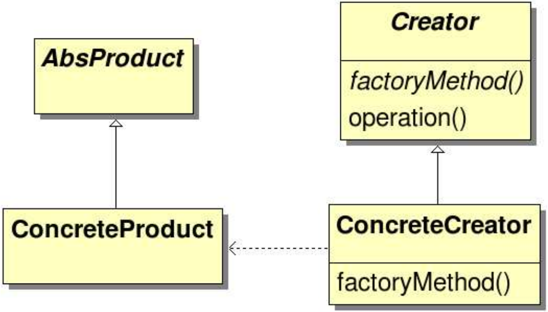
Example in UML



Participants

- **Product** (Document)
  - defines the interface of the objects the factory method creates
- **ConcreteProduct** (MyDocument)
  - implements the Product's interface
- **Creator** (Application)
  - declares the factory method
- **ConcreteCreator** (MyApplication)
  - overrides the factory method to return an instance of a ConcreteProduct

UML representation



Implementation

- It may be useful to select the factory method by using a parameter, to allow the creation of multiple types of products
  - For example, suppose that you want to save a bunch of different objects on the disk (Triangle, Rectangle, Circle, etc, they are all of type shape)
  - one possibility would be to enumerate the types with an integer `id`, and save the `id` as first element in the disk record
  - when loading the objects again you may read the `id` first, and then pass it to a factory which creates the correct type of object and loads it from the disk
  - further, to avoid a switch-case in the factory, we could implement a registry (will see in a little how to do this)

Outline

How to create objects

- Usually, objects are created by invoking the constructor
- however, sometimes the constructor is not as flexible as we wish
- an alternative technique is to use a static method in the class, whose purpose is to create objects of the class in a more flexible way
- this technique is called *static factory method*
  - has almost nothing to do with the GoF's factory method

```
class MyClass {
    public MyClass(int param);
// std constructor
    static public MyClass create(int param);
// static fact. method
};
```

Advantages

- The first advantage is that factory methods can have descriptive names
- This is especially useful when there are many different ways to create an object
  - the standard way is to implement many constructors with different argument lists
  - however, the code readability of this technique is poor: it is difficult to understand what a certain constructor does by just looking at the list of parameters
  - sometimes, constructors differ just in the order of the parameters!
- with static factory methods, instead:
  - It is possible to create different methods with different, more descriptive names

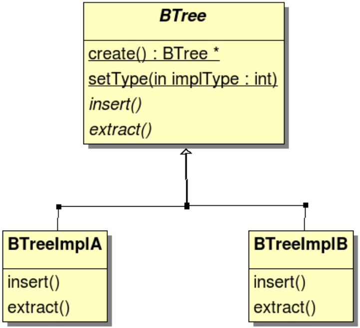
Advantages

- The second important advantage is that, unlike constructors, static factory methods must not necessarily create an object
    - This can be useful for example when you want to control how many objects are around, and eventually reuse them
    - For example, this technique is very useful when implementing an enumeration of constant objects

Advantages

- The second important advantage is that, unlike constructors, static factory methods must not necessarily create an object
  - This can be useful for example when you want to control how many objects are around, and eventually reuse them
  - For example, this technique is very useful when implementing an enumeration of constant objects
- The third advantage is the fact that they can create an object of a subtype of the original type, without the client knowing this fact
  - Suppose for example that you implemented a `BTree` class
  - The client code uses the interface of BTree to perform operations like insert/extract
  - Then, you realize that you need different implementation of BTree in different contexts, because of performance / efficiency reasons
  - If the BTree is created with a factory method, you can simply switch between the implementations by configuring the method differently

Implementation



- Notice that the two implementation classes need not to be exposed to the client: they can be completely hidden, and changed at any time without even informing the customer
- the extra function `setType()` can be optionally used to let the client select the preferred implementation
- therefore, we have maximum separation of concerns

Hiding the constructor?

- The static factory method looks similar to the singleton pattern (except that there is no limit to the number of instances)
- You might be tempted to make the constructor private, so the only way to construct an instance is to use the static factory method
- however, keep in mind that, if the constructor is private, the class cannot be sub-classed
    - The derived class cannot call the base class constructor!
- therefore, if you want to sub-class, the constructor must be at least protected

Other advantages

- Another advantage is the fact that you can easily specify default parameters between successive calls
- this reduces the list of parameters of complex constructors
  - This is sometimes called *telescoping constructor*

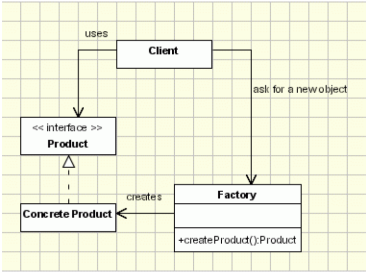
```
class NutritionFacts {
    public NutritionFacts(int servingSize, int servings)
           {...}
    public NutritionFacts(int servingSize, int servings, int calories)
           {...}
    public NutritionFacts(int servingSize, int servings, int calories,
                   int fat) {...}
    public NutritionFacts(int servingSize, int servings, int calories,
                   int fat, int sodium) {...}
};
...
NutritionFacts label1(240, 8, 100, 0, 35, 27);
NutritionFacts label2(240, 8, 100, 0, 42, 25);
NutritionFacts label3(300, 10, 100, 0, 42, 25);
```

With static factory method

- see `SimpleBuilder` example
- notice how much more readable it is
- Notes:
  - once all parameters have been set, they can be checked in the `NutritionFacts` constructor
  - The setting order does not matter
  - This method can be extended to consistently build more complex objects step by step (see Builder Pattern)

Outline

The UML diagram:



- Suppose you have to create one of many product types
  - For example, you could use an ID (an integer, or a String) to identify the product type
  - Therefore, `createProduct()` should take the ID and return the specific product

The switch/case approach

```
public class ProductFactory{
    public Product createProduct(String id){
        if (id==ID1)
            return new OneProduct();
        if (id==ID2)
            return new AnotherProduct();
        ... // so on for the other Ids

        //if the id doesn't have any of the expected values
        return null;
    }
    ...
}
```

- Can you tell why this is bad? Which principle does it violate?

Reflection

- We will now see how to use *reflection* to solve the problem
- The idea is to maintain a data structure that establishes a correspondence between the ID and the class to be created
  - At program start-up, each class *registers* itself on the data structure
  - the `createProduct(ID)` function will perform a look-up in the data structure to select the class, and invokes the corresponding constructor
- *Reflection enables Java code to discover information about the fields, methods and constructors of loaded classes, and to use reflected fields, methods, and constructors to operate on their underlying counterparts*

Registry with reflection

The product family ReflectDemo.java

```
interface Product {
    String getName();
}

class ProductOne implements Product {
    static {
        ProductFactory.instance().register("One", ProductOne.class);
    }
    public String getName() { return "instance of ProductOne"; }
}

class ProductTwo implements Product {
    static {
        ProductFactory.instance().register("Two", ProductTwo.class);
    }
    public String getName() { return "instance of ProductTwo"; }
}
```

- Note how the products get registered in the Factory

The ProductFactory

ReflectDemo.java

```
class ProductFactory {
    // Singleton
    static private ProductFactory inst = new ProductFactory();
    static public ProductFactory instance() { return inst; }

    // The registry
    private HashMap registry = new HashMap();
    public void register(String productID, Class productClass) {
        registry.put(productID, productClass);
    }
    public Product create(String ID) {
        Class pClass = (Class)registry.get(ID);
        if (pClass == null) {
            System.err.println("Product " + ID + " not registered");
            return null;
        }
        try {
            Constructor pConstructor = pClass.getDeclaredConstructor(null);
            return (Product)pConstructor.newInstance(null);
        } catch (NoSuchMethodException e) {
```

- This code snippet uses `Class`, and `Constructor` classes from `java.lang.reflect`

Usage

- We must get sure that all classes get loaded before we access use the factory

ReflectDemo.java

```
public class ReflectDemo {
    static public void main(String args[]) {
        try {
            Class.forName("ProductOne");
            Class.forName("ProductTwo");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        Product p1 = ProductFactory.instance().create("One");
        System.out.println("I have created a " + p1.getName());

        Product p2 = ProductFactory.instance().create("Two");
        System.out.println("I have created a " + p2.getName());
    }
}
```

Without reflection

- Now, suppose we do not want to use reflection
  - The idea is to *map* ID to objects, and then call their factory method (virtual constructor) to create a new object of the kind

RegistryDemo.java

```
interface Product {
    String getName();
    Product create();
}

class ProductOne implements Product {
    static {
        ProductFactory.instance().register("One", new ProductOne());
    }
    public String getName() { return "instance of ProductOne"; }
    public ProductOne create() { return new ProductOne(); }
}

class ProductTwo implements Product {
    static {
        ProductFactory.instance().register("Two", new ProductTwo());
    }
    public String getName() { return "instance of ProductTwo"; }
```

The factory

RegistryDemo.java

```
class ProductFactory {
    // Singleton
    static private ProductFactory inst = new ProductFactory();
    static public ProductFactory instance() { return inst; }

    // The registry
    private HashMap<String, Product> registry = new HashMap<String, Produc
    public void register(String ID, Product p) {
        registry.put(ID, p);
    }
    public Product create(String ID) {
        Product p = registry.get(ID);
        if (p == null) {
            System.err.println("Product " + ID + " not registered");
            return null;
        }
        return p;
    }
}
```

Usage

RegistryDemo.java

```
public class RegistryDemo {
    static public void main(String args[]) {
        try {
            Class.forName("ProductOne");
            Class.forName("ProductTwo");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }

        Product p1 = ProductFactory.instance().create("One");
        System.out.println("I have created a " + p1.getName());

        Product p2 = ProductFactory.instance().create("Two");
        System.out.println("I have created a " + p2.getName());
    }
}
```

Generalization?

- Is it possible to generalise this code by using generics?

```
public class RegistryFactory<E extends VirtualConstructor> {
    // Singleton
    static private RegistryFactory<E> inst = new RegistryFactory<E>();
    static public RegistryFactory<E> instance() { return inst; }

    // The registry
    private HashMap<String, E> registry = new HashMap<String, E>();
    public void register(String ID, E p) {
        registry.put(ID, p);
    }
    public E create(String ID) {
        E p = registry.get(ID);
        if (p == null) {
            System.err.println("Product " + ID + " not registered");
            return null;
        }
        return p.create();
    }
}
```

Singleton and generics

- Unfortunately the previous solution does not work
  - It is not possible to have a static variable (`inst`) that depends on a generic parameter (E)
  - This is a limitation of Java (it is possible in C++)
  - It has to do with how generics are implemented
- Therefore, the only solution is to make the `create()` return a reference to the interface

Outline

Bibliography

- Cristopher Alexander, Sara Ishikawa, Murray Silverstein, Max Jacobson, Ingrid Fiksdhal-King, and Shlomo Angel.
  A pattern language.
  Oxford University Press, 1997.

- Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides.
  Design patterns: elements of reusable object-oriented software.
  Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

- Barbara Liskov.
  Data abstraction and hierarchy.
  SIGPLAN Notice, 23(5), 1988.

- Bertrand Meyer.
  Object-Oriented Software Construction.
  Prentice Hall, 1988.