Comp-303 : Programming Techniques
Lecture 18

Alexandre Denault
Computer Science
McGill University
Winter 2004

Last lecture . . .

- The origins of Design Patterns can be traced back to Architecture.
- By using Design Patterns, we are re-using solutions to well know problems.
- The Singleton pattern should be used when only one copy of an object should be created.
- There are two ways to implement singletons in Java, each with their own advantages.

Typical Situations

- Image you are building the GUI for a new graphic intensive application.
- Given that some users have very old computers, you want to offer two graphical interfaces.
- However, you don't want to create two separate applications, since most of the code remains the same.

Graphical Interface

- Your GUI was built using a fixed number of components:
  - Panels
  - Buttons
  - Labels
  - Text Boxes
  - Scroll Bars
- How can you build your applications without having to code both GUIs separately?

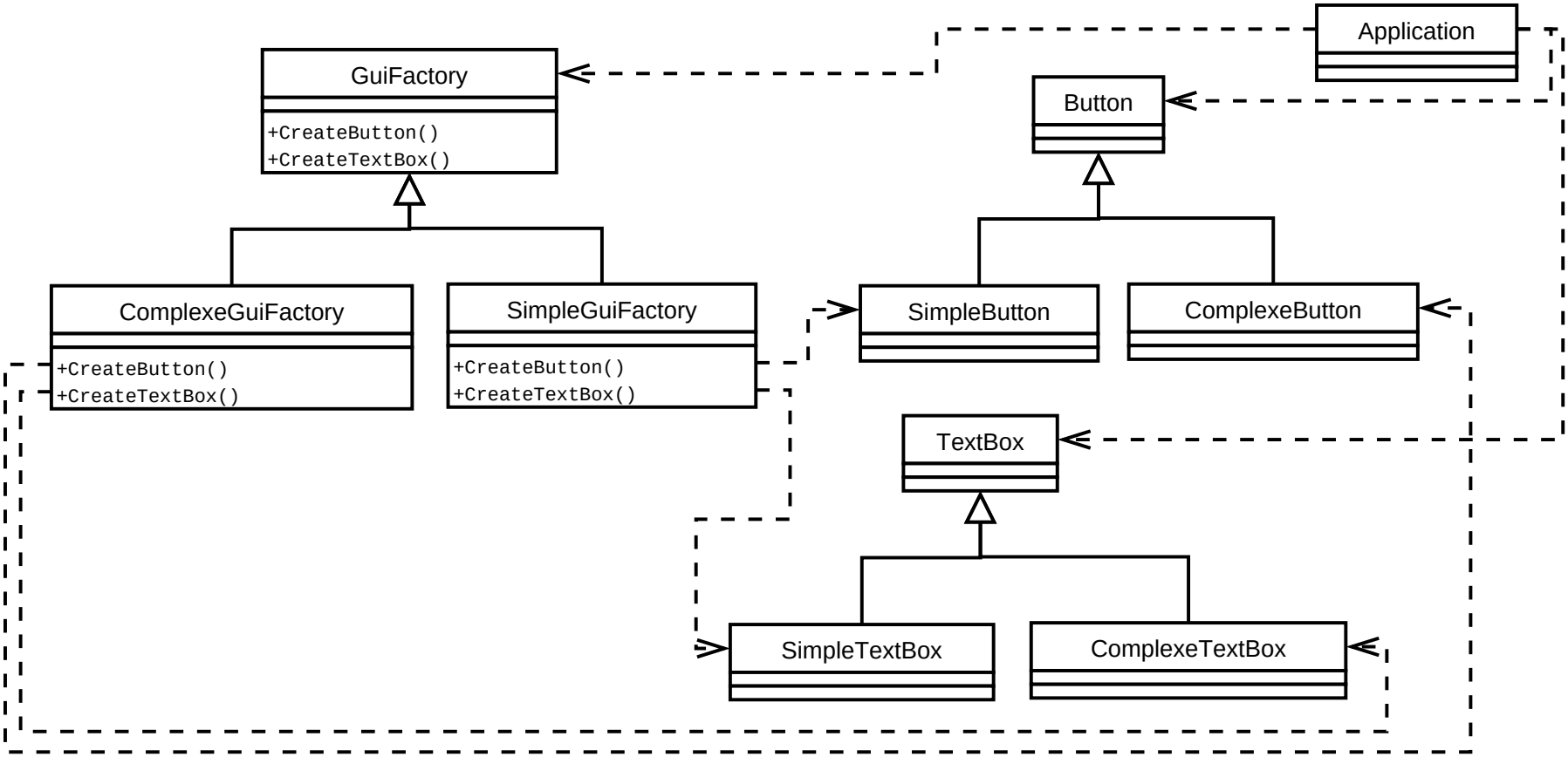
Abstract Factory

- *Pattern Name* : Abstract Factory
- *Classification* : Creational
- *Also known as* : Kit
- *Intent* : Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Motivation

- In our example, we have two families of GUIs, a complex and a simple one.
- Abstract Factory will allow us to build code compatible for both GUIs.
- First step is to build two classes for each components.
  - ComplexPanel and SimplePanel
  - ComplexButton and SimpleButton
  - ComplexLabel and SimpleLabel
  - ComplexTextBox and SimpleTextBox
  - ComplexScrollBar and SimpleScrollBar

Motivation (cont.)



Motivation (cont.)

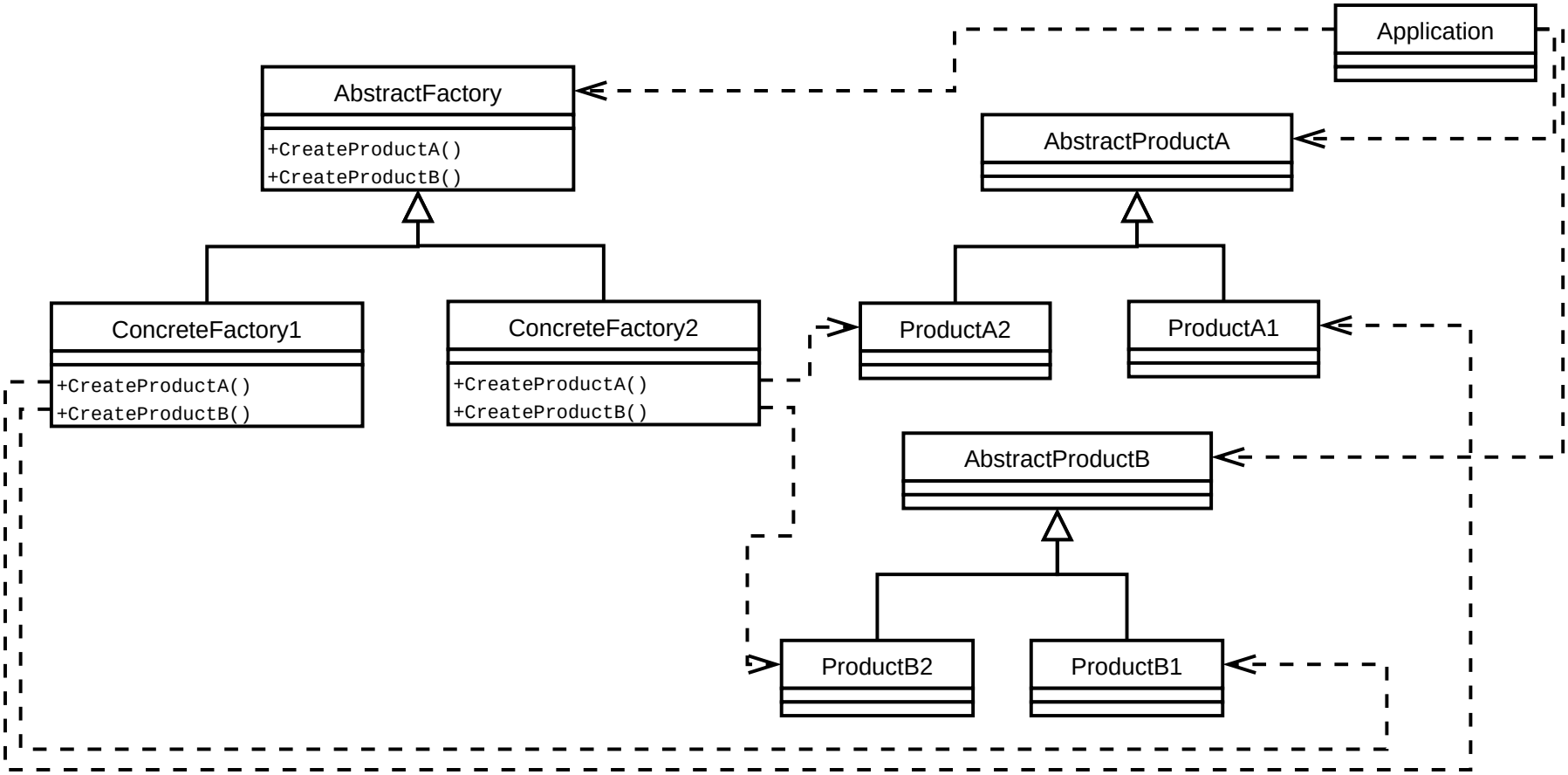
- Users would only know about the GuiFactory, TextBox and Button classes.
- Depending on the type of Factory that is actually create, we would be using either Simple objects or Complex objects.

Applicability

The Abstract Factory pattern should be used when . . .

- . . . a system should be independent of how its products are created, composed and represented.
- . . . a system should be configured with one of multiple families of products.
- . . . a family of related product objects is designed to be used together, and you need to enforce this constant.
- . . . you want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.

Structure


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

Participants

- Abstract Factory: declares an interface for operations that create abstract products objects.
- Concrete Factory: implements the operations to create concrete products objects.
- Abstract Product: declares an interface for a type of product object.
- Concrete Product: defines a product object to be create by corresponding concrete factory.
- Application: uses only the Abstract Factory and the Abstract Product.

Collaborations

- In a normal situation, only one concrete factory should exist at any given time.
- However, if an application requires objects from both family, then a second concrete factory must be created.
- Even though the application uses abstract factory, objects are create by concrete factory.

Consequences

- The concrete class are easy to isolate. You can make sure that only one family of class can be created.
- You can change between families pretty easily.
- You get an increase consistency with your concrete objects.
- It is difficult to add new products.

Implementation

- Factories are typically singletons. You should only have one factory at any given time.
- The abstract factory and the abstract products should be interfaces or abstract classes. You should not be able to instantiate them.
- Concrete factories are subclasses of the abstract factory.

Example

```
public abstract class GuiFactory {

  public abstract Button createButton();
  public abstract TextBox createTextBox();
}

public abstract class Button {

  // Insert methods related to button here
  public abstract void draw();
}

public abstract class TextBox {

  // Insert methods related to TextBox here
  public abstract void draw();
}
```

Using the example

```
...

FactoryGui myGui = SimpleFactoryGui.instance();

Button myButton = myGui.createButton();
TextBox myTextBox = myGui.createTextBox();

myButton.draw();
myTextBox.draw();

...
```

Known Uses

- Many graphic interfaces uses the abstract factory patterns.
- It allows them to change the *motif* the screen without having to rewrite the application.

Related Patterns

- Factory Methods
- Singleton
- Prototype

Something simpler

- In some situation, building an abstract factory is overkill.
- For example, we saw the Poly class with had two concrete implementation (densePoly and sparsePoly).
- The choice between a densePoly and sparsePoly should be done at run time.
- We only need a method to choose between two implementations, not two families.

Factory Method

- *Pattern Name* : Factory Method
- *Classification* : Creational
- *Also known as* : Virtual Constructor
- *Intent* : Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Motivation

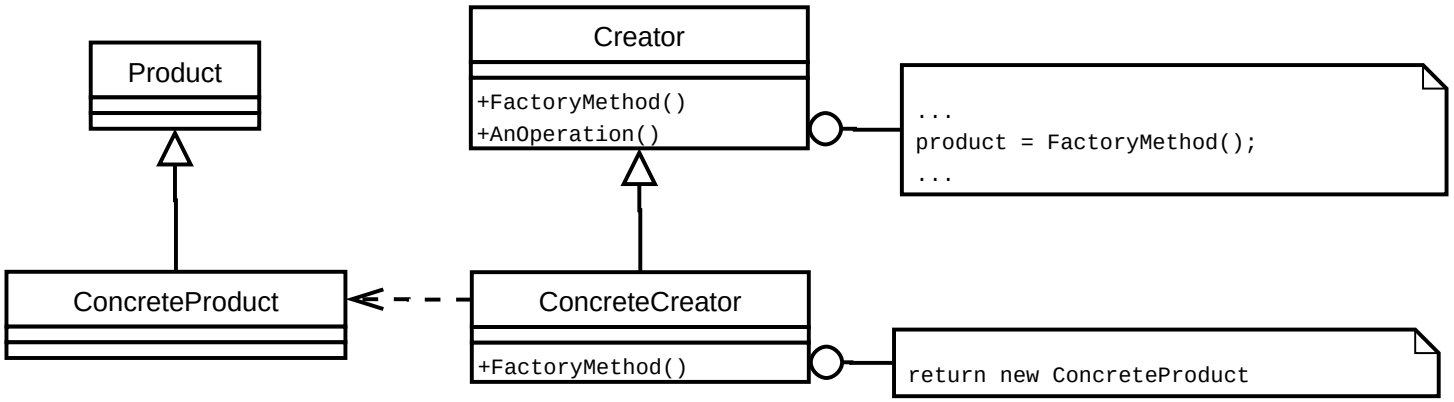
- Consider an application that heavily uses polynomial calculations.
- When low on memory, the application has a tendency to create sparsePoly because they save space.
- However, sparsePoly require more time to add and multiply, thus the application favors densePoly in normal situations.
- This application has two factory method, one that is used in normal situations and one that is used when the application is low on memory.

Applicability

The Factory Method pattern should be used when ...

- ... a class can't anticipate the class of objects it must create.
- ... a class wants its subclasses to specify the objects it creates.
- ... classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Structure


Product
Creator
+FactoryMethod()
+AnOperation()
...
product = FactoryMethod();
...
ConcreteProduct
ConcreteCreator
+FactoryMethod()
return new ConcreteProduct

Participants

- Product: define the interface of objects the factory method creates.
- ConcreteProduct: implements the Product Interface.
- Creator: declares the factory method, which returns an object of type Product.
- ConcreteCreator: overrides the factory method to return an instance of a concreteProduct.

Collaborations

- Creator relies on its subclass to define the factory method and return the appropriate product.

Consequences

- Provides hooks for subclasses. In other word, it's easy to add another decision process since we only need to subclass Creator.

Implementation

- In certain case, you might want the creator class to be concrete. This would give the method factory a default implementation.
- Some factory methods are parametrized. In other words, a parameter is used to influence to choice of which object to create.

Example

```
public class PolyCreator {

  public Poly createPoly(String polynomialString) {
    return factory(polynomialString);
  }

  private Poly factory (String polynomialString) {

    int i = number of terms in polynomialString
    int j = highest degree of polynomialString

    if ( j/i > 6 ) {
      return a densePoly
    } else {
      return a sparsePoly
    }
  }
}
```

Example

```
public class PolyCreatorLowMemory extends PolyCreator {

  private Poly factory (String polynomialString) {

    int i = number of terms in polynomialString
    int j = highest degree of polynomialString

    if ( j/i > 3 ) {
      return a densePoly
    } else {
      return a sparsePoly
    }
  }
}
```

Using the example

```
...

PolyCreator myPolyCreator;

if (free memory is low) {
  myPolyCreator = new PolyCreatorLowMemory();
} else {
  myPolyCreator = new PolyCreator();
}

Poly myPoly = myPolyCreator.createPoly("x^2+3*x+1");

...
```

Known Uses

- Factory methods are common in frameworks.
- The Java API uses some Factory methods when dealing with I/O streams.

Related Patterns

- Abstract Factory
- Template
- Prototype

Abstract Factory vs Factory Method

- Scope
  - An abstract factory will product many objects of different types (families).
  - Factory method produce objects of one type.
- Variance
  - An abstract factory is typically initialize only once.
  - Factory methods have a tendency to change.

Switch Statements

- Often, the presence of a switch statement will indicate
  - the need for polymorphic behavior.
  - the presence of misplaced responsibilities.
- Consider instead a more general solution such as abstraction or giving the responsibility to other objects.

Summary

- Abstract Factories allow you to use families of objects in a generic way.
- Factory methods allow you to choose the appropriate subclass at runtime.

Tool of the day: ProGuard

- ProGuard is a free Java class file shrinker and obfuscator.
- It can detect and remove unused classes, fields, methods, and attributes.
- It can then rename the remaining classes, fields, and methods using short meaningless names.
- The resulting jars are smaller and harder to reverse-engineer.
- More information on ProGuard is available at:

  `http://proguard.sourceforge.net/`

References

- These slides are inspired (i.e. copied) from these three books.
  - Design Patterns, Elements of Reusable Object-Oriented Software; Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides; Addison Wesley; 1995
  - Java Design Patterns, a Tutorial; James W. Cooper Addison Wesly; 2000
  - Design Patterns Explained, A new Perspective on Object Oriented Design; Alan Shalloway, James R. Trott; Addison Wesley; 2002