CSCI 253

Object Oriented Design:
Factory Method Pattern
George Blankenship

Overview

<u>Creational Patterns</u>
- 🖫 Singleton
- 🗐 Abstract factory
- 🗐 Factory Method
- 🗐 Prototype
- 🗐 Builder

<u>Structural Patterns</u>
- 🗐 Composite
- 🗐 Façade
- 🗐 Proxy
- 🗐 Flyweight
- 🖫 Adapter
- 🖫 Bridge
- 🖫 Decorator

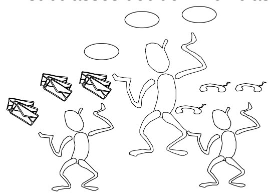
<u>Behavioral Patterns</u>
- 🖫 Chain of Respons.
- 🖫 Command
- 🖫 Interpreter
- 🗐 Iterator
- 🖫 Mediator
- 🖫 Memento
- 🗐 Observer
- 🗐 State
- 🗐 Strategy
- 🖫 Template Method
- 🗐 Visitor

The Elements of a Design Pattern

- A pattern name
- The problem that the pattern solves
  - Including conditions for the pattern to be applicable
- The solution to the problem brought by the pattern
  - The elements (classes-objects) involved, their roles, responsibilities, relationships and collaborations
  - Not a particular concrete design or implementation
- The consequences of applying the pattern
  - Time and space trade off
  - Language and implementation issues
  - Effects on flexibility, extensibility, portability

The Factory Method Pattern: The Problem

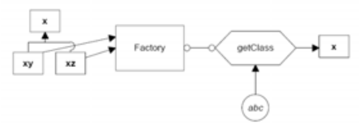
Define an interface for creating an object but let subclasses decide which class to instantiate



Framework (toolkit) uses abstract classes to define and maintain relationships between objects and is responsible for creating the objects as well

Also known as : Virtual constructor

Factory Method Mechanics



- ***x*** is a base class and classes ***xy*** and ***xz*** are derived from it.
- The factory is a class that decides which of these subclasses to return depending on the arguments you give it.
- The *getClass* method receives the value *abc*, and that returns an instance of the class ***x***.
- Each instance has the same methods, but different implementations.
- The instance decision is entirely embedded in the factory and could be very complex but is often quite simple; the factory manufactures the object.

Entry Form

- Entry form that allows the user to enter a name either as “firstname lastname” or “lastname, firstname”
- Simplifying assumption that the name order is indicated by the existence of a comma between the last and first name.

Factory Method Class

```
class NameFactory {
// returns an instance of LastFirst or FirstFirst
// depending on whether a comma is found
    static 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
    }
}
```

Factory Method Pattern George Blankenship 7

Factory Method Worker Classes

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

Factory Method Pattern George Blankenship 8

Name Divider Example

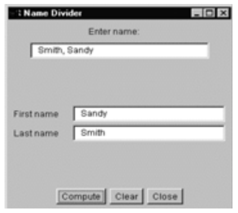
Callback for “Enter name:”

Namer name =
  Name.getName(EnterName.getText())

name is LastFirst object

FirstName.setText(name.firstName());

LastName.setText(name.lastName());



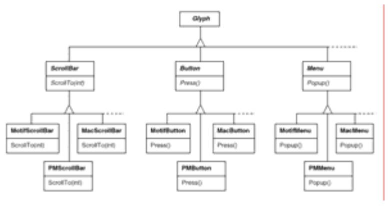
Enter name: Smith, Sandy

First name: Sandy

Last name: Smith

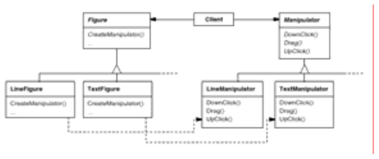
Factory Method Pattern George Blankenship 9

GUI Widgets



Factory Method Pattern George Blankenship 10

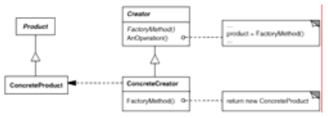
Toolset Generation



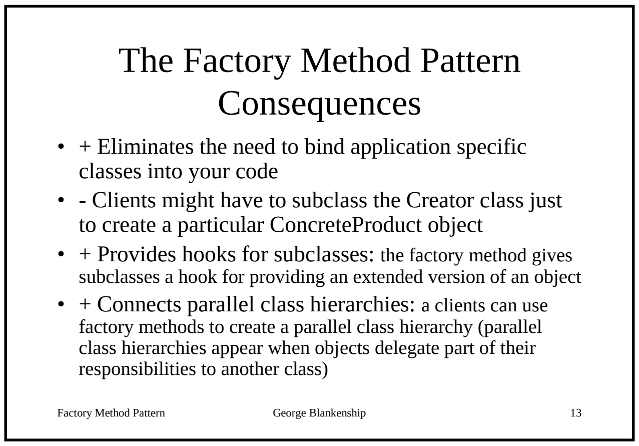
Factory Method Pattern George Blankenship 11

Factory Method Classes

- *Creator* is parent
  - May use static method *FactoryMethod*()
  - May be instantiated as a Factory object
- Set of *ConcreateCreator* classes used to create *ConcreteProduct* (objects)



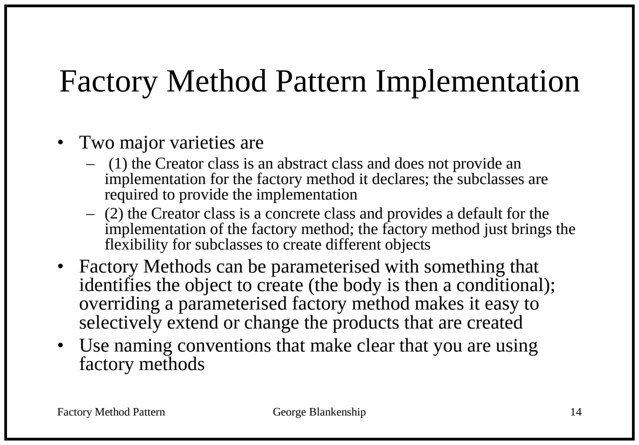
Factory Method Pattern George Blankenship 12



The Factory Method Pattern Consequences

- + Eliminates the need to bind application specific classes into your code
- - Clients might have to subclass the Creator class just to create a particular ConcreteProduct object
- + Provides hooks for subclasses: the factory method gives subclasses a hook for providing an extended version of an object
- + Connects parallel class hierarchies: a clients can use factory methods to create a parallel class hierarchy (parallel class hierarchies appear when objects delegate part of their responsibilities to another class)

Factory Method Pattern George Blankenship 13



Factory Method Pattern Implementation

- Two major varieties are
  - (1) the Creator class is an abstract class and does not provide an implementation for the factory method it declares; the subclasses are required to provide the implementation
  - (2) the Creator class is a concrete class and provides a default for the implementation of the factory method; the factory method just brings the flexibility for subclasses to create different objects
- Factory Methods can be parameterised with something that identifies the object to create (the body is then a conditional); overriding a parameterised factory method makes it easy to selectively extend or change the products that are created
- Use naming conventions that make clear that you are using factory methods

Factory Method Pattern George Blankenship 14