Supplementary Materials to
Design Patterns

ATSD Fall 2003
10/20/2003

Recommended Readings

Thinking in Patterns with Java
Recommended Readings

- Patterns-Discussion FAQ
  http://gee.cs.oswego.edu/dl/pd-FAQ/pd-FAQ.html
- Java Design Patterns in JavaWorld
- A Learning Guide To Design Patterns
- Patterns Home Page
  http://hillside.net/patterns/

Idioms, Patterns, Frameworks

- Idiom: a small language-specific pattern or technique
  - A more primitive building block
- Design pattern: a description of a problem that reoccurs and an outline of an approach to solving that problem
  - Generally domain, language independent
  - Also, analysis patterns
- Framework:
  - A partially completed design that can be extended to solve a problem in a domain
    - Horizontal vs. vertical
Examples of C++ Idioms

• Use of an Init() function in constructors
  – If there are many constructors, make each one call a private function Init()
    • Init() guarantees all possible attributes are initialized
    • Initialization code in one place despite multiple constructors
• Don’t do real work in a constructor
  – Define an Open() member function
    • Constructors just do initialization
    • Open() called immediately after construction
  – Constructors can’t return errors
    • They can throw exceptions

Another Example of the Idiom

• Memory acquisition in C++
  – C++ doesn’t do garbage collection
  – For every ‘new’ operation you write you have to write a corresponding ‘delete’

```cpp
void use_buffer(size_t x) {
  char* buffer = new char[x];
  // use buffer
  delete[] buffer;
}
```

```cpp
class Buffer { // Uses Idiom “Resource acquisition is initialisation”
  char* p;
  public: Buffer(size_t x) { p = new char[x]; }
  ~Buffer() { delete[] p; }
};
```
Patterns Are (and Aren’t)

- Name and description of a proven solution to a problem
- Documentation of a design decision
- They’re not:
  - Reusable code, class libraries, etc. (At a higher level)
  - Do not require complex implementations
  - Always the best solution to a given situation
  - Simply “a good thing to do”

Creational Patterns

- Creational design patterns abstract the instantiation process
- Encapsulate knowledge about which concrete classes the system uses ..
- Hide how instances of these classes are created and put together
- Result: Modules can be designed independent of how objects are created, composed and represented ...
Singleton Pattern

- Context: Only one instance of a class is created. Everything in the system that needs this class interacts with that one object.
- Controlling access: Make this instance accessible to all clients
- Solution:
  - The class has a static variable called `theInstance` (etc)
  - The constructor is made `private` (or `protected`)
  - Clients call a public operation `getInstance()` that returns the one instance
    - This may construct the instance the very first time or be given an initializer

Singleton: Java implementation

```java
public class MySingleton {
    private static MySingleton theInstance =
        new MySingleton();
    private MySingleton() { // constructor
        ...
    }

    public static MySingleton getInstance() {
        return theInstance;
    }
}
```
Factory Method

General Description
• Problem
  – Sometimes, an Application cannot anticipate the class of object that it must create. The Application may know that it has to instantiate classes, but it may only know about abstract classes (or interfaces), which it cannot instantiate. Thus the Application class may only know when it has to instantiate a new Object of a class, not what kind of subclass to create. Another motivation could be that a class may want its subclasses to specify the objects to be created.
• Solution
  – Model an interface for creating an object which at creation time can let its subclasses decide which class to instantiate. This is known as a Factory Pattern since it is responsible for “Manufacturing” an Object. It helps instantiate the appropriate subclass by creating the right object from a group of related classes.
  – Factory Method is a class creational pattern – “uses inheritance to vary the class that’s instantiated. Thus, the factory method lets a class defer instantiation to subclasses” (DP - 81,107)

Why is this pattern significant?
• If you just need to create an object, a factory is not always needed, this can be easily done with a simple call to new(). However, the use of this pattern gives the programmer the opportunity to abstract the attributes of an object to the specific subclasses that create them.
• Factory methods promote loose coupling and encourages encapsulation by “eliminating the need to bind application-specific classes into the code
• Using this pattern gives the designer more flexibility in the code, say for example – adding new types of objects that the class creates, which are not known initially. This can be done without modifying the code in the parent abstract class, which only provides a definition for the factory method. The new subclasses will have their own implementation for creating the objects.
• Why would the pattern be used? Drawing from the problem domain, the pattern would be most useful when it is necessary to encapsulate the knowledge of which subclasses a framework would create and help move that knowledge out of the framework.
Factory Method

Participants

• Product
  – Defines 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. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
• ConcreteCreator
  – Overrides the factory method to return an instance of a ConcreteProduct

Factory Method: UML (Class Diagram)
Factory Pattern

- Sometimes, an Application cannot anticipate the class of object that it must create.
  ```java
  aClass anObject = new aClass();
  // hardcoding class name
  ```

- Instead, we use a factory:

```
public class Parent
{    protected String first;    protected String second;    ... }

public class Child1 extends Parent {}    public class Child2 extends Parent {}    public class ParentFactory {        public Parent getParent(String argument) // Factory method        {            if (argument.equals(...))                return new Child1(argument);            else                return new Child2(argument);        }    }
```

Factory pattern: sample code
Creating Account Objects in the Banking System

Example

```java
Account selectedAccount = new Account(accountID);
```

Add a “createAccount(String)” method to the Bank class.
The Abstract Factory Method

- The Problem: We want to be able to encapsulate the implementation of class creation so that parallel hierarchies of classes can be created and clients can avoid hard-coding the name of the class to be constructed.

![Diagram of Window, MacWindow, Win95 Window, MotifWindow]

The Bank Factory

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes

![Diagram of BankFactory, Client, Bank, Customer, Account, Transaction]
The MegaBankFactory

*Only one instance of a ConcreteFactory (Singleton) per product family*

The Generic Model for the Abstract Factory Method