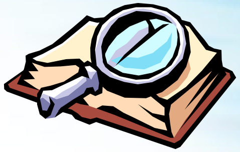
The Factory Pattern

Chapter Content

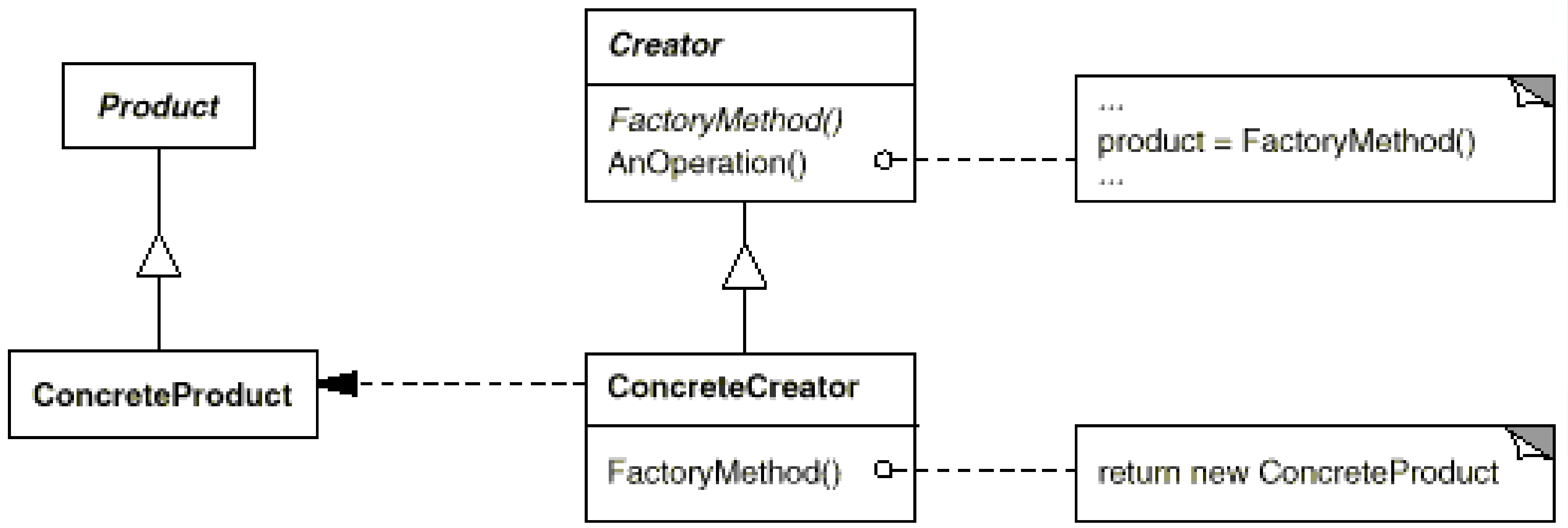


- The Factory Pattern
- Factory UML Diagram
- Factory – the basic pattern
- Simple Example
- Bank Example
  - Products
  - Factory
  - Usage
- Factory variations

The Factory Pattern

- A **very common** pattern.
- Factory is responsible for allocating Product instances (instead of direct constructor calls).
- Factory can select between **several possible sub-classes** depending on conditions:
  - System properties, optimizations, resource bundles...
- Benefits:
  - Centralized, easily configurable instantiation.
  - Factories may **pool** & re-use objects.

Factory UML Diagram



- ConcreteCreator (the factory) is responsible for creating “product” objects. It may select between several concrete sub-classes of product.
- NOTE: The abstract *Creator* super-class is optional here, and will be more relevant for Abstract Factories.

Factory – the basic pattern

```
// Factory responsible for creating Products:
public class ProductFactory {
     public Product createProduct(){
          Product result;
          if (…)       result = new ConcreteProduct1();
          else if(…)  result = new ConcreteProduct2();
          ... // More subclasses
          ... // Configure result (e.g: if it’s a socket, set timeout)
          return result;
     }
}

// Usage:
ProductFactory factory = new ProductFactory();
Product prod = factory.createProduct();
```

Select sub-class

Central configuration point

Simple Example

- Configuration only (no subclasses):

```
// Factory which creates & configures sockets:
public class CustomSocketFactory {

    public Socket createSocket() throws SocketException{
        Socket socket= new Socket();
        socket.setSoTimeout(myTimeout);
        socket.setReuseAddress(true);
        socket.setSendBufferSize(myBufferSize);
        return socket;
    }
}

// Usage:
CustomSocketFactory factory = new CustomSocketFactory();
Socket s= factory.createSocket();
```

  - Why didn’t we just define subclass (MySocket extends Socket) and configure it in the constructor ?

Bank Example

- A Bank application which needs to be configured for different storage techniques (Relational DB, flat file…).
- Assume storage is set once at system startup (or is rarely changed).

```
// The Product : BankStorage object
package bank;
public Interface BankStorage {
     Account readAccount(long accId) ;
     void saveAccount(Account acc);
}
```

Bank Example – Products

```
// concrete Products:
package bank;
public class SqlBankStorage implements BankStorage {
     protected SqlBankStorage(...) {
     }
     public Account readAccount(long accId){ ...}
     public void saveAccount(Account acc){ ... }
}
```

```
package bank;
public class FlatFileBankStorage implements BankStorage {
     protected FlatFileBankStorage(...) {
     }
     public Account readAccount(long accId){ ...}
     public void saveAccount(Account acc){ ... }
}
```

You may choose to force programmers to use your factory, by declaring protected/package friendly constructors.

However, this implies your Factory class (and probably your unit tests) must be in the same package as the products.

Bank Example – Factory

```
package bank;
import java.util.*;
import java.io.*;

public class BankStorageFactory {
    private static final String PROP_FILE = "bankstorage.properties";
    private Class concreteClass;
    public BankStorageFactory(){
     FileInputStream propInp =null;
     try {
          propInp=new FileInputStream(PROP_FILE);
          PropertyResourceBundle bundle=new PropertyResourceBundle(propInp);
          String classname= bundle.getString("storage.classname");
          concreteClass = Class.forName(classname);
     }catch(Exception ex){
          ...
     }finally{
          try { propInp.close(); } catch(Exception ex){}
     }
    }

    public BankStorage createBankStorage(){
        try{
           return (BankStorage) concreteClass.newInstance();
        }catch(Exception ex){…}
    }
}
```

We chose this over multiple “if”s.

More flexible, less efficient … (WHY ?)

Bank Example – Usage

```
# file  bankstorage.properties
storage.classname=bank.SqlBankStorage
```

```
// Usage:

BankStorageFactory factory = new BankStorageFactory ();
BankStorage storage = factory.createBankStorage();
Account acc = new Account ( “yossi levi”, 10000);
storage.save(acc);
```

Note

- The same approach of configuring class-names through property files can be seen in Java API’s such as:
  - **XMLReaderFactory.**
  - **Security Providers.**

Factory variations

- Factory may be a **static method** (rather than a dedicated factory class):

```
// Using java.util.Calendar
Calendar c = Calendar.getInstance();
```

Factory variations - Pool

- Factory may **pool & re-use objects.**
- Not considered a unique pattern, but worth mentioning since it:
- **May dramatically** improve performance, especially in garbage-collection applications.
- **Requires care:**
  - Returning objects to pool (no Smart Pointers in Java).
  - Pool mutex (keep in mind locks are expensive).
  - Pool size management.