Creational Patterns

Christopher Simpkins
chris.simpkins@gatech.edu
Creational Design Patterns

Abstracts the instantiation process.

- Encapsulate knowledge about which concrete classes the system uses.
- Hide how instances of these classes are created and put together.
Abstract Factory

**Intent:** Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

**Structure**

**Participants**

- **AbstractFactory** declares an interface for operations that create abstract product objects.
- **ConcreteFactory** implements the operations to create concrete product objects.
- **AbstractProduct** declares an interface for a type of product.
- **ConcreteProduct** defines a product object to be created by the corresponding concrete factory; implements the AbstractProduct interface.

![Diagram of Abstract Factory pattern](attachment:diagram.png)
Abstract Factory Example: `java.sql.Connection`

```java
public interface Connection ... {
    public Blob createBlob();
    public Statement createStatement();
    public PreparedStatement prepareStatement();
    ...
}
```

- The `Connection` interface has factory methods for a family of related classes.
- A particular `Connection` instance would return database-specific implementations of `Statement`, etc.

```java
String URL = "jdbc:oracle:thin:username/password@amrood:1521:EMP";
Connection conn = DriverManager.getConnection(URL);
```
**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.

**Structure**

**Participants**

- **Product** defines the interface of objects the factory creates.
- **ConcreteProduct** implements the Product interface.
- **Creator** declares the factory method, which returns an object of type Product.
- **ConcreteCreator** overrides factory method to return a ConcreteProduct object.
Say we have a solution domain object that represents a problem domain entity:

```java
public class Person {
    protected final int id;
    protected String name;

    public Person(int id, String name) {
        this.id = id;
        this.name = name;
    }

    public int getId() { return id; }
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }
}
```

How can we add persistence capability in an abstract way so that we can swap out different persistence implementations (database, etc.)?
Active Records are objects that know how to store and retrieve themselves from a data store. The simplest implementation of an ActiveRecord uses an abstract class:

```java
public abstract class ActivePerson extends Person {
    public ActivePerson(int id, String name) {
        super(id, name);
    }

    public abstract Person createNew(String name);

    public abstract Person findById(int id);

    public abstract void save();
}
```

ActivePerson extends Person with persistence capabilities. Now applications that use a particular data store can subclass ActivePerson and implement data store-specific versions of these persistence methods.
Here’s a subclass of `ActivePerson` that uses a `HashMap`:

```java
public class HashMapPerson extends ActivePerson {

    private static HashMap<Integer, Person> persons = new HashMap<>();
    private static int lastUsedId = 0;

    protected HashMapPerson(int id, String name) {
        super(id, name);
    }

    public Person createNew(String name) {
        Person newPerson = new HashMapPerson(lastUsedId++, name);
        persons.put(newId, newPerson);
        return newPerson;
    }

    public Person findById(int id) {
        return persons.get(id);
    }

    public void save() {
        // nothing to do - client has alias to object in HashMap
    }
}
```
Benefits of using **ActivePerson**:

- A **MySQLPerson** would implement MySQL-specific code that maps relational database representations of objects to their Java object counterparts.

- Application is coded to **ActivePerson** interface so versions of **ActivePerson** that use different data stores can be swapped out by changing only the client code that instantiates the **ActivePerson** objects.

- You could put all of your active record-instantiating code in an Abstract Factory or a registry (which could be a singleton) so there's only one place to make this change for all kinds of persisted objects.

There are other ways of doing this, but active records are easy to understand. All object-relational mapping and data store frameworks use these concepts.
Implementing Factories with Reflection

Reflection is an advanced Java programming technique often used to implement factories. Consider:

```
MyClass instance = new MyClass();
```

You can also do this with reflection:

```
MyClass instance = (MyClass) Class.forName("MyClass").newInstance();
```

You can store the string "MyClass" in a properties file, which could be changed without changing any code. Take a look at `greeter` for a simple but complete example of this technique.
Intent: Ensure a class only has one instance, and provide a global point of access to it.

Structure

Participants

- **Singleton** defines an Instance operation that lets clients access its unique instance.
  - Instance is a class operation (that is, a class method in Smalltalk and a static member function in C++, or static method in Java).
  - May be responsible for creating its own unique instance.
Singleton Example: \texttt{java.text.NumberFormat}

Remember \texttt{NumberFormat} from CS 1331?

\begin{verbatim}
public abstract class NumberFormat extends Format {
    protected NumberFormat() {}

    public final static NumberFormat getInstance() { ... }

    public static NumberFormat getInstance(Locale inLocale) { ... }

}
\end{verbatim}

- \texttt{NumberFormat} \texttt{instance} is instantiated once; this instance is shared by all users of \texttt{NumberFormat}
- \texttt{getInstance()} is also a factory method: creates a \texttt{NumberFormat} \texttt{instance} for a particular \texttt{Locale}
Implementing a Singleton

Three things to make a singleton:

- hide constructor,
- store singleton instance in some cache,
- provide public access to singleton instance.

A minimum example:

```java
public class MySingleton {
    protected static instance;

    // Hidden with private visibility - can only instantiate inside class
    private MySingleton() {} // instantiates

    public static MySingleton getInstance() {
        if (instance == null) {
            instance = new MySingleton();
        }
        return instance;
    }
}
```
Creational patterns address design goals

- loose coupling to specific classes
  - program to interfaces, factories return specific implementing classes
- designing for change
  - swapping out implementing classes is done in one place, the factory, and even this can be done with configuration files
  - little or no change to existing code

Many consider the `new` operator to be a code smell. `new` couples your code to a particular class. Factories remove that coupling.