CSE 223: Programming -2
06-Creational Design Patterns (II)

Prof. Dr. Khaled Nagi

Department of Computer and Systems Engineering,

Faculty of Engineering, Alexandria University, Egypt.

Agenda

- Prototype
- Singleton
- Object pool

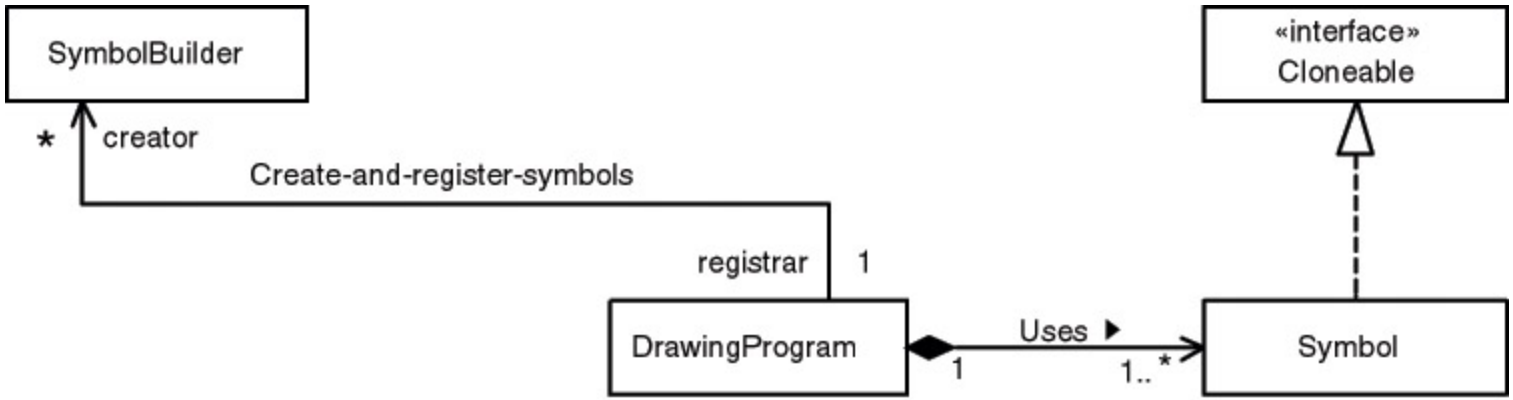
Prototype Pattern

Prototype Pattern

- Intention
  - The Prototype pattern allows an object to create customized objects without knowing their exact class or the details of how to create them.
  - It works by giving prototypical objects to an object that initiates the creation of objects.
  - The creation-initiating object then creates objects by asking the prototypical objects to make copies of themselves.
- Example
  - Suppose that you are writing a Computer-Assisted Design (CAD) program that allows its users to draw diagrams from a palette of symbols.
  - The program will have a core set of built-in symbols. However, people with different and specialized interests will use the program. These people will want additional symbols that are specific to their interests.
  - This presents the problem of how to provide these palettes of additional symbols. You can easily organize things so that all symbols, both core and additional, are descended from a common ancestor class.

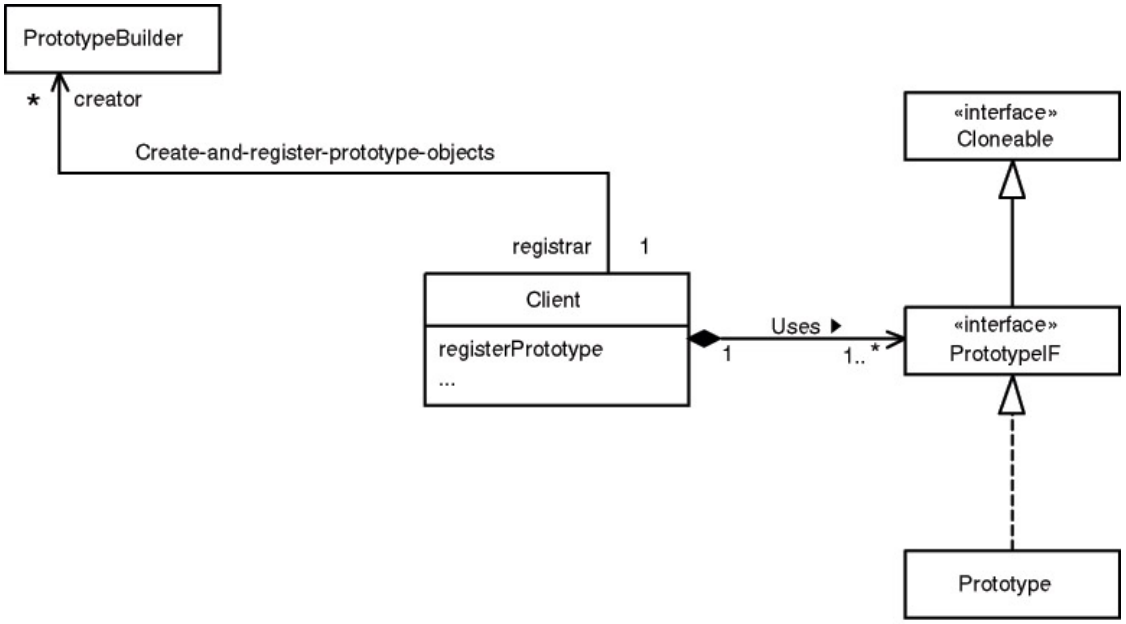
Prototype Pattern

- This will give the rest of your diagram-drawing program a consistent way of manipulating symbol objects. It does leave open the question of how the program will create these objects.
- Creating objects such as these is often more complicated than simply instantiating a class.



Prototype Pattern

- A system must be able to create objects without knowing their exact class, how they are created, or what data they represent.
- Classes to be instantiated are not known by the system until runtime, when they are acquired on the fly by techniques such as dynamic linkage.



Prototype Pattern

- **Client**. The client class represents the rest of the program for the purposes of the Prototype pattern. The client class needs to create objects that it knows little about. Client classes will have a method that can be called to add a prototypical object to a client object's collection (Ex: registerPrototype).
- **Prototype**. Classes in this role implement the **PrototypeIF** interface and are instantiated for the purpose of being cloned by the client.
- **PrototypeIF**. All prototype objects must implement the interface that is in this role. The client class interacts with prototype objects through this interface. Interfaces in this role should extend the Cloneable interface so that all objects that implement the interface can be cloned.
- **PrototypeBuilder**. This corresponds to any class instantiated to supply prototypical objects to the client object. Such classes should have a name that denotes the type of prototypical object that they build, such as **SymbolBuilder**.

Prototype Pattern

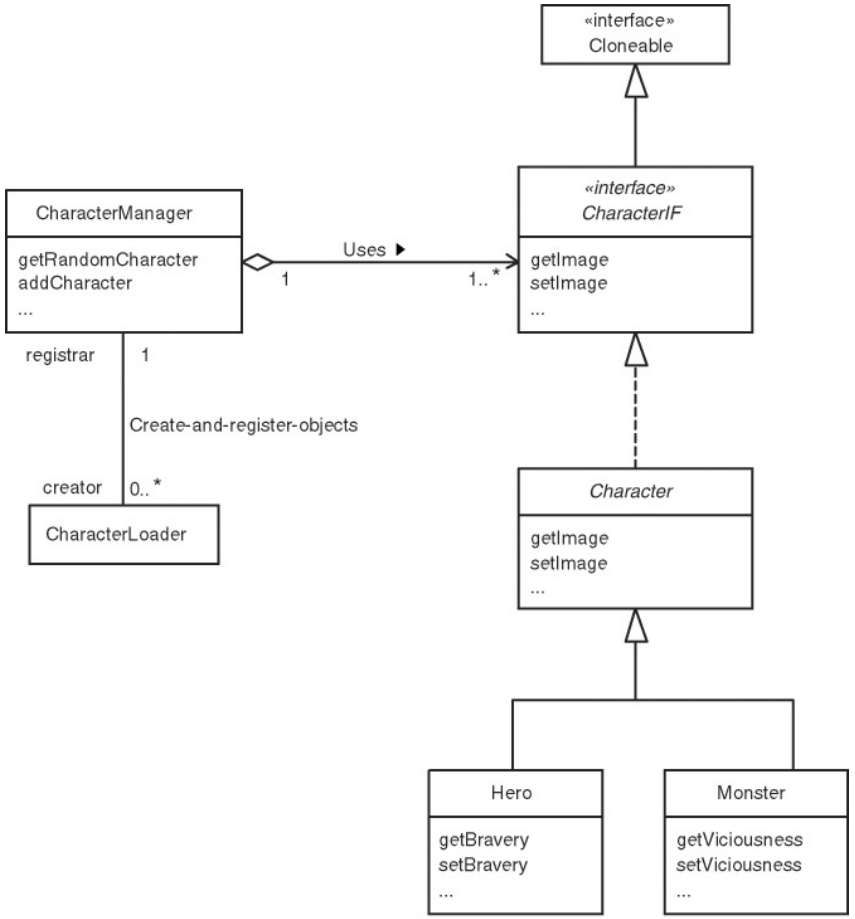
- How to implement the clone operation for the prototypical objects is another important implementation issue.
  - Reference Copy
  - Shallow Copy
  - Deep Copy

Prototype Pattern Consequences

- A program can dynamically add and remove prototypical objects at runtime
- The client object may also be able to create new kinds of prototypical objects. In the drawing program example we looked at previously, the client object could very reasonably allow the user to identify a sub-drawing and then turn the sub-drawing into a new symbol.
- The client class is independent of the exact class of the prototypical objects that it uses. Also, the client class does not need to know the details of how to build the prototypical objects.
- The PrototypeBuilder objects encapsulate the details of constructing prototypical objects.
- By insisting that prototypical objects implement an interface such as PrototypeIF, the Prototype pattern ensures that the prototypical objects provide a consistent set of methods for the client object to use.
- There is no need to organize prototypical objects into any sort of class hierarchy (like in Factory Method/Abstract Factory).
- A drawback of the Prototype pattern is the additional time spent writing PrototypeBuilder classes.

Prototype Pattern
Code Example

- Suppose that you are writing an interactive role-playing game.
- One of the expectations for this game is that people who play it will grow tired of interacting with the same characters and want to interact with new characters.
- For this reason, you are also developing an add-on to the game that consists of a few pre-generated characters and a program to generate additional characters.



Prototype Pattern
Code Example

```
public interface CharacterIF extends Cloneable {
    public String getName() ;
    public void setName(String name) ;
    public Image getImage() ;
    public void setImage(Image image) ;
    public int getStrength() ;
    public void setStrength(int strength) ;
}  // class CharacterIF
```

```
public abstract class Character implements CharacterIF {
    /* Override clone to make it public. */
    public Object clone() {
        try {
            return super.clone();
        }  catch (CloneNotSupportedException e) {
            // should never happen because this class
            // implements Cloneable.
            throw new InternalError();
        }  // try
    } // clone()
    public String getName() {  return name; }
    public void setName(String name) {  this.name = name; }
    public Image getImage() {  return image; }
    public void setImage(Image image) {this.image = image;}
} // class Character
```

Prototype Pattern
Code Example

```
public class CharacterManager {
    private Vector characters = new Vector();
/* Return a copy of random character from collection.*/
    Character getRandomCharacter() {
        int i = (int)(characters.size()*Math.random());
        Character c = (Character)characters.elementAt(i);
        return (Character)c.clone();
    }  // getRandomCharacter()

    /**
     * Add a prototypical object to the collection.
     */
    void addCharacter(Character character) {
        characters.addElement(character);
    } // addCharacter(Character)
...
} // class CharacterManager
```

Prototype Pattern
Code Example

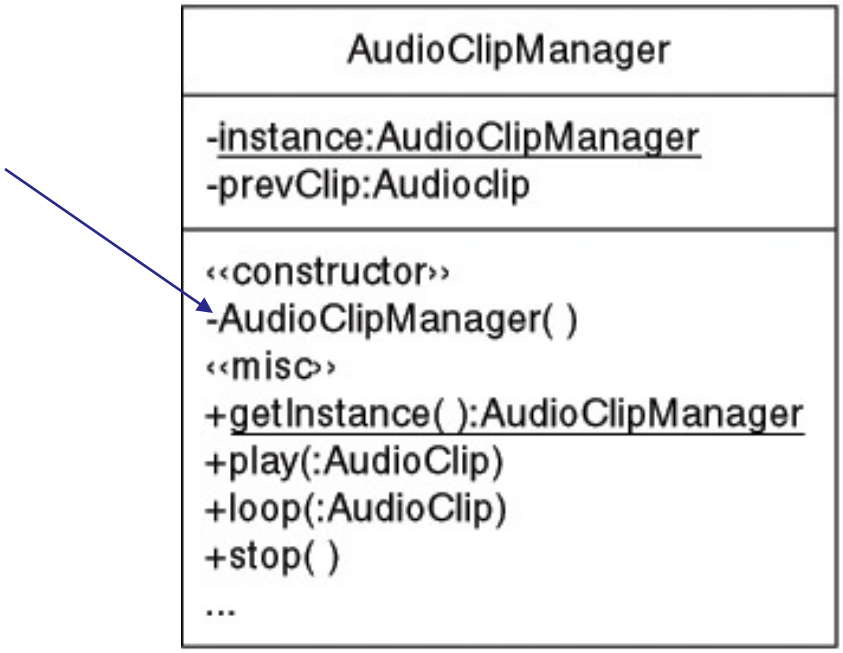
```
class CharacterLoader {
    private CharacterManager mgr;
   CharacterLoader(CharacterManager cm) {
        mgr = cm;
    }  // Constructor(CharacterManager)
    int loadCharacters(String fname) {
        int objectCount = 0;   // The number of objects loaded
        // If construction of InputStream fails, just return
        try {
            InputStream in;
            in = new FileInputStream(fname);
            in = new BufferedInputStream(in);
            ObjectInputStream oIn = new ObjectInputStream(in);
            while(true) {
                Object c = oIn.readObject();
                if (c instanceof Character) {
                    mgr.addCharacter((Character)c);
                }  // if
            } // while
        }  catch (Exception e) {
        } // try
        return objectCount;
    } // loadCharacters(String)
}  // class CharacterLoader
```

Singleton Pattern

Singleton Pattern

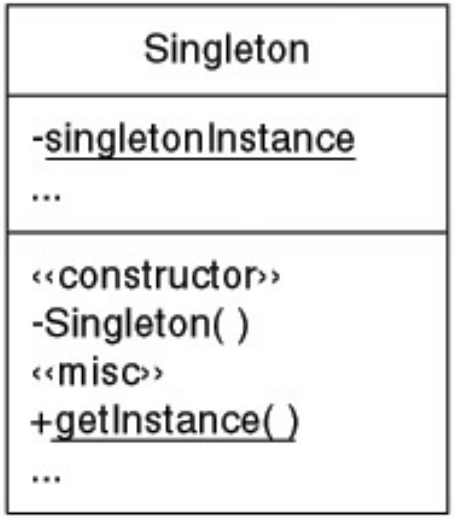
- Intention
  - The Singleton pattern ensures that only one instance of a class is created. All objects that use an instance of that class use the same instance.
- Example
  - Some classes should have exactly one instance. These classes usually involve the central management of a resource. The resource may be external, as is the case with an object that manages the reuse of database connections. The resource may be internal, such as an object that keeps an error count and other statistics for a compiler.
  - Suppose you need to write a class that an applet can use to ensure that no more than one audio clip is played at a time.
  - If an applet contains two pieces of code that independently play audio clips, then it is possible for both to be playing at the same time.
  - When two audio clips play at the same time, the results depend on the platform. The results may range from confusing, with users hearing both audio clips together, to terrible, with the platform's sound-producing mechanism unable to cope with playing two different audio clips at once.

Singleton Pattern



Singleton Pattern
Forces

- There must be at least one instance of a class.
- There should be no more than one instance of a class. This may be the case because you want to have only one source of some information. For example, you may want to have a single object that is responsible for generating a sequence of serial numbers.
- The one instance of a class must be accessible to all clients of that class.
- Beware of *Concurrent* **GetInstance** Calls!



Object Pool Pattern

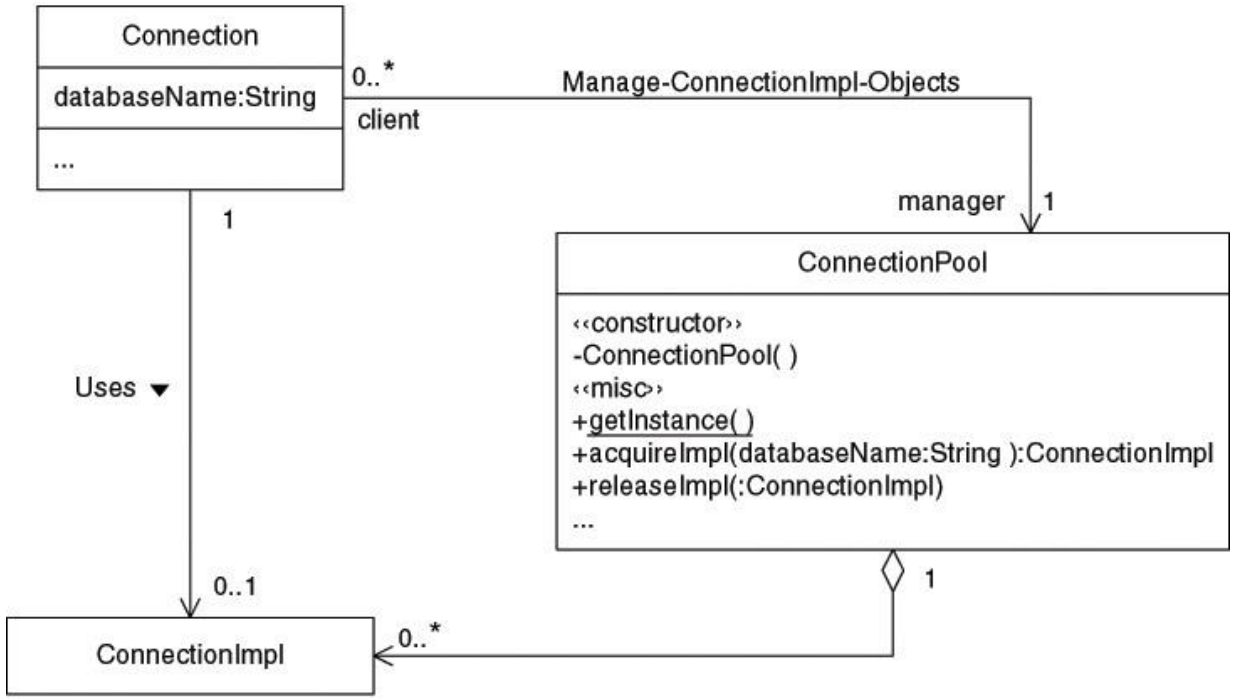
Object Pool Pattern

- Intention
  - Manage the reuse of objects when a type of object is expensive to create or only a limited number of a kind of object can be created.
- Example
  - Suppose you have been given the assignment of writing a class library to provide access to a proprietary database. Clients will send queries to the database through a network connection. The database server will receive queries through the network connection and return the results through the same connection.
  - A convenient way for programmers who will use the library to manage connections is for each part of a program that needs a connection to create its own connection. However, creating database connections that are not needed is bad for a few reasons:
    - It can take a few seconds to create each database connection.
    - The more connections there are to a database, the longer it takes to create new connections.
    - Each database connection uses a network connection. Some platforms limit the number of network

Object Pool Pattern

- Forces
  - A program may not create more than a limited number of instances of a particular class.
  - Creating instances of a particular class is sufficiently expensive that creating new instances of that class should be avoided.
  - A program can avoid creating some objects by reusing objects that it has finished with rather than letting them be garbage-collected.
  - The instances of a class are interchangeable. If you have multiple instances on hand, you can arbitrarily choose one to use for a purpose. It does not matter which one you choose.
  - Resources can be managed centrally by a single object or in a decentralized way by multiple objects. It is easier to achieve predictable results by managing resources centrally with a single object.
  - Some objects consume resources that are in short supply.

Object Pool Pattern



- ***Reusable***. Instances of classes in this role collaborate with other objects for a limited amount of time, then they are no longer needed for that collaboration.
- ***Client***. Instances of classes in this role use Reusable objects.
- ***ReusablePool***. Instances of classes in this role manage Reusable objects for use by Client objects.

Object Pool Pattern

- Implementation Issues
  - Ensuring a Maximum Number of Instances
  - Data Structure
  - Using Soft References
    - With its pros and cons
  - Limiting the Size of the Pool
- Consequences
  - avoids the creation of objects.
  - Keeping the logic to manage the creation and reuse of a class's instances in a separate class from the class whose instances are being managed results in a more cohesive design. It eliminates interactions between the implementation of a creation and reuse policy and the implementation of the managed class's functionality.