Design Patterns

The Monostate Design Pattern

The Issue:

The Singleton pattern is a fine design pattern for insuring that only one instance of class can ever be created and for insuring client objects all have access to the common Singleton object and it’s shared data, but it has some drawbacks:

- Destruction is undefined. There is no good way to destroy or decommission a singleton. If you add a decommission method that sets the one “instance” to “null”, other modules in the system may still be holding a reference to the singleton. Subsequent calls to the Singleton will cause another instance to be created, resulting in two concurrent instances. This problem is particularly acute in C++ where the instance can be explicitly destroyed, leading to possible dereferencing of a destroyed object.

- Not inherited. A class derived from a singleton is not a singleton. If it needs to be a singleton, the static function and variable need to be added to it.

- Efficiency. Each Singleton access invokes the “if” statement. For most of those calls, the “if” statement is useless.

- Non-transparent. Users of a singleton know that they are using a singleton because they must invoke the instance accessor method.

Monostate

Design Pattern

The Issue:

There are times when you want an object to behave like a Singleton but look like a “regular” object, thereby avoiding the previously described disadvantages of the Singleton pattern.

One of the main reasons one might avoid using the Singleton pattern is that it is difficult to subclass, and it is difficult to “convert” a Singleton class into a “regular” class. If there is a chance that you might need to convert a class from a single-object context to a multi-object context (i.e., a “regular” class) then the Singleton is not the best option.

For example, assume you needed to create a class that maintains and provides environmental data (temperature, humidity, etc.) to many client objects with an application you are developing. You would only want one consistent set of environmental data to be provided at any given time so you want to insure you only ever have a single “environment manager” object created. You also know that your application needs will require you to create subclasses of the “environment manager” to extend the set of data attributes and methods.

How can you create a class such that it displays single-object behavior, that singular nature is transparent to the users, and leave you the opportunity to employ polymorphic derivatives of the single object?

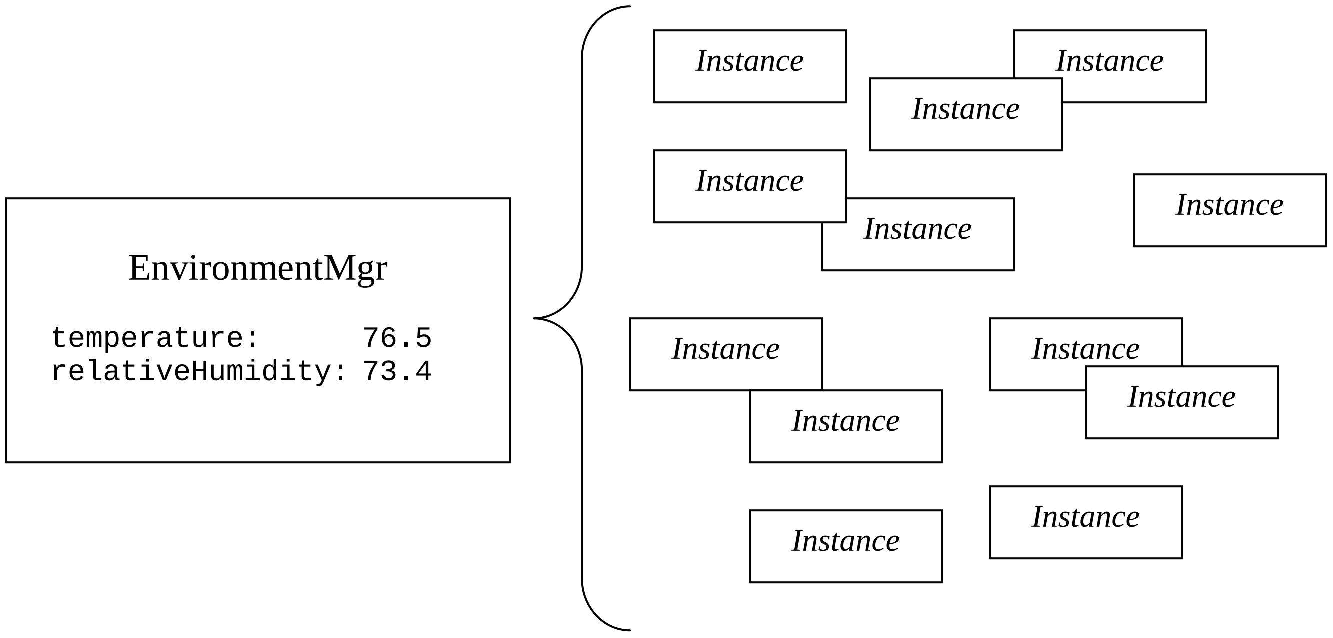
The Solution:

The Monostate Pattern

Using the Monostate design pattern we can avoid these problems.

The Monostate design pattern defines a “conceptual singleton”.
It is not a true singleton because multiple instances of the Monostate class can be created, but *all* data attributes of a Monostate class are declared as “static” so that all instances of the Monostate class share the same (static) data attribute values.

In this way, you can create one or one thousand instances of a Monostate class and all of those objects will share and see the same values for the Monostate’s data attributes.



The Solution:

Monostate Implementation

The following shows how out EnvironmentMgr class would look if implemented as a Monostate:

```
public class EnvironmentMgr
{
     private static double temperature;
     private static double humidity;

     public double getTemperature()
     {
          return temperature;
     }

     public void setTemperature(double temperatureIn)
     {
          temperature = temperatureIn;
     }

     public double getHumidity()
     {
          return humidity;
     }

     public void setHumidity(double humidityIn)
     {
          humidity = humidityIn;
     }
}
```

All the Monostate class' data attributes are declared as "static"

These accessor & modifiers will "get" and "set" the same static data values to any instance that invokes them

The Solution:

Monostate Usage

The following shows how out Monostate EnvironmentMgr class is used in client code:

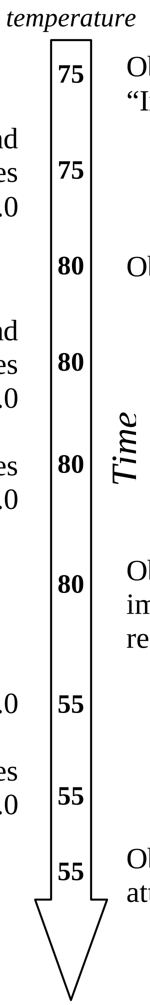
```
[…]

EnvironmentMgr em1 = new EnvironmentMgr();
em1.setTemperature(88.0);

[…]
EnvironmentMgr em2 = new EnvironmentMgr();
double x = em2.getTemperature(); // x is set to 88.0

[…]
EnvironmentMgr em3 = new EnvironmentMgr();
double s = em3.getTemperature(); // s is set to 88.0

[…]
EnvironmentMgr em4 = new EnvironmentMgr();
em4.setTemperature(72.0);

[…]
double a = em1.getTemperature(); // a is set to 72.0
double b = em2.getTemperature(); // b is set to 72.0
double c = em3.getTemperature(); // c is set to 72.0
double d = em4.getTemperature(); // d is set to 72.0

[…]
```

Though this illustrates how the Monostate behaves, in reality many other classes could be accessing and modifying the Monostate class' data attributes as well.

The Solution:

Monostate Usage

The following shows how a Monostate class behaves when used across multiple classes:



Object A creates a new Environment object (Instance 1) and sets the "Instance 1" temperature to 75.0

Object B creates a new Environment object (Instance 2) and immediately accesses the "Instance 2" temperature attribute - receives 75.0

Object A sets the "Instance 1" temperature to 80.0

Object B creates another new Environment object (Instance 3) and immediately accesses the "Instance 3" temperature attribute - receives 80.0

Object B accesses the "Instance 2" temperature attribute - receives 80.0

Object A creates a new Environment object (Instance 4) and immediately accesses the "Instance 4" temperature attribute - receives 80.0

Object B sets the "Instance 2" temperature to 55.0

Object B accesses the "Instance 3" temperature attribute - receives 55.0

Object A accesses the "Instance 1" and "Instance 4" temperature attribute - receives 55.0.

Monostate

Design Pattern

Summary:

- Users of a Monostate object do not behave differently than users of a regular object. The users do not need to know that the object is monostate.
- Subclasses of a Monostate class are Monostates. Indeed, all the derivatives of a Monostate are part of the same Monostate. They all share the same static variables.
- Since the methods of a Monostate are not static, they can be overridden in a derivative. Thus different derivatives can offer different behavior over the same set of static variables.
- Well defined creation and destruction. The variables of a Monostate, being static, have well defined creation and destruction times.