This article is derived from a chapter of The Principles, Patterns, and Practices of Agile Software Development, Robert C. Martin, Prentice Hall, 2002.

SINGLETON and MONOSTATE

"Infinite beatitude of existence! It is; and there is none else beside It."
-- The point. Flatland. Edwin A. Abbott

Usually there is a one to many relationship between classes and instances. You can create many instance of most classes. The instances are created when they are needed and disposed of when their usefulness ends. They come and go in a flow of memory allocations and deallocations.

But there are some classes that should have only one instance. That instance should appear to have come into existence when the program started, and should only be disposed of when the program ends. Such objects are sometimes the roots of the application. From the roots you can find your way to many other objects in the system. Sometimes they are factories which you can use to create the other objects in the system. Sometimes these objects are managers, responsible for keeping track of certain other objects and driving them through their paces.

Whatever these objects are, it is a severe logic failure if more than one of them is created. If more than one root is created then access to objects in the application may depend upon a chosen root. Programmers, not knowing that more than one root exists, may find themselves looking at a subset of the application objects without knowing it. If more than

one factory exists, clerical control over the created objects may be compromised. If more than one manager exists, activities that were intended to be serial may become concurrent.

It may seem that mechanisms to enforce the singularity of these objects is overkill. After all, when you initialize the application, you can simply create one of each and be done with it. In fact, this is usually the best course of action. Mechanism should be avoided when there is no immediate and significant need. However, we also want our code to communicate our intent. If the mechanism for enforcing singularity is trivial, the benefit of communication may outweigh the cost of the mechanism.

This chapter is about two patterns that enforce singularity. These patterns have very different cost/benefit trade-offs. In most contexts their cost is low enough to more than balance the benefit of their expressiveness.

SINGLETON[1]

SINGLETON is a very simple pattern. The test case in Listing 16-1 shows how it should work. The first test function shows that the `Singleton` instance is accessed through the public static method `Instance`. It also shows that if `Instance` is called multiple times, a reference to the exact same instance is returned each time. The second test case shows that the `Singleton` class has no public constructors, so there is no way for anyone to create an instance without using the `Instance` method.

Listing 16-1
Singleton Test Case

```
import junit.framework.*;
import java.lang.reflect.Constructor;

public class TestSimpleSingleton extends TestCase
{
  public TestSimpleSingleton(String name)
  {
    super(name);
  }

  public void testCreateSingleton()
  {
    Singleton s = Singleton.Instance();
    Singleton s2 = Singleton.Instance();
    assertSame(s, s2);
  }

  public void testNoPublicConstructors() throws Exception
  {
    Class singleton = Class.forName("Singleton");
    Constructor[] constructors = singleton.getConstructors();
```

1. [GOF95], p127

Listing 16-1
Singleton Test Case

```
    assertEquals("Singleton has public constructors.",
                 0, constructors.length);
  }
}
```

This test case is a specification for the SINGLETON pattern. It leads directly to the code shown in Listing 16-2. It should be clear, by inspecting this code, that there can never be more than one instance of the `Singleton` class within the scope of the static variable `Singleton.theInstance`.

Listing 16-2
Singleton Implementation

```
public class Singleton
{
  private static Singleton theInstance = null;
  private Singleton() {}

  public static Singleton Instance()
  {
    if (theInstance == null)
      theInstance = new Singleton();
    return theInstance;
  }
}
```

Benefits of the SINGLETON

- **Cross Platform:** Using appropriate middleware, (e.g. RMI) Singleton can be extended to work across many JVMs and many computers.
- **Applicable to any class.** You can change any class into a singleton simply by making its constructors private and by adding the appropriate static functions and variable.
- **Can be created through derivation.** Given a class, you can create a subclass that is a singleton.
- **Lazy Evaluation**. If the singleton is never used, it is never created.

Costs of the SINGLETON

- **Destruction is undefined.** There is no good way to destroy or decommission a singleton. If you add a `decommission` method that nulls out `theInstance`, other modules in the system may still be holding a reference to the singleton. Subsequent calls to `Instance` will cause another instance to be created, causing two concurrent instances to exist. This problem is particularly acute in C++ where the instance *can be 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 call to `Instance` 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` method.

SINGLETON in Action

Assume that we have a web based system that allows users to log in to secure areas of a web server. Such a system will have a database containing user names, passwords, and other user attributes. Assume further that the database is accessed through a third party API. We could access the database directly in every module that needed to read and write a user. However this would scatter usage of the third party API throughout the code, and would leave us no place to enforce access or structure conventions.

A better solution is to use the FACADE pattern and create a `UserDatabase` class that provides methods for reading and writing `User` objects. These methods access the third party API of the database, translating between `User` objects and the tables and rows of the database. Within the `UserDatabase` we can enforce conventions of structure and access. For example, we can guarantee that no `User` record gets written unless it has a non-blank `username`. Or we can serialize access to a `User` record, making sure that two modules cannot simultaneously read and write it.

The code in Listing 16-3 and Listing 16-4 show a SINGLETON solution. The SINGLETON class is named `UserDatabaseSource`. It implements the `UserDatabase` interface. Notice that the static `instance()` method does not have the traditional `if` statement to protect against multiple creations. Instead, it takes advantage of the Java initialization facility.

Listing 16-3
`UserDatabase Interface`

```
public interface UserDatabase
{
  User readUser(String userName);
  void writeUser(User user);
}
```

Listing 16-4
`UserDatabaseSource Singleton`

```
public class UserDatabaseSource implements UserDatabase
{
  private static UserDatabase theInstance =
    new UserDatabaseSource();

  public static UserDatabase instance()
  {
    return theInstance;
  }

  private UserDatabaseSource()
  {
```

Listing 16-4 (Continued)
UserDatabaseSource Singleton

```
  }

  public User readUser(String userName)
  {
    // Some Implementation
    return null;  // just to make it compile.S
  }

  public void writeUser(User user)
  {
    // Some Implementation
  }
}
```

This is an extremely common use of the SINGLETON pattern. It assures that all database access will be through a single instance of `UserDatabaseSource`. This makes it easy to put checks, counters and locks in `UserDatabaseSource` that enforce the access and structure conventions mentioned earlier.

MONOSTATE

The MONOSTATE pattern is another way to achieve singularity. It works through a completely different mechanism. We can see how that mechanism works by studying the Monostate test case in Listing 16-5.

The first test function simply describes an object whose `x` variable can be set and retrieved. But the second test case shows that two instances of the same class behave *as though they were one*. If you set the `x` variable on one instance to a particular value, you can retrieve that value by getting the `x` variable of a different instance. It's as though the two instances are just different names for the same object.

Listing 16-5
Monostate Test Case

```
import junit.framework.*;

public class TestMonostate extends TestCase
{
  public TestMonostate(String name)
  {
    super(name);
  }

  public void testInstance()
  {
    Monostate m = new Monostate();
    for (int x = 0; x<10; x++)
    {
      m.setX(x);
      assertEquals(x, m.getX());
```

Listing 16-5
Monostate Test Case

```
    }
  }

  public void testInstancesBehaveAsOne()
  {
    Monostate m1 = new Monostate();
    Monostate m2 = new Monostate();

    for (int x = 0; x<10; x++)
    {
      m1.setX(x);
      assertEquals(x, m2.getX());
    }
  }
}
```

If we were to plug the `Singleton` class into this test case, and replace all the `new Monostate` statements with calls to `Singleton.Instance`, the test case should still pass. So this test case describes the *behavior* of `Singleton` without imposing the constraint of a single instance!

How can two instances behave as though they were a single object? Quite simply it means that the two objects must share the same variables. This is easily achieved by making all the variables static. Listing 16-6 shows the implementation of `Monostate` that passes the above test case. Note that the `itsX` variable is static. Note also that *none of the methods are static*. This is important as we'll see later.

Listing 16-6
Monostate Implementation

```
public class Monostate
{
  private static int itsX = 0;
  public Monostate() {}

  public void setX(int x)
  {
    itsX = x;
  }

  public int getX()
  {
    return itsX;
  }
}
```

I find this to be a delightfully twisted pattern. No matter how many instances of `Monostate` you create, they all behave as though they were a *single object*. You can even destroy or decommission all the current instances without losing the data.

Note that the difference between the two patterns is one of behavior vs. structure. The SINGLETON pattern enforces the structure of singularity. It prevents any more than one instance from being created. Whereas MONOSTATE enforces the *behavior* of singularity

without imposing structural constraints. To underscore this difference consider that the Monostate test case is valid for the `Singleton` class, but the singleton test case is not even close to being valid for the `Monostate` class.

Benefits of MONOSTATE

- **Transparency**. Users of a monostate do not behave differently than users of a regular object. The users do not need to know that the object is monostate.
- **Derivability**. Derivatives of a monostate are monostates. Indeed, all the derivatives of a monostate are part of the *same* monostate. They all share the same static variables.
- **Polymorphism**. 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

Costs of MONOSTATE

- **No conversion**. A non monostate class cannot be converted into a monostate class through derivation.
- **Efficiency**. A monostate may go through many creations and destructions because it is a real object. These operations are often costly.
- **Presence**. The variables of a monostate take up space, even if the monostate is never used.
- **Platform local.** You can't make a monostate work across several JVM instances or across several platforms.

MONOSTATE in Action

Consider implementing the simple finite state machine for a subway turnstile shown in Figure 16-1. The turnstile begins its life in the `Locked` state. If a coin is deposited, it transitions to the `Unlocked` state, and unlocks the gate, resets any alarm state that might be present, and deposits the coin in its collection bin. If a user passes through the gate at this point, the turnstile transitions back to the `Locked` state and locks the gate.

There are two abnormal conditions. If the user deposits two or more coins before passing through the gate, the coins will be refunded, and the gate will remain unlocked. If the user passes through without paying, then an alarm will sound and the gate will remain locked.

The test program that describes this operation is shown in Listing 16-7. Note that the test methods assume that the `Turnstyle` is a monostate. It expects to be able to send

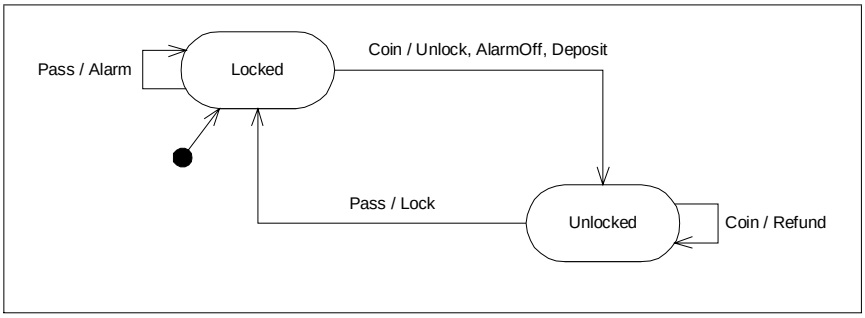

Figure 16-1
Subway Turnstile Finite State Machine

events and gather queries from different instances. This makes sense if there will never be more than one instance of the `Turnstyle`.

Listing 16-7
TestTurnstyle

```
import junit.framework.*;

public class TestTurnstyle extends TestCase
{
  public TestTurnstyle(String name)
  {
    super(name);
  }

  public void setUp()
  {
    Turnstyle t = new Turnstyle();
    t.reset();
  }

  public void testInit()
  {
    Turnstyle t = new Turnstyle();
    assert(t.locked());
    assert(!t.alarm());
  }

  public void testCoin()
  {
    Turnstyle t = new Turnstyle();
    t.coin();
    Turnstyle t1 = new Turnstyle();
    assert(!t1.locked());
    assert(!t1.alarm());
    assertEquals(1, t1.coins());
  }

  public void testCoinAndPass()
  {
    Turnstyle t = new Turnstyle();
```

Listing 16-7 (Continued)
TestTurnstyle

```
      t.coin();
      t.pass();

      Turnstyle t1 = new Turnstyle();
      assert(t1.locked());
      assert(!t1.alarm());
      assertEquals("coins", 1, t1.coins());
    }

    public void testTwoCoins()
    {
      Turnstyle t = new Turnstyle();
      t.coin();
      t.coin();

      Turnstyle t1 = new Turnstyle();
      assert("unlocked", !t1.locked());
      assertEquals("coins",1, t1.coins());
      assertEquals("refunds", 1, t1.refunds());
      assert(!t1.alarm());
    }

    public void testPass()
    {
      Turnstyle t = new Turnstyle();
      t.pass();
      Turnstyle t1 = new Turnstyle();
      assert("alarm", t1.alarm());
      assert("locked", t1.locked());
    }

    public void testCancelAlarm()
    {
      Turnstyle t = new Turnstyle();
      t.pass();
      t.coin();
      Turnstyle t1 = new Turnstyle();
      assert("alarm", !t1.alarm());
      assert("locked", !t1.locked());
      assertEquals("coin", 1, t1.coins());
      assertEquals("refund", 0, t1.refunds());
    }

    public void testTwoOperations()
    {
      Turnstyle t = new Turnstyle();
      t.coin();
      t.pass();
      t.coin();
      assert("unlocked", !t.locked());
      assertEquals("coins", 2, t.coins());
      t.pass();
      assert("locked", t.locked());
    }
  }
```

The implementation of the monostate `Turnstyle` is in Listing 16-8. The base `Turnstyle` class delegates the two event functions, `coin` and `pass`, to two derivatives of `Turnstyle`, `Locked` and `Unlocked`, that represent the states of the finite state machine.

Listing 16-8
`Turnstyle`

```
public class Turnstyle
{
  private static boolean isLocked = true;
  private static boolean isAlarming = false;
  private static int itsCoins = 0;
  private static int itsRefunds = 0;
  protected final static Turnstyle LOCKED = new Locked();
  protected final static Turnstyle UNLOCKED = new Unlocked();
  protected static Turnstyle itsState = LOCKED;

  public void reset()
  {
    lock(true);
    alarm(false);
    itsCoins = 0;
    itsRefunds = 0;
    itsState = LOCKED;
  }

  public boolean locked()
  {
    return isLocked;
  }

  public boolean alarm()
  {
    return isAlarming;
  }

  public void coin()
  {
    itsState.coin();
  }

  public void pass()
  {
    itsState.pass();
  }

  protected void lock(boolean shouldLock)
  {
    isLocked = shouldLock;
  }

  protected void alarm(boolean shouldAlarm)
  {
    isAlarming = shouldAlarm;
  }

  public int coins()
  {
```

Listing 16-8 (Continued)
Turnstyle

```
    return itsCoins;
  }

  public int refunds()
  {
    return itsRefunds;
  }

  public void deposit()
  {
    itsCoins++;
  }

  public void refund()
  {
    itsRefunds++;
  }
}

class Locked extends Turnstyle
{
  public void coin()
  {
    itsState = UNLOCKED;
    lock(false);
    alarm(false);
    deposit();
  }

  public void pass()
  {
    alarm(true);
  }
}

class Unlocked extends Turnstyle
{
  public void coin()
  {
    refund();
  }

  public void pass()
  {
    lock(true);
    itsState = LOCKED;
  }
}
```

This example shows some of the useful features of the MONOSTATE pattern. It takes advantage of the ability for monostate derivatives to be polymorphic, and that monostate derivatives are themselves monostates. This example also shows how difficult it can sometimes be to turn a monostate into a non-monostate. The structure of this solution strongly depends upon the monostate nature of `Turnstyle`. If we needed to control more

than one turnstile with this finite state machine, the code would require some significant refactoring.

Perhaps you are concerned about the unconventional use of inheritance in this example. Having `Unlocked` and `Locked` derived from `Turnstyle` seems a violation of normal OO principles. However, since Turnstile is a Monostate, there are no separate instances of it. Thus, `Unlocked` and `Locked` aren't really separate objects. Instead they are part of the `Turnstyle` abstraction. `Unlocked` and `Locked` have access to the same variables and methods that `Turnstyle` does.

Conclusion

It is often necessary to enforce that a particular object have only a single instantiation. This chapter has shown two very different techniques. SINGLETON makes use of private constructors, a static variable, and a static function to control and limit instantiation. MONOSTATE simply makes all variables of the object static.

SINGLETON is best used when you have an existing class that you want to constrain through derivation, and you don't mind that everyone will have to call the `instance()` method to gain access. `Monostate` is best used when you want the singular nature of the class to be transparent to the users, or when you want to employ polymorphic derivatives of the single object.

Bibliography

[GOF95]: Design Patterns, Gamma, et. al., Addison Wesley, 1995

[PLOPD3]: Pattern Languages of Program Design 3, Robert C. Martin, et. al., Addison Wesley, 1998.