Intent

Ensure a class only has one instance, and provide a global point of access to it.

Motivation

It's important for some classes to have exactly one instance. Although there can be many printers in a system, there should be only one printer spooler. There should be only one file system and one window manager. A digital filter will have one A/D converter. An accounting system will be dedicated to serving one company.

How do we ensure that a class has only one instance and that the instance is easily accessible? A global variable makes an object accessible, but it doesn't keep you from instantiating multiple objects.

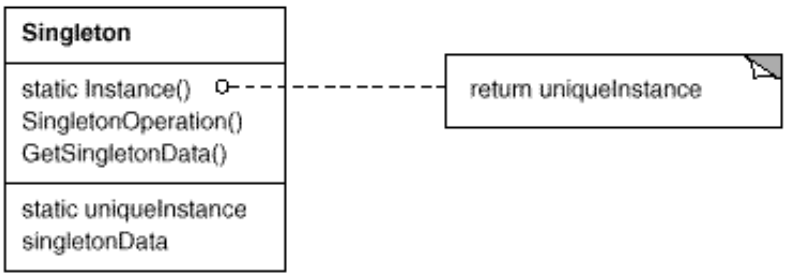
A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. This is the Singleton pattern.

Applicability

Use the Singleton pattern when

- there must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- when the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.

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++).
  - may be responsible for creating its own unique instance.

Collaborations

- Clients access a Singleton instance solely through Singleton's Instance operation.

Consequences

The Singleton pattern has several benefits:

1. *Controlled access to sole instance*. Because the Singleton class encapsulates its sole instance, it can have strict control over how and when clients access it.

2. *Reduced name space*. The Singleton pattern is an improvement over global variables. It avoids polluting the name space with global variables that store sole instances.

3. *Permits refinement of operations and representation*. The Singleton class may be subclassed, and it's easy to configure an application with an instance of this extended class. You can configure the application with an instance of the class you need at run-time.

4. *Permits a variable number of instances*. The pattern makes it easy to change your mind and allow more than one instance of the Singleton class. Moreover, you can use the same approach to control the number of instances that the application uses. Only the operation that grants access to the Singleton instance needs to change.

5. *More flexible than class operations*. Another way to package a singleton's functionality is to use class operations (that is, static member functions in C++ or class methods in Smalltalk). But both of these language techniques make it hard to change a design to allow more than one instance of a class. Moreover, static member functions in C++ are never virtual, so subclasses can't override them polymorphically.

Implementation

Here are implementation issues to consider when using the Singleton pattern:

1. *Ensuring a unique instance*. The Singleton pattern makes the sole instance a normal instance of a class, but that class is written so that only one instance can ever be created. A common way to do this is to hide the operation that creates the instance behind a class operation (that is, either a static member function or a class method) that guarantees only one instance is created. This operation has access to the variable that holds the unique instance, and it ensures the variable is initialized with the unique instance before returning its value. This approach ensures that a singleton is created and initialized before its first use.

   You can define the class operation in C++ with a static member function `Instance` of the `Singleton` class. `Singleton` also defines a static member variable `_instance` that contains a pointer to its unique instance.

   The `Singleton` class is declared as

```
class Singleton {
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton* _instance;
};
```

   The corresponding implementation is

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance () {
    if (_instance == 0) {
        _instance = new Singleton;
    }
    return _instance;
}
```

Clients access the singleton exclusively through the `Instance` member function. The variable `_instance` is initialized to 0, and the static member function `Instance` returns its value, initializing it with the unique instance if it is 0. `Instance` uses lazy initialization; the value it returns isn't created and stored until it's first accessed.

Notice that the constructor is protected. A client that tries to instantiate `Singleton` directly will get an error at compile-time. This ensures that only one instance can ever get created.

Moreover, since the `_instance` is a pointer to a Singleton object, the `Instance` member function can assign a pointer to a subclass of Singleton to this variable. We'll give an example of this in the Sample Code.

There's another thing to note about the C++ implementation. It isn't enough to define the singleton as a global or static object and then rely on automatic initialization. There are three reasons for this:

1. We can't guarantee that only one instance of a static object will ever be declared.
2. We might not have enough information to instantiate every singleton at static initialization time. A singleton might require values that are computed later in the program's execution.
3. C++ doesn't define the order in which constructors for global objects are called across translation units [ES90]. This means that no dependencies can exist between singletons; if any do, then errors are inevitable.

An added (albeit small) liability of the global/static object approach is that it forces all singletons to be created whether they are used or not. Using a static member function avoids all of these problems.

In Smalltalk, the function that returns the unique instance is implemented as a class method on the Singleton class. To ensure that only one instance is created, override the `new` operation. The resulting Singleton class might have the following two class methods, where `SoleInstance` is a class variable that is not used anywhere else:

```
new
    self error: 'cannot create new object'

default
    SoleInstance isNil ifTrue: [SoleInstance := super new].
    ^ SoleInstance
```

2. *Subclassing the Singleton class*. The main issue is not so much defining the subclass but installing its unique instance so that clients will be able to use it. In essence, the variable that refers to the singleton instance must get initialized with an instance of the subclass. The simplest technique is to determine which singleton you want to use in the Singleton's `Instance` operation. An example in the Sample Code shows how to implement this technique with environment variables.

   Another way to choose the subclass of Singleton is to take the implementation of `Instance` out of the parent class (e.g., `MazeFactory`) and put it in the subclass. That lets a C++ programmer

decide the class of singleton at link-time (e.g., by linking in an object file containing a different implementation) but keeps it hidden from the clients of the singleton.

The link approach fixes the choice of singleton class at link-time, which makes it hard to choose the singleton class at run-time. Using conditional statements to determine the subclass is more flexible, but it hard-wires the set of possible Singleton classes. Neither approach is flexible enough in all cases.

A more flexible approach uses a **registry of singletons**. Instead of having `Instance` define the set of possible Singleton classes, the Singleton classes can register their singleton instance by name in a well-known registry.

The registry maps between string names and singletons. When `Instance` needs a singleton, it consults the registry, asking for the singleton by name. The registry looks up the corresponding singleton (if it exists) and returns it. This approach frees `Instance` from knowing all possible Singleton classes or instances. All it requires is a common interface for all Singleton classes that includes operations for the registry:

```
class Singleton {
public:
    static void Register(const char* name, Singleton*);
    static Singleton* Instance();
protected:
    static Singleton* Lookup(const char* name);
private:
    static Singleton* _instance;
    static List<NameSingletonPair>* _registry;
};
```

`Register` registers the Singleton instance under the given name. To keep the registry simple, we'll have it store a list of `NameSingletonPair` objects. Each `NameSingletonPair` maps a name to a singleton. The `Lookup` operation finds a singleton given its name. We'll assume that an environment variable specifies the name of the singleton desired.

```
Singleton* Singleton::Instance () {
    if (_instance == 0) {
        const char* singletonName = getenv("SINGLETON");
        // user or environment supplies this at startup

        _instance = Lookup(singletonName);
        // Lookup returns 0 if there's no such singleton
    }
    return _instance;
}
```

Where do Singleton classes register themselves? One possibility is in their constructor. For example, a `MySingleton` subclass could do the following:

```
MySingleton::MySingleton() {
    // ...
    Singleton::Register("MySingleton", this);
}
```

Of course, the constructor won't get called unless someone instantiates the class, which echoes the problem the Singleton pattern is trying to solve! We can get around this problem in C++ by defining a static instance of `MySingleton`. For example, we can define

```
static MySingleton theSingleton;
```

in the file that contains `MySingleton`'s implementation.

No longer is the Singleton class responsible for creating the singleton. Instead, its primary responsibility is to make the singleton object of choice accessible in the system. The static object approach still has a potential drawback—namely that instances of all possible Singleton subclasses must be created, or else they won't get registered.

Sample Code

Suppose we define a `MazeFactory` class for building mazes as described on page 92. `MazeFactory` defines an interface for building different parts of a maze. Subclasses can redefine the operations to return instances of specialized product classes, like `BombedWall` objects instead of plain `Wall` objects.

What's relevant here is that the Maze application needs only one instance of a maze factory, and that instance should be available to code that builds any part of the maze. This is where the Singleton pattern comes in. By making the `MazeFactory` a singleton, we make the maze object globally accessible without resorting to global variables.

For simplicity, let's assume we'll never subclass `MazeFactory`. (We'll consider the alternative in a moment.) We make it a Singleton class in C++ by adding a static `Instance` operation and a static `_instance` member to hold the one and only instance. We must also protect the constructor to prevent accidental instantiation, which might lead to more than one instance.

```
class MazeFactory {
public:
    static MazeFactory* Instance();

    // existing interface goes here
protected:
    MazeFactory();
private:
    static MazeFactory* _instance;
};
```

The corresponding implementation is

```
MazeFactory* MazeFactory::_instance = 0;

MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        _instance = new MazeFactory;
    }
    return _instance;
}
```

Now let's consider what happens when there are subclasses of `MazeFactory`, and the application must decide which one to use. We'll select the kind of maze through an environment variable and add code that instantiates the proper `MazeFactory` subclass based on the environment variable's value. The `Instance` operation is a good place to put this code, because it already instantiates `MazeFactory`:

```
MazeFactory* MazeFactory::Instance () {
    if (_instance == 0) {
        const char* mazeStyle = getenv("MAZESTYLE");

        if (strcmp(mazeStyle, "bombed") == 0) {
            _instance = new BombedMazeFactory;

        } else if (strcmp(mazeStyle, "enchanted") == 0) {
            _instance = new EnchantedMazeFactory;
```

```
            // ... other possible subclasses

            } else {        // default
                _instance = new MazeFactory;
            }
        }
        return _instance;
    }
```

Note that `Instance` must be modified whenever you define a new subclass of `MazeFactory`. That might not be a problem in this application, but it might be for abstract factories defined in a framework.

A possible solution would be to use the registry approach described in the Implementation section. Dynamic linking could be useful here as well—it would keep the application from having to load all the subclasses that are not used.

Known Uses

An example of the Singleton pattern in Smalltalk-80 [Par90] is the set of changes to the code, which is `ChangeSet current`. A more subtle example is the relationship between classes and their metaclasses. A metaclass is the class of a class, and each metaclass has one instance. Metaclasses do not have names (except indirectly through their sole instance), but they keep track of their sole instance and will not normally create another.

The InterViews user interface toolkit [LCI+92] uses the Singleton pattern to access the unique instance of its Session and WidgetKit classes, among others. Session defines the application's main event dispatch loop, stores the user's database of stylistic preferences, and manages connections to one or more physical displays. WidgetKit is an Abstract Factory (87) for defining the look and feel of user interface widgets. The `WidgetKit::instance()` operation determines the particular WidgetKit subclass that's instantiated based on an environment variable that Session defines. A similar operation on Session determines whether monochrome or color displays are supported and configures the singleton Session instance accordingly.

Related Patterns

Many patterns can be implemented using the Singleton pattern. See Abstract Factory (87), Builder (97), and Prototype (117).

▲
▶ Discussion of Creational Patterns
◀ Prototype