Department of Computer Science

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Tackling Design Patterns
Chapter 28: Singleton Design Pattern

Contents

28.1 Introduction

In this lecture you will learn about the Singleton Design Pattern. When looking at it in terms of its class diagram and design, it is the simplest pattern. Its class diagram has only one class. However, it turns out to be quite complicated when considering its implementation and the consequences of the various ways in which it can be implemented. The singleton is probably the most debated pattern of all the classical design patterns.

The Singleton pattern is applicable in situations where there is a need to have only one instance of a class instantiated during the lifetime of the execution of a system. Sometimes because it makes logical sense to have a single instance of an object and sometimes because it might be dangerous to have more than one instance alive at the same time. Dangerous situations arise when resources are shared. For example if a system would have two instances of its file system running at the same time it can easily create a state where the two instances destroy the operations of one another resulting in unpredictable system state. The Singleton Design Pattern provide the means to write code that is guaranteed not to instantiate more than one instance of a specific class.

The Singleton pattern should be used with caution. We agree with the following remark by [2]

> *If you use a large number of Singletons in your application, you should take a hard look at your design. Singletons are meant to be used sparingly.*

28.2 Programming Preliminaries

28.2.1 Information hiding

The Singleton Design Pattern applies information hiding to ensure that its constructors are used in a strictly controlled manner. We assume that the reader is familiar with the concept of information hiding and the visibility of members of a class (i.e. public, protected and private) and their respective consequences. These are also discussed in more detail in Section **??**.

28.2.2 Destructors and virtual destructors

In C++ a destructor is generally used to deallocate memory and do some other cleanup for a class object and its class members whenever an object is destroyed. Destructors are distinguished by the tilde, the $\sim$ that appears in front of the destructor name. When no destructor is specified, the compiler usually creates a default destructor. In our experience we have learnt that relying on the default destructor is sometimes disastrous. Consequently we have specified in our coding standards that every class definition of a class with dynamic instance variables should explicitly containing a default constructor, copy constructor, assignment operator and destructor.

In a situation where a derived class is destroyed through a pointer to its base class, the base class destructor will be executed. This destructor will be unaware of any additional memory allocated by the derived class, leading to a memory leak where the object is

destroyed but all extra memory that was allocated beyond the base object memory, will not be freed. To remedy this, the base class destructor should be virtual. If it is virtual the destructor for any class that derives from base will be called before the base class destructor. For this reason [8] specify that destructors of polymorphic base classes should be declared virtual. An implication of this important design principle of class design is that if a class has one or more virtual functions, then that class should have a virtual destructor.

28.2.3 Static variables and class members

In C++ the `static` keyword is used to specify that a local variable in a function, an instance variable of a class or a member function of a class is static.

The use of static keyword when declaring a local variable inside a function means that once the variable has been initialised, it remains in memory until the program terminates. Although the variable is local to the function and cannot be accesses from outside the function it maintains its value in subsequent calls to the function. For instance, you can use a static variable to record the number of times a function has been called simply by including the following code in the body of the function:

```
static int count = 0;
count++;
```

The memory for local variables of a function are allocated each time the function is called and released when the function goes out of scope. When declaring a local variable of a function static, however, the memory for it will be allocated once and will not be released when the function goes out of scope. Thus, the line `static int count = 0;'` will only be executed once. Whenever the function is subsequently called, `count` will contain the value that was assigned to it during the previous call the the function.

Variables declared inside a class are used to specify the state of an instance of the class. Thus each instance of a class, allocates memory for all its variables and each of these variables can have a different value in the different instances of the class. On the contrary, a static instance variable has the same value over all instances of the class. Its memory is allocated only once and is shared by all the instances. It acts like a global variable that is visible only to instances of the class that defines it. In fact its memory is allocated even if the class is never instantiated.

Methods of class a may be declared static. Static methods can be accessed without instantiating the class. The following example code defines a utility class in Figure 1 called `FinanceTools`. It contains a number of static member functions.

```
class FinanceTools
{
    public:
        static double calculateSimpleInterest
            (double principal, float rate, int term);
        static double calculateCompoundInterest
            (double principal, float rate, int term);
        static double calculatePayment
            (double loan, float rate, int term);
};
```

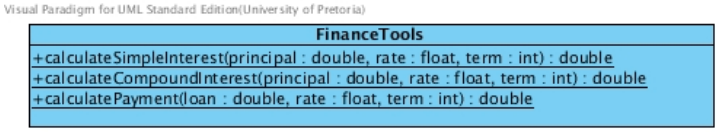

Figure 1: The FinanceTools class

Note that the member functions are underlined in the class diagram. It shows that they are declared static. Because all the member functions of this class are static and the class has no instance variables, it is not required that the class be instantiated. The member functions can be used without an instance of the class. Thus there is no need to implement constructors or a destructor for this class.

To use the static member functions of a class one can simply refer to them through the use of a class name and the scope operator, `::`. For example the following code uses the `calculatePayment` function without instantiating the class:

```
double installment;
installment = FinanceTools :: calculatePayment(220000, 23.7, 40);
```

28.2.4 Lazy initialisation

Sometimes classes that are implemented as part of a system are not always used while an application executes. Often software applications may contain functionality that most users of the program hardly use. An example may be the mail-merge functionality in a word processor that may not be needed when the user is using the word processor to write a scientific report. Another example is the math equation editor that may not be needed if the user is someone who is typing the minutes of a meeting.

To save resources the concept of lazy initialisation may be applied. This means that classes are not instantiated unless they are required. This is achieved by instantiating the classes only when their functionality is requested by the user.

28.3 Singleton Design Pattern

28.3.1 Identification

Name	**Classification**	**Strategy**
Singleton	Creational	Delegation
Intent		
Ensure a class only has one instance, and provide a global point of access to it. ([4]:127)		

28.3.2 Structure

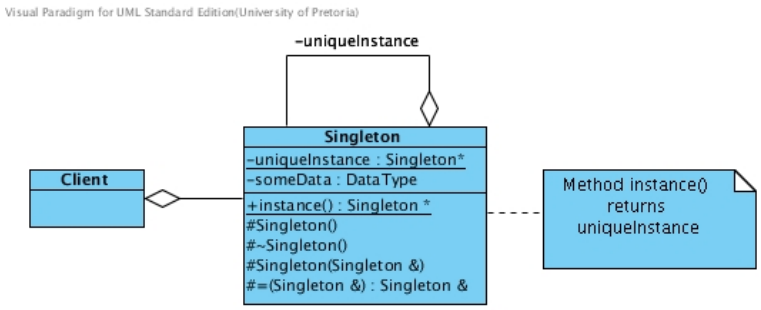

Figure 2: The structure of the Singleton Design Pattern

28.3.3 Problem

The Singleton design pattern address problems such as inconsistent results, unpredictable program behaviour, and overuse of resources that may arise when more than one object operate on some shared resources.

Examples of objects that only one instance is desired are thread pools, caches, dialog boxes, objects that handles preferences (like registry settings), objects used for logging, device drivers, etc. [2]

28.3.4 Participants

The Singleton pattern has only one participant

Singleton

- defines an `instance()` operation that lest clients access its unique instance. This method is a static member function in C++.
- may be responsible for creating and destroying its own unique instance.

28.3.5 Discussion

The Singleton pattern takes control over its own instantiation by hiding its constructor. This is done by declaring the constructor private or protected. If the constructor is private no class other than the singleton class itself can call the constructor. The consequence is that the class can not be instantiated at all because it cannot call its own constructor if it does not yet exist. This problem is solved by providing a public static member function that can be called by any class that needs to use the instance of the singleton class. Recall that a static method can be called by using the class name and the scope operator :: without the need to instantiate the class. This member function is responsible for creating the instance. The UML activity diagram shown in Figure 3 shows the creational logic that has to be implemented by this member function.

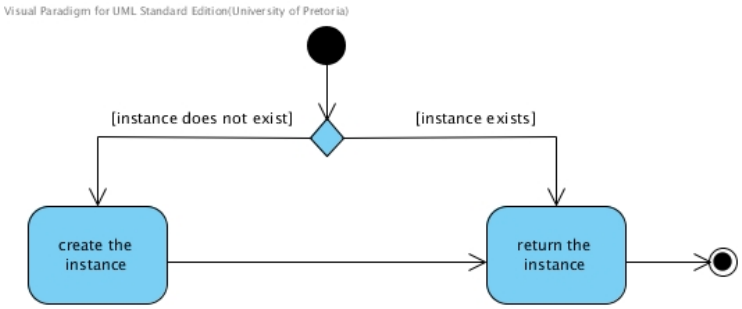

Figure 3: The creational logic of the Singleton Design Pattern

To allow the option of having derived singletons, the constructor can be declared protected. When allowing this it is assumed that the programmer still intend to have only one concrete instance of the singleton to to be instantiated, but would like to decide at runtime what variation is to be instantiated. The system should not allow the co-existence of different classes that are derived from the abstract singleton. The use of derived singletons are beyond the scope of these notes. [2] point to the dangers of the practice to subclass a singleton. The interested reader can find examples of how subclassing of singletons should be done in [5].

28.4 Singleton Implementations in C++

28.4.1 GoF Implementation

Various authors have implemented of the Singleton design pattern with subtle differences to address some of the consequences and flaws of the original implementation suggested by [4] shown in the following code:

```
class GoFSingleton {
  public:
    static GoFSingleton* getInstance();
  protected:
    GoFSingleton();
  private:
    static GoFSingleton* onlyInstance;
};

GoFSingleton* GoFSingleton :: onlyInstance = 0;

GoFSingleton* GoFSingleton :: getInstance() {
  if(onlyInstance == 0) {
      onlyInstance = new GoFSingleton();
  }
  return onlyInstance;
}
```

Observe the following:

- The `onlyInstance` member is declared static. Since this member is accessed for the first time before the class is instantiated it is declared static to ensure that it is assigned memory before the class is instantiated.
- The `onlyInstance` member is initiated outside the constructor. This is required because it is is declared to be static. If this initiation is not done, a call to the singleton will cause a segmentation fault.
- The constructor is protected. This hides it from other classes but allows for subclassing the `GoFSingleton` class.

28.5 Larman implementation

A serious consequence of the combination of multithreaded environments and optimising compilers is unpredictable behaviour where complicated creational logic is implemented. The implementation of the Singleton pattern is particularly at risk. The problems that may arise as a consequence of lazy initialisation can be remedied by applying **eager initialisation**. This is the opposite of lazy initialisation. The object is initialised at the beginning when the program is executed regardless if it is needed or not. The application of eager instantiation avoids complex conditional creation logic at the cost of instantiating an object that might not be used during execution. The following code adapted from [6] shows how this can be implemented:

```
class LarmanSingleton{
    public:
        static LarmanSingleton* getInstance();
    protected:
        LarmanSingleton();
    private:
        static LarmanSingleton* onlyInstance;
};
LarmanSingleton* LarmanSingleton ::
    onlyInstance = new LarmanSingleton();
}

LarmanSingleton* LarmanSingleton :: getInstance(){
    return onlyInstance;
}
```

The above code differs from the code in Section **??** in the following ways:

- Instead of intiating the `onlyInstance` to 0, and instantiating it only on the first call to the `getInstance()`-method it is initiated at startup.
- Because it is guaranteed that `onlyInstance` already exist when the `getInstance()`-method is called, the need to apply the prescribed creational logic is eliminated.

This, however, is usually not preferred because the instantiation of the object which is never actually accessed may be wasteful. Besides, one of the aspects of the intent of the Singleton pattern is to apply lazy initialisation. Technically this implementation is not really as Singleton and does not differ from having a static global variable pointing to the Singleton class.

28.5.1 Müldner Implementation

[11] points out that the unique instance of an object is not guaranteed by hiding only the constructor of the class. It might still possible for other classes to instantiate copies of the Singleton object through calling a copy constructor or assignment operator. Consequently the Singleton pattern is unable to comply with its intent to guarantee that only one instance can exist. This problem is remedied in the following code that is adapted from a Singleton implementation given by [11]:

```
class MuldnerSingleton {
  public:
    static MuldnerSingleton* getInstance();
    void updateSingletonData(int);
    void printSingletonData();
  protected:
    MuldnerSingleton();
    virtual ~MuldnerSingleton(){cout << "Destructing" << endl;};
    MuldnerSingleton(const MuldnerSingleton&){};
    MuldnerSingleton& operator=(const MuldnerSingleton&){};
  private:
    static MuldnerSingleton* onlyInstance;
    int  singletonData;
};

void MuldnerSingleton :: updateSingletonData(int i){
  singletonData = i;
}

void MuldnerSingleton :: printSingletonData(){
  cout << "Singleton data: " << singletonData << endl;
}

MuldnerSingleton :: MuldnerSingleton() : singletonData(0){}

MuldnerSingleton* MuldnerSingleton :: onlyInstance = 0;

MuldnerSingleton* MuldnerSingleton :: getInstance() {
  if(onlyInstance == 0) {
      onlyInstance = new MuldnerSingleton();
  }
  return onlyInstance;
}
```

The above code differs from the code in Section 27.4.1 in the following ways:

- The copy constructor and assignment operator are also hidden by declaring them protected. Since they are now declared, the compiler requires an implementation for each. The implementations provided, however, are empty because they can anyway not be called. They are just stubs to prevent the public ones from being created.
- Since its constructors are protected it is assumed that subclassing may be applied. It is declared virtual to comply with the principle discussed in Section 27.2.2.
- Additional instance variables and member functions were implemented to show how the rest of the Singleton class (its actual functionality) should be implemented, and also to provide detail to be able to see how it behaves during execution.

If the `getInstance()` method of the Singleton class returns a pointer to the only instance of the class, [11] argue that the client should avoid using a variable holding this pointer. One should rather call the operations of the Singleton class through the instantiating operation. For example in the above implementation the `printSingetonData()` operation should be called with the following statement:

```
getInstance()->printSingetonData();
```

This is to avoid the danger that the variable (which is of pointer type) can be used to delete the Singleton object while it is still in use. However, it is generally not a good idea to rely on programmers to treat the objects of your classes properly. Therefore, it would be much better to provide a mechanism to ensure that the Singleton is properly destructed. Understanding the destruction issues and how to address them is beyond the scope of these notes. The interested reader may read [13, 3, 1]

28.5.2 Meyers Singleton

[14] discuss a singleton he calls the Meyers Singleton. It is based on a remark by Meyers in an earlier version of [7]. The remark by Meyers was not intended to be a suggestion about how the Singleton pattern can be implemented. It was intended to offer a solution to address problems related to not being able to control the order of initiation of non-local static objects. Incidentally it provided a neat implementation of the Singleton pattern. This implementation overcomes most problems that may occur when Singletons are constructed. It also eliminates the destruction problems associated with most other implementations of the Singleton pattern. This solution is shown in the following code on the next page. This code makes use of references rather than pointers. It differs from the code in Section 27.5.1 in the following ways:

- The `onlyInstance` variable is no longer a static pointer to `self`. Instead it is now a static local variable (not a pointer) of the `getInstance()` method. The global initialisation of the `onlyInstance` variable is thus no longer required.
- The `getInstance()` method is changed. Instead of returning a pointer, it returns the Singleton itself as a reference. The creation logic is no longer needed. Lazy instantiation of the Singleton is achieved as a consequence of the language feature that static variables of functions are instantiated when the function is first called.

- Since `onlyInstance` is not a pointer, its memory is allocated on the stack. Consequently the destruction of the Singleton is no longer an issue.
- The rest of the implementation (not shown here) is *mutatis mutandis* the same as the implementation shown in Section 27.5.1.

The following is an implementation of the Meyers Singleton adapted from an implementation of this principle given by [11]

```
class MeyersSingleton {
  public:
    static MeyersSingleton& getInstance();
    void updateSingletonData(int);
    void printSingletonData();
  protected:
    virtual ~MeyersSingleton(){};
    MeyersSingleton();
    MeyersSingleton(const MeyersSingleton&){};
    MeyersSingleton& operator=(const MeyersSingleton&){};
  private:
    int singletonData;
};

MeyersSingleton& MeyersSingleton :: getInstance(){
  static MeyersSingleton onlyInstance;
  return onlyInstance;
}
```

Unfortunately the Meyers Singleton is not thread-safe [14]. The reader is referred to [9] and [10] for an interesting discussion about creating thread-safe Singleton.

28.6 Singleton Pattern Explained

28.6.1 Improvements achieved

- If the Singleton class provide access to resources that are shared. The usage of these resources is more robust than would be without the pattern. The Singleton encapsulates its sole instance, hence it can have strict control over how and when clients access it.
- If the Singleton class is resource intensive, the application of the pattern may contribute to more optimal resource usage.
- The Singleton Pattern is an improvement over global variables. Although global variables can provide global access (part of the intent of this pattern) it cannot guarantee that only one instance is instantiated (the more crucial aspect of the intent of this pattern).

28.6.2 Disadvantages

- The application of the Singleton pattern violates the *One class, one responsibilty* principle to an extent. This principle is aimed at increasing cohesion in classes. If a class has unrelated responsibilities the cohesion of such class is low. See Section **??** for a discussion of cohesion and coupling and why it is important. Inadvertently the Singleton class of the Singleton pattern has two unrelated responsibilities, namely to manage its own creation (and possibly destruction) as well as providing the functionality it was designed for in the first place.

- In multithreaded applications the creation step in the singleton design pattern should be made a critical section requiring thread concurrency control. If it is not properly managed it may happen that two different threads both create an instance on the class which may lead to undesired consequences. Furthermore, the overhead when implementing and executing the creation step in a thread-safe manner is needed only once (when the function is called the first time) but may be executed with every call to the function. If not implemented properly it may lead to unacceptable performance degradation. The implementation of the singleton in a thread-safe manner is beyond the scope of these notes. The interested reader can find an example of how this should be done in [12].

28.6.3 Implementation Issues

1. The intent of the Singleton design pattern can in some cases be obtained by means of a static or global variable. This, however, is deemed bad practice. Besides a global variable always applies **eager initialisation**. Thus, the benefits of lazy instantiation is forfeited when a global instance is used as opposed to applying the singleton pattern. Furthermore, sometimes the initialisation of the Singleton object may need additional information which might not be known at static initialisation time, which make it impossible to to apply a static global variable to implement the intent of the Singleton design pattern.

2. When [4] first proposed the Singleton pattern, different techniques for implementations with subclassing was discussed but nothing was said about the destruction of singletons. The problems related to creating Singletons in a multithreaded environment was also not addressed.

28.6.4 Related Patterns

- Singleton and Prototype both sometimes apply lazy instantiation. Singleton does this to save memory while Prototype uses it to save processing power.

- Abstract Factory, and Builder can use Singleton in their implementation.

- Façade objects are often Singletons because only one Façade object is required.

- State objects are often Singletons.

28.7 Example

Figure 4 is a class diagram of a PrintSpooler class. It implements the Muldner implementation of the Singleton Pattern.

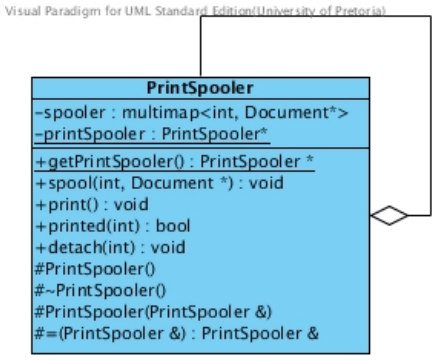

Figure 4: Class Diagram of a PrintSpooler Class

The following table shows the participant and its prescribed members.

Participant	**Entity in application**
Singleton	PrintSpooler
onlyInstance	printSpooler
getInstance()	getPrintSpooler

28.8 Conclusion

The Singleton pattern is a innocent looking pattern that provides a simple solution to ensure that only one instance of a class is instantiated for the duration of program execution. Unfortunately it can potentially give rise to many problems. It is extremely difficult to subclass, its destruction is controversial and it is almost impossible to implement it in a thread-safe manner. If the need to have only one instance of the class is not an absolute necessity, it is probably not worth the risk to implement the pattern.

References

[1] Andrei Alexandrescu. *Modern C++ design : generic programming and design patterns applied*. Addison-Wesley, Boston, MA, 2001.

[2] Eric Freeman, Elisabeth Freeman, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, Sebastopol, CA95472, 1 edition, 2004.

[3] Evgeniy Gabrilovich. Destruction-Managed Singleton: a compound pattern for reliable deallocation of singletons. `http://www.cs.technion.ac.il/~gabr/papers/singleton_cppr.pdf`, 2000. [Online: Accessed 10 November 2012].

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, Reading, Mass, 1995.

[5] Greg Ippolito. C++ Singleton design pattern. `http://www.yolinux.com/TUTORIALS/C++Singleton.html`, 2008. [Online: Accessed 10 November 2012].

[6] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Prentice-Hall, New York, 3^{rd} edition, 2004.

[7] Scott Meyers. *Effective C++: 55 Specific Ways to Improve your Programs and Designs*. Pearson Education Inc, Upper Saddle River, NJ 074548, 3^{rd} edition, 2005.

[8] Scott Meyers. *Effective C++: 55 specific ways to improve your programs and designs*. Pearson Education Inc, Upper Saddle River, NJ 074548, 3^{rd} edition, 2008.

[9] Scott Meyers and Andrei Alexandrescu. C++ and the Perils of Double-Checked Locking: Part I. *Dr. Dobb's Journal*, 29(7):46 – 49, July 2004.

[10] Scott Meyers and Andrei Alexandrescu. C++ and the Perils of Double-Checked Locking: Part II. *Dr. Dobb's Journal*, 29(8):57 – 61, August 2004.

[11] Tomasz Müldner. *C++ Programming with Design Patterns Revealed*. Addison Wesley, 2002.

[12] Arena Red. Revisiting the Thread-Safe C++ Singleton. `http://www.bombaydigital.com/arenared/2005/10/25/1`, October 2005. [Online: Accessed 10 November 2012].

[13] John Vlissides. To Kill a Singleton. `http://www.research.ibm.com/designpatterns/pubs/ph-jun96.txt`, June 1996. [Online: Accessed 10 November 2012].

[14] John Vlissides. *Pattern Hatching: Design Patterns Applied*. Addison-Wesley, 1998.