SOFTWARE ENGINEERING INSTITUTE
华东师范大学软件学院

Object oriented Analysis &Design
面向对象分析与设计

Lecture_17 Design Pattern-Singleton

主讲: 陈小红

日期:

GoF设计模式的分类

（1）Creational（创建型）5个

（2）Structural（结构型）7个

（3）Behavioral（行为型）11个

	创建型	结构型	行为型
类	Factory Method	Adapter_Class	Interpreter Template Method
对象	Abstract Factory Builder Prototype Singleton	Adapter_Object Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

一个类可以有多少个实例?

类的多重性

单例模式Singleton

- 有些类也需要计划生育

Problem

- Application needs one, and only one, instance of an object.
  - examples: keyboard reader, bank data collection
  - we'd like to make it illegal to have more than one, just for safety's sake

Intent

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

只有一个实例的类

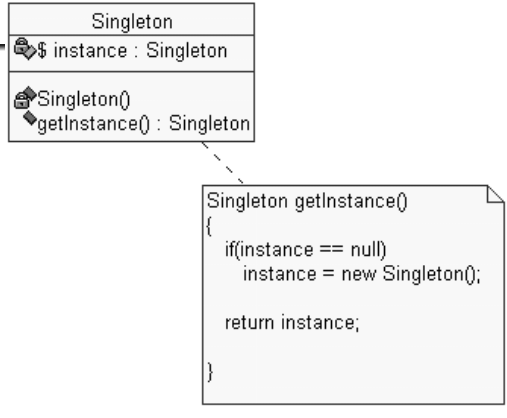
- 这个类谁来实例化?
  - 保证它是唯一的实例
- 这个类的实例可以随时访问

自己

提供访问该实例的方法

Pattern: Singleton

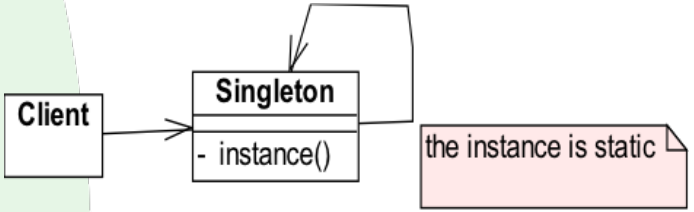
a class that has only one instance



Structure Summary

Singleton

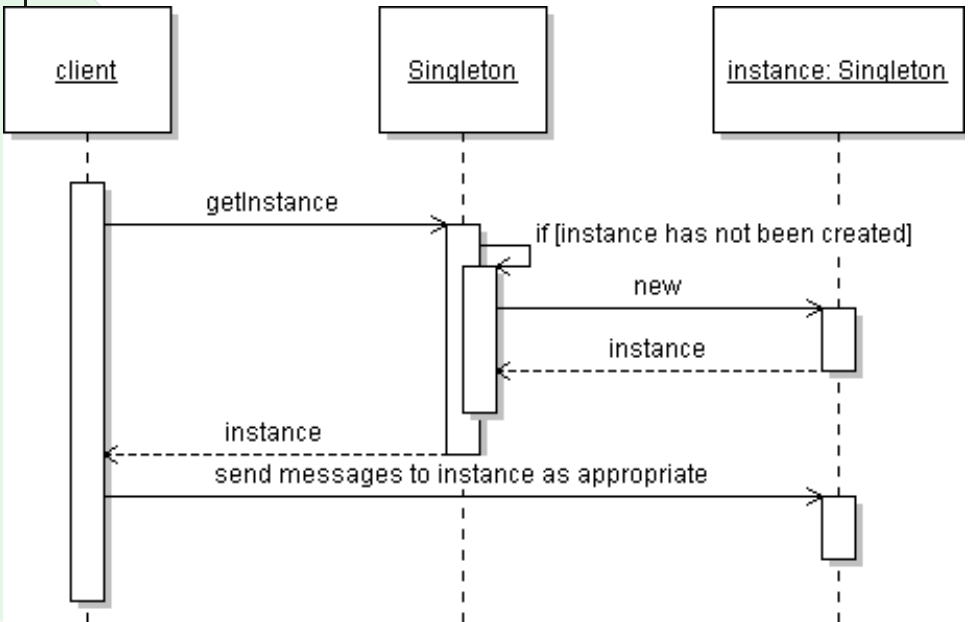
1) Lock-up a class so that clients cannot create their own instances, but must use the single instance hosted by the class itself.



- When to do the instantiation?

- **Constructor: Hungry instantiation**
- **When used: Lazy instantiation**
- creation on first use
- **Which one is better?**

Singleton sequence diagram



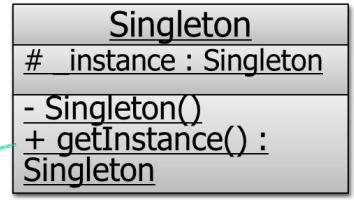
Singleton

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

```
public class Singleton {
 private Singleton() {}
 private static Singleton _instance = null;
 public static Singleton getInstance() {
    if (_instance == null) _instance = new Singleton();
    return _instance;
```

Lazy instantiation

Tactic of delaying the creation of an object, the calculation of a value, or some other exp
process until the first time

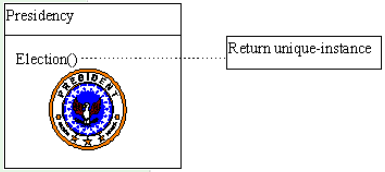



```
return _instance;
```

Singleton – implementation 2


```
class Singleton {
public:
   static Singleton* Instance(); // gives back a real object!
   static proof(void); // proof that the object was made
protected:
    Singleton(); // constructor
private: static Singleton* _singleton;
};
Singleton* Singleton::_singleton = 0;
Singleton* Singleton::Instance() {
  if (_singleton == 0) {
      singleton = new Singleton;
  } // endif
  return _singleton;
} // end Instance()
```

Example



国不可一日无君

一山不容二虎

Forces

There are two forces that affect the Singleton:

- There must be exactly one instance of the class
- The instance must be (easily) accessible to all potential clients

Solution (Check list)

- Define a private static attribute in the "single instance" class.
- Define a public static getInstance() function in the class.
- Do "lazy initialization" (creation on first use) in the getter function.
  - class itself responsible for creating, maintaining, and providing global access to its own single instance.
- Define all constructors to be protected or private.
- Clients may only use the getter function to manipulate the Singleton.

Java Code Fragment: Singleton Patt

How the Singleton pattern works: A special method

```
class USTax {
  public:
    static USTAX* getInstance(); // public static
  private:
    USTax();
    static USTax* instance;
}

USTax* USTax::instance = 0;
USTax* USTax::getInstance(){
  if (instance == 0) {
    instance = new USTax;
  }
  return instance;
}
```

constructors to be private

private static attribute

Singleton example

- consider a singleton class RandomGenerator that generates random numbers

```
public class RandomGenerator {
  private static RandomGenerator gen = new
     RandomGenerator();

 public static RandomGenerator getInstance() {
   return gen;
 }

 private RandomGenerator() {}

 public double nextNumber() {
   return Math.random();
 }
}
```

- possible problem: always creates the instance, even if it isn't used

Singleton example 2

- variation: don't create the instance until needed

```
// Generates random numbers.
public class RandomGenerator {
 private static RandomGenerator gen = null;
 public static RandomGenerator getInstance() {
   if (gen == null)
     gen = new RandomGenerator();
   return gen;
 }
}
```

- What could go wrong with this version?

Singleton example 3

- variation: solve concurrency issue by locking

```
// Generates random numbers.
public class RandomGenerator {
 private static RandomGenerator gen = null;
 public static synchronized
              RandomGenerator getInstance() {
   if (gen == null)
     gen = new RandomGenerator();
   return gen;
 }
}
```

- Is anything wrong with this version?

Singleton example 4

- variation: solve concurrency issue without unnecessary locking

```
// Generates random numbers.
public class RandomGenerator {
 private static RandomGenerator gen = null;
 public static RandomGenerator getInstance() {
   if (gen == null) {
     synchronized (RandomGenerator.class) {
       // must test again -- can you see why?
       if (gen == null)
         gen = new RandomGenerator();
     }
   }
   return gen;
 }
}
```

Conclusion

- The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance.
- But, The Singleton design pattern is one of the most inappropriately used patterns
  - The Singleton does not do away with the global; it merely renames it
- When is Singleton unnecessary?
  - Short answer: most of the time.
  - Long answer: when it's simpler to pass an object resource as a reference to the objects that need it, rather than letting objects access the resource globally

Lab related Implement (Before):

A global variable is default initialized when it is declared - but it is not initialized in earnest until its first use.
This requires that the initialization code be replicated throughout the application.

```
class GlobalClass {
  int  m_value;
  public:
  GlobalClass( int v=0 )  { m_value = v; }
  int  get_value()        { return m_value; }
  void set_value( int v ) { m_value = v; }
};

// Default initialization
GlobalClass*   global_ptr = 0;

void foo( void ) {
  // Initialization on first use
  if ( ! global_ptr )
    global_ptr = new GlobalClass;
  global_ptr->set_value( 1 );
  cout << "foo: global_ptr is "
       << global_ptr->get_value() << '\n';
}
```

```
void bar( void ) {
  if ( ! global_ptr )
    global_ptr = new GlobalClass;
  global_ptr->set_value( 2 );
  cout << "bar: global_ptr is "
       << global_ptr->get_value() << '\n';
}
```

```
int main( void ) {
  if ( ! global_ptr )
    global_ptr = new GlobalClass;
  cout << "main: global_ptr is "
       << global_ptr->get_value() << '\n';
  foo();
  bar();
}
// main: global_ptr is 0
// foo: global_ptr is 1
// bar: global_ptr is 2
```

Lab

// Purpose. Singleton design pattern lab.

// Problem. The application would like a single instance of globalObject to exist, and chooses to implement it as a global. Globals should always be discouraged. Additionally, any code that references the global object, has to first check if the pointer has been initialized, and initialize it if it has not.

// Assignment.

- Replace the global variable globalObject with a private static data member.
- Provide the pattern-specified accessor function.
- Provide for initialization and init testing in the GlobalObject class.
- All client code should now use the Singleton accessor function instead of referencing the globalObject variable.
- Remove any client code dealing with globalObject initialization.
- Guarantee that the GlobalObject class cannot be instantiated.

要求

- 要求：上交电子版
- 写出代码
- 写出最后运行结果
- Deadline: 1月5日