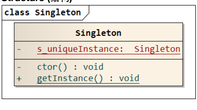
Intent (意图)
Ensure a class only has one instance, and provide a global point of access to it.

Motivation (动机)
It's important for some classes to have exactly one instance.
- one printer spooler
- one file system
- one window manager
- one Director in cocos

- 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)
- provide a way to access the instance.

Applicability (适用性)
- 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 (结构)


Participants (参与者)
- defines an Instance operation that lets clients access its unique instance. Instance is a class operation(that is, a static member function in C++)
- may be responsible for creating its own unique instance.

Collaborations (协作)
- Clients access a Singleton instance solely through Singleton's Instance operation.

Consequences (结果)
Several benefits:
- Controlled access to sole instance. It can have strict control over how and when clients access it.
- Reduced name space. The Singleton pattern is an improvement over global variables.
- Permits refinement of operations and representation. (by subclassed)
- Permits a variable number of instance.
- More flexible than class operations.

Implementation (实现)
- Ensuring a unique instance.
Prefer Using a static member function not the global/ static object.
- We can't guarantee that only one instance of a static object will ever be declared.
- We might not have enough information to instantiate every singleton at static initialization time.
- initialization order.

- Subclassing the Singleton class.

Sample Code (代码示例)
- Sample 1
```
class Singleton
{
public:
static Singleton *getInstance();
protected:
Singleton();
Singleton(const Singleton &copy);
~Singleton();

Singleton &operator=(const Singleton &copy);

private:
static Singleton *_instance;
};

Singleton *Singleton::_instance = 0;
Singleton *Singleton::getInstance()
{
if (_instance == 0) {
_instance = new Singleton;
}

return _instance;
}
```

- Sample 2
```
class Singleton
{
private:
typedef std::pair<std::string, Singleton> NameSingletonPair;

public:
static void registerSingleton(const char *name, Singleton *);
static Singleton *getInstance();

protected:
Singleton();
Singleton(const Singleton &copy);
~Singleton();

Singleton &operator=(const Singleton &copy);

static Singleton *lookUp(const char *name);

private:
static Singleton *_instance;
static std::list<NameSingletonPair> * _registry;
};


Singleton *Singleton::_instance = 0;
Singleton *Singleton::getInstance()
{
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;
}

//register in their constructor.
class MySingleton : public Singleton
{
public:
MySingleton();
};

MySingleton::MySingleton() {
Singleton::registerSingleton("MySingleton", this);
}

static MySingleton theSingleton;
```

- Sample 3
```
Singleton *Singleton::getInstance()
{
if (_instance == 0) {
const char *singletonName = getenv("SINGLETON");
//user or environment supplies this at startup

if (strcmp(singletonName, "bombed") == 0) {
_instance = new BombedMazeSingleton();
} else if (strcmp(singletonName, "enchanted") == 0) {
_instance = new EnchantedMazeInstance();
} else {
_instance = new MazeInstance();
}
}

return _instance;
}
```

- Sample 4
```
template <class T>
class Singleton {
public:
static T *getInstance();
static void destroyInstance();

protected:
Singleton();
Singleton(const T &s);
T &operator=(const T &);

//@note:no virtual, so I prefer inherit this class by private
// public inherit may not occur Error, because of destructor is protected
~Singleton();

static T *s_instance;
};


template <class T>
T *Singleton<T>::s_instance = nullptr;

template <class T>
T *Singleton<T>::getInstance() {
if (s_instance == nullptr) {
s_instance = new (std::nothrow)T;
}

return s_instance;
}

template <class T>
void Singleton<T>::destroyInstance() {
if (s_instance) {
delete s_instance;
}
}


template <class T>
Singleton<T>::Singleton()
{
}

template <class T>
Singleton<T>::~Singleton()
{
s_instance = nullptr;
}

//inherit from Singleton Template class
class AudioManager : private Singleton<AudioManager>
{
private:
class SoundEffectInfo;
friend class Singleton<AudioManager>;

AudioManager();
~AudioManager();

public:
using Singleton<AudioManager>::getInstance;
using Singleton<AudioManager>::destroyInstance;
};
```

Known Uses (已知应用)
...

Related Patterns (相关模式)
- Abstract Factory
- Builder
- Prototype

Summary
确保一个类只有一个实例，并提供全局访问点

单件模式是经得起时间考验的方法，可以确保只有一个实例会被创建。

不建议使用静态类的原因：静态初始化的控制权是在Java手上，这么做有可能导致和初始化次序有关的bug。

所以推荐使用对象的单件，不推荐类的单件

Java中，如果程序有多个类加载器又同时使用了单件模式，请小心。解决：自行指定类加载器，并指定同一个类加载器。

违反"单一职责"原则

责任一：单件类负责管理自己的实例，并提供全局访问

责任二：在应用程序中担当角色。