

Patrones de Diseño

Singleton

TPV2

Samir Genaim

¿Que es el Patrón Singleton?

- ♦ Hay objetos que sólo necesitamos una instancia: thread pools, dialog boxes, device drivers, etc.
- ♦ Además, tener más de una instancia puede causar problemas como el comportamiento incorrecto, uso excesivo de recursos o resultados inconsistentes.
- ♦ En videojuegos, por ejemplo la clase que maneja el sonido, las texturas, etc.

El Patrón Singleton

Asegura que una clase tiene sólo una instancia y proporciona un punto de acceso global

Singleton Object

```
class SomeClass {  
    static SomeClass* instance_;  
    SomeClass() { ... }  
public:  
    static SomeClass* instance() {  
        if (instance_ == nullptr)  
            instance_ = new SomeClass();  
        return instance_;  
    }  
  
    // more methods  
    void m(...) { ... }  
    ...  
};  
SomeClass* SomeClass::instance_ = nullptr; // en el .cpp
```

Un atributo para la única instancia de **SomeClass**.

La constructora es private, imposible crear instancias fuera de la clase.

Un método para obtener la instancia. El 'new' se ejecuta sólo un vez

Se usa llamando a instance() para obtener el objeto:

SomeClass::instance()->m(...)

Siempre devuelve la misma instancia

Ejemplo: FileSystem

```
class FileSystem {
    static FileSystem* instance_;
    FileSystem() { ... }
public:
    static FileSystem* instance() {
        if (instance_ == nullptr)
            instance_ = new FileSystem();
        return instance_;
    }

    void read(...) { ... }
    void write(...) { ... }
};

FileSystem* fs = FileSystem::instance();
fs->read(...);
fs->write(...);

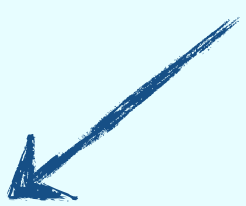
FileSystem* fs2 = FileSystem::instance();
fs2->read(...);

...
cout << "Equal? " << (fs == fs2) << endl;
```

Singleton con Parámetros

```
class SomeClass {  
    static SomeClass* instance_;  
    SomeClass(...) { ... }  
public:  
    static void init(...) {  
        if (instance_ == nullptr)  
            instance_ = new SomeClass(...);  
    }  
  
    static SomeClass* insatnce() {  
        assert(instance_ != nullptr);  
        return instance_;  
    }  
    // more methods  
    ...  
};
```

Un método para crear la instancia — hay que llamar a este método al principio de programa.



Obtener la instancia. Lanza una excepción si no se ha creado todavía ...



importante!

Hay que ocultar la constructora de copia y el operador de asignación (porque siempre existen por defecto) para prohibir hacer copias del Singleton ...

```
class SomeClass {  
    SomeClass(SomeClass&) = delete;  
    SomeClass& operator=(const SomeClass&) = delete;  
    ...  
};
```

Usando delete en C++11

```
class SomeClass {  
    SomeClass(SomeClass&);  
    SomeClass& operator=(const SomeClass&);  
    ...  
};
```

declaración sin implementación

¿Como borrar un Singleton?

- ✦ ¿Deberíamos liberar la memoria del Singleton antes de salir? ¿Quién debe hacerlo?
- ✦ Si se trata de sólo liberar la memoria, entonces se puede no liberarla. El sistema operativo reclama esa memoria al salir del programa — normalmente no se considera como **memory leak** porque **no es una memoria que potencialmente se puede reusar durante la ejecución**.
- ✦ Si necesitamos hacer alguna tarea antes de salir del programa, p.ej., cerrar archivos, etc., entonces hay que invocar al destructor del Singleton (mediante la liberación de la memoria)

¿Como borrar un Singleton?

- ✦ Una solución sencilla es borrar la instancia antes de salir del programa, simplemente ejecutando la instrucción `delete SomeClass::instance()`
- ✦ Pero eso no es recomendable, porque tenemos que hacerlo para cada Singleton en el programa, sería mucho mejor dejar a cada Singleton la tarea de borrarse ...
- ✦ Recuerda que el destructor de un atributo estático se invoca antes de salir del programa, pero `instance_` es un puntero, ¿cómo podemos invocar al destructor del objeto al que señala?

Usar un Destructor

```
class SomeClass {  
  
    static SomeClass* instance_;  
    static Destructor<SomeClass> instanceManager_;  
  
    SomeClass() { ... }  
public:  
    static SomeClass* instance() {  
        if (instance_ == nullptr) {  
            instance_ = new SomeClass();  
            instanceManager_.setObject(instance_);  
        }  
        return instance_;  
    }  
    ...  
};
```

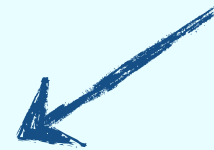
Antes de salir del programa se destruye instanceManager_ porque es estático y no es un puntero. La idea es borrar instance_ en el destructor de Destructor

Al crear la instancia del Singleton, la pasamos al instanceManager_ para destruirla en su destructor ...

La clase Destructor

```
template<typename T>
class Destructor {
public:
    Destructor() :
        managedObj_(nullptr) {
    }
    virtual ~Destructor() {
        if (managedObj_ != nullptr)
            delete managedObj_;
    }
    void setObject(T* o) {
        managedObj_ = o;
    }
private:
    T* managedObj_;
};
```

Borra la instancia que
está manejando en su
destructor ...



Usar Smart Pointers

```
class SomeClass {  
    static unique_ptr<SomeClass> instance_;  
    SomeClass() { ... }  
public:  
    static SomeClass* instance() {  
        if (instance_.get() == nullptr) {  
            instance_.reset( new SomeClass() );  
        }  
        return instance_.get();  
    }  
    ...  
};
```

A partir de C++11 se puede usar smart pointers que proporcionan (entre otras cosas) algo parecido a la la clase Destructor ...

get() devuelve la instance que el smart pointer está manejando y reset(...) la cambia.

Singleton sin memoria dinámica ...

```
class SomeClass {  
    static SomeClass instance_;  
    static bool instantiated_;  
    SomeClass() {}  
public:  
    static SomeClass* instance() {  
        if ( !instantiated_ )  
            init(...);  
        return &instance_;  
    }  
    static void init(...) {  
        assert(!instantiated_); ... ; instantiated_ = true;  
    }  
    void m(...) { ... }  
};
```

La constructora por defecto no hace nada



Usar init para inicializar la instancia. Se pueden pasar parámetros



`SomeClass::instance()->m(...);`

```
SomeClass SomeClass::instance_; // en el .cpp  
bool SomeClass::instantiated_ = false; // en el .cpp
```

Ten cuidado con la inicialización ...

- ✦ La instancia se crea la primera vez que se usa ...
- ✦ Esto puede ser bueno en algunos contextos, porque si no se usa no se crea ...
- ✦ Por otro lado, en otros contextos, como videojuegos, esto puede dar problemas porque si la inicialización tarda lo vamos a notar durante el juego. ...
- ✦ A veces mejor inicializar siempre al principio de programa ...

Ventajas y Desventajas

Las ventajas y desventajas dependen del contexto ...

- + No crea la instancia si nadie la usa ...
- + Se inicializa en tiempo de ejecución ...
- + ...

- Más difícil razonar sobre el código, porque es como una variable global ...
- Anima el acoplamiento de clases ...
- No son muy compatibles con la programación concurrente (hay que tener cuidado al crear la instancia) ...
- ...

El Patrón Singleton: alternativas

Esto es nuestro objetivo principal ...



Asegura que una clase tiene sólo una instancia y

proporciona un punto de acceso global



Esto es para conseguir que ese objeto sea accesible desde todas partes del programa. Pero la solución no es ideal en general porque es una variable global y anima el acoplamiento de clases — pero es una solución muy muy muy cómoda ...

Instancia única: alternativas

```
class SomeClass {  
public:  
    SomeClass( ... ) {  
        assert(!instantiated_);  
        instantiated_ = true;  
        ...  
    }  
  
    // methods  
    void m(...) { ... }  
    ...  
private:  
    static bool instantiated_;  
};
```

La constructora es public,
cualquiera puede crear
una instancia ...

... pero si intentamos crear más
de una instancia lanza una
excepción

```
bool SomeClass::instantiated_ = false; // en .cpp
```

Singleton Template

Cada vez que definimos una clase como Singleton hacemos lo mismo: definir init, definir instance, etc. Sería ideal hacerlo esto sólo una vez y reusarlo.

```
class SomeClass {  
    ...  
    SomeClass(...) { ... }  
public:  
    ...  
    void m(...) { ... }  
    ...  
}
```

¿Cómo definimos la clase Singleton?

```
Singleton<SomeClass>::init(...)  
Singleton<SomeClass>::instance()->m(...)  
...
```

Singleton Template (I)

```
template<typename T>
class Singleton {
public:
    Singleton() = delete;
```

```
template<typename ...Targs>
inline static T* init(Targs &&...args) {
    assert(instance_.get() == nullptr);
    instance_.reset(new T(std::forward<Targs>(args)...));
    return instance_.get();
}
```

```
inline static T* instance() {
    if (instance_.get() == nullptr) init();
    return instance_.get();
}
```

private:

```
    static std::unique_ptr<T> instance_;
};

template<typename T>
std::unique_ptr<T> Singleton<T>::instance_;
```

¡La clase Singleton tiene que tener acceso a las constructoras de T. Pero el resto de clases no tienen que tener acceso!

La instancia es de tipo T

Uso de Singleton Template (I)

```
class SomeClass {  
    friend Singleton<SomeClass>;
```

```
    ...
```

```
    SomeClass() { ... }
```

```
    SomeClass(...) { ... }
```

```
public:
```

```
    SomeClass& operator=(const SomeClass& o) = delete;
```

```
    SomeClass(const SomeClass& o) = delete;
```

```
    void m(...) { ... }
```

```
    ...
```

```
}
```

Para que Singleton<SomeClass>
tenga acceso a las constructoras.
Nadie más tiene acceso!

Borrar todo lo que permite copiar

```
Singleton<SomeClass>::init(...)
```

```
Singleton<SomeClass>::instance()->m(...)
```

```
...
```

Singleton Template (II - Herencia)

```
template<typename T>  
class Singleton {  
protected:  
    Singleton() {}
```

La constructora por defecto es protected, para poder usarla desde de la subclase

public:

```
Singleton<T>& operator=(const Singleton<T>& o) = delete;
```

```
Singleton(const Singleton<T>& o) = delete;
```

```
virtual ~Singleton() {}
```

```
// instance y init como antes ...
```

private:

```
static std::unique_ptr<T> instance_;
```

```
};
```

```
template<typename T>
```

```
std::unique_ptr<T> Singleton<T>::instance_;
```

Borrar lo que permite copiar, no hace falta borrarlo en SomeClass

La instancia es de tipo T

Uso de Singleton Template (II)

```
class SomeClass : public Singleton<SomeClass> {  
    friend Singleton<SomeClass>;
```

```
    ...
```

```
    SomeClass() { ... }
```

```
    SomeClass(...) { ... }
```

```
public:
```

```
    ...
```

```
    void m(...) { ... }
```

```
    ...
```

```
}
```

Para que Singleton<SomeClass> tenga acceso a las constructoras. Nadie más tiene acceso!

SomeClass tiene métodos init y instance! Las hereda de Singleton<SomeClass>

```
SomeClass::init(...)  
SomeClass::instance()->m(...)
```

```
...
```

Resumen

No es perfecto porque es como una variable global, pero es tan cómodo que no se puede resistir a usarlo ...

Usalo con moderación ...