C++ Design Patterns: Singleton in Detail

Christoph Schmidt

Seminar Advanced C++ Programming

11.06.2013

Outline:

1. What is a design pattern and why should I use it?
- General advantages and disadvantages
- Different types of design patterns

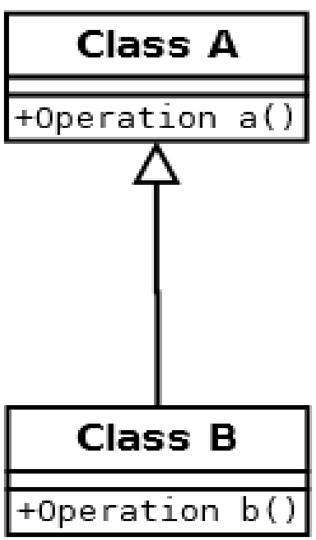
2. The singleton pattern
- Typical problem(s)
- Implementation of singletons
- Thread safety issues and solutions
- Destruction of a singleton
- Life time control

3. Conclusion

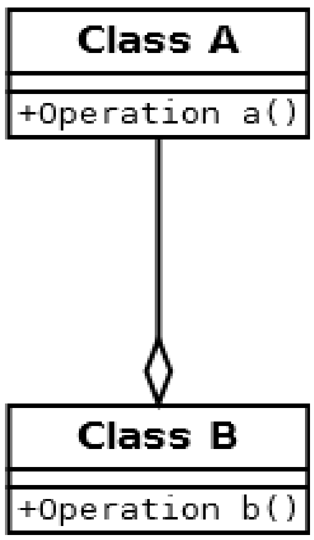
1. What is a design pattern and why should I use it?

Origin:

Problems that appear frequently for many programmers while designing Code



?



- General, reusable solutions to problems occuring while designing code
- Typically found in descripted form
- Programmers have to implement them themselves
- 3 different types of patterns

Advantages and disadvantages of using design patterns

Advantages:

- Always correct
- Usability only depends on the problem
- Speeds up the development process of software
- Makes communication between developers easier

Disadvantages:

- May decrease understandability of the code and design
- Danger of understanding design patterns as an allround solution
- Risk of higher Memory consumption due to generalized format

Different types of design patterns

1. **Creational**
   - Singleton
   - Abstract Factory

2. **Structural**
   - Adapter
   - Composite

3. **Behavioral**
   - Observer
   - Visitor

2. The singleton pattern

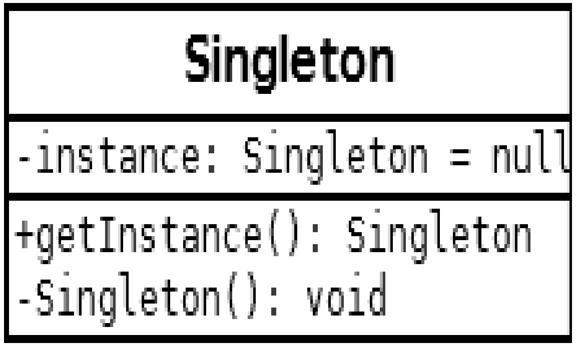
Typical problem(s)

- There shall be only one object of a class during the execution of a programm
- I need a global instance of my object

Examples

- A central object for producing output to a file
- Jobs for a printer that a written to a single buffer
- Accessing the GPU in video games
- Database connections

Implementation of singletons



Implementation of singletons

```
class MySingleton
{
    public:
        MySingleton (const MySingleton&) = delete;
        MySingleton& operator=(const MySingleton&) = delete;
        static MySingleton * getInstance()
        {
            if(theOnlyInstance == nullptr)
            {
                theOnlyInstance = new MySingleton();
            }
            return theOnlyInstance;
        }

        void release(); //will be the focus later

    private:
        static MySingleton * theOnlyInstance;

        MySingleton();
        ~MySingleton();
};
```

Implementation of singletons

```
class MySingleton
{
    public:
        MySingleton (const MySingleton&) = delete;
        MySingleton& operator=(const MySingleton&) = delete;
        static MySingleton * getInstance()
        {
            static MySingleton theOnlyInstance;
            return &theOnlyInstance;
        }

    private:
        MySingleton();
        ~MySingleton();
};
```

Thread safety issues and solutions

```
int main()
{
    MySingleton *s1;
    //create 2 threads
    s1 = MySingleton::getInstance();
    //..
    return 0;
}
```

```
//..
static MySingleton * getInstance()
{
    if(theOnlyInstance == nullptr)
    {
        theOnlyInstance = new MySingleton();
    }
    return theOnlyInstance;
}
//..
```

Solution 1 (Eager Instantiation)

```
int main()
{
    MySingleton *s1;
    s1 = MySingleton::getInstance();
    //do every thing with multithreading
    //here and below
    return 0;
}
```

```
//..
static MySingleton * getInstance()
{
    if(theOnlyInstance == nullptr)
    {
        theOnlyInstance = new MySingleton();
    }
    return theOnlyInstance;
}
//..
```

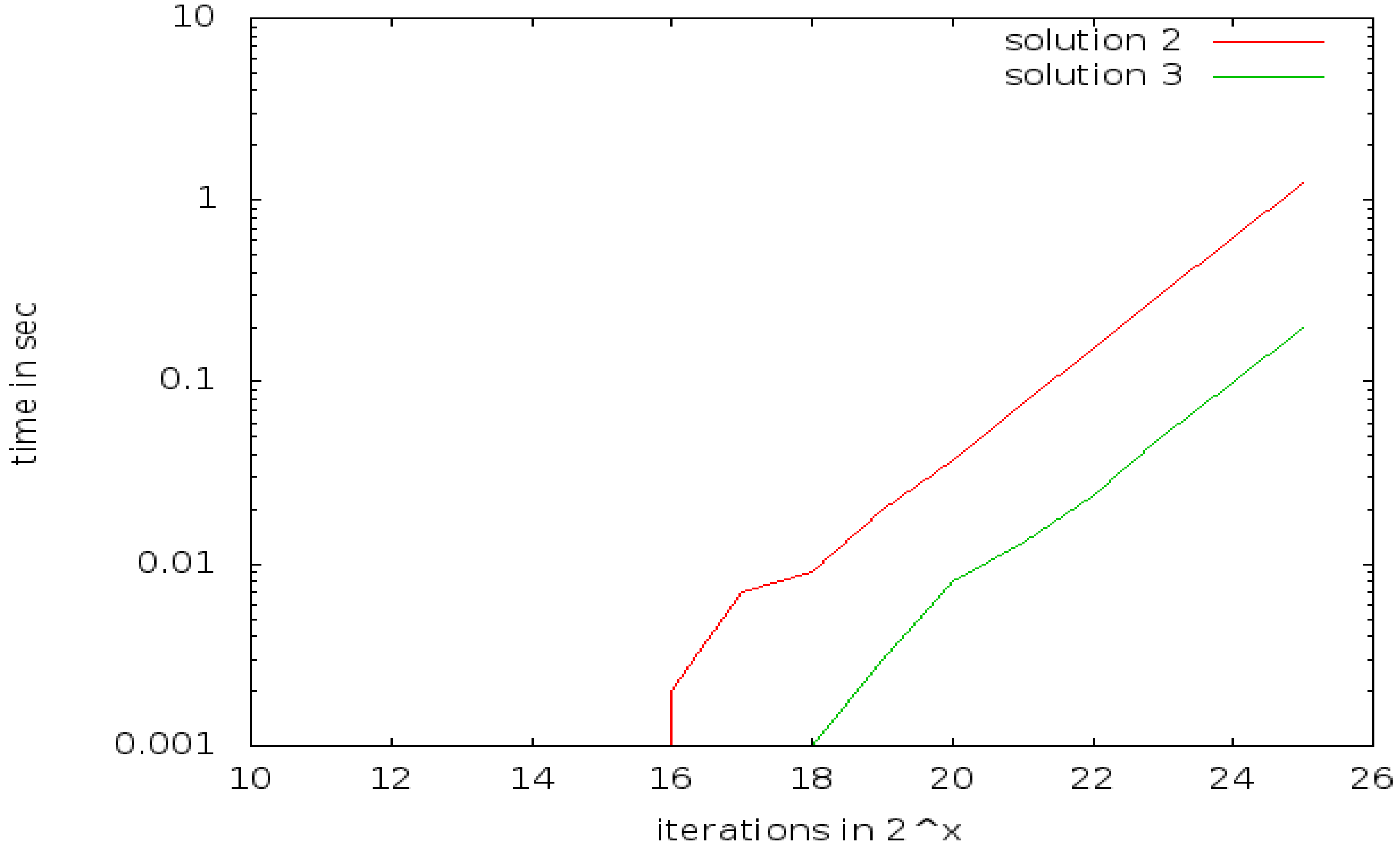
Solution 2

```
public:
    static MySingleton * getInstance()
    {
        std::unique_lock<std::mutex> lock(m);
        if(theOnlyInstance == nullptr)
        {
            theOnlyInstance = new MySingleton();
        }
        return theOnlyInstance;
    }
```

Solution 3 (solution 2 improved)

```
public:
    static MySingleton * getInstance()
    {
        if(theOnlyInstance == null)
        {
            std::unique_lock<std::mutex> lock(m);
            if(theOnlyInstance == null)
            {
                theOnlyInstance = new MySingleton();
            }
        }
        return theOnlyInstance;
    }
```

Solution 2 vs solution 3



Destruction of a singleton

```
class MySingleton
{
    public:
        static MySingleton * getInstance()
        {
            static MySingleton theOnlyInstance;  //guaranteed to be destroyed
            return &theOnlyInstance;
        }

    private:
        MySingleton();
        ~MySingleton();
};
```

No problem for this version! → The MySingleton object is guaranteed to be destroyed

```
class MySingleton
{
    public:
        static MySingleton * getInstance()
        {
            if(theOnlyInstance == nullptr)
            {
                theOnlyInstance = new MySingleton();
            }
            return theOnlyInstance;
        }

        void release()
        {
            //will be the focus NOW!
        }

    private:
        static MySingleton * theOnlyInstance;

        MySingleton();
        ~MySingleton();
};
```

```
void release()
{
    delete theOnlyInstance;
    theOnlyInstance = nullptr;
}
```

```
int main()
{
    MySingleton *s1;
    s1 = MySingleton::getInstance();
    //do all the funny stuff
    s1->release();
    return 0;
}
```

Life time control

Problem:

No general control over the lifetime of static objects.

Construction:

Singleton objects are constructed with the first call to getInstance()

Destruction:

Static objects like singletons are destroyed in reverse order as they were created.
This might cause troubles!

C++ Design Patterns: Singleton in Detail

```cpp
class MySingletonA
{
    public:
        static MySingletonA * getInstance();

    private:
        MySingletonA();
        ~MySingletonA()
        {
            MySingletonB::getInstance()->foo();
        }
};

class MySingletonB
{
    public:
        static MySingletonB * getInstance();
        void foo();

    private:
        MySingletonB();
        ~MySingletonB();
};
```

```cpp
int main()
{
    MySingletonA * a;
    MySingletonB * b;
    a = MySingletonA::getInstance();
    b = MySingletonB::getInstance();
    //...
    return 0;
}
```

a is calling foo() after the destruction of b, a is talking to a dead reference!

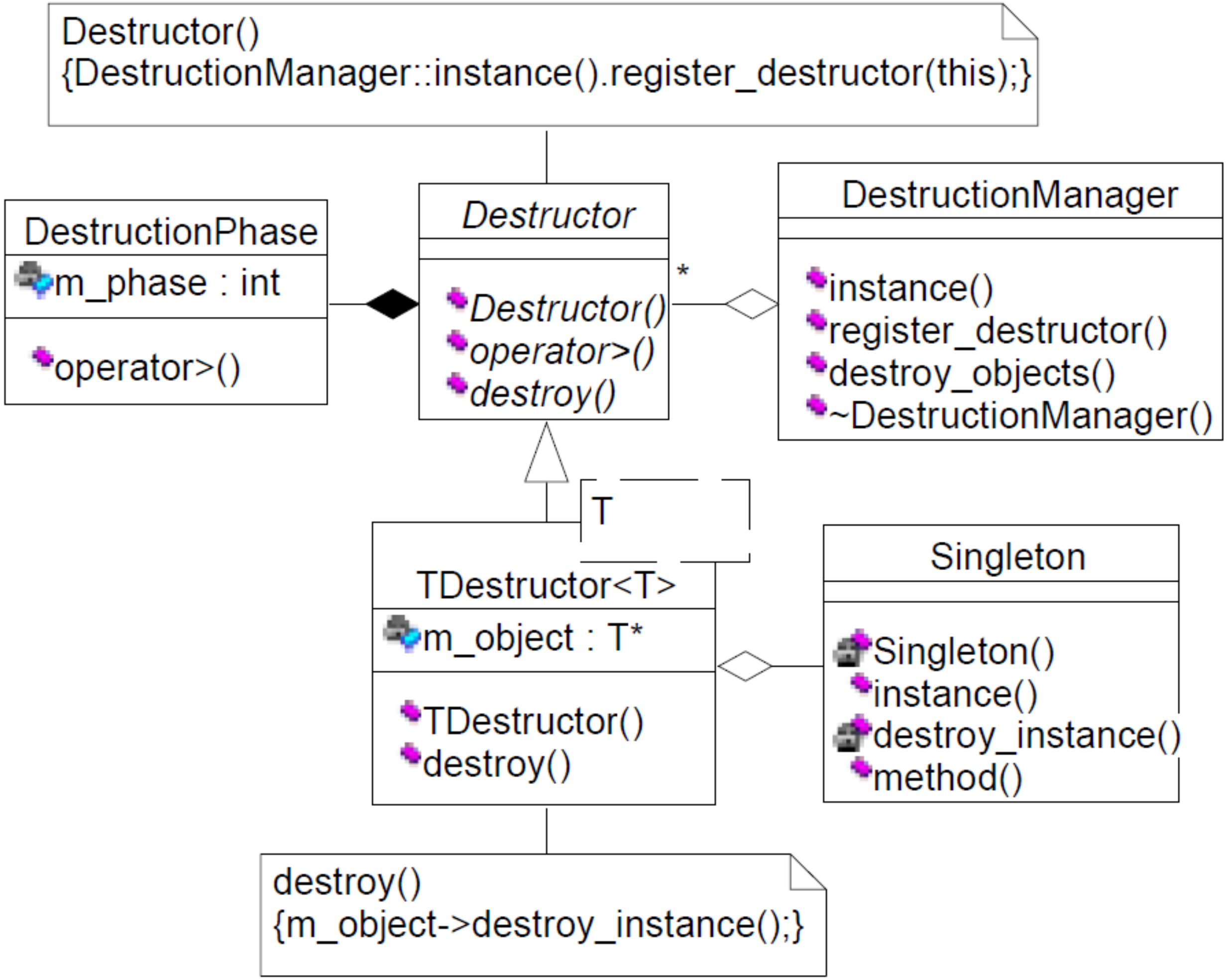
A possible solution: The Phoenix Singleton

```
//in Si.h
class Si
{
    Public:
        static Si & getInstance(){
            static Si inst;
            instance_ = &inst;
            if(destroyed_){
                new(instance_) Si();
                std::atexit(destroy);
                destroyed_ = false;
            }
            return inst;
        }
    private:
        Si();
        ~Si(){ destroyed = true; }
        static void destroy(){ instance->~Si(); }
        static bool destroyed_;
        static Si *instance_;

};
```

```
//in Si.cpp
Si * Si::instance_ = nullptr;
bool Si::destroyed = false;
```

Other possible solution: Destruction-Managed Singleton



3. Conclusion

- If a design problem occures, check if there is a pattern for it
- Make sure the pattern really fits your problem
- Check if there is an easier solution
- If not, use it but
- Make sure to watch out for problems like multithread safety and life time control

Design patterns provide good solutions for typical design problems, but they aren't the answer to everything!

Sources:

http://www.codeproject.com/Articles/96942/Singleton-Design-Pattern-and-Thread-Safety

http://www2.cs.uni-paderborn.de/cs/ag-wehrheim/vorlesungen/ss06/prosem/folien/Warneke.pdf

http://de.wikipedia.org/wiki/Entwurfsmuster

http://de.wikipedia.org/wiki/Singleton_%28Entwurfsmuster%29

http://oette.wordpress.com/2009/09/11/singletons-richtig-verwenden/

http://stackoverflow.com/questions/2496918/singleton-pattern-in-c

http://www10.informatik.uni-erlangen.de/Teaching/Courses/SS2010/CPP/sembritzki.pdf

http://www.cs.technion.ac.il/~gabr/papers/singleton_cppr.pdf/