
Singleton



Singleton – obsah

- **Motivace**
- **Základní myšlenka**
- **Implementace**
- **Problémy**
 - destrukce
 - vícevláknovost
- **Dědičnost**
- **Obecná implementace**
- **Shrnutí**



Singleton – motivace

■ Co mají následující příklady společného?

- DatabaseConnectionPool – přiděluje spojení do DB jednotlivým částem aplikace
- WindowManager – v okenním systému existuje jeden objekt spravující okna
- Keyboard, display, log – (KDL)
- PrinterSpooler – více tiskáren, jeden manažer
- FileSystem
- System clock

■ Odpověď:

- Třídy řešící tyto problémy vyžadují jedinou instanci. Mít více instancí je nežádoucí až nebezpečné.
- Měly by být lehce přístupné z libovolného místa.



Singleton – naivní implementace

→ základní myšlenka

■ Globální proměnná

- zaručuje globální přístupový bod
- znehledňuje namespace a kód
- nedokáže zaručit jedinou instanci objektu

■ Statická data + statické metody

- zaručuje globální přístupový bod
- zaručuje jedinou instanci
- problém rozšiřitelnosti – statické metody nemohou být virtuální
- problém inicializace (závislosti)
- problém úklidu (kdo to má dělat)

■ Singleton

- třída se o svou jedinou instanci stará sama
- inicializace: konstruktor
- úklid: destruktork



Singleton – požadavky

■ Zaručení jediné instance

- ❑ zakázat vytvoření více instancí
- ❑ zakázat kopírování existující instance
- ❑ zakázat rušení
- ❑ instanci spravuje singleton

■ Globální přístupový bod

- ❑ použití na libovolném místě kódu
- ❑ přístup poskytován samotnou třídou

■ Možnost rozšiřitelnosti

- ❑ bez nutnosti zásahu do kódu předka
- ❑ zachovává všechny nástroje objektového programování
 - virtuální metody, abstraktní metody, přetěžování metod
- ❑ pomocí šablon

```
class Singleton {
    public static Singleton instance();

    protected Singleton();
    protected Singleton(final Singleton s);

    /* C++
    Singleton& operator=(const Singleton&);
    virtual ~Singleton();
    */
    ...

};
```



Singleton – implementace

```
class Singleton {  
private static Singleton instance;  
  
public static Singleton instance() {  
    if (instance == null) {  
        instance = new Singleton;  
    }  
    return instance;  
}  
  
public void doSomethingUseful() {...}  
  
protected Singleton() {...}  
protected Singleton(final Singleton s);  
/* C++  
Singleton& operator=(const Singleton&);  
~Singleton() {...}  
*/  
  
void main(String[] args) {  
    Singleton.instance().doSomethingUseful();  
    ...  
}
```

Ukazatel na instanci si spravuje třída sama.
Je static

Statická metoda pro přístup k jediné
instanci.

Instance se vytvoří při prvním volání
`instance()` – až když je to potřeba.

Konstruktor nesmí být public. Inicializuje
třidu. Volán jenom zevnitř třídy.

Destruktor nesmí být public, někdo by mohl
smazat instanci.

Jednoduchý příklad použití.



Singleton – problémy

■ GoF '95

- relativně jednoduchý vzor
- od té doby rozsáhlé diskuse
- Vlissides '98:

- “I used to think it was one of the more trivial of our patterns ... Boy, I was wrong!”

■ Problém destrukce (lifetime)

- zodpovědnost za zrušení
- kdy je objekt zrušen
- některé objekty je potřeba mít přístupné vždy

■ Problém vícevláknových aplikací

- zabezpečení jedinečnosti

■ Dědičnost



Singleton – destrukce Ostrich



- **'Ostrichovo' řešení (leaking singleton)**
 - problém destrukce ignorovat
 - **statická paměť se uvolní automaticky při ukončení procesu**
 - co když je třeba říci vzdálené straně že končím?
 - jako každá kulturní třída by měl mít destruktory
 - zápis do logu, uzavření spojení, odhlášení, ...
 - diskuze o tom co je a co není leak (memory leak, resource leak)

- **V čem je problém?**
 - destruktory se nezavolá
 - ukládáme statický ukazatel, ne statický objekt



Singleton – destrukce killer



■ Idea

- ukazatel zabalit do třídy starající se o destrukci
- kompilátor se stará o zrušení statického objektu singletonKiller, který ve svém destruktoru instanci Singleton zabíjí



Singleton – destrukce killer (C++)



```
class SingletonKiller {
public:
    void setSingleton(Singleton* _instance) {
        instance = _instance;
    }
    Singleton* getSingleton() {
        return instance;
    }
    ~SingletonKiller() {
        delete instance;
    }
private:
    static Singleton* instance;
};
```

```
class Singleton {
public:
    static Singleton* instance();
private:
    Singleton();
    ~Singleton();
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);

    static SingletonKiller singletonKiller;
};
```

```
Singleton* SingletonKiller::instance = 0;
SingletonKiller Singleton::singletonKiller;
```

■ Idea

- ukazatel zabalit do třídy starající se o destrukci
- kompilátor se stará o zrušení statického objektu singletonKiller, který ve svém destruktoru instanci Singleton zabíjí

2. destruktorkillera

killer obsahuje ukazatel na náš singleton

3. destruktorkillera

1. destrukce statické proměnné

```
Singleton* Singleton::instance() {
    if (!singletonKiller.getSingleton()) {
        singletonKiller.setSingleton(
            new Singleton);
    }
    return singletonKiller.getSingleton();
}
```



Singleton – Meyers (C++)



■ Scott Meyers

- ❑ místo operátoru `new`, statická lokální proměnná
- ❑ instanci nedržíme ve statickém ukazateli
- ❑ funkce vracející referenci na statický objekt ve funkci

```
class Singleton {  
public:  
    static Singleton& instance() {  
        static Singleton inst;  
        return inst;  
    }  
  
private:  
    Singleton() {...}  
    Singleton(const Singleton&);  
    Singleton& operator=(const Singleton&);  
    ~Singleton() {...}  
};  
  
int main(int argc, char** argv) {  
    Singleton& s = Singleton::instance();  
    ...  
}
```

2. inicializace statického objektu
pouze při prvním průchodu
registrace atexit

4. návrat zkonstruovaného objektu

3. konstruktor objektu

6. destruktor objektu

1. zavolá se metoda

5. konec programu, destrukce
statických proměnných



Singleton – funkce atexit = generované kompilátorem

■ Odstraňování statických proměnných

- LIFO – nejdříve se odstraní naposledy inicializované
- `int atexit(void* (pFunction) ()) ;`
 - při vytváření objektu se zaregistruje funkce pro zrušení
 - při ukončení programu se postupně zavolají registrované funkce

```
Singleton& Singleton::instance() {  
    extern void __constructSingleton(void* memory);  
    extern void __destroySingleton();  
  
    static bool __initialized = false;  
    static char __buffer[sizeof(Singleton)];  
  
    if (!__initialized) {  
        __constructSingleton(__buffer);  
        atexit(__DestroySingleton);  
        __initialized = true;  
    }  
  
    return *reinterpret_cast<Singleton*>(__buffer);  
}
```

funkce generované kompilátorem

proměnné generované kompilátorem

`__buffer` obsahuje Singleton

volání funkce
`__constructSingleton()` zavolá
konstruktor na paměti `__buffer`

zaregistruje destrukci



Singleton – problém s pořadí destrukce



■ Dead reference problem

- při nevhodném pořadí mohou vzniknout reference na neexistující objekty
- příklad: singletony Keyboard, Display, Log
 - vytvoření instance Log pouze při chybě

inicializace Keyboard
inicializace Display s chybou
inicializace Log a zapsání chyby
konec programu
destrukce Log
destrukce Display
destrukce Keyboard s chybou
reference neexistujícího objektu Log



- destrukce Logu by měla následovat až po destrukcích ostatních singletonů
- nebo aspoň poznat problém a slušně umřít



Singleton – detekce mrtvé reference

■ Detekce destruovaného objektu

- přidání statické proměnné, její nastavení v destruktoru



```
class Singleton {
public:
    static Singleton& instance() {
        if (!instance) {
            if (destroyed) throw ...;
            else create();
        }
        return *instance;
    }
private:
    static void create() {
        static Singleton inst;
        instance = &inst;
    }
    Singleton() {...}
    Singleton(const Singleton&);
    Singleton& operator=(const Singleton&);
    ~Singleton() {
        destroyed = true;
        instance = NULL;
    }
    static bool destroyed;
    static Singleton* instance;
};
Singleton* Singleton::instance = 0;
bool Singleton::destroyed = false;
```

detekce problému

inicializace přesunuta
do privátní metody

nastavení zrušení

příznak zrušení



Singleton – Fénix



- **Detekce někdy nestačí - nutnost přístupu k singletonu kdykoliv**
 - idea: bájný pták Fénix vstane ze svého popela
 - znovuvytvoření při detekci zrušeného objektu
 - příklad KDL - Keyboard a Display obyčejné singletony, Log Fénix
 - C++: paměť statických objektů zůstane alokována až do konce běhu programu
 - problém: stav starého mrtvého Singletonu je navždy ztracen

```
class Singleton {  
    ...  
    void KillPhoenix();  
};  
  
void Singleton::OnDeadRef() {  
    Create();  
    new(instance) Singleton;  
    atexit(KillPhoenix);  
    destroyed = false;  
}  
  
void Singleton::KillPhoenix() {  
    instance->~Singleton();  
}
```

zbytek třídy nezměněn

při detekci zrušení se uloží reference na paměť zrušeného objektu

placement new
zavolání konstrukturu na daném místě

registrace destrukturu fénixu

explicitní zavolání destrukturu nelze delete!



Singleton – dlouhověkost



■ Problémy Fénixe

- ❑ ztráta stavu, uložení, uzavření, ...
- ❑ Nejasnost C++ standardů ohledně funkce atexit()

■ Singleton s dlouhověkostí

- ❑ při vytváření singletonu priorita destrukce
- ❑ KDL – Log bude mít větší dlouhověkost
- ❑ explicitní mechanismus destrukce objektů
 - nelze použít destrukci řízenou kompilátorem - pouze dynamické objekty

```
class SomeSingleton { ... };
class SomeClass { ... };
SomeClass* pGlobalObject(new SomeClass);

template <typename T>
void SetLongevity(T* pDynObj, int longevity);

int main()
{
    SetLongevity(&SomeSingle().Inst(), 5);
    SetLongevity(pGlobalObject, 6);
    ...
}
```

mechanismus by měl fungovat na jakékoliv (*dynamické*) objekty

šablona pro nastavování dlouhověkosti

prioritní fronta na zabití

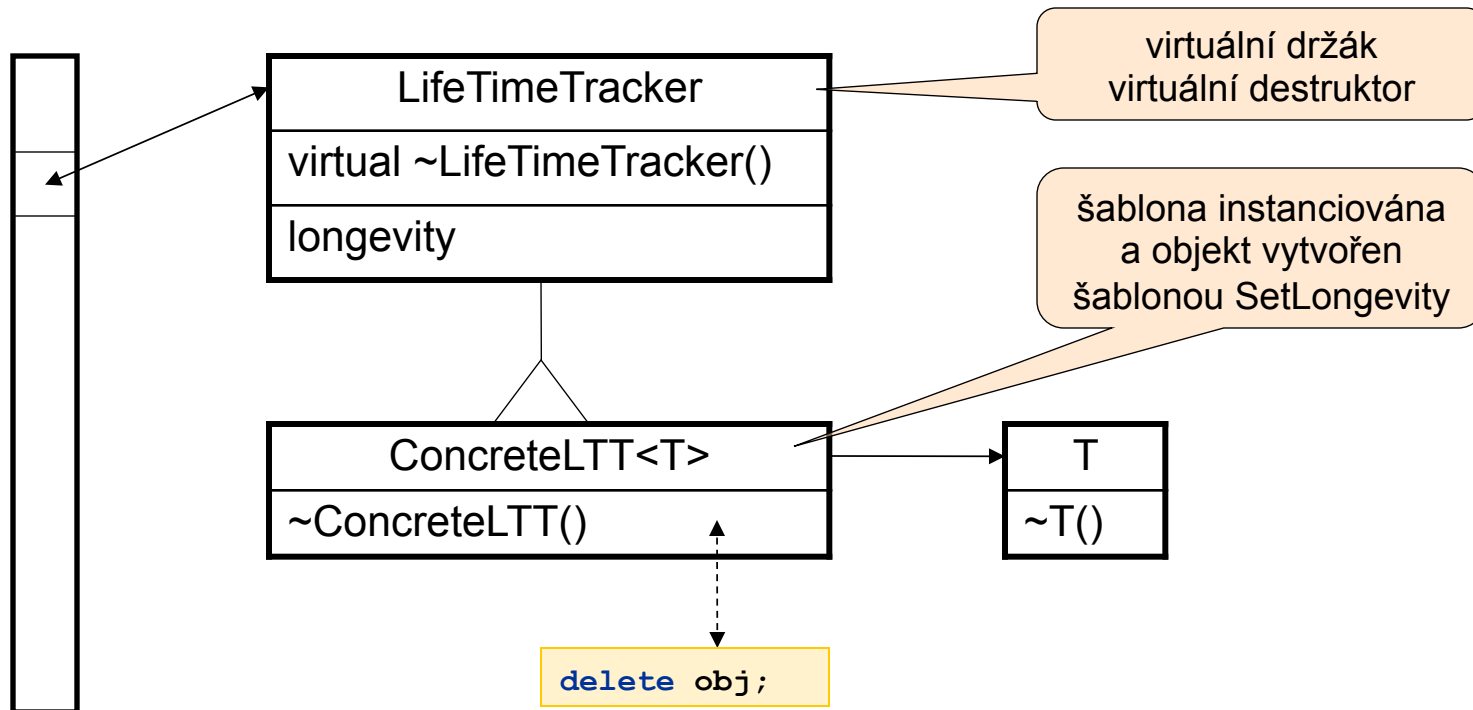
po ukončení programu se objekty destrukují v tomto pořadí



Singleton – implementace dlouhověčnosti

■ Prioritní fronta

- při stejných prioritách se chová jako zásobník
 - C++ pravidlo: dříve inicializované objekty se destruuují později
- čeho to bude fronta?
 - neexistuje společný předek, registrační funkce je šablona
 - ukazatel na abstraktního předka šablon obsahujících ukazatel na objekt





Singleton – implementace dlouhověkosti

```
class LifetimeTracker {
public:
    LifetimeTracker(unsigned int x): longevity(x) {}
    virtual ~LifetimeTracker() = 0;
    friend inline bool Compare(
        unsigned int longevity,
        const LifetimeTracker* p) {
        return p->longevity_ > longevity;
    }
private:
    unsigned int longevity;
};

inline LifetimeTracker::~~LifetimeTracker() {}

typedef LifetimeTracker** TrackerArray;
extern TrackerArray pTA;
extern unsigned int elements;

template <typename T> struct Deleter {
    static void Delete(T* pObj) {
        delete pObj;
    }
};
```

virtuální držák

umí zabíjet...

... a porovnávat stáří

vlastní fronta na zabití

způsob zabití
defaultně delete
lze i free, ...

Bylo dříve vejce nebo slepice?

- TrackerArray se musí chovat jako Singleton
- čímžpádem ale trpí všemi jeho problémy



Singleton – multithreading

■ Vícevláknové prostředí

- samo o sobě není pokryté standardy
 - platformově závislé knihovny

```
Singleton instance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```

sem se dostanou obě vlákna

dva Singleton objekty 💣

■ Ochrana mutexem

```
Singleton instance() {  
    synchronized(this) {  
        if (!instance) {  
            instance = new Singleton();  
        }  
    }  
    return *instance;  
}
```

konstruktor objektu Lock zamkne
destruktor odemkne

- velmi neefektivní – zamykání při každém přístupu



Singleton – Double-Checked Locking

■ Double-Checked Locking Pattern

- Doug Schmidt & Tim Harrison, *C++ Report 1996*

```
Singleton instance() {  
    if (instance == null) {  
        synchronized(this) {  
            if (instance == null) {  
                instance = new Singleton();  
            }  
        }  
    }  
    return instance;  
}
```

test inicializace

sem se může dostat víc vláken

odsud už jen jedno vlákno

zde mi to už nikdo nezmění

■ Už je to konečně správně?

- C++ – není!
- víc vláken – norma nezaručuje vůbec nic
- speciálně multi- a vícejaderné procesory
 - zápisy a čtení různých vláken nemusí být sekvenční
- řešení:
 - platformově závislá synchronizační primitiva
 - memory barriers, ...



Singleton – kategorizace

■ Dělení z hlediska vytvoření

- dynamické (operátor `new`, `malloc`)
- statické (Meyers singleton)

■ Dělení z hlediska životnosti (lifetime)

- Ostrichovo řešení (Leaking singleton)
- životnost určená kompilátorem (Meyers singleton)
- Fénix singleton
- dlouhověkost (singleton with longevity)

■ Dělení z hlediska použití ve vláknech

- jednovláknové – bez synchronizace
- thread safe – s platformově závislou synchronizací



Singleton – obecná implementace

■ Politiky (Policy classes)

- mnoho kombinací možností implementace
- šablona s možností vlastních voleb

```
template <
    class T,
    template <class> class CreationPolicy = CreateUsingNew,
    template <class> class LifetimePolicy = DefaultLifetime,
    template <class> class ThreadingModel = SingleThreaded
> class SingletonHolder {
    ...
};
```

```
class KeyboardImpl { ... };
class LogImpl { ... };

inline unsigned int GetLongevity(KeyboardImpl*) { return 1; }
inline unsigned int GetLongevity(LogImpl*) { return 2; }

typedef SingletonHolder<KeyboardImpl, SingletonWithLongevity> Keyboard;
typedef SingletonHolder<LogImpl, SingletonWithLongevity> Log;
```

■ Alexandrescu: Modern C++ Design

- knihovna Loki (@SourceForge)



Singleton – implementace dědičnosti

```
#include <iostream>

template<typename T> class Base {
public:
    static T &instance() {
        static T inst;
        return inst;
    }
    virtual void echo () = 0;
protected:
    Base() {}
    virtual ~Base() {}
private:
    Base(const Base &);
    Base& operator=(const Base &);
};

class Derived: public Base<Derived> {
    friend class Base<Derived>;
public:
    void echo () { std::cout << "I'm Derived" << std::endl; }
protected:
    Derived() {}
    virtual ~Derived() {}
};

int main(int argc, char **argv) {
    Derived &derived = Derived::instance();
    derived.echo();
}
```

Dědičnost implementována pomocí šablon

Možnost šablonování metody `instance()`, místo třídy

Možnost vytváření abstraktních metod

Třída `Base<Derived>` musí být náš friend (volání konstruktoru ze statické metody `instance()`).

Jednoduché použití



Singleton – implementace C#

Dědičnost implementována pomocí šablon

```
using System;
public abstract class DataProvider {
    static DataProvider instance = makeProvider();

    public static DataProvider Instance {
        get { return instance; }
    }

    // nested classes can call private members
    private DataProvider() {}
    public abstract void Connect();

    static DataProvider makeProvider() {
        // Use Oracle on a Sunday, SQL Server otherwise
        if (DateTime.Now.DayOfWeek==DayOfWeek.Sunday) { }
            return new OracleProvider();
        } else {
            return new SqlServerProvider();
        }
    }
}

// There's no need to make the constructors
// for the nested types non-public, as the classes
// themselves are private to DataProvider.

class OracleProvider : DataProvider {
    public override void Connect() {
        Console.WriteLine ("Connecting to Oracle");
    }
}

class SqlServerProvider : DataProvider {
    public override void Connect() {
        Console.WriteLine ("Connecting to SQL Server);
    }
}
```




Singleton – související návrhové vzory

■ Možné použití

- zjednodušení stavu aplikace - zajištění pouze jedné instance nějakého objektu
- řízení přístupu do DB, tiskových zdrojů...
 - omezení plýtvání zdrojů (lze přidělit existující spojení)
 - možnost regulovat počty, frekvence... přístupů
 - přidělení spojení s vhodnými právy

■ Související NV

- Abstract Factory, Builder, Prototype
 - častá implementace pomocí singletonu
- Facade
 - v případě potřeby pouze jednoho vstup do systému
- State
 - stavové objekty jsou často `Singletony`



Singleton – shrnutí

- **Dělení z hlediska vytvoření**
 - dynamické (operátor `new`, `malloc`)
 - destruktor se nezavolá
 - statické (Meyers singleton)
 - funkce vracějící referenci na statický objekt
- **Životnost (lifetime)**
 - Ostrichovo řešení (Leaking singleton)
 - resource, memory leaks
 - killer
 - obalení ukazatele třídou starající se o destrukci
 - životnost určená kompilátorem (Meyers singleton)
 - destrukce v pořadí podle LIFO
 - detekce mrtvé reference
 - při detekci zrušeného objektu výjimka
 - Fénix singleton
 - po zrušení objektu jeho znovuvytvoření
 - dlouhověkost (singleton with longevity)
 - specifikace pořadí destrukcí
- **Threading model**
 - jednovláknové
 - bez synchronizace
 - Double-checked locking
 - problémy se synchronizací – platformě závislé

pretty simple pattern 😊



Konec

- **Děkujeme za pozornost.**