Singleton

Jakub Mudra, David Jaromír Šebánek

"When discussing which patterns to drop, we found that we still love them all. (Not really—I'm in favor of dropping Singleton. Its use is almost always a design smell.)"

- Erich Gamma

Úvod

- Jedna unikátní instance
- Globální přístupový bod
- Např.: logger, DB spojení, systémové zdroje

Příklad: logger

```cpp
#include <iostream>
#include <string>

class Logger {
private:
    Logger(const Logger&) = delete;
    Logger(Logger&&) noexcept = delete;
    Logger& operator=(const Logger&) = delete;
    Logger& operator=(Logger&&) noexcept = delete;

    Logger() { /* constructor code */ }
    ~Logger() { /* destructor code */ }

    static Logger* instance_;
    /* fields needed for class */
public:
    static Logger& getInstance() {
        if (instance_ == nullptr) {
            instance_ = new Logger();
        }
        return *instance_;
    }
    /* class functionality */
    // E.g.:
    void log(std::string message) { /* log message */ }
};

int main()
{
    Logger::getInstance().log("Hello, World!");
    Logger logger = Logger::getInstance();
    Logger logger2;
    logger2 = logger;
}
```

Struktura singletonu

```cpp
#include <iostream>
#include <string>

class Logger {
private:
    Logger(const Logger&) = delete;
    Logger(Logger&&) noexcept = delete;
    Logger& operator=(const Logger&) = delete;
    Logger& operator=(Logger&&) noexcept = delete;

    Logger() { /* constructor code */ }
    ~Logger() { /* destructor code */ }

    static Logger* instance_;
    /* fields needed for class */
```

Zajištění jediné instance

Odstranění přístupu uživatele k vytváření a destrukci singletonu

Data business logiky a třídou držená jediná instance

Struktura singletonu

Globální přístupový bod & zajištění jedinečnosti

Funkce pro business logiku

Příklad získání a použití singletonu

Vytváří kompilační chyby

```
public:
    static Logger& getInstance() {
        if (instance_ == nullptr) {
            instance_ = new Logger();
        }
        return *instance_;
    }
    /* class functionality */
    // E.g.:
    void log(std::string message) { /* log message */ }
};
```

```
int main()
{
    Logger::getInstance().log("Hello, World!");
    Logger logger = Logger::getInstance();
    Logger logger2;
    logger2 = logger;
}
```

A nešlo by to jinak…?

A nešlo by to jinak…?

- Poprosíme v komentářích

```
// Please instantiate only once or it will break! :(
class Logger { /* class body */ };

int main()
{
    Logger log1;
    Logger log2;
    // Happily ignoring the plea
    log1.log("Hello World!");
    log2.log("Goodbye World!");
}
```

A nešlo by to jinak...?

- Globální proměnná

```
class Logger { /* class body */ };

Logger logger; // global instance of Logger
int main()
{
    logger.log("Hello, World!");
    logger = Logger(); // New instance is possible!
    logger.log("New Hello, World!");
    Logger logger2 = Logger(); // 2 instances possible!
    logger2.log("Hello, World 2!");
}
```

A nešlo by to jinak…?

- Statická třída

```
class Logger { // Cannot inherit static members
private:
    Logger() = delete; // delete default constructor - no instance
    /* static members */
public:
    /* static member functions, e.g.: */
    static void log(std::string message) { /* log message */ }
};

int main()
{
    Logger::log("Hello, World!"); // call static member function
    Logger logger; // error: constructor is deleted
}
```

A nešlo by to jinak...?

- Monostate

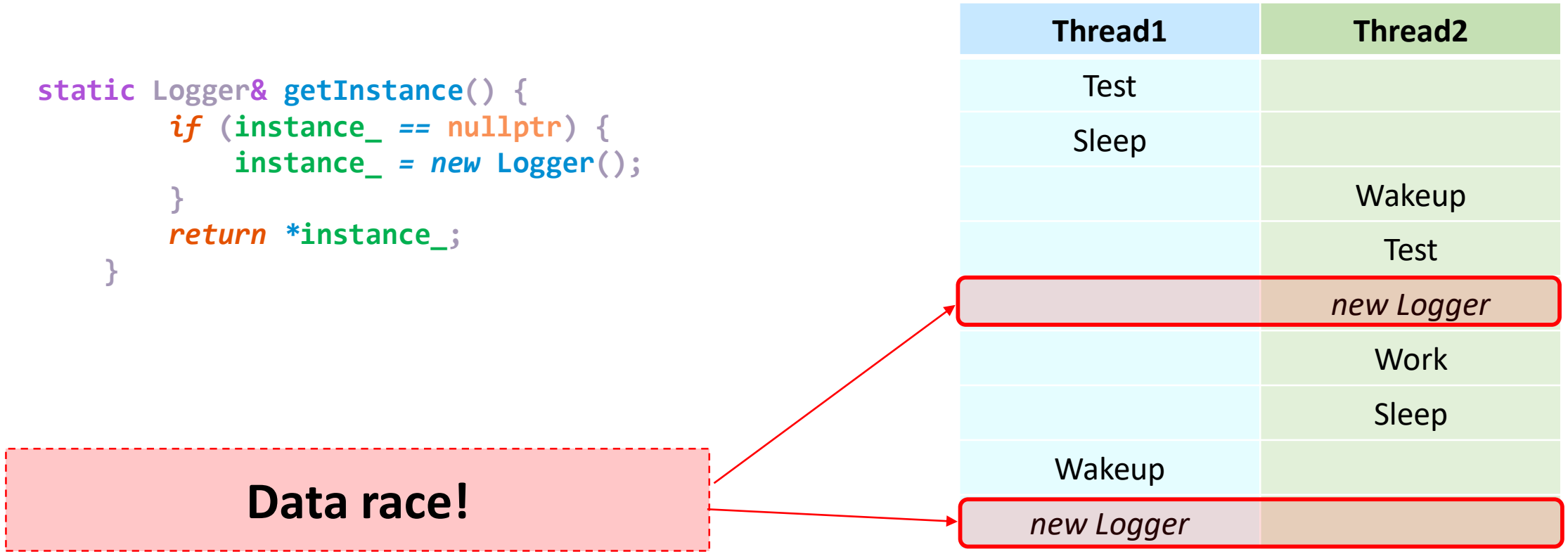
```
class Logger { // Cannot inherit static members
private:
    /* static members */
public:
    /* member functions, e.g.: */
    void log(std::string message) { /* log message */ }
};

int main()
{
    Logger logger;
    logger.log("Hello, World!");
    Logger logger2; // Multiple instances, same static data!
    logger2.log("Hello, World!");
}
```

Singleton vs paralelizace

Singleton vs **paralelizace**

- Konstrukce v multithreaded prostředí

```
static Logger& getInstance() {
        if (instance_ == nullptr) {
            instance_ = new Logger();
        }
        return *instance_;
    }
```



Singleton vs **paralelizace**

- Lock-guard

```
#include <mutex>
#include <string>

class Logger {
private:
    /* delete copy-/move-ctors/assigns */

    Logger() { /* constructor code */ }
    ~Logger() { /* destructor code */ }

    inline static Logger* instance_ = nullptr;
    inline static std::mutex mutex_;
    /* fields needed for class */
...
```

```
...
public:
    static Logger& getInstance() {
        std::lock_guard<std::mutex> lock(mutex_);
        if (instance_ == nullptr) {
            instance_ = new Logger();
        }
        return *instance_;
    }

    /* class functionality */
    // E.g.:
    void log(std::string message) { /* log message */ }
};
```

Singleton vs **paralelizace**

- Double-checked locking

```
#include <mutex>
#include <string>

class Logger {
private:
    /* delete copy-/move-ctors/assigns */

    Logger() { /* constructor code */ }
    ~Logger() { /* destructor code */ }

    inline static std::atomic<Logger*>
        instance_ = nullptr;
    inline static std::mutex mutex_;
    /* fields needed for class */
...
```

```
...
public:
    static Logger& getInstance() {
        if (instance_ == nullptr) {
            std::lock_guard<std::mutex> lock(mutex_);
            if (instance_ == nullptr) {
                instance_ = new Logger();
            }
        }
        return *instance_;
    }
    /* class functionality */
    // E.g.:
    void log(std::string message) { /* log message */ }
};
```

Singleton vs **paralelizace**

- Call_once

```
#include <mutex>
#include <string>

class Logger {
private:
    /* delete copy-/move-ctors/assigns */

    Logger() { /* constructor code */ }
    ~Logger() { /* destructor code */ }

    inline static std::atomic<Logger*>
        instance_ = nullptr;
    inline static std::once_flag flag_;
    /* fields needed for class */
...
```

```
...
public:
    static Logger& getInstance() {
        std::call_once(flag_, [&]() {
            instance_ = new Logger();
        });
        return *instance_;
    }

    /* class functionality */
    // E.g.:
    void log(std::string message) { /* log message */ }
};
```

Singleton vs **paralelizace**

- Meyersův singleton (od C++11)

```
class Logger {
private:
    /* delete copy-/move-ctors/assigns */

    Logger() { /* constructor code */ }
    ~Logger() { /* destructor code */ }

    /* fields needed for class */
public:
    static Logger& getInstance() {
        static Logger instance;
        return instance;
    }
    /* class functionality */
    // E.g.:
    void log(std::string message)
    { /* log message */ }
};
```

```
// Under the hood
static Logger& getInstance() {
    extern void __ConstructLogger(void* memory);
    extern void __DestroyLogger();

    static bool __initialized = false;
    static char __buffer[sizeof(Logger)];

    if (!__initialized) {
        __ConstructLogger(__buffer);
        atexit(__DestroyLogger);
        __initialized = true;
    }
    return *reinterpret_cast<Logger*>(__buffer);
}
```

Singleton vs destrukce

Singleton vs **destrukce**

- Resource leak
  - File handles, OS-mutexy, spojení
- Mrtvý odkaz
  - "Neřeším"
  - Fénix
  - Dlouhověkost

Singleton vs **destrukce**

- Keyboard
- Display
- Log
  - Na chyby při inicializaci a destrukci

Singleton vs **destrukce**

- Detekce mrtvého odkazu

```
class Logger {
private:
    /* delete copy-/move-ctors/assigns */
    Logger() { /* constructor code */ }
    ~Logger() {
        destroyed_ = true;
        instance_ = nullptr;
    }

    static void Create() {
        static Logger instance;
        instance_ = &instance;
    }

    static void OnDeadReference() {
        throw std::runtime_error("Dead reference detected");
    }

    inline static Logger* instance_ = nullptr;
    inline static bool destroyed_ = false;
    /* fields needed for class */
...
```

```
...
public:
    static Logger& getInstance() {
        if (!instance_) {
            // Check for dead reference
            if (destroyed_) {
                OnDeadReference();
            }
            else {
            // First call—initialize
                Create();
            }
        }
        return *instance_;
    }
    /* class functionality */
};
```

Singleton vs **destrukce**

- Mrtvý odkaz - fénix

```
static Logger& getInstance() {
        if (!instance_) {
            // Check for dead reference
            if (destroyed_) {
                OnDeadReference();
            }
            else {
            // First call—initialize
                Create();
            }
        }
        return *instance_;
    }
```

```
static void OnDeadReference() {
        Create();
        new(instance_) Logger;
        atexit(KillPhoenix);
        destroyed_ = false;
    }
```

```
void KillPhoenix() {
        instance_->~Logger();
    }
```

Singleton vs **destrukce**

- Mrtvý odkaz - dlouhověkost: příprava

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

typedef LifetimeTracker** TrackerArray;
extern TrackerArray trackerArray;
extern unsigned int elements;
```

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

template <typename T, typename Destroyer>
class ConcreteLifetimeTracker : public LifetimeTracker {
public:
    ConcreteLifetimeTracker(T* p, unsigned int longevity, Destroyer d)
        : LifetimeTracker(longevity), tracked_(p), destroyer_(d) {}
    ~ConcreteLifetimeTracker() {
        destroyer_(tracked_);
    }
private:
    T* tracked_;
    Destroyer destroyer_;
};
```

Singleton vs **destrukce**

- Mrtvý odkaz - dlouhověkost: příprava

```
static void AtExitFn() {
    LifetimeTracker* top = trackerArray[elements - 1];
    trackerArray = static_cast<TrackerArray>(
        std::realloc(trackerArray, sizeof(LifetimeTracker*) * (--elements))
    );
    delete top; // delete AFTER popping
}
```

Singleton vs **destrukce**

- Mrtvý odkaz - dlouhověkost: příprava

```
template <typename T, typename Destroyer>
void SetLongevity(T* object, unsigned int longevity, Destroyer d) {
    TrackerArray newArray = static_cast<TrackerArray>(
        std::realloc(trackerArray, sizeof(T*) * (elements + 1))
    );
    if (newArray == nullptr) {
        throw std::bad_alloc();
    }
    trackerArray = newArray;

    LifetimeTracker* p
        = new ConcreteLifetimeTracker<T, Destroyer> (
          object, longevity, d);

    TrackerArray pos = std::upper_bound(
        trackerArray, trackerArray + elements,
        longevity, Compare);
...
```

```
...
    std::copy_backward(
        pos,
        trackerArray + elements,
        trackerArray + elements + 1);

    *pos = p;
    ++elements;
    atexit(AtExitFn);
}

template <typename T>
void SetLongevity(T* object, unsigned int longevity) {
    SetLongevity(object, longevity, Deleter<T>::Delete);
}
```

Singleton vs **destrukce**

- Mrtvý odkaz - dlouhověkost

```
class Logger {
private:
    /* delete copy-/move-ctors/assigns */
    Logger() { /* constructor code */ }
    ~Logger() {
        destroyed_ = true;
        instance_ = nullptr;
    }

    static void Create() {
        static Logger instance;
        instance_ = &instance;
        SetLongevity(instance_, longevity_);
    }

    static void OnDeadReference() {
        throw std::runtime_error(
            "Dead reference detected");
    }
...
```

```
...
    inline static Logger* instance_ = nullptr;
    inline static bool destroyed_ = false;
    inline static const unsigned int longevity_ = 2;
    /* fields needed for class */
public:
    static Logger& getInstance() {
        if (!instance_) {
            // Check for dead reference
            if (destroyed_) {
                OnDeadReference();
            }
            else {
            // First call—initialize
                Create();
            }
        }
        return *instance_;
    }
    /* class functionality */
};
```

Singleton vs testování

Singleton vs **testování**

- Vnitřní závislosti v kódu
- Trpí hlavně unit test
  - Mock objekty se musejí vytvářet speciálněji

Singleton vs **testování**

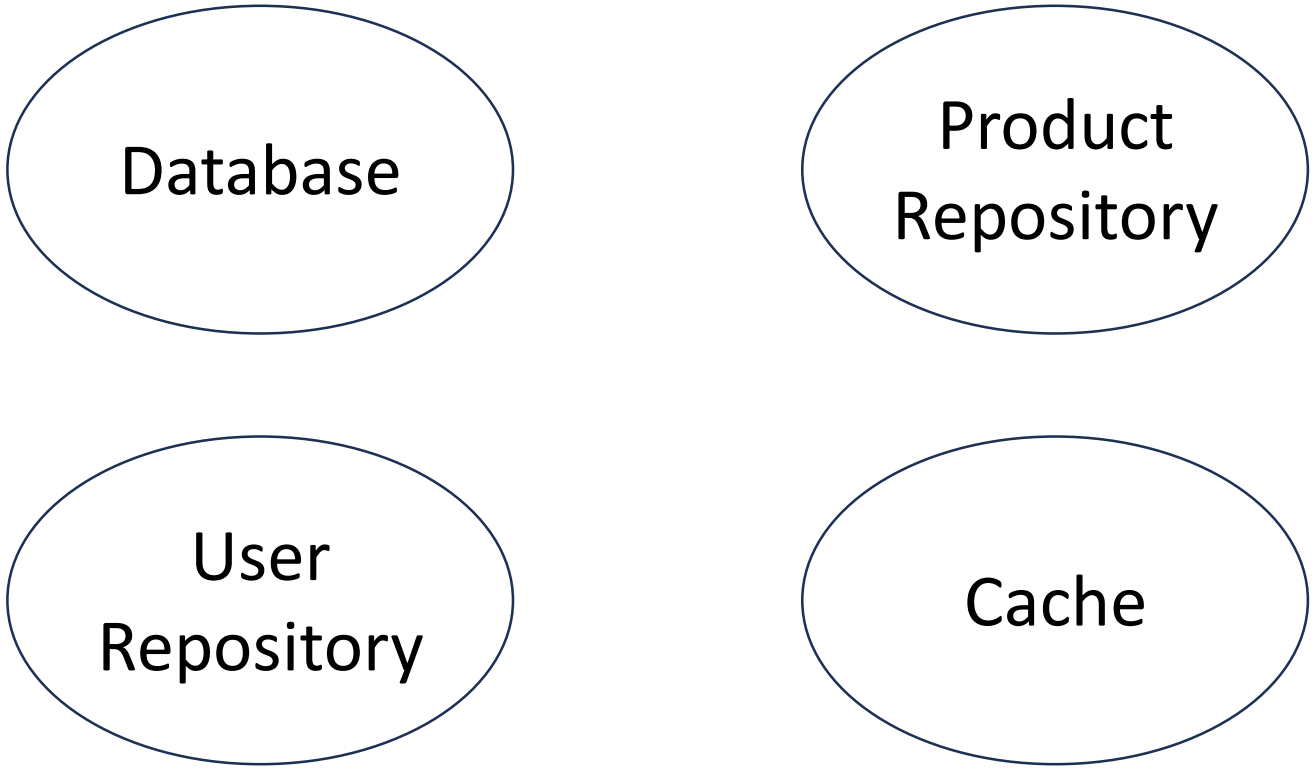
- Graf závislostí (hlavičkový soubor)

```cpp
#include <string>
#include <map>

class Database {
public:
    static Database& getInstance();
    std::string getUserData(const std::string& username);
    std::string getProductData(const std::string& productName);

private:
    Database() = default;
    ~Database() = default;
    /* delete copy-/move-ctors/assigns */
};

class UserRepository {
public:
    std::string getUserInfo(const std::string& username);
};
```

```cpp
class ProductRepository {
public:
    std::string getProductInfo(
        const std::string& productName
    );
};

class Cache {
private:
    std::map<std::string, std::string> cache;

public:
    void add(
        const std::string& key,
        const std::string& value
    );
    std::string get(const std::string& key);
};
```

Singleton vs **testování**

- Graf závislostí (z jen hlaivčkového souboru)



"Veřejná rozhraní tříd nevykazují závislost, je to dobrý OO návrh."

...ale Database je **používaný** *singleton*

Singleton vs **testování**

- Graf závislostí (implementace)

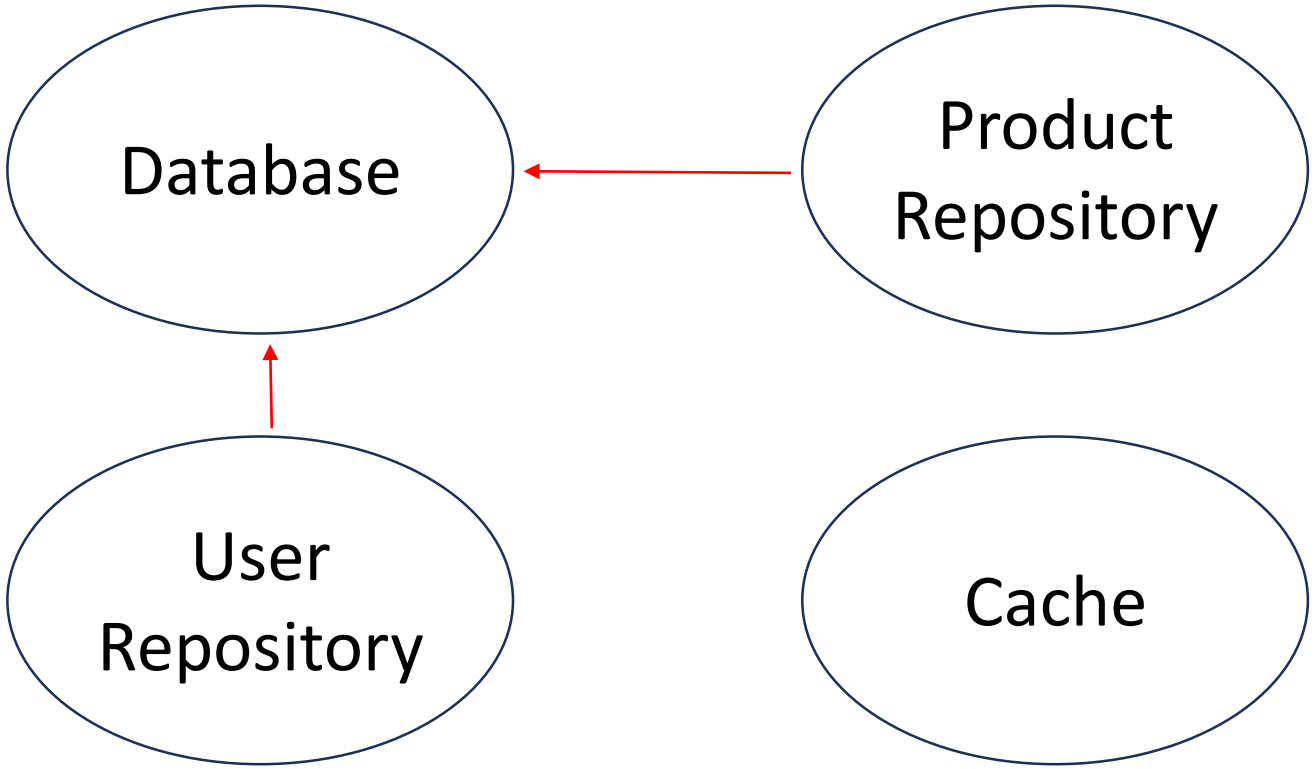
```cpp
Database& Database::getInstance() {
    static Database instance;
    return instance;
}
std::string Database::getUserData(const std::string& username) {
    return "User data for " + username;
}
std::string Database::getProductData(const std::string& productName) {
    return "Product data for " + productName;
}

std::string UserRepository::getUserInfo(const std::string& username) {
    auto data = Database::getInstance().getUserData(username);
    return "Returning user info for " + username + " with data: " + data;
}

std::string ProductRepository::getProductInfo(const std::string& productName) {
    auto data = Database::getInstance().getProductData(productName);
    return "Returning product info for " + productName + " with data: " + data;
}
```

```cpp
void Cache::add(
    const std::string& key,
    const std::string& value) {
    cache[key] = value;
}

// No dependency on Database!
std::string Cache::get(const std::string& key) {
    if (cache.find(key) != cache.end()) {
        return cache[key];
    }
    return "No data found for " + key;
}
```

Schované vnitřní závislosti!

Singleton vs **testování**

- Graf závislostí



*"Objekty na sobě závisí **v kódu**."*

Programátor bude možná muset při změnách či testování zkoumat implementační kód...

...pokud k němu má přístup.

Singleton vs **testování**

```
class Database {
private:
    /* delete copy-/move-ctors/assigns */
    Database() {
        std::ifstream ifs("cities.txt");
        std::string cityEntry, populationEntry;
        while (getline(ifs, cityEntry)) {
            getline(ifs, populationEntry);
            int population = std::stoi(populationEntry);
            cityPopulations[cityEntry] = population;
        }
    }
    ~Database() { /* destructor code */ }

    std::map<std::string, int> cityPopulations;
    /* other db members */
public:
    static Database& get() {
        static Database db;
        return db;
    }
    int getPopulation(const std::string& city) const {
        return cityPopulations.at(city);
    }
    /* other db methods */
};
```

```
class RecordFinder {
public:
    int totalPopulation(const std::vector<std::string>& names) {
        int result = 0;
        for (auto& name : names)
            // Tight coupling!
            result += Database::get().getPopulation(name);
        return result;
    }
};

TEST(RecordFinderTests, SingletonTotalPopulationTest) {
    RecordFinder rf;
    ASSERT_EQ(
        1800000,
        rf.totalPopulation(std::vector<std::string>
            {"Prague", "Bratislava"}
        )
    );
}
```

Nutí testovat i Database → nutí využívat **produkční** databázi!

Singleton vs **testování**

```
class SingletonDatabase : public Database {
private:
    /* delete copy-/move-ctors/assigns */
    SingletonDatabase() {
        std::ifstream ifs("cities.txt");
        std::string cityEntry, populationEntry;
        while (getline(ifs, cityEntry)) {
            getline(ifs, populationEntry);
            int population = std::stoi(populationEntry);
            cityPopulations[cityEntry] = population;
        }
    }
    ~SingletonDatabase() { /* destructor code */ }
    std::map<std::string, int> cityPopulations;
public:
    static SingletonDatabase& get() {
        static SingletonDatabase db;
        return db;
    }
    int getPopulation(const std::string& city) const override
 {
        return cityPopulations.at(city);
  }
};
```

```
class Database {
public:
    virtual int getPopulation(
        const std::string& city) const = 0;
};
```

Singleton vs **testování**

```
class DummyDatabase : public Database {
    std::map<std::string, int> cityPopulations;
public:
    DummyDatabase() {
        cityPopulations["alpha"] = 1;
        cityPopulations["beta"] = 2;
        cityPopulations["gamma"] = 3;
    }
    int getPopulation(const std::string& city) const override {
        return cityPopulations.at(city);
    }
};

class RecordFinder {
    Database& db;
public:
    RecordFinder(Database& db) : db(db) {}
    int totalPopulation(const std::vector<std::string>& names) {
        int result = 0;
        for (auto& name : names)
            result += db.getPopulation(name);
        return result;
    }
};
```

```
TEST(RecordFinderTests, TotalPopulationTest) {
    DummyDatabase db {};
    RecordFinder rf {db};
    ASSERT_EQ(
        3,
        rf.totalPopulation(
            std::vector<std::string>
                {"alpha", "beta"}
        )
    );
}
```

Závěr

Výhody Singletonu

- Čistší globální namespace
- Zajištění unikátnosti
- Lazy initialization (většinou)
- Globální přístup

Nevýhody Singletonu

- Globální přístup :
  - Závislosti schované v kódu programu
  - Komplikace v multithreaded prostředí
- Komplikace při testování
- Porušuje *Single Responsibility Principle*
- Dnes neoblíbený
  - Dokonce přiřazován k anti-patterns

Související vzory

- *Abstract Factory, Builder, Prototype*
- *Facade*
- *Monostate*

Děkujeme za vaši pozornost!

GitLab s kódy, materiály, poznámkami

Reference

[1] A. Alexandrescu, *Modern C++ design: Generic programming and design patterns applied*. Boston, MA: Addison Wesley, 2001.

[2] *Design patterns: Elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[3] D. Nesteruk, *Design patterns in modern C++: Reusable approaches for object-oriented software design*, 1st ed. New York, NY: APRESS, 2018.

[4] E. Freeman, E. Robson, E. Freeman, K. Sierra, and B. Bates, *Head first design patterns*. "O'Reilly Media, Inc.," 2004.

[5] "Singleton," *Refactoring.guru*, 01-Jan-2023. [Online]. Available: https://refactoring.guru/design-patterns/singleton. [Accessed: 05-Mar-2024].

[6] N. Byers, "The Singleton pattern: Pros, cons, and best practices," *Medium*, 14-Jan-2023. [Online]. Available: https://medium.com/@nathanbyers13/the-singleton-pattern-pros-cons-and-best-practices-9e4d256132de. [Accessed: 05-Mar-2024].

[7] D. Soni, "What is a singleton?," *Better Programming*, 31-Jul-2019. [Online]. Available: https://betterprogramming.pub/what-is-a-singleton-2dc38ca08e92. [Accessed: 05-Mar-2024].

[8] S. Yegge, "Singleton considered stupid," *Google.com*, 03-Sep-2004. [Online]. Available: https://sites.google.com/site/steveyegge2/singleton-considered-stupid. [Accessed: 05-Mar-2024].

[9] "Why Singletons Are Controversial," *Archive.org*, 06-Feb-2016. [Online]. Available: https://web.archive.org/web/20210506162753/https://code.google.com/archive/p/google-singleton-detector/wikis/WhySingletonsAreControversial.wiki. [Accessed: 05-Mar-2024].

[10] "Why Singletons Are Evil," *Archive.org*, 25-May-2004. [Online]. Available: https://web.archive.org/web/20210715184717/https://docs.microsoft.com/en-us/archive/blogs/scottdensmore/why-singletons-are-evil. [Accessed: 05-Mar-2024].

[11] J. B. Rainsberger, "Use your singletons wisely," *Archive.org*, 01-Jul-2001. [Online]. Available: https://web.archive.org/web/20210224180356/https://www.ibm.com/developerworks/library/co-single/. [Accessed: 05-Mar-2024].