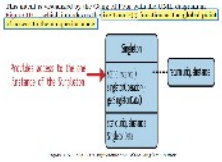
10-Singleton Pattern

segunda-feira, 14 de abril de 2025 18:22

Guideline 37: TREAT SINGLETON AN IMPLEMENTATION PATTERN, NOT A DESIGN PATTERN

- Bad reputation because of its the global nature
  - Solution to represent the few global aspects in our code
- Use as implementation pattern rather than design pattern
  - Will be discussed
  - It's not a design patter because it doesn't have the properties of a design pattern
- Usual problems,
  - global state
  - many, strong and artificial dependencies
  - impeded changeability and testibility



The Singleton Pattern Explained

- garantee that only exists one instance of a particular class
- might make sense for database access, logger clocks, configurations or any class that should not be instantiated multiple times because it represents something that exists only once
- Ensure a class has only one instance, and provide a global point of access to it

Meyers' Singleton

- a way to reresent and limit the number of objects of a class
- Defin the constructor and all assigned operations in as private methods and delete the operators. Not allowing overloads
- The only way to access the single instance is throug the public static instance() function
  The instance() function is implemented in terms of a static local variable. This means that the first time control passes through the declaration, the variable is initialized in a thread-safe way, and on all further calls the initialization is skipped. the function returns a reference to the static local variable

Singleton is not a design pattern.

- Design pattern has name; carries na intent; introduces na abstraction; has been proven
  - Most of the things are true, but there is no kind of abstraction: no base class, no template parameters, nothing!!!
  - Singleton is focused on restricting the number of instantiations to exactly one. Thus, Singleton is not a design pattern but merely na implementation pattern.
- Reasons it's listed as design patter
  - There are languages that every class can be considered na abstraction, it's very commonly used, and they are in the process of understanding software design and design patterns
- So... we don't use singletons to decouple software entities. It's only dealing with implementation details. Aas such, we should treat it as impementaton pattern

GUIDELINE 37: TREAT SINGLETON AN IMPLEMENTATION PATTERN, NOT A DESIGN PATTERN

- The goal of Singleton is not to decouple or manage dependencies, and thus it does not fulfill the expectations of a design pattern.
- Apply the Singleton pattern with the intent to restrict the number of instances of a particular class to exactly one.

Guideline 38: Design Singletons for Change and Testability

A lot of people says to avoid singletons....

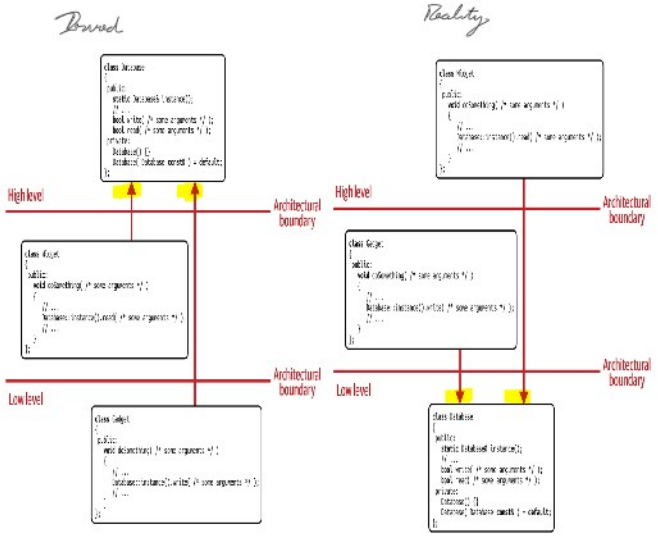
- It often causes artificial dependencies and obstructs testability
- But still is often used
  - Sometimes, objects MUST exists only once and be acceble in other parts of the code
  - There are global aspects to represent
- Let's focus on these usage and design it for change and testability

Singletons Represent Global State

- Represents something that must exists only once and be acceble from multiple palces
  - representing global available functionality or data
- Global variables are a bad idea
  - the term variable represents mutability, and in a global scope, there is none
  - hard to read about because a lot of places chan chage it (bad for maintainability)
  - ...
  - Avoid asmuch as possible
- Singletons taht provide a unidirectional data flow to or from some global state are acceptable
  - Only allow to write data, not to readly. Or the other way around
  - This restrictions helps wih alot of the problems of global variables

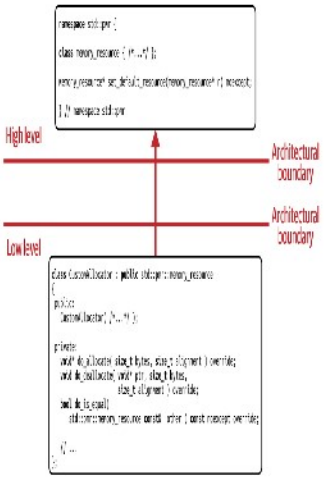
Singletons Impede Changeability and Testability

- Functions that uses singletons depends on the represented global data
  - harder to change and harder to test
- Desired - Since they use the Database and thus depend on it, we would like them to reside in architecture levels below the level of the Database Singleton. That is because, as you remember from "Guideline 2: Design for Change", we can call it a proper architecture only if all dependency arrows run toward the high levels
- Reality - Desirable but unfortunately it is only na illusion: the Database class is not an abstraction but a concrete implementation. Dpends on a very spesific database!!! All dependency arrows point toward the lower level. In other words, right now there is no software architecture!
- Diabase is a concrete class [] and not na abstraction - class that can be instantiated and create objects - Because of that there are several dependences from all over the code to specify implementation details
- This maiinclude dependencies related to the concrete class that will manifest in many different places of the code nd, later maes changes harder or even impossible (affects maintainability)
- This also affects the changes on the tests.. because all test will depends on the singleton as well. In this case we always will need to prowed a database class to the methods to be valid. With no way to replace with a tuck or mock
- To make it more changable and testable, we need to add na abstraction. A real one, since singltons are not



Inverting the Dependencies on a Singleton

- Represents one kind of global state
- In other words, this function provides you with the ability to inject the dependency on this allocator.
  - Thinks on the strategy design pattern
    - **Encapsulate interchangeable behaviors or algorithms** and allow a context class to use there without knowing their concrete types.
    - Represents na abstraction of all possible concrete implementations reside on the lower level of the architecture and the abstraction in the higher level.
    - The Strategy pattern causes na inversion of dependencies, the main logic depends on na abstraction instead of the concrete implementaion details



2. Concrete Strategies

```cpp
#include <iostream>
#include <fstream>

class FileLogStrategy : public ILogStrategy {
public:
    void log(const std::string& message) override {
        std::ofstream file("log.txt", std::ios::app);
        file << message << std::endl;
    }
};

class ConsoleLogStrategy : public ILogStrategy {
public:
    void log(const std::string& message) override {
        std::cout << message << std::endl;
    }
};
```

Call an interface that is a base class for 2 types of concrete classes!

1. Strategy Interface

```cpp
class ILogStrategy {
public:
    virtual ~ILogStrategy() = default;
    virtual void log(const std::string& message) = 0;
};
```

3. Context Class (Logger)

```cpp
class Logger {
public:
    Logger(std::unique_ptr<ILogStrategy> strategy)
        : m_strategy(std::move(strategy)) {}

    void setStrategy(std::unique_ptr<ILogStrategy> strategy) {
        m_strategy = std::move(strategy);
    }

    void log(const std::string& message) {
        if (m_strategy) {
            m_strategy->log(message);
        }
    }

private:
    std::unique_ptr<ILogStrategy> m_strategy;
};
```

Pointer to the base interface class

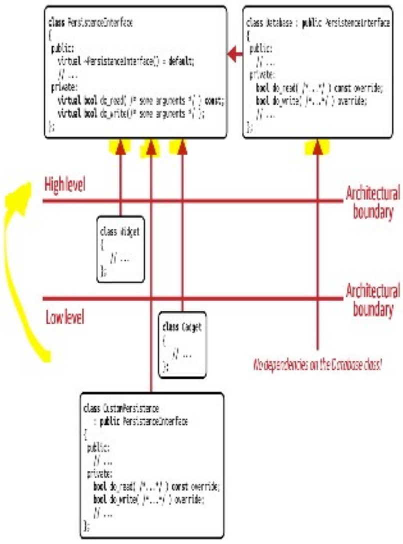
4. Usage Example

```cpp
int main() {
    // Start with console logging
    Logger logger(std::make_unique<ConsoleLogStrategy>());
    logger.log("Logging to console");

    // Switch to file logging
    logger.setStrategy(std::make_unique<FileLogStrategy>());
    logger.log("Logging to file");

    return 0;
}
```

```
    // ...
};
```

Applying the Strategy Design Pattern

- The foal is to make the concrete classes as impementation detail and not the center o the dependencies
- Apply th strategy design patter to remove the dependenceis and remove the database dependency from the high level of the architecture. Introducing na abstraction
- This would make it easier to test and intodauce new concrete classes that peforms the same actions
- introduce the following PersistenceInterface abstraction (5) which provides the interface for all possible concrete implementations.
    - Introduces a read and write functions (public scope) and the implementation | int he private scope
        - The public just call the right implementation
    - Introduces a small overhead but make the adding functionality process easier
    - The cconcrete class now inheris from the abstraction and implement the required set of functions
    - Since i have to inherit from na interface class, this is na intrusive solution



Moving Toward Local Dependency Injection

- Since we added na abstraction to the higher lelvel of the architecture, we now can discuss some options
- While we still provide the previous doSomething() function, we now additionally provide an overload that accepts a PersistenceInterface as a function argument. The second function does all the work, whereas the first function now merely acts as a wrapper, which injects the globally set PersistenceInterface. In this combination, it's possible to make local decisions and to locally inject the desired dependency, but at the same time it is not necessary to pass the dependency through many layers of function calls.
- It might not be the right approach for every single Singleton problem. So don't believe that this is the only possible solution. After all, it depends. However, it is a great example of the general process of software design: identify the aspect that changes or causes dependencies, then separate concerns by extracting a fitting abstraction. Depending on your intent, you will just have applied adesign pattern.

GUIDELINE 38: DESIGN SINGLETONS FOR CHANGE AND TESTABILITY

- Be aware that Singleton represents global state, with all its flaws.
- Avoid global state as much as possible.
- Use Singleton judiciously and just for the few global aspects in your code.
- Prefer Singletons with unidirectional data flow.
- Use the Strategy design pattern to invert dependencies on your Singleton to remove the usual impediments to changeability and testability.

```
        m_strategy->log(message);
    }
}

private:
    std::unique_ptr<ILogStrategy> m_strategy;
};
```

→ Pointer to the base interface class

```
    // Switch to file logging
    logger.setStrategy(std::make_unique<FileLogStrategy>());
    logger.log("Logging to file");

    return 0;
}
```

→ Good to change at runtime or template method interface to change behaviour based on file type

```
//---- <PersistenceInterface.h> ----------------

class PersistenceInterface ❺
{
```

```
 public:
   virtual ~PersistenceInterface() = default;

   bool read( /*some arguments*/ ) const ❻
   {
      return do_read( /*...*/ );
   }
   bool write( /*some arguments*/ ) ❼
   {
      return do_write( /*...*/ );
   }

   // ... More database specific functionality

 private:
   virtual bool do_read( /*some arguments*/ ) const = 0; ❻
   virtual bool do_write( /*some arguments*/ ) = 0; ❼
};

PersistenceInterface* get_persistence_interface(); ❽
void set_persistence_interface( PersistenceInterface* persistence ); ❾

// Declaration of the one 'instance' variable
extern PersistenceInterface* instance; ❿
```