



# Lesson #5

## Pattern “Singleton”

## Порождающие паттерны

*Порождающие паттерны* проектирования абстрагируют процесс создания экземпляров. Они позволяют сделать систему независимой от способа создания, композиции и представления объектов.

Для порождающих паттернов характерны два аспекта:

- 1 - инкапсулируют знания о конкретных классах, которые применяются в системе;
- 2 - скрывают подробности создания и компоновки экземпляров этих классов.

## Порождающие паттерны

Единственная информация об объектах, известная системе, – это их интерфейсы, определенные с помощью абстрактных классов.

Следовательно, порождающие паттерны обеспечивают большую гибкость в отношении того, что создается, кто это создает, как и когда.

Это позволяет настроить систему «готовыми» объектами с самой различной структурой и функциональностью статически (на этапе компиляции) или динамически (во время выполнения).

# Паттерн «Singleton» (Одиночка)

## Название и классификация паттерна

Одиночка – паттерн, порождающий объекты.

## Назначение

Гарантирует, что у класса существует только один экземпляр, и предоставляет к нему глобальную точку доступа.

## Мотивация

Для некоторых классов важно, чтобы существовал только один экземпляр.

**Конфигурационные настройки.** Представим, что у нас есть класс с настройками приложения – параметрами базы данных или внешнего вида интерфейса. Имеет смысл реализовать его как Singleton. Это обеспечит одну точку доступа к настройкам, и весь код сможет ссылаться на одни и те же настройки.

## Паттерн «Singleton» (Одиночка)

**Подключение к базе данных.** Если наше приложение использует базу данных, Singleton гарантированно создаст только один экземпляр класса, отвечающего за подключение к ней. Так мы предотвратим лишние соединения и упростим подключение в целом.

**Логирование.** Singleton удобно использовать для логов. Вместо создания нового логгера каждый раз, когда нужно что-то залогировать, мы записываем всё в один объект.

**Счётчики и глобальные объекты.** Паттерн «одиночка» подходит для оценки состояния приложения или сбора статистики с его модулей. С классом можно будет взаимодействовать из любой части программы.

**Пул ресурсов.** Если у нас ограниченный пул соединений к внешнему сервису или к другим ресурсам, то Singleton гарантирует, что доступ к ним всегда будет идти через единственный экземпляр.

## Паттерн «Singleton» (Одиночка)

### Мотивация (*продолжение*)

Как гарантировать, что у класса есть единственный экземпляр и что этот экземпляр легко доступен? Глобальная переменная дает доступ к объекту, но не запрещает создать несколько экземпляров класса. Более удачное решение – возложить на сам класс ответственность за то, что у него существует только один экземпляр. Класс может запретить создание дополнительных экземпляров, перехватывая запросы на создание новых объектов, и он же способен предоставить доступ к своему экземпляру. Это и есть назначение паттерна одиночка.

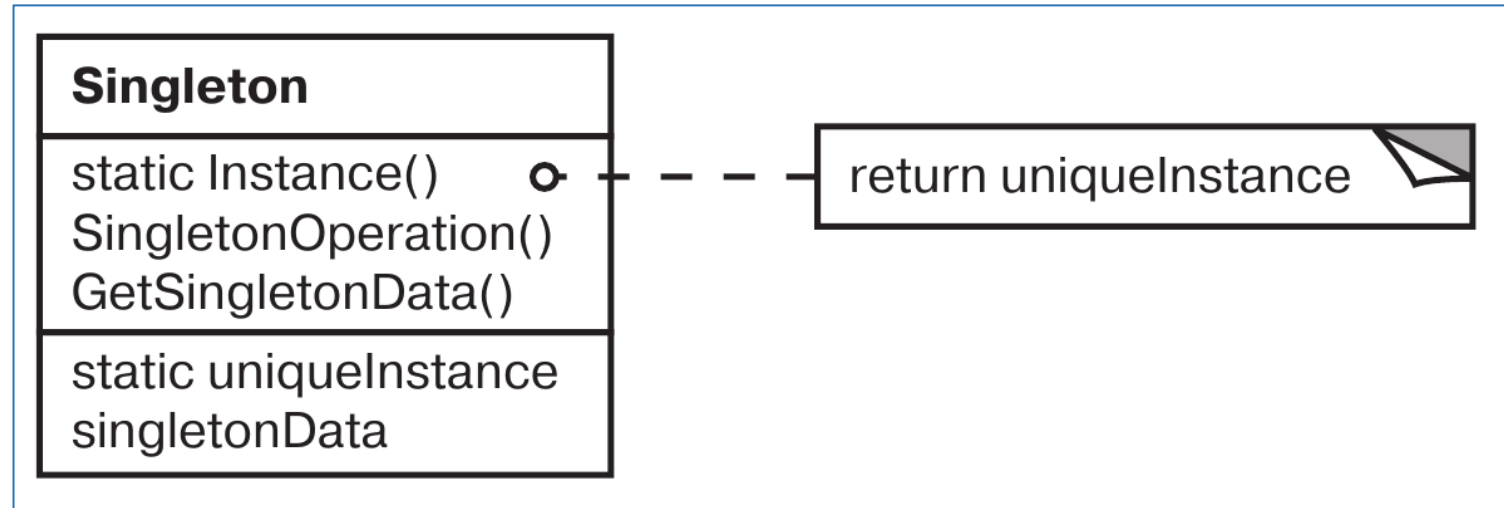
# Паттерн «Singleton» (Одиночка)

## Применимость

- должен существовать ровно один экземпляр некоторого класса, к которому может обратиться любой клиент через известную точку доступа;
- единственный экземпляр должен расширяться путем порождения подклассов, а клиенты должны иметь возможность работать с расширенным экземпляром без модификации своего кода.

# Паттерн «Singleton» (Одиночка)

## Структура



## Участники

- *Singleton* – одиночка
  - определяет операцию *Instance*, которая позволяет клиентам получить доступ к единственному экземпляру. *Instance* – это операция класс
  - может нести ответственность за создание собственного уникального экземпляра.



# Паттерн «Singleton» (Одиночка)

## Результаты

Паттерн обладает рядом достоинств:

- *контролируемый доступ к единственному экземпляру*. Поскольку класс Singleton инкапсулирует свой единственный экземпляр, он полностью контролирует то, как и когда клиенты получают доступ к нему;
- *сокращение пространства имен*. Паттерн одиночка – шаг вперед по сравнению с глобальными переменными. Он позволяет избежать засорения пространства имен глобальными переменными, в которых хранятся уникальные экземпляры;

## Паттерн «Singleton» (Одиночка)

### Результаты (продолжение)

- *возможность уточнения операций и представления.* От класса Singleton можно породить подклассы, а приложение легко настраивается экземпляром расширенного класса. Приложение можно настроить экземпляром нужного класса во время выполнения;

- *возможность использования переменного числа экземпляров.*

Паттерн позволяет легко изменить решение и разрешить появление более одного экземпляра класса Singleton. Более того, тот же подход может использоваться для управления числом экземпляров, используемых в приложении. Изменить нужно будет лишь операцию, дающую доступ к экземпляру класса Singleton;

## Паттерн «Singleton» (Одиночка)

### Результаты (продолжение)

- *большая гибкость, чем у операций класса.* Другой способ реализации функциональности одиночки – использование операций класса, то есть статических функций класса в C++ и методов класса в Smalltalk. Но оба этих приема препятствуют изменению дизайна, если потребуются разрешить наличие нескольких экземпляров класса. Кроме того, статические функции классов в C++ не могут быть виртуальными, что делает невозможной их полиморфную замену в подклассах.

# Паттерн «Singleton» (Одиночка)

## Реализация

Необходимо обеспечить гарантию существованию единственного экземпляра

PATTERN "SINGLETON"

```
1  #pragma once
2  class Settings
3  {
4  public:
5      static Settings* Instance();
6  protected:
7      Settings();
8  private:
9      static Settings* _pInstance;
10 }
```

```
1  #include "Settings.h"
2
3  Settings* Settings::_pInstance = nullptr;
4
5  Settings::Settings()
6  {
7      _pInstance = nullptr;
8  }
9
10 Settings* Settings::Instance()
11 {
12     if (_pInstance == nullptr)
13         _pInstance = new Settings();
14
15     return _pInstance;
16 }
```

# Паттерн «Singleton» (Одиночка)

## Реализация (расширенная версия)

### PATTERN "SINGLETON"

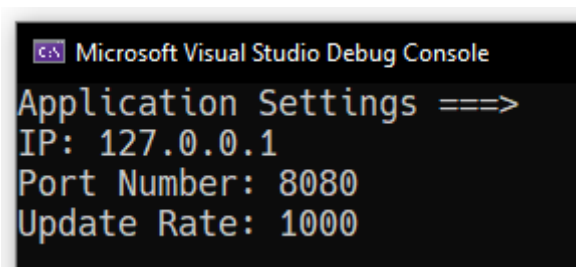
```
1  #pragma once
2  #include <string>
3  class Settings
4  {
5  public:
6      static Settings* Instance();
7      int getUpdateRate();
8      std::string getIP();
9      int getPortNumber();
10 protected:
11     Settings();
12 private:
13     static Settings* _pInstance;
14     int _updateRate;
15     std::string _IP;
16     int _portNumber;
17 };
```

```
1  #include "Settings.h"
2
3  Settings* Settings::_pInstance = nullptr;
4
5  Settings* Settings::Instance()
6  {
7      if (_pInstance == nullptr)
8          _pInstance = new Settings();
9
10     return _pInstance;
11 }
12 Settings::Settings()
13 {
14     _pInstance = nullptr;
15
16     _updateRate = 1000;
17     _IP = "127.0.0.1";
18     _portNumber = 8080;
19 }
20
21 int Settings::getUpdateRate()
22 {
23     return _updateRate;
24 }
25 std::string Settings::getIP()
26 {
27     return _IP;
28 }
29 int Settings::getPortNumber()
30 {
31     return _portNumber;
32 }
```

# Паттерн «Singleton» (Одиночка)

## Реализация (расширенная версия)

```
1  ✓ #include <iostream>
2  [ #include "Settings.h"
3
4  ✓ int main()
5  {
6      std::cout << "Application Settings ==>" << std::endl;
7      std::cout << "IP: " << Settings::Instance()->getIP() << std::endl;
8      std::cout << "Port Number: " << Settings::Instance()->getPortNumber() << std::endl;
9      std::cout << "Update Rate: " << Settings::Instance()->getUpdateRate() << std::endl;
10 }
```



Microsoft Visual Studio Debug Console

```
Application Settings ==>
IP: 127.0.0.1
Port Number: 8080
Update Rate: 1000
```

# Паттерн «Singleton» (Одиночка)

## Реализация (расширенная версия)

PATTERN "SINGLETON"

```
1  √ #include <iostream>
2  | #include "Settings.h"
3
4  √ int main()
5  | {
6      std::cout << "Application Settings ==>" << std::endl;
7      std::cout << "IP: " << Settings::Instance()->getIP() << std::endl;
8      std::cout << "Port Number: " << Settings::Instance()->getPortNumber()
9      std::cout << "Update Rate: " << Settings::Instance()->getUpdateRate()
10
11  Settings *sett = new Settings();|
12  }
```

Problem Details

✖ C2248 'Settings::Settings': cannot access protected member declared in class 'Settings'  
lesson-05.cpp (Line 11)

see declaration of 'Settings::Settings' Settings.h (Line 11)

see declaration of 'Settings' Settings.h (Line 3)

# Паттерн «Singleton» (Одиночка)

## Реализация

(с помощью умных указателей)

PATTERN "SINGLETON"

```

1  #pragma once
2  #include <string>
3  #include <memory>
4  class Settings
5  {
6  public:
7      static std::shared_ptr<Settings> &Instance();
8      int getUpdateRate();
9      std::string getIP();
10     int getPortNumber();
11 protected:
12     Settings();
13 private:
14     static std::shared_ptr<Settings> _pInstance;
15     int _updateRate;
16     std::string _IP;
17     int _portNumber;
18 };

```

```

1  #include "Settings.h"
2
3  std::shared_ptr<Settings> Settings::_pInstance = nullptr;
4
5  std::shared_ptr<Settings> &Settings::Instance()
6  {
7      if (_pInstance == nullptr)
8          _pInstance = std::shared_ptr<Settings>(new Settings());
9
10     return _pInstance;
11 }
12 Settings::Settings()
13 {
14     _pInstance = nullptr;
15
16     _updateRate = 1000;
17     _IP = "127.0.0.1";
18     _portNumber = 8080;
19 }
20
21 int Settings::getUpdateRate()
22 {
23     return _updateRate;
24 }
25 std::string Settings::getIP()
26 {
27     return _IP;
28 }
29 int Settings::getPortNumber()
30 {
31     return _portNumber;
32 }
33

```



# Паттерн «Singleton» (Одиночка)

## Реализация

(с помощью умных указателей)

PATTERN "SINGLETON"

```
1  ~#include <iostream>
2  ~#include "Settings.h"
3
4  ~int main()
5  ~{
6      std::shared_ptr<Settings>& settings = Settings::Instance();
7      std::cout << "Application Settings ==>" << std::endl;
8      std::cout << "IP: " << settings->getIP() << std::endl;
9      std::cout << "Port Number: " << settings->getPortNumber() << std::endl;
10     std::cout << "Update Rate: " << settings->getUpdateRate() << std::endl;
11 }
```

```
Microsoft Visual Studio Debug Console
Application Settings ==>
IP: 127.0.0.1
Port Number: 8080
Update Rate: 1000

D:\SourceCode\design-patterns\x64\Debug\lesson-05.exe (process 29112) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->
le when debugging stops.
Press any key to close this window . . .
```

# Паттерн «Singleton» (Одиночка)

## Реализация

(с помощью умных указателей)

PATTERN "SINGLETON"

```
1  #pragma once
2  #include <string>
3  #include <memory>
4  class Settings
5  {
6  public:
7      static std::shared_ptr<Settings> &Instance();
8      int getUpdateRate();
9      std::string getIP();
10     int getPortNumber();
11     void setIP(std::string ip);
12 protected:
13     Settings();
14 private:
15     static std::shared_ptr<Settings> _pInstance;
16     int _updateRate;
17     std::string _IP;
18     int _portNumber;
19 };
```

```
25  std::string Settings::getIP()
26  {
27      return _IP;
28  }
29  void Settings::setIP(std::string ip)
30  {
31      _IP = ip;
32  }
```

Использование нескольких одиночек

<http://cpp-reference.ru/patterns/creational-patterns/singleton/>

# Паттерн «Singleton» (Одиночка)

## Реализация

(с помощью умных указателей)

PATTERN "SINGLETON"

```
4  int main()
5  {
6      std::shared_ptr<Settings>& settings = Settings::Instance();
7      std::cout << "Application Settings ===>" << std::endl;
8      std::cout << "IP: " << settings->getIP() << std::endl;
9      std::cout << "Port Number: " << settings->getPortNumber() << std::endl;
10     std::cout << "Update Rate: " << settings->getUpdateRate() << std::endl;
11
12     std::shared_ptr<Settings>& settingsA = Settings::Instance();
13     settings->setIP("192.168.1.1");
14     std::cout << "IP: " << settings->getIP() << std::endl;
15     std::cout << "IP: " << settingsA->getIP() << std::endl;
16 }
```

```
Microsoft Visual Studio Debug Console
Application Settings ===>
IP: 127.0.0.1
Port Number: 8080
Update Rate: 1000
IP: 192.168.1.1
IP: 192.168.1.1

D:\SourceCode\design-patterns\x64\Debug\lesson-05.exe (process 17180) exited with code 0.
```

# Паттерн «Singleton» (Одиночка)

## Результат

### *Достоинства*

- класс сам контролирует процесс создания единственного экземпляра;
- паттерн легко адаптировать для создания нужного числа экземпляров;
- возможность создания объектов классов, производных от Singleton.

### *Недостатки*

- в случае использования нескольких взаимозависимых одиночек их реализация может резко усложниться;