Паттерн "Одиночка" (Порождающий паттерн)

Общее представление

Одиночка — это порождающий паттерн проектирования, который гарантирует, что у класса есть всего один единственный экземпляр, и предоставляет к нему глобальную точку доступа.

Проще говоря, он запрещает создавать новые объекты класса, а заставляет использовать один и тот же, созданный единожды.

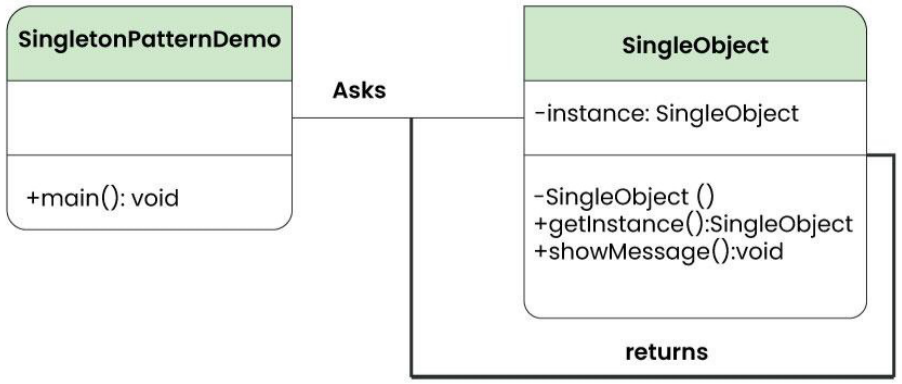
Какую проблему решает?

Гарантирует наличие единственного экземпляра класса. Это полезно, когда в программе должен быть единственный объект, отвечающий за какой-то общий ресурс (например, подключение к базе данных, файл конфигурации, логгер, кэш).

Предоставляет глобальный доступ к этому экземпляру. Хотя глобальные переменные — это плохо, Одиночка предоставляет контролируемый доступ, а не любую возможность изменить объект откуда угодно.

Одиночка: структура

UML диагамма паттерна "Одиночка"



Данная реализация обеспечивает создание единственного экземпляра класса с ленивой* инициализацией. Класс предоставляет глобальную точку доступа через статический метод getInstance() и запрещает прямое создание объектов.

Ленивая инициализация — это создание объекта не при запуске программы, а в момент первого обращения к нему.

Описание класса и реализация

```
class SingleObject {
private:
   SingleObject() {
      std::cout << "SingleObject instance created!" <<
std::endl;
   }
   SingleObject(const SingleObject&) = delete;
   SingleObject& operator=(const SingleObject&) = delete;
   static SingleObject* instance;
public:
   static SingleObject* getInstance() {
      if (instance == nullptr) {
         instance = new SingleObject();
      }
      return instance;
   }
   static void destroyInstance() {
      if (instance != nullptr) {
         delete instance;
         instance = nullptr;
      }
   }
};
SingleObject* SingleObject::instance = nullptr;
```

Одиночка: плюсы и минусы паттерна

✓ **Плюсы**:
Гарантирует наличие единственного экземпляра класса.
Предоставляет удобный глобальный доступ.
Объект создается только в момент первого обращения (ленивая инициализация).

❌ **Минусы**:
Нарушает принципы чистой архитектуры и превращается в "глобальную переменную".
Усложняет тестирование из-за скрытых зависимостей.
Может быть причиной скрытых связей между модулями.
Требует особой осторожности в многопоточных средах, чтобы не создать несколько экземпляров.

Одиночка: плохой пример

Пример, в котором всё плохо:

```
class BadConfig {
public:
   BadConfig() {}
   std::string getSetting() { return "setting_value"; }
};

void functionA() {
   BadConfig config;
   std::cout << "A: " << config.getSetting() << std::endl;
}

void functionB() {
   BadConfig config;
   std::cout << "B: " << config.getSetting() << std::endl;
}

int main() {
   functionA();
   functionB();
   return 0;
}
```

Проблемы:

- Много одинаковых объектов - каждый создает свой экземпляр
- Бесконтрольное создание - кто угодно может сделать new BadConfig()
- Трата ресурсов - одинаковые объекты дублируются в памяти
- Рассогласованность - разные части программы работают с разными объектами

Одиночка: хороший пример

Как должно быть:

```
class Config {
private:
   static Config* instance;
   Config() {}
public:
   static Config* getInstance() {
      if (!instance)
         instance = new Config();
      return instance;
   }
   std::string getSetting() { return "setting_value"; }
};
Config* Config::instance = nullptr;

int main() {
   Config* config1 = Config::getInstance();
   Config* config2 = Config::getInstance();

   std::cout << (config1 == config2 ? "Same instance!" : "Different!") << std::endl;
   return 0;
}
```

В хорошем примере:

- Один объект на всё приложение - все получают один и тот же экземпляр
- Контролируемое создание - только через getInstance()
- Экономия ресурсов - не дублируется в памяти