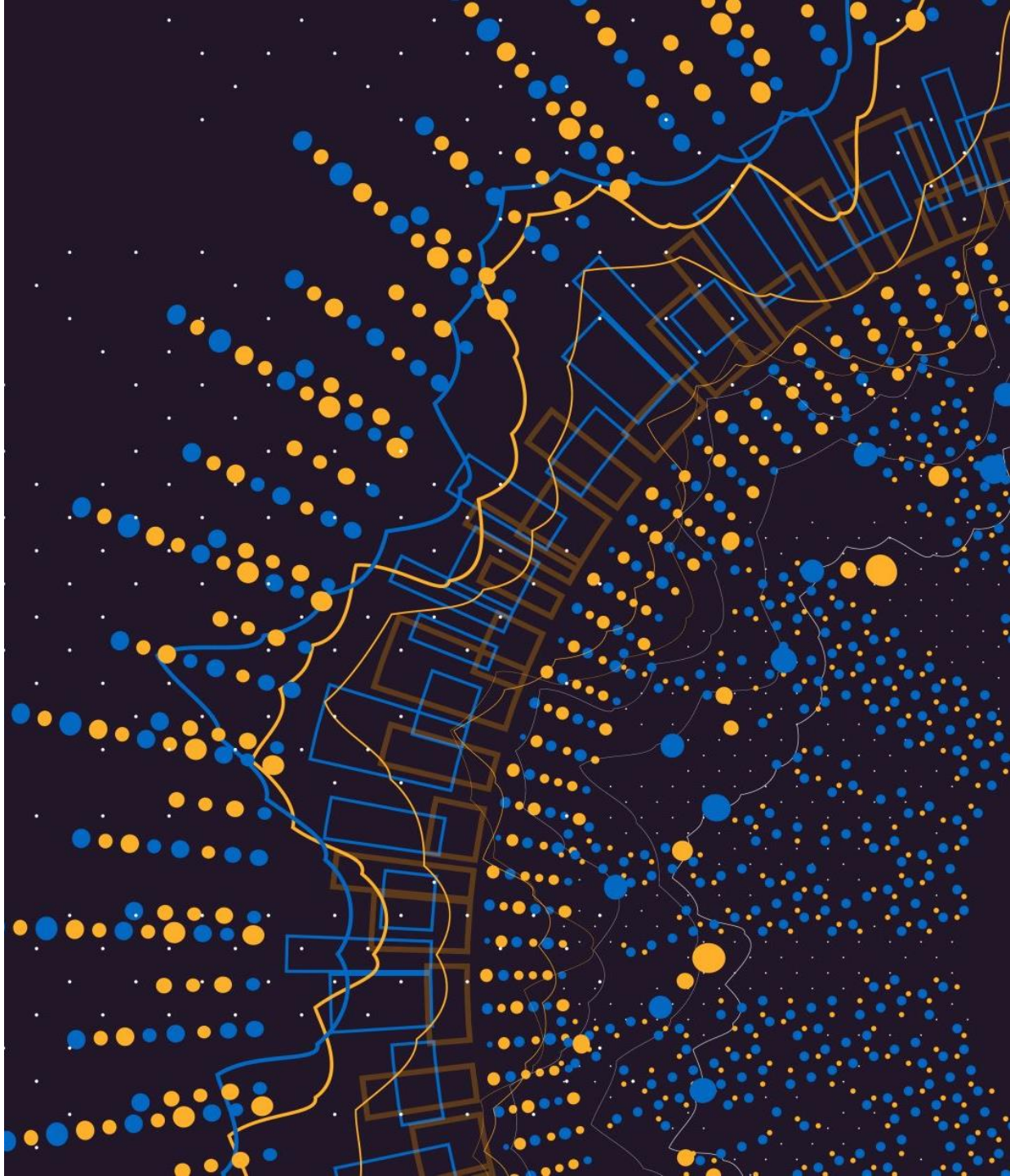
ОБЗОР ПАТТЕРНА SINGLETON

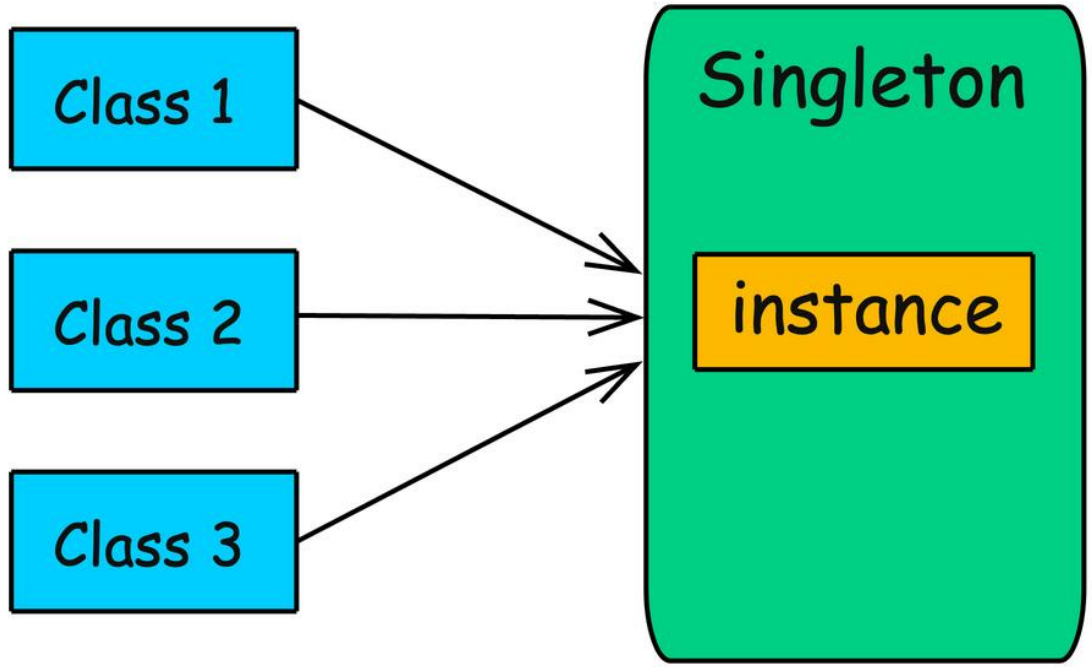


ПЛАН ЗАНЯТИЯ

- понятие паттерна Singleton
- UML-диаграмма Singleton
- разные способы реализации
- плюсы и минусы использования Singleton
- отношения с другими паттернами

ЧТО ТАКОЕ SINGLETON?

Singleton – паттерн проектирования, гарантирующий создание только одного объекта класса. Поэтому он выполняет функцию контроля количества экземпляров класса.



ГДЕ НУЖЕН SINGLETON?

- Логгер позволяет записывать информацию со всех частей программы в один лог
- Конфигурация, откуда все модули программы будут читать одинаковые настройки
- Пул подключений к БД даёт возможность использовать общее подключение, а не создавать новое при каждом запросе
- Менеджер сцен как единая точка переключения уровней в игре

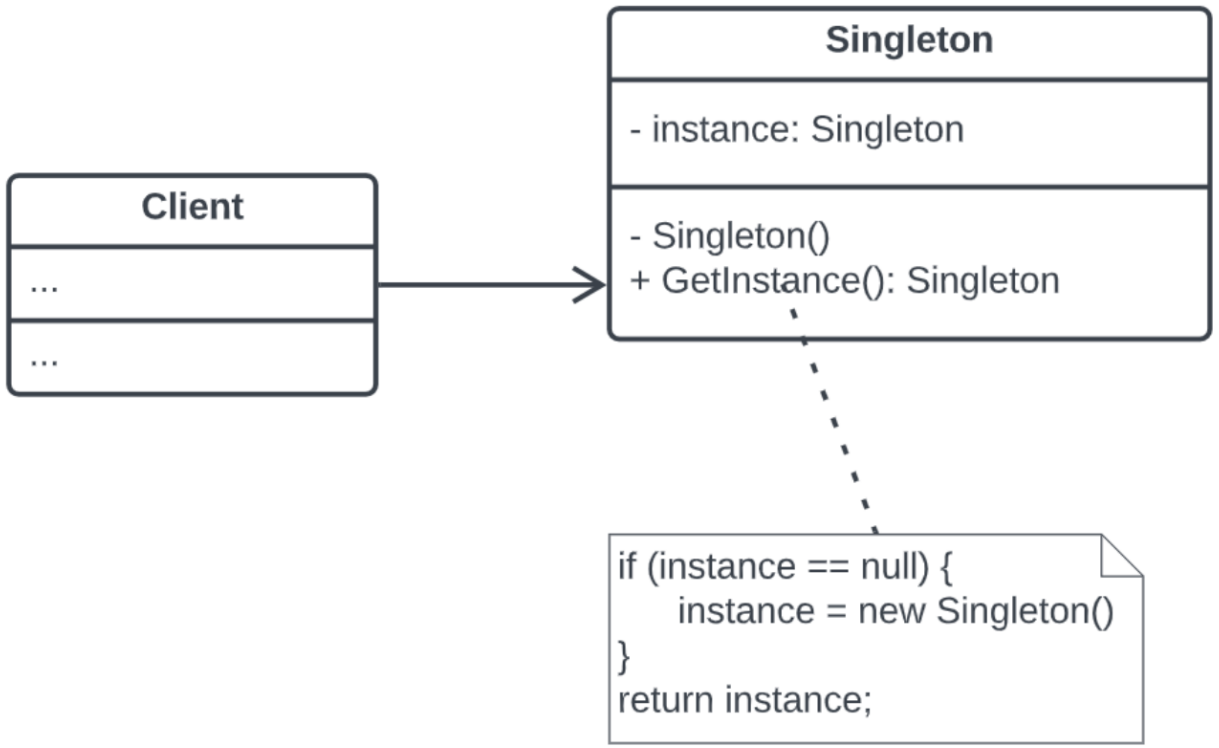
UML-ДИАГРАММА SINGLETON

Класс **Singleton** имеет приватное статическое поле **instance** типа **Singleton**. В нём хранится единственный экземпляр класса.

Конструктор **Singleton()** тоже приватный, чтобы нельзя было создать экземпляр извне.

Метод **GetInstance()** – публичный, возвращает единственный экземпляр Singleton.

Класс **Client** взаимодействует с **Singleton** через вызов **GetInstance().**



ЛЕНИВЫЙ SINGLETON

Здесь метод **getInstance()** создаёт объект только при первом вызове, что может вызывать некоторые проблемы, например, в многопоточном окружении может создаться несколько экземпляров. Это может произойти при одновременном прохождении двумя потоками проверки **cls._instance is None**, что приводит к созданию двух объектов.

```python
class LazySingleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super(LazySingleton, cls).__new__(cls)
        return cls._instance

singleton1 = LazySingleton()
singleton2 = LazySingleton()

print(singleton1 is singleton2)
```

МНОГОПОТОЧНЫЙ SINGLETON

Чтобы избежать проблемы многопоточности, нужно синхронизировать потоки при создании объекта-Одиночки.

```python
import threading
import time

class SafeSingleton:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    time.sleep(0.01)
                    cls._instance = super(SafeSingleton, cls).__new__(cls)
        return cls._instance

instances = []

def create_instance():
    instance = SafeSingleton()
    instances.append(instance)

threads = [threading.Thread(target=create_instance) for _ in range(10)]

for t in threads:
    t.start()

for t in threads:
    t.join()

unique_instances = set(instances)
print(f"Создано {len(unique_instances)} уникальных экземпляров Singleton.")
```

МНОГОПОТОЧНЫЙ SINGLETON

Чтобы избежать проблемы многопоточности, нужно синхронизировать потоки при создании объекта-Одиночки.

```python
import threading
import time

class SafeSingleton:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    time.sleep(0.01)
                    cls._instance = super(SafeSingleton, cls).__new__(cls)
        return cls._instance

instances = []

def create_instance():
    instance = SafeSingleton()
    instances.append(instance)

threads = [threading.Thread(target=create_instance) for _ in range(10)]

for t in threads:
    t.start()

for t in threads:
    t.join()

unique_instances = set(instances)
print(f"Создано {len(unique_instances)} уникальных экземпляров Singleton.")
```

ПЛЮСЫ И МИНУСЫ

+ Гарантирован один экземпляр

+ Централизованный доступ

+ Экономия ресурсов

- Скрытая зависимость от глобального состояния

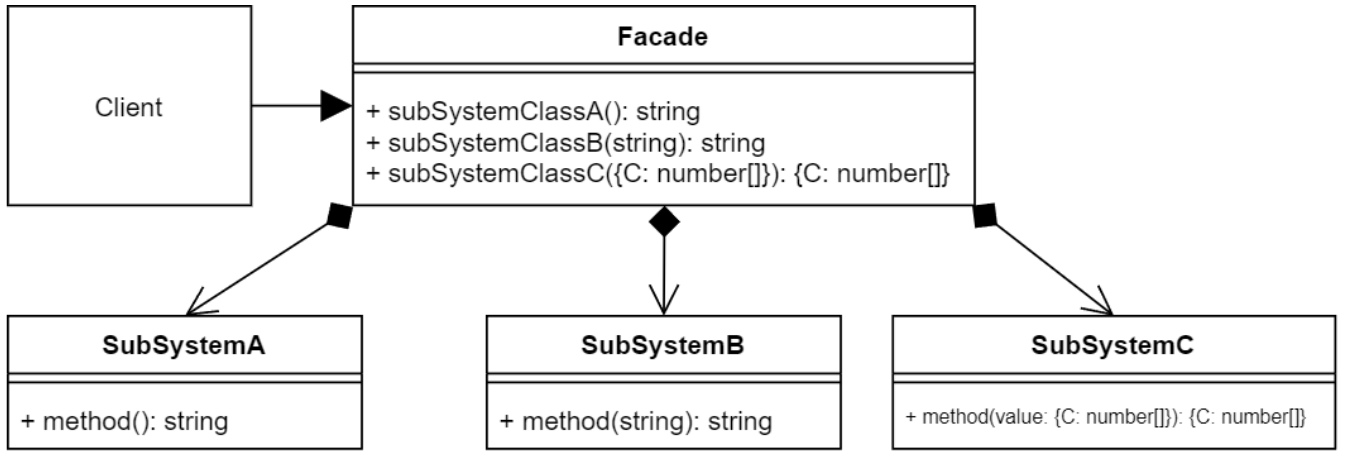
- Затруднённое тестирование

- Нарушает принцип единственной ответственности класса

ОТНОШЕНИЯ С ДРУГИМИ ПАТТЕРНАМИ

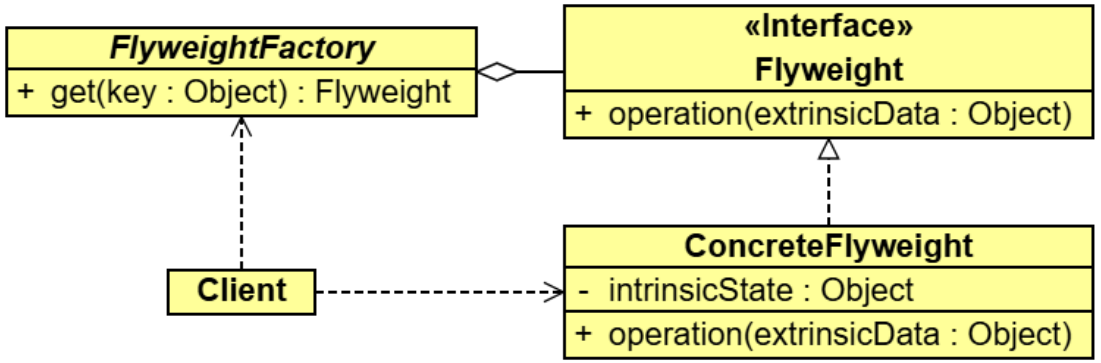
Паттерн Фасад (Façade) предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Фасад можно сделать **Одиночкой**, так как обычно нужен только один объект-фасад.



ОТНОШЕНИЯ С ДРУГИМИ ПАТТЕРНАМИ

Паттерн Приспособленец (Flyweight) – структурный шаблон проектирования, который позволяет использовать разделяемые объекты сразу в нескольких контекстах. Данный паттерн используется преимущественно для оптимизации работы с памятью.



Паттерн **Приспособленец** может напоминать **Одиночку**, если для конкретной задачи получается свести количество объектов к одному. Но между паттернами есть два кардинальных отличия: возможность нескольких объектов и неизменяемость объектов-приспособленцев.

ОТНОШЕНИЯ С ДРУГИМИ ПАТТЕРНАМИ

Паттерн **Абстрактная фабрика (Abstract Factory)** предоставляет интерфейс для создания семейств взаимосвязанных объектов с определенными интерфейсами без указания конкретных типов данных объектов.

Паттерн **Строитель (Builder)** – шаблон проектирования, который инкапсулирует создание объекта и позволяет разделить его на различные этапы.

Паттерн **Прототип (Prototype)** позволяет создавать объекты на основе уже ранее созданных объектов-прототипов. Иначе говоря, данный паттерн предлагает технику клонирования объектов.

Все перечисленные паттерны могут быть реализованы при помощи Одиночки.