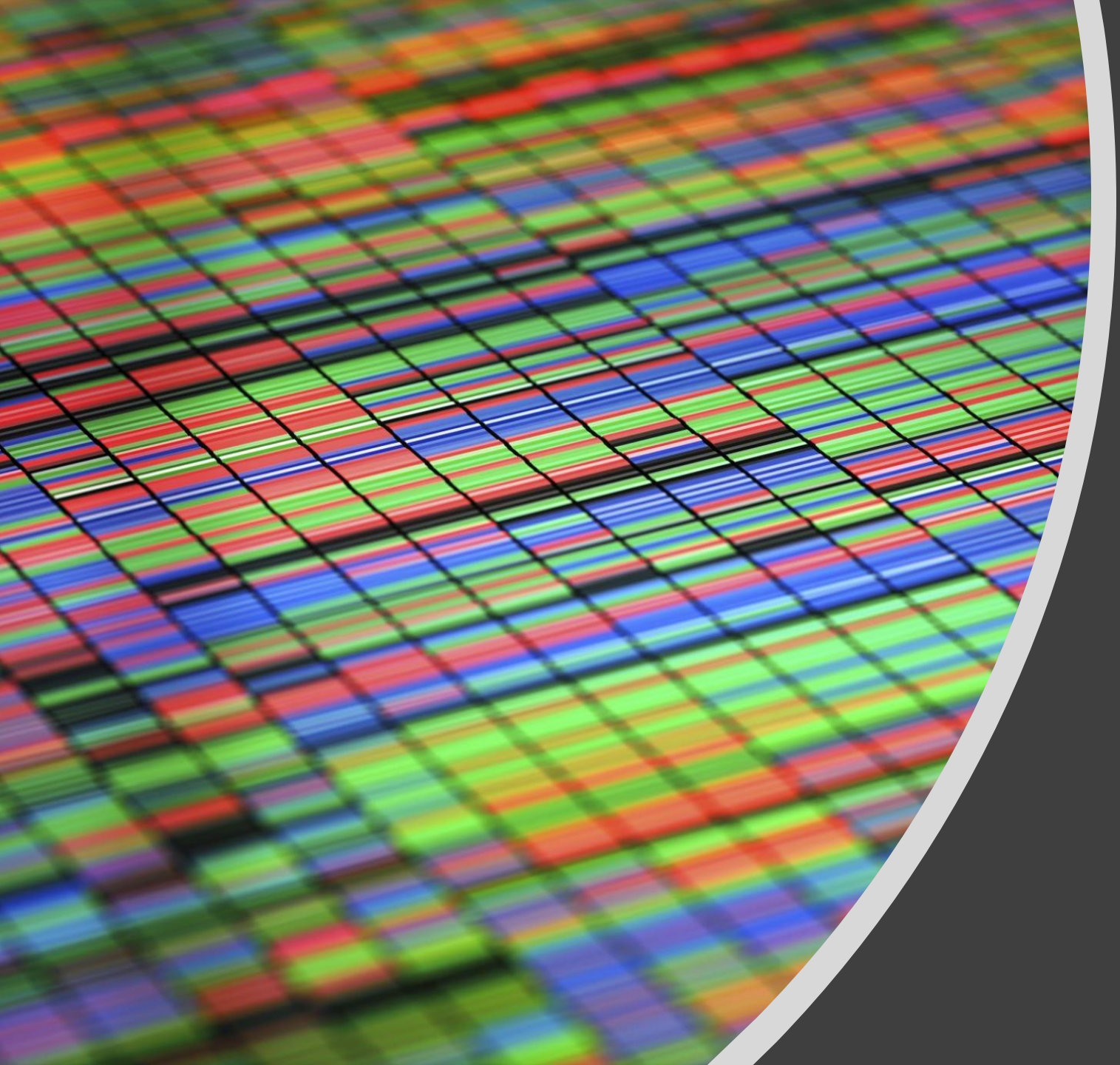


Proxy pattern

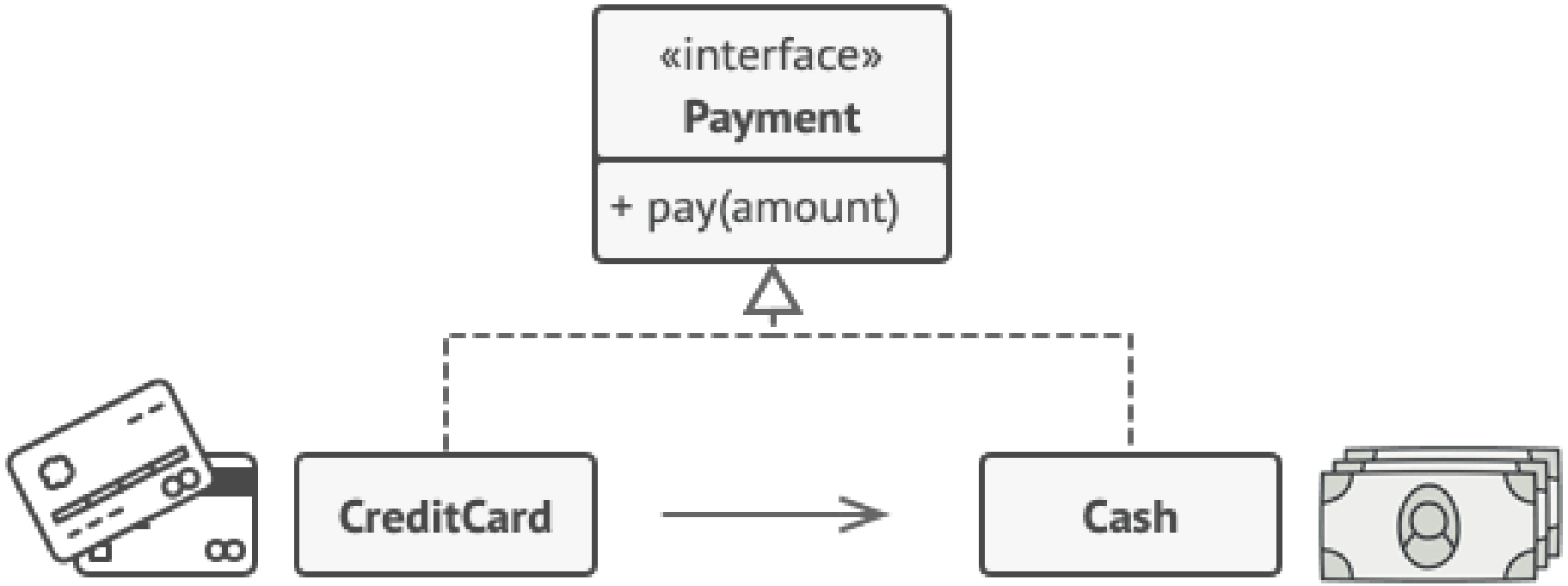
Заместитель (англ. *Proxy*) — структурный шаблон проектирования, предоставляющий объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера).

Прокси-объекты используются для выполнения важных операций перед доступом к реальным объектам.

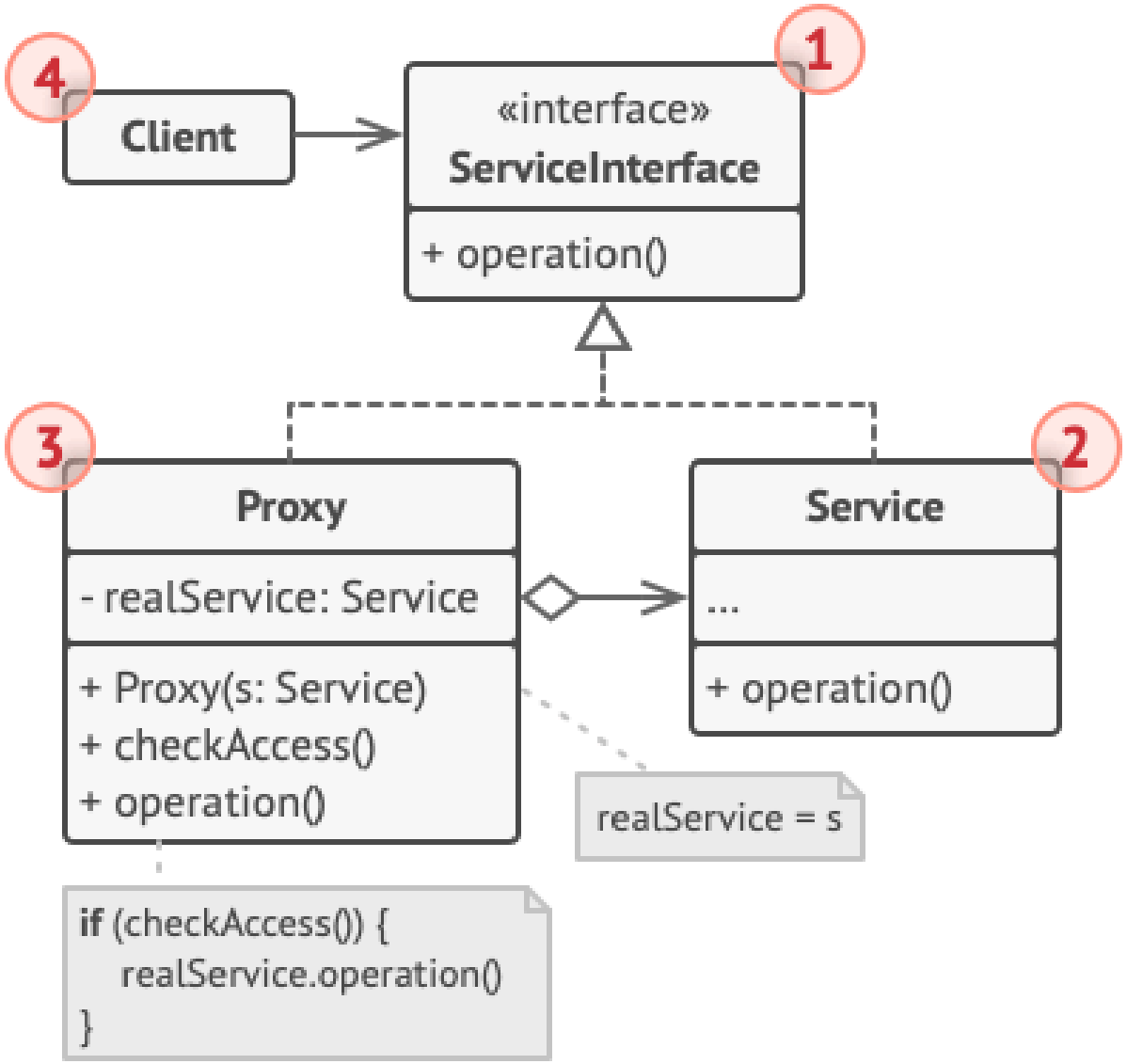
Этот паттерн схож с фасадом или адаптером, только у прокси интерфейс полностью повторяет интерфейс замещающего объекта.

Шаблон прокси-сервера позволяет вам легко отделить вашу основную логику от дополнительных функций, которые могут потребоваться в дополнение к этому. Модульный характер кода делает поддержание и расширение функциональных возможностей основной логики намного быстрее и проще.

Примером из реального мира может быть чек или кредитная карта, которые являются прокси для того, что находится на нашем банковском счете. Он может быть использован вместо наличных денег и обеспечивает возможность доступа к этим деньгам при необходимости. И это именно то, что делает шаблон прокси–сервера - “Контролирует и управляет доступом к объекту, который они защищают“.



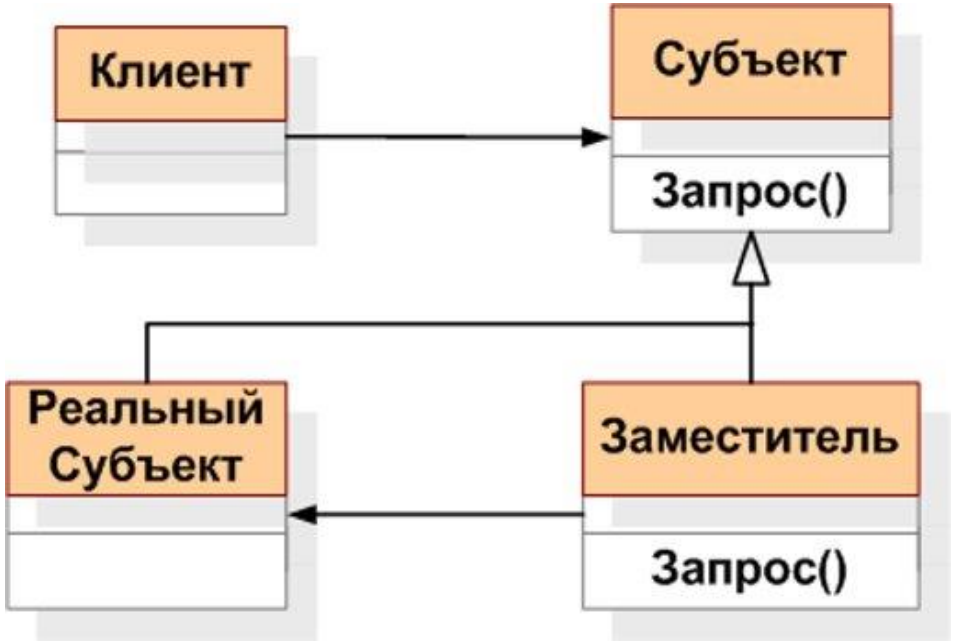
Структура


4
Client
1
«interface»
ServiceInterface
+ operation()
3
Proxy
- realService: Service
+ Proxy(s: Service)
+ checkAccess()
+ operation()
2
Service
...
+ operation()
realService = s
if (checkAccess()) {
realService.operation()
}

- **Сервис** содержит полезную бизнес-логику.
- **Заместитель** хранит ссылку на объект сервиса. После того как заместитель заканчивает свою работу (например, инициализацию, логирование, защиту или другое), он передаёт вызовы вложенному сервису.
  Заместитель может сам отвечать за создание и удаление объекта сервиса.
- **Клиент** работает с объектами через интерфейс сервиса. Благодаря этому, его можно «одурачить», подменив объект сервиса объектом заместителя.

- **Интерфейс сервиса** определяет общий интерфейс для сервиса и заместителя. Благодаря этому, объект заместителя можно использовать там, где ожидается объект сервиса.

Проще говоря, прокси-объект —
прослойка между клиентским кодом и целевым объектом.



```python
import logging
from typing import Union

logging.basicConfig(level=logging.INFO)


def division(a: Union[int, float], b: Union[int, float]) -> float:
    try:
        result = a / b
        return result

    except ZeroDivisionError:
        logging.error(f"Argument b cannot be {b}")

    except TypeError:
        logging.error(f"Arguments must be integers/floats")


print(division(1.9, 2))
```

Вы можете видеть, что эта функция уже выполняет три вещи одновременно, что нарушает принцип единой ответственности. SRP говорит, что функция или класс должны иметь только одну причину для изменения. В этом случае изменение любой из трех обязанностей может привести к изменению функции. Также это означает, что изменение или расширение функции может быть трудно отследить.

Вместо этого вы можете написать два класса. Основной класс Division будет реализовывать только основную логику, в то время как Proxy-Division расширит функциональность Division, добавив обработчики исключений и логгеры.

```python
import logging
from typing import Union

logging.basicConfig(level=logging.INFO)


class Division:
    def div(self, a: Union[int, float], b: Union[int, float]) -> float:
        return a / b
```

```python
class ProxyDivision:
    def __init__(self) -> None:
        self._klass = Division()

    def div(self, a: Union[int, float], b: Union[int, float]) -> float:
        try:
            result = self._klass.div(a, b)
            return result

        except ZeroDivisionError:
            logging.error(f"Argument b cannot be {b}")

        except TypeError:
            logging.error(f"Arguments must be integers/floats")


klass = ProxyDivision()
print(klass.div(2, 0))
```

Еще пример

```
class College:
    '''Resource-intensive object'''

    def studyingInCollege(self):
        print("Studying In College....")


class CollegeProxy:
    '''Relatively less resource-intensive proxy acting as middleman.
     Instantiates a College object only if there is no fee due.'''

    def __init__(self):

        self.feeBalance = 1000
        self.college = None

    def studyingInCollege(self):

        print("Proxy in action. Checking to see if the balance of student is clear or not...")
        if self.feeBalance <= 500:
            # If the balance is less than 500, let him study.
            self.college = College()
            self.college.studyingInCollege()
        else:

            # Otherwise, don't instantiate the college object.
            print("Your fee balance is greater than 500, first pay the fee")
```

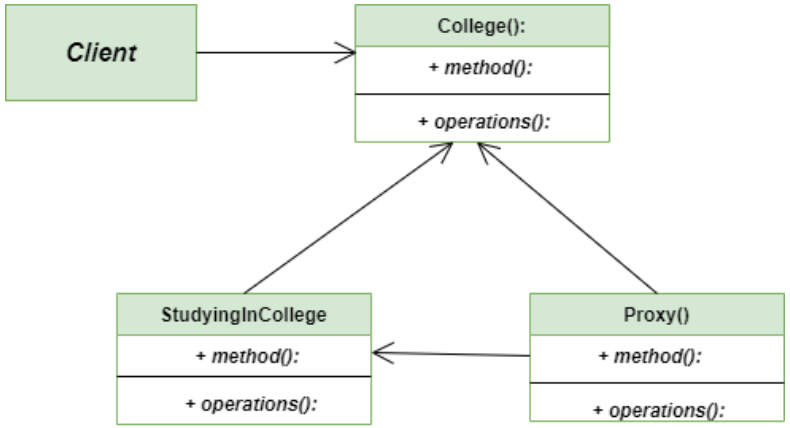
```
if __name__ == "__main__":

    # Instantiate the Proxy
    collegeProxy = CollegeProxy()

    # Client attempting to study in the college at the default balance of 1000.
    # Logically, since he / she cannot study with such balance,
    # there is no need to make the college object.
    collegeProxy.studyingInCollege()

    # Altering the balance of the student
    collegeProxy.feeBalance = 100

    # Client attempting to study in college at the balance of 100. Should succeed.
    collegeProxy.studyingInCollege()
```



В реальном мире ваш класс не будет выглядеть как простой класс Division, имеющий только один метод. Обычно ваш основной класс будет иметь несколько методов, и они будут выполнять несколько сложных задач. Прокси-классы должны реализовывать все методы основного класса. При написании прокси-класса для сложного основного класса автор этого класса может забыть реализовать все методы основного класса. Это приведет к нарушению шаблона прокси-сервера. Здесь решение представляет собой интерфейс, который может сигнализировать автору прокси-класса обо всех методах, которые необходимо реализовать.

```
class Proxy(IMath):
    """Прокси"""
    def __init__(self):
        self.math = Math()

    def add(self, x, y):
        return x + y

    def sub(self, x, y):
        return x - y

    def mul(self, x, y):
        return self.math.mul(x, y)

    def div(self, x, y):
        return float('inf') if y == 0 else self.math.div(x, y)

p = Proxy()
x, y = 4, 2
print '4 + 2 = ' + str(p.add(x, y))
print '4 - 2 = ' + str(p.sub(x, y))
print '4 * 2 = ' + str(p.mul(x, y))
print '4 / 2 = ' + str(p.div(x, y))
```

```python
class IMath:
    """Интерфейс для прокси и реального субъекта"""
    def add(self, x, y):
        raise NotImplementedError()

    def sub(self, x, y):
        raise NotImplementedError()

    def mul(self, x, y):
        raise NotImplementedError()

    def div(self, x, y):
        raise NotImplementedError()

class Math(IMath):
    """Реальный субъект"""
    def add(self, x, y):
        return x + y

    def sub(self, x, y):
        return x - y

    def mul(self, x, y):
        return x * y

    def div(self, x, y):
        return x / y
```

Для каких задач лучше использовать Proxy

- Кэширование.
- Отложенная реализация, также известная как ленивая. Зачем загружать объект сразу, если можно загрузить его по мере необходимости?
- Логирование запросов.
- Промежуточные проверки данных и доступа.
- Запуск параллельных потоков обработки.

В этом примере **Заместитель** помогает добавить в программу механизм кеширования результатов работы библиотеки интеграции с YouTube.

Оригинальный объект начинал загрузку по сети, даже если пользователь запрашивал одно и то же видео. Заместитель же загружает видео только один раз, используя для этого служебный объект, но в остальных случаях возвращает закешированный файл.