- **Структурные паттерны**

Структурные паттерны определяют отношения между классами и объектами, позволяя им работать совместно. При этом могут использоваться следующие механизмы:

Наследование, когда базовый класс определяет интерфейс, а подклассы - реализацию. Структуры на основе наследования получаются статичными.
Композиция, когда структуры строятся путем объединения объектов некоторых классов. Композиция позволяет получать структуры, которые можно изменять во время выполнения.

Структурные паттерны

№	Название паттерна	Перевод	Назначение паттерна
1	Adapter(синоним - Wrapper)	Адаптер (Обертка)	Преобразует существующий интерфейс класса в другой интерфейс, который понятен клиентам. При этом обеспечивает совместную работу классов, невозможную без данного паттерна из-за несовместимости интерфейсов.
2	Decorator	Декоратор	Применяется для расширения имеющейся функциональности и является альтернативой порождению подклассов на основе динамического назначения объектам новых операций.
3	Proxy	Заместитель	Подменяет выбранный объект другим объектом для управления контроля доступа к исходному объекту.
4	Composite	Компоновщик	Группирует объекты в иерархические структуры для представления отношений типа "часть-целое", что позволяет клиентам работать с единичными объектами так же, как с группами объектов.
5	Bridge	Мост	Отделяет абстракцию класса от его реализации, благодаря чему появляется возможность независимо изменять то и другое.
6	Flyweight	Приспособленец	Использует принцип разделения для эффективной поддержки большого числа мелких объектов.
7	Facade	Фасад	Предоставляет единый интерфейс к множеству операций или интерфейсов в системе на основе унифицированного интерфейса для облегчения работы с системой.

- **Паттерн Adapter**

Назначение:

Преобразует существующий интерфейс класса в другой интерфейс, который понятен клиентам. При этом обеспечивает совместную работу классов, невозможную без данного паттерна из-за несовместимости интерфейсов.

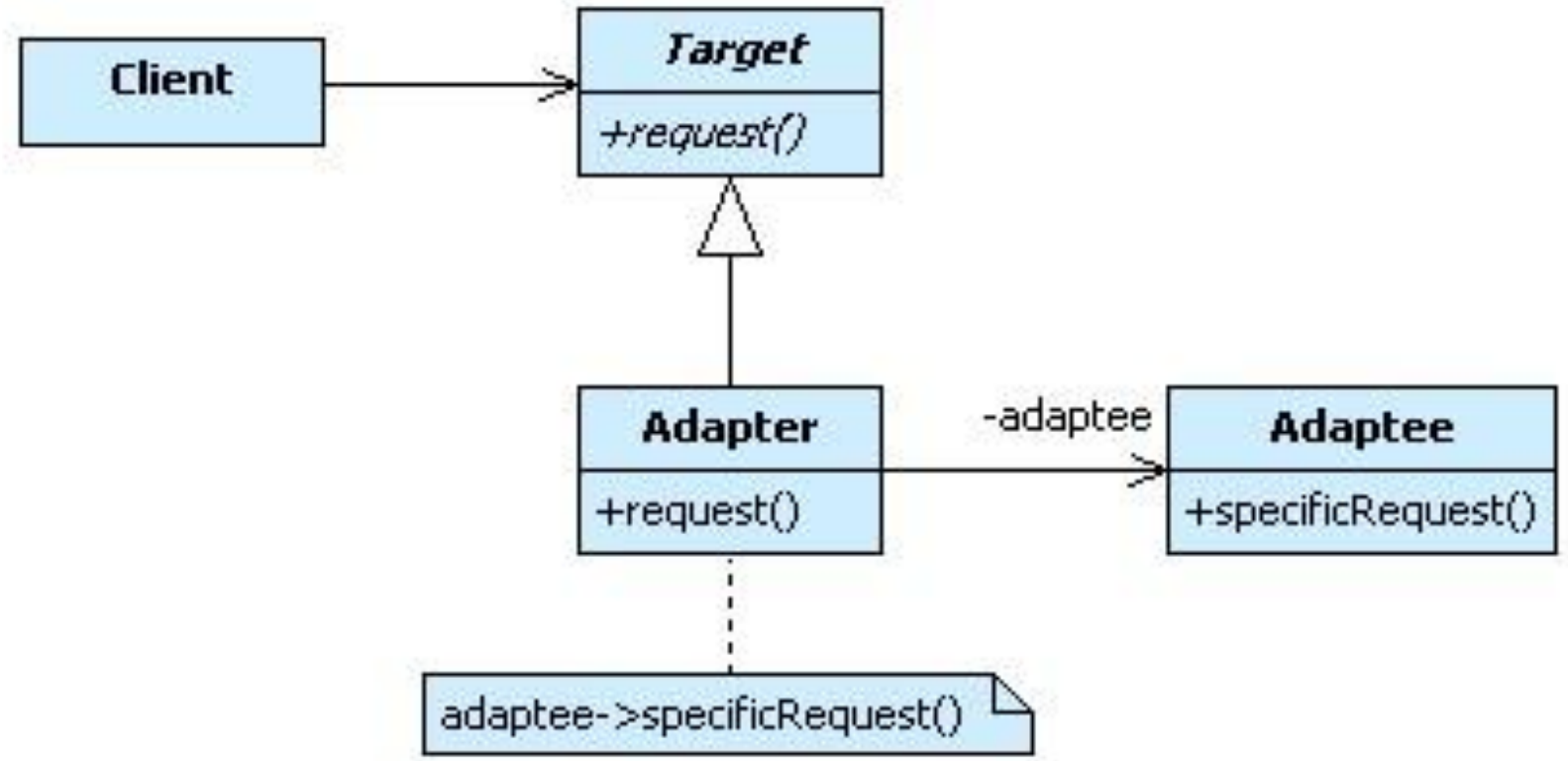
Часто в новом программном проекте не удается повторно использовать уже существующий код. Например, имеющиеся классы могут обладать нужной функциональностью, но иметь при этом несовместимые интерфейсы. В таких случаях следует использовать паттерн Adapter (адаптер).

Описание паттерна Adapter:

Пусть класс, интерфейс которого нужно адаптировать к нужному виду, имеет имя Adaptee. Для решения задачи преобразования его интерфейса паттерн Adapter вводит следующую иерархию классов:

- Виртуальный базовый класс Target. Здесь объявляется пользовательский интерфейс подходящего вида. Только этот интерфейс доступен для пользователя.
- Производный класс Adapter, реализующий интерфейс Target. В этом классе также имеется указатель или ссылка на экземпляр Adaptee. Паттерн Adapter использует этот указатель для перенаправления клиентских вызовов в Adaptee. Так как интерфейсы Adaptee и Target несовместимы между собой, то эти вызовы обычно требуют преобразования.

- **UML-диаграмма классов паттерна Adapter**



Паттерн Adapter представляет собой программную обертку над уже существующими классами и предназначен для преобразования их интерфейсов к виду, пригодному для последующего использования в новом программном проекте.

- **Пример паттерна Adapter**

Рассмотрим простой пример, когда следует применять паттерн Adapter. Пусть мы разрабатываем систему климат-контроля, предназначенной для автоматического поддержания температуры окружающего пространства в заданных пределах. Важным компонентом такой системы является температурный датчик, с помощью которого измеряют температуру окружающей среды для последующего анализа. Для этого датчика уже имеется готовое программное обеспечение от сторонних разработчиков, представляющее собой некоторый класс с соответствующим интерфейсом. Однако использовать этот класс непосредственно не удастся, так как показания датчика снимаются в градусах Фаренгейта. Нужен адаптер, преобразующий температуру в шкалу Цельсия.

Пример реализации паттерна Adapter

```
// Уже существующий класс температурного датчика окружающей среды
class FahrenheitSensor // считаем что этот класс закрыт для изменения, в терминах описания паттерна это Adaptee
{
public:
    // Получить показания температуры в градусах Фаренгейта
    float getFahrenheitTemp() { float t = 32.0; return t; }
};
class Sensor // в терминах описания паттерна - Target
{
public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;  // в терминах описания паттерна request()
};


class Adapter : public Sensor //  в терминах описания паттерна - Adapter
{
public:
    Adapter(FahrenheitSensor* p) : p_fsensor(p) {}
   ~Adapter() { delete p_fsensor; }
    float getTemperature() { // в терминах описания паттерна — request()
                                    return (p_fsensor->getFahrenheitTemp() - 32.0f)*5.0f / 9.0f;
                                }
private:
    FahrenheitSensor* p_fsensor; // в терминах описания паттерна -adaptee
};

int main()
{
   setlocale(LC_ALL, "rus");
   Sensor* p = new Adapter(new FahrenheitSensor);
   cout << "Температура в градусах: " << p->getTemperature() << endl;
   delete p;
}
```

- ***Результаты применения паттерна Adapter***

Достоинства паттерна Adapter:

Паттерн Adapter позволяет повторно использовать уже имеющийся код, адаптируя его несовместимый интерфейс к виду, пригодному для использования.

Недостатки паттерна Adapter:

Задача преобразования интерфейсов может оказаться непростой в случае, если клиентские вызовы и (или) передаваемые параметры не имеют функционального соответствия в адаптируемом объекте.

• Паттерн Bridge (мост)

Паттерн Bridge отделяет абстракцию от реализации так, что то и другое можно изменять независимо.

Назначение:

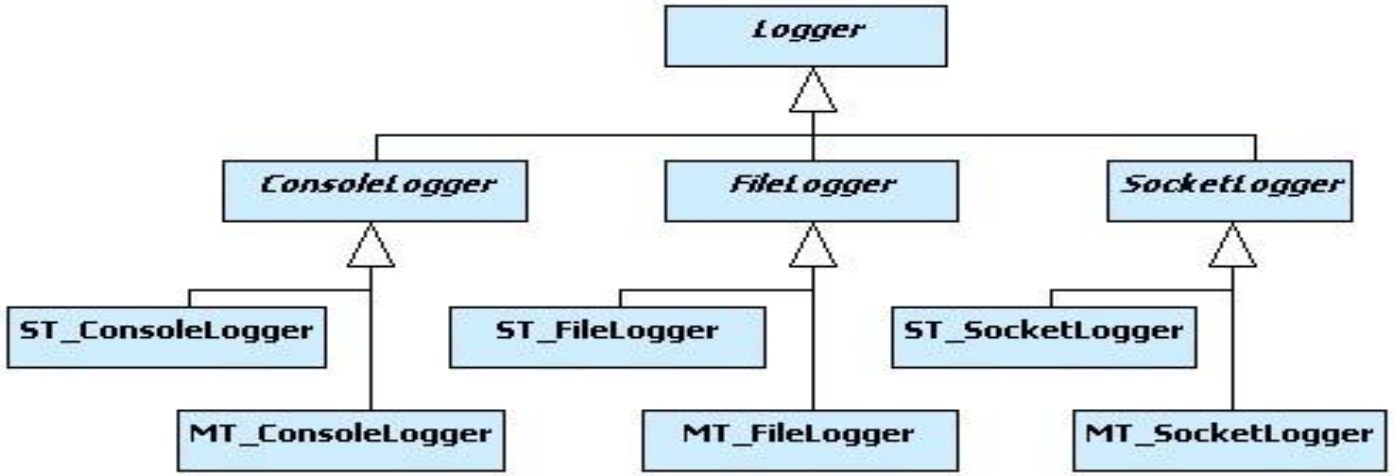
В системе могут существовать классы, отношения между которыми строятся в соответствии со следующей объектно-ориентированной иерархией: абстрактный базовый класс объявляет интерфейс, а конкретные подклассы реализуют его нужным образом. Такой подход является стандартным в ООП, однако, ему свойственны следующие недостатки:

- Система, построенная на основе наследования, является ***статичной***. Реализация жестко привязана к интерфейсу. Изменить реализацию объекта некоторого типа в процессе выполнения программы уже невозможно.
- Система становится трудно поддерживаемой, если число родственных производных классов становится большим.

• Паттерн Bridge (мост)

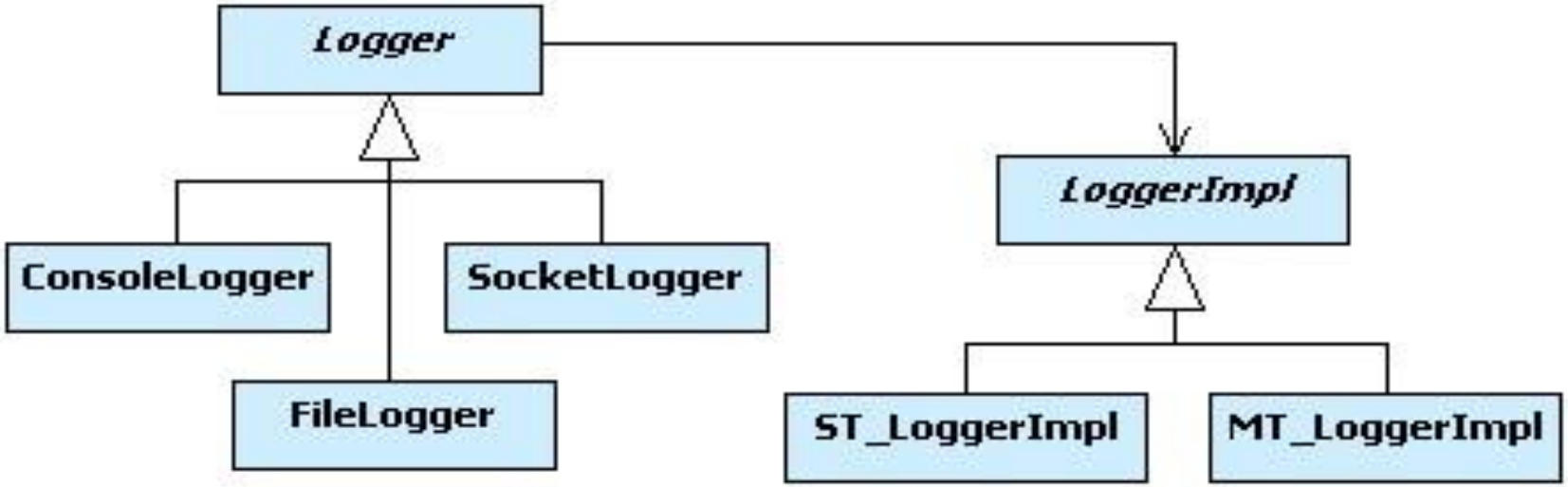
Рассмотрим сложности расширения системы новыми типами на примере разработки логгера. Логгер это система протоколирования сообщений, позволяющая фиксировать ошибки, отладочную и другую информацию в процессе выполнения программы. Разрабатываемый нами логгер может использоваться в одном из трех режимов: выводить сообщения на экран, в файл или отсылать их на удаленный компьютер. Кроме того, необходимо обеспечить возможность его применения в однопоточной и многопоточной средах.

Станд ледующую иерархию

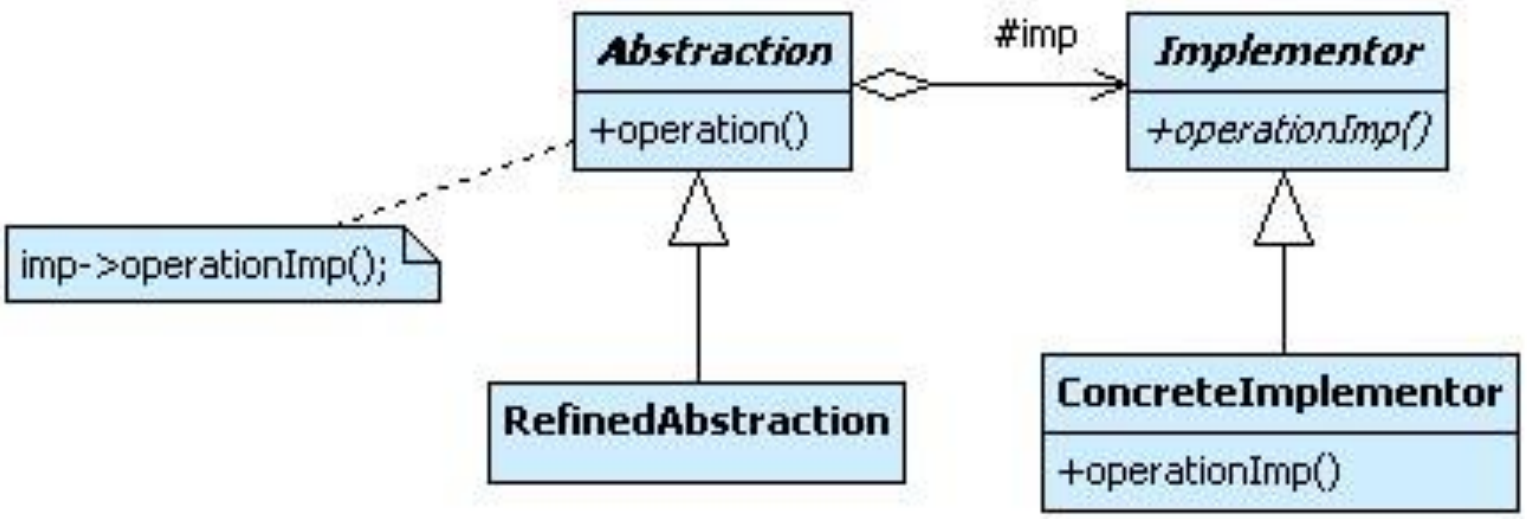


Видно, что число родственных подклассов в системе равно 6. Добавление еще одного вида логгера увеличит его до 8, двух - до 10 и так далее. Система становится трудно управляемой.

Пример иерархии на основе паттерна Bridge



- **UML-диаграмма классов паттерна Bridge**



Описание паттерна Bridge

Первая иерархия определяет интерфейс абстракции, доступный пользователю. Для случая проектируемого нами логгера абстрактный базовый класс Logger мог бы объявить интерфейс метода log() для вывода сообщений. Класс Logger также содержит указатель на реализацию pimpl, который инициализируется должным образом при создании логгера конкретного типа. Этот указатель используется для перенаправления пользовательских запросов в реализацию. Заметим, в общем случае подклассы ConsoleLogger, FileLogger и SocketLogger могут расширять интерфейс класса Logger.

Все детали реализации, связанные с особенностями среды скрываются во второй иерархии. Базовый класс LoggerImpl объявляет интерфейс операций, предназначенных для отправки сообщений на экран, файл и удаленный компьютер, а подклассы ST_LoggerImpl и MT_LoggerImpl его реализуют для однопоточной и многопоточной среды соответственно. В общем случае, интерфейс LoggerImpl необязательно должен в точности соответствовать интерфейсу абстракции. Часто он выглядит как набор низкоуровневых примитивов.

Паттерн Bridge позволяет легко изменить реализацию во время выполнения программы. Для этого достаточно перенастроить указатель pimpl на объект-реализацию нужного типа. Применение паттерна Bridge также позволяет сократить общее число подклассов в системе, что делает ее более простой в поддержке.

Пример реализации паттерна Bridge

```
class LoggerImpl; // Опережающее объявление
class Logger
{
 public:
   Logger(LoggerImpl* p);
   virtual ~Logger();
   virtual void log(string & str) = 0;
   protected:
   LoggerImpl * pimpl;
};
class ConsoleLogger : public Logger
{
 public:
   ConsoleLogger();
   void log(string & str);
};
class FileLogger : public Logger
{
 public:
   FileLogger(string & file_name);
   void log(string & str);
 private:
   string file;
};
class SocketLogger : public Logger
{
 public:
   SocketLogger(string & remote_host, int remote_port);
   void log(string & str);
 private:
   string host;
   int    port;
};
```

Пример реализации паттерна Bridge — продолжение

```
Logger::Logger(LoggerImpl* p) : pimpl(p) { }

Logger::~Logger()  { delete pimpl; }
ConsoleLogger::ConsoleLogger() : Logger(
#ifdef MT
        new MT_LoggerImpl()
#else
        new ST_LoggerImpl()
#endif
)
{ }
void ConsoleLogger::log(string & str)
{
        pimpl->console_log(str);
}
FileLogger::FileLogger(string & file_name) : Logger(
#ifdef MT
        new MT_LoggerImpl()
#else
        new ST_LoggerImpl()
#endif
), file(file_name)
{ }
void FileLogger::log(string & str)
{
        pimpl->file_log(file, str);
}
```

Пример реализации паттерна Bridge — продолжение

```
SocketLogger::SocketLogger(string & remote_host, int remote_port) : Logger(
#ifdef MT
            new MT_LoggerImpl()
#else
            new ST_LoggerImpl()
#endif
      ), host(remote_host), port(remote_port)
{ }

void SocketLogger::log(string & str)
{
      pimpl->socket_log(host, port, str);
}
```

Пример реализации паттерна Bridge

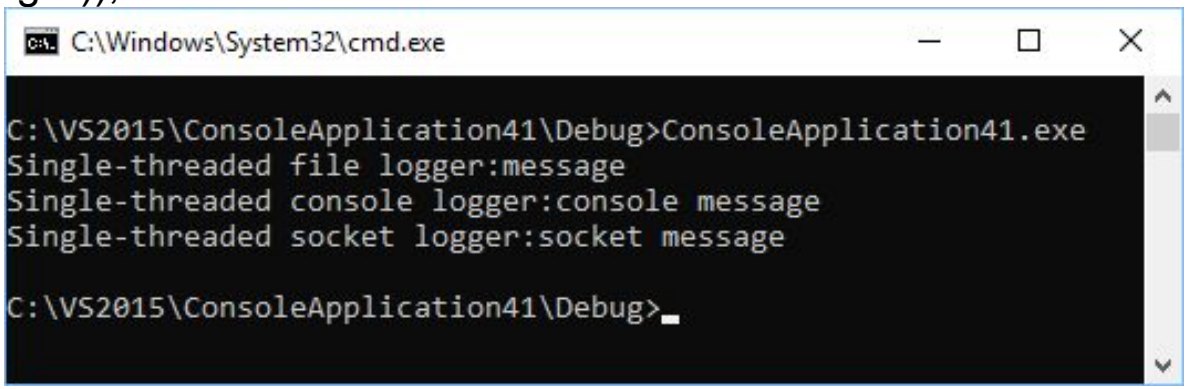
```
class LoggerImpl
{
 public:
   virtual ~LoggerImpl() {}
   virtual void console_log(string & str) = 0;
   virtual void file_log(string & file, string & str) = 0;
   virtual void socket_log(string & host, int port, string & str) = 0;
};

class ST_LoggerImpl : public LoggerImpl
{
 public:
   void console_log(string & str);
   void file_log(string & file, string & str);
   void socket_log(string & host, int port, string & str);
};

class MT_LoggerImpl : public LoggerImpl
{
 public:
   void console_log(string & str);
   void file_log(string & file, string & str);
   void socket_log(string & host, int port, string & str);
};
```

Пример реализации паттерна Bridge — продолжение

```
void ST_LoggerImpl::console_log(string & str)
{ cout << "Single-threaded console logger:"<< str << endl; }

void ST_LoggerImpl::file_log(string & file, string & str)
{ cout << "Single-threaded file logger:" << str << endl; }

void ST_LoggerImpl::socket_log(string & host, int port, string & str)
{ cout << "Single-threaded socket logger:" << str << endl; }

void MT_LoggerImpl::console_log(string & str)
{ cout << "Multithreaded console logger:" << str << endl; }

void MT_LoggerImpl::file_log(string & file, string & str)
{ cout << "Multithreaded file logger:" << str << endl; }

void MT_LoggerImpl::socket_log(    string & host, int port, string & str)
{ cout << "Multithreaded socket logger:" << str << endl; }
```

Пример реализации паттерна Bridge — продолжение

```
// ConsoleApplication41.cpp: пример паттерна Bridge
//

#include <string>
#include "Logger.h"

int main()
{
	Logger * p = new FileLogger(string("log.txt"));
	p->log(string("message"));
	delete p;
	Logger * pcons = new ConsoleLogger();
	pcons->log(string("console message"));
	delete pcons;
	Logger * psoc = new SocketLogger(std::string("127.0.0.1"),32765);
	psoc->log(string("socket message"));
	delete psoc;
	return 0;
}
```



- **паттерн Bridge**

Паттерны Bridge и Adapter имеют схожую структуру, однако, цели их использования различны. Если паттерн Adapter применяют для адаптации уже существующих классов в систему, то паттерн Bridge используется на стадии ее проектирования.

Отметим несколько важных моментов приведенной реализации паттерна Bridge:

- При модификации реализации клиентский код перекомпилировать не нужно. Использование в абстракции указателя на реализацию (**идиома pimpl**) позволяет заменить в файле Logger.h включение include "LoggerImpl.h" на опережающее объявление class LoggerImpl. Такой прием снимает зависимость времени компиляции файла Logger.h (и, соответственно, использующих его файлов клиента) от файла LoggerImpl.h.
- Пользователь класса Logger не видит никаких деталей его реализации.

- *Результаты применения паттерна Bridge*

- Достоинства паттерна Bridge:
- Проще расширять систему новыми типами за счет сокращения общего числа родственных подклассов.
- Возможность динамического изменения реализации в процессе выполнения программы.
- Паттерн Bridge полностью скрывает реализацию от клиента. В случае модификации реализации пользовательский код не требует перекомпиляции.

паттерн Компоновщик (Composite)

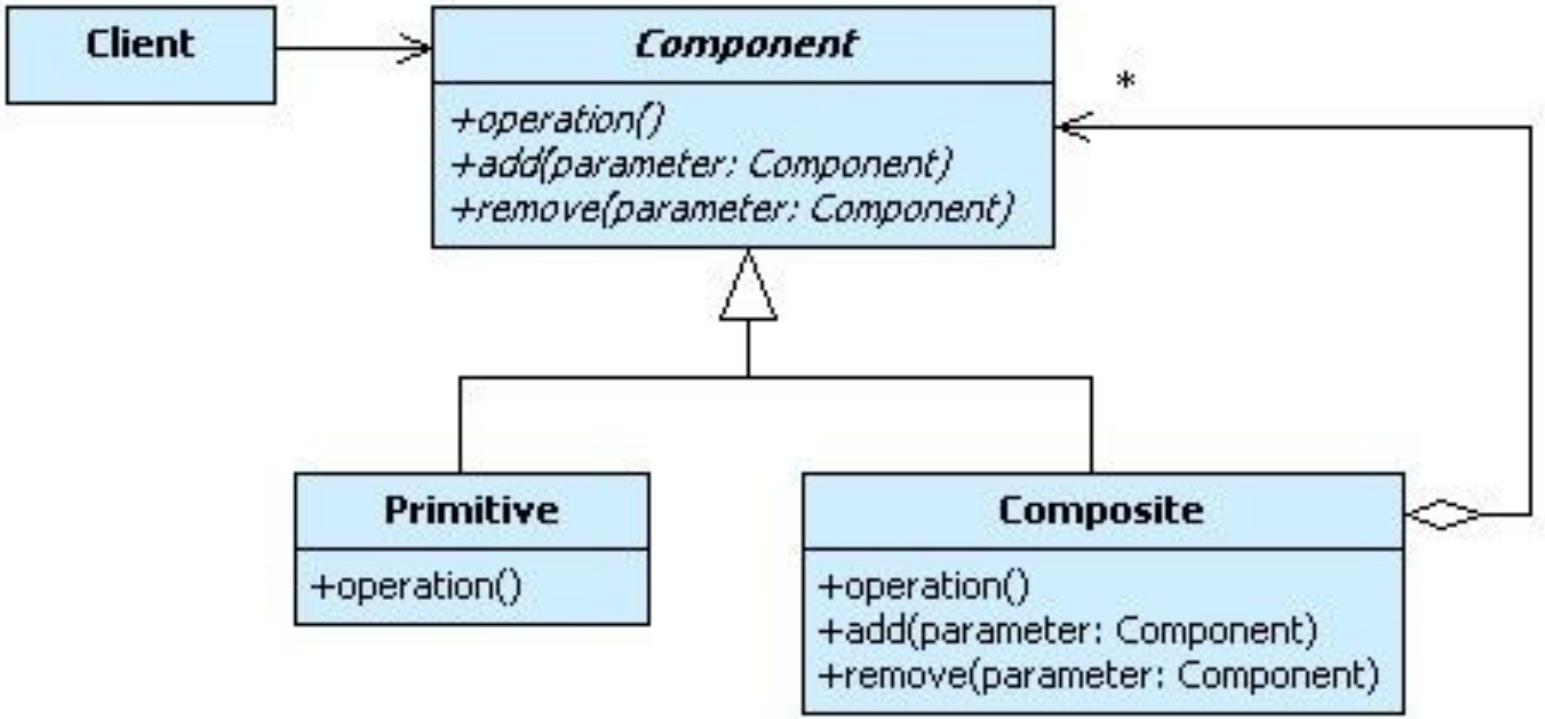
Паттерн Composite группирует схожие объекты в древовидные структуры. Рассматривает единообразно простые и сложные объекты.

Назначение паттерна Composite:

паттерн Composite используется если:

- Необходимо объединять группы схожих объектов и управлять ими.
- Объекты могут быть как примитивными (элементарными), так и составными (сложными). Составной объект может включать в себя коллекции других объектов, образуя сложные древовидные структуры. Пример: директория файловой системы состоит из элементов, каждый их которых также может быть директорией.
- Код клиента работает с примитивными и составными объектами единообразно.

- **UML-диаграмма классов паттерна Composite**



Для добавления или удаления объектов-потомков в составной объект Composite, класс Component определяет интерфейсы add() и remove().

. Пример использования паттерна Компоновщик (Composite)

Управление группами объектов может быть непростой задачей, особенно, если эти объекты содержат собственные объекты.

Для военной стратегической игры ”Пунические войны ”, описывающей военное противостояние между Римом и Карфагеном каждая боевая единица (всадник, лучник, пехотинец) имеет свою собственную разрушающую силу. Эти единицы могут объединяться в группы для образования более сложных военных подразделений, например, римские легионы, которые, в свою очередь, объединяясь, образуют целую армию. Как рассчитать боевую мощь таких иерархических соединений?

Паттерн Composite предлагает следующее решение. Он вводит абстрактный базовый класс Component с поведением, общим для всех примитивных и составных объектов. Для случая стратегической игры - это метод getStrength() для подсчета разрушающей силы. Подклассы Primitive и Composite являются производными от класса Component. Составной объект Composite хранит компоненты-потомки абстрактного типа Component, каждый из которых может быть также Composite.

Для добавления или удаления объектов-потомков в составной объект Composite, класс Component определяет интерфейсы add() и remove().

Пример реализации паттерна «Компоновщик»

```
// Component
class Unit
{
public:
virtual int getStrength() = 0;
virtual void addUnit(Unit* p) { assert(false); }
virtual ~Unit() {}
};

// Primitives
class Archer : public Unit
{
public:
virtual int getStrength() { return 1; }
};

class Infantryman : public Unit
{
public:
virtual int getStrength() { return 2; }
};

class Horseman : public Unit
{
public:
virtual int getStrength() { return 3; }
};
```

Пример реализации паттерна «Компоновщик»

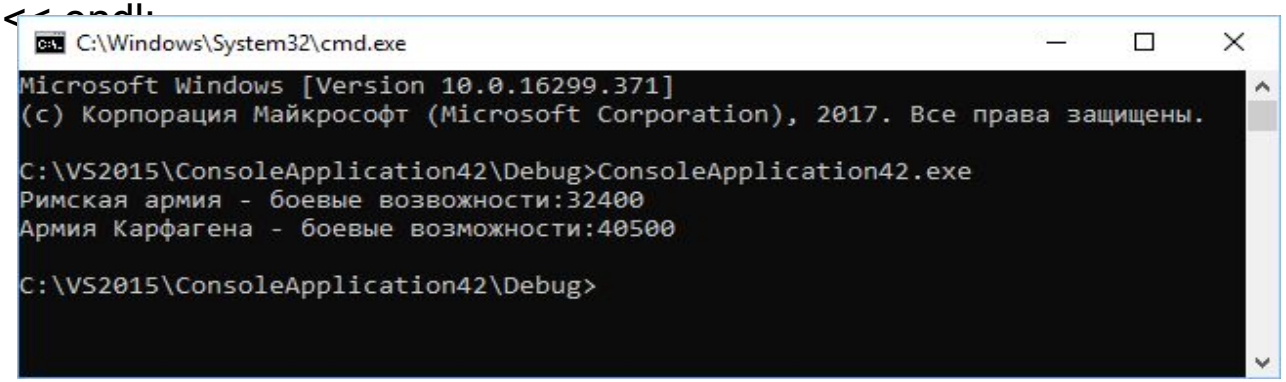
```
// Composite
#pragma once
#include <vector>
#include "Component.h"

class CompositeUnit : public Unit
{
public:
	int getStrength()
		{
			int total = 0;
			for (int i = 0; i<c.size(); ++i)
				total += c[i]->getStrength();
			return total;
		}
	void addUnit(Unit* p) { c.push_back(p); }
	~CompositeUnit()
		{
			for (int i = 0; i<c.size(); ++i)
			delete c[i];
		}
private:
	std::vector<Unit*> c;
};
```

Пример реализации паттерна «Компоновщик»

```
// Вспомогательная функция для создания легиона
CompositeUnit* createLegion()
{
	// легион содержит:
	CompositeUnit* legion = new CompositeUnit;

	// 3000 тяжелых пехотинцев
	for (int i = 0; i<3000; ++i)
		legion->addUnit(new Infantryman);

	// 1200 легких пехотинцев
	for (int i = 0; i<1200; ++i)
		legion->addUnit(new Archer);

	// 300 всадников
	for (int i = 0; i<300; ++i)
		legion->addUnit(new Horseman);

	return legion;
}
```

Пример реализации паттерна «Компоновщик»

```
int main()
{
	setlocale(LC_ALL, "rus");

	// Римская армия состоит из 4-х легионов
	CompositeUnit* roman_army = new CompositeUnit;
	for (int i = 0; i<4; ++i)
		roman_army->addUnit(createLegion());

	cout << "Римская армия - боевые возвожности:"
		<< roman_army->getStrength() << endl;
	// …
	// Армия Карфагена состоит из 5-х легионов
	CompositeUnit* puni_army = new CompositeUnit;
	for (int i = 0; i<5; ++i)
		puni_army->addUnit(createLegion());

	cout << "Армия Карфагена - боевые возможности:"
		<< puni_army->getStrength() << endl;

	delete roman_army, puni_army;
	return 0;
}
```



Результаты применения паттерна Composite

Достоинства паттерна Composite:

В систему легко добавлять новые примитивные или составные объекты, так как паттерн Composite использует общий базовый класс Component.

Код клиента имеет простую структуру – примитивные и составные объекты обрабатываются одинаковым образом.

Паттерн Composite позволяет легко обойти все узлы древовидной структуры.

Недостатки паттерна Composite:

Неудобно осуществить запрет на добавление в составной объект Composite объектов определенных типов.

Так, например, в состав римской армии не могут входить боевые слоны.

Паттерн Декоратор (Decorator)

Паттерн Decorator используется для расширения функциональности объектов. Являясь гибкой альтернативой порождению классов, паттерн Decorator динамически добавляет объекту новые обязанности.

Назначение паттерна Decorator:

Паттерн Decorator динамически добавляет новые обязанности объекту. Декораторы являются гибкой альтернативой порождению подклассов для расширения функциональности.

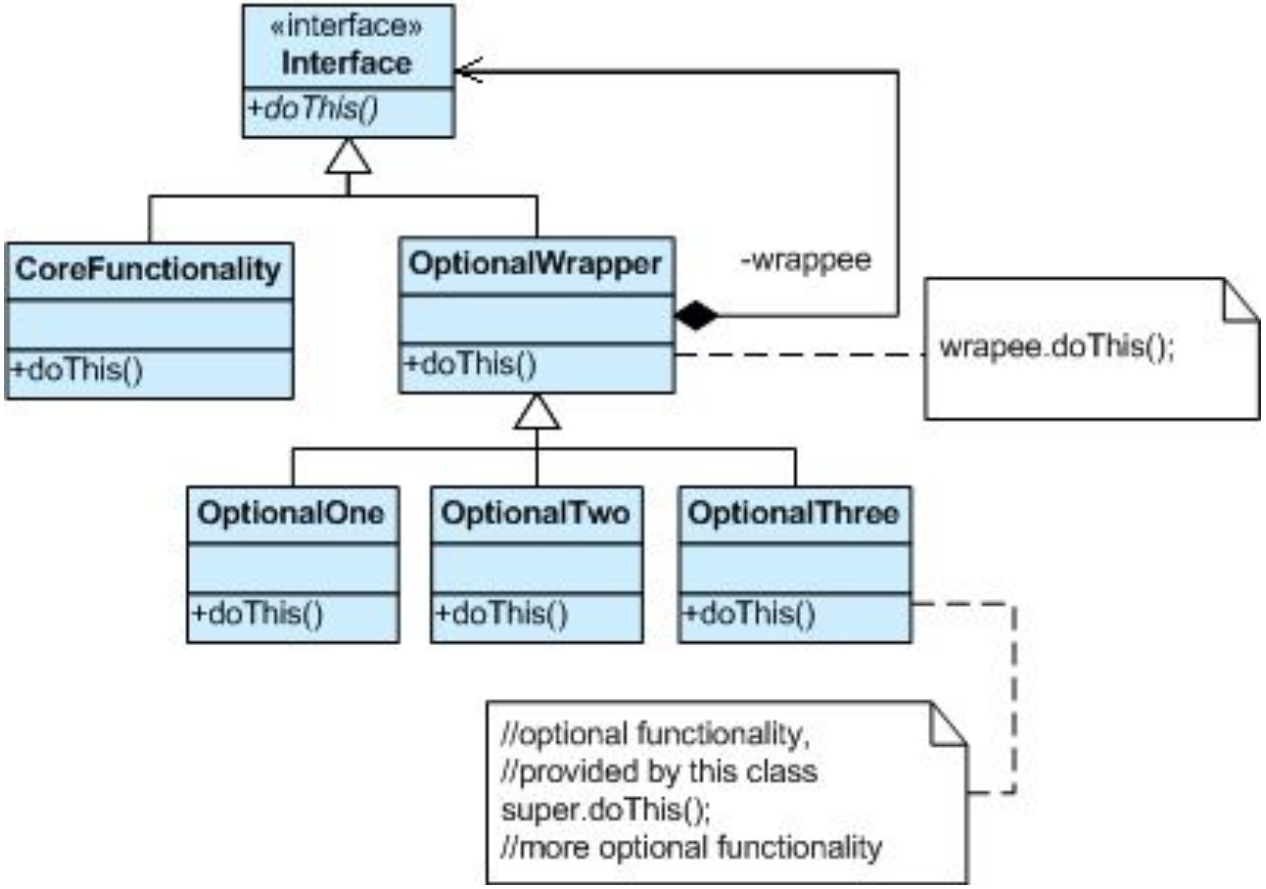
Рекурсивно декорирует основной объект.

Паттерн Decorator использует схему "обертываем подарок, кладем его в коробку, обертываем коробку".

Решаемая проблема:

Вы хотите добавить новые обязанности в поведении или состоянии ***отдельных объектов во время выполнения программы***. Использование наследования не представляется возможным, поскольку это решение статическое и распространяется целиком на весь класс.

- **UML-диаграмма классов паттерна Decorator**



Пример реализации паттерна «Декоратор»

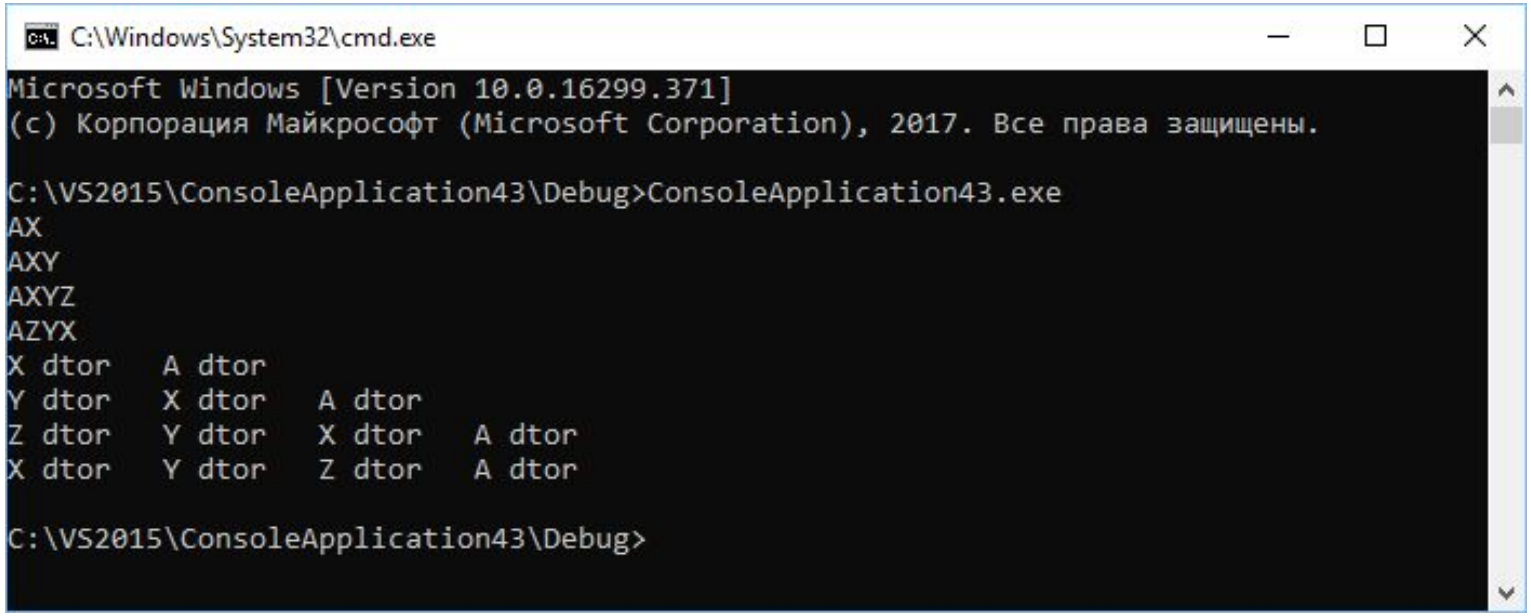
```
#include <iostream>
using namespace std;

class I { // НОЗ (LCD) - Interface
public:
virtual ~I() {}
virtual void doThis() = 0;
};

class A : public I { // Основной объект с базовой функциональностью CoreFunctionality
public:
~A() { cout << "A dtor" << endl; }
/*virtual*/
void doThis() { cout << 'A'; }
};

class D : public I { // OptionalWrapper
public:
D(I *inner) { m_wrappee = inner; }
~D() {         delete m_wrappee; }
/*virtual*/
void doThis() { m_wrappee->doThis(); }
private:
I *m_wrappee;  // указатель на LCD
};
```

Пример реализации паттерна «Декоратор»

```
class X : public D {  // доп.функциональность OptionalOne
public:
        X(I *core) : D(core) {}
        ~X() {        cout << "X dtor" << "    ";     }
        /*virtual*/
        void doThis() {  D::doThis(); cout << 'X'; }
};

class Y : public D {// доп.функциональность OptionalTwo
public:
        Y(I *core) : D(core) {}
        ~Y() {        cout << "Y dtor" << "    ";     }
        /*virtual*/
        void doThis() { D::doThis(); cout << 'Y'; }
};

class Z : public D {// доп.функциональность OptionalThree
public:
        Z(I *core) : D(core) {}
        ~Z() { cout << "Z dtor" << "    "; }
        /*virtual*/
        void doThis() { D::doThis(); cout << 'Z'; }
};
```

Пример реализации паттерна «Декоратор»

```
int main() {
    I *anX = new    X(new A);
    I *anXY = new   Y(new X(new A));
    I *anXYZ = new  Z(new Y(new X(new A)));
    I *anZYX = new  X(new Y(new Z(new A)));
    anX->doThis();      cout  << endl;
    anXY->doThis();     cout << endl;
    anXYZ->doThis(); cout << endl;
    anZYX->doThis(); cout << endl;
    delete anX;
    delete anXY;
    delete anXYZ;
    delete anZYX;
}
```



Паттерн Декоратор (Decorator)

- ***Использование паттерна Decorator:***

- Подготовьте исходные данные: один основной компонент и несколько дополнительных (необязательных) "оберток".
- Создайте общий для всех классов интерфейс по принципу "наименьшего общего знаменателя НОЗ" (lowest common denominator LCD). Этот интерфейс должен делать все классы взаимозаменяемыми.
- Создайте базовый класс второго уровня (Decorator) для поддержки дополнительных декорирующих классов.
- Основной класс и класс Decorator наследуют общий НОЗ-интерфейс.
- Класс Decorator использует отношение композиции. Указатель на НОЗ-объект инициализируется в конструкторе.
- Класс Decorator делегирует выполнение операции НОЗ-объекту.
- Для реализации каждой дополнительной функциональности создайте класс, производный от Decorator.
- Подкласс Decorator реализует дополнительную функциональность и делегирует выполнение операции базовому классу Decorator.
- Клиент несет ответственность за конфигурирование системы: устанавливает типы и последовательность использования основного объекта и декораторов.

Паттерн Фасад (Facade)

Паттерн Facade предоставляет высокоуровневый унифицированный интерфейс к набору интерфейсов некоторой подсистемы, что облегчает ее использование.

Назначение паттерна Facade:

Паттерн Facade предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Facade определяет интерфейс более высокого уровня, упрощающий использование подсистемы.

Паттерн Facade "обертывает" сложную подсистему более простым интерфейсом.

Решаемая проблема:

Клиенты хотят получить упрощенный интерфейс к общей функциональности сложной подсистемы.

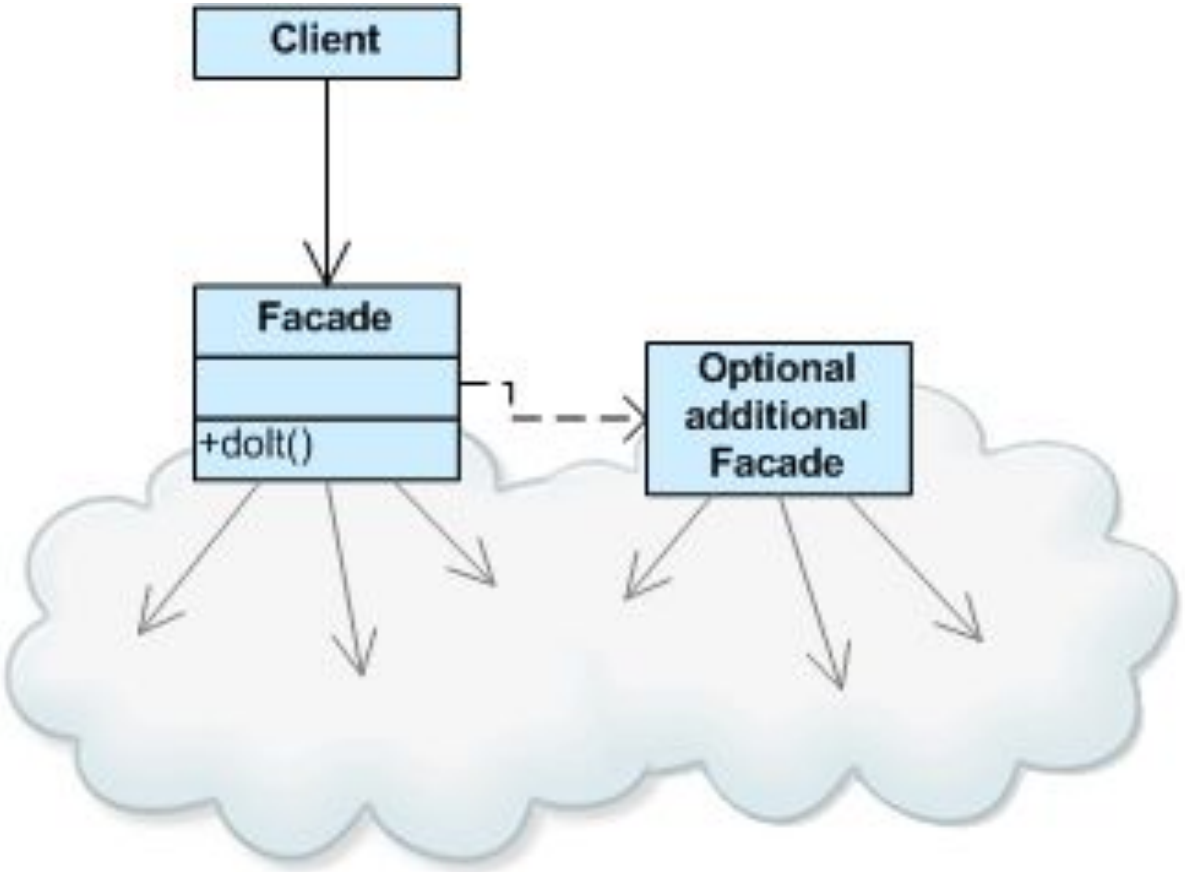
Паттерн Facade инкапсулирует сложную подсистему в единственный интерфейсный объект. Это позволяет сократить время изучения подсистемы, а также способствует уменьшению степени связанности между подсистемой и потенциально большим количеством клиентов. С другой стороны, если фасад является единственной точкой доступа к подсистеме, то он будет ограничивать возможности, которые могут понадобиться "продвинутым" пользователям.

Объект Facade, реализующий функции посредника, должен оставаться довольно простым и не быть всезнающим "оракулом".

Структура паттерна Facade:

Клиенты общаются с подсистемой через Facade. При получении запроса от клиента объект Facade переадресует его нужному компоненту подсистемы. Для клиентов

- **UML-диаграмма классов паттерна Facade**



Пример реализации паттерна Facade

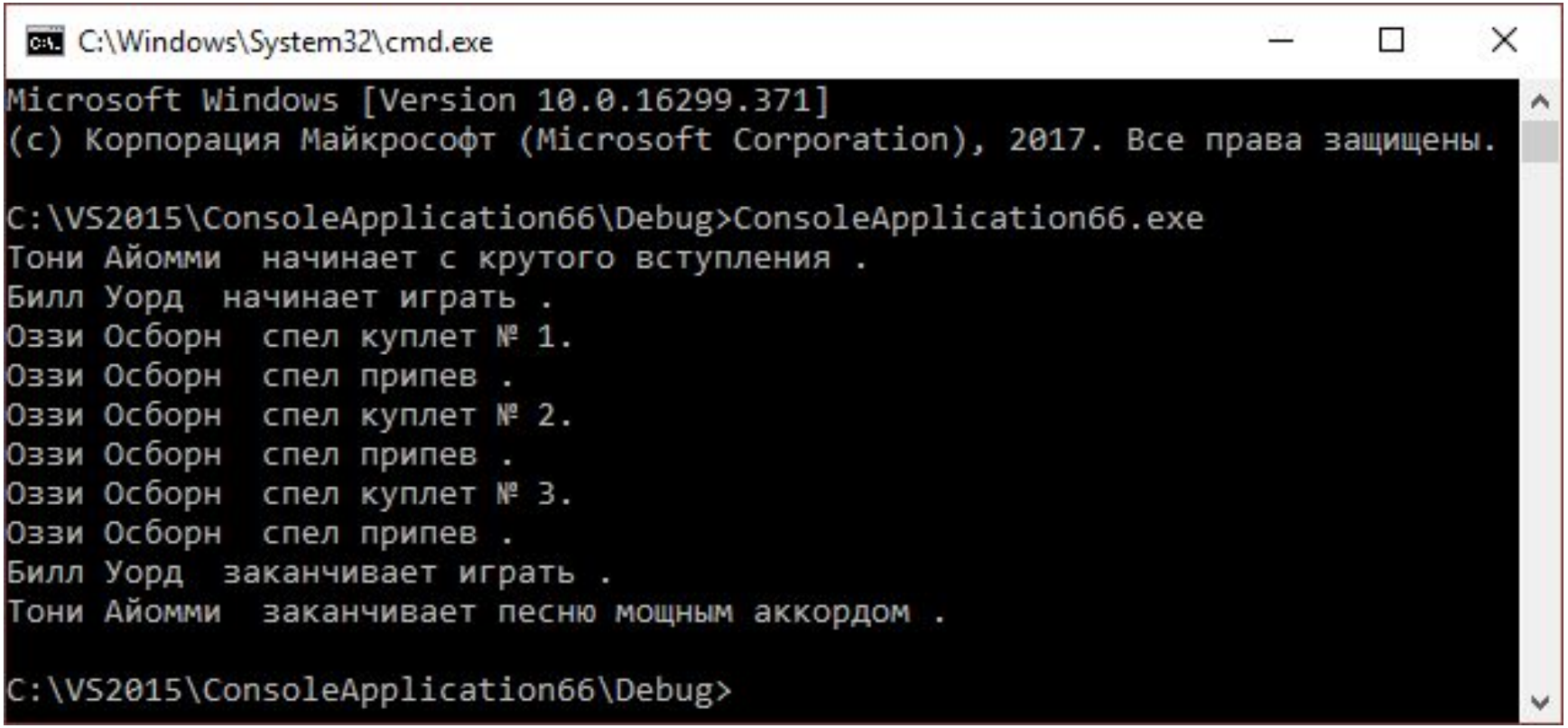
```
/** Абстрактный музыкант - не является обязательной составляющей паттерна, введен
для упрощения кода */
class Musician {
	std::string name;
public:
	Musician(const std::string &name) { this->name = name; }
	virtual ~Musician() {}
protected:
	void output(const std::string &text) { std::cout << this->name << " " << text << "." <<
std::endl; }
};
/** Конкретные музыканты */
class Vocalist : public Musician {
public:
	Vocalist(std::string &name) : Musician(name) {}
	void singCouplet(const int coupletNumber) { output(std::string(" спел куплет № ") +
std::to_string(coupletNumber)); }
	void singChorus() { output(std::string(" спел припев ")); }
};
class Guitarist : public Musician {
public:
	Guitarist(std::string &name) : Musician(name) {}
	void playCoolOpening() { output(std::string(" начинает с крутого вступления ")); }
	void playFinalAccord() { output(std::string(" заканчивает песню мощным аккордом "));}
};
```

Пример реализации паттерна Facade

```
class Drummer : public Musician {

public:
        Drummer(std::string &name) : Musician(name) {}

        void startPlaying() { output(std::string(" начинает играть "));         }

        void stopPlaying() { output(std::string(" заканчивает играть ")); }
};

/** Фасад, в данном случае - рок-группа */
class Band {

        Vocalist* vocalist;
        Guitarist* guitarist;
        Drummer* drummer;

public:

        Band() {    vocalist = new Vocalist(std::string("Оззи Осборн"));
                    guitarist = new Guitarist(std::string("Тони Айомми"));
                    drummer = new Drummer(std::string("Билл Уорд"));
                }

        ~Band() {  delete vocalist; delete guitarist; delete drummer; }

        void playCoolSong() { guitarist->playCoolOpening(); drummer->startPlaying();
                vocalist->singCouplet(1); vocalist->singChorus();
                vocalist->singCouplet(2);   vocalist->singChorus();
                vocalist->singCouplet(3);   vocalist->singChorus();
                drummer->stopPlaying();      guitarist->playFinalAccord();
```

Пример реализации паттерна Facade

```
int main()
{
    setlocale(LC_ALL, "rus");
    Band* band = new Band();
    band->playCoolSong();
    return 0;
}
```



Паттерн Фасад (Facade)

Использование паттерна Facade:

- Определите для подсистемы простой, унифицированный интерфейс.
- Спроектируйте класс "обертку", инкапсулирующий подсистему.
- Вся сложность подсистемы и взаимодействие ее компонентов скрыты от клиентов. "Фасад" / "обертка" переадресует пользовательские запросы подходящим методам подсистемы.
- Клиент использует только "фасад".
- Рассмотрите вопрос о целесообразности создания дополнительных "фасадов".

Особенности паттерна Facade:

Facade определяет новый интерфейс, в то время как Adapter использует уже имеющийся. Помните, Adapter делает работающими вместе два существующих интерфейса, не создавая новых.

Если Flyweight показывает, как сделать множество небольших объектов, то Facade показывает, как сделать один объект, представляющий целую подсистему.

Mediator похож на Facade тем, что абстрагирует функциональность существующих классов. Однако Mediator централизует функциональность между объектами-коллегами, не присущую ни одному из них. Коллеги обмениваются информацией друг с другом через Mediator. С другой стороны, Facade определяет простой интерфейс к подсистеме, не добавляет новой функциональности и не известен классам подсистемы.

Abstract Factory может применяться как альтернатива Facade для сокрытия платформенно-зависимых классов.

Объекты "фасадов" часто являются Singleton, потому что требуется только один объект Facade.

Adapter и Facade являются "обертками", однако эти "обертки" разных типов. Цель Facade – создание

Паттерн «Приспособленец» (FlyWeight)

Паттерн Flyweight использует разделение для эффективной поддержки большого числа мелких объектов.

Решаемая проблема:

Проектирование системы из объектов самого низкого уровня обеспечивает оптимальную гибкость, но может быть неприемлемо "дорогим" решением с точки зрения производительности и расхода памяти.

Паттерн Flyweight описывает, как совместно разделять очень мелкие объекты без чрезмерно высоких издержек. Каждый объект-приспособленец имеет две части: внутреннее и внешнее состояния.

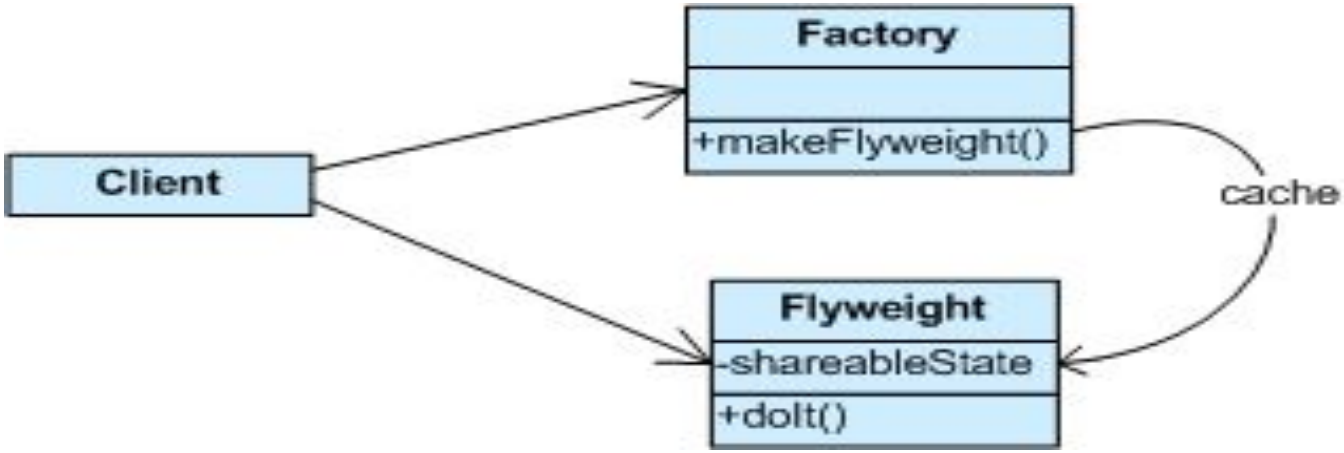
Внутреннее состояние хранится (разделяется) в приспособленце и состоит из информации, не зависящей от его контекста.

Внешнее состояние хранится или вычисляется объектами-клиентами и передается приспособленцу при вызове его методов.

Структура паттерна Flyweight:

Клиенты не создают приспособленцев напрямую, а запрашивают их у фабрики. Любые атрибуты (данные-члены класса), которые не могут разделяться, являются внешним состоянием. Внешнее состояние передается приспособленцу при вызове его методов. При этом наибольшая экономия памяти достигается в том случае, если внешнее состояние не хранится, а вычисляется при вызове.

- **UML-диаграмма классов паттерна Flyweight**



Использование паттерна Flyweight

- Убедитесь, что существует проблема повышенных накладных расходов.
- Разделите состояние целевого класса на разделяемое (внутреннее) и неразделяемое (внешнее).
- Удалите из атрибутов (членов данных) класса неразделяемое состояние и добавьте его в список аргументов, передаваемых методам.
- Создайте фабрику, которая может кэшировать и повторно использовать существующие экземпляры класса.
- Для создания новых объектов клиент использует эту фабрику вместо оператора new.
- Клиент (или третья сторона) должен находить или вычислять неразделяемое состояние и передавать его методам класса.

Пример реализации паттерна «приспособленец»

```
class Icon
{
public:
Icon(char *fileName) // приспособленец
{
        strcpy_s(_name, sizeof(_name), fileName);
        if (!strcmp(fileName, "go"))      {  _width = 20; _height = 20;  }
        if (!strcmp(fileName, "stop"))    {  _width = 40; _height = 40;  }
        if (!strcmp(fileName, "select")) {  _width = 60; _height = 60;  }
        if (!strcmp(fileName, "undo"))   { _width = 30; _height = 30;   }
}
const char *getName() { return _name; }
void draw(int x, int y)
{ cout << "   drawing " << _name << ": upper left (" << x << "," << y
 << ") - lower right (" << x + _width << "," << y + _height << ")" << endl;
}
private:
        char _name[20];
        int _width;
        int _height;
};
```

Пример реализации паттерна «приспособленец»

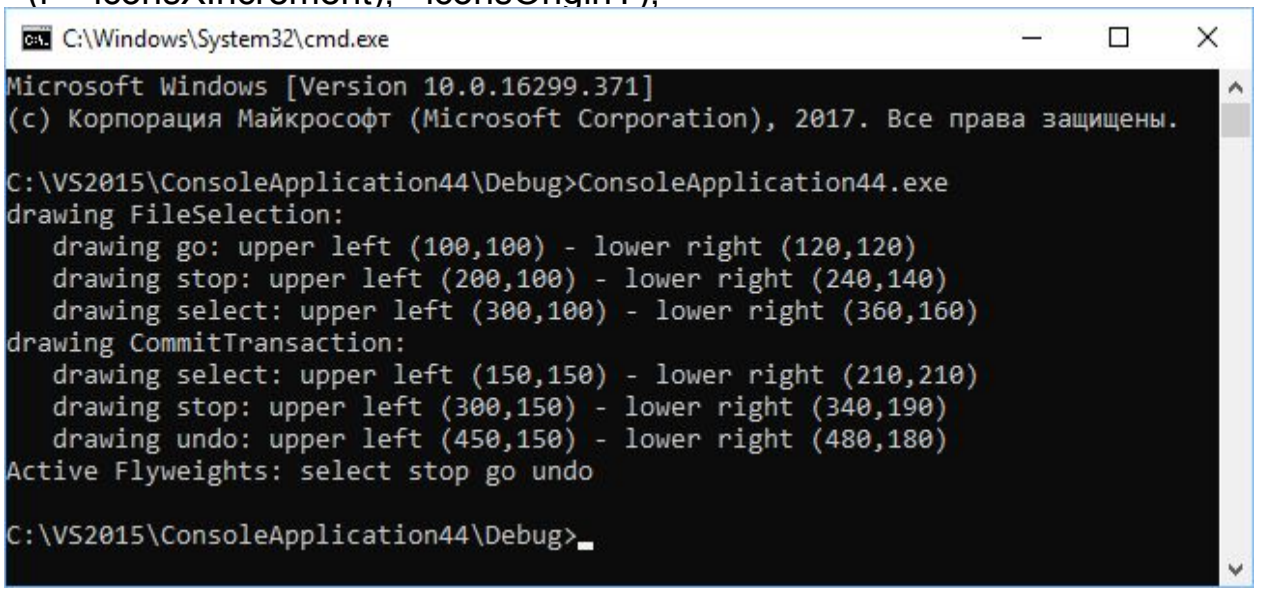
```
class FlyweightFactory
{
public:
	static Icon *getIcon(char *name)
	{
		for (int i = 0; i < _numIcons; i++)
			if (!strcmp(name, _icons[i]->getName()))
				return _icons[i];
		_icons[_numIcons] = new Icon(name);
		return _icons[_numIcons++];
	}
	static void reportTheIcons()
	{
		cout << "Active Flyweights: ";
		for (int i = 0; i < _numIcons; i++)
			cout << _icons[i]->getName() << " ";
		cout << endl;
	}
private:
	enum { MAX_ICONS = 5 };
	static int _numIcons;
	static Icon *_icons[MAX_ICONS];
};

int FlyweightFactory::_numIcons = 0;
Icon *FlyweightFactory::_icons[];
```

Пример реализации паттерна «приспособленец»

```
class DialogBox
{
public:
	DialogBox(int x, int y, int incr) : _iconsOriginX(x), _iconsOriginY(y), _iconsXIncrement(incr) {}
	virtual void draw() = 0;
protected:
	Icon *_icons[3];
	int _iconsOriginX;
	int _iconsOriginY;
	int _iconsXIncrement;
};

class FileSelection : public DialogBox
{
public:
	FileSelection(Icon *first, Icon *second, Icon *third) : DialogBox(100, 100, 100)
	{
		_icons[0] = first; _icons[1] = second; _icons[2] = third;
	}
	void draw()
	{
		cout << "drawing FileSelection:" << endl;
		for (int i = 0; i < 3; i++)
			_icons[i]->draw(_iconsOriginX + (i *_iconsXIncrement), _iconsOriginY);
	}
};
```

Пример реализации паттерна «приспособленец»

```
class CommitTransaction : public DialogBox
{
public:
	CommitTransaction(Icon *first, Icon *second, Icon *third) : DialogBox(150, 150, 150)
	{
		_icons[0] = first; _icons[1] = second; _icons[2] = third;
	}
	void draw()
	{
		cout << "drawing CommitTransaction:" << endl;
		for (int i = 0; i < 3; i++)
			_icons[i]->draw(_iconsOriginX + (i *  iconsXIncrement),   iconsOriginY);
	}
};

int main()
{
	DialogBox *dialogs[2];
	dialogs[0] = new FileSelection(
		FlyweightFactory::getIcon("go"),
		FlyweightFactory::getIcon("stop"),
		FlyweightFactory::getIcon("select"));

	dialogs[1] = new CommitTransaction(
		FlyweightFactory::getIcon("select"),
		FlyweightFactory::getIcon("stop"),
		FlyweightFactory::getIcon("undo"));

	for (int i = 0; i < 2; i++)
		dialogs[i]->draw();

	FlyweightFactory::reportTheIcons();
}
```



Паттерн «Приспособленец» (FlyWeight)

Паттерн Flyweight показывает, как эффективно разделять множество мелких объектов. Ключевая концепция - различие между внутренним и внешним состояниями.

Внутреннее состояние состоит из информации, которая не зависит от контекста и может разделяться (например, имя иконки, ее ширина и высота). Оно хранится в приспособленце (то есть в классе Icon).

Внешнее состояние не может разделяться, оно зависит от контекста и изменяется вместе с ним (например, координаты верхнего левого угла для каждого экземпляра иконки).

Внешнее состояние хранится или вычисляется клиентом и передается приспособленцу при вызове операций. Клиенты не должны создавать экземпляры приспособленцев напрямую, а получать их исключительно из объекта FlyweightFactory для правильного разделения.

Особенности паттерна Flyweight:

Если Flyweight показывает, как сделать множество небольших объектов, то Facade показывает, как представить целую подсистему одним объектом.

Flyweight часто используется совместно с Composite для реализации иерархической структуры в виде графа с разделяемыми листовыми вершинами.

Терминальные символы абстрактного синтаксического дерева Interpreter могут разделяться при помощи Flyweight.

Flyweight объясняет, когда и как могут разделяться объекты State.

Паттерн «прокси» (Proxy) (заместитель, суррогат)

Назначение паттерна Proxy:

Паттерн Proxy является суррогатом или заместителем другого объекта и контролирует доступ к нему.

Предоставляя дополнительный уровень косвенности при доступе к объекту, может применяться для поддержки распределенного, управляемого или интеллектуального доступа.

Являясь "оберткой" реального компонента, защищает его от излишней сложности.

Решаемая проблема:

Вам нужно управлять ресурсоемкими объектами. Вы не хотите создавать экземпляры таких объектов до момента их реального использования.

Существует четыре ситуации, когда можно использовать паттерн Proxy:

- Виртуальный proxy является заместителем объектов, создание которых обходится дорого. Реальный объект создается только при первом запросе/доступе клиента к объекту.
- Удаленный proxy предоставляет локального представителя для объекта, который находится в другом адресном пространстве ("заглушки" в RPC).
- Защитный proxy контролирует доступ к основному объекту. "Суррогатный" объект предоставляет доступ к реальному объекту, только когда вызывающий объект имеет соответствующие права.
- Интеллектуальный proxy выполняет дополнительные действия при доступе к объекту.

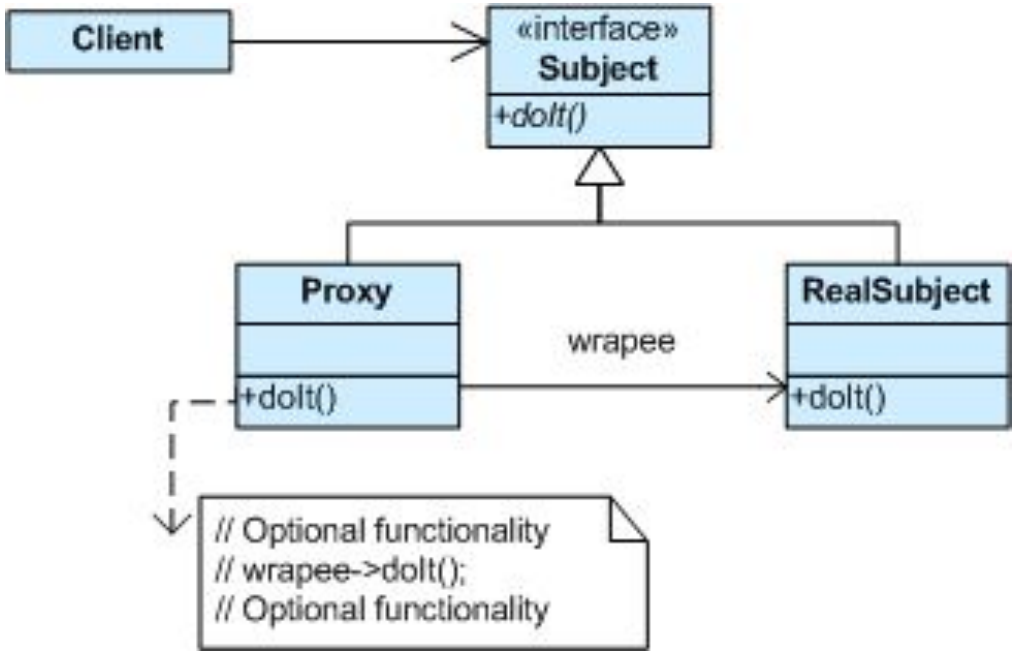
Типичные области применения интеллектуальных proxy:

Подсчет числа ссылок на реальный объект. При отсутствии ссылок память под объект автоматически освобождается (известен также как интеллектуальный указатель или smart pointer).

Загрузка объекта в память при первом обращении к нему.

Установка запрета на изменение реального объекта при обращении к нему других объектов.

- **UML-диаграмма классов паттерна Proxy**

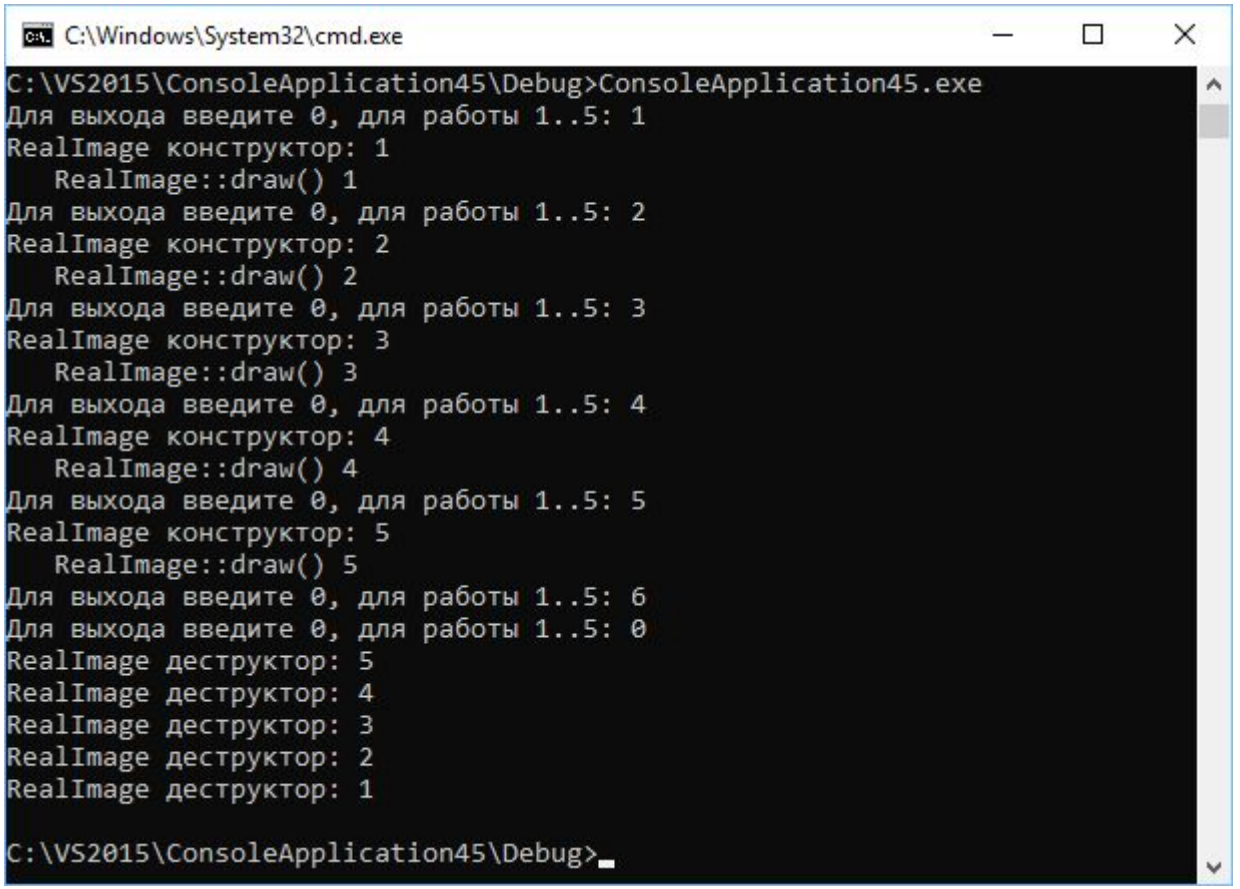


Пример реализации паттерна «прокси»

```
class RealImage
{
	int m_id;

public:
	RealImage(int i)	{  m_id = i;	cout << "RealImage конструктор: " << m_id << endl;  }
	~RealImage()	{  cout << "RealImage деструктор: " << m_id << endl;	}
	void draw()	{  cout << "   RealImage::draw() " << m_id << endl;	}
};

// 1. Класс-обертка с "дополнительным уровнем косвенности"
class Image	// Proxy - класс
{
	// 2. Класс-обертка содержит закрытый указатель на реальный класс
	RealImage *ptrRealImage;
	int m_id;
	static int s_next;
public:
	Image()
	{
		m_id = s_next++;
		// 3. Инициализируется нулевым значением
		ptrRealImage = nullptr;
	}
	~Image() { delete ptrRealImage;  }
	void draw()
	{
		// 4. Реальный объект создается при поступлении
		//    запроса "на первом использовании"
		if (!ptrRealImage)
			ptrRealImage = new RealImage(m_id);
		// 5. Запрос всегда делегируется реальному объекту
		ptrRealImage->draw();
	}
};

int Image::s_next = 1;
```

Пример реализации паттерна «прокси»

```
int main()
{
	setlocale(LC_ALL, "rus");
	Image images[5];

	for (int i; true;)
	{
		cout << "Для выхода введите 0, для работы 1..5: ";
		cin >> i;
		if (i == 0)
			break;
		if (i < 6)
			images[i - 1].draw();
	}
}
```



Паттерн «прокси» (Proxy) (заместитель, суррогат)

Использование паттерна Proxy:

- Определите ту часть системы, которая лучше всего реализуется через суррогата.
- Определите интерфейс, который сделает суррогата и оригинальный компонент взаимозаменяемыми.
- Рассмотрите вопрос об использовании фабрики, инкапсулирующей решение о том, что желательно использовать на практике: оригинальный объект или его суррогат.
- Класс суррогата содержит указатель на реальный объект и реализует общий интерфейс.
- Указатель на реальный объект может инициализироваться в конструкторе или при первом использовании.
- Методы суррогата выполняют дополнительные действия и вызывают методы реального объекта.

Особенности паттерна Proxy:

Adapter предоставляет своему объекту другой интерфейс . Proxy предоставляет тот же интерфейс. Decorator предоставляет расширенный интерфейс.

Decorator и Proxy имеют разные цели, но схожие структуры. Оба вводят дополнительный уровень косвенности: их реализации хранят ссылку на объект, на который они отправляют запросы.