Паттерны поведения

Паттерны поведения определяют алгоритмы и способы реализации взаимодействия различных объектов и классов. Они обеспечивают гибкость взаимодействия между объектами.

№	Название паттерна	Перевод	Назначение паттерна
1	Chain of Responsibility	Цепочка обязанностей	Позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом объекты-получатели связываются в цепочку, а запрос передается по цепочке, пока какой-то объект его не обработает.
2	Command	Команда	Инкапсулирует запрос в виде объекта, обеспечивая параметризацию клиентов типом запроса, установление очередности запросов, протоколирование запросов и отмену выполнения операций.
3	Interpreter	Интерпретатор	Для заданного языка определяет представление его грамматики на основе интерпретатора предложений языка, использующего это представление.
4	Iterator	Итератор	Дает возможность последовательно перебрать все элементы составного объекта, не раскрывая его внутреннего представления.
5	Mediator	Посредник	Определяет объект, в котором инкапсулировано знание о том, как взаимодействуют объекты из некоторого множества. Способствует уменьшению числа связей между объектами, позволяя им работать без явных ссылок друг на друга и независимо изменять схему взаимодействия.
6	Memento	Хранитель	Дает возможность получить и сохранить во внешней памяти внутреннее состояние объекта, чтобы позже объект можно было восстановить точно в таком же состоянии, не нарушая принципа инкапсуляции.

Паттерны поведения (продолжение)

№	Название паттерна	Перевод	Назначение паттерна
7	Observer	Наблюдатель	Специфицирует зависимость типа "один ко многим" между различными объектами, так что при изменении состояния одного объекта все зависящие от него получают извещение и автоматически обновляются.
8	State	Состояние	Позволяет выбранному объекту варьировать свое поведение при изменении внутреннего состояния. При этом создается впечатление, что изменился класс объекта.
9	Strategy	Стратегия	Определяет множество алгоритмов, инкапсулируя их все и позволяя подставлять один вместо другого. При этом можно изменять алгоритм независимо от клиента, который им пользуется.
10	Template Method	Шаблонный метод	Определяет структуру алгоритма, перераспределяя ответственность за некоторые его шаги на подклассы. При этом подклассы могут переопределять шаги алгоритма, не меняя его общей структуры.
11	Visitor	Посетитель	Позволяет определить новую операцию, не меняя описаний классов, у объектов которых она вызывается.

Паттерн «Команда» (Command)

– ***Паттерн Command используется, если:***

- Система управляется событиями. При появлении такого события (запроса) необходимо выполнить определенную последовательность действий.
- Необходимо параметризировать объекты выполняемым действием, ставить запросы в очередь или поддерживать операции отмены (undo) и повтора (redo) действий.
- Нужен объектно-ориентированный аналог функции обратного (callback) вызова в процедурном программировании.

– ***Описание паттерна Command***

Паттерн ***Command*** преобразовывает запрос на выполнение действия в отдельный ***объект-команду***. Этот объект запроса на действие и называется командой.

При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию.

Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

В паттерне Command может быть до трех участников:

- ***клиент,*** создающий экземпляр командного объекта;
- ***инициатор запроса,*** использующий командный объект;
- ***получатель запроса*.**

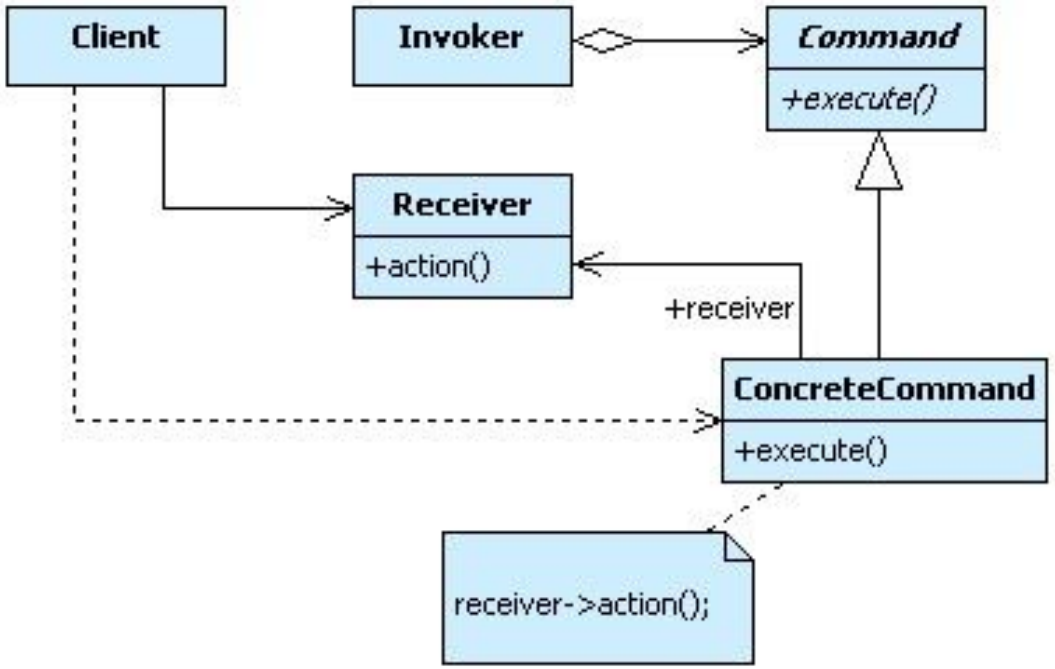
Паттерн ***Command*** отделяет объект, инициирующий операцию, от объекта, который знает, как ее выполнить. Единственное, что должен знать инициатор, это как отправить команду.

Это придает системе гибкость: позволяет осуществлять динамическую замену команд, использовать сложные составные команды, осуществлять отмену операций.

Достоинства паттерна Command

Придает системе гибкость, отделяя инициатора запроса от его получателя.

UML-диаграмма классов паттерна «Команда» (Command)



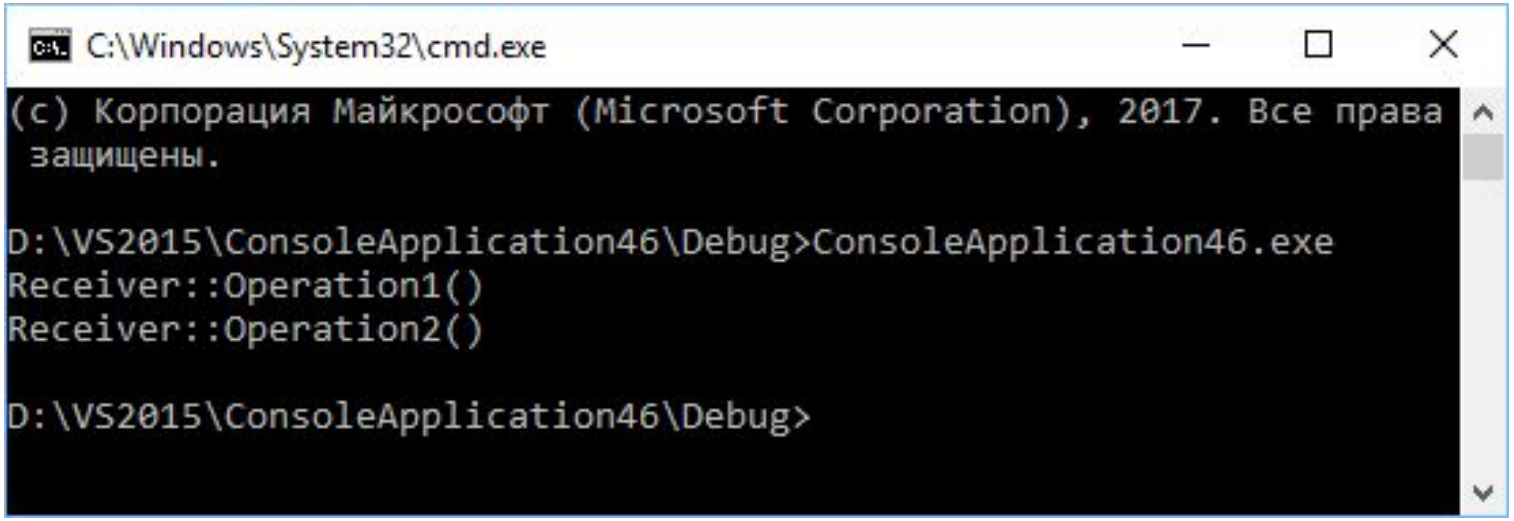
Пример реализации паттерна «Команда» (Command)

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

// получатель команды
class Receiver
{
  public:
   void Operaiton1()  { cout << "Receiver::Operation1() " << endl;  }
   void Operaiton2()  { cout << "Receiver::Operation2() " << endl;  }
};

class Command // Базовый класс для объектов — команд
{
  public:
    virtual ~Command() {}
    virtual void Execute()=0;

  protected:
    Command(Receiver *p) : ptrReceiver(p) {}
    Receiver *ptrReceiver;
};

// конкретная команда
class ConcreteCommand1 : public Command
{
  public:
    ConcreteCommand1(Receiver *p) : Command(p) {}

    void Execute()
   {
    ptrReceiver->Operaiton1();
   }
};
```

Пример реализации паттерна «Команда» (Command) (продолжение)

```
class ConcreteCommand2 : public Command
{
  public:
    ConcreteCommand2(Receiver *p) : Command(p) {}

    void Execute() { ptrReceiver→Operaiton2(); }
};

// инициатор команды
class Invoker
{
    Command *ptrCommand;

  public:
    void SetCommand(Command *ptrC) {  ptrCommand = ptrC;  }
    void Run()  {  ptrCommand->Execute(); }
};

//class Client
int main()
{
  Receiver receiver;
  Invoker invoker;
  Command *command1 = new ConcreteCommand1(&receiver);
  Command *command2 = new ConcreteCommand2(&receiver);

  invoker.SetCommand(command1);
  invoker.Run();

  invoker.SetCommand(command2);
  invoker.Run();
```

Результат работы программы:



Паттерн «Итератор» (Iterator)

Назначение паттерна Iterator

Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления. Пример: абстракция в стандартных библиотеках C++ и Java, позволяющая разделить классы коллекций и алгоритмов.

Придает обходу коллекции "объектно-ориентированный статус".

Полиморфный обход.

Решаемая проблема

Предлагается реализация механизма "абстрактного" обхода различных структур данных так, что могут определяться алгоритмы, способные взаимодействовать со структурами прозрачно.

Структура паттерна Iterator

Для манипулирования коллекцией клиент использует открытый интерфейс класса Collection. Однако доступ к элементам коллекции *инкапсулируется* дополнительным уровнем абстракции, называемым ***Iterator***. Каждый производный от Collection класс знает, какой производный от ***Iterator*** класс нужно создавать и возвращать. После этого клиент использует интерфейс, определенный в базовом классе ***Iterator***.

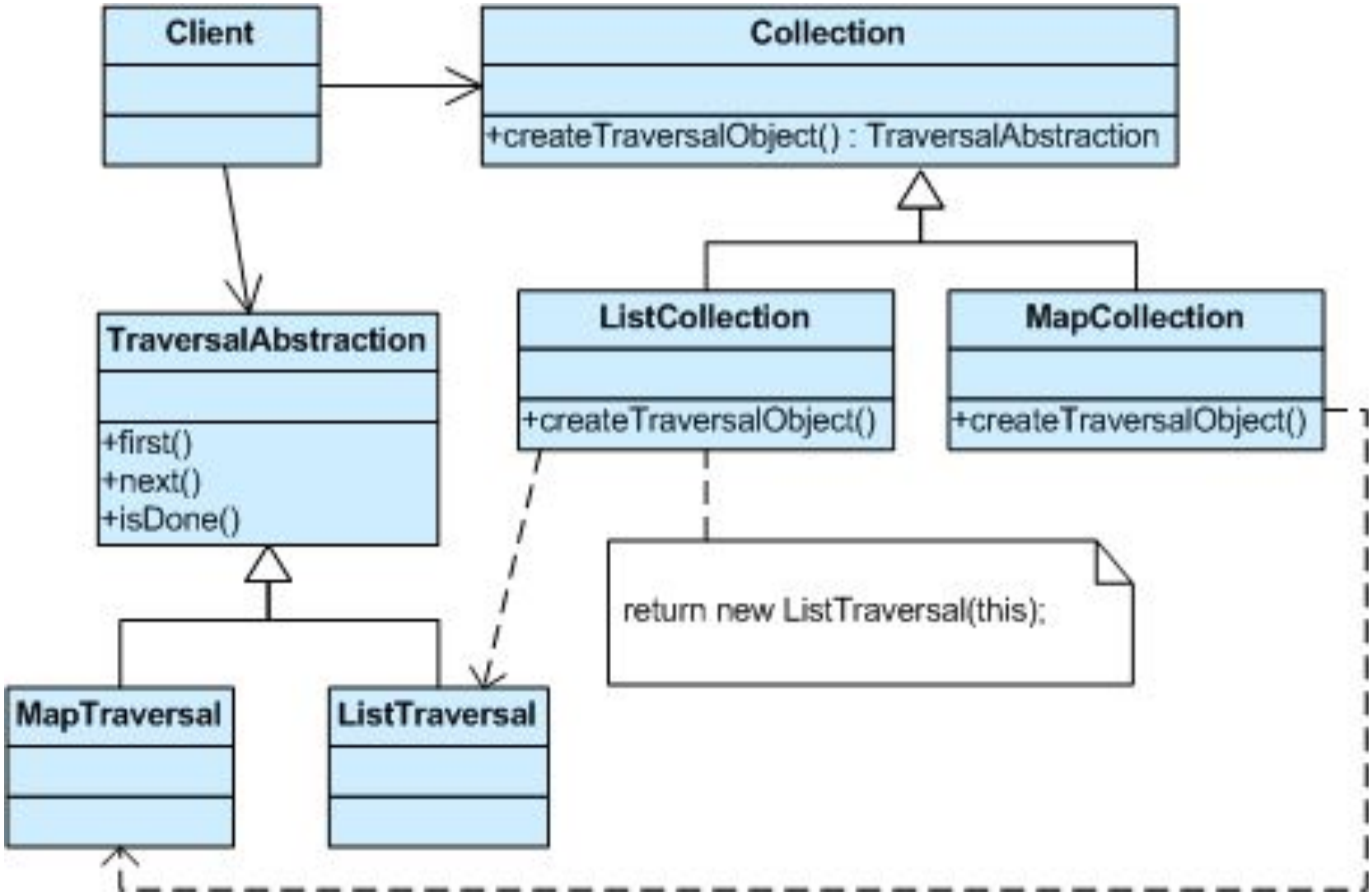
Особенности паттерна Iterator

Iterator может применяться для обхода сложных структур, создаваемых с помощью «компоновщика» Composite.

Для создания экземпляра подкласса Iterator полиморфные итераторы используют «фабричный метод» (Factory Method).

Часто «Хранитель» (Memento) и Iterator используются совместно. Iterator может использовать «хранителя» (Memento) для сохранения состояния итерации и содержит его внутри себя.

UML-диаграмма классов паттерна Iterator



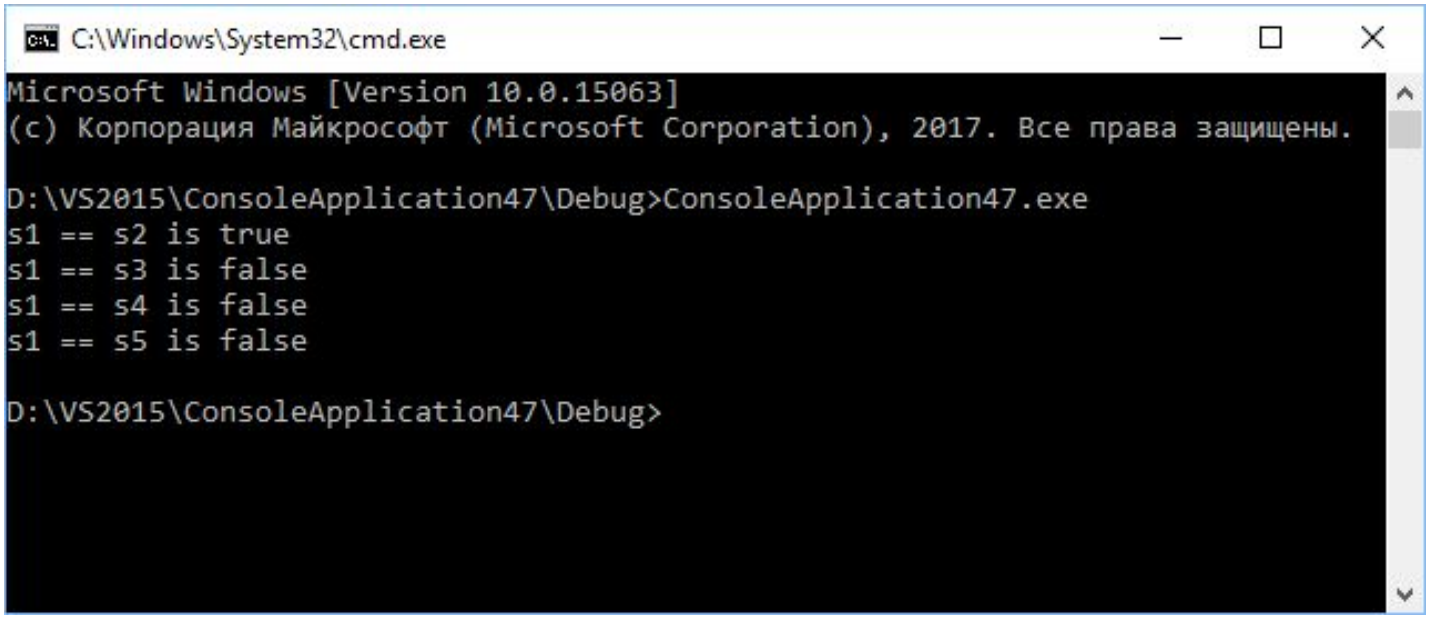
Пример реализации паттерна «Итератор» (Iterator)

```
#include <iostream>
using namespace std;
class Stack
{
        int items[10];
        int sp;
public:
        Stack()                 { sp = -1; }
        void push(int in){ items[++sp] = in; }
        int pop()               { return items[sp--]; }
        bool isEmpty()          { return (sp == -1); }
        int size()    const{ return (sp + 1); }
        friend class StackIter; // разрешаю итератору доступ к закрытым членам класса Stack
        StackIter* createIterator() const;
};
class StackIter  // класс "iterator"
{
        const Stack *stk;
        int index;
public:
        StackIter(const Stack *s) { stk = s; }
        void first()            { index = 0; }
        void next()             { index++; }
        bool isDone()           { return index == stk->sp + 1; }
        int currentItem(){ return stk->items[index]; }
};
```

Пример реализации паттерна «Итератор» (Iterator) продолжение

```
StackIter *Stack::createIterator() const { return new StackIter(this); }
bool operator == (const Stack &l, const Stack &r)  // перегруженный оператор ==
{
        if (l.size() != r.size())  return false;
        // Клиенты запрашивают создание объекта StackIter у объекта Stack
        StackIter *itl = l.createIterator();
        StackIter *itr = r.createIterator();
        // Клиенты используют first(), isDone(), next(), and currentItem()
        for (itl->first(), itr->first(); !itl->isDone(); itl->next(), itr->next())
                if (itl->currentItem() != itr→currentItem())  break;
        bool ans = itl->isDone() && itr->isDone();
        delete itl; delete itr;
        return ans;
}
int main()
{
        Stack s1;
        for (int i = 1; i < 6; i++)  s1.push(i);  // формирую стек из 5 элементов [5,4,3,2,1]
        Stack s2(s1), s3(s1), s4(s1), s5(s1); // создаю копии стека s1 в s2, s3, s4, s5
        s3.pop();    // удаляю первый элемент из s3 [4,3,2,1]
        s5.pop();    // удаляю первый элемент из s5 [4,3,2,1]
        s4.push(2);          // добавляю 2 в s4 [2,5,4,3,2,1]
        s5.push(9);          // добавляю 9 в s5 [9,4,3,2,1]
        cout << "s1 == s2 is " << ((s1 == s2) ? "true" : "false") << endl;
        cout << "s1 == s3 is " << ((s1 == s3) ? "true" : "false") << endl;
        cout << "s1 == s4 is " << ((s1 == s4) ? "true" : "false") << endl;
        cout << "s1 == s5 is " << ((s1 == s5) ? "true" : "false") << endl;
}
```

Результат работы программы:



Паттерн «Посредник» (Mediator)

Назначение паттерна Mediator

Паттерн Mediator определяет объект, инкапсулирующий взаимодействие множества объектов.

Mediator делает систему слабо связанной, избавляя объекты от необходимости ссылаться друг на друга, что позволяет изменять взаимодействие между ними независимо.

Паттерн Mediator вводит посредника для развязывания множества взаимодействующих объектов.

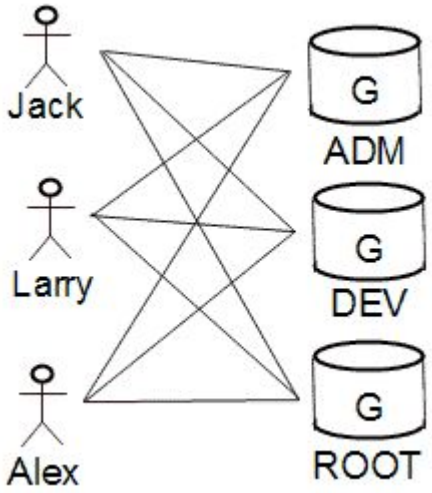
Заменяет взаимодействие "все со всеми" взаимодействием "один со всеми".

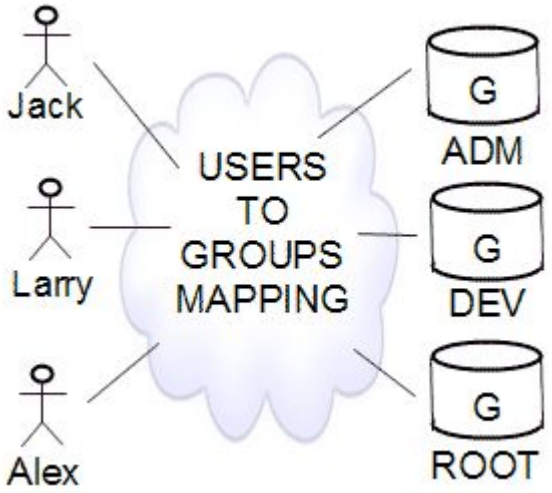
Решаемая проблема

Мы хотим спроектировать систему с повторно используемыми компонентами, однако существующие связи между этими компонентами можно охарактеризовать феноменом "спагетти-кода".

Спагетти-код - плохо спроектированная, слабо структурированная, запутанная и трудная для понимания программа. Спагетти-код назван так, потому что ход выполнения программы похож на миску спагетти, то есть извилистый и запутанный.

Паттерн «Посредник» (Mediator)

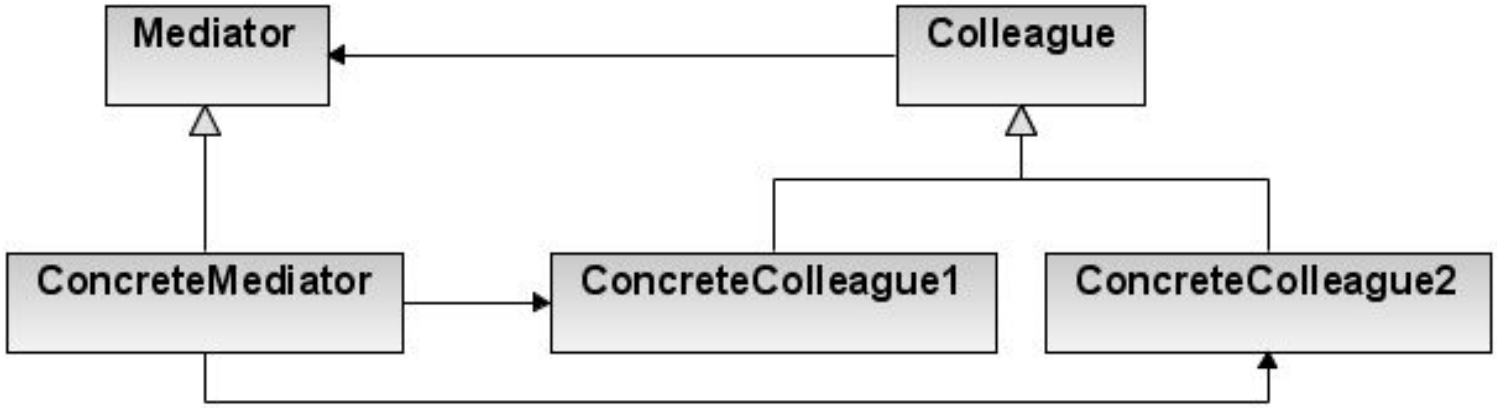




Если нам нужно было бы построить программную модель такой системы, то мы могли бы связать каждый объект User с каждым объектом Group, а каждый объект Group - с каждым объектом User. Однако из-за наличия множества взаимосвязей модифицировать поведение такой системы очень непросто, пришлось бы изменять все существующие классы.

Альтернативный подход - введение "дополнительного уровня косвенности" или построение абстракции из отображения (соответствия) пользователей в группы и групп в пользователей. Такой подход обладает следующими преимуществами: пользователи и группы отделены друг от друга, отображениями легко управлять одновременно и абстракция отображения может быть расширена в будущем путем определения производных классов.

UML-диаграмма классов паттерна «Посредник» (Mediator)



Mediator – "Посредник"
ConcreteMediator – "Конкретный посредник"
Классы Colleague – "Коллеги"

Использование паттерна Mediator

Определите совокупность взаимодействующих объектов, связанность между которыми нужно уменьшить.

Инкапсулируйте все взаимодействия в абстракцию нового класса.

Создайте экземпляр этого нового класса. Объекты-коллеги для взаимодействия друг с другом используют только этот объект.

Найдите правильный баланс между принципом слабой связанности и принципом распределения ответственности.

Будьте внимательны и не создавайте объект-"контроллер" вместо объекта-посредника.

Особенности паттерна Mediator

Паттерны “Цепочка обязанностей» (Chain of Responsibility), «Команда» (Command), «Посредник» (Mediator) и «Наблюдатель» (Observer) показывают, как можно разделить отправителей и получателей запросов с учетом их особенностей. Chain of Responsibility передает запрос отправителя по цепочке потенциальных получателей. Command номинально определяет связь - "оправитель-получатель" с помощью подкласса. В Mediator отправитель и получатель ссылаются друг на друга косвенно, через объект-посредник. В паттерне Observer связь между отправителем и получателем слабее, при этом число получателей может конфигурироваться во время выполнения.

Mediator и Observer являются конкурирующими паттернами. Если Observer распределяет взаимодействие с помощью объектов "наблюдатель" и "субъект", то Mediator использует объект-посредник для инкапсуляции взаимодействия между другими объектами. Легче сделать повторно используемыми Наблюдателей и Субъектов, чем Посредников.

С другой стороны, Mediator может использовать Observer для динамической регистрации коллег и их взаимодействия с посредником.

Mediator похож Facade в том, что он абстрагирует функциональность существующих классов. Mediator абстрагирует/централизует взаимодействие между объектами-коллегами, добавляет новую функциональность и известен всем объектам-коллегам (то есть определяет двунаправленный протокол взаимодействия). Facade, наоборот, определяет более простой интерфейс к подсистеме, не добавляя новой функциональности, и неизвестен классам подсистемы (то есть имеет однонаправленный протокол взаимодействия, то есть запросы отправляются в подсистему, но не наоборот).

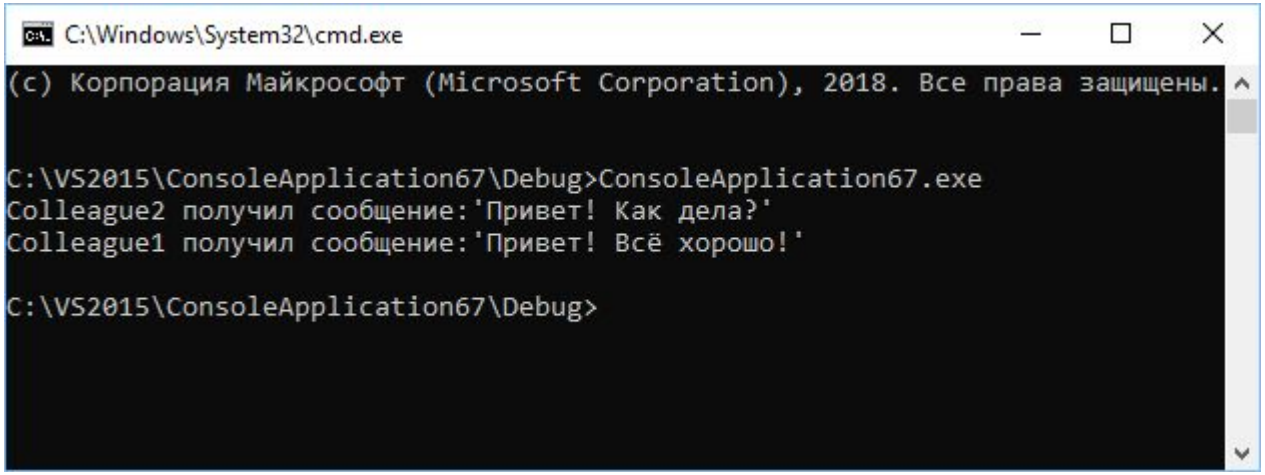
Пример реализации паттерна «Посредник» (Mediator)

```
#include <iostream>
#include <string>

class Colleague;
class Mediator;
class ConcreteMediator;
class ConcreteColleague1;
class ConcreteColleague2;

class Mediator
{
public:
	virtual void Send(std::string const& message, Colleague *colleague) const = 0;
};
class Colleague
{
protected:
	Mediator* mediator_;
public:
	explicit Colleague(Mediator *mediator) :mediator_(mediator) { }
};

class ConcreteColleague1 :public Colleague
{
public:
	explicit ConcreteColleague1(Mediator* mediator) :Colleague(mediator) {}

	void Send(std::string const& message) { mediator_->Send(message, this); }

	void Notify(std::string const& message)  {
		std::cout << "Colleague1 получил сообщение:'" << message << "'" << std::endl;
	 }
};
```

Пример реализации паттерна «Посредник» (Mediator) продолжение

```
class ConcreteColleague2 :public Colleague
{
public:
	explicit ConcreteColleague2(Mediator *mediator) :Colleague(mediator) {}
	void Send(std::string const& message) { mediator_->Send(message, this); }
	void Notify(std::string const& message) { std::cout << "Colleague2 получил сообщение:'" << message << "'" << std::endl;
}
};
class ConcreteMediator :public Mediator
{
protected:
	ConcreteColleague1	*m_Colleague1;
	ConcreteColleague2	*m_Colleague2;
public:
	void SetColleague1(ConcreteColleague1 *c) { m_Colleague1 = c; }
	void SetColleague2(ConcreteColleague2 *c) { m_Colleague2 = c; }
	virtual void Send(std::string const& message, Colleague *colleague) const
	{
		if (colleague == m_Colleague1)  m_Colleague2->Notify(message);
		else
		if (colleague == m_Colleague2)  m_Colleague1->Notify(message);
	}
};
int main()
{
	setlocale(LC_ALL, "rus");
	ConcreteMediator m;
	ConcreteColleague1 c1(&m);
	ConcreteColleague2 c2(&m);

	m.SetColleague1(&c1);
	m.SetColleague2(&c2);

	c1.Send("Привет! Как дела?");
	c2.Send("Привет! Всё хорошо!");
```

Результат работы программы:



Паттерн “Хранитель» (Memento)

Назначение паттерна Memento

Не нарушая инкапсуляции, паттерн Memento получает и сохраняет за пределами объекта его внутреннее состояние так, чтобы позже можно было восстановить объект в таком же состоянии.

Является средством для инкапсуляции "контрольных точек" программы.

Паттерн Memento придает операциям "Отмена" (undo) или "Откат" (rollback) статус "полноценного объекта".

Решаемая проблема

Вам нужно восстановить объект обратно в прежнее состояние (те есть выполнить операции "Отмена" или "Откат").

Обсуждение паттерна Memento

Клиент запрашивает Memento (хранителя) у исходного объекта, когда ему необходимо сохранить состояние исходного объекта (установить контрольную точку).

Исходный объект инициализирует Memento своим текущим состоянием. Клиент является "посыльным" за Memento, но только исходный объект может сохранять и извлекать информацию из Memento (Memento является "непрозрачным" для клиентов и других объектов).

Если клиенту в дальнейшем нужно "откатить" состояние исходного объекта, он передает Memento обратно в исходный объект для его восстановления.

Использование паттерна Memento

Определите роли "смотрителя" и "хозяина".

Создайте класс Memento и объявите хозяина другом.

Смотритель знает, когда создавать "контрольную точку" хозяина.

Хозяин создает хранителя Memento и копирует свое состояние в этот Memento.

Смотритель сохраняет хранителя Memento (но смотритель не может заглянуть в Memento).

Смотритель знает, когда нужно "откатить" хозяина.

Хозяин восстанавливает себя, используя сохраненное в Memento состояние.

Особенности паттерна Memento

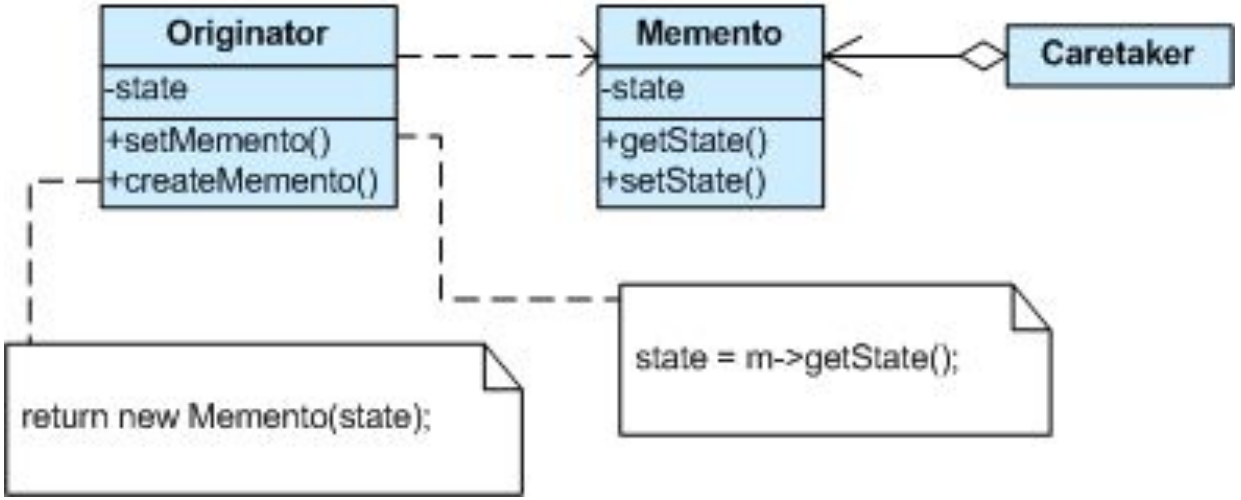
Паттерны Command и Memento определяют объекты "волшебная палочка", которые передаются от одного владельца к другому и используются позднее.В Command такой "волшебной палочкой" является запрос,в Memento - внутреннее состояние объекта в некоторый момент времени.

Полиморфизм важен для Command, но не важен для Memento потому, что интерфейс Memento настолько "узкий", что его можно передавать как значение.

Command может использовать Memento для сохранения состояния, необходимого для выполнения отмены действий.

Memento часто используется совместно с Iterator. Iterator может использовать Memento для сохранения состояния итерации.

UML-диаграмма классов паттерна «Хранитель» (Memento)



Паттерн проектирования Memento определяет трех различных участников:
Originator (хозяин) - объект, умеющий создавать хранителя, а также знающий, как восстановить свое внутреннее состояние из хранителя.
Caretaker (смотритель) - объект, который знает, почему и когда хозяин должен сохранять и восстанавливать себя.
Memento (хранитель) - "ящик на замке", который пишется и читается хозяином и за которым присматривает смотритель.

Пример реализации паттерна Memento

```
#include <iostream>
using namespace std;
class Number; // опережающее объявление класса

class Memento
{
public:
Memento(int val) {     _state = val;      }
private:
friend class Number;
int _state;
};

class Number
{
public:
Number(int value)     {       _value = value; }
void dubble()         {       _value = 2 * _value; }
void half()           {       _value = _value / 2; }
int getValue()        {       return _value;    }
Memento *createMemento() {     return new Memento(_value); }
void reinstateMemento(Memento *mem) { _value = mem->_state; }
private:
int _value;
};
```

Пример реализации паттерна Memento

Memento - это объект, хранящий "снимок" внутреннего состояния другого объекта.

Memento может использоваться для поддержки "многоуровневой" отмены действий паттерна Command.

В этом примере перед выполнением команды по изменению объекта Number, текущее состояние этого объекта сохраняется в статическом списке истории хранителей Memento, а сама команда сохраняется в статическом списке истории команд.

Undo() просто восстанавливает состояние объекта Number, получаемое из списка истории хранителей. Redo() использует список истории команд.

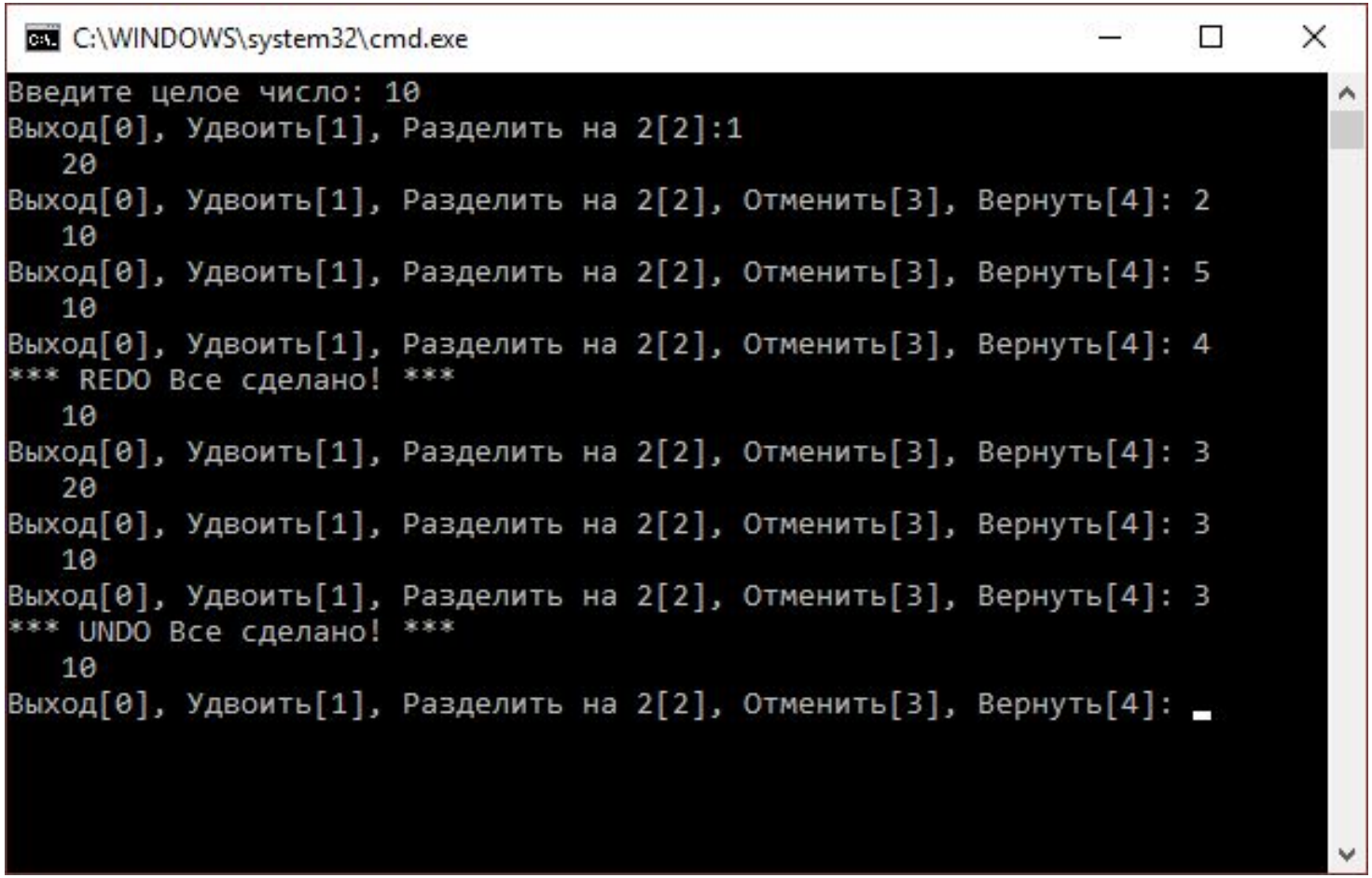
Пример реализации паттерна Memento (продолжение)

```
class Command
{
public:
	typedef void(Number:: *Action)();
	Command(Number *receiver, Action action)
	{
		_receiver = receiver;  _action = action;
	}
	virtual void execute()
	{
		_mementoList[_numCommands] = _receiver->createMemento();
		_commandList[_numCommands] = this;
		if (_numCommands > _highWater)
			_highWater = _numCommands;
		_numCommands++;
		(_receiver->*_action)();
	}
	static void undo()
	{
		if (_numCommands == 0) { cout << "*** UNDO Все сделано! ***" << endl; return; }
		_commandList[_numCommands - 1]->_receiver->reinstateMemento(_mementoList[_numCommands - 1]);
		_numCommands--;
	}
	void static redo()
	{
		If (_numCommands > _highWater) { cout << "*** REDO Все сделано! ***" << endl; return; }
		(_commandList[_numCommands]->_receiver->*(_commandList[_numCommands]->_action))();
		_numCommands++;
	}
protected:
	Number *_receiver;
	Action _action;
	static Command *_commandList[20];
	static Memento *_mementoList[20];
	static int _numCommands;
```

Пример реализации паттерна Memento (продолжение)

```
Command *Command::_commandList[];
Memento *Command::_mementoList[];
int Command::_numCommands = 0;
int Command::_highWater = 0;

int main()
{
	int i;
	setlocale(LC_ALL, "rus");
	cout << "Введите целое число: ";
	cin >> i;
	Number *object = new Number(i);

	Command *commands[3];
	commands[1] = new Command(object, &Number::dubble);
	commands[2] = new Command(object, &Number::half);

	cout << "Выход[0], Удвоить[1], Разделить на 2[2]:";
	cin >> i;

	while (i)
	{
		switch (i) {
		case 0: break;
		case 1:
		case 2:		commands[i]->execute();break;
		case 3:		Command::undo(); break;
		case 4:		Command::redo();
			break;
		default: break;
		}
		cout << "    " << object->getValue() << endl;
		cout << "Выход[0], Удвоить[1], Разделить на 2[2], Отменить[3], Вернуть[4]: ";
		cin >> i;
	}
}
```

Результат работы программы:



Паттерн Observer (наблюдатель, издатель-подписчик)

Назначение паттерна Observer

Паттерн Observer определяет зависимость "один-ко-многим" между объектами так, что при изменении состояния одного объекта все зависящие от него объекты уведомляются и обновляются автоматически.

Паттерн Observer инкапсулирует главный (независимый) компонент в абстракцию Subject и изменяемые (зависимые) компоненты в иерархию Observer.

Паттерн Observer определяет часть "View" в модели Model-View-Controller (MVC) .

Решаемая проблема

Имеется система, состоящая из множества взаимодействующих классов. При этом взаимодействующие объекты должны находиться в согласованных состояниях. Вы хотите избежать монолитности такой системы, сделав классы слабо связанными (или повторно используемыми).

Обсуждение паттерна Observer

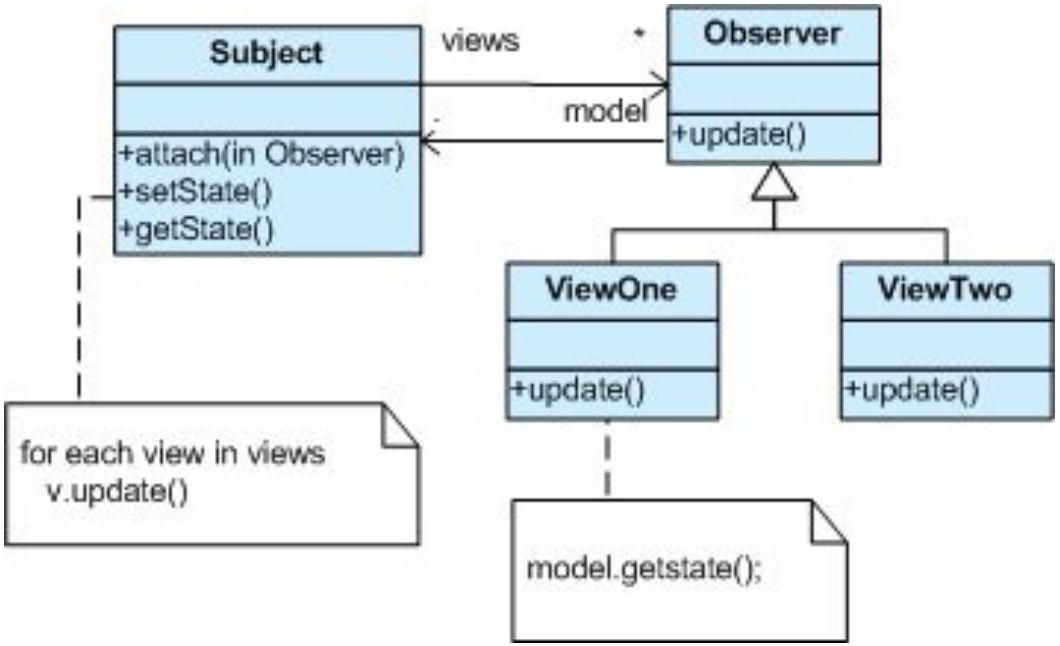
Паттерн Observer определяет объект Subject, хранящий данные (модель), а всю функциональность "представлений" делегирует слабосвязанным отдельным объектам Observer. При создании наблюдатели Observer регистрируются у объекта Subject. Когда объект Subject изменяется, он извещает об этом всех зарегистрированных наблюдателей. После этого каждый наблюдатель запрашивает у объекта Subject ту часть состояния, которая необходима для отображения данных.

Такая схема позволяет динамически настраивать количество и "типы" представлений объектов.

Использование паттерна Observer

- Проведите различия между основной (или независимой) и дополнительной (или зависимой) функциональностями.
- Смоделируйте "независимую" функциональность с помощью абстракции "субъект".
- Смоделируйте "зависимую" функциональность с помощью иерархии "наблюдатель". Класс Subject связан только с базовым классом Observer.
- Клиент настраивает количество и типы наблюдателей.
- Наблюдатели регистрируются у субъекта.
- Субъект извещает всех зарегистрированных наблюдателей.
- Субъект может "протолкнуть" информацию в наблюдателей, или наблюдатели могут "вытянуть" необходимую им информацию от объекта Subject.

UML-диаграмма классов паттерна «Наблюдатель» (Observer)



Особенности паттерна «Наблюдатель» (Observer)

Особенности паттерна Observer

Паттерны Chain of Responsibility, Command, Mediator и Observer показывают, как можно разделить отправителей и получателей запросов с учетом своих особенностей. Chain of Responsibility передает запрос отправителя по цепочке потенциальных получателей. Command определяет связь - "оправитель-получатель" с помощью подкласса. В Mediator отправитель и получатель ссылаются друг на друга косвенно, через объект-посредник. В паттерне Observer связь между отправителем и получателем получается слабой, при этом число получателей может конфигурироваться во время выполнения.

Mediator и Observer являются конкурирующими паттернами. Если Observer распределяет взаимодействие с помощью объектов "наблюдатель" и "субъект", то Mediator использует объект-посредник для инкапсуляции взаимодействия между другими объектами.

Mediator может использовать Observer для динамической регистрации коллег и их взаимодействия с посредником.

Реализация паттерна Observer

1. Смоделируйте "независимую" функциональность с помощью абстракции "субъект".
2. Смоделируйте "зависимую" функциональность с помощью иерархии "наблюдатель".
3. Класс Subject связан только с базовым классом Observer.
4. Наблюдатели регистрируются у субъекта.
5. Субъект извещает всех зарегистрированных наблюдателей.
6. Наблюдатели "вытягивают" необходимую им информацию от объекта Subject.
7. Клиент настраивает количество и типы наблюдателей.

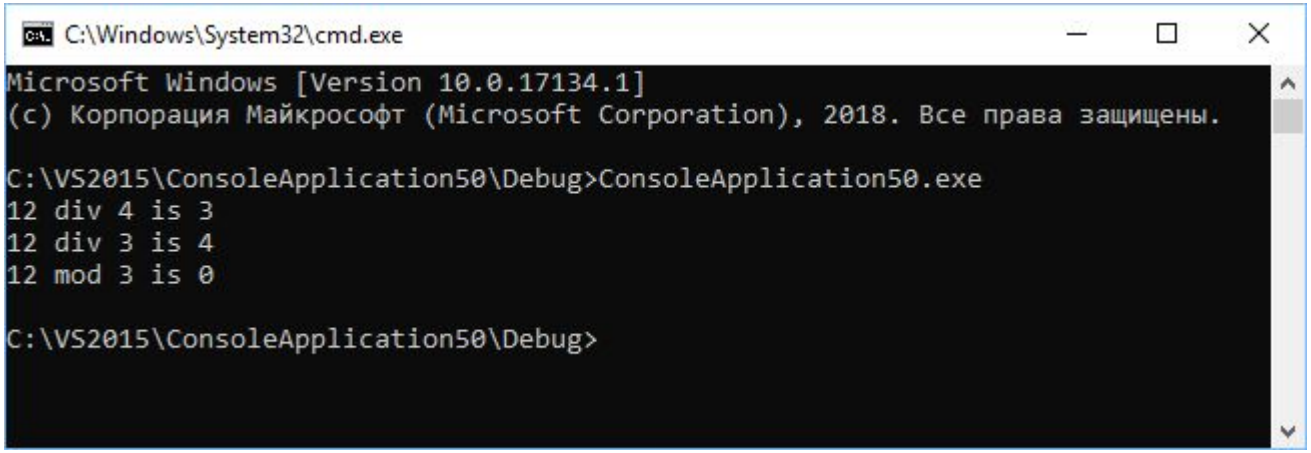
Пример реализации паттерна «Наблюдатель» (Observer)

```
// 1. "Независимая" функциональность
class Subject {
        // 3. Связь только с базовым классом Observer
        vector < class Observer * > views;
        int value;
public:
        void attach(Observer *obs) { views.push_back(obs); }
        void setVal(int val) {  value = val;  notify();    }
        int getVal() { return value; }
        void notify();
};
// 2. "Зависимая" функциональность
class Observer {
        Subject *model;
        int denom;
public:
        Observer(Subject *mod, int div) {
                model = mod;
                denom = div;
                // 4. Наблюдатели регистрируются у субъекта
                model->attach(this);
        }
        virtual void update() = 0;
protected:
        Subject *getSubject() { return model; }
        int getDivisor() {     return denom;}
};
```

Пример реализации паттерна «Наблюдатель» (Observer)

```
void Subject::notify() {  // 5. Извещение наблюдателей
	for (size_t i = 0; i < views.size(); i++)
		views[i]->update();
}
class DivObserver : public Observer {
public:
	DivObserver(Subject *mod, int div) : Observer(mod, div) {}
	void update() {
		// 6. "Вытягивание" интересующей информации
		int v = getSubject()->getVal(), d = getDivisor();
		cout << v << " div " << d << " is " << v / d << '\n';
	}
};
class ModObserver : public Observer {
public:
	ModObserver(Subject *mod, int div) : Observer(mod, div) {}
	void update() { // 6. "Вытягивание" интересующей информации
		int v = getSubject()->getVal(), d = getDivisor();
		cout << v << " mod " << d << " is " << v%d << '\n';
	}
};
int main() {
	Subject subj;
	DivObserver divObs1(&subj, 4); // 7. Клиент настраивает число
	DivObserver divObs2(&subj, 3); //    и типы наблюдателей
	ModObserver modObs3(&subj, 3);
	subj.setVal(12);
}
```

Результат работы программы:



Паттерн “Состояние» (State)

Назначение паттерна State

Паттерн State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния. Создается впечатление, что объект изменил свой класс.

Паттерн State является объектно-ориентированной реализацией конечного автомата.

Решаемая проблема

Поведение объекта зависит от его состояния и должно изменяться во время выполнения программы. Такую схему можно реализовать, применив множество условных операторов: на основе анализа текущего состояния объекта предпринимаются определенные действия. Однако при большом числе состояний условные операторы будут разбросаны по всему коду, и такую программу будет трудно поддерживать.

Обсуждение паттерна State

Паттерн State решает указанную проблему следующим образом:

Вводит класс Context, в котором определяется интерфейс для внешнего мира.

Вводит абстрактный класс State.

Представляет различные "состояния" конечного автомата в виде подклассов State.

В классе Context имеется указатель на текущее состояние, который изменяется при изменении состояния конечного автомата.

Использование паттерна State

Определите существующий или создайте новый класс-"обертку" Context, который будет использоваться клиентом в качестве "конечного автомата".

Создайте базовый класс State, который повторяет интерфейс класса Context. Каждый метод принимает один дополнительный параметр: экземпляр класса Context. Класс State может определять любое полезное поведение "по умолчанию".

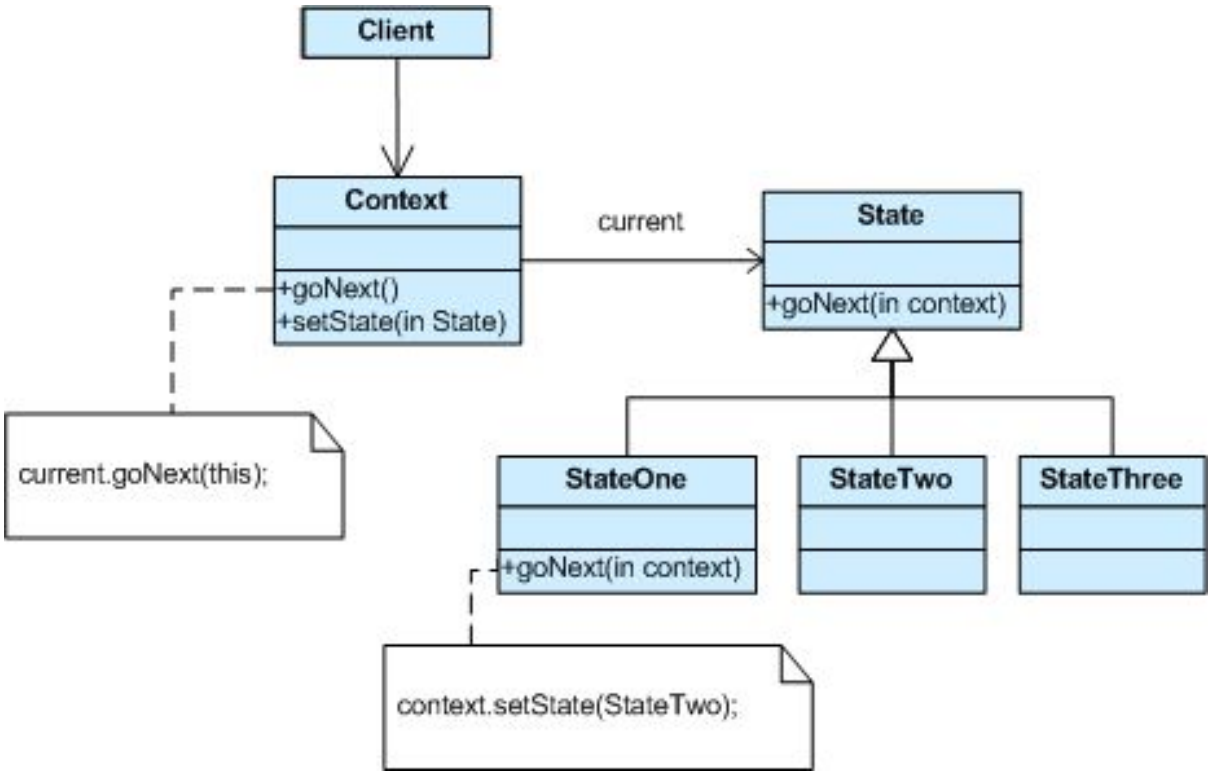
Создайте производные от State классы для всех возможных состояний.

Класс-"обертка" Context имеет ссылку на объект "текущее состояние".

Все полученные от клиента запросы класс Context просто делегирует объекту "текущее состояние", при этом в качестве дополнительного параметра передается адрес объекта Context.

Используя этот адрес, в случае необходимости методы класса State могут изменить "текущее состояние" класса Context.

UML-диаграмма классов паттерна «Состояние» (State)



Особенности паттерна State

Объекты класса State часто бывают одиночками.

Flyweight показывает, как и когда можно разделять объекты State.

Паттерн Interpreter может использовать State для определения контекстов при синтаксическом разборе.

Паттерны State и Bridge имеют схожие структуры за исключением того, что Bridge допускает иерархию классов-конвертов (аналогов классов-"оберток"), а State-нет. Эти паттерны имеют схожие структуры, но решают разные задачи: State позволяет объекту изменять свое поведение в зависимости от внутреннего состояния, в то время как Bridge разделяет абстракцию от ее реализации так, что их можно изменять независимо друг от друга.

Реализация паттерна State основана на паттерне Strategy. Различия заключаются в их назначении.

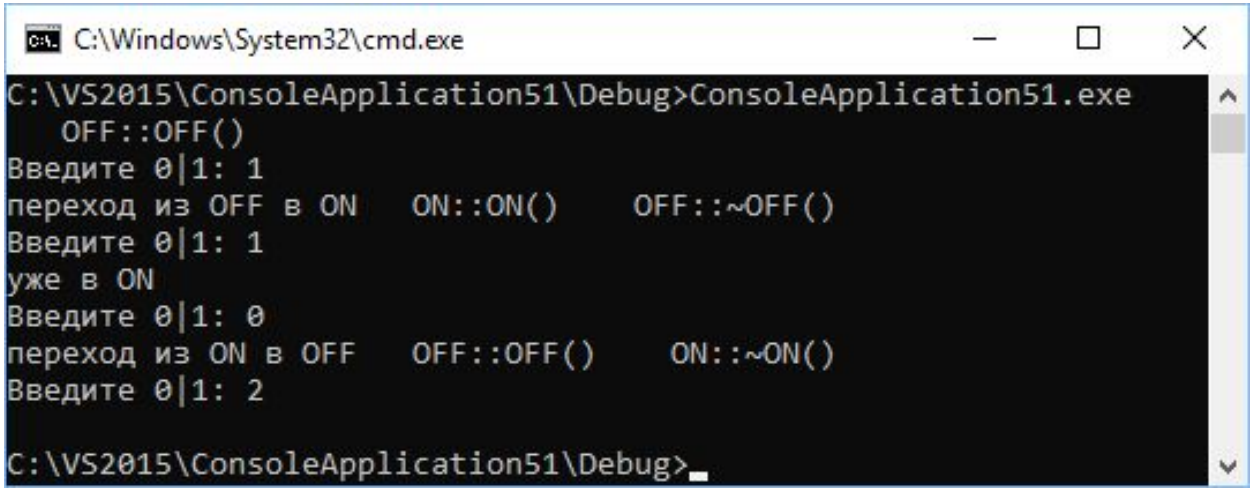
Пример реализации паттерна «Состояние» (State)

```
// Рассмотрим пример конечного автомата с двумя возможными состояниями и двумя событиями.

class Machine;
class State
{
public:
        virtual void on(Machine *m) {    cout << "уже в ON" << endl; }
        virtual void off(Machine *m) {    cout << "уже в OFF" << endl;}
};
class Machine  // Context
{
        State *current;// указатель на текущее состояние
public:
        Machine();
        void setCurrent(State *s) { current = s; }
        void on() { current->on(this); }
        void off() { current->off(this); }
};

class ON : public State
{
public:
        ON()  { cout << "   ON::ON() "; };
        ~ON(){ cout << "   ON::~ON()\n"; };
        void off(Machine *m);
};
class OFF : public State
{
public:
        OFF()   { cout << "   OFF::OFF() "; };
        ~OFF() { cout << "   OFF::~OFF()\n";  };
        void on(Machine *m)
        {
                cout << "переход из OFF в ON";  m->setCurrent(new ON());
                delete this;
        }
```

Пример реализации паттерна «Состояние» (State) (продолжение)

```
void ON::off(Machine *m)
{
	cout << "переход из ON в OFF";
	m->setCurrent(new OFF());
	delete this;
}
Machine::Machine()
{
	current = new OFF();
	cout << '\n';
}
int main()
{
	void(Machine:: *ptrs[])() =
	{
		&Machine::off, &Machine::on
	};
	Machine fsm;
	int num;
	setlocale(LC_ALL, "rus");
	while (1)
	{
		cout << "Введите 0|1: ";
		cin >> num;
		if (num == 0 || num == 1)
			(fsm.*ptrs[num])();
		else
			break;
	}
}
```

Результат работы программы:



Паттерн Chain of Responsibility (цепочка обязанностей)

Назначение паттерна Chain of Responsibility

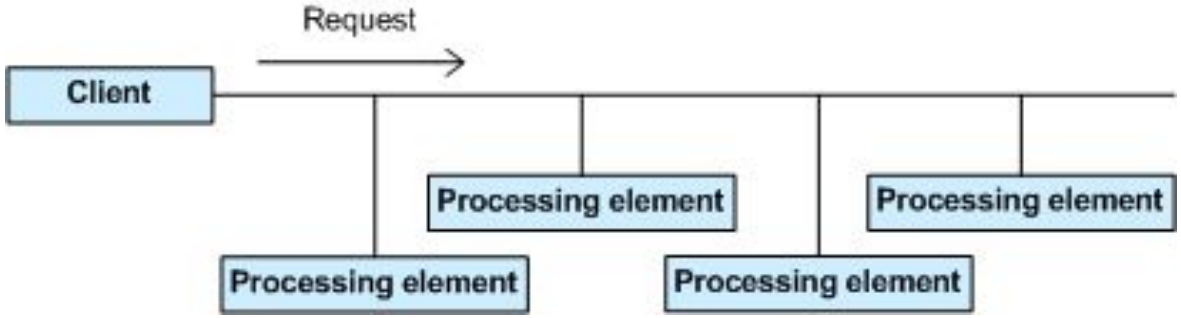
Паттерн Chain of Responsibility позволяет избежать жесткой зависимости отправителя запроса от его получателя, при этом запрос может быть обработан несколькими объектами. Объекты-получатели связываются в цепочку. Запрос передается по этой цепочке, пока не будет обработан.

Вводит конвейерную обработку для запроса с множеством возможных обработчиков.

Обычно представляет собой объектно-ориентированный связанный список с рекурсивным обходом.

Решаемая проблема

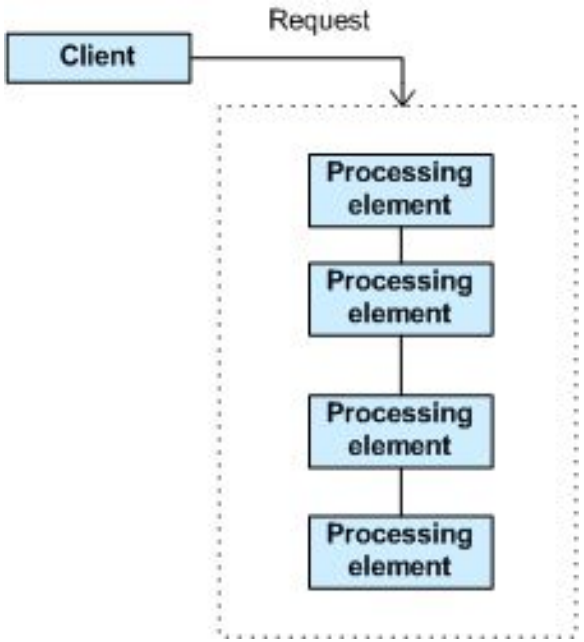
Имеется поток запросов разного типа и переменное число "обработчиков" этих запросов. Необходимо эффективно обрабатывать запросы без жесткой привязки к их обработчикам, при этом запрос может быть обработан любым обработчиком.



Паттерн Chain of Responsibility (цепочка обязанностей)

Реализация

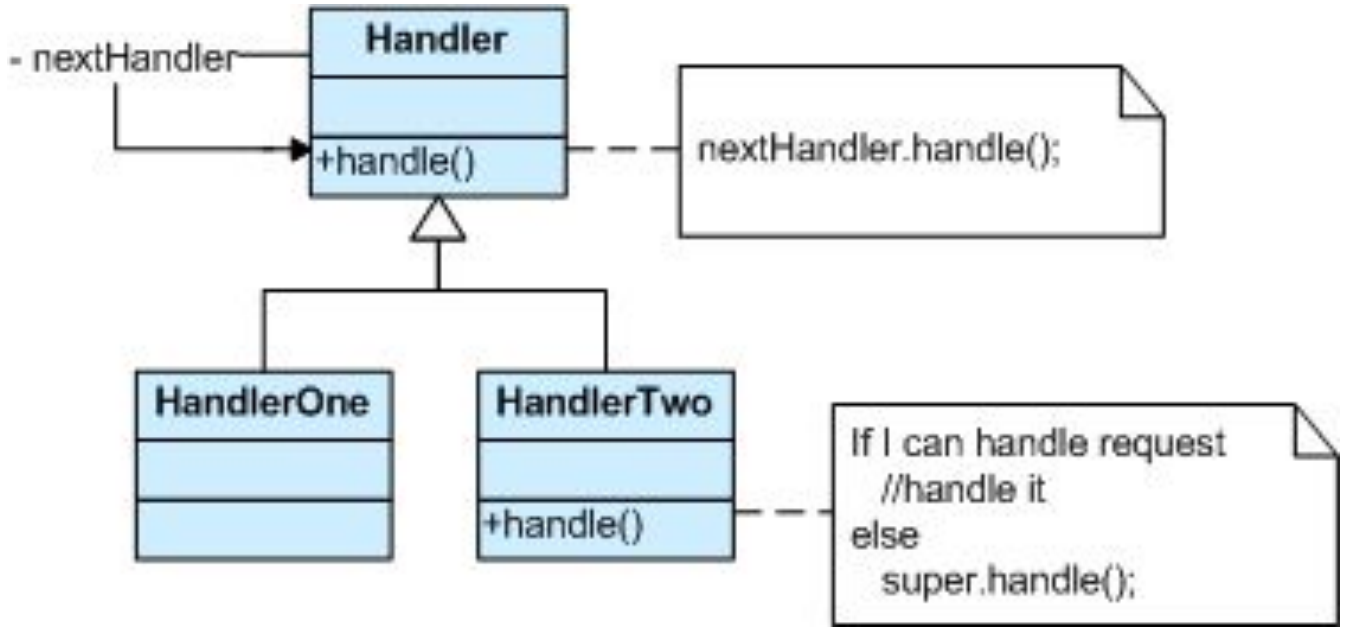
Паттерн Chain of Responsibility связывает в цепочку объекты-получатели, а затем передает запрос-сообщение от одного объекта к другому до тех пор, пока не достигнет объекта, способного его обработать. Число и типы объектов-обработчиков заранее неизвестны, они могут настраиваться динамически. Механизм связывания в цепочку использует рекурсивную композицию, что позволяет использовать неограниченное число обработчиков.



Достоинства

Паттерн Chain of Responsibility упрощает взаимосвязи между объектами. Вместо хранения ссылок на всех кандидатов-получателей запроса, каждый отправитель хранит единственную ссылку на начало цепочки, а каждый получатель имеет единственную ссылку на своего преемника - последующий элемент в цепочке.

Паттерн Chain of Responsibility (цепочка обязанностей) UML — диаграмма классов



Производные классы знают, как обрабатывать запросы клиентов. Если "текущий" объект не может обработать запрос, то он делегирует его базовому классу, который делегирует "следующему" объекту и так далее.

Обработчики могут вносить свой вклад в обработку каждого запроса. Запрос может быть передан по всей длине цепочки до самого последнего звена.

Паттерн Chain of Responsibility (цепочка обязанностей)

Использование паттерна Chain of Responsibility

- Базовый класс имеет указатель на "следующий обработчик".
- Каждый производный класс реализует свой вклад в обработку запроса.
- Если запрос должен быть "передан дальше", то производный класс "вызывает" базовый класс, который с помощью указателя делегирует запрос далее.
- Клиент (или третья сторона) создает цепочку получателей (которая может иметь ссылку с последнего узла на корневой узел).
- Клиент передает каждый запрос в начало цепочки.

Особенности паттерна Chain of Responsibility

- Паттерны Chain of Responsibility, Command, Mediator и Observer показывают, как можно разделить отправителей и получателей с учетом их особенностей. Chain of Responsibility передает запрос отправителя по цепочке потенциальных получателей.
- Chain of Responsibility может использовать Command для представления запросов в виде объектов.
- Chain of Responsibility часто применяется вместе с паттерном Composite. Родитель компонента может выступать в качестве его преемника.

реализация паттерна Chain of Responsibility

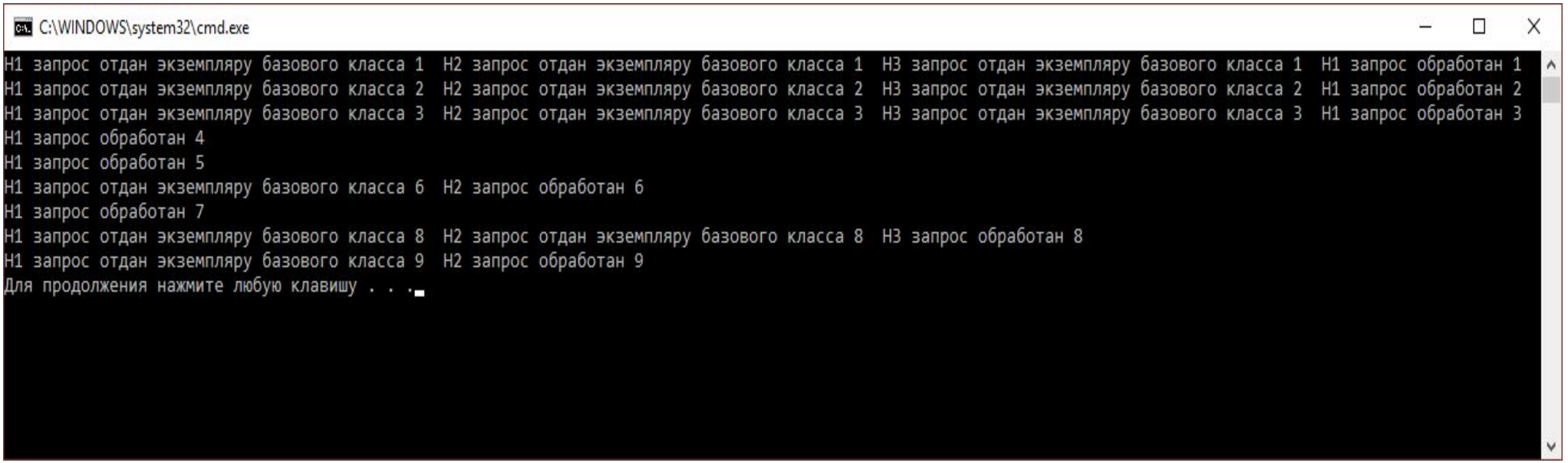
```
class Base
{
	// 1. Указатель "next" в базовом классе
	Base *next;
public:
	Base()	{	next = 0; }
	void setNext(Base *n)  {	next = n; }
	void add(Base *n)
	{
		if (next)
			next->add(n);
		else
			next = n;
	}
	// 2. Метод базового класса, делегирующий запрос next-объекту
	virtual void handle(int i) { next→handle(i); }
};

class Handler1 : public Base
{
public:
	void handle(int i)
	{
		if (rand() % 3)
		{
			cout << "H1 запрос отдан экземпляру базового класса " << i << "  ";
			Base::handle(i);
		}
		else
			cout << "H1 запрос обработан " << i << "  ";
		}
};
```

реализация паттерна Chain of Responsibility (продолжение)

```
class Handler2 : public Base
{
public:
	void handle(int i)
	{
		if (rand() % 3)
		{
			cout << "H2 запрос отдан экземпляру базового класса " << i << "  ";
			Base::handle(i);
		}
		else
			cout << "H2 запрос обработан " << i << "  ";
	}
};

class Handler3 : public Base
{
public:
	void handle(int i)
	{
		if (rand() % 3)
		{
			cout << "H3 запрос отдан экземпляру базового класса " << i << "  ";
			Base::handle(i);
		}
		else
			cout << "H3 запрос обработан " << i << "  ";
	}
};
```

Результат работы программы реализации паттерна Chain of Responsibility

```
int main()
{
	setlocale(LC_ALL, "rus");
	srand((unsigned int) time(nullptr)); // инициализация датчика случайных чисел
	Handler1 root;
	Handler2 two;
	Handler3 thr;
	root.add(&two);
	root.add(&thr);
	thr.setNext(&root); // замыкаю обработку по кругу
	for (int i = 1; i < 10; i++)  { root.handle(i); cout << '\n';    }
}
```



Паттерн Interpreter (интерпетатор)

Назначение паттерна Interpreter

Для заданного языка определяет представление его грамматики, а также интерпретатор предложений этого языка.

Отображает проблемную область в язык, язык – в грамматику, а грамматику – в иерархии объектно-ориентированного проектирования.

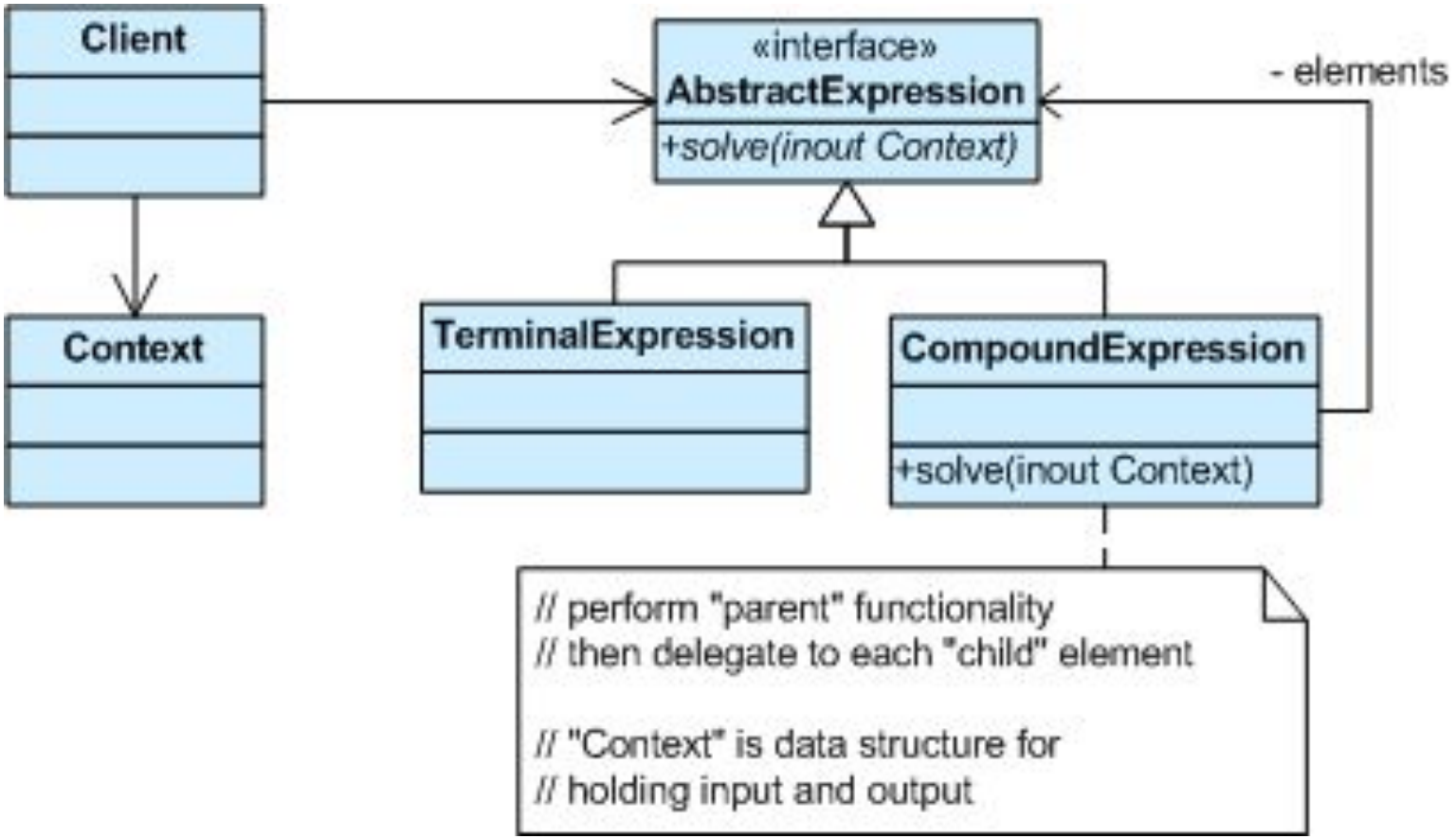
Решаемая проблема

Пусть в некоторой, хорошо определенной области периодически случается некоторая проблема. Если эта область может быть описана некоторым “языком“, то проблема может быть легко решена с помощью “интерпретирующей машины“.

Паттерн Interpreter определяет грамматику простого языка для проблемной области, представляет грамматические правила в виде языковых предложений и интерпретирует их для решения задачи. Для представления каждого грамматического правила паттерн Interpreter использует отдельный класс. А так как грамматика, как правило, имеет иерархическую структуру, то иерархия наследования классов хорошо подходит для ее описания.

Абстрактный базовый класс определяет метод *interpret()*, принимающий (в качестве аргумента) текущее состояние языкового потока. Каждый конкретный подкласс реализует метод *interpret(),* добавляя свой вклад в процесс решения проблемы.

- **UML-диаграмма классов паттерна Interpreter**



Использование паттерна Interpreter

- Определите “малый“ язык, “инвестиции” в который будут оправданными.
- Разработайте грамматику для языка.
- Для каждого грамматического правила (продукции) создайте свой класс.
- Полученный набор классов организуйте в структуру с помощью паттерна Composite.
- В полученной иерархии классов определите метод interpret(Context).
- Объект Context инкапсулирует информацию, глобальную по отношению к интерпретатору. Он используется классами во время процесса ” интерпретации”.

Особенности паттерна Interpreter

- Абстрактное синтаксическое дерево интерпретатора – пример паттерна Composite.
- Для обхода узлов дерева может применяться паттерн Iterator.
- Терминальные символы могут разделяться с помощью Flyweight.
- Паттерн Interpreter не рассматривает вопросы синтаксического разбора. Когда грамматика очень сложная, должны использоваться другие методики.

Пример реализации паттерна Interpreter

Рассмотрим задачу интерпретирования (вычисления) значений строковых представлений римских чисел. Используем следующую грамматику:

romanNumeral ::= {thousands} {hundreds} {tens} {ones}
thousands,hundreds,tens,ones ::= nine | four | {five} {one} {one} {one}
nine ::= "CM" | "XC" | "IX"
four ::= "CD" | "XL" | "IV"
five ::= 'D' | 'L' | 'V'
one ::= 'M' | 'C' | 'X' | 'I'

Для проверки и интерпретации строки используется иерархия классов с общим базовым классом RNInterpreter, имеющим 4 под-интерпретатора.

Каждый под-интерпретатор получает "контекст" (оставшуюся неразобранную часть строки и накопленное вычисленное значение разобранной части) и вносит свой вклад в процесс обработки. Под-интерпретаторы просто определяют шаблонные методы, объявленные в базовом классе RNInterpreter.

реализация паттерна Interpreter

```
// Совместное использование паттернов Interpreter и Template Method
// опережающие объявления классов
// опережающие объявления классов
class Thousand;
class Hundred;
class Ten;
class One;

class RNInterpreter
{
public:
	RNInterpreter(); // конструктор для клиента
	RNInterpreter(int) {} // конструктор для классов-наследников предотвращающий бесконечный цикл
	int solve(char*); // interpret() для клиента
	virtual void solve(char *input, int &total);
protected:
	// эти члены-функции нельзя делать чисто-виртуальными
	virtual char one() { return '\0'; }
	virtual char *four() { return '\0'; }
	virtual char five() { return '\0'; }
	virtual char *nine() { return '\0'; }
	virtual int multiplier() { return '\0'; }
private:
	RNInterpreter *thousands;
	RNInterpreter *hundreds;
	RNInterpreter *tens;
	RNInterpreter *ones;
};
```

реализация паттерна Interpreter (продолжение)

```
void RNInterpreter::solve(char *input, int &total)  //
{       // выполняется разбор входной строки
        int index = 0;
        if (!strncmp(input, nine(), 2)) // если 9
        {
                total += 9 * multiplier();
                index += 2;
        }
        else if (!strncmp(input, four(), 2))  // если 4
        {
                total += 4 * multiplier();
                index += 2;
        }
        else
        {
                if (input[0] == five()) // если 5
                {
                        total += 5 * multiplier();
                        index = 1;
                }
                else
                        index = 0;
                for (int end = index + 3; index < end; index++)
                        if (input[index] == one())
                                total += 1 * multiplier();
                        else
                                break;
        }
        strcpy(input, &(input[index]));// удаляю из входной строки обработанные символы
}
```

реализация паттерна Interpreter (продолжение)

```
class Thousand : public RNInterpreter
{
public:
// определение конструктора с 1 аргументом для предотв. бесконечного цикла в конструкторе базового класса
        Thousand(int) : RNInterpreter(1) {}
protected:
        char one()  { return 'M'; }
        char *four(){ return "";  }
        char five() { return '\0';}
        char *nine(){ return "";  }
        int multiplier() { return 1000;  }
};
class Hundred : public RNInterpreter
{
public:
        Hundred(int) : RNInterpreter(1) {}
protected:
        char one()  { return 'C'; }
        char *four(){ return "CD";      }
        char five()  { return 'D'; }
        char *nine(){ return "CM";      }
        int multiplier(){ return 100; }
};
class Ten : public RNInterpreter
{
public:
        Ten(int) : RNInterpreter(1) {}
protected:
        char one()  { return 'X';  }
        char *four(){ return "XL"; }
        char five() { return 'L';  }
        char *nine(){ return "XC"; }
        int multiplier() { return 10;}
```

реализация паттерна Interpreter (продолжение)

```
class One : public RNInterpreter
{
public:
        One(int) : RNInterpreter(1) {}
protected:
        char one() {       return 'I';   }
        char *four(){      return "IV";}
        char five() {      return 'V';  }
        char *nine(){      return "IX";}
        int multiplier(){  return 1;    }
};

RNInterpreter::RNInterpreter()
{
        thousands = new Thousand(1);
        hundreds = new Hundred(1);
        tens = new Ten(1);
        ones = new One(1);
}

int RNInterpreter::solve(char *input)
{
        int total = 0;
        thousands->solve(input, total);
        hundreds->solve(input, total);
        tens->solve(input, total);
        ones->solve(input, total);
        if (strcmp(input, "")) return 0;
        return total;
}
```

реализация паттерна Interpreter (результат)

```
int main()
{
	RNInterpreter interpreter;
	char input[20];
	setlocale(LC_ALL, "rus");
	cout << "Введите число в римской записи: ";
	while (cin >> input)
	{
		cout << input;
		cout << " = " << interpreter.solve(input) << endl;
		cout << "Введите число в римской записи: ";
	}
}
```