Паттерны поведения

Паттерны поведения

- Паттерны поведения связаны с алгоритмами и распределением обязанностей между объектами.
  - Речь в них идет не только о самих объектах и классах, но и о типичных способах взаимодействия
  - Паттерны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы
    - Внимание акцентировано не на потоке управления как таковом, а на связях между объектами

Паттерны поведения

- Цепочка обязанностей (Chain of Responsibility)
- Команда (Command)
- Интерпретатор (Interpreter)
- Итератор (Iterator)
- Посредник (Mediator)
- Хранитель (Memento)
- Наблюдатель (Observer)
- Состояние (State)
- Стратегия (Strategy)
- Шаблонный метод (Template Method)
- Посетитель (Visitor)

Command (Команда)

Паттерн «Команда»

- Инкапсулирует запрос как объект, позволяя тем самым:
  - Задавать параметры клиентов для обработки соответствующих запросов
  - Ставить запросы в очередь или протоколировать их
  - Поддерживать отмену операций
- Альтернативные названия – Action, Transaction

Применимость

- Параметризация объектов выполняемым действием
  - Реализация пунктов меню
- Определение, постановка в очередь и выполнение запросов в разное время
  - Возможность выполнения команды в другом процессе
- Поддержка отмены операций
  - Undo/Redo
- Протоколирование изменений
  - Восстановление состояния после сбоя
- Структурирование системы на основе высокоуровневых операций, составленных из примитивных (транзакции)

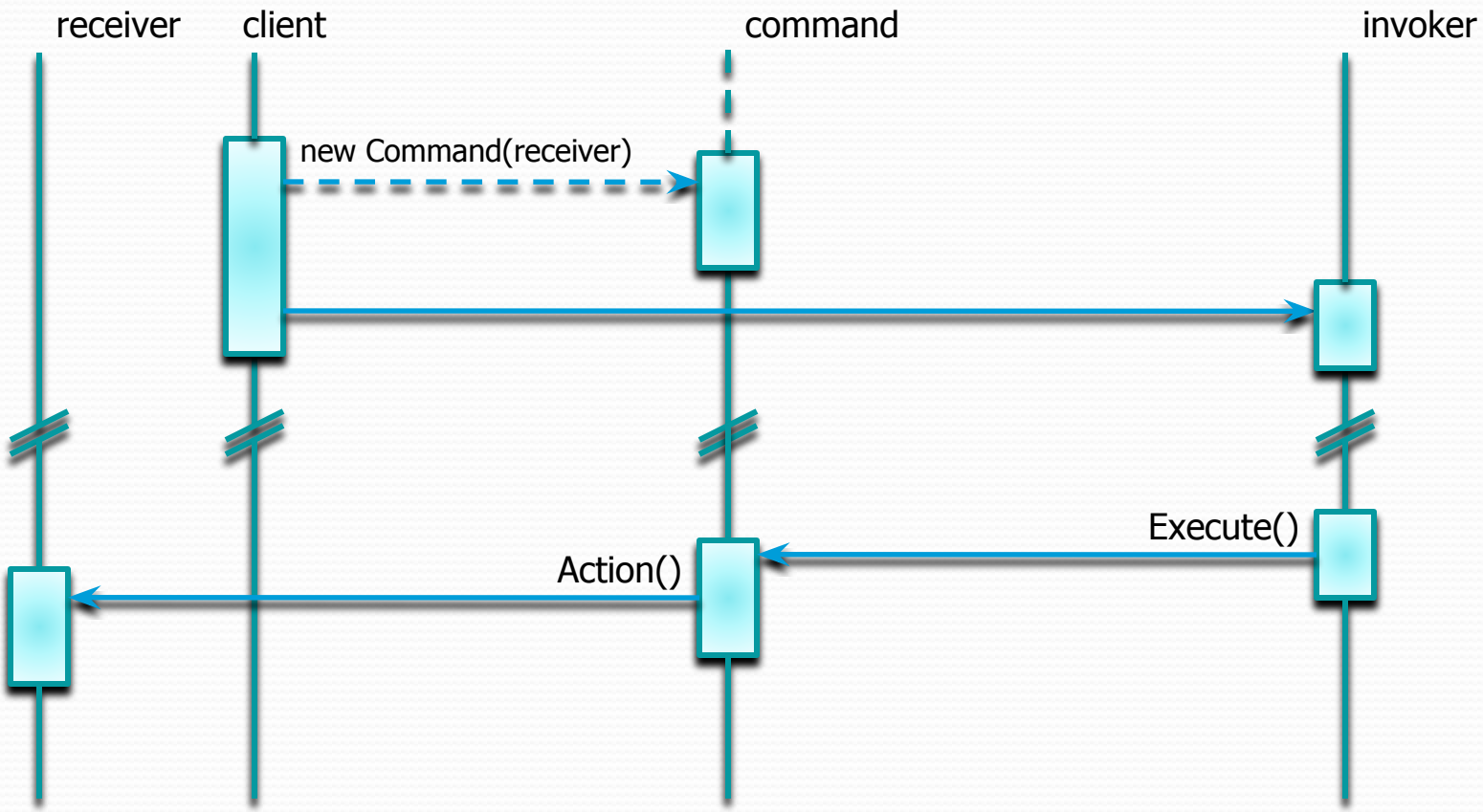
Структура


Client
Invoker
Command
Execute()
Объявляет интерфейс для выполнения операции
•Определяет связь между объектом-получателем Receiver и действием
•Реализует операцию Execute путем вызова соответствующих операций получателя
Инициатор. Обращается к команде для выполнения запроса
receiver
Receiver
Action()
ConcreteCommand
Execute()
Состояние
receiver->Action();
Создает объект класса ConcreteCommand и устанавливает его получателя
•Располагает информацией о способах выполнения операций, необходимых для удовлетворения запроса
•В роли получателя может выступать любой класс.

Отношения

- Клиент создает объект ConcreteCommand и устанавливает для него получателя
- Инициатор Invoker сохраняет объект ConcreteCommand
- Инициатор отправляет запрос, вызывая операцию команды Execute
  - Если поддерживается отмена выполненных действий, то ConcreteCommand перед вызовом Execute сохраняет информацию о состоянии, достаточную для выполнения отката
- Объект ConcreteCommand вызывает операции получателя для выполнения запроса

Диаграмма взаимодействий



Результаты

- Команда разрывает связь между объектом, инициирующим операцию, и объектом, имеющим информацию о том, как ее выполнить
- Команды - это самые настоящие объекты
  - Допускается манипулировать ими и расширять их точно так же, как в случае с любыми другими объектами
- Из простых команд можно собирать составные
  - В общем случае составные команды описываются паттерном Компоновщик
- Добавлять новые команды легко, поскольку никакие существующие классы изменять не нужно

Поддержка отмены и повтора операций

- В классе ConcreteCommand должна сохраняться информация для отмены операции
  - Объект-получатель
  - Аргументы операции, выполненное пользователем
  - Исходные значения различных атрибутов получателя, которые могли измениться в результате обработки запроса
    - Получатель должен предоставлять операции, позволяющие команде вернуть его в исходное состояние
- Для нескольких уровней отмены необходимо вести историю выполненных команд
  - Отмена операций осуществляется в обратном порядке, повтор – в прямом
  - Объекты команд, возможно, потребуется скопировать перед помещением в список истории команд
    - Используется паттерн «Прототип»

Пример использования

- В текстовом редакторе предоставить возможность отмены и повтора операций по редактированию текста
  - Вставка текста
  - Удаление текста
- Количество уровней отмены команд ограничивается лишь объемом доступной памяти

Иерархия классов

CDocument – получатель команд

```
class CDocument
{
public:
	std::string const& GetText()const{return m_text;}

	void InsertText(std::string const& text, size_t pos = std::string::npos)
	{
		if (pos == std::string::npos)		pos = m_text.length();
		m_text.insert(pos, text);
	}

	void RemoveText(size_t pos, size_t length)
	{
		m_text.erase(pos, length);
	}
private:
	std::string m_text;
};

std::ostream & operator<<(std::ostream & strm, CDocument const& doc)
{
	return strm << doc.GetText() << "\n";
}
```

CCommand и CCommandImpl

```
class CCommand
{
public:
	virtual ~CCommand(){}
	virtual void Execute() =
0;
	virtual void Unexecute()
= 0;
};

typedef boost::shared_ptr
<CCommand> CCommandPtr;
```

Данные методы реализовываются конкретными командами

```
class CCommandImpl : public CCommand
{
public:
	CCommandImpl(bool executed = false):m_executed(executed){
	}
	virtual void Execute()
	{
		if (m_executed)
			throw std::logic_error("The command has been already
executed");
		DoExecute();	m_executed = true;
	}

	virtual void Unexecute()
	{
		if (!m_executed)
			throw std::logic_error("The command has not been
executed yet");
		DoUnexecute();	m_executed = false;
	}
protected:
	virtual void DoExecute()=0;
	virtual void DoUnexecute()=0;
private:
```

Команда вставки текста

```
class CInsertTextCommand : public CCommandImpl
{
public:
	CInsertTextCommand(CDocument & doc, std::string const& text, size_t pos =
std::string::npos)
		:m_doc(doc),m_text(text)
		,m_pos(pos == std::string::npos ? doc.GetText().length() : pos)
	{
	}
protected:
	virtual void DoExecute()
	{
		m_doc.InsertText(m_text, m_pos);
	}

	virtual void DoUnexecute()
	{
		m_doc.RemoveText(m_pos, m_text.length());
	}
private:
	CDocument&	m_doc;
	std::string	m_text;
	size_t		m_pos;
};
```

При выполнении команды происходит вставка текста в документ

При отмене команды происходит удаление вставленного фрагмент из документа

Команда стирания текста

```
class CEraseTextCommand : public CCommandImpl
{
public:
	CEraseTextCommand(CDocument & doc, size_t pos, size_t length = std::string::npos)
		:m_doc(doc),m_pos(pos),m_length(length)
	{
	}
protected:
	virtual void DoExecute()
	{
		m_text = m_doc.GetText().substr(m_pos, m_length
		m_doc.RemoveText(m_pos, m_length);
	}
	virtual void DoUnexecute()
	{
		m_doc.InsertText(m_text, m_pos);
		m_text.clear();
	}
private:
	CDocument&	m_doc;
	std::string	m_text;
	size_t		m_pos, m_length;
};
```

При выполнении команды происходит сохранение стираемого текста в поле m_text команды

При отмене команды происходит восстановление удаленного текста из поля m_text

Класс CCommands (начало)

```
class CCommands
{
private:
	std::vector<CCommandPtr> m_commands;
	size_t m_currentCommand;
public:
	CCommands()
		:m_currentCommand(0)
	{
	}

	bool UndoAvailable()const
	{
		return m_currentCommand != 0;
	}

	bool RedoAvailable()const
	{
		return m_currentCommand < m_commands.size();
	}
	...
```

Проверка доступности операции Undo()

Проверка доступности операции Redo

Класс CCommands (окончание)

```
        void AddAndExecute(CCommandPtr pCommand)
        {
                m_commands.reserve(m_currentCommand + 1);
                pCommand->Execute();

                if (RedoAvailable())
m_commands.resize(m_currentCommand);
                m_commands.push_back(pCommand);
                ++m_currentCommand;
        }
        void Undo()
        {
                if (!UndoAvailable()) throw std::logic_error("Undo is not
available");
                size_t commandIndex = m_currentCommand - 1;
                m_commands[commandIndex]->Unexecute();
                m_currentCommand = commandIndex;
        }
        void Redo()
        {
                if (!RedoAvailable()) throw std::logic_error("Redo is not
available");
                m_commands[m_currentCommand]->Execute();
                ++m_currentCommand;
```

Добавляем команду в конец очереди команд, при необходимости стирая «будущие» команды

Отменяем текущую команду, сдвигаем указатель команд на одну команду назад

Повторяем отмененную ранее команду, сдвигаем указатель команд на одну команду вперед

Редактор текста (Client)

```
class CTextEditor
{
public:
	CTextEditor(CDocument & doc):m_doc(doc){}
	void InsertText(std::string const& text, size_t pos = std::string::npos)
	{
		m_commands.AddAndExecute(CCommandPtr(new CInsertTextCommand(m_doc, text, pos)));
	}
	void EraseText(size_t pos, size_t length = std::string::npos)
	{
		m_commands.AddAndExecute(CCommandPtr(new CEraseTextCommand(m_doc, pos, length)));
	}
	void Undo()
	{
		if (m_commands.UndoAvailable())  m_commands.Undo();
	}
	void Redo()
	{
		if (m_commands.RedoAvailable())  m_commands.Redo();
	}
private:
	CDocument & m_doc;
	CCommands m_commands;
};
```

Пример использования

```
int main(int argc, char* argv[])
{
	CDocument doc;
	CTextEditor editor(doc);

	editor.InsertText("Hello, ");
	std::cout << doc;
	editor.InsertText("World!");
	std::cout << doc;
	editor.Undo();
	std::cout << doc;
	editor.InsertText("world! :)");
	std::cout << doc;
	editor.EraseText(0, 5);
	std::cout << doc;
	editor.InsertText("Good bye", 0);
	std::cout << doc;
	editor.Undo();
	std::cout << doc;
	editor.Undo();
	std::cout << doc;
	return 0;
}
```

```
Output:
Hello,
Hello, World!
Hello,
Hello, world! :)
, world! :)
Good bye, world! :)
, world! :)
Hello, world! :)
```