Тема 5. Паттерны поведения

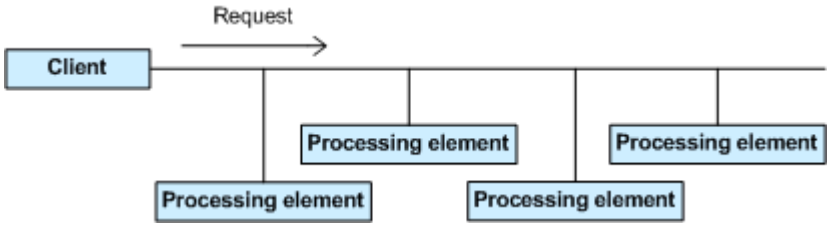
Паттерн Цепочка обязанностей

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.
- Объектно-ориентированный связанный список с рекурсивным обходом.

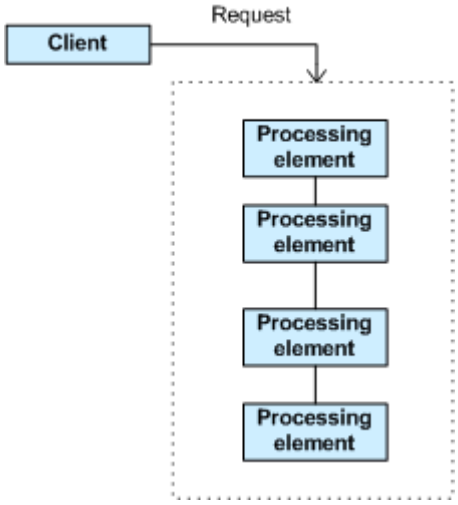
Решаемая проблема

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



Обсуждение паттерна Chain of Responsibility

Инкапсулирует элементы по обработке запросов внутри абстрактного "конвейера". Клиенты "кидают" свои запросы на вход этого конвейера.



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.

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.

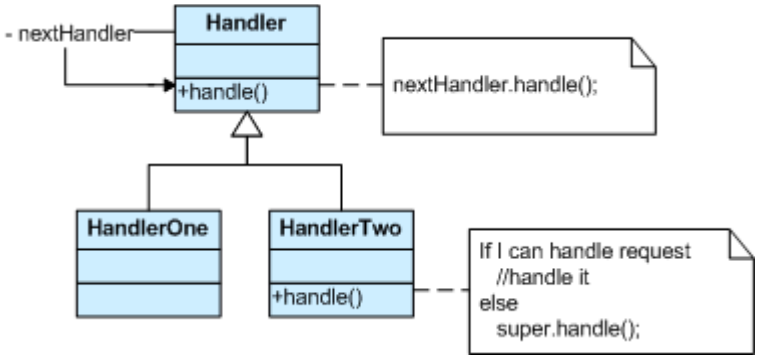
Убедитесь, что система корректно "отлавливает" случаи необработанных запросов.

Не используйте паттерн Chain of Responsibility, когда каждый запрос обрабатывается только одним обработчиком, или когда клиент знает, какой именно объект должен обработать его запрос.

Структура паттерна Chain of Responsibility

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.

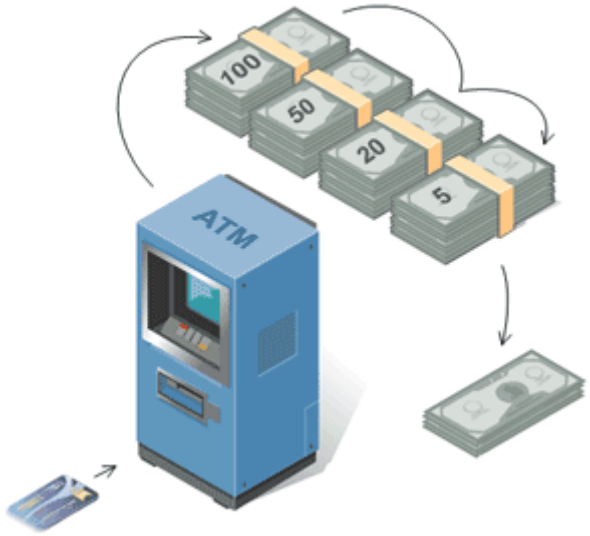
UML-диаграмма классов паттерна Chain of Responsibility



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

Пример паттерна Chain of Responsibility

Паттерн Chain of Responsibility позволяет избежать привязки отправителя запроса к его получателю, давая шанс обработать запрос нескольким получателям. Банкомат использует Chain of Responsibility в механизме выдачи денег.



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.
- Рекурсивное делегирование создает иллюзию волшебства.

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

Реализация паттерна Chain of Responsibility по шагам

1. Создайте указатель на следующий обработчик next в базовом классе.
2. Метод handle() базового класса всегда делегирует запрос следующему объекту.
3. Если производные классы не могут обработать запрос, они делегируют его базовому классу.

```
1   #include <iostream>
2   #include <vector>
3   #include <ctime>
4   using namespace std;
5
6   class Base
7   {
8       // 1. Указатель "next" в базовом классе
9       Base *next;
10    public:
11      Base()
12      {
13          next = 0;
14      }
15      void setNext(Base *n)
16      {
17          next = n;
18      }
19      void add(Base *n)
20      {
21          if (next)
22            next->add(n);
23          else
24            next = n;
25      }
26      // 2. Метод базового класса, делегирующий запрос next-объекту
27      virtual void handle(int i)
28      {
```

```
29          next->handle(i);
30      }
31  };
32
33  class Handler1: public Base
34  {
35    public:
36      /*virtual*/void handle(int i)
37      {
38          if (rand() % 3)
39          {
40              // 3. 3 из 4 запросов не обрабатываем
41              cout << "H1 passsed " << i << "  ";
42              // 3. и делегируем базовому классу
43              Base::handle(i);        }
44          else
45            cout << "H1 handled " << i << "  ";
46      }
47  };
48
49  class Handler2: public Base
50  {
51    public:
52      /*virtual*/void handle(int i)
53      {
54          if (rand() % 3)
55          {
56              cout << "H2 passsed " << i << "  ";
```

```
57              Base::handle(i);
58          }
59          else
60            cout << "H2 handled " << i << "  ";
61      }
62  };
63
64  class Handler3: public Base
65  {
66    public:
67       /*virtual*/void handle(int i)
68      {
69          if (rand() % 3)
70          {
71              cout << "H3 passsed " << i << "  ";
72              Base::handle(i);
73          }
74          else
75            cout << "H3 handled " << i << "  ";
76      }
77  };
78
79  int main()
80  {
81    srand(time(0));
82    Handler1 root;
83    Handler2 two;
84    Handler3 thr;
```

```
85     root.add(&two);
86     root.add(&thr);
87     thr.setNext(&root);
88     for (int i = 1; i < 10; i++)
89     {
90       root.handle(i);
91       cout << '\n';
92     }
93   }
```

Вывод программы:

```
1   H1 passsed 1  H2 passsed 1  H3 passsed 1  H1 passsed 1  H2 handled 1
2   H1 handled 2
3   H1 handled 3
4   H1 passsed 4  H2 passsed 4  H3 handled 4
5   H1 passsed 5  H2 handled 5
6   H1 passsed 6  H2 passsed 6  H3 passsed 6  H1 handled 6
7   H1 passsed 7  H2 passsed 7  H3 passsed 7  H1 passsed 7  H2 handled 7
8   H1 handled 8
9   H1 passsed 9  H2 passsed 9  H3 handled 9
```

Реализация паттерна Chain of Responsibility: Chain and Composite

1. Создайте указатель на следующий обработчик next в базовом классе.
2. Метод handle() базового класса всегда делегирует запрос следующему объекту.
3. Если производные классы не могут обработать запрос, они делегируют его базовому классу.

```
1    #include <iostream>
2    #include <vector>
3    #include <ctime>
4    using namespace std;
5
```

```
6   class Component
7   {
8       int value;
9       // 1. Указатель "next" в базовом классе
10      Component *next;
11    public:
12      Component(int v, Component *n)
13      {
14          value = v;
15          next = n;
16      }
17      void setNext(Component *n)
18      {
19          next = n;
20      }
21      virtual void traverse()
22      {
23          cout << value << ' ';
24      }
25      // 2. Метод базового класса, делегирующий запрос next-объекту
26      virtual void volunteer()
27      {
28          next->volunteer();
29      }
30  };
31
32  class Primitive: public Component
33  {
```

```
34    public:
35      Primitive(int val, Component *n = 0): Component(val, n){}
36       /*virtual*/void volunteer()
37      {
38          Component::traverse();
39          // 3. Примитивные объекты не обрабатывают 5 из 6 запросов
40          if (rand() *100 % 6 != 0)
41            // 3. Делегируем запрос в базовый класс
42            Component::volunteer();
43      }
44  };
45
46  class Composite: public Component
47  {
48      vector < Component * > children;
49    public:
50      Composite(int val, Component *n = 0): Component(val, n){}
51      void add(Component *c)
52      {
53          children.push_back(c);
54      }
55       /*virtual*/void traverse()
56      {
57          Component::traverse();
58          for (int i = 0; i < children.size(); i++)
59            children[i]->traverse();
60      }
61      // 3. Составные объекты никогда не обрабатывают запросы
```

```
62      /*virtual*/void volunteer()
63      {
64          Component::volunteer();
65      }
66  };
67
68  int main()
69  {
70      srand(time(0));                 // 1
71      Primitive seven(7);             // |
72      Primitive six(6, &seven);       // +-- 2
73      Composite three(3, &six);       // |   |
74      three.add(&six);
75      three.add(&seven);              // |   +-- 4 5
76      Primitive five(5, &three);      // |
77      Primitive four(4, &five);       // +-- 3
78      Composite two(2, &four);        // |   |
79      two.add(&four);
80      two.add(&five);                 // |   +-- 6 7
81      Composite one(1, &two);         // |
82      Primitive nine(9, &one);        // +-- 8 9
83      Primitive eight(8, &nine);
84      one.add(&two);
85      one.add(&three);
86      one.add(&eight);
87      one.add(&nine);
88      seven.setNext(&eight);
89      cout << "traverse: ";
```

```
90      one.traverse();
91      cout << '\n';
92      for (int i = 0; i < 8; i++)
93      {
94        one.volunteer();
95        cout << '\n';
96      }
97    }
```

Вывод программы:

```
1   traverse: 1 2 4 5 3 6 7 8 9
2   4
3   4 5 6 7
4   4 5 6 7 8 9 4 5 6 7 8 9 4
5   4
6   4 5 6
7   4 5
8   4 5
9   4 5 6 7 8 9 4 5 6 7 8 9 4 5 6
```

Паттерн Команда

Назначение паттерна Command

Используйте паттерн Command 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 вызова в процедурном программировании.

> Пример событийно-управляемой системы – приложение с пользовательским интерфейсом. При выборе некоторого пункта меню пользователем

> вырабатывается запрос на выполнение определенного действия (например, открытия файла).

Описание паттерна Command

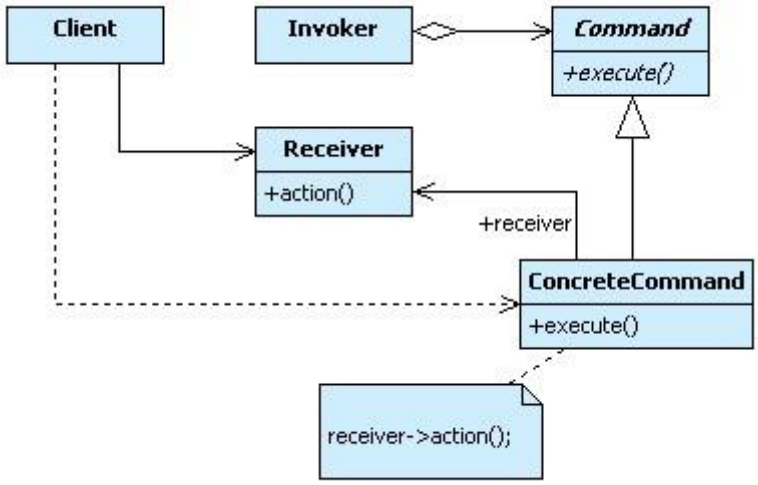
Паттерн Command преобразовывает запрос на выполнение действия в отдельный объект-команду. Такая инкапсуляция позволяет передавать эти действия другим функциям и объектам в качестве параметра, приказывая им выполнить запрошенную операцию. Команда – это объект, поэтому над ней допустимы любые операции, что и над объектом.

Интерфейс командного объекта определяется абстрактным базовым классом `Command` и в самом простом случае имеет единственный метод `execute()`. Производные классы определяют получателя запроса (указатель на объект-получатель) и необходимую для выполнения операцию (метод этого объекта). Метод `execute()` подклассов `Command` просто вызывает нужную операцию получателя.

В паттерне Command может быть до трех участников:

- Клиент, создающий экземпляр командного объекта.
- Инициатор запроса, использующий командный объект.
- Получатель запроса.

UML-диаграмма классов паттерна Command



Сначала клиент создает объект `ConcreteCommand`, конфигурируя его получателем запроса. Этот объект также доступен инициатору. Инициатор использует его при отправке запроса, вызывая метод `execute()`. Этот алгоритм напоминает работу функции обратного вызова в процедурном программировании – функция регистрируется, чтобы быть вызванной позднее.

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.

Паттерн Интерпретатор

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“.

Обсуждение паттерна Interpreter

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.

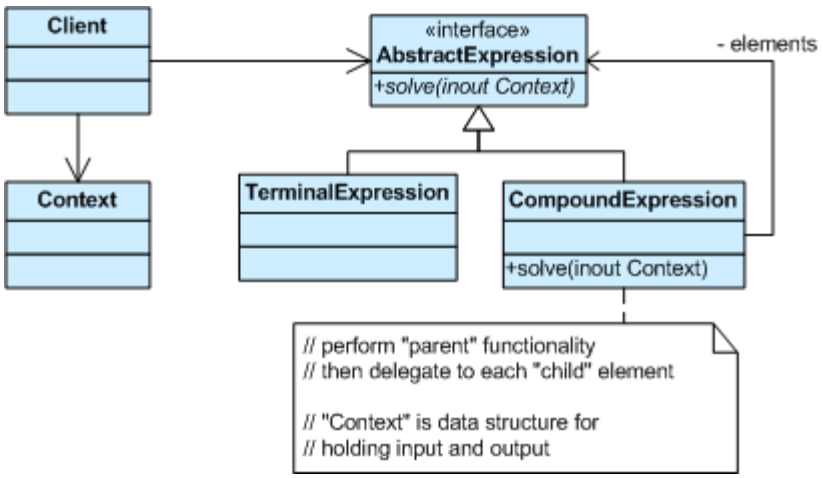
Абстрактный базовый класс определяет метод `interpret()`, принимающий (в качестве аргумента) текущее состояние языкового потока. Каждый конкретный подкласс реализует метод `interpret()`, добавляя свой вклад в процесс решения проблемы.

Структура паттерна Interpreter

Паттерн Interpreter моделирует проблемную область с помощью рекурсивной грамматики. Каждое грамматическое правило может быть либо составным (правило ссылается на другие правила) либо терминальным (листовой узел в структуре ”дерево”).

Для рекурсивного обхода ”предложений” при их интерпретации используется паттерн Composite.

UML-диаграмма классов паттерна Interpreter



Паттерн Итератор

Назначение паттерна Iterator

- Предоставляет способ последовательного доступа ко всем элементам составного объекта, не раскрывая его внутреннего представления.
- Абстракция в стандартных библиотеках C++ и Java, позволяющая разделить классы коллекций и алгоритмов.
- Придает обходу коллекции "объектно-ориентированный статус".
- Полиморфный обход.

Решаемая проблема

Вам необходим механизм "абстрактного" обхода различных структур данных так, что могут определяться алгоритмы, способные взаимодействовать со структурами прозрачно.

Обсуждение паттерна Iterator

Составной объект, такой как список, должен предоставлять способ доступа к его элементам без раскрытия своей внутренней структуры. Более того, иногда нужно перебирать элементы списка различными способами, в зависимости от конкретной задачи. Но вы, вероятно, не хотите раздувать интерфейс списка операциями для различных обходов, даже если они необходимы. Кроме того, иногда нужно иметь несколько активных обходов одного списка одновременно. Было бы хорошо иметь единый интерфейс для обхода разных типов составных объектов (т.е. полиморфная итерация).

Паттерн Iterator позволяет все это делать. Ключевая идея состоит в том, чтобы ответственность за доступ и обход переместить из составного объекта на объект Iterator, который будет определять стандартный протокол обхода.

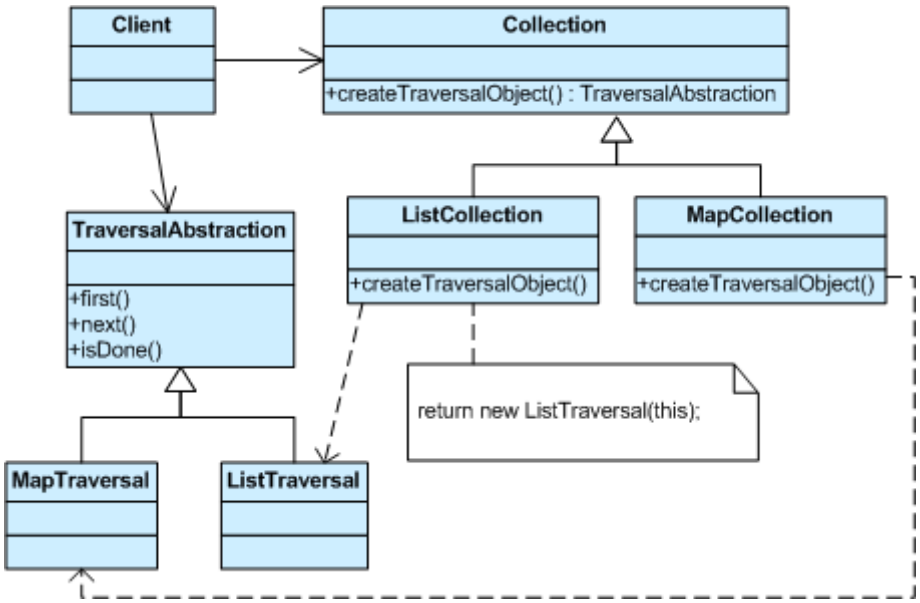
Абстракция Iterator имеет основополагающее значение для технологии, называемой "обобщенное программирование". Эта технология четко разделяет такие понятия как "алгоритм" и "структура данных". Мотивирующие факторы: способствование компонентной разработке, повышение производительности и снижение расходов на управление.

Рассмотрим пример. Если вы хотите одновременно поддерживать четыре вида структур данных (массив, бинарное дерево, связанный список и хэш-таблица) и три алгоритма (сортировка, поиск и слияние), то традиционный подход потребует 12 вариантов конфигураций (четыре раза по три), в то время как обобщенное программирование требует лишь 7 (четыре плюс три).

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`.

UML-диаграмма классов паттерна Iterator



Паттерн Посредник

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".

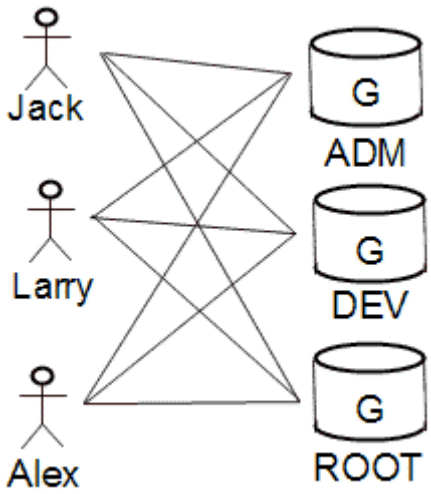
Решаемая проблема

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.

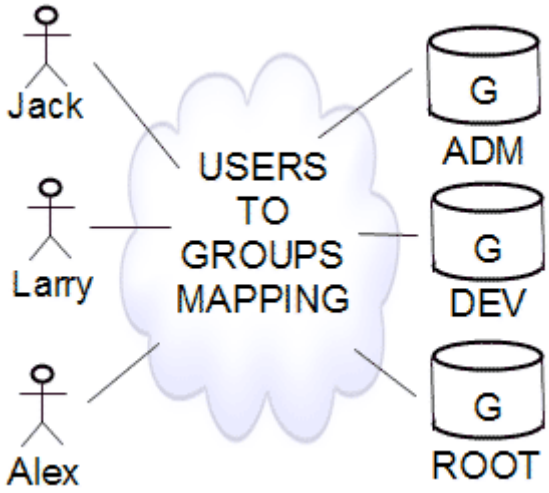
Обсуждение паттерна Mediator

В Unix права доступа к системным ресурсам определяются тремя уровнями: владелец, группа и прочие. Группа представляет собой совокупность пользователей, обладающих некоторой функциональной принадлежностью. Каждый пользователь в системе может быть членом одной или нескольких групп, и каждая группа может иметь 0 или более пользователей, назначенных этой группе. Следующий рисунок показывает трех пользователей, являющихся членами всех трех групп.



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.



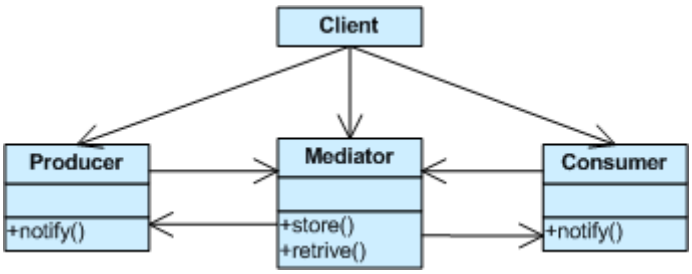
Разбиение системы на множество объектов в общем случае повышает степень повторного использования, однако множество взаимосвязей между этими объектами, как правило, приводит к обратному эффекту. Чтобы этого не допустить, инкапсулируйте взаимодействия между объектами в объект-посредник. Действуя как центр связи, этот объект-посредник контролирует и координирует взаимодействие группы объектов. При этом объект-посредник делает взаимодействующие объекты слабо связанными, так как им больше не нужно хранить ссылки друг на друга – все взаимодействие идет через этого посредника. Расширить или изменить это взаимодействие можно через его подклассы.

Паттерн Mediator заменяет взаимодействие "все со всеми" взаимодействием "один со всеми".

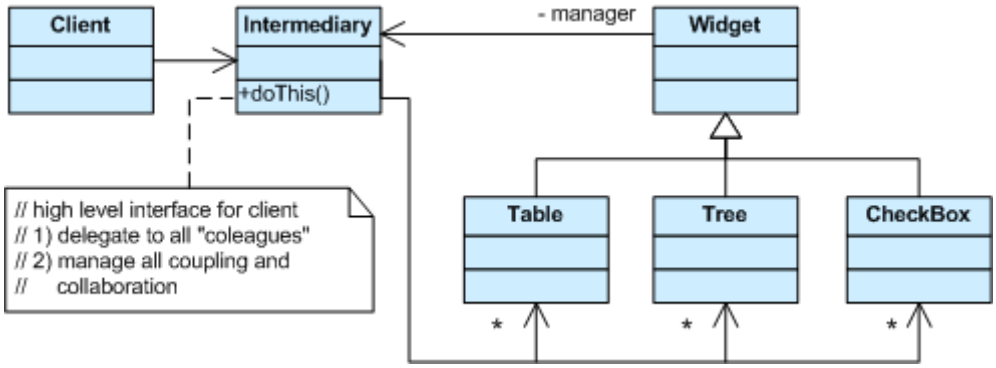
Пример рационального использования паттерна Mediator – моделирование отношений между пользователями и группами операционной системы. Группа может иметь 0 или более пользователей, а пользователь может быть членом 0 или более групп. Паттерн Mediator предусматривает гибкий способ управления пользователями и группами.

Структура паттерна Mediator

UML-диаграмма классов паттерна Mediator



Коллеги (или взаимодействующие объекты) не связаны друг с другом. Каждый из них общается с посредником, который, в свою очередь, знает об остальных и управляет ими. Паттерн Mediator делает статус взаимодействия "все со всеми" "полностью объектным".



Пример паттерна Mediator

Паттерн Mediator определяет объект, управляющий набором взаимодействующих объектов. Слабая связанность достигается благодаря тому, что вместо непосредственного взаимодействия друг с другом коллеги общаются через объект-посредник. Башня управления полетами в аэропорту хорошо демонстрирует этот паттерн. Пилоты взлетающих или идущих на посадку самолетов в районе аэропорта общаются с башней вместо непосредственного общения друг с другом. Башня определяет, кто и в каком порядке будет садиться или взлетать. Важно отметить, что башня контролирует самолеты только в районе аэродрома, а не на протяжении всего полета.

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 Chain of Responsibility, Command, Mediator и 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. Мы обнаружили, что 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

Паттерн Наблюдатель

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) .

> Паттерн Observer находит широкое применение в системах пользовательского интерфейса, в которых данные и их представления ("виды") отделены друг от друга. При изменении данных должны быть

> изменены все представления этих данных (например, в виде таблицы, графика и диаграммы).

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 обозреватель запрашивает у объекта Subject ту часть состояния, которая необходима для отображения данных.

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

Описанный выше протокол взаимодействия соответствует модели вытягивания (pull), когда субъект информирует наблюдателей о своем изменении, и каждый наблюдатель ответственен за "вытягивание" у Subject нужных ему данных. Существует также модель проталкивания, когда субъект Subject посылает ("проталкивает") наблюдателям детальную информацию о своем изменении.

Существует также ряд вопросов, о которых следует упомянуть, но обсуждение которых останется за рамками данной статьи:

- Реализация "компрессии" извещений (посылка единственного извещения на серию последовательных изменений субъекта Subject).
- Мониторинг нескольких субъектов с помощью одного наблюдателя Observer.
- Исключение висячих ссылок у наблюдателей на удаленные субъекты. Для этого субъект должен уведомить наблюдателей о своем удалении.

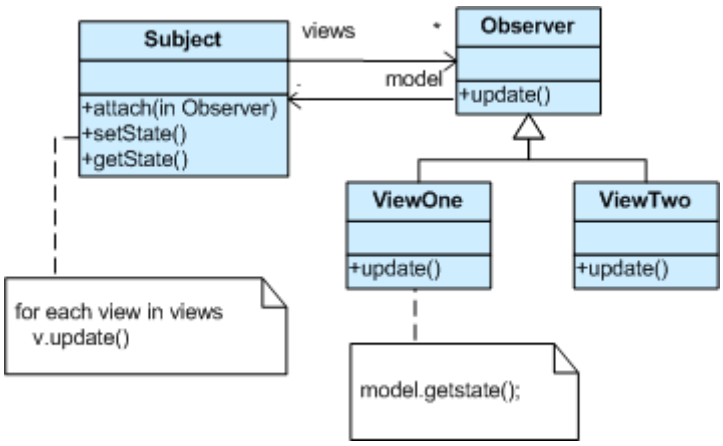
Паттерн Observer впервые был применен в архитектуре Model-View-Controller языка Smalltalk, представляющей каркас для построения пользовательских интерфейсов.

Структура паттерна Observer

Subject представляет главную (независимую) абстракцию. Observer представляет изменяемую (зависимую) абстракцию. Субъект извещает

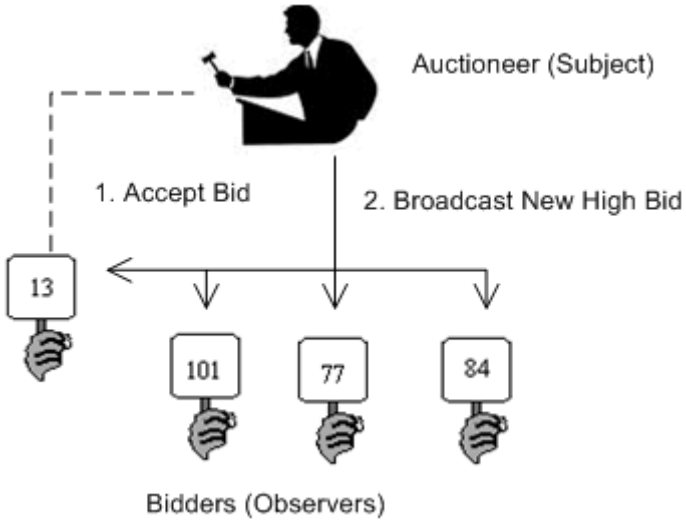
наблюдателей о своем изменении, на что каждый наблюдатель может запросить состояние субъекта.

UML-диаграмма классов паттерна Observer



Пример 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. Некоторые аукционы демонстрируют этот паттерн. Каждый участник имеет карточку с цифрами, которую он использует для обозначения предлагаемой цены (ставки). Ведущий аукциона (Subject) начинает торги и наблюдает, когда кто-нибудь поднимает карточку, предлагая новую более высокую цену. Ведущий принимает заявку, о чем тут же извещает всех участников аукциона (Observers).



Использование паттерна Observer

1. Проведите различия между основной (или независимой) и дополнительной (или зависимой) функциональностями.
2. Смоделируйте "независимую" функциональность с помощью абстракции "субъект".

3. Смоделируйте "зависимую" функциональность с помощью иерархии "наблюдатель".
4. Класс Subject связан только с базовым классом Observer.
5. Клиент настраивает количество и типы наблюдателей.
6. Наблюдатели регистрируются у субъекта.
7. Субъект извещает всех зарегистрированных наблюдателей.
8. Субъект может "протолкнуть" информацию в наблюдателей, или наблюдатели могут "вытянуть" необходимую им информацию от объекта Subject.

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. Мы обнаружили, что легче сделать повторно используемыми Наблюдателей и Субъектов, чем Посредников.
- Mediator может использовать Observer для динамической регистрации коллег и их взаимодействия с посредником.

Реализация паттерна Observer

Реализация паттерна Observer по шагам

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.

```
1  #include <iostream>
2  #include <vector>
```

```
3   using namespace std;
4
5   // 1. "Независимая" функциональность
6   class Subject {
7       // 3. Связь только базовым классом Observer
8       vector < class Observer * > views;
9       int value;
10    public:
11      void attach(Observer *obs) {
12          views.push_back(obs);
13      }
14      void setVal(int val) {
15          value = val;
16          notify();
17      }
18      int getVal() {
19          return value;
20      }
21      void notify();
22  };
23
24  // 2. "Зависимая" функциональность
25  class Observer {
26      Subject *model;
27      int denom;
28    public:
29      Observer(Subject *mod, int div) {
30          model = mod;
```

```
31          denom = div;
32          // 4. Наблюдатели регистрируются у субъекта
33          model->attach(this);
34      }
35      virtual void update() = 0;
36    protected:
37      Subject *getSubject() {
38          return model;
39      }
40      int getDivisor() {
41          return denom;
42      }
43  };
44
45  void Subject::notify() {
46    // 5. Извещение наблюдателей
47    for (int i = 0; i < views.size(); i++)
48      views[i]->update();
49  }
50
51  class DivObserver: public Observer {
52    public:
53      DivObserver(Subject *mod, int div): Observer(mod, div){}
54      void update() {
55          // 6. "Вытягивание" интересующей информации
56          int v = getSubject()->getVal(), d = getDivisor();
57          cout << v << " div " << d << " is " << v/d << '\n';
58      }
```

```
59  };
60
61  class ModObserver: public Observer {
62    public:
63      ModObserver(Subject *mod, int div): Observer(mod, div){}
64      void update() {
65          int v = getSubject()->getVal(), d = getDivisor();
66          cout << v << " mod " << d << " is " << v%d << '\n';
67      }
68  };
69
70  int main() {
71    Subject subj;
72    DivObserver divObs1(&subj, 4); // 7. Клиент настраивает число
73    DivObserver divObs2(&subj, 3); //    и типы наблюдателей
74    ModObserver modObs3(&subj, 3);
75    subj.setVal(14);
76  }
```

Вывод программы:

```
1   14 div 4 is 3 14 div 3 is 4 14 mod 3 is 2
```

Реализация паттерна Observer: до и после

До

Количество и типы "зависимых" объектов определяются классом `Subject`. Пользователь не имеет возможности влиять на эту конфигурацию.

```
1   class DivObserver
2   {
3       int m_div;
4     public:
5       DivObserver(int div)
```

```
6       {
7           m_div = div;
8       }
9       void update(int val)
10      {
11          cout << val << " div " << m_div
12              << " is " << val / m_div << '\n';
13      }
14  };
15
16  class ModObserver
17  {
18      int m_mod;
19    public:
20      ModObserver(int mod)
21      {
22          m_mod = mod;
23      }
24      void update(int val)
25      {
26          cout << val << " mod " << m_mod
27              << " is " << val % m_mod << '\n';
28      }
29  };
30
31  class Subject
32  {
33      int m_value;
```

```
34      DivObserver m_div_obj;
35      ModObserver m_mod_obj;
36    public:
37      Subject(): m_div_obj(4), m_mod_obj(3){}
38      void set_value(int value)
39      {
40          m_value = value;
41          notify();
42      }
43      void notify()
44      {
45          m_div_obj.update(m_value);
46          m_mod_obj.update(m_value);
47      }
48  };
49
50  int main()
51  {
52    Subject subj;
53    subj.set_value(14);
54  }
```

Вывод программы:

```
1   14 div 4 is 3 14 mod 3 is 2
```

После

Теперь класс Subject не связан с непосредственной настройкой числа и типов объектов `Observer`. Клиент установил два наблюдателя `DivObserver` и одного `ModObserver`.

```
1   class Observer
2   {
```

```
3     public:
4       virtual void update(int value) = 0;
5     };
6
7     class Subject
8     {
9       int m_value;
10      vector m_views;
11    public:
12      void attach(Observer *obs)
13      {
14        m_views.push_back(obs);
15      }
16      void set_val(int value)
17      {
18        m_value = value;
19        notify();
20      }
21      void notify()
22      {
23        for (int i = 0; i < m_views.size(); ++i)
24          m_views[i]->update(m_value);
25      }
26    };
27
28    class DivObserver: public Observer
29    {
30      int m_div;
```

```
31    public:
32      DivObserver(Subject *model, int div)
33      {
34          model->attach(this);
35          m_div = div;
36      }
37       /* virtual */void update(int v)
38      {
39          cout << v << " div " << m_div << " is " << v / m_div << '\n';
40      }
41  };
42
43  class ModObserver: public Observer
44  {
45      int m_mod;
46    public:
47      ModObserver(Subject *model, int mod)
48      {
49          model->attach(this);
50          m_mod = mod;
51      }
52       /* virtual */void update(int v)
53      {
54          cout << v << " mod " << m_mod << " is " << v % m_mod << '\n';
55      }
56  };
57
58  int main()
```

```
59  {
60    Subject subj;
61    DivObserver divObs1(&subj, 4);
62    DivObserver divObs2(&subj, 3);
63    ModObserver modObs3(&subj, 3);
64    subj.set_val(14);
65  }
```

Вывод программы:

```
1   14 div 4 is 3 14 div 3 is 4 14 mod 3 is 2
```