Тема 4. Структурные паттерны

Структурные паттерны рассматривают вопросы о компоновке системы на основе классов и объектов. При этом могут использоваться следующие механизмы:

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

Кратко рассмотрим особенности структрурных паттернов (шаблонов).

Паттерн Adapter представляет собой программную обертку над уже существующими классами и предназначен для преобразования их интерфейсов к виду, пригодному для последующего использования в новом программном проекте.
Паттерн Bridge отделяет абстракцию от реализации так, что то и другое можно изменять независимо.
Паттерн Composite группирует схожие объекты в древовидные структуры. Рассматривает единообразно простые и сложные объекты.
Паттерн Decorator используется для расширения функциональности объектов. Являясь гибкой альтернативой порождению классов, паттерн Decorator динамически добавляет объекту новые обязанности.
Паттерн Facade предоставляет высокоуровневый унифицированный интерфейс к набору интерфейсов некоторой подсистемы, что облегчает ее использование.
Паттерн Flyweight использует разделение для эффективной поддержки множества объектов.
Паттерн Proxy замещает другой объект для контроля доступа к нему.

Паттерн Адаптер

Назначение паттерна Adapter

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

Паттерн Adapter, представляющий собой программную обертку над существующими классами, преобразует их интерфейсы к виду, пригодному для последующего использования.

Рассмотрим простой пример, когда следует применять паттерн Adapter. Пусть мы разрабатываем систему климат-контроля, предназначенной для

автоматического поддержания температуры окружающего пространства в заданных пределах. Важным компонентом такой системы является температурный датчик, с помощью которого измеряют температуру окружающей среды для последующего анализа. Для этого датчика уже имеется готовое программное обеспечение от сторонних разработчиков, представляющее собой некоторый класс с соответствующим интерфейсом. Однако использовать этот класс непосредственно не удастся, так как показания датчика снимаются в градусах Фаренгейта. Нужен адаптер, преобразующий температуру в шкалу Цельсия.

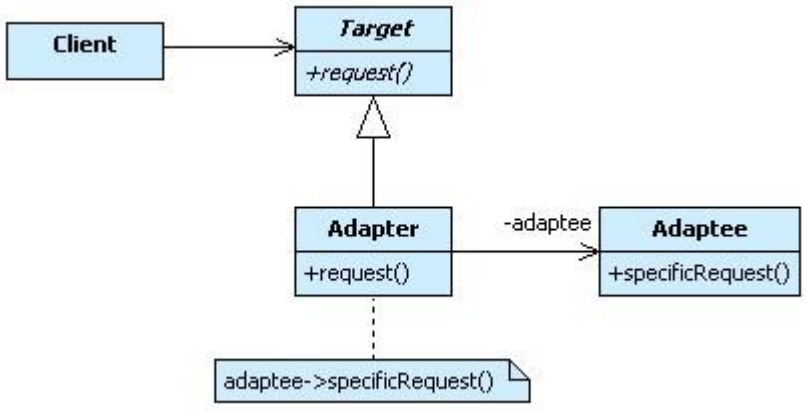
> Контейнеры queue, priority_queue и stack библиотеки стандартных шаблонов STL реализованы на базе последовательных контейнеров list, deque и vector, адаптируя их интерфейсы к нужному виду. Именно поэтому эти контейнеры называют контейнерами-адаптерами.

Описание паттерна Adapter

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

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

UML-диаграмма классов паттерна Adapter



Реализация паттерна Adapter

Классическая реализация паттерна Adapter

Приведем реализацию паттерна Adapter. Для примера выше адаптируем показания температурного датчика системы климат-контроля, переведя их из градусов Фаренгейта в градусы Цельсия (предполагается, что код этого датчика недоступен для модификации).

```
#include <iostream>

// Уже существующий класс температурного датчика окружающей среды
class FahrenheitSensor
{
  public:
    // Получить показания температуры в градусах Фаренгейта
    float getFahrenheitTemp() {
      float t = 32.0;
      // ... какой то код
      return t;
    }
};

class Sensor
{
  public:
    virtual ~Sensor() {}
    virtual float getTemperature() = 0;
};

class Adapter : public Sensor
{
  public:
    Adapter( FahrenheitSensor* p ) : p_fsensor(p) {
```

```
27      }
28     ~Adapter() {
29        delete p_fsensor;
30      }
31      float getTemperature() {
32        return (p_fsensor->getFahrenheitTemp()-32.0)*5.0/9.0;
33      }
34    private:
35      FahrenheitSensor* p_fsensor;
36  };
37
38  int main()
39  {
40    Sensor* p = new Adapter( new FahrenheitSensor);
41    cout << "Celsius temperature = " << p->getTemperature() << endl;
42    delete p;
43    return 0;
      }
```

Реализация паттерна Adapter на основе закрытого наследования

Пусть наш температурный датчик системы климат-контроля поддерживает функцию юстировки для получения более точных показаний. Эта функция не является обязательной для использования, возможно, поэтому соответствующий метод adjust() объявлен разработчиками защищенным в существующем классе FahrenheitSensor.

Разрабатываемая нами система должна поддерживать настройку измерений. Так как доступ к защищенному методу через указатель или ссылку запрещен, то классическая реализация паттерна Adapter здесь уже не подходит. Единственное решение - наследовать от класса FahrenheitSensor. Интерфейс этого класса должен оставаться недоступным пользователю, поэтому наследование должно быть закрытым.

> Цели, преследуемые при использовании открытого и закрытого наследования различны. Если открытое наследование применяется для наследования

> интерфейса и реализации, то закрытое наследование - только для наследования реализации.

```
1   #include <iostream>
2
3   class FahrenheitSensor
4   {
5     public:
6       float getFahrenheitTemp() {
7         float t = 32.0;
8         // ...
9         return t;
10      }
11    protected:
12      void adjust() {} // Настройка датчика (защищенный метод)
13  };
14
15  class Sensor
16  {
17    public:
18      virtual ~Sensor() {}
19      virtual float getTemperature() = 0;
20      virtual void adjust() = 0;
21  };
22
23  class Adapter : public Sensor, private FahrenheitSensor
24  {
25    public:
26      Adapter() { }
27      float getTemperature() {
```

```
28        return (getFahrenheitTemp()-32.0)*5.0/9.0;
29      }
30      void adjust() {
31        FahrenheitSensor::adjust();
32      }
33  };
34
35  int main()
36  {
37    Sensor * p = new Adapter();
38    p->adjust();
39    cout << "Celsius temperature = " << p->getTemperature() << endl;
40    delete p;
41    return 0;
42  }
```

Результаты применения паттерна Adapter

Достоинства паттерна Adapter

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

Недостатки паттерна Adapter

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

Паттерн Мост

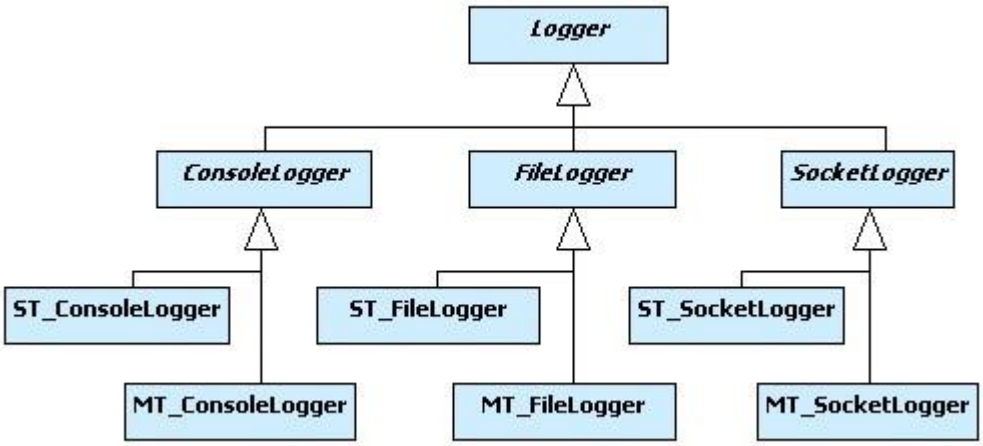
Назначение паттерна Bridge

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

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

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

Стандартный подход на основе полиморфизма использует следующую иерархию классов.



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

Указанные недостатки отсутствует в системе, спроектированной с применением паттерна Bridge.

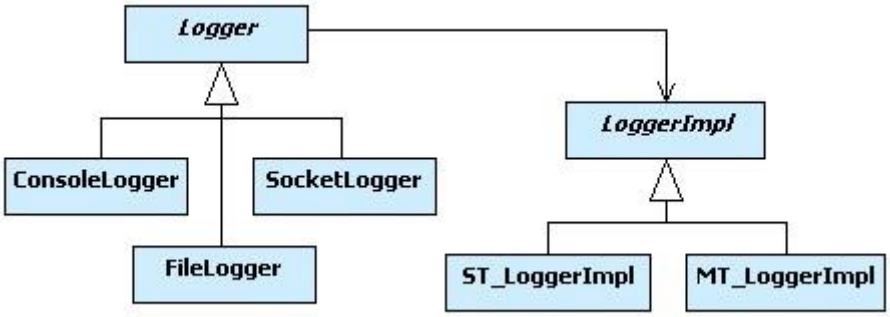
Описание паттерна Bridge

Паттерн Bridge разделяет абстракцию и реализацию на две отдельные иерархии классов так, что их можно изменять независимо друг от друга.

Первая иерархия определяет интерфейс абстракции, доступный пользователю. Для случая проектируемого нами логгера абстрактный базовый класс Logger мог бы объявить интерфейс метода log() для вывода сообщений. Класс Logger также содержит указатель на реализацию pimpl, который инициализируется должным образом при создании логгера конкретного типа. Этот указатель используется для перенаправления

пользовательских запросов в реализацию. Заметим, в общем случае подклассы ConsoleLogger, FileLogger и SocketLogger могут расширять интерфейс класса Logger.

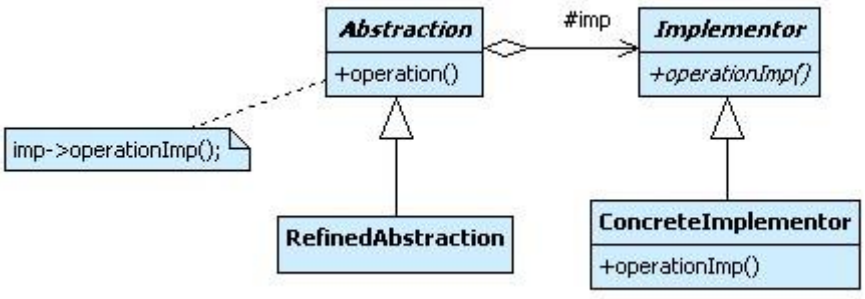
Все детали реализации, связанные с особенностями среды скрываются во второй иерархии. Базовый класс LoggerImpl объявляет интерфейс операций, предназначенных для отправки сообщений на экран, файл и удаленный компьютер, а подклассы ST_LoggerImpl и MT_LoggerImpl его реализуют для однопоточной и многопоточной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



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

Структура паттерна Bridge

UML-диаграмма классов паттерна Bridge



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

Реализация паттерна Bridge

Приведем реализацию логгера с применением паттерна Bridge.

```
1   // Logger.h - Абстракция
2   #include <string>
3
4   // Опережающее объявление
5   class LoggerImpl;
6
7   class Logger
8   {
9     public:
10      Logger( LoggerImpl* p );
11      virtual ~Logger( );
12      virtual void log( string & str ) = 0;
13    protected:
14      LoggerImpl * pimpl;
15  };
16
17  class ConsoleLogger : public Logger
18  {
19    public:
20      ConsoleLogger();
21      void log( string & str );
22  };
23
24  class FileLogger : public Logger
25  {
26    public:
27      FileLogger( string & file_name );
28      void log( string & str );
```

```
29      private:
30        string file;
31    };
32
33    class SocketLogger : public Logger
34    {
35      public:
36        SocketLogger( string & remote_host, int remote_port );
37        void log( string & str );
38      private:
39        string host;
40        int    port;
41    };
42
43
44    // Logger.cpp - Абстракция
45    #include "Logger.h"
46    #include "LoggerImpl.h"
47
48    Logger::Logger( LoggerImpl* p ) : pimpl(p)
49    { }
50
51    Logger::~Logger( )
52    {
53        delete pimpl;
54    }
55
56    ConsoleLogger::ConsoleLogger() : Logger(
```

```
57          #ifdef MT
58            new MT_LoggerImpl()
59          #else
60            new ST_LoggerImpl()
61          #endif
62        )
63    { }
64
65    void ConsoleLogger::log( string & str )
66    {
67        pimpl->console_log( str);
68    }
69
70    FileLogger::FileLogger( string & file_name ) : Logger(
71          #ifdef MT
72            new MT_LoggerImpl()
73          #else
74            new ST_LoggerImpl()
75          #endif
76        ), file(file_name)
77    { }
78
79    void FileLogger::log( string & str )
80    {
81        pimpl->file_log( file, str);
82    }
83
84    SocketLogger::SocketLogger( string & remote_host,
```

```
85                          int remote_port ) : Logger(
86        #ifdef MT
87          new MT_LoggerImpl()
88        #else
89          new ST_LoggerImpl()
90        #endif
91       ), host(remote_host), port(remote_port)
92   { }
93
94   void SocketLogger::log( string & str )
95   {
96      pimpl->socket_log( host, port, str);
97   }
98
99
100  // LoggerImpl.h - Реализация
101  #include <string>
102
103  class LoggerImpl
104  {
105    public:
106      virtual ~LoggerImpl( ) {}
107      virtual void console_log( string & str ) = 0;
108      virtual void file_log(
109                  string & file, string & str ) = 0;
110      virtual void socket_log(
111            tring & host, int port, string & str ) = 0;
112  };
```

```
113
114  class ST_LoggerImpl : public LoggerImpl
115  {
116    public:
117      void console_log( string & str );
118      void file_log   ( string & file, string & str );
119      void socket_log (
120              string & host, int port, string & str );
121  };
122
123  class MT_LoggerImpl : public LoggerImpl
124  {
125    public:
126      void console_log( string & str );
127      void file_log   ( string & file, string & str );
128      void socket_log (
129              string & host, int port, string & str );
130  };
131
132
133  // LoggerImpl.cpp - Реализация
134  #include <iostream>
135  #include "LoggerImpl.h"
136
137
138  void ST_LoggerImpl::console_log( string & str )
139  {
140    cout << "Single-threaded console logger" << endl;
```

```
141  }
142
143  void ST_LoggerImpl::file_log( string & file, string & str )
144  {
145    cout << "Single-threaded file logger" << endl;
146  }
147
148  void ST_LoggerImpl::socket_log(
149                      string & host, int port, string & str )
150  {
151    cout << "Single-threaded socket logger" << endl;
152  };
153
154  void MT_LoggerImpl::console_log( string & str )
155  {
156    cout << "Multithreaded console logger" << endl;
157  }
158
159  void MT_LoggerImpl::file_log( string & file, string & str )
160  {
161    cout << "Multithreaded file logger" << endl;
162  }
163
164  void MT_LoggerImpl::socket_log(
165                      string & host, int port, string & str )
166  {
167    cout << "Multithreaded socket logger" << endl;
168  }
```

```
169
170
171  // Main.cpp
172  #include <string>
173  #include "Logger.h"
174
175  int main()
176  {
177    Logger * p = new FileLogger( string("log.txt"));
178    p->log( string("message"));
179    delete p;
180    return 0;
181  }
```

Отметим несколько важных моментов приведенной реализации паттерна Bridge:

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

Результаты применения паттерна Bridge

Достоинства паттерна Bridge

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

Паттерн Компоновщик

Назначение паттерна Composite

Используйте паттерн Composite если:

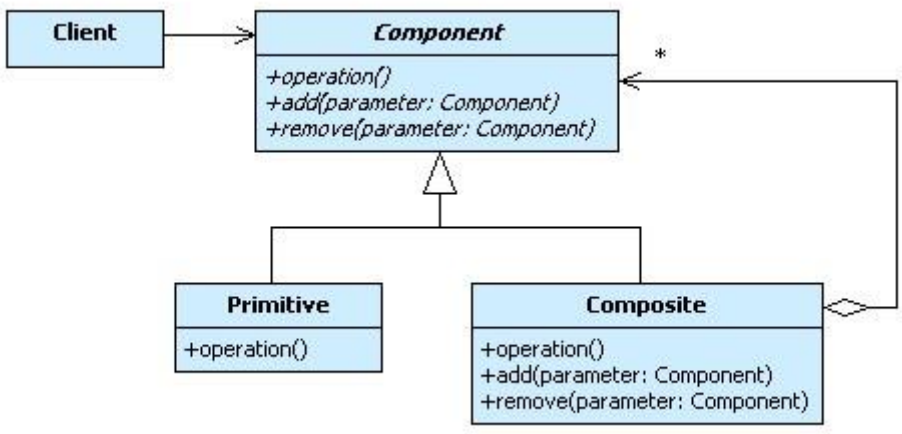
- Необходимо объединять группы схожих объектов и управлять ими.
- Объекты могут быть как примитивными (элементарными), так и составными (сложными). Составной объект может включать в себя коллекции других объектов, образуя сложные древовидные структуры. Пример: директория файловой системы состоит из элементов,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

Описание паттерна Composite

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

Для военной стратегической игры ”Пунические войны ”, описывающей военное противостояние между Римом и Карфагеном (см. раздел Порождающие паттерны), каждая боевая единица (всадник, лучник, пехотинец) имеет свою собственную разрушающую силу. Эти единицы могут объединяться в группы для образования более сложных военных подразделений, например, римские легионы, которые, в свою очередь, объединяясь, образуют целую армию. Как рассчитать боевую мощь таких иерархических соединений?
Паттерн Composite предлагает следующее решение. Он вводит абстрактный базовый класс `Component` с поведением, общим для всех примитивных и составных объектов. Для случая стратегической игры - это метод `getStrength()` для подсчета разрушающей силы.
Подклассы `Primitive` and `Composite` являются производными от класса `Component`. Составной объект `Composite` хранит компоненты-потомки абстрактного типа `Component`, каждый из которых может быть также `Composite`.

UML-диаграмма классов паттерна Composite



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

Реализация паттерна Composite

Применим паттерн Composite для нашей стратегической игры. Сначала сформируем различные военные соединения римской армии, а затем рассчитаем разрушающую силу.

```
1   #include <iostream>
2   #include <vector>
3   #include <assert.h>
4
5   // Component
6   class Unit
7   {
8     public:
9       virtual int getStrength() = 0;
10      virtual void addUnit(Unit* p) {
11        assert( false);
12      }
13      virtual ~Unit() {}
14  };
15
16  // Primitives
17  class Archer: public Unit
18  {
19    public:
20      virtual int getStrength() {
21        return 1;
22      }
23  };
```

```
24
25  class Infantryman: public Unit
26  {
27    public:
28      virtual int getStrength() {
29        return 2;
30      }
31  };
32
33  class Horseman: public Unit
34  {
35    public:
36      virtual int getStrength() {
37        return 3;
38      }
39  };
40
41
42  // Composite
43  class CompositeUnit: public Unit
44  {
45    public:
46      int getStrength() {
47        int total = 0;
48        for(int i=0; i<c.size(); ++i)
49          total += c[i]->getStrength();
50        return total;
51      }
```

```
52      void addUnit(Unit* p) {
53          c.push_back( p);
54      }
55      ~CompositeUnit() {
56        for(int i=0; i<c.size(); ++i)
57          delete c[i];
58      }
59    private:
60     std::vector<Unit*> c;
61  };
62
63
64  // Вспомогательная функция для создания легиона
65  CompositeUnit* createLegion()
66  {
67    // Римский легион содержит:
68    CompositeUnit* legion = new CompositeUnit;
69    // 3000 тяжелых пехотинцев
70    for (int i=0; i<3000; ++i)
71      legion->addUnit(new Infantryman);
72    // 1200 легких пехотинцев
73    for (int i=0; i<1200; ++i)
74      legion->addUnit(new Archer);
75    // 300 всадников
76    for (int i=0; i<300; ++i)
77      legion->addUnit(new Horseman);
78
79    return legion;
```

```
80  }
81
82  int main()
83  {
84    // Римская армия состоит из 4-х легионов
85    CompositeUnit* army = new CompositeUnit;
86    for (int i=0; i<4; ++i)
87      army->addUnit( createLegion());
88
89    cout << "Roman army damaging strength is "
90         << army->getStrength() << endl;
91    // …
92    delete army;
93    return 0;
94  }
```

Следует обратить внимание на один важный момент. Абстрактный базовый класс `Unit` объявляет интерфейс для добавления новых боевых единиц addUnit(), несмотря на то, что объектам примитивных типов (`Archer`, `Infantryman`, `Horseman`) подобная операция не нужна. Сделано это в угоду прозрачности системы в ущерб ее безопасности. Клиент знает, что объект типа `Unit` всегда будет иметь метод `addUnit()`. Однако его вызов для примитивных объектов считается ошибочным и небезопасным. Можно сделать систему более безопасной, переместив метод `addUnit()` в составной объект `CompositeUnit`. Однако при этом возникает следующая проблема: мы не знаем, содержит ли объект `Unit` метод `addUnit()`.

Рассмотрим следующий фрагмент кода.

```
1   class Unit
2   {
3     public:
4       virtual CompositeUnit* getComposite() {
5         return 0;
6       }
```

```
7       // …
8   };
9
10  // Composite
11  class CompositeUnit: public Unit
12  {
13    public:
14     void addUnit(Unit* p);
15      CompositeUnit* getComposite() {
16        return this;
17      }
18      // …
19  };
```

В абстрактном базовом классе `Unit` появился новый виртуальный метод `getComposite()` с реализацией по умолчанию, которая возвращает 0. Класс `CompositeUnit` переопределяет этот метод, возвращая указатель на самого себя. Благодаря этому методу можно запросить у компонента его тип. Если он составной, то можно применить операцию `addUnit()`.

```
1   if (unit->getComposite())
2   {
3     unit->getComposite()->addUnit( new Archer);
4   }
```

Результаты применения паттерна Composite

Достоинства паттерна Composite

- В систему легко добавлять новые примитивные или составные объекты, так как паттерн Composite использует общий базовый класс Component.
- Код клиента имеет простую структуру – примитивные и составные объекты обрабатываются одинаковым образом.
- Паттерн Composite позволяет легко обойти все узлы древовидной структуры

Недостатки паттерна Composite

- Неудобно осуществить запрет на добавление в составной объект Composite объектов определенных типов. Так, например, в состав римской армии не могут входить боевые слоны.

Паттерн Декоратор

Назначение паттерна Decorator

- Паттерн Decorator динамически добавляет новые обязанности объекту. Декораторы являются гибкой альтернативой порождению подклассов для расширения функциональности.
- Рекурсивно декорирует основной объект.
- Паттерн Decorator использует схему "обертываем подарок, кладем его в коробку, обертываем коробку".

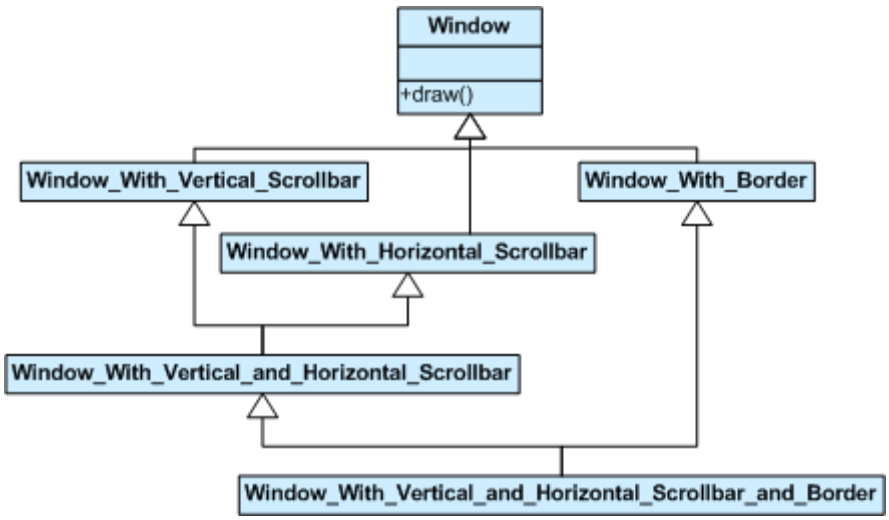
Решаемая проблема

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

Обсуждение паттерна Decorator

Предположим, вы работаете над библиотекой для построения графических пользовательских интерфейсов и хотите иметь возможность добавлять в окно рамку и полосу прокрутки. Тогда вы могли бы определить иерархию наследования следующим образом...

UML-диаграмма классов паттерна Decorator

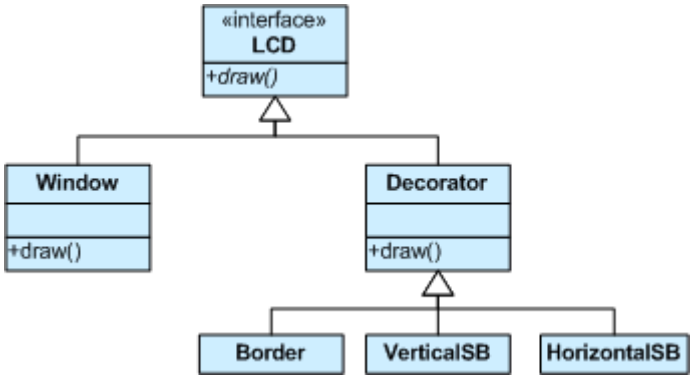


Эта схема имеет существенный недостаток - число классов сильно разрастается.

Паттерн Decorator дает клиенту возможность задавать любые комбинации желаемых "особенностей".

```
1   Widget*  aWidget = new BorderDecorator(
2                       new HorizontalScrollBarDecorator(
3                         new VerticalScrollBarDecorator(
4                           new Window( 80, 24 ))));
5   aWidget->draw();
```

Гибкость может быть достигнута следующим дизайном.



Другой пример каскадного соединения свойств (в цепочку) для придания объекту нужных характеристик…

```
1   Stream*  aStream = new CompressingStream(
2                       new ASCII7Stream(
3                         new FileStream( "fileName.dat" )));
4   aStream->putString( "Hello world" );
```

Решение этого класса задач предполагает инкапсуляцию исходного объекта в абстрактный интерфейс. Как объекты-декораторы, так и основной объект наследуют от этого абстрактного интерфейса. Интерфейс использует рекурсивную композицию для добавления к основному объекту неограниченного количества "слоев" - декораторов.

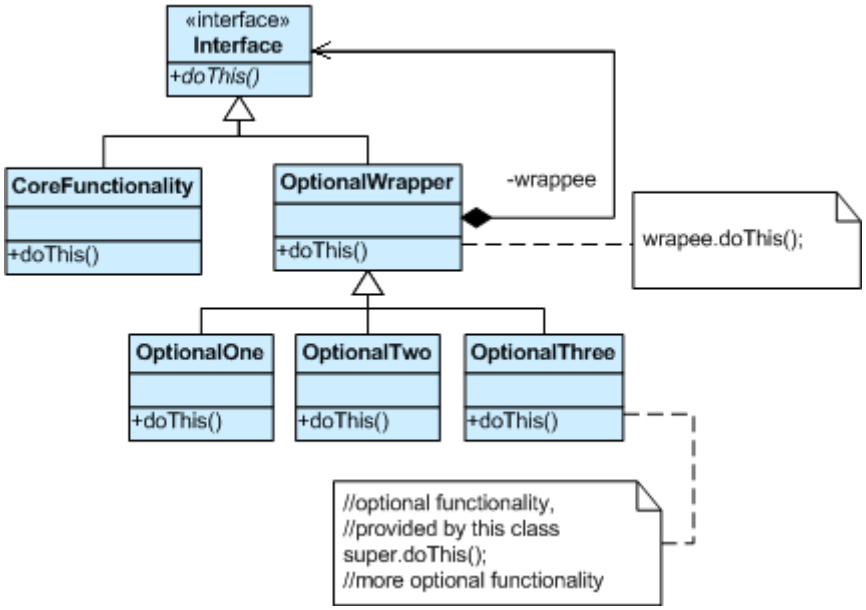
Обратите внимание, паттерн Decorator позволяет добавлять объекту новые обязанности, не изменяя его интерфейс (новые методы не добавляются). Известный клиенту интерфейс должен оставаться постоянным на всех, следующих друг за другом "слоях".

Отметим также, что основной объект теперь "скрыт" внутри объекта-декоратора. Доступ к основному объекту теперь проблематичен.

Структура паттерна Decorator

Клиент всегда заинтересован в функциональности `CoreFunctionality.doThis()`. Клиент может или не может быть заинтересован в

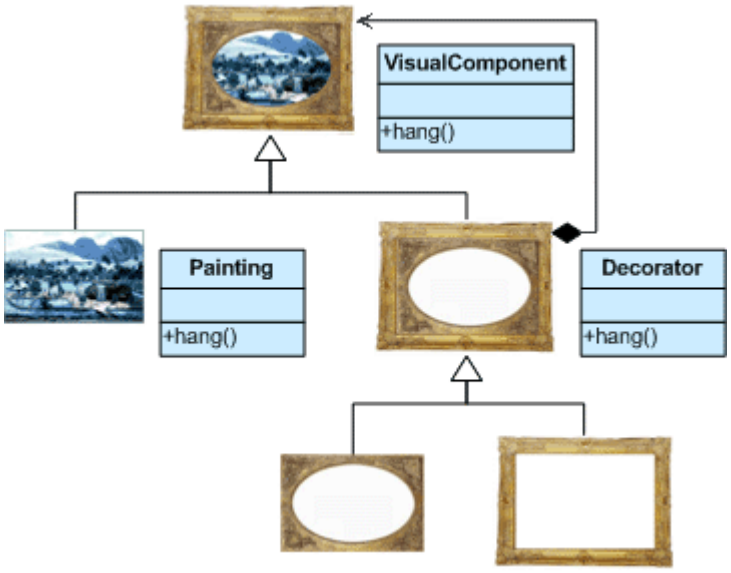
методах `OptionalOne.doThis()` и `OptionalTwo.doThis()`. Каждый из этих классов переадресует запрос базовому классу `Decorator`, а тот направляет его в декорируемый объект.



Пример паттерна Decorator

Паттерн Decorator динамически добавляет новые обязанности объекту. Украшения для новогодней елки являются примерами декораторов. Огни, гирлянды, игрушки и т.д. вешают на елку для придания ей праздничного вида. Украшения не меняют саму елку, а только делают ее новогодней.

Хотя картины можно повесить на стену и без рамок, рамки часто добавляются для придания нового стиля.



Использование паттерна Decorator

- Подготовьте исходные данные: один основной компонент и несколько дополнительных (необязательных) "оберток".

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

Особенности паттерна Decorator

- Adapter придает своему объекту новый интерфейс, Proxy предоставляет тот же интерфейс, а Decorator обеспечивает расширенный интерфейс.
- Adapter изменяет интерфейс объекта. Decorator расширяет ответственность объекта. Decorator, таким образом, более прозрачен для клиента. Как следствие, Decorator поддерживает рекурсивную композицию, что невозможно с чистыми адаптерами.
- Decorator можно рассматривать как вырожденный случай Composite с единственным компонентом. Однако Decorator добавляет новые обязанности и не предназначен для агрегирования объектов.
- Decorator позволяет добавлять новые функции к объектам без наследования. Composite фокусирует внимание на представлении, а не декорировании. Эти характеристики являются различными, но взаимодополняющими, поэтому Composite и Decorator часто используются вместе.
- Decorator и Proxy имеют разное назначение, но схожие структуры. Их реализации хранят ссылку на объект, которому они отправляют запросы.
- Decorator позволяет изменить внешний облик объекта, Strategy – его внутреннее содержание.

Реализация паттерна Decorator

Паттерн Decorator: до и после

До

Используется следующая иерархия наследования:

```
1   class A {
2     public:
3       virtual void do_it() {
4           cout << 'A';
5       }
6   };
7
8   class AwithX: public A {
9     public:
10      /*virtual*/
11      void do_it() {
12          A::do_it();
13          do_X();
14      };
15    private:
16      void do_X() {
17          cout << 'X';
18      }
19  };
20
21  class AwithY: public A {
22    public:
23      /*virtual*/
24      void do_it() {
25          A::do_it();
26          do_Y();
27      }
28    protected:
```

```
29      void do_Y() {
30          cout << 'Y';
31      }
32  };
33
34  class AwithZ: public A {
35    public:
36      /*virtual*/
37      void do_it() {
38          A::do_it();
39          do_Z();
40      }
41    protected:
42      void do_Z() {
43          cout << 'Z';
44      }
45  };
46
47  class AwithXY: public AwithX, public AwithY
48  {
49    public:
50      /*virtual*/
51      void do_it() {
52          AwithX::do_it();
53          AwithY::do_Y();
54      }
55  };
56
```

```
57  class AwithXYZ: public AwithX, public AwithY, public AwithZ
58  {
59    public:
60      /*virtual*/
61      void do_it() {
62          AwithX::do_it();
63          AwithY::do_Y();
64          AwithZ::do_Z();
65      }
66  };
67
68  int main() {
69    AwithX anX;
70    AwithXY anXY;
71    AwithXYZ anXYZ;
72    anX.do_it();
73    cout << '\n';
74    anXY.do_it();
75    cout << '\n';
76    anXYZ.do_it();
77    cout << '\n';
78  }
```

Вывод программы:

```
1  AX
2  AXY
3  AXYZ
```

После

Заменим наследование делегированием.

Обсуждение. Используйте агрегирование вместо наследования для декорирования "основного" объекта. Тогда клиент сможет динамически добавлять новые обязанности объектам, в то время как архитектура, основанная на множественном наследовании, является статичной.

```
1   class I {
2     public:
3       virtual ~I(){}
4       virtual void do_it() = 0;
5   };
6
7   class A: public I {
8     public:
9       ~A() {
10          cout << "A dtor" << '\n';
11      }
12      /*virtual*/
13      void do_it() {
14          cout << 'A';
15      }
16  };
17
18  class D: public I {
19    public:
20      D(I *inner) {
21          m_wrappee = inner;
22      }
23      ~D() {
24          delete m_wrappee;
25      }
26      /*virtual*/
```

```
27      void do_it() {
28          m_wrappee->do_it();
29      }
30    private:
31      I *m_wrappee;
32  };
33
34  class X: public D {
35    public:
36      X(I *core): D(core){}
37      ~X() {
38          cout << "X dtor" << "    ";
39      }
40      /*virtual*/
41      void do_it() {
42          D::do_it();
43          cout << 'X';
44      }
45  };
46
47  class Y: public D {
48    public:
49      Y(I *core): D(core){}
50      ~Y() {
51          cout << "Y dtor" << "    ";
52      }
53      /*virtual*/
54      void do_it() {
```

```
55          D::do_it();
56          cout << 'Y';
57      }
58  };
59
60  class Z: public D {
61    public:
62      Z(I *core): D(core){}
63      ~Z() {
64          cout << "Z dtor" << "    ";
65      }
66      /*virtual*/
67      void do_it() {
68          D::do_it();
69          cout << 'Z';
70      }
71  };
72
73  int main() {
74    I *anX = new X(new A);
75    I *anXY = new Y(new X(new A));
76    I *anXYZ = new Z(new Y(new X(new A)));
77    anX->do_it();
78    cout << '\n';
79    anXY->do_it();
80    cout << '\n';
81    anXYZ->do_it();
82    cout << '\n';
```

```
83     delete anX;
84     delete anXY;
85     delete anXYZ;
86   }
```

Вывод программы:

```
1   AX
2   AXY
3   AXYZ
4   X dtor   A dtor
5   Y dtor   X dtor   A dtor
6   Z dtor   Y dtor   X dtor   A dtor
```

Паттерн проектирования Decorator по шагам

- Создайте "наименьший общий знаменатель", делающий классы взаимозаменяемыми.
- Создайте базовый класс второго уровня для реализации дополнительной функциональности.
- Основной класс и класс-декоратор используют отношение "является".
- Класс-декоратор "имеет" экземпляр "наименьшего общего знаменателя".
- Класс Decorator делегирует выполнение операции объекту "имеет".
- Для реализации каждой дополнительной функциональности создайте подклассы Decorator.
- Подклассы Decorator делегируют выполнение операции базовому классу и реализуют дополнительную функциональность.
- Клиент несет ответственность за конфигурирование нужной функциональности.

```
1    #include <iostream>
2    using namespace std;
3
4    // 1. " Наименьший общий знаменатель"
5    class Widget
6    {
7      public:
```

```
8       virtual void draw() = 0;
9   };
10
11  // 3. Основной класс, использующий отношение "является"
12  class TextField: public Widget
13  {
14      int width, height;
15    public:
16      TextField(int w, int h)
17      {
18          width = w;
19          height = h;
20      }
21
22      /*virtual*/
23      void draw()
24      {
25          cout << "TextField: " << width << ", " << height << '\n';
26      }
27  };
28
29  // 2. Базовый класс второго уровня
30  class Decorator: public Widget  // 3. использует отношение
    "является"
31  {
32      Widget *wid; // 4. отношение "имеет"
33    public:
34      Decorator(Widget *w)
35      {
```

```
36          wid = w;
37      }
38
39      /*virtual*/
40      void draw()
41      {
42          wid->draw(); // 5. делегирование
43      }
44  };
45
46  // 6. Дополнительное декорирование
47  class BorderDecorator: public Decorator
48  {
49    public:
50      BorderDecorator(Widget *w): Decorator(w){}
51
52      /*virtual*/
53      void draw()
54      {
55          // 7. Делегирование базовому классу и
56          Decorator::draw();
57          // 7. реализация дополнительной функциональности
58          cout << "   BorderDecorator" << '\n';
59      }
60  };
61
62  // 6. Дополнительное декорирование
63  class ScrollDecorator: public Decorator
```

```
{
  public:
    ScrollDecorator(Widget *w): Decorator(w){}

    /*virtual*/
    void draw()
    {
        // 7. Delegate to base class and add extra stuff
        Decorator::draw();
        cout << "   ScrollDecorator" << '\n';
    }
};

int main()
{
  // 8. Клиент ответственен за конфигурирование нужной функциональности
  Widget *aWidget = new BorderDecorator(
                    new BorderDecorator(
                      new ScrollDecorator
                        (new TextField(80, 24))));
  aWidget->draw();
}
TextField: 80, 24
   ScrollDecorator
   BorderDecorator
   BorderDecorator
```

Паттерн Фасад

Назначение паттерна Facade

- Паттерн Facade предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Facade определяет интерфейс более высокого уровня, упрощающий использование подсистемы.
- Паттерн Facade "обертывает" сложную подсистему более простым интерфейсом.

Решаемая проблема

Клиенты хотят получить упрощенный интерфейс к общей функциональности сложной подсистемы.

Обсуждение паттерна Facade

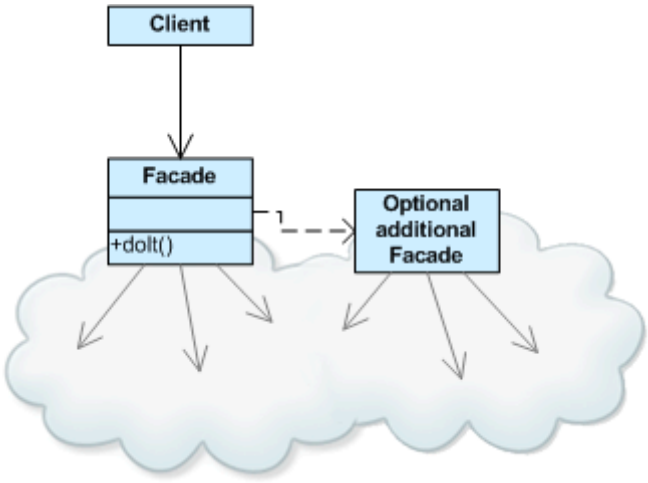
Паттерн Facade инкапсулирует сложную подсистему в единственный интерфейсный объект. Это позволяет сократить время изучения подсистемы, а также способствует уменьшению степени связанности между подсистемой и потенциально большим количеством клиентов. С другой стороны, если фасад является единственной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

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

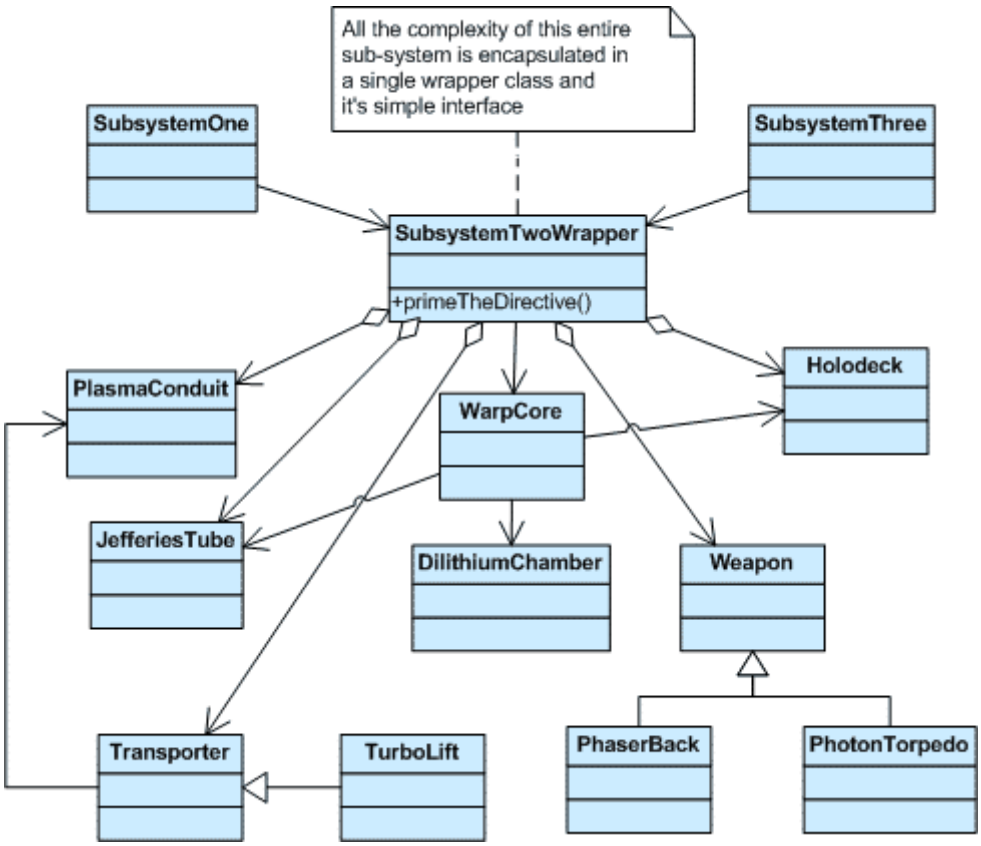
Структура паттерна Facade

Клиенты общаются с подсистемой через Facade. При получении запроса от клиента объект Facade переадресует его нужному компоненту подсистемы. Для клиентов компоненты подсистемы остаются "тайной, покрытой мраком".

UML-диаграмма классов паттерна Facade

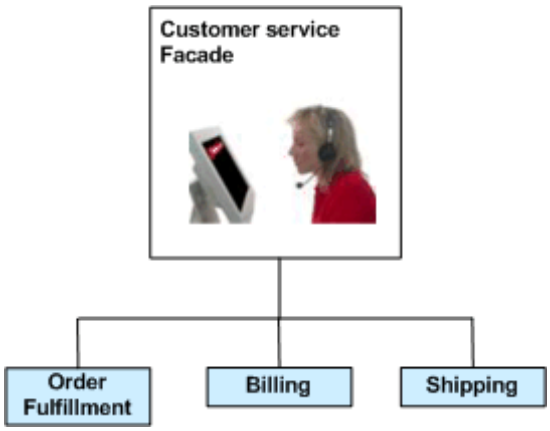


Подсистемы SubsystemOne и SubsystemThree не взаимодействуют напрямую с внутренними компонентами подсистемы SubsystemTwo. Они используют "фасад" SubsystemTwoWrapper (т.е. абстракцию более высокого уровня).



Пример паттерна Facade

Паттерн Facade определяет унифицированный высокоуровневый интерфейс к подсистеме, что упрощает ее использование. Покупатели сталкиваются с фасадом при заказе каталожных товаров по телефону. Покупатель звонит в службу поддержки клиентов и перечисляет товары, которые хочет приобрести. Представитель службы выступает в качестве "фасада", обеспечивая интерфейс к отделу исполнения заказов, отделу продаж и службе доставки.



Использование паттерна Facade

- Определите для подсистемы простой, унифицированный интерфейс.
- Спроектируйте класс "обертку", инкапсулирующий подсистему.

- Вся сложность подсистемы и взаимодействие ее компонентов скрыты от клиентов. "Фасад" / "обертка" переадресует пользовательские запросы подходящим методам подсистемы.
- Клиент использует только "фасад".
- Рассмотрите вопрос о целесообразности создания дополнительных "фасадов".

Особенности паттерна Facade

- Facade определяет новый интерфейс, в то время как Adapter использует уже имеющийся. Помните, Adapter делает работающими вместе два существующих интерфейса, не создавая новых.
- Если Flyweight показывает, как сделать множество небольших объектов, то Facade показывает, как сделать один объект, представляющий целую подсистему.
- Mediator похож на Facade тем, что абстрагирует функциональность существующих классов. Однако Mediator централизует функциональность между объектами-коллегами, не присущую ни одному из них. Коллеги обмениваются информацией друг с другом через Mediator. С другой стороны, Facade определяет простой интерфейс к подсистеме, не добавляет новой функциональности и не известен классам подсистемы.
- Abstract Factory может применяться как альтернатива Facade для сокрытия платформенно-зависимых классов.
- Объекты "фасадов" часто являются Singleton, потому что требуется только один объект Facade.
- Adapter и Facade в являются "обертками", однако эти "обертки" разных типов. Цель Facade – создание более простого интерфейса, цель Adapter – адаптация существующего интерфейса. Facade обычно "обертывает" несколько объектов, Adapter "обертывает" один объект.

Реализация паттерна Facade

Разбиение системы на компоненты позволяет снизить ее сложность. Ослабить связи между компонентами системы можно с помощью паттерна Facade. Объект "фасад" предоставляет единый упрощенный интерфейс к компонентам системы.

В примере ниже моделируется система сетевого обслуживания. Фасад FacilitiesFacade скрывает внутреннюю структуру системы. Пользователь, сделав однажды запрос на обслуживание, затем 1-2 раза в неделю в течение 5 месяцев справляется о ходе выполнения работ до тех пор, пока его запрос не будет полностью обслужен.

```
1    #include <iostream.h>
2
```

```
3   class MisDepartment
4   {
5     public:
6       void submitNetworkRequest()
7       {
8           _state = 0;
9       }
10      bool checkOnStatus()
11      {
12          _state++;
13          if (_state == Complete)
14            return 1;
15          return 0;
16      }
17    private:
18      enum States
19      {
20          Received, DenyAllKnowledge, ReferClientToFacilities,
21          FacilitiesHasNotSentPaperwork, ElectricianIsNotDone,
22          ElectricianDidItWrong, DispatchTechnician, SignedOff,
23          DoesNotWork, FixElectriciansWiring, Complete
24      };
25      int _state;
26  };
27
28  class ElectricianUnion
29  {
30    public:
```

```
31      void submitNetworkRequest()
32      {
33          _state = 0;
34      }
35      bool checkOnStatus()
36      {
37          _state++;
38          if (_state == Complete)
39            return 1;
40          return 0;
41      }
42    private:
43      enum States
44      {
45          Received, RejectTheForm, SizeTheJob, SmokeAndJokeBreak,
46          WaitForAuthorization, DoTheWrongJob, BlameTheEngineer,
47          WaitToPunchOut, DoHalfAJob, ComplainToEngineer,
48          GetClarification, CompleteTheJob, TurnInThePaperwork,
49          Complete
50      };
51      int _state;
52  };
53
54  class FacilitiesDepartment
55  {
56    public:
57      void submitNetworkRequest()
58      {
```

```
59          _state = 0;
60      }
61      bool checkOnStatus()
62      {
63          _state++;
64          if (_state == Complete)
65            return 1;
66          return 0;
67      }
68    private:
69      enum States
70      {
71          Received, AssignToEngineer, EngineerResearches,
72          RequestIsNotPossible, EngineerLeavesCompany,
73          AssignToNewEngineer, NewEngineerResearches,
74          ReassignEngineer,EngineerReturns,
75          EngineerResearchesAgain, EngineerFillsOutPaperWork,
76          Complete
77      };
78      int _state;
79  };
80
81  class FacilitiesFacade
82  {
83    public:
84      FacilitiesFacade()
85      {
86          _count = 0;
```

```
87      }
88      void submitNetworkRequest()
89      {
90          _state = 0;
91      }
92      bool checkOnStatus()
93      {
94          _count++;
95          /* Запрос на обслуживание получен */
96          if (_state == Received)
97          {
98              _state++;
99              /* Перенаправим запрос инженеру */
100             _engineer.submitNetworkRequest();
101             cout << "submitted to Facilities - " << _count
102                 << " phone calls so far" << endl;
103         }
104         else if (_state == SubmitToEngineer)
105         {
106             /* Если инженер свою работу выполнил,
107               перенаправим запрос электрику */
108             if (_engineer.checkOnStatus())
109             {
110                 _state++;
111                 _electrician.submitNetworkRequest();
112                 cout << "submitted to Electrician - " << _count
113                     << " phone calls so far" << endl;
114             }
```

```
115         }
116         else if (_state == SubmitToElectrician)
117         {
118            /* Если электрик свою работу выполнил,
119               перенаправим запрос технику */
120            if (_electrician.checkOnStatus())
121            {
122               _state++;
123               _technician.submitNetworkRequest();
124               cout << "submitted to MIS - " << _count
125                    << " phone calls so far" << endl;
126            }
127         }
128         else if (_state == SubmitToTechnician)
129         {
130            /* Если техник свою работу выполнил,
131               то запрос обслужен до конца */
132            if (_technician.checkOnStatus())
133              return 1;
134         }
135         /* Запрос еще не обслужен до конца */
136         return 0;
137      }
138      int getNumberOfCalls()
139
140      {
141         return _count;
142      }
```

```
143   private:
144     enum States
145     {
146         Received, SubmitToEngineer, SubmitToElectrician,
147         SubmitToTechnician
148     };
149     int _state;
150     int _count;
151     FacilitiesDepartment _engineer;
152     ElectricianUnion _electrician;
153     MisDepartment _technician;
154 };
155
156 int main()
157 {
158   FacilitiesFacade facilities;
159
160   facilities.submitNetworkRequest();
161   /* Звоним, пока работа не выполнена полностью */
162   while (!facilities.checkOnStatus())
163     ;
164   cout << "job completed after only "
165        << facilities.getNumberOfCalls()
166        << " phone calls" << endl;
167 }
```

Вывод программы:

```
1   submitted to Facilities - 1 phone calls so far
2   submitted to Electrician - 12 phone calls so far
```

3 submitted to MIS - 25 phone calls so far

4 job completed after only 35 phone calls

Паттерн Единая точка входа

Единая точка входа (англ. Front controller) — обеспечивает унифицированный интерфейс для интерфейсов в подсистеме. Front Controller определяет высокоуровневый интерфейс, упрощающий использование подсистемы.

В сложных веб-сайтах есть много одинаковых действий, которые надо производить во время обработки запросов. Это, например, контроль безопасности, многоязычность и настройка интерфейса пользователя. Когда поведение входного контроллера разбросано между несколькими объектами, дублируется большое количество кода. Помимо прочего возникают сложности смены поведения в реальном времени.

Паттерн Front Controller объединяет всю обработку запросов, пропуская запросы через единственный объект-обработчик. Этот объект содержит общую логику поведения, которая может быть изменена в реальном времени при помощи декораторов. После обработки запроса контроллер обращается к конкретному объекту для отработки конкретного поведения.

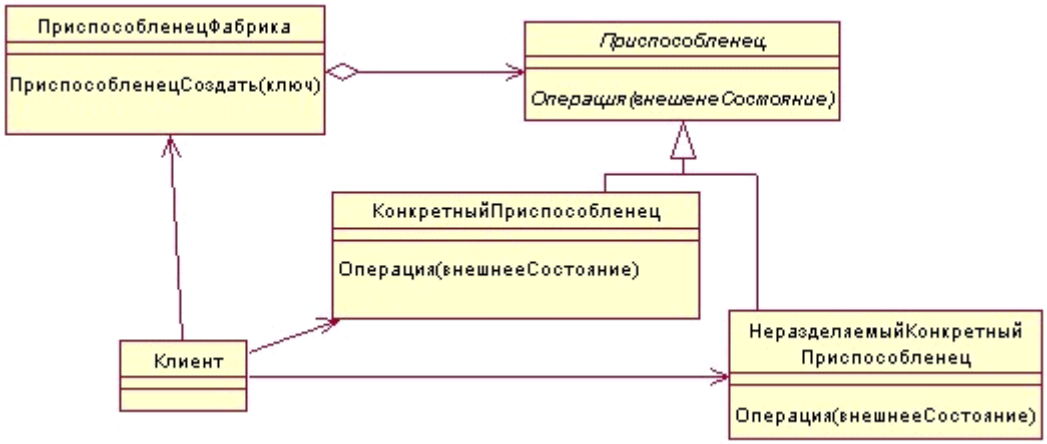
Паттерн Приспособленец

Необходимо обеспечить поддержку множества мелких обьектов. Приспособленцы моделируют сущности, число которых слишком велико для представления объектами. Имеет смысл использовать данный паттерн если одновременно выполняются следующие условия:

- в приложении используется большое число обьектов, из-за этого расходы на хранение высоки,
- большую часть состояния обьектов можно вынести вовне,
- многие группы обьектов можно заменить относительно небольшим количеством обьектов, поскольку состояния обьектов вынесены вовне.
- Создать разделяемый объект, который можно использовать одновременно в нескольких контекстах, причем, в каждом контексте он выглядит как независимый объект (неотличим от экземпляра, который не разделяется). "Приспособленец" объявляет интерфейс, с помощью которого приспособленцы могут получить внешнее состояние или как-то воздействовать на него, "КонкретныйПриспособленец" реализует интерфейс класса "Приспособленец" и добавляет при необходимости внутреннее состояние. Внутреннее состояние хранится в объекте "КонкретныйПриспособленец", в то время как внешнее состояние

хранится или вычисляется "Клиентами" ("Клиент" передает его "Приспособленцу" при вызове операций).

- Объект класса "КонкретныйПриспособленец" должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть независимым от контекста, "ПриспособленецФабрика" - создает объекты - "Приспособленцы" (или предоставляет существующий экземпляр) и управляет ими. "НеразделяемыйКонкретныйПриспособленец" - не все подклассы "Приспособленца" обязательно должны быть разделяемыми. "Клиент" - хранит ссылки на одного или нескольких "Приспособленцев", вычисляет и хранит внешнее состояние "Приспособленцев".



-

Паттерн Заместитель

Назначение паттерна Proxy

- Паттерн Proxy является суррогатом или заместителем другого объекта и контролирует доступ к нему.
- Предоставляя дополнительный уровень косвенности при доступе к объекту, может применяться для поддержки распределенного, управляемого или интеллектуального доступа.
- Являясь "оберткой" реального компонента, защищает его от излишней сложности.

Решаемая проблема

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

Обсуждение паттерна Proxy

Суррогат или заместитель это объект, интерфейс которого идентичен интерфейсу реального объекта. При первом запросе клиента заместитель создает реальный объект, сохраняет его адрес и затем отправляет запрос этому реальному объекту. Все последующие запросы просто переадресуются инкапсулированному реальному объекту.

Существует четыре ситуации, когда можно использовать паттерн Proxy:

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

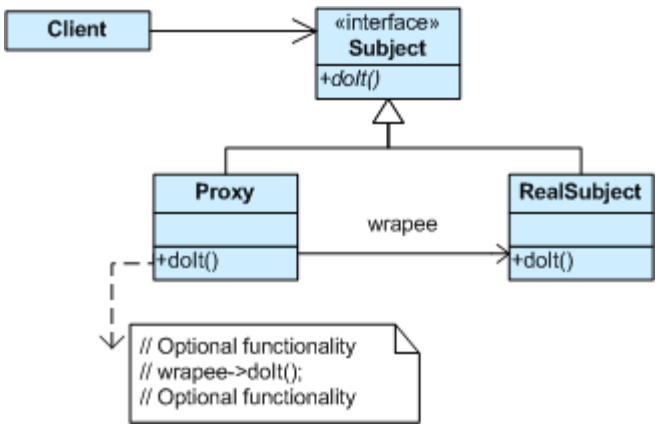
Вот типичные области применения интеллектуальных proxy:

- Подсчет числа ссылок на реальный объект. При отсутствии ссылок память под объект автоматически освобождается (известен также как интеллектуальный указатель или smart pointer).
- Загрузка объекта в память при первом обращении к нему.
- Установка запрета на изменение реального объекта при обращении к нему других объектов.

Структура паттерна Proxy

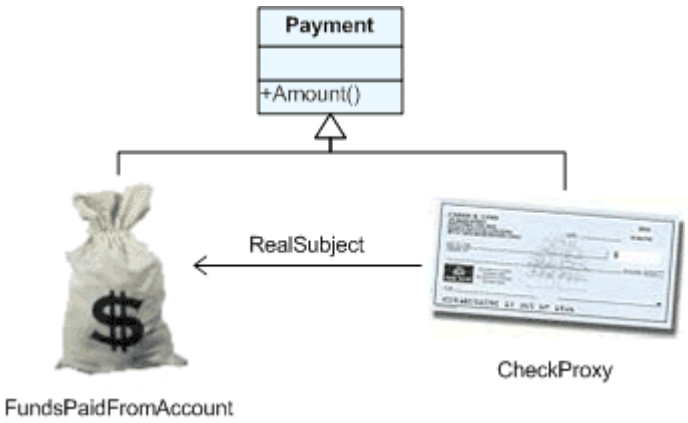
Заместитель Proxy и реальный объект RealSubject имеют одинаковые интерфейсы класса Subject, поэтому заместитель может использоваться "прозрачно" для клиента вместо реального объекта.

UML-диаграмма классов паттерна Proxy



Пример паттерна Proxy

Паттерн Proxy для доступа к реальному объекту использует его суррогат или заместитель. Банковский чек является заместителем денежных средств на счете. Чек может быть использован вместо наличных денег для совершения покупок и, в конечном счете, контролирует доступ к наличным деньгам на счете чекодателя.



Использование паттерна Proxy

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

Особенности паттерна Proxy

- Adapter предоставляет своему объекту другой интерфейс . Proxy предоставляет тот же интерфейс. Decorator предоставляет расширенный интерфейс.
- Decorator и Proxy имеют разные цели, но схожие структуры. Оба вводят дополнительный уровень косвенности: их реализации хранят ссылку на объект, на который они отправляют запросы.