# Паттерн Decorator (декоратор, wrapper, обертка)

## 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".

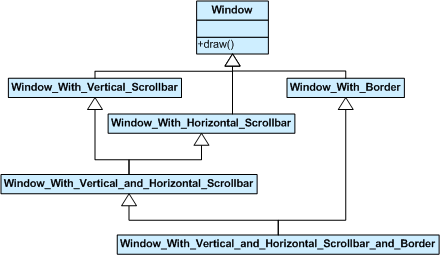
## Решаемая проблема

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

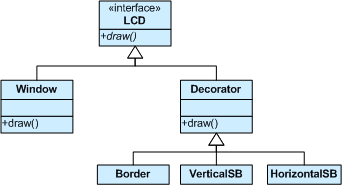


Эта схема имеет существенный недостаток - число классов сильно разрастается.

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

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

Гибкость может быть достигнута следующим дизайном.



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

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

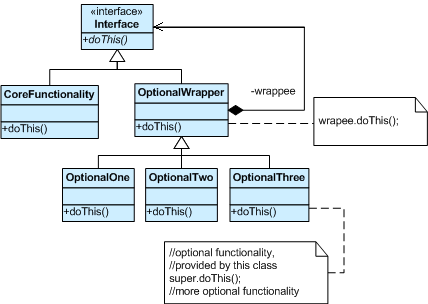
Решение этого класса задач предполагает инкапсуляцию исходного объекта в абстрактный интерфейс. Как объекты-декораторы, так и основной объект наследуют от этого абстрактного интерфейса. Интерфейс использует рекурсивную композицию для добавления к основному объекту неограниченного количества "слоев" -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

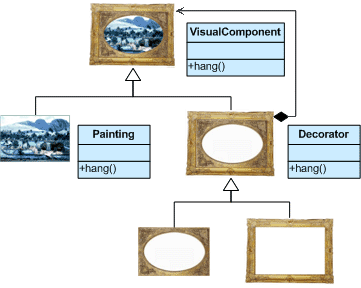
Клиент всегда заинтересован в функциональности CoreFunctionality.doThis(). Клиент может или не может быть заинтересован в методах OptionalOne.doThis() и OptionalTwo.doThis(). Каждый из этих классов переадресует запрос базовому классу Decorator, а тот направляет его в декорируемый объект.



## 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](http://cpp-reference.ru/patterns/structural-patterns/adapter/) придает своему объекту новый интерфейс, [Proxy](http://cpp-reference.ru/patterns/structural-patterns/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](http://cpp-reference.ru/patterns/structural-patterns/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](http://cpp-reference.ru/patterns/behavioral-patterns/strategy/) – его внутреннее содержание.

## Реализация паттерна Decorator

### Паттерн Decorator: до и после

#### До

Используется следующая иерархия наследования:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71 | class A {    public:      virtual void do\_it() {          cout << 'A';      }  };    class AwithX: public A {    public:      /\*virtual\*/      void do\_it() {          A::do\_it();          do\_X();      };    private:      void do\_X() {          cout << 'X';      }  };    class AwithY: public A {    public:      /\*virtual\*/      void do\_it() {          A::do\_it();          do\_Y();      }    protected:      void do\_Y() {          cout << 'Y';      }  };    class AwithZ: public A {    public:      /\*virtual\*/      void do\_it() {          A::do\_it();          do\_Z();      }    protected:      void do\_Z() {          cout << 'Z';      }  };    class AwithXY: public AwithX, public AwithY  {    public:      /\*virtual\*/      void do\_it() {          AwithX::do\_it();          AwithY::do\_Y();      }  };    class AwithXYZ: public AwithX, public AwithY, public AwithZ  {    public:      /\*virtual\*/      void do\_it() {          AwithX::do\_it();          AwithY::do\_Y();          AwithZ::do\_Z();      }  };    int main() {    AwithX anX;    AwithXY anXY;    AwithXYZ anXYZ;    anX.do\_it();    cout << '\n';    anXY.do\_it();    cout << '\n';    anXYZ.do\_it();    cout << '\n';  } |

Вывод программы:

|  |  |
| --- | --- |
| 1  2  3 | AX  AXY  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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78 | class I {    public:      virtual ~I(){}      virtual void do\_it() = 0;  };    class A: public I {    public:      ~A() {          cout << "A dtor" << '\n';      }      /\*virtual\*/      void do\_it() {          cout << 'A';      }  };    class D: public I {    public:      D(I \*inner) {          m\_wrappee = inner;      }      ~D() {          delete m\_wrappee;      }      /\*virtual\*/      void do\_it() {          m\_wrappee->do\_it();      }    private:      I \*m\_wrappee;  };    class X: public D {    public:      X(I \*core): D(core){}      ~X() {          cout << "X dtor" << "   ";      }      /\*virtual\*/      void do\_it() {          D::do\_it();          cout << 'X';      }  };    class Y: public D {    public:      Y(I \*core): D(core){}      ~Y() {          cout << "Y dtor" << "   ";      }      /\*virtual\*/      void do\_it() {          D::do\_it();          cout << 'Y';      }  };    class Z: public D {    public:      Z(I \*core): D(core){}      ~Z() {          cout << "Z dtor" << "   ";      }      /\*virtual\*/      void do\_it() {          D::do\_it();          cout << 'Z';      }  };    int main() {    I \*anX = new X(new A);    I \*anXY = new Y(new X(new A));    I \*anXYZ = new Z(new Y(new X(new A)));    anX->do\_it();    cout << '\n';    anXY->do\_it();    cout << '\n';    anXYZ->do\_it();    cout << '\n';    delete anX;    delete anXY;    delete anXYZ;  } |

Вывод программы:

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | AX  AXY  AXYZ  X dtor   A dtor  Y dtor   X dtor   A dtor  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  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82 | #include <iostream>  using namespace std;    // 1. " Наименьший общий знаменатель"  class Widget  {    public:      virtual void draw() = 0;  };    // 3. Основной класс, использующий отношение "является"  class TextField: public Widget  {      int width, height;    public:      TextField(int w, int h)      {          width = w;          height = h;      }        /\*virtual\*/      void draw()      {          cout << "TextField: " << width << ", " << height << '\n';      }  };    // 2. Базовый класс второго уровня  class Decorator: public Widget  // 3. использует отношение "является"  {      Widget \*wid; // 4. отношение "имеет"    public:      Decorator(Widget \*w)      {          wid = w;      }        /\*virtual\*/      void draw()      {          wid->draw(); // 5. делегирование      }  };    // 6. Дополнительное декорирование  class BorderDecorator: public Decorator  {    public:      BorderDecorator(Widget \*w): Decorator(w){}        /\*virtual\*/      void draw()      {          // 7. Делегирование базовому классу и          Decorator::draw();          // 7. реализация дополнительной функциональности          cout << "   BorderDecorator" << '\n';      }  };    // 6. Дополнительное декорирование  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 |

Источник: <http://sourcemaking.com/design_patterns/decorator/>

**http://cpp-reference.ru/patterns/structural-patterns/flyweight/**