Component для промежуточной формы представления программы SSA (Single Static Assignment). Листовые подклассы RegisterTransfer определяют различные статические присваивания, например:

- примитивные присваивания, которые выполняют операцию над двумя регистрами и сохраняют результат в третьем;
- присваивание, у которого есть исходный, но нет целевого регистра. Следовательно, регистр используется после возврата из процедуры;
- присваивание, у которого есть целевой, но нет исходного регистра. Это означает, что присваивание регистру происходит перед началом процедуры.

Подкласс RegisterTransferSet является примером класса Composite для представления присваиваний, изменяющих сразу несколько регистров.

Другой пример применения паттерна компоновщик – финансовые программы, когда инвестиционный портфель состоит их нескольких отдельных активов. Можно поддержать сложные агрегаты активов, если реализовать портфель в виде компоновщика, согласованного с интерфейсом каждого актива [BE93].

Паттерн команда описывает, как можно компоновать и упорядочивать объекты Command с помощью класса компоновщика MacroCommand.

Родственные паттерны

Отношение компонент-родитель используется в паттерне цепочка обязанностей.

Паттерн декоратор часто применяется совместно с компоновщиком. Когда декораторы и компоновщики используются вместе, у них обычно бывает общий родительский класс. Поэтому декораторам придется поддержать интерфейс компонентов такими операциями, как Add, Remove и GetChild.

Паттерн приспособленец позволяет разделять компоненты, но ссылаться на своих родителей они уже не могут.

Итератор можно использовать для обхода составных объектов.

Посетитель локализует операции и поведение, которые в противном случае пришлось бы распределять между классами Composite и Leaf.

Паттерн Decorator

Название и классификация паттерна

Декоратор – паттерн, структурирующий объекты.

Назначение

Динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Известен также под именем

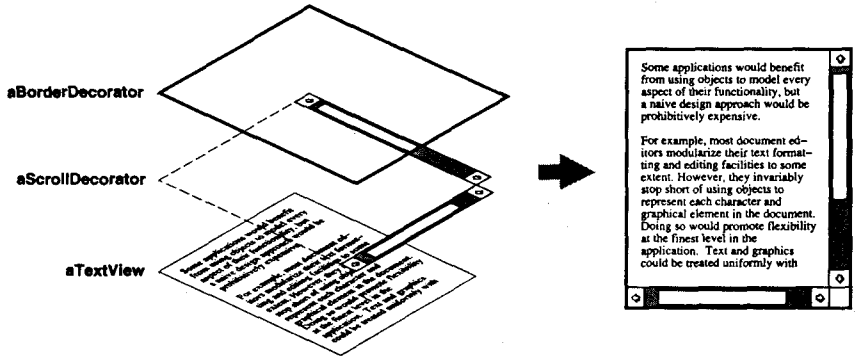
Wrapper (обертка).

Мотивация

Иногда бывает нужно возложить дополнительные обязанности на отдельный объект, а не на класс в целом. Так, библиотека для построения графических интерфейсов пользователя должна «уметь» добавлять новое свойство, скажем, рамку или новое поведение (например, возможность прокрутки к любому элементу интерфейса).

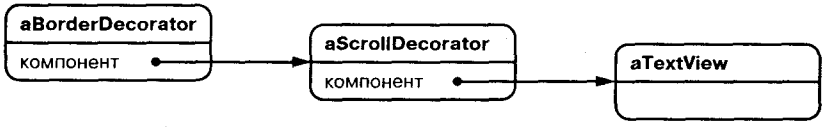
Добавить новые обязанности допустимо с помощью наследования. При наследовании классу с рамкой вокруг каждого экземпляра подкласса будет рисоваться рамка. Однако это решение статическое, а значит, недостаточно гибкое. Клиент не может управлять оформлением компонента рамкой.

Более гибким является другой подход: поместить компонент в другой объект, называемый *декоратором*, который как раз и добавляет рамку. Декоратор следует интерфейсу декорируемого объекта, поэтому его присутствие прозрачно для клиентов компонента. Декоратор переадресует запросы внутреннему компоненту, но может выполнять и дополнительные действия (например, рисовать рамку) до или после переадресации. Поскольку декораторы прозрачны, они могут вкладываться друг в друга, добавляя тем самым любое число новых обязанностей.



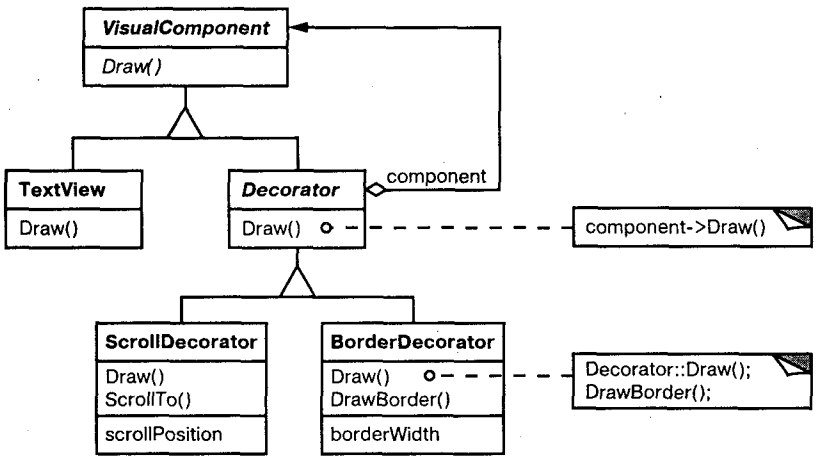
Предположим, что имеется объект класса `TextView`, который отображает текст в окне. По умолчанию `TextView` не имеет полос прокрутки, поскольку они не всегда нужны. Но при необходимости их удастся добавить с помощью декоратора `ScrollDecorator`. Допустим, что еще мы хотим добавить жирную сплошную рамку вокруг объекта `TextView`. Здесь может помочь декоратор `BorderDecorator`. Мы просто компонуем оба декоратора с `BorderDecorator` и получаем искомый результат.

Ниже на диаграмме показано, как композиция объекта `TextView` с объектами `BorderDecorator` и `ScrollDecorator` порождает элемент для ввода текста, окруженный рамкой и снабженный полосой прокрутки.



Классы `ScrollDecorator` и `BorderDecorator` являются подклассами `Decorator` – абстрактного класса, который представляет визуальные компоненты, применяемые для оформления других визуальных компонентов.

`VisualComponent` – это абстрактный класс для представления визуальных объектов. В нем определен интерфейс для рисования и обработки событий. Отметим, что класс `Decorator` просто переадресует запросы на рисование своему компоненту, а его подклассы могут расширять эту операцию.



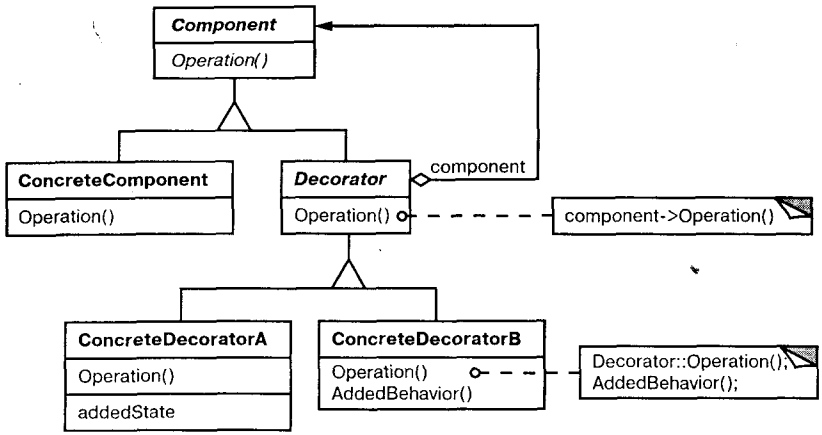
Подклассы `Decorator` могут добавлять любые операции для обеспечения необходимой функциональности. Так, операция `ScrollTo` объекта `ScrollDecorator` позволяет другим объектам выполнять прокрутку, если им известно о присутствии объекта `ScrollDecorator`. Важная особенность этого паттерна состоит в том, что декораторы могут употребляться везде, где возможно появление самого объекта `VisualComponent`. Поэтому клиент не может отличить декорированный объект от недекорированного, а значит, и никоим образом не зависит от наличия или отсутствия оформлений.

Применимость

Используйте паттерн декоратор:

- для динамического, прозрачного для клиентов добавления обязанностей объектам;
- для реализации обязанностей, которые могут быть сняты с объекта;
- когда расширение путем порождения подклассов по каким-то причинам неудобно или невозможно. Иногда приходится реализовывать много независимых расширений, так что порождение подклассов для поддержки всех возможных комбинаций приведет к комбинаторному росту их числа. В других случаях определение класса может быть скрыто или почему-либо еще недоступно, так что породить от него подкласс нельзя.

Структура



Участники

- **Component** (`VisualComponent`) – компонент:
  - определяет интерфейс для объектов, на которые могут быть динамически возложены дополнительные обязанности;
- **ConcreteComponent** (`TextView`) – конкретный компонент:
  - определяет объект, на который возлагаются дополнительные обязанности;
- **Decorator** – декоратор:
  - хранит ссылку на объект `Component` и определяет интерфейс, соответствующий интерфейсу `Component`;
- **ConcreteDecorator** (`BorderDecorator`, `ScrollDecorator`) – конкретный декоратор:
  - возлагает дополнительные обязанности на компонент.

Отношения

`Decorator` переадресует запросы объекту `Component`. Может выполнять и дополнительные операции до и после переадресации.

Результаты

У паттерна декоратор есть, по крайней мере, два плюса и два минуса:

- *большая гибкость, нежели у статического наследования.* Паттерн декоратор позволяет более гибко добавлять объекту новые обязанности, чем было бы возможно в случае статического (множественного) наследования. Декоратор может добавлять и удалять обязанности во время выполнения программы. При использовании же наследования требуется создавать новый класс для каждой дополнительной обязанности (например, `BorderedScrollableTextView`, `BorderedTextView`), что ведет к увеличению числа классов и, как следствие, к возрастанию сложности системы. Кроме того, применение нескольких

декораторов к одному компоненту позволяет произвольным образом сочетать обязанности.

Декораторы позволяют легко добавить одно и то же свойство дважды. Например, чтобы окружить объект `TextView` двойной рамкой, нужно просто добавить два декоратора `BorderDecorators`. Двойное наследование классу `Border` в лучшем случае чревато ошибками;

- *позволяет избежать перегруженных функциями классов на верхних уровнях иерархии.* Декоратор разрешает добавлять новые обязанности по мере необходимости. Вместо того чтобы пытаться поддержать все мыслимые возможности в одном сложном, допускающем разностороннюю настройку классе, вы можете определить простой класс и постепенно наращивать его функциональность с помощью декораторов. В результате приложение уже не платит за неиспользуемые функции. Нетрудно также определять новые виды декораторов независимо от классов, которые они расширяют, даже если первоначально такие расширения не планировались. При расширении же сложного класса обычно приходится вникать в детали, не имеющие отношения к добавляемой функции;
- *декоратор и его компонент не идентичны.* Декоратор действует как прозрачное обрамление. Но декорированный компонент все же не идентичен исходному. При использовании декораторов это следует иметь в виду;
- *множество мелких объектов.* При использовании в проекте паттерна декоратор нередко получается система, составленная из большого числа мелких объектов, которые похожи друг на друга и различаются только способом взаимосвязи, а не классом и не значениями своих внутренних переменных. Хотя проектировщик, разбирающийся в устройстве такой системы, может легко настроить ее, но изучать и отлаживать ее очень тяжело.

Реализация

Применение паттерна декоратор требует рассмотрения нескольких вопросов:

- *соответствие интерфейсов.* Интерфейс декоратора должен соответствовать интерфейсу декорируемого компонента. Поэтому классы `ConcreteDecorator` должны наследовать общему классу (по крайней мере, в C++);
- *отсутствие абстрактного класса Decorator.* Нет необходимости определять абстрактный класс `Decorator`, если планируется добавить всего одну обязанность. Так часто происходит, когда вы работаете с уже существующей иерархией классов, а не проектируете новую. В таком случае ответственность за переадресацию запросов, которую обычно несет класс `Decorator`, можно возложить непосредственно на `ConcreteDecorator`;
- *облегченные классы Component.* Чтобы можно было гарантировать соответствие интерфейсов, компоненты и декораторы должны наследовать общему классу `Component`. Важно, чтобы этот класс был настолько легким, насколько возможно. Иными словами, он должен определять интерфейс, а не хранить данные. В противном случае декораторы могут стать весьма тяжеловесными, и применять их в большом количестве будет накладно. Включение большого

числа функций в класс Component также увеличивает вероятность, что конкретным подклассам придется платить за то, что им не нужно;

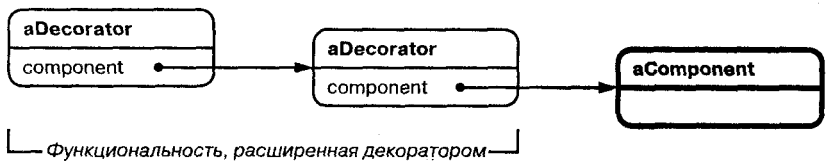
- *изменение облика, а не внутреннего устройства объекта.* Декоратор можно рассматривать как появившуюся у объекта оболочку, которая изменяет его поведение. Альтернатива – изменение внутреннего устройства объекта, хорошим примером чего может служить паттерн стратегия.

Стратегии лучше подходят в ситуациях, когда класс Component уже достаточно тяжел, так что применение паттерна декоратор обходится слишком дорого. В паттерне стратегия компоненты передают часть своей функциональности отдельному объекту-стратегии, поэтому изменить или расширить поведение компонента допустимо, заменив этот объект.

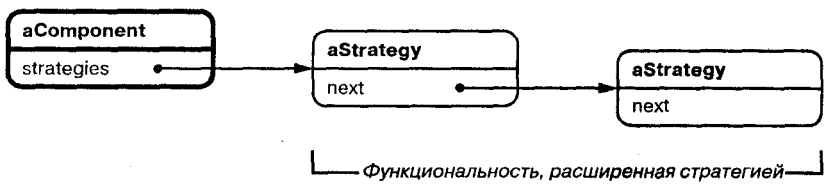
Например, мы можем поддержать разные стили рамок, поручив рисование рамки специальному объекту Border. Объект Border является примером объекта-стратегии: в данном случае он инкапсулирует стратегию рисования рамки. Число стратегий может быть любым, поэтому эффект такой же, как от рекурсивной вложенности декораторов.

Например, в системах MacApp 3.0 [App89] и Bedrock [Sym93a] графические компоненты, называемые видами (views), хранят список объектов-оформителей (adorner), которые могут добавлять различные оформления вроде границ к виду. Если к виду присоединены такие объекты, он дает им возможность выполнить свои функции. MacApp и Bedrock вынуждены предоставить доступ к этим операциям, поскольку класс View весьма тяжел. Было бы слишком расточительно использовать полномасштабный объект этого класса только для того, чтобы добавить рамку.

Поскольку паттерн декоратор изменяет лишь внешний облик компонента, последнему ничего не надо «знать» о своих декораторах, то есть декораторы прозрачны для компонента.



В случае стратегий самому компоненту известно о возможных расширениях. Поэтому он должен располагать информацией обо всех стратегиях и ссылаться на них.



При использовании подхода, основанного на стратегиях, может возникнуть необходимость модифицировать компонент, чтобы он соответствовал новому расширению. С другой стороны, у стратегии может быть свой собственный специализированный интерфейс, тогда как интерфейс декоратора должен повторять интерфейс компонента. Например, стратегии рисования рамки необходимо определить всего лишь интерфейс для этой операции (`DrawBorder`, `GetWidth` и т.д.), то есть класс стратегии может быть легким, несмотря на тяжеловесность компонента.

Системы MacApp и Bedrock применяют такой подход не только для оформления видов, но и для расширения особенностей поведения объектов, связанных с обработкой событий. В обеих системах вид ведет список объектов поведения, которые могут модифицировать и перехватывать события. Каждому зарегистрированному объекту поведения вид предоставляет возможность обработать событие до того, как оно будет передано незарегистрированным объектам такого рода, за счет чего достигается переопределение поведения. Можно, например, декорировать вид специальной поддержкой работы с клавиатурой, если зарегистрировать объект поведения, который перехватывает и обрабатывает события нажатия клавиш.

Пример кода

В следующем примере показано, как реализовать декораторы пользовательского интерфейса в программе на C++. Мы будем предполагать, что класс компонента называется `VisualComponent`:

```
class VisualComponent {
public:
    VisualComponent();

    virtual void Draw();
    virtual void Resize();
    // ...
};
```

Определим подкласс класса `VisualComponent` с именем `Decorator`, от которого затем породим подклассы, реализующие различные оформления:

```
class Decorator : public VisualComponent {
public:
    Decorator(VisualComponent*);

    virtual void Draw();
    virtual void Resize();
    // ...
private:
    VisualComponent* _component;
};
```

Объект класса `Decorator` декорирует объект `VisualComponent`, на который ссылается переменная экземпляра `_component`, инициализируемая в конструкторе.

Для каждой операции в интерфейсе `VisualComponent` в классе `Decorator` определена реализация по умолчанию, передающая запросы объекту, на который ведет ссылка `_component`:

```
void Decorator::Draw () {
    _component->Draw();
}

void Decorator::Resize () {
    _component->Resize();
}
```

Подклассы `Decorator` определяют специализированные операции. Например, класс `BorderDecorator` добавляет к своему внутреннему компоненту рамку. `BorderDecorator` – это подкласс `Decorator`, где операция `Draw` замещена так, что рисует рамку. В этом классе определена также закрытая вспомогательная операция `DrawBorder`, которая, собственно, и изображает рамку. Реализации всех остальных операций этот подкласс наследует от `Decorator`:

```
class BorderDecorator : public Decorator {
public:
    BorderDecorator(VisualComponent*, int borderWidth);

    virtual void Draw();
private:
    void DrawBorder(int);
private:
    int _width;
};

void BorderDecorator::Draw () {
    Decorator::Draw();
    DrawBorder(_width);
}
```

Подклассы `ScrollDecorator` и `DropShadowDecorator`, которые добавят визуальному компоненту возможность прокрутки и оттенения можно реализовать аналогично.

Теперь нам удастся скомпоновать экземпляры этих классов для получения различных оформлений. Ниже показано, как использовать декораторы для создания прокручиваемого компонента `TextView` с рамкой.

Во-первых, нужен какой-то способ поместить визуальный компонент в оконный объект. Предположим, что в нашем классе `Window` для этой цели имеется операция `SetContents`:

```
void Window::SetContents (VisualComponent* contents) {
    // ...
}
```

Теперь можно создать поле для ввода текста и окно, в котором будет находиться это поле:

```
Window* window = new Window;
TextView* textView = new TextView;
```

`TextView` – подкласс `VisualComponent`, значит, мы могли бы поместить его в окно:

```
window->SetContents(textView);
```

Но нам нужно поле ввода с рамкой и возможностью прокрутки. Поэтому предварительно мы его надлежащим образом оформим:

```
window->SetContents(
    new BorderDecorator(
        new ScrollDecorator(textView), 1
    )
);
```

Поскольку класс `Window` обращается к своему содержимому только через интерфейс `VisualComponent`, то ему неизвестно о присутствии декоратора. Клиент при желании может сохранить ссылку на само поле ввода, если ему нужно работать с ним непосредственно, например вызывать операции, не входящие в интерфейс `VisualComponent`. Клиенты, которым важна идентичность объекта, также должны обращаться к нему напрямую.

Известные применения

Во многих библиотеках для построения объектно-ориентированных интерфейсов пользователя декораторы применяются для добавления к виджетам графических оформлений. В качестве примеров можно назвать InterViews [LVC89, LCI+92], ET++ [WGM88] и библиотеку классов ObjectWorks\Smalltalk [Par90]. Другие варианты применения паттерна декоратор – это класс `DebuggingGlyph` из библиотеки InterViews и `PassivityWrapper` из ParcPlace Smalltalk. `DebuggingGlyph` печатает отладочную информацию до и после того, как переадресует запрос на размещение своему компоненту. Эта информация может быть полезна для анализа и отладки стратегии размещения объектов в сложном контейнере. Класс `PassivityWrapper` позволяет разрешить или запретить взаимодействие компонента с пользователем.

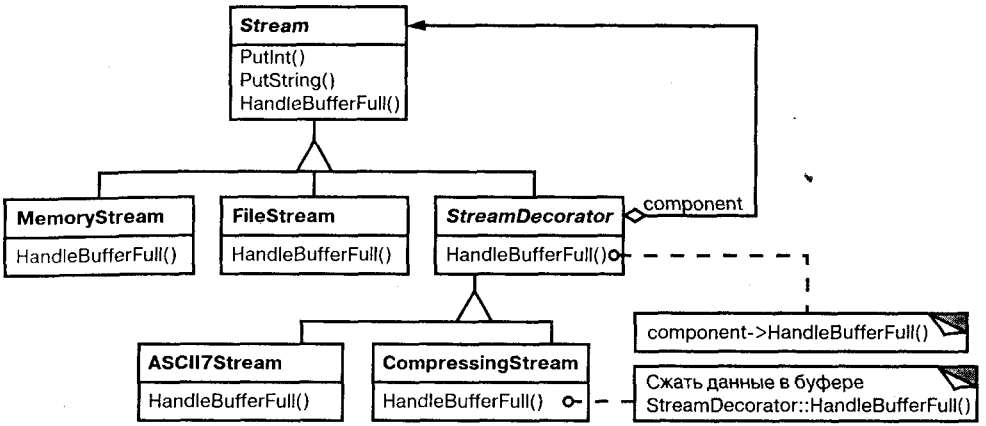
Но применение паттерна декоратор никоим образом не ограничивается графическими интерфейсами пользователя, как показывает следующий пример, основанный на потоковых классах из каркаса ET++ [WGM88].

Поток – это фундаментальная абстракция в большинстве средств ввода/вывода. Он может предоставлять интерфейс для преобразования объектов в последовательность байтов или символов. Это позволяет записать объект в файл или буфер в памяти и впоследствии извлечь его оттуда. Самый очевидный способ сделать это – определить абстрактный класс `Stream` с подклассами `MemoryStream` и `FileStream`. Предположим, однако, что нам хотелось бы еще уметь:

- компрессировать данные в потоке, применяя различные алгоритмы сжатия (кодирование повторяющихся серий, алгоритм Лемпеля-Зива и т.д.);

- преобразовывать данные в 7-битные символы кода ASCII для передачи по каналу связи.

Паттерн декоратор позволяет весьма элегантно добавить такие обязанности потокам. На диаграмме ниже показано одно из возможных решений задачи.



Абстрактный класс `Stream` имеет внутренний буфер и предоставляет операции для помещения данных в поток (`PutInt`, `PutString`). Как только буфер заполняется, `Stream` вызывает абстрактную операцию `HandleBufferFull`, которая выполняет реальное перемещение данных. В классе `FileStream` эта операция замещается так, что буфер записывается в файл.

Ключевым здесь является класс `StreamDecorator`. Именно в нем хранится ссылка на тот поток-компонент, которому переадресуются все запросы. Подклассы `StreamDecorator` замещают операцию `HandleBufferFull` и выполняют дополнительные действия, перед тем как вызвать реализацию этой операции в классе `StreamDecorator`.

Например, подкласс `CompressingStream` сжимает данные, а `ASCII7Stream` конвертирует их в 7-битный код ASCII. Теперь, для того чтобы создать объект `FileStream`, который *одновременно* сжимает данные и преобразует результат в 7-битный код, достаточно просто декорировать `FileStream` с использованием `CompressingStream` и `ASCII7Stream`:

```
Stream* aStream = new CompressingStream(
    new ASCII7Stream(
        new FileStream("aFileName")
    )
);
aStream->PutInt(12);
aStream->PutString("aString");
```

Родственные паттерны

Адаптер: если декоратор изменяет только обязанности объекта, но не его интерфейс, то адаптер придает объекту совершенно новый интерфейс.

Компоновщик: декоратор можно считать вырожденным случаем составного объекта, у которого есть только один компонент. Однако декоратор добавляет новые обязанности, агрегирование объектов не является его целью.

Стратегия: декоратор позволяет изменить внешний облик объекта, стратегия – его внутреннее содержание. Это два взаимодополняющих способа изменения объекта.

Паттерн Facade

Название и классификация паттерна

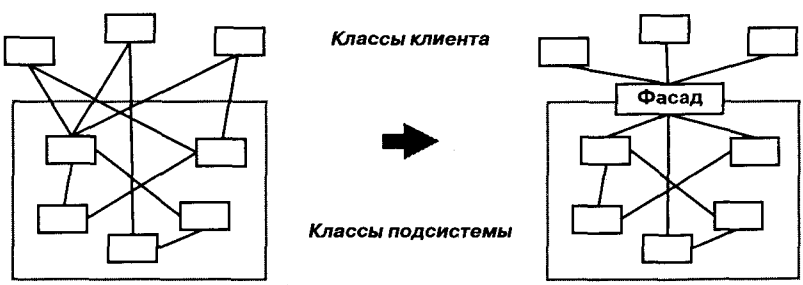
Фасад – паттерн, структурирующий объекты.

Назначение

Предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Мотивация

Разбиение на подсистемы облегчает проектирование сложной системы в целом. Общая цель всякого проектирования – свести к минимуму зависимость подсистем друг от друга и обмен информацией между ними. Один из способов решения этой задачи – введение объекта фасад, предоставляющий единый упрощенный интерфейс к более сложным системным средствам.



Рассмотрим, например, среду программирования, которая дает приложениям доступ к подсистеме компиляции. В этой подсистеме имеются такие классы, как `Scanner` (лексический анализатор), `Parser` (синтаксический анализатор), `ProgramNode` (узел программы), `BytecodeStream` (поток байтовых кодов) и `ProgramNodeBuilder` (строитель узла программы). Все вместе они составляют компилятор. Некоторым специализированным приложениям, возможно, понадобится прямой доступ к этим классам. Но для большинства клиентов компилятора такие детали, как синтаксический разбор и генерация кода, обычно не нужны; им просто требуется откомпилировать некоторую программу. Для таких клиентов применение мощного, но низкоуровневого интерфейса подсистемы компиляции только усложняет задачу.