Мотивация

Такие приложения, как графические редакторы и редакторы электрических

схем, позволяют пользователям строить сложные диаграммы из более простых

компонентов. Проектировщик может сгруппировать мелкие компоненты для фор-

мирования более крупных, которые, в свою очередь, могут стать основой для со-

здания еще более крупных. В простой реализации допустимо было бы определить

классы графических примитивов, например текста и линий, а также классы, вы-

ступающие в роли контейнеров для этих примитивов.

Но у такого решения есть существенный недостаток. Программа, в которой

эти классы используются, должна по-разному обращаться с примитивами и кон-

тейнерами, хотя пользователь чаще всего работает с ними единообразно. Необхо-

димость различать эти объекты усложняет приложение. Паттерн компоновщик

описывает, как можно применить рекурсивную композицию таким образом, что

клиенту не придется проводить различие между простыми и составными объек-

тами.

Ключом к паттерну компоновщик является абстрактный класс, который

представляет одновременно и примитивы, и контейнеры. В графической системе

этот класс может называться Graphic. В нем объявлены операции, специфичные

для каждого вида графического объекта (такие как Draw) и общие для всех со-

ставных объектов, например операции для доступа и управления потомками.

Подклассы Line, Rectangle и Text (см. диаграмму выше) определяют при-

митивные графические объекты. В них операция Draw реализована соответствен-

но для рисования прямых, прямоугольников и текста. Поскольку у примитивных

объектов нет потомков, то ни один из этих подклассов не реализует операции,

относящиеся к управлению потомками.

Класс Picture определяет агрегат, состоящий из объектов Graphic. Реали-

зованная в нем операция Draw вызывает одноименную функцию для каждого

потомка, а операции для работы с потомками уже не пусты. Поскольку интерфейс

класса Picture соответствует интерфейсу Graphic, то в состав объекта Picture

могут входить и другие такие же объекты.

Участники

Э Component (Graphic) - компонент:

- объявляет интерфейс для компонуемых объектов;

- предоставляет подходящую реализацию операций по умолчанию, общую

для всех классов;

- объявляет интерфейс для доступа к потомкам и управления ими;

- определяет интерфейс для доступа к родителю компонента в рекурсив-

ной структуре и при необходимости реализует его. Описанная возмож-

ность необязательна;

a Leuf (Rectangle, Line, Text, и т.п.) - лист:

- представляет листовые узлы композиции и не имеет потомков;

- определяет поведение примитивных объектов в композиции;

a Composite (Picture) - составной объект:

- определяет поведение компонентов, у которых есть потомки;

- хранит компоненты-потомки;

- реализует относящиеся к управлению потомками операции в интерфей-

се клдсса Component;

a Client - клиент:

- манипулирует объектами композиции через интерфейс Component.

Отношения

Клиенты используют интерфейс класса Component для взаимодействия с объек-

тами в составной структуре. Если получателем запроса является листовый объект

Leaf, то он и обрабатывает запрос. Когда же получателем является составной объект

Composite, то обычно он перенаправляет запрос своим потомкам, возможно, вы-

полняя некоторые дополнительные операции до или после перенаправления.

Результаты

Паттерн компоновщик:

а определяет иерархии классов, состоящие из примитивных и составных объ-

ектов. Из примитивных объектов можно составлять более сложные, кото-

рые, в свою очередь, участвуют в более сложных композициях и так далее.

Любой клиент, ожидающий примитивного объекта, может работать и с со-

ставным;

а упрощает архитектуру клиента. Клиенты могут единообразно работать

с индивидуальными и объектами и с составными структурами. Обычно

клиенту неизвестно, взаимодействует ли он с листовым или составным

объектом. Это упрощает код клиента, поскольку нет необходимости пи-

сать функции, ветвящиеся в зависимости от того, с объектом какого клас-

са они работают;

а облегчает добавление новых видов компонентов. Новые подклассы классов

Composite или Leaf будут автоматически работать с уже существующи-

ми структурами и клиентским кодом. Изменять клиента при добавлении но-

вых компонентов не нужно;\* Структурные паттерны

а способствует созданию общего дизайна. Однако такая простота добавле-

ния новых компонентов имеет и свои отрицательные стороны: становится

трудно наложить ограничения на то, какие объекты могут входить в со-

став композиции. Иногда желательно, чтобы составной объект мог вклю-

чать только определенные виды компонентов. Паттерн компоновщик не

позволяет воспользоваться для реализации таких ограничений статичес-

кой системой типов. Вместо этого следует проводить проверки во время

выполнения.

Реализация

При реализации паттерна компоновщик приходится рассматривать много

вопросов:

а явные ссылки на родителей. Хранение в компоненте ссылки на своего роди-

теля может упростить обход структуры и управление ею. Наличие такой

ссылки облегчает передвижение вверх по структуре и удаление компонен-

та. Кроме того, ссылки на родителей помогают поддержать паттерн цепоч-

ка обязанностей.

Обычно ссылку на родителя определяют в классе Component. Классы Leaf

и Composite могут унаследовать саму ссылку и операции с ней.

При наличии ссылки на родителя важно поддерживать следующий инвари-

ант: если некоторый объект в составной структуре ссылается на другой со-

ставной объект как на своего родителя, то для последнего первый является

потомком. Простейший способ гарантировать соблюдение этого условия -

изменять родителя компонента только тогда, когда он добавляется или уда-

ляется из составного объекта. Если это удается один раз реализовать в опе-

рациях Add и Remove, то реализация будет унаследована всеми подкласса-

ми и, значит, инвариант будет поддерживаться автоматически;

а разделение компонентов. Часто бывает полезно разделять компоненты, на-

пример для уменьшения объема занимаемой памяти. Но если у компонента

может быть более одного родителя, то разделение становится проблемой.

Возможное решение - позволить компонентам хранить ссылки на несколь-

ких родителей. Однако в таком случае при распространении запроса по

структуре могут возникнуть неоднозначности. Паттерн приспособленец

показывает, как следует изменить дизайн, чтобы вовсе отказаться от хране-

ния родителей. Работает он в тех случаях, когда потомки могут не посылать

сообщений своим родителям, вынеся за свои границы часть внутреннего со-

стояния;

а максимизация интерфейса класса Component. Одна из целей паттерна ком-

поновщик - избавить клиентов от необходимости знать, работают ли они

с листовым или составным объектом. Для достижения этой цели класс

Component должен сделать как можно больше операций общими для клас-

сов Composite и Leaf. Обычно класс Component предоставляет для этих

операций реализации по умолчанию, а подклассы Composite и Leaf заме-

щают их.Паттерн Composite

Однако иногда эта цель вступает в конфликт с принципом проектирования

иерархии классов, согласно которому класс должен определять только ло-

гичные для всех его подклассах операции. Класс Component поддерживает

много операций, не имеющих смысла для класса Leaf. Как же тогда предоста-

вить для них реализацию по умолчанию?

Иногда, проявив изобретательность, удается перенести в класс Component

операцию, которая, на первый взгляд, имеет смысл только для составных

объектов. Например, интерфейс для доступа к потомкам является фунда-

ментальной частью класса Composite, но вовсе не обязательно класса Leaf.

Однако если рассматривать Leaf как Component, у которого никогда не бы-

вает потомков, то мы можем определить в классе Component операцию до-

ступа к потомкам как никогда не возвращающую потомков. Тогда подклас-

сы Leaf могут использовать эту реализацию по умолчанию, а в подклассах

Composite она будет переопределена, чтобы возвращать потомков.

Операции для управления потомками довольно хлопотны, мы обсудим их

в следующем разделе;

а объявление операций для управления потомками. Хотя в классе Composite

реализованы операции Add и Remove для добавления и удаления потомков,

но для паттерна компоновщик важно, в каких классах эти операции объяв-

лены. Надо ли объявлять их в классе Component и тем самым делать до-

ступными в Leaf, или их следует объявить и определить только в классе

Composite и его подклассах?

Решая этот вопрос, мы должны выбирать между безопасностью и прозрач-

ностью:

- если определить интерфейс для управления потомками в корне иерархии

классов, то мы добиваемся прозрачности, так как все компоненты удает-

ся трактовать единообразно. Однако расплачиваться приходится безопас-

ностью, поскольку клиент может попытаться выполнить бессмысленное

действие, например добавить или удалить объект из листового узла;

- если управление потомками сделать частью класса Composite, то безо-

пасность удастся обеспечить, ведь любая попытка добавить или удалить

объекты из листьев в статически типизированном языке вроде C++ бу-

дет перехвачена на этапе компиляции. Но прозрачность мы утрачиваем,

ибо у листовых и составных объектов оказываются разные интерфейсы.

В паттерне компоновщик мы придаем особое значение прозрачности, а не

безопасности. Если для вас важнее безопасность, будьте готовы к тому, что

иногда вы можете потерять информацию о типе и придется преобразовы-

вать компонент к типу составного объекта. Как это сделать, не прибегая

к небезопасным приведениям типов?

Можно, например, объявить в классе Component операцию Composite\*

GetComposite ( ) . Класс Component реализует ее по умолчанию, возвра-

щая нулевой указатель. А в классе Composite эта операция переопределе-

на и возвращает указатель this на сам объект:Структурные паттерны

Аналогичные проверки на принадлежность классу Composite в C++ вы-

полняют и с помощью оператора dynamic\_cast.

Разумеется, при таком подходе мы не обращаемся со всеми компонентами

единообразно, что плохо. Снова приходится проверять тип, перед тем как

предпринять то или иное действие.

Единственный способ обеспечить прозрачность - это включить в класс

Component реализации операций Add и Remove по умолчанию. Но появит-

ся новая проблема: нельзя реализовать Component : : Add так, чтобы она

никогда не приводила к ошибке. Можно, конечно, сделать данную опера-

цию пустой, но тогда нарушается важное проектное ограничение; попытка

class Composite;

class Component {

public:

//...

virtual Composite\* GetComposite() { return 0; }

};

class Composite : public Component {

public:

void Add(Component\*);

// ...

virtual Composite\* GetComposite(} { return this; }

};

class Leaf : public Component {

// ...

};

Благодаря операции Get Composite можно спросить у компонента, явля-

ется ли он составным. К возвращаемому этой операцией составному объек-

ту допустимо безопасно применять операции Add и Remove:

Composite\* aComposite = new Composite;

Leaf\* aLeaf = new Leaf;

Component \* aComponent;

Composite\* test;

aComponent = aComposite;

if (test = aComponent->GetComposite()) {

test->Add(new Leaf);

}

aComponent = aLeaf;

if (test = aComponent->GetComposite()) {

test->Add(new Leaf); // не добавит лист

}Паттерн Composite

добавить что-то в листовый объект, скорее всего, свидетельствует об ошиб-

ке. Допустимо было бы заставить ее удалять свой аргумент, но клиент мо-

жет быть не рассчитанным на это.

Обычно лучшим решением является такая реализация Add и Remove по

умолчанию, при которой они завершаются с ошибкой (возможно, возбуж-

дая исключение), если компоненту не разрешено иметь потомков (для Add)

или аргумент не является чьим-либо потомком (для Remove).

Другая возможность - слегка изменить семантику операции «удаления».

Если компонент хранит ссылку на родителя, то можно было бы считать, что

Component: : Remove удаляет самого себя. Но для операции Add по-преж-

нему нет разумной интерпретации;

а должен ли Component реализовывать список компонентов. Может возник-

нуть желание определить множество потомков в виде переменной экземп-

ляра класса Component, в котором объявлены операции доступа и управле-

ния потомками. Но размещение указателя на потомков в базовом классе

приводит к непроизводительному расходу памяти во всех листовых узлах,

хотя у листа потомков быть не может. Такой прием можно применить, толь-

ко если в структуре не слишком много потомков;

а упорядочение потомков. Во многих случаях порядок следования потомков

составного объекта важен. В рассмотренном выше примере класса Graphic

под порядком может пониматься Z-порядок расположения потомков. В со-

ставных объектах, описывающих деревья синтаксического разбора, состав-

ные операторы могут быть экземплярами класса Composite, порядок сле-

дования потомков которых отражает семантику программы.

Если порядок следования потомков важен, необходимо учитывать его при

проектировании интерфейсов доступа и управления потомками. В этом

может помочь паттерн итератор;

а кэширование для повышения производительности. Если приходится часто

обходить композицию или производить в ней поиск, то класс Composite мо-

жет кэшировать информацию об обходе и поиске. Кэшировать разрешает-

ся либо полученные результаты, либо только информацию, достаточную

для ускорения обхода или поиска. Например, класс Picture из примера,

приведенного в разделе «Мотивация», мог бы кэшировать охватывающие

прямоугольники своих потомков. При рисовании или выборе эта инфор-

мация позволила бы пропускать тех потомков, которые не видимы в теку-

щем окне.

Любое изменение компонента должно делать кэши всех его родителей не-

действительными. Наиболее эффективен такой подход в случае, когда ком-

понентам известно об их родителях. Поэтому, если вы решите воспользо-

ваться кэшированием, необходимо определить интерфейс, позволяющий

уведомить составные объекты о недействительности их кэшей;

а кто должен удалять компоненты. В языках, где нет сборщика мусора, луч-

ше всего поручить классу Composite удалять своих потомков в момент

уничтожения. Исключением из этого правила является случай, когда лис-

товые объекты постоянны и; следовательно, могут разделяться;\_ •» \_ Структурные паттерны

а какая структура данных лучше всего подходит для хранения компонентов.

Составные объекты могут хранить своих потомков в самых разных структу-

рах данных, включая связанные списки, деревья, массивы и хэш-таблицы.

Выбор структуры данных определяется, как всегда, эффективностью. Соб-

ственно говоря, вовсе не обязательно пользоваться какой-либо из универ-

сальных структур. Иногда в составных объектах каждый потомок представ-

ляется отдельной переменной. Правда, для этого каждый подкласс Composite

должен реализовывать свой собственный интерфейс управления памятью.

См. пример в описании паттерна интерпретатор.

Пример кода

Такие изделия, как компьютеры и стереокомпоненты, часто имеют иерархичес-

кую структуру. Например, в раме монтируются дисковые накопители и плоские

электронные платы, к шине подсоединяются различные карты, а корпус содержит

раму, шины и т.д. Подобные структуры моделируются с помощью паттерна ком-

поновщик.

Класс Equipment определяет интерфейс для всех видов аппаратуры в иерар-

хии вида часть-целое:

class Equipment {

public:

virtual -Equipment ( ) ;

const char\* NameO { return \_name; }

virtual Watt Power ();

virtual Currency NetPrice();

virtual Currency DiscountPrice ( ) ;

virtual void Add ( Equipment \*);

virtual void Remove (Equipment\*) ;

virtual Iterator<Equipment\*>\* Createlterator ();

protected:

Equipment (const char\*);

private:

const char\* \_name;

};

В классе Equipment объявлены операции, которые возвращают атрибуты ап-

паратного блока, например энергопотребление и стоимость. Подклассы реализу-

ют эти операции для конкретных видов оборудования. Класс Equipment объяв-

ляет также операцию Createlterator, возвращающую итератор Iterator (см.

приложение С) для доступа к отдельным частям. Реализация этой операции по

умолчанию возвращает итератор Null Iterator, умеющий обходить только пус-

тое множество.

Среди подклассов Equipment могут быть листовые классы, представляющие

дисковые накопители, СБИС и переключатели:Паттерн Composite

class FloppyDisk : public Equipment {

public:

FloppyDisk(const char\*);

virtual -FloppyDisk();

virtual Watt Power();

virtual Currency NetPriceO;

virtual Currency DiscountPrice();

};

CompositeEquipment - это базовый класс для оборудования, содержащего

другое оборудование. Одновременно это подкласс класса Equipment:

class CompositeEquipment : public Equipment {

public:

virtual -CompositeEquipment();

virtual Watt Power();

virtual Currency NetPriceO;

virtual Currency DiscountPrice();

virtual void Add(Equipment\*);

virtual void Remove(Equipment\*);

virtual Iterator<Equipment\*>\* Createlterator();

protected:

CompositeEquipment(const char\*);

private:

List<Equipment\*> „equipment;

};

CompositeEquipment определяет операции для доступа и управления внут-

ренними аппаратными блоками. Операции Add и Remove добавляют и удаляют обо-

рудование из списка, хранящегося в переменной-члене \_equipment. Операция

Createlterator возвращает итератор (точнее, экземпляр класса List Iterator),

который будет обходить этот список.

Подразумеваемая реализация операции Net Price могла бы использовать

Createlterator для суммирования цен на отдельные блоки:

1

Currency CompositeEquipment::NetPrice () {

Iterator<Equipment\*>\* i = Createlterator();

Currency total = 0;

for (i->First(); !i->IsDone(); i->Next()) {

total += i->CurrentItem()->NetPrice();

}

delete i;

return total;

}

1

Очень легко забыть об удалении итератора после завершения работы с ним. При обсуждении паттер-

на итератор рассказано, как защититься от таких ошибок.\_^ ч Структурные паттерны

Теперь мы можем представить аппаратный блок компьютера в виде подкласса

к CompositeEquipment под названием Chassis. Chassis наследует порожден-

ные операции класса CompositeEquipment.

class Chassis : public CompositeEquipment {

public:

Chassis(const char\*);

virtual -Chassis();

virtual Watt Power();

virtual Currency NetPriceO;

virtual Currency DiscountPrice();

};

Мы можем аналогично определить и другие контейнеры для оборудования,

например Cabinet (корпус) и Bus (шина). Этого вполне достаточно для сборки

из отдельных блоков довольно простого персонального компьютера:

Cabinet\* cabinet = new Cabinet("PC Cabinet");

Chassis\* chassis = new Chassis("PC Chassis");

cabinet->Add(chassis);

Bus\* bus = new Bus("MCA B u s " );

bus->Add(new Card("16Mbs Token R i n g " ) );

chassis->Add(bus);

chassis->Add(new FloppyDisk("3.Sin Floppy"));

cout « "Полная стоимость равна " « chassis->NetPrice() « endl;