в этих примерах сделан на вопросах управления памятью, например разделении представления строк и поддержке объектов переменного размера. Нас же в первую очередь интересует поддержка независимых расширений абстракции и ее реализации.

В библиотеке libg++ [Lea88] определены классы, которые реализуют универсальные структуры данных: `Set` (множество), `LinkedSet` (множество как связанный список), `HashSet` (множество как хэш-таблица), `LinkedList` (связанный список) и `HashTable` (хэш-таблица). `Set` – это абстрактный класс, определяющий абстракцию множества, а `LinkedList` и `HashTable` – конкретные реализации связанного списка и хэш-таблицы. `LinkedSet` и `HashSet` – реализаторы абстракции `Set`, перекидывающие мост между `Set` и `LinkedList` и `HashTable` соответственно. Перед вами пример вырожденного моста, поскольку абстрактного класса `Implementor` здесь нет.

В библиотеке NeXT AppKit [Add94] паттерн мост используется при реализации и отображении графических изображений. Рисунок может быть представлен по-разному. Оптимальный способ его отображения на экране зависит от свойств дисплея и прежде всего от числа цветов и разрешения. Если бы не AppKit, то для каждого приложения разработчикам пришлось бы самостоятельно выяснять, какой реализацией пользоваться в конкретных условиях.

AppKit предоставляет мост `NXImage/NXImageRep`. Класс `NXImage` определяет интерфейс для обработки изображений. Реализация же определена в отдельной иерархии классов `NXImageRep`, в которой есть такие подклассы, как `NXEPSImageRep`, `NXCachedImageRep` и `NXBitMapImageRep`. В классе `NXImage` хранятся ссылки на один или более объектов `NXImageRep`. Если имеется более одной реализации изображения, то `NXImage` выбирает самую подходящую для данного дисплея. При необходимости `NXImage` также может преобразовать изображение из одного формата в другой. Интересная особенность этого варианта моста в том, что `NXImage` может одновременно хранить несколько реализаций `NXImageRep`.

Родственные паттерны

Паттерн абстрактная фабрика может создать и сконфигурировать мост.

Для обеспечения совместной работы не связанных между собой классов прежде всего предназначен паттерн адаптер. Обычно он применяется в уже готовых системах. Мост же участвует в проекте с самого начала и призван поддержать возможность независимого изменения абстракций и их реализаций.

Паттерн Composite

Название и классификация паттерна

Компоновщик – паттерн, структурирующий объекты.

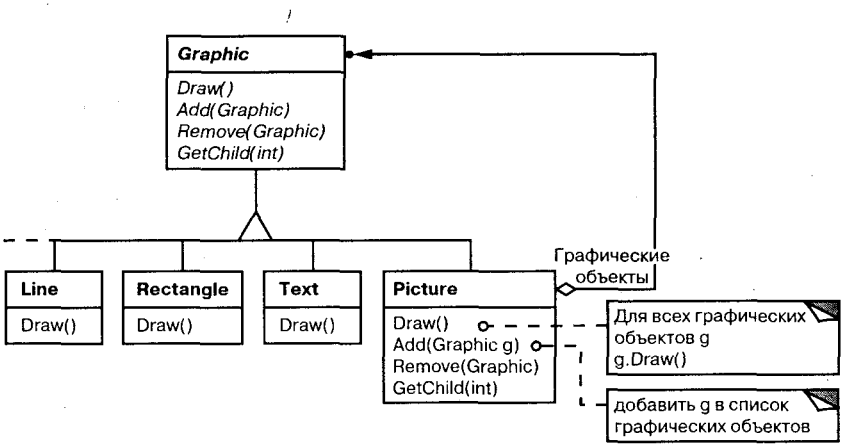
Назначение

Компонует объекты в древовидные структуры для представления иерархий часть-целое. Позволяет клиентам единообразно трактовать индивидуальные и составные объекты.

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 формирования более крупных, которые, в свою очередь, могут стать основой для создания еще более крупных. В простой реализации допустимо было бы определить классы графических примитивов, например текста и линий, а также классы, выступающие 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 контейнерами, хотя пользователь чаще всего работает с ними единообразно. Необходимость 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 объектами.

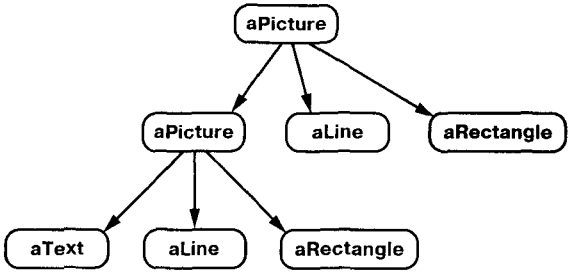


Ключом к паттерну КОМПОНОВЩИК является абстрактный класс, который представляет *одновременно* и примитивы, и контейнеры. В графической системе этот класс может называться `Graphic`. В нем объявлены операции, специфичные для каждого вида графического объекта (такие как `Draw`) и общие для всех составных объектов, например операции для доступа и управления потомками.

Подклассы `Line`, `Rectangle` и `Text` (см. диаграмму выше) определяют примитивные графические объекты. В них операция `Draw` реализована соответственно 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`. Реализованная 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.

Ниже на диаграмме показана типичная структура составного объекта, рекурсивно скомпонованного из объектов класса `Graphic`.

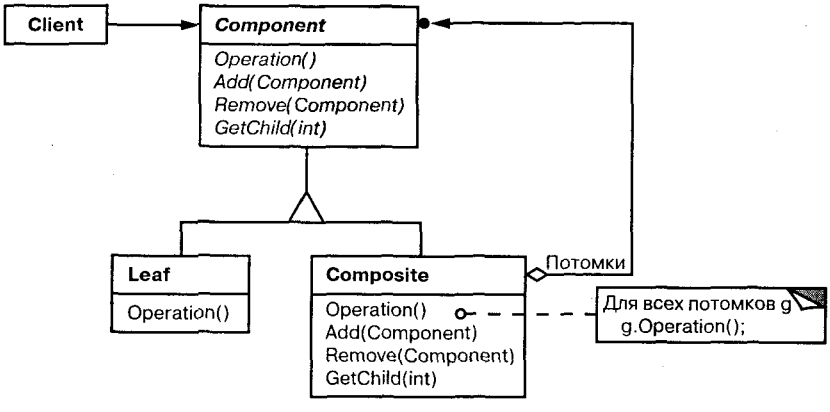


Применимость

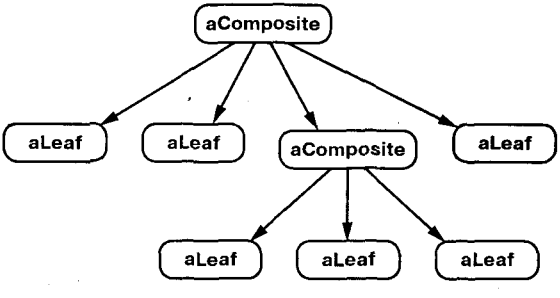
Используйте паттерн КОМПОНОВЩИК, когда:

- нужно представить иерархию объектов вида часть-целое;
- хотите, чтобы клиенты единообразно трактовали составные и индивидуальные объекты.

Структура



Структура типичного составного объекта могла бы выглядеть так:



Участники

- **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 рекурсивной структуре и при необходимости реализует его. Описанная возможность необязательна;
- **Leaf** (`Rectangle`, `Line`, `Text`, и т.п.) – лист:
  - представляет листовые узлы композиции и не имеет потомков;
  - определяет поведение примитивных объектов в композиции;
- **Composite** (`Picture`) – составной объект:
  - определяет поведение компонентов, у которых есть потомки;
  - хранит компоненты-потомки;
  - реализует относящиеся к управлению потомками операции в интерфейсе класса `Component`;
- **Client** – клиент:
  - манипулирует объектами композиции через интерфейс `Component`.

Отношения

Клиенты используют интерфейс класса `Component` для взаимодействия с объектами 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, выполняя некоторые дополнительные операции до или после перенаправления.

Результаты

Паттерн компоновщик:

- *определяет иерархии классов, состоящие из примитивных и составных объектов.* Из примитивных объектов можно составлять более сложные, которые, в свою очередь, участвуют в более сложных композициях и так далее. Любой клиент, ожидающий примитивного объекта, может работать и с составным;
- *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 писать функции, ветвящиеся в зависимости от того, с объектом какого класса они работают;
- *облегчает добавление новых видов компонентов.* Новые подклассы классов `Composite` или `Leaf` будут автоматически работать с уже существующими структурами и клиентским кодом. Изменять клиента при добавлении новых компонентов не нужно;

- *способствует созданию общего дизайна.* Однако такая простота добавления новых компонентов имеет и свои отрицательные стороны: становится трудно наложить ограничения на то, какие объекты могут входить в состав композиции. Иногда желательно, чтобы составной объект мог включать только определенные виды компонентов. Паттерн компоновщик не позволяет воспользоваться для реализации таких ограничений статической системой типов. Вместо этого следует проводить проверки во время выполнения.

Реализация

При реализации паттерна компоновщик приходится рассматривать много вопросов:

- *явные ссылки на родителей.* Хранение в компоненте ссылки на своего родителя может упростить обход структуры и управление ею. Наличие такой ссылки облегчает передвижение вверх по структуре и удаление компонента. Кроме того, ссылки на родителей помогают поддержать паттерн цепочка 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 инвариант: если некоторый объект в составной структуре ссылается на другой составной 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 удаляется из составного объекта. Если это удается один раз реализовать в операциях `Add` и `Remove`, то реализация будет унаследована всеми подклассами и, значит, инвариант будет поддерживаться автоматически;
- *разделение компонентов.* Часто бывает полезно разделять компоненты, например 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 нескольких родителей. Однако в таком случае при распространении запроса по структуре могут возникнуть неоднозначности. Паттерн приспособленец показывает, как следует изменить дизайн, чтобы вовсе отказаться от хранения родителей. Работает он в тех случаях, когда потомки могут не посылать сообщений своим родителям, вынеся за свои границы часть внутреннего состояния;
- *максимизация интерфейса класса Component.* Одна из целей паттерна компоновщик – 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 классов `Composite` и `Leaf`. Обычно класс `Component` предоставляет для этих операций реализации по умолчанию, а подклассы `Composite` и `Leaf` замещают их.

Однако иногда эта цель вступает в конфликт с принципом проектирования иерархии классов, согласно которому класс должен определять только логичные для всех его подклассах операции. Класс `Component` поддерживает много операций, не имеющих смысла для класса `Leaf`. Как же тогда предоставить 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 фундаментальной частью класса `Composite`, но вовсе не обязательно класса `Leaf`. Однако если рассматривать `Leaf` как `Component`, у которого *никогда* не бывает потомков, то мы можем определить в классе `Component` операцию доступа к потомкам как никогда не возвращающую потомков. Тогда подклассы `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;

- *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 *объявлены*. Надо ли объявлять их в классе `Component` и тем самым делать доступными в `Leaf`, или их следует объявить и определить только в классе `Composite` и его подклассах?

  Решая этот вопрос, мы должны выбирать между безопасностью и прозрачностью:

  - если определить интерфейс для управления потомками в корне иерархии классов, то мы добиваемся прозрачности, так как все компоненты удается трактовать единообразно. Однако расплачиваться приходится безопасностью, поскольку клиент может попытаться выполнить бессмысленное действие, например добавить или удалить объект из листового узла;
  - если управление потомками сделать частью класса `Composite`, то безопасность удастся обеспечить, ведь любая попытка добавить или удалить объекты из листьев в статически типизированном языке вроде C++ будет 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 преобразовывать 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, возвращая нулевой указатель. А в классе `Composite` эта операция переопределена и возвращает указатель `this` на сам объект:

```
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 `GetComposite` можно спросить у компонента, является ли он составным. К возвращаемому этой операцией составному объекту 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
}
```

Аналогичные проверки на принадлежность классу `Composite` в C++ выполняют 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 появится новая проблема: нельзя реализовать `Component::Add` так, чтобы она никогда не приводила к ошибке. Можно, конечно, сделать данную операцию пустой, но тогда нарушается важное проектное ограничение: попытка

добавить что-то в листовый объект, скорее всего, свидетельствует об ошибке. Допустимо было бы заставить ее удалять свой аргумент, но клиент может быть не рассчитанным на это.

Обычно лучшим решением является такая реализация `Add` и `Remove` по умолчанию, при которой они завершаются с ошибкой (возможно, возбуждая 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-прежнему нет разумной интерпретации;

- *должен ли Component реализовывать список компонентов.* Может возникнуть желание определить множество потомков в виде переменной экземпляра класса `Component`, в котором объявлены операции доступа и управления 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, только если в структуре не слишком много потомков;
- *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 составных объектах, описывающих деревья синтаксического разбора, составные операторы могут быть экземплярами класса `Composite`, порядок следования 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;
- *кэширование для повышения производительности.* Если приходится часто обходить композицию или производить в ней поиск, то класс `Composite` может кэшировать информацию об обходе и поиске. Кэшировать разрешается 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 информация позволила бы пропускать тех потомков, которые не видимы в текущем окне.

  Любое изменение компонента должно делать кэши всех его родителей недействительными. Наиболее эффективен такой подход в случае, когда компонентам известно об их родителях. Поэтому, если вы решите воспользоваться кэшированием, необходимо определить интерфейс, позволяющий уведомить составные объекты о недействительности их кэшей;
- *кто должен удалять компоненты.* В языках, где нет сборщика мусора, лучше всего поручить классу `Composite` удалять своих потомков в момент уничтожения. Исключением из этого правила является случай, когда листовые объекты постоянны и, следовательно, могут разделяться;

- *какая структура данных лучше всего подходит для хранения компонентов.* Составные объекты могут хранить своих потомков в самых разных структурах данных, включая связанные списки, деревья, массивы и хэш-таблицы. Выбор структуры данных определяется, как всегда, эффективностью. Собственно говоря, вовсе не обязательно пользоваться какой-либо из универсальных структур. Иногда в составных объектах каждый потомок представляется 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 иерархическую 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 компоновщик.

Класс `Equipment` определяет интерфейс для всех видов аппаратуры в иерархии вида часть-целое:

```
class Equipment {
public:
    virtual ~Equipment();

    const char* Name() { return _name; }

    virtual Watt Power();
    virtual Currency NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();
protected:
    Equipment(const char*);
private:
    const char* _name;
};
```

В классе `Equipment` объявлены операции, которые возвращают атрибуты аппаратного блока, например энергопотребление и стоимость. Подклассы реализуют эти операции для конкретных видов оборудования. Класс `Equipment` объявляет также операцию `CreateIterator`, возвращающую итератор `Iterator` (см. приложение С) для доступа к отдельным частям. Реализация этой операции по умолчанию возвращает итератор `NullIterator`, умеющий обходить только пустое множество.

Среди подклассов `Equipment` могут быть листовые классы, представляющие дисковые накопители, СБИС и переключатели:

```
class FloppyDisk : public Equipment {
public:
    FloppyDisk(const char*);
    virtual ~FloppyDisk();

    virtual Watt Power();
    virtual Currency NetPrice();
    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 NetPrice();
    virtual Currency DiscountPrice();

    virtual void Add(Equipment*);
    virtual void Remove(Equipment*);
    virtual Iterator<Equipment*>* CreateIterator();

protected:
    CompositeEquipment(const char*);
private:
    List<Equipment*> _equipment;
};
```

`CompositeEquipment` определяет операции для доступа и управления внутренними аппаратными блоками. Операции `Add` и `Remove` добавляют и удаляют оборудование из списка, хранящегося в переменной-члене `_equipment`. Операция `CreateIterator` возвращает итератор (точнее, экземпляр класса `ListIterator`), который будет обходить этот список.

Подразумеваемая реализация операции `NetPrice` могла бы использовать `CreateIterator` для суммирования цен на отдельные блоки:[1]

```
Currency CompositeEquipment::NetPrice () {
    Iterator<Equipment*>* i = CreateIterator();
    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 паттерна **итератор** рассказано, как защититься от таких ошибок.

Теперь мы можем представить аппаратный блок компьютера в виде подкласса к `CompositeEquipment` под названием `Chassis`. `Chassis` наследует порожденные операции класса `CompositeEquipment`.

```
class Chassis : public CompositeEquipment {
public:
    Chassis(const char*);
    virtual ~Chassis();

    virtual Watt Power();
    virtual Currency NetPrice();
    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 Bus");
bus->Add(new Card("16Mbs Token Ring"));

chassis->Add(bus);
chassis->Add(new FloppyDisk("3.5in Floppy"));

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

Известные применения

Примеры паттерна **компоновщик** можно найти почти во всех объектно-ориентированных системах. Первоначально класс `View` в схеме **модель/вид/контроллер** в языке Smalltalk [KP88] был **компоновщиком**, и почти все библиотеки для построения пользовательских интерфейсов и каркасы проектировались аналогично. Среди них ET++ (со своей библиотекой VObjects [WGM88]) и InterViews (классы Styles [LCI+92], Graphics [VL88] и Glyphs [CL90]). Интересно отметить, что первоначально вид `View` имел несколько подвидов, то есть он был одновременно и классом `Component`, и классом `Composite`. В версии 4.0 языка Smalltalk-80 схема **модель/вид/контроллер** была пересмотрена, в нее ввели класс `VisualComponent`, подклассами которого являлись `View` и `CompositeView`.

В каркасе для построения компиляторов RTL, который написан на Smalltalk [JML92], паттерн **компоновщик** используется очень широко. `RTLExpression` – это разновидность класса `Component` для построения деревьев синтаксического разбора. У него есть подклассы, например `BinaryExpression`, потомками которого являются объекты класса `RTLExpression`. В совокупности эти классы определяют составную структуру для деревьев разбора. `RegisterTransfer` – класс

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).