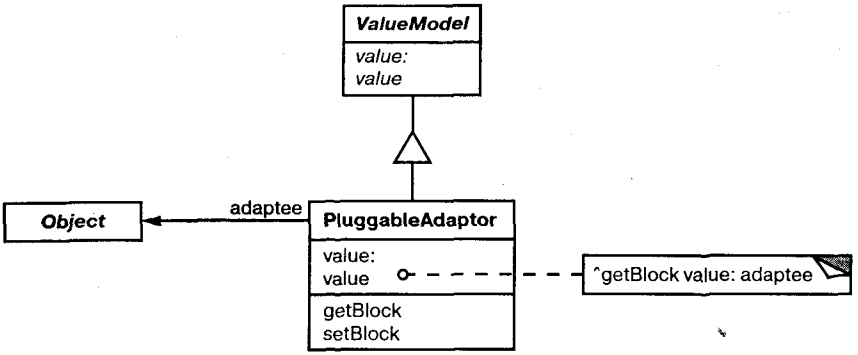


Еще один пример из ObjectWorks\Smalltalk – это класс `TableAdaptor`. Он может адаптировать последовательность объектов к табличному представлению. В таблице отображается по одному объекту в строке. Клиент параметризует `TableAdaptor` множеством сообщений, которые используются таблицей для получения от объекта значения в колонках.

В некоторых классах библиотеки NeXT AppKit [Add94] используются объекты-уполномоченные для реализации интерфейса адаптации. В качестве примера можно привести класс `NXBrowser`, который способен отображать иерархические списки данных. `NXBrowser` пользуется объектом-уполномоченным для доступа и адаптации данных.

Придуманная Скоттом Мейером (Scott Meyer) конструкция «брак по расчету» (Marriage of Convenience) [Mey88] это разновидность адаптера класса. Мейер описывает, как класс `FixedStack` адаптирует реализацию класса `Array` к интерфейсу класса `Stack`. Результатом является стек, содержащий фиксированное число элементов.

Родственные паттерны

Структура паттерна Мост аналогична структуре Адаптера, но у Моста иное назначение. Он отделяет интерфейс от реализации, чтобы то и другое можно было изменять независимо. Адаптер же призван изменить интерфейс *существующего* объекта.

Паттерн Декоратор расширяет функциональность объекта, изменяя его интерфейс. Таким образом, Декоратор более прозрачен для приложения, чем Адаптер. Как следствие, Декоратор поддерживает рекурсивную композицию, что для «чистых» Адаптеров невозможно.

Заместитель определяет представителя или суррогат другого объекта, но не изменяет его интерфейс.

Паттерн Bridge

Название и классификация паттерна

Мост – паттерн, структурирующий объекты.

Назначение

Отделить абстракцию от ее реализации так, чтобы то и другое можно было изменять независимо.

Известен также под именем

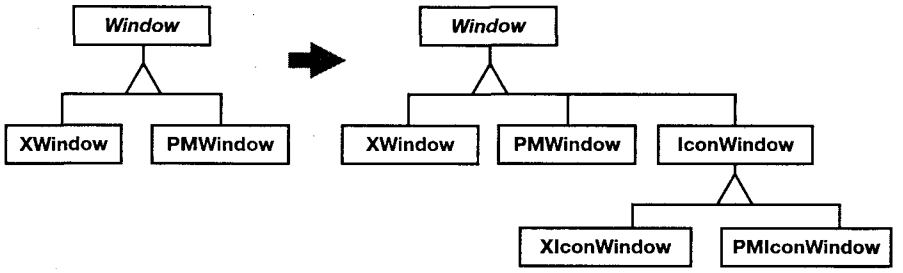
Handle/Body (описатель/тело).

Мотивация

Если для некоторой абстракции возможно несколько реализаций, то обычно применяют наследование. Абстрактный класс определяет интерфейс абстракции, а его конкретные подклассы по-разному реализуют его. Но такой подход не всегда обладает достаточной гибкостью. Наследование жестко привязывает реализацию к абстракции, что затрудняет независимую модификацию, расширение и повторное использование абстракции и ее реализации.

Рассмотрим реализацию переносимой абстракции окна в библиотеке для разработки пользовательских интерфейсов. Написанные с ее помощью приложения должны работать в разных средах, например под X Window System и Presentation Manager (PM) от компании IBM. С помощью наследования мы могли бы определить абстрактный класс `Window` и его подклассы `XWindow` и `PMWindow`, реализующие интерфейс окна для разных платформ. Но у такого решения есть два недостатка:

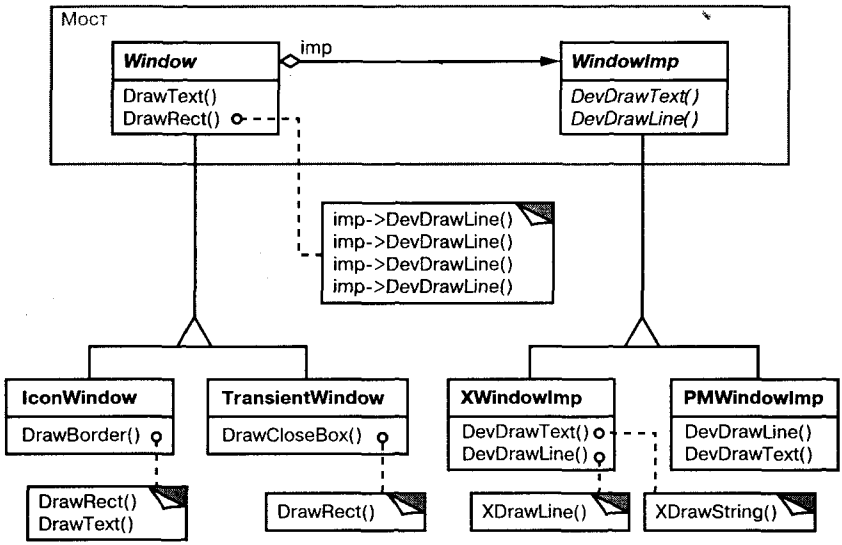
- неудобно распространять абстракцию `Window` на другие виды окон или новые платформы. Представьте себе подкласс `IconWindow`, который специализирует абстракцию окна для пиктограмм. Чтобы поддержать пиктограммы на обеих платформах, нам придется реализовать *два* новых подкласса `XIconWindow` и `PMIconWindow`. Более того, по два подкласса необходимо определять для *каждого* вида окон. А для поддержки третьей платформы придется определять для всех видов окон новый подкласс `Window`;



- клиентский код становится платформенно-зависимым. При создании окна клиент инстанцирует конкретный класс, имеющий вполне определенную реализацию. Например, создавая объект `XWindow`, мы привязываем абстракцию окна к ее реализации для системы X Window и, следовательно, делаем код клиента ориентированным именно на эту оконную систему. Таким образом усложняется перенос клиента на другие платформы.

Клиенты должны иметь возможность создавать окно, не привязываясь к конкретной реализации. Только сама реализация окна должна зависеть от платформы, на которой работает приложение. Поэтому в клиентском коде не может быть никаких упоминаний о платформах.

С помощью паттерна МОСТ эти проблемы решаются. Абстракция окна и ее реализация помещаются в раздельные иерархии классов. Таким образом, существует одна иерархия для интерфейсов окон (`Window`, `IconWindow`, `TransientWindow`) и другая (с корнем `WindowImp`) – для платформенно-зависимых реализаций. Так, подкласс `XWindowImp` предоставляет реализацию в системе X Window System.



Все операции подклассов `Window` реализованы в терминах абстрактных операций из интерфейса `WindowImp`. Это отделяет абстракцию окна от различных ее платформенно-зависимых реализаций. Отношение между классами `Window` и `WindowImp` мы будем называть *мостом*, поскольку между абстракцией и реализацией строится мост, и они могут изменяться независимо.

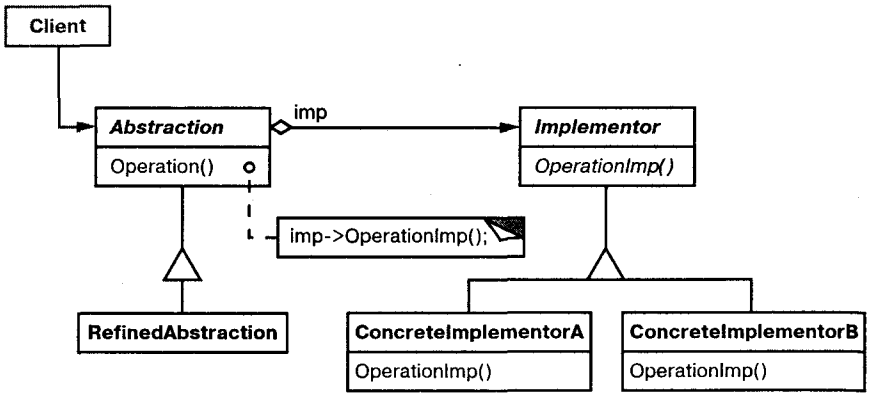
Применимость

Используйте паттерн МОСТ, когда:

- хотите избежать постоянной привязки абстракции к реализации. Так, например, бывает, когда реализацию необходимо выбирать во время выполнения программы;
- и абстракции, и реализации должны расширяться новыми подклассами. В таком случае паттерн МОСТ позволяет комбинировать разные абстракции и реализации и изменять их независимо;
- изменения в реализации абстракции не должны сказываться на клиентах, то есть клиентский код не должен перекомпилироваться;

- *(только для C++!)* вы хотите полностью скрыть от клиентов реализацию абстракции. В C++ представление класса видимо через его интерфейс;
- число классов начинает быстро расти, как мы видели на первой диаграмме из раздела «Мотивация». Это признак того, что иерархию следует разделить на две части. Для таких иерархий классов Рамбо (Rumbaugh) использует термин «вложенные обобщения» [RBP+91];
- вы хотите разделить одну реализацию между несколькими объектами (быть может, применяя подсчет ссылок), и этот факт необходимо скрыть от клиента. Простой пример – это разработанный Джеймсом Коплиеном класс `String` [Cop92], в котором разные объекты могут разделять одно и то же представление строки (`StringRep`).

Структура



Участники

- **Abstraction** (`Window`) – абстракция:
  - определяет интерфейс абстракции;
  - хранит ссылку на объект типа `Implementor`;
- **RefinedAbstraction** (`IconWindow`) – уточненная абстракция:
  - расширяет интерфейс, определенный абстракцией `Abstraction`;
- **Implementor** (`WindowImp`) – реализатор:
  - определяет интерфейс для классов реализации. Он не обязан точно соответствовать интерфейсу класса `Abstraction`. На самом деле оба интерфейса могут быть совершенно различны. Обычно интерфейс класса `Implementor` предоставляет только примитивные операции, а класс `Abstraction` определяет операции более высокого уровня, базирующиеся на этих примитивах;
- **ConcreteImplementor** (`XWindowImp`, `PMWindowImp`) – конкретный реализатор:
  - содержит конкретную реализацию интерфейса класса `Implementor`.

Отношения

Объект `Abstraction` перенаправляет своему объекту `Implementor` запросы клиента.

Результаты

Результаты применения паттерна **мост** таковы:

- *отделение реализации от интерфейса.* Реализация больше не имеет постоянной привязки к интерфейсу. Реализацию абстракции можно конфигурировать во время выполнения. Объект может даже динамически изменять свою реализацию.

  Разделение классов `Abstraction` и `Implementor` устраняет также зависимости от реализации, устанавливаемые на этапе компиляции. Чтобы изменить класс реализации, вовсе не обязательно перекомпилировать класс `Abstraction` и его клиентов. Это свойство особенно важно, если необходимо обеспечить двоичную совместимость между разными версиями библиотеки классов.

  Кроме того, такое разделение облегчает разбиение системы на слои и тем самым позволяет улучшить ее структуру. Высокоуровневые части системы должны знать только о классах `Abstraction` и `Implementor`;
- *повышение степени расширяемости.* Можно расширять независимо иерархии классов `Abstraction` и `Implementor`;
- *сокрытие деталей реализации от клиентов.* Клиентов можно изолировать от таких деталей реализации, как разделение объектов класса `Implementor` и сопутствующего механизма подсчета ссылок.

Реализация

Если вы предполагаете применить паттерн **мост**, то подумайте о таких вопросах реализации:

- *только один класс Implementor.* В ситуациях, когда есть только одна реализация, создавать абстрактный класс `Implementor` необязательно. Это вырожденный случай паттерна **мост** – между классами `Abstraction` и `Implementor` существует взаимно-однозначное соответствие. Тем не менее разделение все же полезно, если нужно, чтобы изменение реализации класса не отражалось на существующих клиентах (должно быть достаточно заново скомпоновать программу, не перекомпилируя клиентский код).

  Для описания такого разделения Каролан (Carolan) [Car89] употребляет сочетание «чеширский кот». В C++ интерфейс класса `Implementor` можно определить в закрытом заголовочном файле, который не передается клиентам. Это позволяет полностью скрыть реализацию класса от клиентов;
- *создание правильного объекта Implementor.* Как, когда и где принимается решение о том, какой из нескольких классов `Implementor` инстанцировать? Если у класса `Abstraction` есть информация о конкретных классах `ConcreteImplementor`, то он может инстанцировать один из них в своем конструкторе; какой именно – зависит от переданных конструктору параметров.

Так, если класс коллекции поддерживает несколько реализаций, то решение можно принять в зависимости от размера коллекции. Для небольших коллекций применяется реализация в виде связанного списка, для больших – в виде хэшированных таблиц.

Другой подход – заранее выбрать реализацию по умолчанию, а позже изменять ее в соответствии с тем, как она используется. Например, если число элементов в коллекции становится больше некоторой условной величины, то мы переключаемся с одной реализации на другую, более эффективную. Можно также делегировать решение другому объекту. В примере с иерархиями `Window/WindowImp` уместно было бы ввести фабричный объект (см. паттерн абстрактная фабрика), единственная задача которого – инкапсулировать платформенную специфику. Фабрика обладает информацией, объекты `WindowImp` какого вида надо создавать для данной платформы, а объект `Window` просто обращается к ней с запросом о предоставлении какого-нибудь объекта `WindowImp`, при этом понятно, что объект получит то, что нужно. Преимущество описанного подхода: класс `Abstraction` напрямую не привязан ни к одному из классов `Implementor`;

- *разделение реализаторов*. Джеймс Коплиен показал, как в C++ можно применить идиому описатель/тело, чтобы несколькими объектами могла совместно использоваться одна и та же реализация [Cop92]. В теле хранится счетчик ссылок, который увеличивается и уменьшается в классе описателя. Код для присваивания значений описателям, разделяющим одно тело, в общем виде выглядит так:

```
Handle& Handle::operator= (const Handle& other) {
    other._body->Ref();
    _body->Unref();

    if (_body->RefCount() == 0) {
        delete _body;
    }
    _body = other._body;

    return *this;
}
```

- *использование множественного наследования*. В C++ для объединения интерфейса с его реализацией можно воспользоваться множественным наследованием [Mar91]. Например, класс может открыто наследовать классу `Abstraction` и закрыто – классу `ConcreteImplementor`. Но такое решение зависит от статического наследования и жестко привязывает реализацию к ее интерфейсу. Поэтому реализовать настоящий мост с помощью множественного наследования невозможно, по крайней мере в C++.

Пример кода

В следующем коде на C++ реализован пример `Window/WindowImp`, который обсуждался в разделе «Мотивация». Класс `Window` определяет абстракцию окна для клиентских приложений:

```
class Window {
public:
    Window(View* contents);

    // запросы, обрабатываемые окном
    virtual void DrawContents();

    virtual void Open();
    virtual void Close();
    virtual void Iconify();
    virtual void Deiconify();

    // запросы, перенаправляемые реализации
    virtual void SetOrigin(const Point& at);
    virtual void SetExtent(const Point& extent);
    virtual void Raise();
    virtual void Lower();

    virtual void DrawLine(const Point&, const Point&);
    virtual void DrawRect(const Point&, const Point&);
    virtual void DrawPolygon(const Point[], int n);
    virtual void DrawText(const char*, const Point&);

protected:
    WindowImp* GetWindowImp();
    View* GetView();

private:
    WindowImp* _imp;
    View* _contents; // содержимое окна
};
```

В классе `Window` хранится ссылка на `WindowImp` – абстрактный класс, в котором объявлен интерфейс к данной оконной системе:

```
class WindowImp {
public:
    virtual void ImpTop() = 0;
    virtual void ImpBottom() = 0;
    virtual void ImpSetExtent(const Point&) = 0;
    virtual void ImpSetOrigin(const Point&) = 0;

    virtual void DeviceRect(Coord, Coord, Coord, Coord) = 0;
    virtual void DeviceText(const char*, Coord, Coord) = 0;
    virtual void DeviceBitmap(const char*, Coord, Coord) = 0;
    // множество других функций для рисования в окне...
protected:
    WindowImp();
};
```

Подклассы `Window` определяют различные виды окон, как то: окно приложения, пиктограмма, временное диалоговое окно, плавающая палитра инструментов и т.д.

Например, класс ApplicationWindow реализует операцию DrawContents для отрисовки содержимого экземпляра класса View, который в нем хранится:

```
class ApplicationWindow : public Window {
public:
    // ...
    virtual void DrawContents();
};

void ApplicationWindow::DrawContents () {
    GetView()->DrawOn(this);
}
```

А в классе IconWindow содержится имя растрового изображения для пиктограммы

```
class IconWindow : public Window {
public:
    // ...
    virtual void DrawContents();
private:
    const char* _bitmapName;
};
```

и реализация операции DrawContents для рисования этого изображения в окне:

```
void IconWindow::DrawContents() {
    WindowImp* imp = GetWindowImp();
    if (imp != 0) {
        imp->DeviceBitmap(_bitmapName, 0.0, 0.0);
    }
}
```

Могут существовать и другие разновидности класса Window. Окну класса TransientWindow иногда необходимо как-то сообщаться с создавшим его окном во время диалога, поэтому в объекте класса хранится ссылка на создателя. Окно класса PaletteWindow всегда располагается поверх других. Окно класса IconDockWindow (контейнер пиктограмм) хранит окна класса IconWindow и располагает их в ряд.

Операции класса Window определены в терминах интерфейса WindowImp. Например, DrawRect вычисляет координаты по двум своим параметрам Point перед тем, как вызвать операцию WindowImp, которая рисует в окне прямоугольник:

```
void Window::DrawRect (const Point& p1, const Point& p2) {
    WindowImp* imp = GetWindowImp();
    imp->DeviceRect(p1.X(), p1.Y(), p2.X(), p2.Y());
}
```

Конкретные подклассы WindowImp поддерживают разные оконные системы. Так, класс XWindowImp ориентирован на систему X Window:

```
class XWindowImp : public WindowImp {
public:
    XWindowImp();

    virtual void DeviceRect(Coord, Coord, Coord, Coord);
    // прочие операции открытого интерфейса...
private:
    // переменные, описывающие специфичное для X Window состояние,
    // в том числе:
    Display* _dpy;
    Drawable _winid; // идентификатор окна
    GC _gc;          // графический контекст окна
};
```

Для Presentation Manager (PM) мы определяем класс `PMWindowImp`:

```
class PMWindowImp : public WindowImp {
public:
    PMWindowImp();
    virtual void DeviceRect(Coord, Coord, Coord, Coord);

    // прочие операции открытого интерфейса...
private:
    // переменные, описывающие специфичное для PM Window состояние,
    // в том числе:
    HPS _hps;
};
```

Эти подклассы реализуют операции `WindowImp` в терминах примитивов оконной системы. Например, `DeviceRect` для X Window реализуется так:

```
void XWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    int x = round(min(x0, x1));
    int y = round(min(y0, y1));
    int w = round(abs(x0 - x1));
    int h = round(abs(y0 - y1));
    XDrawRectangle(_dpy, _winid, _gc, x, y, w, h);
}
```

А для PM – так:

```
void PMWindowImp::DeviceRect (
    Coord x0, Coord y0, Coord x1, Coord y1
) {
    Coord left = min(x0, x1);
    Coord right = max(x0, x1);
    Coord bottom = min(y0, y1);
    Coord top = max(y0, y1);

    PPOINTL point[4];
```

```
    point[0].x = left;  point[0].y = top;
    point[1].x = right; point[1].y = top;
    point[2].x = right; point[2].y = bottom;
    point[3].x = left;  point[3].y = bottom;

    if (
        (GpiBeginPath(_hps, 1L) == false) ||
        (GpiSetCurrentPosition(_hps, &point[3]) == false) ||
        (GpiPolyLine(_hps, 4L, point) == GPI_ERROR) ||
        (GpiEndPath(_hps) == false)
    ) {
    // сообщить об ошибке

    } else {
        GpiStrokePath(_hps, 1L, 0L);
    }
}
```

Как окно получает экземпляр нужного подкласса `WindowImp`? В данном примере мы предположим, что за это отвечает класс `Window`. Его операция `GetWindowImp` получает подходящий экземпляр от абстрактной фабрики (см. описание паттерна абстрактная фабрика), которая инкапсулирует все зависимости от оконной системы.

```
WindowImp* Window::GetWindowImp () {
    if (_imp == 0) {
        _imp = WindowSystemFactory::Instance()->MakeWindowImp();
    }
    return _imp;
}
```

`WindowSystemFactory::Instance()` возвращает абстрактную фабрику, которая изготавливает все системно-зависимые объекты. Для простоты мы сделали эту фабрику одиночкой и позволили классу `Window` обращаться к ней напрямую.

Известные применения

Пример класса `Window` позаимствован из ET++ [WGM88]. В ET++ класс `WindowImp` называется `WindowPort` и имеет такие подклассы, как `XWindowPort` и `SunWindowPort`. Объект `Window` создает соответствующего себе реализатора `Implementor`, запрашивая его у абстрактной фабрики, которая называется `WindowSystem`. Эта фабрика предоставляет интерфейс для создания платформенно-зависимых объектов: шрифтов, курсоров, растровых изображений и т.д.

Дизайн классов `Window/WindowPort` в ET++ обобщает паттерн мост в том отношении, что `WindowPort` сохраняет также обратную ссылку на `Window`. Класс-реализатор `WindowPort` использует эту ссылку для извещения `Window` о событиях, специфичных для `WindowPort`: поступлений событий ввода, изменениях размера окна и т.д.

В работах Джеймса Коплиена [Cop92] и Бьерна Страуструпа [Str91] упоминаются классы описателей и приводятся некоторые примеры. Основной акцент

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.