Например, для каркаса виртуальной памяти фасадом служит Domain. Класс Domain представляет адресное пространство. Он обеспечивает отображение между виртуальными адресами и смещениями объектов в памяти, файле или на устройстве длительного хранения. Базовые операции класса Domain поддерживают добавление объекта в память по указанному адресу, удаление объекта из памяти и обработку ошибок отсутствия страниц.

Как видно из вышеприведенной диаграммы, внутри подсистемы виртуальной памяти используются следующие компоненты:

- MemoryObject представляет объекты данных;
- MemoryObjectCache кэширует данные из объектов MemoryObjects в физической памяти. MemoryObjectCache – это не что иное, как объект Стратегия, в котором локализована политика кэширования;
- AddressTranslation инкапсулирует особенности оборудования трансляции адресов.

Операция RepairFault вызывается при возникновении ошибки из-за отсутствия страницы. Domain находит объект в памяти по адресу, где произошла ошибка и делегирует операцию RepairFault кэшу, ассоциированному с этим объектом. Поведение объектов Domain можно настроить, заменив их компоненты.

Родственные паттерны

Паттерн абстрактная фабрика допустимо использовать вместе с фасадом, чтобы предоставить интерфейс для создания объектов подсистем способом, не зависимым от этих подсистем. Абстрактная фабрика может выступать и как альтернатива фасаду, чтобы скрыть платформенно-зависимые классы.

Паттерн посредник аналогичен фасаду в том смысле, что абстрагирует функциональность существующих классов. Однако назначение посредника – абстрагировать произвольное взаимодействие между «сотрудничающими» объектами. Часто он централизует функциональность, не присущую ни одному из них. Коллеги посредника обмениваются информацией именно с ним, а не напрямую между собой. Напротив, фасад просто абстрагирует интерфейс объектов подсистемы, чтобы ими было проще пользоваться. Он не определяет новой функциональности, и классам подсистемы ничего неизвестно о его существовании.

Обычно требуется только один фасад. Поэтому объекты фасадов часто бывают одиночками.

Паттерн Flyweight

Название и классификация паттерна

Приспособленец – паттерн, структурирующий объекты.

Назначение

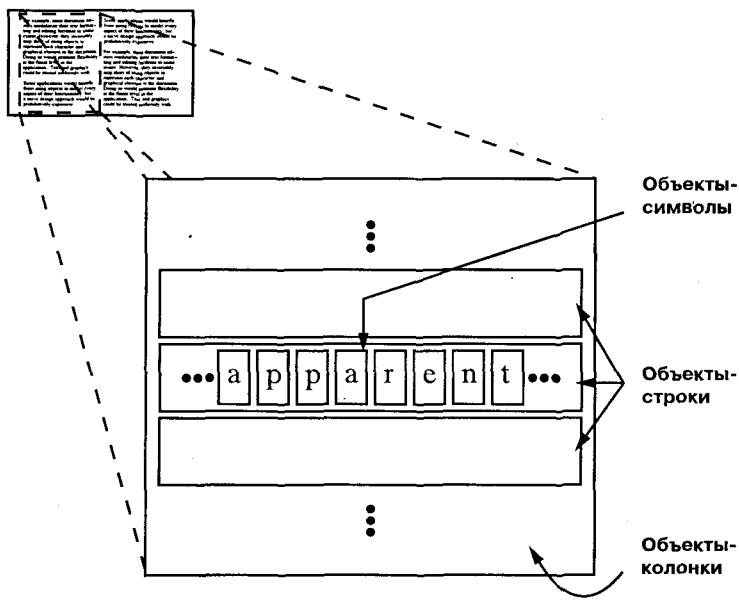
Использует разделение для эффективной поддержки множества мелких объектов.

Мотивация

В некоторых приложениях использование объектов могло бы быть очень полезным, но прямолинейная реализация оказывается недопустимо расточительной.

Например, в большинстве редакторов документов имеются средства форматирования и редактирования текстов, в той или иной степени модульные. Объектно-ориентированные редакторы обычно применяют объекты для представления таких встроенных элементов, как таблицы и рисунки. Но они не используют объекты для представления каждого символа, несмотря на то что это увеличило бы гибкость на самых нижних уровнях приложения. Ведь тогда к рисованию и форматированию символов и встроенных элементов можно было бы применить единообразный подход. И для поддержки новых наборов символов не пришлось бы как-либо затрагивать остальные функции редактора. Да и общая структура приложения отражала бы физическую структуру документа. На следующей диаграмме показано, как редактор документов мог бы воспользоваться объектами для представления символов.

У такого дизайна есть один недостаток – стоимость. Даже в документе скромных размеров было бы несколько сотен тысяч объектов-символов, а это привело бы к расходованию огромного объема памяти и неприемлемым затратам во время выполнения. Паттерн приспособленец показывает, как разделять очень мелкие объекты без недопустимо высоких издержек.

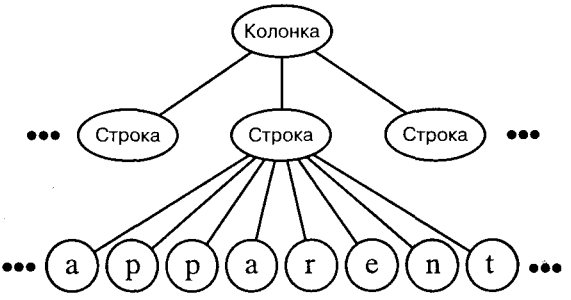


Приспособленец – это разделяемый объект, который можно использовать одновременно в нескольких контекстах. В каждом контексте он выглядит как независимый объект, то есть неотличим от экземпляра, который не разделяется. Приспособленцы не могут делать предположений о контексте, в котором работают.

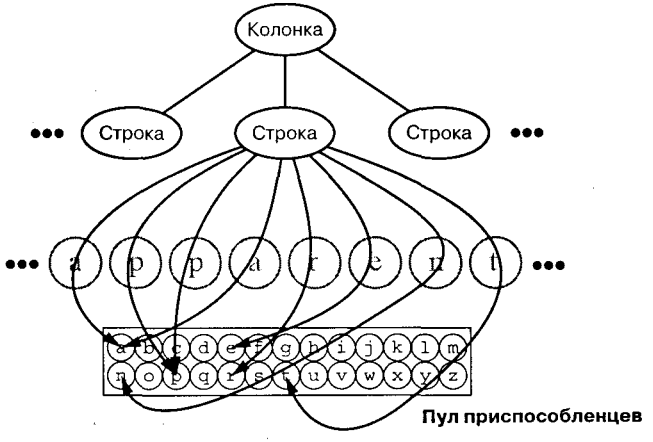
Ключевая идея здесь – различие между *внутренним* и *внешним* состояниями. Внутреннее состояние хранится в самом приспособленце и состоит из информации, не зависящей от его контекста. Именно поэтому он может разделяться. Внешнее состояние зависит от контекста и изменяется вместе с ним, поэтому не подлежит разделению. Объекты-клиенты отвечают за передачу внешнего состояния приспособленцу, когда в этом возникает необходимость.

Приспособленцы моделируют концепции или сущности, число которых слишком велико для представления объектами. Например, редактор документов мог бы создать по одному приспособленцу для каждой буквы алфавита. Каждый приспособленец хранит код символа, но координаты положения символа в документе и стиль его начертания определяются алгоритмами размещения текста и командами форматирования, действующими в том месте, где символ появляется. Код символа – это внутреннее состояние, а все остальное – внешнее.

Логически для каждого вхождения данного символа в документ существует объект.

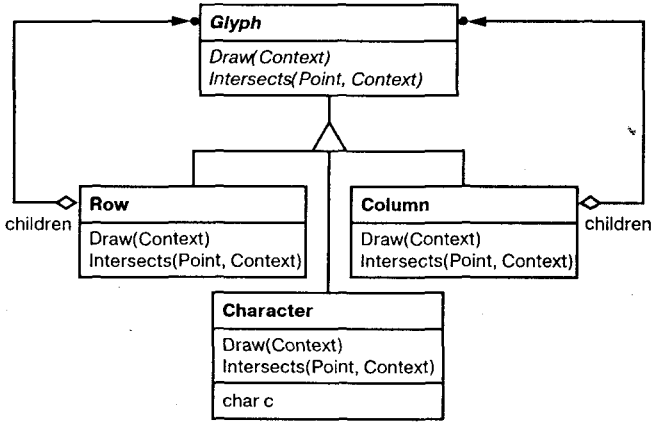


Физически, однако, есть лишь по одному объекту-приспособленцу для каждого символа, который появляется в различных контекстах в структуре документа. Каждое вхождение данного объекта-символа ссылается на один и тот же экземпляр в разделяемом пуле объектов-приспособленцев.



Пул приспособленцев

Ниже изображена структура класса для этих объектов. `Glyph` – это абстрактный класс для представления графических объектов (некоторые из них могут быть приспособленцами). Операции, которые могут зависеть от внешнего состояния, передают его в качестве параметра. Например, операциям `Draw` (рисование) и `Intersects` (пересечение) должно быть известно, в каком контексте встречается глиф, иначе они не смогут выполнить то, что от них требуется.



Приспособленец, представляющий букву «а», содержит только соответствующий ей код; ни положение, ни шрифт буквы ему хранить не надо. Клиенты передают приспособленцу всю зависящую от контекста информацию, которая нужна, чтобы он мог изобразить себя. Например, глифу `Row` известно, где его потомки должны себя показать, чтобы это выглядело как горизонтальная строка. Поэтому вместе с запросом на рисование он может передавать каждому потомку координаты.

Поскольку число различных объектов-символов гораздо меньше, чем число символов в документе, то и общее количество объектов существенно меньше, чем было бы при простой реализации. Документ, в котором все символы изображаются одним шрифтом и цветом, создаст порядка 100 объектов-символов (это примерно равно числу кодов в таблице ASCII) независимо от своего размера. А поскольку в большинстве документов применяется не более десятка различных комбинаций шрифта и цвета, то на практике эта величина возрастет несущественно. Поэтому абстракция объекта становится применимой и к отдельным символам.

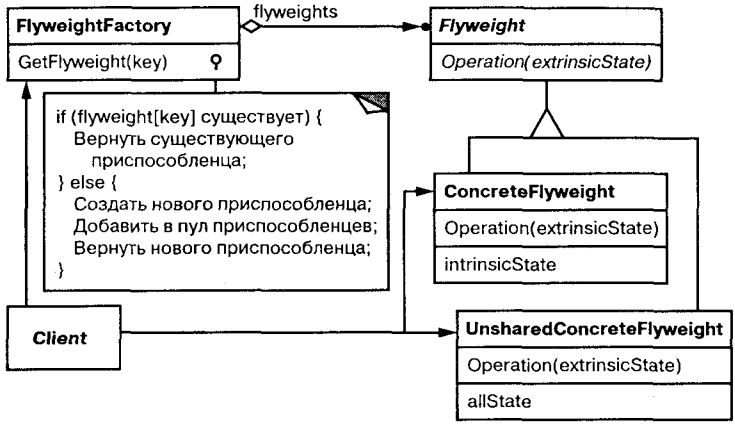
Применимость

Эффективность паттерна приспособленец во многом зависит от того, как и где он используется. Применяйте этот паттерн, когда выполнены *все* нижеперечисленные условия:

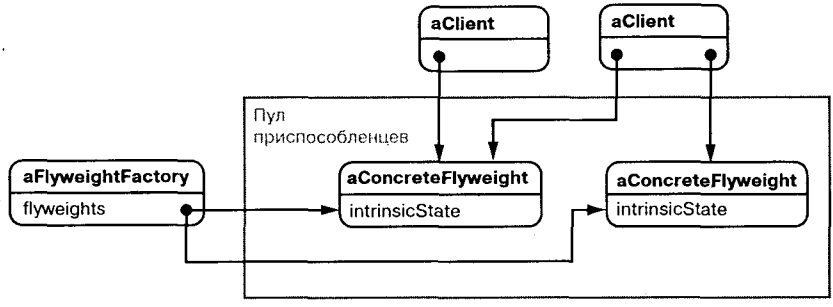
- в приложении используется большое число объектов;
- из-за этого накладные расходы на хранение высоки;
- большую часть состояния объектов можно вынести вовне;
- многие группы объектов можно заменить относительно небольшим количеством разделяемых объектов, поскольку внешнее состояние вынесено;

- приложение не зависит от идентичности объекта. Поскольку объекты-приспособленцы могут разделяться, то проверка на идентичность возвратит «истину» для концептуально различных объектов.

Структура



На следующей диаграмме показано, как приспособленцы разделяются.



Участники

- **Flyweight** (Glyph) – приспособленец:
  - объявляет интерфейс, с помощью которого приспособленцы могут получать внешнее состояние или как-то воздействовать на него;
- **ConcreteFlyweight** (Character) – конкретный приспособленец:
  - реализует интерфейс класса `Flyweight` и добавляет при необходимости внутреннее состояние. Объект класса `ConcreteFlyweight` должен быть разделяемым. Любое сохраняемое им состояние должно быть внутренним, то есть не зависящим от контекста;
- **UnsharedConcreteFlyweight** (`Row`, `Column`) – неразделяемый конкретный приспособленец:
  - не все подклассы `Flyweight` обязательно должны быть разделяемыми. Интерфейс `Flyweight` допускает разделение, но не навязывает его. Часто у объектов `UnsharedConcreteFlyweight` на некотором уровне структуры

приспособленца есть потомки в виде объектов класса `ConcreteFlyweight`, как, например, у объектов классов `Row` и `Column`;

- **FlyweightFactory** – фабрика приспособленцев:
  - создает объекты-приспособленцы и управляет ими;
  - обеспечивает должное разделение приспособленцев. Когда клиент запрашивает приспособленца, объект `FlyweightFactory` предоставляет существующий экземпляр или создает новый, если готового еще нет;
- **Client** – клиент:
  - хранит ссылки на одного или нескольких приспособленцев;
  - вычисляет или хранит внешнее состояние приспособленцев.

Отношения

- состояние, необходимое приспособленцу для нормальной работы, можно охарактеризовать как внутреннее или внешнее. Первое хранится в самом объекте `ConcreteFlyweight`. Внешнее состояние хранится или вычисляется клиентами. Клиент передает его приспособленцу при вызове операций;
- клиенты не должны создавать экземпляры класса `ConcreteFlyweight` напрямую, а могут получать их только от объекта `FlyweightFactory`. Это позволит гарантировать корректное разделение.

Результаты

При использовании приспособленцев не исключены затраты на передачу, поиск или вычисление внутреннего состояния, особенно если раньше оно хранилось как внутреннее. Однако такие расходы с лихвой компенсируются экономией памяти за счет разделения объектов-приспособленцев.

Экономия памяти возникает по ряду причин:

- уменьшение общего числа экземпляров;
- сокращение объема памяти, необходимого для хранения внутреннего состояния;
- вычисление, а не хранение внешнего состояния (если это действительно так).

Чем выше степень разделения приспособленцев, тем существеннее экономия. С увеличением объема разделяемого состояния экономия также возрастает. Самого большого эффекта удается добиться, когда суммарный объем внутренней и внешней информации о состоянии велик, а внешнее состояние вычисляется, а не хранится. Тогда разделение уменьшает стоимость хранения внутреннего состояния, а за счет вычислений сокращается память, отводимая под внешнее состояние.

Паттерн **приспособленец** часто применяется вместе с **компоновщиком** для представления иерархической структуры в виде графа с разделяемыми листовыми узлами. Из-за разделения указатель на родителя не может храниться в листовом узле-приспособленце, а должен передаваться ему как часть внешнего состояния. Это оказывает заметное влияние на способ взаимодействия объектов иерархии между собой.

Реализация

При реализации приспособленца следует обратить внимание на следующие вопросы:

- *вынесение внешнего состояния*. Применимость паттерна в значительной степени зависит от того, насколько легко идентифицировать внешнее состояние и вынести его за пределы разделяемых объектов. Вынесение внешнего состояния не уменьшает стоимости хранения, если различных внешних состояний так же много, как и объектов до разделения. Лучший вариант – внешнее состояние вычисляется по объектам с другой структурой, требующей значительно меньшей памяти.

  Например, в нашем редакторе документов мы можем поместить карту с типографской информацией в отдельную структуру, а не хранить шрифт и начертание вместе с каждым символом. Данная карта будет отслеживать непрерывные серии символов с одинаковыми типографскими атрибутами. Когда объект-символ изображает себя, он получает типографские атрибуты от алгоритма обхода. Поскольку обычно в документах используется немного разных шрифтов и начертаний, то хранить эту информацию отдельно от объекта-символа гораздо эффективнее, чем непосредственно в нем;
- *управление разделяемыми объектами*. Так как объекты разделяются, клиенты не должны инстанцировать их напрямую. Фабрика `FlyweightFactory` позволяет клиентам найти подходящего приспособленца. В объектах этого класса часто есть хранилище, организованное в виде ассоциативного массива, с помощью которого можно быстро находить приспособленца, нужного клиенту. Так, в примере редактора документов фабрика приспособленцев может содержать внутри себя таблицу, индексированную кодом символа, и возвращать нужного приспособленца по его коду. А если требуемый приспособленец отсутствует, он тут же создается.

  Разделяемость подразумевает также, что имеется некоторая форма подсчета ссылок или сбора мусора для освобождения занимаемой приспособленцем памяти, когда необходимость в нем отпадает. Однако ни то, ни другое необязательно, если число приспособленцев фиксировано и невелико (например, если речь идет о представлении набора символов кода ASCII). В таком случае имеет смысл хранить приспособленцев постоянно.

Пример кода

Возвращаясь к примеру с редактором документов, определим базовый класс `Glyph` для графических объектов-приспособленцев. Логически глифы – это составные объекты, которые обладают графическими атрибутами и умеют изображать себя (см. описание паттерна КОМПОНОВЩИК). Сейчас мы ограничимся только шрифтом, но тот же подход применим и к любым другим графическим атрибутам:

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

    virtual void Draw(Window*, GlyphContext&);

    virtual void SetFont(Font*, GlyphContext&);
    virtual Font* GetFont(GlyphContext&);
```

```
    virtual void First(GlyphContext&);
    virtual void Next(GlyphContext&);
    virtual bool IsDone(GlyphContext&);
    virtual Glyph* Current(GlyphContext&);

    virtual void Insert(Glyph*, GlyphContext&);
    virtual void Remove(GlyphContext&);
protected:
    Glyph();
};
```

В подклассе `Character` хранится просто код символа:

```
class Character : public Glyph {
public:
    Character(char);

    virtual void Draw(Window*, GlyphContext&);
private:
    char _charcode;
};
```

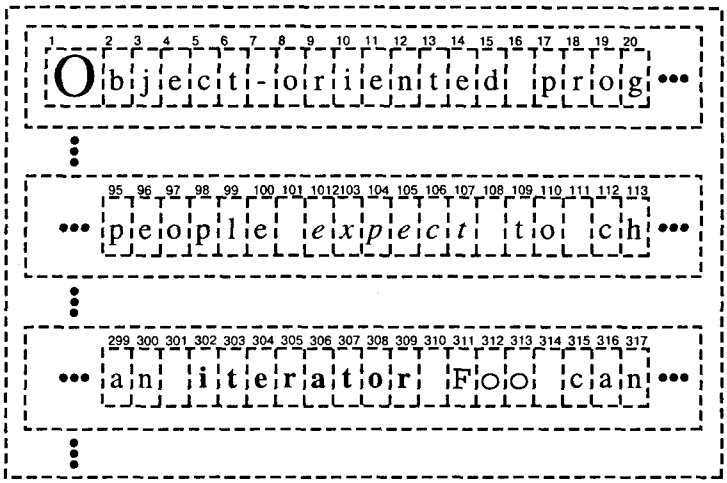
Чтобы не выделять память для шрифта каждого глифа, будем хранить этот атрибут во внешнем объекте класса `GlyphContext`. Данный объект поддерживает соответствие между глифом и его шрифтом (а также любыми другими графическими атрибутами) в различных контекстах. Любой операции, у которой должна быть информация о шрифте глифа в данном контексте, в качестве параметра будет передаваться экземпляр `GlyphContext`. У него операция и может запросить нужные сведения. Контекст определяется положением глифа в структуре. Поэтому операциями обхода и манипулирования потомками обновляется `GlyphContext`:

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

    virtual void Next(int step = 1);
    virtual void Insert(int quantity = 1);

    virtual Font* GetFont();
    virtual void SetFont(Font*, int span = 1);
private:
    int _index;
    BTree* _fonts;
};
```

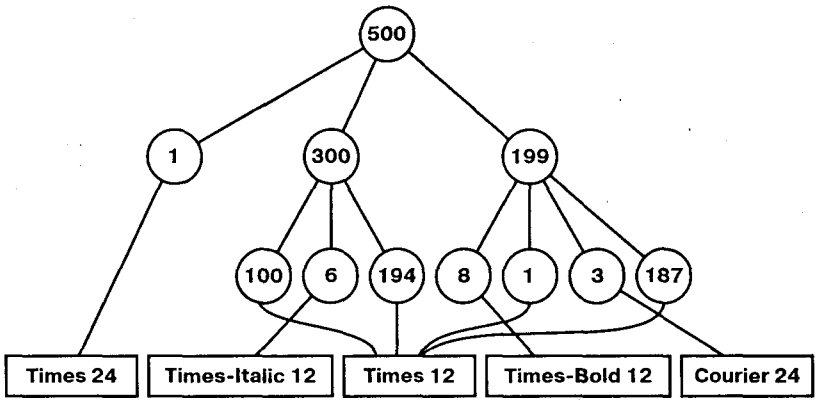
Объекту `GlyphContext` должно быть известно о текущем положении в структуре глифов во время ее обхода. Операция `GlyphContext::Next` увеличивает переменную `_index` по мере обхода структуры. Подклассы класса `Glyph`, имеющие потомков (например, `Row` и `Column`), должны реализовывать операцию `Next` так, чтобы она вызывала `GlyphContext::Next` в каждой точке обхода.

Операция `GlyphContext::GetFont` использует переменную `_index` в качестве ключа для структуры `BTree`, в которой хранится отображение между глифами и шрифтами. Каждый узел дерева помечен длиной строки, для которой он предоставляет информацию о шрифте. Листья дерева указывают на шрифт, а внутренние узлы разбивают строку на подстроки – по одной для каждого потомка.

Рассмотрим фрагмент текста, представляющий собой композицию глифов.



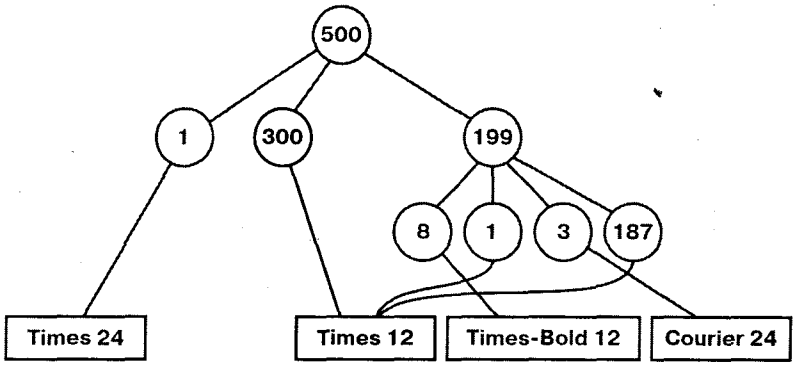
Структура `BTree`, в которой хранится информация о шрифтах, может выглядеть так:



Внутренние узлы определяют диапазоны индексов глифов. Дерево обновляется в ответ на изменение шрифта, а также при каждом добавлении и удалении глифов из структуры. Например, если предположить, что текущей точке обхода соответствует индекс 102, то следующий код установит шрифт каждого символа в слове «expect» таким же, как у близлежащего текста (то есть `times12` – экземпляр класса `Font` для шрифта Times Roman размером 12 пунктов):

```
GlyphContext gc;
Font* times12 = new Font("Times-Roman-12");
Font* timesItalic12 = new Font("Times-Italic-12");
// ...

gc.SetFont(times12, 6);
```

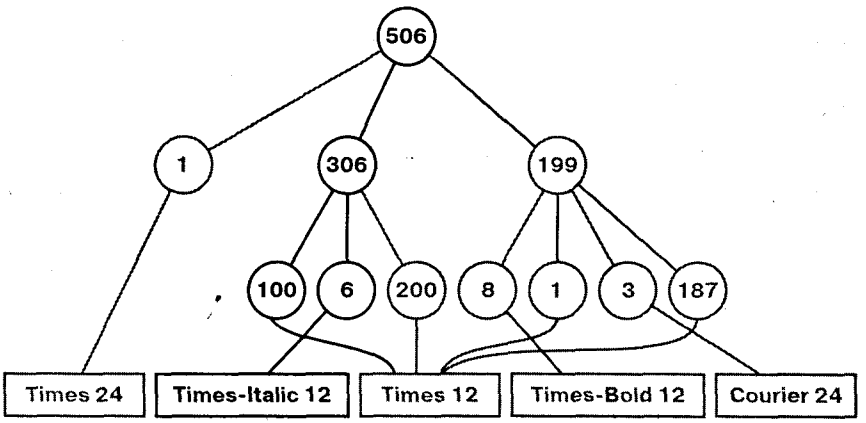
Новая структура `BTree` выглядит так (изменения выделены более ярким цветом):



Добавим перед «expect» слово «don't » (включая пробел после него), написанное шрифтом Times Italic размером 12 пунктов. В предположении, что текущей позиции все еще соответствует индекс 102, следующий код проинформирует объект `gc` об этом:

```
gc.Insert(6);
gc.SetFont(timesItalic12, 6);
```

Теперь структура `BTree` выглядит так:



При запрашивании шрифта текущего глифа объект `GlyphContext` спускается вниз по дереву, суммируя индексы, пока не будет найден шрифт для текущего

индекса. Поскольку шрифт меняется нечасто, размер дерева мал по сравнению с размером структуры глифов. Это позволяет уменьшить расходы на хранение без заметного увеличения времени поиска.[1]

И наконец, нам нужна еще фабрика `FlyweightFactory`, которая создает глифы и обеспечивает их корректное разделение. Класс `GlyphFactory` создает объекты `Character` и глифы других видов. Разделению подлежат только объекты `Character`. Составных глифов гораздо больше, и их существенное состояние (то есть множество потомков) в любом случае является внутренним:

```
const int NCHARCODES = 128;

class GlyphFactory {
public:
    GlyphFactory();
    virtual ~GlyphFactory();

    virtual Character* CreateCharacter(char);
    virtual Row* CreateRow();
    virtual Column* CreateColumn();
    // ...
private:
    Character* _character[NCHARCODES];
};
```

Массив `_character` содержит указатели на глифы `Character`, индексированные кодом символа. Конструктор инициализирует этот массив нулями:

```
GlyphFactory::GlyphFactory () {
    for (int i = 0; i < NCHARCODES; ++i) {
        _character[i] = 0;
    }
}
```

Операция `CreateCharacter` ищет символ в массиве и возвращает соответствующий глиф, если он существует. В противном случае `CreateCharacter` создает глиф, помещает его в массив и затем возвращает:

```
Character* GlyphFactory::CreateCharacter (char c) {
    if (!_character[c]) {
        _character[c] = new Character(c);
    }

    return _character[c];
}
```

Остальные операции просто создают новый объект при каждом обращении, так как несимвольные глифы не разделяются:

---

[1] Время поиска в этой схеме пропорционально частоте смены шрифта. Наименьшая производительность бывает, когда смена шрифта происходит на каждом символе, но на практике это бывает редко.

```
Row* GlyphFactory::CreateRow () {
    return new Row;
}

Column* GlyphFactory::CreateColumn () {
    return new Column;
}
```

Эти операции можно было бы опустить и позволить клиентам инстанцировать неразделяемые глифы напрямую. Но если позже мы решим сделать разделяемыми и их тоже, то придется изменять клиентский код, в котором они создаются.

Известные применения

Концепция объектов-приспособленцев впервые была описана и использована как техника проектирования в библиотеке InterViews 3.0 [CL90]. Ее разработчики построили мощный редактор документов Doc, чтобы доказать практическую полезность подобной идеи. В Doc объекты-глифы используются для представления любого символа документа. Редактор строит по одному экземпляру глифа для каждого сочетания символа и стиля (в котором определены все графические атрибуты). Таким образом, внутреннее состояние символа состоит из его кода и информации о стиле (индекс в таблицу стилей).[1] Следовательно, внешней оказывается только позиция, поэтому Doc работает быстро. Документы представляются классом `Document`, который выполняет функции фабрики `FlyweightFactory`. Измерения показали, что реализованное в Doc разделение символов-приспособленцев весьма эффективно. В типичном случае для документа из 180 тысяч знаков необходимо создать только 480 объектов-символов.

В каркасе ET++ [WGM88] приспособленцы используются для поддержки независимости от внешнего облика.[2] Его стандарт определяет расположение элементов пользовательского интерфейса (полос прокрутки, кнопок, меню и пр., в совокупности именуемых виджетами) и их оформления (тени и т.д.). Виджет делегирует заботу о своем расположении и изображении отдельному объекту `Layout`. Изменение этого объекта ведет к изменению внешнего облика даже во время выполнения.

Для каждого класса виджета имеется соответствующий класс `Layout` (например, `ScrollbarLayout`, `MenubarLayout` и т.д.). В данном случае очевидная проблема состоит в том, что удваивается число объектов пользовательского интерфейса, ибо для каждого интерфейсного объекта есть дополнительный объект `Layout`. Чтобы избавиться от расходов, объекты `Layout` реализованы в виде приспособленцев. Они прекрасно подходят на эту роль, так как заняты преимущественно определением поведения и им легко передать тот небольшой объем внешней информации о состоянии, который необходим для изображения объекта.

---

[1] В приведенном выше примере кода информация о стиле вынесена наружу, так что внутреннее состояние – это только код символа.

[2] Другой подход к обеспечению независимости от внешнего облика см. в описании паттерна **абстрактная фабрика**.

Объекты Layout создаются и управляются объектами класса Look. Класс Look – это абстрактная фабрика, которая производит объекты Layout с помощью таких операций, как GetButtonLayout, GetMenuBarLayout и т.д. Для каждого стандарта внешнего облика у класса Look есть соответствующий подкласс (MotifLook, OpenLook и т.д.).

Кстати говоря, объекты Layout – это, по существу, стратегии (см. описание паттерна стратегия). Таким образом, мы имеем пример объекта-стратегии, реализованный в виде приспособленца.

Родственные паттерны

Паттерн приспособленец часто используется в сочетании с компоновщиком для реализации иерархической структуры в виде ациклического направленного графа с разделяемыми листовыми вершинами.

Часто наилучшим способом реализации объектов состояния и стратегии является паттерн приспособленец.

Паттерн Proxy

Название и классификация паттерна

Заместитель – паттерн, структурирующий объекты.

Назначение

Является суррогатом другого объекта и контролирует доступ к нему.

Известен также под именем

Surrogate (суррогат).

Мотивация

Разумно управлять доступом к объекту, поскольку тогда можно отложить расходы на создание и инициализацию до момента, когда объект действительно понадобится. Рассмотрим редактор документов, который допускает встраивание в документ графических объектов. Затраты на создание некоторых таких объектов, например больших растровых изображений, могут быть весьма значительны. Но документ должен открываться быстро, поэтому следует избегать создания всех «тяжелых» объектов на стадии открытия (да и вообще это излишне, поскольку не все они будут видны одновременно).

В связи с такими ограничениями кажется разумным создавать «тяжелые» объекты *по требованию*. Это означает «когда изображение становится видимым». Но что поместить в документ вместо изображения? И как, не усложняя реализации редактора, скрыть то, что изображение создается по требованию? Например, оптимизация не должна отражаться на коде, отвечающем за рисование и форматирование.

Решение состоит в том, чтобы использовать другой объект – *заместитель* изображения, который временно подставляется вместо реального изображения. Заместитель ведет себя точно так же, как само изображение, и выполняет при необходимости его инстанцирование.