Доступ к распределенной информации

Для многих видов анализа необходимо рассматривать текст посимвольно. Но анализируемый текст рассеян по иерархии структур, состоящих из объектов-глифов. Чтобы исследовать текст, представленный в таком виде, нужен механизм доступа, знающий о структурах данных, в которых хранится текст. Для некоторых глифов потомки могут храниться в связанных списках, для других – в массивах, а для третьих и вовсе используются какие-то экзотические структуры. Наш механизм доступа должен справляться со всем этим.

К сожалению, для разных видов анализа методы доступа к информации могут различаться. Обычно текст сканируется от начала к концу. Но иногда требуется сделать прямо противоположное. Например, для реверсивного поиска нужно проходить по тексту в обратном, а не в прямом направлении. А при вычислении алгебраических выражений необходим внутренний порядок обхода.

Итак, наш механизм доступа должен уметь приспосабливаться к разным структурам данных и поддерживать разные способы обхода.

Инкапсуляция доступа и порядка обхода

Пока что в нашем интерфейсе глифов для доступа к потомкам со стороны клиентов используется целочисленный индекс. Хотя для тех классов глифов, которые содержат потомков в массиве, это, может быть, и разумно, но совершенно неэффективно для глифов, пользующихся связанным списком. Роль абстракции глифов в том, чтобы скрыть структуру данных, в которой содержатся потомки. Тогда мы сможем изменить данную структуру, не затрагивая другие классы.

Поэтому только глифу разрешено знать, какую структуру он использует. Отсюда следует, что интерфейс глифов не должен отдавать предпочтение какой-то одной структуре данных. Например, не следует оптимизировать его в пользу массивов, а не связанных списков, как это делалось до сих пор.

Мы можем решить проблему и одновременно поддержать несколько разных способов обхода. Разумно включить возможности множественного доступа и обхода прямо в классы глифов и предоставить способ выбирать между ними, возможно, передавая константу, являющуюся элементом некоторого перечисления. Выполняя обход, классы передают этот параметр друг другу, чтобы гарантировать, что все они обходят структуру в одном и том же порядке. Так же должна передаваться любая информация, собранная во время обхода.

Для поддержки данного подхода мы могли бы добавить в интерфейс класса `Glyph` следующие абстрактные операции:

```
void First(Traversal kind)
void Next()
bool IsDone()
Glyph* GetCurrent()
void Insert(Glyph*)
```

Операции `First`, `Next` и `IsDone` управляют обходом. `First` производит инициализацию. В качестве параметра передается вид обхода в виде перечисляемой константы типа `Traversal`, которая может принимать такие значения, как

CHILDREN (обходить только прямых потомков глифа), PREORDER (обходить всю структуру в прямом порядке), POSTORDER (в обратном порядке) или INORDER (во внутреннем порядке). Next переходит к следующему глифу в порядке обхода, а IsDone сообщает, закончился ли обход. GetCurrent заменяет операцию Child – осуществляет доступ к текущему в данном обходе глифу. Старая операция Insert переписывается, теперь она вставляет глиф в текущую позицию.

При анализе можно было бы использовать следующий код на C++ для обхода структуры глифов с корнем в g в прямом порядке:

```
Glyph* g;

for (g->First(PREORDER); !g->IsDone(); g->Next()) {
    Glyph* current = g->GetCurrent();

    // выполнить анализ
}
```

Обратите внимание, что мы исключили целочисленный индекс из интерфейса глифов. Не осталось ничего, что предполагало бы какой-то предпочтительный контейнер. Мы также уберегли клиенты от необходимости самостоятельно реализовывать типичные виды доступа.

Но этот подход еще не идеален. Во-первых, здесь не поддерживаются новые виды обхода, не расширяется множество значений перечисления и не добавляются новые операции. Предположим, что нам нужен вариант прямого обхода, при котором автоматически пропускаются нетекстовые глифы. Тогда пришлось бы изменить перечисление Traversal, включив в него значение TEXTUAL_PREORDER.

Но нежелательно менять уже имеющиеся объявления. Помещение всего механизма обхода в иерархию класса Glyph затрудняет модификацию и расширение без изменения многих других классов. Механизм также трудно использовать повторно для обхода других видов структур. И еще нельзя иметь более одного активного обхода над данной структурой.

Как уже не раз бывало, наилучшее решение – инкапсулировать изменяющуюся сущность в класс. В данном случае это механизмы доступа и обхода. Допустимо ввести класс объектов, называемых итераторами, единственное назначение которых – определить разные наборы таких механизмов. Можно также воспользоваться наследованием для унификации доступа к разным структурам данных и поддержки новых видов обхода. Тогда не придется изменять интерфейсы глифов или трогать реализации существующих глифов.

Класс Iterator и его подклассы

Мы применим абстрактный класс Iterator для определения общего интерфейса доступа и обхода. Конкретные подклассы вроде ArrayIterator и ListIterator реализуют данный интерфейс для предоставления доступа к массивам и спискам, а такие подклассы, как PreorderIterator, PostorderIterator и им подобные, реализуют разные виды обходов структур. Каждый подкласс класса Iterator содержит ссылку на структуру, которую он обходит. Экземпляры подкласса инициализируются этой ссылкой при создании. На рис. 2.13 показан класс