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

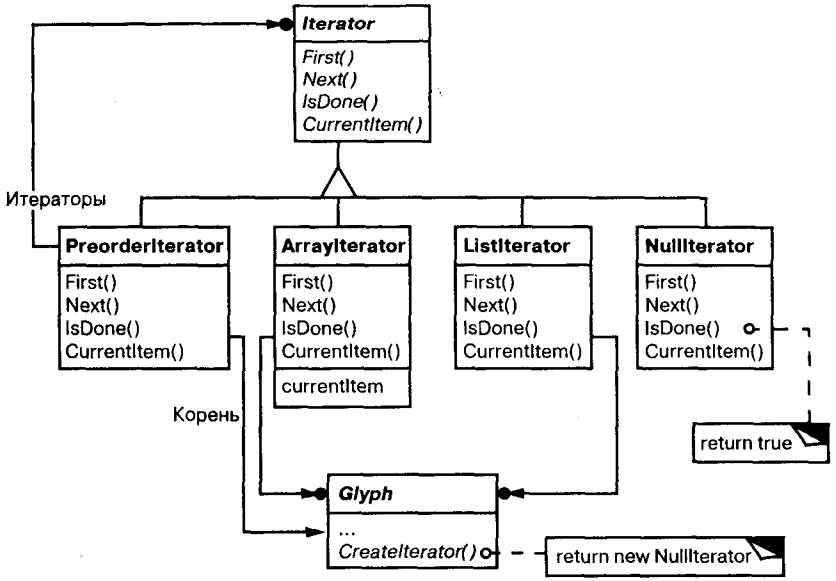

Рис. 2.13. Класс Iterator и его подклассы

`Iterator` и несколько его подклассов. Обратите внимание, что мы добавили в интерфейс класса `Glyph` абстрактную операцию `CreateIterator` для поддержки итераторов.

Интерфейс итератора предоставляет операции `First`, `Next` и `IsDone` для управления обходом. В классе `ListIterator` реализация операции `First` указывает на первый элемент списка, а `Next` перемещает итератор на следующий элемент. Операция `IsDone` возвращает признак, говорящий о том, перешел ли указатель за последний элемент списка. Операция `CurrentItem` разыменовывает итератор для возврата глифа, на который он указывает. Класс `ArrayIterator` делает то же самое с массивами глифов.

Теперь мы можем получить доступ к потомкам в структуре глифа, не зная ее представления:

```
Glyph* g;
Iterator<Glyph*>* i = g->CreateIterator();

for (i->First(); !i->IsDone(); i->Next()) {
    Glyph* child = i->CurrentItem();

    // выполнить действие с потомком
}
```

`CreateIterator` по умолчанию возвращает экземпляр `NullIterator`. `NullIterator` – это вырожденный итератор для глифов, у которых нет потомков, то есть листовых глифов. Операция `IsDone` для `NullIterator` всегда возвращает истину.

Подкласс глифа, имеющего потомков, замещает операцию `CreateIterator` так, что она возвращает экземпляр другого подкласса класса `Iterator`. Какого именно – зависит от структуры, в которой содержатся потомки. Если подкласс `Row` класса `Glyph` размещает потомков в списке, то его операция `CreateIterator` будет выглядеть примерно так:

```
Iterator<Glyph*>* Row::CreateIterator () {
    return new ListIterator<Glyph*>(_children);
}
```

Для обхода в прямом и внутреннем порядке итераторы реализуют алгоритм обхода в терминах, определенных для конкретных глифов. В обоих случаях итератору передается корневой глиф той структуры, которую нужно обойти. Итераторы вызывают `CreateIterator` для каждого глифа в этой структуре и сохраняют возвращенные итераторы в стеке.

Например, класс `PreorderIterator` получает итератор от корневого глифа, инициализирует его так, чтобы он указывал на свой первый элемент, а затем помещает в стек:

```
void PreorderIterator::First () {
    Iterator<Glyph*>* i = _root->CreateIterator();

    if (i) {
        i->First();
        _iterators.RemoveAll();
        _iterators.Push(i);
    }
}
```

`CurrentItem` должна будет просто вызвать операцию `CurrentItem` для итератора на вершине стека:

```
Glyph* PreorderIterator::CurrentItem () const {
    return
        _iterators.Size() > 0 ?
        _iterators.Top()->CurrentItem() : 0;
}
```

Операция `Next` обращается к итератору с вершины стека с тем, чтобы элемент, на который он указывает, создал свой итератор, спускаясь тем самым по структуре глифов как можно ниже (это ведь прямой порядок, не так ли?). `Next` устанавливает новый итератор так, чтобы он указывал на первый элемент в порядке обхода, и помещает его в стек. Затем `Next` проверяет последний встретившийся итератор; если его операция `IsDone` возвращает `true`, то обход текущего поддерева (или листа) закончен. В таком случае `Next` снимает итератор с вершины стека и повторяет всю последовательность действий, пока не найдет следующее не полностью обойденное дерево, если таковое существует. Если же необойденных деревьев больше нет, то мы закончили обход:

```
void PreorderIterator::Next () {
    Iterator<Glyph*>* i =
        _iterators.Top()->CurrentItem()->CreateIterator();

    i->First();
    _iterators.Push(i);

    while (
        _iterators.Size() > 0 && _iterators.Top()->IsDone()
    ) {
        delete _iterators.Pop();
        _iterators.Top()->Next();
    }
}
```

Обратите внимание, что класс `Iterator` позволяет вводить новые виды обходов, не изменяя классы глифов, – мы просто порождаем новый подкласс и добавляем новый обход так, как проделали это для `PreorderIterator`. Подклассы класса `Glyph` пользуются тем же самым интерфейсом, чтобы предоставить клиентам доступ к своим потомкам, не раскрывая внутренней структуры данных, в которой они хранятся. Поскольку итераторы сохраняют собственную копию состояния обхода, то одновременно можно иметь несколько активных итераторов для одной и той же структуры. И, хотя в нашем примере мы занимались обходом структур глифов, ничто не мешает параметризовать класс типа `PreorderIterator` типом объекта структуры. В C++ мы воспользовались бы для этого шаблонами. Тогда описанный механизм итераторов можно было бы применить для обхода других структур.

Паттерн итератор

Паттерн итератор абстрагирует описанную технику поддержки обхода структур, состоящих из объектов, и доступа к их элементам. Он применим не только к составным структурам, но и к группам, абстрагирует алгоритм обхода и экранирует клиентов от деталей внутренней структуры объектов, которые они обходят. Паттерн итератор – это еще один пример того, как инкапсуляция изменяющейся сущности помогает достичь гибкости и повторной используемости. Но все равно проблема итерации оказывается глубокой, поэтому паттерн итератор гораздо сложней, чем было рассмотрено выше.

Обход и действия, выполняемые при обходе

Итак, теперь, когда у нас есть способ обойти структуру глифов, нужно заняться проверкой правописания и расстановкой переносов. Для обоих видов анализа необходимо аккумулировать собранную во время обхода информацию.

Прежде всего следует решить, на какую часть программы возложить ответственность за выполнение анализа. Можно было бы поручить это классам `Iterator`, тем самым сделав анализ неотъемлемой частью обхода. Но решение стало бы более гибким и пригодным для повторного использования, если бы обход был отделен от действий, которые при этом выполняются. Дело в том, что для одного и того же вида обхода могут выполняться разные виды анализа. Поэтому один и тот же