```
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