```
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

набор итераторов можно было бы использовать для разных аналитических операций. Например, прямой порядок обхода применяется в разных случаях, включая проверку правописания, расстановку переносов, поиск в прямом направлении и подсчет слов.

Итак, анализ и обход следует разделить. Кому еще можно поручить анализ? Мы знаем, что разновидностей анализа достаточно много, и в каждом случае в те или иные моменты обхода будут выполняться различные действия. В зависимости от вида анализа некоторые глифы могут оказаться более важными, чем другие. При проверке правописания и расстановке переносов мы хотели бы рассматривать только символьные глифы и пропускать графические – линии, растровые изображения и т.д. Если мы занимаемся разделением цветов, то желательно было бы принимать во внимание только видимые, но никак не невидимые глифы. Таким образом, разные глифы должны просматриваться разными видами анализа.

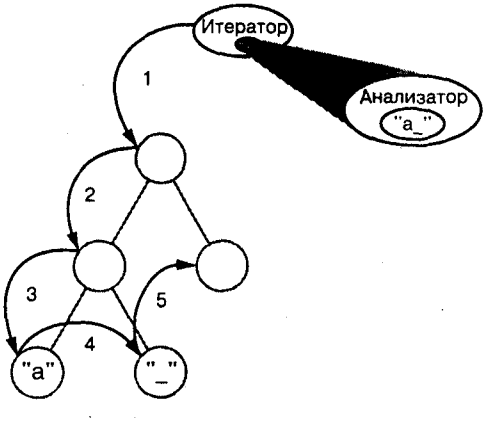
Поэтому данный вид анализа должен уметь различать глифы по их типу. Очевидное решение – встроить аналитические возможности в сами классы глифов. Тогда для каждого вида анализа мы можно было добавить одну или несколько абстрактных операций в класс `Glyph` и реализовать их в подклассах в соответствии с той ролью, которую они играют при анализе.

Однако неприятная особенность такого подхода состоит в том, что придется изменять каждый класс глифов при добавлении нового вида анализа. В некоторых случаях проблему удается сгладить: если в анализе участвует немного классов или если большинство из них выполняют анализ одним и тем же способом, то можно поместить подразумеваемую реализацию абстрактной операции прямо в класс `Glyph`. Такая операция по умолчанию будет обрабатывать наиболее распространенный случай. Тогда мы смогли бы ограничиться только изменениями класса `Glyph` и тех его подклассов, которые отклоняются от нормы.

Но, несмотря на то что реализация по умолчанию сокращает объем изменений, принципиальная проблема остается: интерфейс класса `Glyph` необходимо расширять при добавлении каждого нового вида анализа. Со временем такие операции затемнят смысл этого интерфейса. Будет трудно понять, что основная цель глифа – определить и структурировать объекты, имеющие внешнее представление и форму; в интерфейсе появится много лишних деталей.

Инкапсуляция анализа

Судя по всему, стоит инкапсулировать анализ в отдельный объект, как мы уже много раз делали прежде. Можно было бы поместить механизм конкретного вида анализа в его собственный класс, а экземпляр этого класса использовать совместно с подходящим итератором. Тогда итератор «переносил» бы этот экземпляр от одного глифа к другому, а объект выполнял бы свой анализ для каждого элемента. По мере продвижения



обхода анализатор накапливал бы определенную информацию (в данном случае – символы).

Принципиальный вопрос при таком подходе – как объект-анализатор различает виды глифов, не прибегая к проверке или приведениям типов? Мы не хотим, чтобы класс SpellingChecker включал такой псевдокод:

```
void SpellingChecker::Check (Glyph* glyph) {
    Character* c;
    Row* r;
    Image* i;

    if (c = dynamic_cast<Character*>(glyph)) {
        // анализировать символ

    } else if (r = dynamic_cast<Row*>(glyph)) {
        // анализировать потомки r

    } else if (i = dynamic_cast<Image*>(glyph)) {
        // ничего не делать
    }
}
```

Такой код опирается на специфические возможности безопасных по отношению к типам приведений. Его трудно расширять. Нужно не забыть изменить тело данной функции после любого изменения иерархии класса Glyph. В общем это как раз такой код, необходимость в котором хотелось бы устранить.

Как уйти от данного грубого подхода? Посмотрим, что произойдет, если мы добавим в класс Glyph такую абстрактную операцию:

```
void CheckMe(SpellingChecker&)
```

Определим операцию CheckMe в каждом подклассе класса Glyph следующим образом:

```
void GlyphSubclass::CheckMe (SpellingChecker& checker) {
    checker.CheckGlyphSubclass(this);
}
```

где GlyphSubclass заменяется именем подкласса глифа. Заметим, что при вызове CheckMe конкретный подкласс класса Glyph известен, ведь мы же выполняем одну из его операций. В свою очередь, в интерфейсе класса SpellingChecker есть операция типа CheckGlyphSubclass для каждого подкласса класса Glyph[1]:

```
class SpellingChecker {
public:
    SpellingChecker();
```

[1] Мы могли бы воспользоваться перегрузкой функций, чтобы присвоить этим функциям-членам одинаковые имена, поскольку их можно различить по типам параметров. Здесь мы дали им разные имена, чтобы было видно, что это все-таки разные функции, особенно при их вызове.

```
    virtual void CheckCharacter(Character*);
    virtual void CheckRow(Row*);
    virtual void CheckImage(Image*);

    // ... и так далее

    List<char*>& GetMisspellings();

protected:
    virtual bool IsMisspelled(const char*);

private:
    char _currentWord[MAX_WORD_SIZE];
    List<char*> _misspellings;
};
```

Операция проверки в классе `SpellingChecker` для глифов типа `Character` могла бы выглядеть так:

```
void SpellingChecker::CheckCharacter (Character* c) {
    const char ch = c->GetCharCode();

    if (isalpha(ch)) {
        // добавить букву к _currentWord

    } else {
        // встретилась не-буква

        if (IsMisspelled(_currentWord)) {
            // добавить _currentWord в _misspellings
            _misspellings.Append(strdup(_currentWord));
        }

        _currentWord[0] = '\0';
            // переустановить _currentWord для проверки
            // следующего слова
    }
}
```

Обратите внимание, что мы определили специальную операцию `GetCharCode` только для класса `Character`. Объект проверки правописания может работать со специфическими для подклассов операциями, не прибегая к проверке или приведению типов, а это позволяет нам трактовать некоторые объекты специальным образом.

Объект класса `CheckCharacter` накапливает буквы в буфере `_currentWord`. Когда встречается не-буква, например символ подчеркивания, этот объект вызывает операцию `IsMisspelled` для проверки орфографии слова, находящегося

в _currentWord.[1] Если слово написано неправильно, то CheckCharacter добавляет его в список слов с ошибками. Затем буфер _currentWord очищается для приема следующего слова. По завершении обхода можно добраться до списка слов с ошибками с помощью операции GetMisspellings.

Теперь логично обойти всю структуру глифов, вызывая CheckMe для каждого глифа и передавая ей объект проверки правописания в качестве аргумента. Тем самым текущий глиф для SpellingChecker идентифицируется и может продолжать проверку:

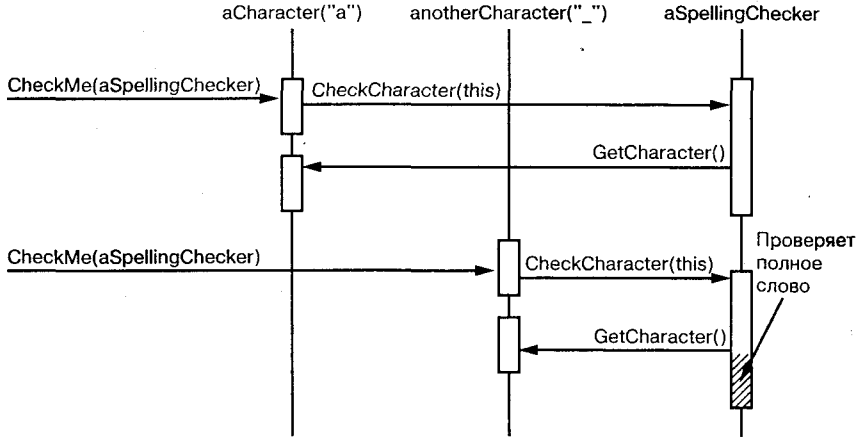
```
SpellingChecker spellingChecker;
Composition* c;

// ...

Glyph* g;
PreorderIterator i(c);

for (i.First(); !i.IsDone(); i.Next()) {
    g = i.CurrentItem();
    g->CheckMe(spellingChecker);
}
```

На следующей диаграмме показано, как взаимодействуют глифы типа Character и объект SpellingChecker.



Этот подход работает при поиске орфографических ошибок, но как он может помочь в поддержке нескольких видов анализа? Похоже, что придется добавлять операцию вроде CheckMe(SpellingChecker&) в класс Glyph и его подклассы

[1] Функция IsMisspelled реализует алгоритм проверки орфографии, детали которого мы здесь не приводим, поскольку мы сделали его независимым от дизайна Lexi. Мы можем поддержать разные алгоритмы, порождая подклассы класса SpellingChecker. Или применить для этой цели паттерн стратегия (как для форматирования в разделе 2.3).

всякий раз, как вводится новый вид анализа. Так оно и есть, если мы настаиваем на независимом классе для каждого вида анализа. Но почему бы не придать всем видам анализа одинаковый интерфейс? Это позволит нам использовать их полиморфно. И тогда мы сможем заменить специфические для конкретного вида анализа операции вроде `CheckMe(SpellingChecker&)` одной инвариантной операцией, принимающей более общий параметр.

Класс Visitor и его подклассы

Мы будем использовать термин «посетитель» для обозначения класса объектов, «посещающих» другие объекты во время обхода, дабы сделать то, что необходимо в данном контексте.[1] Тогда мы можем определить класс `Visitor`, описывающий абстрактный интерфейс для посещения глифов в структуре:

```
class Visitor {
public:
    virtual void VisitCharacter(Character*) { }
    virtual void VisitRow(Row*) { }
    virtual void VisitImage(Image*) { }

    // ... и так далее
};
```

Конкретные подклассы `Visitor` выполняют разные виды анализа. Например, можно было определить подкласс `SpellingCheckingVisitor` для проверки правописания и подкласс `HyphenationVisitor` для расстановки переносов. При этом `SpellingCheckingVisitor` был бы реализован точно так же, как мы реализовали класс `SpellingChecker` выше, только имена операций отражали бы более общий интерфейс класса `Visitor`. Так, операция `CheckCharacter` называлась бы `VisitCharacter`.

Поскольку имя `CheckMe` не подходит для посетителей, которые ничего не проверяют, мы использовали бы имя `Accept`. Аргумент этой операции тоже пришлось бы изменить на `Visitor&`, чтобы отразить тот факт, что может приниматься любой посетитель. Теперь для добавления нового вида анализа нужно лишь определить новый подкласс класса `Visitor`, а трогать классы глифов вовсе не обязательно. Мы поддержали все возможные в будущем виды анализа, добавив лишь одну операцию в класс `Glyph` и его подклассы.

О выполнении проверки правописания говорилось выше. Такой же подход будет применен для аккумулирования текста в подклассе `HyphenationVisitor`. Но после того как операция `VisitCharacter` из подкласса `HyphenationVisitor` закончила распознавание целого слова, она ведет себя по-другому. Вместо проверки орфографии применяется алгоритм расстановки переносов, чтобы определить, в каких местах можно перенести слово на другую строку (если это вообще возможно). Затем для каждой из найденных точек в структуру вставляется разделяющий

---

[1] «Посетить» – это лишь немногим более общее слово, чем «проанализировать». Оно просто предвосхищает ту терминологию, которой мы будем пользоваться при обсуждении следующего паттерна.