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.

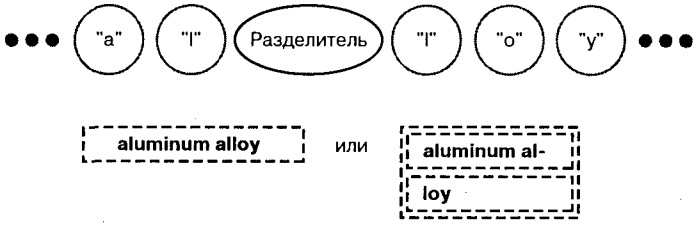
О выполнении проверки правописания говорилось выше. Такой же подход будет применен для аккумулирования текста в подклассе `HyphenationVisitor`. Но после того как операция `VisitCharacter` из подкласса `HyphenationVisitor` закончила распознавание целого слова, она ведет себя по-другому. Вместо проверки 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.

(discretionary) глиф. Разделяющие глифы являются экземплярами подкласса `Glyph` – класса `Discretionary`.

Разделяющий глиф может выглядеть по-разному в зависимости от того, является он последним символом в строке или нет. Если это последний символ, глиф выглядит как дефис, в противном случае не отображается вообще. Разделяющий глиф запрашивает у своего родителя (объекта `Row`), является ли он последним потомком, и делает это всякий раз, когда от него требуют отобразить себя или вычислить свои размеры. Стратегия форматирования трактует разделяющие глифы точно так же, как пропуски, считая их «кандидатами» на завершающий символ строки. На диаграмме ниже показано, как может выглядеть встроенный разделитель.



Паттерн посетитель

Вышеописанная процедура – пример применения паттерна посетитель. Его главными участниками являются класс `Visitor` и его подклассы. Паттерн посетитель абстрагирует метод, позволяющий иметь заранее неопределенное число видов анализа структур глифов без изменения самих классов глифов. Еще одна полезная особенность посетителей состоит в том, что их можно применять не только к таким агрегатам, как наши структуры глифов, но и к любым структурам, состоящим из объектов. Сюда входят множества, списки и даже направленные ациклические графы. Более того, классы, которые обходит посетитель, необязательно должны быть связаны друг с другом через общий родительский класс. А это значит, что посетители могут пересекать границы иерархий классов.

Важный вопрос, который надо задать себе перед применением паттерна посетитель, звучит так: «Какие иерархии классов наиболее часто будут изменяться?» Этот паттерн особенно удобен, если необходимо выполнять действия над объектами, принадлежащими классу со стабильной структурой. Добавление нового вида посетителя не требует изменять структуру класса, что особенно важно, когда класс большой. Но при каждом добавлении нового подкласса вы будете вынуждены обновить все интерфейсы посетителя с целью включить операцию `Visit...` для этого подкласса. В нашем примере это означает, что добавление подкласса `Foo` класса `Glyph` потребует изменить класс `Visitor` и все его подклассы, чтобы добавить операцию `VisitFoo`. Однако при наших проектных условиях гораздо более вероятно добавление к Lexi нового вида анализа, а не нового вида глифов. Поэтому для наших целей паттерн посетитель вполне подходит.