Глава 3

Реализация предметно-ориентированных языков

Теперь, когда вы хорошо себе представляете, что такое DSL и в чем преимущества его использования, наступило время углубиться в методы его построения. Хотя многие из этих методов различны для внутренних и внешних DSL, у них много общего. В этой главе я сконцентрируюсь на общих для внутренних и внешних предметно-ориентированных языков вопросах и перейду к подробностям в следующей главе. Я также пока что проигнорирую языковые инструментальные средства (я вернусь к ним гораздо позже).

3.1. Архитектура обработки DSL

Возможно, одна из наиболее важных вещей, о которых следует поговорить, — это как работают реализации предметно-ориентированных языков (то, что можно назвать архитектурой системы DSL; рис. 3.1).

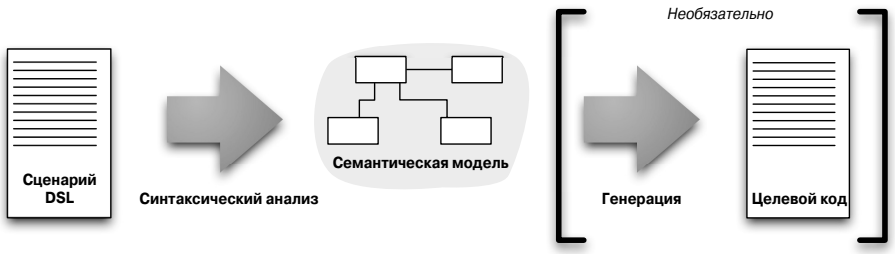

Рис. 3.1. Предпочитаемая автором архитектура обработки DSL

К настоящему времени вам, должно быть, ужасно надоело мое постоянное повторение, что DSL представляет собой тонкий слой над моделью. Говоря о модели в этом контексте, я именую ее шаблоном семантической модели (Semantic Model (171)). Ее суть в том, что все важное семантическое поведение охватывается моделью, а роль DSL — заполнить эту модель посредством этапа синтаксического анализа. Это означает, что семантическая модель играет центральную роль в представлении о DSL (почти во всей этой книге предполагается ее использование). Об альтернативах семантической модели я скажу в конце этого раздела, когда появится достаточный контекст для их обсуждения.

Будучи сторонником объектно-ориентированного программирования, я, естественно, предполагаю, что семантическая модель является объектной. Мне нравятся богатые объектные модели, в которых сочетаются данные и обработка. Но семантиче-

ская модель не обязана быть такой; она также может представлять собой только структуры данных. Хотя я всегда предпочитаю по возможности работать с объектами, использование семантической модели в форме данных лучше, чем отсутствие семантической модели вообще. Таким образом, хотя я буду считать в этой книге, что у нас имеются надлежащие поведенческие объекты, помните, что вы можете встретиться и только со структурами данных.

Во многих системах используется модель предметной области (**Domain Model** [10]) для охвата базового поведения программной системы. Часто DSL заполняет значительную часть модели предметной области. Я предпочитаю разделять понятия модели предметной области и семантической модели. Семантическая модель DSL обычно является подмножеством модели предметной области приложения, так как не все части модели предметной области наилучшим образом обрабатываются DSL. Кроме того, DSL могут быть использованы для выполнения иных задач, помимо заполнения модели предметной области (при ее наличии).

Семантическая модель — это обычная объектная модель, с которой можно работать точно так же, как и с любой иной имеющейся в вашем распоряжении объектной моделью. В случае примера с состояниями можно заполнить конечный автомат с помощью API командных запросов модели, а затем запустить его, чтобы получить требуемое поведение. В известном смысле модель независима от DSL, хотя на самом деле это близкие родственники.

(Если вы знакомы с технологиями компиляторов, у вас может возникнуть вопрос, не является ли семантическая модель тем же, что и абстрактное синтаксическое дерево. Краткий ответ — нет, это разные понятия. Я еще вернусь к этой теме в разделе “Работа синтаксического анализатора”, с. 67.)

Отделение семантической модели от DSL имеет несколько преимуществ. Основное из них то, что вы можете думать о семантике предметной области, не беспокоясь о синтаксисе DSL или синтаксическом анализаторе. Если вы вообще используете DSL, то, как правило, потому что вы работаете с чем-то довольно сложным, иначе вы бы его не использовали. А так как это что-то довольно сложное, то оно заслуживает собственной модели.

В частности, это позволяет тестировать семантическую модель путем создания объектов в модели и непосредственной работы с ними. Я могу создать много состояний и переходов и протестировать их, чтобы посмотреть, насколько корректно работают события и команды, не прибегая к синтаксическому анализу вообще. Если при этом возникли проблемы в работе конечного автомата, я могу выделить проблему в модели, не понимая, как работает синтаксический анализ.

Явная семантическая модель позволяет поддерживать несколько предметно-ориентированных языков для ее заполнения. Вы можете начать с простого внутреннего DSL, а затем добавить внешний DSL в качестве более простого для чтения альтернативного варианта. Поскольку у вас уже имеются написанные на внутреннем DSL сценарии и работающие с ним пользователи, вы можете сохранить существующий внутренний DSL и поддерживать два предметно-ориентированных языка. Поскольку у них одна семантическая модель, это не так трудно и, кроме того, помогает избежать дублирования между языками.

В продолжение темы укажу, что отдельная семантическая модель позволяет развивать модель и язык в отдельности. Если я хочу изменить модель, я могу исследовать этот вопрос без изменения DSL, добавив необходимые конструкции в предметно-ориентированный язык после того, как получу работающую модель. Я также могу экспериментировать с новым синтаксисом для DSL и просто убедиться, что он создает те же объекты в модели. Я могу сравнить два синтаксиса по заполнению семантической модели.

Во многом это разделение семантической модели и синтаксиса DSL отражает разделение модели предметной области и представления, которые мы видим в разработке корпоративного программного обеспечения. Зачастую я думаю о DSL как об еще одной форме пользовательского интерфейса.

Сравнение DSL с представлением накладывает определенные ограничения. Предметно-ориентированный язык и семантическая модель по-прежнему связаны, и при добавлении в DSL новых конструкций нужно обеспечить их поддержку в семантической модели, что зачастую означает одновременное изменение их обоих. Однако разделение означает, что я могу анализировать семантические вопросы отдельно от вопросов синтаксического анализа, что упрощает мою задачу.

Разница между внутренними и внешними DSL заключается в стадии синтаксического анализа — как в том, что именно анализируется, так и в том, как это делается. Оба стиля DSL будут производить семантическую модель одного вида, и, как я уже говорил ранее, нет никаких причин, чтобы не позволить одной семантической модели быть заполненной и внутренними, и внешними DSL. По сути, это именно то, что я сделал, когда программировал пример конечного автомата (когда у меня было несколько DSL, заполняющих одну семантическую модель).

В случае внешнего DSL имеется четкое разделение между сценариями DSL, синтаксическим анализатором и семантической моделью. Сценарии DSL пишутся на отдельном языке; синтаксический анализатор читает их и заполняет семантическую модель. В случае внутреннего DSL гораздо проще все это перепутать. Я сторонник явного слоя объектов (Expression Builder (349)), работа которых заключается в предоставлении необходимых свободных интерфейсов, действующих в качестве языка. Затем сценарии DSL выполняются путем применения методов построителя выражений, которые заполняют семантическую модель. Таким образом, в случае внутреннего DSL синтаксический анализ сценариев DSL выполняется комбинацией синтаксического анализатора базового языка и построителей выражений.

В этой связи возникает интересный момент — вам может показаться немного странным использование термина “синтаксический анализ” в контексте внутреннего DSL. Признаюсь, мне это тоже не совсем нравится. Но я пришел к выводу, что проводить параллели между внутренним и внешним DSL весьма полезно. При использовании традиционных методов синтаксического анализа вы получаете поток текста, преобразуете его в дерево синтаксического анализа и обрабатываете это дерево для получения полезного выхода. При синтаксическом анализе внутреннего DSL ваш вход представляет собой ряд вызовов функций. Вы также организуете их в иерархию (обычно неявно в стеке) с целью получения полезного выхода.

Еще одним фактором в пользу использования здесь термина “синтаксический анализ” является то, что в ряде случаев отсутствует непосредственная обработка текста. Во внутреннем DSL текст обрабатывает синтаксический анализатор базового языка, а процессор DSL обрабатывает дальнейшие языковые конструкции. Но то же самое происходит и с предметно-ориентированными языками, использующими XML: синтаксический анализатор XML преобразует текст в элементы XML, после чего с ними работает процессор DSL.

Сейчас стоит вернуться к различиям между внутренними и внешними DSL. Использованное ранее отличие — используется ли для написания сценария базовый язык программирования приложения — как правило, верно, но не на все 100%. Например, рассмотрим случай, когда приложение написано на Java, а сценарии DSL — на JRuby. В этом случае я бы классифицировал DSL как внутренний, по крайней мере по тому признаку, что вам нужно использовать приемы из части этой книги, посвященной внутренним DSL.

Истинное различие между ними состоит в том, что внутренние DSL пишутся на выполняемом языке, а синтаксический анализ осуществляется путем выполнения DSL в этом языке. Как в JRuby, так и в XML, DSL встраивается в синтаксис носителя, но при этом мы выполняем код JRuby, но только читаем структуры XML-данных. Бо́льшую часть времени внутренний DSL выполняется в основном языке приложения, так что это определение в общем случае оказывается более полезным.

Теперь, когда у нас имеется семантическая модель, необходимо заставить ее делать то, что нам нужно. В примере с конечным автоматом это задача управления системой безопасности. Есть два основных способа, как это сделать. Самый простой и обычно наилучший — просто выполнить семантическую модель. Семантическая модель представляет собой код и как таковая может быть запущена и выполнить все необходимые действия.

Другой вариант заключается в использовании генерации кода. Генерация кода означает, что мы создаем код, который отдельно компилируется и запускается. В некоторых кругах генерация кода рассматривается как неотъемлемая часть DSL. Я сталкивался с мнением, что при любой работе с DSL необходимо генерировать код. В редких случаях, когда я вижу кого-то, говорящего или пишущего о генераторах синтаксических анализаторов (Parser Generator (277)), они неизбежно говорят о генерации кода. Однако DSL не присуща необходимость генерации кода! В большинстве случаев лучше просто выполнить семантическую модель.

Веским доводом в пользу генерации кода является ситуация, когда запуск модели и синтаксический анализ DSL должны выполняться в разных местах. Хорошим примером этого является выполнение кода в среде с ограниченным выбором языка, например в случае ограниченных аппаратных средств или в реляционной базе данных. Вы не хотите запускать синтаксический анализатор в небольшом устройстве или в SQL, поэтому реализуете синтаксический анализатор и семантическую модель на более подходящем языке и генерируете код C или SQL. Еще один случай — при наличии зависимостей синтаксического анализатора от библиотек, которых не должно быть в производственной среде. Это достаточно распространенная ситуация, если вы используете сложный инструментарий для своего DSL, в частности именно поэтому языковые инструментальные средства, как правило, прибегают к генерации кода.

В этих ситуациях все равно полезно иметь семантическую модель, которая может работать без генерации кода. Запустив семантическую модель, можно экспериментировать с выполнением DSL без необходимости разбираться в процессе генерации кода. Вы можете протестировать синтаксический анализ и семантику без генерации кода, что зачастую помогает выполнять тесты быстрее и для отдельных задач. Так вы можете выполнить проверки семантической модели и выявить ее ошибки до генерации кода.

Еще одним аргументом в пользу генерации кода, даже в среде, в которой можно ограничиться непосредственной интерпретацией семантической модели, является то, что многие разработчики считают логику богатых семантических моделей сложной для понимания. Создание кода на основе семантической модели упрощает понимание и может быть решающим моментом для команды из не очень опытных и способных разработчиков.

Но самое важное, о чем следует помнить, — это то, что генерация кода не является обязательным компонентом ландшафта DSL. Это одна из тех вещей, которые очень важны, когда в них есть нужда, но бо́льшую часть времени можно легко обходиться и без них. Генераторы кода похожи на снегоступы: зимой при большом количестве снега они необходимы, но никто не носит их в жаркий летний день.

Генерируя код, мы видим еще одно преимущество использования семантической модели: она отделяет генераторы кода от синтаксических анализаторов. Я могу написать генератор кода, ничего не зная о процессе синтаксического анализа, и независимо его про-

верить. Одно это стоит разработки семантической модели. Кроме того, в случае необходимости многоцелевой генерации кода модель упрощает ее поддержку.

3.2. Работа синтаксического анализатора

Итак, различия между внутренними и внешними DSL кроются в их синтаксическом анализе. И в самом деле: между ними много различий, но есть и очень много общего.

Одно из наиболее важных сходств состоит в том, что синтаксический анализ является строго иерархической операцией. Анализируя текст, мы организуем его части в виде древовидной структуры. Рассмотрим простую структуру списка событий конечного автомата. При использовании внешнего синтаксиса это выглядит примерно так.

```
events
  doorClosed D1CL
  drawerOpened D2OP
end
```

Мы можем рассматривать эту составную конструкцию как список событий, каждое из которых имеет имя и код.

Рассмотрим аналог на внутреннем DSL в Ruby.

```
event :doorClosed "D1CL"
event :drawerOpened "D2OP"
```

Здесь нет явного понятия списка, но каждое событие по-прежнему представляет собой иерархию: событие содержит символьное имя и строку кода.

Всякий раз, когда вы смотрите на сценарий, подобный приведенному, вы можете представить его в виде иерархии. Такая иерархии называется **синтаксическим деревом** (или деревом разбора) (рис. 3.2). Любой сценарий может быть преобразован в множество потенциальных синтаксических деревьев — в зависимости от того, каким образом вы будете его разбирать. Синтаксическое дерево является гораздо более полезным представлением сценария, чем слова, поскольку с ним можно работать различными способами, поразному обходя дерево.

При использовании семантической модели (Semantic Model (171)) мы получаем синтаксическое дерево и преобразуем его в семантическую модель. Если почитать материалы языковых сообществ, то в них можно заметить, что основной упор сделан на синтаксические деревья — либо они выполняются непосредственно, либо на их основе генерируется код. Фактически синтаксическое дерево можно использовать как семантическую модель. Впрочем, по возможности я бы этого не делал, потому что синтаксическое дерево очень тесно связано с синтаксисом сценария DSL и, таким образом, привязывает обработку DSL к его синтаксису.

Я говорю о синтаксическом дереве как если бы это была материальная структура данных в вашей системе наподобие модели XML DOM. Зачастую это вовсе не так. По большей части синтаксическое дерево формируется в стеке вызовов и обрабатывается при его обходе. В результате вы никогда не видите все дерево в целом, а видите только обрабатываемые в настоящий момент ветви (аналогично способу работы SAX с XML документом). Несмотря на это, как правило, полезно рассматривать такое призрачное синтаксическое дерево как скрывающееся в тени стека вызовов. Для внутренних DSL это дерево формируется аргументами в вызове функции (Nested Function (361)) и вложенными объектами (Method Chaining (375)). Иногда вы не видите строгой иерархии и должны ее имитировать (Function Sequence (357)) с иерархией, моделируемой с помощью переменных контекста (Context Variable (187)). Синтаксическое дерево может быть призрачным, но при

этом оно остается полезным инструментом. Использование внешнего DSL приводит к более явному синтаксическому дереву; более того, иногда вы действительно создаете структуру данных полномасштабного синтаксического дерева (Tree Construction (289)). Но даже внешние DSL обычно обрабатывают синтаксические деревья, формируя и отсекая ветви в стеке вызовов. (Выше я сослался на несколько моделей, которые еще не были описаны. Вы можете спокойно игнорировать их при первом чтении, но позже эти ссылки вам пригодятся.)

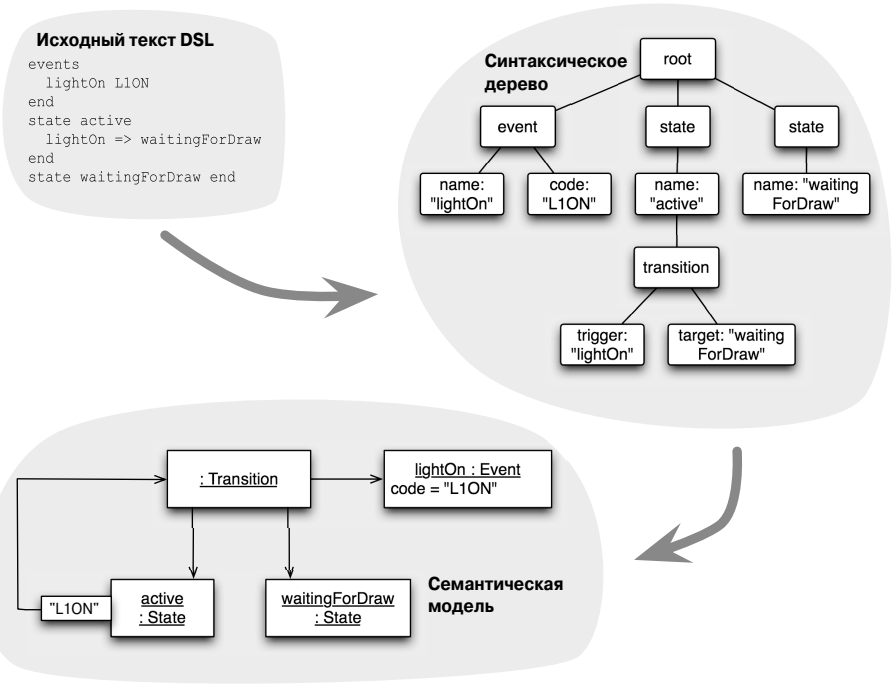

Рис. 3.2. Синтаксическое дерево и семантическая модель обычно являются различными представлениями сценария DSL

3.3. Грамматики, синтаксис и семантики

В работе с синтаксисом языка важным инструментом является грамматика. **Грамматика** представляет собой набор правил, которые описывают, как текстовый поток превращается в синтаксическое дерево. Большинство программистов сталкиваются с грамматиками в тот или иной момент своей деятельности, так как они часто используются для описания языков программирования, с которыми мы все работаем. Грамматика состоит из списка продукций (правил вывода), в котором каждое правило имеет некий заголовок (нетерминал) и тело (инструкцию вывода). Таким образом, грамматика для инструкции сложения может выглядеть как `additionStatement := number '+' number`. Эта продукция говорит о том, что если мы видим предложение языка `5+3`, синтаксический анализатор может распознать его как инструкцию сложения. Правила ссылаются одно на другое, так что у нас должно быть правило, которое говорит о том, как распознать корректное число. Из таких правил и составляется грамматика языка.

Важно понимать, что язык может иметь несколько определяющих его грамматик. Не существует *единственной* грамматики языка. Грамматика определяет структуру синтаксического дерева, которое генерируется для данного языка, и мы можем распознать несколько синтаксических деревьев для некоторого фрагмента исходного текста на этом языке. Грамматика просто определяет одну из форм синтаксического дерева; выбранные вами фактическая грамматика и синтаксическое дерево будут зависеть от многих факторов, включая особенности языка, с которым вы работаете, и то, как вы намерены обрабатывать синтаксическое дерево.

Грамматика также определяет только синтаксис языка, т.е. как он представляется в виде синтаксического дерева. Она ничего не говорит нам о его семантике, т.е. о том, что означает то или иное выражение. В зависимости от контекста `5+3` может означать и `8`, и `53`; синтаксис при этом один и тот же, а семантики различны. В случае семантической модели (Semantic Model (171)) определение семантики сводится к тому, как мы заполняем семантическую модель из синтаксического дерева и что мы делаем с этой семантической моделью. В частности, можно сказать, что если два выражения дают одну и ту же структуру в семантической модели, то они имеют одну и ту же семантику, даже если их синтаксисы различны.

Используя внешний DSL, в частности при синтаксически управляемой трансляции (Syntax-Directed Translation (229)), вы, вероятно, явно применяете грамматику для построения синтаксического анализатора. В случае внутреннего DSL явной грамматики нет, но все равно полезно думать о своем предметно-ориентированном языке в терминах грамматики. Эта грамматика помогает выбрать, какой из внутренних шаблонов DSL *стоит* использовать.

Рассматривая внутренние DSL, мы сталкиваемся с еще одной специфичной для них сложностью — проблема заключается в наличии двух анализирующих проходов и, таким образом, участии двух грамматик. Первый синтаксический анализ — самого базового языка программирования, который, очевидно, зависит от базовой грамматики. Этот анализ создает выполняемые инструкции базового языка. DSL-часть базового языка при выполнении будет создавать "призрачные" синтаксические деревья DSL в стеке вызовов. И только в ходе этого второго синтаксического анализа в игру вступает условная грамматика DSL.

3.4. Анализ данных

В процессе работы анализатор должен хранить различные данные, связанные с проводимым анализом. Эти данные могут представлять собой полное синтаксическое дерево, но в основном это некоторая другая информация. И даже если хранится полное синтаксическое дерево, есть и множество других данных, которые необходимо хранить для эффективной и корректной работы.

Анализ по своей природе представляет собой обход дерева, и всякий раз, когда вы обрабатываете часть сценария DSL, у вас имеются некоторые сведения о контексте, связанном с обрабатываемой вами ветвью синтаксического дерева. Однако зачастую вам нужна и информация, находящаяся за пределами этой ветви. Давайте вновь рассмотрим фрагмент из примера конечного автомата.

```
commands
  unlockDoor D1UL
end

state idle
  actions {unlockDoor}
end
```

Здесь мы видим распространенную ситуацию: команда определяется в одной части языка, а используется — в другой. Когда команда используется как часть действий состояния, мы находимся в ветви синтаксического дерева, отличной от той, где была определена эта команда. Если единственное представление синтаксического дерева находится в стеке вызовов, то определение команды к настоящему времени уже отсутствует. Таким образом, чтобы можно было разрешить ссылку в инструкции действия, следует хранить объект команды для последующего использования.

Для того чтобы сделать это, мы используем таблицу символов (Symbol Table (177)), которая является, по существу, словарем, ключ которого — идентификатор `unlockDoor`, а значение — объект, представляющий команду в нашем синтаксическом анализе. Обрабатывая текст `unlockDoor D1UL`, мы создаем объект для хранения соответствующих данных и вносим его в таблицу символов с ключом `unlockDoor`. Этот объект может быть объектом семантической модели для команды или промежуточным объектом, локальным для синтаксического дерева. Позже, обрабатывая действия `{unlockDoor}`, мы находим объект с помощью таблицы символов для выяснения взаимосвязей между состоянием и его действиями. Таблица символов, таким образом, является важнейшим инструментом для создания перекрестных ссылок. Если вы действительно создаете полное синтаксическое дерево во время синтаксического анализа, теоретически можно обойтись без таблицы символов, хотя обычно она все равно остается полезной конструкцией, упрощающей сборку программы воедино (рис. 3.3).

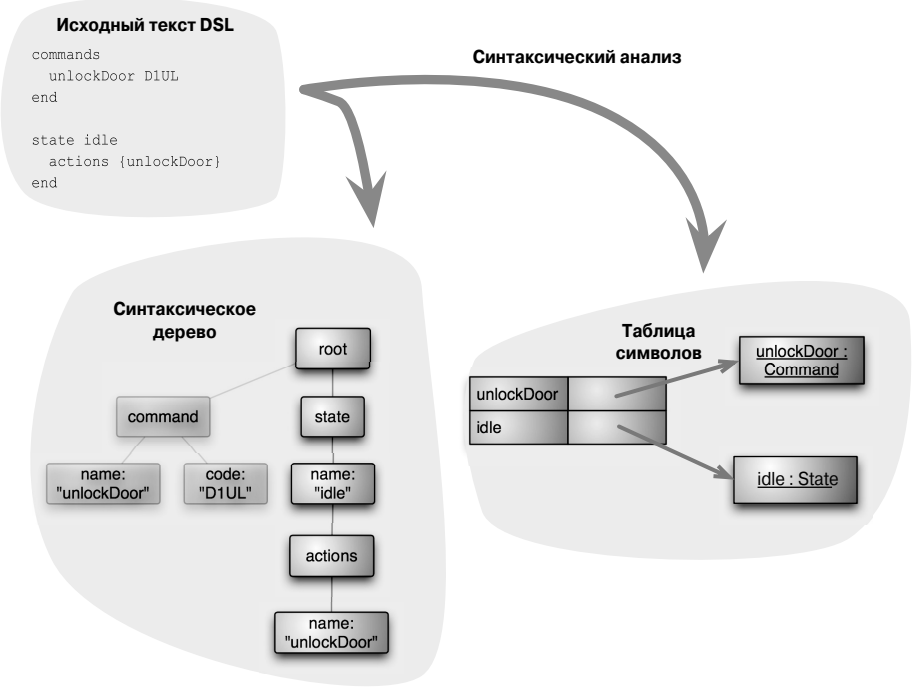

Рис. 3.3. Синтаксический анализ создает как синтаксическое дерево, так и таблицу символов

В конце этого раздела я более подробно остановлюсь на паре шаблонов. Я упоминаю их здесь, потому что они используются и во внутренних, и во внешних DSL, так что это

вполне подходящее место для такого упоминания, хотя в большей части этой главы рассматриваются вопросы на более высоком уровне.

В процессе синтаксического анализа необходимо сохранять его результаты. Иногда все результаты можно внести в таблицу символов, иногда большое количество информации можно хранить в стеке вызовов, а иногда синтаксическому анализатору требуются дополнительные структуры данных. Во всех этих случаях наиболее очевидные действия — создание объектов семантической модели (Semantic Model (171)). Зачастую, однако, придется создавать промежуточные объекты из-за невозможности создания объектов семантической модели до некоторого более позднего момента синтаксического анализа. Типичным примером такого промежуточного объекта является построитель конструкции (Construction Builder (191)), который представляет собой объект, собирающий все данные для объекта семантической модели. Это полезно, когда ваш объект семантической модели после создания имеет только данные для чтения, но в процессе синтаксического анализа для него накапливаются дополнительные данные. Построитель имеет те же поля, что и объект семантической модели, но делает их доступными для чтения и записи, так что в них можно сохранять информацию. Как только будут готовы все необходимые данные, можно будет создать объект семантической модели. Использование построителей усложняет анализатор, но я предпочитаю этот путь изменению семантической модели для отказа от преимуществ свойств, доступных только для чтения.

В действительности иногда можно отложить создание объектов семантической модели до окончания обработки всего сценария DSL. В этом случае синтаксический анализ имеет различные фазы: во-первых, чтение сценария DSL и создание промежуточных данных синтаксического анализа; во-вторых, проход по этим промежуточным данным и заполнение семантической модели. Выбор, что именно предстоит сделать в ходе обработки текста, а что отложить, как правило, зависит от того, как должна быть заполнена семантическая модель.

Способ синтаксического анализа выражения часто зависит от контекста. Рассмотрим следующий текст.

```
state idle
  actions {unlockDoor}
end

state unlockedPanel
  actions {lockDoor}
end
```

Обрабатывая действия `{lockDoor}`, важно знать, что при этом мы находимся в контексте состояния `unlockedPanel`, а не простаиваем. Часто этот контекст получается в процессе построения и обхода синтаксического дерева синтаксическим анализатором, но во многих случаях его трудно получить. Если мы не можем найти контекст путем изучения синтаксического дерева, то хорошим решением может быть хранение контекста, в данной ситуации — текущего состояния, в переменной. Я называю этот вид переменной контекста (Context Variable (187)). Такая переменная контекста, как и таблица символов, может хранить объект семантической модели или некоторый промежуточный объект.

Хотя переменная контекста представляет собой очень простой в использовании инструмент, обычно я все же предпочитаю избегать их применения, насколько это возможно. Следовать синтаксическому анализу кода легче, если читать его без необходимости мысленно манипулировать переменными контекста (так же, как большое количество изменяемых переменных усложняют отслеживание процедурного кода). Конечно, есть моменты, когда невозможно избежать использования переменных контекста, но я предпочитаю их избегать.

3.5. Макросы

Макросы (Macros (195)) являются инструментом, который может использоваться как с внутренними, так и с внешними DSL. В свое время они применялись довольно широко, но сегодня стали существенно менее распространенными. В большинстве контекстов я предлагаю обходиться без них, но иногда они все же полезны, так что следует выяснить, как они работают и когда их можно использовать.

Макросы бывают двух видов: текстовые и синтаксические. Проще всего понять текстовые макросы, которые позволяют заменить какой-нибудь текст некоторым другим текстом. Хороший пример, где они могут оказаться удобными, — указание цвета в CSS-файле. В CSS цвета указываются с помощью числовых кодов, таких как `#FFB595` (кроме некоторых фиксированных именованных цветов). Такой код не слишком значим, но, что еще хуже, используя один и тот же цвет в разных местах, следует везде повторять его код. Это плохо, как и любой иной вид дублирования кода. Было бы лучше дать такому цвету имя, значимое в данном контексте, наподобие `MEDIUM_SHADE`, и в некотором одном месте определить, что `MEDIUM_SHADE` представляет собой `#FFB595`.

Хотя язык CSS, по крайней мере в настоящее время, не позволяет это сделать, для обработки таких ситуаций можно использовать макропроцессор. Для этого просто создайте файл, который представляет собой ваш CSS файл, но с применением `MEDIUM_SHADE` везде, где нужен соответствующий цвет. Затем макропроцессор выполнит простую замену текста `MEDIUM_SHADE` текстом `#FFB595`.

Это очень простая форма обработки макросов; чаще используются макросы, в которых можно использовать параметры. Классическим примером является препроцессор C, в котором можно определить, например, макрос для замены всех `sqr(x)` на `x*x`.

Макросы предоставляют богатые возможности для создания DSL как в рамках базового языка (как препроцессор C), так и в виде отдельных файлов, преобразуемых в базовый язык. Недостатком является ряд проблем при применении макросов, которые делают их трудно используемыми на практике. В результате текстовые макросы в значительной степени потеряли привлекательность, и большинство специалистов вроде меня советуют их не использовать.

Синтаксические макросы также выполняют замены, но работают с синтаксически корректными элементами базового языка, преобразуя выражения из одного вида в другой. Самый известный своим интенсивным использованием синтаксических макросов — язык Lisp; хорошо также известен своими шаблонами C++. Использование синтаксических макросов для DSL является основной технологией написания внутренних DSL в Lisp, но дело в том, что вы можете применять синтаксические макросы только в том языке, который их поддерживает. В данной книге о них сказано немного, но это связано с тем, что синтаксические макросы поддерживаются лишь несколькими языками.

3.6. Тестирование DSL

За последние два десятилетия я существенно углубился в тему тестирования. Я большой поклонник разработки на основе тестирования (test-driven development [4]) и других подобных методов, которые выводят тестирование в авангард программирования. В результате я не могу говорить о DSL, не задумываясь об их тестировании.

В случае DSL тестирование можно разделить на три отдельные области: тестирование семантической модели (Semantic Model (171)), тестирование синтаксического анализатора и тестирование сценариев.

3.6.1. Тестирование семантической модели

Первая область тестирования DSL — тестирование семантической модели (Semantic Model (171)). Эти тесты должны гарантировать корректность поведения семантической модели, т.е. то, что при ее выполнении она дает верный вывод, зависящий от того, что помещается в модель. Это стандартная практика тестирования, такая же, как и при использовании любых других наборов объектов. Для проведения такого тестирования сам DSL не нужен — модель можно заполнить с помощью базового интерфейса самой модели. Это хорошо, так как позволяет тестировать модели независимо от DSL и синтаксических анализаторов.

Позвольте мне проиллюстрировать сказанное на примере контроллера системы безопасности. В данном случае это семантическая модель конечного автомата. Я могу протестировать семантическую модель путем ее наполнения с помощью кода с применением API командных запросов, как это делалось в главе 1, "Вводный пример", на с. 34, где не требовалось наличие какого-либо DSL.

```
@Test
public void event_causes_transition() {
  State idle = new State("idle");
  StateMachine machine = new StateMachine(idle);
  Event cause = new Event("cause", "EV01");
  State target = new State("target");
  idle.addTransition(cause, target);
  Controller controller =
    new Controller(machine, new CommandChannel());
  controller.handle("EV01");
  assertEquals(target, controller.getCurrentState());
}
```

В приведенном выше коде показано, что я могу тестировать семантическую модель совершенно отдельно. Однако следует заметить, что реальный код теста в данном случае должен быть существенно бо́льшим, лучше организованным и разделенным.

Вот несколько способов достижения наилучшего разделения подобного кода. Во-первых, можно создать много маленьких конечных автоматов с минимальной функциональностью для тестирования различных возможностей семантической модели. Все, что нужно для того, чтобы убедиться, что события запускают переходы, — это простой конечный автомат с состоянием покоя и двумя исходящими переходами в другие состояния.

```
class TransitionTester...
  State idle, a, b;
  Event trigger_a, trigger_b, unknown;

  protected StateMachine createMachine() {
    idle = new State("idle");
    StateMachine result = new StateMachine(idle);
    trigger_a = new Event("trigger_a", "TRGA");
    trigger_b = new Event("trigger_b", "TRGB");
    unknown = new Event("Unknown", "UNKN");
    a = new State("a");
    b = new State("b");
    idle.addTransition(trigger_a, a);
    idle.addTransition(trigger_b, b);
    return result;
  }
```

Для того чтобы протестировать команды, достаточно и меньшего автомата с одним состоянием, достижимым из состояния покоя.

```
class CommandTester...
  Command commenceEarthquake =
    new Command("Commence Earthquake", "EQST");
  State idle = new State("idle");
  State second = new State("second");
  Event trigger = new Event("trigger", "TGGR");

  protected StateMachine createMachine() {
    second.addAction(commenceEarthquake);
    idle.addTransition(trigger, second);
    return new StateMachine(idle);
  }
```

Все эти разные тесты могут быть выполнены единообразно. Я могу поступить проще, обеспечив им общий суперкласс. Первое, что предоставляет этот класс, — возможность настройки общих устройств, в данном случае — инициализации контроллера и командного канала к прилагаемому конечному автомату.

```
class AbstractStateTesterLib...
  protected
    CommandChannel commandChannel = new CommandChannel();
  protected StateMachine machine;
  protected Controller controller;

  @Before
  public void setup() {
    machine = createMachine();
    controller = new Controller(machine, commandChannel);
  }

  abstract protected StateMachine createMachine();
```

Теперь можно написать тесты для запуска событий контроллера и проверки состояний.

```
class TransitionTester...
  @Test
  public void event_causes_transition() {
    fire(trigger_a);
    assertCurrentState(a);
  }
  @Test
  public void event_without_transition_is_ignored() {
    fire(unknown);
    assertCurrentState(idle);
  }

class AbstractStateTesterLib...
  //-------- Вспомогательные методы -------------
  protected void fire(Event e) {
    controller.handle(e.getCode());
  }
  //------- Пользовательские проверки -----------
  protected void assertCurrentState(State s) {
    assertEquals(s, controller.getCurrentState());
  }
```

Суперкласс предоставляет *вспомогательные тестовые методы* (Test Utility Methods [20]) и *пользовательские проверки* (Custom Assertions [20]) для упрощения чтения кода тестов.

Альтернативный подход к тестированию семантической модели заключается в заполнении большей модели, демонстрирующей многие возможности, и в выполнении над

ней ряда тестов. В нашем случае в качестве тестового устройства можно использовать контроллер мисс Грант.

```
class ModelTest...
  private Event doorClosed, drawerOpened, lightOn,
                doorOpened, panelClosed;
  private State activeState, waitingForLightState,
                unlockedPanelState, idle,
                waitingForDrawerState;
  private Command unlockPanelCmd, lockDoorCmd,
                lockPanelCmd, unlockDoorCmd;
  private CommandChannel channel = new CommandChannel();
  private Controller con;
  private StateMachine machine;

@Before
public void setup() {
  doorClosed = new Event("doorClosed", "D1CL");
  drawerOpened = new Event("drawerOpened", "D2OP");
  lightOn = new Event("lightOn", "L1ON");
  doorOpened = new Event("doorOpened", "D1OP");
  panelClosed = new Event("panelClosed", "PNCL");
  unlockPanelCmd = new Command("unlockPanel", "PNUL");
  lockPanelCmd = new Command("lockPanel", "PNLK");
  lockDoorCmd = new Command("lockDoor", "D1LK");
  unlockDoorCmd = new Command("unlockDoor", "D1UL");

  idle = new State("idle");
  activeState = new State("active");
  waitingForLightState = new State("waitingForLight");
  waitingForDrawerState = new State("waitingForDrawer");
  unlockedPanelState = new State("unlockedPanel");

  machine = new StateMachine(idle);

  idle.addTransition(doorClosed, activeState);
  idle.addAction(unlockDoorCmd);
  idle.addAction(lockPanelCmd);

  activeState.addTransition(drawerOpened,
                            waitingForLightState);
  activeState.addTransition(lightOn,
                            waitingForDrawerState);

  waitingForLightState.addTransition(lightOn,
                                     unlockedPanelState);
  waitingForDrawerState.addTransition(drawerOpened,
                                      unlockedPanelState);

  unlockedPanelState.addAction(unlockPanelCmd);
  unlockedPanelState.addAction(lockDoorCmd);
  unlockedPanelState.addTransition(panelClosed, idle);

  machine.addResetEvents(doorOpened);
  con = new Controller(machine, channel);
  channel.clearHistory();
}

@Test
public void event_causes_state_change() {
  fire(doorClosed);
```

```
  assertCurrentState(activeState);
}

@Test
public void ignore_event_if_no_transition() {
  fire(drawerOpened);
  assertCurrentState(idle);
}
```

Здесь я вновь наполняю семантическую модель с помощью ее собственного интерфейса командных запросов. Поскольку в данном случае тестовые устройства более сложные, я могу упростить тестирующий код, применив DSL для создания устройств. Я могу сделать это, если у меня есть тесты для синтаксического анализатора.

3.6.2. Тестирование синтаксического анализатора

При использовании семантической модели (Semantic Model (171)) работа синтаксического анализатора состоит в ее наполнении. Поэтому наше тестирование синтаксического анализатора заключается в написании небольших фрагментов DSL и проверке того факта, что они создают корректные структуры в семантической модели.

```
@Test
public void loads_states_with_transition() {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end " +
    "state target end ";
  StateMachine actual = StateMachineLoader.loadString(code);

  State idle = actual.getState("idle");
  State target = actual.getState("target");
  assertTrue(idle.hasTransition("TGGR"));
  assertEquals(idle.targetState("TGGR"), target);
}
```

Работать с семантической моделью так, как показано здесь, довольно неудобно, и это может привести к нарушению инкапсуляции объектов в семантической модели. Поэтому другой способ тестирования выхода синтаксического анализатора состоит в определении и использовании методов для сравнения семантических моделей.

```
@Test
public void loads_states_with_transition_using_compare() {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end " +
    "state target end ";
  StateMachine actual = StateMachineLoader.loadString(code);

  State idle = new State("idle");
  State target = new State("target");
  Event trigger = new Event("trigger", "TGGR");
  idle.addTransition(trigger, target);
  StateMachine expected = new StateMachine(idle);

  assertEquivalentMachines(expected, actual);
}
```

Проверка равенства сложных структур должна включать нечто большее, чем предполагает простой оператор равенства. Кроме того, при проверке недостаточно простого логического значения, поскольку нужно точно знать, в чем именно состоит различие между объектами. В результате в моем сравнении используется шаблон уведомления (Notification (205)).

```
class StateMachine...
  public Notification probeEquivalence(StateMachine other) {
    Notification result = new Notification();
    probeEquivalence(other, result);
    return result;
  }

  private void probeEquivalence(StateMachine other,
                                Notification note) {
    for (State s : getStates()) {
      State otherState = other.getState(s.getName());
      if (null == otherState)
        note.error("missing state: %s", s.getName()) ;
      else s.probeEquivalence(otherState, note);
    }
    for (State s : other.getStates())
      if (null == getState(s.getName()))
        note.error("extra state: %s", s.getName());
    for (Event e : getResetEvents()) {
      if (!other.getResetEvents().contains(e))
        note.error("missing reset event: %s", e.getName());
    }
    for (Event e : other.getResetEvents()) {
      if (!getResetEvents().contains(e))
        note.error("extra reset event: %s", e.getName());
    }
  }

class State...
  void probeEquivalence(State other, Notification note) {
    assert name.equals(other.name);
    probeEquivalentTransitions(other, note);
    probeEquivalentActions(other, note);
  }

  private void probeEquivalentActions(State other,
                                      Notification note) {
    if (!actions.equals(other.actions))
      note.error("%s has different actions %s vs %s",
                 name, actions, other.actions);
  }

  private
    void probeEquivalentTransitions(State other,
                                    Notification note) {
    for (Transition t : transitions.values())
      t.probeEquivalent(
        other.transitions.get(t.getEventCode()), note);
    for (Transition t : other.transitions.values())
      if (!this.transitions.containsKey(t.getEventCode()))
        note.error("%s has extra transition with %s", name,
                   t.getTrigger());
  }
```

В данном тесте выполняется обход объектов семантической модели с записью всех отличий в объект уведомления. Таким образом выявляются все различия, а не только первое. Затем выполняется проверка, имеются ли в уведомлении какие-то данные об ошибках.

```
class AntlrLoaderTest...
  private void assertEquivalentMachines(StateMachine left,
                                        StateMachine right){
    assertNotificationOk(left.probeEquivalence(right));
    assertNotificationOk(right.probeEquivalence(left));
  }

  private void assertNotificationOk(Notification n) {
    assertTrue(n.report(), n.isOk());
  }

class Notification...
  public boolean isOk() {return errors.isEmpty();}
```

Можете считать меня параноиком, проверяющим эквивалентность в обоих направлениях, но так поступаю не только я.

Проверка некорректных входных данных

Только что рассмотренные тесты положительны в том плане, что они гарантируют, что корректный входной текст DSL создает правильные структуры в семантической модели (Semantic Model (171)). Другая категория тестов — отрицательные тесты, которые определяют, что произойдет при неверном входном тексте DSL. Это относится ко всей области обработки ошибок и диагностики в целом. И хотя она выходит за рамки данной книги, я не могу не упомянуть о тестах некорректных входных данных.

Суть такого тестирования состоит в подаче на вход синтаксического анализатора неверной информации различных видов. При первом выполнении такого рода тестов просто интересна реакция синтаксического анализатора. Чаще всего получается непонятная, но критичная ошибка. В зависимости от того, какую диагностическую поддержку необходимо предоставить с DSL, этого может оказаться достаточно. Гораздо хуже, если в случае некорректного исходного текста он будет проанализирован без сообщений о каких-либо ошибках. Это нарушило бы принцип “быстрой ошибки”, заключающийся в том, что все ошибки должны выявляться на как можно более ранней стадии, а сообщения о них должны быть такими, чтобы не заметить их было бы невозможно. Если вы заполните модель недопустимым состоянием и у вас нет соответствующих проверок, то об этой проблеме вы узнаете намного позднее. Наличие расстояния между исходной ошибкой (загрузка неверной входной информации) и некорректной работой в дальнейшем затрудняет поиск ошибок, и чем больше это расстояние, тем сложнее поиск.

Мой пример конечного автомата включает минимальную обработку ошибок, которая, увы, свойственна всем примерам из данной книги. Один из примеров синтаксического анализатора, впрочем, будет проверен таким образом, просто чтобы посмотреть, что из этого выйдет.

```
@Test public void targetStateNotDeclaredNoAssert () {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end ";
  StateMachine actual = StateMachineLoader.loadString(code);
}
```

Тест пройден — и это плохо. После этого, когда я пытаюсь сделать что-то с моделью (даже просто ее распечатать), я получаю исключение нулевого указателя. Этот пример достаточно прост (в конце концов, единственная его цель — дидактическая), но опечатка во входном тексте DSL может привести к существенному увеличению времени на отладку. Поскольку это мое личное время и я высоко его ценю, я хотел бы, чтобы хорошо работал принцип "быстрой ошибки".

Поскольку проблема в том, что я создаю некорректную структуру в семантической модели, ответственность за соответствующую проверку лежит на семантической модели, в данном случае — на методе, который добавляет переход к состоянию. Я добавляю утверждение для выявления проблем такого рода.

```
class State...
  public void addTransition(Event event,
                            State targetState) {
    assert null != targetState;
    transitions.put(event.getCode(),
             new Transition(this, event, targetState));
  }
```

Теперь я могу изменить тест, чтобы перехватить исключение. Помимо всего прочего, он документирует тип ошибки, вызываемой таким неверным входом.

```
@Test public void targetStateNotDeclared () {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "trigger => target " +
    "end ";
  try {
    StateMachine actual =
      StateMachineLoader.loadString(code);
    fail();
  } catch (AssertionError expected) {}
```

В утверждении проверяется целевое состояние, а не событие, которое также может быть нулевым. Причина в том, что нулевое событие вызовет немедленное исключение нулевого указателя из-за наличия вызова `event.getCode()`. Это соответствует принципу "быстрой ошибки". Проверить это можно с помощью другого теста.

```
@Test public void triggerNotDeclared () {
  String code =
    "events trigger TGGR end " +
    "state idle " +
    "wrongTrigger => target " +
    "end " +
    "state target end ";
  try {
    StateMachine actual =
        StateMachineLoader.loadString(code);
    fail();
  } catch (NullPointerException expected) {}
```

Исключение нулевого указателя соответствует принципу "быстрой ошибки", но при этом оно не столь ясно, как невыполненное утверждение. Вообще говоря, я не применяю проверки на равенство `null` к аргументам своих методов, так как, по моему мнению, выгода от них не оправдывает необходимости читать дополнительный код. Исключением являются ситуации, в которых нулевые значения не приводят к немедленным сбоям в работе, как в рассмотренном выше случае нулевого целевого состояния.

3.6.3. Тестирование сценариев

Тестирование семантической модели (Semantic Model (171)) и синтаксического анализатора представляет собой модульное тестирование обобщенного кода. Однако сценарии DSL также являются кодом, так что мы должны рассмотреть и их тестирование. Я уже слышу аргументы типа “сценарии DSL слишком просты и очевидны, чтобы их тестировать”, но я, естественно, отношусь к ним отрицательно. Я представляю такое тестирование как механизм двойной проверки. В процессе написания кода и тестов мы определяем одно и то же поведение, используя совершенно разные механизмы, один из которых включает абстракции (код), а другой — примеры (тесты). Чтобы такое тестирование имело смысл, всегда следует выполнять двойную проверку.

Детали тестирования сценариев во многом зависят от того, что именно тестируется. Общий подход заключается в предоставлении тестовой среды, которая позволяет создавать исходные тексты, запускать сценарии DSL и сравнивать результаты. Подготовка такой среды, как правило, требует некоторых усилий, но то, что DSL легко читается, еще не означает, что программисты не будут делать ошибки. Если вы не предоставите тестовую среду (и, таким образом, у вас не будет механизма двойной проверки), вы значительно увеличите риск возникновения ошибок в сценариях DSL.

Тесты сценариев действуют и как тесты интеграции, поскольку любые ошибки в синтаксическом анализаторе или семантической модели должны приводить к их сбоям. Так что стоит выбрать несколько сценариев DSL для их использования для этой цели.

Часто альтернативные визуализации сценария оказывают помощь в тестировании и отладке DSL-сценариев. Различные текстовые и графические визуализации логики сценария выполнить относительно легко. Представление информации несколькими разными способами часто помогает людям находить ошибки. Понятие двойной проверки — главная причина написания самотестируемого кода.

Для нашего конечного автомата я начал с примеров, имеющих смысл для данного типа автомата. На мой взгляд, логичнее было бы запускать сценарии, каждый из которых представляет собой последовательность событий, посылаемых автомату. Затем следует проверить конечное состояние автомата и команды, которые были им посланы. Построение системы наподобие описанной удобочитаемым способом естественным образом приводит к другому DSL. Это не редкость — тестирование сценариев является обычным применением DSL, так как они хорошо отвечают потребности в ограниченном, декларативном языке.

```
events("doorClosed", "drawerOpened", "lightOn")
       .endsAt("unlockedPanel")
       .sends("unlockPanel", "lockDoor");
```

3.7. Обработка ошибок

Всякий раз, когда я пишу книгу, наступает момент, когда я понимаю, что следует ограничить рассматриваемый материал, иначе эту книгу никогда не увидит читатель. Хотя это и означает, что некоторая важная тема не рассматривается надлежащим образом, я все же считаю, что неполная книга полезнее, чем неизданная. Есть много тем, которые я бы хотел осветить в этой книге, и первой в списке является обработка ошибок.

Когда я изучал компиляторы в университете, преподаватель сказал, что самое простое при написании компилятора — это создание синтаксического анализатора и генератора кода, а самое трудное — обеспечение информативных сообщений об ошибках. Соответ-

ственно, диагностика ошибок оказалась за рамками изучаемого мною в университете курса, точно так же, как сейчас она находится за рамками этой книги.

Но эта "зарамочность" идет еще дальше: хорошая диагностика — большая редкость даже в успешных DSL. Многие очень полезные пакеты DSL выдают очень мало полезной информации об ошибках. Graphviz, один из моих любимых инструментов DSL, просто сообщает о синтаксической ошибке вблизи строки 4, и я чувствую себя счастливым от того, что меня хотя бы сориентировали по номеру строки. Я оказывался в ситуациях, когда приходилось по сути выполнять бинарный поиск (состоящий в комментировании половины строк рассматриваемого блока), чтобы найти место расположения проблемы.

Можно справедливо критиковать такие системы за их слабую диагностику ошибок, но уровень диагностики — это результат компромиссов. Время на улучшение обработки ошибок — это время, которое могло бы быть затрачено на добавление других возможностей. Практика свидетельствует, что в реальных DSL их пользователи, как правило, вполне мирятся со слабой диагностикой ошибок. В конце концов, сценарии DSL малы, так что грубые методы поиска ошибок в этом случае работают существенно лучше, чем с языками программирования общего назначения.

Я говорю это не для того, чтобы убедить вас не работать над диагностикой ошибок. В интенсивно используемых библиотеках хорошая диагностика может сэкономить много времени. Каждый компромисс уникален, и вы должны достигать его, исходя из собственных обстоятельств. Однако это соображение позволяет моей совести не так сильно мучить меня из-за того, что я не уделил данному вопросу должного внимания в этой книге.

Я не могу изложить в этой книге все, что мне хотелось бы написать по данной теме, но надеюсь, что приведенного материала будет достаточно для того, чтобы, если вы решите усовершенствовать свои разработки, вы уделили диагностике ошибок больше внимания.

(Один момент, который я должен упомянуть, — это самая грубая методика поиска ошибок, состоящая в комментировании больших блоков исходного текста. В случае внешнего DSL убедитесь, что вы поддерживаете в нем комментарии, — не только в силу очевидных причин, но и для того, чтобы помочь людям найти проблемы в своем коде. Работать с такими комментариями легче всего тогда, когда они продолжаются до конца строки. В зависимости от целевой аудитории я обычно использую либо "#" (стиль сценариев), либо "//" (стиль C). Это делается с помощью очень простого правила лексического анализатора.)

Если вы следуете моему глобальному совету использовать семантическую модель (Semantic Model (171)), то учтите, что для размещения обработки ошибок есть два места: модель и синтаксический анализатор. Очевидным местом обработки синтаксических ошибок является синтаксический анализатор. Некоторые синтаксические ошибки будут обрабатываться вместо вас: синтаксические ошибки базового языка во внутреннем DSL или грамматические ошибки при использовании генератора синтаксических анализаторов (Parser Generator (277)) во внешнем DSL.

В ситуации с семантическими ошибками у вас есть выбор между обработкой их синтаксическим анализатором и моделью. В смысле семантики у обоих есть свои сильные стороны. Модель — действительно подходящее место для проверки правил семантически корректно сформированных структур. У вас есть вся необходимая информация, структурированная необходимым образом, так что вы можете написать ясный код для проверки наличия ошибок. Кроме того, если модель заполняется более чем из одного места (например, при использовании нескольких DSL или с помощью интерфейса командных запросов), то выполнять проверки следует именно здесь.

Размещение обработки ошибок только в семантической модели имеет один серьезный недостаток: отсутствует связь с источником проблемы в сценарии DSL, так что но-

мер строки неизвестен даже приблизительно. Это существенно затрудняет выяснение, что же пошло не так, но вовсе не является неразрешимой проблемой. Имеющийся опыт позволяет предположить, что даже сообщения об ошибке от модели достаточно, чтобы в самых разных ситуациях найти источник проблемы.

Существует несколько способов получения контекста сценария DSL, если в нем возникает необходимость. Наиболее очевидным является размещение правил обнаружения ошибок в синтаксическом анализаторе. Однако проблемой при использовании этой стратегии является существенное усложнение написания правил, так как работа ведется на уровне синтаксического дерева, а не семантической модели. Существенно вырастает и риск дублирования правил со всеми вытекающими из дублирования кода проблемами.

В качестве альтернативы можно внести синтаксическую информацию в семантическую модель. Можно добавить поле номера строки в объект семантического перехода, так что семантическая модель, обнаружив ошибки в переходе, может вывести номер строки из сценария. В этом случае проблема кроется в усложнении семантической модели, которая должна отслеживать синтаксическую информацию. Кроме того, отображение сценария на модель может не быть совершенно ясным и однозначным и привести к сообщениям об ошибках, которые еще больше запутают пользователя.

Третья стратегия (которая мне нравится больше всех) заключается в использовании для обнаружения ошибок семантической модели и запуске системы обнаружения ошибок в синтаксическом анализаторе. Таким образом, синтаксический анализатор будет анализировать фрагмент сценария DSL, заполнять семантическую модель и запрашивать проверку этой модели (если это не делается непосредственно при заполнении модели). Если модель находит ошибки, синтаксический анализатор может получить информацию о них и сопоставить с известным ему контекстом сценария DSL. Таким образом, происходит отделение синтаксических знаний (в синтаксическом анализаторе) от знаний семантических (в модели).

Полезным подходом является разделение обработки ошибок на инициацию, обнаружение и отчетность. Последняя стратегия переносит инициацию в синтаксический анализатор, обнаружение — в модель, а отчетность — в оба места. При этом модель поставляет семантику ошибки, а синтаксический анализатор добавляет к ней синтаксический контекст.

3.8. Миграция DSL

Одна из опасностей, о которой предупреждают сторонники DSL, заключается в том, что сначала вы проектируете DSL, а затем люди его используют. Как и любое другое программное обеспечение, успешные DSL будут развиваться. Это означает, что сценарии, написанные в ранней версии DSL, могут не работать в более поздней версии.

Подобно многим другим свойствам DSL, хорошим и плохим, это очень похоже на то, что происходит с библиотеками. Если вы получили от кого-то библиотеку, написали использующий ее код, а библиотека затем была обновлена, ваш код может перестать работать. DSL в этом плане ничем не отличаются; определение DSL, по сути, представляет собой опубликованный интерфейс, так что вы имеете дело с теми же последствиями, что и при работе с библиотеками.

Я начал использовать термин **опубликованный интерфейс** в своей книге о рефакторинге [12]. Разница между опубликованным (published) и более распространенным открытым (public) интерфейсом состоит в том, что опубликованный интерфейс используется кодом, написанным иной, отдельной командой. Поэтому когда команда, которая определяет интерфейс, хочет его изменить, она не может просто переписать вызываю-

щий код. Изменение опубликованного DSL является проблемой как для внутренних, так и для внешних предметно-ориентированных языков. В случае неопубликованных DSL, пожалуй, легче поддаются изменению внутренние DSL, если рассматриваемый язык имеет автоматизированные средства рефакторинга.

Один из способов решения проблемы изменения DSL — предоставление инструментов, которые автоматически выполняют миграцию DSL с одной версии на другую. Они могут быть запущены либо во время обновления, либо автоматически при попытках запуска старых версий сценариев.

Есть два основных способа справиться с задачей миграции. Первый — стратегия **инкрементной миграции**. Это по сути то же понятие, что и используемое при эволюционном проектировании баз данных [9]. Для каждого изменения, вносимого в определение DSL, создается программа миграции, которая автоматически преобразует все старые сценарии в сценарии новой версии. Таким образом, выпуская новую версию DSL, вы одновременно обеспечиваете возможность перехода на новую версию любого кода, использующего DSL.

Важной частью инкрементной миграции является то, что вы делаете вносимые изменения минимальными. Представьте, что вы выполняете обновление с версии 1 до версии 2 и у вас имеется десять изменений, вносимых в определение вашего DSL. В этом случае не создавайте только один сценарий для миграции с версии 1 до 2; создайте вместо этого по крайней мере десять сценариев. Изменяйте определение DSL по одной возможности за раз и пишите сценарий миграции для каждого вносимого изменения. Вы можете решить, что стоит разбивать процесс еще сильнее и добавлять некоторые возможности более чем за один шаг (а значит, более чем одной миграцией). Может показаться, что такой способ приведет к большему объему работы, чем потребовал бы один сценарий, но дело в том, что сценарии миграции намного легче писать, если они маленькие, и достаточно просто соединить несколько таких последовательных миграций вместе. В результате вы сможете написать десять сценариев гораздо быстрее, чем один.

Другой подход заключается в миграции на основе моделей. Эту тактику можно использовать с семантической моделью (Semantic Model (171)). **Миграция на основе моделей** позволяет поддерживать несколько синтаксических анализаторов вашего языка, по одному для каждой версии. (Таким образом, вы делаете это только для версий 1 и 2, но не для промежуточных шагов.) Каждый синтаксический анализатор заполняет семантическую модель. При использовании семантической модели поведение синтаксического анализатора довольно простое, так что иметь их несколько не доставляет особых хлопот. Затем вы запускаете синтаксический анализатор, соответствующий версии сценария, с которой вы работаете. Так вы можете работать с несколькими версиями сценариев, не прибегая к их миграции. Для выполнения миграции вы пишете генератор на основе семантической модели, который генерирует представление сценария DSL. Таким образом, вы можете запустить синтаксический анализатор для сценария версии 1, заполнить семантическую модель и получить сценарий для версии 2 с помощью генератора.

Одна из проблем, возникающих в процессе применения данной миграции на основе модели, заключается в том, что в сценариях легко потерять то, что не имеет значения для семантики, но что авторы сценариев хотели бы сохранить. Очевидный пример — комментарии.

Если изменений в DSL достаточно много, может оказаться невозможным преобразование сценария версии 1 в сценарий версии 2 семантической модели. В этом случае, возможно, потребуется сохранить модель версии 1 (или промежуточную модель) и придать ей возможность генерировать сценарии версии 2.

У меня нет явно выраженного предпочтения того или иного из описанных путей.

Сценарии миграции могут быть запущены при необходимости самими программистами сценариев или автоматически системой DSL. При автоматическом выполнении

очень полезно иметь возможность записи версии DSL в сценарии, чтобы синтаксический анализатор мог легко его выяснить и выполнить миграцию. Действительно, некоторые авторы DSL утверждают, что все DSL обязательно должны содержать указание версии в сценарии, чтобы можно было легко обнаруживать устаревшие сценарии и обеспечивать поддержку миграции сценариев. Хотя инструкция указания версии несколько "засоряет" сценарий, ее очень трудно усовершенствовать.

Конечно, есть еще один вариант — отказаться от миграции, т.е. поддерживать синтаксический анализатор версии 1 и позволить ему заполнять модель версии 2. Вы должны помочь людям перейти на новую версию, и они должны будут это сделать, если хотят использовать дополнительные возможности. Но непосредственная поддержка старых сценариев, если таковая возможна, весьма полезна, так как позволяет пользователям мигрировать в собственном темпе.

Хотя такие методы весьма привлекательны, возникает вопрос об их стоимости на практике. Как я уже говорил, эта проблема точно такая же, как и в случае широко используемых библиотек, а здесь автоматизированные схемы миграции практически не применяются.