内部 DSL によるシステム開発に関する検討

A Study on System Development with Internal DSLs

前田和昭
Kazuaki Maeda
中部大学　経営情報学部
School of Business Administration and Information Science, Chubu University

要旨

これまで，構造化データ表現とそのためのスキーマ定義言語，パーザ生成系のための構文ルール記述などで内部 DSL を活用してきた．内部 DSL を使うことによって，簡潔で強力な記述が可能になる．本稿では，これらの経験を踏まえ，内部 DSL をシステム開発に活用する上でのポイントについて検討する．

1. はじめに

インターネットの発達とコンピュータの飛躍的な性能向上を背景に，XML を使ってデータを表現することが多くなってきた．XML は，主要プログラミング言語とは全く異なる構文を使って，構造化データを表現し，ネットワークを介したアプリケーション間のデータ交換や，データベースからデータを取り出して他の製品で活用するなど，多くの分野で利用されている．最近では，構造化データを簡潔に表現できることから，XML を使わずに JSON[1] が使われる場面を見かけることも増えてきている．

XML や JSON は，構造化データを表現することに特化した言語であり，このように特定ドメインに特化して設計された言語のことを DSL（Domain-Specific Language，ドメイン特化言語）と呼んでいる．Martin Fowler の著書 [2] では，DSL に関して以下の４つの特徴が述べられている[1]．DSL は，

- 人間がコンピュータに処理を指示するための言語であり，
- 流れるような感覚を備えていて，
- 対象となるドメインをサポートするための必要最低限の機能を提供し，
- 特定のドメインに集中することで，限定された価値のある言語になる

特徴を持つ．さらには，２種類の DSL について，

- 外部 DSL とは，開発すべきアプリケーションの主要言語から独立し，その主要言語とは異なった記述能力を持つ言語である
  （例：SQL, AWK, Struts や Hibernate で利用する XML 設定ファイルなど）
- 内部 DSL とは，汎用言語の構文のサブセットを使って表現される言語である
  （例：Ruby on Rails や Rake の設定ファイルなど）

と記述されている．外部 DSL は，独自の構文を持ち，Java, C++, Ruby などの汎用言語とは全く異なる．したがって，外部 DSL を利用したアプリケーションを開発するには，DSL で記述したコードを解析するための構文解析プログラム（以下，パーザと記す）が必要となる．XML は，構造化データを表現するための外部 DSL として頻繁に使われ，XML を読み込むために多くのがパーザが開発されている．

筆者は，構造化データ表現のための RibON，そのためのスキーマ定義言語，パーザ生成系のための構文定義言語 RibLR などを内部 DSL として設計し実験的に使用してきた．これらは Ruby 構文のサブセットを使ったもので，簡潔で強力な記述が可能となっている．本稿では，これまで内部 DSL を設計してきた経験を述べ，システム開発に内部 DSL を活用するためのポイントについて検討する．

[1] 日本語版 [3] の記述を土台にして筆者が意訳した．

2. 構造化データ表現と RibON

コンパイラは，ソースコードを読んで解析し，ソースコードの構文と意味を表現するための抽象構文木（以下，AST と記す）を構築する．この AST を XML で表現するために，JavaML[4] と XMLizer[5] が提案されている．ソースコードを入力とし，図 1 に示すような XML で表現された AST を出力するツールがあれば，パーザを作成するという厄介な作業を省く事ができる．XML を操作するためのライブラリが多数利用可能であり，これらを活用してソースコード解析のためのツール開発の手間が軽減できる．

ネットワーク上でデータを交換する場面で，構造化データを簡潔に表現するために，JSON を見かけることが多くなってきた．JSON は，JavaScript のサブセットを使って設計された構造化データ表現のための内部 DSL であり，単純に JSON データを読み込むだけであれば，解析のためのパーザは必要ない．JavaScript を使って構造化データを処理するためのプログラムを開発するならば，JSON による構造化データの表現は開発者にとって有利となる．しかし，開発のために JavaScript 以外のプログラミング言語を使うのであれば，外部 DSL の場合と同じように，JSON による構造化データを読み込むためのパーザを準備する必要がある．

```
<Node class="Add">
  <left class="IntConst">
    <ival>1</ival>
  </left>
  <right class="IntConst">
    <ival>2</ival>
  </left>
</Node>
```

図 1: 二項演算の XML による表現

```
option java_package = "org.rugson";
message Node {
  enum NodeType {
    Add = 1; Mul = 2; IntConst = 3;
  }
  required Node left  = 1;
  required Node right = 2;
  required int32 ival = 3;
}
```

図 2: Protocol Buffers のためのスキーマ定義

Java の標準クラスライブラリでは，オブジェクト直列化が提供されている．構造化データをコード化してファイルに書き出すには，単に Serializable インタフェースを実装するだけでよい．Java のオブジェクト直列化を使うと，バイナリ形式でファイルに書き出される．デバッグ等のために人間が書き出されたデータを読むことを考えると，バイナリ形式よりもテキスト形式（XML や JSON など）を使った方が好ましい．

XML を使って構造化データを直列化するための Java ライブラリとして，XStream[6] がある．XStream では，実行時にリフレクションを使ってオブジェクトを精査するため，Serializable インタフェースを実装するような余分な記述は必要ない．しかし，直列化の性能実験の結果から，性能面でのペナルティを抱えていることが分かっている [7].

効率的な実装を目指した Protocol Buffers[8] では，スキーマ定義を図 2 のように記述し，コード生成系が生成するプログラムを使って，独自のバイナリ形式でオブジェクトを直列化する．各フィールドにはデータ型（整数，実数，文字列，列挙型など）とユニークな番号を割り当てる．このスキーマ定義は，Protocol Buffers に特化した外部 DSL で記述するため，開発者が習得するまでに時間がかかる．もし，慣れ親しんだプログラミング言語に基づいた内部 DSL を使えば，開発者の負担は減るであろう．

以上の議論をふまえ，構造化データをテキスト形式で表現するための言語 RibON を設計した．また，慣れ親しんだ言語を土台とした内部 DSL でスキーマ定義を記述できるようにした．RibON は，構造化データを表現するための Ruby を土台とした内部 DSL である．例えば，二項演算 `1 + 2` を表現した例の一部を図 3 に示す．RibON では，データの最小単位を要素と呼び，名前と値のペアで表現する．図中の要素 `ival 1` は，要素名が `ival` で，その値が `1` であることを示す．これは単なるデータ表現ではな

く，メソッド名が ival ，引数が 1 のメソッド呼び出しになっている．また，Ruby のブロックを使って構造を表現する．図 3 では，ブロックを記述することで，expr が 3 つの子要素 kind，left，right を持つことを示している．

```
expr{
  kind :Add
  left{
    kind :IntConst
    ival 1
  }
  right{
    kind :IntConst
    ival 2
  }
}
```

図 3: 二項演算の RibON による表現

```
:expr.has :left, :right
:left.is_alias_of  :expr
:right.is_alias_of :expr
:add.is_a :expr
:intConst.is_a :expr
:intConst.has :ival
:ival.is_type :int
```

図 4: RibON のためのスキーマ定義

RibON のためのスキーマ定義では，図 4 に示すように，Ruby のシンボルとメソッド呼び出しを使う．has は要素の構成を示し，is_a は汎化関係を示し，is_type は要素のデータ型を示し，is_alias_of は要素とデータ型を別の名前で扱うことを示している．このスキーマ定義は，Ruby で構築されたツールによって処理され，プログラム開発を支援するためのコードが生成される．

3. パーザ生成系と RibLR

1970 年以前，汎用的なプログラミング言語のためのパーザを開発することは複雑な作業が必要であった．しかし，1970 年代になり，パーザ生成系 Yacc[9] が提供され，パーザ開発の手間が軽減された．それ以降，Bison[10], SableCC[11], Antlr[12] など多数のパーザ生成系が開発されている．

パーザ生成系は，ユーザが定義する構文ルールにアクションコードを埋め込んだファイルを読み込み，特定のプログラミング言語で記述されたパーザを生成する．生成されたパーザがソースコードを読み込むとき，ある構文ルールがマッチすると，その構文ルールに埋め込まれたアクションコードが実行される．図 5 に，単純な二項演算のための構文ルールの一部を示す．これは，Berkeley Yacc 互換の BYACC/J[13] の入力の一部であり，各構文ルールには Java によるアクションコードが埋め込まれている．

```
expr : expr PLUS NUMBER
     { System.out.println("plus expr");
       $$ = Util.add($1,$3); }
     | NUMBER
     { $$ = $1; }
     ;
```

図 5: 二項演算のための構文ルール

```
expr : expr PLUS term
     { System.out.println("plus expr")
       $$ = Util.add($1,$3); }
     | term
     { $$ = $1 }
     ;

% byaccj -J -Jclass=CalcParser parse-error.y

% javac CalcParser.java
CalcParser.java:395: ';' expected
{ System.out.println("plus expr")
                                ^
CalcParser.java:400: ';' expected
{ yyval.obj = val_peek(0).obj }
                              ^
2 errors
```

図 6: アクションコード中にエラーがある例

例えば，パーザが 1+2 を読み込んだとすると，

```
expr :  expr PLUS NUMBER
```

がマッチし，その直下のアクションコードが実行される．構文ルールの記述は，Java や C などのような汎用言語の構文とは全く異なる外部 DSL で記述される．

BYACC/J は，アクションコード中の構文エラーを見つけることができない．図 6 に，文末にセミコロンがない些細な構文エラーをアクションコードに含む例を示す．BYACC/J はアクションコードの構文を解析しないため，そのエラーを含んだままコードが生成される．コード生成の後，Java コンパイラが 395 行目と 400 行目に構文エラーを見つけるが，Java コンパイラは，どの構文ルールに埋め込まれたアクションコードなのか知らせてくれることはない．このエラーを修正するには，パーザ生成系が生成した Java ソースコードと構文ルールを見比べながら，どの構文ルールにエラーがあるのか開発者が判断する必要がある．

```
:expr .> :expr,:PLUS,:NUMBER
:expr .> :NUMBER

:PLUS.is_token
:NUMBER.is_token
```

図 7: RibLR で記述した構文ルール

```
action(:expr,:expr,:PLUS, :NUMBER) do
  $params[0] = $params[1] + $params[3]
end
action(:expr,:NUMBER) do
  $params[0] = $params[1]
end
```

図 8: RibLR のためのアクションコード記述

この問題を解消するために，Ruby を土台にした内部 DSL で構文ルールを記述できるように RibLR を設計した．図 7 に，図 5 で記述した構文ルールを RibLR で書き直した例を示す．RibLR では，シンボルと，. > メソッド呼び出しを使って構文ルールを定義する．アクションコードは，図 8 に示すように，action メソッド呼び出しを使って指定し，これらの定義をコード生成ツールが読み込んで，Ruby で記述されたパーザのソースコードを生成する．

4. 検討

外部 DSL は，汎用プログラミング言語とは全く異なる構文を使うため，開発で使えるよう慣れ親しむまで時間がかかる．しかも，外部 DSL にアクションコードを埋め込み，それをコード生成系が入力として受け付ける場合，構文エラーが出力されても，その原因を探し出すのに手間がかかる場合が多々ある．

これに対して，内部 DSL では，汎用プログラミング言語のサブセットを使うため，その言語に馴染みのある開発者に受け入れられやすい．また，内部 DSL は，プログラミング言語と密接に結びつき，処理系がそのまま解析するため，構文エラーが出力された場合はすぐに対応できる．

内部 DSL が土台とするプログラミング言語を選定するには注意が必要である．本稿では，Ruby を土台にして構造化データを表現する RibON とそのスキーマ定義，構文ルールを記述する RibLR を紹介した．Ruby は柔軟な構文を備えている．Ruby では，メソッド呼び出しのための引数を括る括弧を省略でき，また，ブロックを使って階層構造を表現できる．他の言語として，Groovy を使って内部 DSL を設計する例が著書 [14] に紹介されている．Java を使って内部 DSL を設計することも可能ではあるが，筆者の経験では，Ruby や Groovy ほど簡潔に記述することは難しかった．過去の実例を十分に検討した上で，内部 DSL の土台となるプログラミング言語を選定すべきであろう．

開発者が XML データを意識しないでデータにアクセスできるようにするために，コード生成系がスキーマ定義を読み込み，クラス群を生成する手法が広く使われている [15][16]．しかし，XML データを読み込み，木構造データを内部で作り上げるプログラムを開発できても，木構造にアクセスするには，オブジェクト間の参照関係に沿って処理を進めるための単純な方法しか準備されていない．開発者が何とかしようとして，コードが生成されるときにユーザ定義のコードを埋め込もうとしても，コード生成

系はブラックボックスとして内部が隠されているため，どうにもならない．このように生成されるコードに，何らかのコードを埋め込みたいときには，内部 DSL を使って定義し，そこにコードを埋め込めるようにした方が有利である．これは，コード生成系の内部をユーザに公開するための仕組みを提供していることになる．内部 DSL による記述とユーザ定義のコードを処理系内に読み込んでから，そこからコードを生成するには，処理系内部からソースコードを逆生成する必要があり，この機能を持つ処理系が必須となることに注意が必要である．

5. おわりに

本稿では，構造化データ表現のための RibON，そのためのスキーマ定義言語，パーザ生成系のための構文定義言語 RibLR などを設計する場面で，内部 DSL を活用した経験を述べた．これらは Ruby 構文のサブセットを使い内部 DSL として設計したもので，簡潔で強力な記述が可能である．さらには，システム開発に内部 DSL を活用するためのポイントについて述べた．今後，さらなる設計と実装を進めていき，製品開発での応用を通して，その評価を進めていきたい．

参考文献

[1] JSON, http://www.json.org/ (accessed at October 29, 2012).
[2] Martin Fowler, Domain-Specific Languages, Addison-Wesley, 2011.
[3] マーチン・ファウラー, ドメイン特化言語, ピアソン桐原, 2012.
[4] Greg Badros, JavaML: A Markup Language for Java Source Code, 9th International World Wide Web Conference, http://www9.org/w9cdrom/342/342.html (accessed at October 29, 2012).
[5] Gregory McArthur, John Mylopoulos and Siu Kee Keith Ng, An Extensible Tool for Source Code Representation Using XML, 9th Working Conference on Reverse Engineering, 2002, pp.199–209.
[6] XStream – About XStream, http://xstream.codehaus.org/ (accessed at October 29, 2012).
[7] Kazuaki Maeda, Comparative Survey of Object Serialization Techniques and the Programming Supports, Journal of Communication and Computer, Vol.9, No.8, 2012, pp.920–928.
[8] protobuf – Protocol Buffers, http://code.google.com/p/protobuf/ (accessed at October 29, 2012).
[9] Steven C. Johnson, Yacc: Yet Another Compiler Compiler, UNIX Programmer's Manual, Vol. 2, 1979, pp.353–387.
[10] Bison - Gnu Parser Generator, http://www.gnu.org/s/bison/ (accessed at October 29, 2012).
[11] E. M. Gagnon and L. J. Hendren, SableCC, an Object-Oriented Compiler Framework, TOOLS 26 Technology of Object-Oriented Languages, 1998, pp.140–154.
[12] T. J. Parr and R. W. Quong, ANTLR: A Predicated-LL(k) Parser Generator, Software Practice & Experience, Vol.25, No.7, 1995, pp.789–810.
[13] BYACC/J Home Page, http://byaccj.sourceforge.net/ (accessed at October 29, 2012).
[14] Debrasish Ghosh, DSLs in Action, Manning, 2011.
[15] JAXB Reference Implementation – Java.net, http://jaxb.java.net/ (accessed at October 29, 2012).
[16] The Castor Project, http://www.castor.org/ (accessed at October 29, 2012).