Kari korpinen Ohjelmistotuotanto k 2012

Tehtävä 8 referenssi

An Approach to Internal Domain-Specific Languages in Java

Posted by **Alex Ruiz and Jeff Bay**

Introduction

A domain-specific language (DSL) is commonly described as a computer language targeted at a particular kind of problem and it is not planned to solve problems outside of its domain.Lately, with the advent of Ruby and other dynamic languages, there has been a growing interest in DSLs amongst programmers. These loosely structured languages offer an approach to DSLs which allow a minimum of grammar and direct representation of a particular language. However, discarding the compiler and the ability to use the most powerful modern IDEs such as Eclipse is a definite disadvantage with this approach.This article describes how it is possible to write domain-specific languages using the Java language and suggests some patterns for constructing them.

Is Java suited for creation of internal Domain-Specific Languages?

An internal DSL is created with the main language of an application without requiring the creation (and maintenance) of custom compilers and interpreters.

When thinking in terms of DSL, we exploit the host language to create a readable API with a limited scope. "Internal DSL" is more or less a fancy name for an API that has been created thinking in terms of readability and focusing on a particular problem of a specific domain.

Any internal DSL is limited to the syntax and structure of its base language. In the case of Java, the obligatory use of curly braces, parenthesis and semicolons, and the lack of closures and meta-programming may lead to a DSL that is more verbose that one created with a dynamic language.

On the bright side, by using the Java language we can take advantage of powerful and mature IDEs like Eclipse and IntelliJ IDEA, which can make creation, usage and maintenance of DSLs easier thanks to features like "auto-complete," automatic refactoring and debugging.

In general, a DSL written in Java will not lead to a language that a business user can create from scratch. It will lead to a language that is quite **readable** by a business user It has the advantage over an external DSL or a DSL written in a dynamic language that the compiler can enforce correctness along the way, and flag inappropriate usage where Ruby or Pearl would happily accept nonsensical input and fail at run-time.Using the compiler to improve quality in this way is an art however, and currently, many programmers are bemoaning the "hard work" of satisfying the compiler instead of using it to build a language that uses syntax to enforce semantics.

Java as a platform for internal DSLs

Dynamically constructing SQL is a great example where building a "DSL"appropriate to the domain of SQL is a compelling advantage.

Traditional Java code that uses SQL would look something like the following:

```
String sql = "select id, name " +
             "from customers c, order o " +
             "where " +
             "c.since >= sysdate - 30 and " +
             "sum(o.total) > " + significantTotal + " and " +
             "c.id = o.customer_id and " +
             "nvl(c.status, 'DROPPED') != 'DROPPED'";
```

An alternative representation taken from a recent system worked on by the authors:

```
Table c = CUSTOMER.alias();
Table o = ORDER.alias();
Clause recent = c.SINCE.laterThan(daysEarlier(30));
Clause hasSignificantOrders = o.TOTAT.sum().isAbove(significantTotal);
Clause ordersMatch = c.ID.matches(o.CUSTOMER_ID);
Clause activeCustomer = c.STATUS.isNotNullOr("DROPPED");
String sql = CUSTOMERS.where(recent.and(hasSignificantOrders)
                       .and(ordersMatch)
                       .and(activeCustomer)
                       .select(c.ID, c.NAME)
                       .sql();
```

The DSL version has several advantages. The latter version was able to accommodate a switch to using `PreparedStatement`s transparently - the `String` version requires extensive modification to switch to using bind variables. The latter will not compile if the quoting is incorrect or an integer parameter is passed to a date column for comparison. The phrase "`nvl(foo, 'X') != 'X'`" is a specific form found in Oracle SQL. It is virtually unreadable to a non-Oracle SQL programmer or anyone unfamiliar with SQL. That idiom in SQL Server, for example, would be "`(foo is null or foo != 'X')`." By replacing this phrase with the more easily understandable and language-like "`isNotNullOr(rejectedValue)`," readability has been enhanced, and the system is protected from a later need to change the implementation to take advantage of facilities offered by another database vendor.

Creating internal DSLs in Java

The best way to create a DSL is by first prototyping the desired API and then work on implementing it given the constraints of the base language. Implementation of a DSL will involve testing continuously to ensure that we are advancing in the right direction. This "prototype and test" approach is what Test-Driven Development (TDD) advocates.

When using Java to create a DSL, we might want to create the DSL through a fluent interface. A fluent interface provides a compact and yet easy-to-read representation of the domain problem we want to model. Fluent interfaces are implemented using method chaining. Here is an example:

```
StringBuilder b = new StringBuilder();
b.append("Hello. My name is ")
  .append(name)
  .append(" and my age is ")
  .append(age);
```

This example does not solve any domain-specific domain.

In addition to method chaining, static factory methods and imports are a great aid in creating a compact, yet readable DSL.

1. Method Chaining

There are two approaches to create a DSL using method chaining, and both are related to the return value of the methods in the chain. Our options are to return `this` or to return an intermediate object, depending on what we are trying to do.

1.1 Returning `this`

We usually return `this` when calls to the methods in the chain can be:

- optional
- called in any order
- called any number of times

We have found two use cases for this approach:

1. chaining of related object behavior
2. simple construction/configuration of an object

1.1.1 Chaining related object behavior

Many times, we only want to chain methods of an object to reduce unnecessary text in our code, by simulating dispatch of "multiple messages" (or multiple method calls) to the same object.The test verifies that an error message is displayed if a user tries to log into a system without entering her password.

```
DialogFixture dialog = new DialogFixture(new LoginDialog());
dialog.show();
dialog.maximize();
TextComponentFixture usernameTextBox = dialog.textBox("username");
usernameTextBox.clear();
usernameTextBox.enter("leia.organa");
dialog.comboBox("role").select("REBEL");
OptionPaneFixture errorDialog = dialog.optionPane();
errorDialog.requireError();
errorDialog.requireMessage("Enter your password");
```

The following are two methods from `TextComponentFixture` that were used in our example:

```
public void clear() {
   target.setText("");
}
```

```
public void enterText(String text) {
   robot.enterText(target, text);
}
```

We can simplify our testing API by simply returning `this`, and therefore enable method chaining:

```
public TextComponentFixture clear() {
   target.setText("");
   return this;
}

 public TextComponentFixture enterText(String text) {
   robot.enterText(target, text);
   return this;
}
```

After enabling method chaining in all the test fixtures, our testing code is now reduced to:

```
DialogFixture dialog = new DialogFixture(new LoginDialog());
dialog.show().maximize();
dialog.textBox("username").clear().enter("leia.organa");
dialog.comboBox("role").select("REBEL");
dialog.optionPane().requireError().requireMessage("Enter your password");
```

In our example, the domain-specific problem was Swing GUI testing.

1.1.2 Simple construction/configuration of an object

we create a "builder" to create and/or configure objects using a fluent interface.

The following example illustrates a "dream car" created using setters:

```
DreamCar car = new DreamCar();
car.setColor(RED);
car.setFuelEfficient(true);
car.setBrand("Tesla");
```

The code for the `DreamCar` class is pretty simple:

```
// package declaration and imports

public class DreamCar {

   private Color color;
   private String brand;
   private boolean leatherSeats;
   private boolean fuelEfficient;
   private int passengerCount = 2;

   // getters and setters for each field
}
```

we can create more compact code using a car builder:

```
// package declaration and imports

public class DreamCarBuilder {

   public static DreamCarBuilder car() {
     return new DreamCarBuilder();
   }

   private final DreamCar car;

   private DreamCarBuilder() {
     car = new DreamCar();
   }

   public DreamCar build() { return car; }

   public DreamCarBuilder brand(String brand) {
     car.setBrand(brand);
     return this;
   }

   public DreamCarBuilder fuelEfficient() {
     car.setFuelEfficient(true);
     return this;
   }

   // similar methods to set field values
}
```

Using the builder we can rewrite the `DreamCar` creation:

```
DreamCar car = car().brand("Tesla")
                     .color(RED)
                     .fuelEfficient()
                     .build();
```

Using a fluent interface, once again, reduced noise in code, which resulted in more readable code. It is imperative to note that, when returning `this`, any method in the chain can be called at any time and any number of times.

Another important observation is that there is no compiler checking to enforce required field values.

1.2 Returning an intermediate object

Returning an intermediate object from methods in a fluent interface has some advantages over returning `this`:

- we can use the compiler to enforce business rules (e.g. required fields)
- we can guide our users of the fluent interface through a specific path by limiting the available options for the next element in the chain
- gives API creators greater control of which methods a user can (or must) call, as well as the order and how many times a user of the API can call a method

The following example illustrates a vacation created using constructor arguments:

```
Vacation vacation = new Vacation("10/09/2007", "10/17/2007",
                                 "Paris", "Hilton",
                                 "United", "UA-6886");
```

The benefit of this approach is that it forces our users to specify all required parameters.

A second approach is to use setters as a way to document each parameter:

```
Vacation vacation = new Vacation();
 vacation.setStart("10/09/2007");
 vacation.setEnd("10/17/2007");
vacation.setCity("Paris");
vacation.setHotel("Hilton");
vacation.setAirline("United");
vacation.setFlight("UA-6886");
```

Our code is more readable now, but it is also verbose. A third approach could be to create a fluent interface to build a vacation, like in the example in the previous section:

```
Vacation vacation = vacation().starting("10/09/2007")
                              .ending("10/17/2007")
                              .city("Paris")
                              .hotel("Hilton")
                              .airline("United")
                              .flight("UA-6886");
```

This version is more compact and readable, but we have lost the compiler checks for missing fields that we had in the first version. At this point, the best we can do with this approach is to throw exceptions at run-time if any of the required fields was not set.

The following is a fourth, more sophisticated version of the fluent interface. This time, methods return intermediate objects instead of `this`:

```
Period vacation = from("10/09/2007").to("10/17/2007");
Booking booking = vacation.book(city("Paris").hotel("Hilton"));
booking.add(airline("united").flight("UA-6886");
```

Here we have introduced the concept of `Period`, a `Booking`, as well as a `Location` and `BookableItem` (`Hotel` and `Flight`), and an `Airline`. The airline, in this context, is acting as a factory for `Flight` objects; the `Location` is acting as a factory for `Hotel` items, etc.

Static factory methods, when used with static imports, can help us create more compact fluent interfaces. For example, without static imports, the previous example will need to be coded like this:

```
Period vacation = Period.from("10/09/2007").to("10/17/2007");
Booking booking = vacation.book(Location.city("Paris").hotel("Hilton"));
booking.add(Flight.airline("united").flight("UA-6886");
```

Here is a second example of a DSL in Java. This time, we are simplifying usage of Java reflection:

```
Person person = constructor().withParameterTypes(String.class)
                             .in(Person.class)
                             .newInstance("Yoda");
```

```
 method("setName").withParameterTypes(String.class)
                  .in(person)
                  .invoke("Luke");

 field("name").ofType(String.class)
              .in(person)
              .set("Anakin");
```

2. Static Factory Methods and Imports

We have found that static factory methods are a convenient way to simulate named parameters in Java, a feature that many developers wish the language had. For example, consider this code, which purpose is to test a GUI by simulating a user selecting a row in a `JTable`:

```
dialog.table("results").selectCell(6, 8); // row 6, column 8
```

We could also declare the row and column indices as variables or better yet, as constants:

```
int row = 6;
int column = 8;
dialog.table("results").selectCell(row, column);
```

To keep code as compact as possible, the ideal solution would to write something like this:

```
dialog.table("results").selectCell(row: 6, column: 8);
```

Unfortunately, we cannot do that because Java does not support named parameters. On the bright side, we can simulate them by using a static factory method and static imports, to get something like:

```
dialog.table("results").selectCell(row(6).column(8));
```

We can start by changing the signature of the method, by replacing all the parameters with one object that will contain them. In our example, we can change the signature of `selectCell(int, int)` to:

```
selectCell(TableCell);
```

`TableCell` will contain the values for the row and column indices:

```
public final class TableCell {

   public final int row;
   public final int column;

   public TableCell(int row, int column) {
     this.row = row;
     this.column = column;
   }
}
```

`TableCell`'s constructor is still taking two `int` values. The next step is to introduce a factory of `TableCell`s, which will have one method per parameter in the original version of `selectCell`. In addition, to force users to use the factory, we need to change `TableCell`'s constructor to `private`:

```
public final class TableCell {

   public static class TableCellBuilder {
     private final int row;

     public TableCellBuilder(int row) {
       this.row = row;
     }

     public TableCell column(int column) {
       return new TableCell(row, column);
     }
   }

   public final int row;
   public final int column;

   private TableCell(int row, int column) {
     this.row = row;
     this.column = column;
   }
}
```

By having the factory `TableCellBuilder` we can create a `TableCell` having one parameter per method call.

```
selectCell(new TableCellBuilder(6).column(8));
```

The last step is to introduce a static factory method to replace usage of `TableCellBuilder` constructor, which is not communicating what 6 stands for. As we did previously, we need to make the constructor `private` to force our users to use the factory method:

```
public final class TableCell {

   public static class TableCellBuilder {
     public static TableCellBuilder row(int row) {
       return new TableCellBuilder(row);
     }

     private final int row;

     private TableCellBuilder(int row) {
       this.row = row;
     }

     private TableCell column(int column) {
       return new TableCell(row, column);
     }
   }

   public final int row;
   public final int column;

   private TableCell(int row, int column) {
     this.row = row;
     this.column = column;
   }
 }
```

Now we only need to add to our code calling `selectCell` is include an static import for the method `row` in `TableCellBuilder`.

```
dialog.table("results").selectCell(row(6).column(8));
```

The following code listing shows an alternative way to solve the same problem of table indices, using a static factory methods and imports in a different way:

```
/**
  * @author Mark Alexandre
  */
public final class TableCellIndex {

   public static final class RowIndex {
     final int row;
     RowIndex(int row) {
       this.row = row;
     }
   }

   public static final class ColumnIndex {
     final int column;
     ColumnIndex(int column) {
       this.column = column;
     }
   }

   public final int row;
   public final int column;
   private TableCellIndex(RowIndex rowIndex, ColumnIndex columnIndex) {
     this.row = rowIndex.row;
     this.column = columnIndex.column;
   }

   public static TableCellIndex cellAt(RowIndex row, ColumnIndex column) {
     return new TableCellIndex(row, column);
   }

   public static TableCellIndex cellAt(ColumnIndex column, RowIndex row) {
     return new TableCellIndex(row, column);
   }

   public static RowIndex row(int index) {
     return new RowIndex(index);
   }

   public static ColumnIndex column(int index) {
     return new ColumnIndex(index);
   }
}
```

The second version of the solution is more flexible than the first one:

```
dialog.table("results").select(cellAt(row(6), column(8));
dialog.table("results").select(cellAt(column(3), row(5));
```

Organizing Code

It is a lot easier to organize code of a fluent interface which methods return `this`, than the one which methods return intermediate objects.

Organizing code of a fluent interface using intermediate objects as return type is trickier because we have the logic of the fluent interface scattered across several small classes. Since these classes, together, as a whole, form our fluent interface, it makes sense to keep them together and we might not want them to mix them with classes outside of the DSL. We have found two options:

- Create intermediate objects as inner classes
- Have intermediate objects in their own top-level classes, all in the same package

Documenting Code

As in organizing code, documenting a fluent interface which methods return `this` is a lot easier than documenting a fluent interface returning intermediate objects, especially if documenting using Javadoc.

We should be careful and not duplicate documentation, because it will increase maintenance costs for the API creators. The best approach is to rely on tests as executable documentation as much as possible.

In Conclusion

Java can be suited to create internal domain-specific languages that developers can find very intuitive to read and write, and still be quite readable by business users.

Users of our APIs will see improvements in their code bases. Their code will be more compact and easier to maintain, which can simplify their lives.