Multi-DSL Applications with Ruby

Sebastian Günther
School of Computer Science
University of Magdeburg
Magdeburg, Germany
sebastian.guenther@ovgu.de

***Abstract*—Domain-specific languages (DSL) are becoming a mature tool in application development. With the help of DSLs, developers express concerns in specifically tailored languages. We focus on internal DSL – a special type which uses an existing programming language as its host. Our research into internal DSL utilization eventually led us to discover *multi-DSL applications*. These applications consist entirely of DSLs and host language code. In this paper, we explain multi-DSL applications in the context of web applications and the Ruby programming language. Exploiting Ruby's built-in support for the imperative, functional, and object-oriented paradigm, which we further extend with feature-oriented programming, we use integrated and interwoven multiparadigm expressions of several DSLs to express all application layers, concerns, and artifacts. In addition to the case study, we discuss how this approach impacts analysis, design, implementation, and testing of applications.**

***Keywords*-D.3.2.m Programming languages/Multiparadigm languages, D.3.2.s Programming languages/Specialized application languages, J.8.n Internet applications/Software engineering**

I. Introduction

Today's web applications are demanding in terms of languages. Chunks of HTML, CSS, JavaScript, XML, JSON, SQL, and a server language (PHP, Ruby, Java, Scala) are used for implementation. Adding web services, protocols, authentication, and sessions as further concerns, and a complex mix emerges. How can we design and implement applications satisfying complex requirements and utilizing different target languages?

We approach this challenge with the help of Domain-Specific Languages. DSLs use domain-specific notations and abstractions [1] to express the domain knowledge as an executable language. We use the special kind of *internal DSL*, which are languages built on top of other existing programming languages [2] – and mean this type when we speak of DSLs in the following. Languages like Ruby and Scala provide suitable semantic modifiability and syntactic flexibility to engineer DSLs.

In order to give a DSL its characteristic syntax and semantic, the support for multiparadigm programming in the host language is important. For example in Ruby, the community produced several internal and external DSLs with a focus on web applications. The DSLs exploit Ruby's support for imperative, functional, and object-oriented programming. And based on this foundation, we can bring other paradigms to Ruby too. We designed *rbFeatures* [3], a DSL that enables feature-oriented programming [4]. This DSL works by using functional programming to put code belonging to a feature inside feature containments that are only evaluated if their corresponding feature is activated.

In the course of our research into DSL utilization, we designed a Ruby web application that consists entirely of DSLs. From the presentation layer with its HTML and CSS, down to database abstractions, and further using rbFeatures to express layer-spanning features, the application consists entirely of interwoven DSL expressions. We eventually discovered *multi-DSL applications*: Applications in which every layer, concern, and artifact is expressed through a DSL. Since all DSLs are based on the same language, we can integrate the different concerns with interwoven multiparadigm expressions. This greatly reduces the complexity of writing web applications.

Our central motivation for this article is to sum up our experiences and reflect upon the development of multi-DSL applications. We illustrate such applications with a web application case study in the context of Twitter, a collaborative microblogging platform. We will explain how different domains and application layers (data, logic, presentation) are integrated using a mix of imperative, functional, object-oriented, and feature-oriented expressions.

We assume our readers to be familiar with HTML, CSS, SQL, application layers, and web application basics (URL handling, query parsing, template rendering). To stay concise, we will only briefly introduce the application and the DSLs. We then focus one particular use case with the application to show the expression integration, and explain our findings and experiences with multi-DSL applications in software development.

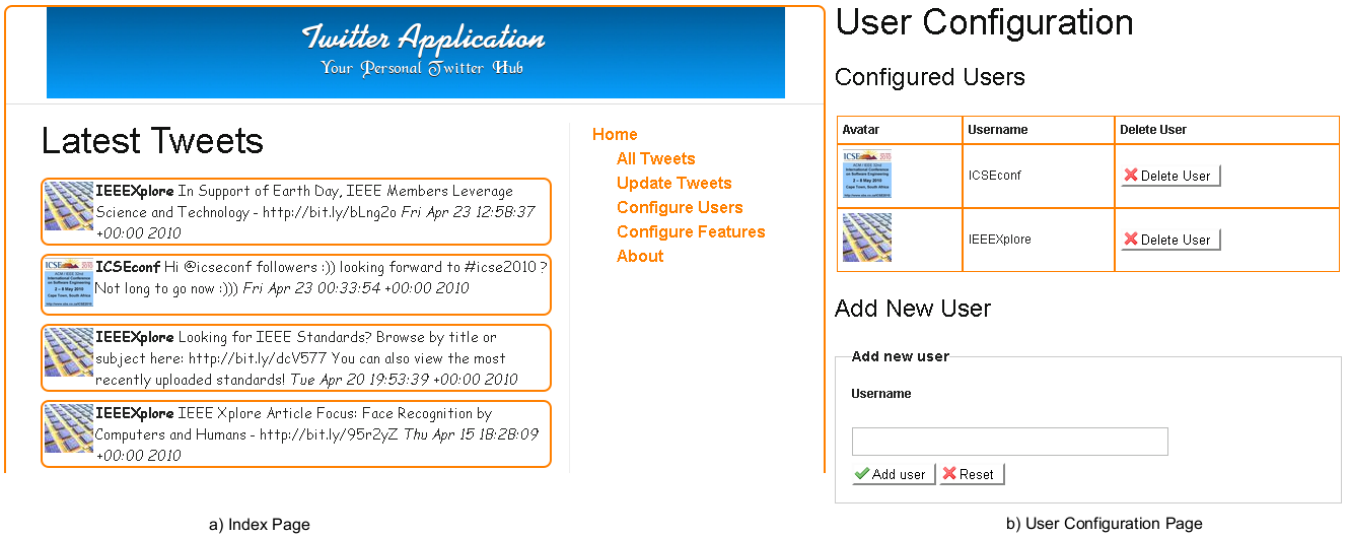
Important keywords are shown in *italics*, while DSL entities and expressions receive a `verbatim` format.

II. TAP - Twitter Application

In Twitter, users exchange small posts (also called tweets) with a maximum of 140 characters. Twitter's well-documented API[1] provides nearly the same functionality as its web-frontend: User-details can be read or updated, tweets created and destroyed. Our Twitter Application (TAP) is a hub to collect users and their posts. In its current form, TAP has the following features:

- Users

[1] http://apiwiki.twitter.com/



a) Index Page

b) User Configuration Page

Fig. 1: TAP – Page Templates

  - Add new users
  - Delete existing users
- Tweets
  - View all/latest ten tweets
  - Update tweets for registered users

We divide TAP into data, logic, and presentation. TAP stores users and tweets in a local database. Communication with the Twitter API only occurs when users are created or their tweets are updated. ►Figure 1 a) shows the index page. The page shows a banner at the top, has a large section at the left side for the tweets, and a navigation bar to the right. The user-configuration dialog in ►Figure 1 b) shows a list of users. Users can be added by inserting their username, and existing users can be deleted together with their corresponding tweets.

TAP is written in a single file with 338 lines of code. Roughly one half is for the data and logic layer, and the other half for the presentation layer with embedded HTML and CSS. The application can be tried at http://tap.admantium.com, and its complete source code is available at http://tap.admantium.com/code.

TAP is a multi-DSL application using five DSLs. Four of them are available in the Ruby community. We have chosen the following DSLs because they facilitate minimal expressions and are mature and proven in web applications.

- *DataMapper*[2] – Provides a declarative DSL to express the mapping of object properties to the database. Properties are defined with specific types and options (primary or composite keys, default values, lazy-loaded). The created objects obtain a rich set of methods to create, read, update, and delete their persistent representation in the database.
- *Sinatra*[3] – The lightweight web-framework is the backbone of our application. Sinatra supports HTTP methods (get, post, put, delete), request handlers, route declarations, filters, and delivering of static files. Sinatra is easy to customize with database handlers, different template languages, and arbitrary custom classes.
- *HAML*[4] – An external DSL that generates HTML. HAML expressions are terse and concise in syntax, because tags are expressed by their names and code indentation replaces manual tag closing. Layouts are separated into a basic file and several templates.
- *SASS*[5] – An external DSL for CSS with an indentation-based syntax to express the nested hierarchies of CSS properties. SASS extends the CSS language with two features: Variable declarations for individual values and mixin declarations for sharing grouped properties.

We designed the fifth DSL *rbFeatures* on our own. rbFeatures brings the paradigm of feature-oriented programming [4] to Ruby. Directly in the source code, Developers use a terse syntax to express which parts of an application (modules, classes, functions, expressions) belong to a certain feature. The features are first-class entities of the applications. When their activation status changes, the behavior of the application changes accordingly. rbFeatures is extensively explained in [3], and the used DSL development process is explained in [5].

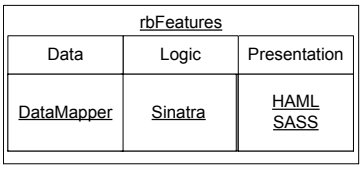


Fig. 2: TAP – Application Layers and used DSLs

How the DSLs are utilized within each layer is shown in ►Figure 2: We see that DataMapper supports the data, Sinatra

[2]http://datamapper.org
[3]http://sinatrarb.com
[4]http://haml-lang.com
[5]http://sass-lang.com

the logic, and HAML and SASS the presentation layer. On top, rbFeatures influences all parts. Now we continue with a use case.

III. Use Case: Add a new User

We can not cover all of TAPs features. Instead we focus on one specific use case: The creation of a new user. Successfully adding a new user begins at the user configuration dialog and involves the following steps:

- Fill in the username field
- Invoke a HTTP post "/adduser" with the username as a form parameter
- Connect to the Twitter API and return user details
- Create a new user instance with login and avatar; store in database
- Connect to the Twitter API and return last 200 tweets
- For each tweet, construct tweet object with id, text, date, and user (foreign-key relationship to login); store in database
- Invoke a HTTP get "/user_config"
- Render HTML templates & CSS

In order to explain the details how the DSLs interact and which paradigms they use for their expressions, it is important to show the relevant source code in ►Figure 3. We suggest to read the entire example for a basic understanding, and then continue to read how request handlers are defined with Sinatra, database objects stored, dynamic content in templates created, and features modify the application.

A. Part 1: Post the Username Parameter

In the user configuration dialog, we add a new username in the provided field and press the "ok" button. This computes a HTTP post request with the "/adduser" URL and the entered username as its parameter.

In Line 12 to 18 we see the relevant post request handler – an example of the Sinatra DSL. Handlers are declaratively expressed with the keywords `post` or `get` to define the corresponding HTTP handler, and they contain a body with imperative expressions detailing the handling steps. In Line 14 we see a command that integrates Sinatra and DataMapper. The latter part is a Sinatra expression that accesses the `params` hash to read the entered username. It returns a string that is passed to `User.construct`, the DataMapper constructor for a new user object. The command in Line 15 is again part of Sinatra and redirects the request to the user-configuration dialog.

And what about Line 13? This is an example how DSL interact. The rbFeatures expressions `AddUserFeature.code` means "only if the `AddUserFeature` is activated, execute the following code". If the feature is not activated, we simply ignore all requests and instead redirect to the index page. The code itself is put inside a feature containment that uses Ruby's functional programming support to create an anonymous block of code with the `do...end` notation. This containment adds a hook into the application that allows runtime modification of the behavior.

```
1  class Tap < Sinatra::Base
2    before do
3      content_type 'text/html; charset=utf-8'
4      @add_user_feature = AddUserFeature
5    end
6
7    get '/user_config' do
8      @users = User.all
9      haml :user_config
10   end
11
12   post '/adduser' do
13     AddUserFeature.code do
14       User.construct params['username']
15       redirect '/tap/user_config'
16     end
17     redirect '/'
18   end
19 end
20
21 class User
22   include DataMapper::Resource
23   property :login,  String, :key => true
24   property :avatar, String
25   has n, :tweets
26
27   def self.construct(user)
28     raw_data = Net::HTTP.get URI.parse(data_url)
29     ...
30     User.create! :login => login, :avatar => avatar
31   end
32 end
33
34 class Tweet
35   include DataMapper::Resource
36   property :id,     Integer, :key => true
37   property :date,   DateTime
38   property :text,   String, :lazy => true
39   belongs_to :user
40   ...
41 end
42
43 @@ user_config
44 - @add_user_feature.code do
45   %h2 Add New User
46   %form{:action=>'/adduser', :method=>'post'}
47     ...
48
49 @@ custom
50 table
51   :padding 2px 2px 2px 2px
52   td
53     +orange-border
54     ...
```

Fig. 3: TAP – Implementation Extract for the Use Case "Add a New User"

In summary, the Lines 12 to 18 are a mix of declarative, imperative, and feature-oriented expressions with three DSLs that address request handling, runtime behavior modification, database operations, and request redirection.

B. Part 2: Constructing User and Tweet Objects

The second step is to construct concrete objects which represent users and their tweets. Before we can instantiate them, we need to define the class they belong to. The class declarations are a hybrid of self defined methods and the DataMapper DSL that defines database properties and relationships.

Lets detail the `User` class shown in Lines 21 to 32. DataMapper provides the `property(name, type, options)` method to designate which attributes are stored in

the database. Lines 23 and 24 define the `login` and `avatar` properties, where `login` is the primary key for the user. Line 25 defines the foreign key relationship to the `Tweet` class. `Tweet`, shown in Lines 34 to 41, has three properties. The `id` is its primary key, the `date` stores the time, and `text` the message of the tweet. The last property has the option `lazy`, which means that the instances retrieved from the database have an empty text field. Only when this field is accessed by the application, the data is fetched from the database.

We augment the property and relationship declaration with custom constructors because we interact with the Twitter API. The user constructor is defined in Line 27 to 31. In Line 28, TAP invokes a HTTP get request to the Twitter API and receives some data. We parse and access this data to assign correct `login` and `avatar` values. Inside this abbreviated method, we use the data to construct all `Tweet` instances which belong to the user too.

We see how DataMapper helps developers to express common concerns with a terse syntax. The declarative expressions will generate attribute assessors and writers for the properties as well as designating the primary key or lazy loaded properties. DataMapper also provide the required functionality of reading, updating, and deleting objects. We just extend the objects with custom constructors to communicate with the Twitter API, and thus fulfill all requirements for TAPs data layer.

C. Part 3: Rendering the User-Dialog Template

After the objects have been created, we are redirected to the user-configuration dialog. We focus the explanation on the DSL interaction, and for brevity we omit the listing of generated HTML and CSS code.

Data Preparation in Sinatra

Sinatra supports filters that are executed before each request and response, typically to prepare common data needed in the handlers. In TAP's case, we use the before filter in Lines 2 to 5 for two things: To configure the response content header as HTML (Line 3) and to define the `@add_user_feature` instance variable as a pointer to the corresponding feature (Line 4). This variable plays an important role in the template.

Afterwards, the HTTP get request handler is invoked in Lines 7 to 10. It has the same declarative and imperative style as the post handler we explained before. In Line 8, we see the creation of an instance variable called `@users` that receives the value of the DataMapper expression `User.all`. This expressions internally transforms to the SQL statement "`SELECT * FROM users;`" and returns an array of `User` objects. The next Sinatra-DSL expression in Line 9 calls HAML to process the `user_config` template.

This example shows how the DataMapper and the Sinatra-DSL interact by using the commands they provide side-by-side to create object that are further processed in the template.

Template Call

The `user_config` template is expressed with the HAML DSL in Line 43 to 47. The template is directly included in the application file because Sinatra supports inline-templates. We see how an HTML `<h2>` tag with the title "Add a new user" is defined in Line 45. The following line starts the definition of the fieldset containing a form to submit new users – please see ►Figure 1 b) for the result. Obviously, this part of the application belongs to the `AddUserFeature`. We express this relationship in Line 43, where we use an HAML inline expression which contains an rbFeatures expression. The `@add_user_feature` variable is needed in this context because the HAML template is executed in another scope that does not see the `AddUserFeature`. But the logic stays the same: All following parts of this template are only shown if the feature is activated. Although HAML is an external DSL, it allows entering arbitrary Ruby code when the suffix "-" is added to the expression. The semantics of both DSLs neatly integrate into each other, and we combine declarative HTML with feature-oriented expressions.

Additionally to the HTML template, we also use SASS for declaring the CSS code. For completeness, we included a small declaration of SASS in the Lines 49 to 54, which shows how indentation helps to express properties for very specific entities. At this point the padding and color properties for the user table are defined. The `+orange-border` is a mixin that uses a previously defined group of properties to add them to the current element declarations.

IV. Multi-DSL Applications and Software Development

Multi-DSL application do not only influence the programming style, but have an impact on the software development phases *analysis*, *design*, and *implementation and testing* too. We explain them in the following.

A. Analysis

Domain engineering is the task to identify the domains and subdomains, concepts and operations where the application is deployed into. The result usually has the form of a domain model, and in general leads to cumulative domain knowledge. This knowledge is important to understand the basic terminology of the domain and to elaborate the applications requirements.

We see a sophisticated DSL as an executable domain model, because the domain vocabulary is present in the form of constants, modules, classes, and functions. The DSL abstracts from the domain and from the used host-language alike. Furthermore, implicit assumptions of the domain are explicitly present in the language. Thus, we propose to learn a DSL in order to understand a domain.

We used Sinatra to familiarize ourselves with the concepts of request, response, and HTTP methods. Because these domain concepts are expressed as language entities, we understood their application in the particular Sinatra context right away. Without the mature and proven DSL, we would have possibly misunderstood the domain and implemented a solution with wrong or missing abstractions.

Another advantage is that a DSL is often used in connection with other DSLs. So, starting from one particular domain, we are brought to another, equally important domain and can incorporate the novel knowledge too.

In total, using multiple DSLs for analysis helps to form the overall vocabulary, to understand and to contrast assumptions better as they are explicitly formulated in the languages, and are possible guided to other important domains. This approach can significantly boost the learning process.

B. Design

Software design receives several impulses when using DSLs. We see modifications to *requirements*, *application architecture*, *reusing specifications*, and *continuous refinement of domain knowledge*.

Choosing a DSL satisfies many functional *requirements* from the start away. DataMapper for example is a well-tested and documented solution for any database interaction or data layer in web applications. With the simple property declaration we explained before, we obtain a continuously updated database scheme, foreign key relationships including composite keys, and adapter to Sqlite3, MySQL, and PostgreSQL.

DSLs also influence the *application architecture*. As one of our recent reports showed, DSLs are designed with different patterns, like language modules or internal interpreter [5]. Patterns are related to other patterns, and using one may lead to another. Sinatra for example uses a minimalistic model-view-controller pattern, supporting a fixed set of templates, but being open for any modification. We could hook a proxy object for view provisioning, forwarding to other template engines or even other applications easily. From an expression composition perspective, we could also facilitate DSL integration by combining them in common patterns and use them in our applications.

Some DSLs have such a high abstraction level that they can express specifications as executable code. This facilitates *reusing specification* as implementation artifacts. That's the case with DataMapper: A model design is the database scheme at the same time.

While designing the application, we usually analyze the tools we use more closely. The extended study of the DSL can lead to *continuous refinement of domain knowledge*. The more mature a DSL is, the more executable domain knowledge it presents and the more complete coverage of a particular domain is provided. For example, we see the need to structure our model objects in the form of a tree. The close study of DataMapper not only shows that tree structures can be enabled for our entities, but that several plug-in exist. Another point are domain specific errors which help to refine assumptions about the domain. For example the HAML DSL refuses to add attributes to entities which are not defined within the HTML specification.

To facilitate this learning, developers can use a specific form of proactive testing. The idea is to write and execute unit tests that reflect developers assumption about a programs' behavior [6]. In the context of DSLs, this means to test the supposed domain properties and behavior as implemented by the language. Successive writing of correct tests refines the developers understanding of the DSL and domain alike.

C. Implementation and Testing

Implementation and testing are also modified if DSLs are used. We separate the explanation into *programming*, *interaction and integration*, *abstraction*, *expression integration*, *separation of concerns*, and *testing*.

One benefit of DSLs is the concise, precise and domain-specific utilization of code while *programming*. All domain knowledge literally leaks out of the language. Developers are forced to understand the domain, have lesser options to err, and write smaller amount of code. DSLs "live" from the vivid integration and intersection of statements and thus are more open to adopt to unforeseen utilization environments. The same mechanisms used for an DSL's creation can be used for its runtime modification. For example, if we want to customize the `create!` method of DataMapper, we can alias the method and execute some code before or around the database call, e.g. to inform an API or to log this event.

DSLs can improve the *interaction and integration* of code among architectural layers and subdomains. TAP uses two layer-independent domain models: User and Tweet. They are created once at the database layer, but accessible as the very same object at the logic and presentation layer. If we decide to add another property or to delete an existing one, modifications mostly hit the data layer and only small parts within other layers. This is feasible for removing code duplication.

The TAP case study showed how multiple DSLs can effectively collaborate to express concerns of different domains as interwoven expressions. This combination further facilitates the *abstraction* of our software. We stop thinking in terms of structural patterns like proxies and gateway to satisfy the requirements, but think about how to combine our DSLs. Ultimately, many artifacts of our application will find their representation as code, even whole application layers become constructs of abstractions. We can generalize this to represent whole layer, components, and even systems as domain models. When we integrate components or systems via a DSL they offer, we speak of *expression integration*. This concept enables new options of system interaction. Structural entities such as components or systems, and workflow entities such as methods and functions become integrateable to form arbitrary systems.

Separation of concerns is an important principle of how to separate software functionality into different composable parts. DSL help to pinpoint concerns: They are implementing a set of concerns in the languages" implementation and they offer a set of concerns in the DSL expressions. Combining multiple DSLs means also to combine multiple concerns. Contrast this to an application in which all concerns are tangled inside the applications architecture. By using multiple DSLs, we are focusing concern refactoring inside DSL expressions or the DSL interaction which are possibly easier to do then in the application architecture.

Finally, we emphasize the importance of *testing*. The flexible composition mechanisms and expression integration capabilities of DSLs may provide new pitfalls to developers. Rigorous test-driven approaches guard against unwanted errors. Ideally, the whole development process of a multi-DSL application is test-driven and agile. Reusable specification artifacts are created, implemented, and validated with extensible DSLs. The application is developed feature by feature. Tests provide the necessary stability, and constant refactoring the code quality.

V. Summary

Ruby is a multiparadigm programming language natively supporting the imperative, functional, and object-oriented paradigms. Additionally, more paradigms can be built on top of Ruby, like we did with rbFeatures that enables feature-oriented programming. The support for extensible multi-paradigm programming is a key point for Domain Specific Languages. With this background, we presented a multi-DSL application in the context of web applications. The application consists of integrated and interwoven expressions of five DSL that use four paradigms. Multi-DSL applications facilitate expressing every layer, concern, and artifact based on one common programming language. In addition to the case study, we discussed the impact of such an approach on analysis, design, implementation, and testing. In summary, Ruby allows effective multiparadigm and multi-DSL programming that facilitates abstraction and separation of concerns using terse and concise expressions.

Acknowledgements

We thank Christian Kästner, Maximilian Haupt, Matthias Splieth, and the anonymous reviewers for comments on an earlier draft.

Bio

Sebastian Günther is a PhD student working with the Very Large Business Applications Lab, School of Computer Science, at the Otto-von-Guericke University Magdeburg. His research interest is into internal DSL engineering, DSL utilization in application development, and metaprogramming. He developed several applications and DSLs to extend the internal language frontier. Sebastian Günther received his Diploma (similar to M.Sc.) in Business Information Systems at the Otto-von-Guericke University Magdeburg.

References

[1] P. Hudak, "Modular Domain Specific Languages and Tools," in *Proceedings of the 5th International Conference on Software Reuse (ICSR)*, P. Devanbu and J. Poulin, Eds., 1998, pp. 134–142.

[2] M. Mernik, J. Heering, and A. M. Sloane, "When and How to Develop Domain-Specific Languages," *ACM Computing Survey*, vol. 37, no. 4, pp. 316–344, 2005.

[3] S. Günther and S. Sunkle, "Feature-Oriented Programming with Ruby," in *Proceedings of the First International Workshop on Feature-Oriented Software Development (FOSD)*. New York: ACM, 2009, pp. 11–18.

[4] C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," in *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, ser. Lecture Notes in Computer Science, vol. 1241. Berlin, Heidelberg: Springer, 1997, pp. 419–443.

[5] S. Günther, "Agile DSL-Engineering and Patterns in Ruby," Otto-von-Guericke-Universität Magdeburg, Technical report (Internet) FIN-018-2009, 2009.

[6] R. C. Martin, *Clean Code - A Handbook of Agile Software Craftsmanship*. Upper Saddle River, Boston, Indianapolis et al.: Prentice Hall, 2009.