Multi-DSL Applications with Ruby

Sebastian Günther, *University of Magdeburg*

Exploiting Ruby's support for the imperative, functional, and object-oriented paradigms, several DSLs' integrated and interwoven multiparadigm expressions can express all concerns, application layers, and artifacts of an application.

Today's Web applications are language intensive. Web applications use chunks of HTML, cascading stylesheets (CSS), JavaScript, XML, Java Script Object Notation, SQL, and a server language (Java, Ruby, or Scala) for implementation. Add Web services, protocols, authentication, and sessions as further concerns, and a complex mix emerges. How can developers design and implement applications satisfying complex requirements and using different target languages?

Domain-specific languages (DSLs) use domain-specific notations and abstractions to express domain knowledge as an executable language.[1] *Internal DSLs*, which is what I'm discussing when I use the term "DSL" in this article, are languages built on top of other existing programming languages.[2] Languages such as Ruby and Scala provide suitable semantic modifiability and syntactic flexibility to engineer DSLs.

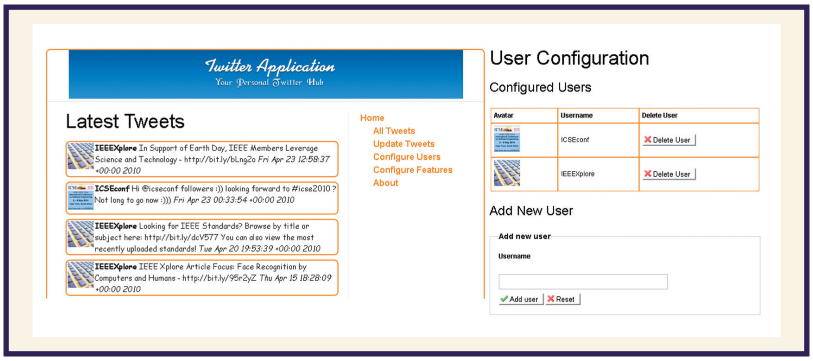
To give a DSL its characteristic syntax and semantics, the support for multiparadigm programming in the host language is important. For example, in Ruby, the community produced several internal and external DSLs with a focus on Web applications. The DSLs exploit Ruby's support for imperative, functional, and object-oriented programming. Based on this foundation, other paradigms also suit Ruby. I designed *rbFeatures*,[3] a DSL that enables feature-oriented programming.[4] This DSL uses functional programming to put code belonging to a feature inside feature containments that the system evaluates only if it activates their corresponding feature.

In the course of my research into DSL usage, I created a Ruby Web application made up 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 comprises interwoven DSL expressions. I eventually discovered *multi-DSL applications*, applications in which developers express every layer, concern, and artifact through a DSL. Because all DSLs are based on the same language, integrating concerns with interwoven multiparadigm expressions is possible. This greatly reduces the complexity of writing applications.

This article illustrates multi-DSL applications with a Web application case study in the context of Twitter (the collaborative microblogging platform). Through this article, this example explains how concerns, application layers (data, logic, and presentation), and artifacts are expressed using a mix of imperative, functional, object-oriented, and feature-oriented expressions.

TAP Twitter Web Application

In Twitter, users exchange small posts (also called tweets) with a maximum of 140 characters. Twitter's well-documented API (http://apiwiki.twitter.com) provides nearly the same



Latest Tweets

IEEEXplore In Support of Earth Day, IEEE Members Leverage Science and Technology - http://bit.ly/bLng2o *Fri Apr 23 12:58:37 +00:00 2010*

ICSEconf Hi @icseconf followers :)) looking forward to #icse2010 ? Not long to go now :))) *Fri Apr 23 00:33:54 +00:00 2010*

IEEEXplore Looking for IEEE Standards? Browse by title or subject here: http://bit.ly/dcV577 You can also view the most recently uploaded standards! *Tue Apr 20 19:53:39 +00:00 2010*

IEEEXplore IEEE Xplore Article Focus: Face Recognition by Computers and Humans - http://bit.ly/95r2yZ *Thu Apr 15 18:28:09 +00:00 2010*

User Configuration

Configured Users

Avatar	Username	Delete User
	ICSEconf	Delete User
	IEEEXplore	Delete User

Add New User

Add new user

Username

Figure 1. Twitter Application (TAP). The TAP Web application is a hub for collecting Twitter users and their posts: the (a) index page and (b) user configuration page.

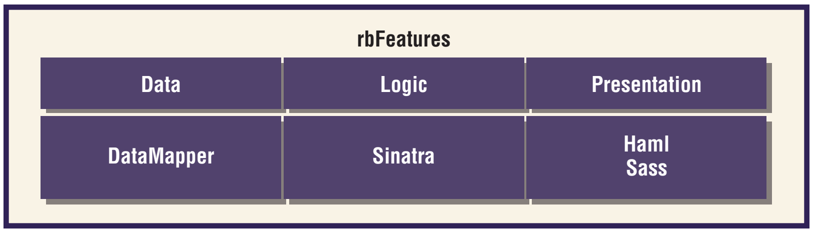


Figure 2. Application layers and used DSLs. DataMapper supports the data, Sinatra the logic, and Haml and Sass the presentation layer. On top, rbFeatures influences all parts.

functionality as its Web front end: users can read or update details and create or destroy tweets. My Twitter Application (TAP) is a hub for collecting users and their posts in a local database.

Communication with the Twitter API occurs only when the new users are added or their tweets updated. As Figure 1a shows, the index page has a banner at the top, a large section on the left for tweets, and a navigation bar to the right. The user-configuration dialog in Figure 1b shows a list of users. Administrators can add people by inserting their username and delete users together with their corresponding tweets.

TAP is 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 is available at http://tap.admantium.com, with complete source code available at http://tap.admantium.com/code.

TAP is a multi-DSL application using five DSLs, four of which are available in the Ruby community. I've chosen the following DSLs because they facilitate minimal expressions and are mature and proven in Web applications:

- *DataMapper*. The DataMapper (http://datamapper.org) DSL declaratively expresses object mapping properties to the database. Developers define properties 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*. The lightweight Sinatra (http://sinatrarb.com) Web framework is the backbone of my application. Sinatra supports HTTP methods (get, post, put, delete), request handlers, route declarations, filters, and delivering static files. Sinatra is easy to customize with database handlers, template languages, and arbitrary custom classes.
- *Haml*. An external DSL, Haml (http://haml-lang.com) 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 separate into a basic file and several templates.
- *Sass*. An external DSL for CSS, Sass (http://sass-lang.com) has an indentation-based syntax to express the nested CSS property hierarchies. Sass extends the CSS language with two features: variable declarations for individual values, and mixin declarations for sharing grouped properties.

The fifth DSL, rbFeatures[3] brings the paradigm of feature-oriented programming to Ruby.[4] 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. Features are first-class entities in TAP. When their activation status changes, the application's behavior changes accordingly. For more on DSL's development, see my earlier work.[5] Figure 2 shows the DSLs used in the application layer.

Use Case: Add a New User

Because of space limitations, I can't cover all of TAP's features. So, I'll focus the use case (and feature) of creating a new user. Successfully adding a new user begins at the user configuration dialog and involves the following steps:

1. Fill in the username field.
2. Invoke an HTTP post "/adduser" with the username as a form parameter.
3. Connect to the Twitter API and return user details.
4. Create a new user instance with login and avatar; store it in the database.
5. Connect to the Twitter API and return the last 200 tweets.
6. For each tweet, construct a tweet object with id,

text, date, and user (foreign-key relationship to login); store it in the database.
7. Invoke a HTTP get "/user_config".
8. Render HTML templates and CSS.

To explain in detail how the DSLs interact and which paradigms they use for their expressions, it's important to show the relevant source code in Figure 3. I suggest reading the entire example for a basic understanding, and then continue to read how Sinatra defines request handlers, stores database objects, and creates dynamic content, and how features modify the application.

Part 1: Post the Username Parameter

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

In Lines 12 to 18 is the relevant post request handler—an example of the Sinatra DSL. TAP declaratively expresses handlers with the keywords `post` or `get` to define the corresponding HTTP request. Handlers contain a body with imperative expressions which are executed upon the request. In Line 14, a command integrates Sinatra and DataMapper. `.construct`, the DataMapper constructer for a new `User` object, receives the Sinatra expression that reads the passed HTTP post parameters. The command in Line 15 is again part of Sinatra and redirects the request to the user-configuration dialog.

The user construction is included inside an rbFeatures DSL expression. Line 13 expresses "only if the `AddUserFeature` is activated, construct a new user." Otherwise, the request will redirect to the index page. The user construction is included in 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 behavior modification.

In summary, 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.

Part 2: Constructing User and Tweet Objects

The second step is to construct concrete objects that represent users and their tweets. Before instantiating them, the developer must define their class. The class declarations are a hybrid of self-defined methods and the DataMapper DSL that defines database properties and relationships.

```
01.    class Tap < Sinatra::Base
02.      before do
03.        content_type 'text/html; charset=utf-8'
04.        @add_user_feature = AddUserFeature
05.      end
06.
07.      get '/user_config' do
08.        @users = User.all
09.        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.          ...
```

Figure 3. TAP source code. Implementation extract for the use case "add a new user."

Multi-DSL applications influence programming style.

Looking at the User class in Lines 21 to 32 in detail, DataMapper provides the property method to designate which attributes the system stores 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 tweet's message. The last property has the option lazy, which means that the instances the system retrieves from the database have an empty text field. Only when the application accesses the lazy field, does the system fetch its content from the database and store it with the instance. Augmenting the property and relationship declaration with custom constructors is necessary because of the interaction with the Twitter API. The user constructor is defined in Lines 27 to 31. In Line 28, TAP invokes an HTTP get request to the Twitter API and receives some data. The data is used to construct a User with correct login and avatar fields as well as construct all Tweet objects that belong to the user.

DataMapper helps developers express common concerns with a terse syntax. The declarative expressions will generate attribute assessors and writers for the properties, as well as designate the primary key or lazy loaded properties. DataMapper also provides the required functionality of reading, updating, and deleting objects. TAP extends the objects with custom constructors to communicate with the Twitter API, and thus fulfills all requirements for the data layer.

Part 3: Rendering the User-Dialog Template

After the objects have been created, TAP redirects to the user-configuration dialog. I focus explaining the DSL interaction and, for brevity, omit listing generated HTML and CSS code.

Data Preparation in Sinatra. Sinatra supports filters that execute before each request and response, typically to prepare common data the handlers need. In TAP's case, I use the before filter in Lines 2 to 5 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.

Afterward, Lines 7 to 10 invoke the HTTP get request handler, which has the same declarative and imperative style as the post handler. In Line 8, an instance variable called @users 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 expression in Line 9 calls Haml to process the user_config template. This example shows how the DataMapper and the Sinatra-DSL interact side by side by using commands to create objects that the system further processes in the template.

Template Call. The Haml DSL in Line 43 to 47 expresses the user_config template. The application file includes the template because Sinatra supports inline-templates. Line 45 defines an HTML <h2> tag with the title "Add New User." The following line starts the definition of the fieldset containing a form to submit new users (see Figure 1b) for the result. Obviously, this part of the application belongs to the AddUserFeature. Line 43 expresses this relationship, using a Haml inline expression that contains an rbFeatures expression. The @add_user_feature variable is needed in this context because the Haml template executes in another scope that doesn't see the AddUserFeature. But the logic stays the same: TAP shows the fieldset and forms only if the feature is activated. Although Haml is an external DSL, it allows entering arbitrary Ruby code when the expression includes the suffix "-". Both DSLs' semantics neatly integrate into each other, so I can combine declarative HTML with feature-oriented expressions.

In addition to the HTML template, I also use Sass for declaring the CSS code. For completeness, I included a small Sass declaration, which shows how indentation helps express properties for very specific entities. At this point, the system defines user table's padding and color properties. The +orange-border is a mixin that uses a previously defined group of properties to add them to the current element declarations.

Multi-DSL Applications and Software Development

Multi-DSL applications influence the programming style as well as the software development phases analysis, design, implementation, and testing.

Analysis

Domain engineering is the task to identify the domains, subdomains, concepts, and operations where developers deploy the application. The result usually has the form of a domain model and generally leads to cumulative domain knowledge. This knowledge is important to understand the domain's basic terminology and to elaborate the application's requirements.

A sophisticated DSL is an executable domain model because the domain's concepts and functions, which also describe the vocabulary, are present in the form of constants, modules, classes, and functions. The DSL abstracts form the domain and the host language. Furthermore, implicit domain assumptions are explicitly present in the language. So, I propose learning a DSL to understand a domain.

I used Sinatra to familiarize myself with the concepts of request, response, and more HTTP protocol concepts. Because the DSL expresses these domain concepts as language entities, I immediately understood their application in the Sinatra context. Without the mature and proven DSL, I would have possibly misunderstood the domain and implemented a solution with wrong or missing abstractions.

Another advantage is that developers often use a DSL in connection with other DSLs. So, starting from one particular domain, the developer can go to another equally important domain and incorporate the novel knowledge there, too.

In total, using multiple DSLs for analysis helps form the overall vocabulary to understand and contrast assumptions better as they explicitly formulate in the languages and to consider adjoining domains right away. This approach can significantly boost the learning process.

Design

Software design receives several impulses when using DSLs: modifications to requirements, application architecture, specifications, and domain knowledge.

Choosing a DSL satisfies many functional requirements from the start. DataMapper for example is a well-tested and documented solution for any database interaction or data layer in Web applications. The simple property declaration produces a continuously updated database scheme, foreign key relationships including composite keys, and an adapter to Sqlite3, MySQL, and PostgreSQL.

DSLs also influence the application architecture. As one of my recent reports showed, DSLs are designed with different patterns, such as language modules or internal interpreter.[5] Patterns are related to other patterns, and using one might lead to another. Sinatra, for example, uses a minimalistic model-view-controller pattern. It supports a fixed set of templates but is open for any modification. I can hook a proxy object for view provisioning and forwarding to other template engines or even other applications easily. From an expression composition perspective, I could also facilitate DSL integration by combining them in common patterns and using them in my 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, in which the declarative expressions implement the data models and express the database scheme.

While designing the application, developers usually analyze the tools they use more closely. The extended DSL study can lead to continuously refined domain knowledge. The more mature a DSL, the more executable domain knowledge it presents and the more complete coverage of a particular domain it provides. For example, the need to structure model objects in the form of a tree becomes apparent. The DataMapper DSL and a plugin support this. Also domain-specific errors help refine assumptions about the domain. For example, the Haml DSL refuses to add attributes to entities that the HTML specification doesn't define.

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' assumptions 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 DSL and domain alike.

Implementation and Testing

DSLs also affect implementation and testing. The explanation addresses programming, interaction and integration, abstraction, expression integration, separation of concerns, and testing.

One of DSLs' benefits is the concise, precise, and domain-specific use of code while programming. All domain knowledge literally leaks out of the language. Developers must 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 so are more open to adapt to unforeseen environments. The same mechanisms used for a DSL's creation can be used for its runtime modification. For example, if developers want to customize the `create!` method of DataMapper, they can alias the method and execute some code before or around the database call, for example, to inform an API or to log this event.

DSLs can improve code interaction and integration among architectural layers and subdomains. TAP uses two layer-independent domain models: `User` and `Tweet`. TAP creates these models only once

A sophisticated DSL is an executable domain model because the domain's concepts and functions, which also describe the vocabulary, are present.

at the database layer, but they're accessible as the very same objects at the logic and presentation layers. If adding another property or deleting an existing one is necessary, modifications mostly hit the data layer and only small parts within other layers. This facilitates 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 my software's abstraction. We as developers should stop thinking in terms of structural patterns such as proxies and gateway to satisfy the requirements, and instead think about how to combine our DSLs. Ultimately, many of our application's artifacts will find their representation as code. We can generalize this to represent whole components, layers, and even systems as constructs of abstractions. When we integrate components or systems through a DSL they offer, we speak of *expression integration*. Structural entities, such as components or systems, and workflow entities, such as methods and functions, interact in new ways because developers implement and represent them with integrated DSL expressions.

Separation of concerns is an important principle of how to separate software functionality into composable parts. DSL helps pinpoint concerns: implementing a set of concerns in the languages implementation and offering a set of concerns in the DSL expressions. Combining multiple DSLs means also combining multiple concerns. Contrast this to an application that tangles all concerns inside its architecture. Using multiple DSLs focuses concern refactoring inside DSL expressions or the DSL interaction, which is possibly easier to do than in the application architecture.

Finally, we must 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. Developers create, implement, and validate reusable specification artifacts with extensible DSLs, developing the application feature by feature. Tests provide the necessary stability and constantly refactor the code quality.

Ruby is a multiparadigm programming language supporting the imperative, functional, and object-oriented paradigms. Further extensions with other paradigms, implemented in the form of a DSL, increase developers' capacity to structure software. Future research must detail DSL integration and interaction for special cases such as using nonorthogonal DSL (languages with possible conflicting semantics) and joint application of DSLs stemming from different host languages (the challenge of semantically integrating expressions).

About the Author

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 interests include internal DSL engineering, DSL use in application development, and metaprogramming. Günther received his Diploma in business information systems from the Otto-von-Guericke University Magdeburg. Contact him at sebastian.guenther@ovgu.de.

Acknowledgments

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

References

1. P. Hudak, "Modular Domain Specific Languages and Tools," *Proc. 5th Int'l Conf. Software Reuse* (ICSR 98), IEEE CS Press, 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, 2005, pp. 316–344.
3. S. Günther and S. Sunkle, "Feature-Oriented Programming with Ruby," *Proc. 1st Int'l Workshop Feature-Oriented Software Development* (FOSD 09), ACM Press, 2009, pp. 11–18.
4. C. Prehofer, "Feature-Oriented Programming: A Fresh Look at Objects," *Proc. 11th European Conf. Object-Oriented Programming* (ECOOP 97), LNCS 1241, Springer, 1997, pp. 419–443.
5. S. Günther, "Agile DSL-Engineering and Patterns in Ruby," tech. report FIN-018-2009, Otto-von-Guericke-University Magdeburg, 2009.
6. R.C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2009.