focus

multiparadigm programming

Separation of Concerns and Linguistic Integration in the WebDSL Web Programming Language

Danny M. Groenewegen, Zef Hemel, and Eelco Visser,
Delft University of Technology

WebDSL is a domain-specific language for Web information systems that maintains separation of concerns while integrating its sublanguages, enabling consistency checking and reusing common language concepts.

Web application development is complex, requiring developers to be polyglot, multiparadigm programmers. They must know numerous frameworks and general-purpose and domain-specific languages, such as HTML, cascading stylesheets (CSS), Java Script, Flash, Java, XML, Extensible Style Sheet Language Transformation (XSLT), Hibernate, and JavaServer Faces (JSF).

Unfortunately, integrated tool support for multilanguage development is limited. For instance, tools to check consistency between application components expressed in different languages are either lacking or nonexistent, resulting in errors and regressions developers discover only at runtime. Building integrated development environments (IDEs) for mixed languages is similarly complex. So, IDE support, such as in-place error highlighting, reference resolving, and refactoring, is generally limited in Web development IDEs. In addition, many languages require some sort of simple expression language, and they each invent their own. So, little reuse between languages occurs.

WebDSL is a domain-specific language for developing interactive Web information systems.[1] Rather than using numerous separate languages, WebDSL is linguistically integrated. It comprises sublanguages that share a common type system and cover the Web application's different aspects (see the "WebDSL's Sublanguages" sidebar). WebDSL reuses sublanguage elements, such as the expressions from the action language, in other sublanguages, such as in the user interface and access control language. The system fully checks WebDSL applications at compile time or, if the WebDSL Eclipse plug-in is in use, as the programmer writes the program.

A Closer Look at the Problem

The Java Web framework JBoss Seam is a typical example of a current Web development practice.[2] Seam uses Java to define business logic, which the Java compiler statistically checks. The application's data model is also Java-defined by annotating Java classes using Java Persistence API's annotations. While the Java compiler detects Java type

0740-7459/10/$26.00 © 2010 IEEE

errors in data models, Seam's data modeling API puts additional constraints on data model code, which the Java type checker doesn't enforce. For instance, the Java compiler doesn't detect inconsistencies in the Java annotations. Therefore, such inconsistencies appear only when the application is in use, often resulting in long, difficult-to-interpret stack traces.

Developers define user interfaces using JSF combined with HTML, CSS, and Java Script. JSF pages contain references to business logic and the data model that a compiler doesn't check; the system reports such errors only at runtime. Data model and logic references are expressed in *Expression Language* (EL), which looks like Java syntactically but has different semantics. For instance, the EL expression `e.name`, rather than accessing the `e` variable's name field (the expected Java behavior), calls the `e` variable's `getName()` method. Similarly, the == operator, rather than performing an equality check on the basis of references as Java does, uses the equals method to compare objects. To define access control rules, developers use a specialized JBoss Rules language that contains references to the data model and user interface. Again, inconsistencies surface only at runtime.

Other Web development frameworks, including Ruby on Rails[3] and Django,[4] have similar problems. Both frameworks rely heavily on metaprogramming techniques, giving the developer the impression of working with a dedicated domain-specific language, while in fact writing code in Ruby or Python. When developers make mistakes in programs, they most often discover the errors late; moreover, the messages aren't domain-specific and are difficult to trace back to their origin.[5]

Linguistic Integration in WebDSL

WebDSL's support for multilanguage integration features a checker that performs many cross-language checks:

- Do referenced properties in the user interface exist? Check the consistency between data model and user interface definitions.
- Do the pages, templates, and actions for which the developer defined access control exist, and do their arguments match? Check access control and user interface definitions.
- Do data model properties that access control rules reference exist? Check access control and data model definitions.
- Do actions referenced using a submit primitive from the user interface exist, and does the developer pass them the right list of arguments? Check user interface definitions and action language.

To demonstrate how WebDSL works, we'll develop a basic blogging application. The application has two entities in its data model: user and post. A user has zero or more posts, and a post links back to the user who authored it. This application's data model is as follows:

```
entity User {
  username  ::   String (id)
  password  ::   Secret
  posts     →    Set <Post>
}
entity Post {
  title     ::   String
  updated   ::   DateTime
  author    →    User (inverse = User.posts)
  text      ::   WikiText (validate (text ! = "",
                     "Text cannot be empty."))
}
```

A data model definition in WebDSL features zero or more `entity` declarations, which comprise a name and a set of properties. The three property types are *value properties* (indicated with ::), *reference properties* (->), and *compound properties* (<>). Reference and compound properties can refer to other entity types or `Set`s or `List`s thereof.

Property annotations control more detailed behavior. The `id` annotation indicates that the property is unique and that the system can use it to identify an entity instance. URLs also use the `id` annotated property. The inverse annotation represents one-to-one, one-to-many, and many-to-many relationships, automatically synchronizing the two properties. The model demonstrates this through the author property, which defines a users' `posts` property as its inverse, thereby creating a one-to-many relationship between users and posts. The `validate` property defines a data validation invariant,

Architecture

A WebDSL application comprises numerous components:

- *Data models* are compiled to JPA-annotated Java classes.
- *Pages and actions* are compiled to Java classes and methods.
- A *dispatch servlet* is generated that dispatches requests to page or action objects.
- *Configuration files* configure the used database and email servers.

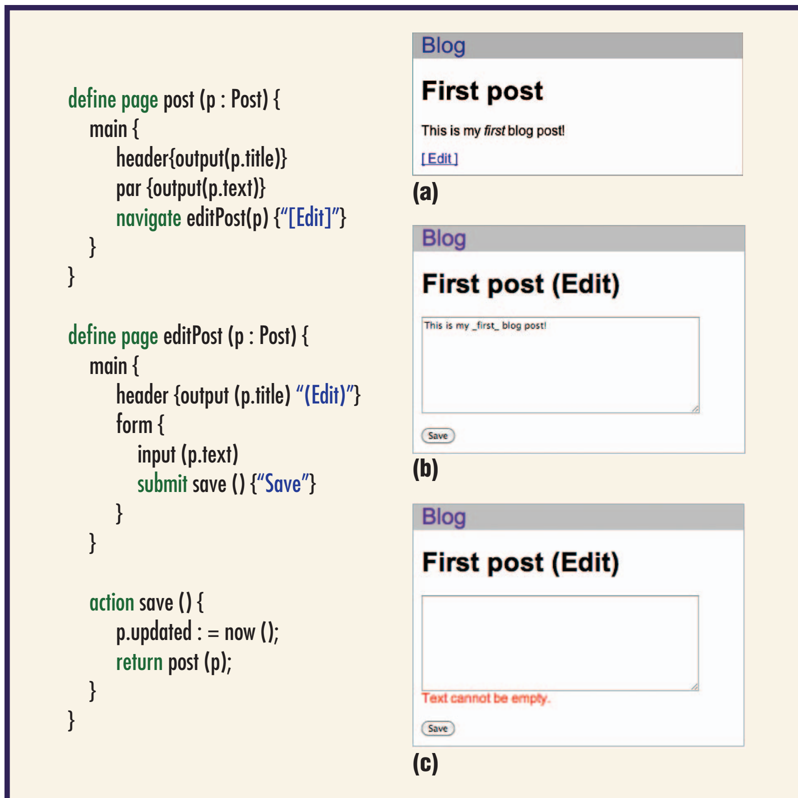


Figure 1. A WebDSL-based blog user interface. The code defines the user interface of pages for viewing and editing a single post.

ensuring that the text property is nonempty at all times. Otherwise, the user will receive a validation error message.

Figure 1 lists code that defines the user interface of pages for viewing and editing a single post. A page definition defines the page name (post and editPost), its arguments, and a definition of the page's structure. The developer defines the page's structure using a combination of template calls, such as main, header, and output, and user interface primitives, such as navigate and submit. The main template is user defined as follows:

```
define main () {
  includeCSS ("blog .css")
  top ()
  block [class : = "content"] {
    elements ()
  }
}
define top () {
  navigate root () {"Blog"}
}
```

Built-in template header renders a header, and output renders its argument type dependently. For instance, when invoked on an expression of type WikiText, it passes the field's contents through a wikitext parser before rendering it on the screen. The navigate primitive creates a link to another page in the application. The first screenshot in Figure 1 shows the post page in action.

The editPost page demonstrates WebDSL's data binding mechanism. The input template creates a text area and binds the expression p.text to it. When the user submits the form, any changes to the text in the text area automatically propagate back to the text property of p. Changes to data are part of an implicit transaction committed after request processing has completed. If an exception occurs during the request, for instance if data validation fails, the transaction rolls back.

The submit primitive creates a form submit button. When clicked it invokes the save action. The developer writes this action, defined at the end of the editPost page, using the action language. It sets the updated property to the current time and subsequently redirects the user back to the post page. The second screenshot in Figure 1 shows the editPost page in action. The third screenshot shows the result of clearing the post's text and submitting the form, activating the validation rule defined earlier. If validation succeeds, changes made to the Post p are automatically persisted.

Both the post and editPost pages in Figure 1 use a custom main template. Defining a template is similar to defining a page, except the page keyword is omitted. The main template uses includeCSS to include a CSS to style the page. It calls another custom template top and creates a block, which is a means to attach a certain CSS class or id to a page section. The special template elements() is called to inline the elements passed to the main template as body. In the post and editPost pages, the main template call wraps around other template calls, which are passed to it as elements.

Naturally, the system doesn't allow everybody to edit *any* post. The access control language provides a declarative language to specify access policies.[6] The following code defines that the system use the User entity as the access control principal:

```
principal is User with credentials username, password

rule page editPost (p : Post) {
  principal == p.author
}
```

A simple access control rule for the editPost page that uses expressions from the action language specifies that the post's author must be the principal (the logged in user)—that is, only authors can edit their own posts. If this isn't the case, the system will present the user with an access denied

page. In addition, the user interface won't show any links to inaccessible pages, created using the `navigate` primitive.

WebDSL in Practice

WebDSL initially was an exercise in domain-specific language design and implementation.[5] Today, however, developers use WebDSL to build Web applications for production.

Researchr (http://researchr.org) is the largest and most complex WebDSL application to date (see Figure 2a). Researchr is a digital library with over a million publication records. It features BibTex import and export, a reputation system, groups, a messaging system, bibliographies and tagging.

We used WebDSL to build the official WebDSL website, http://webdsl.org. It features an editable manual with revision control. TweetView (http://tweetview.net) is a Twitter archival and search tool that archives tweets about certain topics and attempts to reconstruct conversations around them. TweetView uses WebDSL for only the Web front end; it uses Java to implement the communication with Twitter and conversation reconstruction algorithms. YellowGrass, built using WebDS, is issue tracking software that we use internally in our department.

The model-driven software development course that our group teaches uses a WebDSL application that handles the organization of the course, examinations, and other processes. During this course, students use WebDSL to build a Web application.

WebDSL is optimized for constructing form-based interactive Web information systems. Our and our students' experience show that isn't well-suited for building applications that mainly rely on heavy client-side Java Script/HTML/CSS work, such as Google Docs and maps-style applications and graphical games. Although developers can use WebDSL to build these types of applications, they gain little from WebDSL's abstractions. However, using HTML and Java Script escapes lets developers use existing client-side widgets.

Figure 2. WebDSL in practice. (a) Researchr (http://researchr.org) is a digital library with more than one million publication records. (b) YellowGrass (http://yellowgrass.org) is a free Web-based issue tracker that we use it to track WebDSL bugs.

Implementation

WebDSL is available as a standalone compiler and as an Eclipse IDE plug-in. The Eclipse plug-in, shown in Figure 3, offers syntax highlighting, code folding, in-place error reporting, code navigation, compilation, and deployment. The WebDSL compiler compiles a WebDSL specification to a `.war` file, ready to be deployed to a Java servlet container such as Tomcat.

As Figure 4 shows, the WebDSL compilation process comprises parse, check, normalize, and generation phases. The parser turns textual WebDSL code into an abstract syntax tree (AST), a tree data structure the compiler uses internally. Subsequently, the system fully checks the AST for inconsistencies and reports errors to the user. If the system finds no errors, it normalizes the AST to core WebDSL, which is a smaller subset of the language.

Normalization transformations range from trivial to complex. For instance, core WebDSL only has `if` statements with `else` clauses, whereas in the "full" WebDSL language `else` clauses are optional. Therefore, a normalization in the action language translates an `if`-statement without `else` clause to an `if` statement with an empty `else` clause. The transformations that implement the access control and workflow[7] languages are examples of more complex transformations. The system

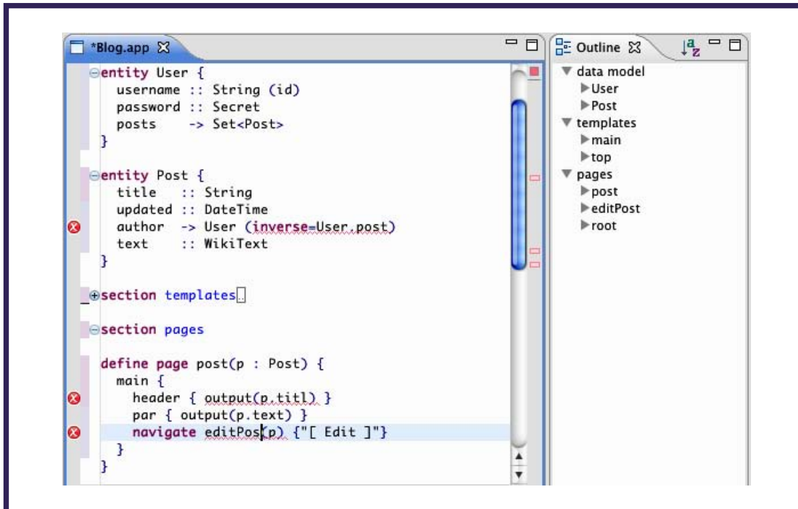


Figure 3. The WebDSL integrated development environment. The Eclipse plug-in offers syntax highlighting, code folding, in-place error reporting, code navigation, compilation, and deployment.

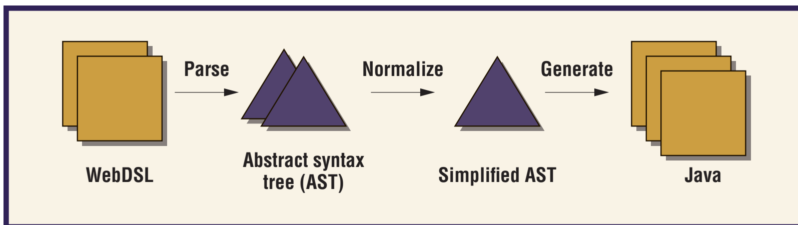


Figure 4. WebDSL compiler stages. The WebDSL compilation process comprises parse, check, normalize, and generation phases.

implements both languages as abstractions on top of core WebDSL and transforms down to core WebDSL during normalization. The access control transformation injects code into pages for each access control rule, checking if the user has access to that page. `if` statements are wrapped around links that link to protected pages, checking whether the user has access to that page and hiding it otherwise.

After applying the normalizing transformations, the compiler translates the simplified WebDSL AST into Java code (see the "Architecture" sidebar). An Ant script compiles the generated Java code and packages it as a ready-to-deploy Web archive (`.war` file).

Talking to the World

Although WebDSL incorporates many aspects of Web application programming, it doesn't have a large ecosystem of reusable libraries like Java does. Fortunately on the server side, developers can harness Java libraries through WebDSL's native Java class interface. The following example shows a native interface definition to a `TwitterReader` class in the `nativejava` package:

```
native class nativejava.TwitterReader as TwitterReader {
  static getLatest (String user) : List <String>
}
define showTweets () {
  for(s : String in TwitterReader.getLatest ("webdsl")) {
    output (s)
  }
}
```

It declares a static method `getLatest` with WebDSL types `String` and `List<String>`. This method's implementation must take care of any conversions necessary to provide the types WebDSL uses. The `showTweets` template invokes the native code and displays the results.

Whereas the native Java class interface takes care of server-side extensibility, HTML and Java Script escapes in templates enable reuse of Java Script libraries on the client. The following code illustrates these extension points with a `fadeInImage` template that displays an image with a fade-in effect using the Java Script jQuery library:

```
define fadeInImage (id : String, imageUrl : String) {
  includeJS ("jquery.min.js")
  <div id = id>
    image (imageUrl)
  </div>
  <script>
  jQuery ('~id'). hide(). fadeIn();
  </script>
}
```

HTML attribute values are normal action language expressions. To inject WebDSL expressions in Java Script code, the system uses the ~ escape.

Most WebDSL applications don't require these extension points. However, these extension points proved invaluable to applications attempting to do things that the WebDSL developers hadn't previously anticipated.

Comparison

The Web application development space has experienced rapid growth over the past years. Today, Web development frameworks exist for almost every software platform.

Internal vs. External DSLs

An *internal DSL* is a library that uses fluent interfaces and metaprogramming techniques, such as reflection and runtime code generation to provide an API that feels like a domain-specific language. Ruby on Rails is a popular internal DSL for Web development.

The advantage of implementing a language as an internal DSL is that it easily integrates with other libraries, as it shares a type system with the rest of the platform. Internal DSLs are also cheaper to develop because they're built on existing platforms, rather than from scratch. Their use avoids the need for an additional language to address a language aspect, avoiding overhead in maintaining a polyglot solution.

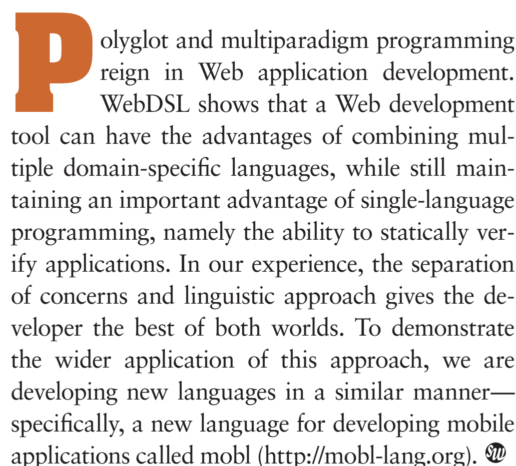
Internal DSLs have disadvantages as well. First is the lack of consistency checking in frameworks like Rails. In addition, external DSL compilers have access to the application as a model that it can analyze and manipulate. This analysis is useful for inconsistency detection, but also for optimization, aspect weaving, and generating views of the application. Also, whereas internal DSLs are tied to the syntax of their host language, external DSLs can use any syntax, typically resulting in more concise programs.

User Interface Definition

All Web frameworks have a means to define user interfaces using a template language. Template languages mix HTML and template-specific tags or escapes to another language, such as Ruby, to build the application's user interface.

Ruby on Rails uses plaintext templates with escapes to Ruby code to iterate and insert property values. JavaServer Faces uses XHTML with JSF-specific tags. The system parses and processes the XHTML, enabling manipulation of the XML tree. JSF uses this to implement databinding similar to WebDSL.

WebDSL user interfaces are a structure of template calls, primitives, and HTML escapes. WebDSL manipulates this structure internally and generates specialized code for each component and for each phase in a component. This allows multiple passes over the template structure, constituting data binding, validation, action handling, and rendering. Specialized code helps prevent security leaks—for example, only page parameters and input parameters for data binding are taken from the request parameters, anything else is shielded from post-data tampering. Users can invoke actions only through a URL on the page, where request handling automatically verifies that the action could be executed by normal usage of the page. Compare this to Ruby on Rails, in which, for example, a generic "mass assignment" operation provides a convenient way for loading all inputs into an entity object directly from the request parameters. However, it is also a security leak because post-data tampering allows setting any property in the entity.[8] To work around this issue, developers can shield a property from mass assignment in the data model definition, something easily neglected and able to break other action handlers that rely on this feature.



Polyglot and multiparadigm programming reign in Web application development. WebDSL shows that a Web development tool can have the advantages of combining multiple domain-specific languages, while still maintaining an important advantage of single-language programming, namely the ability to statically verify applications. In our experience, the separation of concerns and linguistic approach gives the developer the best of both worlds. To demonstrate the wider application of this approach, we are developing new languages in a similar manner—specifically, a new language for developing mobile applications called mobl (http://mobl-lang.org).

References

1. E. Visser, "WebDSL: A Case Study in Domain-Specific Language Engineering," *Generative and Transformational Techniques in Software Engineering II*, LNCS 5235, R. Lämmel, J. Visser, and J. Saraiva, eds., Springer, 2008, pp. 291–373.
2. S. Kittoli, ed., *Seam - Contextual Components: A Framework for Enterprise Java*, RedHat Middleware, 2008.

WebDSL's Sublanguages

WebDSL sublanguages each cover a Web application aspect:

- *User interface language* for creating pages and reusable templates.[1]
- *Data modeling language* for defining the application's data model.[1]
- *Action language* for defining the application's (business) logic.[1]
- *Access control language* for restricting access to parts of the application to specific user groups.[2]
- *Data validation language* for defining constraints on data and inputs.[3]
- *Workflow language* for defining business workflows in the application.[4]

References

1. E. Visser, "Domain-Specific Language Engineering," *Pre-Proc. Generative and Transformational Techniques in Software Eng.* (GTTSE 07), Int'l Summer School GTTSE, 2007, pp. 265–318.
2. D.M. Groenewegen and E. Visser, "Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns," *Proc. 8th Int'l Conf. on WebEngineering* (ICWE 08), IEEE CS Press, July 2008, pp. 175–188.
3. D.M. Groenewegen and E. Visser, "Integration of Data Validation and User Interface Concerns in a DSL for Web Applications," *Software Language Engineering*, LNCS 5969, M. van den Brand and J. Gray, eds., Springer, 2009, pp. 164–173.
4. Z. Hemel, R. Verhaaf, and E. Visser, "WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications," *Model Driven Engineering Languages and Systems*, LNCS 5301, K. Czarnecki et al., eds., Springer, 2008, pp. 113–127.

IEEE Software

HOW TO REACH US

Writers

For detailed information on submitting articles, write for our Editorial Guidelines (software@computer.org) or access www.computer.org/software/author.htm.

Letters to the Editor

Send letters to

Editor, *IEEE Software*
10662 Los Vaqueros Circle
Los Alamitos, CA 90720
software@computer.org

Please provide an email address or daytime phone number with your letter.

On the Web

Access www.computer.org/software for information about *IEEE Software*.

Subscribe

Visit www.computer.org/subscribe.

Subscription Change of Address

Send change-of-address requests for magazine subscriptions to address.change@ieee.org. Be sure to specify *IEEE Software*.

Membership Change of Address

Send change-of-address requests for IEEE and Computer Society membership to member.services@ieee.org.

Missing or Damaged Copies

If you are missing an issue or you received a damaged copy, contact help@computer.org.

Reprints of Articles

For price information or to order reprints, send email to software@computer.org or fax +1 714 821 4010.

Reprint Permission

To obtain permission to reprint an article, contact the Intellectual Property Rights Office at copyrights@ieee.org.

About the Authors

Danny M. Groenewegen is a PhD student in the Software Engineering Research Group at Delft University of Technology. His research interests include abstractions for the Web domain and their implementation as domain-specific languages and code generators. Groenewegen has a master's degree in computer science from the Delft University of Technology, The Netherlands. Contact him at d.m.groenewegen@tudelft.nl.

Zef Hemel is a PhD student in the Software Engineering Research Group at Delft University of Technology. His research interests include the design and implementation of (domain-specific) languages, specifically for the Web and mobile application domain. Hemel has a master's degree in computer science from Trinity College Dublin. Contact him at z.hemel@tudelft.nl.

Eelco Visser is an associate professor in the Software Engineering Research Group at Delft University of Technology. His research interests include model-driven engineering, domain-specific languages, program transformation, and software deployment. Visser has a PhD in computer science from the University of Amsterdam, The Netherlands. He is a member of the ACM Special Interest Group Programming Languages and IEEE. Contact him at visser@acm.org.

3. S. Ruby, D. Thomas, and D. Heinemeier Hansson, *Agile WebDevelopment with Rails*, 3rd ed., Pragmatic Programmers, 2009.
4. A. Holovaty and J. Kaplan-Moss, *The Definitive Guide to Django: WebDevelopment Done Right*, Apress, 2007.
5. Z. Hemel, "When Rails Fails," 2010; http://zef.me/2308/when-rails-fails.
6. D.M. Groenewegen and E. Visser, "Declarative Access Control for WebDSL: Combining Language Integration and Separation of Concerns," *Proc. 8th Int'l Conf. on Web Engineering* (ICWE 08), IEEE CS Press, 2008, pp. 175–188.
7. Z. Hemel, R. Verhaaf, and E. Visser, "WebWorkFlow: An Object-Oriented Workflow Modeling Language for Web Applications," *Model Driven Engineering Languages and Systems*, LNCS 5301, K. Czarnecki et al., eds., Springer, 2008, pp. 113–127.
8. R. Bates, "Hackers Love Mass Assignment," 2007; http://railscasts.com/episodes/26-hackers-love-mass-assignment.

Selected CS articles and columns are also available for free at http://ComputingNow.computer.org.