Mashups are flourishing on the programmable Web. Developers create mashups by combining components of existing Web sites and applications. Although the rapid mashup proliferation offers many opportunities, a lack of standardization and compatibility presents considerable challenges. To address this, the IBM Sharable Code online service platform uses an innovative domain-specific language that provides a common structure and design for mashups as well as facilities to help share and reuse mashup components.
current mashups are point solutions, lacking principled approaches to architecture and design; and

many similar mashups use the same APIs but share nothing else in common.

However, the mashup explosion also offers considerable opportunities. Just as Web 2.0 social features have led to collaborative content creation, developers can collaborate on mashups, or at least share parts. The availability of data as service, coupled with collaborative application development, can help realize the vision of the open programmable Web.

With such challenges and opportunities in mind, we developed IBM Sharable Code (née Swashup), an online mashup platform that offers a principled approach to designing mashups, sharing their parts, and facilitating their management (see http://services.alphaworks.ibm.com/isc). Using our solution, developers can also use mashups to solve Web integration and service composition problems. As we discuss later, an informal comparison shows that our platform performs better, in some aspects, compared to other leading mashup platforms.

Motivation
To illustrate the programmable Web paradigm shift, we'll use two abbreviated use cases. The first is in Web 2.0, or the social Web realm, where end users collaborate to create content and value. The second is within the so-called Enterprise 2.0, marked by the move of Web 2.0 concepts into the enterprise realm.

Web 2.0 Use Case
In the new social Web, users contribute, comment on, tag, rank, and aggregate content, which is often exposed in RSS or Atom data feeds. Many current Web applications are also using REST and SOAP services to expose business functions. To imagine our mashups, we'll use the following three Web 2.0-style services:

- Digg.com or IBM Web Highlights (http://services.alphaworks.ibm.com/webhighlights), which use Atom data services to syndicate the most popular items as voted upon by service users;
- Flickr.com, a Web-based photo-sharing application with a REST API that lets users access photos and associated photo tags that other users in the community have created; and
- Eventful.com, an event database with information on theater, sporting events, and so on, that uses a REST API to search, find, add, and manage events. Users can also add tags to each event.

Given these services, we can create a variety of mashups. A simple one would be to build an end-user interface to find events using location and keyword input, such as “Silicon Valley” for location and “book signings” for keywords. With the Eventful events data and each event’s tags and title information, we can also find Flickr.com photos and Digg.com news items (from the Atom feed) tagged with same keywords. Essentially, our mashup offers a simple, integrated, and automated view of information in three different Web 2.0 services. Also, because the data and APIs are live, the displayed information is constantly updated.

Next-Generation Enterprise IT
In the enterprise, mashups can address the idiosyncratic needs of employees and help them innovate by creating Web-based solutions to problems that might not warrant the IT division’s time and investments. As IBM’s Anant Jhingran discusses, enterprise IT is often slow to integrate and migrate to newer technologies.2 Mashups — with their easy and purposeful Web interfaces — offer an excellent venue for creating applications to support day-to-day activities for specific, recurring tasks.

For example, at IBM Research Labs, employees routinely update calendars with internal talks from various international labs, as well as from external public talks at local universities and organizations. A simple mashup might aggregate Atom feeds from various talk sources and offer employees a common location to easily update their calendars using a series of
checks and a one-click submit mechanism. We could further integrate the mashup with other services that rank talks and presenters and offer forums in which users could provide feedback on attended talks.

As another example, let’s assume a company has a complex set of legacy system Web services for order management and customer-relationship management. Although the company has integrated with existing partners, it hasn’t yet branched out to other ephemeral partners, such as those that might submit purchase orders using RosettaNet PIP 3A4 order purchasing schemas. To address these potential partners’ vocabulary and processes, our example company could create a mashup to mediate between new partners and legacy services. The mashup could transform RosettaNet messages into a format that the legacy services could comprehend, resolve choreographic conflicts with the legacy services, and provide the necessary RosettaNet confirmation output messages. Conceptually, the company could create a mashup for each new partner that has an incompatible order messaging vocabulary or proprietary message formats.

**IBM Sharable Code**

The IBM Sharable Code platform attempts to provide a comprehensive solution to creating, reusing, deploying, and managing Web API mashups. To achieve our goals, we constrain our definition of “mashup” and give them a general and common structure.

**Defining Mashups**

Conceptually, mashups are simply new Web applications that repurpose existing Web data and APIs. Well-structured mashups therefore include all three aspects of an equivalently well-designed Web application: data models, views, and interaction controllers. Also, mashups often mediate between heterogeneous providers’ Web APIs. So, for us, mashing up different Web APIs typically involves one or more of the following activities:

- **Data mediation** involves converting, transforming, and combining data elements from one or more data feeds or APIs. The mashup can thereby create a new data element or satisfy an API’s operational needs, such as determining the correct address data fields to invoke a mapping service.
- **Process (or protocol) mediation** choreographs different APIs to create a new process and present the necessary interaction points to the user.
- **User interface customization** elicits user information and displays intermittent process information to the user. This interface can be as simple as Web forms and static pages with dynamically generated content, or more complex, with rich Ajax interfaces that
refresh content or dynamically generate new pages based on user interactions.

Informed by these activity requirements, the IBM Sharable Code platform provides a language and a set of Web-based tools to facilitate construction of well-structured mashups.

**Platform Overview**

We implemented our platform as a collection of Ruby on Rails (RoR) Web applications and APIs, all backed by a relational database. Figure 1 shows a high-level platform overview with the various Web applications and APIs. Our domain-specific language (DSL) has a well-defined grammar, and we could implement it using some other language or its own parser, compiler, and code-generation engine. However, we chose instead to use RoR because of its flexible syntax and dynamic features.

Our platform currently comprises three RoR Web applications. At the platform's core is the DSL engine, which takes a DSL-defined mashup and generates the necessary code for a complete RoR Web application. We expose the engine as Web APIs. As we now describe, the platform also contains creator and community applications that let users create and share mashups, respectively.

**Mashup Creator**

Figure 2 shows the mashup creator's main user interface. User projects can contain multiple mashups, all of which follow the DSL's structure (described in detail later). The creator tool provides basic facilities to browse the DSL, create new DSL constructs via wizards, and search and easily reuse DSL parts from either the user's own projects or those of other users. When the user searches a specific keyword, the creator provides a list of indexed items that he or she can add directly to the project. The system creates the index using the DSL construct names or the user's own provided descriptions and tags. The search term “news” yields news-related components and provides a one-click add facility.

After creating and testing a mashup, users can deploy IBM Sharable Code mashups as packaged RoR Web applications or Ruby gems (www.rubygems.org), which they can download and install on their local server or on a remote server. The only requirement is that the server has the RoR framework and a relational database such as DB2 or MySQL. All other depen-
Service Mashups

We divide related works into two main categories: mashup tools and frameworks and domain-specific languages (DSLs).

Tools and Frameworks
Yahoo Pipes (http://pipes.yahoo.com) is an example of a mashup tool available on the Web. Pipes provides a graphical interface that lets users combine data feeds into other feeds. The user interface is quite sophisticated and uses advanced Ajax features to make it responsive and intuitive.

IBM’s Damia (http://services.alphaworks.ibm.com/damia) is available online as part of IBM’s MashupHub product. Damia offers a principled approach to data mashups by exposing data operators such as extract, combine, and merge. Damia provides a structured approach to creating data feed combinations by bringing well-known relational data manipulation operators to Web-feed data.

IBM Sharable Code differs from both Pipes and Damia in that it treats mashups as more than a combination of data feeds, offering process choreography as well as user interfaces.

Unlike our platform, IBM’s QEDwiki (http://services.alphaworks.ibm.com/qedwiki) focuses principally on the user interface, letting users create Web widgets (such as JavaScript and HTML components) and connections to Web APIs and feeds. QEDwiki lets users organize their projects into workspaces and pages that they can share and reuse. Although QEDwiki provides similar functionality as IBM Sharable Code, their approaches differ. QEDwiki doesn’t expose a mashup language, nor does it let users export the resulting mashup in a packaged form that they can deploy to other servers.

Google Web Toolkit (http://code.google.com/webtoolkit) is a Java framework; its goal is to let users create Ajax Web applications as pure Java Model-View-Controller (MVC) applications. Users therefore write all aspects of a GWT application in Java. Using a compiler, the toolkit then statically compiles the Java view code into HTML and JavaScript, accounting for browser differences.

Our toolkit differs from GWT in three main respects. First, our goal is to address the service mashup construction, and not the entire MVC stack (we use Ruby on Rails [RoR] to enable the latter). Second, we use metaprogramming to compile our language constructs into RoR code, which makes it highly dynamic. Finally, we achieve view support using RoR views through HTML and JavaScript templating languages (such as Ruby HTML and Ruby JavaScript).

Domain-Specific Languages
A good example of a platform and language that developers could use to implement our DSL is the Smalltalk Seaside (www.seaside.org) framework. Like RoR, Seaside provides a complete framework for creating Web applications. However, the framework, like other Smalltalk environments, also contains tooling that lets users browse, edit, and modify the resulting Web application’s code directly in their Web browsers.

Finally, Charles Simonyi and his colleagues at Intentional Software (www.intentsoft.com) are creating a complete set of tools for WYSIWYG programming. Their approach is based on DSLs but is more generic than ours and attempts to enable DSLs for a variety of domains. Scott Rosenberg has discussed both their ambitions and directions for the Intentional Software Workbench, as well as how they use DSLs to achieve metaprogramming for different domains.

References

Related Work in Mashup Development

dependencies are either included in the application gem or downloaded when the gem installs.

Mashup Community
The IBM Sharable Code Community is an RoR Web application that shares deployed mashups. The application’s primary objective is to build a community of mashup users and create a mashup catalog. Community members can discuss and rate the mashups, and either manually add mashups to the community or have them automatically added when they deploy from the mashup creator.

Domain-Specific Language
A DSL is a “mini” language typically built on top of a hosting language that provides a common syntax and semantics to represent a particular domain’s concepts and behaviors. Generally, using or designing a DSL helps developers meet one or more key goals.

• Achieve higher-level abstraction. A DSL enables programming at a level higher than the host language constructs or libraries allow. It also lets developers represent the domain concepts, actions, and behaviors directly in the new syntax.
• Write terse code. This is a side effect of programming at a higher abstraction level. DSL programs are typically smaller than their
Web APIs and Service Mashups

equivalent in general-purpose programming languages.

- **Create simple and natural syntax.** This leads to code that’s easy to write and read.
- **Program easily.** This is desirable in any programming language and also somewhat difficult to judge. However, because DSL enables the expression of constructs that map directly to a domain, it generally makes programming easier (for domain applications) than using a general-purpose language.
- **Generate code.** This is a DSL's primary function. Essentially, the system translates DSL statements, at runtime, into code that uses the underlying language and its libraries. Developers can accomplish this using metaprogramming techniques or by generating program files that the system compiles and loads or interprets at runtime.

Language Overview

Our language’s features map to both top-level constructs from our conceptual definition and to the mashup conceptual model. Because a mashup represents new data or functionality outside what the service provider originally intended, our language explicitly represents a mashup designer’s activities, shows our language’s primary constructs, along with examples.

Figure 3 shows the various constructs of Sharable Code DSL using UML (www.uml.org) class diagrams, annotated with stereotypes we created. For example, we use the stereotype << api >>, which means that the UML element represents our language’s api construct. This doesn’t have to translate to a class in the underlying language — although it could.

DSL examples and deployed mashups are available for study, usage, discussions, and download through one IBM Sharable Code Sample Repository (http://knoesis1.wright.edu/isc).

DSL Example

To better illustrate our DSL, we'll briefly describe a sample mashup that combines Flickr photos with YouTube videos based on a user’s search information. We chose this example for three reasons: it’s simple, it shows the mashup
<table>
<thead>
<tr>
<th>Sharable Code language constructs</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
</table>
| data                             | Describes a data element used by a service. A data element can correspond to an XML schema complex type or a JavaScript Object Notation (JSON) type. Each data element has a name and a series of member attributes that can be a simple type or other data elements. Caching is enabled using `cache:policyname`. | data :Error do  
name :error  
member :code, :string, :attribute, :desc=>'description', :tag=>'err'  
cache :fifo end |
| api                              | Gives a complete description of a service’s interface, including API descriptions, such as operation names, parameters, and data types, as well as HTTP verbs and headers. An operation’s data is either a simple type (such as a string or integer) or refers to a data element or to an array of types using `[:type_name]`. | desc “ Defines the Digg REST API”  
desc “ All stories”  
:expects => []  
:returns => [:Story] |
| service                          | Binds a service API with a concrete service endpoint. Part of the binding is to indicate the service’s type — such as Representational State Transfer (REST), RSS, Atom, or AtomPub — and its endpoint, and to give an alias for the service instance. | service:digg_rest_service,:type=>:rest, :api=>:DiggApi,:alias=> :digg,:endpoint=>'http://services.digg .com’ |
| mashup                           | Consists of one or more services that comprise a collection of wiring and mediation declarations. Each mashup also translates into a composed service that can be exposed externally and used for further mashups. | mashup :digg cnn mashup do [digg, cnn]  
#step(s)  
#mediation(s)  
#wiring(s) end |
| mediation                        | Defines a mediation declaration with instances of the mediated data elements. Each mediation can be called by its name, like a method call. The result is a primitive type instance or another data element instance. | mediation :flickr_to_youtube do [f_ data, y_data]  
... end |
| wiring                           | Enables interaction with the users, which can include invocation to service operations via available callbacks. When service data caching is enabled, a wiring can also use the cached data instead of performing a remote operation call. | wiring :interesting_photos do  
# calls to step methods end |
| step                             | Constitutes one atomic step in a wiring, which can be invoked multiple times as part of a wiring. Like a method call, a step is invoked by its name. | step :combine_feeds do [stanford, ibm_w3_media]  
... end |
| view                             | Describes a mashup’s view component; usually written in HTML, with embedded Ruby code (RHTML). Other templating languages are also possible. | view :view_feed,:mashups =>[:feeds_mashup], :content => %{  
  #: RHTML view code } end |
| css                              | Contains the mashup’s CSS; CSS-defined elements can be used in all mashup views. | css :default,:mashups=>  
  [:eventful_mashup], :content => %{  
    #: CSS declarations} end |
| recipe                           | Contains, at the top level, all other DSL constructs. | recipe :eventful_digg_recipe do  
  #: all other DSL constructs end |
| desc                             | Lets users provide textual descriptions: text is indexed to enable searching. | desc “Flickr photo data” |
| tag, tags                        | Lets users tag various recipe components and add comments and idiosyncratic semantics. | tags ['errors', 'api__ error']  
tag ‘rest’ |

---

Table 1. DSL constructs and simplified examples.
desc "Recipe for YouTube AtomPub and flickr.com"

REPT mashups"

tags ['mashup', 'rest', 'atom']

recipe :youtube_flickr_rest_recipe do
  desc "Flickr's error response part"
  data :FlickrErr do
    name :err
    member :code, :string, :attribute
    member :msg, :string, :attribute
  end

  desc "Flick's <username/> data"
  data :FlickrUsername do
    name :username
  ...
end

desc "Flickr's <photo/> data"
  data :FlickrPhoto do
    name :photo
    member :id, :integer, :attribute
    # many more members
  end

# Various other Flickr data elements to
# match Flickr's XML data

desc "Flick's REST API"

tags ['api', 'rest', 'flickr']

api :FlickrApi do
  desc "Returns list of interesting photos"
  api_method :flickr_interestingness_getList,
    :http_method => :get,
    :method_name_as_path => false,
    :fixed => [{:method => 'flickr.
      interestingness.getList},
      :api_key => '...',
      :api => :FlickrApi, :type => :rest]
end

desc "YouTube APP and Atom data feed"

tags ['atom', 'app']

service :youtube_app_service,
  :type => :atom,
  :api => :AtomApi, :entry => :AtomEntry,
  :alias => :flickr,
  :endpoint => 'http://gdata.youtube.com/feeds/standardfeeds/toprated?time=today'
end

desc "Shows both YouTube and Flickr data"

mashup :combined_mashup do |youtube, flickr|
  wiring :view do
    end
  end
  wiring :search do
    end
end

desc "Flickr's REST service definition"

tags ['flickr', 'rest', 'service']

service :flickr_rest_service, :type => :rest,
  :api => :FlickrApi, :alias => :flickr,
  :endpoint => 'http://api.flickr.com/services/rest'
end

desc "Common Atom data feed author tag"
data :AtomAuthor do
  name :author
  member :name, :string, :text
  member :uri, :string, :text
  member :email, :string, :text
end

desc "Common Entry for Atom feed"
data :AtomEntry do
  name :entry
  member :id, :string, :text
  member :title, :string, :text
  member :title, :string, :text
  member :updated, :datetime, :text
  # other members
end

desc "Common Atom API"

tags ['api', 'atom']

api :AtomApi do
  api_method :metadata,
    :returns => [:AtomMetadata]
end

desc "Flickr's REST service definition"

tags ['flickr', 'rest', 'service']

service :flickr_rest_service, :type => :rest,
  :api => :FlickrApi, :alias => :flickr,
  :endpoint => 'http://api.flickr.com/services/rest'
end

Figure 4. Example recipe for a Flickr-YouTube mashup. The mashup uses a Representational State Transfer (REST) service with an Atom/AtomPub feed and API.
of a REST service with an Atom/AtomPub feed and API, and it illustrates our language’s user interface capabilities.

Figure 4 shows the DSL code for this mashup. We simplified some sections, which can be inferred from similar constructs; the complete recipe is available on our Web site. Figure 5 shows the Ruby HTML (RHTML) view to generate the photo and video listing, as well as the search form for user input. As the figure shows, the view’s name corresponds to the name of the mashup’s appropriate wiring. This, in turn, corresponds to portions of the deployed mashup’s routing. So, for a recipe named youtube_flickr_recipe, a mashup named combined_mashup, and a wiring named view, we’d construct the URL as follows: http://server-name:server-port/youtube_flickr/combined/view, where server-name is the host location and server-port is the host port.

**Evaluation**

To evaluate our platform, we analyzed the code it generated for three simple mashups of different Web API types. Our analysis is not intended to be comprehensive, but rather to offer a general idea of how much time developers can expect to save by programming their mashups in a higher-level language such as our DSL.

Table 2 summarizes the initial code generation statistics for our three simple mashups. The data suggests that using our tool lets programmers create a common, reusable representation for their mashups as well as save development time, as the average ratio of recipe to code generated is greater than 75 percent. The savings come primarily from the fact that using our DSL and tooling lets us generate the mashup’s plumbing classes, such as XML and JavaScript Object Notation (JSON) parsing, REST, Atom, AtomPub, and RSS proxy classes;

database mapping and migration classes, to move data elements into the database; and

controllers and view code, along with methods mapping to the step, mediation, and wiring constructs.
Our platform also contains Web-based tools, such as the mashup creator tool, which contains various wizards to facilitate some DSL construct generation. As Figure 6 shows, for example, developers can use the creator tool to generate various DSL constructs (including api, data, and service) for an Atom feed by simply loading the feed and selecting the data fields that the data declaration needs.

**Discussion**

Table 3 shows how our tools compare with popular mashup tools on five criteria. The data is based on our own experiences using the different tools, and thus isn't a scientific survey. However, because we're all domain experts, we do consider the results to be expert opinion on the technologies and tools.

Our language and platform have some important limitations. Some of these limits are due to the fact that we need to test our approach in many more use cases. Doing so will evolve and improve our language by allowing us to survey and study a wider audience, and thereby improve the platform’s ease of use. It should also give us insights into how easy it is for people to reuse and extend our recipes.

Other limitations are due to our general DSL-based approach. Although the platform lets users indicate limited automatic service-data caching in the DSL, this is not a generic solution. Extending the DSL's API definition con-
structs could let users indicate which services and operations data the system should cache, and which caching policy it should use.

As with any metaprogramming or code-generation approach, debugging the generated code is another challenge. One approach is to explicitly introduce debugging facilities in the generated code and let users turn debugging on or off, depending on the development stage.

Another major limitation is that our DSL doesn’t directly include primitives for user-interface descriptions. Instead, each mashup wiring implicitly associates with a view by the same name. This was a design choice; we chose the RoR platform as a base and wanted to leverage its view facilities. One advantage of this choice is that our views can be complex HTML pages, which can incorporate advanced features such as Ajax callbacks. However, this view-controller separation means that we must sometimes include logic in the view templates. Arguably, including complex logic and data manipulations in the views could defeat the DSL’s purpose and duplicate some of its content. It’s too early to say if this problem will be widespread when the tool is widely used. We might be able to mitigate this by enabling graphical-based tooling to help users customize their mashup user interfaces by hooking into the mashup recipe descriptions.

Finally, we restrict our definition of mashups to services with APIs. Our platform therefore doesn’t support direct editing and mashup of pure Web resources, such as video streams. However, we could use Web applications supporting such mashups in our platform if they expose a Web API that allows Web resource manipulation, such as YouTube.com video sharing and editing APIs.

We’ve deployed an initial version of the Sharable Code platform on IBM alphaWorks services. We plan to improve and complete the platform by updating the current toolset, testing DSL with more complex examples, and adding new features that facilitate scalable mashup deployments and deployments into social platforms.

Following the RoR philosophy of small tools written in Ruby, we plan to release platform sections as RoR generators and plugins to facilitate command-line generation of the IBM Sharable Code DSL recipe and its components. This would both encourage RoR programmers to adopt our DSL in their applications (independent of using the platform and Web user interface tools) and allow us to contribute back to the RoR community.

We also plan to provide tools for mashup management. One of the reasons to create mashups is to provide innovative solutions to business problems and distribute the solutions around the Web. As a result, there’s a need to help these mashups grow into full-blown Web applications. Providing tools that can facilitate scalable mashup deployments and automate data component caching would go a long way toward making IBM Sharable Code solutions widely available. For example, it would let users automatically deploy our mashups to compute clouds such as Amazon’s EC2 (http://aws.amazon.com/ec2) and Google App Engine (http://appengine.google.com). Furthermore, using cloud computing intermediaries such as MorphExchange (http://morphexchange.com) and Hekoku (http://heroku.com) could completely automate the cloud deployment and management process and help our mashups automatically scale as more users access them.

Finally, we’d like users to be able to deploy our mashups in social utility platforms such as Facebook and OpenSocial (http://code.google.com/apis/opensocial). Such platforms let users integrate and blend Web applications into social networking sites, which offer a rich platform and user community for mashups. Because such sites provide information about users’ friends and networks, they also open viral usage possibilities.

Acknowledgments
We thank N.C. Narendra of IBM’s India Research Lab for initial discussions on an early version of the platform. We also thank Stefan Tai of Karlsruhe University, Pietro Mazzeoli of IBM’s T.J. Watson Research Center, and Nirmi Desai of North Carolina State University for collaborating on early prototypes of the IBM Sharable Code platform tools and DSL language.

References
E. Michael Maximilien is a research staff member at IBM’s Almaden Research Center in San Jose, California. Prior to joining ARC, he spent 10 years at IBM’s Research Triangle Park in software development and architecture. His primary research interests are in distributed systems and software engineering, focusing on Web APIs and services, mashups, Web 2.0, cloud computing, service-oriented architecture, social software, and Agile methods and practices. Maximilien has a PhD in computer science from North Carolina State University. Contact him at maxim@us.ibm.com; www.maximilien.com and blog.maximilien.com.

Ajith Ranabahu is a PhD student in computer science at Wright State University’s Kno.sis Center. His research interests include situational Web applications and services for mobile computing. He is a member of the Apache Software Foundation and a major committer to the Apache Axis2 project. Ranabahu has a bachelor’s degree in computer science from the University of Moratuwa, Srilanka. Contact him at ajith@knoesis.org.

Karthik Gomadam is a PhD student in computer science at Wright State University’s Kno.sis Center, where he is student coordinator of the Services Research Lab (http://knoesis.org/srl). He was previously an intern at IBM Research in Hawthorne and in Bangalore. His research interests include Web 2.0, mobile platforms and services, and social network analysis. Gomadam has a master’s degree in mathematics and computer science from Sri Satya Sai Institute of Higher Learning. Contact him at karthik@knoesis.org.

### Web APIs and Service Mashups

**EXECUTIVE COMMITTEE**

President: Rangachar Kasturi*

* voting member of the Board of Governors
† nonvoting member of the Board of Governors

### BOARD OF GOVERNORS

Term Expiring 2008: Richard H. Eckhouse; James D. Isak; James Moore, CSDP; Gary McGraw; Robert H. Sloan; Makoto Takizawa; Stephanie M. White

Term Expiring 2009: Van L. Eden; Robert Dupuis; Frank E. Ferrante; Roger U. Fujii; Ann Q. Gates, CSDP; Juan E. Gilbert; Don F. Shafer

Term Expiring 2010: André Ivanov; Phillip A. Laplante; Itaru Mimura; Jon G. Rokne; Christina M. Schober; Ann E.K. Sobel; Jeffrey M. Voas

Next Board Meeting: 18 Nov. 2008, New Brunswick, NJ, USA

### EXECUTIVE STAFF

Executive Director: Angela R. Burgess;
Director, Business & Product Development: Ann Vu; Director, Finance & Accounting: John Miller; Director, Governance, & Associate Executive Director: Anne Marie Kelly; Director, Membership Development: Violet S. Doan; Director, Products & Services: Evan Butterfield; Director, Sales & Marketing: Dick Price

### COMPUTER SOCIETY OFFICES

Washington Office. 1828 L St. N.W., Suite 1202, Washington, D.C. 20036-5104
Phone: +1 202 371 0101
Fax: +1 202 728 9614
Email: help@computer.org

Los Alamitos Office. 10662 Los Vaqueros Circle, Los Alamitos, CA 90720-1314
Phone: +1 714 821 8380
Fax: +1 714 821 4641
Email: help@computer.org

Asia/Pacific Office. Watanabe Building, 1-4-2 Minami-Aoyama, Minato-ku, Tokyo 107-0062, Japan
Phone: +81 3 3408 3118
Fax: +81 3 3408 3553
Email: tokyo.ofc@computer.org

### IEEE OFFICERS

President: Lewis M. Terman; President-Elect: John R. Vtg; Past President: Leah H. Jamieson; Executive Director & COO: Jeffrey W. Raynes; Secretary: Barry L. Shoo; Treasurer: David G. Green; VP, Educational Activities: Evangelia Michalis-Tzanakou; VP, Publication Services & Products: John Baillieul; VP, Membership & Geographic Activities: Joseph V. Lillie; VP, Standards Association Board of Governors: George W. Arnold; VP, Technical Activities: J. Roberto B. deMarca; IEEE Division V Director: Deborah M. Cooper; IEEE Division VIII Director: Thomas W. Williams; President, IEEE-USA: Russell J. Lefevre