Dr. Dobb's Journal

AUGUST 2011

Next

# Producer-Consumer Hand-Offs in C

Handing data from one thread to another concurrently

## ALSO INSIDE

Benevolent Language Dictators >>

Creating Your Own Domain-Specific Language >>

From the Vault: Naïve Bayesian Text Classification >>

UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

# CONTENTS

August 2011

## COVER STORY

## 7 Implementing Producer-Consumer Hand-Offs in C

**By Andrew Binstock**

It's important to know how to code the hand-off of data between concurrent threads correctly and efficiently.

## 12 Creating Your Own Domain-Specific Language

**By Sebastian Zarnekow**

Users of technical software often want to extend a package's capabilities by writing scripts. To do this, developers need to provide DSL capabilities. This article introduces the tools for easily writing and verifying a DSL grammar.

## 5 Benevolent Language Dictators

**By Andrew Binstock**

The programming languages that enjoy the most esteem by their users are all designed and managed by benevolent dictators, rather than committees.

## 18 From the Vault: Naïve Bayesian Text Classification

**By John Graham-Cumming**

John Graham-Cumming uses Perl to show that while spam filtering may be the best-known use of naïve Bayesian text classification, it's not the only application.

## 3 Letters

**By you**

This month, *Dr. Dobb's* readers recommend another C library, discuss defining "immutability," surrender in the quest for small classes, and propose a new use for virual machines

## 25 Links

Snapshots of the most interesting articles on drdobbs.com, including Using JDK 7's Fork/Join Framework and Sizing Android Visual Elements Correctly.

## 26 Editorial and Business Contacts


## More on DrDobbs.com

**C++AMP: Accelerated Massive Parallelism**

C++ AMP introduces an STL-like library for multidimensional data, allowing you to use your existing C++ STL knowledge for parallel programming.

http://drdobbs.com/cpp/231000963

**Sharding, Replication, Caches, and In-Memory Databases**

Caching, replication, and sharding have proven to be important tools in the modern database architect's toolbox.

http://drdobbs.com/blogs/high-performance-computing/231000846

**Interview with Scala's Martin Odersky**

Martin Odersky, the developer of Scala, was recently in the Bay Area to kick off his new company, Typesafe. He sat down with *Dr. Dobb's* Editor-in-Chief Andrew Binstock to discuss the language, its beauty and its warts, and where it is headed.

http://drdobbs.com/architecture-and-design/231001802

**Secure Login in AJAX Applications**

Implementing login correctly in Web apps is surprisingly complicated. Allen Holub demonstrates the two principal ways to keep users safe and logged in.

http://drdobbs.com/web-development/231000884

**Making the Move to Your Own DSL**

Your Own DSL is a drop-in replacement for Maven 2 and fully backwards compatible. So aside from addressing duplicate dependencies and plugin declarations, you don't have to make any changes to your Project Object Models (POM) file.

http://drdobbs.com/open-source/230800045

IN THIS ISSUE

Editorial >>
Hand-Offs in C>>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

# Mailbag

This month, *Dr. Dobb's* readers recommend another C library, discuss defining "immutability," surrender in the quest for small classes, and propose a new use for virual machines

## General-Purpose C Libraries

In addition to the C libraries you mention in your editorial (http://drdobbs.com/cpp/231000967), I'd recommend Newlib (http://sourceware.org/newlib/) . It is a C library intended for use on embedded systems. It is a conglomeration of several library parts, all under free software licenses that make them easily usable on embedded products. Newlib is only available in source form. It can be compiled for a wide array of processors, and will usually work on any architecture with the addition of a few low-level routines.

**—Bob Paddock**

## The Need For Immutability

Regarding your June editorial on immutability (http://drdobbs.com/architecture-and-design/231000092), it strikes me that there are really three distinct kinds of immutability you refer to. The first is a true constant that never changes. The second is a pass-by-value parameter where you know the value at a point in time, you want to provide it to a function (or the caller), and you don't want that entity to be able to change it. The third is a value that you only want one entity to be able to change, but multiple entities (e.g., threads) to be able to "see" even though it changes over time (and you want the readers to be able to see the latest value).

"Immutable variable" is a bit of an oxymoron and the latter two cases seem to fit this mold. Elaborating the three cases (maybe there are more) might help the reader.

Can a compiler really optimize pass-by-value? It would seem that for every reference to the value, the compiler would need to build an unchanging value on the stack to store a "constant" that is only really constant in a certain context. That same constant could have a different value in a different context (e.g., a second call to another module that needs the same variable's value). To make it truly invariate, you would need to incorporate memory protection of some kind, but perhaps

IN THIS ISSUE

Editorial >>
Hand-Offs in C>>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

that's beyond the scope of your concern in this editorial. (However, it's clearly not when it comes to security!)

**—Roger Skyhouse**
**General Motors**

**In Praise of Small Classes**

Having been coding professionally in R&D and maintenance for almost 30 years and managing coders for half of that [time], I have additional input to your editorial (http://drdobbs.com/230300002 ).

I have tried various techniques over the years, and while some are easier for the initial creation of the code base, I prefer to err on the side of code maintainability.

Classes that can be viewed entirely in one screen-full encourage subclass calls, sub-subclass calls, and so on. It's typically somewhat difficult for coders to think more than 2 or 3 levels deep in logic while dealing with the normal interruptions of the work day. On many occasions, I have tried to maintain a sub-sub-sub class of code and spent hours trying to figure out the impact the modification would have on other programs that call it. The original coder would have you think that their approach to nesting is supremely efficient. However, the same neural pathways of the intelligent creator of the code are seldom shared by the individuals who maintain the code.

In theory, I agree that small, compact code is a good idea. However, in practical implementation and with maintainability foremost on the mind, I prefer and teach "longer" code that can be quickly repaired and understood with less concentration as opposed to dealing with dozens of subclasses nested many levels deep. Obviously, there are exceptions to the rule and one cannot apply any one technique to all situations, but I thought it may be helpful to share my thoughts.

**— Tim Hauser**
**Hauser Tech**
**Richardson, TX**

**Use More VMs**

A good article (http://drdobbs.com/tools/229402296), I am glad you wrote it. I also agree with VMs for Web browsing. There is so much nonsense out there.

Something I have been thinking about is VMs as a distribution device. That is, when writing software, simply install it on a VM (usually something open source for licensing questions), and send it along to the customer. They install the image, twiddle some network settings, and voila! — an installed application.
Yes, third-party hook ups are a question...

**— Scott Augé**
**Amduus Information Works, Inc.**
**Flint, MI**

**Have a correction or a thoughtful opinion on *Dr. Dobb's* content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of *Dr. Dobb's*.**

IN THIS ISSUE

Editorial >>
Hand-Offs in C>>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

# In Praise of Benevolent Language Dictators

## What's wrong with codification by committee?

Writing new programming languages is a fun task, but dismally unsatisfying. The vast majority of languages that make it to a complete implementation are used primarily, if not exclusively, by their designers. If the designers are part of a company that hired them to write the language, then some cabal of employees or customers will compose the entire, quite-finite universe of users. Only the rare language breaks out into the Tiobe top 100 languages. And even among those victors, the greatest number will not ever enjoy usage above ¼ percent (as defined by Tiobe). Still, language creation is fun — a hotbed of interesting ideas — and is generally implemented by smart programmers who have a unique slant on the world.

The comparatively low rate of new language adoption is, in my view, not very different from the success rate of other software segments favored by many competing solutions: How many Web frameworks are there? Content management systems? Social networks?

The secret to language success is not easy to define and often relies more on luck than on quality of implementation. Luck being defined as the ability to tap into the zeitgeist and have the right product at the right time. Quality of implementation is typically not a decisive factor in achieving success. Many languages that are popular are no better than "good enough."

Quality of implementation, however, is critically important in the satisfaction enjoyed by users of the language. It is the experiential difference between writing Ruby and JavaScript. The factor that, in my view, most affects this quality of implementation is the vision of the original creator. Where the vision is maintained by a single individual, quality thrives. Where committees determine features, quality declines inexorably: Each new release saps vitality from the language even as it appears to remedy past faults or provide new, awaited capabilities.

It is hard to argue that C (and separately, C++) have become more vital since control of their design passed from the original creators to committees. Likewise, Java since it migrated from the original design group to the JCP process. Language extension by committee roll call tends to result in functional but not inspiring changes. So the language gets no new lease on life and simply lives inside its original skin getting progressively fatter, more complex, and dragged down by otiose features.

It's too easy to dismiss this process as the natural aging of all languages, but the success of the other model — the benevolent dictator — argues otherwise. Benevolent dictators make decisions about the language in consultation with the community of users. They are notable for being willing to extend the spirit of the language by making hard decisions that large committees almost always eschew. The best example of this is surely Guido van Rossum of Python fame. In 2008,

## IN THIS ISSUE

Editorial >>
Hand-Offs in C>>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

after the language had become well established, Python 3, which was not backwards compatible with previous versions, was released. Over time, many of the features of the new release were back-ported to the old 2.x trunk. This invigoration of the language deepened the high esteem felt by Pythonistas for their language.

A similar benevolent dictatorship exists with C#. The language, as I have mentioned before, is remarkably well tended by a core group in Microsoft headed by Anders Hejlsberg. (Although there is an ECMA standard for C#, Microsoft is the reference implementation, and the company decides the new features.) The result is a language widely loved by its users (Miguel de Icaza: "It's a beautiful languages that is such a pleasure to program in") and admired by others. Lua, Ruby, and Perl are other successful instances of this model.

Fundamentally, a programming language is the product of a viewpoint on a specific problem domain. To keep the viewpoint sharp, rather than diffuse, the original vision has to be maintained, even if the implementation evolves. This is almost impossible for committees to do, as they are made up of members with differing agendas and a reluctance towards bold action.

As popular languages mature, they might be tempted to go the route of codification by committee. Codification is fine, but implementation of new features must remain the privilege of a benevolent dictator if the languages are to enjoy the continued love and trust of their users.

*— Andrew Binstock is Editor in Chief for* Dr. Dobb's *and can be contacted at alb@drdobbs.com.*

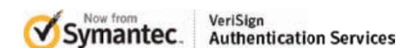
Comment

SHRINKWRAP YOUR APP

WITH AWARD-WINNING VERISIGN® CODE SIGNING

You developed the software. Now, deliver it with the same care and vigilance by using VeriSign® Code Signing. Why? Code signing not only protects the identity and reputation of the author, but it also verifies the authenticity and version of your software. Then, go a step further. VeriSign Code Signing can create a unique digital signature every time the code is signed. Plus, we support more certification programs and development platforms than any other Certificate Authority. Leverage the reputation of the most recognized and trusted name in online security.

Learn how VeriSign Code Signing can help make sure your applications are more trusted and adopted at www.VeriSign.com/CodeSigning or call 1-866-893-6565.




Copyright © 2011 Symantec Corporation. All rights reserved. Symantec, the Symantec Logo, and the Checkmark Logo are trademarks or registered trademarks of Symantec Corporation or its affiliates in the U.S. and other countries. VeriSign and other related marks are the trademarks or registered trademarks of VeriSign, Inc. or its affiliates or subsidiaries in the U.S. and other countries and licensed to Symantec Corporation. Other names may be trademarks of their respective owners.

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

# Implementing Producer-Consumer Hand-Offs in C

Handing data from one thread to another while the threads run concurrently requires care and a little finesse

By Andrew Binstock

**Producer-consumer situations,** in which one thread generates data for a second thread to use, occur frequently in multithreaded code. As a result, it's important to know how to code the hand-off of data between threads correctly and efficiently. This article examines an implementation in C using threading primitives from Microsoft Windows. The concepts, however, are universal and the primitives used here map to most other threading APIs.

When working with a producer-consumer situation a finite-sized queue is often the data structure of choice. It is convenient to think of the queue as circular, as in Figure 1.

Using this design, the producer thread adds chunks of data at the tail, while the consumer thread reads them from the head of the queue. Figure 2 shows a queue after the addition of a single word: "It."

Notice that the head is pointing at the first entry. The tail is pointing at the element in which we will place the next item. When the queue is full, but before we update tail, the queue appears as it does in Figure 3.

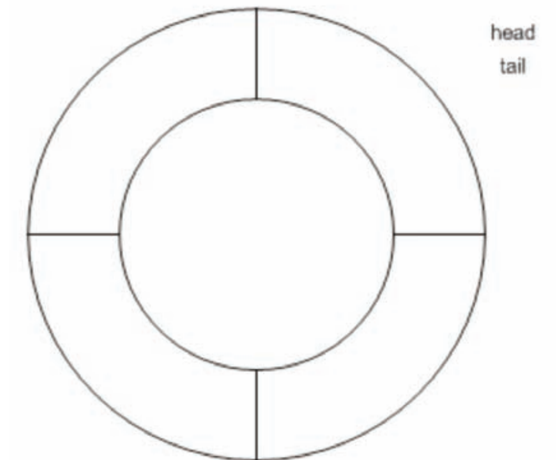



Figure 1: An empty fixed-size circular queue, waiting for the first item. Both head and tail point to the first element.

IN THIS ISSUE
Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>



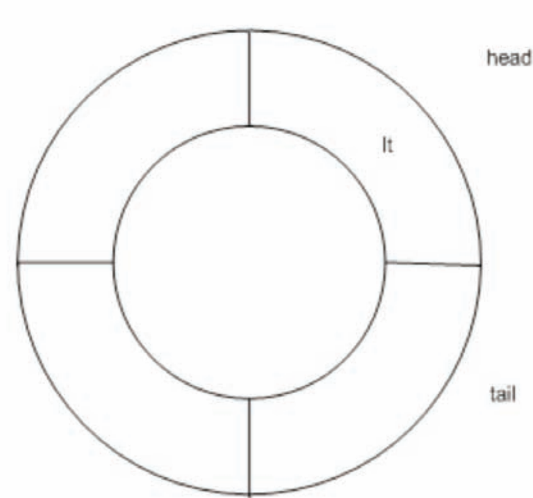



Figure 2: A word has been added as the first entry in the queue.

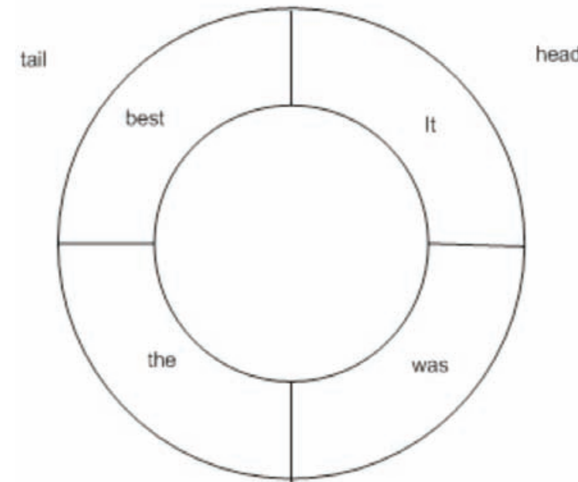



Figure 3: The full queue before updating the tail pointer.

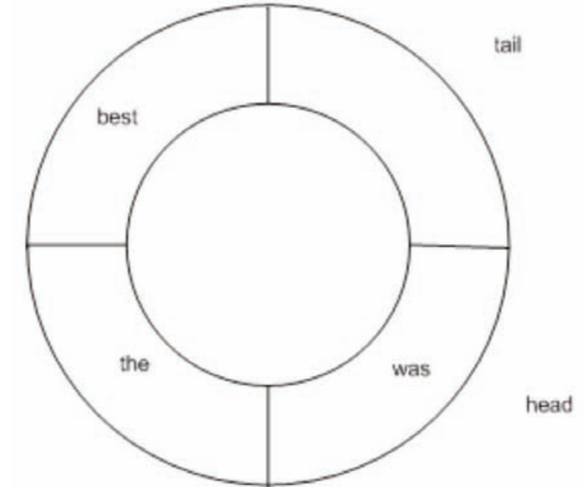



Figure 4: An item has been removed from the queue and room now exists for a new item to be added.

When we advance the tail to point to the next element where an item would go, we will have both head and tail pointing to the upper-right element, as in Figure 1. The difference is that unlike Figure 1, the queue is full, not empty. This situation tells us that we cannot use solely the relative positions of the head and tail to determine whether the queue is full or empty. We need a flag of some sort. In practice, a counter of elements is favored over a flag. This counter will, of course, disambiguate the situation, as well as make it simple for functions to check on waiting data.

Returning to the diagrams, the queue is now full and no more words can be added until at least one is removed from the head, as in Figure 4.

As can be seen from the sequence of figures, head and tail chase each other around the queue, stopping only when the queue is full.

### Implementing the Queue in Code

Queues are typically implemented as a structure containing a pointer to the head and tail, a counter of the number of items in the queue, and the set of elements that make up the queue itself. In our code, we will use a simple array of 10 elements.

```
#define Q_SIZE          10
#define Q_END       (Q_SIZE-1)
struct queue {       // the queue data structure
    int  head;
    int  tail;
    long count;
    struct item entry[Q_SIZE];
} q;
```

As explained, the counter helps us keep track of how many items are in the queue, which is necessary to disambiguate when the queue is full or empty.

IN THIS ISSUE
Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>



These elements will be used in a program in which the producer thread reads data from a file in blocks that are placed in a queue. The consumer thread reads those blocks and prints them to the screen. Each queue element consists of a text area and two integers: one containing the amount of data in the block, the other the block number.

```
#define BLK_SIZE        100
struct item {                   // the items in the queue:
    int    blk_num;             // the block sequence number
    size_t data_len;            // how much data in the block
    char   data [BLK_SIZE+1];   // the block data
   };
   bool atEOF;                  // so we know when to stop
```

Initializing the queue is trivial: The `head`, `tail`, and `counter` are all set to zero. `head` and `tail` are subscripts into the array of `items` rather than actual pointers, so setting them to zero points them both at the first element, as illustrated in Figure 1.

Adding items to the queue is done with the following code:

```
int addQueue ( struct queue *pq, char *text, size_t len )
{
static block_count = 0;
if ( pq->count == Q_SIZE )      // if queue is full, wait
                                // until space opens
{
    WaitForSingleObject ( hSpaceOpen, INFINITE );
    ResetEvent ( hSpaceOpen );  // and reset the event
}
memcpy ( pq->entry[pq->tail].data, text, len );
pq->entry[pq->tail].data_len = len;
pq->entry[pq->tail].blk_num  = ++block_count;
InterlockedExchangeAdd ( &(pq->count), 1L );
if ( pq->count == 1L )          // only true if previously was 0
    SetEvent ( hDataReady );    // so announce that there is
                                //  data ready
if ( pq->tail == Q_END )
     pq->tail = 0;
else
     pq->tail++;
return ( 0 );
}
```

The function takes three parameters: a pointer to the queue, the text to be added to the queue, and an integer specifying the length of the text. The function first checks to see whether the queue is full. If so, it must halt the current thread (which is both reading and adding to the queue). It does this by waiting on a Windows event, `hSpaceOpen`.

This event is signaled only when the consumer thread removes an item from a full queue, which tells the function the addition of an element may proceed. Because of this, if the queue is full, the thread will wait for the consumer thread. This is handled by the call to `WaitForSingleObject()`. Note that when Windows signals an event (that is, allows threads waiting on it to proceed), that event remains signaled until it is reset. So here, the full-queue event would remain signaled if we did not reset it immediately. However, we always want the thread to hold up when the queue is full. So, the first instruction once the event is signaled is to reset the event, so that the next time through, the thread will have to wait again if the queue is full.

VISIT GO PARALLEL
PROGRAMMING FOR PERFORMANCE
EXPERT TIPS AND TRICKS: C++, OPEN MP, THREADING AND MORE

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>



After determining that there is room in the queue, the code loads the data into the queue item indicated by the tail. It then increments the queue's item counter via Windows' interlocked functions. The interlocked functions are an efficient mechanism for updating counters on a mutually exclusive basis — no other thread can access the counter during the interlock operation. Note that in our implementation of the queue, in which one thread writes to the queue and the other removes (simply by reading) items, the only data item they might both need to modify at the same time is the counter. Hence the use of mutual exclusion on this one data item.

After incrementing the counter, the code checks to see whether the new item is the only one in the queue. If it is, the function knows that the consumer thread was waiting for it, because the consumer thread (whose code follows) always goes into a wait state when the queue is empty. Hence, the producer code checks for this condition, and if it's found, it sets the event that alerts the consumer code to retrieve the newly enqueued item.

Finally, the code increments `tail`, making the proper adjustment to wrap around to the beginning of the array/queue as needed.

The consumer thread is coded as follows:

```
int getQueue ( struct queue *pq, struct item *dest )
{
if ( pq->count <= 0L )    // if queue is empty, check EOF.
                          // If not EOF,
   if ( atEOF == TRUE )   // then wait for more data
       return ( 0 );
   else
   {
       WaitForSingleObject ( hDataReady, INFINITE );
       ResetEvent ( hDataReady );
   }
   memcpy ( dest->data, pq->entry[pq->head].data,
            pq->entry[pq->head].data_len );
   dest->blk_num = pq->entry[pq->head].blk_num;
   dest->data_len = pq->entry[pq->head].data_len;
```

Over 60%* of new projects will use multicore or multiprocessor architectures.
Is your software ready?

Multicore and multiprocessor software projects are 4.5x more expensive, have 25% longer schedules and require almost 3x as many software engineers. Eliminate the risk of critical software defects and boost development productivity with Klocwork's source code analysis solutions.

WANT TO LEARN MORE?
ACCESS OUR MULTICORE ANALYST & TECHNICAL RESOURCES OR REQUEST A FREE TRIAL.

klocwork.com/multicore

*VDC Research, an Executive White Paper on "Next Generation Embedded Hardware Architectures", September 2010.
Copyright ©2011 Klocwork. All Rights Reserved.

Klocwork
www.klocwork.com

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

```
    InterlockedExchangeAdd ( &(pq->count), -1L );
    if ( pq->count == ( Q_SIZE - 1 ))   // only true if
                                        // queue was full
        SetEvent ( hSpaceOpen );        // so announce
                                        // there's room
    if ( pq->head == Q_END )
        pq->head = 0;
    else
        pq->head++;
    return ( 0 );
}
```

This code bears a lot of similarity to the producer routine. Because it's trying to remove items from the queue, it first checks whether the queue is empty. If so, it must make an additional check: Is there any more data to come? It does this by checking the `atEOF` flag.

If it's not end of file (EOF), then the thread waits to be signaled by the `addQueue()` function, as was just explained. It immediately resets the event, copies the file chunk, decrements the item counter via interlocked functions, and then checks to see whether the queue was full prior to removing this item. If the queue was indeed full, it knows the producer thread is waiting or will eventually be waiting to add an item (since we're not at EOF), and so it signals the `hSpaceOpen` event, which informs the producer thread that a slot is available and to proceed. It then adjusts the head of the queue, making the adjustment for wrapping around to the beginning of the array, as needed.

Notice, that it signals the producer thread before it adjusts the head. It could just as well signal it after. But since the producer thread never accesses the head data item, the function does not need to make the producer thread wait. A central idea of parallel programming is to keep threads waiting on each other as little as possible. Hence, once the code knows the slot is available and it's safe to announce it, it does so before doing anything else.

Now, let's look at the consumer thread:

```
unsigned __stdcall ProcessData ( void* pv )
{
    int i;
    struct item *data_out;
    data_out = (struct item*) malloc ( sizeof (struct item));
    // wait to be told there's data waiting
    WaitForSingleObject ( hDataReady, INFINITE );
    ResetEvent ( hDataReady );
    while ( 1 )
    {
        if ( q.count == 0 && atEOF == TRUE )
            break;
        else
        {
            getQueue ( &q, data_out );
            // print the fetched data
            for ( i = 0; i < data_out->data_len; i++ )
                printf ( "%c", data_out->data[i] );
        }
    }
    return ( 0 );
}
```

This code is straightforward. It waits for a Windows event to signal that there is an item in the queue. It then enters an infinite loop that exits only when the queue is empty and the producer thread has reached end of file (EOF). The loop simply gets queue items and prints the data to the screen. Notice that all the negotiation between threads as to when data is ready is handled within the queue primitives. Hence, the logic of this consumer thread is minimal.

*This article was updated and adapted from "Programming with Hyper-Threading Technology" by Rich Gerber and Andrew Binstock, published by Intel Press. Portions are copyright Intel Corp.*

Comment

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

# Creating Your Own Domain-Specific Language

How to put together an industrial-grade DSL without all the agony of writing a whole new language

**By Sebastian Zarnekow**

**The Eclipse Xtext framework [1] provides a straightforward** way to craft external domain specific languages. Simply define the syntax of your language and you'll end up with an Eclipse-based language-aware IDE. In this tutorial, I will illustrate how to perform the first steps towards designing your own language. You'll become familiar with the syntax of the Xtext grammar definition language and learn how to define validation rules for your concepts.

For this tutorial, we will use a well known abstraction that has been addressed by different frameworks in one or the other way: the entity model. The entity model is handled differently by various frameworks to overcome the mismatch between relational databases and object-oriented software systems. Ruby On Rails [2] derives a lot of information from the database structure, Groovy's Grails [3] framework uses an internal DSL to describe the structure of the entities, and Spring Roo [4] has a strong emphasize on scaffolding.

Because it is such a common use case, Xtext 2.0 already ships with an example that demonstrates how such a domain model language looks. In this tutorial series, I want to develop that language from scratch , explaining the grammar and APIs step by step.

The first iteration of the language is pretty simple and contains a few intuitive concepts. It allows you to define packages that contain entities and data types. In turn, an entity defines properties. A property refers to a type and may be multi- or single-valued. An example file looks like this:

```
import com.drDobbs.common.String
package com.drDobbs {
        entity Author {
                name : String
                many articles: Article
        }

        entity Article {
                author: Author
                title: String
        }
}

package com.drDobbs.common {
        datatype String
}
```

## Getting Started

To follow this tutorial, you must ensure that Xtext is properly installed in your Eclipse IDE. Once this is done, launch the wizard that guides you through the necessary steps to create a new Xtext project:

```
File -> New -> Project... -> Xtext -> Xtext project
```

IN THIS ISSUE
Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

Then choose the following settings (Figure 1) on the next wizard page before you hit Finish:

**Main project name:** *org.example.domainmodel*
**Language name:** *org.example.domainmodel.Domainmodel*
**DSL-File extension:** *dmodel*

The "Hello World" grammar in the Domainmodel.xtext file illustrates a trivially simple language that allows you to write a number of "Hello World!" sentences.

```
grammar org.example.domainmodel.Domainmodel
       with org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Model:
       greetings+=Greeting*;

Greeting:
       'Hello' name=ID '!';
```

In addition to the preamble, the grammar file contains two parser rules that specify the structure of a valid sentence of the language. It's important to realize that an Xtext grammar is not only a description of the syntax of a DSL, it's also a concise notation that describes the mapping to a type-safe object model. The types and the expected structure of these objects is called "abstract syntax." It is either derived from the grammar definition itself, or defined externally and imported into the grammar definition. These imports allow for language modularization. During this tutorial lesson, we will stick with the defaults and the generated `domainmodel`, which is declared in the preamble:

```
generate domainmodel "http://www.example.org/domainmodel/Domainmodel"
```

## Entity Grammar Step-By-Step

Let's have a closer look at the grammar notation and then develop the actual syntax for the entity language step by step. The first parser rule in a grammar is the entry point for the language. Each time the parser reads a file, it starts with the entry rule and walks through the grammar to consume the input file.

By providing meaningful names for each rule, you can ensure that the resulting object graph has an intuitive structure. The root object should be a `Domainmodel`. It provides access to a number of elements which are either `packages`, `types`, or `imports`. These three concepts will be derived from `AbstractElement` to make sure that we can use them in arbitrary order in our grammar.

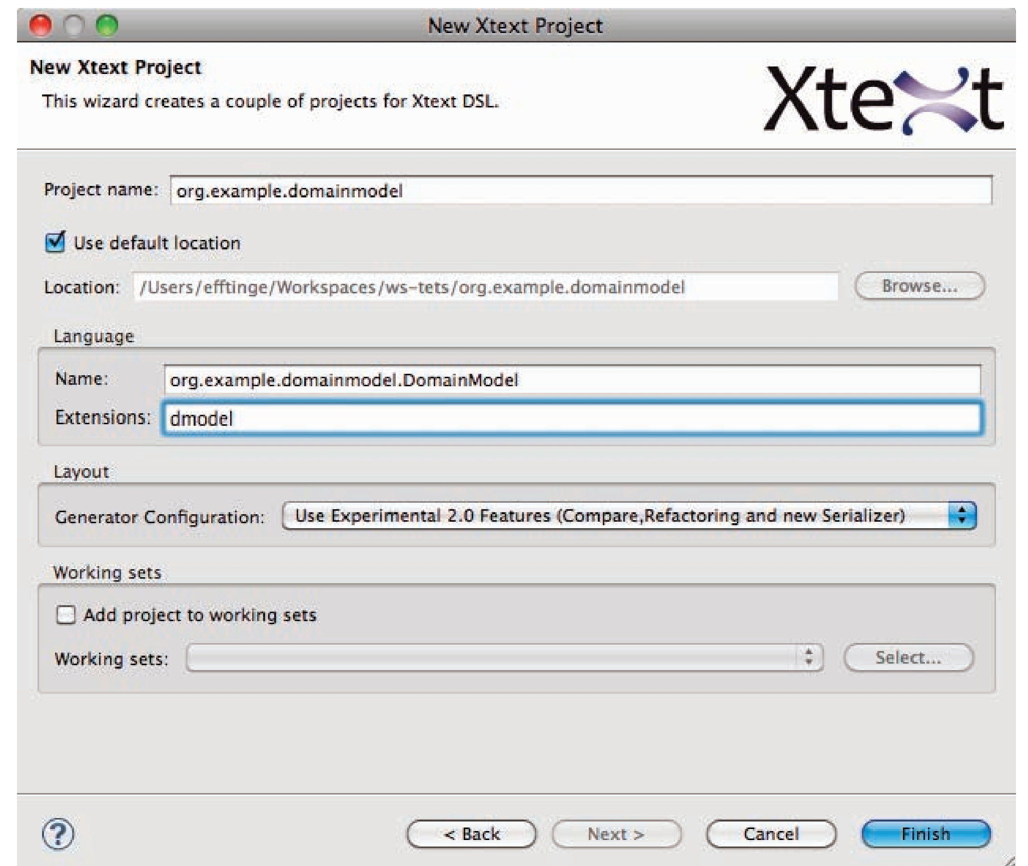


Figure 1: Setting up a DSL project in Xtext.

IN THIS ISSUE
Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>



```
Domainmodel:
  (elements += AbstractElement)*
;

AbstractElement:
  PackageDeclaration | Type | Import
;
```

The `+=` assignment is used here to denote that an instance of `Domainmodel` holds a list of `AbstractElements`. There are two other assignment operators available. The plain equals sign (`=`) sets a single value instead of adding to a list, while the boolean assignment (`?=`) sets the feature on the left-hand side to `true` if the right-hand side was parsed successfully.

The second interesting syntax element is the cardinality (`*`). It denotes that the syntax allows any number of `AbstractElements` in the body of a `Domainmodel`. Xtext has operators indicating this, as shown in Table 1.

The third thing that I want to emphasize is the alternative in the body of the rule `AbstractElement`. The pipe symbol (|) is used here to define different valid paths in a rule (e.g., an `AbstractElement` is either a `PackageDeclaration`, a `Type`, or an `Import`). The overall syntax definition of an Xtext grammar is very close to the Extended Backus-Naur Form (EBNF) [5] or the input format used by many parser generators. It's concise, intuitive, and easy to understand after little exposure.

To continue with the grammar definition, we add a rule to describe the structure of a package. A `PackageDeclaration` is conceptually similar to the root `Domainmodel` except that it has a name. Because

| Operator | Meaning |
|---|---|
| no operator, default | Exactly one occurrence |
| ? | Optional: zero or one occurrence |
| * | Optional multi value: Occurrence arbitrary times |
| + | Multi value: Expected at least ones in a valid sentence |

Table 1: Operators for specifying cardinality.

the name of a package consists of a number of segments that are delimited by a dot (.), we add a parser rule for it:

```
PackageDeclaration:
  'package' name = QualifiedName '{'
    (elements += AbstractElement)*
  '}'
;

QualifiedName:
  ID ('.' ID)*
;
```

The rule `QualifiedName` is special. The language processor recognizes that it does not contain any assignments but simply consumes atomic values. That's why it is interpreted as a data type rule. Data type rules allow you to define a complex syntax for a simple type (e.g., for a string or a decimal number). They have significant advantages compared to atomic terminal rules like ID or INT: After an input sequence is parsed, the framework automatically strips superfluous whitespace along with any nested comments from the qualified name and assigns the clean value to the package's name.

The next step is to define types. The sample file illustrates that two different kinds of types exist in the domain model language. In addition to entities, there are simple data types that don't expose any structural information. The latter have a concise notation: After the keyword `datatype`, a name is expected, which is an atomic ID.

```
Type:
  DataType | Entity
;

DataType:
  'datatype' name=ID
;

Entity:
  'entity' name = ID ('extends' superType = [Entity | QualifiedName])?
  '{'
    (features += Feature)*
  '}'
;
```

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

The rule for entities leverages a several concepts introduced in this tutorial. There are simple assignments, keywords, optional cardinality, and multi-value assignments. However, there is something new, too. An entity may optionally extend another entity to represent an inheritance relationship. A cross reference is used for this. All the other assignments that were used so far in this grammar, established a has-a or contains-a relationship. The "super" type of an entity is different. The referenced super type is not contained in the entity. Instead, it must be defined somewhere else, and the inheriting entity simply references it. A cross reference is the grammar element that allows you to describe exactly this concept. Between the squared brackets, the type of the referenced object is used. In this context, the pipe symbol is not used to denote an alternative but rather serves as a delimiter. On its right side is the definition of how the reference appears in the concrete syntax of the language. Because entities may be referenced by their qualified name, we reuse the previously defined data type rule here.

However, qualified names are not the best choice for readability and editing comfort. That's why the notion of imports is added to the language. Xtext provides convenient default behavior for imports that are based on namespaces and qualified names. It is a good match for many common use cases. If a feature `importedNamespace` is used in your language, the framework will process the information that is stored in that feature and enrich the scope accordingly when the cross references are resolved. Due to this convention, two new rules are sufficient to implement imports. A container object `Import` is necessary as is a data type rule defining the imported namespace itself to allow wildcards. It adds an optional suffix to a qualified name.

```
Import:
  'import' importedNamespace = QualifiedNameWithWildcard
;

QualifiedNameWithWildcard:
  QualifiedName '.*'?
;
```

There is one last missing piece of the grammar definition: The parser rule for the features of an entity must still be defined:

```
Feature:
  (many ?= 'many')? name = ID ':' type = [Type | QualifiedName]
;
```

A feature definition is a sequence of an optional keyword `many`, an ID that becomes the feature's name, a colon, and a type reference. The type is again a cross-reference that refers to a type. Because cross-references are polymorphic, the type of a feature can be either a simple data type or an entity.

### Start the Entity Editor

That's all that is necessary to create an entity language, including support for data types and nested packages. The complete grammar now looks like this:

```
grammar org.example.domainmodel.Domainmodel
  with org.eclipse.xtext.common.Terminals

generate domainmodel "http://www.example.org/domainmodel/Domainmodel"

Domainmodel:
  (elements += AbstractElement)*
;

PackageDeclaration:
  'package' name = QualifiedName '{'
    (elements += AbstractElement)*
  '}'
;
```

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

```
AbstractElement:
  PackageDeclaration | Type | Import
;

QualifiedName:
  ID ('.' ID)*
;

Import:
  'import' importedNamespace = QualifiedNameWithWildcard
;

QualifiedNameWithWildcard:
  QualifiedName '.*'?
;

Type:
  DataType | Entity
;

DataType:
  'datatype' name=ID
;

Entity:
  'entity' name = ID ('extends' superType = [Entity | QualifiedName])?
  '{'
    (features += Feature)*
  '}'
;

Feature:
  (many ?= 'many')? name = ID ':' type = [Type | QualifiedName]
;
```

### Now What?

The next step is to fire up the generator, which will process the grammar definition and derive some infrastructure and configuration information from it.

Locate the `GenerateDomainmodel.mwe2` next to the grammar file and choose:

```
Run As -> MWE2 Workflow
```

from the context menu. After the generator was executed successfully, you can test drive the editor in a new Eclipse application. Therefore, select :

```
Run As -> Eclipse Application
```

on the project org.example.domainmodel to launch a new Eclipse process. You are already in the home stretch: Create a new Project using a sample file with the right extension, e.g., Sample.dmodel; and then you can try the editor (Figure 2).

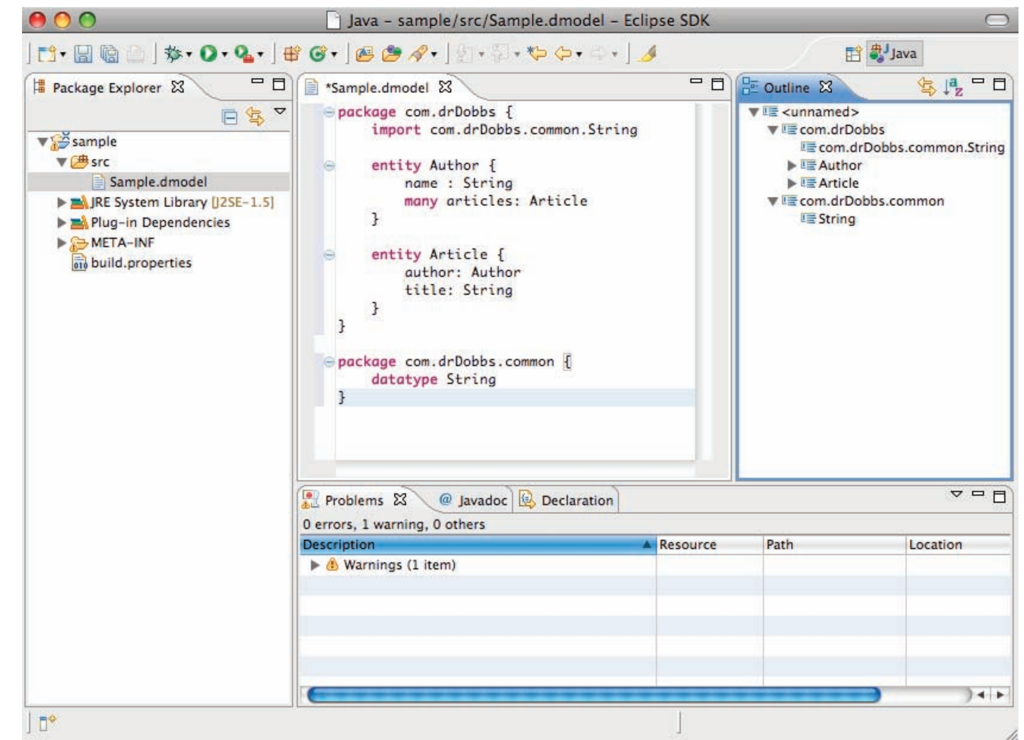


Figure 2: The grammar in the Xtext editor.

IN THIS ISSUE
Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>



**Custom Checks**

One main advantages of external DSLs (that is, DSLs that are a separate part of a package) is the ability to specify the semantics and the validation rules. Because you have full control over the language, you can define constraints that are specific to your project and environment. To illustrate how this can be done with Xtext, let's add a check to ensure that multi-value features only refer to entities.

The generator that processed the grammar definition already created stubs for commonly used customization points among them the hook for static analysis. The class `DomainmodelJavaValidator` should be enhanced with constraints specific for the language. It provides a convenient API to write checks. Each method written in that class that is annotated with `@Check` is invoked by the framework in the validation phase. The methods specify the type of the validated model element by means of their first argument. When you identify a problem, `error(..)`, `warning(..)` or `info(..)` can be used to report it as shown in this code.

```
package org.example.domainmodel.validation;

import org.eclipse.xtext.validation.Check;
import org.example.domainmodel.domainmodel.DomainmodelPackage;
import org.example.domainmodel.domainmodel.Entity;
import org.example.domainmodel.domainmodel.Feature;


public class DomainmodelJavaValidator
    extends AbstractDomainmodelJavaValidator {
  @Check
  public void checkFeature(Feature f) {
    if (f.isMany() && !(f.getType() instanceof Entity)) {
      error("Multi value features have to point to entities",
          DomainmodelPackage.Literals.FEATURE__MANY);
    }
  }
}
```

The validation hook is certainly not the only customization point provided by Xtext. Fine-grained customization with other customization techniques is demonstrated by the documentation on the Xtext website.[5]

In a future article on www.drdobbs.com, I'll show how to take this language definition and put it to use.

**References**

[1] Xtext: http://www.eclipse.org/Xtext
[2] Ruby On Rails: http://rubyonrails.org/
[3] Grails: http://www.grails.org/
[4] Spring Roo: http://www.springsource.org/roo
[5] EBNF: http://en.wikipedia.org/wiki/Extended_Backus-Naur_Form
[6] Xtext Documentation: http://www.eclipse.org/Xtext/documentation/

*— Sebastian Zarnekow is core committer to the Xtext project.*

Comment

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

## From the Vault

# Naïve Bayesian Text Classification

This entry from *Dr. Dobb's* May 2005 issue employed Perl to illustrate that while spam filtering may be the best known use of naïve Bayesian text classification, it's not the only application.

*—DDJ*

**By John Graham-Cumming**

**Paul Graham popularized the term "Bayesian Classification"** (or more accurately "Naïve Bayesian Classification") after his "*A Plan for Spam*" article was published (http://www.paulgraham.com/spam.html). In fact, text classifiers based on naïve Bayesian and other techniques have been around for many years. Companies such as Autonomy and Interwoven incorporate machine-learning techniques to automatically classify documents of all kinds; one such machine-learning technique is naïve Bayesian text classification.

Naïve Bayesian text classifiers are fast, accurate, simple, and easy to implement. In this article, I present a complete naïve Bayesian text classifier written in 100 lines of commented, nonobfuscated Perl.

A text classifier is an automated means of determining some metadata about a document. Text classifiers are used for such diverse needs as spam filtering, suggesting categories for indexing a document created in a content management system, or automatically sorting help desk requests.

The classifier I present here determines which of a set of possible categories a document is most likely to fall into and can be used in any of the ways mentioned with appropriate training. Feed it samples of spam and nonspam e-mail and it learns the difference; feed it documents on various medical fields and it distinguishes an article on, say, "heart disease" from one on "influenza." Show it samples of different types of help desk requests and it should be able to sort them so that when 50 e-mails come in informing you that the laser printer is down, you'll quickly know that they are all the same.

### The Math

You don't need to know any of the underlying mathematics to use the sample classifier presented here, but it helps.

The underlying theorem for naïve Bayesian text classification is the Bayes Rule:

$$P(A|B) = ( P(B|A) * P(A) ) / P(B)$$

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

The probability of *A* happening given *B* is determined from the probability of *B* given *A*, the probability of *A* occurring, and the probability of *B*. The Bayes Rule enables the calculation of the likelihood of event *A* given that *B* has happened. This is used in text classification to determine the probability that a document *B* is of type *A* just by looking at the frequencies of words in the document. You can think of the Bayes Rule as showing how to update the probability of event *A* happening given that you've observed *B*. A far more extensive discussion of the Bayes Rule and its general implications can be found in Wikipedia at http://en.wikipedia.org/wiki/Bayes%27_Theorem. For the purposes of text classification, the Bayes Rule is used to determine the category a document falls into by determining the most probable category. That is, given this document with these words in it, which category does it fall into?

A category is represented by a collection of words and their frequencies; the frequency is the number of times that each word has been seen in the documents used to train the classifier.

Suppose there are *n* categories $C_0$ to $C_{n-1}$. Determining which category a document *D* is most associated with means calculating the probability that document *D* is in category $C_i$, written $P(C_i|D)$, for each category $C_i$.

Using the Bayes Rule, you can calculate $P(C_i|D)$ by computing:

$$P(C_i|D) = ( P(D|C_i) * P(C_i) ) / P(D)$$

$P(C_i|D)$ is the probability that document *D* is in category $C_i$; that is, the probability that given the set of words in *D*, they appear in category $C_i$. $P(D|C_i)$ is the probability that for a given category $C_i$, the words in *D* appear in that category.

$P(C_i)$ is the probability of a given category; that is, the probability of a document being in category $C_i$ without considering its contents. $P(D)$ is the probability of that specific document occurring.

To calculate which category *D* should go in, you need to calculate $P(C_i|D)$ for each of the categories and find the largest probability. Because each of those calculations involves the unknown but fixed value $P(D)$, you just ignore it and calculate:

$$P(C_i|D) = P(D|C_i) * P(C_i)$$

$P(D)$ can also be safely ignored because you are interested in the relative — not absolute — values of $P(C_i|D)$, and $P(D)$ simply acts as a scaling factor on $P(C_i|D)$.

*D* is split into the set of words in the document, called $W_0$ through $W_{m-1}$. To calculate $P(D|C_i)$, calculate the product of the probabilities for each word; that is, the likelihood that each word appears in $C_i$. Here's the "naïve" step: Assume that words appear independently from other words (which is clearly not true for most languages) and $P(D|C_i)$ is the simple product of the probabilities for each word:

$$P(D|C_i) = P(W_0|C_i) * P(W_1|C_i) * ... * P(W_{m-1}|C_i)$$

For any category, $P(W_j|C_i)$ is calculated as the number of times $W_j$ appears in $C_i$ divided by the total number of words in $C_i$. $P(C_i)$ is calculated as the total number of words in $C_i$ divided by the total number of words in all the categories put together. Hence, $P(C_i|D)$ is:

$$P(W_0|C_i) * P(W_1|C_i) * ... * P(W_{m-1}|C_i) * P(C_i)$$

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

for each category, and picking the largest determines the category for document *D*.

A common criticism of naïve Bayesian text classifiers is that they make the naïve assumption that words are independent of each other and are, therefore, less accurate than a more complex model. There are many more complex text classification techniques, such as Support Vector Machines, *k*-nearest neighbor, and so on. In practice, naïve Bayesian classifiers often perform well, and the current state of spam filtering indicates that they work very well for e-mail classification.

A useful toolkit that implements different algorithms is the freely available Bow toolkit from Carnegie Mellon University (CMU) (http://www-2.cs.cmu.edu/~mccallum/bow/). It makes a useful testbed for comparing the accuracy of different techniques. A good starting point for reading more about naïve Bayesian text classification is the Wikipedia article on the subject (http://en.wikipedia.org/wiki/Naïve_Bayesian_classification).

## Implementation

The Perl implementation (Listing One) uses the hash (associative array) `%words` to store the word counts for each word and for each category.

**Listing One**

```
use strict;
use DB_File;

# Hash with two levels of keys: $words{category}{word} gives count
of
# 'word' in 'category'.  Tied to a DB_File to keep it persistent.

my %words;
tie %words, 'DB_File', 'words.db';

# Read a file and return a hash of the word counts in that file
```

```
sub parse_file
{
    my ( $file ) = @_;
    my %word_counts;

    # Grab all the words with between 3 and 44 letters

    open FILE, "<$file";
    while ( my $line = <FILE> ) {
        while ( $line =~ s/([[:alpha:]]{3,44})[ \t\n\r]// ) {
            $word_counts{lc($1)}++;
        }
    }
    close FILE;
    return %word_counts;
}

# Add words from a hash to the word counts for a category
sub add_words
{
    my ( $category, %words_in_file ) = @_;

    foreach my $word (keys %words_in_file) {
        $words{"$category-$word"} += $words_in_file{$word};
    }
}

# Get the classification of a file from word counts
sub classify
{
    my ( %words_in_file ) = @_;

    # Calculate the total number of words in each category and
    # the total number of words overall

    my %count;
    my $total = 0;
    foreach my $entry (keys %words) {
        $entry =~ /^(.+)-(.+)$/;
        $count{$1} += $words{$entry};
        $total += $words{$entry};
    }

# Run through words and calculate the probability for each category

    my %score;
    foreach my $word (keys %words_in_file) {
        foreach my $category (keys %count) {
            if ( defined( $words{"$category-$word"} ) ) {
```

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

```
                    $score{$category} += log( $words{"$category-$word"} /
                                              $count{$category} );
                } else {
                    $score{$category} += log( 0.01 /
                                              $count{$category} );
                }
            }
        }
    # Add in the probability that the text is of a specific category

    foreach my $category (keys %count) {
        $score{$category} += log( $count{$category} / $total );
    }
    foreach my $category (sort { $score{$b} <=> $score{$a} } keys %count) {
        print "$category $score{$category}\n";
    }
}

# Supported commands are 'add' to add words to a category and
# 'classify' to get the classification of a file

if ( ( $ARGV[0] eq 'add' ) && ( $#ARGV == 2 ) ) {
    add_words( $ARGV[1], parse_file( $ARGV[2] ) );
} elsif ( ( $ARGV[0] eq 'classify' ) && ( $#ARGV == 1 ) ) {
    classify( parse_file( $ARGV[1] ) );
} else {
    print <<EOUSAGE;
Usage: add <category> <file> - Adds words from <file> to category <category>
       classify <file>       - Outputs classification of <file>
EOUSAGE
}

untie %words;
```

The hash is stored to disk using a Perl construct called a "tie" that, when used with the `DB_File` module, results in the hash being stored automatically in a file called "words.db" so that its contents persist between invocations.

```
use DB_File;
my %words;
tie %words, 'DB_File', 'words.db';
```

The hash keys are strings of the form category-word: For example, if the word "potato" appears in the category "veggies" with a count of three, there will be a hash entry with key "potato-veggies" and value "3." This data structure contains enough information to compute the probability of a document and do a naïve Bayesian classification.

The subroutine `parse_file` reads the document to be classified or trained on and fills in a hash called `%words_in_file` that maps words to the count of the number of times that word appeared in the document. It uses a simple regular expression to extract every 3- to 44-letter word that is followed by whitespace; in a real classifier, this word splitting could be made more complex by accounting for punctuation, digits, and hyphenated words.

```
sub parse_file
{
    my ( $file ) = @_;
    my %word_counts;
    open FILE, "<$file";
    while ( my $line = <FILE> ) {
        while ( $line =~
            s/([[:alpha:]]{3,44})[ \t\n\r]// ){
            $word_counts{lc($1)}++;
        }
    }
    close FILE;
    return %word_counts;
}
```

The output of `parse_file` can be used in two ways: It can be used to train the classifier by learning the word counts for a particular category and updating the `%words` hash, or it can be used to determine the classification of a particular document.

To train the classifier, call the `add_words` subroutine with the output of `parse_file` and a category. In the Perl code, a category is any string and the classifier is trained by passing sample documents into `parse_file` and then into `add_words`: `add_words( <category>, parse_file( <sample document>));`

IN THIS ISSUE
Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>



```
sub add_words
{
   my ( $category, %words_in_file ) = @_;
   foreach my $word (keys %words_in_file) {
      $words{"$category-$word"} +=
         $words_in_file{$word};
   }
}
```

Once document training has been done, the `classify` subroutine can be called with the output of `parse_file` on a document. `classify` will print out the possible categories for the document in order of most likely to least likely:

```
classify ( parse_file( <document to classify> ) );

sub classify
   my ( %words_in_file ) = @_;
   my %count;
   my $total = 0;
   foreach my $entry (keys %words) {
      $entry =~ /^(.+)-(.+)$/;
      $count{$1} += $words{$entry};
      $total += $words{$entry};
   }
   my %score;
   foreach my $word (keys %words_in_file) {
      foreach my $category (keys %count) {
         if (defined($words{"$category-$word"})) {
            $score{$category} +=
               log( $words{"$category-$word"} /
                  $count{$category} );
         } else {
            $score{$category} +=
               log( 0.1 /
                  $count{$category} );
         }
      }
   }
   foreach my $category (keys %count) {
      $score{$category} +=
         log( $count{$category} / $total );
   }
   foreach my $category (sort { $score{$b} <=> $score
{$a} } keys %count) {
      print "$category $score{$category}\n";
   }
}
```

`classify` first calculates the total word count (`$total`) for all categories (which it needs to calculate $P(C_i)$) and the word count for each category (`%count` indexed by category name, which it needs to calculate $P(W_j|C_i)$). Then `classify` calculates the score for each category: The score is the value of $P(C_i|D)$. It's preferable to call it a score for two reasons: Ignoring $P(D)$ means that, strictly speaking, the value is being calculated incorrectly, and `classify` uses logs to reduce overflow errors and replace multiplication by addition for speed. The score is in fact log $P(C_i|D)$, which is:

$$log\ P(W_0|C_i) + log\ P(W_1|C_i) + ... + log\ P(W_{m-1}|C_i) + log\ P(C_i)$$

(Recall the equality $log\ (A*B)=log\ A+log\ B$). In that log form, it is still suitable for comparison. After the score has been calculated, `classify` calculates $log\ P(C_i)$ for each category and then sorts the scores in descending order to output the classifier's opinion of the document. `classify` makes an estimate of the probability for a word that doesn't appear in a particular category by calculating a very small, nonzero probability for that word based on the word count for the category:

```
$score{$category} += log( 0.1 / $count{$category} );
```

A small amount of Perl code wraps these three subroutines into a usable classifier that accepts commands to add a document to the word list for a category (and hence, train the classifier), and to classify a document.

```
if ( ( $ARGV[0] eq 'add' ) && ( $#ARGV == 2 ) ) {
   add_words( $ARGV[1],
                  parse_file( $ARGV[2] ) );
} elsif ( ( $ARGV[0] eq 'classify' ) && ( $#ARGV == 1 )
) {
```

IN THIS ISSUE
Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>



```
   classify( parse_file( $ARGV[1] ) );
} else {
   print <<EOUSAGE;
Usage: add <category> <file> - Adds words from <file>
to category <category>
   classify <file> - Outputs classification
of <file>
EOUSAGE
}
untie %words;
```

If the Perl code is stored in file bayes.pl, then the classifier is trained like this:

```
perl bayes.pl add veggies article-about-vegetables
perl bayes.pl add fruits article-about-fruits
perl bayes.pl add nuts article-about-nuts
```

to create three categories (veggies, fruits, and nuts). Asking bayes.pl to classify a document will output the likelihood that the document is about vegetables, fruits, or nuts:

```
% perl bayes.pl classify article-I-just-wrote
fruits -4.11700258611469
nuts -6.60190923590268
veggies -11.9002266024507
```

Here, bayes.pl shows that the new article is most likely about fruits.

## E-Mail Classification

If you are interested in classifying e-mail, there are a couple of tweaks that improve accuracy in practice: Don't fold case on values from headers and count words differently if they appear in the subject or body.

In the aforementioned Perl implementation, there is no difference between the words *From*, *FROM*, and *fRoM*: They are all considered to be instances of `from`. The `parse_file` subroutine lowercases the word before counting it. In practical e-mail classifiers, the names of e-mail headers turn out to be a better indicator of the type of an e-mail if case is preserved. For example, the header *MIME-Version* was written *MiME-Version* by one piece of common spamming software.

**"A text classifier [can] accurately identify worms and viruses, such as W32.Bagle, and it was able to spot even mutated versions of the worms"**

Distinguishing words found in the subject versus the body also increases the accuracy of a naïve Bayesian text classifier on e-mail. The simplest way to do this is to store a word like *forward* as `subject:forward` when it comes from the subject line, and simply `forward` when it is seen in the body.

## Performance

The Perl code presented here isn't optimized at all. Each time `classify` is called, it has to recalculate the total word count for each category and it would be easy to cache the log values between invocations. The use of a Perl hash will not scale well in terms of memory usage.

However, the algorithm is simple and can be implemented in any language. A highly optimized version of this code is used in the POPFile e-mail classifier to do automatic classification. It uses a combination of Perl and SQL queries. The Bow toolkit from CMU has a fast C implementation of naïve Bayesian classification.

IN THIS ISSUE
Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>



## Uses of Text Classification

Although spam filtering is the best-known use of naïve Bayesian text classification, there are a number of other interesting uses on the horizon. IBM researcher Martin Overton has published a paper concerning the use of naïve Bayesian e-mail classification to detect e-mail-borne malware (http://arachnid.homeip.net/papers/VB2004-Canning-more-than-SPAM-1.02.pdf). In Overton's paper, presented at the Virus Bulletin 2004 conference, he demonstrated that a text classifier could accurately identify worms and viruses, such as W32.Bagle, and that it was even able to spot even mutated versions of the worms. All this was done without giving the classifier any special knowledge of viruses.

The POPFile Project is a general e-mail classifier that can classify incoming e-mail into any number of categories. Users of POPFile have reported using its naïve Bayesian engine to classify mail into up to 50 different categories with good accuracy; and one journalist uses it to sort "interesting" from "uninteresting" press releases.

At LISA 2004, four Norwegian researchers presented a paper concerning a system called DIGIMIMIR, which was capable of automatically classifying requests coming into a typical IT help desk and in some cases responding automatically (http://www.digimimir.org/). They use a document clustering approach that, while not naïve Bayesian, is similar in implementation complexity and allowed the clustering together of "similar" e-mails without knowing the initial set of possible topics.

*— At the time this article was originally published, John was chief scientist at Electric Cloud. He is also the creator of POPFile. John can be contacted at http://blog.jgc.org/.*

Comment

IN THIS ISSUE
Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>



# This Month on DrDobbs.com

Interesting items posted on www.drdobbs.com over the past month that you may have missed

## USING JDK 7'S FORK/JOIN FRAMEWORK

Fork/Join is an enhancement to the *ExecutorService* implementation that allows you to more easily break up processing to be executed concurrently, and recursively, with little effort on your part.

**http://drdobbs.com/blogs/java/231000556**

## PROJECT OF THE MONTH: XTEXT DSL FRAMEWORK

Develop extensive domain-specific languages — including the parser, interpbayesian classificationr, code generator, and IDE — without all the toil and hard work.

**http://drdobbs.com/open-source/230700063**

## CODE CORRECTNESS AND UNIT TESTING IN .NET APPLICATIONS

Implementation details are verified with unit tests and Code Contracts verify the correct flow of data — that's why they go happily hand in hand.

**http://drdobbs.com/windows/231000213**

## TRUST, BUT VERIFY

Pretty much all engineering is based on abstractions.

**http://drdobbs.com/blogs/embedded-systems/231000389**

## HOW NOT TO COMPARE AGGREGATES

One way of avoiding trouble when you try to define an order relation for sorting is to start by asking what "unordered" means as a concept, in the context of the data structure that you are trying to sort.

**http://drdobbs.com/blogs/cpp/231000237**

## SIZING ANDROID VISUAL ELEMENTS CORRECTLY

With iOS, from the moment you start designing an app, you know how Apple wants you to build it. The design plans are exhaustive and accessible. Unfortunately, there is no comparable resource for Android that gives consideration to asset creation or creative direction. This article tries to fill that gap.

**http://drdobbs.com/mobility/230600075**

## FREE SMTP

Back in the day, if you wanted to send an email, you simply found a handy SMTP server that was accepting incoming connections. There were thousands, and they were undiscriminating, so this was no big deal...The invention of spam ruined that.

**http://drdobbs.com/blogs/web-development/231000912**

Previous Next

## IN THIS ISSUE


Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>


# Dr. Dobb's


**Andrew Binstock** Editor in Chief, Dr. Dobb's
alb@drdobbs.com

**Deirdre Blake** Managing Editor, Dr. Dobb's
dblake@techweb.com

**Amy Stephens** Copyeditor, Dr. Dobb's
astephens@techweb.com

**Sean Coady** Webmaster, Dr. Dobb's
scoady@techweb.com

**Jon Erickson** Editor in Chief Emeritus , Dr. Dobb's

**CONTRIBUTING EDITORS**
**Mike Riley**
**Herb Sutter**
**Scott Ambler**

**DR DOBB'S**
**UBM TECHWEB**
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

**INFORMATIONWEEK ADVISORY BOARD**

**Dave Bent**
Senior VP and CIO
United Stationers

**Robert Carter**
Executive VP and CIO
FedEx

**Michael Cuddy**
VP and CIO
Toromont Industries

**Laurie Douglas**
Senior CIO
Publix Super Markets

**Dan Drawbaugh**
CIO
University of Pittsburgh
Medical Center

**Jerry Johnson**
CIO
Pacific Northwest National
Laboratory

**Kent Kushar**
VP and CIO
E.&J. Gallo Winery

**Carolyn Lawson**
Director, E-Services
California Office of the CIO

**Jason Augustnard**
Managing Director
Wells Fargo Securities

**Randall Mott**
Sr. Executive VP and CIO
Hewlett-Packard

**Denis O'Leary**
Former Executive VP
Chase.com

**Mykolas Rambus**
CEO
Wealth-X

**M.R. Rangaswami**
Founder
Sand Hill Group

**Manjit Singh**
CIO
Las Vegas Sands

**David Smoley**
CIO
Flextronics

**Ralph J. Szygenda**
Former Group VP and CIO
General Motors

**Peter Whatnell**
CIO
Sunoco

**INFORMATIONWEEK**

**Rob Preston** VP and Editor In Chief, InformationWeek
rpreston@techweb.com 516-562-5692

**John Foley** Editor, InformationWeek
jpfoley@techweb.com 516-562-7189

**Chris Murphy** Editor, InformationWeek
cjmurphy@techweb.com 414-906-5331

**Art Wittmann** VP and Director, Analytics, InformationWeek
awittmann@techweb.com 408-416-3227

**Alexander Wolfe** Editor In Chief, InformationWeek.com
awolfe@techweb.com 516-562-7821

**Stacey Peterson** Executive Editor, Quality, InformationWeek speterson@techweb.com 516-562-5933

**Lorna Garey** Executive Editor, Analytics, InformationWeek
lgarey@techweb.com 978-694-1681

**Stephanie Stahl** Executive Editor, InformationWeek
sstahl@techweb.com 703-266-6030

**Fritz Nelson** VP and Editorial Director
fnelson@techweb.com 949-223-3608

**David Berlind** Chief Content Officer, TechWeb
dberlind@techweb.com 978-462-5315

**REPORTERS**

**Charles Babcock** Editor At Large
Open source, infrastructure, virtualization
cbabcock@techweb.com 415-947-6133

**Thomas Claburn** Editor At Large
Security, search, Web applications
tclaburn@techweb.com 415-947-6820

**Paul McDougall** Editor At Large
Software, IT services, outsourcing
pmcdougall@techweb.com

**Marianne Kolbasuk McGee** Senior Writer IT management and careers
mmcgee@techweb.com 508-697-0083

**J. Nicholas Hoover** Senior Editor
Desktop software, Enterprise 2.0, collaboration
nhoover@techweb.com 516-562-5032

**Andrew Conry-Murray** New Products and Business Editor
Information and content management
acmurray@techweb.com 724-266-1310

**W. David Gardner** News Writer
Networking, telecom
wdavidg@earthlink.net

**Antone Gonsalves** News Writer
Processors, PCs, servers
antoneg@pacbell.net

**Eric Zeman**
Mobile and Wireless
eric@zemanmedia.com

**CONTRIBUTORS**

**Michael Biddick** mbiddick@nwc.com
**Michael A. Davis** mdavis@nwc.com
**Jonathan Feldman** jfeldman@nwc.com
**Randy George** rgeorge@nwc.com
**Michael Healey** mhealey@nwc.com

**EDITORS**

**Jim Donahue** Chief Copy Editor
jdonahue@techweb.com

**ART/DESIGN**

**Mary Ellen Forte** Senior Art Director
mforte@techweb.com

**Sek Leung** Senior Designer
sleung@techweb.com

**INFORMATIONWEEK ANALYTICS**
analytics.informationweek.com

**Art Wittmann** VP and Director
awittmann@techweb.com 408-416-3227

**Lorna Garey** Executive Editor, Analytics
lgarey@techweb.com 978-694-1681

**Heather Vallis** Managing Editor, Research
hvallis@techweb.com 508-416-1101

**INFORMATIONWEEK.COM**

**Benjamin Tomkins** Managing Editor
btomkins@techweb.com 516-562-5336

**Roma Nowak** Senior Director,
Online Operations and Production
rnowak@techweb.com 516-562-5274

**Tom LaSusa** Managing Editor, Newsletters
tlasusa@techweb.com

**Jeanette Hafke** Web Production Manager
jhafke@techweb.com

**Joy Culbertson** Web Producer
jculbertson@techweb.com

**Nevin Berger** Senior Director, User Experience
nberger@techweb.com

**Steve Gilliard** Senior Director, Web Development
sgilliard@techweb.com



Copyright 2011 United Business Media LLC. All rights reserved.

**UBM TECHWEB**

**Tony L. Uphoff** CEO

**John Dennehy** CFO

**David Michael** CIO

**Bob Evans** Sr. VP and Global CIO Director

**Joseph Braue** Sr. VP, Light Reading Communications Network

**Scott Vaughan** CMO

**Ed Grossman** Executive Vice President, InformationWeek Business Technology Network

**John Ecke** VP and Group Publisher, Financial Technology Network, InformationWeek Government, and InformationWeek Healthcare

**Martha Schwartz** EVP, Group Sales, InformationWeek Business Technology Network

**Beth Rivera** Senior VP, Human Resources

**David Berlind**
Chief Content Officer, TechWeb, and Editor in Chief, TechWeb.com

**Fritz Nelson** VP and Editorial Director, InformationWeek Business Technology Network, and Executive Producer, TechWeb TV

**UNITED BUSINESS MEDIA LLC**

**Pat Nohilly** Sr. VP, Strategic Development and Business Administration

**Marie Myers** Sr. VP, Manufacturing

**INFORMATIONWEEK VIDEO**
informationweek.com/tv

**Fritz Nelson** Executive Producer
fnelson@techweb.com

**INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK**

**DarkReading.com**
Security
**Tim Wilson**, Site Editor
wilson@darkreading.com

**IntelligentEnterprise.com**
App Architecture
**Doug Henschen**,
Editor in Chief
dhenschen@techweb.com

**NetworkComputing.com**
Networking , Communications, and Storage
**Mike Fratto**, Site Editor
mfratto@techweb.com

**PlugIntoTheCloud.com**
Cloud Computing
**John Foley**, Site Editor
jpfoley@techweb.com

**InformationWeek SMB**
Technology for Small and Midsize Business
**Benjamin Tomkins**,
Site Editor
btomkins@techweb.com

**Dr. Dobb's**
The World of Software Development
**Andrew Binstock**
Executive Editor
alb@drdobbs.com

IN THIS ISSUE

Editorial >>
Hand-Offs in C >>
Your Own DSL >>
Bayesian Classification >>
Letters >>
Links >>
Table of Contents >>

# Dr. Dobb's Business Contacts

## DR. DOBB'S

**Sales Director, Michele Hurabiell**
(415) 378-3540, mhurabiell@techweb.com

**Account Executive, Shaina Guttman**
(212) 600-3106, sguttman@techweb.com

## INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

**CMO, Scott Vaughan**
(949) 223-3662, svaughan@techweb.com

**VP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz**
(212) 600-3015, mschwartz@techweb.com

**Sales Assistant, Group Sales, Kelly Glass**
(212) 600-3327, kglass@techweb.com

**Publisher's Assistant, Esther Rodriguez**
(949) 223-3656, erodriguez@techweb.com

## SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

**Inside Sales Manager, Vesna Beso**
(415) 947-6104, vbeso@techweb.com

**Sales Assistant, Ian Doyle**
(415) 947-6105, idoyle@techweb.com

### Strategic Accounts

**Account Director, Sandra Kupiec**
(415) 947-6922, skupiec@techweb.com

**Account Manager, Shoshana Freisinger**
(415) 947-6349, sfreisinger@techweb.com

**Sales Assistant, Matthew Cohen-Meyer**
(415) 947-6214, mmeyer@techweb.com

## SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

**District Manager, Jenny Hanna**
(516) 562-5116, jhanna@techweb.com

**District Manager, Michael Greenhut**
(516) 562-5044, mgreenhut@techweb.com

**Account Manager, Cori Gordon**
(516) 562-5181, cgordon@techweb.com

**Inside Sales Manager East, Ray Capitelli**
(212) 600-3045, rcapitelli@techweb.com

**Sales Assistant, Elyse Cowen**
(212) 600-3051, ecowen@techweb.com

### Strategic Accounts

**District Manager, Mary Hyland**
(516) 562-5120, mhyland@techweb.com

**Account Manager, Tara Bradeen**
(212) 600-3387, tbradeen@techweb.com

**Account Manager, Jennifer Gambino**
(516) 562-5651, jgambino@techweb.com

**Sales Assistant, Kathleen Jurina**
(212) 600-3170, kjurina@techweb.com

### Dr. Dobb's

**Sales Director, Michele Hurabiell**
(415) 378-3540, mhurabiell@techweb.com

**Account Executive, Shaina Guttman**
(212) 600 3106, sguttman@techweb.com

## MARKETING

**VP, Marketing, Winnie Ng-Schuchman**
(631) 406-6507, wng@techweb.com

**Marketing Manager, Monique Luttrell**
(949) 223-3609, mluttrell@techweb.com

## AUDIENCE DEVELOPMENT

**Director, Karen McAleer**
(516) 562-7833, kmcaleer@techweb.com

## BUSINESS OFFICE

**General Manager, Marian Dujmovits**

**United Business Media LLC**
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000
**Copyright 2011. All rights reserved.**

**Entire contents Copyright© 2011, Techweb/United Business Media LLC, except where otherwise noted. No portion of this publication august be reproduced, stored, transmitted in any form, including computer retrieval, without written permission from the publisher. All Rights Reserved. Articles express the opinion of the author and are not necessarily the opinion of the publisher. Published by Techweb, United Business Media, 303 Second Street, Suite 900 South Tower, San Francisco, CA 94107 USA 415-947-6000.**

## UBM TECHWEB

**Tony L. Uphoff** CEO

**John Dennehy** CFO

**David Michael** CIO

**Bob Evans** Sr. VP and Global CIO Director

**Joseph Braue** Sr. VP, Light Reading Communications Network

**Scott Vaughan** CMO

**Ed Grossman** Executive Vice President, InformationWeek Business Technology Network

**John Ecke** VP and Group Publisher, Financial Technology Network, InformationWeek Government, InformationWeek Healthcare

**Martha Schwartz** VP, Group Sales, InformationWeek Business Technology Network

**Beth Rivera** Senior VP, Human Resources

**David Berlind** Chief Content Officer, TechWeb, and Editor in Chief, TechWeb.com

**Fritz Nelson** VP, Editorial Director, InformationWeek Business Technology Network, and Executive Producer, TechWeb TV

## UNITED BUSINESS MEDIA LLC

**Pat Nohilly** Sr. VP, Strategic Development and Business Admin.

**Marie Myers** Sr. VP, Manufacturing