



Dr. Dobb's DIGEST

The Art and Business of Software Development March, 2010

Editor's Note by Jonathan Erickson	2
Techno-News Truly Random Numbers Researchers use a "flip-flop" to improve random number generators.	3
Features Team Building Goes Viral by Jerry Tubbs Ready for an Agile development environment? Success depends on making sure everyone buys in.	4
When Teams Break Down, Business Loses by Johanna Rothman What happens when team members don't work well together?	6
Your Own MP3 Duration Calculator: Part 2 by Gigi Sayfan Rolling out Version 2.0.	7
C Snippet #3 by Bob Stout Sorting an array of strings.	14
Java and C++ Socket Communication by Eric J. Bruno Integrating C++ native Windows applications with Java applications via network sockets.	15
The Jazz Process by Adrian Cho What can software developers learn from jazz musicians?	19
Estimating on Agile Projects by Scott W. Ambler Upfront estimating from an agile viewpoint.	21
Columns Book Review by Mike Riley Mike Riley examines the second edition of Erica Sadun's <i>The iPhone Developer's Cookbook</i> .	24
Conversations by Jonathan Erickson Q&A with Andrew Stellman and Jennifer Greene on teamwork: It's all about trust and techniques.	25
Other Voices by Su-Laine Yeo The Sim Ship Standard.	27
Swaine's Flames by Michael Swaine Michael takes a spin at Chatroulette.	29

Hub-a, Hub-a, Hub-a



By Jonathan Erickson,
Editor In Chief

A couple of weeks ago, the headlines generally read “Microsoft Unveils Windows Phone 7 Series.” More accurately, they probably should have said something like “Microsoft takes yet another run at mobile phones.” Of course, that doesn’t mean that the Phone 7 isn’t interesting and newsworthy. What I find interesting about the Phone 7 is the implementation of “hubs,” although I probably would have shied away from describing them as “unrivaled.” But more on that shortly.

According to Microsoft, “Hubs bring together related content from the Web, applications, and services into a single view to simplify common tasks.” The company goes on to say that the Phone 7 Series includes six hubs built on specific themes reflecting activities that, according to focus groups I suppose, matter most to people:

- **People.** This hub delivers a social experience by bringing together relevant content based on the person, including his or her live feeds from social networks and photos. It also provides a central place from which to post updates to Facebook and Windows Live in one step.
- **Pictures.** This hub is for sharing pictures and video to a social network.
- **Games.** This hub delivers the first and only official Xbox LIVE experience on a phone, including Xbox LIVE games, Spotlight feed, and the ability to see a gamer’s avatar, achievements, and profile.
- **Music and Video.** This hub creates a media experience that includes content from a user’s PC, online music services, and a built-in FM radio.
- **Marketplace.** This hub lets users discover and load the phone with “certified” applications and games.
- **Office.** This hub enables access to Office, OneNote, SharePoint Workspace, and Outlook Mobile.

Again these are the six out-of-the-box Microsoft hubs, with the suggestion that more can come in the future, both from Microsoft and third-party developers. This will likely require special tools, of course, and an SDK will be made available this month.

Microsoft isn’t the inventor of the hub concept, and in fact, hubs aren’t even new to the world of Windows developers. Infragistics (<http://www.infragistics.com>) for instance, provides a Hub and Spoke pattern as part of its Quince pattern library (<http://quince.infragistics.com/#/Main>).

All of which prompted me to get in touch with Tobias Komischke, Director of User Experience at Infragistics, who told me: “I like the idea of the hub concept, because it helps to consolidate and organize various content into just a few, manageable access points — and that’s just what is needed for a phone in times of convergence, where you have most information and data available anywhere and anytime.

“The hubs thus help greatly to provide a welcoming first impression when starting the phone experience: non-cluttered, clearly segmented information, and access points that still allow for customization.

“It appears to me that the hubs are the reloaded portals that we saw on the Web 10 years ago. Back then, there was a website — a portal — that allowed access to an entire world of related information. But we had to go there — we had to dare entering the portal to access this information. Now, the hub concept not only provides 6 and not only one “portal”, but the hub tiles live and show updated information (e.g. new emails), so the transition between seeing a hub and entering it feels more natural.”

According to Infragistics, the hub and spoke pattern is most commonly seen on phones and other small devices where having global navigation (and even the ability to interact with it were it displayed) is difficult, or in situations where there is an open plug-in model for applications, such as Facebook, where the hub is more of a facilitator for disparate services and applications to come together in one place. The iPhone and now the Phone 7 fit both of these constraints. Other classes of software solutions that warrant this approach include games — people expect to and usually want to be immersed in them. Sensitive tasks that require a person’s full attention could be another.

[Return to Table of Contents](#)

Truly Random Numbers

German researchers use a “flip-flop” to improve random number generators

A new approach to generating truly random numbers could lead to improved Internet security and better weather forecasts, according to researchers writing in the *International Journal of Critical Computer-Based Systems*.

Random numbers are a critical part of computer and Internet security. They allow websites and browsers to encrypt the data sent between them using a session key. Weather forecasters, climate scientists, economists, and epidemiologists also use random numbers to generate simulated data for their predictive models. Such simulations can test theories of hurricane formation, climate change, and the spread of disease epidemics, for instance.

According to Bernhard Fechner of the University of Hagen, and Andre Osterloh of BTC AG, in Germany, the “quality” of a random number is a measure of how truly random the number is. This quality affects significantly any security or simulation in which it is used. If a so-called random number is not truly random, then someone could predict a security key and crack the Internet encryption on bank accounts, e-commerce sites or secure government websites, for instance. Similarly, if the random numbers used in scientific models of the weather, climate, or the spread of disease and economic boom and bust are predictable, then systematic errors will creep into the models and make the predictions unreliable.

Fechner and Osterloh explain in their paper, “A Meta-level True Random Number Generator,” that a good random number in computer binary would usually comprise discrete and uniformly distributed ones and zeroes. In such a sequence there is no way to predict what digit will appear next, the number is thus random. In reality, it is unfeasible to generate a sequence that is totally unpredictable by computational means. Therefore, physical means are used to generate true random numbers.

The German team has now developed a true random number generator that uses an extra layer of randomness by making a computer memory element, a flip-flop, twitch randomly between its two states 1 or 0. Immediately prior to the switch, the flip-flop is in a “metastable state” where its behaviour cannot be predicted. At the end of the metastable state, the contents of the memory are purely random.

The researchers' experiments with an array of flip-flop units show that for small arrays the extra layer makes the random number almost twenty times more “random” than conventional methods. The degree of randomness possible depends on the size of the array and so a bigger array could be even more effective. Essentially, this means that the bigger the array the better the quality of random numbers. As such, it is many times more difficult to predict with any degree of certainty the next number in a sequence compared to current random number generators. This could be used to improve everything from climate change models to stock market predictions.

The team adds that the efforts of a cracker attempting to influence the array will be wholly obvious to a simple statistical analysis as — depending on the type of attack — either the whole array or single elements will be disturbed, whereas these are again selected randomly. So this true random number generator can protect systems against third-party snooping, potentially making private and sensitive transactions on the Internet more secure.

[Return to Table of Contents](#)

Dr.Dobb's

EDITOR-IN-CHIEF
Jonathan Erickson

EDITORIAL
MANAGING EDITOR
Deirdre Blake
COPY EDITOR
Amy Stephens
CONTRIBUTING EDITORS
Mike Riley, Herb Sutter
WEBMASTER
Sean Coady

VICE PRESIDENT, GROUP PUBLISHER
Brandon Friesen
VICE PRESIDENT GROUP SALES
Martha Schwartz

AUDIENCE DEVELOPMENT
CIRCULATION DIRECTOR
Karen McAleer
MANAGER
John Slesinski

DR. DOBB'S
600 Harrison Street, 6th Floor, San
Francisco, CA, 94107. 415-947-6000.
www.ddj.com

UBM LLC

Pat Nohilly Senior Vice President,
Strategic Development and Business
Administration
Marie Myers Senior Vice President,
Manufacturing

TechWeb

Tony L. Uphoff Chief Executive Officer
John Dennehy, CFO
David Michael, CIO
John Siefert, Senior Vice President and
Publisher, InformationWeek Business
Technology Network
Bob Evans Senior Vice President and
Content Director, InformationWeek
Global CIO
Joseph Braue Senior Vice President,
Light Reading Communications
Network
Scott Vaughan Vice President,
Marketing Services
John Ecke Vice President, Financial
Technology Network
Beth Rivera Vice President, Human
Resources
Fritz Nelson Executive Producer,
TechWeb TV

techweb™

Team Building Goes Viral

Ready for an Agile development environment? Success depends on making sure everyone buys in

by Jerry Tubbs

DST Systems' large development group of about 1,200 people over the years adopted a hodgepodge of tools, processes, and source code control systems. It didn't have a single central repository for source code nor one developer toolset shared across the organization. Some teams used Serena PVCS, others didn't. Some tools worked in Eclipse, others were standalone. Moving between tools was cumbersome. Many processes were manual and time consuming, especially if they involved Excel spreadsheets or Word documents.

Development teams complained regularly about the tools they had to work with. Managers had difficulty answering basic questions about who was working on what and determining the status of a particular asset, because they had to look in so many places to get answers.

This situation became a serious problem as we looked to accelerate the development cycle for DST Automated Work Distributor, business process management and workflow software that's been around for nearly 20 years. Financial services companies use AWD to process financial transactions. Healthcare organizations use it to process insurance claims. It's a mature, sophisticated product with lots of features. But moving it through a traditional waterfall methodology — from design to coding to testing to integration — was taking two years for each major release. To remain competitive, we needed to speed things up.

So after decades of using waterfall methodologies, we switched last year to the Scrum Agile software development methodology. In a short time, we achieved some impressive results. Our developers accelerated software releases by a factor of four, reducing the software release cycle from 24 months to 6 months. We also boosted developer productivity by at least 20%.

But getting there wasn't easy. Running Scrum sprints with our existing tools caused immediate problems. Processes broke down, and our disparate tools and manual processes kept Scrum teams from reaching their objectives. We knew that Agile was a team-based approach to software development. But to make it work, we quickly realized that we needed an easy-to-use, comprehensive development environment that would put tools and data within easy reach of everyone on the team.

The Right Approach

We put together a product evaluation team to identify the right development environment in mid-2008. It considered about a dozen application life-cycle management (ALM) products, including tools from CollabNet, IBM, MKS, and Serena, as well as several open source options. We settled on three finalists and ran pilot projects with each.

Cost-effectiveness and ease of adoption were key selection criteria. We also examined feature-effectiveness, assessing whether the core features were packaged in a way that made them immediately accessible. We wanted to be able to easily make use of each feature without extensive training that would impinge on productivity. Whichever product we chose, we needed to adopt it quickly so we could continue developing and supporting products in a competitive market. We were, in effect, changing the tires while the car was moving.

From Chaos To ALM

For the three pilot projects, we assembled a typical Scrum team that included a project owner, a Scrum master, a business analyst, and someone from quality control. We then replicated source

code into the tool's repository, established some Agile use cases, created processes within the tool to model the use cases, and staged a dress rehearsal to test the day-to-day interaction with the tool. We allowed time following each pilot to assess how the tools would let us meet the requirements we had set.

Once the pilot phase was over, the evaluation team chose CollabNet's TeamForge as our ALM platform, with Subversion as a single, centralized repository for all development projects. We completed the transition to TeamForge in just less than 10 weeks. By June 2009, our core development team was up and running in the new environment.

Now all members of the development team — from product owners to Scrum masters to developers and testers to business analysts — have what they need for all phases of the application life cycle. We're also using Agile templates included with the TeamForge to manage our processes and workflows.

The switch has made development teams more effective. Now, developers aren't struggling to move files among disparate tools, but instead, they do all their work within the ALM platform and Eclipse. Developers aren't complaining about tools anymore.

Our goal was to increase productivity by 20% in one year. We've met and probably exceeded that goal. The combination of Agile development and TeamForge has delivered benefits for our development work, as well as the overall business, including:

- **Competitiveness and market readiness.** We're delivering new features every six months instead of every two years, and we're ready to move aggressively in our markets this year.
- **Transparency.** We can now answer management's questions about people and products immediately. We have visibility into project status and can drill down to the level of individual assets and users. My team no longer spends hours looking for answers and building reports. All of the information we need is readily available in the new development environment.
- **Happier developers.** We're not hearing complaints about tools and processes. Everything the developers need is in TeamForge and Eclipse. The tool's ease-of-use has led to viral adoption by our development teams.
- **More effective support.** The new ALM platform provides support engineers with better visibility into our product, including new features under development and relevant test results.

Lessons Learned

If you're software group is struggling with old, inadequate toolsets, it's time to evaluate ALM options. Based on our experience, here's what to look for:

First, keep it simple. Find a product that meets your needs, but be wary of complex "Cadillac" offerings that promise every feature under the sun. Over the years, I've learned that Cadillac-style products generally give you higher costs along with all those features. Focus on what you really need, and don't fall for showy vendor presentations. Simpler is better, and it's almost always cheaper.

7 Key Factors Effective Development Teams Start Here

- 1. Common Purpose. Get everyone on the same page.**
- 2. Commitment. Do what's necessary to get the job done.**
- 3. Trust. Establish trust because it's mandatory even when you don't always agree.**
- 4. Understand The Process. Master the tools and processes before coding begins.**
- 5. Communication. Share knowledge and information constantly.**
- 6. Resources. Have adequate resources at the outset so team can focus on the project, not the tools.**
- 7. Leadership. Ensure leaders are in place to make technical or business decisions.**

Involve key players every step of the way. When evaluating ALM options, put together a team of senior developers who'll ask vendors hard questions and give you excellent advice on selection criteria. Their endorsement of the final selection will carry tremendous weight with the rest of the organization. Your developers will have faith in the decision, because the senior developer team was involved. They're going to trust other developers before they trust a bunch of managers who went off by themselves and then came down from the mountain with product brochures and a purchase order.

Finally, strive for viral adoption, rather than mandating change. Find a platform that's so compelling that people will adopt it without coercion. If you've selected a product that solves problems, meets people's needs, and has earned the endorsement of senior developers, you won't have to force it on your developers. They'll adopt it on their own.

After several months, we already have more teams on the new tools and processes than used our previous ones. It's part of our culture now. People don't question it. There's a groundswell of adoption going on. It's that compelling.

While it would be tempting to give Agile practices all the credit for these results, if you've really selected a product that solves problems, meets people's needs, and has earned the endorsement of senior developers, you won't have to force it on your developers. They'll adopt it on their own. Mandating change is hard. Viral adoption is easier — and more fun.

— *Jerry Tubbs is systems development manager for DST Systems.*

[Return to Table of Contents](#)

When Teams Break Down, Business Loses

What happens when team members don't work well together?

by Johanna Rothman

What happens when team members don't work well together? A decade ago, when I consulted at AVS, a company that makes data visualization software, there were multiple ways to access its database. Think about it — a database-oriented product with multiple ways to get and put.

Several years before, two senior managers couldn't agree on an approach to access the database, so they formed two competing teams and implemented their respective ideas. By the time I arrived, the multiple library calls were entrenched in the product, and customers weren't happy because the calls to the database were different. Likewise, the developers were unhappy because it was difficult to incorporate new features into the existing, disparate calls. So they simply kept creating new calls. When my engagement ended, sales were down, staff was reduced, and the company was trying to recreate itself. It now has new management and a new look for its legacy products.

The competing teams contributed to AVS's stumble. Customers depend on companies to be consistent with their products, and AVS's product was inconsistent. I've also done some consulting at a large insurance company that has a huge Agile program aimed at unifying all products under one umbrella to let agents sell anything, anytime, anywhere. The project teams at first refused to synchronize their iterations. Some teams wanted two-week iterations, some wanted three-week ones. After several iterations, it was clear the program wouldn't meet the first deliverable, so the program manager mandated that each team move to two-week iterations and synchronize their work. They finally released the first chunk — a month late.

A senior VP estimated the company lost up to \$100,000 per business day that the system was late. That's a cost of \$2 million for a one-month slip, and a high price to pay for teams that couldn't agree on iteration length. Sometimes it's difficult to separate technical teamwork from bad architectural and requirements decisions. But those decisions are part of teamwork, too. When we don't pay attention to teams and how they work, we risk alienating customers, losing business, and being unresponsive to the rest of the organization.

— Johanna Rothman is a management consultant and author of *Behind Closed Doors: Secrets Of Great Management* (www.jrothman.com/Books/behind-closed-doors.html).

[Return to Table of Contents](#)

Your Own MP3 Duration Calculator: Part 2

Rolling out Version 2.0

by Gigi Sayfan

In “Your Own MP3 Duration Calculator: Part 1” (<http://www.drdobbs.com/embedded/222500141>), I presented a program called “MP3 Duration Calculator” (MDC), which I wrote for my wife Liat who recently completed certification as a spinning instructor. Spinning instructors need to create CDs with music for each class that matches the class content. MDC allows Liat to quickly select different song and see the total duration of all the songs.

Liat was excited and started browsing her music collection and selecting songs for her first spinning class. She was marginally pleased with the interface but complained that the total duration text box is a little obscure. Then something terrible happened! Liat had already selected a bunch of songs when she accidentally clicked without holding control on another song. Her entire selection was gone and replaced with the single last song. That was pretty late (about 2:00 a.m.) and her class was scheduled for 8:00 a.m. Liat was upset and reselected all the songs again. Finally, all the songs were there and it was time to burn the CD. That's when we discovered that we had run out of blank CDs. Alas, Liat brought several commercial CDs to her first spinning class and swapped them manually.

So, it wasn't the best software debut ever. The good thing is that there was a lot of room for improvement. Liat ran into a few other issues. When selecting songs, she sometimes didn't recognize the song's name or just wanted to listen to it to make sure it was an appropriate selection. She had to go to Windows Explorer, find the song, and click on it to bring up the Windows Media Player. This process was tedious and broke the flow of song selection. After selecting all the songs, she had to manually select them again in Windows Explorer to burn them. Finally, Liat also wanted to print a list of all the songs and their duration. Obviously, it was time for Version 2.0.

New Requirements

Before launching into full-blown development, we sat down and compiled the following list of requirements for MDC 2.0:

- Make total duration more prominent
- Check boxes for selection (one-song toggle)
- Show # of selected songs
- Play songs from within MDC
- Copy selected songs to a new folder
- Export song list to a CSV file
- Remember last folder

One of the main features of MDC 1.0 was its interactivity. Collecting the track information can take a while for folders with tens or hundreds of songs. MDC was never blocked. You could always do all the selection operations you wanted on the available subset of songs and MDC displayed the total duration. I wanted to preserve this capability for MDC 2.0 so the user could simultaneously select and play songs while collecting more tracks.

MP3 Duration Calculator 2.0

Version 2.0 is usually where you discover if your system is well designed. If your design had too many implicit assumptions about what it should support, then you will have a hard time accommodating the changes and new requirements for version 2.0. In general, there are two types of requirements that are difficult: changes to existing functionality, and the addition of new functionality. In a well-designed application, adding new functionality should be straightforward and have little impact on the existing code. The complete source code and related files for MDC Version 2.0 are available at http://i.cmpnet.com/ddj/images/article/2010/code/MP3DurationCalculator_v2.zip.

In this case, most of the requirements are for new functionality. The changes to existing functionality are making the total duration more prominent and changing the way songs are selected.

Make Total Duration More Prominent

In Version 1.0, the total duration text box was at bottom of the window squished between the select/unselect buttons and the status text box; see Figure 1.

In Version 2.0, it moved to the top-right corner of the window and its text became blue and bold; see Figure 2.

WPF was designed to support this kind of change, where you move elements around and modify properties without any code changes (unless you think of XAML as code, which is a perfectly valid point of view). I only had to modify the XAML a bit. Here is the code for the bottom strip in Version 1.0:

```
<DockPanel
    DockPanel.Dock="Bottom"
    LastChildFill="True"
    Height="30"
    Margin="0" VerticalAlignment="Stretch"
>
    <Button DockPanel.Dock="Left" Margin="3" Click="SelectAll_Click">
```



Figure 1

```
    Select All
  </Button>
  <Button DockPanel.Dock="Left" Margin="3"
    Click="UnselectAll_Click">
    Unselect All
  </Button>
  <TextBox DockPanel.Dock="Left" Height="22"
    Name="total" Width="60" Margin="3"
    Text="{Binding Path=Self, Converter={StaticResource
    DurationConverter}}">
  </TextBox>
  <TextBox Height="22" Name="status" MinWidth="100" Margin="3" />
</DockPanel>
```

In Version 2.0, it appears in the top strip. Note that the font size is 14 (compared to 11 for all the other controls) and the foreground color is blue. That was trivial to do in the properties pane of Visual Studio.

```
<DockPanel
    DockPanel.Dock="Top"
    LastChildFill="True"
    Height="30"
    Margin="0" VerticalAlignment="Stretch"
>
    <Button DockPanel.Dock="Left" Height="22"
    Name="btnSelectFolder" Width="84" Margin="3"
    Click="btnSelectFolder_Click">
        Browse...
    </Button>
    <TextBox DockPanel.Dock="Right" Margin="3" Height="22"
    Name="total"
        Width="60" Text="{Binding Path=Self,
    Converter={StaticResource DurationConverter}}"
        Foreground="Blue" TextDecorations="None" FontWeight="Bold"
        FontSize="14" Background="White" TextAlignment="Center"
        VerticalContentAlignment="Center"
    >
    </TextBox>
    <TextBox Height="22" Name="tbTargetFolder" MinWidth="258"
    Margin="3"
        TextChanged="tbTargetFolder_TextChanged"
    >
    </TextBox>
</DockPanel>
```

The data binding that populates the *total* field doesn't really care about its location or font, and continues to work just fine.

Check Boxes for Selection (One-Song Toggle)

Adding the check boxes is straightforward (see Figure 3). Remember that WPF and XAML are all about composition and nesting. But making the checkbox of each item in the list reflect the



Figure 2



Figure 3

selection state is not trivial. This is a classic use case for two-way binding. If the check box is clicked and becomes “checked,” you want the item to be selected (one way) and if the item is selected through some other mechanism like the user navigating with the arrow keys and hitting the space bar, you want the check box to become checked (the other way). After some “binging” I discovered that it requires a data template in XAML (of course, you can do anything in code too). You define the data template in the resources section of the main window and bind the *isChecked* property of the check box to the *isSelected* property of its *ListViewItem*. It sounds elegant, but the arcane incantations it requires can't be monitored easily just by reading the documentation. Here is the data template:

```
<DataTemplate x:Key="FirstCell">
  <StackPanel Orientation="Horizontal">
    <CheckBox IsChecked="{Binding Path=IsSelected,
      Mode=TwoWay,
      RelativeSource={RelativeSource FindAncestor,
        AncestorType={x:Type ListViewItem}}}"
    />
  </StackPanel>
</DataTemplate>
```

Once the data template is defined, it is simple to extend the list view to have a check box next to every item. All it takes is adding a new column to *GridView.Columns* (the first column):

```
<GridView.Columns>
  <GridViewColumn
    CellTemplate="{StaticResource FirstCell}"
    Width="30"
  />
  <GridViewColumn
    Header="Filename"
    DisplayMemberBinding="{Binding Path=Name}"
    Width="Auto"
  />
  <GridViewColumn
    Header="Duration"
    DisplayMemberBinding="{Binding Path=Duration,
      Converter={StaticResource DurationConverter}}"
    Width="Auto"
  />
</GridView.Columns>
```

That wasn't good enough though. My nifty little check boxes were nicely synchronized with the selected items, but the basic problem remained: If you select an item without holding control,

all your previous selections were replaced by the new item. The check boxes didn't enforce the behavior I wanted (item select/unselect toggles just the current item). I tried various solutions with ever more desperation but I couldn't fully override the selection behavior of the list view (I tried hooking into the routed event processing and directly manipulating the selected items). Eventually, I noticed that the list view has a little property called *SelectionMode*. The MSDN documentation is a little confused and doesn't mention it, but it does mention that it's a property of a list box (a list view is derived from list box and just extends it with a view). Anyway, the *SelectionMode* is set to “extended” by default, which is the bad behavior, but it also has a value of “Multiple”, which is exactly the behavior I was after. There is a “Single” value that allows you to select at most one item at a time. So, I set the *SelectionMode* to “Multiple” and my work was done (including seamless integration with the check boxes).

Here is the XAML if you must:

```
<ListView
  MinHeight="223"
  Name="files"
  MinWidth="355"
  Background="White"
  SelectionChanged="files_SelectionChanged"
  SelectionMode="Multiple"
  PreviewMouseRightButtonDown="files_PreviewMouseRightButtonDown"
>
  ...
</ListView>
```

Show the Number of Selected Songs

This was another trivial request, but it required some code changes. The “status” text box displays various status messages during the lifetime of the application (I probably should add a status bar for that). In theory, I could have a status object bound to the “status” text box, but that seemed like overkill just to change the contents of a text box every now and then (you would still need to update the status object from the code). The number of selected songs should be displayed in two separate cases. The first one is when the collection of tracking info is complete. During the collection process, the user may select already-collected songs and view their total duration, but the status text box displays the progress of the collection process. Once the collection is complete, it makes sense to display the current number of selected songs. The second case is of course when the number of selected songs is changed. The code is just two lines so I didn't even bother with a special method to update the status and just used the two-liner in both places. I use *String.Format()* to construct the message. Here is the first usage in *onTrackInfo()* when there are no more tracks:

```
void onTrackInfo(TrackInfo ti)
{
  if (ti == null)
  {
    // Update the status line
    var st = String.Format("{0}/{1} files are selected",
                          files.SelectedItems.Count,
                          _fileCount);

    status.Text = st;
    ...
  }
  ...
}
```

The second usage is in the *selectionChanged* event handler of the files list view:

```
void files_SelectionChanged(object sender,
SelectionChangedEventArgs e)
{
    ...
    // Update the status line
    var st = String.Format("{0}/{1} files are selected",
        files.SelectedItems.Count,
        _fileCount);
    status.Text = st;
    ...
}
```

Play Songs From Within MDC

Okay, here is a feature with some substance. The media element is fully capable of playing tracks and we already have one in our main window. However, it is not a good idea to reuse the same media element used for collecting track info for playing tracks because I want the user to be able to play tracks during the collection process. This is a very important feature because the collection process can last seconds (even minutes if a folder contains hundreds of tracks). Another dedicated media element for playing songs solves the problem. Note that the media element I use for collecting track info doesn't interfere with the player media element because it never tries to play anything. It just opens each file, extracts the duration, and closes the file.

I decided to go with a very spartan interface where you start and stop playing a song by right clicking with the mouse. There is no way to pause or skip ahead and no way to see how far you are into the current song.

The act of playing a song doesn't change its selection state. You may play selected or unselected songs, during and after the collection process.

The player media element is an invisible element because it just plays audio in this case, so it can be placed anywhere. I placed it next to the original media element in the *Grid* container of the main window:

```
<Grid>
  <MediaElement
    Height="0"
    Width="0"
    Name="mediaElement"
    LoadedBehavior="Manual"
  />
  <MediaElement
    Height="0"
    Width="0"
    Name="player"
    LoadedBehavior="Manual"
  />
  ...
</Grid>
```

The code to enable play/stop is simple. It is all located in the event handler for the *PreviewMouseRightButtonDown*. WPF uses routed events. Routed events travel through the control hierarchy, and any control along the way can handle the event and it will not be passed further. There are two types of routed events: tunneling events and bubbling events. Tunneling events travel from parent to child and allow the parent to handle the event on behalf of many children or do some global handling and then let the children

process them further. Bubbling events start at the child and bubble up to parents. The routed events travel in pairs. First, there is a tunneling event called *PreviewXXX* followed by a bubbling event called *XXX*. If some control handled the tunneling event, the bubbling event will NOT be fired. So, the *PreviewMouseRightButtonDown* event is a tunneling event that's fired when the user clicks the right mouse button on the files list view. The main window is the parent of the list view so it gets first shot at handling it. Immediately, it sets the *Handled* property to *true* so the list view will never get a chance to process it. The reason is that the right mouse button click will toggle the selection if allowed to reach the list view. This is behavior I want to suppress. There is a somewhat complex check to find the parent of the original source of the right mouse click and see if it's a *GridViewRowPresenter*. If this check succeeds, it means that the right click was indeed on one of songs in the list and not just a click in an empty area. Of course, that's when we want to play or stop the current song. First, I need to figure out the path to the song file, which I retrieve from the clicked item. Note that the *Content* property of the clicked item is a *TrackInfo* object.

Then, I stop the player in case it was playing (it may be stopped but this causes no harm). If the player *Source* property is null (no song) or anything other than the current clicked item, then I set the *Source* property to the new song and it starts playing. Otherwise (clicked on the same playing stop), I set the *Source* to null for cleanup purposes.

```
private void files_PreviewMouseRightButtonDown(object
sender,
MouseButtonEventArgs e)
{
    e.Handled = true;
    var x = ((FrameworkElement)e.OriginalSource).Parent;
    if (x is GridViewRowPresenter)
    {
        var ti = ((GridViewRowPresenter)x).Content as TrackInfo;
        string path =
System.IO.Path.Combine(tbTargetFolder.Text, ti.Name);
        player.Stop();
        if (player.Source == null || player.Source.AbsolutePath
!= path)
        {
            player.Source = new System.Uri(path);
            player.Play();
        }
        else
        {
            player.Source = null;
        }
    }
}
```

Copy Selected Songs To a New Folder

Okay, so you play with MDC, you select a bunch of songs out of your source directory, and now you want to burn them. Liat preferred to have all the selected songs in a dedicated folder so she could rename songs, rearrange them, and examine previous CDs.

With MDC 1.0 it was a grueling manual process. Liat fired up an Explorer window next to MDC, selected all the files she selected previously in MDC, created a new folder, and copied the files there. In MDC 2.0, this task can be done directly. I added a "Make CD..." button (see Figure 4) that opens a folder selection dialog,



Figure 4

which allows you to browse an existing folder or create a new one. Once you select a folder, the selected items are copied to the target folder.

As usual, XAML makes it easy to add a button. The button is docked to the right side of the bottom panel; it has a tooltip and it is disabled initially (you can't make a CD before some songs are selected). The *Click* event is hooked up to the *MakeCD_Click()* event handler:

```
<Button Name="makeCD" DockPanel.Dock="Right" Margin="3"
ToolTip="Copy selected songs to a new folder"
Click="MakeCD_Click" IsEnabled="False" >
```

```
Make CD...
```

```
</Button>
```

The button is enabled/disabled based on the selection state of the files list. This can be done with data binding, but I opted in this case to use imperative code in the *files_SelectionChanged()* event handler that's being called whenever the selection changes. It is just one line, and I handle enabling/disabling and the 'Select All' and 'Unselect All' buttons in the same place. The 'Select All' button is enabled if all the items are not selected. The 'Unselect All' and 'Make CD...' buttons are enabled if at least one item is selected.

```
selectAll.IsEnabled = files.SelectedItems.Count <
files.Items.Count;
```

```
unselectAll.IsEnabled = files.SelectedItems.Count > 0;
makeCD.IsEnabled = files.SelectedItems.Count > 0;
```

The *MakeCD_Click()* event handler implements two requested features: copying selected files to a folder and also exporting the song list to a CSV file.

The first step is to select the target folder. I reuse the *_folderBrowserDialog* for selecting the source folder. In order to remember the source folder, I save it in a temporary variable and restore it after the user selected a target folder. If the user selects the same folder as the source folder, MDC displays a message box and takes no action (no point in copying the files to the same folder, there are already there).

```
private void MakeCD_Click(object sender, RoutedEventArgs e)
{
```

```
    string savedPath = _folderBrowserDialog.SelectedPath;
    _folderBrowserDialog.ShowNewFolderButton = true;
    DialogResult r = _folderBrowserDialog.ShowDialog();
    if (r == System.Windows.Forms.DialogResult.OK)
    {
        string path = _folderBrowserDialog.SelectedPath;
        // check that the target folder is different than the
source folder
        if (path == savedPath)
        {
            System.Windows.MessageBox.Show(
                "Target folder is the same as source folder. no
action taken");
            return;
        }
        // restore selected path
        _folderBrowserDialog.SelectedPath = savedPath;
```

Now, it's time to actually copy all the selected files to the target folder. The .NET Framework provides in the *System.IO* namespace several classes to help with path manipulation and file operations. Many of these operations are implemented as static methods, which makes a lot of sense for operations like *Path.Combine* or *File.Copy* that don't have any state worth remembering for future operations:

```
// copy selected songs to target folder

foreach (TrackInfo t in files.SelectedItems)
{
    var src = System.IO.Path.Combine(savedPath, t.Name);
    var dst = System.IO.Path.Combine(path, t.Name);
    System.IO.File.Copy(src, dst);
}
```

Exporting the Song List To a CSV File

This is another feature Liat requested and it is also implemented in the *MakeCD_Click()* event handler immediately after copying the selected songs. The format is very simple. It is just a two-column list of *Name*, *Duration* for each selected item (see Figure 5). Liat can edit it later in Excel and add her class program next to the songs. It goes something like: go fast for 5 minutes, then accelerate for 8 more minutes and when you can't go any faster, increase the resistance until you start crying.

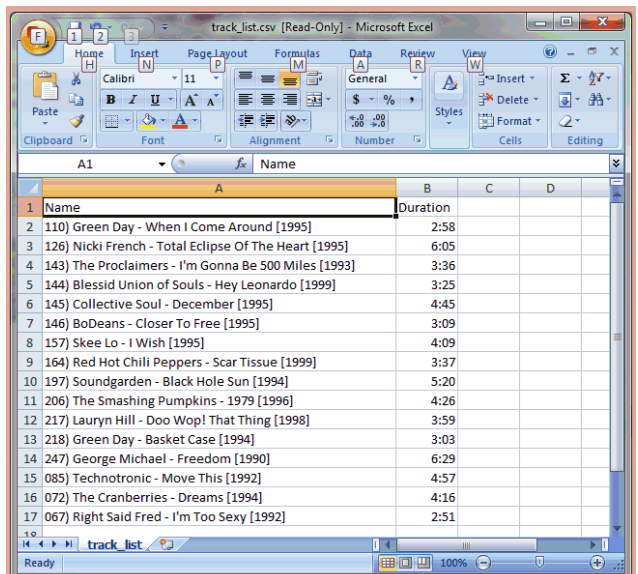


Figure 5

The Excel native file format is very complicated because it supports a lot of functionality. But, just for importing tabular data into Excel you can use a CSV (Comma-separated values) file. It is very simple to create such a file. You have to be careful to make sure your values don't contain a comma (I don't do it here).

The filename is *track_list.csv* and it is stored in the target folder together with all the songs. I utilize the *using* statement that ensures correct use of *IDisposable* objects. These objects have a *Dispose()* method that should be called to ensure proper cleanup and the *using* statement does it at the end of the block even if an exception is thrown. I like to think of it as the C# equivalent of the C++ RAII design pattern (http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization).

The *using* statement creates the *track_list.csv* text file, which is guaranteed to be closed properly at the end of the block. Inside the *using* block the code writes a header "Name", "Duration" to the *StreamWriter* object and then iterates over the selected *TrackInfo* items and writes each item's name and duration with some formatting to make it look nicer.

```
// Create CSV file with track list
var filename = System.IO.Path.Combine(path, "track_list.csv");
using (var sw = File.CreateText(filename))
{
    sw.WriteLine("Name,Duration");
    foreach (TrackInfo t in files.SelectedItems)
    {
        var d = new DateTime(t.Duration.Ticks);
        var name =
            System.IO.Path.GetFileNameWithoutExtension(t.Name);
        sw.WriteLine(name + "," + d.ToString("mm:ss"));
    }
}
```

Remember the Last Folder

Programs that remember where the user left off are much nicer to use. The .NET Framework provides standard infrastructure to persist application settings and it's well integrated with Visual Studio (see Figure 6). I only use it to remember the last source folder, but it can be extended to store any number of things like the size and position of the main window, the selected items in the current session and more.

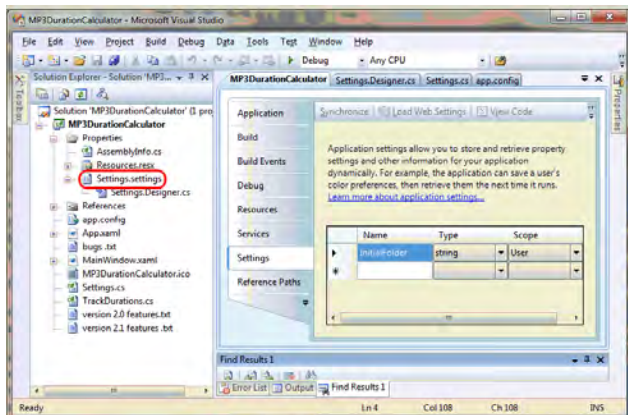


Figure 6

There are several other issues you have to take into account when thinking about persistent application settings such as write access and shared settings between all users vs. per-user settings.

Once you figured it out, it is pretty easy to use application settings (especially in Visual Studio). You define your properties in the settings tab. Visual Studio takes care of adding two files: *Settings.Designer.cs* and *Settings.cs*. These files together define the sealed *Settings* class (via partial classes in each file) under the *MP3DurationCalculator.Properties* namespace. The *Settings.Designer.cs* file contains a static instance of the default settings and get/set properties for each setting:

```
//-----
// <auto-generated>
// This code was generated by a tool.
// Runtime Version:2.0.50727.3603
//
// Changes to this file may cause incorrect behavior and
// will be lost if
// the code is regenerated.
// </auto-generated>
//-----

namespace MP3DurationCalculator.Properties {

    [global::System.Runtime.CompilerServices.CompilerGeneratedAttribute()]

    [global::System.CodeDom.Compiler.GeneratedCodeAttribute("Microsoft.VisualStudio.Editors.SettingsDesigner.SettingsSingleFileGenerator", "9.0.0.0")]

    internal sealed partial class Settings :
        global::System.Configuration.ApplicationSettingsBase {
        private static Settings defaultInstance = ((Settings)
            (global::System.Configuration.ApplicationSettingsBase.Synchronize
            (
                new Settings())));
        public static Settings Default {
            get { return defaultInstance; }
        }
        [global::System.Configuration.UserScopedSettingAttribute()]
        [global::System.Diagnostics.DebuggerNonUserCodeAttribute()]
        [global::System.Configuration.DefaultSettingValueAttribute("")]
        public string InitialFolder {
            get { return ((string)(this["InitialFolder"])); }
            set { this["InitialFolder"] = value; }
        }
    }
}
```

The *Settings.cs* file contains optional event handlers you can implement to respond to a change in settings and to intercept the saving process:

```
namespace MP3DurationCalculator.Properties
{
    // This class allows you to handle specific events on the
    // settings class:
    // The SettingChanging event is raised before a setting's
    // value is changed.
    // The PropertyChanged event is raised after a setting's
    // value is changed.
    // The SettingsLoaded event is raised after the setting
    // values are loaded.
    // The SettingsSaving event is raised before the setting
    // values are saved.
    internal sealed partial class Settings
    {
        public Settings()
        {
            // To add event handlers for saving and changing
            // settings, uncomment the
            // lines below:
            //
        }
    }
}
```

```
// this.SettingChanging +=
this.SettingChangingEventHandler;
//
// this.SettingsSaving += this.SettingsSavingEventHandler;
//
}
private void SettingChangingEventHandler(
    object sender,
    System.Configuration.SettingChangingEventArgs e)
{
    // Add code to handle the SettingChangingEvent event here.
}
private void SettingsSavingEventHandler(
    object sender,
    System.ComponentModel.CancelEventArgs e)
{
    // Add code to handle the SettingsSaving event here.
}
}
```

MDC saves its settings in the *OnExit* method of the *App* class:

```
namespace MP3DurationCalculator
{
    public partial class App : Application
    {
        protected override void OnExit(ExitEventArgs e)
        {
            base.OnExit(e);
            MP3DurationCalculator.Properties.Settings.Default.Save();
        }
    }
}
```

When MDC is loaded, the settings are loaded automatically and are available in the *MainWindow* constructor. If the initial folder doesn't exist, then the user's MyMusic special folder is used as an initial folder:

```
public partial class MainWindow : Window
{
    public MainWindow()
    {
        ...

        var initialFolder =
            Properties.Settings.Default.InitialFolder;
        if (!Directory.Exists(initialFolder))
            initialFolder =

            Environment.GetFolderPath(Environment.SpecialFolder.MyMusic);
        ...
    }
    ...
}
```

Give Me More...

Liat was pleased with all the changes and new features, but with the food comes the appetite. Once the initial excitement over MDC 2.0 subsided, Liat wanted more features:

- Make CD from multiple source folders (will require merging of track_list.csv files)
- A full-fledged media control bar: play, pause, stop
- A seek bar that allows skipping and also displays the current position of the song
- Maybe a little database to manage prepared CDs and mark songs that were used already.

I guess this is a good sign. There are only two types of programs: the programs that users complain about and the programs they don't use.

Conclusion

The .NET Framework and WPF are a superb platform for developing rich client applications. It is designed well and supports both enterprise-grade applications as well as small, fun utilities like MDC. I was able to extend MDC with a significant amount of user-requested functionality with minimal changes to the XAML and the code.

— *Gigi Sayfan specializes in cross-platform object-oriented programming in C/C++/ C#/Python/Java with emphasis on large-scale distributed systems. He is currently trying to build intelligent machines inspired by the brain at Numenta (www.numenta.com).*

[Return to Table of Contents](#)

Windows 7 Libraries

Windows 7 libraries are explored and the Sensor and Location platform is explained. Click screen to watch a video about Windows 7 libraries on Dr. Dobb's Microsoft Resource Center.



Click screen to link to video

C Snippet #3

Sorting an array of strings

by Bob Stout



: HOW DO I...sort an array of strings?

A: Sorting jobs come in all sizes. One of the most common is simply sorting an array of strings. Unless the array is very large, the standard `qsort()` function is often inefficient for such tasks. In such situations, you might consider the Shell sort presented in this Snippet. Call it with only two arguments, a pointer to the string array to sort and the number of strings in the array.

```
/*
** strsort() -- Shell sort an array of string pointers via strcmp()
** public domain by Ray Gardner   Denver, CO
*/

#include <string.h>
#include <stddef.h>

void strsort (char **array, size_t array_size)
{
    size_t gap, i, j;
    char **a, **b, *tmp;

    for (gap = 0; ++gap < array_size; )
        gap *= 2;
    while (gap /= 2)
    {
        for (i = gap; i < array_size; i++)
        {
            for (j = i - gap; j -= gap)
            {
                a = array + j;
                b = a + gap;
                if (strcmp(*a, *b) <= 0)
                    break;
                tmp = *a;
                *a = *b;
                *b = tmp;
                if (j < gap)
                    break;
            }
        }
    }
}
```

More C Snippets

- Determining a file's size (<http://www.drdobbs.com/cpp/219100141>)
- Rounding floating-point values (<http://www.drdobbs.com/cpp/219200409>)
- Sorting an array of strings (<http://www.drdobbs.com/cpp/219500313>)
- Computing the wind chill factor (<http://www.drdobbs.com/cpp/219700315>)
- Timers and default actions (<http://www.drdobbs.com/cpp/220001077>)

All the code in C Snippets is either public domain or freeware and may therefore freely be used by the C programming community without restrictions. In most cases, if the original author is someone other than myself, he or she will be identified. Thanks to all who have contributed to this collection over the years. I hope *Dr. Dobb's* readers will find these useful.

—Bob Stout

rbs@snippets.org

[Return to Table of Contents](#)

Java and C++ Socket Communication

Integrating C++ native Windows applications with Java applications via network sockets

by Eric J. Bruno

Performing all of your application development with a single language on a single platform may be ideal, but it's not always practical. There are times when you may need to integrate a new application with a legacy one, and communication between the two can be an issue. For instance, you may desire to isolate the two applications so that the new application's design isn't compromised, and the older one can be upgraded later without impacting the newer application.

Potential Solutions

In the past, I've explored distributed computing solutions that have discussed integrating Java and C++ applications via the Java Native Interface (JNI), Java Message System (JMS), and web services; see the "Conclusion" section at the end of this article. Although these approaches are good in the right situations, there may be times where these solutions are too complicated or just not ideal. For instance, calling into native code from Java via JNI can be complex, time-consuming, and error-prone. Using JMS requires a JMS provider be licensed, installed, and configured. And a web service requires significant development and web-based infrastructure.

Another solution is to use socket-based network communication directly between the Java and C++ applications. Although this is a relatively low-level approach, it's still an effective solution. With XML as the message protocol between them, you can maintain a degree of platform and language independence. Figure 1 illustrates this in a very simple way; here we show that a C++ application that uses Windows sockets can communicate with a Java application that uses Java IO (scenario c) just as easily as in the two homogenous examples (scenarios a and b), and with no code changes.

The Socket Solution

Let's explore a sample integration solution where a Java application uses *java.io* to communicate with a C++ Windows application that uses Windows sockets. The Windows application supports three simple requests for the following data:

- The Windows host name
- The amount of memory installed
- A pseudorandom number

These requests are simple for illustration only, but in reality they may be requests for data only available from a legacy application or a native platform interface. All requests and responses are formatted as simple XML character strings. Specifically, both the client and server wait for data and read bytes from the network stream, with each full message delineated by a NULL — or '\n' — character.

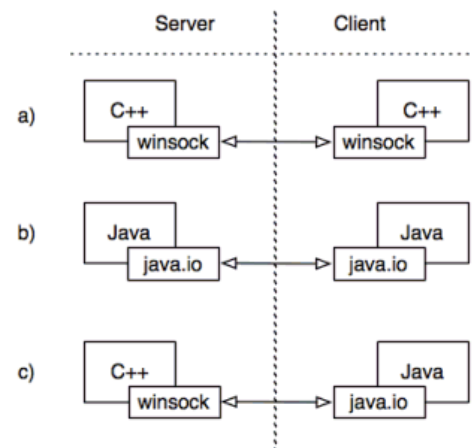


Figure 1: A Java application can communicate directly with a C++ winsock application.

The simple XML request messages are in Example 1. Each message has the same basic XML structure, with only the request name differing between them.

Example 1: The XML request messages sent from the client, to the server.

```
<Request>
  <Name>GetHostname</Name>
</Request>

<Request>
  <Name>GetMemory</Name>
</Request>

<Request>
  <Name>GetRandomNumber</Name>
</Request>
```

If you need to send data along with each request, simply add one or more XML elements to the request message. For example, if you want to change the *GetMemory* message to indicate the type of memory (physical or virtual) to request, the XML can be changed to look like the following:

```
<Request>
  <Name>GetMemory</Name>
  <Type>Physical</Type>
</Request>
```

The response messages are similar to the requests with some obvious changes; see Example 2. The main differences are the XML response types, and the inclusion of the data being requested.

Example 2: The XML response messages sent from the server, in response to client requests.

```
<Response>
  <Name>HostnameResponse</Name>
  <Hostname>MyServer</Hostname>
</Response>

<Response>
  <Name>MemoryResponse</Name>
  <TotalPhysicalMemory>1073201152</TotalPhysicalMemory>
</Response>

<Response>
  <Name>RandomNumberResponse</Name>
  <Number>20173</Number>
</Response>
```

In this sample implementation, the actual data (bold in Example 2) returned in each response will vary according to the computer it's run on. Let's begin to dive into the implementation of the solution, beginning with the Java client application.

The Java Client

The sample Java client is written as a Java Swing application, which includes a user interface with buttons to make requests, and a list view to display the server responses (see Figure 2). There's also a button labeled *Connect*, and when pressed, the client attempts to connect to the server at the supplied address. The address defaults to localhost, but you can change this to any valid hostname or IP address. As a result, you have the choice to run

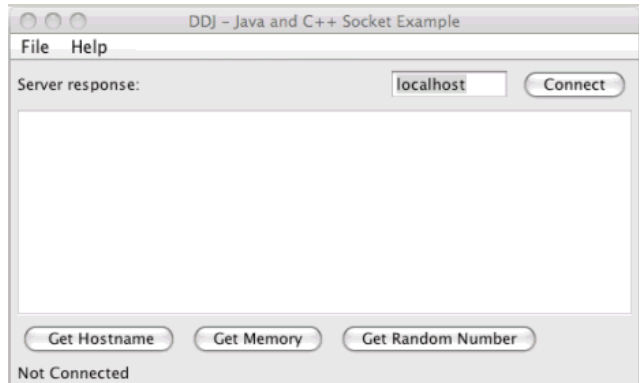


Figure 2: The sample Java client application user interface.

both the client and server applications on one computer, or on separate computers, without issue. We'll examine the sample C++ server application later.

The heart of the application is the Java network and IO code, which is encapsulated inside one class, *SocketClient*, shown partially in Example 3. This class contains two embedded classes called *Listener* and *Sender*, which listen for requests and send responses, respectively.

Example 3: The *SocketClient* class implements all of the code to send requests and listen for server responses.

```
public class SocketClient {
    private Listener listener = null; // listens for requests
    private Sender sender = null;    // sends responses
    public boolean connected = false;
    ServerResponses viewer = null;

    class Listener extends Thread {
        // ...
    }

    class Sender {
        static final String HOSTNAME = ...
        static final String MEMORY = ...
        static final String RANDOM_NUM = ...
        // ...
    }

    public SocketClient(String IPaddress,
                        ServerResponses viewer) {
        // ...
    }

    public boolean isConnected() {
        return connected;
    }

    public void requestHostname() {
        sender.requestHostname();
    }

    public void requestMemory() {
        sender.requestMemory();
    }

    public void requestRandomNumber() {
        sender.requestRandomNumber();
    }
}
```

A connection to the server is established when a *SocketClient* object is created. The first parameter of the constructor is the address of the server to connect to, and the second parameter is a reference to an object that implements the *ServerResponses* interface, shown in Example 4.

Example 4: The `ServerResponses` interface methods are called as response messages arrive.

```
public interface ServerResponses {
    public void onHostnameResponse(String msg);
    public void onMemoryResponse(String msg);
    public void onRandomNumberResponse(String msg);
}
```

As a result, the user interface code drives the connection and request process by instantiating the `SocketClient` class when the `Connect` button is pressed, and later calling any of its public methods — `requestHostname()`, `requestMemory()`, and `requestRandomNumber()` — when the associated button is pressed. When a response is received from a prior request, the appropriate method is called on the `ServerResponses` reference provided.

In this sample Java client, each time you press one of the request buttons, a request message is sent to the server, and the associated response message text is displayed at the top of the list view (see Figure 3).

Here, you also see that the display has been updated to show a connection to the server at address 192.168.0.180. Let's examine the Java network communication code in more detail.

Java Network Programming

When the `SocketClient` class is created (Listing 1, the complete implementation, and related files are available at <http://i.cmpnet.com/ddj/images/article/2010/code/JavaCppSockets.zip>.

The code in the constructor attempts to connect to a server at the given IP address on port 8080. This code looks similar to the following:

```
try {
    java.net.Socket socket =
        new java.net.Socket("localhost", 8080);
    this.listener = new Listener(socket);
    this.sender = new Sender(socket);
}
catch (Exception e) {
    e.printStackTrace();
}
```

The code is simple:

- If the connection attempt fails, an *Exception* is thrown.
- If the connection attempt succeeds, the active socket is passed to the *Listener* and *Sender* classes to perform the real work.

To send a network message using this active socket, you need to use Java IO streams, which are part of the *java.io* package. The *Sender* class encapsulates this implementation, but in summary, to send a network message you need to begin with the socket's output stream, which is used to send data. For efficiency, we wrap this stream in a *java.io.BufferedOutputStream* object, and then write the message's bytes to this stream:

```
String xml = "<Request><Name>GetHostname</Name></Request>";
BufferedOutputStream bos =
    new BufferedOutputStream( socket.getOutputStream() );
bos.write( xml.getBytes() ); // writes to the buffer
bos.flush(); // writes the buffer to the network
```

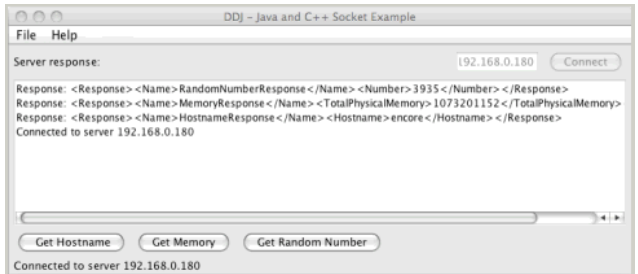


Figure 3: The Java client displaying some server responses.

The code to listen for network messages is similar in that it also uses Java IO streams. However, due to the nature of waiting for a message, you typically perform this task in a separate thread. The *Listener* class extends class *Thread*, and makes its call to one of the blocking `read()` methods in the *Thread.run()* method when started.

Executing this code in a separate thread ensures that the rest of the application is able to execute (and respond to user input, for instance) while the blocking `read()` method waits for data for an unpredictable length of time. This solution is simple and elegant enough for this application, but for applications that need to communicate over many socket connections, creating a thread per socket may not scale well. Alternatively, you can specify a timeout on the `read()` method and call it periodically (polling), or perform non-blocking IO (beyond the scope of this article) in those situations.

The code in the *Listener* class that performs this task is similar to the following:

```
public void run() {
    // Create a reader to wrap the socket's InputStream
    InputStreamReader inputReader =
        new InputStreamReader(socket.getInputStream());
    BufferedReader bufferedReader =
        new BufferedReader(inputReader);
    while (true) {
        // readLine blocks until a message arrives
        String xml = reader.readLine();
        // process the XML ...
    }
}
```

As stated earlier, the Java Swing code controls the sending of request messages as the user clicks on the appropriate buttons, and subsequently displays the server response messages in a list view. Listing 2, which contains this implementation and the complete Java client application and related files are available at <http://i.cmpnet.com/ddj/images/article/2010/code/JavaCppSockets.zip>.

The C++ Windows Application

Let's examine the C++ Windows server application, which also uses sockets to communicate over the network. This implementation is unaware of whether the client connecting to it is written in Java, C/C++, or any other language. The use of XML further ensures that the communication will succeed.

When the server application starts, an instance of the *ServerSocket* class is created (Listing 3 and the complete C++ server are available at <http://i.cmpnet.com/ddj/images/article/2010/code/JavaCppSockets.zip>), which listens on port 8080 for new client connections. In summary,

this class uses Windows Sockets to create a network socket, and then listen on it and accept new client connection requests as they arrive:

```
// Start winsock and create the socket
WSAStartup(...);
listenSocket = socket(...);

// TCP bind and listen
bind( listenSocket, ... );
listen( listenSocket, SOMAXCONN);

while ( listening ) {
    // Accept a client socket (
    clientSocket = accept(listenSocket, NULL, NULL);
    // ...
}
```

Given that the call to *accept()* blocks until a new client connects, this code is executed in its own thread. Once a client does connect, the resulting code is similar to the Java socket implementation in that objects are created to handle sending data over the socket, and listening for data to arrive:

```
// Setup the socket sender and listener
sender = new SocketSender(clientSocket);
listener = new SocketListener(clientSocket);
listener->setSender(sender);
listener->startThread();
```

Again, as with the Java socket code, the *SocketListener* class executes the Windows Sockets *recv()* method, which blocks until data arrives, in its own thread:

```
char recvbuf[BUFFERLEN];
int bytes;
while ( true ) {
    bytes = recv(clientSocket, recvbuf, BUFFERLEN, 0);
    if (bytes > 0) {
        recvbuf[bytes-1] = 0; // null terminate the string
        HandleMessage(recvbuf);
    }
}
```

Each request is further processed in *HandleMessage()*, and the appropriate response is sent back to the client:

```
int num = rand();
sprintf_s(buffer, "<Response><Name>RandomNumberResponse</Name>\n<Number>%i</Number></Response>\n", num);
send( clientSocket, buffer, strlen(buffer), 0 );
```

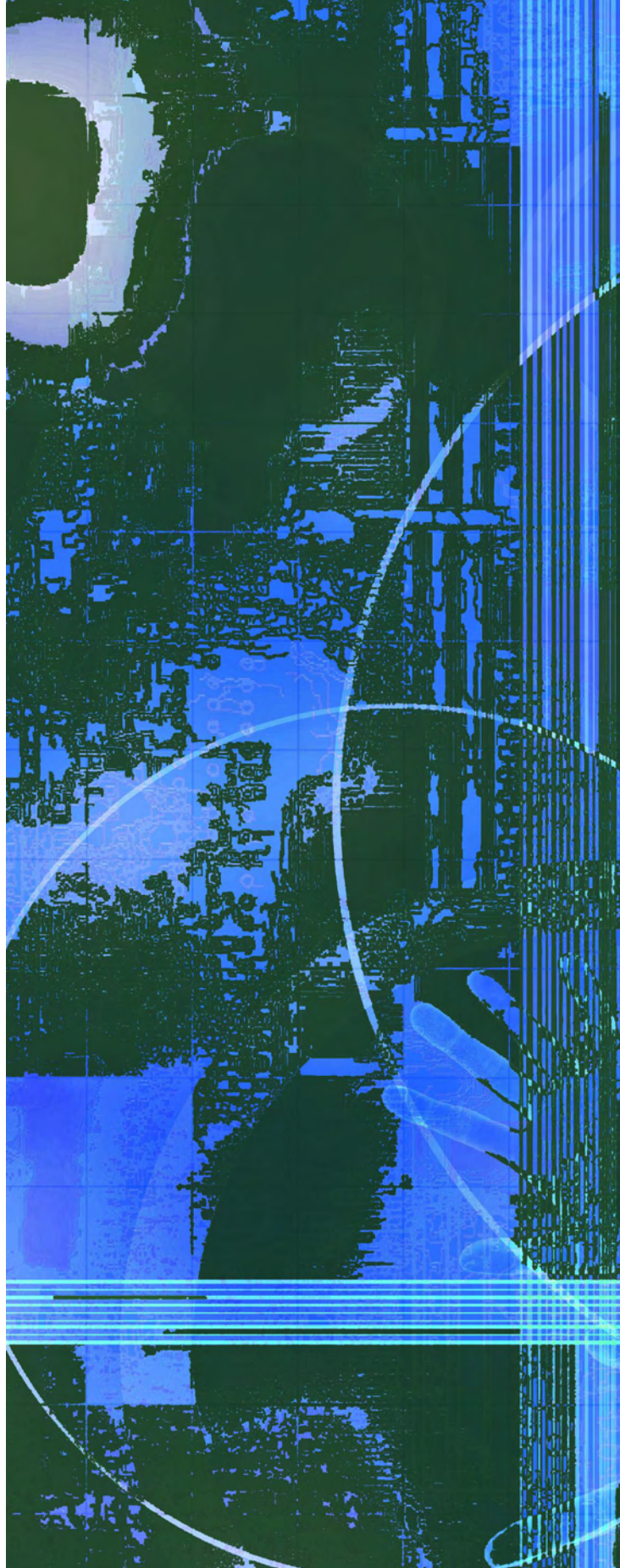
Conclusion

You should now have a basic understanding of how Java and C/C++ applications can communicate via network socket programming. In fact, although the sample server application explored here uses Windows sockets, a comparable Linux application can be developed to work equally well with the Java client presented here. Combined with XML, this should provide a suitable solution in some cases. To learn about other application integration techniques, such as with JNI, JMS, or web services, take a look at the following articles:

- C++, Java, and XML Processing (www.drdobbs.com/cpp/184401817)
- The Java Message Service (www.drdobbs.com/java/200001958)
- Java, C++, and SOA (www.drdobbs.com/cpp/184401849)

— Eric J. Bruno is an author and professional technology consultant.

[Return to Table of Contents](#)



The Jazz Process

What can software developers learn from jazz musicians?

by Adrian Cho

Software development projects fail all the time. Teams deliver late, blow out budgets, underwhelm or disappoint users with missing or deficient functionality, or simply crash-and-burn before reaching their goal. Why do some projects and teams perform so poorly? For decades the practice of software development has focused intently on tools, technology, and intricate, heavyweight processes. Wave after wave of programming interfaces, frameworks, toolkits, and methods have come and gone, each accompanied by a horde of evangelists and zealous practitioners. These people will be the first to tell you that failed projects are a result of using the wrong technology or process. They may even try to sell you the right one. While such things can make a difference, there is an easier way to fail.

In software development, multidisciplined teams must collaborate and innovate with agility. Disciplines of requirements analysis, design, development, testing, finance, and in the case of software products, sales, marketing, and support, must work in concert to produce creative solutions while constantly reacting and responding to change. Change may be generated from within an organization or externally. Changing business conditions or the actions of partners or competitors are examples of external change. A software development team is similar to a basketball team, a squad of soldiers, or a group of jazz musicians. This may be surprising since each activity has completely different domain-specific practices. Basketball players must be able to handle a ball and implement game strategy. Soldiers must be skilled in various aspects of combat while following orders. Jazz musicians must be able to play instruments and navigate through a musical form. Software developers must be able to write code and adhere to a development process. Yet none of these groups can succeed without effective collaboration and the ability to execute on a plan, or as it sometimes happens, in the absence of a plan.

Like software developers, jazz musicians must work in synergy to deliver unique, high-quality offerings that will attract and retain customers. They must deliver on-time and in real-time, often under continuous scrutiny. They do this by integrating strong individual contributions from passionate and committed practitioners and ensuring success with best principles. The Jazz Process (<http://www.jazzprocess.com/>) is a framework for improving collaboration, innovation, and agility with a method for execution and 14 best principles that act on that method. While its primary inspiration is jazz performance, the process draws on examples of excellence in business, software development, music, military operations, and sports, while applying laws of sociology, psychology, physics, biology, and systems theory.

The execution method of the Jazz Process is based on the OODA loop (http://www.valuebased-management.net/methods_boyd_ooda_loop.html) developed by Colonel John Boyd, a legendary maverick U.S. Air Force fighter pilot, military strategist, and aircraft designer. Boyd believed that agility was the key to winning dogfights or any other competitive situation. Tactics used in Desert Storm were patterned after Boyd's theories and many business methods such as Plan-Do-Check-Act (PDCA; see <http://www.asq.org/learn-about-quality/project-planning-tools/overview/pdsa-cycle.html>) are similar to his method. The name "OODA" comes from the four phases in the loop:

- **Observe:** Acquire all potentially relevant data including that which originates from collaborators (those we work with), consumers (those we work for), and competitors (those we work against).
- **Orient:** Interpret that data as useful information.
- **Decide:** Make the best decisions while avoiding groupthink, the not-invented-here syndrome, and the many other cognitive biases that can skew good individual or group decision-making.
- **Act:** Execute on the decision in a correct and rapid manner.

The goal is to do all of this more rapidly than the competition, although it's important that execution speed shouldn't sacrifice quality and correctness. Paths of reobservation (look twice), reinterpretation (reconsider what was observed), and evaluation (try something before committing to a decision) may increase cycle time while feed-forward (predict by guessing or experience) may shorten it. Individuals, teams, and organizations all execute in OODA loops.

The principles of the Jazz Process affect the execution of the loop. There are five principles in support of working.

- **Use Just Enough Rules** is right-sized governance with just enough process to avoid chaos and confusion while affording people the autonomy to get things done. Processes must be continually improved by being honest about problems and less-than-optimal execution and updating rules and practices for greater future success. Tools such as IBM's Rational Team Concert (<http://www-01.ibm.com/software/awdtools/rtc/>), built on IBM's Jazz team collaboration platform, allow processes to be defined and customized in a way that tools can enact and enforce the process and guide people to do the right thing.
- **Employ Top Talent** focuses on hiring and retaining the best people possible. Aim for quality and not quantity. One genius may succeed where 1,000 others failed. The Manhattan Project's race to develop the atomic bomb employed more than 130,000 people but it was a small group of individuals, the project's scientific and engineering geniuses, who were most vital to the project.
- **Put the Team First** places the team's goals before those of any individual. The team succeeds together and absorbs mistakes together.
- **Build Trust and Respect and Commit with Passion** are essential. Trust enables everything to happen more quickly. Respect allows people to contribute without fear. Commitment wills people to push through in spite of challenges. Infectious passion excites and motivates others to participate and consume.

There are four principles in support of collaborating.

- **To Listen for Change** is to know what's going on at all times. Not surprisingly, jazz musicians place great importance on the skill of listening. If one jazz musician remarks that another jazz musician has "big ears," it's a compliment that means the person is constantly aware and ready to respond to change and that they are open to exchanging ideas. Listening in the abstract sense is acquiring all relevant data to understand what's happening in the team, what customers are thinking, and what competitors are doing. Team awareness is vital to working together. It helps avoid conflicting and duplicate efforts and facilitates synergy from complementary efforts. Rational Team Concert makes team awareness a high priority, and this is especially important for geographically-distributed teams.
- **Lead on Demand** is about taking initiative. However, this can only happen when organizations decentralize leadership and give people the power to make decisions and act.
- **Act Transparently** focuses on open and honest communication. When others can see what's going on it makes it easier for them to collaborate. Customers also appreciate openness, hence the success of open source projects and IBM's Open Commercial Software Development taking place at jazz.net.
- **Making Contributions Count** is to carefully consider the effort, value, and impact of everything we do. It only takes one careless contribution to break a software development build.

Jazz Process Principles

1. WORKING

1. Use Just Enough Rules
2. Employ Top Talent
3. Put the Team First
4. Build Trust and Respect
5. Commit with Passion

2. COLLABORATING

6. Listen for Change
7. Lead on Demand
8. Act Transparently
9. Make Contributions Count

3. EXECUTING

10. Reduce Friction
11. Maintain Momentum
12. Stay Healthy

4. INNOVATING

13. Exchange Ideas
14. Take Measured Risks

In support of executing are three principles.

- **Reduce Friction** is actually about optimizing friction. Friction is resistance that can increase the difficulty and/or time taken to complete a task. Some causes of friction don't seem like a problem until you have to perform that task frequently. Yet you must have some friction. Without it we couldn't walk and objects would slide off even slightly sloped surfaces. Business controls such as legal and finance are often seen as impediments by software developers, but without any such controls an organization exposes itself to intolerable risk.
- **Maintain Momentum** promotes continual progress by establishing regular cycles and rhythms for such tasks as planning, software builds, meetings, and so forth. This is the essence of iterative software development methods.
- **Stay Healthy** alerts us to the fact that injury to a project or team can impair performance. Left untended, poor health may degrade to a total loss of stability. It also discourages people from consuming or participating — the Broken Window Theory. In software development, build health is everything and broken builds must be repaired immediately.

Finally, innovating is pursued through two principles:

- **Exchange Ideas** is the key to innovation fueled by diversity. A team with diversity in technical experience increases divergent thinking, which enables more possible solutions to be generated. At the same time, it increases convergent thinking, which improves selection of the best possible solution.
- **Take Measured Risks** is reality. While it makes sense to only undertake tasks that are free of risk, it's not really practical. Few things that are truly risk-free, and anything that is will most likely be exploited by your competitors, thereby reducing the likelihood that it could provide you with a competitive advantage.

— *Adrian Cho has worked in the software development industry for more than 20 years. He currently manages the development of IBM's Rational Team Concert and Rational's Collaborative Application Lifecycle Management project at jazz.net. Adrian is also a freelance jazz musician, the Artistic Director of the Impressions in Jazz Orchestra, and the author of upcoming The Jazz Process: Collaboration, Innovation and Agility (www.jazzprocess.com/the-book/). You can contact Adrian at jazzprocess.com.*

[Return to Table of Contents](#)

Estimating on Agile Projects

Upfront estimating from an agile viewpoint

by Scott W. Ambler

Many people who are new to agile seem to struggle with how estimating occurs on agile projects. I suspect that this is due in part to the fact that many people still cling to very serious misunderstandings about estimation, such as being able to predict up front with precision what the cost of a project will be, and due in part to the rhetoric within the agile community around the “evil practices” of traditional project management. In this article, I start with a brief discussion for the need for realistic expectations for initial estimating, then I describe the process of initial estimating from an agile viewpoint, and then I describe how to accurately update your estimates throughout the agile lifecycle.

As Barry Boehm has shown us again and again, we need to give ranged estimates that reflect the amount of uncertainty in the information on which our estimates are based. This holds for both cost and schedule estimates. Early in the project there is greater uncertainty so the range is greater, but as the project progresses the quality of the information improves, decreasing the uncertainty and therefore the range in your estimates decreases. This is something that Boehm calls the “cone of uncertainty.” At the beginning of a project, due to the inherent uncertainty, it is reasonable to give estimates in the $\pm 30\%$ range. Sadly, senior management, ever optimistic and unfailingly unable to learn from previous experiences, asks for a $\pm 11\%$ range on average (see the *Dr. Dobb's* July 2009 State of the IT Union Survey; <http://www.ambysoft.com/surveys/stateOfITUnion200907.html>) and on average gets $\pm 19\%$ in practice. If we're actually getting $\pm 19\%$, then why am I recommending $\pm 30\%$? In addition to the rather large body of knowledge about estimation, some of which recommends a far greater range, the *Dr. Dobb's* survey also found that to get $\pm 19\%$ project managers often had to do a fair bit

of fudging such as padding the budget, dropping scope once the budget ran out, or even updating the original estimates at the end of the project to better align with the actual results (yikes). This implies that we need an initial range far greater, on average, than 19%.

There are many strategies for initially estimating the cost and schedule of an IT project, but they all boil down to the same basic idea: Gather some information about the requirements, do some thinking about the potential solution(s), and then put this information through your estimation algorithm of choice. The individual strategies will potentially vary by the level of detail in the requirements, the formatting of those requirements, the amount of detail of the architecture/design specification, and in the factors addressed by the algorithm. For example, COCOMO II and SLIM both require fairly detailed requirements and design specifications and vary mostly on algorithm minutiae (yes, as you'd expect there are raging religious debates about this within the estimation community). Both techniques require you to analyze your detailed specifications to count certain functional and technical aspects to plug into their complex estimation algorithms. On the other hand, estimation strategies around use case points and user story points vary mainly on the format of the requirement, both of them requiring significantly less specification but relying on the ability of the team to guess at the size of the individual requirement. Once each high-level requirement, use case, or user story, respectively, is given a point rating, the points are merely added up and multipliers applied to give cost and time estimates. Other estimation strategies go even further in that they skip over the detailed use case/user story point counting step and go straight to educated guess(es) from experienced people.

On the surface of things, intuition tells us that the estimation techniques based on detailed specifications should be more accurate than the techniques which rely on the best guesses of the people involved. Sadly, our intuition is pretty much wrong on this issue. The challenge is that there is still a lot of guessing going on with the detailed approaches. In theory, the errors in each of the little guesses should balance out and the overall error should be much smaller as a result. This would be true if the little guesses were independent of each other, but the problem is that they're not. All of the little guesses are dependent on the ability of the stakeholders to describe their requirements, something we know that people aren't good at; on the ability of the people doing the analysis; on the ability of the people developing the detailed architecture; on the ability of the estimator(s); and on the estimator's knowledge of the skills and ability of the people doing the actual work. In short, the guesses based on detailed information still have an incredible amount of uncertainty built into them, yet they appear to be less risky than the estimates based on less-detailed specifications because of the amount of effort put into calculating them. Arguably the only thing that these complex estimating techniques are doing is putting a "scientific facade" over an activity that is still mostly an art.

The reality is that you're going to need to do some initial estimating because at the beginning of a project someone is going to ask you to indicate what you think you're going to deliver, how you're going to do so, how much it's going to cost, and how long you think it will take. Because of the inherent uncertainty in the information that you have available to you at the beginning of a project, you'll need to give ranged answers at first and then tighten up your answers as the project progresses. The Agile Modeling methodology (<http://www.agilemodeling.com/>) describes techniques applicable to the first two questions. Let's explore how to address the questions about the cost and schedule estimates.

A common progress reporting technique in the agile community is something called a "burn down chart". Burn down charts show the amount of work on the project along the vertical "Y" axis and time along the horizontal "X" axis. The change in remaining work to do is called the "burndown rate," something that varies throughout the project for several reasons. Based on how much work is left to do, and the burn down rate, you can easily project when you expect to be done. Then, given the expected end date and your current labor costs, you can estimate the overall labor portion of the expected cost of the project (non-labor costs, such as hardware costs, travel costs, and so on will still need to be guessed).

Let's work through an example. You're a member of an agile project team that started on January 1. You spent the month of January in "sprint 0" — which is also called iteration 0, the warm up phase,

or the inception phase — where you performed requirements envisioning to identify the initial high-level requirements, architecture envisioning to identify a realistic technical strategy, developed a shared vision, obtained support for the project, put together the team, and obtained a working environment. You've been asked by senior management to deliver the system by August 31. Your sprint length is two weeks, your guess at your initial velocity is 20 points per sprint, your burdened labor costs are \$5000 per week per employee, you think you need four people on the team, and you have 300 points of work on your product backlog. So, with a burn-down rate of 20 points, your team projects will require 15 sprints to complete construction ($300/20$), which is 30 weeks (15×2). You also believe that it will take an additional four weeks to release the system into production, with an expected release date of 34 weeks from February 1st, which is September 24th. There is also an expected labor cost of \$680,000 ($34 \text{ weeks} \times \$5,000 \times 4 \text{ people}$).

Because you're also overly optimistic and unable to learn from previous experiences, your team feels that it's possible that they can deliver by the end of August as requested. Worse yet, because of cultural dysfunction within your organization, the team chooses not to give a ranged estimate because in the past senior management simply forces the team to commit to the lower estimate in a naive belief that this will improve the team's motivation to deliver. In the end, senior management takes your August estimate as a commitment, even though this estimate was based on guesses about the scope and the ability of the team to deliver functionality.

Let's move ahead to March 29, the beginning of the fourth sprint where you're doing sprint planning that morning. In the previous sprint you delivered 24 points of functionality, a bit better than you had initially guessed. But, because you've been getting great feedback from your stakeholders in the first and second sprint you identified a lot of missing requirements (this is a common occurrence on software delivery projects regardless of paradigm), and in the third sprint you still identified six points of new functionality. At this point in time you now have 320 points of functionality due to the influx of new requirements.

Now the estimate of the delivery date becomes a bit more interesting because there are two options for your burndown rate: the gross velocity, which is the number of points of functionality delivered by the team each sprint, or the net velocity, which is the number of points that the product backlog shrank by the previous sprint. In this case, the gross velocity is 24 points and the net velocity is 18 points ($24 - 6$). Using the gross velocity the construction phase is now 14 sprints ($320/24$), and using net velocity the construction phase is 18 sprints ($320/18$). Including the four weeks required for release, the gross velocity indicates that the team will deliver in 32 weeks ($14 \times 2 + 4$) on November 5 for a total labor cost of \$760,000 ($(32 + 6) \text{ weeks} \times \$5,000 \times 4 \text{ people}$), and

the net velocity in 40 weeks ($18 \times 2 + 4$) on December 31st for a total labor cost of \$920,000 ($(40+6) \text{ weeks} \times \$5000 \times 4 \text{ people}$).

The most realistic thing to do is to take the conservative approach with net velocity, but either way it doesn't look like this team will make its date. Interestingly, we now have an easy strategy for giving a realistic ranged estimate for both the cost and schedule based on our actuals. We could have also given a ranged estimate at the very beginning of the project by estimating the initial velocity as a range.

At this point in time, you should inform senior management about the situation. For your IT governance strategy to be effective, project teams must be as open and honest about their current status as possible, even when the status is bad. You should discuss strategies for rectifying the problem, which could include letting the delivery date slip, cutting functionality, or increasing the size of the team. Adding people to the team runs the risk of running afoul of Brook's Law (http://en.wikipedia.org/wiki/Brooks%27s_law), which states that adding more people to a late project makes it later, although the team is still early in the lifecycle so this shouldn't be a problem. Sadly, senior management tells you to stick to the existing date with the existing team and still deliver all the required functionality. The project status goes red and team morale goes down.

Move forward to April 26, the beginning of the sixth sprint. There are now 285 points on the product backlog, with 22 points delivered the previous sprint and four points added to your product backlog. Now the expected delivery date is between 30 weeks ($285/22 \times 2 + 4$) and 36 weeks ($285/18 \times 2 + 4$), which is November 20 and December 31, respectively. The cost is now estimated to be between \$800,000 ($40 \times \$5000 \times 4$) and \$920,000 ($46 \times \$5000 \times 4$). Alternatively, to make a delivery date of August 27 (the last Friday in August) you have only 14 weeks of construction left, implying that you can do between 126 ($14/2 \times 18$) and 154 ($14/2 \times 22$) points of work — so you need to drop between 131 ($285 - 154$) and 159 points of functionality from the existing product backlog given your current gross and net velocities. Time for another discussion with senior management.

You can see where the scenario is going, the important thing is that it is relatively straightforward to update your estimates as you progress throughout an agile development project. I just wanted to leave you with a few more points.

- First, it is common to have a negative net velocity in the first few sprints due to an increase in requirements as your stakeholders see the working solution at first. In this case the top end of your estimated range would be infinite, a problem which will correct itself once your stakeholders have a better understanding of what they actually want.
- Second, a slightly more complicated approach, although more accurate, would be to use average rates for your velocity calcu-

lated over several sprints instead of just the previous sprint's velocity.

- Third, you can have intelligent conversations based on actual data, not on wishful thinking. Senior management might not like what they're hearing, but the numbers speak for themselves. If your project is in trouble, it's better to correct the problems as early as possible instead of letting them fester over time.
- Fourth, doing these calculations by hand is part of the hidden bureaucracy in agile software development that we don't often talk about. The good news is that with integrated and instrumented tooling it's possible to automate these calculations.

Hot Links

In "Questioning Earned Value Management (EVM) on IT Projects" (<http://www.drdoobbs.com/architect/207801786>), I examine the traditional strategy of EVM for monitoring your project actuals against the plan.

The *Dr. Dobb's* 2009 State of the IT Union Survey (<http://www.ambysoft.com/surveys/stateOfITUnion200907.html>) explored project management and governance issues.

"Lies, Great Lies, and Software Development Project Plans" (<http://www.drdoobbs.com/architect/218700176>) explores the results of the *Dr. Dobb's* July 2009 State of the IT Union Survey.

The Danger of Detailed Speculations summarizes the evidence around how detailed specifications can increase, not decrease, the risk on your IT projects.

At www.jazz.net you can see the project dashboard of the Jazz Project team, including the burn down chart that is automatically generated by the Jazz-based development tools used by the team to develop Jazz.

"Introduction to the Agile Scaling Model" (https://www.ibm.com/developerworks/mydeveloperworks/blogs/ambler/entry/detailed_speculations?lang=en) summarizes the ASM framework, which provides advice for scaling agile strategies to meet the unique needs of your project.

The "Surveys Exploring the Current State of Information Technology Practices" page (<http://www.ambysoft.com/surveys/>) links to the results of all the *Dr. Dobb's* surveys that I've run over the years.

My Agility@Scale blog discusses strategies for adopting and applying agile strategies in the complex environments.

— Scott Ambler is Chief Methodologist/Agile and SOA at IBM Rational.

[Return to Table of Contents](#)

The iPhone Developer's Cookbook

Reviewed by Mike Riley

**The iPhone Developer's Cookbook:
Building Applications with the iPhone 3.0 SDK, 2nd Edition**
Erica Sadun
Addison-Wesley Professional
\$44.99

With the impending release of the Apple iPad, renewed interest in the iPhone SDK has garnered accelerating exploration into the Objective-C/Cocoa framework combination. Learning the nuances of this platform can be challenging. Fortunately, author and iPhone hacker Erica Sadun has come to the rescue with the publication of her second edition of the iPhone Developer's Cookbook. Read on for my review.

Originally published in 2008, the *iPhone Developer's Cookbook* was immediately recognized as a treasure trove of tips and tricks for both aspiring and experienced iPhone developers. Indeed, the book was even considered for a Jolt award during that timeframe. Since then, Apple has rolled out its major 3.x release of the iPhone operating system adding GPS, compass, faster graphics, and more to the platform. To respond to these additions as well as address the shortcomings of the original, the second edition has more than doubled in its page count by adding new chapters on everything from the Objective-C 2.0 language to programming the latest iPhone capabilities like gestures and push notifications and learning and leveraging the Accessibility Plus, Core Data, GameKit, and StoreKit In-App purchasing services.

The book begins with the usual SDK introduction of obtaining, installing, and configuring the Xcode environment. A sample Hello World project helps familiarize readers with the IDE, iPhone Simulator, and the Objective-C language. It goes further than other iPhone introductory texts by detailing the procedures and considerations of purchasing an iPhone developer's program to deploy iPhone applications to actual hardware devices as well as to Apple's App Store.

After the new 50+ page crash course chapter on Objective-C, the numerous code sample recipes are presented in the remaining 18 chapters. These chapters cover such diverse topics as working with tab bars, fading and flipping views, using core animation for view transitions, e-mailing pictures, live camera overlays, han-

dling gestures including shake events, touches and multi-touches, dealing with text entries, alerts and table views, leveraging GameKit for data transfer and voice chat, over a dozen networking recipes for dealing with synchronous and asynchronous downloads and even building a simple web-based server. Other recipes include monitoring battery state, playing and recording audio and video, a 30+ page chapter on working with push notifications followed by a chapter on using the new Core Location and MapKit services for reading and interpreting the iPhone's built-in GPS. Finally, chapters on interacting with the iPhone's Address Book, Core Data and In-App Purchasing and Accessibility Plus services cap an amazing volume of valuable code snippets and explanations that will provide iPhone developers with the tips needed to take their applications to a higher, more polished and professional level. Erica's explanations and code deconstructions are clear and easy to follow, making the journey toward iPhone SDK mastery a knowledge-rich and memorable one.

The book's sample code can be downloaded from Erica's Github account, a practice that is gaining favor among tech book writers. The Github hosting concept works because of the community feedback aspects that help refine code examples and contributions from the readers toward future editions.

Overall, this second edition of *The iPhone Developer's Cookbook* represents an outstanding collection of immensely useful routines, tips, and tricks for iPhone application developers spanning the spectrum of experience. Reading the book from cover to cover will elevate both the breadth and depth of knowledge needed by iPhone and future iPad application creators everywhere.

[Return to Table of Contents](#)

Q&A: Teamwork

It's all about trust and techniques

by Jonathan Erickson

Andrew Stellman and Jennifer Greene are the editors of *Beautiful Teams: Inspiring and Cautionary Tales from Veteran Team Leaders* (<http://oreil.ly.com/catalog/9780596518028>) and principals of Stellman & Greene. They recently spoke with *Dr. Dobb's* editor-in-chief Jonathan Erickson.

Dr. Dobb's: What qualities define a “beautiful” team?

Stellman: There's no single magic formula to make a team work. One thing that can set the stage for an effective — even “beautiful” — team is that team members aren't afraid to fail. A team that's afraid of failure is a team that isn't going to take risks, and risk-taking is how innovation happens. Another quality is trust. If team members trust each other, then when one person goes out on a limb, everyone else won't have a knee-jerk reaction and pull him or her back. Finally, a great team has great leadership.

Dr. Dobb's: Is an ugly team simply “none of the above” or something else?

Stellman: Teams that might be considered flawed — people who don't get along, problems with the projects or their companies or the world in general, a complete lack of those great qualities that we just talked about — can still accomplish great things. This was a realization for us because it helped us see that the whole exploration of how teams work and what makes them tick is more complicated than we thought at first. There are lots of contradictions and messiness.

Dr. Dobb's: Can tools alone turn an ugly team into a beautiful one?

Stellman: A good team tool can help a good team be better. But if you've got a team that's deeply flawed, just adding a tool won't fix the problem.

At best, it will help you make mistakes faster.

Dr. Dobb's: Can teams thrive in non-agile environments?

Stellman: Absolutely. We've gotten so used to Agile practices like test-driven development, continuous integration, and constant communication between the team and users that we have trouble remembering that a lot of great software was built by great teams before these things had even been invented. Indeed, great software is still being built by teams that don't use agile techniques.

Dr. Dobb's: Why do we see more ugly team than beautiful teams?

Stellman: Because it's very easy for a team to get sunk by its obstacles, and those obstacles come in many forms: people who can't work together, bosses who sink your project, problems with buildings and facilities, bad product ideas that you can't shake. Sometimes they can break up a team permanently.

Every team faces obstacles. It's the nature of working on projects. If a team can get past its obstacles, it bonds them together. Most of the time, unremarkable teams muddle through the obstacles and deliver something. We wouldn't call that “ugly” as much as “plain” — not beautiful, but not too unattractive, either.

There are definitely truly ugly teams out there. All of us have been on at least one. Maybe it's the one with the boss who yelled at the junior team member until he started crying, or the project that you worked 70 hours a week on only to see it canceled at the last minute. The ugliest teams are the ones with the biggest obstacles that don't manage to get over them. The funny thing is that if the team can figure out a way to get past the obstacle, they have a shot at turning ugly into beautiful.

This is exactly why in our book that we wanted to talk to Peter Gluck, who manages software teams at NASA. We specifically asked him about tools and techniques you'd find on an agile project, like continuous integration. He pointed out that their integration process generally involved putting a rover or satellite in a clean room and loading software onto it. That's not exactly something that you can automate and run continuously.

That said, there's a lot in common with the teams and teamwork that Peter described at NASA and a good agile team. The biggest one is that everyone on the team takes quality very, very seriously. That's something that we really credit the agile movement with: making quality the developer's responsibility. In fact, we're really pleased that it's possible to ask a question like the one you asked about whether or not teams can thrive in non-agile environments. Agile practices like test-driven development, continuous integration, well-planned time-boxed releases with high enough quality that they can actually ship, user stories and other agile requirements techniques, and most importantly, constant communication between the team and the users (and, in the best case, bringing the users right onto the team) — we've gotten so used to these as part of our day-to-day work that we have trouble remembering that a lot of great software was built by great teams before these things had even been invented. Indeed, a lot of great software is still built by teams that don't use agile techniques. As Peter showed us, NASA has some very good examples in its past.

Dr. Dobb's: What's the best example of a software development team you've run across?

Stellman: To be honest, the best team either of us have been on was the team that was just the two of us. We were working on a great project that was interesting, challenging, and, in our opinion, important. We were building software for public health researchers who were studying the effects of herbicide exposure on Vietnam veterans. We had great rapport as a team and with our client, and we got a lot out of working on something that was going to actually help peoples' lives. We built a really high quality piece of software: It was used to do many thousands of calculations (some runs lasted for weeks at a time), and we never had to fix a single bug after the software was released.

[Return to Table of Contents](#)



The Sim Ship Standard

Meeting the growing expectation of simultaneous shipment with XML

by Su-Laine Yeo

In the global economy, “sim ship” — simultaneously shipping in multiple languages — is becoming the new normal. Companies are expected to deliver their messages to a global audience, in a wide variety of languages, at the same time. The message might be about a product, or the message might be the product itself, in the form of news, analysis or other content. The audience may be customers, partners, employees, industry watchers and others who, together, speak 20 or more languages. Regardless, sim ship is expected for even rapidly and frequently updated messages and content. And XML is the most efficient way for companies to meet that expectation.

Without XML, weeks can pass before a company is ready to sim ship. That's what happened at BusinessObjects. The company writes content in English and localizes it for up to 10 other languages, and its build times were as high as two days per deliverable per language. What took so long? Formatting the content for each language. Basically, content would come back from the translator, and somebody on the documentation team would spend a day or two making that FrameMaker, Word, or HTML file look good and read correctly in the given language.

Since BusinessObjects moved to XML, its build times have shrunk from days to minutes. XML saves time by separating content and formatting, and all of the formatting work is automated. That's a huge advantage in the world of sim ship. No matter how many revisions the content goes through, the formatting is automatically generated according to a style sheet. Nobody has to manually reformat for each revision, for each language and each deliverable. Now, content is formatted with the push of a button.

Of course, XML brings other advantages to companies that sim ship or want to do so.

- First, XML lets companies mark what parts of a document should be translated and, just as important, what parts should not be translated. Often, a document includes sections that are supposed to remain in the native language, regardless of the target language into which that document is being translated. The ability to mark such sections “do not translate” eliminates the errors that occur when content is accidentally translated.
- Second, XML enables automatic information sorting based on the target language. With XML, the computer — not the writer — sorts content that's alphabetized, such as a glossary. Without XML, a writer is going to sort content in one language, and then translators have to resort that content into each language. It's a huge manual effort that's now automated by XML.
- Third, XML supports content reuse to drive significant cost savings. Content reuse prevents content duplication, and more important for sim shipping, it prevents translation duplication. When a company pays by the word to translate content, minimizing content redundancy saves a lot of money. Before adopting XML, BusinessObjects faced huge translation costs. The marketing organization alone supported 12 languages in print and on the Web. Any given product description could be used in marketing materials, on the website, in tech support materials, as well as product materials. Whenever a product description changed, each one of those business units had to retranslate the description into 12 languages, effectively generating 40 to 50

projects for each change. Now, XML streamlines that process. The product description is translated into the target languages one time, and then the business units format the content for their deliverables.

- Fourth, XML supports incremental translation — the ability to translate only those portions of a document (or other content) that have changed since the last translation. Incremental translation is yet another way to drive down translation costs. Now, XML isn't the only technology that supports this feature. Companies also use translation memory systems to reduce the number of words that need to be re-translated. And while those systems do, indeed, reduce translation costs, adding XML to the mix helps reduce costs across the entire content lifecycle.

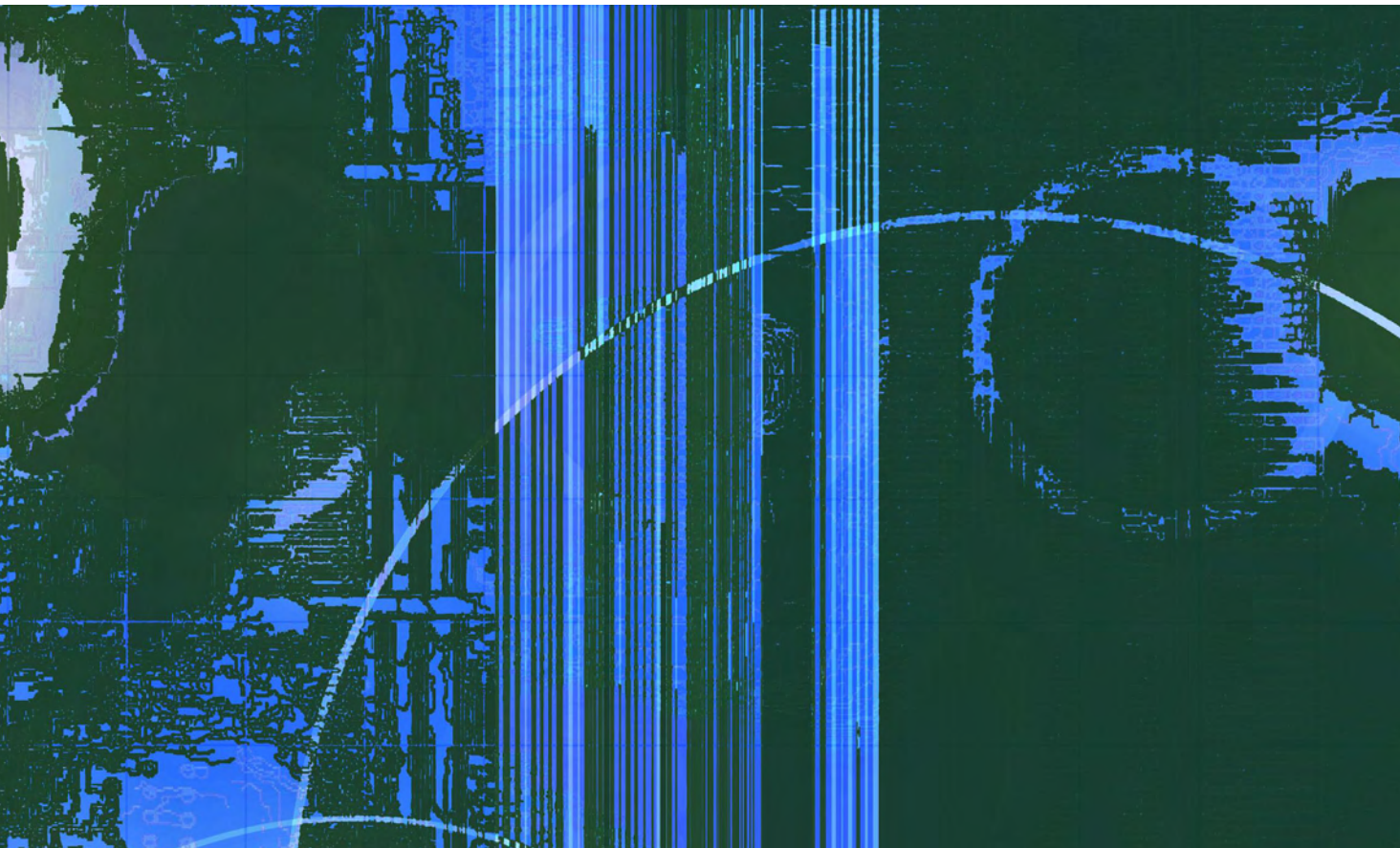
XML doesn't just reduce costs; it reduces the risk of errors that are magnified by multiple languages. For example, take two different departments in the same organization. They take the same English content and get it translated, but they use two different translation firms and end up with essentially different content. Same input, different outputs because even though they use the same target language, the translation firms use different phrases and word structures. And they generate different content

as a result. And each time somebody has to manually edit that content, the risk of error is magnified by the number of languages supported.

Sure, XML has been around for a while, but sim shipping is driving more organizations to embrace XML and deliver content more effectively in multiple languages. Whether a company is trying to sim ship products (and product documentation) to consumers or messaging documents to employees in multiple languages, manual and redundant processes are crippling. Nothing can be released until the content is published in all of the target languages, to all of the target formats. XML dramatically accelerates time to message, offering an alternative to days and weeks spent formatting, sorting and performing other tasks by hand. For companies committed to sim ship, XML is the new normal.

— *Su-Laine Yeo is a solutions consultant for XMetaL at JustSystems and an active member of the OASIS DITA Technical Committee. You can contact Su-Laine at su-laine.yeo@justsystems.com.*

[Return to Table of Contents](#)



Variations on a Meme

Social media maturing? You're kidding, right?

Web technologies like Facebook and Twitter go through stages as they mature. Okay, maybe "mature" is not the word I want. "As they age."

So isn't it reasonable to assume that Chatroulette (<http://chatroulette.com/>) will go through similar stages? You know about Chatroulette, right? It takes more time to explain it than for you to follow the link, but for those who like some telling along with their showing, it's a website that connects you via webcam with random strangers. Who then either disgust you or reject you. It's really popular. Although your experience may be different from mine. Anyway, I think that the Chatroulette phenomenon is bound to evolve, and I'm betting that the next stage will be Themed Chatroulette. Rather than seeing the same random people everybody else sees doing the same random things, you'll customize your random selection by keying in a theme or phrase. That's not far-fetched; even an iPod Shuffle can provide degrees of randomness. So let's see how this might play out.

As our example theme, I choose "What is cloud computing?" Since this is a text-only article, you'll have to imagine the video. And since we don't yet have Themed Chatroulette, there isn't anything to describe, so I'll have to make it up. What I'll do is, I'll piece it together from found video on the web. That ought to produce a fair test of my theory. Or whatever it is we're doing here. Anyway, here we go.

You click "New game."

You see some guy with a Giant Pac-Man hanging over his head: "Cloud computing is a bit like Liquid Paper."

You click "Next." This is the main attraction of Chatroulette, it seems to me, the ability to dismiss annoying people with the click of a button.

You see Larry Ellison waving his hands and ranting: "What is cloud computing? Take Microsoft Word, change 'internet' to 'cloud', give it back to these nitwits on Sand Hill Road...."

You click "Next." Chatroulette presents you with plenty of annoying people to feed your need to click them away.

You see some guy with a beard thrusting his face into the camera: "You've got Cloud A and you've got Cloud B and there's storms and rain. And then lightning goes between the clouds. Kshhh! And in that bolt of lightning you get one tera peta flops of information!"

You click "Next." Sometimes the people are scary.

You see Rafe Needleman wearing a namebadge so you know who he is: "Yeah, in an airplane. That's cloud computing."

You click "Next."

You see Larry Ellison waving an Evian bottle: "Our industry is so bizarre. You just change a term and they think you've invented technology."

You click "Next."

You see a random rock star: "You're tryin' to trick me. That's one of those catch questions."

You click "Next."

You see an IT guy looking mystified: "I have no idea. That's why I'm asking Wikipedia."

You ask yourself why you're doing this. You click "Next."

You see a gentleman of the redneck persuasion: "It's like a angelic kinda — thing. It's a experience. It's spirichal."

You click "Next" and say, amen, brother.

You see a folk singer: "LANs and WANs and endless cords...."

You click "Next."

You see Larry Ellison leaning into the camera: "...SalesForce became cloud computing for the same reason Chanel last year was fuschia, this year it's puce."

You click "Next."

You see Matt Mullenweg: "It should be like Care Bears. Perfect. Happy."

You know Matt Mullenweg is somebody, except you don't remember offhand just what he did to get marginally famous. You wait to see if he will elaborate. He doesn't. You click "Next."

You see Adolph Hitler surrounded by his senior officers: "You n00bs! You fell for a bunch of marketing and slick glossies from some tradeshow booth bimbo! You outsourced all our data to a Cloud Computing provider that hosts in Leningrad?"

That one seems staged. You click "Next."

You see Larry Ellison sniffing glue: "Yeah, 'Let's call that cloud.' Sure beats innovation."

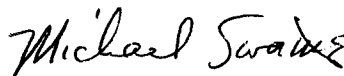
You click "Next."

You see Marc Benioff of SalesForce smirking: "...and if you can understand that then you'll know why cloud computing is what it is."

You've had enough. You disconnect.

Well, I think we've all learned something from that experiment.

(All of these snippets can be found, word-for-word, by looking for "cloud computing" on YouTube. Pretty nearly, anyway. The part about Larry Ellison sniffing glue is a total lie.)



— Michael Swaine is a senior contributing editor to Dr. Dobb's and editor of PragPub (<http://www.pragprog.com/magazines>).

Return to Table of Contents