United Business Media

Dr.Dobb's DIGEST

The Art and Business of Software Development April, 2010

DrDobbs.com: New AND Improved

By Jonathan Erickson,
Editor In Chief

If you had a feather we'd both be tickled. Why? Because I get to welcome you to DrDobbs.com…make that the "new and improved" DrDobbs.com. If you're a long-time reader of Dr. Dobb's, you probably think that the "new and improved" part is like the library books stacked up in the corner of my office — overdue. And you'd be right. Which is why I'm especially pleased to usher in the redesigned Dr. Dobb's website.

Photos of your's truly aside, the "improved" stuff is easy to spot: brighter colors, smoother layouts, and a design that makes finding actual articles possible. In other words, the kind of UI that enhances the user experience you'd expect from www.DrDobbs.com.

Improved UIs are one thing, but "new" features is where the fun begins — and we've loaded up DrDobbs.com with new features, including:

- A spotlight on the Dr. Dobb's Guru Bloggers, that merry band of experts (well, for the most part; there are a couple of grumpy ones in the mix) who stay on top of what's happening in the world of software development
- New search functionality that helps you find what you want faster and more accurately
- Improved community experience thanks to new commenting functionality allows deeper user interaction and engagement.
- Social network integration with deeper embedding of social media outlets such as Facebook and Twitter.
- Taxonomy refresh with streamlined "channel" navigation across key developer topics
- Blog site integration that merges DobbsCodeTalk.com into DrDobbs.com for a seamless user experience
- Single-sign-on membership that gives users simple access to not only DrDobbs.com, but to other leading IT sites such as InformationWeek.com, NetworkComputing.com, DarkReading.com, WallStreetandTech.com, and TechWeb.com

That's some, but not all, of the changes you'll start seeing as you begin poking around in the site. What hasn't changed, however, is Dr. Dobb's commitment to excellence in technical information. What? You don't believe me? Okay, that's fair. I'm admittedly biased when it comes to Dr. Dobb's, so will you believe your peers?

In a recent survey, Evans Data (http://www.evansdata.com), the leading marketing intelligence company for the software development space, asked developers to identify and rate the websites they visit on a regular basis. I'm happy to report that DrDobbs.com came out as #1 in a number of categories, including quality of technical articles, usefulness of technical articles, blogs, unbiased product coverage, and overall credibility.

We're honored that the software development community has recognized DrDobbs.com in this regard, and I'd like to personally say thanks for the kind words and support.

I also hope that you'll find DrDobbs.com is both new and improved. At the same time, I look forward to hearing your suggestions and recommendations for improving the website even more. Here's what you can do: Drop me a comment using our new and vastly improved "threaded commenting" feature. Not only can you comment to the authors (and editors) of the articles you're reading, but you can interact with your fellow readers. Think an article is great? Tell the author. Think a reader's comment is wrong-headed? Say so. Believe an editor is wacko? Be gentle now…

Remember — we can do the "new," but it's you who brings the "improved."

Return to Table of Contents

A Grand Unified Theory of AI

A new approach unites two prevailing but often opposed strains in the history of AI research

By Larry Hardesty

In the 1950s and '60s, artificial-intelligence researchers saw themselves as trying to uncover the rules of thought. But those rules turned out to be way more complicated than anyone had imagined. Since then, artificial-intelligence (AI) research has come to rely, instead, on probabilities — statistical patterns that computers can learn from large sets of training data.

The probabilistic approach has been responsible for most of the recent progress in artificial intelligence, such as voice recognition systems, or the system that recommends movies to Netflix subscribers. But Noah Goodman (http://www.mit.edu/~ndg/), an MIT research scientist whose department is Brain and Cognitive Sciences but whose lab is Computer Science and Artificial Intelligence, thinks that AI gave up too much when it gave up rules. By combining the old rule-based systems with insights from the new probabilistic systems, Goodman has found a way to model thought that could have broad implications for both AI and cognitive science.

Early AI researchers saw thinking as logical inference: If you know that birds can fly and are told that the waxwing is a bird, you can infer that waxwings can fly. One of AI's first projects was the development of a mathematical language — much like a computer language — in which researchers could encode assertions like "birds can fly" and "waxwings are birds." If the language was rigorous enough, computer algorithms would be able to comb through assertions written in it and calculate all the logically valid inferences. Once they'd developed such languages, AI researchers started using them to encode lots of commonsense assertions, which they stored in huge databases.

The problem with this approach is, roughly speaking, that not all birds can fly. And among birds that can't fly, there's a distinction between a robin in a cage and a robin with a broken wing, and another distinction between any kind of robin and a penguin. The mathematical languages that the early AI researchers developed were flexible enough to represent such conceptual distinctions, but writing down all the distinctions necessary for even the most rudimentary cognitive tasks proved much harder than anticipated.

Embracing Uncertainty

In probabilistic AI, by contrast, a computer is fed lots of examples of something — like pictures of birds — and is left to infer, on its own, what those examples have in common. This approach works fairly well with concrete concepts like "bird," but it has trouble with more abstract concepts — for example, flight, a capacity shared by birds, helicopters, kites, and superheroes. You could show a probabilistic system lots of pictures of things in flight, but even if it figured out what they all had in common, it would be very likely to misidentify clouds, or the sun, or the antennas on top of buildings as instances of flight. And even flight is a concrete concept compared to, say, "grammar," or "motherhood."

As a research tool, Goodman has developed a computer programming language called "Church" named after the great American logician Alonzo Church that, like the early AI languages, includes rules of inference. But those rules are probabilistic. Told that the cassowary is a bird, a program

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™

written in Church might conclude that cassowaries can probably fly. But if the program was then told that cassowaries can weigh almost 200 pounds, it might revise its initial probability estimate, concluding that, actually, cassowaries probably can't fly.

"With probabilistic reasoning, you get all that structure for free," Goodman says. A Church program that has never encountered a flightless bird might, initially, set the probability that any bird can fly at 99.99 percent. But as it learns more about cassowaries — and penguins, and caged and broken-winged robins — it revises its probabilities accordingly. Ultimately, the probabilities represent all the conceptual distinctions that early AI researchers would have had to code by hand. But the system learns those distinctions itself, over time — much the way humans learn new concepts and revise old ones.

"What's brilliant about this is that it allows you to build a cognitive model in a fantastically much more straightforward and transparent way than you could do before," says Nick Chater, a professor of cognitive and decision sciences at University College London. "You can imagine all the things that a human knows, and trying to list those would just be an endless task, and it might even be an infinite task. But the magic trick is saying, 'No, no, just tell me a few things,' and then the brain — or in this case the Church system, hopefully somewhat analogous to the way the mind does it — can churn out, using its probabilistic calculation, all the consequences and inferences. And also, when you give the system new information, it can figure out the consequences of that."

Modeling Minds

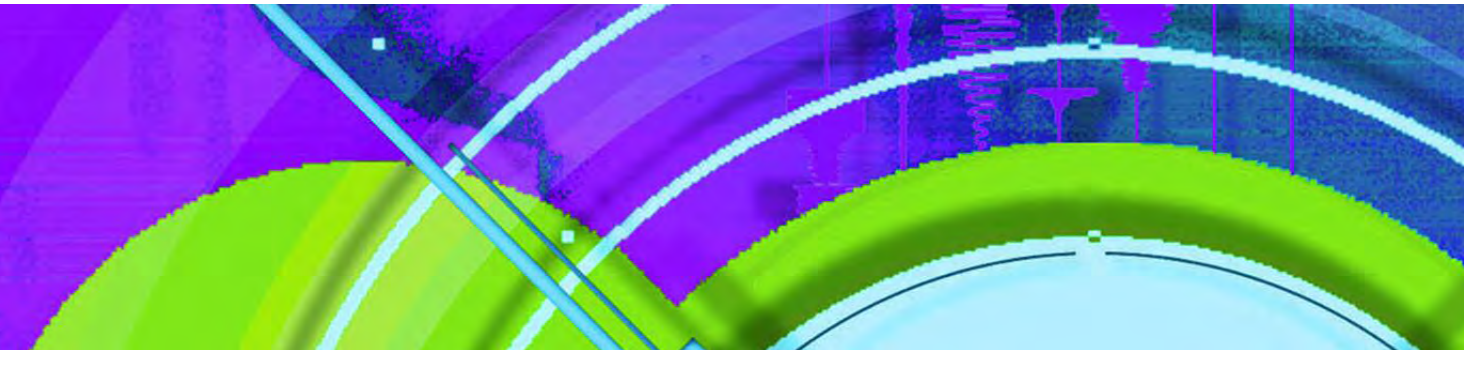
Programs that use probabilistic inference seem to be able to model a wider range of human cognitive capacities than traditional cognitive models can. At the 2008 conference of the Cognitive Science Society, for instance, Goodman and Charles Kemp, who was a PhD student in BCS at the time, presented work in which they'd given human subjects a list of seven or eight employees at a fictitious company and told them which employees sent e-mail to which others. Then they gave the subjects a short list of employees at another fictitious company. Without any additional data, the subjects were asked to create a chart depicting who sent e-mail to whom at the second company.

If the e-mail patterns in the sample case formed a chain — Alice sent mail to Bob who sent mail to Carol, all the way to, say, Henry — the human subjects were very likely to predict that the e-mail patterns in the test case would also form a chain. If the e-mail patterns in the sample case formed a loop — Alice sent mail to Bob who sent mail to Carol, and so on, but Henry sent mail to Alice — the subjects predicted a loop in the test case, too.

A program that used probabilistic inference, asked to perform the same task, behaved almost exactly like a human subject, inferring chains from chains and loops from loops. But conventional cognitive models predicted totally random e-mail patterns in the test case: They were unable to extract the higher-level concepts of loops and chains. With a range of collaborators in the Department of Brain and Cognitive Sciences, Goodman has conducted similar experiments in which subjects were asked to sort stylized drawings of bugs or trees into different categories, or to make inferences that required guessing what another person was thinking. In all these cases — several of which were also presented at the Cognitive Science Society's conference — Church programs did a significantly better job of modeling human thought than traditional artificial-intelligence algorithms did.

Chater cautions that, while Church programs perform well on such targeted tasks, they're currently too computationally intensive to serve as general-purpose mind simulators. "It's a serious issue if you're going to wheel it out to solve every problem under the sun," Chater says. "But it's just been built, and these things are always very poorly optimized when they've just been built." And Chater emphasizes that getting the system to work at all is an achievement in itself: "It's the kind of thing that somebody might produce as a theoretical suggestion, and you'd think, 'Wow, that's fantastically clever, but I'm sure you'll never make it run, really.' And the miracle is that it does run, and it works."

Return to Table of Contents



The iPhone Isn't Easy

How to get started building an app

By Tom Thompson

With more than 20 years of software development under his belt, Ray Floyd was ready to tackle the next great platform, Apple's iPhone. He'd developed for the Macintosh, although not recently. He was familiar with Objective-C, and he had developed software for other mobile platforms. With a software development kit in hand and up to 75 million potential customers, Floyd couldn't wait to roll out his first iPhone app.

Two months later, he was still waiting, and his enthusiasm had largely changed to exasperation. Why? Because writing native applications for the iPhone platform can be an intimidating process.

For one thing, as Floyd quickly discovered, there is a huge amount of information to digest before you can start coding. The iPhone/iPod Touch/iPad is a new platform in its own right. These are sophisticated general-purpose, handheld computers. And as a platform, it has its own operating systems, and a host of APIs (some new, some recycled) for programming. Not surprisingly, the documentation on operating systems and APIs spans hundreds of pages.

Even knowing the Mac isn't a free pass. The iPhone operating system is a variant of Mac OS X, but slimmed down to function on a mobile device. As such, a lot of familiar OS X APIs, such as those that manage a keyboard and mouse, are absent. This means that even experienced Mac developers, which Floyd didn't consider himself to be, may not have the features and facilities they're used to having.

The APIs are based on Cocoa, an object-oriented application framework used to write Mac OS X apps. Like the operating system, the Cocoa frameworks have been stripped to the bare essentials for the iPhone OS, and lightweight touch interface APIs added. This minimalist version of Cocoa, termed "Cocoa Touch," provides another hurdle.

To use the APIs, you need to learn Objective-C. Objective-C is an object-oriented superset of C, and while it has many valuable features, its idiosyncratic syntax means it'll take a lot more than a cursory study to make sense of code. The language's syntax is quite different from C++, and until you have a grasp of the syntax, example code will appear almost unreadable.

Like Floyd, I was initially lulled into thinking iPhone development was probably simple for experienced developers. How else could you explain the many thousands of iPhone apps available on Apple's App Store? In fact, many of these apps — and games written in C and C++, in particular — have been ported to the iPhone, where an Objective-C-based wrapper interfaces to the Cocoa Touch frameworks, while the game engine remains written in C and C++. In other words, since the Apple Objective-C compiler accepts C and C++ code, you don't have to start entirely from scratch.

But as I began researching iPhone apps, I grimly realized even my background in writing INITs and FKEYs for the classic Mac OS and J2ME programming for various mobile phones was of little use. The iPhone platform, its OS and APIs, and the peculiarities of Objective-C were radically different. I decided to break the task up to reduce the steepness of the learning curve, hoping to find a magic bullet along the way that would streamline development.

First, what should my first app do? I wanted a specific project so I could focus on a subset of APIs, and not become overwhelmed by the big picture. I settled on porting the SpaceActivity app I had written for the Android platform some time ago (see "The Android Mobile Phone Platform"; http://www.drdobbs.com/mobility/210300551).

The app displayed a spaceship you steered with handset buttons. That iPhone APIs would be utterly different from Android's was a given. I'd have to learn how to draw the ship on the iPhone's screen,

then figure out how to implement controls for it. I hoped that SpaceActivity's "core code" — the physics routine that calculated the ship's velocity in response to rocket thrusts, plotted its position, and kept the image on-screen — could be reused. This code was debugged and tested and had the virtue of using a minimum of API calls. It also had ready-to-use images of the spaceship with a transparent background, so I didn't have to design any new graphics. Finally, it would also be an interesting test of porting Android code. The complete source code for SpaceActivity is available at http://i.cmpnet.com/ddj/images/article/2010/code/iphone_space.zip.

Variations on C

The iPhone runs a stripped-down version of Mac OS X and uses a subset of the Cocoa application frameworks, along with some new ones, to support its touch interface. These revised frameworks are known as Cocoa Touch and comprise the iPhone platform's APIs. (For further information on the iPhone platform, see "Smartphone Operating Systems: A Developer's Perspective"; http://www.drdobbs.com/mobility/216300179.) OS X programmers will have an advantage here in that they are familiar with the programming language and frameworks. There are some differences between the frameworks, however. For example, the primary Cocoa framework for OS X is AppKit, while on the iPhone the primary Cocoa Touch framework is UIKit. UIKit, as its name implies, consists of lightweight UI classes tailored for the resource constraints of a hand-held device. Also, the keyboard and mouse interfaces are absent. I'll point out more differences between the two platforms as I go.

Again, you write an iPhone with Objective-C. Objective-C consists of extensions to the C that implement many OOP techniques, such as object encapsulation, inheritance, and polymorphism. Because the language is a superset of C, Apple's Objective-C compiler accepts C and C++ source, which makes it ideal for reusing existing code written in those languages. In fact, some games ported to the iPhone consist mostly of C code, and Objective-C code is used only to interface with the Cocoa Touch frameworks. For example, id Software's Wolfenstein 3D takes this route, since its core logic was written in C. John Carmack, the game's creator, was able to write Objective-C wrapper code that communicated with the iPhone's APIs, while the complex pieces of the game, such as the game's ray-casting logic and its rendering engine, were left intact. Carmack has made the source code of Wolfenstein 3D available for download from the id Software website (http://www.idsoftware.com/wolfenstein-3d-classic-platinum/).

Objective-C has its own language idiosyncrasies for defining objects and implementing other OOP-based design patterns. How you invoke methods differs, for example. In Objective-C, you don't "call" a method, rather you send a "message" to the object that owns the method, like so:

```
[shipView upDateShip:kNoThrust];
```

The content within the square brackets contains the gist of the message. To wit, this message is directed to the *shipView* object, and invokes its *upDateShip* method, while supplying it with an argument that consists of the constant value *kNoThrust*. Another important difference is that for certain complex UIKit classes you don't extend their behavior by subclassing them and writing override methods. Instead, you edit the supplied "delegate" class that implements the desired behavior. For example, you do not extend the *UIApplication* class that implements the core application functions. You modify and add overrides to a *UIApplicationDelegate* class. Messages unknown to the *UIApplication* object are routed to your *UIApplicationDelegate* object, where your custom methods can respond to them. You can get detailed information on Objective-C from the Objective-C 2.0 Programming Language manual, and more information on UIKit and message passing from the iPhone Application Programming Guide (http://developer.apple.com/programs/iphone/). Both of these documents are available on the Apple iPhone developer site. Given all of the changes implemented by Objective-C and UIKit, the structure of an iPhone app is an interesting mixture of the old and new. When the iPhone OS launches an app, it calls the standard C function, *main()* (Listing 1). However, all that *main()* does is set up a memory pool for your app and optionally parses any *argc* and *argv* arguments.

```
LISTING ONE
#import <UIKit/UIKit.h>

int main(int argc, char *argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];
    int retVal = UIApplicationMain(argc, argv, nil, nil);
    [pool release];
    return retVal;
}
```

However, here the resemblance to C ends: *main()* immediately calls the *UIApplicationMain()* function, which generates an instance of the *UIApplication* class. It also creates your delegate object, which is your modified *UIApplicationDelegate* class. *UIApplication* then starts receiving events from the iPhone OS and sends them to your delegate. For example, when your delegate receives an *applicationDidFinishLaunching* message, this signals that the app's runtime environment is established and running. You respond to this message by executing the delegate's *applicationDidFinishLaunching* method, which contains custom initialization code, and creates any required objects. (Listing 2).

```
LISTING TWO
#import "SpaceAppDelegate.h"
#import "SpaceViewController.h"

@implementation SpaceAppDelegate
@synthesize window;
@synthesize spaceViewController;

- (void)applicationDidFinishLaunching:(UIApplication *)application
{
          application.statusBarHidden = YES;

// Make instance of our controller
          SpaceViewController *aViewController =
[[SpaceViewController alloc]
                    initWithNibName:@"SpaceView" bundle:[NSBundle
mainBundle]];
          self.spaceViewController = aViewController;
          [aViewController release];
// Restore state if necessary
```

```
            [spaceViewController restoreState];
// Add view managed by controller to main window
            [window addSubview:spaceViewController.view];
    [window makeKeyAndVisible];
}
// Application will be terminated soon, save state data
- (void)applicationWillTerminate:(UIApplication *)application {
            [spaceViewController setMode:PAUSE];
            [spaceViewController saveState];
}
// Release all objects
- (void)dealloc {
            [spaceViewController release];
    [window release];
    [super dealloc];
}

@end
```

Another difference from C is that the iPhone APIs don't provide a method that allows you to terminate an application. Instead, your app receives an *applicationWillTerminate* message, which signals that the iPhone OS is preparing to shut the app down. In response to this event, your delegate executes an optional *applicationWillTerminate* method that saves the app's data context (if required) and tears down any storage that it set up in *applicationDidFinishLaunching*. This routine should execute quickly, as the iPhone OS usually terminates the app within a few seconds.

Now that I had a grasp of Objective-C and an idea how the iPhone OS interacted with an app, my next step was to figure out how to draw something onto the device's screen. That meant delving further into the frameworks.

A Window...With Views

An iPhone app displays UI elements and content on the device's screen, and the user responds by tapping the screen or making gestures across it. A singleton window object and multiple views are the visible components that manage the iPhone's UI and any content display. The window object encompasses the entire screen and lacks a border and a title bar with controls. The window and views are implemented by instances of the *UIWindow* and various *UIView* classes, respectively.

The *UIWindow* class is actually a subclass of *UIView*. However, the window instance serves as the root container for all of the other views, and it is always the size of the screen. It serves as a backdrop on which you add any number of views and subviews. The *UIApplication* object sends events to the window object, which in turn delivers them to the appropriate view objects.

An iPhone app typically generates an instance of the window object immediately, using information stored in a NextStep Interface Builder (nib) file. Nib files describe the window's dimensions and other information necessary to make an instance of the window. I'll have more to say about nib files later.

Alternatively, you can generate the window instance from code that you write in the delegate. Once the window is set up, you then add the views that make up the app's content and controls as subviews to this window. The *UIView* class has methods that allow you to add, remove, or change the order of how views are presented in the window. After you add the app's views, often you are done with the window and deal only with events delivered to the views.

You can see some of how this setup is accomplished in Listing 2.

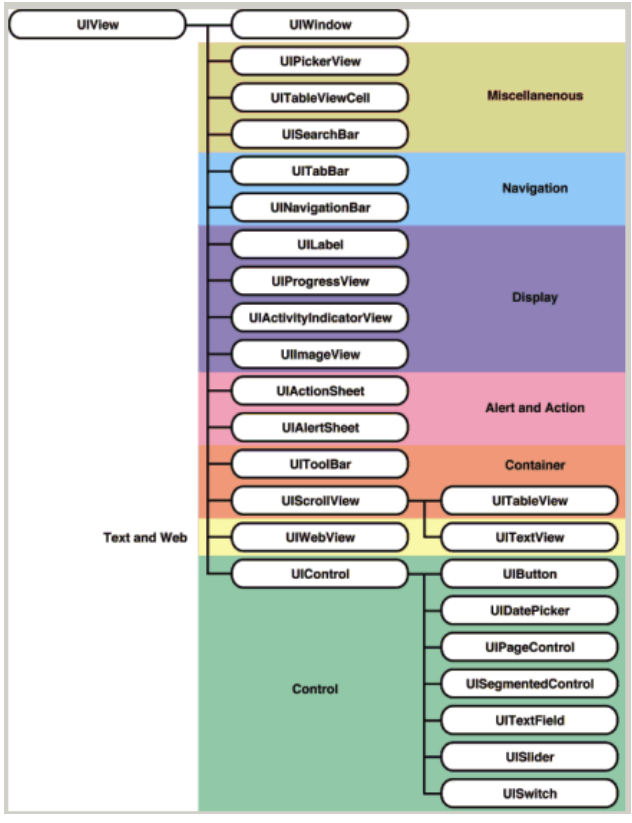


Figure 1

The generation of the window is done for you automatically if you use a nib file, as is the case here. All that you need to do is provide a pointer, termed window in the listing, which references the window instance. Apple's XCode development tools automatically generate the window nib file, MainWindow.xib, along with a *UIApplicationDelegate* header file and method file for you.

Views, like the window instance, can be generated either from a nib file or programmatically. The choice of which option to use depends upon how dynamic you intend the view's UI elements to be. For example, a help display with information presented in a static layout is best handled with a nib file. If you have controls that change in response to the user's interaction, you may want to use code to generate a portion of the views and then manage their position and appearance. Likewise, a view with the spaceship needs to move about the screen, so it should be generated by code, and not by a nib.

Besides the *UIWindow* class, *UIView* has a number of child classes that provide useful capabilities (Figure 1). A control group of views implements visual elements such as buttons, sliders, progress indicators, switches, and text labels. Another group serves as containers for dynamic content such as pages or tables. A navigation group provides tab bars and navigation bars that facilitate moving swiftly from one screen of content to another. Other subclasses support scrolling of content, and a toolbar for grouping items.

Views also display content, which might be text, vector graphics,

rendered HTML material, or images. For games, you can use cross-platform OpenGL ES (http://www.drdobbs.com/cpp/187203532) APIs to draw and render 3D content, but the OpenGL drawing region must be presented within a view. My program's goals were modest: I only wanted to paint pixels on the screen and move some of them about. The basic *UIView* class would be suitable for this purpose.

The origin of a view's coordinate system starts at the upper left corner and the axes extend to the right and downwards; see Figure 2.

Interestingly, for OS X, the situation is reversed: The origin is at the lower left corner and the axes extend to the right and upwards. In fact, the vector graphics APIs, Quartz (http://lists.apple.com/archives/QuickTime-API/2007/Jul/msg00129.html), still use this orientation in the iPhone, but UIKit automatically corrects the orientation for you before rendering the graphics calls. Coordinates are specified in floating-point numbers for device-resolution independence. A frame describes the view's rectangular area, and its midpoint is defined by an instance of point (a structure with *x* and *y* values) termed center. The center point information is handy to have when you move or rotate a view — particularly when the view might contain a spaceship image.

Views can be nested in one another, to form complex visual elements, as Figure 2 shows. In addition, views have characteristics known as properties, and the center point is one. Another useful property is alpha, which manages the view's alpha channel. This property can be modified to apply transparency effects to the view's contents. There is also a visibility property, hidden, that can be used to hide or reveal views.

Rotating an image is handled through the view's transform property. This property contains a transformation matrix that can be applied to the view. For me, a transformation that did a simple rotation would be sufficient. However, more complex transformations can be applied to a view's contents, and UIKit provides a number of predefined matrices that can be used to generate special effects with the view's content, or when switching to another view.

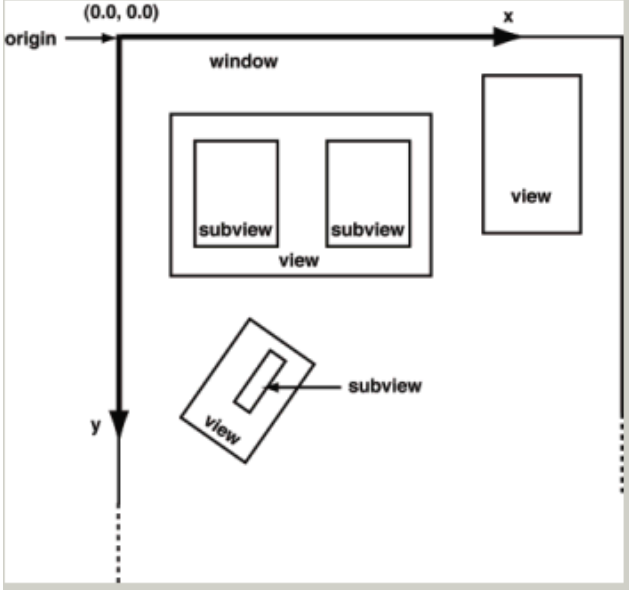


Figure 2

At this point I understood the UIKit framework adequately to know what classes I needed to build the basic core of an iPhone app. This core (an instance of *UIApplicationDelegate*) would start by making the required window object, to which I would add a view for a background, and another view that contained the spaceship image. A transformation matrix would rotate the spaceship view. I was a bit fuzzy on the details as how to control the spaceship, but I figured that another view could display a virtual joystick or control pad. That view would then react to touches on it. So far, so good.

It's All About Controllers

As I brought the details of implementing UI controls into sharp focus, I discovered that while I'd be still have the app add the views to a window, there was an intermediary involved: a *UIViewController* object. This object, known as a view controller, supports the OOP-based model-view-controller (MVC) scheme used by iPhone apps. In this program design, a model object manages the app's data and implements its custom logic, while the view displays the model's data. The controller acts as an intermediary that manages user interactions detected by the view, and has the model respond to them, either by editing or updating its data content. In addition, the controller then has the view refresh its content with the model's revised data. MVC keeps the app's unique data and behaviors within the model object and separate from the view assigned to display its data. Can you say encapsulation?

My planned app had only one screen where the spaceship would appear, plus a help screen. This arrangement could be neatly managed by a single instance of the generic *UIViewController* class, which would implement the physics model and jump to the help screen and back. While it's possible to implement all of the app's logic within a view object or in other ways, I found it better to follow the MVC design pattern that the Cocoa Touch frameworks support.

Unlike the *UIApplication* class that requires you edit a delegate object, the *UIView* and *UIViewController* classes can be subclassed so that you can extend their behavior with custom code. The custom code in the view controller would receive touch events from the view representing a virtual control pad, and convert them to commands that would affect the physics of the spaceship. The controller would then update the position and appearance of the subview displaying the spaceship. To recap the program design, both the control pad and spaceship would be subviews within the background view. This view in turn would be attached to the mother-of-all views, the window object. The program's design was coming together.

Apple's XCode development tools, as mentioned previously, provide ready-to-use iPhone application templates to help get you started. I launched XCode, and in its New Project dialog, chose the Cocoa Touch frameworks, and then chose an app template that was window-based. This I named *SpaceAppDelegate*. While there were templates that generated a view-based app template, I wanted to

add the view controller and view classes as I went.

Next, I added a view controller class. This was just a matter of selecting New File in XCode, followed by choosing the *UIViewController* class from the dialog. I named my view controller *SpaceViewController*. XCode generates the basic header file and method file for the class. The latter has stub routines for some of the class methods, allowing you the opportunity to add custom code.

The first thing the view controller would do is generate a view that covered the app's window and display an image of a star field as a background. Since the background was a static image, I decided that storing the view and its image in a nib file would be a good idea. It was, but getting a properly made nib file that didn't crash the app (the phone itself wasn't affected) was a major problem until I sorted things out.

Interface Building

To make nib files for your app's visual interface, you use the appropriately named Interface Builder program, often just called "IB." It provides a number of preconfigured objects such as *UIImageViews* (that display images), scrolling views, buttons, sliders, and other visual controls. You drag and drop these objects from IB's library window onto a mockup of the view to assemble the app's UI. IB thus allows you to design the visual elements of the app's UI without worrying about its underlying code. This fits nicely into the MVC design pattern, as IB keeps most of the UI elements uncoupled from the app code.

However, IB does much more than assemble an app's UI. The visual elements that you place into a view's layout become full-blown objects when they are loaded. (Apple characterizes the information stored in the nib file as "freeze-dried objects".) Put another way, the XML generated by IB that describes the object and its properties can be reconstituted at run time into actual UIKit objects. For example, that button you place in a view becomes a *UIButton* object that can detect a touch event on it. Furthermore, it responds by highlighting itself — no action on the part of the programmer is required. More important, the button object can send one of many pre-defined messages to the application. In short, when you use IB to put together the app's UI, you are not only constructing its visual interface, but also its underlying event and messaging apparatus that communicates with the program.

IB lets you route the messages generated by these UI elements to specific destinations, which in my case are methods in *SpaceViewController*. This is accomplished by clicking and dragging a link representing a message generated by the UI object to the name of a method in the receiving object. For example, for the view representing a button, you Control-click on it, and a menu of possible messages that can be generated in response to the action appear. You next click on the desired message link and drag it to the File's Owner icon, which represents the view controller receiving the messages; see Figure 3. A pop-up menu of methods appears, and when you select one, it appears in the Connections

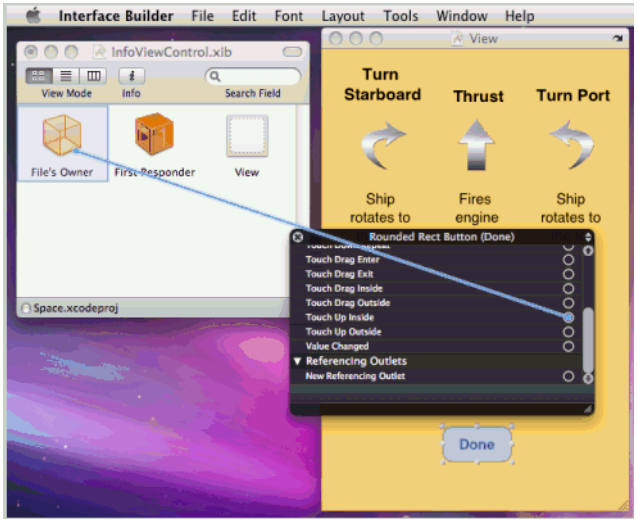


Figure 3

browser window. The visual metaphor used to match UI messages to methods is a powerful one, and makes coupling the UI's actions to specific program responses easy to do. Having said that, IB's assignments aren't foolproof or bulletproof. Mistakes can be made, and I made a good one at the outset.

I started my foray into nib files by launching IB, and in the New File dialog selected a view, and named it *SpaceView*. I made it the size of the iPhone screen (320 x 480 pixels). From the library window, I dragged a *UIImageView* onto the view, and made it the same size. From the object inspector, I assigned *UIImageView*'s content to be an image of the Horsehead Nebula that I had trimmed to the screen dimensions and saved in a file named Backdrop.png. With the view thus set up, the existing stub code in the view controller should be sufficient to load the nib and display the nebula image in the iPhone simulator. Displaying the image would make for a good initial test.

No such luck. My trial app died a horrible death, usually with an uncaught exception.

It took me a while to track down the culprit. In IB, if you botch specifying the object that a view sends its messages to, bad things happen. That's because Objective-C's message-forwarding and delegation mechanism builds the links between objects and methods at run time. Therefore, if you fail to specify say, a view controller as the message recipient, the result is that the iPhone OS loads a view with broken links. When the view attempts to pass messages to the view controller, they go spinning into oblivion, carrying your app with them.

The gotcha is in how IB sets the destination object for the view's messages. IB uses a placeholder object, called the File's Owner, to represent the destination object. You've met this object already, and can see it in the IB document window for SpaceView in Figure 3. Put another way, the File's Owner information designates the object that "owns" the interface, which means that it is the message recipient. Stated another way, the owner is the instance of the class that loads the nib. For my program, that would be

IB Tricks and Traps

Interface Builder (IB) is valuable for visually constructing an iPhone's UI and choosing the messages that it generates. However, like any software tool, mistakes in construction can result in the app crashing. The UI objects, their attributes, and message assignments are buried in either in .xib file (a text file consisting of dense XML that's generated for development purposes) or binary format (a .nib file that's generated for the release version of the app). These objects are loaded where you can't study or trace them, which makes trouble-shooting difficult. Generally, if the debugger informs you that it is terminating the application due to an uncaught exception, the problem is rooted in a view not loading properly, or due to a broken link in the nib file.

However, there are a number of ways you can safeguard your efforts, which I have fashioned into rules. These rules have been derived from my own experience.

Rule 1: When you make a view in IB, the first thing to do before all else is configure the File's Owner object. That is, you specify the object that is the recipient of the view's messages. Click on the File's Owner icon in IB's document window, and choose the Identity Inspector. In the Class field, enter the class's name (typically a view controller). Type-ahead in this field is supported, but be careful: If you have a lot of classes with similar names, a wrong keystroke could select the wrong one. Save the nib file with an appropriate name.

Rule 2: For view objects, be sure to assign an outlet. This is the inverse of the purpose described in rule 1: You're assigning a variable in the File's Owner that points to the view. Control-click on the File's Owner icon in IB's document window, then drag a link to the view icon in this window. You should get a pop-up menu specifying a view instance. Choose it, and save the file.

Rule 3: Build your nib files from scratch. While you can learn valuable interface techniques by studying the nib files of example programs, reusing them is not a good idea. The reason is that as you change things to suit your app, you break the pre-existing links in the example program's nib. Unless you are very thorough, finding all of the broken links is difficult and tedious. You're better off spending the effort making your own nib files from the ground up. You'll learn more in the process as well.

Rule 4: If you change the name of a class in your source code, be sure to update the changes in IB as well as any source files. In particular, update the File's Owner object. See rule 1.

Rule 5: When loading a nib file in code, be sure that the nib file name you present the initWithNibName method matches the actual file name, or is the correct one. Where this burns you is when you copy and paste some code, and forget to modify the nib file name. While I find it useful to have the nib file describe its class (such as SpaceView for a view object), there's no requirement that the nib file name has to match anything.

Rule 6: Have XCode, along with your project, up and running as you build the nib file, as suggested by Apple. These two programs intercommunicate changes, which is handy as you modify or add methods that will be targets of messages sent by the UI. Sometimes one of these programs can miss a beat and the nib file falls out of sync with the app code. This can be avoided by performing a clean build once and a while.

Rule 7: When IB gives you a hint, take it. Occasionally as you re-arrange the UI's elements, you might notice a warning icon at the bottom of IB's document window. IB's telling you something isn't right. Click on the icon to get a nearly plain-English explanation of the warning, which should help you correct the problem.

Apple has an excellent tutorial on building a nib file from scratch (see "Your First iPhone Application") that covers Rules 1 and 2. However, it doesn't stress sufficiently that failing to observe these procedures is going to cause you grief.

SpaceViewController. Granting ownership isn't accomplished by pointing and clicking in IB as was the case with the other assignments. This is done by typing the class name into IB's Identity Inspector. I had overlooked doing this. Once I entered the class name and rebuilt the project, the background image of the nebula appeared in the iPhone simulator. I was one step closer to my goal. For more information on IB's pitfalls and how to have problem free interface construction work, see the sidebar, "IB Tricks and Traps".

Views Within Views

Unlike the static scene in SpaceView where I used a nib file, I decided that the view displaying the spaceship should be generated by code. The reason is that it is a dynamic object, moving about in response to data from the physics engine. The view's image would also change in appearance, depending on whether the spaceship engine was firing. I created a simple view, *ShipView*, to manage the rocket's appearance (Listing 3). Two images, rocket_firing .png and rocket.png, would show the ship with its engine either firing or not. The method *upDateShip* changes the view's appearance by loading one or the other of these images. Besides drawing itself, *ShipView* doesn't do anything else: *SpaceViewController* is responsible for rotating the *ShipView*, and plotting its position. Once these updates are done, *SpaceViewController* directs *ShipView* to redraw itself, so that appears in the new position, orientation, and with the proper image.

```
LISTING THREE
#import "ShipView.h"

@implementation ShipView

@synthesize shipImage;
@synthesize shipThrustingImage;
@synthesize shipNotThrustingImage;

- (id)init {
          // Retrieve the images for the view and determine the
size of one
          shipNotThrustingImage = [UIImage
imageNamed:@"rocket.png"];
          shipThrustingImage = [UIImage
imageNamed:@"rocket_firing.png"];

          CGRect shipFrame = CGRectMake(0, 0,
shipNotThrustingImage.size.width,
shipNotThrustingImage.size.height);
          // Set self's frame to encompass the image
          if (self = [self initWithFrame:shipFrame]) {
                    self.opaque = NO;
                    shipImage = shipNotThrustingImage;
          }
          return self;
}
```

```
- (id)initWithFrame:(CGRect)frame {
    if (self = [super initWithFrame:frame]) {
        // Initialization code
    }
    return self;
}

- (void)upDateShip:(int)imageSelection {

          if (imageSelection == kNoThrust) {
                    shipImage = shipNotThrustingImage;
          }
          if (imageSelection == kThrust) {
                    shipImage = shipThrustingImage;
          }
}

- (void)drawRect:(CGRect)rect {

          // Drawing code
          [shipImage drawAtPoint:(CGPointMake(0.0, 0.0))];
}

- (void)dealloc {
          [shipImage release];
          [shipThrustingImage release];
          [shipNotThrustingImage release];
    [super dealloc];
}

@end
```

Creating the virtual control pad was a bit tougher. One of the best features about the iPhone is that its touch screen allows you to design the UI to suit the needs of your app. One of the worst features of the iPhone is that you have to design the UI. You don't want come up with an unwieldy interface that could frustrate a user. However, the first order of business was to figure out to respond to the touch events received by the control pad's view. The intuitive interface could come later. Although the control pad's position was fixed, I designed the *ControlPadView* class, like *ShipView*'s, to be generated programmatically. This allowed its appearance to change in response to a touch, and to potentially support a user's preference to change its position. *ControlPadView*'s code was much simpler than *ShipView*'s, as it only had to load one image. For a control pad image, I started with three buttons: a left one to turn the ship starboard, one in the middle to fire the rocket, and the third to command a turn to port. A view responds only to touch events that fall within its frame, and these events correspond to several types of messages. There's *touchesBegan*, which is sent when you place your finger on the screen, and *touchesEnded*, which is sent when you lift your finger from it. A *touchesMoved* message is sent when the iPhone OS detects finger motion across the screen. If you touch the screen and release it quickly, the iPhone OS interprets the two closely spaced events as a tap and advances a counter. This counter can be read to determine how many times the screen was tapped. With this set of events, I could implement the control pad and other actions, as in Listing 4. For a *touchesBegan* event, I first determined if it was on the *ControlPadView* or a view with an information button. If the touch occurred in *ControlPadView*, I next obtained the touch's coordinates within the view to determine what button had been touched. I had designed the buttons to fill the entire view vertically, so all the code had to do was compare the horizontal displacement with specific constants to determine which button was "pushed." For the two steering buttons, a *shipTurning* variable is set to signal a turn and its direction. A touch on the thrust button sets a variable named thrust. When *ShipView* redraws itself, it would display the image of the ship firing its engine. A touch on the information button loads the help view.

LISTING FOUR

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event
{

          UITouch *touch = [touches anyObject];
          if ([touch view] == controlPad) {

                    location = [touch locationInView:controlPad];

                    int disp;
                    disp = location.x;
                    if (disp <= kPortPad) {
                              shipTurning = kPortTurn;
                    } else if (disp > kPortPad && disp <=
kFireEnginePad) {
                              thrust = kThrust;
                    } else if (disp >= kStarboardPad) {
                              shipTurning = kStarBoardTurn;
                    }
                    controlPad.alpha = 0.75;
                    return;
          }
          if ([touch tapCount] == 2) {
                    [self doReset];
                    return;
          }
}

- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event
{

          thrust = kNoThrust;
          shipTurning = 0.0;

          [shipView upDateShip:kNoThrust];
          controlPad.alpha = 0.5;
          infoIcon.alpha = 0.5;
}
```

To provide visual feedback when the control pad was touched, I cranked up *ControlPadView*'s alpha channel property up so that the buttons would be less translucent and appear to light up. Finally, just in case things went horribly wrong with the program, I had *touchesBegan* watch the tap counter for two taps anywhere on the screen, which would reset the physics engine and the app's operating mode.

When the user lifts their figure, the *touchesEnded* method cleans up by clearing the *shipTurning* and thrust variables, and having *ShipView* update its image. I also reset the alpha property of the *ControlPadView* back its default value, which made the buttons partially translucent again. The ability to manipulate the properties of views, particularly their alpha channel and visibility, allows you to craft a slick interface with little effort and code.

Now it was time to put things together. The *UIApplicationDelegate* loads, and it in turn loads the main window nib and *SpaceViewController*. *SpaceViewController* in turn loads *SpaceView*'s nib and receives a message, *viewDidLoad*. This message signals *SpaceViewController* that *SpaceView* loaded without trouble. I used the *viewDidLoad* method to add the views of the spaceship, the control pad, and info button into *SpaceView*. The order in which the views are drawn is important, because you can layer views over one another. By drawing the ship first, it would be under the control pad and info button, but this wasn't a problem because their alpha property made them translucent. The *UIView* class also provides methods for changing the order of views on-the-

fly if your design requires it. The last thing *viewDidLoad* does is make an instance of *NSTimer*. It implements a periodic timer that calls the game loop method every 0.075 seconds.

The Loop That Isn't and Modes

The master method that drives the app and periodically updates its interfaces is — appropriately enough — called *gameLoop*; see Listing 5. Note that the method is a loop in name only: It calls housekeeping methods that update the physics model and the screen, and exits. The *gameLoop* method keeps executing because of the timer.

```
LISTING FIVE
-(void) gameLoop {
          if (programMode == RUNNING) {
                    [self updatePhysics];

                    [controlPad setNeedsDisplay];
                    [infoIcon setNeedsDisplay];
                    CGAffineTransform shipRotate =
CGAffineTransformMakeRotation(shipHeading / 180.0 * M_PI);
                    [shipView setTransform:shipRotate];
                    [shipView upDateShip:thrust];
                    [shipView setNeedsDisplay];

                    [self.view setNeedsDisplay];
                    return;
          }
          if (programMode == PAUSE) {
                    return;
          }

} // end gameLoop
```

The logic in *gameLoop* is starkly simple. The code calls *doPhysics* to update the physics model. *doPhysics* calculates the spaceship's latest position, velocity, and heading, based on the changes in the variables for rotation and thrust received from the UI. Next, *gameLoop* has the control pad and info button views redraw themselves. This is done so that the controls can appear with the proper translucent value, depending upon whether the user is touching them or not. Updating the spaceship view is slightly more complicated, but not much. The *ShipView*'s transform property is updated with the ship's latest heading, and then the transformation matrix in this property is applied to the spaceship view, which rotates it. *ShipView*'s *upDateShip* method is called to load the corresponding image into the view, which depends upon whether the thrust variable is set. Finally, the code has *ShipView* redraw itself. The code of *doPhysics* I dropped in straight from the Android program. I modified its API calls that obtain the absolute time and trigonometric values from Android's to those for Coca Touch (Listing 6). The one hiccup I had was that I couldn't just jam the values for the ship's *x* and *y* coordinates into *ShipView*'s center property. This property is a structure of type point and Objective-C syntax doesn't allow you to modify individual elements in a structure if it is an *l*-value. The easiest solution was to place the coordinates into a point with the *CGPointMake()* function and write this point into the center property.

```
LISTING SIX
-(void) updatePhysics {
   CFTimeInterval now = CFAbsoluteTimeGetCurrent();

// Calculate and respond to situation where ship tries
// to go off screen.
   CGRect rect = [[UIScreen mainScreen] applicationFrame];

   if (shipX <= kEdgeCondition){
      shipX = kEdgeCondition;
      mDX = -mDX;
   } else if (shipX>>= (rect.size.width - kEdgeCondition)){
      shipX = (rect.size.width - kEdgeCondition);
      mDX = -mDX;
   } // end else
   // Handle top and bottom
   if (shipY <= 10){
      shipY = 10;
      mDY = -mDY;
   } else if (shipY >= (rect.size.height - kEdgeCondition)){
      shipY = (rect.size.height - kEdgeCondition);
      mDY = -mDY;
   } // end else

   double elapsed = (now - lastTime) * 10.0;

   // Ship is rotating — update its heading
   if (shipTurning != 0.0) {
      shipHeading += shipTurning * (SLEW_SEC * elapsed);

      // Bring value back into the range 0..360
      if (shipHeading < 0.0) shipHeading += 360.0;
      else if (shipHeading >= 360.0) shipHeading -= 360.0;

   } // end if mRotating != 0

   // Base accelerations
   double ddx = 0.0;
   double ddy = 0.0;

   if (thrust) {
      // Taking 0 as up, 90 as to the right
      // cos(deg) is ddy component, sin(deg) is ddx component
      double elapsedFiring = elapsed;

      // Have this much acceleration from the engine
      double accel = (FIRE_ACCEL_SEC * elapsedFiring) / 10.0;
      double radians = shipHeading / 180.0 * M_PI;
      ddx = sinf(radians) * accel;
      ddy = -(cosf(radians) * accel);
   } // end if

   double dxOld = mDX;
   double dyOld = mDY;

   // figure speeds for the end of the period
   mDX += ddx;
   mDY += ddy;

   // figure position based on average speed during the period
   shipX += elapsed * (mDX + dxOld)/2;
   shipY += elapsed * (mDY + dyOld)/2;

   CGPoint shipLocation = CGPointMake(shipX, shipY);
   shipView.center = shipLocation;

   lastTime = now;
}
```

Finally, because the absolute time values returned by the iPhone API were different from those returned by Android, I had to tinker with some constants to get the spaceship's motions working smoothly. Overall, the changes to the physics code were minor and it worked almost "as is" in the app, due to the fact that Java's arithmetic operations are nearly identical to C's. In fact, porting the physics algorithm was probably the easiest part of the project, because I was working with proven code. Porting more sophisticated code from Android or other mobile platforms would require extra work, depending upon how much the algorithms rely on the other platform's APIs.

Touching the information button loads the *InfoViewController* class, which is a controller view that manages a view displaying help information. There's some nifty code that performs an animation that rotates the help screen over *SpaceView* as it is added as a subview. I had originally considered making the help screen just

another view, but I added a view controller, *InfoViewController*, since I might add other interactive elements to it in the future. Such elements might be the ability to let you choose where the control pad appears.

Once I got the basic help screen working, I realized I had a problem: the spaceship would keep drifting merrily about while the help screen concealed it. I therefore made a *setmode* method that modifies a variable, *programMode*. If *programMode* was set to the constant value PAUSED, then the *gameLoop* immediately exits, rather the calling *doPhysics* and updating the views. With this change, when you switch to the help screen, *SpaceView*'s actions stop until you dismiss it. The little bit of code present in *InfoViewController* performs the rotating animation back the *SpaceView* screen, and has the app resume its execution.

Saving State

Being able to halt the app's physics calculations also served another valuable purpose. Suppose a phone call interrupts the app? In this case, the calculations and updates must be stopped while the user decides whether to take the call. If she answers the call, the iPhone OS warns the app that it is going to be shut down. The app receives an *applicationWillTerminate* message, and in response it must preserve its critical state variables. If this is done properly, when the app is relaunched it restores its state and can resume where it left off.

There are a number of ways to store an app's state. Cocoa Touch offers archiving features that allow you to save data and objects. The most comprehensive archiving mechanism is *NSCoding*, which lets you save the complete object graph of a group of objects. *NSCoding* preserves objects into nib files, and when they are later reloaded, the complete state of the archived objects is restored. You got a glimpse of this capability when working with views in IB. The UIKit class documents specify upfront whether each class implements *NSCoding*. The problem is that you must write *NSCoding* methods to preserve and restore your custom-made class. That seemed like a lot of work, particularly when my app can be easily reconstituted by preserving just a few variables.

Cocoa Touch also offers a serialization mechanism that lets you store certain data objects. The serialized data is stored as a property list, which could be part of the application's property list (the Info.plist file) or stored as user preferences, using *NSUserDefaults*. I chose *NSUserDefaults* because it was simple to set up and met my needs.

With *NSUserDefaults*, you stash the data into a shared data object, using a key (a string) to uniquely identify it. You also use key names to both identify and access the variables you store within the object. The methods I wrote to store and retrieve the variables are shown in Listing 7.

```
LISTING SEVEN
- (void)saveState {
   NSUserDefaults *savedData = [NSUserDefaults
standardUserDefaults];
   [savedData setDouble:shipHeading forKey:@"shipHeading"];
   [savedData setDouble:shipX forKey:@"shipX"];
   [savedData setDouble:shipY forKey:@"shipY"];
   [savedData setDouble:mDX forKey:@"mDX"];
   [savedData setDouble:mDY forKey:@"mDY"];
   [savedData synchronize];
}

- (void)restoreState {
            NSUserDefaults *savedData = [NSUserDefaults
standardUserDefaults];
          int programStateSaved = [savedData
integerForKey:@"programStateSaved"]; // Get saved state flag

   if (programStateSaved) {
      shipHeading = [savedData doubleForKey:@"shipHeading"]; //
Get heading
      shipX = [savedData doubleForKey:@"shipX"]; // Get X coord
      shipY = [savedData doubleForKey:@"shipY"]; // Get Y coord
      mDX = [savedData doubleForKey:@"mDX"]; // Get X velocity
      mDY = [savedData doubleForKey:@"mDY"]; // Get Y velocity
   } else {
                  // First run
      [savedData setInteger:1 forKey:@"programStateSaved"]; //
Create save flag
      shipX = self.view.center.x;
      shipY = self.view.center.y;
      mDX = 0.0;
      mDY = 0.0;
      shipHeading = 0.0;
   }
}
```

Saving data in the user preferences object is straightforward. You retrieve a pointer to access the object, and save the critical velocity, position, and bearing information tagged with descriptive key strings. This code is invoked when the *UIApplicationDelegate* object receives the *applicationWillTerminate* message.

Retrieving the stored data is easy, and its method is called from the delegate's *applicationDidFinishLaunching* method. The code first checks for a flag, *programStateSaved*, to see if the app's stored data exists. If not, it makes such a flag and stores some default values into the object. If the flag exists, then the code uses the same keys used to store the data to retrieve it. The code places these values into the appropriate variables. Now all that's left to do is have *gameLoop* resume execution, and the physics engine picks up where it left off.

Finally, because of the iPhone's OS X underpinnings, you have a third option in which to save the app's context. You can use UNIX BSD-style file system calls to store and read the app's state, rather than Cocoa Touch. A good example of this technique appears in Wolfenstein 3D, where its *SaveTheGame* method uses a passel of *fwrite()* calls to save the game's critical variables, and its *LoadTheGame* method uses *fread()* calls to restore the game's state.

Final Polish

Getting a working app isn't enough, however. There are other design details that need to be dealt with. The first was checking the usability of the interface that I'd cobbled together. For that, I sought the feedback of a beta-tester, which was my son, John. While waiting in line for *Avatar*, I handed him the iPhone and had him fire up the app and fool with it. His feedback was swift and sobering: The control buttons were too small, and they needed to better indicate their function. I had drawn some crude arrows for the control pad and obviously they were lacking.

While I have some graphic arts skills, the app's UI would fare far better if I found some professionally designed arrow images on the Web. Ideally, they should be in PNG format (the iPhone's preferred image format, although it can handle JPEG, TIFF, GIF,

Windows BMP, and XWindows bitmap files), and with a transparent background. I wasn't asking for much, was I?

After a little scouting, I found the Icons web page, which is part of the MySiteMyWay site (http://icons.mysitemyway.com/). It provides a vast array of attractive icons in different colors and designs. There are icons for all sorts of images and concepts, and they are 512 x 512 pixels in size with transparent backgrounds. You can use any pixel-editing program to scale them down for the iPhone, although the original size might be handy for iPad app interfaces. Best of all, they're free!

I downloaded the glossy silver icons archive, and used Lemke Software's Graphic Converter X to edit the images. I selected several arrow images to represent the various control pad actions, scaled them down, and then merged them together into one image. This became the control pad displayed on the screen (Figure 4).

To complete the app, two other images had to be made. The first was a 320 x 480-pixel PNG image that's stored in a file named Default.png. When the iPhone app launches, this image is loaded automatically as a background scene until your app initializes and adds its views to the window. Otherwise, the user becomes alarmed when they stare at a blank screen for more than several seconds. A properly designed default image shows the user that the application is starting up and doing things. You can see this behavior in Apple's weather app, in that it first displays the blank days of the week graphic while it fetches weather information. For simple apps, this default image could be either a splash screen, or present a message that states the app is loading. More complex apps with longer load times should display a progress bar.

I had used Blender3D, an open source 3D modeling program, to generate the spacecraft image for Android. I had the original model file handy, so I used Blender3D render a larger image for use as the splash screen. Next, I had Blender3D render the ship into 57 x 57 pixel image file named Icon.png. This image serves as the application icon. I added these two files to the XCode Space project and after rebuilding the app, the icon and splash screen appeared with no further coding effort on my part. My tester, John, was much happier with the UI.

Lessons Learned

Looking back, there are a few things I would have done differently. For example, for my info button I use a graphic, but late in the design I learned that there are two button types, *UIButtonTypeInfoLight* and *UIButtonTypeInfoDark*, that display the info icon for you. In addition, I'd group the initialization of the control pad and info button views into one method, and probably do the same for updating them. Contrary to how things are done on a desktop application, where one updates the UI as things happen, on a mobile platform it's better to group the graphics redraws together into bursts that allow the processor to fall idle and converse battery life. The reason for the separate code for these views is that my app grew incrementally as I learned things. Finally, the simple update code just hammers out redraw messages in response to touches, particularly for the ship's engine firing. It might be worth investigating if a little logic could implement smarter screen updates that occur only when something really changes.

Figure 4

Looking ahead, improvements that I'd like to make to Space are its ability to handle a change in screen orientation. Also, I'd like to make the screen controls configurable. I'm left-handed and put the control pad in the iPhone's left corner, but I realize my handedness is in the minority and other users might want the control pad on the right. I'd also like to add guard logic so that the app could react if critical image files were missing, or if the saved variables were corrupt.

However, as such my iPhone app shows that I got off to a good start on my journey. I now understand Cocoa Touch's View classes sufficiently to draw to the screen, display controls, handle finger taps, and update the display in response to them. In addition, the app saves and restores its state when shut down, or when I take a call. The program accomplishes all of this using little code, which speaks well for the capabilities of Cocoa Touch and the iPhone mobile platform.

— Tom Thompson is an experienced developer for multiple mobile platforms. He can be contacted at tom_thompson@lycos.com.

Return to Table of Contents

What's New in Visual Studio 2010 & .NET 4 for Web Developers

Simplifying the development cycle, while still building quality applications

By Scott Guthrie and Laurence Moroney

After what we think is an amazing engineering effort, Microsoft is ready to release Visual Studio 2010 (http://www.microsoft.com/visualstudio/en-us/) with the .NET Framework version 4. The guiding principle behind Visual Studio 2010 and .NET Framework 4 is pretty straightforward: We wanted to simplify the entire development cycle, while at the same time making it easier for developers to unleash their creativity, and build quality applications.

Since the days of Visual Basic 1.0, we have aimed to deliver tools that let developers harness new technologies with languages and environments they already know. Visual Studio 2010 continues that tradition with integrated support for Windows 7, Microsoft SharePoint 2010, Windows Azure, and more. Built on .NET Framework 4, Visual Studio helps developers extend and transfer existing skills to new technologies, providing greater flexibility and power.

We've worked hard to make sure that Visual Studio 2010 and .NET Framework 4 have something for every developer, and this is particularly the case for those focused on Web development. This is where we'll spend the bulk of the time in this article, as we take a tour of some of the new features for Web developers. We'll start with a look at IDE improvements, then we'll go into some of the API improvements for ASP.NET Web Forms and ASP.NET MVC.

However, this is only a portion of the new functionality we've introduced in these releases, so check out www.microsoft.com/visualstudio for more information or to download the trial version.

Multiple Monitor Support

One of the first productivity improvements is multi-monitor support in Visual Studio 2010. In previous versions of Visual Studio, we joked that you could support multiple monitors, but you would have to do that by launching multiple instances of Visual Studio and running one on each monitor! With Visual Studio 2010, we've added real multi-monitor support, so you can "tear off" any window, such as a code window, and dock it in a separate monitor. So, for example, you can be running your application by stepping through the code in one window on one monitor, and looking at trace information on another. It's one of those features that once you start using it; you'll wonder how you lived without it!

Improved Intellisense

Intellisense is one of those great features that make using an IDE a real pleasure. With Visual Studio 2010 we've provided a subtle but wonderful upgrade to the Intellisense feature. In the past, when you started typing, the Intellisense window would look for methods, properties, or events that are available to you and that begin with the characters that you just typed. With the updated Intellisense in Visual Studio 2010, the characters will match any part of the available classes. So, for example, if you type *cart*, then Intellisense will find all available APIs where the characters are present, and provide some basic documentation on that API; see Figure 1.

Note also that the search is sensitive enough to understand Pascal casing, so, for example, in Figure 1, we could have typed *db.ATSC*, and found the *AddToShoppingCarts* method.

The Intellisense improvements aren't limited to .NET — you can also use it with JavaScript, including JavaScript libraries like jQuery. Figure 2 is example where you can see not only does the Intellisense pick up the available APIs for the *myItem Var,* but it also picks up the jQuery documentation as a tooltip!

This documentation is also fully customizable. In the case of jQuery, you'll see a file called "jQuery&ltversion>-vsdoc.js" in your solution. Find the desired method (in this case *load*) and you'll see the <summary></summary> tags containing this documentation. Follow this syntax in your own JavaScript libraries, and you'll be able to provide intellisense and documentation tips in the same way.

```
load: function( url, params, callback ) {
///       <summary>
///                 Loads HTML from a remote file and injects it into the DOM.  By default
///            performs a GET request, but if parameters are included
///                 then a POST will be performed.
///       </summary>
///       <param name="url" type="String">The URL of the HTML page to load.</param>
///       <param name="data" optional="true" type="Map">Key/value pairs that will be sent to the server.</param>
///       <param name="callback" optional="true" type="Function">The function called when the AJAX request is complete.
///          It should map function(responseText, textStatus, XMLHttpRequest) such that this maps the injected DOM element.</param>
///       <returns type="jQuery" />

  if ( typeof url !== "string" )
    return this._load( url );
```

Code Navigation

Visual Studio 2010 provides a new way of finding code within your application. If you hold down the Control key, and hit the "comma" key, the new Navigate To window will open. This lets you type in some characters, and Visual Studio shows you all the code that contains those characters, be it a variable name, a method, an event handler, or even an external file such as a graphic. Imagine a scenario where you know that the code you're looking for has something to do with a "*cart*", but you can't remember exactly what it is called. Figure 3 is an example of where the user has typed the value cart, and Visual Studio has found all the code containing that word. This should make it a lot easier for you to find what you need. The Pascal casing previously mentioned for Intellisense also works in the Navigate To window.

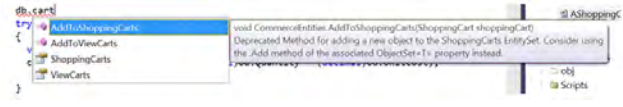


Figure 1: An example of improved Intellisense.

Figure 2: Using Intellisense with JavaScript.

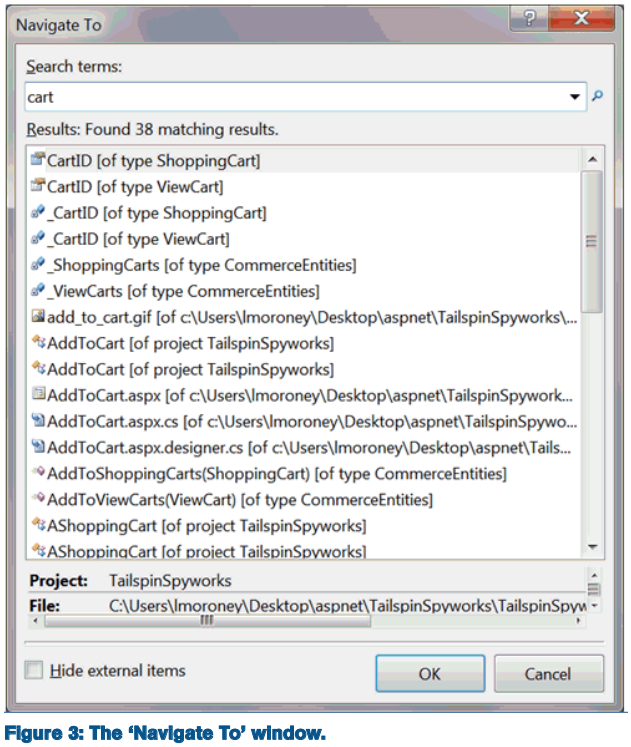


Figure 3: The 'Navigate To' window.

Visualization Tools

There are enough visualization tools in Visual Studio 2010 to fill several full articles in their own right! For example, Web developers are able to generate a dependency graph of their site, which shows not only how pages are dependent on each other, but also how they are dependent on controls and how they use any code-behind logic. You can dive directly into the pages, master templates, controls definitions, or code directly from the graph. Figure 4 is an example of a Web Application where there are a number of *Services*, a number of *Controllers*, and a number of *Models*. You can see that the *Models* entity has been opened up and we can see the classes within the application. You could also drill further down into the code behind these classes.

Profiling and Debugging

Another upgrade that we'd like to highlight is where we've improved profiling and debugging of your applications. So, for example, you can profile your application to see where the bottlenecks are, and where you need to tweak it to improve performance. So, for example in Figure 5 you can see where the Visual Studio has measured each

of the executing elements within the application and its effect on CPU usage. This helps make your applications and the servers that they run on much more effective.

Intellitrace

In addition to finding performance optimization opportunities for your application, you also need the best possible way to find problems in your application so that you can debug and fix them. An upgrade to Visual Studio is the addition of Intellitrace which can keep track of the complete flow of your application, and which can be externalized and sent to a developer to follow through to reproduce the situation. It gives you a historical view of the execution of the application which you can use to quickly pinpoint bugs. You can see an example of using Intellitrace in Figure 6.

Deployment

Of course, you still need to deploy your application for folks to be able to use it. With Visual Studio 2010, there are a couple of things to make your experience in doing this a little easier. First is the ability to have multiple Web.config files within your web application. So, for example, you can have different Web.config files for an application on a Debug server, a Staging server and your release server. In addition to this, you can right-click anywhere in the solution explorer window and find a new "Publish" option. This gives you a wizard that packages up all your application assets, and deploys them according to the config profile that you want to use. You can see the dialog in Figure 7. This is preconfigured for the debug profile, so it may, for example, be using a different database from your release version.

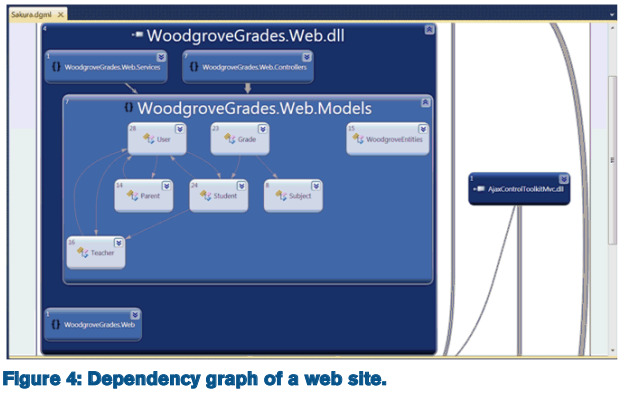


Figure 4: Dependency graph of a web site.

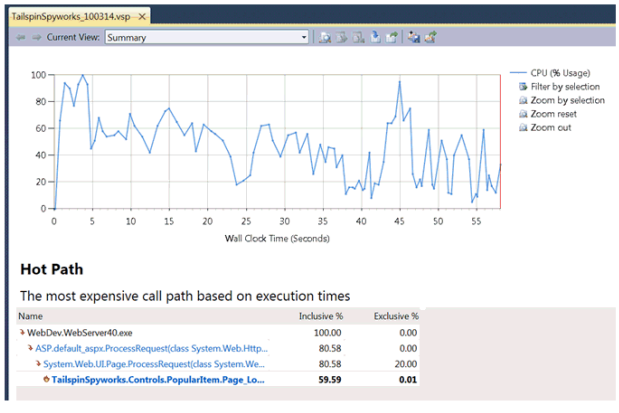


Figure 5: Using the Visual Studio profiler.

ASP.NET Web Forms

Now let's switch gears a little and take a look at ASP.NET Web Forms. We get a lot of email from people worried that with the release of ASP.NET MVC, ASP.NET Web Forms may be going away. We want to let you know that this is certainly not the case. Microsoft has invested a lot in Web forms for the .NET 4 release, with the aim of making your productive experience in building Web applications using ASP.NET and ASP.NET Web forms easier

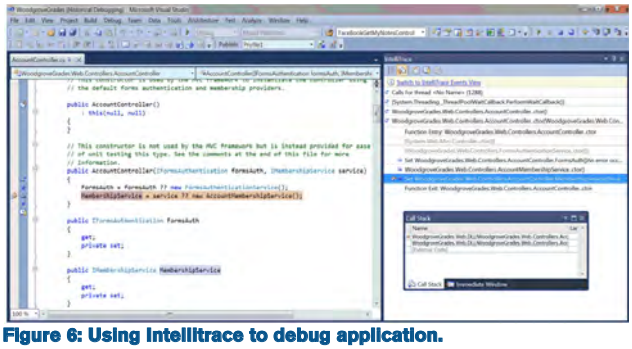


Figure 6: Using Intellitrace to debug application.

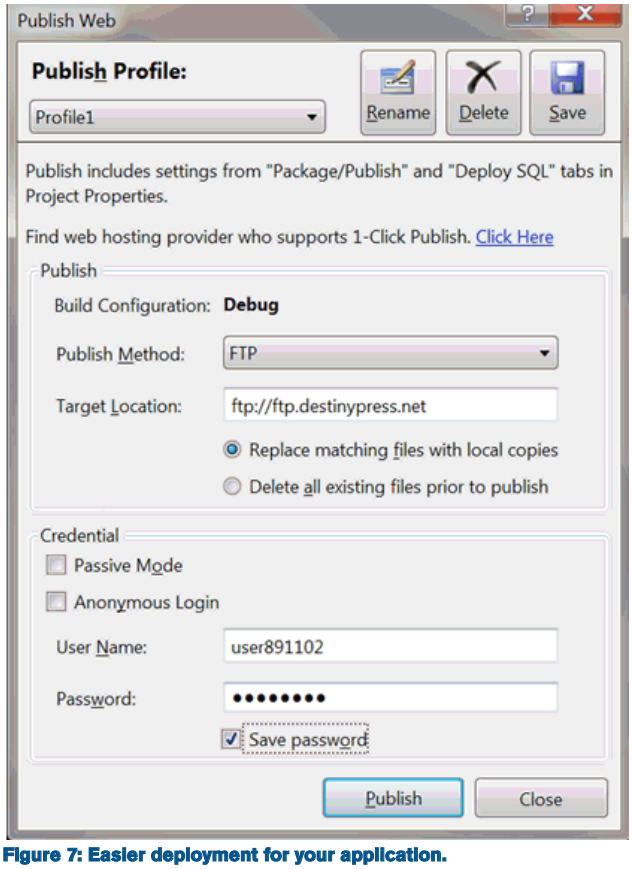


Figure 7: Easier deployment for your application.

and more powerful than ever.

Client ID flexibility. One major piece of feedback that we've heard from developers involves controlling the Client ID of controls. With the explosion in popularity of JavaScript-based client-side APIs such as the JQuery JavaScript library, many people have found that the controls generated by an ASP.NET page on the server side have obscure or hard-to-recognize IDs because the framework would modify the client-side IDs to uniquely identify each control. It could give you the ID that you had previously defined, or could leave you with something long and obscure like *ctl00_MasterPageBody_ctl07_TextBox1*, and if you needed to write some client-side script against this, the experience wasn't optimal. Now, with ASP.NET 4, all controls have a *ClientIDMode* property which can be used to control this behavior. Note also that the overall *ClientID* mode can be set for the application in Web.config.

So, for example, consider this simple ASP.NET Web Form that uses a master page:

```
<asp:TextBox ID="MyTextBox1" runat="server"></asp:TextBox>
```

that will produce the following HTML:

```
<input name="ctl00$ContentPlaceHolder1$MyTextBox1"
    type="text" id="ctl00_ContentPlaceHolder1_MyTextBox1" />
```

The *ClientIDMode* property has a number of settings that you can use. In this case, as you have a single control, setting it to *Static* lest you generate the ID on the client exactly as it is specified on the server, so the following ASP.NET markup:

```
<asp:TextBox runat="server"
   ID="MyTextBox1" ClientIDMode="Static" ></asp:TextBox>
```

will produce the following HTML:

```
<input name="ctl00$ContentPlaceHolder1$MyTextBox1"
   type="text" id="MyTextBox1" />
```

Another option for the *ClientID* mode property are *Predictable* which is typically used in a data binding scenario where *n* controls are created at runtime where *n* is determined by how many records you are binding to. So, for example, consider a *GridView* control like this:

```
<asp:GridView ID="Friends" runat="server"
  AutoGenerateColumns="false" ClientIDMode="Predictable" >
    <Columns>
        <asp:TemplateField HeaderText="ID">
            <ItemTemplate>
                <asp:Label ID="FriendID" runat="server"
  Text='<%# Eval("ID") %>' />
            </ItemTemplate>
        </asp:TemplateField>
        <asp:TemplateField HeaderText="Name">
            <ItemTemplate>
                <asp:Label ID="FriendName" runat="server"
  Text='<%# Eval("Name") %>' />
            </ItemTemplate>
        </asp:TemplateField>
    </Columns>
</asp:GridView>
```

The HTML that is generated will have a table whose ID is the name of the *Grid*, and elements whose names are derived from the name of the grid, the name of the field and an incremental value, so you can predict that a field will be called *GridName_FieldName_X*.

```
<table id="Friends">
..
<td><span id="Friends_FriendID_0">1</span></td>
<td><span id="Friends_FriendName_0">Scott Guthrie</span></td>
..
<td><span id="Friends_FriendID_1">2</span></td>
<td><span id="Friends_FriendName_1">Laurence Moroney</span></td>
..
</table>
```

Other options are *Inherit* where the name will inherit from the parent container, and *AutoID* where the legacy behavior of pre-NET 4 is maintained for backwards compatibility.

Cleaner HTML

Going beyond ClientID control, ASP.NET 4 also gives you the ability to emit very clean HTML. This makes it a lot easier for you to do CSS styling on your HTML, make AJAX programming simpler and be more bandwidth efficient. ASP.NET 4 gives you clean markup through removing many of the extraneous inline styles, and by removing outer table rendering on template controls. As a simple example of this, consider an Image control, which in ASP.NET is defined like this:

```
<asp:Image ID="Image1" runat="server" />
```

When you'd execute the application, this would generate markup like this, where a style with border width of 0 has been added for you whether you want it or not:

```
<img id="MainContent_Image1" src="" style="border-width:0px;" />
```

With ASP.NET 4, this behavior is turned off by default, so you'll get cleaner HTML, and its xHTML conformance is set to be strict. The earlier *Image* will look like this:

```
<img id=" Image1" src="" />
```

There are many other instances of using controls where styles like this would be added, or the output would be wrapped in a table, so that inline styles could be applied. In .NET 4, this is removed, but, for backwards compatibility, you can turn it back on. You do this using the *<pages>* node in your web.config. If you load an older project into Visual Studio 2010, this will be set to the "old" mode so as not to break your markup. Here's the Web.config setting for the "old" way. Change the value to "4.0" for cleaner HTML.

```
<pages controlRenderingCompatibilityVersion="3.5"/>
```

ViewState

Another major piece of feedback that we received is around *ViewState* in ASP.NET Web Forms. *ViewState* gives you the ability to store the state of your application encoded as a string in a hidden form on your page. In earlier versions of ASP.NET, *ViewState* was turned on for all controls by default, whether you needed it or not, resulting in very large values in these hidden forms, affecting

performance and manageability of your page. With .NET 4 we've turned this around, by turning it off by default, and allowing you to enable it for the controls that need it, providing you with the ability to optimize the size of your *ViewState*. You control this using the *ViewStateMode* property on either the page itself, or on any of the controls on the page.

For example, take a look at this *Grid* Control:

```
<asp:GridView ID="GridView1" runat="server" AllowPaging="True"
    AllowSorting="True" AutoGenerateColumns="False"
      BackColor="White"
    BorderColor="#3366CC" BorderStyle="None" BorderWidth="1px"
      CellPadding="4"
    DataKeyNames="id" DataSourceID="SqlDataSource1"
      ViewStateMode="Enabled">
```

When this is run on a page, the ViewState of the page will look something like this:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="Ri70Kt932u5kPOG5Zst6eq6fqmbiORZJcKv2ci2dhihsohA7cc2NUN8e0N
hVdW7+XHOaL5qIsxcpYHk7HCvanrASFuMIK+7NUuYWweSA4Fi8QgSCxFKLiplx2WY
u2NY5MR0Z7HnJGaj749Mduub4xO5dtOrZYF35/ujVrrWYVQl3ZNJvSpGHyqKBbsEk
2Ry3bQAOPkdewt3eQ29uPq9v5T1B8FCdVFcOb6Uz6tgQsfqiqD3KG3MitqzP+x2d3
f1AgOAnsEW363SyPOdrSPebvx3ng3EoxegtfKEpPlBXLL84HCqDV0yP2TNytEBZuq
ml5o1J7uUwP6ve9Epnf0cDZ/jIh16fNd0zUcHZcchl1OhMiIILEkHGxCSqsE79Jno
hSz3HEQs4hRGbGaYXCs7OKEYeZxWeQBC/l0GCUGz2kfu5/dh4E5by7aeWbw2TjgER
CTYts6q0uPlnzyYzk0TFaQ0IkY0Ni2yzhtLNix8YYng6ifmqnV6ogrLlbzzptchyO
PlM0WOXLY0BkExE3AzybdjvYUsftACdoT2L8RYNYA+Xo7GEnhgJxGmMQ/Nwm1yVt7
aPeFtSs3BRma4HrfmSGeEa4buyou+AaX8vre6IkU8nb3DOV+0Frf7XAnN97PsyXoa
GQu/1VhymqWcVj6OzoH0Oay6fJJVzCU5rlFVvBqHkDjDnKDkxXlYgwH8w2A+G17h+
SDjrI2Xkq+dckDF+NZLf2jrOGOW8M0qrs00xJRNpwOHeL5vv09qRqJQjELIrOury/
uBQICcQ6iy7f3pQ+g5KArmz+n7PXf/SZlyOBPow+iRkuDa293xGt+2zOSOH5Y0QP9
6UwegRJraTha99mnfT25Xv3/6vjwIP74cu2+mdfOmtjILGyHxkXWParuvMyq99pr7
Ts8+MhY1/er9RqccOB3rKLbBTAOWDrbZo0t7c9P0Dj50Q3G7qkuGVJc+HnBd5I104
xUVQDu8KOdAnH4ZeeVUjtrjywFWcLZV0rvna9nThhBhGQreU7nCZbLZGMz38HyzkI
/HdxXxTmu0nlg==" />
</div>
```

Now, if I disable the *ViewState* for the *Grid* (by setting the *ViewStateMode* attribute to 'Disabled'), the *ViewState* for the page is drastically reduced to this:

```
<input type="hidden" name="__VIEWSTATE" id="__VIEWSTATE"
value="P7TsNJO0o83Ega0yoyRXs+hV4rIR6f7kZ2PQoMesCBLPN1/eRlBBD56zVl
Ybzlmkx4snc2eVxCliiCUIYPld3oko/g6hILIeDJmL82wjhmP7zFpHCWhpmVw1no5
zZAPGC7uVr8QUxt/c5HEtQFRyJxSkcWdXfTzXq6YheRAnmd0=" />
</div>
```

ViewState is useful in allowing you to manage data across page postbacks and making it easier for you to be productive in building applications — we hope that the new feature to have fine control over your *ViewState* will give you a sweet spot in optimizing your applications.

Routing

When it comes to SEO, an important feature is to create SEO-friendly URLs. This is typically seen as the realm of MVC, where you can have a route, such as */Home/Accounts/Laurence* instead of the location of a page with a bunch of parameters, such as */home/Accounts.aspx?id=Laurence*. The good news is that this routing is available to you as an ASP.NET Web forms developer to provide these SEO-Friendly URLs. You can define these routes in your global.asax file using a new property on your page called *RouteData*. Also, if you are using data controls, and want to pull the parameter in from the route, there is a new control called *<asp:RouteParameter>*, which provides this functionality. Your Global.asax should look something a little like this:

```
using System.Web.Routing;
namespace RouteSampler
{
    public class Global : System.Web.HttpApplication
    {
        void Application_Start(object sender, EventArgs e)
        {
            // Code that runs on application startup
            RegisterRoutes(RouteTable.Routes);
        }
        void RegisterRoutes(RouteCollection routes)
        {
            routes.MapPageRoute(
                "home",
                "home/{Accounts}/{name}",
                "~/Accounts.aspx",
                true);
        }
```

Here you can see that we are registering a route *home/Accounts/parameter* to map to the page called *Accounts.aspx*. So if the user types that at the end of the URL, then ASP.NET will locate the *Accounts.aspx* page and execute it. The browser will never see the aspx page address, just its output. You can derive the parameter(s) of the route from the *RouteData* property of the page. Here's an example:

```
string name = this.RouteData.Values["name"] as string;
```

ASP.NET MVC

The first good news with Visual Studio 2010 and .NET 4 is that ASP.NET MVC 2.0 is built-in. It's no longer an out-of-band release, though it is also available for earlier versions of Visual Studio and .NET 3.5 as a separate release.

Strongly Typed Templated Helpers. Templated Helpers provide a way to automatically build your user interface based on a data model that is market with attributes from the *System.ComponentModel.DataAnnotations* namespace. When you specify a data type, the template helper will render the value as the appropriate type using a control, so, for example, if your data type is a string, and you want to have a *Create* page that renders a *TextBox* letting users to enter the string, it's simply a case of using code such as this:

```
<%= Html.TextBox("Name") %>
```

Client and Server Validation Support. We've greatly improved Client and Server validation support, making it easier for you to build input forms, that will directly validate your data off your data models. So, for example, you can create a class for a person like this, and use attributes from the *System.ComponentModel.DataAnnotations* namespace to specify which fields are required, and what their ranges are:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.ComponentModel.DataAnnotations;

namespace MVCValidate2
{
    public class Friend
    {
        [Required(ErrorMessage="Name is Required!")]
        public string Name { get; set; }
        [Required(ErrorMessage="Twitter Page is Required!")]
        public string TwitterPage { get; set; }
        [Required(ErrorMessage="Age is Required!")]
        [Range(0,140,ErrorMessage="Age should be between 0 and 140")]
        public int age { get; set; }
    }
}
```

A view that uses these provides a *ValidationMessage* like this:

```
<p>
      <label for="Name">Name:</label>
      <%= Html.TextBox("Name") %>
      <%= Html.ValidationMessage("Name", "*") %>
  </p>
  <p>
      <label for="TwitterPage">TwitterPage:</label>
      <%= Html.TextBox("TwitterPage") %>
      <%= Html.ValidationMessage("TwitterPage", "*") %>
  </p>
  <p>
      <label for="age">age:</label>
      <%= Html.TextBox("age") %>
      <%= Html.ValidationMessage("age", "*") %>
  </p>
```

If you try to enter data that doesn't meet the validation criteria, and post it back, code in the controller like this will reject the invalid input and redisplay the page, this time with the validation messages rendered. You can see this in Figure 8.

Here's the code:

```
[HttpPost]
public ActionResult Create(Friend friendToCreate)
{
    if (ModelState.IsValid)
    {
        // Add the data to the DB...it's valid
        // Code would go here.
        return Redirect("/");
    }
    return View(friendToCreate);
}
```

Now you might be thinking, this is all very good, but it requires me to do a postback to the server before I get any validation. Modern, AJAX-based UIs are snappier than that and provide basic validation before you send it to the server. Well, the good news is that not only is this supported using .NET 4 and ASP.NET MVC 2, but that the same code that you used for Model-based validation can also be used to validate on the client. You don't need to write any extra logic — just add some JavaScript and use a single line of code to activate it in your *View*. Here's what it would look like:

```
<script src="../../Scripts/MicrosoftAjax.js"
type="text/javascript"></script>
<script src="../../Scripts/MicrosoftMvcValidation.js"
type="text/javascript"></script>
<% Html.EnableClientValidation(); %>
```

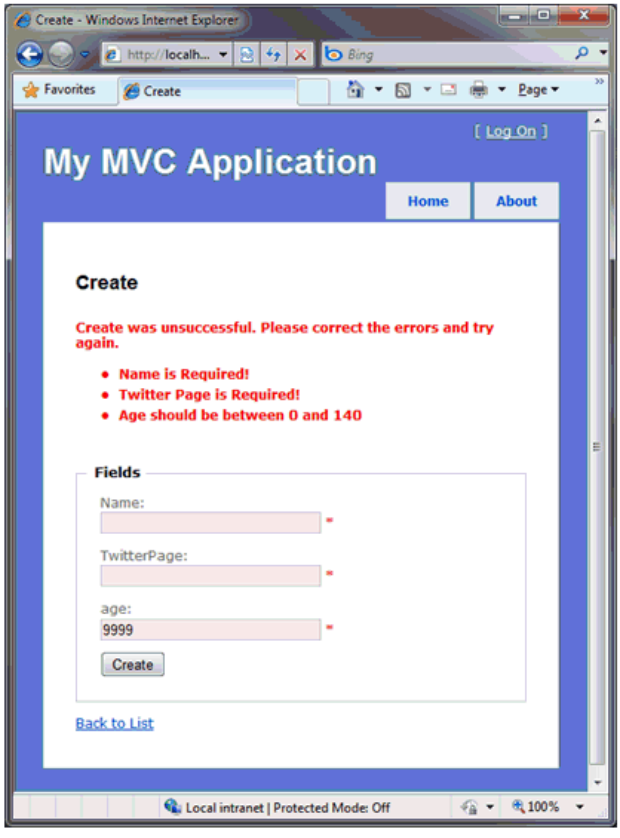


Figure 8: Using Validation in MVC.

This keeps up with the principle of "Don't Repeat Yourself" (DRY) in development, allowing you to use the same attribution to have ASP.NET MVC create the code that will validate your data on both the client and the server.

Areas

ASP.NET MVC 2 adds the concept of "areas", which let you more easily partition and group functionality across an application. They provide a way to group controllers and views allowing you to build

and maintain subsections of your application in relative isolation to other sections of your site. Each area is implemented as a separate MVC project which can then be referenced from the main application. It helps separate your code and concerns, as well as allowing diverse teams to work on a single application together. Figure 9 is an example of a Financial Portal where the main application (FinanceSite) supports areas for Fundamental Reference Data, Real Time and Trading.

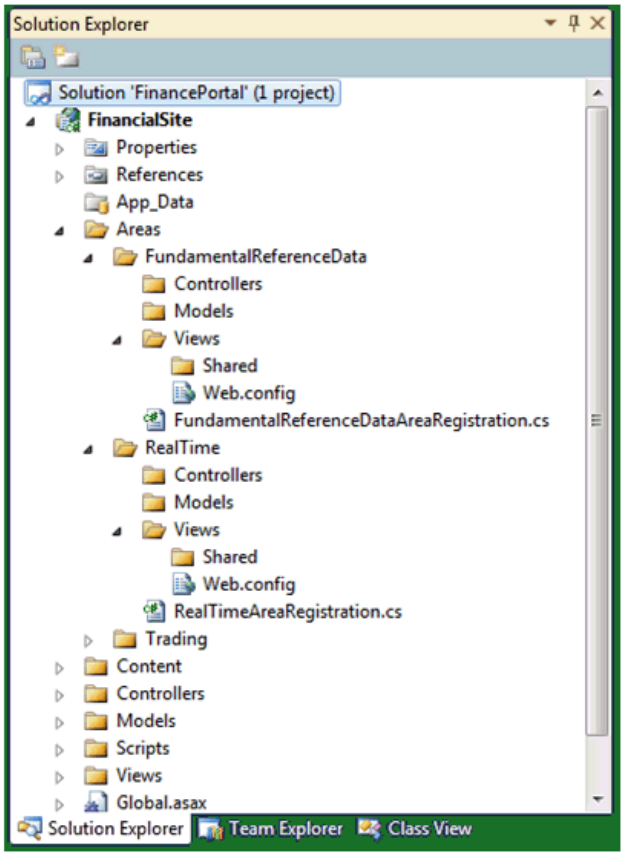


Figure 9: Areas in ASP.NET MVC.

Asynchronous Controllers

The controllers in ASP.NET MVC were synchronous in nature, and thus were not optimized for all scenarios. ASP.NET MVC2 allows you to use asynchronous controllers that you call, and they will call you back without blocking once their function is complete. So, for example, if your controller needed to call a network service to get some data back, and the transaction was long running, in ASP.NET MVC 1.0, your application would be blocked until the transaction was complete. In ASP.NET MVC 2.0, you can call the controller, and it will perform the network service call without blocking you, calling you back once its complete. So, for example a synchronous controller for a stock quote from a Web service and returning it to the view may look like this:

```
public class QuoteController: Controller {
    public ActionResult GetQuote(string ticker) {
        QuoteService quoteService = new QuoteService();
        ViewStringModel quoteValue =
quoteService.GetQuote(ticker);
        return View(quoteValue);
    }
}
```

This would be upgraded to an asynchronous controller like this:

```
public class QuoteController : AsyncController {
    public void QuoteAsync(string ticker) {
        AsyncManager.OutstandingOperations.Increment();
        QuoteService quoteService = new QuoteService();
        quoteService.GetQuoteCompleted += (sender, e) =>
        {
            AsyncManager.Parameters["quote"] = e.Value;
            AsyncManager.OutstandingOperations.Decrement();
        };
        quoteService.GetQuoteAsync(ticker);
    }
    public ActionResult GetQuoteCompleted(string quoteValue) {
        return View("Quote", new ViewStringModel
        {
            QuoteValue = quoteValue
        });
    }
}
```

If you're familiar with WebClient or Web Service calls, the paradigm is very familiar — you specify the callback function, you make the asynchronous call, and you catch the return values in the callback function. In this case the callback function drives the view.

Conclusion

There are so many new goodies in Visual Studio 2010 and .NET 4 for web developers that it's impossible to list them all without writing several books. In this article we took you on a tour of some our favorite new features, from IDE improvements that let you be a more productive developer through to some of the great extras that AJAX, ASP.NET Web Forms and ASP.NET MVC developers can take advantage of. You saw the trend that Visual Studio has been taking — making you more productive, and making your applications more Web friendly.

— Scott Guthrie is corporate vice president of Microsoft's .NET Developer Platform, where he runs the development teams responsible for delivering Visual Studio developer tools and .NET Framework technologies. Laurence Moroney is a Web Development Strategist at Microsoft, focusing on ASP.NET, Silverlight, user experience, Java, and PHP.

Return to Table of Contents

Effectively Managing Distributed Agile Teams

The do's and don'ts of Agile with onshore and offshore resources

By Geoffrey Bourn

In the early years of my technology management career, I practiced standard Waterfall/Software Development Lifecycle (SDLC) methodologies that all too often fell short as a tool to enable my team and I to effectively reach our objectives. The memories are still vivid: Mountains of unnecessary documentation and late-night project management meetings where the focus was discussing the SDLC's process rather than meeting our objectives.

So learning about Agile was a revelation. I still remember my "light-bulb moment" when reading a borrowed copy of Ken Schwaber's "Agile Project Management with Scrum" (http://www.controlchaos.com/) and realizing that there is a different way to look at how we deliver products. Schwaber's book is now my "Agile Bible" (not the borrowed copy). Since adopting Agile, my distributed development teams have risen to a new level of productivity and enjoy a more satisfying work environment. Regular, value-added communication and meeting project objectives are now our focus rather than navigating methodology.

Since finding success with Agile, it's no surprise that I've emerged as an evangelist of Agile's virtues. However, a few years ago my understanding of Agile was put to the test after a colleague challenged me to apply Agile to a team made up of distributed onshore and offshore members. Over the course of our conversation, I was cornered into admitting that Agile doesn't speak to a distributed environment. Instead, I realized that over the course of many years working with a distributed Agile environment, I unconsciously compensated for gaps by modifying practices,which begs the question, "Is Agile still Agile in a distributed environment?"

I experienced a re-awakening of my commitment to Agile at a talk given by Agile Scrum guru and co-creator Jeff Sutherland (http://jeffsutherland.com/). Just as my colleague had challenged me, a member of the audience asked Sutherland to apply Agile to distributed teams. Needless to say, Sutherland's response was similar to mine but with a lot less stammering! He, too, ran into inconsistencies with distributed Agile teams and worked around them by applying new processes and techniques that are not necessarily prescribed by the standard Agile process. All was right with the world again.

Realizing that there was a need to face concerns about applying Agile to a distributed team (rather than run from the challenge!), I began to coalesce and document my experiences in effectively running an Agile team within a distributed environment. And what is my conclusion upon meditating on Agile in a distributed environment? Distributed Agile is still Agile, but with a twist.

The Challenge

A primary tenet of the Agile process is the co-location of the team. Ideally, developers sit only a few feet away from the Product Owner or business partners. This co-location typically facilitates several benefits of Agile: frequent communication and feedback, a sense of ownership, and team building. However, managers are often forced, due to budget constraints and corporate policy, to manage a distributed team comprised of both onshore and offshore resources. These distributed Agile teams often have unique challenges that the Agile process doesn't address.

The biggest challenge managers face with a distributed Agile team is a breakdown in communication. Typically a distributed team is composed of an onshore Product/Business Owner and development staff partially or fully offshore. From a U.S. perspective, the time zone difference can vary from a few hours in Latin America to 12 hours in parts of India. (The time zones I discuss here are from an Eastern Standard Time, EST, perspective.) This time zone difference coupled with the lack of co-location is conducive to a breakdown in communication, and dealing with this challenge needs to be of primary focus. In this article, I discuss optimizing communication when managing a distributed Agile team. Although many forms of Agile exist, I focus here on principles from a SCRUM management and an XP engineering perspective and presupposes a basic understanding of both.

Laying the Foundation

How do you optimally arrange a distributed Agile team and put in place "best practices"? Agile teams are expected to function as self-organizing entities that find their own rhythm and flow within the constructs of the Agile process. However, a distributed environment requires more structure, management, and rules to effectively work. Laying the foundation of your team structure and the rules of engagement is a difficult, but incredibly beneficial first step.

The Team

Establishing strong leadership both onshore and offshore is essential for success. Since communication turn-around times can reach up to 24 hours, a distributed Agile environment can't rely upon the standard structure of a single ScrumMaster (i.e., Coach) who leads and guides the team through the process. Within a distributed Agile environment, a manager should designate a "Facilitator" who locally leads the Stand-Ups (Daily Scrums), prevents outside interference and primarily "makes [Agile] work" (Schwaber, 2003). The "Facilitator" should be as experienced as a ScrumMaster or as an Agile technical lead and act as the proxy for the onshore ScrumMaster (or visa-versa where the ScrumMaster is offshore and the "Facilitator" is onshore). They will lead the local Stand-Ups, give guidance to the team and remove roadblocks. Just remember that we are not setting up two Agile teams, but rather one team with a designated onshore ScrumMaster and an offshore "Facilitator" who acts locally on behalf of the ScrumMaster. As Conway's Law states, "The interface structure of a software system necessarily will show a congruence with the social structure of the organization that produced it" (Brooks, 1995). Certainly a mouthful, but a notable gem of wisdom: if we treat the social structure of our Agile team as onshore and offshore, our work will negatively reflect this distribution. Instead of two teams marching to two beats, we must strive for one team marching to one beat.

While distance separates the ScrumMaster and the "Facilitator", they must work together to create a cohesive team by promoting communication. As the saying goes, it is great to "put a name to the face," so if possible have the onshore and offshore teams initially co-locate. Time spent together is invaluable for learning colleagues' work style and building personal relationships. Schedule events where team members can socialize and build relationships (e.g., working lunches and after-work drinks). Taking the initiative to create strong interpersonal relationships in the early stages of a project is excellent for building a work environment that facilitates growth, efficiency, and self-organization; for example, by discovering the birthdays of all your team members and on their big day send them a birthday e-card or mention it during the Stand-Up. A simple gesture that can have a profound impact.

Iterations

What is the proper length of an Iteration in a distributed Agile environment? Due to the distributed environment and potential communication issues, it is very important to obtain frequent Product Owner feedback so that the team can adjust and refocus as appropriate. On the one hand, shorter Iterations are conducive to frequent Product Owner feedback but require a greater time commitment due to frequent Planning and Demoing (possibly requiring longer hours for onshore and offshore members). On the other hand, longer Iterations reduce the time-commitment but can exasperate communications issues due to infrequent Product Owner feedback. A reasonable compromise is two-week or three-week Iterations where the team is asked to make at most a bi-weekly time sacrifice. In this way, communication chasms are discovered at worst tri-weekly (but hopefully are found sooner in the more frequent Stand-Up).

However, the time zone differences create a difficulty. No matter the length of the Iterations, when we have the Demo (Sprint Review), Planning and Estimation, and Retrospective with the Product Owner the onshore and offshore teams must be present to listen and to ask questions (by phone, WebEx, or video conference). Their presence and understanding is critical for the success of the Iteration and cannot be substituted with emails or post-meeting written summaries. Unfortunately, this requires the team to make a large time sacrifice once per Iteration by coming in either very early or very late to participate in the Planning, Demo, and Retrospective. For example, if working with India with a possible time zone difference of 10 to 12 hours, the onshore team members will have to come in early, and the offshore team members will need to stay late (switching the order every few Iterations).

Given the sacrifice the team is making, the manager must be sensitive to the time zone difference. One way to do so is by recognizing team's sacrifice and thanking the team members at the start and conclusion of the meeting. Also, make sure that the offshore team members have transportation to get home from the meeting, and schedule time for a dinner break. Once during a late night (for the offshore team) Demo/Iteration Planning session, I ignored the clock and attempted to plow straight on through the agenda. Slowly but surely members of the offshore team started disappearing. I admittedly got a bit irritated and stated that we all need to focus to finish as soon as possible. An onshore member, who had

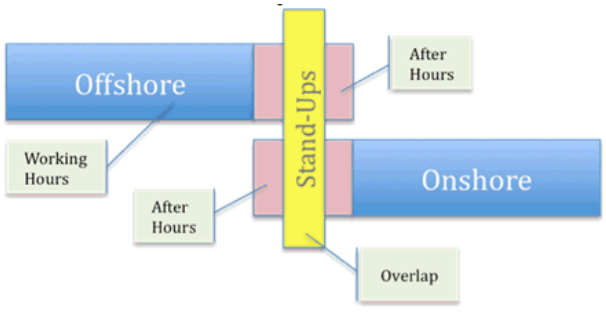


Figure 1: Joint Stand-Ups with Distributed Teams

previously worked at the offshore site, discretely took me aside to remind me of the late hour in India and that people were just stepping away to grab dinner and arrange for transportation home. Brought to my senses I apologized and took the lesson to heart.

Finally, while the team should jointly create the high-level feature point estimation with the Product Owner, the actual detailed engineering task breakdown (e.g., creation of Data Access Objects task) can be completed offline and communicated and reviewed via email or during the next Stand-Up. While this is not ideal, it allows a respite from the time commitment. Figure 2 illustrates an example of an onshore and offshore team's time commitment at the end of an Iteration.

Stand-Ups

The daily Stand-Up meetings should be face-to-face and happen, well, daily. Unfortunately with a distributed Agile team, daily Stand-Ups just aren't practical. During the initial Iteration, the onshore and offshore teams should have local Stand-Ups twice a week led by the ScrumMaster or the "Facilitator", respectively. Joint Stand-Ups (i.e., onshore and offshore present, typically via phone) should occur the other three days of the week, led only by the ScrumMaster. The Joint Stand-Ups will require the onshore and offshore to agree upon a common meeting time. Once a team's rhythm is established the frequency of the Joint Stand-Up meetings can be reduced down to bi-weekly. While we aren't utilizing the powerful communication tool of daily face-to-face Stand-Ups, with a system of partial local and partial Joint Daily Stand-Ups we will see a significant increase in communication with moderate personal sacrifice. Explore allowing flexible work hours, such as the offshore team starting their day at noon. I have had some of the younger team members prefer to work from noon until 8:00-9:00 PM (taking advantage of the active local party scene), but be sure you gain team buy-in before proceeding. Figure 1 illustrates the Joint Stand-Up time commitment (bi- or tri-weekly) where the offshore ends their day with the Stand-Up and the onshore begins their day with the Stand-Up.

The Conversation Cut-Off (Time-Boxing)

Cognition "of the clock" is critical to preventing team burnout. In the Agile process, preserving team members' personal time and quality of life are important. At critical junctures, the team will be required to make sacrifices, but keep those moments as the exception and not the rule. This means that the ScrumMaster should be observant of the team having too many early mornings or late night calls. Individuals, and by extension teams, have a natural desire to do the best they can and often members will slip into making themselves available at anytime and working longer hours to meet deadlines. A work-life balance is necessary for maintaining a high-functioning team. At all costs prevent a "Death March organization, which requires (or strongly encourages) employees to work extensive overtime week after week" (Shore & Warden, 2008). Death Marches may seem beneficial or necessary at first, but in the long run you are merely creating a self-imploding team.

Vigilantly monitor meeting conversations and ensure they do not run on, especially when you have joint meetings with onshore and offshore. Always ask yourself, "Does further discussion at this point bring us closer to a resolution or will it provide more information?" Time-box conversations by either making a decision based on what you currently know and revisiting only when new information presents itself or ask for a specific party to come back with a decision. Never leave the decision timeframe open; set a timeframe of a day or two. This is not only applicable for the Stand-Ups, but also for the Demos and Iteration Planning. If the business has an extended internal discussion in front of the whole team, politely, but firmly, ask that the business get back to you within a reasonable timeframe. I have found that a way to help time-box is to setup an agenda with specific discussion times. For example, if the agenda states that the demo of the "Done" functionality is to take from 9:00 AM until 10:00 AM it is easy to say, "time check, we have 15 minutes left" or "time check, we are 15 minutes over and eating into the next section's allotted time." Figure 2 is an example of time-boxing with an agenda.

As a reminder, don't forget the sacrifices the team has made to attend the meeting. A heart felt "Thank You" accompanied by food and drinks go a long way. If within your power, give the team comp days (incentives should be team based). These small tokens of appreciation will earn the respect and gratitude of your team. I've even received a half page long "Thank You" email after giving the team a comp day — and they were the ones doing the hard work!

Facilitating Communication

Another significant question is "How does the team communicate in a distributed environment?" Communication starts with the manager or ScrumMaster setting forth the vision of the company, team, and project. If you don't know why you're doing something, it is hard to do it right. Also, don't assume just because you are deeply aware of the vision that everyone else is, particularly the offshore team. Sometime the offshore team feels left out of the loop, so bringing them into the fold is critical to a successful project.

Agile greatly relies upon the close proximity of the team and the Product Owner, where communicating is as simple as turning around and asking if your teammate has a minute. In a distributed environment it is difficult to overcome the delay in communication; email responses are often delayed and misunderstandings may require immediate rectification. One can potentially lose a whole day due to a misunderstanding and that is not the Agile way. In a distributed environment you must diligently establish the lines of communication. Furthermore, it is important to set guidelines on how and when to use them ; see Figure 3. For example, email might be appropriate for a non-time sensitive question on a front-end interface; however, updates to your estimated remaining hours in an Iteration might be best put in a SharePoint site or Xplanner. A time-sensitive or a confusing issue might be best discussed immediately following the Stand-Up. By establishing lines of communication with specific guidelines, you are reducing the occurrence of the classic excuse, "I was waiting for a response back."

As suggested above, a great time to have further joint discussions is immediately following the Stand-Up. Since we are dealing with time zone differences, allot at most 30-45 minutes following the bi-weekly Stand-Up to have a general team discussion. This is a time when the team can hear each other's voice, experience each other's persona and quickly resolve open issues. I have seen some of the most significant conversations arise from these post Stand-Ups discussions. Remember that this is not the ScrumMaster's time, but rather the developer's time. The ScrumMaster should stay silent unless a critical point needs to be addressed.

Staying on the same page is especially challenging with a distributed team. A great way to overcome this issue is to have peer code inspections and formal reviews involving the entire team. Before checking in any code, the developer should have another team member inspect their work (encourage cross-site inspections). Also, group code review should occur every other Iteration where someone is chosen to present (i.e., brown-bag lunch/dinner) the goal, solution and delivery. These inspections and reviews are wonderful quality controls, learning exercises and in the case of the brown-bag lunches, presentation practice.

Critical to staying on the same page is getting to know one another. As in any relationship, the personal side is where bonds and kinship form. Create a cohesive team (not a vapor team) by spending the time to learn about each other's interest and culture. As Martin Fowler points out, face-to-face meetings are the best vehicle for communication where gossip and information tidbits can be related; have the team work co-located for a short while or send periodic Ambassadors to the each other's site (Fowler, 2006). Video conferencing during demos is an incredible way to put the name to the face.

Again, another major obstacle to communication is the time zone difference. If possible within your organization, attempt to select an offshore team that has more overlapping hours with the onshore team such as Latin America or Eastern Europe for a U.S. based onshore team. Working with an offshore team in a time zone

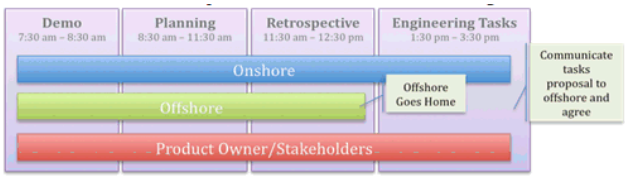


Figure 2: Example of Time Commitments with Time-Boxing

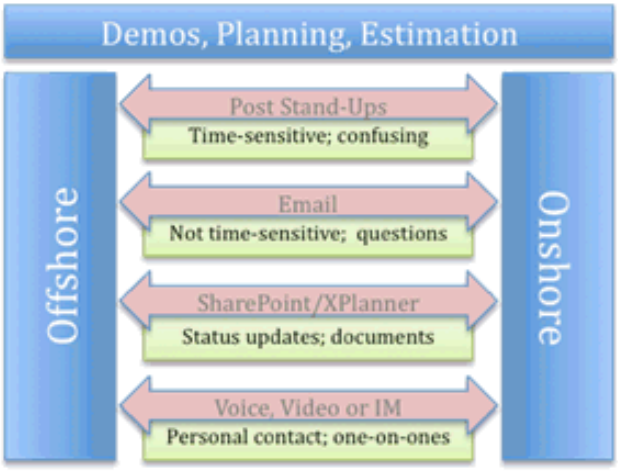


Figure 3: Lines of Communication

with more overlapping hours can make a world of difference to improving the communication gap and output velocity. Also, be sure to select an offshore team that is reasonably fluent in the same language as the onshore team, thus keeping language miscommunication to a minimum (Krym, 2009).

Finally, a word on the role of documentation in communication. In the traditional Waterfall/SDLC methodology, documentation is a primary deliverable. Agile, of course, differs greatly on the role of documentation. A principle of the Agile Manifesto is delivering working software over comprehensive documentation; reduce complexity to focus on value. However, in a distributed Agile environment, we actually reduce complexity by strategically increasing the amount of required documentation, to a degree. By creating appropriate documentation, the team can better communicate designs, work-flows, issues, etc. A new design the onshore team came up with might most easily be communicated to the offshore team via a sequence, component or class diagram (stored on SharePoint or a shared drive). A word of caution is to not treat the documentation as Waterfall documentation that needs continuous updates. Agile documentation should be created and updated as long as it is directly useful; once the usefulness is outlived, abandon the document (Fowler, 2006).

Keeping the Team Productive

A happy team is a productive team. The essence of creating a happy and productive team is to treat every member equally, respectfully and professionally. Building on Conway's Law, if you treat the onshore and offshore members differently (i.e., expect

less from the offshore team) you'll have those expectations met. Expect top quality from people and they usually will attempt to meet those expectations (Carnegie, 1937). Begin by removing preconceptions on what can be done onshore versus offshore, motivating the individual members (and by extension the whole team) by giving them autonomy, and removing roadblocks and bottlenecks that prevent the team from reaching its goals.

Preconceptions: Remove the preconception that offshore can't do the complex coding. Teams self-organize around goals. If the goals have been properly communicated, the team will efficiently break down the responsibilities and tasks. Furthermore, remove the preconception that the offshore resources are fungible. The team as a cohesive unit is an asset and should be treated as such. Treating offshore team members as fungible and replacing them frequently means you are in continual team rebuild mode; a stagnation on the velocity.

Motivation: "Autonomy is a great motivator, allowing people to be both more productive and able to grow into greater responsibility" (Fowler, 2006). People want to feel that they are a part of something and at the same time have the freedom to make their own decisions. Start by treating everyone equally, respecting differing opinions and promoting an autonomous work environment. Also, create career growth and learning opportunities for all the team members (e.g., rotate the presenter of the Demos to provide learning opportunities).

Roadblocks and Bottlenecks: Retrospectives are extremely important for improving the velocity by vetting out productivity and communication issues. It is an opportunity for the team to air what has been done right and what can be done better. If an offshore team member makes a comment about a difficulty they face (i.e., network connectivity issues or feeling that onshore is making decisions without them), treat the issue seriously and follow-up. If necessary, perform root cause analysis to uncover the underlying problem.

As cautioned above, but certainly worth repeating, prevent team burn-out. A distributed Agile team can easily fall into the trap of working longer or odd hours. Short jolts of sacrifice can actually create team solidarity, but it is a slippery slope that can be destructive to the team. Remember, it is the ScrumMaster's responsibility to maintain a healthily productive team.

Using Technology Effectively

Teams are often so eager to get the project ball rolling that employing fundamental delivery tools or creating a delivery platform comes as an afterthought. How many times has the need for a source code control mechanism been left until late in the project? Given the typical requirement for rapid product delivery and the communication barriers of a distributed Agile team, the early establishment of the proper tools and technologies is essential.

Distributed Agile teams typically rely heavily on tools to facilitate communication and the broadcasting of ideas. Begin by picking the best tool for the job. To quote the philosopher Abraham Maslow, "If the only tool you have is a hammer, you tend to treat everything as if it were a nail" (Krym, 2009). Email might not be the best mechanism to share documentation, so look for a better solution such as a shared drive or SharePoint. XPlanner is an excellent open-source project planning and tracking tool for Agile projects (specifically XP) and SVN has shown itself to be a versatile open-source source code control application. Furthermore, don't be afraid to experiment and discover what works and what doesn't work during the Retrospective.

A common Retrospective item is the distributed team having environmental issues, e.g., the onshore and offshore have different system setups or connectivity issues. Achieving a solid distributed environment is a critical success factor. These types of issues, no matter the difficulty, must be resolved. If they are left to fester, the project will possibly require an enormous unplanned time commitment and the team will become increasingly frustrated. For example, I performed an analysis of the team's velocity and found that over the past few Iterations the velocity had sharply declined. I looked back to the Retrospectives and found a reoccurring and, at this point, an ignored issue: offshore network connectivity problems (the offshore would be randomly kicked off the network and lose unsaved data). With some elbow grease the network connectivity problems were resolved, thereby restoring the velocity to an upward trend and removing a painful offshore frustration.

Understanding Cultural Differences

Every culture has its own accepted practices, social norms and ways of respecting authority. It is important to realize that Agile's practice of open communication and team member equality might be daunting for some. *In Outliers: The Story of Success,* Malcolm Gladwell points out that we all have "tendencies and assumptions and reflexes handed down to us by the history of the community we grew up in." Certain cultures are more deferential towards authority, and thus don't contradict a superior (Gladwell, 2008). Gladwell provides an example of deferential attitude by quoting the Korean linguist Ho-min Shon:

> *At a dinner table, a lower-ranking person must wait until a higher-ranking person sits down and starts eating, while the reverse does not hold true. All social behavior and actions are conducted in the order of seniority or ranking; as the [Korean] saying goes, "chanmul to wi alay ka inssta," there is order even to drinking cold water.*

You may find that during the Joint Stand-Ups, some offshore members might not be comfortable giving their status or voicing a dissenting opinion. Their local manager/tech lead might provide the status or answer questions on the whole offshore team's behalf. I once asked, via email, an offshore developer a question, and instead of responding back directly, the developer responded to his local manager, who responded to his onshore manager, who finally responded to me. Given this, it is the duty of the ScrumMaster or manager to help all team members act pro-actively and voice their opinions. Attempt to assuage their discomfort by creating a

safe and open environment where dissenting opinions, new ideas and participation are not only welcome but also expected. Push back if you see the team passively acquiescing. Make sure every individual speaks his or her task estimation and you may well find that the consensus wasn't a consensus after all. Ensuring everyone has a voice will not only strengthen the team but is also conducive to a well-adjusted work environment.

Respect for the traditions and practices of your team's culture can also go a long way. For example, scheduling a Demo on Diwali, a major Indian Holiday in October/November, is disrespectful and will irk the offshore team. On the other hand, remembering and asking what their plans are for the holiday shows respect, understanding of the culture and creates a stronger team bond. Try it and you may even be treated to some wonderful Indian Diwali sweets.

A Final Thought

As we have seen, the removal of communication roadblocks is key to managing a successful distributed Agile environment. The Agile process is naturally flexible, being able to adapt and improve as necessary. However, Agile is a well-defined process with practices resulting from the lessons learned of thousands upon thousands of Agile projects. When appropriate, replace rather than remove Agile practices. Find the root cause of an issue and look for an alternative practice (Miller, 2008).

Agile greatly facilitates frequent communication and feedback, a sense of ownership, and team building. Applying Agile to a distributed environment creates a unique set of challenges, primarily around communication. However, these challenges can be overcome through forethought, structure and diligence. By following the guidelines outlined in this article, a high functioning and successful distributed Agile team can be achieved. To quote the early 20th century oil baron J. Paul Getty, "The formula for success: Rise early, work hard, strike oil."

For More Information

Brooks, F. P. (1995). *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition.* Addison-Wesley Professional.

Carnegie, D. (1937). *How To Win Friends and Influence People.* Simon & Schuster.

Fowler, M. (2006, July 18). *Using an Agile Software Process with Offshore Development.*

Gladwell, M. (2008). *Outliers: The Story of Success. Little,* Brown and Company.

Krym, N. (2009, February 05). *Using Agile with Offshore.*

Manifesto, A. (2001). *Manifesto for Agile Software Development. Retrieved from The Agile Manifesto*: http://agilemanifesto.org/

Miller, A. (2008, October). *Distributed Agile Development at Microsoft Patterns & Practices*

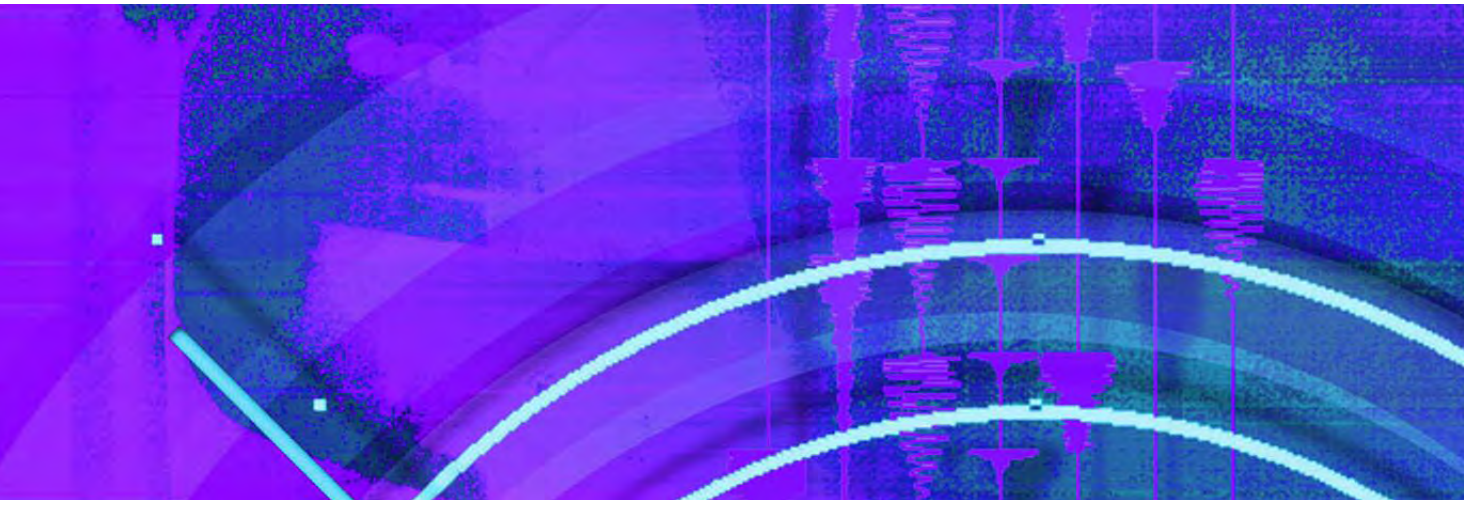
Schwaber, K. (2003). *Agile Project Management With Scrum.* Microsoft Press.

Shore, J., & Warden, S. (2008). *The Art of Agile Development.*

Yourdon, E. (1999). *Death March: The Complete Software* Developer's Guide to Surviving 'Mission Impossible' Projects. Prentice Hall.

— Geoffrey Bourne has 11 years of experience in the financial IT field, successfully managing several globally distributed Agile teams in Mumbai, Bangalore, Hong Kong, Japan, and the U.S. He has worked at J.P. Morgan, Goldman Sachs and several start-ups. He has a B.S. in Computer Science from Washington & Lee University and is a certified Sun Java Developer and PMP. Geoffrey is currently a Vice President at a major financial institution in the New York Private Bank and Personal Wealth Management divisions and can be contacted at gbourne@gmail.com.

Return to Table of Contents



Voice: It's The New UI

Windows 7 does speech recognition one better

By Gaston Hillar

Voice has yet to gain a foothold with developers, but Windows 7 is about to change that. Exchange Server 2010 includes Voice Mail Preview, so the push is on to include voice and speech awareness and interaction in applications.

"Voice is the new touch," says Zig Serafin, GM of Microsoft's speech group. "It's the natural evolution from keyboards and touch screens."

Speech-aware apps recognize human speech and react to commands. They talk back to users instead of displaying text, letting people interact with their computers in the same way they interact with other people. These apps have two components:

- Speech recognition to convert spoken words and sentences to text. Windows 7 lets users train the speech-recognition system to transform it into a voice-recognition system. This way, the speech-recognition engine improves its accuracy based on the user's unique vocal sounds.
- Speech synthesis to artificially produce human speech and talk to users. Windows 7's Text-To-Speech (TTS) engine converts text in a specific language into speech.

The Speech Recognition Control Panel application (Figure 1) offers everything you need to configure your microphone and train your computer to better understand you. You can find this application in the Ease of Access category. It offers access to the Text to Speech tab in the Speech Properties dialog box. This tab lets you select the default voice to use for the TTS engine. You can use a text to preview the voice and configure its speed and output settings (see Figure 2).

However, one of the great problems about speech-related services in Windows 7 is that the APIs and the wrappers for managed code are a

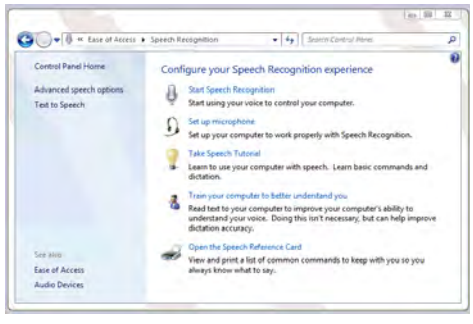


Figure 1: The Speech Recognition options in Windows 7 Control Panel.

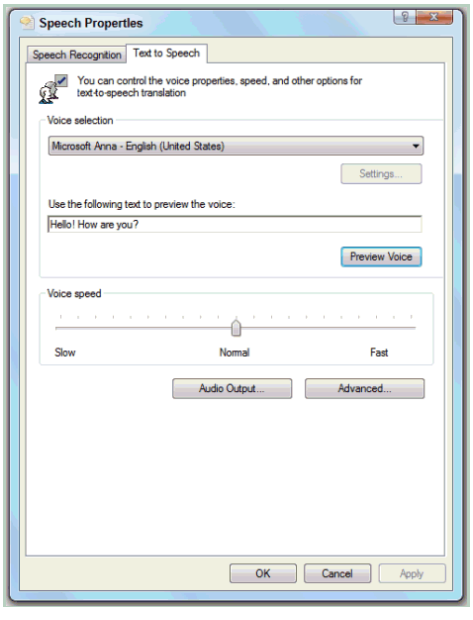


Figure 2: The default configuration for the TTS engine in Windows 7.

bit complex and lack documentation. Thus, I present in this article example C# programs to help you to create speech-aware applications.

Talking to People

To get an app to talk to users, you use speech synthesis services, wrappers provided by both .NET Framework 3.5 and .NET Framework 4 (Release Candidate). First, add the System.Speech.dll assembly as a reference to an existing C# project, then include the *System.Speech.Synthesis* namespace to access the classes, types, and enumerations offered by the speech synthesis wrapper. You can create a new instance of the *SpeechSynthesizer* class and call its *Speak* method with the text to speak:

```
using System.Speech.Synthesis;
```

This way, the TTS engine uses the default voice, its parameter values, and audio output to turn the received text into human speech:

```
var synthesizer = new SpeechSynthesizer();
synthesizer.Speak ("Hello! How are you?");
```

The statement after the call to the *Speak* method isn't executed until the TTS engine finishes saying "Hello! How are you?" To create a more responsive speech-aware application, call the *SpeakAsync* method, which produces the same effect as *Speak* but continues to the next statement after scheduling an asynchronous operation to transform the received text to speech:

```
synthesizer.SpeakAsync("Good morning!");
```

You may need to cancel an asynchronous scheduled speak command. It is possible to create a *Prompt* instance for each text to speak, then call the *SpeakAsyncCancel* for the *SpeechSynthesizer* instance, with the *Prompt* instance to be canceled as a parameter. This way, it is possible to cancel a specific text as needed. The following lines show an example that cancels "How are you?"

```
var prompt1 = new Prompt("Good morning!",
SynthesisTextFormat.Text);
var prompt2 = new Prompt("How are you?",
SynthesisTextFormat.Text);

synthesizer.SpeakAsync(prompt1);
synthesizer.SpeakAsync(prompt2);

// Cancel prompt1 -> "How are you?"
synthesizer.SpeakAsyncCancel(prompt1);
```

You can also cancel all the scheduled asynchronous speak operations by calling the *SpeakAsyncCancellAll* for the *SpeechSynthesizer* instance, without parameters.

Configuring Voices

The *SpeechSynthesizer* class also offers methods to perform the following operations:

- Retrieve the installed voices with *GetInstalledVoices*.
- Specify the voice to use by its name with *SelectVoice*.
- Specify the voice to use by hints with *SelectVoicyByHints*.

The *GetInstalledVoices* method returns a read-only collection of *InstalledVoice* instances. You can access the *VoiceInfo.Name* property for each element and use it as a parameter to the *SelectVoice* method. It is possible to fill a list box or a combo box with these values and let the user select the desired voice in your speak-aware application. The following two lines retrieve the installed voices according to the current UI culture and then select the first voice. In a default Windows 7 English (United States) installation, the value for the *VoiceInfo.Name* property is going to be "Microsoft Anna".

```
var installedVoices =
  synthesizer.GetInstalledVoices(System.Globalization.Culture-
      Info.CurrentUICulture);
synthesizer.SelectVoice(installedVoices[0].VoiceInfo.Name);
```

Besides, the *SpeechSynthesizer* class fires events to allow you to write code at specific times. For example, you can program event handlers for *SpeakStarted*, *SpeakProgress*, and *SpeakCompleted*. There are many other advanced options that let you control the way the TTS engine works.

Processing Voice User Commands

To process user voice commands use speech-recognition services that are part of the System.Speech.dll assembly. If you include the *System.Speech.Recognition* namespace, you'll be able to access the classes, types, and enumerations offered by the speech-recognition wrapper:

```
using System.Speech.Recognition;
```

Speech-recognition engines are complex and require dozens of parameters. This example simplifies recognition, using a limited list of user commands. You have to create a new instance of the *SpeechRecognitionEngine* class, accessible to many methods and events that will interact with it:

```
private SpeechRecognitionEngine _recognitionEngine=
  new SpeechRecognitionEngine();.
```

You can then define a list of alternative items to make up an element in a *Grammar*. This list is a *Choices* instance, which creates a new *GrammarBuilder* and then a *Grammar* that's loaded to the engine. The following code defines five possible voice commands:

```
string[] voiceCommands = new string[]
{
    "Favorite news",
    "Favorite movies",
    "Weather forecast",
    "New blog entry",
    "New word document"
};
var comChoices = new Choices(voiceCommands);
var comGrammarBuilder = new GrammarBuilder(comChoices);
var comGrammar = new Grammar(comGrammarBuilder);
_recognitionEngine.LoadGrammar(comGrammar);
```

There are many other ways to create *Grammars* that can accept more complex commands. In fact, you can also work with XML elements defined under the Speech Recognition Grammar

Specification (SRGS); http://www.w3.org/TR/speech-Grammar/. Now, it is necessary to add event handlers to the following events that the recognition engine is going to fire:

- *SpeechDetected*. The user started talking.
- *SpeechRecognized*. The recognition engine was capable of recognizing one of the voice commands. It is possible to check the results by adding code in an event handler attached to this event.
- *RecognizeRejected*. The user began talking but the recognition engine wasn't capable of understanding a voice command.
- *RecognizeCompleted*. The recognition engine finished its asynchronous execution. The code written in an event handler attached to this event will be executed after calling the event handler for either *SpeechRecognized* or *RecognizeRejected*.

```
_recognitionEngine.SpeechDetected +=
   new EventHandler<SpeechDetectedEventArgs>
      (recognitionEngine_SpeechDetected);
_recognitionEngine.SpeechRecognized +=
   new EventHandler<SpeechRecognizedEventArgs>
      (recognitionEngine_SpeechRecognized);
_recognitionEngine.RecognizeCompleted += new
   EventHandler<RecognizeCompletedEventArgs>
      (recognitionEngine_RecognizeCompleted);
_recognitionEngine.SpeechRecognitionRejected += new
   EventHandler<SpeechRecognitionRejectedEventArgs>
      (recognitionEngine_SpeechRecognitionRejected);
```

The following sample code shows definitions for the four event handlers. The *e.Result.Text* property in the *SpeechRecognized* event handler contains the recognized phrase that is going to match one of the previously added commands. Therefore, according to the recognized phrase, it is possible to execute an action.

```
private void recognitionEngine_RecognizeCompleted(object sender,
RecognizeCompletedEventArgs e)
{
    // Start another recognition
    _recognitionEngine.RecognizeAsync();
}

private void recognitionEngine_SpeechRecognitionRejected(object
sender, SpeechRecognitionRejectedEventArgs e)
{
    // Tell the user you didn't understand him
}

private void recognitionEngine_SpeechDetected(object sender,
SpeechDetectedEventArgs e)
{
    // Do something when the user's speech is detected
}

private void recognitionEngine_SpeechRecognized(object sender,
SpeechRecognizedEventArgs e)
{
    if (e.Result.Text == "Favorite news")
    {
        // Start the application or go to the Web Site with the
        // favorite news
    }
     // Check the other possible commands
}
```

It is very easy to add simple speech recognition capabilities to an existing C# application targeting. There are dozens of additional options because the speech recognition engine is very powerful and it allows a very complex customization to improve its efficiency. However, it was necessary to start with a simple example.

Conclusion

Speech may be a natural evolution from keyboards and touch screens, but the APIs required to work with speech-related services are complex. Windows 7 gives developers the ability to create speech-aware apps through performance and accuracy improvements to the speech-recognition engine. Once you begin working with speech-aware apps, you'll find great opportunities to take advantage of this natural user interface.

—Gaston Hillar is an IT consultant and author of more than 40 books on topics ranging from systems programming to IT project management.

Return to Table of Contents

C Snippet #4

Computing the wind chill factor

by Bob Stout

Q: HOW DO I…compute the wind chill factor from temperature and wind speed? A: Okay, so I've only seen this question come up twice. It's still a fascinating Snippet, which could potentially save someone hours of research.

```
#include <math.h> float wind_chill(int wind_speed, int temp) { return (((10.45 + (6.686112 *
sqrt((double) wind_speed)) - (.447041 * wind_speed)) / 22.034 * (temp - 91.4)) + 91.4); }

Updates to C Snippet #4

/*********************************************************/ /**** weather.h ****/

/* ** SNIPPETS header file for weather functions */

#ifndef WEATHER__H #define WEATHER__H

#include "metric.h" #include "sniptype.h"

double wind_chill(int flag, double temp, double wind_speed); double heat_index(int flag, double temp,
double rh);

#endif /* WEATHER__H */

/*********************************************************/ /**** weather.c ****/

/* ** SNIPPETS weaather functions ** ** Public domain, based on the standard NOAA formuolas. */

/* ** Wind Chill for exposed human skin, expressed as a function of wind ** speed in Miles per Hour
and temperature in degrees Fahrenheit. ** ** Parameters: 1 - Metric flag ** 2 - Temperature (degrees
F or C) ** 3 - Wind speed (mph or kph) ** ** Returns: Wind chill in degreers F or C to match the
input */

#include <math.h> #include "weather.h" double wind_chill(int flag, double temp, double wind_speed) {
double retval;

if (flag) // Convert to English if metric { temp = C_to_F(temp); wind_speed = KPH_to_MPH(wind_speed);
} retval = temp;

if (wind_speed >= 3.0 && temp <= 50.0) { double V2 = pow(wind_speed, 0.16); retval = 35.74 + (0.6215
* temp) - (35.75 * V2) + (0.4275 * temp * V2); } if (flag) retval = F_to_C(retval); return retval; }

/* ** Heat Index for an everage size human in late morning or early afternoon with a wind ** speed of
~6 MPH, expressed as a function of relative humidity and temperature. ** ** Parameters: 1 - Metric
flag ** 2 - Temperature (degrees F or C) ** 3 - Relative huidity (percent) ** ** Returns: Heat index
in degreers F or C to match the input */

double heat_index(int flag, double temp, double rh) { double retval;

if (flag) // Convert to English if metric temp = C_to_F(temp); retval = temp;

if (rh >= 40.0 && temp >= 80.0) { double T2 = temp*temp, R2 = rh*rh; double coeff[] = {-42.379,
2.04901523, 10.14333127, -0.22475541, -6.83783E-3, -5.481717E-2, 1.22874E-3, 8.5282E-4, -1.99E-6};

retval = coeff[0] + coeff[1] * temp + coeff[2] * rh + coeff[3] * temp * rh + coeff[4] * T2 + coeff[5]
* R2 + coeff[6] * T2 * rh + coeff[7] * temp * R2 + coeff[8] * T2 * R2; } if (flag) retval =
F_to_C(retval); return retval; }

#ifdef TEST #include <stdio.h>

int main(void) { // Test data double Tf[] = // Temp in degrees F {65, 20, 0, -20, 80, 90, 100};
double Tc[] = // Temp in degrees C {18.33333, -6.666667, -17.77778, -28.88889, 26.66667, 32.22222,
37.77778}; double Wm[] = // Wind speed in MPH {0, 100, 10, 30}; double Wk[] = // Wind speed in KPH
{0, 160.9344, 16.09344, 48.28032}; double Hr[] = // Relative humidity in percent {0, 100, 60, 90};
double WCf[] = // Wind chill in degrees F {9, 1, -16, -26, -40, -53}; double WCc[] = // Wind chill in
degrees C {-12.77778, -17.22222, -26.66667, -32.22222, -40, -47.22222}; double HIf[] = // Heat index
in degrees F {81, 85, 99, 121, 129}; double HIc[] = // Heat index in degrees F {27.22222, 29.44444,
37.22222, 49.44444, 53.88889};

puts("Apparent Temperaures"); puts("--------------------"); puts("The following tests use equivalent
data for English and metric units:"); puts("\nThese should all return 65F or 18.33333C");
printf("wind_chill(%d, %fF, %fMPH) = %fF\n", False_, Tf[0], Wm[0], wind_chill(False_, Tf[0], Wm[0]));
printf("wind_chill(%d, %fF, %fMPH) = %fF\n", False_, Tf[0], Wm[1], wind_chill(False_, Tf[0], Wm[1]));
printf("wind_chill(%d, %fC, %fKPH) = %fC\n", True_, Tc[0], Wk[0], wind_chill(True_, Tc[0], Wk[0]));
printf("wind_chill(%d, %fC, %fKPH) = %fC\n", True_, Tc[0], Wk[1], wind_chill(True_, Tc[0], Wk[1]));
printf("heat_index(%d, %fF, %f%%) = %fF\n", False_, Tf[0], Hr[0], heat_index(False_, Tf[0], Hr[0]));
printf("heat_index(%d, %fF, %f%%) = %fF\n", False_, Tf[0], Hr[1], heat_index(False_, Tf[0], Hr[1]));
printf("heat_index(%d, %fC, %f%%) = %fC\n", True_, Tc[0], Hr[0], heat_index(True_, Tc[0], Hr[0]));
```

```
printf("heat_index(%d, %fC, %f%%) = %fC\n", True_, Tc[0], Hr[1],
heat_index(True_, Tc[0], Hr[1]));

puts("\nThese should all return the indicated values");
printf("wind_chill(%d, %fF, %fMPH) = %fF (should be %fF)\n",
False_, Tf[1], Wm[2], wind_chill(False_, Tf[1], Wm[2]), WCf[0]);
printf("wind_chill(%d, %fF, %fMPH) = %fF (should be %fF)\n",
False_, Tf[1], Wm[3], wind_chill(False_, Tf[1], Wm[3]), WCf[1]);
printf("wind_chill(%d, %fF, %fMPH) = %fF (should be %fF)\n",
False_, Tf[2], Wm[2], wind_chill(False_, Tf[2], Wm[2]), WCf[2]);
printf("wind_chill(%d, %fF, %fMPH) = %fF (should be %fF)\n",
False_, Tf[2], Wm[3], wind_chill(False_, Tf[2], Wm[3]), WCf[3]);
printf("wind_chill(%d, %fF, %fMPH) = %fF (should be %fF)\n",
False_, Tf[3], Wm[2], wind_chill(False_, Tf[3], Wm[2]), WCf[4]);
printf("wind_chill(%d, %fF, %fMPH) = %fF (should be %fF)\n",
False_, Tf[3], Wm[3], wind_chill(False_, Tf[3], Wm[3]), WCf[5]);

printf("wind_chill(%d, %fC, %fKPH) = %fC (should be %fC)\n",
True_, Tc[1], Wk[2], wind_chill(True_, Tc[1], Wk[2]), WCc[0]);
printf("wind_chill(%d, %fC, %fKPH) = %fC (should be %fC)\n",
True_, Tc[1], Wk[3], wind_chill(True_, Tc[1], Wk[3]), WCc[1]);
printf("wind_chill(%d, %fC, %fKPH) = %fC (should be %fC)\n",
True_, Tc[2], Wk[2], wind_chill(True_, Tc[2], Wk[2]), WCc[2]);
printf("wind_chill(%d, %fC, %fKPH) = %fC (should be %fC)\n",
True_, Tc[2], Wk[3], wind_chill(True_, Tc[2], Wk[3]), WCc[3]);
printf("wind_chill(%d, %fC, %fKPH) = %fC (should be %fC)\n",
True_, Tc[3], Wk[2], wind_chill(True_, Tc[3], Wk[2]), WCc[4]);
printf("wind_chill(%d, %fC, %fKPH) = %fC (should be %fC)\n",
True_, Tc[3], Wk[3], wind_chill(True_, Tc[3], Wk[3]), WCc[5]);
puts(""); printf("heat_index(%d, %fF, %f%%) = %fF (should be
%fF)\n", False_, Tf[4], Hr[2], heat_index(False_, Tf[4], Hr[2]),
HIf[0]); printf("heat_index(%d, %fF, %f%%) = %fF (should be
%fF)\n", False_, Tf[4], Hr[3], heat_index(False_, Tf[4], Hr[3]),
HIf[1]); printf("heat_index(%d, %fF, %f%%) = %fF (should be
%fF)\n", False_, Tf[5], Hr[2], heat_index(False_, Tf[5], Hr[2]),
HIf[2]); printf("heat_index(%d, %fF, %f%%) = %fF (should be
%fF)\n", False_, Tf[5], Hr[3], heat_index(False_, Tf[5], Hr[3]),
HIf[3]); printf("heat_index(%d, %fF, %f%%) = %fF (should be
%fF)\n", False_, Tf[6], Hr[2], heat_index(False_, Tf[6], Hr[2]),
HIf[4]); printf("heat_index(%d, %fC, %f%%) = %fC (should be
%fC)\n", True_, Tc[4], Hr[2], heat_index(True_, Tc[4], Hr[2]),
HIc[0]); printf("heat_index(%d, %fC, %f%%) = %fC (should be
%fC)\n", True_, Tc[4], Hr[3], heat_index(True_, Tc[4], Hr[3]),
HIc[1]); printf("heat_index(%d, %fC, %f%%) = %fC (should be
%fC)\n", True_, Tc[5], Hr[2], heat_index(True_, Tc[5], Hr[2]),
HIc[2]); printf("heat_index(%d, %fC, %f%%) = %fC (should be
%fC)\n", True_, Tc[5], Hr[3], heat_index(True_, Tc[5], Hr[3]),
HIc[3]); printf("heat_index(%d, %fC, %f%%) = %fC (should be
%fC)\n", True_, Tc[6], Hr[2], heat_index(True_, Tc[6], Hr[2]),
HIc[4]);

puts("\nNote: The formulas are approximations. Exposure to direct
sunlight can"); puts(" cause the apparent tmperatures to be
higher.");

return EXIT_SUCCESS; }

#endif // TEST

/*****************************************************/ /****
sniptype.h ****/

/* ** SNIPTYPE.H - Include file for SNIPPETS data types and
commonly used macros */

#ifndef SNIPTYPE__H #define SNIPTYPE__H

#include <stdlib.h> /* For free() */ #include <string.h> /* For
NULL & strlen() */

typedef enum {Error_ = -1, Success_, False_ = 0, True_} Boolean_T;

#ifdef BYTE #undef BYTE #endif #ifdef WORD #undef WORD #endif #
ifdef DWORD #undef DWORD #endif typedef unsigned char BYTE;
typedef unsigned short WORD; typedef unsigned long DWORD;

typedef union { signed char c; BYTE b; } VAR8_;

typedef union { VAR8_ v8[2]; signed short s; WORD w; } VAR16_;

typedef union { VAR16_ v16[2]; signed long l; DWORD dw; float f;
void *p; } VAR32_;

typedef union { VAR32_ v32[2]; double d; } VAR64_;

#define NUL '\0' #define LAST_CHAR(s) (((char *)s)[strlen(s) - 1])
#define TOBOOL(x) (!(!(x))) #define FREE(p) (free(p),(p)=NULL)

#endif /* SNIPTYPE__H */

/*****************************************************/ /****
metric.h ****/

/********************************************** * * ANSI C
Metric Conversion Macros * Mike Smith, Freeware * 6 Grenville
Road, Plymouth, England, PL4 9PX * * mike@msmith.demon.co.uk * *
Version: 1.03 * History: 5th of May, 1997 * * 1.01 Taken out
formulas and added values only Mon 17th April 1995 * 1.02 Error
corrections Sat 13th July 1996 * 1.03 Added macros 5th of May,
1997 * 1.04 Added dram <-> fl.oz. macros (RBS) 21-Aug-1998 * 1.05
Added in.^3 <-> l. macros (RBS) 05-Sep-1998 * 2.00 Added C++,
velocity and temperature macros (RBS) 23-Sep-2009 *
**********************************************/

#ifndef METRIC__H #define METRIC__H

#if !(defined(__cplusplus) && __cplusplus)

/******** Length Conversion Macros **********/

#define CM_to_IN(x) (0.3937 * (x)) #define IN_to_CM(x) (2.54 *
(x)) #define IN_to_M(x) (0.0254 * (x)) #define IN_to_MM(x) (25.4 *
(x)) #define KM_to_MILE(x) (0.6214 * (x)) #define M_to_IN(x)
(39.37 * (x)) #define M_to_FT(x) (3.28083 * (x)) #define
M_to_YARD(x) (1.093611 * (x)) #define MILE_to_KM(x) (1.609 * (x))
#define MM_to_IN(x) (0.03937 * (x)) #define YARD_to_M(x) (0.9144 *
(x)) #define IN_to_FT(x) ((x) / 12.0) #define IN_to_YARD(x) ((x) /
36.0) #define IN_to_MILE(x) ((x) / 63360.0) #define FT_to_IN(x)
(12.0 * (x)) #define FT_to_M(x) (0.3048 * (x)) #define
FT_to_YARD(x) ((x) / 3.0) #define FT_to_MILE(x) ((x) / 5280.0)
#define MILE_to_IN(x) (63360.0 * (x)) #define MILE_to_FT(x)
(5280.0 * (x)) #define MILE_to_YARD(x) (1760.0 * (x))

#define AWG_to_IN(awg) (0.46/pow(1.1229283027,(awg)+3))

/******* Weight/Mass Macros ************/

#define G_to_OZ(x) (0.03527 * (x)) #define KG_to_LBS(x) (2.2046 *
(x)) #define LBS_to_KG(x) (0.4545 * (x)) #define LBS_to_OZ(x)
(16.0 * (x)) #define OZ_to_G(x) (28.3527 * (x)) #define
OZ_to_LBS(x) ((x) / 16.0)

/******* Velocity Conversion Macros ********/

#define MPH_to_KPH(x) MILE_to_KM(x) #define KPH_to_MPH(x)
KM_to_MILE(x) #define FPS_to_MPS(x) FT_to_M(x) #define
MPS_to_FPS(x) M_to_FT(x)

/******* Temperature Conversion Macros ********/

#define C_to_F(x) (1.8 * (x) + 32) #define F_to_C(x) (((x) - 32) /
1.8)

/******* Liquid Conversion Macros ********/

#define CL_to_OZ(x) (0.338 * (x)) #define ML_to_OZ(x) (0.0338 *
(x)) #define OZ_to_CL(x) (2.95857988 * (x)) #define OZ_to_ML(x)
(29.5857 * (x))

#define OZ_to_DRAM(x) (16.0 * (x)) #define DRAM_to_OZ(x) (0.0625 *
(x))

/********************************************** * * In case you
are unaware there are: * 16 ounces in a US quart * 20 ounces in a
UK quart * * This, of course, makes gallons different * too.
Supplied below are US to UK * conversions as well. *
**********************************************/

#define KL_to_UKGAL(x) (211.344 * (x)) #define KL_to_USGAL(x)
(264.18 * (x)) #define L_to_UKGAL(x) (0.2199 * (x)) #define
L_to_UKQ(x) (0.945 * (x)) #define L_to_USGAL(x) (0.26455 * (x))
#define L_to_USQ(x) (1.0582 * (x)) #define L_to_CM3(x) (1000.0 *
(x)) #define UKGAL_to_KL(x) (0.0047316 * (x)) #define
UKGAL_to_L(x) (4.546 * (x)) #define UKGAL_to_USGAL(x) (1.25 * (x))
#define UKQ_to_L(x) (1.0582 * (x)) #define UKQ_to_USQ(x) (1.25 *
(x)) #define USGAL_to_KL(x) (0.003785 * (x)) #define USGAL_to_L(x)
(3.78 * (x)) #define USGAL_to_UKGAL(x) (0.8 * (x)) #define
USQ_to_L(x) (0.945 * (x)) #define USQ_to_UKQ(x) (0.8 * (x))
#define CM3_to_L(x) (0.001 * (x))

/**** Area Conversion Macros ****/

#define CM2_to_IN2(x) (0.155 * (x)) #define FT2_to_M2(x) (0.0930 *
(x)) #define IN2_to_CM2(x) (6.4516 * (x)) #define IN2_to_MM2(x)
(645.161 * (x)) #define KM2_to_MILE2(x) (0.386 * (x)) #define
M2_to_FT2(x) (10.75 * (x)) #define MILE2_to_KM2(x) (2.5906 * (x))
#define MM2_to_IN2(x) (0.00155 * (x))

/******** Volume Conversion Macros *********/

#define CM3_to_IN3(x) (0.061 * (x)) #define IN3_to_CC(x) (16.3934
* (x)) #define IN3_to_CM3(x) (16.3934 * (x)) #define
KM3_to_MILE3(x) (0.25 * (x)) #define M3_to_YARD3(x) (1.308 * (x))
#define MILE3_to_KM3(x) (4.0 * (x)) #define YARD3_to_M3(x)
(0.764526 * (x)) #define M3_to_FT3(x) (35.3144754 * (x)) #define
FT3_to_M3(x) (2.8317e-2 * (x))

#define L_to_IN3(x) (CM3_to_IN3(L_to_CM3(x))) #define IN3_to_L(x)
(CM3_to_L(IN3_to_CM3(x))) #define L_to_FT3(x) (L_to_IN3(x) / 1728)
#define FT3_to_L(x) (IN3_to_L((x) * 1728))

#else // C++
```

```
/******** Length Conversion Functions **********/

inline double CM_to_IN(double x) {return 0.3937 * x;} inline
double IN_to_CM(double x) {return 2.54 * x;} inline double
IN_to_M(double x) {return 0.0254 * x;} inline double
IN_to_MM(double x) {return 25.4 * x;} inline double
KM_to_MILE(double x) {return 0.6214 * x;} inline double
M_to_IN(double x) {return 39.37 * x;} inline double M_to_FT(double
x) {return 3.28083 * x;} inline double M_to_YARD(double x) {return
1.093611 * x;} inline double MILE_to_KM(double x) {return 1.609 *
x;} inline double MM_to_IN(double x) {return 0.03937 * x;} inline
double YARD_to_M(double x) {return 0.9144 * x;} inline double
IN_to_FT(double x) {return x / 12.0;} inline double
IN_to_YARD(double x) {return x / 36.0;} inline double
IN_to_MILE(double x) {return x / 63360.0;} inline double
FT_to_IN(double x) {return 12.0 * x;} inline double FT_to_M(double
x) {return 0.3048 * x;} inline double FT_to_YARD(double x) {return
x / 3.0;} inline double FT_to_MILE(double x) {return x / 5280.0;}
inline double MILE_to_IN(double x) {return 63360.0 * x;} inline
double MILE_to_FT(double x) {return 5280.0 * x;} inline double
MILE_to_YARD(double x) {return 1760.0 * x;}

inline double AWG_to_IN(double awg) {return
0.46/pow(1.1229283027,awg+3);}

/******* Weight/Mass Functions ************/

inline double G_to_OZ(double x) {return 0.03527 * x;} inline
double KG_to_LBS(double x) {return 2.2046 * x;} inline double
LBS_to_KG(double x) {return 0.4545 * x;} inline double
LBS_to_OZ(double x) {return 16.0 * x;} inline double
OZ_to_G(double x) {return 28.3527 * x;} inline double
OZ_to_LBS(double x) {return x / 16.0;}

/******* Velocity Conversion Functions ********/

inline double MPH_to_KPH(double x) {return MILE_to_KM(x);} inline
double KPH_to_MPH(double x) {return KM_to_MILE(x);} inline double
FPS_to_MPS(double x) {return FT_to_M(x);} inline double
MPS_to_FPS(double x) {return M_to_FT(x);}

/******* Temperature Conversion Functions ********/

inline double C_to_F(double x) {return 1.8 * x + 32;} inline
double F_to_C(double x) {return (x - 32) / 1.8;}

/******* Liquid Conversion Functions ********/

inline double CL_to_OZ(double x) {return 0.338 * x;} inline double
ML_to_OZ(double x) {return 0.0338 * x;} inline double
OZ_to_CL(double x) {return 2.95857988 * x;} inline double
OZ_to_ML(double x) {return 29.5857 * x;}

inline double OZ_to_DRAM(double x) {return 16.0 * x;} inline
double DRAM_to_OZ(double x) {return 0.0625 * x;}

/************************************************ * * In case you
are unaware there are: * 16 ounces in a US quart * 20 ounces in a
UK quart * * This, of course, makes gallons different * too.
Supplied below are US to UK * conversions as well. *
************************************************/

inline double KL_to_UKGAL(double x) {return 211.344 * x;} inline
double KL_to_USGAL(double x) {return 264.18 * x;} inline double
L_to_UKGAL(double x) {return 0.2199 * x;} inline double
L_to_UKQ(double x) {return 0.945 * x;} inline double
L_to_USGAL(double x) {return 0.26455 * x;} inline double
L_to_USQ(double x) {return 1.0582 * x;} inline double
L_to_CM3(double x) {return 1000.0 * x;} inline double
UKGAL_to_KL(double x) {return 0.0047316 * x;} inline double
UKGAL_to_L(double x) {return 4.546 * x;} inline double
UKGAL_to_USGAL(double x) {return 1.25 * x;} inline double
UKQ_to_L(double x) {return 1.0582 * x;} inline double
UKQ_to_USQ(double x) {return 1.25 * x;} inline double
USGAL_to_KL(double x) {return 0.003785 * x;} inline double
USGAL_to_L(double x) {return 3.78 * x;} inline double
USGAL_to_UKGAL(double x) {return 0.8 * x;} inline double
USQ_to_L(double x) {return 0.945 * x;} inline double
USQ_to_UKQ(double x) {return 0.8 * x;} inline double
CM3_to_L(double x) {return 0.001 * x;}

/**** Area Conversion Functions ****/

inline double CM2_to_IN2(double x) {return 0.155 * x;} inline
double FT2_to_M2(double x) {return 0.0930 * x;} inline double
IN2_to_CM2(double x) {return 6.4516 * x;} inline double
IN2_to_MM2(double x) {return 645.161 * x;} inline double
KM2_to_MILE2(double x) {return 0.386 * x;} inline double
M2_to_FT2(double x) {return 10.75 * x;} inline double
MILE2_to_KM2(double x) {return 2.5906 * x;} inline double
MM2_to_IN2(double x) {return 0.00155 * x;}

/******** Volume Conversion Functions *********/

inline double CM3_to_IN3(double x) {return 0.061 * x;} inline
double IN3_to_CC(double x) {return 16.3934 * x;} inline double
IN3_to_CM3(double x) {return 16.3934 * x;} inline double
KM3_to_MILE3(double x) {return 0.25 * x;} inline double
M3_to_YARD3(double x) {return 1.308 * x;} inline double
MILE3_to_KM3(double x) {return 4.0 * x;} inline double
YARD3_to_M3(double x) {return 0.764526 * x;} inline double
M3_to_FT3(double x) {return 35.3144754 * x;} inline double
FT3_to_M3(double x) {return 2.8317e-2 * x;}

inline double L_to_IN3(double x) {return CM3_to_IN3(L_to_CM3(x));}
inline double IN3_to_L(double x) {return CM3_to_L(IN3_to_CM3(x));}
inline double L_to_FT3(double x) {return L_to_IN3(x / 1728);}
inline double FT3_to_L(double x) {return IN3_to_L(x * 1728);}

#endif // C++

#endif /* METRIC__H */

/*********************************************************/ /****
weather.out ****/

Apparent Temperaures -------------------- The following tests use
equivalent data for English and metric units:

These should all return 65F or 18.33333C wind_chill(0, 65.000000F,
0.000000MPH) = 65.000000F wind_chill(0, 65.000000F, 100.000000MPH)
= 65.000000F wind_chill(1, 18.333330C, 0.000000KPH) = 18.333330C
wind_chill(1, 18.333330C, 160.934400KPH) = 18.333330C
heat_index(0, 65.000000F, 0.000000%) = 65.000000F heat_index(0,
65.000000F, 100.000000%) = 65.000000F heat_index(1, 18.333330C,
0.000000%) = 18.333330C heat_index(1, 18.333330C, 100.000000%) =
18.333330C

These should all return the indicated values wind_chill(0,
20.000000F, 10.000000MPH) = 8.854038F (should be 9.000000F)
wind_chill(0, 20.000000F, 30.000000MPH) = 1.298559F (should be
1.000000F) wind_chill(0, 0.000000F, 10.000000MPH) = -15.934472F
(should be -16.000000F) wind_chill(0, 0.000000F, 30.000000MPH) = -
25.864927F (should be -26.000000F) wind_chill(0, -20.000000F,
10.000000MPH) = -40.722982F (should be -40.000000F) wind_chill(0,
-20.000000F, 30.000000MPH) = -53.028413F (should be -53.000000F)
wind_chill(1, -6.666667C, 16.093440KPH) = -12.859030C (should be -
12.777780C) wind_chill(1, -6.666667C, 48.280320KPH) = -17.056550C
(should be -17.222220C) wind_chill(1, -17.777780C, 16.093440KPH) =
-26.630478C (should be -26.666670C) wind_chill(1, -17.777780C,
48.280320KPH) = -32.147439C (should be -32.222220C) wind_chill(1,
-28.888890C, 16.093440KPH) = -40.401922C (should be -40.000000C)
wind_chill(1, -28.888890C, 48.280320KPH) = -47.238323C (should be
-47.222220C)

heat_index(0, 80.000000F, 60.000000%) = 81.810923F (should be
81.000000F) heat_index(0, 80.000000F, 90.000000%) = 85.641892F
(should be 85.000000F) heat_index(0, 90.000000F, 60.000000%) =
99.677718F (should be 99.000000F) heat_index(0, 90.000000F,
90.000000%) = 121.901204F (should be 121.000000F) heat_index(0,
100.000000F, 60.000000%) = 129.489027F (should be 129.000000F)
heat_index(1, 26.666670C, 60.000000%) = 27.672739C (should be
27.222220C) heat_index(1, 26.666670C, 90.000000%) = 29.801060C
(should be 29.444440C) heat_index(1, 32.222220C, 60.000000%) =
37.598727C (should be 37.222220C) heat_index(1, 32.222220C,
90.000000%) = 49.945103C (should be 49.444440C) heat_index(1,
37.777780C, 60.000000%) = 54.160579C (should be 53.888890C)

Note: The formulas are approximations. Exposure to direct sunlight
can cause the apparent tmperatures to be higher.

Note: The formulas are approximations. Exposure to direct sunlight
can cause the apparent temperatures to be higher.
```

More C Snippets

- Determining filesize (www.drdobbs.com/cpp/219100141)
- Rounding floating-point values (www.drdobbs.com/cpp/219200409)
- Sorting an array of strings (www.drdobbs.com/cpp/219500313)
- Timers and default actions (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

Probability Selector

For when you really do need randomness

By Craig Lindle

We programmers usually strive to write deterministic code that has predictable, repeatable behavior. Behavior that is not predictable and repeatable usually indicates a bug that must be found and fixed. Sometimes, however, the introduction of controlled randomness may be exactly what is needed to make our presentations more dynamic or to better test software we have already written.

In my case, I was writing a ray tracing application for the iPhone/iPod Touch called Art Rays Lite (http://download.cnet.com/Art-Rays-Lite/3000-2381_4-75008502.html). This app generates unique, one of a kind, ray-traced images in real time. Initially, I used unconstrained randomness to select which geometric shapes (spheres, cones, cylinders, boxes, planes) to use within a scene, where these shapes would be placed in the 3D space, what their surface characteristics were (dull, shiny, reflective, etc.) and what colors would be used for these objects and for the scene light sources.

While the images produced were almost always interesting in some aspect, the random selection of colors produced images that were sometimes jarring and not aesthetically pleasing. Since I actually wanted to sell this app, it was important that most of the generated images were interesting and pleasant to look at.

This led me to the concept of using palettes for color selection instead of randomly generating colors. In this context, a palette is a collection of 256 related colors. I decided to define numerous palettes because I wanted a lot of variation in the generated images. To this end, I came up with the set of palettes in Table 1 for use in my ray tracer.

I started to formulate the problem as follows: I would like to use the R_GRAYSCALE palette approximately X percent of the time, the R_COLOR palette approximately Y percent of the time, the R_COLOR_FROM_IMAGE palette Z percent of the time, and so on. The *ProbabilitySelector* class was developed to encapsulate this probability selection process.

Palette Identifier	Palette Description
R_GRAYSCALE	Random gray scale values
R_COLOR	Random color values
R_COLOR_FROM_IMAGE	Random color values extracted from a randomly selected bitmapped image
R_BLUE	Random blue color values
R_GREEN	Random green color values
R_GREEN_BLUE	Random color values with the red component set equal to zero.
R_RED	Random red color values
R_RED_BLUE	Random color values with the green component set equal to zero.
R_RED_GREEN	Random color values with the blue component set equal to zero.

Table 1: Ray tracer palettes.

From its initial use for palette selection within the Art Rays Lite app, *ProbabilitySelectors* are now being used in three other areas of the code to constrain randomness and hopefully create more pleasing images. *ProbabilitySelectors* might be useful in your code if you can formulate the problem to be solved like I did above.

Other possible uses might be in picking directories of digital photographs for display in a digital picture frame. You might want to select pictures from a family directory 60% of the time. Pictures from your pets directory 30% of the time. Pictures from your sports car directory 10% of the time.

ProbabilitySelectors might also come in handy for software testing. You might use a *ProbabilitySelector* for picking suites of tests to run and then use another *ProbabilitySelector* to pick which tests within a suite to run. Executing test suites and test cases in an unpredictable order might point out bugs in the code you might not have seen otherwise.

Using ProbabilitySelectors

Although I developed *ProbabilitySelectors* originally in Objective-C for the iPhone, I decided to present the code for this article in Java, as it might be more widely understood. Table 2 shows the Java API for the *ProbabilitySelector* class. As you can see, there are very few methods in this API. (The source code accompanying this article is available at http://i.cmpnet.com/ddj/images/article/2010/code/ProbabilitySelector.zip.)

A typical use would be as follows:

```
// Instantiate a named selector
ProbabilitySelector ps = new ProbabilitySelector("Test1
Selector");

// Initialize it by adding the required number of selectors with
their
// corresponding percentage values
ps.addSelector(SELECTOR_1, SELECTOR1_PERCENTAGE);
ps.addSelector(SELECTOR_2, SELECTOR2_PERCENTAGE);
 add as many selectors as needed
ps.addSelector(SELECTOR_N, SELECTORN_PERCENTAGE);

// Validate the ProbabilitySelector
if (! ps.validate()) {
   // Houston we have a problem.
   // Make sure all percentages add up to exactly 100%
}
```

Constructor Summary

`ProbabilitySelector(java.lang.String name)`	Create a named ProbabilitySelector instance

Method Summary

void	`addSelector(int selector, int percentage)` Add a selector with the specified value that will be returned the specified percentage of time
java.lang.String	`getName()` Return name of this probability selector as String
void	`reset()` Reset the ProbabilitySelector instance for a new run
int	`select()` Return the next selector
java.lang.String	`toString()` Show internal representation of ProbabilitySelector
boolean	`validate()` Verify the ProbabilitySelector instance is completely and accurately initialized.

Table 2: ProbabilitySelector API

```
// Call the select method over and over to return the next
selector in sequence
int selector = ps.select();
```

The code accompanying this article (available online) includes test code, which defines five selectors: 0,1,2,3,4 each with a 20% probability. In the test code, the *select()* method is called 100 times. Running this test code three times results in the following three selector output sequences. As you can see, the returned selectors are randomly intermixed and always occur in a different order. If you add up the number of times each selector occurs, it will match the percentage associated with the selector.

```
13214243101101302300443420322000203403443131121224001114141441332
04042034302303312044242011223342132

20430132224124034214424312000234104010204123440204023334124320101
01313233242444431320031111130023131

31114101430312204240404031321321412422400331124434234000112331233
24212402000140204422230130304413314
```

Using a selector to control program flow could be accomplished as follows:

```
int selector = ps.select();
switch (selector) {

case SELECTOR_1:
           do some stuff
          break;
case SELECTOR_2:
           do some different stuff
          break;

case SELECTOR_N:
           do some different stuff
          break;
}
```

It would also be possible to redefine the *addSelector* method to take a class instance implementing a Java interface instead of an integer selector so that calling the *select()* method would call a specified method directly. By doing this, the *switch* statement used in the example above could be done away with.

How They Work

You may have already guessed how *ProbabilitySelectors* are implemented, but I'll spend just a minute describing my implementation. Each *ProbabilitySelector* instance has an array of 100 integers, which are each initialized to -1 by the constructor. The *toString()* method of the class shows the following internal state:

```
<Probability Selector>
Name: Test1 Selector
Count: 0
Data:
 0 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
10 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
20 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
30 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
40 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
50 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
60 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
70 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
80 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
90 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
</Probability Selector>
```

When the *addSelector* method is called to add selectors, the value of the selector is written into the array in sequential order. So, for example, if selector 0 is set to 20%, selector 1 to 20%, and selector 3 to 30%, the *ProbabilitySelector* would look like this:

```
<Probability Selector>
Name: Test1 Selector
Count: 0
Data:
 0 - 0 0 0 0 0 0 0 0 0 0
10 - 0 0 0 0 0 0 0 0 0 0
20 - 1 1 1 1 1 1 1 1 1 1
30 - 1 1 1 1 1 1 1 1 1 1
40 - 2 2 2 2 2 2 2 2 2 2
50 - 2 2 2 2 2 2 2 2 2 2
60 - 2 2 2 2 2 2 2 2 2 2
70 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
80 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
90 - -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
</Probability Selector>
```

Once the *ProbabilitySelector* is fully initialized ,there shouldn't be any -1's left in the array. The *validate()* method checks for this, and therefore determines if the *ProbabilitySelector* is ready to be used or not.

Each time *select()* is called, a random index into the array of 100 *ints* is picked, and the entry at that index is examined. If the entry is a selector value, that is the value returned. Before doing so however, a large value (USED_TAG in the code) is added to the selector, and that is stored back into the array. This value indicates this selector has been returned previously, so it should not be returned again. On a subsequent call to *select()*, if the array entry is marked as previously returned, a new random index is generated, and that array entry is examined. After 100 calls to *select()*, all of the entries in the array have the USED_TAG value subtracted from them, marking each selector as unused and available again. This will make sense when you look at the code for the class.

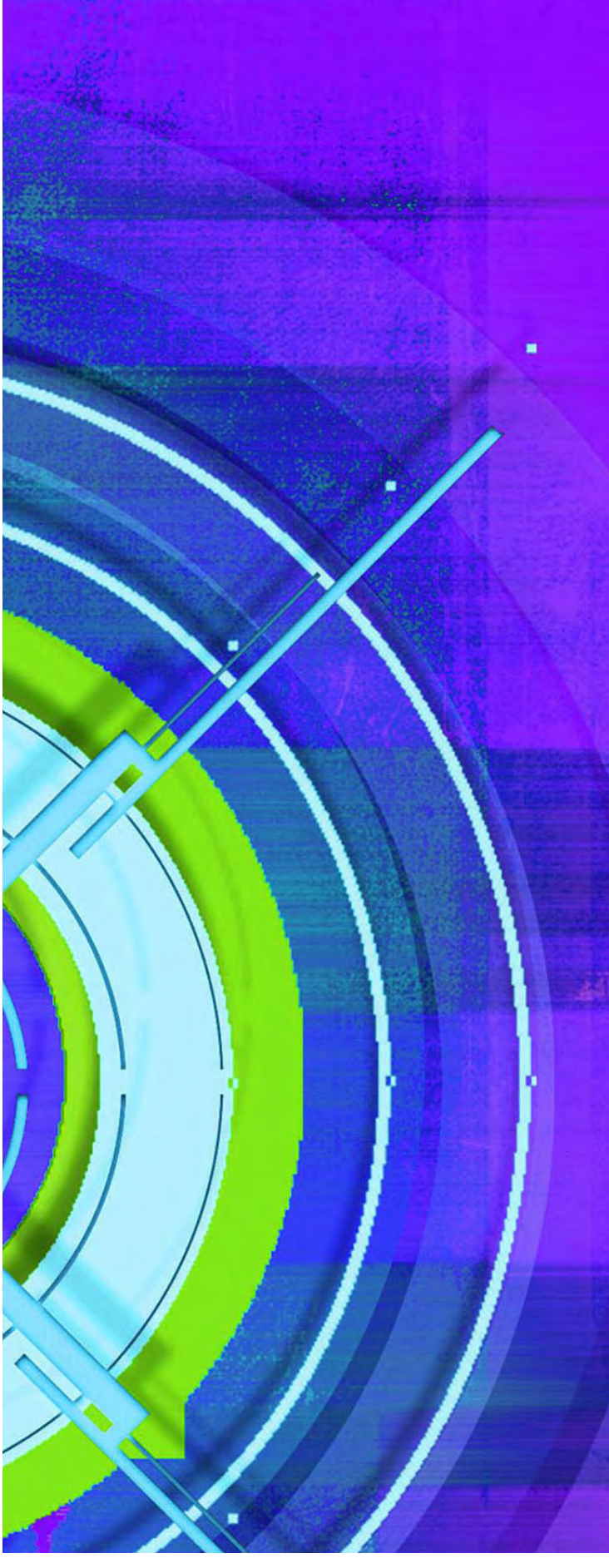
(With this implementation, it is possible that some selectors will not be returned after 100 calls to *select()*. This happens when the index of that selector's array entry is never generated by the random *int* function. This can affect the accuracy of the selector percentages, but hasn't yet been an issue in my code. Just a heads up.)

Conclusion

Only a certain class of computer problems require randomness in their solutions. For those that do, constraining randomness is sometimes necessary, and the *ProbabilitySelector* class maybe useful in those cases. I seem to find more and more uses for it all of the time. Thanks to Jerry Jackson for discussing this idea with me.

— Craig Lindley is a degreed hardware engineer who, until recently, had been writing large-scale Java applications. Now in retirement, he designs, builds, and writes about the projects he wants to. He can be contacted at calhjh@gmail.com.

Return to Table of Contents



Why Software Really Fails And What To Do About It

Software is a machine built on the principles of good machine design

By Chuck Connell

What is software? What is it about software that takes so long to create? And why does software development so often go wrong, compared to other kinds of engineering?

Along with my day job as a software consultant, I like to think about the essence of this stuff I work with every day. I wonder why it is different from other "stuff" that humans build things with, such as bricks and steel and chemicals.

Computer scientists have proposed many ways of looking at software, including as a function with inputs and an output, and as instructions for transforming computer memory from one state to another. I like to think of software, however, as a machine. It is a machine we cannot touch, but it is a machine nonetheless. (I am not the first to suggest this view.) Equivalently, we can think of software as one part of a machine that includes the computer hardware, allowing us to see software as a component of a physical machine.

When we create a new software system we are creating a new machine. The principles that apply to good machine design also apply to good software design, such as durability, maintainability, and simplicity. But, if this is true, why does software development seem harder than mechanical or structural engineering? After all, people build airplanes, bridges, skyscrapers, and factories, on time and on budget. (Not always of course, but often.) It is rare that large physical engineering projects are simply abandoned half-built, after years of effort, with millions of dollars spent, left to rust under the elements. But there are many examples of this scale of failure for software projects, including the Denver Airport (www.nytimes.com/2005/08/27/national/27denver.html?_r=1), the FBI's computer upgrade (www.washingtonpost.com/wp-dyn/content/article/2006/08/17/AR2006081701485.html), and the IRS's modernization effort (http://www.gao.gov/new.items/d06310.pdf). Many, many others were never made public because their failures were hidden by organizations not wishing to be embarrassed by the scope of their incompetence.

So what, specifically, is it about software that causes large software projects to go wrong more often than with other kinds of engineering? This question has been examined before, including in "Dreaming in Code" (www.dreamingincode.com/), "Patterns in Failure" (www.sei.cmu.edu/acquisition/research/archetypes.cfm), and "Why Software Is So Bad" (www.technologyreview.com/InfoTech/12887/). Some of the proposed answers are poor requirements documentation, and that software is just plain harder than other kinds of engineering. I believe there is another answer, however, which has not been stated clearly.

The problem is not the nature of software; software is just a type of machine and is amenable to known methods for good machine design and project management. The problem is the way we approach software projects and our expectations for their outcomes. In short, we expect too much. Too often we try to "invent the world" with a new software project, instead of relying on well-known designs and methods that are likely to succeed.

Consider this fictional mechanical engineering project that is run like many software projects.

> *The motivation for this project is that cars are a very poor form of transportation for individual people. This has been widely recognized for a long time. We want a smaller, lighter, cleaner, less expensive device for personal, local transportation.*
>
> *We will call the new invention "Personal Transportation Device 1000", stating its intended selling price in US Dollars. For individual commuting and errands within 50 miles, we want PTD-1000 to make the current automobile obsolete. We do not want the*

device to use existing, already congested, roads so PTD-1000 will fly.

Our goals include low fuel cost and no pollution, so the motor will be powered by helium fusion. Fusion is an emerging standard, but we believe that this project will provide synergy to fusion research, both driving the research and serving as a test bed for it.

We want the device to be light, which will contribute to efficiency and allow the single user to pick it up, so we will construct PTD-1000 primarily from Rearden Metal (http://en.wikibooks.org/wiki/Atlas_Shrugged/Technology). The design team recognizes that the formulation for this metal is not yet finished, so we will assign our A-team of engineers to finish this work in parallel with the other subsystems.

The final key design criterion is that a person who is physically disabled must be able to enter a building after getting there. So PTD-1000 will convert to a wheelchair for use indoors.

The participants in this project all understand that it is a substantial undertaking, but enthusiasm is high for the benefits that will be realized at its completion. There is buy-in by all stakeholders. The investors and engineers have committed to a budget and a completion date of Q4 2011. Everyone has agreed to forgo their vacations for the next year in order to meet this schedule.

Of course, this fictional invention is absurd. Anyone with common sense can see that it will fail dramatically. But take the above scenario, and substitute software concepts for the physical details, and you have an accurate description of many real software projects.

In contrast to software projects, traditional engineering relies on tried-and-true materials and methods, applied in well-understood ways to new applications. An aeronautical engineer may design a new aircraft that does not look or function exactly like any previous plane, but the structural members, outer skin, wiring components, and engines usually are copies of proven parts and techniques. Manufacturing plants, skyscrapers, roads, and bridges are built with a similar philosophy — substantial reuse of known pieces, put together largely in known ways.

Traditional engineers, of course, sometimes advance the state of their art, by using new materials or techniques or both. The Burj Khalifa recently set a new height record for buildings. The engineers did this partly by employing a special formulation of concrete (http://www.concreteconstructiononline.com/industry-news.asp?sectionID=0&articleID=651219&artnum=1) to achieve the weight-bearing strength required, and by pouring the concrete at night to aid in proper hardening. But these innovations are in tightly constrained situations, making incremental improvements to well-studied topics. Concrete has been around for thousands of years (see http://www.concretenetwork.com/concrete-history/linear-timeline.html) in some format and 200+ years in its modern version. New bridges similarly use known methods, with possibly some incremental innovations to solve special problems for that location. The public would not have it any other way.

Software projects often take the opposite approach, attempting to use many radical new materials (data structures) operated on by grossly unproven methods (programming code) to produce a machine that performs a never-before-seen function. Our large software projects often resemble precisely the sort of absurd dream invention described above. Why do we do this? The reason is that since we cannot see or touch software, it appears that all the parts of software, tested or novel, are quite similar and can be used equally well as machine subassemblies. In fact, software components do not all have the same approximate reliability, not by a long shot. We fail to appreciate fully that well-tested software modules can have a high probability of successful reuse, while novel software components are often a crap-shoot.

When we visualize software as a machine, it becomes clear just how unwise it is to invent too much in a new software system. Picture the overall software as a factory assembly line of robots, or a new kind of automobile. The major software modules are sections of the factory, or important pieces of the automobile. The software subroutines are parts making up the larger mechanical components. Individual lines of source code are single pieces of metal in a robot, or springs, or gears, or levers. Function parameters are rods or lasers reaching into another mechanical subassembly. When the assembly line or car is started for the first time, the parts may not work together correctly. They may rub or bang into each other, preventing the whole machine from working right. This might occur in hundreds of places. Some problems may not be seen until a certain sequence of actions is attempted at the same time.

In the same way, a large software project is an incredibly complex machine, with millions of possible interactions among overlapping parts, compounded by interactions of the interactions. The full behavior of many software systems is well beyond human understanding. This is why we cannot accurately predict bugs in complex software; we are trying to build machines we cannot comprehend.

Oversimplifying a bit, there are two common approaches to software projects.

1. Design and build software in a conservative manner, using tried-and-true components, assembled by a stable team of engineers, who have successfully built similar systems. These projects usually can be estimated accurately, and completed on time and budget.
2. Attempt to create software that is substantially new. These are really research projects, not engineering endeavors. They have uncertain outcomes and no reliable time/cost estimates.

An example of #1 is the creation of a new compiler by a software development company that has produced many compilers for dozens of languages and target machines. When this company takes on a new compiler project, for a variation of an existing source language, with a carefully specified target instruction set, by an experienced team of compiler engineers, then this project has a high likelihood of success. Techniques such as reusable class libraries and design patterns help software projects conform to this model.

An example of #2 was the FBI's Virtual Case File, previously mentioned. No one had ever created a software system to perform the functions envisioned for it. Creating it was like trying to construct a wholly new type of machine, from a new kind of metal, using a yet-to-be-invented welding technique.

Either of these two approaches to software is valid. The key problem is that we take on projects like #2, but pretend they are like #1. This is what ails the world of software development.

"The solution is to get over our hubris that software development is some special kind of animal"

We fool ourselves about how well we understand the complex new software machines we are trying to build. Just because we plan to code a new project in a known programming language, say Java, and our engineers are good at Java, this does not mean we have answers to all the challenges that will arise in the project. Using the mechanical analogy, just because our inventors have put together many machines that use springs and gear and levers, does not mean we can correctly build any machine using these parts. We can't have it both ways. If we want an accurate budget and completion time, we cannot engage in significant research during a software project. Conversely, there is nothing wrong with research and trial-and-error, but we should not think we know when it will be finished.

But what is the solution in the real world? Everyone would like to make software engineering as predictable as traditional engineering. There are many important pending software projects with large unknowns. We cannot simply say, "Oh, this software poses some new challenges, so let's give up." The solution is to get over our hubris that software development is some special kind of animal, unlike other engineering endeavors, and that we programmers are so much smarter than our traditional engineering brethren. Software is just a machine, and people have been building machines for a very long time.

To wit, here is my prescription for improving the success rate and reputation of software developers…

- Stop fooling ourselves about how much we know and how clever we are. Large software projects are impossible to understand fully. No one can grasp all of the overlapping effects of each component. Picturing software as a physical machine helps to illuminate just how complex these systems are.

  In a large software project, there may be one person who fully understands each particular component, but that is a different person for each component. No one has a grasp of the whole system, and we have no way to meld isolated individual knowledge into a collective whole. This was precisely the problem with the embarrassing tale of the Metric-English measurement error on the Mars Climate Orbiter (http://oig.nasa.gov/old/inspections_assessments/g-00-021.pdf). (Wasting $300 million in taxpayer money.)
- Incremental improvement to existing systems is good. Sometimes this means adding one additional feature to a working system. Sometimes it means combining two working systems with a new interface. And it is helpful if the interface method itself has been used elsewhere.
- Iterative development is good. This applies the above principle, again and again, to one particular software system. The first release of the software does little more than say "Hello" to the user. The next release adds one basic feature. The next, one more, etc. The idea is that each software release only has one major problem to solve. If it does not work, there is one thing to fix. Each release is an incremental improvement to working software. In practice, of course, we may stretch a bit and include a few new features in each release, but we never attempt to create a huge, complex piece of software all at once. (See the Agile Manifesto; http://agilemanifesto.org/principles.html.) The iterative approach also can be applied to estimating the size of software projects (http://www.stevemcconnell.com/est.htm).
- Research projects are great, but be honest about them. The Denver Airport software disaster, cited above, could have been avoided if it were handled in this way…

  Admit that we don't know how to sort airline baggage automatically, but would like to solve this problem. Start such a research project in an empty warehouse, using a few conveyor belts, some bar-coded suitcases, and some sorting gates with embedded software. After working out the kinks, try the system at a small airport, for incoming flights from one other city. When that works, try it for all flights to this small airport. After success there, and further hardware/software refinements, create a similar system at a mid-size airport. Improve the hardware/software again. Install at several mid-size airports and fix any problems.

  Then you are finally in a position to say, "Let's think about handling baggage at a large airport this way."

Software development need not be a mystical process, undertaken only by the most brilliant, with no hope of predicting the outcome. Software is a machine, and over many years we have learned the principles of good machine design. Unfortunately, because software is so new and is impossible to see or touch, we get clouds in our eyes when we think about software projects. We forget that we know how to plan, design, and construct high-quality machines — by incremental improvement to previous machines, using proven materials and methods.

—Chuck Connell is a software consultant and writer in Bedford, MA. He can be reached at www.BeautifulSoftware.com.

Return to Table of Contents

Making it Big in Software Book Review

Reviewed by Mike Riley

Making it Big in Software: Get the Job. Work the Org. Become Great.
Sam Lightstone
Prentice Hall
$24.99

After reviewing the less-than-stellar *97 Things Every Programmer Should Know* (http://www.drdobbs.com/blog/archives/2010/03/97_things_every.html), I was skeptical about reviewing yet another book that interviewed developers for their professional wisdom. However, unlike *97 Things*, *Making it Big in Software* takes a different approach with a much more recognizable cast. Is it worth your while? Read on to find out.

Unlike the overpriced *97 Things, Making it Big* is sold at lower cost and has a higher page count. More importantly, the people interviewed in the book are rock stars of the computer software industry. People like Steve Wozniak, Linus Torvalds, Bjarne Stroustrup, Ray Tomlinson, and Robert Kahn are just a few of the names lucky author Sam Lightstone somehow managed to interview for the book. Given the high profile of these characters, pages could have been wasted on praise for these noteworthy figures. Instead, the author asks many of the same insightful questions I would have asked these legendary people and collected meaningful responses. In addition to having access to such recognizable powerhouse personalities, the author brings his own street cred to the conversations. Sam works for IBM's Software Group, is the founder of the IEEE Data Engineering Workgroup on Self Managing Databases Systems, and has worked on software projects big and small.

Each chapter in the book follows the formula of the author presenting a 3–4 page essay on topics ranging from work ethic and team-building to customer management and work-life issues. These write-ups are followed by an interview with one of the 17 people highlighted in the book. Similar questions are asked of each person, such as what first ignited their passion in the software development profession, how do they manage work-life balance, what do they predict the software landscape will look like in 10 to 15 years, and so on. While Google is the company that gets the most ink (two of the interviewees are Googlers, and their services are mentioned by name by others in the book), Microsoft, Adobe, IBM, and Oracle receive an ample amount of representation in the pages as well.

Having heard several of these people speak at conferences or interviewed on technology podcasts, their voices have been accurately captured in print. Ego flexing does exist for some of the multi-degreed and/or CEO-types, but a majority of the participants remain humbly self-deprecating. All of them emphasize the importance of loving your work, working harder than anyone else you know, and knowing how to effectively communicate your ideas to a cohesive group ready to help make your vision a reality.

The author accurately concludes that the road to software development career success is attained by following mythologist Joseph Campbell's advice to "Follow your bliss" (http://www.jcf.org/new/index.php?categoryid=31). Journalist Bill Moyers asked Mr. Campbell over 20 years ago, "What happens when you follow your bliss?" Joe stated the obvious — "You come to bliss." For me, reading this book was a blissful experience from cover to cover.

Return to Table of Contents

Q&A: It's A Mobile World

The more open source, the better off we'll be

By Jonathan Erickson

Vincent Ricciardi is the technical lead at Vokal Interactive, a mobile application development shop. He recently spoke with Dr. Dobb's editor-in-chief Jonathan Erickson.

Dr. Dobb's: Are all mobile apps the same?

Ricciardi: There are two types of mobile apps: "ground apps" and "cloud apps." Ground apps never communicate over the cell network to retrieve or manipulate data. Cloud apps, which we focus on, connect to remote sources of data for the content and configuration of the application. The development challenges of cloud apps relate to the density of the technology stack necessary to support them. Data sources hosted remotely require a host, server-side programming knowledge, relational database management system knowledge, web-service development knowledge, and server administration knowledge. On top of this is the client-side component that has an application and database tier itself. Throw in a UI, and you have a 4-tier application.

Dr. Dobb's:: Are tools emerging that make iPhone application development easier?

Ricciardi: Yes, Joe Hewitt's Three20 library (Facebook App) has been a great resource for the iPhone development community. The Three20 library is loaded with sample code and an API of resources that quickly equip an individual developer with powerful tools, UI and otherwise. The more open source code out there, the easier app development will get. Also, the vast network of forums and iPhone dev communities helps support a shared knowledge bank with regards to debugging and optimizing code.

Dr. Dobb's: Is cross-platform possible?

Ricciardi: Cross-platform competency, strategy, and development are vital to a successful mobile strategy. The iPhone is cool, but it doesn't have the lion's share of the market just yet. Blackberry and the Android OS are big contenders in the market. Vokal has successfully launched cross-platform Apps for iPhone and BlackBerry and is looking forward to our first wave of Android Apps, so this is not just a notion I play with, it is an approach I advocate to each of our customers.

Dr. Dobb's: How has mobile changed IT?

Ricciardi: For one thing, mobile has introduced heightened security risks. From the company's standpoint, highly sensitive intellectual property is now floating around, constantly under the risk of being lost or stolen. From a productivity standpoint, at Vokal we have noticed that having an entire team deployed on a device like the iPhone allows us to stay in close communication no matter our location or time of day. This is but one example of how mobile has liberated and empowered today's workforce.

Return to Table of Contents

DevOps — Why Now?

Tackling the development vs. operations bottleneck

By Jake Sorofman

There's tremendous passion surrounding the burgeoning DevOps movement (http://dev2ops.org/blog/2010/2/22/what-is-devops.html), which aspires to address the longstanding bottleneck between development and operations. It's no surprise, because the cause is worthwhile: This bottleneck is the primary barrier to agility, responsiveness, and delivering business value.

DevOps is a reaction against the dysfunction in most enterprises today: Code is "thrown over the wall," and IT is left to figure out how to construct, provision, and maintain resulting systems. Dependencies are poorly documented, systems often fall over at deployment, and change is avoided at all costs. In the meantime, deadlines are missed, lines of business wait, and blame is assigned:

> *IT operations: "The application folks handed me a poorly documented pile of goo that has dependencies beyond our supported OS and middleware platforms."*
> *App dev: "The platform version we built this against supported all of our dependencies. IT never makes us aware of platform changes. It's their issue."*

But more than a few have raised the question: How in the world is this new? It's a fair question; dev and ops have long been divided by conflicting cultures, goals, processes, and tools. And the motivation to bridge this gap is far from new. What is new is an organized movement to shine a light on the problem space and emerging solution patterns. A community is finally organizing around the issue.

But why now? Because the time is right — for several reasons:

- Lean mandates. Last year's crushing recession forced us to do more with less. Automation became the necessary alternative to headcount. It also made us smarter and more efficient at scaling without adding people. This is one of the reasons that economists are predicting a "jobless recovery" in 2010. Automation remains necessary and very much in vogue.
- The need for speed. IT as competitive advantage is nothing new, but many companies are realizing that the velocity created by Agile hits a brick wall when it comes time to deploy an application — which is when business value is realized. That's why agility needs to be extended into the operational context. At the same time, emerging self-service and elastic computing initiatives dictate the need for zero-latency provisioning — automated, policy-driven and conflict-free.
- Crushing scale. IT is staring down a tidal wave of scale as systems transition from physical to virtual to cloud. What they're realizing is that these new architectures are a cap ex boon and an op ex bust — every dime of ROI realized could be washed out by attendant growth in op ex costs as the management burden is shifted to these new architectures, which promise compounding growth in system volume.

The DevOps future depends on:

- A version-controlled software library, which ensures all system artifacts are well defined, consistently shared, and up-to-date across the release lifecycle. Development and QA organizations draw from the same platform version, and production groups deploy the exact same version that has been certified by QA.
- Deeply modeled systems where a versioned system manifest describes all of the components, policies, and dependencies related to a software system, making it simple to reproduce a system on demand or to introduce change without conflicts.
- Automation of manual tasks taking the manual effort out of processes like dependency discovery and resolution, system construction, provisioning, update, and rollback. Automation — not hoards of people — becomes the basis for command and control of high-velocity, conflict-free, and massive-scale system administration.

And all of these pressures converge on the gap between dev and ops. So, is the pain new? No, but it's more acute than ever and it's only getting worse. That's why the time is right for DevOps.

— Jake Sorofman is chief marketing officer for rPath and can be contacted jsorofman@rpath.com.

Return to Table of Contents

Application Approval: A Scary Trend?

By Eric J. Bruno

When Apple launched the iPhone with AT&T as its partner, it later introduced the iPhone SDK and the application store. However, as you probably know, applications must be submitted to Apple for approval before being allowed in the application store. This means you, as a developer, need to risk non-approval based on Apple's nebulous approval process. It also means that you, as an iPhone user, may not have access to certain useful applications because Apple didn't approve it. Some applications that were not approved include those that had questionable content, and others that were deemed competitive to existing iPhone applications. The first reason makes some sense, but the second reason is quite scary.

Apple's justification for the approval process was to protect AT&T's network from errant applications. That seemed reasonable enough, and although there were those who complained about being disapproved, most have been tolerant of this requirement. However, the application store and its approval process seems to still exist even for the iPad. Why would it be needed, especially for the WIFI model? The WIFI model really is just a computer, like a MacBook without a keyboard. For that matter, so is an iPod Touch — there's no cell carrier's network to protect in either case.

This is a concerning trend, where application approval by Apple on its products has grown beyond the reasoning to protect the iPhone carrier's cell network. What's next, application approval for all Apple products, including its Macs? Although it's intriguing to think that all applications must meet some sort of minimal set of requirements and testing before you're ever exposed to them (i.e. eliminating potential malware), it's concerning that a single company could control this. Personally, even if it were a consortium of independent parties running the application approval process, I'd still be somewhat concerned. However, such a process could be used to stamp an application as "Tested and Approved", knowing that it passed a set of tests that check for things such as the presence of malware. But I still prefer a free market for third-party applications on the computers (of all shapes and sizes) that I buy.

I'm curious to see how the approval process plays out for Apple products like the iPad. It can either turn into a concerning trend, or a process to test and label applications as meeting a level of quality. As owners and users of these products, it's my opinion that we need to fight for a free and open application marketplace.

Happy coding!

CLICK HERE TO COMMENT ON THIS POST
(http://www.drdobbs.com/blog/archives/2010/03/application_app.html)

Return to Table of Contents