United Business Media

Dr. Dobb's DIGEST

The Art and Business of Software Development December 2010

Agile Documentation

By Deirdre Blake,
Managing Editor

What are the benefits and risks of creating deliverable documentation continuously throughout your project's lifecycle? Scott Ambler addresses the issue in his most recent Agile Update (see http://www.drdobbs.com/architecture-and-design/228300465), and there are a few key factors to consider.

As Scott notes, "By writing the documentation in sync with the rest of the solution, you know you have sufficient documentation to support what you've built to date. The implication is that your solution will in fact be potentially shippable at the end of each iteration." But there are caveats: "Unfortunately, adoption of a continuous approach to documentation potentially increases three project risks: Financial risk, schedule risk, and accuracy risk," according to Scott. Check out the the complete story (you can comment at http://www.drdobbs.com/architecture-and-design/228300465) and let us know if you agree with Scott's assessment.

And while you're at it, please take five minutes to complete the December 2010 edition of Scott Ambler's "State of the IT Union" Survey (the URL is http://www.surveymonkey.com/s/StateOfITUnion); the goal which is to find out what IT professionals are actually doing in practice. It should take you less than 5 minutes to complete and your privacy will be completely protected.

At the end of the survey, you will be given the chance to be entered into a draw for one of ten copies of *Glitch: The Hidden Impact of Faulty Software* by Jeff Papows, which is published by Pearson. This book is a business technology expose that examines the technical, business, and consumer impact of software glitches that continue to erode brands, consumer confidence, productivity, and profitability.

The results of this survey will be summarized in a forthcoming Agile Update. Furthermore, this is an open survey, so the source data (without identifying information to protect your privacy), a summary slide deck, and the original source questions will be posted at www.ambysoft.com/surveys/ so that others may analyze the data for their own purposes. Data from previous surveys have been used by university students and professors for their research papers, and hopefully the same will be true of the data from this survey. The results from several other surveys are already posted there, so please feel free to take advantage of this resource.

Return to Table of Contents

A How-To Primer on Mashing-Up Web Data

The Rensselaer team shows how to analyze raw government data

Extracting meaning from the mounds of data available on the Web is no easy task. For example, take the case of Data.gov, where the Obama Administration has posted 272,000 or more sets of raw data from its departments, agencies, and offices to the World Wide Web. "Data.gov mandates that all information is accessible from the same place, but the data is still in a hodgepodge of different formats using differing terms, and therefore challenging at best to analyze and take advantage of," explained James Hendler (http://www.cs.rpi.edu/~hendler/), the Tetherless World Research Constellation professor of computer and cognitive science at Rensselaer Polytechnic Institute. "We are developing techniques to help people mine, mix, and mash-up this treasure trove of data, letting them find meaningful information and interconnections.

"An unfathomable amount of data resides on the Web," Hendler said. "We want to help people get as much mileage as possible out of that data and put it to work for all mankind."

The Rensselaer team has figured out how to find relationships among the literally billions of bits of government data, pulling pieces from different places on the Web, using technology that helps the computer and software understand the data, then combine it in new and imaginative ways as "mash-ups," which mix or mash data from two or more sources and present them in easy-to-use, visual forms.

By combining data from different sources, data mash-ups identify new, sometimes unexpected relationships. The approach makes it possible to put all that information buried on the Web to use and to answer myriad questions.

"We think the ability to create these kinds of mash-ups will be invaluable for students, policy makers, journalists, and many others," said Deborah McGuinness (http://www.cs.rpi.edu/~dlm/), another constellation professor in Rensselaer's Tetherless World Research Constellation. "We're working on designing simple yet robust web technologies that allow someone with absolutely no expertise in web science or semantic programming to pull together data sets from Data.gov and elsewhere and weave them together in a meaningful way."

While the Rensselaer approach makes government data more accessible and useful to the public, it also means government agencies can share information more readily.

"The inability of government agencies to exchange their data has been responsible for a lot of problems," said Hendler. "For example, the failure to detect and scuttle preparations for 9/11 and the 'underwear bomber' were both attributed in a large part to information-sharing failures." The website (http://data-gov.tw.rpi.edu/wiki) developed by Hendler, McGuinness, and Peter Fox (http://www.rpi.edu/dept/ees/people/faculty/fox.html) — the third professor in the Tetherless World Research Constellation — and students, provides stunning examples of what this approach can accomplish. It also has video presentations and step-by-step do-it-yourself tutorials for those who want to mine the treasure trove of government data for themselves.

Hendler started Rensselaer's Data-Gov project in June 2009, one month after the government launched Data.gov, when he saw the new program as an opportunity to demonstrate the value of Semantic Web languages and tools. Hendler and McGuinness are both leaders in semantic web technologies, sometimes called Web 3.0, and were two of the first researchers working in that field.

Using semantic web representations, multiple data sets can be linked even when the underlying structure, or format, is different. Once data is converted from its format to use these representations, it becomes accessible to any number of standard web technologies.

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.drdobbs.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
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 Vice President and Editorial Director

techweb™

One of the Rensselaer demonstrations (http://data-gov.tw.rpi.edu/demo/exhibit/demo-8-castnet.php) deals with data from CASTNET, the Environmental Protection Agency's Clean Air Status and Trends Network. CASTNET measures ground-level ozone and other pollutants at stations all over the country, but CASTNET doesn't give the location of the monitoring sites, only the readings from the sites. The Rensselaer team located a different data set that described the location of every site. By linking the two along with historic data from the sites, using RDF, a semantic web language, the team generated a map that combines data from all the sets and makes them easily visible.

This data presentation, or mash-up, that pairs raw data on ozone and visibility readings from the EPA site with separate geographic data on where the readings were taken had never been done before. This demo and several others developed by the Rensselaer team are now available from the official US Data.gov site (http://www.data.gov/semantic).

The aim is not to create an endless procession of mash-ups, but to provide the tools and techniques that allow users to make their own mash-ups from different sources of data, the Rensselaer researchers say. To help make this happen, Rensselaer researchers have taught a short course showing government data providers how to learn to do it themselves, allowing them to build their own data visualizations to release to the public.

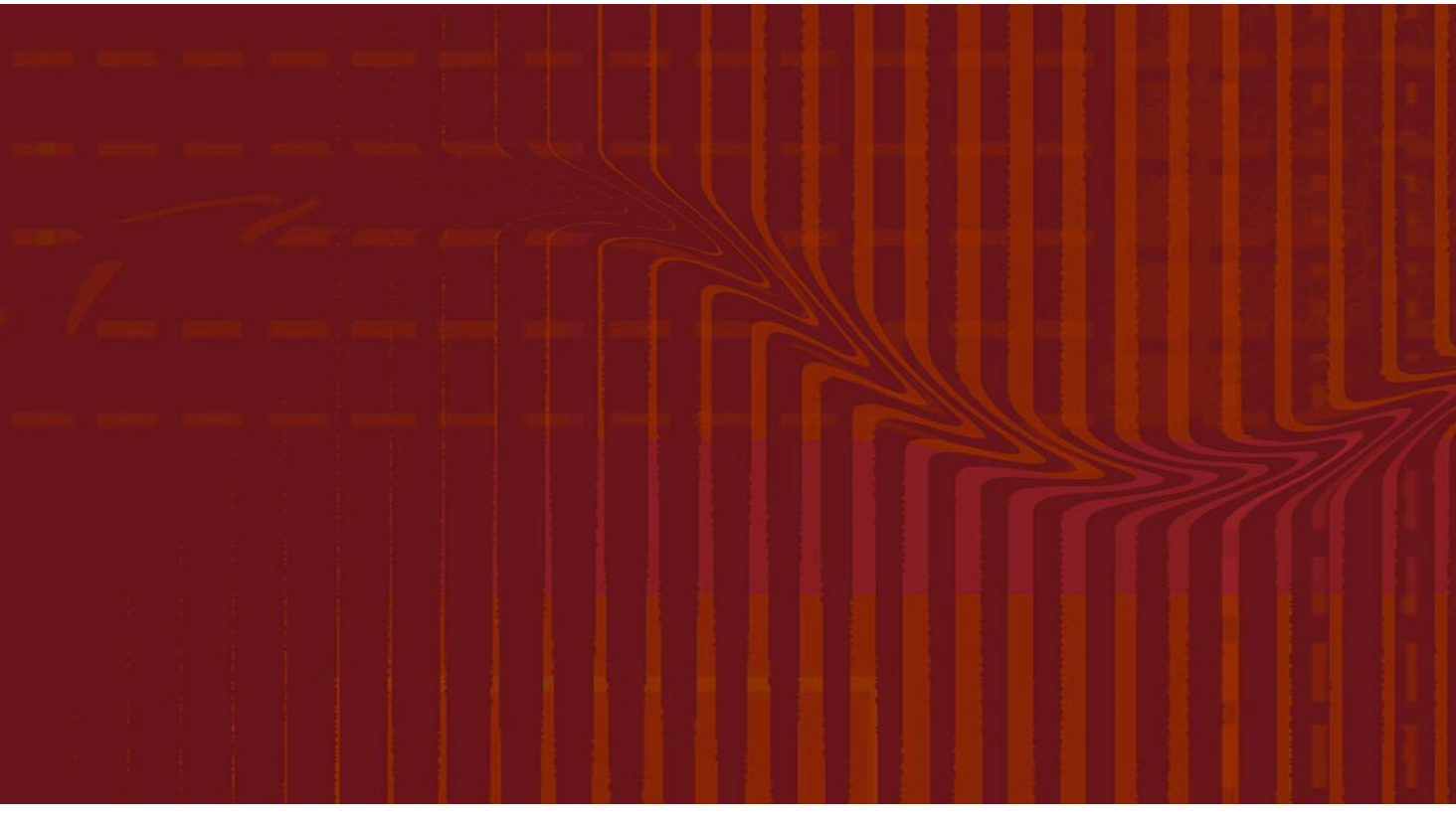
The same techniques can be applied to data from other sources. For example, public safety data can show a user which local areas are safe, where crimes are most likely to occur, accident prone intersections, proximity to hospitals, and other information that may help a decision on where to shop, where to live, even areas to avoid at night. In an effort McGuinness is leading at Rensselaer along with collaborators at NIH, the team is exploring how to make medical information accessible to both the general public and policy makers to help explore policies and their potential impact on health. For example, one may want to explore taxation or smoking policies and smoking prevalence and related health costs.

The semantic web describes techniques that allow computers to understand the meaning, or "semantics," of information so that it can find and combine information, and present it in usable form.

"Computers don't understand; they just store and retrieve," explained Hendler. "Our approach makes it possible to do a targeted search and make sense of the data, not just using keywords. This next version of the Web is smarter. We want to be sure electronic information is increasingly useful and available."

"Also, we want to make the information transparent and accountable," added McGuinness. "Users should have access to the meta data — the data describing where the data came from and how and when it was derived — as well as the base information so that end users can make better informed decisions about when to rely on the information."

Return to Table of Contents



GE Healthcare Goes Agile

An imaging unit takes control of its development environment and likes the results

By Andrew Deitsch and Ross Hughes

GE Healthcare is a $17 billion business unit of General Electric, making everything from multispectral high definition CT scanners to diagnostic pharmaceutical devices. One year ago, our Imaging Solutions unit, which has 375 engineers supporting 18 products that increase clinician productivity, faced several challenges meeting commitments in this multiproduct distributed environment.

First, we struggled with the predictability of our program execution. The cycle time on projects was too long, taking from 12 to 24 months, often with significant delays. These long cycle times frequently caused the business to push to add features beyond the initial requirements, fearing that the market couldn't wait for another cycle to get those features. That, in turn, often increased a program's scope, causing further delays and increasing the cycle time even more. A longer cycle time puts a project at risk since the requirements gathered at the beginning are out of date by the time the product hits the market.

Second, our waterfall process followed the typical phased-gate approach, which begins with gathering requirements, creating a high-level design followed by detailed designs, and then creating a traceability matrix showing how those design details tie back to the system and user requirements. At that point, a formal design review occurs and once the various approvals have happened, coding begins.

Coding typically takes several months, and then we release the product into a test environment where we can collect customer feedback. This is usually the first time customers see the new product before we begin a rigorous verification and validation effort prior to release.

The challenge with this approach is that the ability to incorporate customer-requested modifications occurs so late in the cycle that any significant misses could require complete changes to the design, causing a lot of wasted time and effort, and delaying the project further.

A third challenge for Imaging Solutions' product development effort was the many artificial barriers that existed among functions, especially marketing and engineering. These barriers weren't any different than in most large organizations, but it was clear that they were becoming more problematic over time.

Agile Transition

To address these issues, early this year, Imaging Solutions replaced the waterfall software development methodology it was using with an agile initiative. We already had pockets of agile development going on within various development teams around the world, but they were run by engineering groups that only used parts of agile. They used Test-Driven Development, Continuous Integration, and ran projects in sprints, but didn't adopt other facets of the methology.

We liked the agile-based scrum approach of having the product owner as an integral part of the development team. We hoped that adopting agile would break down these barriers and get the whole business working in unison to release the right product to our customers on time. We especially liked the idea of bi-weekly sprints, where product increments were completed, and the chance to demonstrate functionality to customers at the end of the each sprint and get immediate feedback. We began by visiting colleagues at one of our joint ventures who used agile methodologies from the beginning of their development process and were having great success with it.

We sent different people a number of times to observe sprint reviews, retrospectives, and sprint planning, and to learn how they use third-party tools — like Rally's Agile ALM platform — to create a single source of record for progress and quality across

their software development teams. We also met with the quality and regulatory team to understand how it was making agile work within its Quality Management System. Those conversations got us excited, and we began to focus on getting senior leadership support. They'd seen the results of using agile development at the joint venture (especially the frequent customer feedback), and were quick to support our move. Our next step was to hire an outside agile coach, who met with the entire team to understand our products, organization, and development processes. Once he assessed our current state, he customized our scrum training. We decided to launch our move to agile with one team. Then, after that team was comfortable, roll it out to one site. And then finally, we could take it to all of our development sites globally.

The objective for our pilot was to acquire scrum experience, understand how we could apply these techniques within our larger business (such as making it work within our Quality Management System), and to build confidence among team members and leadership so that we could be successful. Everyone involved in the pilot—executive leadership, managers, marketers, developers, testers, and technical writers—was trained in the scrum methodology. We needed the whole crew on board; we didn't want this to be just an engineering effort. We staffed a strong cross-functional team for the pilot and protected it from outside distractions. We defined a manageable scope with a short time release horizon of about four months. We established clear success criteria so that we could evaluate whether we achieved our goals. Yet, the project was meaty enough that the team could learn scrum skills while delivering something meaningful to the business.

What We Learned

The pilot identified important lessons. First, we operate in a highly regulated environment so there are a number of additional quality and regulatory steps that must be completed before we can accept a "user story"— that scenario written in the business language of the user that captures what he or she wants to achieve. Therefore, our "definition of done" — that is, the list of activities that add value to the product such as unit tests, code coverage, and code reviews — turned out to be lengthy.

Our development teams need to plan for that when estimating what they accomplish in a two-week sprint. We also learned the importance of communicating, communicating, and then communicating some more. It can't be emphasized enough how important it is to make sure everyone from the CIO to the developers knows what's happening. Often people don't hear the message after the first, second, and even third time it's said. So, while it may feel repetitive, it's valuable to over communicate and keep everyone aligned. Finally, we found that we can be agile, but the rigors of being in a regulated industry require us to operate a hybrid development model with more up-front planning and post-sprint testing than would be found in a pure agile environment.

Five Lessons Learned

Be realistic: Your organization's unique needs will dictate what can be accomplished in a two-week sprint.

Over communicate: Don't assume everyone will get it the first time. They won't.

Modify: It's OK to use a hybrid approach to agile. GE Imaging Solutions needed more up-front planning and postsprint testing, for example.

Coordinate teams: They can learn from and help each other; so the closer in alignment they are, the better.

Cultural change is key: People will have problems with the changes agile brings. Identify passionate individuals and get them to help with adoption.

Following the pilot, we brought our agile coach back in to train everyone who hadn't already been trained. We formed 10 scrum teams of seven to nine people and allowed them to self organize. Even the leaders got engaged by forming their own scrum team. With more people getting involved, we needed to coordinate the various teams that were all contributing towards a common release. We instituted scrum of scrum meetings with a representative from each of the teams to coordinate activities.

We also scheduled our sprint reviews so that they're all on the same day. So now, every other Wednesday, the teams conduct their sprint reviews together in the morning; and after lunch, hold planning meetings for the upcoming sprint. This ensures shared learning among the teams and visibility into what's going on outside any one team's activities. We also found we needed to identify cross-team dependencies early in the sprint or else teams got in one another's way. Rally's Agile ALM platform provided insight into cross-team dependencies and real-time status updates. With these capabilities, we started to see teams swapping user stories and tasks. Teams that complete their own tasks early are helping ones that are slower. There's, indeed, an art to balancing the decentralized control of independent scrum teams.

Cultural changes are the hardest part of adopting agile. That's something we'd heard from others prior to jumping into the methodology, and it turned out to be true. People often find it difficult to change, and so it's important to identify change agents within the organization who are passionate and can help with the adoption.

A key aspect of the culture change is the role of managers and individual contributors on scrum teams. Managers need to avoid a command-and-control style where they're pushing work, but rather build empowered teams. Individual contributors need to start pulling work, make commitments around that work, and then be accountable to deliver on those commitments. Trust is an important part of people being comfortable enough to embrace change,

along with providing a safe environment where teams can learn, fail, and bring up issues without fear of repercussion—this is critical for success.

While we've only just begun our journey, we've seen positive results already. Getting feedback early and frequently from customers has let us prioritize features correctly and, in one example, identify a clinical workflow that we hadn't known about. We've seen much more transparency and accountability among our teams. Team ownership has increased, and scrum processes have brought the entire team, from individual contributors to leadership, together, asking the right questions.

The pilot project was delivered successfully with the correct features and functionality. The release ran over by two sprints, so we're still working on the predictability of our execution. Understanding a team's velocity and using it to predict future execution is a learning process that will take some time—and some more sprints—to perfect. However, we're making progress, and we feel that the benefits so far of our agile adoption are worth the effort. We're now beginning the next phase of our transition by rolling out scrum globally to the rest of GE Healthcare.

—Andrew Deitsch is VP and general manager for GE Healthcare IT's Imaging Solutions group. Ross Hughes is GE Healthcare IT's ScrumMaster.

Return to Table of Contents

Window's Phone 7 App Challenges

It's no more difficult than other mobile platforms, but that doesn't guarantee success

By Dino Esposito

Application developers start your engines: Microsoft's Windows Phone 7 is finally on its way. Apps for the new platform can be based on Silverlight and work like classic event-driven Windows applications, or they can be based on XNA and behave like a classic XBox game.

Each app will run in its own sandbox and interact with users through sensors such as the camera, microphone, accelerometer, and multitouch. They'll interact with the outside world via geolocation and notification services, cloud services, and custom web services.

The layout of a Phone 7 app is based on XAML markup that you create using Visual Studio or Expression Blend. The functionality of a Phone 7 application results from the combination of Silverlight native components and Phone 7-specific components and "launchers" and "choosers" to access contacts, place phone calls, send email and text messages, and launch the built-in media player, web browser, or search application.

Beneath the specific capabilities of the Phone 7 and Silverlight frameworks, you'll find a subset of the .NET Framework that supports threading, LINQ, text, networking, XML, configuration, security, and reflection. You interact with native Windows Phone 7 applications using either a launcher or chooser task.

A launcher is a fire-and-forget command that the custom application sends to the system. Typical examples: placing a phone call, sending an email, and shooting a picture.

A chooser is a step in which the custom application visually interacts with a native app. Selecting a picture and displaying a web page are examples. Invoking a launcher or a chooser deactivates your application, and tombstoning — the process where an app is terminated when the user navigates away from it — must be used to preserve state (see http://www.informationweek.com/1286/silverlight).

When it comes to tombstoning, Phone 7 lets only one application at a time take full control of the device. To preserve battery life and avoid serious performance hits, custom apps aren't allowed to stay loaded and run in the background. As soon as the user starts a new app, the current one is suspended and added to the journal of applications and pages accessible via the Back button. The application receives proper notification when it's deactivated and activated. It's the developer's responsibility to preserve any useful data when an app is suspended.

Phone 7 offers an in-memory state dictionary that will be automatically restored when an application resumes execution. This dictionary is ideal for retaining transient data. Its content, though, gets lost if the system terminates the app or if the user restarts it. For any persistent state your application may have, you must use isolated storage.

Phone 7 applications must be published to the Windows Phone Marketplace for users to buy and download them. Developers also can install a limited number of applications on a registered device directly using Visual Studio Express for Windows Phone. In January 2011, though, there may be updates to this aspect of development.

If you're coming from .NET, Phone 7 development isn't hard but requires a strong commitment. If you're new to .NET, my gut feeling is that the learning curve is no steeper than with other mobile platforms. But the success of a mobile platform doesn't come from ease of development; it can only be achieved with appealing devices, nice applications, and rich content.

— Dino Esposito is a consultant and frequent contributor to Dr. Dobb's who specializes in Windows development.

Return to Table of Contents

Programming with Reason: Why is *goto* Bad?

And do *break* and *continue* deserve the same scrutiny?

By Eric J. Bruno

I've been part of software teams and projects where the leaders start by putting together a "programming standards" document. The first entry always seems to be "don't use *goto*" if the language in question supports it. I'll start by saying this: Aside from BASIC programs I wrote in high school, I've never used the *goto* keyword in any of my code. In fact, with Java, there is no *goto* statement. Or is there?

Most programmers will agree that properly written, well-structured code doesn't require the use of *goto*, and I agree with this. The problem is, if you simply state "no *goto*" you might as well say "no *JMP* command in the generated assembly code" as well. When we write and compile code, the result is assembly/machine code that the processor can execute. Even higher level languages like Java get compiled to machine code when executed, thanks to technology such as the Hot Spot Java VM.

If you've ever written assembly code, or viewed the output of a compiler, you'll notice the *JMP* command all over the place, usually coupled with a test for some condition. This is one way that branching and code flow control is performed in the CPU. Basically, it's the same as a *goto* statement, and if you subscribe to the "no *goto*" rule, why not eliminate it also? The answer is, if you did, you probably couldn't get any of your software to run. I use this as proof that the "no *goto*" rule is flawed; we need to state something more concise.

It's More than *Goto*

I've seen a lot of code that violates "no *goto*" rule without even using *goto*. For instance, if *goto* violates the principals of structured code, does calling *return* in the middle of a method violate it also? I tend to say yes, and I try to structure my code to avoid this. Look at the following as an example:

```
private boolean processTransaction(Transaction trans) {
        // do some session security checking
        User user = trans.user;
        log("security check for user " + user.name);
        if ( user.loggedIn() == false)
           return false;

        if ( user.sessionTimeout() == true )
            return false;

        boolean success = true;

        // process transaction here

        return success;
    }
```

This method checks two things before attempting to process a transaction: First, it checks that the user is logged in; and second, it checks that the user's session hasn't timed out. If either check fails, it aborts the transaction by returning *false*, but it does so in two places in the middle of the method. I refer to this as a mid-method return (MMR) and there's something I don't like about it. For instance, it's not always obvious where the checks begin and end, and the actual processing occurs.

In this case, the code above the checks stores the user object and logs the user's name. The code after the checks does the actual work of processing the transaction, but what if another programmer needs to modify this code? He could accidentally insert some code between the *if* statements, inadvertently doing some work that shouldn't be done until after all the checks are complete. In this simple example, it would be obvious to most programmers that this is wrong;

but in a lot of code I've seen, it's easy to make this mistake. To resolve this, I propose that the following code is better:

```
private boolean processTransaction(Transaction transaction) {
        // do some session security checking
        User user = trans.user;
        log("security check for user " + user);

        boolean success = false;
        if ( user.loggedIn() ) {
            if ( ! user.sessionTimeout() ) {
                // process transaction here
            }
        }

        return success;
    }
```

It's now more obvious where the checks are complete and the real processing begins; the code actually draws your eyes to the right spot. Also, with only one line in the method that returns a value, you have less "mental accounting" to do as you read from top to bottom. As a pleasant side effect, it's easy to discover where the *success* variable is declared. (How many times have you hunted through a long method to find declarations?)

Some people don't like nesting all of the *if* statements, but I find that this can usually be limited or avoided outright by combining related checks. The following code is an example:

```
if ( user.loggedIn() && ! user.sessionTimeout() ) {
    // process transaction here
}
```

Since the user's login and session timeout status are related checks, I have no problem checking them both in the same *if* statement. If there were a requirement to also check the date (i.e. check that it's not Sunday), then I'd prefer to see that in a separate *if* statement — but that's a matter of taste.

The point is that mid-method returns can be just as bad as using a *goto* statement in your code; they disrupt code structure, flow, and readability. However, that's not where it ends.

When is *break* Broken?

The *break* keyword in languages such as C/C++ and Java accomplishes the same thing: It abnormally exits from a block of control code (i.e. a loop or *switch..case* statement). In fact, to avoid mid-method returns, you may need to place a *break* within a *while(true)* loop to escape from it. The use of *break* in these situations makes sense to me.

Another occasion where a *break* makes sense is when performance is a consideration, such as when checking for a condition within a *for* loop as this example shows:

```
private boolean loginUser(String username) {
    boolean found = false;
    for ( User user: Users )
        if ( user.username.equals(username) ) {
            found = true;
            // login user here…
            break;
        }
    }
    return found;
}
```

In my opinion, this *break* is fine, although it does upset code flow somewhat, and it can be considered as offensive as *goto* (it essentially jumps to the *return* statement in the method). However, since it does so only to avoid needless processing of all of the users in the list once the appropriate one is found, it's acceptable. Also, since the user-login processing is still done in place, and it's obvious when reading the code where that is, this use of *break* is even more acceptable.

Using more than one *break* in a loop, however, is not acceptable to me. This is just as bad as a mid-method return, as it makes the code more difficult to read and requires more analysis to determine where to make modifications to the code. Experience has shown me that in just about every case where multiple *break*s exist within a loop, the code structure can be improved to avoid this and make the code more readable.

— In his 15+ years of developing software, Eric Bruno has acted as technical advisor, chief architect, and led teams of developers. He can be contacted at www.ericbruno.com.

Return to Table of Contents

A Highly Configurable Logging Framework In C++

A flexible, highly configurable, and easily customizable logging framework

By Michael Schulze

Logging is both a crucial technique during development and a great tool for investigating problems that arise while running software on a client's system. One of the major benefits of logging is that users of software can provide helpful information to software maintainers by simply delivering the logged data — a job for which they don't need to know anything about programming languages, debuggers, and the like.

Of course, there are numerous implementations of logging, which begs the questions "So why do we need yet another logging utility?" One reason is that many logging tools suffer from drawbacks of one kind or another:

- Logging can consume a lot of resources even if it is fully disabled in release builds. Log messages are still part of the build and in case of special debug messages (e.g. for a proprietary algorithm) it enables reverse-engineering of your code. Moreover the logging framework can be ever present, due to disabling logging only at runtime.
- Preprocessor implemented logging often has side effects. For instance, if you want to call a function always and for debugging purpose you print its return value with the help of a logging macro like *LOGGING(ret=f())* the function is only called if logging is active. If the macro is empty defined to disable logging, the function will never be called. Such Heisenbugs are hard to find.
- Adding new features like different back-ends (sinks) or customized prefixes on logging messages requires careful study.

So what are the feature requirements that a basic logging framework should have?

1. Easy to use (maybe with the typical behavior of *operator<<*)
2. Easy to extend
3. Portable
4. Type-safe
5. Possible to enable/disable certain parts both at compile time and at runtime, depending on the configuration
6. Possible to be disabled at compile time, leading to zero overhead at runtime. Thus, neither code nor memory is wasted
7. Efficient and economical in terms of resource so that it can be used in resource-constrained environments (like microcontrollers)
8. Able to avoid macro-related pitfalls in logging statements

In this article, I present a flexible, highly configurable, and easily customizable logging framework that uses standard C++ in meeting these requirements. The complete source code and related files are available at http://i.cmpnet.com/ddj/images/article/2010/code/logging-cpp.zip.

The following code snippet illustrates a typical use case:

```
log::emit() << "Hello World!" << log::endl;
```

This example outputs "Hello World! with linefeed and is easy to use (Requirement #1). Can you also see similarities to *std::cout*? I use *log* instead of *std* and *emit()* instead of *cout*. *emit()* is a function returning "something," not an object like *cout*. However, it is usable with *operator<<()* for outputting logging data. Furthermore, no macro is used at the statement level (Requirement #8).

Behind the Scenes

The architecture of the framework has three parts:

- Front-end
- Core
- Back-end

You can configure and adapt each part, and the configuration determines what the framework does. My implementation uses C++ features like function overloading, free operator functions, templates, and template-specialization to name a few, and all of the logging framework functionality is encapsulated in a separate "logging" namespace, avoiding name clashes.

The front-end is the top of the framework, presenting the API. The *log* structure that provides the *emit()* functions and two enumerations is shown in Listing One.

LISTING ONE

```
struct log {

    template<typename Level>
    static inline
    typename Logger<Level>::return_type& emit () {
        return Logger<Level>::logging() << Level::level()
                                        << Level::desc();
    }

    static inline
    Logger<>::return_type& emit () {
        return Logger<>::logging() << Void::level();
    }

    enum Numerative {
        bin  =  2,   ///< switch to binary output
        oct  =  8,   ///< switch to octal output
        dec  = 10,   ///< switch to decimal output
        hex  = 16    ///< switch to hexadecimal output
    };

    enum Manipulator {
        tab  =  '\t',   ///< prints a tabulator to the output
        endl =  '\n'    ///< adds a line feed to the output
    };
};
```

All of the logging levels are not shown. However, I do present the definition of *Error* as an example that's typical of all logging levels:

```
struct Error {
    static Level::levels level () {
        return Level::error;
    }

    static const char * desc() {
        return "[ ERROR ]";
    }
};
```

Again, each *log* statement starts with the *emit()* function. *emit()* exists in two flavors — a general form and one with a template-parameter, giving the logging level (see Listing One). The template version of the *emit()* function is used this way:

```
log::emit< Error >()<<Logging an Error << log::endl;
```

The code outputs "*[ERROR] Logging an Error*" and sets the logging level to *Error* for this statement. The return type of *emit()*, directly usable with *operator<<()* for outputting logging data, is determined by the *Logger* component (part of the core). The return type depends first on the logging level, and second, on a default template parameter of the *Logger* component (not visible here; see Listing Two). The configuration of this type is key to meeting Requirements #2, #3, and #6 because, depending on how you deploy it, different behaviors are realized. Furthermore, configuring the return type can let you enhance the framework functionality (for example with coloration, as I'll show shortly). If you look carefully at Listing One, you'll see that *endl*, *tab*, and so on are defined within an enumeration — not as manipulator functions. If I use manipulator functions in the same way as in *std::cout*, then code generation always occurs because it implies taking the address of the function. Because of Requirement #6, I have avoided manipulator functions in the general design. However, users have the ability to write manipulator functions, and the core is able to execute manipulators.

Most of the core is not interesting from the concepts point of view, because there is a lot of housekeeping code (e.g. logging level treatment) and code responsible for converting numbers to a given base, pointers, and so on to character streams. However, the *Logger* component is important because of its job for determining and handling the correct return type — the arcane template argument *loggingReturnType* (Listing Two).

LISTING TWO

```
struct loggingReturnType;

template<typename Level = ::logging::Void, typename R =
          loggingReturnType>
class Logger {
    public:
        typedef R return_type;

        static return_type& logging () {
            return Obj<typename return_type::output_base_type,
                    int>::obj();
        }

    private:
        template <typename log_t, typename T>
        struct Obj {
            static return_type& obj () {
                typedef singleton<return_type> log_output;
                return log_output::instance();
            }
        };

        template <typename T>
        struct Obj<NullOutput, T> {
            static return_type& obj () {
                return *reinterpret_cast<return_type*>(0x0);
            }
        };
};
```

loggingReturnType is forward declared to allow including the framework and later defining the type by the user of the framework. Enabling this needs an indirection level via an additional

template parameter *R* that is the *loggingReturnType* by default. Through that indirection, it is possible to defer the type definition to a later point, but allowing the use of this not-yet-defined type within the *Logger* component. Later at *log* statements, when the template code is evaluated, *loggingReturnType* has to be defined.

"How do you configure and setup the framework to a working state?"

In general, the *Logger* implements a compile-time selector, using the C++ feature of partial template-specialization. The selection mechanism uses an embedded *output_base_type* definition within the *loggingReturnType* (which is ensured through its declaration) for choosing the right implementation. If the *output_base_type* is not of type *NullOutput*, a singleton of type *loggingReturnType* is constructed and returned as reference. On the reference, methods for switching levels (printing numbers or characters) can be called. If the *output_base_type* is of type *NullOutput*, a reference to an object at address zero will be returned. "What is that?" you ask. "A reference to an object that does not really exist and calling methods on it? Is this allowed?" No, calling methods on it is not legal, but like the standard says, it leads to "undefined behavior." Therefore, I do not call methods on this reference. The *NullOutput* is defined as:

```
struct NullOutput {};
```

…and it is empty; thus it is also impossible to call methods on it. However, the type (in this case the returned reference) is usable to select an implementation. Now I use an overloaded version of a free operator function.

```
template<typename T> static inline NullOutput& operator <<
          (NullOutput& no, T) { return no; }
```

The overloaded *operator<<()* has a template parameter matching on everything, meaning it catches every call having a *NullOutput* as reference, and also allows chaining of consecutive calls. Due to its definition as static inline and through its empty implementation, the compiler can fully inline the free *operator* function and omit the call as well as the generation of the function itself, allowing a full deactivation of the logging framework (Requirement #6). To meet Requirement #5 — deactivation of certain parts/levels at compile-time — I use the *NullOutput* again. A full template specialization of the *Logger* component enables the desired functionality. The following code snippet disables Info logging level output:

```
template<>
class Logger<Info, loggingReturnType> {
     public:
          typedef NullOutput return_type;
          static return_type& logging (){
               return *reinterpret_cast<return_type*>(0x0);
          }
}
```

Such code needs not be written by hand, but instead it is generated by a user-friendly macro (*LOGGING_DISABLE_LEVEL(level)*). Yes, you read it right — a macro — but this is not critical because this kind of macro is not part of a *log* statement within user code; instead, it is only part of the logging framework configuration.

The used *NullOutput* is from the architecture point of view a special back-end used for deactivation; however, one will not always want to deactivate logging. Usually, you want to output something. Hence, the consideration of the lowest part, the back-end, is still missing. In general, a back-end, or sink, defines an output device for logged data. It could be a file, a console, a serial port, or something that you imagine. For that, sinks have to provide a simple interface to be a possible back-end.

```
struct Sink {
     Sink& operator<<(const char c) {
          // implement the sink specific behavior for printing c
          return *this;
     }
};
```

A sink takes a single character and delivers a reference to itself, allowing the chaining of consecutive output actions. However, the performed action depends on the back-end's type.

Having all parts of the architecture together, one question remains: How do you configure and setup the framework to a working state? In the first instance, you have to choose what back-end you want. As an example, I show a configuration with *StdOutput* as a back-end parameterized with *std::clog* stream as output medium. Furthermore, for the example, I do not want to switch logging levels on or off at runtime, so I configure *OutputLevelSwitchDisabled*. The contained *OutputStream* is responsible for handling numbers, pointers, conversions, and so on. I provide the configuration as a *typedef*.

```
typedef OutputLevelSwitchDisabled <
             OutputStream <
                  StdOutput< ::std::clog>
             >
        > StdLogType;
```

The architecture is implemented as mixin-layers, thus it is easy to extend, allowing adding new functionality. (See "Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs" by Y. Smaragdakis and D. Batory, *ACM Transactions on Software Engineering and Methodology*, vol. 11, 2002.) The defined *StdLogType* is directly usable. However, if I would do so, Requirements #5 and #6 are not able to be met because this type is not bound to the frameworks front-end; therefore, *log::emit()* in its flavors is not configured yet. To get the conjunction, I need the following macro:

```
#define LOGGING_DEFINE_OUTPUT(BASE)                                    \
    namespace logging {                                                \
        struct loggingReturnType : public BASE {                       \
                // The provided typedef is used for compile-time       \
                // selection of different implementations of the       \
                // logging framework. Thus, it is necessary            \
                // that any output type supports this type             \
                // definition, why it is defined here.                 \
                typedef BASE    output_base_type;                      \
            };                                                         \
    }
```

that is used this way:

```
LOGGING_DEFINE_OUTPUT( StdLogType)
```

After that, the logging framework is ready for action!

— Michael Schulze is a Ph.D. candidate at Otto-von-Guericke University working on adaptable event-based middleware for resource-constraint embedded systems. He can be contacted at mschulze@ivs.cs.ovgu.de.

Return to Table of Contents

Embedding Data Visualization in Rich Internet Applications

Visualizing complex data facilitates rapid understanding — so how should you go about adding data visualization to your apps?

By Andrew Shorten

For developers, demand for applications that contain data visualization (DV) capabilities is on the rise. Gone are the days when simple tables of rows and columns met the decision-making needs of customers evaluating product options. Today, changing user expectations make Rich Internet Applications (RIAs) a requirement for delivering the types of experiences that users demand — and rich data visualization plays an integral part.

Seeing Is Understanding

Whether embedded in an enterprise or customer-facing application, DV can simplify, organize, and present complex data in graphic patterns that help people understand, absorb, and respond to information more quickly and accurately. Presented visually, map-based data can help shipping professionals manage fleets and routes more effectively or help emergency-rescue teams respond better during disasters. Business intelligence data, presented visually via dashboards, helps executives spot business opportunities or threats in just a glance. Manufacturing data can be visualized to support product configurators. Personal checking account data can be visualized for consumers to depict deposit and spending behaviors. Ultimately, as long as the data is accurate and presented in a way that makes sense to users, data visualization leads to faster, more-informed decision making.

Developers are increasingly expected to embed data visualization in existing product offerings or to develop standalone data visualization applications. To date, this has been challenging — even for experienced Java-based developers. In part, the difficulty stems from the growing number of one-off data visualization components that are proliferating all over the Internet. Most of these component libraries are built by third parties that specialize in DV, rather than application development. Because these libraries are not always built with proven applications in mind, practical use by professional developers can be questionable.

Contributing to further fragmentation, licensing models vary from library to library. Some components are commercial. Some are open source. Some are extensible and some are not. Some are supported by small contributor bases that may not be able or willing to keep pace with emerging trends. And other components do not integrate at all with existing tools, meaning that authoring DV capabilities into your application becomes a manual process. Yet mature, well-tested technology is available for developers who seek proven, integrated data visualization tools.

What to Look For

To build high-quality data visualization applications as productively and simply as possible, developers should seek DV components that are part of larger, proven frameworks that enable sophisticated integration with complex business applications.

High-Performance, Ubiquitous Runtimes

For maximum effect, it is important that DV components run in all major browsers and computing platforms, freeing developers to focus on application logic instead of cross-platform testing. A ubiquitous client runtime that resides on desktop computers and laptops, as well as mobile devices, will help achieve a "write once, run everywhere" universal reach. The aim is to have a runtime capable of supporting responsive, interactive graphics and charting — as well as displaying large amounts of data quickly — across multiple browsers, platforms, and devices.

Adobe (the company I work for) offers the Flash Platform runtimes, Flash Player, and Adobe AIR — all are cross-platform runtimes that work consistently on a variety of hardware, operating systems, and browsers. The Flash Player is installed on more than 99 percent of Internet-enabled PCs worldwide; while Adobe AIR, used to deliver RIAs outside of browsers, has been installed more than 300 million times to date.

Consistent, Familiar Programming Languages and Environments

Development time can be further accelerated through the use of languages and environments that are familiar. Programmers who work with Java, PHP, C#, and others will be most productive working in environments that follow similar modern development patterns that enable them to leverage existing skills. This simplifies and expedites ongoing maintenance, as well. With the free Adobe Flex SDK (http://www.adobe.com/products/flex/), for example, developers write object-oriented code using the ActionScript 3, MXML, and CSS languages — which are easily mastered by experienced Java, PHP, and HTML programmers.

Integrated Tooling

Many developers use the Eclipse tooling framework (http://www.eclipse.org/) for their day-to-day development activities. Leveraging a framework for data visualization that provides a robust Eclipse-based tool can be an important consideration in the choice of a data visualization solution. An Eclipse-based tool helps ensure an accelerated development cycle because it simply plugs right into the existing environment, so new visualizations can be easily added and new data sources introduced without rewriting the application. A rich charting framework should be part of the tooling, and should allow access to source code so the developer can get going quickly. It should contain an evolving and customizable library of base-level components used in most business applications. And, most important to productive development of data visualization applications, the tooling should support drag-and-drop capabilities of charting components as well as data.

An auto manufacturer, for example, might need to drag a mechanical design component onto a design surface — and then bind data to the component through a simple drag-and-drop, as well. Or a developer who builds Human Resource applications might want to customize the GUI to appear one way when an HR specialist views the data, and another way when an employee views the same data. Using professional DV tools, the developer can drag-and-drop different skins onto the same grid, enabling fast, easy customization according to end-user need.

Using my Flex example, the SDK contains hundreds of out-of-the-box components. DataGrids, Charts, Formatters, Validators, and numerous other UI controls are the building blocks for applications of all sizes. Components can be styled and skinned to fit the desired look and feel. Additionally, all Flex components are easily extensible. They can be derived to add custom functionality, behavior, or look and feel. There are also hundreds of third-party open source and commercial components available for Flex. Mapping APIs, Data Visualization libraries, and Cloud APIs provide the building blocks for assembling great applications.

A Mature, Open Source Framework

Another critical factor to consider is the strength, maturity, and openness of the framework used to build the DV application. While some commercially available frameworks are tested and mature, their component libraries may be created by a range of third-party vendors — which adds additional cost to development, and contributes to potentially fragmented vendor offerings that often cannot evolve at the pace of the associated runtimes. Instead, look for a mature, tested open source framework that provides the highest levels of advancement, support, and integration.

A good open source framework should include a library of data visualization components that can be used as is, customized, or extended at no additional cost — and that are integral to the kinds of applications being built. Third-party partners typically supplement component libraries by offering additional charting and graphic components, as well as more complex DV applications that can be integrated into the framework. In this way, the developer doesn't have to build everything from scratch, but can write custom code around grouped components as required. Helping further expedite development, a mature framework is usually supported by a community of professional developers that ensures a growing library of business-critical components as more applications are developed.

Sophisticated DV apps also include drill-down capabilities that let users dig deeply into the details. For example, a stock market RIA might visualize current trading prices on Fortune 500 companies. If an investor clicks on one company's name, a new window could pop up that graphically depicts trading prices over a period of time for that particular company. From there, the investor may be able to drill even deeper to view a graphical depiction that breaks down that company's trading values by the minute, hour, or day.

Conclusion

Leveraging a framework that offers an easy process of data binding and connecting with existing data and services can accelerate application development. A well-designed RIA can improve access to data stored within the organization, connect isolated components in an enterprise's infrastructure, and then utilize embedded DV capabilities to merge and present data — at the UI level — from disparate systems. Utilizing DV components that are ubiquitous, integrated, and written in a familiar language on a mature framework is an effective means of adding data visualization to any application development project.

— Andrew Shorten is Senior Product Manager for Flash Builder at Adobe.

Return to Table of Contents

Q&A: Scrum Success

A conversation with CollabNet's ScrumWorks expert Victor Szalvay

by Deirdre Blake

Scrum is an iterative methodology often seen in agile software development. Victor Szalvay is a Scrum project expert who leads development for CollabNet's ScrumWorks suite, which is used by more than half of the Fortune100. Szalvay joined CollabNet in February when it acquired Danube Technologies, a provider of Scrum products and services that Szalvay cofounded. *Dr. Dobb's* managing editor Deirdre Blake recently spoke with him about the state of Agile.

Dr. Dobb's: What's the most compelling reason for an organization to adopt Agile processes?

Szalvay: Agile development makes clear responsibilities associated with a product's direction and quality. Before adopting agile, there's typically a game of "hot-potato" at most organizations regarding who's responsible for schedules and the features delivered. At the same time, managerial pressure to deliver puts developers in a position to cut quality. Agile frameworks like Scrum appoint a single person who consciously makes these trade-offs, while affording development teams the ability to implement without shouldering quality decisions.

Dr. Dobb's: Agile requires effective communication to work. What are some techniques teams can use to keep everyone in sync when transitioning to a truly collaborative workflow?

Szalvay: The ideal technique is to collocate cross-functional teams in a bullpen-style environment. If collocation isn't possible, then start with cross-functional teams. This requires breaking people out of their functional silos and asking them to work as a real team — developers, quality assurance staff, documentation writers, all together. Then require that work be delivered incrementally, not just when the code is complete. The biggest obstacle to agility is a siloed organization with gated hand-offs between roles, because it makes it easy for upstream roles to rely on downstream roles for quality.

Dr. Dobb's: In what ways has the Agile methodology been affected by the shift toward cloud computing?

Szalvay: As teams and organizations start thinking Agile, they realize the value of rapid builds, continuous integration, and heavy test automation. These practices enable agility by providing a safety net of instant feedback when team members embark on radical code changes. The faster the feedback, the better, but most organizations are hardware-resource constrained. Enter cloud computing. Once there's an infrastructure in place, developers can deploy to the cloud and get feedback from test automation suites running in the cloud. And since it's on demand, resources are well utilized and costs stay reasonable.

Return to Table of Contents

Jolt Product Excellence Awards: Testing and Debugging

TestComplete 8

By Mike Riley

Congratulations to TestComplete 8 (http://www.automatedqa.com/products/testcomplete/), winner of this year's Jolt Product Excellence Award in the Testing and Debugging category.

If you're looking for an easy, script-free point-and-click test harness assembly toolkit that can be promoted to a sophisticated, flexible, script-driven system capable of running a range of application technologies through their paces, SmartBear Software's merger with AutomatedQA's TestComplete 8 is good news, and the tool should be at the top of any Windows developer's list. In addition to TestComplete's full support of Windows frameworks and libraries spanning the .NET family of languages along with Silverlight and Windows Mobile emulator support, it also can be used to test Adobe AIR, Flex, and Flash as well as Java and JavaFX applications.

All the major test types can be executed, from functional, unit, and load testing to regression, data-driven, and distributed test configurations — along with several other types. Check out the nearly 30 screencasts of TestComplete 8 in action at http://www.automatedqa.com/products/testcomplete/screencasts/ and see for yourself why TestComplete 8 deserved the top spot in this year's Jolt Award test tools category.

Return to Table of Contents

Real World Instrumentation with Python Book Review

By Mike Riley

Real World Instrumentation with Python
John M. Hughes
O'Reilly Media
$43.99

Computers are at their best when interacting with the world around them, whether that be via a network connection or a sensor or actuator of some sort. How well does this new book written by embedded systems engineer John Hughes capture the essentials of programmatic instrumentation? Read on to find out.

Nearly a quarter of the book is spent on introducing the concept of data acquisition, basic electronics, and brief tutorial chapters on learning the Python and C languages. Chapters 6 and 7 set the stage with a discussion of the essential tools and the types of physical interfaces connecting measurement and actuating devices (of which "old doesn't mean bad") to computational machines.

Things don't get really rolling until Chapter 8, aptly titled "Getting Started." The standard list of project definition, requirements, design, testing, implementing and documenting an instrumentation project are reviewed. Chapter 9 offers a quick review of basic control system concepts and how to implement these in Python. The next four chapters are really what the book is all about, from building simulators in Python to acquiring instrumentation data I/O, and reading and writing out that data to ASCII or binary files. Finally, the author instructs readers on best practices for building user interfaces in Python (via console-based text to ugly Tkinter and more attractive wxPython GUIs) to interpret and visualize all that data flow information. The book concludes with a chapter on the author's own real-world implementations via serial port and USB data conduits. Readers interested in following the author's footsteps should be prepared to spend a couple hundred dollars on a good RS-232 digital multimeter (DMM) and the LabJack U3 USB data acquisition and control devices (although given the rising popularity of USB data analysis, an evaluation of the increasingly popular Total Phase Beagle USB 480 Protocol Analyzer would have been ideal). The book contains two appendixes on FOSS resources and a list of equipment sources, respectively.

One overlooked opportunity was showing examples using the showcase of the open hardware microcontroller world, the Arduino Uno, given the number of interesting control programs that people have posted on blogs, websites like Instructables and videos of their work on YouTube. It's also too bad that the author released the book just weeks before Limor Fried, aka Ladyada of Adafruit fame, released her guide on hacking the Microsoft Kinect using Python, a Beagle USB480 logic analyzer, and a handful of open source tools and libraries.

And while the omission of Arduino examples is a downer, the fact that the author has a resume of experience that most aeronautic-oriented software developers could only dream makes him well qualified to write about the subject matter. How many other O'Reilly authors can claim to have written software for the Phoenix Mars Lander and the James Webb Space Telescope?

Overall, the book offers a self-contained foundation to learn, explore, and write applications that collect and process data from simple and not-so-simple control systems. The code is digestible, especially for the experienced C and Python programmer, and the writing is densely packed with plenty of new concepts and abbreviations. DMM, GPIB, SCPI, VISA, and VXIbus are just a sample of such, and the ones I demonstrated all came from the same page in the book!

If you have started working with or have an interest in taking the next step toward machine-driven data collection and making sense of the information that transacts between physical interfaces, *Real World Instrumentation with Python* can offer readers a great primer and a notable head start.

Return to Table of Contents

Is Your Edge Bleeding?

By Paul Kimmel

I guess it has been a while since I have been on the "bleeding edge." When everything installs using the standard configuration and you don't need technical support, then you probably aren't out on the bleeding edge. I was reminded of this recently when I purchased a 16GB Windows 7 64-bit machine and tried to add it to my Windows Server 2003 domain and install SharePoint 2010 Server on that laptop. Everything was so far from a custom installation; my edge was hemorrhaging.

By hook or crook, no less than a dozen MS support guys, and eight nights — sometimes fourteen-hour stints — could not get that laptop to join the domain, enable Outlook 2010 to find my exchange server, or permit the SharePoint server to be installed on that laptop. I am not exaggerating.

First, joining a domain is usually a 30-second process, but not with a Windows 7 64-bit laptop. Registering Outlook 2010 with an Exchange server is about a two minute process: provide the exchange server name and a user account and you are in, but not with a Windows 7 Ultimate 64-bit machine. And, configuring SharePoint 2010 Server to run on a laptop, well that takes a long time, too.

The odyssey began with adding the 64-bit Windows laptop to the domain. I received RPC failure errors, trust relationship errors, and the Active Directory computer account was just wrong. I had to use ADSIEDIT and configure the computer record by adding the *servicePrincipalName* attribute to contain the HOST and TERMSRV values with the computer name and FQDN — fully qualified domain name. (Adding and removing the computer to the domain using Active Directory still doesn't work, whether I preconfigure it on the AD server or use a change name on the laptop.)

Fixing the AD record manually permitted the laptop to log into the domain. The Microsoft techs tried disabling the firewall, turning off IPv 6 — Internet Protocol 6 with its longer IP address schemes — and close to fifty hours of other changes, none of which worked.

Microsoft Office 2010 64-bit never installed and recognized the MS Exchange server. I had to uninstall Office 2010 64-bit, install Office 2007 32-bit, register that (Outlook) with the exchange server, and then upgrade to Office 2010 32-bit. (Kind of a bummer.) Keep in mind, this was no simple choice for me. I had worked more than 50 hours with Microsoft technical support over 8 days — including Office, Exchange, Active Directory, Networking, and ISA (Internet Security and Acceleration/firewall) folks — before throwing in the towel on Office 64-bit. These guys tried everything, including changing global policy, opening ports, running diagnostics, and running a dozen knowledge base patches to no avail. Their efforts were valiant, but I got the distinct impression they were guessing, too.

Finally, just tonight—one day before Thanksgiving in the US — I got the SharePoint Server 2010 to install, but this took a lot of finagling and no less than a half dozen attempts. This article helped — http://msdn.microsoft.com/en-us/library/ee554869.aspx — but itself is incomplete.

To install SharePoint Server 2010 64-bit on a Windows Ultimate 7 64-bit machine, follow the steps in this article carefully, **very carefully** — but Step 3, item 6, which runs psconfgui.exe is going to fail a couple of times and you will need to make some changes and re-run it, too. To get the products and configuration wizard to complete successfully, you will need read the diagnostics log, Google each error, and make the necessary changes. This blog — http://vinaybhatia.blogspot.com/2010_01_01_archive.html — had all of the answers for each error and almost all of them needed to be made.

For this error "Microsoft.SharePoint.SPException: User cannot be found," make sure you are connected to a domain controller.

For this error: "System.Security.Cryptography.CryptographicException: The data is invalid" back up the registry and delete the registry key "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Shared Tools\Web Server Extensions\14.0\Secure\FarmAdmin", although since I selected a standalone installation, I am not sure why this key was present.

For this error "Failed to create sample data": apply the hotfix KB976462.

And, finally, the configuration wizard fails on step 8 of 10 with "SharePoint Products and Technologies Configuration Wizard Error : Failed to create sample" you can apply the fix at http://social.technet.microsoft.com/Forums/en/sharepointadmin/thread/4747863c-ab4f-4dda-9a79-519a8afc32a6 and run the configuration wizard one last time. (Note: even the steps at the aforementioned TechNet article aren't correct for Windows 7 Ultimate.) Listed as:

1. Click Start, point to Administrative Tools, and then click Component Services.
2. In the left pane of the Component Services snap-in, expand Component Services, and then expand Computers.
3. Right-click My Computer and then click Properties.
4. Click the MSDTC tab, and then click Security Configuration under Transaction Configuration.
5. Under Security Settings, click to select the Network DTC Access check box, if it is not already selected. Then, click to select the Allow Remote Clients check box
6. Click OK.
7. When you receive the message that states that the DTC service will be stopped and restarted, click Yes to confirm that you want to continue.
8. Click OK when you receive the message that the DTC service was restarted.
9. Click OK.

Follow these steps instead:

1. Click Start and in the Run field start typing component services and run Component Services
2. Expand Component Services|Computers|My Computer|Distributed Transaction Coordinator
3. Right click on Local DTC and select properties
4. In the Local DTC Properties dialog click Security
5. Check Network DTC Access and under Client and Administration check Allow Remote Clients
6. Click OK and let the DTC restart
7. Close the Component Services console and re-run the SharePoint configuration tool

Sometimes I wonder why big companies don't immediately upgrade to the latest and greatest product releases. Over the last two weeks, I was humbly reminded. Upgrading on the bleeding edge can be a big time suck, a huge productivity risk, and even the blogs and tech support haven't got the configuration issues worked out very well.

The bleeding edge can be a lot of fun, but if you have real work to do it can be a huge pain in the butsky. If I had to do it over again, the new-new laptop and 64-bit OS could wait six more months at least.

CLICK HERE TO COMMENT ON THIS POST (http://www.drdobbs.com/blog/archives/2010/12/is_your_edge_bl.html)

Return to Table of Contents