United Business Media

Dr. Dobb's DIGEST

The Art and Business of Software Development

October 2010

Say What You Will — In Another Language

By Jonathan Erickson,
Editor In Chief

Learning to speak a language other than English has never been high on my list. Yes, I travel a lot, but I get by. Knowing enough to say "Un grande Caffe Mocha muy caliente, por favor" at a Starbucks will get you a hot cup of coffee in a lot of places, which suited me fine until a recent conversation with Michael McCool, a founder of RapidMind and now an architect for Intel Software where he works on the Array Building Blocks Library (see http://software.intel.com/en-us/articles/intel-array-building-blocks/).

All was fine until one point in the conversation when Michael mentioned that he was teaching himself Japanese, and all of a sudden I realized that I'm a slacker, and that learning another language might be fun. But short of RosettaStone software, where should I start? Then as luck would have it, I stumbled across "Engkoo," an application out of Microsoft Research that loosely translates to "English vault" in Mandarin Chinese. Engkoo takes advantage of natural-language-processing and speech technologies to build massive sets of bilingual terms and sentences. Engkoo currently helps Chinese speakers who are learning English, but its developers think the technology could be applied to any two languages that are widespread on the Web.

"Engkoo is a new kind of language-assistance technology for Chinese people, to enable them to ultimately master English as a native speaker might," says Michael Scott (http://research.microsoft.com/en-us/people/mrscott), development lead of the Innovation Engineering group and Engkoo project manager. "It unifies human translation mined from the Web, machine translation, and a language-learning experience into one user-friendly search-and-explore interface."

But then Ming Zhou, a senior researcher and research manager of the Natural Language Computing group (http://research.microsoft.com/en-us/groups/nlc/) chimed in by pointing out that "Engkoo is not designed primarily for learning a new language from the ground up. Rather, it's designed to be an asset to those who already use or study English, such as English-as-a-second-language (ESL) students."

Drat. Engkoo doesn't appear to be my portal to other languages. Which is okay, since I enjoyed exploring it, and Chinese probably isn't where I should start learning new languages anyway.

Additionally, as many of you have pointed out over the years, I still have a lot of work to do when it comes to English. Okay, back to Strunk and White (http://en.wikipedia.org/wiki/The_Elements_of_Style).

Return to Table of Contents

Linux and Many-Core Scalability? No Problem, Say Researchers

Research suggests that Linux will keep pace with the addition of more cores

By Larry Hardesty, *MIT News Office*

To get a sense of how well Linux will run on the many-core processors of the future, a group of MIT researchers — Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich — has built a system in which eight six-core chips simulated the performance of a 48-core chip. Then, as they describe in their paper entitled "An Analysis of Linux Scalability to Many Cores" (http://pdos.csail.mit.edu/papers/linux:osdi10.pdf), they tested a battery of applications that placed heavy demands on the operating system, activating the 48 cores one by one and observing the consequences.

At some point, the addition of extra cores began slowing the system down rather than speeding it up. But that performance drag had a surprisingly simple explanation. In a multicore system, multiple cores often perform calculations that involve the same chunk of data. As long as the data is still required by some core, it shouldn't be deleted from memory. So when a core begins to work on the data, it ratchets up a counter stored at a central location, and when it finishes its task, it ratchets the counter down. The counter thus keeps a running tally of the total number of cores using the data. When the tally gets to zero, the operating system knows that it can erase the data, freeing up memory for other procedures.

As the number of cores increases, however, tasks that depend on the same data get split up into smaller and smaller chunks. The team found that the separate cores were spending so much time ratcheting the counter up and down that they weren't getting nearly enough work done. Slightly rewriting the Linux code so that each core kept a local count, which was only occasionally synchronized with those of the other cores, greatly improved the system's overall performance.

"That basically tells you how scalable things already are," says Frans Kaashoek, one of three computer-science professors who, along with four students, conducted the research. "The fact that that is the major scalability problem suggests that a lot of things already have been fixed. You could imagine much more important things to be problems, and they're not. You're down to simple reference counts. Our claim is not that our fixes are the ones that are going to make Linux more scalable," Kaashoek says. "The Linux community is completely capable of solving these problems, and they will solve them. That's our hypothesis. In fact, we don't have to do the work. They'll do it."

Kaashoek does say, however, that while the problem with the reference counter was easy to repair, it was not easy to identify. "There's a bunch of interesting research to be done on building better tools to help programmers pinpoint where the problem is," he says. "We have written a lot of little tools to help us figure out what's going on, but we'd like to make that process much more automated."

"The big question in the community is, as the number of cores on a processor goes up, will we have to completely rethink how we build operating systems," says Remzi Arpaci-Dusseau, a professor of computer science at the University of Wisconsin. "This paper is one of the first to systematically address that question."

Someday, Arpaci-Dusseau says, if the number of cores on a chip gets "significantly beyond 48," new architectures and operating systems may become necessary. But "for the next five, eight years," he says, "I think this paper answers pretty definitively that we probably don't have to completely rethink things, which is great, because it really helps direct resources and research toward more relevant problems."

Arpaci-Dusseau points out, too, that the MIT researchers "showed that finding the problems is the hard part. What that hints at for the rest of the community is that building techniques — whether they're software techniques or hardware techniques or both — that help to identify these problems is going to be a rich new area as we go off into this multicore world."

Return to Table of Contents

Dr.Dobb's

EDITOR-IN-CHIEF
Jonathan Erickson

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

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

AUDIENCE DEVELOPMENT
CIRCULATION DIRECTOR
Karen McAleer
MANAGER
John Slesinski

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

Tame The Code Beast

How one shop uses the software product line approach to churn out software more efficiently

By John D. McGregor and Peter Bel

Think about where your business was a year ago and where it is now. Whether better or worse, it's likely that your business environment has changed, and chances are those changes have altered how you think about software. Tasks that you haven't automated now have to be done so frequently that they're soaking up too much precious time. And that elegant abstraction you designed? It now takes too long to tailor for each new product.

This pretty much describes SystemsForge, a company that develops e-commerce, content management, and custom web applications — building more than 200 apps over the last four years alone, in fact. Originally written in ColdFusion, the first version of SystemsForge's software used a component-based architecture built around a content management system, with "Pages" within a "Site." When building a new Site, developers ran a wizard to generate the code forthe project, then used the content management system to add Pages with features such as product catalogs, checkout, e-mail newsletter systems, and event calendars. Each feature was actually a component — a series of interdependent scripts implementing the iFeature and iFeatureAdmin interfaces. iFeature handled runtime client requests and iFeatureAdmin provided a wizard to allow non-technical site developers to configure common options.

This approach worked well for the first few clients. When it started, the product catalog feature had 30 options and was implemented in a manageable 2,500 lines of code. But by the time SystemsForge had 40 clients, the feature catalog had more than 450 configuration settings, nearly 13,000 lines of code. That was unwieldy, didn't scale, and was littered with configuration variables, conditionals, and loops that obfuscated the intent of the code. That's when the company opted for a software product line approach.

A software product line (SPL) is an innovative approach to software development that lets organizations develop, deliver, and evolve an entire line of similar products with much higher degrees of efficiency. The key characteristic of an SPL that distinguishes it from other development efforts: the extensive reuse of core assets. Those assets — referred to as "essential practices" by the Carnegie Mellon Software Engineering Institute — include requirements, architectures, designs, source code, and test cases. SPLs even include people, who share their expertise around the assets and procedures associated with a product line.

An SPL is most often used to manage a software portfolio, but it can be extended to companies managing a portfolio of end products, like manufactured goods, where the reuse of practices as well as parts is key to lower costs. Companies such as Cummins, General Motors, Hewlett-Packard, Lockheed Martin, and Nokia have invested in using SPLs, looking for better productivity, quality, time to market, and product line scalability.

The organization that builds the products in a software product line provides more than just continuity across a set of product development projects. The organization develops a production capability that will produce the products that are within the scope of the product line. This capability is somewhat like the production capability of a manufacturer such as Toyota. Each step required to produce a product is planned before the assets are finalized, and the steps are engineered into an effective and efficient sequence. This plan is used across the set of products; however, the inherent fluidity of software requires a flexible, craftsman-like ability to customize products like the Ferrari personalization program.

The production capability of the product line is described in a production plan. The production plan coordinates the actions of those who create core assets and those who use them to create products. The plan effects this coordination by defining a production method that describes how to build a product using the core assets. Although

the production method places constraints on the technical approach to product building, a wide range of approaches has been successful. From this method, the core asset developers determine how they will design the core assets to support product building.

The choice of production method can mitigate some of the effects of change by decoupling the means of expressing concepts in the problem domain from the implementation of those concepts. A rainbow of general-purpose languages, including C++, Java, Ruby, and others, has been used successfully for their power to express abstractions. More focused frameworks, such as Grails, are particularly suitable for building the discrete assets to be assembled into products. Product line organizations have invested in domain specific languages (DSLs) because of their flexibility and expressive power and the fact that with sufficient products under one management, the cost of the language is more likely to be recouped. These DSLs target specific features of the products such as the user interface or the persistence capability.

The approaches to development and evolution of the core asset base of a software product line vary over a continuum. At one extreme, the proactive adoption approach builds a full set of reusable parts before any products are built. This is feasible when the decision to build multiple similar products is made before any of them have been built and there is a reasonable certainty of what products will constitute the product line, a specific strategic business decision. With this approach a large investment is made before any of the parts are tried out in building a product; so this approach is useful in a context where the organization has much experience in building similar products. At the other extreme, the reactive approach first builds a product and then builds reusable pieces from the pieces used to build that first product. Product builders mine the initial product for functionality that will be useful in many products and create a reusable asset for it. Reacting is appropriate when the priority of product delivery is higher than creation of the reuse infrastructure.

Whether the iterations begin before any products are built or after some products have been built, virtually all software product line organizations build the assets using an iterative process. Iteration reduces the risk of prematurely producing the wrong assets but begins to build a reuse infrastructure quickly without having to risk waiting to react to the results of the first product development effort. Both RUP and agile-based methods have been effective for guiding the iterative product development in a software product line. The manner in which releases of the core assets to the product building teams are handled has been as varied as in any development organization. Some choose to release continuously, some release on a regular, time-boxed schedule, and others release when significant new functionality or high-priority repairs are ready. The longevity of a software product line organization is a virtual guarantee that things will change along the way. Recently some software product line organizations have begun to switch to a compositional approach to product building as opposed to integrating components. Compositional techniques stress parallel development of features that are released asynchronously so that other component developers can build on top of them. This change is partly a reaction to a phenomenon that many successful product lines experience: growth. Once a software product line is proven successful, many product teams ask for their product to be added to the product line and more customers want the lower prices and quicker delivery possible with products of the product line.

Managing Complexity

Before moving to an SPL approach, the biggest problem SystemsForge had was managing the complexity of product features: There were just too many configuration options to implement them all at runtime. To tackle that complexity, SystemsForge created a software product line that used a concatenation-based code generator — meaning the code was linked together in a chain.

This approach worked all right to create multiple applications for clients with similar needs, but it wasn't flexible enough to scale to multiple clients with differing requirements. Consequently, SystemsForge decided to move its SPL to a template-based generator that promised to be more concise, more flexible, and more powerful than the concatenation-based one.

There are a number of product options to support an SPL. The SystemsForge team looked at various ones, including MetaEdit+ for building modeling environments, Microsoft's domain-specific language tools, the openArchitectureWare project (now part of Eclipse Modeling Tools), various UML-based code generators, and products such as CodeFluent, a model-driven tool for creating .NET applications that's described as a "software factory." The company also examined products such as pure::variants and BigLever, which specialize in SPL management market. In the end, SystemsForge opted to write its own code-generation framework, in ColdFusion, in less than 2,500 lines of code. It decided the benefits from control outweighed the robustness of third-party software. While it lacked the sophistication of commercial alternatives, it was a useful starting point.

Over time, the SystemsForge team raised the runtime framework's level of abstraction to replace a lot of the repetitive generated code with a runtime interpretation of XML models. It also created a standard interface to the metadata so metadata storage could be decoupled from the implementation of the generator. And finally, SystemsForge reimplemented a front-end wizard that made it possible to build applications such as e-commerce systems faster than before.

Working With Custom Code

For most SPLs, developers need to support some degree of truly custom, project-specific code. When the company built its first version of SystemsForge, before using SPL, it used a large core framework

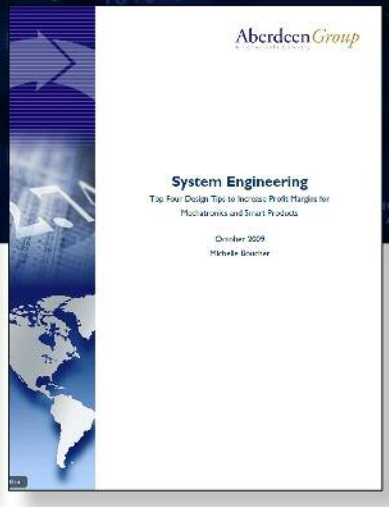
Click this button to download the Systems Content Resource Bundle!

Systems Engineering

The foundation for success in complex systems development

System Engineering

Top four design tips to increase profit margins for mechatronics and smart products



Building smart products

Best practices for multicore software development

Exclusive Downloads!

Compliments of IBM

Powered by Dr.Dobb's

with project-specific directories for assets (images, CSS, etc.) and custom code. Generally developers did ad hoc customization of features rather than having a well-structured approach to application extension.

When moving to the SPL template-based generator, SystemsForge developed a smaller framework with numerous extension points. Events/listeners let it add custom functionality such as login, *saveDomainEntity*, and before/after controller actions. Subclassing generated code let SystemsForge overload specific generated methods. Mixins, partial classes, and includes let it mix generated and handwritten code into one runtime script — while still saving it as two separate files. That allowed active regeneration without losing hand-coded changes. Generated custom tags let SystemsForge reuse complex generated logic such as list pagination and complex administrative user interface widgets in view code. Protected regions let the company customize specific parts of a view template while still allowing for active regeneration of the code as configuration options were changed.

These changes extended the viability of the product line. By having a well-structured, template-based, code-generation step with well-defined extension points, SystemsForge was able to create more than 100 applications with a code base of about 85,000 lines of code, including framework and generator code but excluding project-specific custom code. That's far less than the 140,000 lines with the concatenation- based generator. If they started with an approved HTML/CSS design, configuration would typically take one to two hours; project communications would add another four to six hours; and the custom code would take another week or two to write and test, depending on the complexity.

Where DSL Meets SPL

SystemsForge clearly had an effective system for managing bounded (limited) variability using configuration options, but some of the configuration settings they needed required a different approach. For example, the company found that different projects required products to have additional properties. In some, they might have a wholesale price; in others, they might have a bit setting for determining whether they could be shipped internationally. Eventually, more than 60 percent of the configuration options were based on these kinds of settings. Also, even at 80,000 lines of code, the code base was still expensive for a small business to maintain and evolve. In particular, making changes to underlying technologies such as the persistence framework required substantial effort.

Consequently, SystemsForge decided to fundamentally change the way that it implemented projects by using domain-specific languages to describe each application. They created languages to describe entities, properties, and their relationships, to describe validation constraints, to describe common controller actions, and to describe common view types and view widgets. Technically, this was both the most challenging and the most interesting step in the evolution of their SPL.

Systems and Software Product Line Engineering at Lockheed Martin MS2

"At Lockheed Martin, the timely and cost-effective delivery of the latest technological advances to our customers is mission critical. Our goal is to constantly 'push the envelope' in employing state-of-the-art product development tools and methods." Norman Mainak, Chief Engineer and VP of Technical Operations, Lockheed Martin MS2.

Lockheed Martin's Maritime Systems and Sensors (LM MS2) Division recently implemented a multi-year Common Product Line engineering initiative to reduce the time, cost, and effort required to create, deploy, and maintain its product lines. The new engineering approach had to minimize duplicate effort, maximize commonality among design and implementation assets, and optimize the reuse of effort across similar systems within the MS2 product lines.

As LM MS2 searched for an enhanced systems and software product line (SPL) engineering solution, it was clear that early generation SPL approaches did not enable the organization to fully integrate and strategically reuse engineering assets across the entire product development lifecycle — from requirements, to design, development, documentation, and testing. To address this challenge, LM MS2 decided to adopt an emerging second-generation SPL solution that provides a common framework and extends SPL variation management constructs across engineering artifacts in each stage of the lifecycle. This unified "Product Line Engineering" approach ensures that traceability, control, and processes flow cleanly across each stage of the lifecycle and enables LM MS2 to realize the benefits of its Common Product Line vision.

— James Cezo, Principal Engineer, Lockheed Martin

There were both benefits and costs to moving to a DSL-based approach. By using DSLs, they substantially increased the flexibility of the product line because the DSLs supported unbounded (unlimited) variability. It didn't matter what property they wanted to add to an entity or what validation rules they wanted to add; they could describe it in the DSL without increasing the complexity of the underlying framework. Also, it substantially decreased the complexity of the framework. They were able to replace an 80,000 LoC project with only 6,500 lines of code — a substantial win in terms of maintainability.

However, moving to a DSL-based approach substantially decreased development speed. For example, if they wanted to create a commerce site, instead of just selecting a handful of common configuration options in a couple of hours, developers had to spend a day or more fully describing all of the entities, properties, relationships, validation rules, controller actions, and views

required to implement an e-commerce application. This step added up to 80 percent to the time required to create a simple solution, but as the complexity of a given project increased, they often won that time back by using the DSLs to replace a lot of custom code.

To improve the speed of development for simple projects, SystemsForge then reimplemented the wizard-based configuration system on top of the new DSL-based framework. For ease of parsing, they used an XML concrete syntax for the DSLs. To create a new project, they would run the configuration wizard to generate a first cut of the DSL models quickly, and then just manually edit the XML files as the details of the requirements changed. The framework still supported the same extension points as the previous system for integrating project-specific custom code. This brought them back to the efficiency of the original system for simple projects, but added the flexibility of the more-sophisticated DSL-based solution.

Around the same time, SystemsForge ported the project from the Adobe version of ColdFusion to Railo, an open-source alternative for processing ColdFusion scripts. By moving to an open-source implementation, they had more control of the language and were able to get speedy patches and improvements.

More recently, SystemsForge ported the product line to use Groovy and Grails. Groovy is a dynamic language on the JVM that enhances developer productivity while offering interoperability with Java class files. Grails is a convention-over-configuration framework that allows for rapid development of web applications based on top of proven enterprise Java frameworks including Spring, Hibernate, and SiteMesh. The choice to rebuild in Grails was driven by overlap in the capabilities of Grails and the in-house LightBase framework, and Grails was a better solution for more complex systems.

Conclusion

SystemsForge is continuing to investigate new tools and processes. Key research areas include DSL evolution, DSL testing, and tooling that integrates SPL variability management with generation of DSL models as SystemsForge contemplates the next generation of its product line.

The changes SystemsForge implemented using a software product line approach were driven by client needs. But they were supported by extensive research into best practices in domain-specific modeling and software product lines. Change is the one thing any company can count on, and being prepared to manage change is a prime benefit of the SPL approach.

— John D. McGregor is a visiting scientist at the Carnegie Mellon Software Engineering Institute. Peter Bell is CTO of SystemsForge.

Return to Table of Contents

Quick Apps With ClickOnce

Easily distribute Windows applications over the Web

By Marcin Kawalerowicz and Craig Berntson

ClickOnce is a Microsoft technology that lets you deploy a Windows Form or Windows Presentation Foundation (WPF) application via a web page. With ClickOnce, users get to decide whether to install an application upgrade. A "smart client" application is installed in a sandbox on the client machine with fewer rights than it would otherwise get. For example, it has no access to local files, and it's installed in a user's private folder, not in the Program Files.

For example, to make the Windows Calculator available via ClickOnce, go to the *CiDotNet.WinCalc* project properties and switch to the Publish tab (Figure 1). You'll need a web server like Microsoft's Internet Information Service (IIS) to host your published apps.

Figure 1 shows a publishing target laying somewhere in the local network. In our test setup, IIS is installed on the ci1 server, sees the WinCalc folder, and is able to immediately host the WinCalc application. If you have Front Page Extensions installed on the remote IIS server you will be able to publish onto it right away.

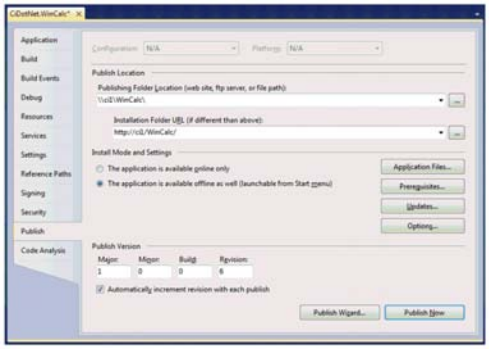


Figure 1: The Publish properties for windows application. The install is deployed to a share in the local network that is available over http.

To make the published version look nice you should define the deployment web page. Click on the Options… button. Fill in the Description, then select Deployment from the listbox (Figure 2).

It's time for a test publication. Press the Publish Now button or use the publishing wizard. Then, launch Internet Explorer and open the remote location. Be advised that Firefox and other browsers do not support ClickOnce without special plug-ins (see Figure 3.)

Now, if you click Install, WinCalc will be installed on your computer. The whole process works like a charm from Visual Studio. With a command line (and eventually Continuous Integration), it is not so easy, as you will see next.

ClickOnce in a Continuous Integration Scenario

Publishing a ClickOnce-based application in Visual Studio is very easy, as opposed to the automated deployment. But it's still tricky and, if you want to do it, you can either use a command-line tool like

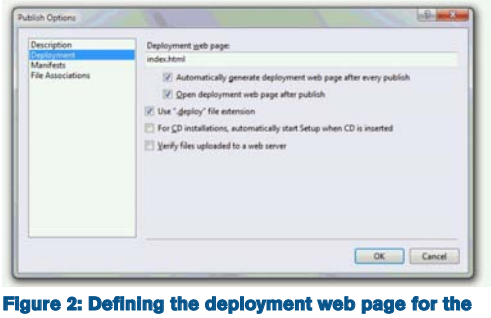


Figure 2: Defining the deployment web page for the ClickOnce deployment.

mage.exe or you can have MSBuild do the work for you. As we are the masters of MSBuild now, we will use this build tool.

You can generate the publication files using MSBuild by using the Publish target on the solution file, like this:

```
msbuild CiDotNet.sln /t:Publish
```

There are several problems with this approach:

- The version number is not incremented.
- The publication files are created in the *bin\$(Configuration)\app.publish* folder and not on the destination location.
- The website HTML file is not generated.

Let's deal with these issues one at the time. The version number could be set from outside by providing the *ApplicationVersion* property on the MSBuild command line:

```
Msbuild CiDotNet.sln /t:Publish /p:ApplicationVersion=1.0.0.3
```

But how do we get the version number on the build server? Visual Studio takes it from the *ApplicationRevision* property inside the .csproj file and it is not a good idea to mess with it on the server. But how about combining the *ApplicationVersion* number with the revision number?

First, make sure the MSBuild community tasks have been copied into the tools directory. Next, get the svn.exe and copy it to svn subdirectory in the tools folder. We'll use them both, as shown in Listing 1.

Listing 1: Publishing an application versioned with SVN revision number

```
<Project DefaultTargets="Publish"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
      <UsingTask
AssemblyFile="tools\MSBuildCommunityTasks\MSBuild.Community.Tasks.dll"
TaskName="MSBuild.Community.Tasks.Subversion.SvnInfo"></UsingTask>
     <PropertyGroup>
        <ApplicationVersion Condition=" '$(ApplicationVersion)' ==
          '' ">  1
          1.0.0.
        </ApplicationVersion>                                  1
        <RevisionNumber Condition=" '$(RevisionNumber)' == '' "> 2
          0 2
        </RevisionNumber> 2
        </PropertyGroup>     2
     <Target Name="Publish">
    <SvnInfo RepositoryPath="http://ci1/svn/CiDotNetCh10/trunk"
        Username="user" Password="password" ToolPath="tools\svn">
       <Output TaskParameter="Revision" PropertyName="RevisionNumber" /> 3
       </SvnInfo>
       <MSBuild Targets="Publish" Projects="CiDotNet.sln" 4
             ContinueOnError="false" Properties=                  4
      "ApplicationVersion=$(ApplicationVersion)$(RevisionNumber)"/> 4
    </Target>
</Project>

1 Defines the immutable version part
2 Defines the mutable version part
3 Gets the version number from Subversion
4 Generates publish files
```

After importing the *SvnInfo* MSBuild Community task we define the immutable #1 and mutable #2 version number parts. In the *Publish* target, we get the *RevisionNumber* #3 using the *SvnInfo* task and apply it to the *ApplicationVersion* property #4. Problem one, setting the version number, is now solved.

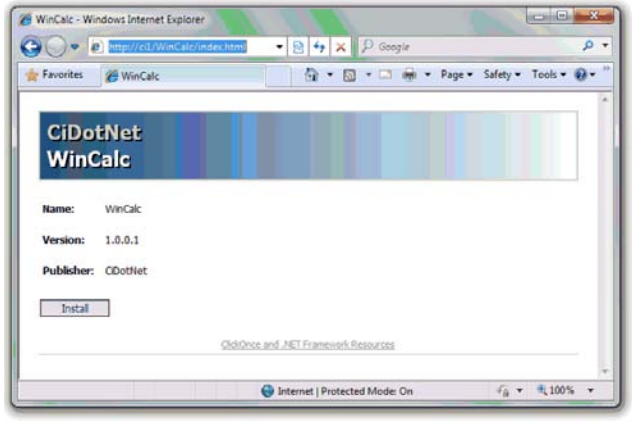


Figure 3: A published ClickOnce application on a web page.

The second problem is that the files are not copied to the target location. The ClickOnce files are created in *bin\$(Configuration)\app.publish* folder. They need to be copied to the web server. The easiest way is to define the target location like:

```
<DeploymentFolder>\\ci1\WinCalc\</DeploymentFolder>
```

Then gather the source files:

```
 <ItemGroup>
   <DeploymentSourceFiles
        Include="CiDotNet.WinCalc\bin\$(Configuration)
        \app.publish\**\*.*" />
</ItemGroup>
```

After the publishing files are created they can be copied:

```
<Copy
      SourceFiles="@(DeploymentSourceFiles)"
     DestinationFiles="@(DeploymentSourceFiles->'
[CA]$(DeploymentFolder)\%(RecursiveDir)%(Filename)%(Extension)')"/>
```

The last problem is the lack of the HTML website. There is no elegant way to deal with that. The best solution is to take the created index.html website and create a kind of template with it. Let's copy the index.html file to the solution project folder and change the current version to a string stored in the *ApplicationVersion* variable. We will use the *FileUpdate* task from the MSBuild Community Tasks. Check out Listing 2 for the file update usage.

Listing 2: Updating publication version in ClickOnce HTML self-made template file

```
<Copy SourceFiles="index.htm"
   DestinationFiles="$(DeploymentFolder)\index.htm"/>
<Copy SourceFiles="wincalc.png" 1
   DestinationFiles="$(DeploymentFolder)\wincalc.png"/> 1
<FileUpdate Files="$(DeploymentFolder)\index.htm"
    IgnoreCase="true"
    Multiline="true"
    Singleline="false"
    Regex="ApplicationVersion"
    ReplacementText="$(ApplicationVersion)$(RevisionNumber)"/> 2

1 Copy template files
2 Update version number
```

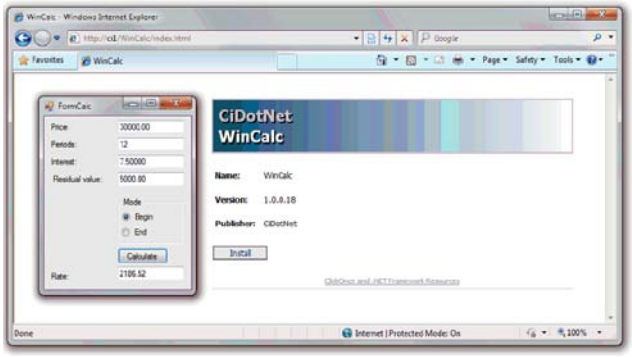


Figure 4: A customized ClickOnce website generated directly from the build server.

The use of the HTML template gives you one more opportunity to customize the ClickOnce website to suit your needs. For example, you can provide additional information to the user. First, we take the custom-made HTML "template" file together with an application screenshot and copy it to the destination folder #1. Afterward we apply the *FileUpdate* task, search for the *ApplicationVersion* string, and replace it with version #2. Don't forget to import the *FileUpdate* task from MSBuild Community Tasks. Your ClickOnce Web site could look like that in Figure 4.

You can take this build script and set it to make your Continuous Integration server use it every time something changes in the repository. You will get a brand new application deployed every time something changes in the repository.

Summary

Automating delivery and deployment and incorporating it into the Continuous Integration process is a very natural thing to do. It feels quite right to take the compiled, tested, analyzed, and documented code that looks like a delicious sweet candy and wrap it up with colorful wrapping.

The delivery and deployment scenarios depend on your individual needs. You can, for example, provision some Virtual PC setups just like the physical ones. You might want to add the automatically generated documentation to the package. You may also need to take into account laws, such as Sarbanes/Oxley (SOX) in the United States, which prohibit development from touching QA or production servers. In this case, you can use agents on QA and production servers to get the latest build. Finally, you might want to create some safety net functionality in your build script to redo the changes if something goes wrong.

— Marcin Kawalerowicz runs a consultancy in Silesia, Poland. Craig Berntson has been Microsoft MVP since 1996. They are the authors of Continuous Integration with .NET.

Return to Table of Contents

Type 5 JDBC Drivers: Welcome to the Revolution

Type 4 drivers have failed to keep up with advancements

By Mike Frost

In recent years, the Java ecosystem has evolved quite dramatically on a number of fronts. As a result, IT organizations are taking advantage of an array of new technologies that promise to streamline Java development cycles and reduce operating costs.

In complex enterprise computing environments, however, everything is connected and not all architectural components evolve in concert. Therefore, the introduction of advanced technologies at specific points in the architectural stack often exposes many downstream technological limitations.

As a result, businesses struggle to realize the return on investment promised by these new technologies and in some cases, even experience problems that take them a step backward rather than forward. Today, enterprise architects and developers experience this phenomenon when they task overmatched Type 4 JDBC database drivers with the ever-increasing demands of the modern Java environment.

The JDBC Driver: An Architectural Afterthought

The JDBC API specification and the drivers it enables have certainly evolved over time, from the original JDBC-ODBC bridge to the native-protocol Type 4 drivers that are so prevalent today. At this point, however, that evolution is stagnant. The majority of Type 4 JDBC drivers essentially offer the same basic features and capabilities, so for many developers, JDBC access is essentially an architectural afterthought that requires little attention.

This perception grew with the increasing adoption of Object-Relational Mapping (ORM) technologies (JPA, Hibernate, and Spring, among others), or application servers such as JBoss that sit on top of JDBC. With these modern development platforms, many developers no longer program directly to JDBC.

With no access to underlying JDBC calls, developers almost never think about JDBC or what JDBC driver to use as part of determining a data access strategy. Therefore, product evaluation means simply selecting a JDBC driver based on "checklisting." If there is a free Type 4 JDBC driver available, the driver is automatically assumed to be adequate for any use case.

Beyond ORM adoption, there have been other key technological advancements at work in the Java universe. Virtualization now delivers massive scalability on an affordable growth curve, placing a much higher value than ever before on optimum performance throughout the entire application stack. The increasingly complex features of relational databases frequently involve complicated and proprietary implementations that make them all but inaccessible to most applications. These strategic technologies are quickly finding acceptance in the enterprise marketplace, directing renewed attention on JDBC components, and placing a revealing spotlight on the shortcomings on Type 4 drivers as well as the negative impact these shortcomings have on IT performance and costs.

Type 4 JDBC Driver: Glaring Limitations

Despite superiority over other JDBC architecture types, Type 4 drivers have failed to keep up with the evolutionary advancement of complimentary Java technologies. As a result, most Type 4 drivers come with glaring limitations in today's Java-based enterprise application environments.

- Slow or Inconsistent Performance. The response time and data throughput performance of many Type 4 drivers is poor or inconsistent, particularly when deployed into certain runtime environments (such as different JVMs) or with modern data access models (ORMs and app servers).
- Unavailable or Inaccessible Functionality. Enabling or tuning critical functionality with many Type 4 JDBC drivers requires access to JDBC code, which is not available to applications deployed with ORM frameworks or in app

servers. New database or driver functionality may also not be available across all supported JVMs or environments.
- Poor Resource Efficiency. Most Type 4 JDBC drivers use excessive amounts of CPU and memory resources during data access. Tuning options, if available, are inaccessible or limited. This leads to excessive usage of CPU resources and a disproportionately large memory footprint.
- Application Deployment Restrictions. Most Type 4 JDBC drivers require multiple JAR files to support different JVMs or database versions. They also typically require the deployment of platform-dependent DLLs or shared libraries to support certain driver or database functionality, limiting what environments they can be deployed to.
- Proprietary Implementation. Many Type 4 JDBC drivers require proprietary code for applications to leverage features such as BLOBs and CLOBs, high availability, and XA. As more and more data sources must be accessed from a single application, the amount of application code, and potential for bugs, increases significantly.

Many developers and architects find out the hard way, after a project has gone into production, that even the most common Type 4 JDBC drivers exhibit most, if not all, of these limitations. As businesses look to new technologies such as virtualization to improve performance and reduce costs, the cost of dealing with the deficiencies of Type 4 JDBC drivers will only increase.

Unlocking the Limitations: Taking the Next Step

There is a movement afoot to address these limitations of Type 4 JDBC drivers by proposing the next step down the evolutionary path of JDBC drivers. Such a step has been dubbed "Type 5", where the problems caused by today's Type 4 JDBC drivers are solved but the benefits of Type 4 architecture are maintained. One proposed definition states that Type 5 JDBC drivers should provide the following:

- Unrestricted Performance. Should be able to easily handle the performance demands of modern data-driven enterprise Java applications.
- Codeless Enhancement. Should offer the ability to add, configure, and tune features without requiring access to or changes made to JDBC code.
- Resource Efficiency. Should use resources such as CPU and memory as efficiently as possible in multiple environments.
- All-in-One Deployment. Should be able to support multiple environments without requiring the deployment of multiple JAR files, or any native code libraries like DLLs.
- Streamlined Standardization. Should not require the use of extensions to the JDBC specification no matter what functionality is required by the application.

Type 5 JDBC Drivers: Evolution or Revolution?

In an "evolutionary" step, Type 5 JDBC drivers will build on all the positive features found with Type 4 products. Given the lack of advancement in the JDBC driver specification over the last decade, it's time that connectivity standards evolved with the myriad other advancements in the Java programming language. By delivering on the promise of unrestricted performance, codeless enhancement capabilities, optimized resource efficiency, an integrated deployment architecture, and streamlined standardization, developers and architects that use Type 5 JDBC drivers can mitigate the technical limitations imposed by outgunned Type 4 drivers that can no longer keep pace within the greater Java ecosystem.

JDBC drivers have not traditionally been viewed as a strategic investment for many businesses. Type 5 JDBC has the potential to turn that trend on its head by offering businesses an easy way to take advantage of years of innovation in database features, data access models, and virtualization technologies without requiring changes to the application. Whether the benefits sought are a simplified data connectivity infrastructure, accelerated front-line productivity, or dramatically reduced IT operational costs, businesses of all shapes and sizes should make it a priority to evaluate Type 5 JDBC drivers and continue to push for innovation over stagnation.

— Mike Frost is a Program Manager for Progress Software.

Return to Table of Contents

Autonomous User Interfaces for Mobile Apps

A new way to customize and differentiate the look, feel, and function of mobile apps

By Robi Karp

The Autonomous User Interface (AUI) is a revolutionary approach to UI design and implementation that goes beyond the custom themes, icon sets, and color schemes common on many mobile phones and other intelligent devices. Through scriptable, autonomous UI coding, AUI lets OEMs, developers, integrators, and other ecosystem participants completely control and customize the look-and-feel of the end-user experience.

With traditional design methodologies, application code "owns" the particulars of UI implementation, determining the type, orientation, placement, and other attributes of objects on the display (buttons, widgets, etc.), the flow of their use, and the callback code that powers those elements. The attributes of a UI design are thereby set in the original design and are only minimally mutable downstream, by channel partners, third-parties, and end-users. Some UI and application frameworks support theming — customization of color schemes, menu text styles, window frames, widget sets, etc. However, the fundamental structure and flow of an application UI remains set in stone — a closed box as imagined by the original design team.

On the other hand, the concept of an Autonomous User Interface lets application developers specify generic or abstract presentation of controls, widgets, and even content, giving downstream developers the freedom to brand and customize. Benefits to this approach include:

- Letting developers bind custom functionality to individual UI elements with runtime scripting
- Supporting addition and/or removal of any item from an application UI, including images, videos, and widgets, without changing any application code (i.e., with binary program images)
- Enabling existing applications to integrate reaction with new device events and capabilities, like shaking and orientation (accelerometer), location and movement (GPS), and definable data and network events (calendars, stock quotes, sports scores, wireless traffic)
- Letting integrators, operators, and end users easily add new UI personalities at runtime without changing shrink-wrap application code
- Supporting transformation of UI elements, including buttons, controls, images, and videos with 2D/3D graphics effects, to scale, reflect, rotate, alpha blend, shadow, etc. with minimum pre-load or post-load software development effort

Autonomous UI Architecture

Breaking out application and presentation code doesn't require radical rethinking of the core application design. Application code can still solicit input and generate output. It is still up to the application design team to determine how much control resides inside the application itself vs. the amount exposed to subsequent modification. But decoupling does require specific support from the underlying graphical and multimedia framework.

Key enablers of an autonomous UI include:

- Providing safe binding between underlying graphical system APIs and an external, open programming environment
- Exposing inventories of (public) application objects that implement UI functions
- Supporting a protocol between presentation code and the application for information exchange

It is also important to provide an open high-level API for developer use. The API is of course the main point of entry for developers, and also simplifies translation of information between the language bindings using the protocol. At Fluffy Spider Technologies (http://www.fluffyspider.com/), we chose C for building FancyPants (our high-performance multimedia and graphics platform for embedded applications) and for underlying libraries. For enabling

Autonomous UI code, we bind to the Lua scripting language at a high level, taking advantage of Lua features and rapid prototyping capabilities.

Lua Scripting Language

Lua is a powerful, fast, lightweight, embeddable scripting language. Lua combines simple procedural syntax with powerful data description constructs based on associative arrays and extensible semantics. Lua is dynamically typed, runs by interpreting bytecode for a register-based virtual machine, and has automatic memory management with incremental garbage collection, making it ideal for configuration, scripting, and rapid prototyping.

Real-World Examples

Most investment in differentiated user interface starts with the device home screen. Unfortunately, the investment often stops there too, providing a marginally unique look-and-feel or shallow theming of window adornment, background image and color of other elements. With AUI, device designers, their channel partners and third parties have myriad opportunities to customize the end-user experience in novel ways.

You can think of AUI in this context as an exercise in object-orient programming. The original, generic home screen is a base object inherited and extended by new objects (realized home screens) that integrate features and capabilities of specific devices and networks. A social media home screen would extend the base version to respond to status updates, messages from friends, location-based information, etc. Sports-themed home screen extensions could include live streaming video during game play, game sounds in real-time (buzzers, whistles, but no vuvuzelas, please), display of game scores and player stats, and social networking of game progress.

On a mobile handset, the device manufacturer will typically include an SMS (Short Messaging System) application, sourced from the mobile OS supplier, a third-party (ISV), or created in-house. This "preload" SMS application likely includes a traditional display of messages and addressees. A mobile network operator (MNO) or other channel partner has few options for customizing or branding this kind of application, and is often forced to pass uninspired software through to end users "as is" or invest in replacing entire preload applications at considerable effort and expense.

Using AUI, an operator could enhance SMS addressee information with status and location-based data supplied by its network, or a third-party ISV could offer an alternate look-and-feel to that same SMS client and to other AUI-enabled application code as well. For example, MNO developers could enhance addressee information with status and location-based data supplied from the operator's network. Similarly, a third-party ISV could offer an alternate look-and-feel to that same SMS client, using previously unavailable functions like accelerometer input or GPS coordinates. All without modifying any original application's code.

Depending on the industry, a new product (not just a new product version) can require between 2 to 10 or more man-years of engineering effort to reach the market. Mobile phones typically have a much quicker sales cycle and a much shorter market window (and usually involve a tremendous engineering investment). A significant part of the engineering effort lies in creating a compelling, differentiated user interface.

For OEM developers creating families of products with multiple members, being able to deploy the same application code base with a range of unique user interfaces saves time, money, and can also help focus development effort on truly differentiating features. For subsequent iterations of the same product line, an

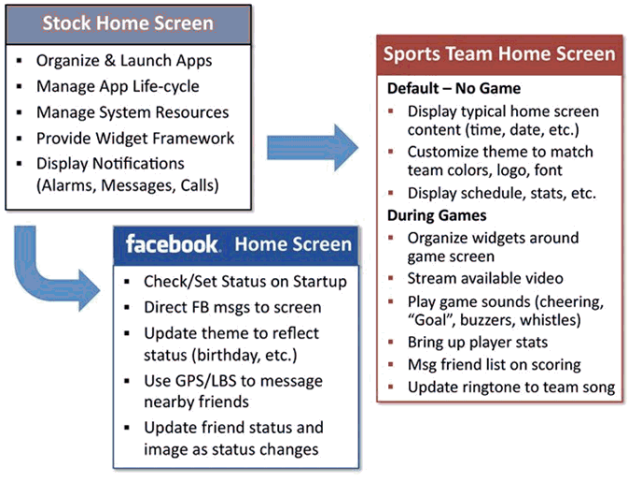


Figure 1: Adding/Extending home screen attributes.

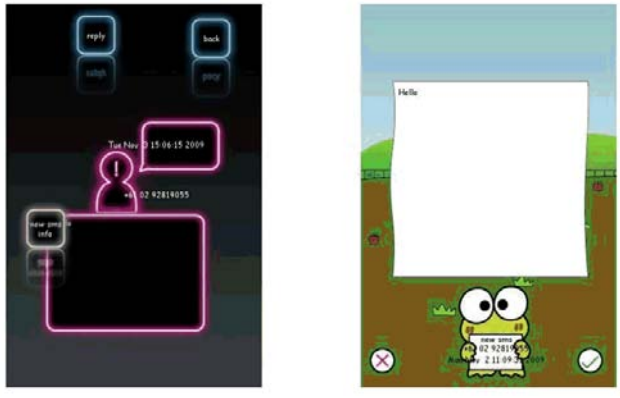


Figure 2: Two presentations of a single SMS application.

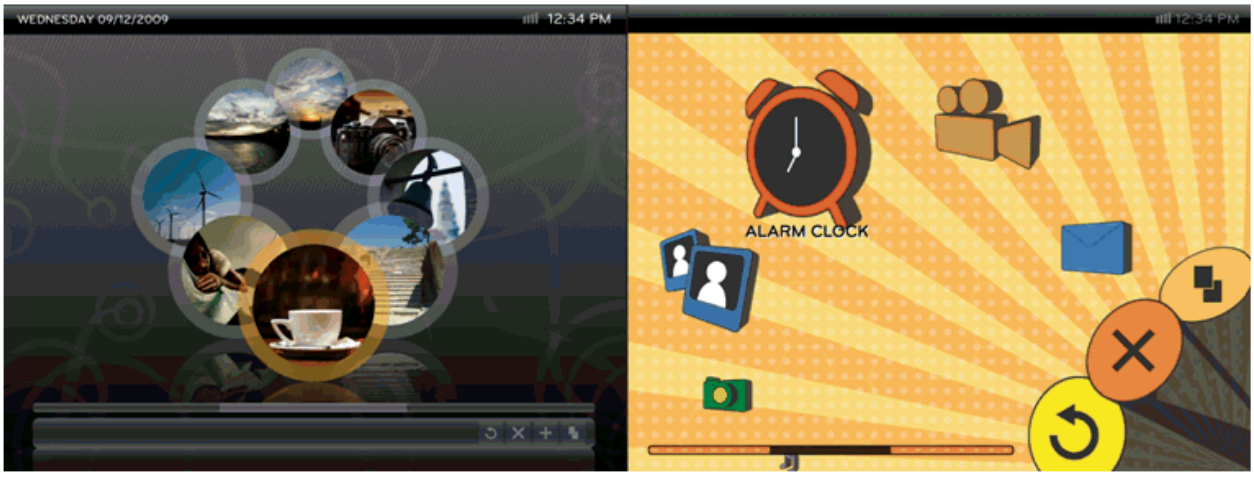


Figure 3: Two Android home screens.

Autonomous UI helps new products in the family arrive to market more quickly and with confidence.

The arena where this phenomenon is most evident is in common operating platforms. Designers may choose a common, interoperable COTS OS like Android or WinCE to save on non-differentiating engineering, and to leverage existing (or evolving) ecosystems that revolve around those platforms, like the Android Market. However, these platforms typically leave little room for OEM branding and customization. Unless developers invest in significant incremental engineering (as Motorola did with BlurUI), users will be greeted with the same UI as on every other gadget running the same OS, relegating the new device to the status of commodity (as in the PC market). And if OEMs do make the required investment, they will likely need to repeat that effort with each new platform release.

In 2009, OEMs launched dozens of devices and the Android Market (applications store) mushroomed to offer more than 30,000 applications. With projections for growing Android deployment in 2010, developers must increasingly invest in customizing Android to avoid delivering "me too" devices and applications. AUI, with its rich graphics and multimedia capabilities, provides the ideal vector to customize Android without forking the platform.

An AUI multimedia applications framework gives original equipment manufacturers (OEMs) of Android-based devices and their channel partners new capabilities to create visually rich applications and completely control and customize the look-and-feel of the end-user experience, thus easily differentiating their wares in an increasingly crowded marketplace.

Benefits of Decoupling Application and UI

Autonomous UI design is not just another way to subdivide application functionality. It actually offers developers benefits that emerge directly from decoupling UI and application code. These include:

- Shorter development time to create brand new, unique user interfaces without modifying application code.
- Ability to add intelligence to existing UI code, e.g., to make decisions and process events unforeseen in the original design.
- Ability to create a family of products based on a single application code base. High-end devices can have different user interface presentation and reactions from low-end phones.
- Shorter time to market for product iterations as the Q/A time and software development time is reduced.
- Enhanced brand retention — the ability to build devices with unique look and feel, even on commodity software (and hardware) platforms.

Conclusion

This article has explored the concept of decoupling UI and application design principles. This capability expands the market and extends the lifetime of application code by offering developers and other ecosystem players new opportunities to brand, differentiate, and refresh device software. In today's dynamic landscape of multiple application OSes (especially in mobile), it's important to build a strong base product that can be easily tailored for different packages, channels, and markets.

— Robi Karp is CEO of Fluffy Spider Technologies.

Return to Table of Contents

Tough Apps: Travelling Games

An empty hotel room represents a lost opportunity. Can you fill the void?

By Dennis Shasha

Long-time *Dr. Dobb's* readers may remember "Dr. Ecco's Omniheurist Corner" where Professor Dennis Shasha posed puzzles that required computer solutions. Dennis is returning to *Dr. Dobb's*, but with a new twist on puzzles. The idea is to take inspiration from the challenging apps that readers have faced and to turn those into puzzles.

If you have written an application that required extensive use of heuristics (because no algorithm would solve it) or that is otherwise algorithmically interesting, please send mail to Dennis at shasha@cs.nyu.edu. Put "Tough Apps" in the subject line. If Dennis likes it, he will contact you. Then he will try to understand what you've done and create a puzzle explaining the algorithmic core. You will write a little blurb about yourself and the problem if you want.

An empty hotel room represents a lost opportunity to the hotel owner. The marginal cost of cleaning a room is minimal compared to the fixed cost of the personnel, rent, and so on. So, the hotel has every interest in filling all its rooms.

Clever travellers and entrepreneurs can take advantage of this by waiting until the last minute to buy cheap rooms. Of course, they run the risk that they may not sleep anywhere. The result is a game. Let's work up to the game in stages. It will lead us to strange and amoral places, I promise.

Warm-Up 1: Suppose you have the only hotel in town. You have three rooms free for tonight. You know from history that there will be one visitor who is willing to spend \$300 per night, another at \$200 per night, and one at \$100 per night. We'll abbreviate this demand profile to 1 @ \$300, 1 @ \$200, 1 @ \$100. If you have to give the same price to all comers, what should you set your price to be?

Solution to Warm-Up 1: \$200 — you get \$400 and leave one room empty. A recurrent theme in this puzzle is that charging the same price to everyone in the service of fairness results in empty hotel rooms and roomless travellers.

Suppose there are two hotels next to one another of the same quality and the demand is twice that before, i.e. 2 @ \$300, 2 @ \$200, 2 @ \$100. Now if your hotel sets the price at \$200, the other hotel can set the price at, say, \$180 and get three guests out of the four. You get only \$200.

Warm-Up 2: If you don't know how much the other hotel will charge, then what can you guarantee to obtain?

Solution to Warm-Up 2: If your hotel charges \$133 and the other charges more, then you will fill your three rooms and get \$399. On the other hand, if the other hotel also charges less but more than \$100, then you will get only \$133. So, if you charge \$100, you will guarantee an income of \$300. If the other hotel reasons the same way, then there will be six rooms for six travellers, each costing \$100 per night. Note how much better this is for consumers.

Warm-Up 3: The hotel owners approach an agent to set room prices for both of them. They

Exclusive Downloads!
Systems Content Resource Bundle

Click here to download the Systems Content Resource Bundle!

Compliments of IBM

Powered by Dr.Dobb's

agree to give the agent a portion of any extra profit they would make beyond the $300 they can already get.

Solution to Warm-Up 3: If the agent sets prices at $200 per room instead of $100, then each hotel's profits will increase to $400 (less the agent's commission) while leaving two travelers without rooms. This is why a single agent (whether online or not) can be as bad for consumers as a hotel monopoly.

Now It's Your Turn

1. Suppose you have the only hotel in town and it has six rooms. The general problem is, given a demand profile and a certain number of rooms, try to find the price per room that will maximize hotel profits if you control all the rooms in a town. If the demand profile is x @ $300, y @ $200, and z @ $100, for which positive values of x, y, and z would you set the price for your six rooms at $300, for which positive values would you set the price at $200, and for which at $100?

2. If you control all the rooms in town, is there ever any profit-maximization reason to charge $250, given the above demand profile? Why or why not?

3. Suppose there are two virtually identical hotels on the same block each having three rooms, but you control the rooms of only one and the demand profile is again x @ $300, y @ $200, and z @ $100. For which x, y, and z values would you charge $200 to guarantee as much revenue as possible? For which x, y, z would you charge $300?

4. Staying in the two hotel scenario, for which x, y, z would you charge something strictly between $200 and $300 in your hotel (assuming you don't know what the other hotel charges)? What if you did know the other hotel's prices and customers would always compare prices before taking a room?

5. Loyalty programs and other perks may direct customers your way. Suppose that your perks program can convince all travelers to come to your hotel first. Each traveler will take a room in your hotel if you charge what he or she is willing to pay. As usual, there are six travelers altogether, where x travelers will be willing to pay $300, y @ $200, and z @ $100. How should you set your prices as a function of x, y, and z? How should your competitor set prices?

6. As you can see, the pricing power of the hotel owner depends as much on consumer behavior (whether consumers compare prices) as on monopoly power, either gained directly or through an agent. Do you see any general pattern?

Solutions

1. If x >= 3 and y <= 4, then setting the price at $300 is best for the hotels, though only x people will get rooms. Otherwise, if x + y >= 3, then charge $200 to get $800 for the two hotels. Otherwise, set the hotel prices to $100.

2. No. You capture no more customers than if you set the price to $300 and receive less revenue.

3. We already saw in the Warm-Ups that you would charge $100 if x = y = z = 2 to guarantee at least $300 in revenue. Any other price would give a lesser guarantee (though the possibility of more). If x >= 5, then each hotel may as well charge $300 as each will get at least two customers at that price. If x <= 4 and x + y >= 5, then $200 is the best price (because charging $300 would guarantee you only one customer and charging $200 gurantees at least two).

4. There are no x, y, z for which it is worthwhile to charge something between $200 and $300 unless you know the prices of the other. If, on the other hand, you knew the other hotel was charging $300 and x = 3 and y = 5, then you could charge say $280 and get all the $300 customers.

5. If x >= 3, then a price of $300 will guaranteee $900. If x + y >= 2, then $200 will guarantee $600. Your competitor can't charge even $200 unless x + y >= 5.

6. With monopoly power, you know all customers will come to you, so you can set the price through direct calculation. In a competitive market, you have to assume the worst case. If you can convince customers to come to you first, then you can charge the most per room.

— Dennis Shasha is a professor of Computer Science at New York University. His latest puzzle book Puzzles for Programmers and Pros *outlined some general approaches to programmatic puzzle solving. This column challenges you to find the solutions to the puzzles that lie at the core of some cool tough applications. He solicits your help in identifying such cool apps.*

Return to Table of Contents

Q&A: Henry Ford Would 'Recoil' at SPL

The role of Software Product Lines

by Jonathan Erickson

Charles Krueger is founder of BigLever, a pioneer in software production lines. He recently spoke with *Dr. Dobb's* editor in chief Jonathan Erickson.

Dr. Dobb's: Henry Ford is given credit for the production line. What would he think of SPL?

Krueger: Henry's initial reaction would likely be negative. His great accomplishment came from engineering innovations that removed all product variation, so that the identical automobile could be reproduced over and over again, with an efficient means of production. He would likely recoil at the notion of deliberately reintroducing product variation into manufacturing — be it systems or software.

With equal certainty, I believe the "SPL epiphany" would eventually bring a smile to his face, once he realized that his accomplishments in the efficient means of production had been extended from merely capitalizing on commonality to also incorporate feature-based product variation.

Dr. Dobb's: Where's SPL having the biggest impact?

Krueger: There are two characteristics that we consistently see in the big SPL success stories. The first is a compelling need or desire by the business for a game-changing advantage. For example, a strategic business opportunity to dramatically increase revenue by rapidly expanding product and feature diversity in a product line. Or, a critical need to quickly respond to market turbulence by significantly increasing efficiency or reducing deployment time in a product line.

The second characteristic of current-day SPL successes is that they come from the revenue-generating side of systems development organizations rather than groups such as internal IT development organizations. I attribute this to the fact that "top line" revenue generation in product development provides stronger SPL motivation than "bottom line" cost savings in IT.

The markets where we currently see the strongest pull for SPL are automotive, aerospace, and defense.

Dr. Dobb's: Can agile and SPL coexist?

Krueger: The correlation between agile and successful SPL practice is very strong, although sometimes not explicit. For example, you wouldn't necessarily expect Aerospace & Defense organizations that are contractually bound by waterfall processes to have agile SPL practices. However, even though the process for deploying any individual system may be waterfall in nature, the overall process for efficiently managing an entire product line must be agile in order to rapidly adapt to unpredictable requests for new products, new features, evolving feature combinations, new diversity, and changing boundaries for commonality.

Dr. Dobb's: When organizations opt for SPL, what hurdles can they initially expect?

Krueger: The biggest hurdles we encounter are around changing people — their roles, responsibilities, job descriptions, and processes. Changing technology is easy by comparison.

Development organizations are often based on a product-centric perspective, where the primary focus for a team or individual is on the development, delivery, schedule, and budget for an individual product. The SPL approach gains its efficiency

and effectiveness from a shift in perspective, changing from the "vertical" silo'ed product-centric perspective to a "horizontal" SPL-asset perspective that cuts across all of the products in the product line.

This organizational shift from vertical to horizontal perspective — and the inevitable resistance to change by some of the individual team members — is by far the biggest hurdle organizations experience. Because the SPL approach ultimately reduces complexity and overhead, the early detractors soon recognize and appreciate the benefits, so "hurdle" is the correct term. It is a short-term upfront obstacle.

Dr. Dobb's: What role does "testing" have in SPL?
Krueger: The SPL approach impacts tools, assets, and processes across the full systems and software development lifecycle — from requirements to design, implementation and, very importantly, testing. The best answer to this question might be in terms of what we see happen when a test organization initially opts out of a transition to SPL practice. The upstream stages in the development lifecycle may increase their output by factors of 2 to 10, quickly overwhelming the test organization that hasn't implemented corresponding SPL efficiencies. The test organization becomes the high-profile bottleneck that prevents the business from realizing SPL benefits.

This illustrates that testing plays a critical role in SPL. We see the most value from two SPL-specific testing techniques. The first is automated unit testing that includes "variants" to match all possible implementation "variants" of a component or subsystem. Nightly or continuous builds and unit test execution for all component variants provides good visibility into the health of the overall product line. The second is integration and system test minimization, by sharing test results from common or similar tests across different products in a product line. Knowing when duplicate testing can be avoided across multiple products is somewhat of an art, but we have seen good results in organizations that strive to put test effort into the areas of highest payoff.

Dr. Dobb's: Is there a place for SPL in small groups or organizations?
Krueger: Absolutely. One of our earliest success stories was with a small startup company that needed speed and efficiency to roll out a large product line of similar systems, without the corresponding need to hire a large engineering team.

Also, we find that the most effective way for organizations to transition to the SPL approach is incrementally by small teams. For example, by first migrating a small subsystem team and then repeating to bring in one small team after another.

Dr. Dobb's: Do SPL and compliance conflict?
Krueger: Compliance introduces some unique challenges to SPL that might initially appear as conflicting, but that can ultimately be turned into positive opportunities. Imagine creating a commercial avionics system product line under strict FAA DO-178B compliance. If you introduce an enhancement to a software component that is shared across multiple avionics products in your product line, have you just introduced multiple recertification burdens for the component in each of your products? If you aren't careful, the answer is a costly "yes". But with well-defined SPL impact analysis and change management processes, the controlled introduction of this change into different product baselines and shared certification effort across multiple products in a single baseline can actually result in less overall certification effort and lower defect density in your systems.

Dr. Dobb's: What's unique about BigLever's approach to SPL?
Krueger: BigLever provides technically innovative and advanced SPL tools, methods, and framework, but what is truly unique is our intense focus on the pragmatic issues that enable predictable and successful SPL deployments and benefits. Our patented Gears SPL Lifecycle Framework provides consistent and integrated SPL engineering throughout the lifecycle, from product line management to requirements, design, implementation, testing, and deployment. We have pioneered a "second generation" SPL approach with an out-of-the-box solution for automated configuration of all product line assets across the lifecycle, using concise "feature profiles" to indicate which product capabilities to include or exclude in any particular product.

BigLever's SPL framework makes it uniquely easy to transition to the SPL approach by allowing organizations to utilize their existing legacy assets and their preferred engineering tools as the basis for their SPL practice. BigLever has a unique 3-tiered SPL Methodology that pioneered and validated techniques that now enable organizations to make incremental and non-disruptive transitions to SPL practice.

Dr. Dobb's: What does it mean when we talk about "second generation SPL"?
Krueger: First generation SPL (1G SPL) approaches showed how organizations could reuse existing SPL architectures and software components to more efficiently create new product variants in a product line. However, the effectiveness of 1G SPL was hindered during product evolution and maintenance, due to the reliance on "application engineering", where each of the individual products in the product line were maintained as cloned copies or branches. This led to an exponential growth in effort and complexity around branching, merging, impact analysis, and

interdependency management. We observed that this complexity in 1G SPL was responsible for the limited degree of success and adoption in the early days of the SPL industry.

Second generation SPL (2G SPL) approaches address this limitation by creating, evolving, and maintaining a single, consolidated "superset" collection of product line assets, for each stage in the engineering lifecycle, and then using an automated "SPL configurator" to automatically assemble the "subset" of the product-specific assets required to match the features for any particular product. Maintaining the consolidated assets with 2G SPL is dramatically simpler than was possible with the multiple clones of 1G SPL. The 2G SPL approach is leading to much higher degrees of success and adoption by the mainstream industry.

Dr. Dobb's: Do changes in technology — new programming languages, new platforms — affect SPL implementations?

Krueger: The impact will be comparable to what engineers are already familiar with in changes in technology for individual products or systems. There is really nothing special about SPL when it comes to technology evolution. For example, changing an embedded software component to a networked service will be similar if that component is part of a one-of-a-kind system or if that component is reused as part of dozens of different systems in a product line.

Dr. Dobb's: Microsoft and others have talked about "software factories." How does this tie into SPL?

Krueger: Unfortunately, the term "software factories" has been used over time to mean different things in different contexts, ranging from very general concepts such as repeatable software development processes to very specific concepts such as a particular SPL approach. Microsoft has used the term "software factory" to describe its solution for creating domain-specific languages (DSL) and DSL compilers for a system of similar systems, thus it is an SPL approach that is appropriate for the subset of product lines that can be expressed with a DSL. But in general, SPL and software factories are not always used to mean the same thing.

Return to Table of Contents

Jolt Product Excellence Awards: Mobile and Web Development

Adobe Creative Suite 5

By Mike Riley

Congratulations to Adobe Creative Suite 5 (http://www.adobe.com/products/creativesuite/), the 2010 winner of the Jolt Product Excellence Award in the Mobile and Web Development category. Adobe Creative Suite 5 comes in four configurations: Design, Web, Production Premium, and Master Collection. The version matching most developer's needs (and budget) is the Web Premium edition, which is packed with all the design and development tools you need to assemble Flash, Flex, and even HTML5-centric mobile-friendly websites and applications.

In addition to Dreamweaver, the de-facto standard web page design application, the Creative Suite 5 Web Premium Edition also includes sophisticated image editors like Photoshop Extended, Illustrator, and Fireworks, as well as Acrobat 9 Pro and Bridge. For Flash and Flex application developers, Flash Professional, Flash Catalyst, and Flash Builder 4 Standard bring all the Flash and Flex programming and design tools under one umbrella. Also included is Adobe's Device Central, emulating a broad range of mobile devices so Flash developers can simulate how their work will appear on a variety of hardware configurations and environmental conditions.

For iPhone developers, Apple's change of heart now reinvites the opportunity to develop iPhone apps using these polished Creative Suite tools. And with the Flash runtime showing up on Android devices, Flash content will continue to be pervasive in the web space for some time to come.

(Stay tuned for my upcoming comprehensive review of Adobe Creative Suite 5 Master Collection from a developer's point of view, coming soon to drdobbs.com.)

Return to Table of Contents

iPad Programming Book Review

By Mike Riley

iPad Programming:
A Quick-Start Guide for iPhone Developers
Daniel H Steinberg and Eric T Freeman
Pragmatic Bookshelf
$34.95

iPhone developers looking for a book that will help them learn how to leverage the new SDK features of the iPad, authors Daniel Steinberg and Eric Freeman may have just what you're looking for. Is it worth the price of admission? Read on to find out.

iPad Programming: A Quick-Start Guide for iPhone Developers, delivers exactly what the title says. It's a book for seasoned iPhone developers seeking to extend their Objective-C and iPhone SDK skills to the iPad platform. While the examples given provide step-by-step detail on constructing the various iPad-specific features, this is not a book that will teach newcomers the fundamentals of iOS programming. Readers are expected to know Objective-C, the iPhone 3 SDK, and have developer access to the iPhone 3.2 SDK. Assuming these minimum requirements are met, this sub-250 page book is quite a bargain, given the smart and concise way the authors demonstrate how iPad-specific features like split views, pop-overs, custom keyboard buttons, advanced video playback, VGA output, and document file transfer via iTunes are coded. If you are a veteran iPad owner, you no doubt have encountered each of these features and wondered how they were done. With *iPad Programming* in hand, wonder no longer.

The book contains 13 chapters, each between 10 and 20+ pages long. It begins with a chapter on upscaling an iPhone application to a Universal, iPad resolution and screen-size friendly program. Then, chapters on using the Split View Controller, UIGestureRecognizer, Popovers, and Modal Dialog Boxes (and dealing with changes portrait or landscape orientation) are also easy to follow along. It wasn't until the chapters on custom keyboards and using the cocoa APIs to output drawings to a PDF did things get even more interesting. Two chapters on the Movie Player cover just about everything you will need to manipulate video elements and playback. The chapter on Apple's HTTP-based streaming protocol was brief (less than 10 pages) but useful, while chapters sending video output to an external VGA display and initiating and managing iPad peer network connections were invaluable. Chapter 12 on transferring, registering, and opening documents supported by your iPad application were also priceless. The final chapter concludes with a series of takeaways reminding readers what makes an iPad application look, feel, and behave like an iPad app. Part Apple design guidelines and part hard-earned experience, these tips were healthy reminders for current and future iPad developers to internalize.

One of the unfortunate aspects of the timing of the book's release is that the examples are all based on the iPhone 3.2 SDK. The book was in beta for some time, but given how close its publication is to the impending release of iOS 4.2 for the iPad, it would have made sense to hold off another month or so to be awarded the first book to cover coding for the new iOS for iPad release out of the gate. As such, those who prefer reading technical books on paper may want to consider purchasing the book+ebook combination instead of just the paper edition. Pragmatic Bookshelf has a good history of updating its electronic publications frequently, and I anticipate that this book may have at least an appendix of changes and/or new features for programmers to explore in the 4.2 iOS release. Indeed, it makes sense for a person interested in a book on iPad programming would want to read it on their iPad. That's how I read the book, and I found it to be even more comfortable than reading a print edition. I propped my iPad up next to my monitor and followed along with the code examples. Unlike a paper book, I never once had the binding flop the book closed and tumble out of my lap. Reading the Epub edition using the iBooks application allowed me to highlight, annotate, and search for passages way faster than the analog ink on paper alternative.

The key aspect I appreciated in *iPad Programming* is how the authors have zeroed in on the cool features of the device and offered their interpretation of Apple's SDK documentation. It's a developer to developer communication style that I found guided me along a much faster path toward understanding how to call upon nifty iPad functions compared to SDK examples. I hope the authors quickly follow up the book with extra chapters on the new iOS 4.2 features such as multitasking and Game Center to keep the book even more relevant for some time to come.

Return to Table of Contents

Overlooked Essentials For Optimizing Code

By Walter Bright

I've been programming for 35 years now, and I've done a lot of work optimizing programs for speed (an example is available at http://biolpc22.york.ac.uk/wx/wxhatch/wxMSW_Compiler_choice.html), and watching others optimize. Two essential techniques are consistently ignored.

Nope, it isn't avoiding premature optimization. It isn't replacing bubble sort with quicksort (i.e. algorithmic improvements). It's not what language used, nor is it how good the compiler is. It isn't writing i<<2 instead of i*4.

It is:

1. Using a profiler
2. Looking at the assembly code being executed

The people who do those are successful in writing fast code, the ones who do not are not. Let me explain.

Using A Profiler

The old programming saw is that a program spends 90% of its time in 10% of the code. I've found that to not be true. Over and over, I've found that a program spends 99% of its time in 1% of the code. But which 1%? A profiler will tell you. Spending 100 hours of dev time on that 1% will yield real benefits, while 100 hours on the other 99% will not produce much of anything worthwhile.

What's the problem? Don't people use profilers? Nope. One place I worked at had a fancy expensive profiler that was still in its shrink wrap 3 years after purchase. Why don't people use profilers? I don't really know. I once got into a heated exchange with a colleague who insisted he knew where the bottlenecks were; after all, he was an experienced professional. I finally ran the profiler myself on his project, and of course the bottleneck was in a completely unexpected place.

Consider auto racing. The team that wins has sensors and logging on just about everything they can stick a sensor on. You can race using seat-of-the-pants tuning and have a jolly good time on the track, but you won't win and you won't even be competitive. You won't know if your poor speeds are caused by the engine, the exhaust, the aerodynamics, the tire pressure, or the driver. Why should programming be any different? You can't improve what you can't measure.

There are lots of profilers available. You can get ones that look at the hierarchy of function calls, function times, times broken down for each statement, and even at the instruction level. I've seen too many programmers eschew profilers, preferring instead to whittle away their time with useless and misdirected "optimizations" and getting trounced by their competitors.

Looking At The Assembly Code

Years ago, I had a colleague, Mary Bailey, who taught remedial algebra at the University of Washington. She told me once that when she wrote on the board:

```
x + 3 = 5
```

and asked her students to "solve for x", they couldn't answer. But, if she wrote:

```
__ + 3 = 5
```

and asked the students to "fill in the blank" all of them could do it. It seems that the magic word "x" seemed to cause them to reflexively think "x means algebra, I don't understand algebra, I can't do this."

Assembler is the algebra of the programming world. If someone asks me "was my function inlined by the compiler" or "if I write i*4, will the compiler optimize it to a left shift" I'll suggest they look at the asm output of the compiler. The reaction is how rude and unhelpful could I be? The person will follow up by saying he doesn't know assembler. Even C++ experts will say this.

Assembler is the simplest language (especially compared with C++!). For example,

```
ADD ESI,x

is (expressed in C style):

ESI += x;

and:

CALL foo

is:

foo();
```

Details vary among CPUs, but that's how it works. It's not even really necessary to know that. Just looking at the assembler output and comparing it to the source code will tell a LOT.

How does this help optimization? For example, I knew a programmer years ago who thought he'd discovered a new, faster algorithm to do X. I'm being deliberately vague to protect him. He had the benchmarks to prove it, and wrote a nice article about it. But then someone looked at the assembler output of the regular way, and his new fast way. It turns out that the way he'd written his improved version had allowed the compiler to replace two DIV instructions with one. This had really nothing to do with his algorithm. But DIV is an expensive instruction, and this was in the inner loop, and so his algorithm appeared to be faster. The regular implementation could also be recoded slightly to use only one DIV, too, and it would perform just as fast as the new algorithm. He had discovered nothing.

For my next example, a D user posted a benchmark showing that dmd (Digital Mars D compiler) was lousy at integer arithmetic, while ldc (LLVM D compiler) was much better. Being very concerned about such a result, I promptly looked at the assembler output. It was pretty much equivalent, nothing stood out as being accountable for a 2:1 difference. But there was a long divide in there, done with a call to a runtime library function. That function call completely dominated the timing results, all the adds and subtracts in the benchmark had no significant impact on the speed. Unexpectedly, the benchmark wasn't about arithmetic code generation at all, it was about long division only. It turns out that dmd's runtime library function had a crummy implementation of long division in it. Fixing that brought the speed up to par. It wasn't the code generation at fault at all, but this was not discoverable without looking at the assembler.

Looking at the assembler often gives unexpected insight into why a program performs as it does. Unexpected function calls, unanticipated bloat, things that shouldn't be there, etc., all are exposed when looking at it. It isn't necessary to be an assembler crackerjack to be able to pick that up.

Conclusion

If you feel the need for speed, the way to get it is to use a profiler and be willing to examine the assembler for the bottlenecks. Only then is it time to think about better algorithms, faster languages, etc.

Conventional wisdom has it that choosing the best algorithm trumps any micro-optimizations. Though that is undeniably true, there are two caveats that don't get taught in schools. First and most importantly, choosing the best algorithm for a part of the program that has no participation to the performance profile has a negative effect on optimization because it wastes your time that could be better invested in making actual progress, and diverts attention from the parts that matter. Second, algorithms' performance always varies with the statistics of the data they operate on. Even bubble sort, the butt of all jokes, is still the best on almost-sorted data that has only a few unordered items. So worrying about using good algorithms without measuring where they matter is a waste of time — your's and computer's.

Just like ordering speed parts from an auto racing catalog isn't going to put you anywhere near the winner's circle (even if you get them installed right), without profiling, you won't know where the problems are without a profiler. Without looking at the assembler, you may know where the problem is, but often won't know why.

Acknowledgments

Thanks to Bartosz Milewski, David Held, and Andrei Alexandrescu for their helpful comments on a draft of this.

CLICK HERE TO COMMENT ON THIS POST
(http://www.drdobbs.com/blog/archives/2010/09/overlooked_esse.html)

Return to Table of Contents