



Dr. Dobb's DIGEST

The Art and Business of Software Development

May 2010

Editor's Note by Jonathan Erickson	2
Techno-News Memory Management as a Separate Thread New technique for hard-to-parallelize programs	3
Features When Quality, Security Count by Sid Sidner Static code analysis can make a big difference.	4
Simple Concurrency with Dataflow Variables in C++ by Anthony Williams What are dataflow variables anyway?	6
An IDE With Lots To Like by Jonathan Erickson Developers pick their favorite Visual Studio 2010 features.	12
Developing for the iPad by Jonathan Branam Ideate is an iPad app for turning ideas into products.	15
JavaFX Database Programming with Java DB by Eric J. Bruno Who said JDBC programming with Java and JavaFX Script was difficult?	20
Columns Book Review by Mike Riley Mike Riley examines <i>The Art of Assembly Language, 2nd Edition</i> by Randall Hyde.	27
Q&A: Catching Up with Lisp by Jonathan Erickson, Support for multicore and Big Data are among the upcoming features of Franz's Lisp-based tool.	28
Blog of the Month by Jocelyn Paine An Online Budget Questionnaire, JavaCC, and the Three Ways of Putting Together.	30
Other Voices by Erik Troan Why scripting is evil.	33
Effective Concurrency by Herb Sutter Herb advises you to associate mutexes with data to prevent races.	34

Not Your Father's Hands-Free Ford



By Jonathan Erickson,
Editor In Chief

The good news for Toyota is that the company has been getting a lot of news coverage recently. The bad news is the kind of news coverage Toyota has been getting. Between the news and all the extra advertising, you sometimes think that Toyota is the only newsworthy automobile company out there. That's hardly the case.

For instance, Ford recently announced that Ford SYNC has been installed in 2-million vehicles. SYNC is the in-car communications and entertainment system based on the Microsoft Auto platform (<http://www.microsoft.com/windowseembedded/en-us/products/windows-embedded-automotive/default.mspx>). Two-million installations. I believe that's what they call a milestone. Since it was first released in 1998, Microsoft Auto has had a number of monikers — “Windows CE for Automotive,” “Windows Automotive,” and “Windows Mobile for Automotive.” By whatever name, Auto is an embedded operating system based on Windows CE. Ford SYNC (<http://www.fordvehicles.com/technology/sync/?searchid=426441|28125565|205373340>), Ford's implementation of Microsoft Auto, was released in 2007.

I have to admit that Ford SYNC has some nifty features — mobile phones that are voice-activated for hands-free calling, audible text messaging, real-time traffic and directions, in-car WiFi, and the like — implemented in everything from a Taurus to a Kenworth. (What? No motor scooters?) That's right a Kenworth, as in an 18-wheeler semi tractor-trailer truck. Kenworth and Peterbilt (another 18-wheeler) have worked with the folks at Microsoft Auto to develop NavPlus and SmartNav, automotive platforms designed specifically for the commercial trucking industry. The Silverlight-based systems give truck drivers vehicle monitoring and diagnostic features, entertainment options such as commercial truck-enhanced navigation and music management, and hands-free phone calling. All features are accessed through voice or touch-enabled commands.

For more on what makes automotive systems like these tick, see “An Architecture for In-Vehicle Infotainment Systems” at <http://www.drdoobbs.com/embedded-systems/22260043>.

Like they said back in the '80s, it sure isn't your father's Oldsmobile — or Ford or Kenworth, for that matter.

A stylized, handwritten signature in blue ink that reads "Jonathan Erickson".

[Return to Table of Contents](#)

Memory Management as a Separate Thread

New technique for hard-to-parallelize programs

Researchers at North Carolina State University (<http://www.ncsu.edu/>) have developed a new approach to software development that will allow common computer programs to run up to 20 percent faster and possibly incorporate new security measures. The researchers have found a way to run different parts of some hard-to-parallelize programs — such as word processors and web browsers — at the same time, which makes the programs operate more efficiently.

Every computer program consists of multiple steps. The program will perform a computation, then perform a memory-management function — which prepares memory storage to contain data or frees up memory storage that is currently in use. It repeats these steps over and over again, in a cycle. And, for difficult-to-parallelize programs, both of these steps have traditionally been performed in a single core.

“We’ve removed the memory-management step from the process, running it as a separate thread,” said Yan Solihin (http://www.csc.ncsu.edu/directories/faculty_info.php?id=2282), an associate professor of electrical and computer engineering at NC State, director of this research project, and co-author of a paper describing the research. Under this approach, the computation thread and memory-management thread are executing simultaneously, allowing the computer program to operate more efficiently.

“By running the memory-management functions on a separate thread, these hard-to-parallelize programs can operate approximately 20 percent faster,” Solihin says. “This also opens the door to development of new memory-management functions that could identify anomalies in program behavior, or perform additional security checks. Previously, these functions would have been unduly time consuming, slowing down the speed of the overall program.”

Using the new technique, when a memory-management function needs to be performed, “the computational thread notifies the memory-management thread — effectively telling it to allocate data storage and to notify the computational thread of where the storage space is located,” said Devesh Tiwari, a Ph.D. student at NC State and lead author of the paper. “By the same token, when the computational thread no longer needs certain data, it informs the memory-management thread that the relevant storage space can be freed.”

The paper, “MMT: Exploiting Fine-Grained Parallelism in Dynamic Memory Management,” was presented at the IEEE International Parallel and Distributed Processing Symposium in Atlanta (<http://www.ipdps.org/>). The research was funded by the National Science Foundation. The paper is coauthored by Tiwari, Solihin, NC State Ph.D. student Sanghoon Lee, and James Tuck, an assistant professor of electrical and computer engineering at NC State.

[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.drdoobs.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™

When Quality, Security Count

Static code analysis can make a big difference

By Sid Sidner

ACI Worldwide is a provider of payments software to banks and merchants around the world. With more than 800 software engineers working in development centers in seven time zones, issues such as software quality and security are critical to ACI's success. As director of security engineering, it's my job to ensure that our code base is bug-free and intruder resistant, while continually improving the software's overall quality. These concerns aren't new. They've been our mantra since the company was founded more than three decades ago. We decided several years ago that the best way to ensure quality and security was to introduce static source code analysis into our development processes.

Static code analysis is the process of examining and evaluating software without actually executing the code. Analyzing software when executing software is known as dynamic analysis. Static code analysis is all about moving the detection of critical security and quality problems upstream, ensuring they're identified and fixed early in the development process.

This approach yields significant productivity gains across the entire process and leads to cleaner, more stable builds, more efficient testing, and of a course, a higher quality product. Besides helping us find bugs that we've missed in unit testing, static code analysis has made all our engineers aware of security issues and helped us teach junior staff better coding techniques.

What's Involved

Static source code analysis tools are almost entirely automated. They're like compilers, but instead of generating machine-executable code, they simply find bugs and issue warnings about security vulnerabilities, logic errors, implementation defects, concurrency violations, boundary conditions, and other glitches in the code. The tools provide a list of problems, each tied to a specific location in the source code. Detailed context is usually provided to explain how the tool arrived at the conclusion.

Static analysis tools use very sophisticated process flow and data flow analysis. The quality

and security issues they identify are often complex and involve obscure logic problems, which is why these tools can be so valuable.

Static source code analysis tools analyze 100% of the source code, far more than any external test tools. For organizations that must comply with the Payment Card Industry Data Security Standard (PCI DSS) or Payment Application Data Security Standard (PCI PADSS), these tools fulfill code review requirement. They also produce valuable metrics, including kilo-lines of code (KLoCs), file counts, and "churn" — that is, the number of files that have changed between two regular builds.

Introducing static code analysis and the requisite tools into the development process isn't always painless, however. At ACI Worldwide, we found many subtle pitfalls in our efforts to roll out this approach company-wide. The tool changes the way many people work and must become a part of the organization and its culture.

For instance, static code analysis tools usually require careful integration into the project build process. For large software products, these builds are often somewhat of a black art, involving the use of Make and Ant. There are many options and dependencies. All static code analysis tools offer powerful utilities to analyze the build process and insert themselves into the right places, but some manual tuning is usually required.

These tools also must be integrated into developers' daily work. Again, tool makers offer both command-line versions of the tools as well as plugins for many of the popular integrated development environments such as Eclipse and Visual Studio.

Most importantly, the tools require that the code base have a subject matter expert (SME) who can also provide the same service for the tools. That person will answer questions not just about how the tool operates but also about the issues that the tool is finding — including identifying when the tool is generating a false positive. The SME will provide training and support to other developers, a fairly heavy workload for the first few weeks, until everyone is familiar with the static analysis tool. After that, that part of the workload should settle down to several hours a week.

1. Do you need a static or dynamic analysis tool?
2. What languages and platforms does it support?
3. How flexible is the reporting component?
4. How easy is it to add or update rules?
5. Does it integrate with your IDE?

Initial Analysis: Panic Time

The biggest challenges with static code analysis tools are problems in existing code. There's an old programmer's joke that says "God made the world in six days because he had no installed base." This is certainly not the case for most businesses, which often have millions of lines of code.

The first time an existing codebase is analyzed, tens of thousands of issues will be found. Don't panic. Remember, these issues have been there for awhile, and the software continues to function and provide users with what they need.

At ACI Worldwide, all the issues from an initial build on existing code are immediately deferred and hidden from sight. That way developers don't get overwhelmed and can stay focused on ensuring that new problems aren't introduced into the code. At some point in the future, product planners and the senior development staff review the deferred issues, prioritize and group them, and decide when remediation can be factored into the planning for a future release. There's no perfect approach, and businesses must always make hard decisions about whether to counter a vulnerability or assume the risk.

Tips For Success

How can you ensure a successful deployment? Here are some hard-won tips from our experience:

- Define an initial issue policy. You may decide to only deal with the most severe issues for the first project cycle.
- Get the global mechanics working. Many of the tools require license managers and centralized result servers.
- Attack one product at a time. Get it working with one group and then move on to the next.
- Identify SMEs. Every product needs at least one subject matter expert. Large products that are broken into major components will naturally need an SME for each one. Be sure that the SME and his or her manager understand the ongoing responsibilities and time commitment.
- Train SMEs. Make them designated experts.
- Work with SMEs. Help them to do build and tool integration for their product or component.
- Train developers. The SME should guide how the tool is integrated into the team's development process.
- Perform initial analysis on existing code and defer all issues. Don't discuss the large quantity of issues with the developers. If any ask, explain to them that they've been set aside and will be considered in a future product cycle.
- Deliver help from SMEs to developers as required. During the first days of the roll-out, the SME should monitor the developers' work. Developers should be analyzing the code often, at least before they submit a completed unit of work into the product build. Just as a developer wouldn't check in a unit of code that doesn't compile, they won't want to check in a unit that still has static code analysis issues.
- Run the build analysis often. If the developers are doing their job and addressing issues as they come up, then no issues should be found at this stage.
- Review deferred issues. After the process is running smoothly and the tool is a routine part of work, review deferred issues and plan whatever remediation is needed for future releases.

The Right Tool For You

There are numerous open source and commercially available static code analysis tools on the market. When choosing one, the place to start is with language support. Some tools like AdaCore's CodePeer and Green Hills' DoubleCheck support a single language. Other static code analysis tools support multiple languages.

But language support isn't the only consideration. When ACI Worldwide was in the market for a static code analysis tool two and a half years ago, we identified five vendors — Coverity, Fortify Software, Klocwork, Ounce Labs, and Veracode. Veracode was eliminated immediately because it only offered code analysis as a service, and we wanted a tool that could be used in-house and provide developer training. Each of the other four vendors performed an in-house proof-of-concept on a large C++ program (2,500 KLoCs) and a large Java program (600 KLoCs).

Coverity was eliminated because, at the time, the tool provided excellent quality checking but had limited security checking. Conversely, we eliminated Ounce Labs because it focused almost exclusively on security, assuming that the prospect already had quality checkers, which wasn't the case for us.

Fortify Software and Klocwork were comparable in their ability to find important quality and security issues. However, Klocwork's licensing model made it less expensive for us. Klocwork used the FlexLM license manager with floating licenses, whereas Fortify Software had a dedicated code contributor model. Since we have development centers spread around the globe in different time zones, we're able to share the licenses very effectively around the clock, so Klocwork was the right fit for us.

Final Analysis

Overall, static code analysis has proven to be a valuable tool for ACI Worldwide. For a reasonable cost per developer, we're finding serious bugs more comprehensively and earlier in the development process. In addition, the Klocwork suite we chose provides a way to connect experienced senior developers with junior developers. The tools include extensive help files that refer developers having difficulty with an issue to a more experienced developer to get advice — always a valuable interaction.

Bottom line: Static code analysis tools help incorporate security and quality awareness into the fabric of the entire development organization. Finding bugs earlier and avoiding security breaches is invaluable to any software development effort.

— Sid Sidner is director of security engineering for ACI Worldwide.

[Return to Table of Contents](#)

Simple Concurrency with Dataflow Variables in C++

What are dataflow variables anyway?

By Anthony Williams

Imagine for a moment a spreadsheet. Some of the cells have values in, but others have formulae that determine the value, possibly based on other cell values. Some are simple calculations such as the sum of a range of cells, while others are more complex and perform intricate calculations involving many other cell values in a multitude of ways. Naturally, the cells that are used in a formula may themselves contain formulae, in which case the spreadsheet calculation engine has to ensure that they are calculated first. If it turns out that the formula in A1 depends on the value of B27, which in turn depends on A1, then you have a circular reference and the answer is undefined.

Dataflow variables are like spreadsheet cells. They may be simple values, or the result of complex calculations involving other variables, but they can only be set once. Any calculation that needs the value of a variable that has not been set must block until the value has been set. This makes dataflow variables ideal for concurrency — the implementation can take care of all the synchronization internally; all you need to do is reference the variables where they are needed and ensure that there are no loops in your logic. Remember the spreadsheet — if a cell needs its own value to resolve its formula then you have an undefined value. The same happens with dataflow variables, though you'll likely get a hung program.

Using Dataflow Variables

Dataflow variables are available for multiple languages, including Oz, Ruby, and Groovy. Also, because of their write-once property, code that uses them is very similar in structure to code written in a Functional language such as Erlang or Haskell. Let's take a look at a few examples, then look at one way of implementing them in C++.

Listing 1: Simple addition with dataflow variables

```
int main()
{
    DataFlow<int> x,y,z;

    z.task([&]() {return x.get()+y.get();});
    y=99;
    x=123;
    std::cout<<"z="<<z.get()<<std::endl;
}
```

This simple example declares three dataflow variables that store integers. It then specifies that *z* is the result of a task, passing in a lambda function that calculates the sum of *x* and *y* as that task. Then it sets the values of *x* and *y* explicitly, before printing the value of *z*, which is now available since *x* and *y* have been set.

Okay, how about something a bit more complex? Listing 2 shows the use of dataflow variables to calculate the mean and standard deviation of a set of data:

Listing 2: Calculating the mean and standard deviation

```
double standard_deviation(std::vector<double> const&
values,double mean)
{
    double const variance=
        std::accumulate(values.begin(),values.end(),0.0,
        [=](double prev,double x){return prev+(x-mean)*
        (x-mean);})/values.size();
    return sqrt(variance);
}

double calculate_mean(std::vector<double> const& values)
{
    return
    std::accumulate(values.begin(),values.end(),0.0)/values.size();
}

int main()
{
    DataFlow<std::vector<double> > data;
    DataFlow<double> sd,mean;

    sd.task([&](){return
    standard_deviation(data.get(),mean.get());});
    mean.task([&](){return calculate_mean(data.get());});

    data=get_data();

    std::cout<<"standard deviation="<<sd.get()<<" , mean="
        "<<mean.get()<<std::endl;
}
```

In this case, both the mean and standard deviation depend on the data, but the standard deviation also depends on the mean. The

implementation ensures that all the dependencies are ordered correctly, so when you wait for the standard deviation value by calling `sd.get()`, the implementation ensures that the task to calculate the standard deviation has run. Since this in turn calls `mean.get()`, the implementation will also ensure that the task to calculate the mean has been run, too.

Dataflow Variables and Concurrency

Now, here's the nifty part: dataflow variables make it really easy to parallelize an application. If the only state shared between the separate tasks that set the dataflow variables is itself encapsulated in dataflow variables then the implementation can choose to run some or all of the tasks on separate threads and everything will "just work." If the code worked fine on a single thread and you only share state through dataflow variables, then it will also work fine with multiple threads.

Of course, it requires discipline to ensure that the only shared state is in dataflow variables. That means no global variables that aren't dataflow variables, so `std::cin` and `std::cout` are off limits. You also need to ensure that if you pass a pointer through a dataflow variable, the data being pointed to isn't going to change.

If you really need to access shared state that isn't covered by dataflow variables, you need to ensure that everything is correctly

What Is the Windows Azure Platform?

Windows Azure provides the platform needed to run applications and store data in the cloud. Click screen to watch a short video overview about Microsoft's Azure platform on Dr. Dobb's Microsoft Resource Center.



Click screen to link to video

synchronized. For example, you could have a global mutex that you used to protect uses of `std::cin` and `std::cout`. If you do this, you can't hold locks when you query a dataflow variable, since that may cause a task to be run, which may in turn try and acquire the same lock, and thus deadlock your program.

Of course, you're not limited to automatic parallelism: You can also spawn your own threads to set the values on dataflow variables. Even if you do so, managing shared state with dataflow variables will still make your code easier to reason and less error-prone. For example, you can only get a deadlock from dataflow variables if there is a circular dependency, in which case you get the deadlock even on a single-threaded execution. The potential for race conditions is also much reduced — since the variables are write-once, the biggest source of race conditions (mutating state) is gone.

Under the Hood

Okay, so we've looked at what the potential benefits are from using dataflow variables, so let's take a look at what makes them tick. I'm going to describe a simple implementation based on the new C++0x thread library. This has the benefit of being portable between platforms, while missing out on potential optimizations, and relinquishing control of the number of threads to the C++0x library.

Our variables have three operations we can do to them:

- Assign a value directly
- Specify that the value is the result of a task
- wait for and retrieve the value

Assigning a value and then waiting for it is easily done with a `std::promise/std::shared_future` pair, whereas specifying that the value is the result of a task is done with a `std::async/std::shared_future` pair. The thing is we don't know which we're going to have, and we can't wait on a future until it's been associated with something. We therefore need a mechanism for waiting that allows us to first wait until the future is valid, and then wait on the future itself.

This is a problem that's been around for as long as there's been multithreaded code, and the solution is almost as old: Use a condition variable. In the implementation of `get()` we use a standard condition-variable wait loop to wait until the future is valid. Once the future is valid we can wait on the future to retrieve the value.

Listing 3: The `DataFlow<T>::get()` member function.

```
T const& get()
{
    std::unique_lock<std::mutex> lk(m);
    while(!data.valid())
    {
        cond.wait(lk);
    }
    lk.unlock();
    return data.get();
}
```

What Is Windows Azure? Part II

In part II, you will learn about the key Azure building blocks, the AppFabric storage, and the resulting developer experience. Click screen to watch a short video explaining these details about Microsoft's Azure platform on Dr. Dobb's Microsoft Resource Center.



Click screen to link to video

In the assignment operator we can create a promise, set the stored future to that from the promise, and set the value of the promise. We can then notify every thread waiting on the condition variable that the future has been set, and that the value is ready to retrieve.

Listing 4: The `DataFlow<T>` assignment operator.

```
DataFlow& operator=(T value)
{
    std::unique_lock<std::mutex> lk(m);
    if(data.valid())
    {
        throw std::runtime_error("Source already set");
    }
    std::promise<T> p;
    p.set_value(std::move(value));
    data=p.get_future();
    lk.unlock();
    cond.notify_all();
    return *this;
}
```

Similarly, in the `task()` member function we can launch the task with `std::async`, set the future to that returned from the `std::async` call, and notify all threads waiting on the condition variable:

Listing 5: The `DataFlow<T>::task()` member function.

```
template<typename F>
void task(F&& f)
{
    {
        std::lock_guard<std::mutex> lk(m);
        if(data.valid())
        {
            throw std::runtime_error("Source already set");
        }
        data=std::async(std::function<T>()>(std::forward<F>(f)));
    }
    cond.notify_all();
}
```

Note that since this is a write-once variable, both the `task()` member function and assignment operator throw an exception if the future is already valid, indicating that the variable has already been set to a value or a task result. Also note that in both cases the mutex is unlocked before the condition variable is notified. This means that the waking threads don't have to block waiting for the notifying thread to unlock the mutex before proceeding.

The use of `std::async` does give us a modicum of control over the use of concurrency. In particular, it gives us three options:

- Completely synchronous execution. The tasks are only executed when the variable's value is requested in a call to `get()`. This

What Is SQL Azure?

High availability, fault tolerance, and cost-effective scaling are just some of the benefits of SQL Azure. Click screen to watch a short video explaining these details about SQL Azure on Dr. Dobb's Microsoft Resource Center.



Click screen to link to video



I KNOW A PLACE THAT'S DIFFERENT, BUT FAMILIAR

Code in the cloud ... with the development technologies you already know

With **Windows Azure™**, you have a flexible, on-demand infrastructure that taps into all the IT resources you need. The advantages?

You have the freedom to focus.

With the cloud as your development platform, you can focus on what you do best: create applications, without the restrictions on performance, capacity, or deployment that come with using your own hardware rather than the cloud.

You can code in the language you choose.

Windows Azure enables you to build applications in multiple languages, including .NET, PHP, and Java.

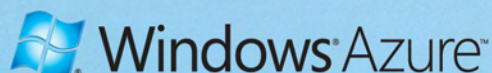
You're already equipped for the cloud.

You can start developing applications for Windows Azure right away, using familiar tools such as Microsoft® Visual Studio® 2008.

Find out more.

See how Windows Azure can help get you coding in the cloud.

Go to **windowsazure.com** now.



allows us to easily check the logic of our code — any circular references will repeatedly and deterministically hang the program.

- Completely asynchronous execution. This allows for maximum concurrency, but potentially leads to oversubscription by spawning a huge number of threads. It might also result in hitting resource limits on the number of threads.
- Allowing the implementation to choose between synchronous and asynchronous execution. This is what the implementation of `task()` shown above does. This is likely to provide at least some level of parallelism, but the maximum potential concurrency might not be realized if the implementation makes poor choices of which tasks to execute asynchronously. In particular, if all the asynchronous tasks block on the same variable then the concurrency is limited, even if there are other tasks that could be run in parallel.

The choice of option can be specified by providing an additional first parameter to the `std::async` call. By specifying `std::async(std::launch::sync, std::forward<F*>(f))`, then the task will be called synchronously by the first thread to call `get()`, whereas by specifying `std::async(std::launch::async, std::forward<F>(f))` the task will run on its own thread. The third option, which is equivalent to not specifying the policy, is to pass `std::launch::any` as the launch policy. This leaves it up to the implementation to choose.

The ideal schedule on a given set of hardware can probably be best obtained with a specially designed thread pool. However, the use

of `std::async` will go a fair way towards that, and can be easily be replaced later if profiling demonstrates that there is appreciable opportunity for concurrency that is not being exploited. It also provides an easy way to debug the dataflow logic by using `std::launch::sync`.

Try It Out

The simple implementation of dataflow variables based on the C++0x thread library described here can be downloaded from my website at <http://www.stdthread.co.uk/dataflow>, and from *Dr. Dobb's* at http://i.cmpnet.com/ddj/images/article/2010/code/cpp_dataflow.zip. Let me know what you think. Has the use of dataflow variables (either in C++ or any other language) enabled you to simplify your multithreaded code?

— Anthony Williams is author of the book *C++ Concurrency in Action* (<http://www.manning.com/williams/>) and of the *just::thread C++0x thread library* (<http://www.stdthread.co.uk/>). He can be contacted at anthony.ajw@gmail.com.

[Return to Table of Contents](#)

What Are the Access Control Service and the Service Bus?

This video presents an outline of the web-based services that are part of the Windows Azure Platform, exploring the traditional ways you have to interconnect applications and the ways you can use the Service Bus to address communication, authentication, and other complexities related to security-based protection. Click screen to watch a short video explaining these details about Windows Azure on Dr. Dobb's Microsoft Resource Center.



Click screen to link to video

An IDE With Lots To Like

Developers pick favorite Visual Studio 2010 features

By Jonathan Erickson

Instead of thinking of recently released Visual Studio 2010 as just another integrated development environment, Microsoft is pitching it as a full-fledged platform for developing all-things Windows — desktop, Web, mobile, enterprise, and everything in between. But being “full-fledged” requires tools and features, and lots of them.

Visual Studio 2010 has features you'd expect of any IDE, like an editor and debugger. It also has visual design tools. But what makes it more than just another IDE is extensibility. Like previous versions, Visual Studio 2010 accommodates third-party and roll-your-own macros, plug-ins, add-ins, and packages. Being able to adapt tools to fit the development environment instead of the other way around is a powerful capability, particularly when it comes to programmer productivity.

All this feature richness is great, but at some point it runs the risk of feature bloat, bogging down resources and performance in the process. Remember the Ada programming language? Oft described as having everything including the kitchen sink, Ada was ahead of its time with object-oriented and other capabilities. But its plethora of features was more than most developers could handle, and it never went mainstream. That's something the Visual Studio team should keep in mind.

That said, to find out more about Visual Studio 2010's features, I asked top-flight Windows programmers what they like best about VS2010.

Code Contracts Guarantee

Visual Studio 2010's Code Contracts is a .NET 4.x feature with tremendous potential. Code Contracts are methods developers use to define the terms of service of a given class and its methods. They're based on Bertrand Meyer's Design by Contract concept, where each method indicates the conditions that must be verified for the code to execute, the conditions that will be verified after the execution completes, and the conditions that never change during the life of any class instance.

Code Contracts let you strictly define the specs for each class and method and go a long way toward proving the semantic correctness of code. Once you define Code Contracts, you can use them as exception handlers in your code or just keep them as specifications.

Visual Studio 2010's Static Checker analysis tool uses Code Contracts. It runs in the background for each build, checking whether each method gets and returns proper values based on the declared contract.

The result? You get warned at compile time about code that compiles okay but will fail at runtime.

—Dino Esposito, software consultant

Intellisense Makes Sense

We already know that Intellisense, which displays a list of every accessible element in a namespace, class, and object, can have a big impact on developer productivity. Improvements to it in Visual Studio 2010 promise to make developers even more productive.

In previous Visual Studio versions, even after you entered characters to refine an Intellisense search list, you still had to search through the list for the grouping you were looking for. For instance, if you

wanted to see all classes that contain the word “Exception” in the *System* namespace, you could refine the Intellisense list. But the only way to actually find the group of classes with “Exception” in the *System* namespace was to manually scroll through all the Intellisense items to find where your target items were grouped.

With the improved Intellisense for Visual Studio 2010, if you want to find all the classes in the *System* namespace that contain “Exception,” all you do is type “System.Exception,” and the list is populated with every class in the *System* namespace that contains the word “Exception.”

Essentially, Intellisense now performs a *strstr*-style search instead of a *strncmp*-style search. It's a simple improvement that leads to great productivity gains.

—Robby Powell, group product manager, GrapeCity

Do More With Multimonitorers

Multiple monitor support in Visual Studio 2010 lets me use both my monitors and see code in full screens. The split screen feature that came out with Visual Studio 2008 was nice, but I could only work on one form at a time and on one monitor. Because I had to keep scrolling the different split screen, I didn't use that feature to its full extent.

Multiple monitor support will let me view as much code as possible using both my monitors. And it's simple to use; I can take any embedded tab item in Visual Studio — a designer, code, or markup — tear it from the main window and drag it anywhere I want.

—Jason Beres, product management director, Infragistics

Think Parallel

Visual Studio 2010 debugging windows — Parallel Stacks and Parallel Tasks — provide invaluable information about the way tasks and threads interact to run code in parallel.

The Parallel Tasks window displays a list of all the tasks and their status — scheduled, running, waiting, waiting-deadlocked, and so on — providing a snapshot of what's going on with each one. You can order and group information shown in the windows. You can double click on a task name to access the code that's being run by it.

The Parallel Tasks grid shows a column, or Thread Assignment, that displays the ID for the thread shown in the Threads Windows, letting you see which managed thread is supporting a task's execution.

The Parallel Stacks window displays a diagram with all the tasks or threads, their status, and their relationships. Switch from the Tasks view to the Threads view to see the whole map of what's going on with tasks and managed threads in your parallelized code.

—Gaston Hillar, IT consultant

Silverlight Shines

The single most-useful new feature is the design-time support and property window available for Silverlight applications. While I use Expression Blend a lot and still prefer it for creating the initial lay-

out of screens, the integrated Visual Studio 2010 visual designer for Silverlight definitely saves time.

Silverlight 3 had no designer support within Visual Studio, so having a way to visually move controls around, change properties, and even hook-up data bindings is welcome. I can open an XAML file and directly adjust properties of controls. Data bindings can be set directly through the data-binding window.

If you're a Silverlight developer, you'll use the new visual designer a lot. It's not as powerful as Expression Blend and doesn't let you do things like edit control templates, but it does provide a more efficient development workflow that boosts productivity.

—Dan Wahlin, founder, The Wahlin Group

Built-In UML

With UML design tools now built-in, no longer do you need to depend on Visio, Rational Rose, or other UML design tools to create UML diagrams. Now you can do it from within Visual Studio.

Visual Studio 2010 provides support for Use Case, Class, Sequence, Activity, Component, and Layer diagrams. You can use the built-in UML modeling tools for conceptual and logical analysis during the software development cycle.

And, there's a new project type called “Modeling Projects” in your “Add New Project” dialog to control and organize the UML diagrams. Moreover, you can save your UML diagrams as images or in XML Paper Specification (.xps) formats.

—Joydip Kanjilal, software consultant

F# Made Easier

When learning a new programming language, immediate feedback from an interactive console helps. That's certainly the case with support provided for F#.

Modern functional languages such as F# Interactive are becoming the paradigm of choice for working with multicore processors. Visual Studio 2010's F# interactive window lets you to run F# code interactively or execute F# scripts. It executes a read, evaluate, and print (REPL) loop cycle for the F# language. Granted, you can run the fsi.exe console application, but the F# Interactive window is more convenient and instructional.

To access this new window, select View, Other Windows, F# Interactive in the IDE's main menu. Once you enter two semicolons (;;) to terminate a line or several lines of code, F# Interactive attempts to compile the code. If successful, it executes the code and prints the signature of the types and values that it compiled. Of course, the interpreter will print error messages if something goes wrong. Code entered in the same session has access to previously entered constructs, a useful feature to test code before creating an F# program.

—Gaston Hillar, IT consultant

Postmortem Debugging

If you are like me, you spend a great deal of time doing “post-mortem” debugging. What that means is that I don't debug a live

process per se, rather a snapshot of a problematic process. Most all of my time I spend debugging these problems using the Debugging Tools for Windows debuggers (i.e., ntsd, cdb, windbg), which are incredibly powerful debuggers with unprecedented commands. At times though, it can get a little tedious and I find myself wondering why I can't have the Visual Studio debugging experience while debugging some of these issues. Well, I wonder no longer. Visual Studio 2010 supports postmortem debugging (i.e., crash dump debugging) with all the bells and whistles that go along with the Visual Studio IDE experience. Switching between frames in a call stack? No problem. The call stack window is just a click away. Do you want source code access? No problem. Setup your source path and Visual Studio will automatically bring up the source code.

—Mario Hewardt, author of *Advanced .NET Debugging*
(<http://my.safaribooksonline.com/9780321584090>)

More Accurate Multitargeting

Multitargeting is a feature that lets you develop applications that target various versions of the .NET Framework — versions 2.0,

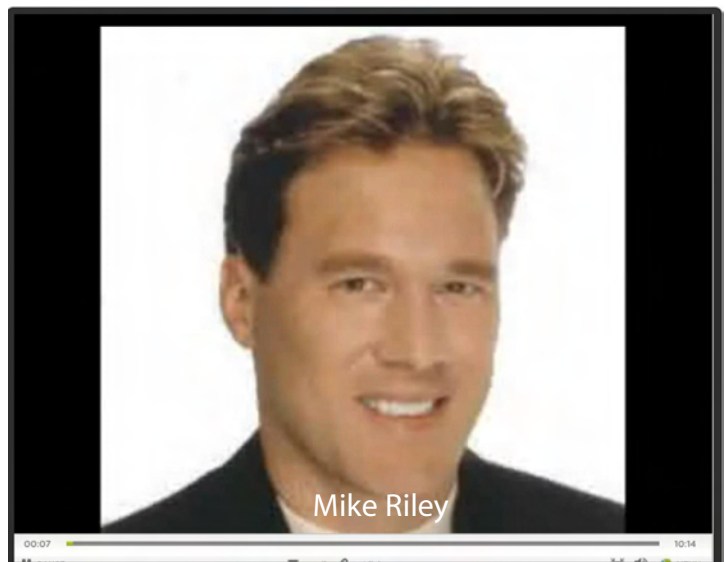
3.0, 3.5, and now 4.0. While multitargeting was available with Visual Studio 2008, it had its warts. For instance, the IDE would on occasion display information not available in the particular version of the .NET Framework your application was targeting. With Visual Studio 2010, you have in your toolbox only items that are specific to the particular version of the .NET Framework that you're focusing on. So, if you are developing an application targeted at .NET Framework 2.0, you won't see your toolbox displaying types that belong to the later versions of the framework. Moreover, in Visual Studio 2010, if you change your project to retarget another version of the .NET Framework that is not available in the system you are working on, you will be prompted to choose the version you need; i.e., the target framework.

—Joydip Kanjilal, software consultant

[Return to Table of Contents](#)

Learn More About Microsoft's Visual Studio 2010

To learn more about Visual Studio 2010, visit **Dr. Dobb's Visual Studio 2010 Channel** at www.drdobbs.com/visualstudio, where you'll find feature articles, news, blogs, videos, and tweets devoted to all things VS 2010! In the video featured here, Dr. Dobb's contributing editor Mike Riley talks with Embarcadero's Mike Rozlog about VS 2010 and Delphi, and previews Delphi Prism 2011. Click the image to play the video on **Dr. Dobb's Visual Studio 2010 Video Library** at www.drdobbs.com/tv/ddjvisualstudio2010.html.



Click screen to link to video

Developing for the iPad

Ideate is an iPad app for turning ideas into products

By Jonathan Branam

Six weeks before the launch of Apple's iPad, I approached my company, EffectiveUI (<http://www.effectiveui.com/>), about building an application for the iPad that would be useful for our particular business — designing and developing user interfaces. Typically our work begins with a number of meetings between our user-experience architects, designers, and developers, as well as the client's stakeholders and users. We sit and chat, we have conference calls, and we generate piles of notes and sketches to keep track of what we learn and how we visualize their new UI.

We use sketch pads for this work as well as taking notes on our laptops, but the end result is a stack of loose papers that we end up needing to keep with us or scan into our computers. The iPad is clearly a device that can help with this problem, and so Juan Sanchez and I pitched the idea of an application designed for the idea generation process of any activity, whether it's user interface design, planning your home landscaping, laying out an ERD, or designing a line of high-fashion clothes.



Figure 1

Ideate Development

With approval in hand, I began developing “Ideate” as my first large CocoaTouch application. I had previously learned iPhone SDK development just enough to launch a basic application and render to some OpenGL ES layers. Building Ideate was a learning experience as much as anything, and I’ll share some of the lessons learned in this article.

At this writing, iPad development is only possible on Mac OS X version 10.6.2. To develop for the iPad, iPhone, or iPod Touch you must join the iPhone development program and download XCode 3.2.2 with iPhone SDK 3.2. If you want to test on an actual iPad or iPhone and submit to the App Store, you also need to become a paid member of the Apple Developer Program (<http://developer.apple.com/>), which currently costs \$100/year. During development you can run your application in the iPhone Simulator, which also simulates an iPad device in version 3.2 (the current device can be changed in the menu of the simulator under Hardware → Device).

Designing for a Touch Tablet

The biggest difference in iPad development isn't the new language or the new SDK. The biggest difference is designing a good user interface for a tablet device that is driven entirely by touch. Since the iPad is such a novel device, application developers are still learning the best ways to use the 9.5-inch touch screen. As we began designing the UI for Ideate, we quickly found that it was difficult to predict how a user would interact with the device. We asked ourselves a number of questions: How far can the users' thumbs stretch over the black bevel? Is it comfortable to hold in one hand and draw with the other? How big is the screen really?

To address these concerns, I went into my garage and cut a piece of poplar to the dimensions of the iPad, sanded the back down to a comfortable bevel, and taped a mockup of the iPad onto it (see Figure 2).

The end result was a physical prototype we could hold in our hands. The top side of the prototype was left open so different sheets of paper could be inserted into it. We were able to use this prototype to get a better sense of the working area and the dimensions of the device. The poplar board is much lighter than the real iPad, however, so we couldn't test the weight.

Using our prototype, we designed an interface that was natural while holding the iPad in the left hand and manipulating the user interface primarily with the right hand. The left hand thumb can read the tool tray and quickly change the active tool while the right hand is sketching. We also quickly realized that the tool tray needs to dock on the left or the right. Using this prototype helped us make good decisions before we were able to hold the device for ourselves.

Custom Controls, Forget Interface Builder

Our first design for Ideate included several custom controls. In particular, we did not use the standard that anchors to the top or bottom of the window; instead we designed a menu that collapsed into itself to give the user the entire screen to use as a sketching canvas. Laying out a standard *UIToolbar* is very easy using the Interface Builder application. It is trivial to drag a new *UIToolbar* into the XIB and then drag buttons, flexible space, and fixed space *UIBarButtonItem*s onto it. Being new to the iPhone SDK, I spent a lot of time trying to get our custom menu to work the same way inside of Interface Builder.

I ran into numerous problems with this approach because to hide the buttons when the menu collapsed, I found it necessary to add a *UIView* into the view hierarchy inside the menu but above the buttons. When I tried to do this in my code, I lost the button layout from Interface Builder; when I did this in Interface Builder it created a hierarchy that was very difficult to navigate. My solution was further frustrated by losing the nice layout options in the *UIToolbar* and being forced to use absolute positioning to align my buttons. The last frustration was that my custom views in Interface Builder rendered as white (since they were drawn in code) and my buttons were only white text against the background, so they were all invisible in Interface Builder until clicked on.

My final determination was that Interface Builder is not the tool for laying out highly custom user interface elements. In the end, working on the custom menu took too much time away from more crucial development tasks, so we returned to using standard *UIToolbar* menus. Using the standard menu allows Ideate look like other iPad applications and helps the user know what to expect from the toolbar. We also used Interface Builder for building our popups since they used standard UI elements.

Later in the project, we implemented a custom tool tray. For the tool tray we did not make use of Interface Builder, but instead built

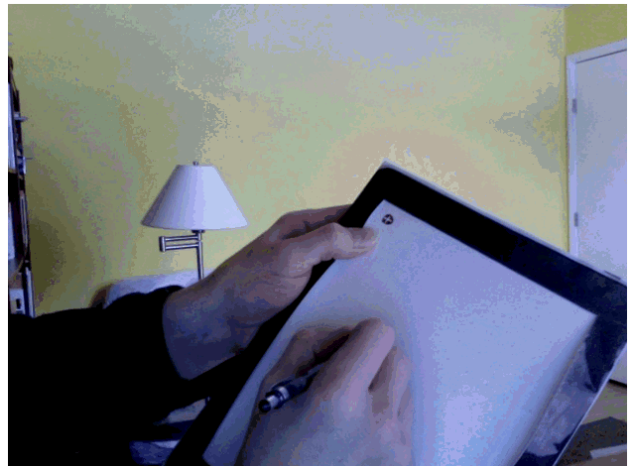


Figure 2



Figure 3

it all in code. This approach was appropriate for the task. In Figure 3, you can see the brush edit popup that is built in Interface Builder, and the tool tray on the left, which is all built in code.

Save Fast and Save Often

The concept of continuous, incremental saving is crucial to the operation of every iPhone OS application. Users on the phone may be interrupted at any time to receive a call and may decide to exit your application at any time. Apple frowns on applications that offer an explicit Save or Load button, instead encouraging developers to save while the user works. For Ideate, I designed a custom file format based on the PNG image file format

(<http://www.libpng.org/pub/png/>). The PNG format supports custom chunks in the file that can contain any data required by an application. Adobe Fireworks uses this approach to store all of its metadata in a PNG file and I thought it was a good idea. This means that browsers, image editors, and mail clients can all render an Ideate image, but only the Ideate app itself will load and save the custom chunks of data inside the file.

I chose the libpng library (<http://www.libpng.org/pub/png/libpng.html>) to implement this solution since it is widely used and supported across many platforms. I also found references to developers using it successfully in existing iPhone applications. After some frustrations getting libpng compiled for both ARM and i386 for use in the iPhone Simulator and on the device, things seemed to progress well. I implemented an initial approach that stopped everything to save the sketch to disk. This was clearly not ideal, so I later moved the saving code to a background thread to keep the application responsive. For safety, I used a locking mechanism on both threads so that the saved sketch would always be consistent. I declared a *lock* object inside my *Sketch* model object like this:

```
NSRecursiveLock *sketchLock;
```

I then wrapped every operation that changed the state of the sketch in a pair of *lock* and *unlock* messages:

```
-(void)insertLayer:(SketchLayer *)sketchLayer
atIndex:(NSUInteger) index {
    [sketchLock lock];
    [self setupNewLayer:sketchLayer];
    [sketchLayers insertObject:sketchLayer atIndex:index];
    [sketchLock unlock];
    [self sketchLayerChanged:sketchLayer];
}
```

The save operation was also wrapped in *lock/unlock* pairs so that it would be an atomic operation. I then ran a timer that would save the sketch every 15 seconds if it changed, and also implemented a call to save when the application terminated. All of this seemed like a great solution and it worked perfectly in the simulator. There was very little interruption during saving and the saving and loading appeared robust. However, I was in for a surprise when I first got an iPad and was able to use our application on it.

Don't Trust the Simulator

What's the first and possibly most important rule of iPad/iPhone development? Don't trust the simulator. The simulator is a fantastic piece of work, and I still use it for most of my work because it loads and runs quickly. However, it is this speed and access to lots of memory that can be deceptive.

When using the device, I didn't notice a pause during background saving, but the actual sketching was not very smooth. The device was doing a lot of work to render the drawing path and composite it with the other layers in the sketch during every screen update. I am still in the process of optimizing the drawing code.

However, I found that, even worse than slow drawing, sometimes the application would corrupt a saved sketch. It didn't take

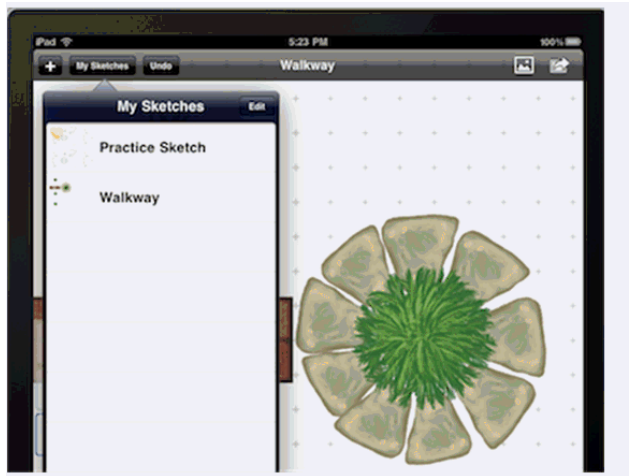


Figure 4

too long to find the problem. When the iPhone OS tells your application to shut down, it isn't sending a polite request, it is informing your application that doomsday is nigh. If your application does not shut down within a few seconds (Apple doesn't specify the exact time limit), it will be forcibly killed.

On the iPad, my saving code did not always complete within the timeout period. Because I used libpng directly to save the files, the application would open the file, write out some data, and then the process would be killed mid-save. When the user returned to the application, the sketch would be corrupt and fail to load.

There are several possible solutions to this issue. The first, immediate solution that I implemented was to do all file I/O on a temporary file so that sketch files would not become corrupt. When the sketch was fully saved, the application then deletes the old file and renames the new file to the sketch file. This largely solves the file corruption issue, but the application still has a problem in that it can lose data if the user has been editing the sketch and shuts down the application before the 15-second save window occurs.

I can reduce the impact of this further by reducing the length of the save window, at the expense of potentially more frequent pauses in the application during use; however, this isn't very palatable either. I can further reduce the impact of the background save by implementing a more granular locking system on the sketch. If a sketch can be locked for a very short window and a read-only copy made of it, then the save can proceed in the background while leaving the sketch available for editing by the user.

I think the real solution is to use a native iPhone OS saving solution rather than my custom PNG solution with libpng. Saving to a PNG for export is still a good idea, but there is no reason that the internal format should be PNG. The sketch can be saved using Core Data (<http://developer.apple.com/iphone/library/referencelibrary/GettingStarted/GettingStartedWithCoreData/index.html>) or SQLite (<http://www.sqlite.org/>), which are both very fast and can be

implemented incrementally much more readily than writing to a large file on disk. A third solution would be to break down the large PNG file into several smaller files that are read in sequence to reconstruct the sketch data. The sketch would be stored as a directory of files and potentially a simple SQLite database or even a binary plist (<http://developer.apple.com/mac/library/documentation/Cocoa/Conceptual/PropertyLists/Introduction/Introduction.html>) to reference the files. I will be experimenting with different solutions and improving Ideate's file management in an upcoming release.

Undo Support: Correcting Mistakes

For many applications, allowing users to undo their work is a crucial piece of functionality. For Ideate it was essential. The process of sketching out an idea is iterative and full of false starts. When I began to think about implementing undo, my mind swam with visions of commands stacked into an undo and a redo stack, and the frustrating process of wrapping all user interactions into these reusable nuggets of state-changing operations.

However, thankfully for me and early Ideate users, Apple has already implemented the necessary functionality and provided it since the iPhone 3.0 SDK. This comes in the form of the *NSUndoManager* and it's a wonderful helper class. *NSUndoManager* supplies an easy interface to managing undo and redo. First, you create a new manager for each "document object" in your system. For Ideate, this is the *Sketch* object. In the *init* method of *Sketch*, I did that:

```
undoManager = [[NSUndoManager alloc] init];
```

The next step is to tell the undo manager that a new operation has occurred that can be undone. There are two ways to do this, but the first way (explained below) is a little more straightforward and works great as long as the operation to undo only requires a single parameter. Inside a *Sketch*, there is a method that adds the newly sketched *path* to the array of paths. This code looks something like this:

```
- (void) appendPath:(DrawPath *)path {
    [paths addObject:path];
    [self setNeedsDisplay];
}
```

This allocates a new *DrawPath* and appends it to the list of paths to draw. Removing a path is also simple and the code is contained in the *removePath* method:

```
- (void) removePath:(DrawPath *)path {
    if ([paths containsObject:path]) {
        [paths removeObject:path];
        [self setNeedsDisplay];
    }
}
```

This removes the requested path and marks the view as needing to redraw itself now that it has changed. These two operations are inverses of each other, and are exactly what we need to implement undo using the *NSUndoManager*. Here is the updated *appendPath* method with the undo functionality added to it:

```
- (void) appendPath:(DrawPath *)path {
    [paths addObject:path];
    [undoManager registerUndoWithTarget:self
      selector:@selector(removePath:) object:
path];
    [self setNeedsDisplay];
}
```

The *registerUndoWithTarget: selector: object:* message gives the *NSUndoManager* all the information it needs to undo the current operation. The target is a pointer to the object and the selector is the name of the message to send to the target. A selector is really just the name of the message — it isn't equivalent to a function pointer in C. The *object* parameter specifies the single parameter to pass along with the message.

After this has been called, the operation can be reversed simply by sending *undo* in response to the user pressing the button in the UI:

```
-(IBAction)undoButtonPressed:(id) sender {
    NSLog(@"undo button pressed");
    [sketch.undoManager undo];
}
```

If your application supports redo, the *NSUndoManager* will automatically build a redo stack when it processes the undo. Any call to the *NSUndoManager* during an undo will add to the redo stack and a call to *NSUnderManager* redo will call those operations. Adding undo/redo support to *removePath:* looks like this:

```
- (void) removePath:(DrawPath *)path {
    if ([paths containsObject:path]) {
        [paths removeObject:path];
        [undoManager registerUndoWithTarget:self
      selector:@selector(appendPath:) object:path];
        [self setNeedsDisplay];
    }
}
```

Communication Through Animation

Animation is also an important topic for any iPad or iPhone application. Meaningful animations can help your users understand what the application is doing and how to interact with it. Animations can show the user what they just did (such as deleting or adding a row in a table, or changing a property) and also help them understand how that change affected the rest of the application. In Ideate we have a few animations and are planning to add more as they are needed.

One current animation in Ideate hides the tool tray when the user is editing an image clip. Image clips can be edited with a "long press," when the user presses and holds their finger on the clip for about half a second. Enabling animations is as simple as telling the view to start animating changes, then telling it when you're done making changes:

```
- (void) hideAnimated {
    [UIView beginAnimations:nil context:NULL];
    [UIView setAnimationDuration:0.15];
    CGRect frame = self.view.frame;
    frame.origin.x = -TOOL_TRAY_WIDTH;
    self.view.frame = frame;
    [UIView commitAnimations];
}
```

This code begins animations, then sets their length to 0.15 seconds, moves the tool tray off screen to the left, then commits animations. The Core Animation framework (available at http://developer.apple.com/iphone/library/documentation/Cocoa/Conceptual/CoreAnimation_guide/Introduction/Introduction.html) records all of the changes to animatable properties that occur between calls to *beginAnimations* and *commitAnimations* and then animates them over the duration of the activity.

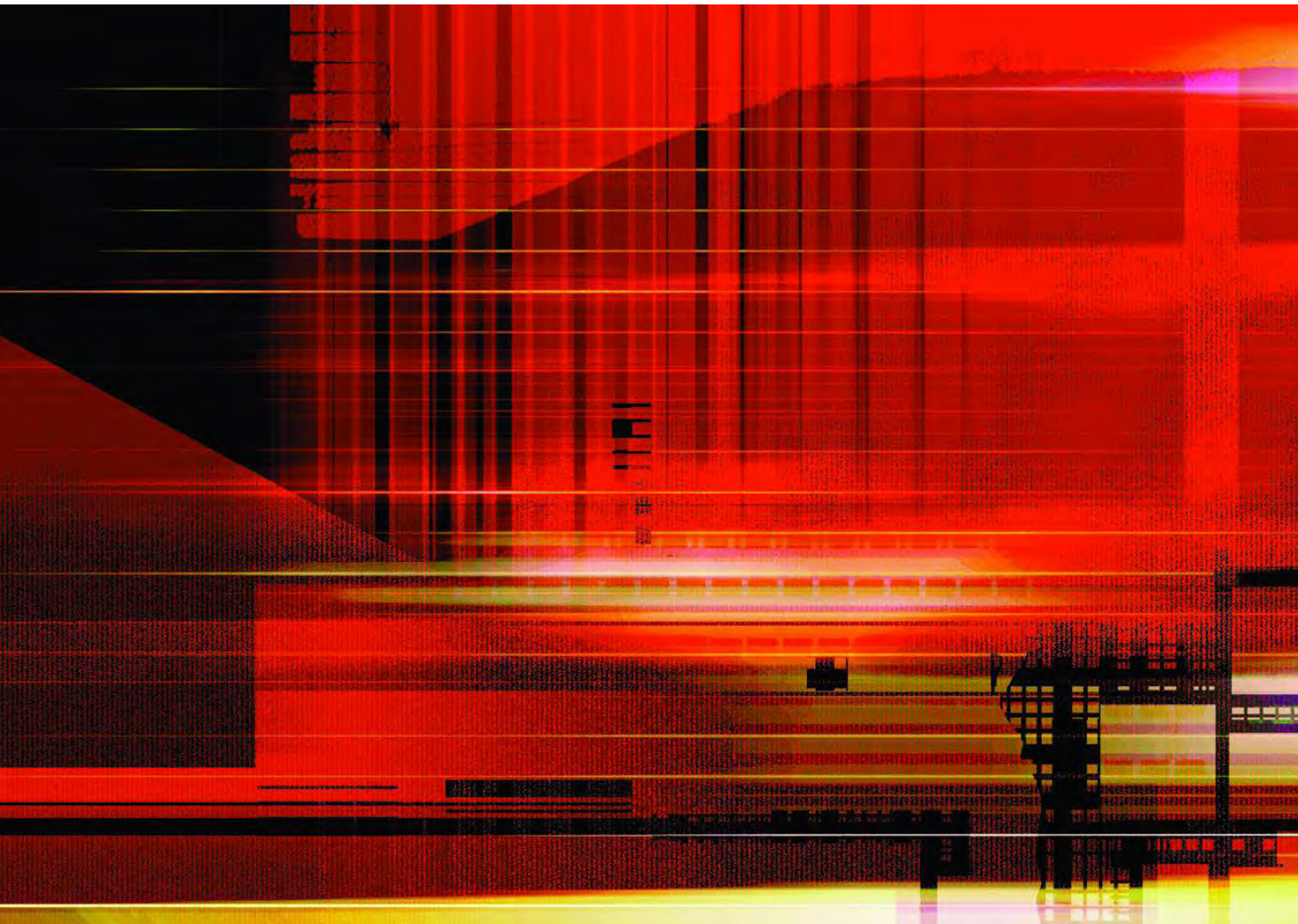
If you read the value of *self.view.frame* immediately after this code, it will look like the animation has already completed. Some animation systems will report the current value of a property while it is animating, but it is more convenient if the final value is reported. This is particularly true if you have code that operates further on animated properties, or if your code checks to see whether it needs to move the view further. If the current value of a property is required, it can still be obtained by asking the main *CALayer* of the *UIView* for its *presentationLayer*.

Bringing It All Together

The iPhone SDK is designed to help you build a great application for the iPad, iPhone, or iPod Touch. Read Apple's Human Interface Guidelines and Programming Guides for the best overview on application development issues. As you work on your application, be aware of how your technology solutions impact the user experience. At EffectiveUI, we are proud to have developed Ideate for the iPad launch, and we expect to leverage our experience to further improve the app and to build future apps as well.

— Jonathan Branam is a senior software developer for EffectiveUI.

[Return to Table of Contents](#)



JavaFX Database Programming with Java DB

Who said JDBC programming with Java and JavaFX Script was difficult?

By Eric J. Bruno

The promises of JavaFX include the ability to quickly and easily create Java GUI applications; run on desktops and mobile devices without changing source code; and the integration of Java code. With it, you can build media-rich applications with far less code than you would with Swing, yet you can still call into your existing Java codebase. This includes databases via JDBC as well, but there are some tricks you'll need to know, which we'll explore here.

Assume you've been asked to build a library front-end system to allow patrons to borrow and return books. The requirements include support for multiple operating systems (i.e. Windows, Mac OS X, and Linux). Java helps to solve that easily, and the addition of JavaFX solves the multimedia requirements as well (such as support for book cover photos, 30-second music samples, and video clips from movies). For our solution's UI, we'll choose JavaFX. The sample code I examine won't include the multimedia features; instead I focus on JDBC integration.

Next, to track library patrons and the books they borrow and return (hopefully), you need a database. For this sample application, I use Java DB since it's open-source, readily available with the Java Development Kit (JDK), and easy to use. Since JavaFX integrates with Java, we should be

able to integrate the UI code with the JDBC code with little effort. First, let's discuss how to get started with JavaFX and Java DB development. The complete source code and related files for the Library Application are available at <http://i.cmpnet.com/ddj/images/article/2010/code/LibraryApp.zip>.

JavaFX Primer

To begin, you must have JavaFX installed on your development machine. You can find the appropriate download bundle at <http://javafx.com/downloads/all.jsp>. If you don't have NetBeans 6.8, it's recommended you download the NetBeans 6.8 and JavaFX bundle; otherwise you can download and install the JavaFX SDK only. If you're committed to Eclipse, you can download the JavaFX plug-in for Eclipse from this page as well. The latest version of JavaFX as of this writing is version 1.2.3.

If you own a Windows Mobile device, you can download JavaFX Mobile to install and run JavaFX applications there as well. For this article, I focus on running JavaFX on the desktop.

If you already have the latest Java and NetBeans installed, you can install the JavaFX plug-in for NetBeans via the Plugins window from the Tools menu (see Figure 1). You'll still need to install the JavaFX SDK separately.

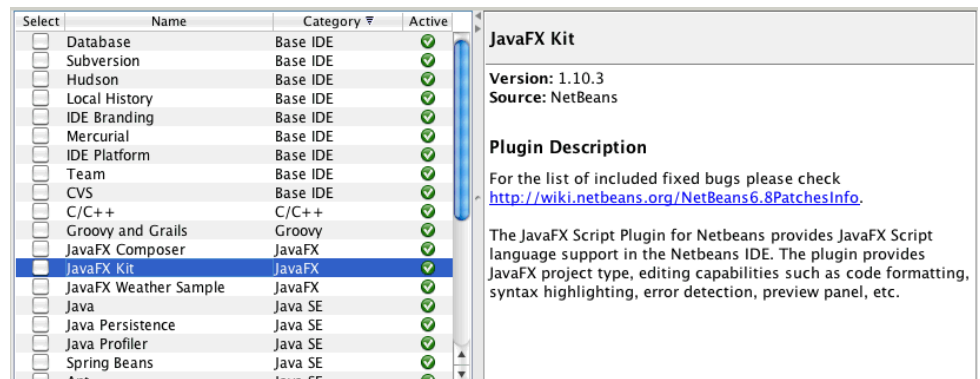


Figure 1: The Plugins window lets you add new features (such as JavaFX development) to NetBeans.

With the JavaFX Kit plug-in, NetBeans allows you to create new JavaFX projects, and the editor will support JavaFX Script development (see Figure 2). This includes syntax highlighting, code completion, in-line error detection, and source code debugging. Project properties allow you to specify how the application is run, such as with Java WebStart, standalone, within a browser, or on a mobile device. You can also specify, if you choose to deploy as a browser-based JavaFX applet, where the applet can be dragged off the browser and onto the desktop — a convenient feature for JavaFX application installation.

You mainly develop JavaFX applications with JavaFX Script, which is an expressive dynamic language that simplifies the GUI development process. It's also meant to be a straightforward way for non-programmers (i.e. graphic artists) to design user interfaces and content to be shared with programmers. More involved logic can still be built with Java, and easily integrated with the JavaFX Script UI code. Example 1 shows a default, very simple, JavaFX application.

```
Stage {
    title: "Application title"
    scene: Scene {
        width: 250
        height: 80
        content: [
            Text {
                font : Font {
                    size : 16
                }
                x: 10
                y: 30
                content: "Application content"
            }
        ]
    }
}
```

Example 1: A basic JavaFX application created by the NetBeans New Project wizard.

Stage is analogous to a window frame, title bar, and other decorations. You can customize these, or turn them on or off, by manipulating properties such as *title*, as shown in this code. The *Scene* represents and contains the content within the window, as well as the overall window size. In this example, the content is a text label which itself has the words *Application content* as its content. There are many JavaFX controls to choose for additional content.

You include more content in the *Scene* by adding more GUI components to the content property between the opening and closing square brackets, which represent an array of items in JavaFX Script. You control the layout of these items through a combination of horizontal, vertical, and grid layout containers, as well as explicitly setting the *x* and *y* placement of each component.

When compiled and run, the script in Example 1 results in an application that resembles Figure 3.

The JavaFX *Stage* results in the window frame and title bar with the text *Application title*. The *Scene* results in the content within the window (outlined in black), such as the text *Application content* in this example. Although the layout of components in JavaFX Script is straightforward, a graphical layout manager makes the task even easier; this is where the JavaFX Composer tool comes in.

The JavaFX Composer

For complex or detailed window layout, such as data entry forms or other detailed displays, the JavaFX Composer makes the task quick and easy. The Composer tool is installed as a NetBeans plug-in, available via the Plugins window (see Figure 1). The “Palette” — in the upper-right corner in Figure 4 — contains the available JavaFX Controls, Shapes, Containers, Effects, and other components that can be dragged onto the window you’re designing. You can then drag the components around on the window for precise placements, and size them appropriately.

When a component is selected, its programmable properties are displayed in the “Properties” editor — the lower-right corner of Figure 4. Here you can set properties such as the name of the

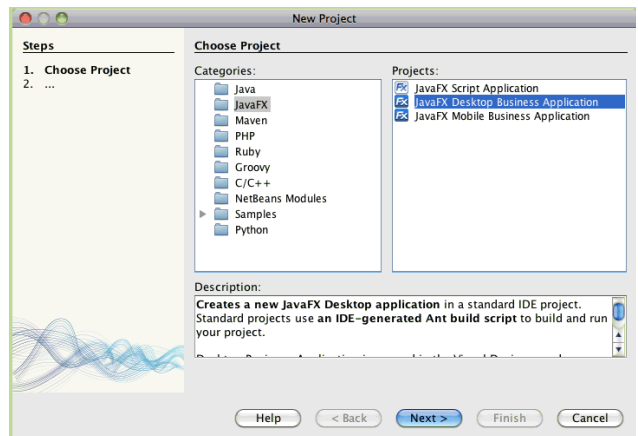


Figure 2: With NetBeans plug-ins, there’s support for multiple programming languages. Now you can add JavaFX to the list.

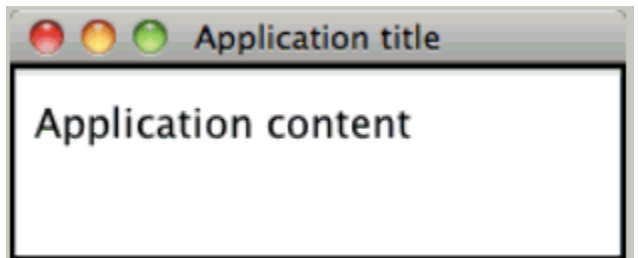


Figure 3: The code from Example 1 results in this application window when run.

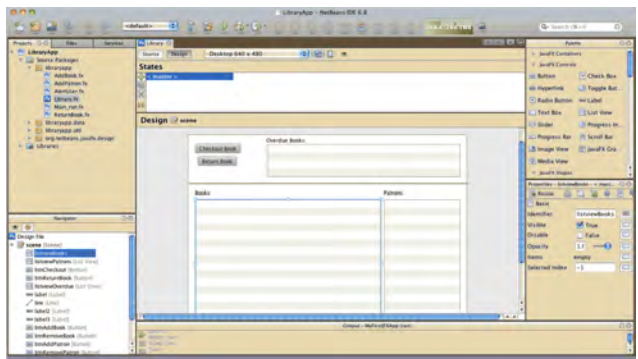


Figure 4: Laying out controls within a window with the NetBeans JavaFX Composer tool.

component, label text, and methods that are called when certain actions occurs (i.e. clicking a button).

The window shown in the Composer tool in Figure 4 is for the Library application I'll examine throughout this article. It contains three main lists — one for all books in the library; a second for the list of library patrons; and a third to highlight books that are overdue. There are buttons to add new books and patrons to the library, as well as to allow patrons to borrow and return books. The data that populates these lists and enables all actions comes from a Java DB database, accessed through JDBC as with any Java application. Let's now take a deeper look at Java DB.

Java DB and the Library Application

Java DB, which is available with most Java Development Kit download bundles, Java EE download bundles, or as a separate download, is a robust relational database based on Apache Derby. The current Java DB release version is v10.5.3.0, which is based on Derby with the same version number. You can run Java DB in its own process, accessible to multiple network clients via the Java DB Client JDBC driver, or you can run it embedded within your application with the embedded JDBC driver. NetBeans contains a Services tab (next to the Projects tab that lists your development projects) that contains a section for Databases (see Figure 5).

When you right-mouse click on the Java DB entry under Databases, you can either start or stop the standalone Java DB serv-

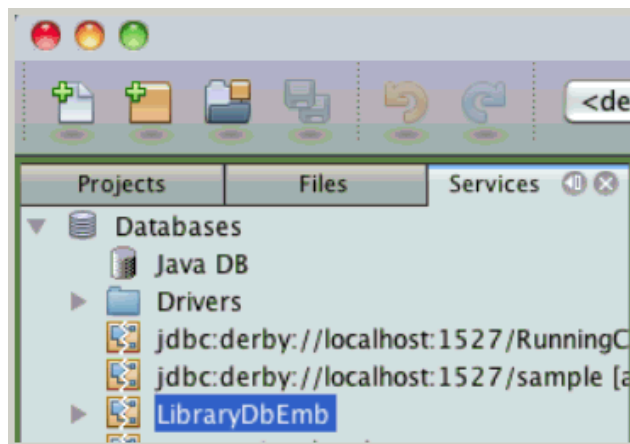


Figure 5: The NetBeans Services tab includes support for various types of databases (Java DB, MySQL, PostgreSQL, and so on).

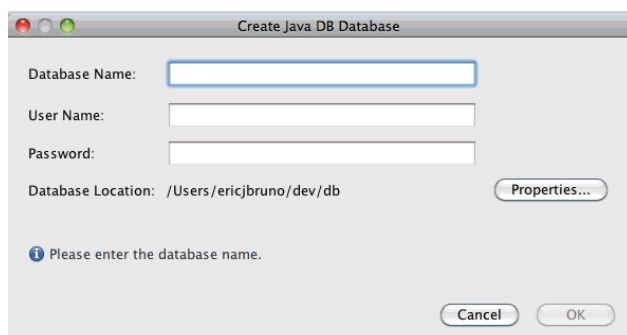


Figure 6: Creating a new standalone Java DB database with NetBeans.

er, modify properties of the Java DB server, or create a new database. When creating a database, you need to provide a name, along with a username and password to access to the database, and location to store the database files (see Figure 6).

However, we won't be using Java DB in standalone mode; instead we'll embed it within our application, and thus include all of the database creation code with the application code. When our Library application is run for the first time, it will create the database along with the users, tables, and indexes that we need. The advantage is that our application won't require a separate database installation process or any input from the user. The downside of embedded mode is that only one application can work with the database at a time. However, for certain applications, this choice works well.

For existing databases, NetBeans allows you to run queries to insert new data, update or view existing data, modify the database schema, and so on. First, you need to connect to the database. To do this, right-mouse click on Databases within the Services tab, and choose New Connection. Next, within the New Database Connection window, choose the driver named Java DB (Embedded), enter the path to the database files, and the username and password to access the database (see Figure 7).

Once connected, you can right-mouse click on tables within the database to view data, add columns, and perform other tasks. The results from queries are displayed within NetBeans, and can be edited in place (see Figure 8).

The JavaFX Library application we're building will include code to create and connect to a Java DB database. It does this through standard JDBC calls. For instance, Example 2 shows the code used to connect to an existing Java DB database named "librarydb".

```
Connection con = null;
try {
    System.setProperty(
        "derby.database.forceDatabaseLock", "true");
    String driver = "org.apache.derby.jdbc.EmbeddedDriver";
    Class.forName(driver).newInstance();
    String url = "jdbc:derby:librarydb";
    con = DriverManager.getConnection(url, "dbuser", "dbuser");
} catch (Exception e) {
    ...
}
```

Example 2: The JDBC code to connect to librarydb.

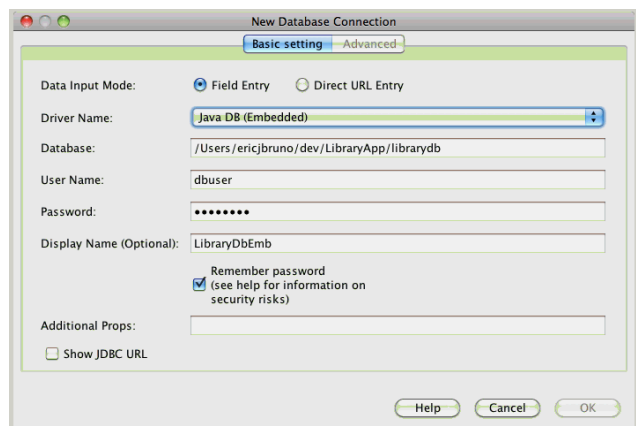


Figure 7: Connecting to a database in NetBeans allows you to easily query and update an existing Java DB/Derby, MySQL, or PostgreSQL database.

First, the code ensures that only one application can connect to *librarydb* at a time by setting the *forceDatabaseLock* property to true. Next, the code attempts to connect to the database with the URL *jdbc:derby:librarydb*, using the Java DB Embedded driver, *org.apache.derby.jdbc.EmbeddedDriver*.

If the connect attempt fails because the database doesn't yet exist, an *Exception* will be thrown. At this point, to create the database, the code will simply append the text *;create=true* to the end of the database URL, and once again attempt a connection. You can do this in the *catch* block of Example 2. As a result, Java DB will create the database and return a valid connection to it for our application to use. Next step: Create the tables.

Our Library application contains three tables; the first named BOOKS; the second named PATRONS; and the third to map books to patrons who have borrowed them (named CHECKED-OUT). Figure 9 shows the tables, columns, and their relation-

ships. When no books are borrowed, the CHECKEDOUT table will be empty. Otherwise when a book is borrowed, a row is added to the CHECKEDOUT table, and the CHECKEDOUT column in the BOOK table is set to the appropriate patron's ID.

These two relationships allow us to quickly know if a particular book is borrowed, get all of the books that are borrowed, as well as who borrowed them. Listing 1 contains the JDBC code to create all three tables.

```
static final String CREATE_BOOK_TABLE =
    "CREATE table APP.BOOK (" +
        "ID          INTEGER NOT NULL " +
        "PRIMARY KEY GENERATED ALWAYS AS IDENTITY " +
        "(START WITH 1, INCREMENT BY 1)," +
        "NAME         VARCHAR(50)," +
        "AUTHOR        VARCHAR(40)," +
        "PUBLISHER      VARCHAR(40)," +
        "CHECKEDOUT     INTEGER," +
        "ACTIVE         INTEGER );";

static final String CREATE_PATRON_TABLE =
    "CREATE table APP.PATRON (" +
        "ID          INTEGER NOT NULL " +
        "PRIMARY KEY GENERATED ALWAYS AS IDENTITY " +
        "(START WITH 1, INCREMENT BY 1)," +
        "FIRSTNAME     VARCHAR(25)," +
        "LASTNAME      VARCHAR(25)," +
        "ADDRESS        VARCHAR(40)," +
        "FINES          DOUBLE," +
        "ACTIVE         INTEGER );";

static final String CREATE_CHECKEDOUT_TABLE =
    "CREATE table APP.CHECKEDOUT (" +
        "ID          INTEGER NOT NULL " +
        "PRIMARY KEY GENERATED ALWAYS AS IDENTITY " +
        "(START WITH 1, INCREMENT BY 1)," +
        "BOOKID        INTEGER," +
        "PATRONID       INTEGER," +
        "MONTHDUE       INTEGER," +
        "DAYDUE         INTEGER," +
        "YEARDUE        INTEGER );";

private void createTables() {
    this.execQuery(CREATE_BOOK_TABLE);
    this.execQuery(CREATE_PATRON_TABLE);
    this.execQuery(CREATE_CHECKEDOUT_TABLE);
}

...
```

Listing 1: The JDBC code to create the Library database tables.

Let's dive deeper into the sample JavaFX Library application now to examine the code to query the database, add new books and patrons, and borrow and return books. We'll also see how these actions are mapped to user actions in the JavaFX application GUI.

The Library Application

All of the JDBC code to create the Library database, connect to it, query it, and otherwise add to or update it is contained within the *Database* class. This class is made a singleton by setting the constructor to private and adding a static *getInstance* method. The *Database* class is part of the *library.data* package within our application project, along with the *Book* and *Patron* classes, which match their respective database tables.

In JavaFX, variables are declared with reverse syntax compared to Java. For instance, the following line declares a variable of type *String*:

```
var name: String;
```

The following declares and assigns a variable in JavaFX:

```
var name: String = "Eric Bruno";
```

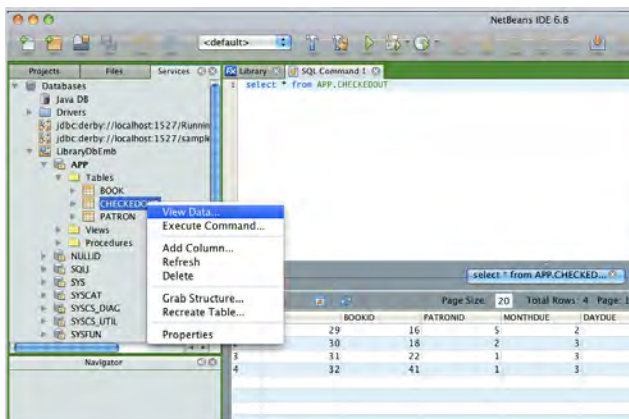


Figure 8: Connecting to a Java DB database, viewing data, and modifying rows are all a mouse-click away in NetBeans.

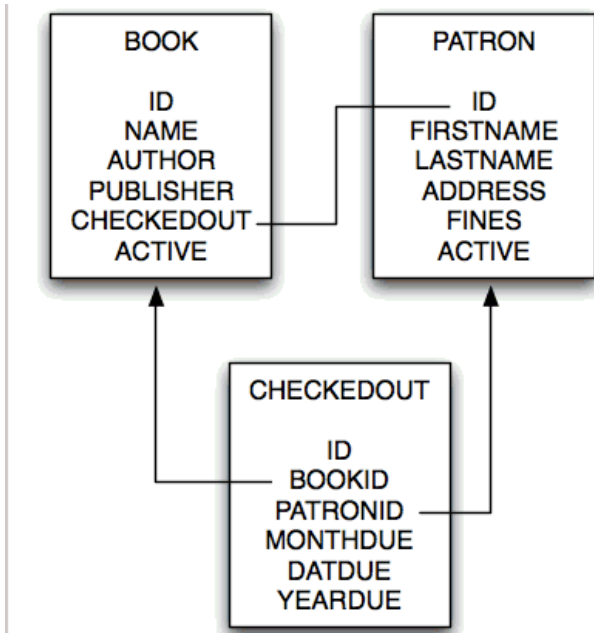


Figure 9: The Library application's database tables and relationships.

In our application, *Library* is the main class, which contains the JavaFX components, layout, and code for the main application window. In this class and others that need to access the database, a reference the singleton *Database* object is added:

```
public class Library {
    public-read var listViewBooks:
javafx.scene.control.ListView;
    ...
    var db: Database = Database.getInstance();
    ...
}
```

The database is used to populate the Books and Patrons lists, and is quite simple to do. For instance, the code below populates the Books list using the book's title, author, and checked-out status:

```
function populateBooksListView(): Void {
    // Get the latest list of books
    var bookArray: Book[] = db.getBooks();

    // Add books to the listview
    listViewBooks.items = null;
    for ( book in bookArray ) {
        var entry: String =
            "\"{book.name}\" (author: {book.author})";

        if ( book.checkedOut > 0 ) {
            entry = "CHECKED OUT: {entry}"
        }
        insert entry into listViewBooks.items;
    }
}
```

Let's look at the query behind the *getBooks* method. To make it easier to execute queries in JDBC and ensure all of the resources are cleaned-up, I've created a class called *QueryTool* (available for download at <http://i.cmpnet.com/ddj/images/article/2010/code/LibraryApp.zip>). It handles getting a JDBC Connection, which works with connection pools as well, and it ensures that transactions are committed and all resources are cleaned up when the query is complete. Example 3 shows *QueryTool* in action, returning a list of all of the books in our library.

```
public Book[] getBooks() {
    QueryTool query = null;
    try {
        query = new QueryTool(con);
        String req =
            "SELECT * FROM APP.BOOK ORDER BY NAME";
        PreparedStatement ps = query.setQuery( req );
        ResultSet rs = query.exec();

        Vector bookVect = new Vector();
        while ( rs.next() ) {
            Book book = new Book();
            book.id = rs.getInt("ID");
            book.name = rs.getString("NAME");
            book.author = rs.getString("AUTHOR");
            book.publisher = rs.getString("PUBLISHER");
            book.checkedOut = rs.getInt("CHECKEDOUT");
            book.active = rs.getInt("ACTIVE");
            if ( book.active > 0 )
                bookVect.add(book);
        }

        // Convert the Vector to an array
        Book[] books = new Book[bookVect.size()];
        bookVect.toArray(books);
        return books;
    }
    catch ( Exception e ) {
        e.printStackTrace();
    }
    finally {
        query.close();
    }
    return null;
}
```

Example 3: The Java DB query (with the *QueryTool* helper class) to retrieve all books.

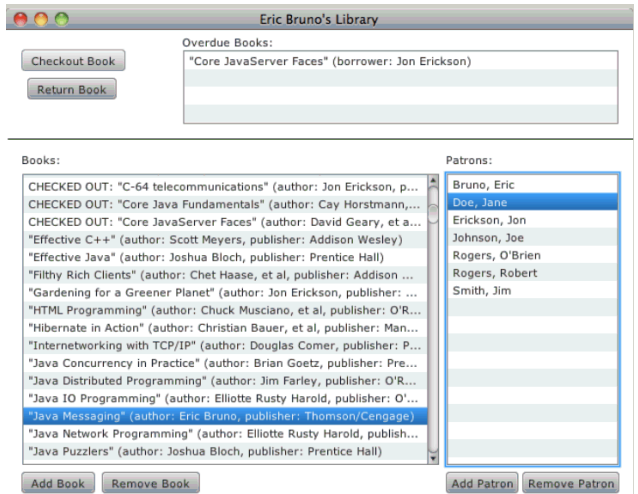


Figure 10: The Library application main window, where books can be borrowed and returned.

The *QueryTool* object is declared outside of the *try..catch..finally* scope as the *close* method is called in the *finally* block. This ensures that the *Statement* and *ResultSet* are closed even if the query raises an *Exception*. The SQL is straightforward, where all books are selected from the BOOK table, and sorted by the BOOK.NAME column (the book title). Each active book (a book may be marked inactive if it's never returned, say) is added to a *Vector*, which is converted to an array and returned to the caller (the JavaFX Script code in this case). For instance, this array is stored and displayed in the Books list in the main Library window.

To borrow a book, you must select an available book from the Books list, a patron from the Patrons list, and click on the button labeled Checkout Book (see Figure 10). An alert will pop up informing you whether or not the book was successfully checked out.

The code that processes the button click gets the selected book and patron objects, verifies that the book can be borrowed, and calls the database to borrow it (see Listing 2). The JDBC code to check out the book updates the BOOK record to set the CHECKEDOUT column to the borrowing patron's ID, and adds a row to the CHECKED-OUT table with both the patron and book IDs (see Listing 3).

```
function btnCheckoutAction(): Void {
    // Get the selected book
    var selected: Integer = listViewBooks.selectedIndex;
    if (selected == -1)
        return;
    var book: Book = bookArray[selected];

    // Get the selected patron
    selected = listViewPatrons.selectedIndex;
    if (selected == -1)
        return;
    var patron: Patron = patronArray[selected];

    // Checkout the book and update the books listview
    if (db.checkoutBook(book.id, patron.id) == true) {
        // Update the books listview to show the book checked out
        populateBooksListView();

        // Open a small window to verify the checkout
        var alert: AlertUser = AlertUser {
            style: StageStyle.TRANSPARENT
            text: "Book checked out successfully"
        }
    }
    else {
        // Open a small window to show it failed
        var alert: AlertUser = AlertUser {
            style: StageStyle.TRANSPARENT
        }
    }
}
```

```

        text: "Book was not checked out!"
        color: Color.RED
    }
}
}

```

Listing 2: The JavaFX code to checkout a book.

Notice that the Overdue Books list shows one book, borrowed by our very own Jon Erickson in this made-up example, and that books that are borrowed are displayed in the list with the text "CHECK OUT" preceding the title. These are all elements that reflect the state of the database.

```

public boolean checkoutBook(int bookid, int patronid) {
    Book book = getBook(bookid);
    if ( book == null )
        return false;

    Patron patron = getPatron(patronid);
    if ( patron == null )
        return false;

    // Is the book already checked out or lost?
    if ( book.checkedOut > 0 || book.active == 0 )
        return false;

    // Is the patron in good standing?
    if ( patron.active == 0 )
        return false;

    // Calculate the due date (today + two weeks)
    Calendar cal = Calendar.getInstance();
    cal.add(Calendar.DAY_OF_MONTH, 14);
    int day = cal.get(Calendar.DAY_OF_MONTH);
    int month = cal.get(Calendar.MONTH)+1;
    int year = cal.get(Calendar.YEAR);

    // Add the check-out record
    QueryTool query = null;
    try {
        query = new QueryTool(con);
        String req = "INSERT INTO APP.CHECKEDOUT " +
            "(BOOKID, PATRONID, MONTHDUE, DAYDUE, YEARDUE) " +
            "VALUES(?, ?, ?, ?, ?)";
        PreparedStatement ps = query.setQuery(req);
        ps.setInt(1, bookid);
        ps.setInt(2, patronid);
        ps.setInt(3, month);
        ps.setInt(4, day);
        ps.setInt(5, year);
        query.exec();
    }
    catch ( Exception e ) {
        e.printStackTrace();
        return false;
    }
    finally {
        query.close();
    }

    // Update the book record (checkedout field is the patron
    id)
    try {
        query = new QueryTool(con);
        String req =
            "UPDATE APP.BOOK SET \"CHECKEDOUT\"=\"" + patronid +
            " WHERE \"ID\"=\"" + bookid;
        query.setQuery(req);
        query.exec();
    }
    catch ( Exception e ) {
        e.printStackTrace();
        return false;
    }
    finally {
        query.close();
    }
    return true;
}

```

Listing 3: The Java/JDBC code to check out a book.

When a book is returned, the user clicks on the Return Book button. As a result, a second window is displayed showing all of the currently borrowed books (see Example 4).

```

function btnReturnBookAction(): Void {
    // Open a new window to return a book
    returnBook = ReturnBook {

```

```

        onClose: function () {
            returnBook = null;

            populateBooksListView();
            populateOverdueListView();
        }
    }
}

```

Example 4: The Return Books button handler code to open a new window.

The *ReturnBook* class extends *Stage* (see Listing 4); instantiating it opens a window as a result (see Figure 11). The *init* block creates the components of the *Scene*, which are arranged in a panel according to their *x* and *y* coordinates. Clicking on the button labeled *Return* updates the database and closes the child window.

```

package libraryapp;
import javafx.stage.Stage;
import libraryapp.data.*;
import javafx.scene.paint.Color;
import javafx.stage.StageStyle;

public class ReturnBook extends Stage {
    override var title = "Return Borrowed Books";

    public-read var label: javafx.scene.control.Label;
    public-read var btnReturn: javafx.scene.control.Button;
    public-read var listViewBooks: javafx.scene.control.ListView;

    var db: Database = Database.getInstance();
    var bookArray: Book[];

    function populateBooksListView() : Void {
        var allBooks: Book[] = db.getBooks();

        // Add only checked-out books to the listView
        listViewBooks.items = null;
        bookArray = null;
        for ( book in allBooks ) {
            if ( book.checkedOut > 0 ) {
                // The checkedout field is the patron id
                var patron: Patron =
                    db.getPatron(book.checkedOut);
                var dueDate: String = db.getDueDate(book.id);
                var entry: String =
                    "\"(book.name)\" (borrower: ";
                entry = entry.concat(
                    "{patron.firstName} {patron.lastName}");
                entry = entry.concat(
                    ") Due: {dueDate}");

                insert entry into listViewBooks.items;
                insert book into bookArray;
            }
        }

        init {
            label = javafx.scene.control.Label {
                layoutX: 6.0
                layoutY: 6.0
                text: "Checked out books:"
            };
            btnReturn = javafx.scene.control.Button {
                layoutX: 413.0
                layoutY: 227.0
                text: "Return"
                action: btnReturnAction
            };
            listViewBooks = javafx.scene.control.ListView {
                layoutX: 6.0
                layoutY: 27.0
                width: 465.0
                height: 194.0
                layoutInfo: javafx.scene.layout.LayoutInfo {
                    width: bind listViewBooks.width
                    height: bind listViewBooks.height
                }
            };
            scene = javafx.scene.Scene {
                width: 480.0
                height: 259.0
                content: javafx.scene.layout.Panel {
                    content: getDesignRootNodes ()
                }
            };

            this.populateBooksListView();
        }
    }
}

```

```

public function getDesignRootNodes () : javafx.scene.Node[] {
    [ label, btnReturn, listViewBooks, ]
}

public function getDesignScene (): javafx.scene.Scene {
    scene
}

```

```
function btnReturnAction(): Void {
    // Get the selected book and mark it as returned
    var selected: Integer = listViewBooks.selectedIndex;
    if ( selected == -1 )
        return;

    var success: Boolean =
        db.returnBook(bookArray[selected].id);

    close();
}
```

Listing 4: The Return Book window.

When the Return Books window is closed, the *ReturnBooks.onClose* method in Example 4 is executed. As a result, the lists of all books and overdue books are updated to reflect the changes.

JavaFX Native Arrays

Passing an object from JavaFX code to Java code is straightforward. However, when passing an array of Java objects, you need to explicitly declare the array as native. For instance, assume we modified the Library application where multiple books can be selected from the books list, and then borrowed by a single patron at once. We simply create an array of *Book* objects to be passed to the *Database* object. This array is declared as a *nativearray*, with the Java type specified as in:

```
var books: nativearray of Book;
```

To populate this array, we must use the `{ }` syntax, such as with the following *String* variable:

```
var bookInfo: String = "{book.name} {author: {book.author}}";
```

The expressions between the braces are evaluated, and their values placed into the *String*. For example, the above line of code may evaluate to the following:

```
Java Messaging {author: Eric Bruno}
```

Notice that the name of the book is inserted into the string in place of `{book.name}`, and so on. The same concept and syntax works in other areas of JavaFX Script as well, such as when populating native arrays. For instance, the code below is used to check out multiple books to one patron:

```
function checkoutMultipleBooks(): Void {
    // Get the selected patron
    var patron: Patron =
        patronArray[listviewPatrons.selectedIndex];

    var booksSelected = getBooksSelected();

    // Create a native array to pass to Java class
    var books : nativearray of Book = [ {
        for ( book in booksSelected ) {
            book;
        }
    } ] as nativearray of Book;

    db.checkoutBooks(patron.id, books);
}
```

Remember that all objects between the square brackets `[ ]` are treated as an array. This, combined with the braces `{ }` used to wrap

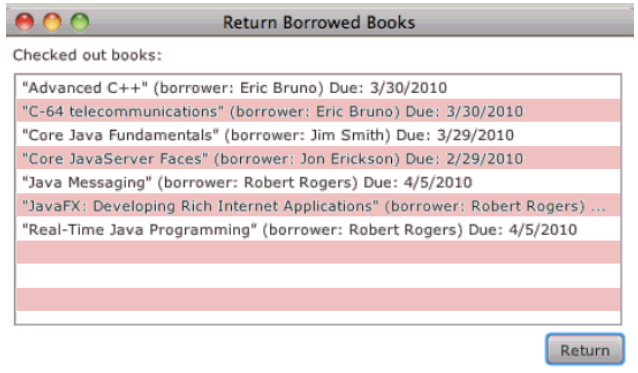


Figure 11: Instantiating a class that extends *Stage* results in a new window, such as the Return Book window.

the *for* loop, takes each *book* object from the *booksSelected* array and inserts them into the *books* native array.

Java DB Clean-Up

Since only one application may connect to the Library database at a time, it's important to disconnect when the Library application terminates. To accomplish this, the *onClose* method of the main window's *Stage* calls the *Database* class to execute the following:

```
DriverManager.getConnection(
    "jdbc:derby:;shutdown=true", usr, pwd );
```

The result is that the current database connection is broken, and it's safe to shut down the application. If you fail to do this, or the application terminates abruptly — i.e., it crashes or its process is killed externally — you'll need to remove the database lock manually in order to connect to it again. To do this, locate the database files (by default, it's in the same directory you execute the application from) and remove the file named *db.lck*.

Conclusion and Further Reading

Because JavaFX applications are also Java applications, integrating with Java classes, JAR files, and other Java APIs and technologies is straightforward. This includes JDBC. However, working with Java code from JavaFX Script requires some special consideration in some cases. This article has shown that JDBC programming with Java and JavaFX Script is possible and straightforward. It also served as a tutorial on how to use Java DB/Derby.

For more information on Java DB, Derby, and JavaFX, check out the following resources:

- Java DB Download at <http://developers.sun.com/Java%20DB/downloads/index.jsp>
- Apache Derby at <http://db.apache.org/derby/>
- JavaFX at <http://db.apache.org/derby/>
- avaFX: Developing Rich Internet Applications at <http://jfx-book.com/JFXBook.html>

— Eric Bruno is a contributing editor to Dr. Dobb's and can be contacted at eric@ericbruno.com.

[Return to Table of Contents](#)

The Art of Assembly Language, 2nd Edition

Reviewed by Mike Riley

The Art of Assembly Language, 2nd Edition

Randall Hyde

No Starch Press

\$59.95 US

I haven't seriously programmed in assembler for nearly 20 years. The last time I did was for a set of routines on the Motorola 68000 chip powering my long-gone Atari ST computer. The computing world has changed quite a bit since then and so have the number of programming options and frameworks. However, a lament made by a number of prominent CS educators is that abstracted frameworks lessen the understanding of what is really going on at the machine level. Does *The Art of Assembly Language* satisfy such educational deficiencies by effectively teaching lower level practices to the uninitiated?

Author Randall Hyde is an ideal educator to write on this subject. He is the author of the excellent two volume (so far) *Write Great Code* books published by No Starch Press (<http://nostarch.com/greatcode.htm>), coauthor of the Microsoft Macro Assembler (MASM; see <http://www.microsoft.com/downloads/details.aspx?familyid=7a1c9da0-0510-44a2-b042-7ef370530c64&displaylang=en>) book titled *MASM 6.0 Bible* (<http://www.amazon.com/Waite-Groups-Microsoft-Macro-Assembler/dp/0672301555>), and has also contributed articles to *Dr. Dobb's*.

The author teaches an abstracted, custom version of assembly called High Level Assembly (HLA) and offers versions of HLA for Linux, Mac, and Windows available for download from his website (<http://homepage.mac.com/randyhyde/webster.cs.ucr.edu/HighLevelAsm/index.html>). In addition to the cross-platform HLA compilers, a full electronic copy of the first edition of the book is available for download at <http://homepage.mac.com/randyhyde/webster.cs.ucr.edu/www.artofasm.com/index.html> as well. This begs the question of considering the hefty cover price of the print edition, especially in the era of Kindle and iPad. Even though the freely available HTML or PDF editions are adequate on the iPad, the print edition still excels at rapid spacial access to book passages and code samples. This will no doubt eventually favor electronic editions as more books migrate to these tablet-based digital reading platforms. But for now, a book of this size and this amount of information density is optimal in paper format and thus worth considering the cover price. Not to mention, its purchase rewards the author with monetary compensation for a job well done.

After reminding readers that what is presented in the book is not raw assembly language programming, Randall reaffirms that HLA is a learning platform for students to acclimate to the con-

structs behind the assembly language principles. As such, readers seeking to program the latest embedded systems in assembly will need further study beyond the book to attain that goal. If readers are seeking enlightenment in the rules, processes, and general syntactic understanding of assembly, HLA is an excellent educational choice. Refreshingly, there isn't a chapter or even appendix on configuring the HLA development environment, since these steps require nothing more than downloading the HLA binary, using a text editor, and compiling the code.

The book progresses from Hello, World in HLA describing basic statements and the HLA Standard Library to data representation, memory access, data types, procedures, and arithmetic expressions. Roughly mid-way through, a chapter on low-level control structures followed by 70+ pages on advanced arithmetic operations are what I found to be the most mentally engaging chapters in the book. These are followed by chapters on the HLA Compile-Time Language (CTL), bit manipulation, and 80x86 string instructions. The book concludes with a useful chapter on leveraging classes and objects in HLA.

The Art of Assembly Language ramps up at an even keel and never once did I feel rushed or required to re-read a passage due to complexity or ambiguous discussions. The author does a great job of keeping the focus on educating the reader with easy-to-follow instructions and explanations. About the only thing I found lacking was a concluding chapter on where to go next based on the modern day computing landscape. How can I leverage these newfound assembly language skills in a multicore scenario? What toolsets are available for ARM development, and how are they different from the 80x86 instructions learned from the book? Will assembly language see a resurgence as the drive toward miniaturization accelerates? Essentially, where do I go from here?

Overall, *The Art of Assembly Language* is a welcome addition to the bookshelf of any programmer who has yet to experience and appreciate a deeper understanding of HLA gained as a result of reading the book. For developers already familiar with assembly language, the book offers a solid reintroduction to the art and a strong reminder why learning assembly language skills are still relevant in today's third-generation language-favored world.

[Return to Table of Contents](#)

Q&A: Catching Up with Lisp

Support for multicore and Big Data are among the upcoming features of Franz's Lisp-based tools

by Jonathan Erickson

Jans Aasman is CEO of Franz Inc. (<http://www.franz.com/>). He recently took time to talk to *Dr. Dobb's* about Lisp and other subjects.

Dr. Dobb's: Jans, when I hear "Franz" I can't help but add "Lisp". Is Lisp still at the core of what Franz is up to these days?

Aasman: I know you journalists want clarity, so my clear answer is a resounding "yes" and "no".

Yes #1, we are still investing in our Lisp (Allegro CL). We recently released a new version, 8.2, and are planning v9.0 for later this year. We run on more platforms than ever, and we have our updated IDE running on the Mac. Also, real soon we'll announce our SMP (Symmetric Multi Processing) version of Allegro CL to automatically take advantage of all the multicore processors in a system.

Yes #2. AllegroGraph is the most ambitious internal project we have ever worked on at Franz. It is a robust commercial graph database that is, in our humble opinion, the most scalable graph database for RDF data available, and it is all written in Lisp. So we are eating our own dog food. Actually, by doing this we have made many low-level advancements in Allegro CL to improve data manipulation capabilities for "Big Data". It is important to note that we own the entire technology stack, which allows us to make optimizations all the way down to the hardware level.

No. What we see is that AllegroGraph is now more than half of our business, so in that sense you could say AllegroGraph is a new core.

Dr. Dobb's: Did Lisp play a part of the recently announced Franz-IO Informatics partnership to deliver "fit for purpose" applications created by a Semantic application framework?

Aasman: No, not really. We used some Lisp in the project, just to do data manipulation, but for the most part the work was done by AllegroGraph and the IO Informatics tools.

Dr. Dobb's: Why is Lisp so ideally suited to "semantics" such as with the Semantic web?

Aasman: There are several ways to answer this question, let me try a few.

- The main argument would be that Lisp is still the premier language for Artificial Intelligence and the Semantic Web is a sane and more modest revival of AI.
- The first applications written in Lisp were semantic networks. People would write knowledge processing applications and expert systems that were always built on top of a semantic network. Some of these semantic networks were more like a frame-based system, where a frame is a thing that has a type (say 'person') and a bunch of arbitrary attributes (say 'first-name' 'last-name' 'dob', etc). These frame systems had, from day one, the capability to add any arbitrary attribute. The whole Semantic Web is just like a frame system. You have things that have a type and then a bunch of (arbitrary) attributes and then some logic around it to make it smart. Lisp was good at frames then, and now with modern hardware it is even better at it now.
- One other reason that our Lisp is ideally suited for semantics is that we have a built in Prolog in our Lisp. Prolog is a declarative



language build around logic and it is on many levels more powerful and practical than all the other logics invented by the Semantic Web community. We can use Prolog as a perfect rule-based system and query language for our triple store. So in our Lisp the rule and query language was built in from day one.

As an aside, many of the founders of the Semantic Web are Lisp people.

- The first triple stores were written in Lisp (Ora Lassila from Nokia Labs).
- Jim Hendler and Dean Allemang wrote the book *The Working Ontologist*. Jim Hendler was also the driving force behind DAML that later became OWL.
- Deborah McGuinness, who was on all the committees that lead ultimately to ontologies.
- The first scalable reasoners for the Semantic Web were all written in Lisp (think of Fact by professor Ian Horrocks, think of Racer by Prof Haarslev and Prof Moller) etc.
- Also see the paper "The Use of Lisp in Semantic Web Applications," at <http://www.mindswap.org/~katz/2003/08/use-of-lisp-on-sw.pd>.

Dr. Dobb's: What's new with Lisp?

Aasman: Do you mean with Lisp in general or the Lisp that Franz provides? Let me take the second interpretation: From our (biased) Franz perspective, here are the exciting things:

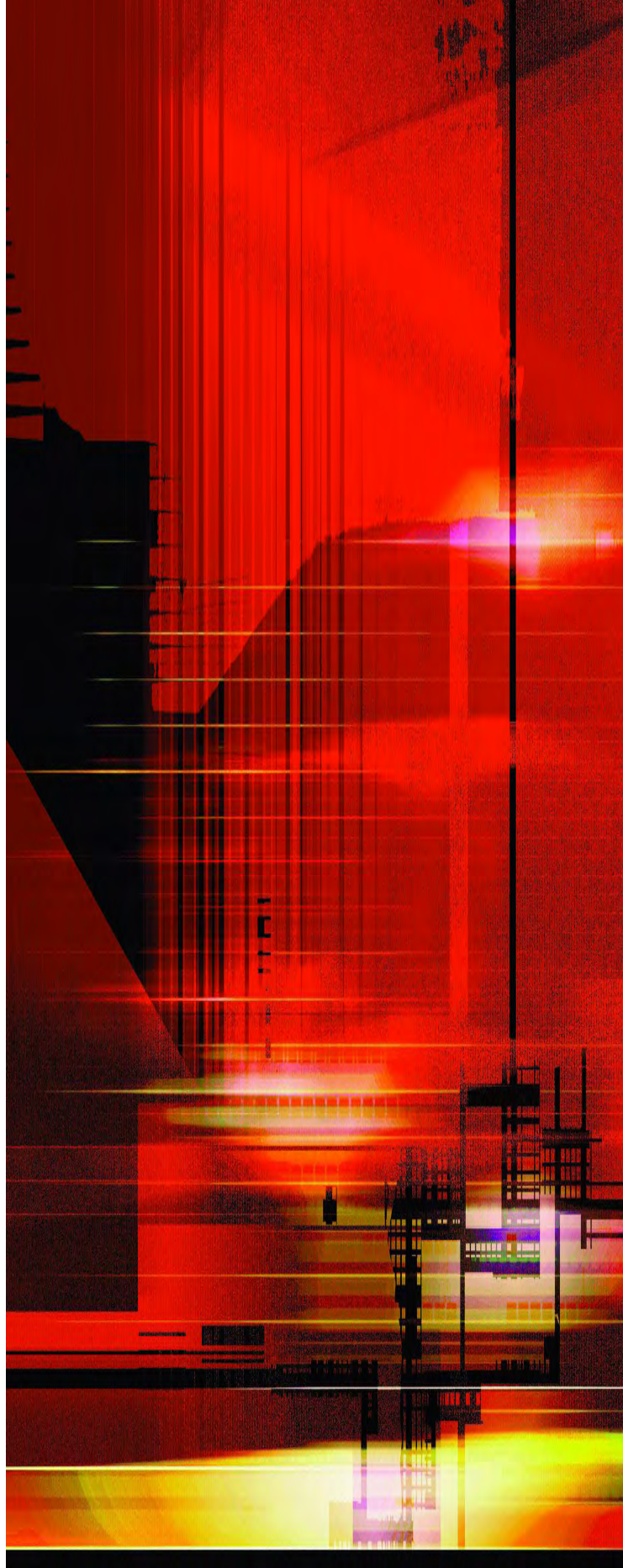
- Symmetric Multi Processing
- The IDE and graphics system now run on all major platform, including the Mac
- You can debug Lisp programs visually in a stepper
- When you download our free version of Allegro CL, you get (also for free) a professional object-oriented database (AllegroCache) and Semantic Web database (AllegroGraph).

Dr. Dobb's: What is a "dynamic object caching database"?

Aasman: Dynamic object caching means that the language makes sure that the objects are in memory when you need them (and more importantly), get rid of them when you don't need them anymore. We also sell an object-oriented database (AllegroCache) that is totally integrated in the language itself. Not some wrapper around a relational database but a completely native object-oriented database that is completely embedded in the language. So you can program as if objects are in memory but they are actually on disk.

Dr. Dobb's: Thanks for your time.

Return to Table of Contents



An Online Budget Questionnaire, JavaCC, and the Three Ways of Putting Together

By Jocelyn Pain

After the ash, the papers will again be full of post-Election austerity. Roads or railways, councils or care homes: where will the Chancellor cut to chop the UK's massive deficit? A colleague and I have put up a poll that surveys how people would trade-off these cuts, and how this correlates with voting preference, at www.sharesim.arachsys.com/login/budget10.jsp and at the Facebook group Balance the Budget (<http://www.facebook.com/group.php?gid=115236488495956>). We used a questionnaire-compiler, which is a nice application of the parser-builder JavaCC (<https://javacc.dev.java.net/>).

The questionnaire was written by my colleague John Green of Research for Today. It is based round four grids, rendered as HTML tables. Each grid has a row for each area of public spending surveyed: affordable housing; Trident nuclear submarines; other defence; nationwide 40Mb broadband; care homes. Each row has cells for possible levels of service: 20,000 more affordable homes per year; 40,000; 60,000. Cells are also equipped with a radio button and a score stating the relative cost of that level of service:

18) Trident nuclear submarine program	Do not upgrade ☐	As Now, upgrade Trident ☒	
19) Other defence capital spending	1 fewer aircraft carrier / associated planes, no eurofighter program. Instead upgrade existing stock ☐	As Now ☐	15% more investment ☐
20) Law and Order	10% less police, new prisons etc ☐	As Now ☐	10% more police, prisons ☐
	Only paid to those	Paid to all	

In each grid, the person doing the poll has to click buttons so that the costs sum to a particular budget saving: £60 billion for the first grid; £40 billion for the second; £20 billion; and As Now. It turns out to be surprisingly difficult — one might say taxing — as you discover just how much the UK cannot afford.

The questionnaire also contains multiple-choice questions having just one row or column of radio buttons or checkboxes. These ask for such characteristics as sex, age band, and political preference: Labour, Tory, or Lib Dem.

These and the rest of the questionnaire are coded in a mini-language that I implemented. Here's an example:

```
text |We are now going to ask you
    some questions about yourself
    and your politics.|
```

```
question Age
  asks |How old are you?|
  options |Under 18|
          |Between 18 to 65|
          |Over 65|
  layout vertical
```

```
question Party
  asks |Which party do you support?|
  options |Labour|
          |Tory|
          |Lib Dem|
          |Other|
  layout vertical
```

page break

(By the way, I used the bar | as a string delimiter rather than double quote, to make it easier to include quotes in strings.)

It is useful to design mini-languages for tasks like this, but of course one needs to specify their grammar. Here is a grammar for the fragment above, written in BNF. I've extended it

with the + operator, which by convention means “one or more occurrences of”:

```
<questionnaire> ::=
  <item>+

<item> ::=
  <question> |
  <non-question>

<question> ::=
  "question"
  "asks" <string>
  "options" <string>+
  <layout-specifier>

<layout-specifier> ::=
  "layout" ( "horizontal" | "vertical" )

<non-question> ::=
  <text> |
  <page-break>

<text> ::=
  "text" <string>

<page-break> ::=
  "page break"
```

To write a parser in JavaCC, you must transcribe BNF into its grammar-description language. This is easy: JavaCC's grammar-description language is designed to be close to BNF.

An important thing JavaCC needs to know is which are the reserved words, or what compiler-writers call “tokens”, in its input. For my mini-language example, they are *asks*, *horizontal*, *layout*, *options*, *page break*, *question*, *text*, and *vertical*.

You must also say how tokens are separated. In common with almost all programming languages — except Fortran, Snobol, and I suppose, Python — our mini-language is completely free-format. So I tell JavaCC to ignore spaces, tabs, and newlines and carriage returns between tokens, by writing a regular expression that specifies the characters to be skipped.

I let questionnaire authors put comments in their files. This is good practice for any mini-language. It allows the author to annotate their file with information about creation and revision dates, why changes were made, the purpose of a particular statement, and so on. I decided to use C-style comments: either two slashes up to the end of the line, or slash-star up to star-slash. Again, one specifies these with regular expressions.

Having told JavaCC about tokens, the gaps between tokens, and comments, one can now code a grammar. In JavaCC's grammar-description language, each rule looks rather like a Java method. For example, the rule for *item* resembles a method named *item*, with a body that says “either call the *question* rule or call the *non-question* rule”.

It isn't enough merely to recognize whether a file holds a valid questionnaire or not. One wants the parser to build a representation of the questionnaire. To do this, one augments the grammar with “semantic actions”. That's the computer scientist's name for pieces of code that take the strings recognized by the right-hand side of a grammar rule and do stuff with them, such as storing them in the fields of a data structure. In JavaCC, the semantic actions are Java statements.

So a JavaCC file contains definitions of tokens and of the spaces

between tokens, definitions of comments, a grammar, and the semantic actions inside the grammar. You must embed this in a class definition. This can contain methods and variables to be called by the semantic actions. You must also set up the input stream that the parser will read text from, and set some options. Amongst other things, these control the diagnostics to be generated. And then, feed the JavaCC file through JavaCC, and out comes a Java file. Compile this, and you have your parser.

The above would apply, *mutatis mutandis*, to any parser-builder, not just JavaCC. Something else that should always apply is the tight connection between grammar, data structure, and control structure. In sequential programming, there are three main control structures. These are: sequencing, or doing one thing and then another thing; alternation, or doing one thing OR another thing; and iteration, or doing one thing over and over again. Java does sequencing with semicolons, alternation with “if” and “switch”, and iteration with loops.

Corresponding notions in a grammar are: recognizing one thing and then another thing; recognizing one thing OR another thing; and recognizing one thing over and over again. The BNF notation I used indicates the first by concatenation, the second by the | operator, and the third by the + operator.

And corresponding to these in data structures are: storing one thing and another thing; storing one thing OR another thing; and storing one thing over and over again. Almost every language uses multiple fields for the first, and arrays, lists, vectors, and so on for the third. A principled language like Pascal uses “variant types” for the second; some other languages call these “unions”. Java doesn't quite have this notion, but bodes it with subclasses.

What is the connection of these “Three Ways of Putting Together” with our questionnaires? I needed to decide what kind of data structure the semantic actions in my grammar would build. The answer is simple: data structures that parallel the structure of the grammar.

So I represented the entire questionnaire as an instance of class *Questionnaire*, with a field that was a vector of *Items*. I defined class *Item*, and its subclasses *Question* and *NonQuestion*. A *Question* had instance variables holding the text that it asks, a vector of options, and a *LayoutSpecifier*. The *LayoutSpecifier* was a Java Enum with two elements: HORIZONTAL and VERTICAL. Class *NonQuestion* had subclasses *Text* and *PageBreak*. And a *Text* contained one instance variable, a *String*. Compare this with my BNF.

By the way, the previous paragraph, and my grammar, are for the questionnaire fragment I showed earlier. The questionnaires that we use in real life contain other kinds of question, such as the grids. They also contain other constructs. For example, conditional jumps that skip part of a questionnaire if the respondent gives certain answers.

How does the structure of questionnaires relate to the Web? When somebody logs in to our budget poll, my Java code looks up the name of the file that contains the mini-language definition for the budget10 questionnaire. It passes the name to the parser

instance I generated with JavaCC. JavaCC has translated my grammar rules into methods, the top-level being named *questionnaire*. My code calls this, passing it an input stream opened onto this file. This method calls the subordinate methods corresponding to subordinate rules, as well as the semantic actions.

The semantic actions build an instance of class *Questionnaire*. This then has to be rendered as HTML. And here we again see the Three Ways of Putting Together. The top-level render-as-HTML method sequences through the fields of the *Questionnaire* instance. It iterates over the vector of *Item* instances in one of these fields. And it has alternative methods for outputting an *Item*, depending on whether this is a *Question* or a *NonQuestion*.

So the Three Ways of Putting Together apply to grammar, to data, and to control. By designing these so that one mirrors another, we can write really clean code.

The same applies to processing the answers to a questionnaire. Once we have answers from enough respondents, John will analyze them with his SIMALTO ("Simultaneous, Multi-Attribute, Level Trade-Off") algorithms, and publish the results, correlating with respondents' expressed voting preference. But how do the answers get to that stage? The respondent's browser sends each page's answers to our server in the normal way, as a string of *name=value*

parameters at the end of a URL. The server passes these to my Java code, which stores them in an instance of my class *AnswersToQuestionnaire*. This parallels the question parts of class *Questionnaire*. (The non-question parts, namely text and page breaks, don't generate answers, and therefore are irrelevant.) Class *AnswersToQuestionnaire* contains a vector of instances of class *AnswerToQuestion*, analogous to my *Question* class.

Having these parallel structures makes it easy to associate answers with the question they belong to. This helps when saving the answers to file, and when displaying already-answered questions. It is also useful in reporting errors, as when the respondent's budget expenditure falls outside the permitted range. Luckily, on the Web, the consequence is one of our red error messages. If you are Chancellor, the consequences of inapposite budget expenditure are less benign.

CLICK HERE TO COMMENT ON THIS POST
(http://www.drdobbs.com/blog/archives/2010/04/an_online_budge.html)

[Return to Table of Contents](#)

Other Voices: Why Scripting Is Evil, Part 1

Scripting is easy to understand and a natural step, but scripting also has serious problems

By Erik Troan

There are two kinds of people in the world — those who divide the world into two kinds of people and those who don't. Okay, an old joke, but I'm clearly the first kind of person. I try and split everything into two buckets. System automation solutions lend themselves to this two-sizes-fits-all mantra, with the approaches splitting between scripting solutions and model-based approaches.

It seems like most people think about scripting as the best approach to automation. Whether it's simple shell (or PowerShell) scripts, attaching scripts to Opsware machine definitions, or running an inscrutable Perl command line, scripting is king. Engineers and system administrators tend to break problems down into steps, and scripting is a way of codifying those steps and running them on lots of machines.

Scripting is easy to understand and a natural step, but scripting also has serious problems. It's a great tool when there is nothing else available, but scripts are difficult to write, impossible to test, unverifiable, and non-invertible.

Why are scripts difficult to write? There are a few reasons; the most obvious is that you're describing how to get to a new state, rather than just describing how things should wind up. Think about how building architects work. They draw up blueprints that completely describe how the important parts of the new building need to end up. What columns support which beams, where the electrical needs to go for code, and what plumbing needs to be put in for fire safety. The construction crews then decide how to get the steel, wires, and pipes into place. Can you imagine if the architect had to write a detailed list of instructions describing how to build a building? Down to what kind of screws to use to hold up the drywall and what kind of drill should be used to put them in place?

Writing scripts for system automation means you have to describe every step. Every time. This forces scripting languages to be Turing-complete; they're powerful languages that can solve any problem. That also means the scripts are impossible to analyze for correctness.

So once you've written a script, how do you test it? Install a machine and run it? Go log into a box and run it there, watching it closely? How do you know the box you're testing it on is a good enough representative of the 1,000 machines you're about to run the script on? Like it or not, machines drift. Configurations change, and software installs change. That handcrafted script has to be able to adapt to every one of these divergent machines. You can test a few cases, but are you really testing it exhaustively? Have you tested the error cases, or will things silently fail? Or even worse, will they fail in a manner that leaves the system unresponsive? Software companies pay a lot of people to test their code under every conceivable situation, and we still wind up with Vista. Does an IT staff test their scripts that carefully?

Let's say you got the script written and tested. How do you know that it's doing the right thing when you run it? If it makes a configuration change, can you verify the change was made correctly? Chances are you haven't had to describe the change anywhere other than a whiteboard. So what checks those 1,000 machines to make sure the script did the right thing? How do you audit the system to make sure the script didn't break the change that the previous script was supposed to have made?

Finally, if you've navigated all of those mines, what if the change was simply incorrect? The script did what it was supposed to do, but it turned out to be a bad idea. How do you undo it? Scripts are, by nature, non-invertible. You can't say "oops, let's just undo that." Instead, you're writing a new "undo" script, and testing it, and (hopefully) checking that it did the right thing. Making non-invertible changes to production systems is crazy.

Face it. Scripts suck.

—Erik Troan is the founder and chief technology officer for rPath. Contact Erik at ewt@rpath.com.

[For a follow-up, see "Why Models Beat Scripting" at <http://www.drdoobbs.com/tools/224700402>.]

[Return to Table of Contents](#)

Associate Mutexes with Data to Prevent Races

Come together:

Associate mutexes with the data they protect, and you can make your code race-free by construction

By Herb Sutter

Race conditions are one of the worst plagues of concurrent code: They can cause disastrous effects all the way up to undefined behavior and random code execution, yet they're hard to discover reliably during testing, hard to reproduce when they do occur, and the icing on the cake is that we have immature and inadequate race detection and prevention tool support available today.

The holy grail of the Consistency pillar is to make a concurrent program race-free and deadlock-free by construction. [1] “By construction” means to write our code in such a way that we can verify with confidence during development and testing that it does not contain races or deadlocks — not just to rely on doing enough stress testing to get a probabilistic level of confidence, but to really *know* that there are no or deadlocks at all, at least for the shared data and locks that we control. I've written about some deadlock prevention techniques in “Use Lock Hierarchies to Avoid Deadlock.” [2] This time, let's consider race prevention.

This article shows how to achieve the “race-free by construction” grail via sound engineering, with automatic and deterministic race identification at test time based on code coverage alone. Notes: For convenience, from now on I'm going to talk about just locks, but the issues and techniques apply to any kind of mutual exclusion. And although the example code I'll show will happen to be in C++, the techniques we'll discuss apply in any mainstream language, including C/Pthreads, C#, and Java.

The Problem

Let's say you want to use some shared data protected using mutexes:

```
// use table1 and table2 (both are shared)
table1.erase( x );
table2.insert( x );
```

Quick: What locks should you take to make this code correct?

The answer is usually to look in some documentation or comment, because the connection between a mutex and the data it protects usually exists primarily in the mind of the programmer, and we rely on discipline to remember which lock to take.

Let's say that *table1* and *table2* are protected by two mutexes *mutTable1* and *mutTable2*, respectively. Then we might write something like this:

```
lock mutTable1 and mutTable2

table1.erase( x );
table2.insert( x );

release mutTable1 and mutTable2 appropriately
(e.g., in finally blocks, or using automatic
dispose/destructor methods)
```

The trouble is that the programmer has to remember that *mutTable1* and *mutTable2* are the “right” mutexes to take to use *table1* and *table2*. This has two main problems.

The minor problem is productivity: We're creating work for the developer by forcing him to look up which mutex to use, because in general he won't get automatic IDE completion/suggestion support to answer questions like, “Which mutex do I need to take to use *table2*?”

The major problem is correctness: We're relying on programmer discipline to make the code correct, and discipline can fail us because we will sometimes make mistakes. What can go wrong? We can fail to take the correct lock, in one of two ways:

- “Oops, forgot to lock.” We can forget to take a lock at all, such as that in this example we might forget to take the lock on *mutTable2*.
- “Oops, took the wrong lock.” We can inadvertently take a lock on the wrong mutex, either because of confusion about which mutex is the right one or because of a simple accident like a typo that misspells *mutTable2* as *mutTable3*.

The trouble is, either way we'll end up writing a race condition with a horrible failure mode: The code will silently compile and appear to work during testing, only to fail nondeterministically in production — typically in the form of intermittent, timing-dependent, hard-to-reproduce bug reports from customer sites.

The Goal

Instead of relying on comments and discipline, what we'd like to do is state in code which mutex protects which data, so that we can build our own guardrails to ensure we did it right. It turns out that this is a powerful approach, because it lets us turn a class of non-deterministic run-time failures into ones we can detect:

- automatically,
- deterministically,
- at test time, and
- based on code coverage alone.

Let's see how.

Solution: Associate Data with Locks

Observation: If we knew which mutex covered which data, we could verify at test time that when we access shared data we're holding the right locks. So let's explicitly group the mutex and the data it protects together in code, in the same struct or class behind an interface that we can check at test time:

```
// Example 1: Illustrating the mutex association pattern
//
struct MyData {
public:
    // provide access to data
    vector<int>& v() { assert( m_.is_held() ); return v_; }
    Widget* w() { assert( m_.is_held() ); return w_; }
    // provide access to mutex
    void lock() { m_.lock(); }
    bool try_lock() { return m_.try_lock(); }
    void unlock() { m_.unlock(); }
private:
    // hide
    vector<int> v_;
    Widget* w_;
    mutex_type m_;
};
```

We require that callers always access the shared object through the accessor methods. That's enough to let us provide automated checking on every use of the shared object, and find many latent race conditions deterministically at test time based on code coverage alone.

Let's consider a few examples of how this helps verify that our program is race-free. First, what if we accidentally try to access the data without taking a lock at all? Consider:

```
MyData data1 = ..., data2 = ...;
// Later, but without having taken any locks:
data1.v().push_back( 10 ); // A: error: good, will assert
data2.w()->ProcessYearEnd(); // B: error: good, will assert
```

In line A, the call to `data1.v()` first performs `assert( data1.m_.is_held() )`, and because the lock isn't held the program will stop with a test-time error and point directly to the offending line. Note that we will catch this error automatically and deterministically based on code coverage alone; as long as our testing exercises line A, we will discover the error the first time we try to execute the

line. Similarly, line B's the call to `data2.w()` first performs `assert( data2.m_.is_held() )`, which will fail deterministically and point directly to the offending line.

What if our testing doesn't do full code coverage, so that we can ship knowing we have no races? For example, what if during testing, we fail to exercise lines A and B at all, or they are callable along multiple paths — each of which must take the lock — and we fail to exercise each path at least once? Even with incomplete testing, the worst case is that the error will be consistently diagnosed in production the very first time the offending code path is actually exercised, with a clear diagnostic that points directly to the offending line — whether or not the potential race condition actually manifests by having another thread performing a conflicting access at the same time. Unlike today, even in the worst case (including no testing at all), the faults are never intermittent, timing-dependent, or hard to reproduce.

Next, what if we try to access the data while having taken a lock, but it's the wrong lock? Consider:

```
{ // acquire lock on data1
  lock_guard<MyData> hold( data1 );
  data1.v().push_back( 10 ); // C: ok
  data2.w()->ProcessYearEnd(); // D: error: good, will assert
}
```

In line C, the call to `data1.v()` again first performs `assert( data1.m_.is_held() )`, and this time because the lock is held the program continues normally, having validated that the access to `data1.v_` is not a race.

In line D, however, the call to `data2.w()` first performs `assert( data2.m_.is_held() )`, which fails because although we are holding a lock, we're not holding the one on `data2`. Again, because the lock isn't held the program will stop with a test-time error and point directly to the offending line.

There is only one specific abusive caller pattern that can't be checked automatically using this technique. Callers that remember a direct pointer or reference to the object for later use must be prevented by programmer education and team policy:

```
// The only hole: Require and enforce that programmers don't do this.
//
vector<int>*& sneaky = nullptr;
{
    // enter critical section
    lock_guard<MyData> hold( data1 );
    sneaky = &data1.v(); // E: compiles, but avoid doing this
    sneaky->push_back( 10 ); // F: ok, but not checked
}
sneaky->push_back( 10 ); // G: error: race, but won't assert
```

In line E, we correctly verify that `data1.m_.is_held()` and allow access to the member, but the caller remembers a sneaky pointer or reference to the object. That's unsafe. It's not inherently always wrong: For example, the access in line F is still correct and not a race, but the problem is that we can no longer verify that it's correct. It's unsafe because it opens up the one hole demonstrated in line G, where the call to `sneaky->push_back( 10 )` is a race, but still compiles and runs and won't be caught as it bypasses the validation.

The mutex association pattern catches both the “Oops, forgot to lock” and “Oops, took the wrong lock” mistakes. As a bonus, it also provides a reasonable migration for existing source code,



I KNOW A PLACE THAT'S DIFFERENT, BUT FAMILIAR

Code in the cloud ... with the development technologies you already know

With **Windows Azure™**, you have a flexible, on-demand infrastructure that taps into all the IT resources you need. The advantages?

You have the freedom to focus.

With the cloud as your development platform, you can focus on what you do best: create applications, without the restrictions on performance, capacity, or deployment that come with using your own hardware rather than the cloud.

You can code in the language you choose.

Windows Azure enables you to build applications in multiple languages, including .NET, PHP, and Java.

You're already equipped for the cloud.

You can start developing applications for Windows Azure right away, using familiar tools such as Microsoft® Visual Studio® 2008.

Find out more.

See how Windows Azure can help get you coding in the cloud.
Go to **windowsazure.com** now.

where the impact on existing calling code can sometimes be made as small as just adding `()` or a similar minor syntactic change.

Finally, and this can't really be overemphasized — this is a vast improvement over intermittent timing-dependent hard-to-reproduce bug reports from customers.

**The holy grail would be to make our programs
race-free by construction**

Automation

The pattern we've presented is sound, but as we apply it to the shared data in our system we'll find that some parts of the pattern are repetitive (e.g., the `lock/try_lock/unlock` code). It would be nice to automate the boring parts. Here's one way.

First, an initial optional step: We might need to make our mutex testable if it isn't already. Many mutex services (including C++0x `std::mutex`) don't provide a way to ask, "Do I already hold a lock on this mutex?" But we need a method like `is_held` to be available. Fortunately, we can easily get it by wrapping together an arbitrary mutex type with a thread ID that stores the ID of the thread that currently holds the mutex. The thread ID initially holds an invalid value that means no threads own the mutex. When we acquire a lock, we set the ID; and when we release a lock, we reset it back to its default nobody-loves-me state:

```
template<typename Mutex>
class TestableMutex {
public:
    void lock()      { m.lock(); id = this_thread::get_id(); }
    void unlock()    { id = 0; m.unlock(); }
    bool try_lock()  { bool b = m.try_lock();
                     if( b ) id = this_thread::get_id();
                     return b; }
    bool is_held()   { return id == this_thread::get_id(); }
private:
    Mutex m;
    atomic<thread::id> id;
    // for recursive mutexes, add a count
};
```

Now we can automate the boring parts of the mutex association pattern. For example, given the following convenience macros:

```
#define PROTECTED_WITH(MutType) \
    public: void lock() { mut_.lock(); } \
    public: bool try_lock() { return mut_.try_lock(); } \
    public: void unlock() { mut_.unlock(); } \
private: TestableMutex<MutType> mut_;
#define PROTECTED_MEMBER(Type, name) \
    public: Type& name() { assert(mut_.is_held()); return \
    name##_; } \
private: Type name##_;
```

Now we associate data with a mutex more easily:

```
// Example 2: Example 1 rewritten using the convenience macros.
//
struct MyData {
    PROTECTED_WITH( some_mutex_type );
    PROTECTED_MEMBER( vector<int>, v );
    PROTECTED_MEMBER( Widget*, w );
};
```

The above is identical to the longer version we wrote out by hand in Example 1. If you don't like macros, or don't have them in your language, you can use other methods to get a similar effect.

Granular Control

But what if we already have struct or data that naturally goes together, but different groups of members are covered by different mutexes? To express that, first just define each group on its own, either by hand as in Example 1 or by using a convenience library as in Example 2. For convenience, I'll use the latter:

```
struct Group1 {
    PROTECTED_WITH( some_mutex_type );
    PROTECTED_MEMBER( vector<int>, v );
    PROTECTED_MEMBER( Widget*, w );
};
struct Group2 {
    PROTECTED_WITH( some_mutex_type );
    PROTECTED_MEMBER( map<string, list<int>>, map_from_2 );
};
struct Group3 {
    PROTECTED_WITH( some_other_mutex_type );
    PROTECTED_MEMBER( complex<float>, cf );
};
```

Then combine them:

```
struct MyData { Group1 group1; Group2 group2; Group3 group3; };
// Sample use
{
    lock_guard<Group1> lock( data.group1 );
    data.group1.v().push_back( 100 );
}
```

Summary

In code, associate mutexes with the data they control. Leverage this to find latent race conditions automatically and deterministically at test time based on code coverage alone, and ensure before shipping that you didn't write a race by forgetting to take a lock, or taking the wrong lock, in any of the code that your team controls. Then your code will be in the best possible place: Not racy, and not even just accidentally race-free, but race-free by construction.

References

- [1] H. Sutter. "The Pillars of Concurrency" (*Dr. Dobb's Journal*, August 2007; <http://www.drdobbs.com/high-performance-computing/200001985>).
- [2] H. Sutter. "Use Lock Hierarchies to Avoid Deadlock" (*Dr. Dobb's Journal*, January 2008; <http://www.drdobbs.com/high-performance-computing/204801163>).

—Herb Sutter is a bestselling author and consultant on software development topics, and a software architect at Microsoft. He can be contacted at www.gotw.ca.

[Return to Table of Contents](#)