ted Business Media

Dr. Dobb's DIGEST

The Art and Business of Software Development July, 2009

Information Security

By Jonathan Erickson,
Editor In Chief

When it comes to computer security, it's a question of "when," not "if": When will you unknowingly download a Trojan horse, when will your web site be defaced, when will your credit card treat someone else to dinner, and when will your company lose intellectual property and critical data about your core business and clients?

How big is the problem? While exact figures are hard to come by, estimates are staggering. In a recent survey commissioned by security firm McAfee, more than 800 CIOs around the world estimated that they lost a combined $4.6 billion worth of intellectual property last year, while spending approximately $600 million repairing damage from one data breach or another. Based on these numbers, McAfee projects that companies worldwide lost more than $1 trillion in 2008 alone. Of this, 42% of the respondents said laid-off employees are the biggest threat, followed by outside data thieves at 39%. Then there's the 2005 FBI report that pegs internal security attacks at costing U.S. businesses $400 billion per year.

While it's unlikely you can totally prevent intrusions, you can mitigate their impact. One way is to build or "bake" security into software, starting in the design phase instead of retrofitting it at the end. As it turns out, that's the goal of Build Security In (buildsecurityin.us-cert.gov), a collaborative program between the Department of Homeland Security and the Software Engineering Institute to provide software developers and architects with practices, tools, guidelines, rules, and principles for building security into software throughout the lifecycle. Build Security In sees software security fundamentally as an engineering problem that must be addressed in a systematic way throughout the software development lifecycle.

The good news is that this heightened awareness of security is forcing companies to become more attuned to how their software is being designed, developed, and tested.

[illegible]

Return to Table of Contents

Finding Java API Methods and Classes

Reducing time and guesswork when searching for the right classes and methods

Researchers at Carnegie Mellon University's School of Computer Science have developed two new tools to help programmers select from among thousands of options within the APIs used to write applications in Java.

The tools — Jadeite and Apatite — take advantage of human-centered design techniques to significantly reduce the time and guesswork associated with finding the right classes and methods of APIs. Choosing APIs for accomplishing a given task is not intuitive, said Brad A. Myers, professor of human-computer interaction. With more than 35,000 methods listed in 4,100 classes in the current Javadoc library of APIs — and more being added in every new version — not even the savviest developer can hope to be familiar with them all.

"This is a fundamental problem for all programmers, whether they are novices, professionals, or the growing number of end-users who just need to modify a web page," Myers said. "It's possible to design APIs so that they are easier to use, but that still leaves thousands of existing APIs that are hard to use but essential for Java programming. Jadeite and Apatite help programmers find what they need among those existing APIs."

Jadeite (Java Documentation with Extra Information Tacked-on for Emphasis) improves usability by enhancing the existing Javadoc documentation. For instance, Jadeite displays the names of API classes in font sizes that correspond with how heavily used they are based on Google searches, helping programmers navigate past little-used classes. The commonly used PrintWriter is in large, prominent letters, while the lesser used PrintEvent is in smaller type.

Jadeite also uses crowd-sourcing to compensate for the fact that an API sometimes doesn't include methods that programmers expect. For instance, the *Message* and *MimeMessage* classes don't include a method for sending an e-mail message. So Jadeite lets users put so-called placeholders for these expected classes and methods within the alphabetical listing of APIs. Users can edit the placeholder to guide programmers to the actual location of the desired method, explain why a desired method is not part of the API, or note that a desired functionality is impossible.

Finding the way to create certain types of objects, such as SSL sockets that enable secure Internet communications, may not be obvious to programmers the first time they encounter these objects. In these cases, Jadeite includes examples of the most popular code used by programmers to create these objects, allowing the user to learn from the examples.

User studies showed that programmers could perform common tasks about three times faster with Jadeite than with the standard Javadoc documentation. Apatite (Associative Perusal of APIs That Identifies Targets Easily) takes a different approach, allowing programmers to browse APIs by association, seeing which packages, classes, and methods tend to go with each other. It also uses statistics about the popularity of each item to provide weighted views of the most relevant items, listing them in larger fonts. Both Jadeite and Apatite remain research tools, Myers said, but are available for public use. Broader use of the tools will enhance the crowd-sourcing aspects of the tools, while giving the researchers important feedback about how the tools can be improved.

Research by Jeffrey Stylos, who was awarded a Ph.D. in computer science this spring, underlies both Jadeite and Apatite. Besides Myers, research programmer Andrew Faulring and undergraduate computer science student Zizhuang Yang contributed to the development of Jadeite and computer science undergraduate Daniel S. Eisenberg led the implementation of Apatite. Eisenberg's work on Apatite earned first place in the Yahoo! Undergraduate Research Awards competition at Carnegie Mellon this spring.

Jadeite and Apatite are part of the Natural Programming Project, www.cs.cmu.edu/~NatProg/, an initiative within Carnegie Mellon's Human-Computer Interaction Institute that is investigating how to make programming easier. Both tools have been funded by grants from the National Science Foundation and SAP AG.

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
SERVICES MARKETING COORDINATOR
Laura Robison

AUDIENCE DEVELOPMENT
CIRCULATION DIRECTOR
Karen McAleer
MANAGER
John Slesinski

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

UBM LLC

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

TechWeb

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

techweb™

How the U.S. Changed Its Security Game

Agencies pool threat data and make practical fixes to common woes

by Alan Paller

On March 12, 2007, the CEO of one of the nation's largest defense contractors learned of a call from the Office of the Secretary of Defense informing his firm that the FBI had evidence that his company had allowed another nation to steal details of some sensitive technology that DOD had contracted to develop. There was no getting the data back. In a meeting at the Pentagon the next week, the executive learned he was not alone. Around the table were other defense contractor executives who had suffered similar breaches.

The meeting was among the catalysts for what has evolved into a change in thinking about information security in U.S. government and the defense industrial base. It includes more emphasis on actions proven to block known or expected attacks, as exemplified in the "20 Critical Security Controls" crafted earlier this year.

The changed thinking involves IT pros from CIO to developer, and is highly relevant to the private sector as well. A bank recently lost $10 million in less than 30 minutes to hackers who had replicated ATM cards and manipulated internal bank computers to increase limits on the amount each card holder could take in a day. The amount of money lost was limited only by the amount of money in the ATM machines that had been targeted.

Throughout most of this decade, the topic of cybersecurity rarely touched senior management, coming up only in the context of regulatory compliance. The problem has been that many of the steps taken to meet compliance requirements weren't geared to match the emerging threat. Rather than doing the tasks needed to ensure that systems were configured securely and that attacks were blocked or found quickly, organizations were forced to pay consultants to write lengthy compliance reports. The reports met the regulator's demands, and the CIO was told that his organization was in compliance. When the CIO learned that the company's systems had been penetrated and its data looted, surprise was a reasonable response.

The Big Questions

Three questions are usually asked following a cyberattack. The first two are:

1. *What do we need to do to fix this problem?*
2. *How much is enough?*

Any CIO will quickly discover security people don't agree on the answers. When outside experts are asked, their opinions also differ, leaving CIOs frustrated and asking the third question:

3. *Whom can I trust to answer the first two questions?*

The CIOs of the major defense contractors and sensitive government sites faced exactly this uncertainty. They found a solution to this problem that may be of value to those who want to avoid those unwanted FBI calls. While the U.S. defense industry was discovering the extent of penetration into its systems, and thousands of other businesses were hearing from the FBI that they were victims, too, the U.S. intelligence community, the departments of Defense, Homeland Security, and Energy, were leading a national effort to transform cybersecurity.

A theme that shaped the national makeover was that defense must be informed by offense. In other words, organizations should prioritize their security investments on actions that can be proven to block known or expected attacks, or that directly help identify and mitigate damage from attacks that get past the defense. This was a huge shift in thinking and in behavior. Its greatest impact was to change who was considered an expert — it answered question No. 3. In the past, consulting firms armed with checklists of questionable value were let loose to point out missing documentation or incomplete awareness programs.

Under the "offense informs defense" approach, the measures of effectiveness are defined by the people who know how attacks are carried out, and are more specific and more directly related to defenses against known attacks — such as the speed with which unauthorized systems are identified and removed from the network. Other examples of the new practices include:

- Automated inventory so every connected system is known and monitored.
- Application software testing so that security flaws are removed from web applications before they're posted.
- Secure configurations of systems and software deployed on the network.

In all cases, the practices include specific tests that can measure the effectiveness of the controls. Most of the new metrics are automated so that CIOs get continuous visibility into their organization-wide security effectiveness, rather than snapshots or compliance summaries. In short, common threats mean common defenses must be implemented first, and extensively automated to continually update.

Another critical change in thinking in government is recognition that, because of the widespread use of common computer and network technology (Windows, UNIX, HTML, Secure Sockets Layer, SQL, and so on), all organizations face many of the same threats. Individual organizations may face additional threats, but unless they engineer their systems to withstand the common threats, even targeted attackers need not worry about specialized tactics — attackers can just use the common attacks that work on any organization not fully prepared.

It's those common threats that make this security challenge a top priority for software development staff. The majority of current attacks exploit programming errors made by developers whose training never included finding and fixing security flaws. One of the most critical controls not in place in most organizations is a secure application training testing program. (Disclosure: SANS Institute operates the Internet Storm Center, the Internet's early-warning system; is a degree-granting institution; and provides training for security professionals and programmers.)

In federal agencies and leading defense industry organizations, these common threats are being countered through a three-part initiative:

- Establish a prioritized set of security controls that the community affirms will stop or mitigate known attacks.
- Use common tools to automate the controls, and even the measurement of the controls, that continuously monitor security.
- Create a dashboard for CIOs and senior managers to be able to monitor the status of security in their organizations.

Agreement On The 20 Most Critical Controls

Although many subdivisions of the U.S. Department of Defense, the civilian government, and various defense contractors have detailed knowledge of attacks that they have experienced, creating an effective national defense means pooling all that knowledge into a prioritized and up-to-date list of critical security controls that represents the most current attack map available.

In February, the Center for Strategic and International Studies (csis.org) announced it had collated that attack knowledge across all relevant agencies and published a first draft of the "20 Critical Security Controls" (the list is at www.sans.org/cag). The controls were the consensus of organizations that understand offense — including the National Security Agency, DOD Joint Task Force Computer-Global Network Operations, the DOD Cyber Crime Center, US-CERT at the Department of Homeland Security, and the nuclear energy research laboratories at the U.S. Department of Energy, plus top commercial forensics and penetration testing organizations. After public review involving more than 60 organizations, the 20 critical controls were published for government and private use. The U.S. State Department has already implemented software and hardware that automate monitoring of the 20 critical controls and is demonstrating how they can be monitored at every U.S. embassy around the world through a centralized dashboard.

The Developer Role: Control No.7

Application software security is the control most often weakly implemented. Effective implementation calls for three processes:

- Testing all applications using source-code analysis tools (Ounce Labs, Fortify, Coverity, and Veracode are among the most widely used); web application scanning tools (such as IBM Rational AppScan, Hewlett-Packard WebInspect, and Cenzic Hailstorm); and, for important applications, application penetration testing. But the control isn't in place when tests are run; it's in place only when the processes can ensure that problems are fixed or vulnerabilities are mitigated with other defenses, such as a web application firewall.
- Training and testing programmers in secure coding skills in their own programming languages. This is focused on finding and fixing the critical errors identified in the "25 Most Dangerous Programming Errors" (www.sans.org/top25errors), developed jointly by NSA, DHS, Mitre, and SANS. The control is in place only if the programmers pass periodic competency exams in each language they use.
- Procurement language requiring software suppliers to implement the first two processes. Putting these requirements into all contracts that result in software being delivered or used on behalf of the organization extends the control to where it can do the most good.

The Way Forward

The outline of a new era is taking shape in security. In the past, security was usually "bolted on" after systems were designed and deployed. That doesn't work. Security is effective only when it's "baked in."

Security is baked in when very large buyers or groups of smaller buyers act jointly to establish minimum security standards for the software and systems and networks they buy, and then demand that vendors deliver technology that meets those standards.

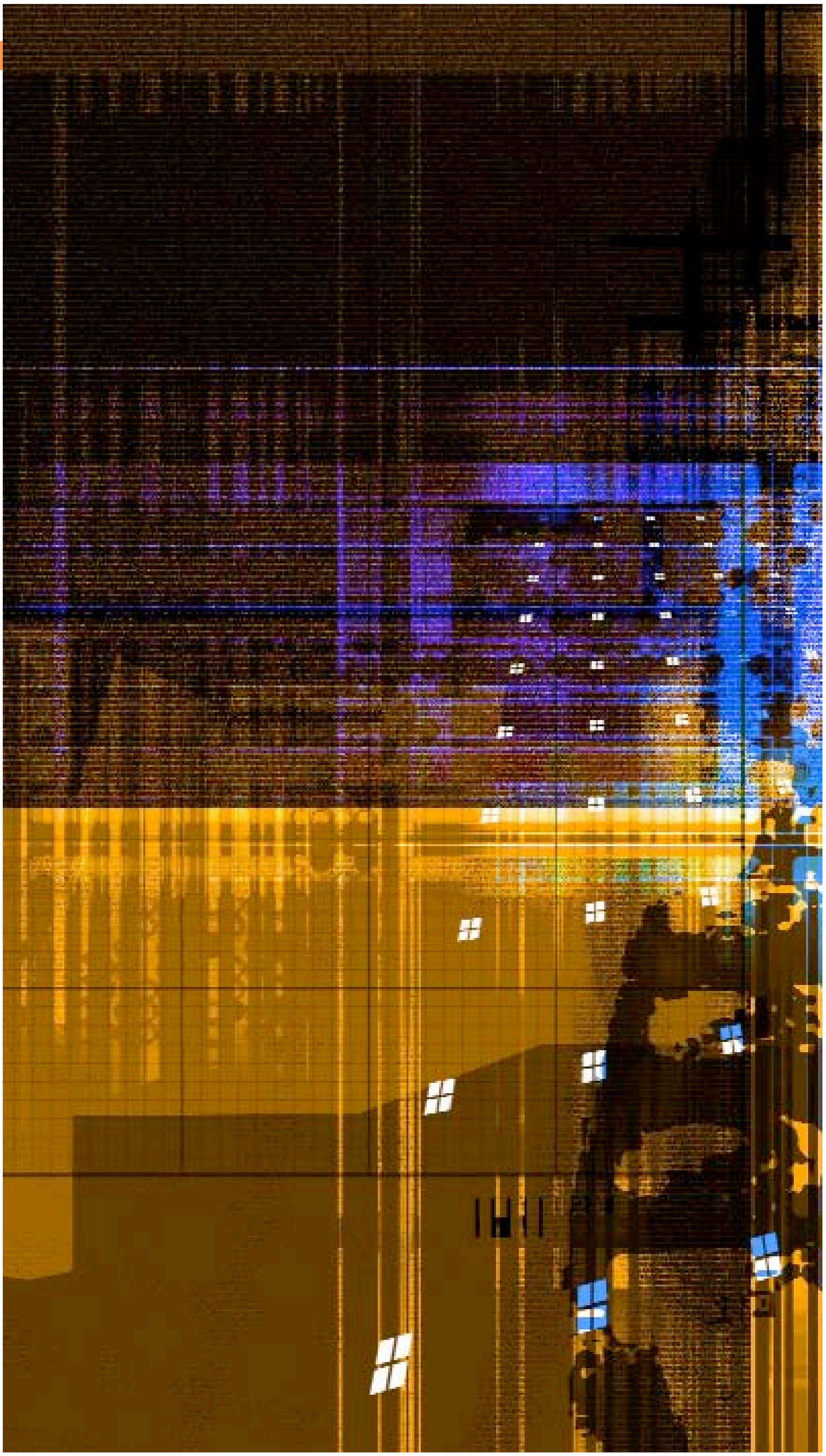
The U.S. Air Force offers the most successful example. With the help of the NSA, the organization that best understands how attacks are launched and why they work, the Air Force identified how Windows should be configured to make it tougher to attack, then persuaded Microsoft to sell 500,000 copies of Windows XP and Vista preconfigured with all key security settings installed. Air Force users could turn on their PCs knowing they were safely configured. The Air Force saved more than $200 million in acquisition and operations cost, radically improved defense against common attacks, and made users happier because systems failed less often. Today, commercial organizations and governments benefit from the more secure version of Windows.

By replicating and expanding the Air Force process, the federal government can use its buying power to provide incentives to bake security into all products and services it buys with the ultimate goal of making security less expensive and easier and more effective for all buyers of the same technologies.

The 20 Most Critical Security Controls automate the measurement of these baked-in controls and can themselves be purchased baked into network and systems monitoring software.

A new era of buying security baked in and continuous monitoring of focused, offense-informed security controls has begun. In government, it's made possible by sharing attack and defense information across the U.S. government and its contractors, and represents the best hope against increasingly sophisticated cyberspace attacks. Any business trying to answer the questions "What do we need to do?" and "How much is enough?" would do well to focus on implementing and automating the 20 critical controls.

— Alan Paller is director of research for the SANS Institute, responsible for projects including the Internet Storm Center and the Top 10 Security Menaces.



Return to Table of Contents

The Case for D

D could be best described as a high-level systems programming language

by Andrei Alexandrescu

Let's see why the D programming language is worth a serious look. Of course, I'm not deluding myself that it's an easy task to convince you. We programmers are a strange bunch in the way we form and keep language preferences. The knee-jerk reaction of a programmer when eyeing a The XYZ Programming Language book on a bookstore shelf is something like, "All right. I'll give myself 30 seconds to find something I don't like about XYZ." Acquiring expertise in a programming language is a long and arduous process, and satisfaction is delayed and uncertain. Trying to find quick reasons to avoid such an endeavor is a survival instinct: The stakes are high and the investment is risky, so having the ability to make a rapid negative decision early in the process can be a huge relief.

That being said, learning and using a programming language can be fun. By and large, coding in a language is fun if the language does a satisfactory job at fulfilling the principles that the coder using it holds in high esteem. Any misalignment causes the programmer to regard the language as, for example, sloppy and insecure or self-righteous and tedious. A language can't possibly fulfill everyone's needs and taste at the same time as many of them are contradictory, so it must carefully commit to a few fundamental coordinates that put it on the landscape of programming languages.

So what's the deal with D? You might have heard of it already — the language with a name like a pun taken a bit too far; annoyingly mentioned now and then on newsgroups dedicated to other languages before the off-topic police reprimands the guilty; praised by an enthusiastic friend all too often; or simply as the result of an idle online search a la "I bet some loser on this big large Internet defined a language called D, let's see... oh, look!"

In this article, I provide a broad overview, which means by necessity I use concepts and features without introducing them rigorously as long as they are reasonably intuitive.

Let's take a brief look at some of D's fundamental features. Be warned that many features or limitations come with qualifications that make their boundaries fuzzy. So if you read something that doesn't quite please you, don't let that bother you too much: The next sentence may contain a redeeming addendum. For example, say you read "D has garbage collection" and you get a familiar frozen chill up the spine that stops in the cerebrum with the imperious command "touch the rabbit foot and stay away." If you are patient, you'll find out that D has constructors and destructors with which you can implement deterministic lifetime of objects.

But Before Getting Into It...

Before getting into the thick of things, there are a few things you should know. First and foremost, if you kind of considered looking into D for whatever reason, this time is not "as good as any," it's in fact much better than others if you're looking for the edge given by early adoption. D has been evolving at a breakneck pace but in relative silence, and a lot of awesome things have been and are being done about it that are starting to become known just about now — some literally in this very article. At this writing, my book *The D Programming Language* is 40% complete and available for pre-order at Amazon. Safari's Rough Cuts subscription-based service makes advance chapters available here.

There are two major versions of the language — D1 and D2. This article focuses on D2 exclusively. D1 is stable (will undergo no other changes but bug fixes), and D2 is a major revision of the language that sacrificed some backwards compatibility for the sake of doing things consistently right, and for adding a few c rucial features related to manycores and generic programming. In the process, the language's complexity has increased, which is in fact a good indicator because no language in actual use has ever gotten smaller. Even languages that started with the stated intent to be "small and beautiful" inevitably grew with use. (Yes, even Lisp. Spare me.) Although programmers dream of the idea of small, simple languages, when they wake up they seem to only want more modeling power. D's state of transition is putting yours truly in the unenviable position of dealing with a moving target. I opted for writing an article that ages nicely at the expense of being occasionally f rustrating in that it describes features that are in the works or are incompletely implemented.

The official D compiler is available for free off digitalmars.com on major desktop platforms (Windows, Mac, and Linux). Other implementations are underway, notably including a .NET port and one using the LLVM infrastructure as a backend. There are also two essential D libraries, the official — Phobos — and a community-driven library called Tango. Tango, designed for D1, is being ported to D2, and Phobos (which was frustratingly small and quirky in its D1 iteration) is undergoing major changes and additions to take full advantage of D2's capabilities. (There is, unsurprisingly, an amount of politics and bickering about which library is better, but competition seems to spur both into being as good as they can be.)

Last but definitely not least, two windowing libraries complete the language's offering quite spectacularly. The mature library DWT is a direct port of Java's SWT. A newer development is that the immensely popular Qt Software windowing library has recently released a D binding (in alpha as of this writing). This is no small news as Qt is a great (the best if you listen to the right people) library for developing portable GUI applications. The two libraries fully take D into "the GUIth dimension."

D Fundamentals

D could be best described as a high-level systems programming language. It encompasses features that are normally found in higher level and even scripting languages — such as a rapid edit-run cycle, garbage collection, built-in hashtables, or a permission to omit many type declarations — but also low-level features such as pointers, storage management in a manual (a la C's *malloc/free*) or semi-automatic (using constructors, destructors, and a unique scope statement) manner, and generally the same direct relationship with memory that C and C++ programmers know and love. In fact, D can link and call C functions directly with no intervening translation layer. The entire C standard library is directly available to D programs. However, you'd very rarely feel compelled to go that low because D's own facilities are often more powerful, safer, and just as efficient. By and large, D makes a strong statement that convenience and efficiency a re not necessarily at odds. Aside from the higher level topics that we'll discuss soon, no description of D would be complete without mentioning its attention to detail: All variables are initialized, unless you initialize them with *void;* a rrays and associative arrays a re intuitive and easy on the eyes; iteration is clean; NaN is actually used; overloading rules can be understood; support for documentation and unit testing is built-in. D is multi-paradigm, meaning that it fosters writing code in object-oriented, generic, functional, and procedural style within a seamless and remarkably small package. The following bite-sized sections give away some generalities about D.

Hello, Cheapshot

Let's get that pesky syntax out of the way. So, without further ado:

```
import std.stdio;
void main()
{
    writeln("Hello, world!");
}
```

Syntax is like people's outfits — rationally, we understand it shouldn't make much of a difference and that it's shallow to care about it too much, but on the other hand we can't stop noticing it. (I remember the girl in red from *The Matrix* to this day.) For many of us, D has much of the "next door" familiar looks in that it adopted the C-style syntax also present in C++, Java, and C#. (I assume you are familiar with one of these, so I don't need to explain that D has par for the course features such as integers, floating-point numbers, arrays, iteration, and recursion.)

Speaking of other languages, please allow a cheapshot at the C and C++ versions of "Hello, world." The classic C version, as lifted straight from the second edition of K&R, looks like this:

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

and the equally classic C++ version is (note the added enthusiasm):

```
#include <iostream>
int main()
{
    std::cout << "Hello, world!\n";
}
```

Many comparisons of the popular first p rogram written in various languages revolve around code length and amount of information needed to understand the sample. Let's take a different route by discussing correctness, namely: What happens if, for whatever reason, writing the greeting to the standard output fails? Well, the C program ignores the error because it doesn't check the value returned by *printf*. To tell the truth, it's actually a bit worse; although on my system it compiles flag-free and runs, C's "hello world" returns an unpredictable number to the operating system because it falls through the end of main. (On my machine, it exits with 13, which got me a little scared. Then I realized why: *"hello, world\n"* has 13 characters; *printf* returns the number of characters printed, so it deposits 13 in the EAX register; the exit code luckily doesn't touch that register; so that's ultimately what the OS sees.) It turns out that the program as written is not even correct under the C89 or C99 standards. After a bit of searching, the Internet seems to agree that the right way to open the hailing frequencies in C is:

```
#include < stdio.h>
int main()
{
    printf("hello, world\n");
    return 0;
}
```

which does little in the way of correctness because it replaces an unpredictable return with one that always claims success, whether or not printing succeeded.

The C++ program is guaranteed to return 0 from *main* if you forgot to return, but also ignores the error because, um, at program start *std::cout.exceptions()* is zero and nobody checks for *std::cout.bad()* after the output. So both programs will claim success even if they failed to print the message for whatever reason. The corrected C and C++ versions of the global greet lose a little of their lip gloss:

```
#include <stdio.h>
int main()
{
   return printf("hello, world\n") < 0;
}
```

and

```
#include <iostream>
int main()
{
   std::cout << "Hello, world!\n";
   return std::cout.bad();
}
```

Further investigation reveals that the classic "hello, world" for other languages such as Java (code omitted due to space limitations), J# (a language completely — I mean **completely** — unrelated to Java), or Perl, also claim success in all cases. You'd almost think it's a conspiracy, but fortunately the likes of Python and C# come to the rescue by throwing an exception.

How does the D version fare? Well, it doesn't need any change: *writeln* throws on failure, and an exception issued by main causes the exception's message to be printed to the standard error stream (if possible) and the program to exit with a failure exit code. In short, the right thing is done automatically. I wouldn't have taken this cheapshot if it weren't for two reasons. One, it's fun to imagine the street riots of millions of betrayed programmers crying how their "Hello, world" program has been a sham. (Picture the slogans: "Hello, world! Exit code 13. Coincidence?" or "Hello, sheeple! Wake up!" etc.) Second, the example is not isolated, but illustrative for a pervasive pattern — D attempts not only to allow you to do the right thing, it systematically attempts to make the right thing synonymous to the path of least resistance whenever possible. And it turns out they can be synonymous more often than one might think. (And before you fix my code, *"void main()"* is legal D and does what you think it should. Language lawyers who destroy noobs writing *"void main()"* instead of *"int main()"* in C++ newsgroups would need to find another favorite pastime if they switch to D.)

Heck, I planned to discuss syntax and ended up with semantics. Getting back to syntax, there is one notable departure from C++, C#, and Java: D uses *T!(X, Y, Z)* instead of *T<X, Y, Z>*(and *T!(X)* or simply *T!X* for *T<X>*) to denote parameterized types, and for good reasons. The choice of angular brackets, when clashing with the use of '<', '>', and '>>' as arithmetic operands, has been a huge source of parsing problems for C++, leading to a hecatomb of special rules and arbitrary disambiguations, not to mention the world's least known syntax *object.template fun<arg>()*. If one of your C++ fellow coders has Superman-level confidence, ask them what that syntax does and you'll see kryptonite at work. Java and C# also adopted the angular brackets but wisely chose to disallow arithmetic expressions as parameters, thus preemptively crippling the option to ever add them later. D extends the traditional unary operator '!' to binary uses and goes with the classic parentheses (which (I'm sure) you always pair properly).

Compilation Model

D's unit of compilation, protection, and modularity is the file. The unit of packaging is a directory. And that's about as sophisticated as it goes. There's no pretense that the program source code would really feel better in a super-duper database. This approach uses a "database" tuned by the best of us for a long time, integrating perfectly with version control, backup, OS-grade protection, journaling, what have you, and also makes for a low entry barrier for development as all you need is an editor and a compiler. Speaking of which, specialized tool support is at this time scarce, but you can find things like the emacs mode d-mode, vim support, the Eclipse plug-in Descent, the Linux debugger ZeroBugs, and the full IDE Poseidon.

Generating code is a classic two-stroke compile and link cycle, but that happens considerably faster than in most similar environments, for two reasons, no, three.

- One, the language's grammar allows separate and highly optimized lexing, parsing, and analysis steps.
- Two, you can easily instruct the compiler to not generate many object files like most compilers do, and instead construct everything in memory and make only one linear commit to disk.
- Three, Walter Bright, the creator and original implementor of D, is an inveterate expert in optimization.

This low latency means you can use D as a heck of an interpreter (the shebang notation is supported, too).

D has a true module system that supports separate compilation and generates and uses module summaries (highbrowspeak for "header files") automatically from source, so you don't need to worry about maintaining redundant files separately, unless you really wish to, in which case you can. Yep, that stops that nag right in mid-sentence.

Memory Model And Manycores

Given that D can call C functions directly, it may seem that D builds straight on C's memory model. That might be good news if it weren't for the pink elephant in the room dangerously close to that Ming-dynasty vase: manycores — massively parallel architectures that throw processing power at you like it's going out of style, if only you could use it. Manycores are here, and C's way of dealing with them is extremely pedestrian and error prone. Other procedural and object-oriented languages made only little improvements, a state of affairs that marked a recrudescence of functional languages that rely on immutability to elegantly sidestep many parallelism-related problems.

Being a relative newcomer, D is in the enviable position of placing a much more informed bet when it comes to threading. And D bets the farm on a memory model that is in a certain way radically different from many others. You see, old-school threads worked like this: You call a primitive to start a new thread and the new thread instantly sees and can touch any data in the program. Optionally and with obscure OS-dependent means, a thread could also acquire the so-called thread-private data for its own use. In short, memory is by default shared across all threads. This has caused problems yesterday, and it makes today a living hell. Yesterday's problems were caused by the unwieldy nature of concurrent updates: It's very hard to properly track and synchronize things such that data stays in good shape throughout. But people were putting up with it because the notion of shared memory was a close model to the reality in hardware, and as such was efficient when gotten right. Now is where we're getting to the "living hell" part — nowadays, memory is in fact increasingly less shared. Today's reality in hardware is that processors communicate with memory through a deep memory hierarchy, a large part of which is private to each core. So not only shared memory is hard to work with, it turns out to be quickly becoming the slower way of doing things because it is increasingly removed from physical reality.

While traditional languages were wrestling with all of these problems, functional languages took a principled position stemming from mathematical purity: we're not interested in modeling hardware, they said, but we'd like to model true math. And math for the most part does not have mutation and is time-invariant, which makes it an ideal candidate for parallel computing. (Imagine the moment when one of those first mathematicians-turned-programmers has heard about parallel computing — they must have slapped their forehead: "Wait a **minute**!...") It was well noted in functional programming circles that such a computational model does inherently favor out-of-order, parallel execution, but that potential was more of a latent energy than a realized goal until recent times. Today, it becomes increasingly clear that a functional, mutation-free style of writing programs will be highly relevant for at least parts of a serious application that wants to tap into parallel processing.

So where's D positioning itself in all this? There's one essential concept forming the basis of D's approach to parallelism:

Memory is thread-private by default, shared on demand.

In D, all memory is by default private to the thread using it; even unwitting globals are allocated per-thread. When sharing is desired, objects can be qualified with *shared*, which means that they are visible from several threads at once. Crucially, the type system knows about shared data and limits what can be done with it to ensure that proper synchronization mechanisms are used throughout. This model avoids very elegantly a host of thorny problems related to synchronization of access in default-shared threaded languages. In those languages, the type system has no idea which data is supposed to be shared and which isn't so it often relies on the honor system — the programmer is trusted to annotate shared data appropriately. Then complicated rules explain what happens in various scenarios involving unshared data, shared annotated data, data that's not annotated yet still shared, and combinations of the above — in a very clear manner so all five people who can understand them will understand them, and everybody calls it a day.

Support for manycores is a very active area of research and development, and a good model has not been found yet. Starting with the solid foundation of a default-private memory model, D is incrementally deploying amenities that don't restrict its options: pure functions, lock-free primitives, good old lock-based programming, message queues (planned), and more. More advanced features such as ownership types are being discussed.

Immutability

So far so good, but what happened to all that waxing about the purity of math, immutability, and functional-style code? D acknowledges the crucial role that functional-style programming and immutability have for solid parallel programs (and not only parallel, for that matter), so it defines *immutable* as a qualifier for data that never, ever changes. At the same time, D also recognizes that mutation is often the best means to a goal, not to mention the style of programming that is familiar to many of us. D's answer is rather interesting, as it encompasses mutable data and immutable data in a seamless whole.

Why is immutable data awesome? Sharing immutable data across threads never needs synchronization, and no synchronization is really the fastest synchronization around. The trick is to make sure that read-only really means read-only, otherwise all guarantees fall apart. To support that important aspect of parallel programs, D provides an unparalleled (there goes the lowest of all literary devices right there) support for mixed functional and imperative programming. Data adorned with the *immutable* qualifier provides a strong static guarantee — a correctly typed program cannot change immutable data. Moreover, immutability is deep — if you are in immutable territory and follow a reference, you'll always stay in immutable territory (Why? Otherwise, it all comes unglued as you think you share immutable data but end up unwittingly sharing mutable data, in which case we're back to the complicated rules we wanted to avoid in the first place.) Entire subgraphs of the interconnected objects in a program can be "painted" *immutable* with ease. The type system knows where they are and allows free thread-sharing for them and also optimizes their access more aggressively in single-threaded code, too.

Is D the first language to have proposed a default-private memory model? Not at all. What sets D apart is that it has integrated default-private thread memory with immutable and mutable data under one system. The temptation is high to get into more detail about that, but let's leave that for another day and continue the overview.

Safety High On the List

Being a systems-level language, D allows extremely efficient and equally dangerous constructs: It allows unmanaged pointers, manual-memory management, and casting that can break into pieces the most careful design.

However, D also has a simple mechanism to mark a module as "safe," and a corresponding compilation mode that forces memory safety. Successfully compiling code under that subset of the language — affectionately dubbed "SafeD" — does not guarantee you that your code is portable, that you used only sound programming practices, or that you don't need unit tests. SafeD is focused only on eliminating memory corruption possibilities. Safe modules (or triggering safe compilation mode) impose extra semantic checks that disallow at compilation time all dangerous language features such as forging pointers or escaping addresses of stack variables.

In SafeD you cannot have memory corruption. Safe modules should form the bulk of a large application, whereas "system" modules should be rare and far between, and also undergo increased attention during code reviews. Plenty of good applications can be written entirely in SafeD, but for something like a memory allocator you'd have to get your hands greasy. And it's great that you don't need to escape to a different language for certain parts of your application. At the time of this writing, SafeD is not finished, but is an area of active development.

Read My Lips: No More Axe

D is multi-paradigm, which is a pretentious way of saying that it doesn't have an axe to grind. D got the memo. Everything is not necessarily an object, a function, a list, a hashtable, or the Tooth Fairy. It depends on you what you make it. Programming in D can therefore feel liberating because when you want to solve a problem you don't need to spend time thinking of ways to adapt it to your hammer (axe?) of choice. Now, truth be told, freedom comes with responsibility: You now need to spend time figuring out what design would best fit a given problem.

By refusing to commit to One True Way, D follows the tradition started by C++, with the advantage that D provides more support for each paradigm in turn, better integration between various paradigms, and considerably less friction in following any and all of them. This is the advantage of a good pupil; obviously D owes a lot to C++, as well as less eclectic languages such as Java, Haskell, Eiffel, JavaScript, Python, and Lisp. (Actually most languages owe their diction to Lisp, some just won't admit it.)

A good example of D's eclectic nature is resource management. Some languages bet on the notion that garbage collection is all you need for managing resources. C++ programmers recognize the merit of RAII and some say it's everything needed for resource management. Each group lacks intimate experience with the tools of the other, which leads to comical debates in which the parties don't even understand each other's arguments. The truth is that neither approach is sufficient, for which reason D breaks with monoculture.

Object-Oriented Features

In D you get *structs* and then you get *classes*. They share many amenities but have different charters: *structs* are value types, whereas *classes* are meant for dynamic polymorphism and are accessed solely by reference. That way confusions, slicing-related bugs, and comments a la *// No! Do NOT inherit!* do not exist. When you design a type, you decide upfront whether it'll be a monomorphic value or a polymorphic reference. C++ famously allows defining ambiguous-gender types, but their use is rare, error-prone, and objectionable enough to warrant simply avoiding them by design.

D's object-orientation offering is similar to Java's and C#'s: single inheritance of implementation, multiple inheritance of interface. That makes Java and C# code remarkably easy to port into a working D implementation. D decided to forgo language-level support for multiple inheritance of implementation, but also doesn't go with the sour-grapes theory "Multiple Inheritance is Evil: How an Amulet Can Help." Instead, D simply acknowledges the difficulty in making multiple inheritance of state work efficiently and in useful ways. To provide most of the benefits of multiple inheritance at controllable cost, D allows a type to use multiple subtyping like this:

```
class WidgetBase { ... }
class Gadget { ... }
class Widget : WidgetBase, Interface1,
Interface2
{
   Gadget getGadget() { ... }
   alias getGadget this; // Widget
subtypes Gadget!
}
```

The alias introduction works like this: Whenever a *Gadget* is expected but all you have is a *Widget*, the compiler calls *getGadget* and obtains it. The call is entirely transparent, because if it weren't, that wouldn't be subtyping; it would be something frustratingly close to it. (If you felt that was an innuendo, it probably was.) Moreover, *getGadget* has complete discretion over completing the task — it may return e.g. a subobject of this or a brand new object. You'd still need to do some routing to intercept method calls if you need to, which sounds like a lot of boilerplate coding, but here's where D's reflection and code generation abilities come to fore (see below). The basic idea is that D allows you to subtype as you need via *alias this*. You can even subtype *int* if you feel like it.

D has integrated other tried and true techniques from experience with object orientation, such as an explicit override keyword to avoid accidental overriding, signals and slots, and a technique I can't mention because it's trademarked, so let's call it contract programming.

Functional Programming

Quick, how do you define a functional-style Fibonacci function?

```
uint fib(uint int n)
{
    return n < 2 ? n : fib(n - 1) + fib(n
- 2);
}
```

I confess to entertaining fantasies. One of these fantasies has it that I go back in time and somehow eradicate this implementation of Fibonacci such that no Computer Science teacher ever teaches it. (Next on the list are bubble sort and the *O(n log n)*-space quicksort implementation. But *fib* outdoes both by a large margin. Also, killing Hitler or Stalin is dicey as it has hard-to-assess consequences, whereas killing *fib* is just good.) *fib* takes exponential time to complete and as such promotes nothing but ignorance of complexity and of the costs of computation, a "cute excuses sloppy" attitude, and SUV driving. You know how bad exponential is? *fib(10)* and *fib(20)* take negligible time on my machine, whereas *fib(50)* takes nineteen and a half minutes. In all likelihood, evaluating *fib(1000)* will outlast humankind, which gives me solace because it's what we deserve if we continue teaching bad programming.

Fine, so then what does a "green" functional Fibonacci implementation look like?

```
uint fib(uint n)
{
    uint iter(uint i, uint fib_1, uint fib_2)
   {
       return i == n
          ? fib_2
          : iter(i + 1, fib_1 + fib_2, fib_1);
   }
    return iter(0, 1, 0);
}
```

The revised version takes negligible time to complete *fib(50)*. The implementation now takes *O(n)* time, and tail call optimization (which D implements) takes care of the space complexity. The problem is that the new *fib* kind of lost its glory. Essentially, the revised implementation maintains two state variables in the disguise of function parameters, so we might as well come clean and write the straight loop that *iter* made unnecessarily obscure:

```
uint fib(uint n)
{
   uint fib_1 = 1, fib_2 = 0;
   foreach (i; 0 .. n)
  {
      auto t = fib_1;
      fib_1 += fib_2;
      fib_2 = t;
   }
   return fib_2;
}
```

but (shriek of horror) this is not functional anymore! Look at all that disgusting mutation going on in the loop! One mistaken step, and we fell all the way from the peaks of mathematical purity down to the unsophisticatedness of the unwashed masses.

But if we sit for a minute and think of it, the iterative *fib* is not that unwashed. If you think of it as a black box, *fib* always outputs the same thing for a given input, and after all pure is what pure does. The fact that it uses private state may make it less functional in letter, but not in spirit. Pulling carefully on that thread, we reach a very interesting conclusion: as long as the mutable state in a function is entirely **transitory** (i.e., allocated on the stack) and private (i.e., not passed along by reference to functions that may taint it), then the function can be considered pure.

And that's how D defines functional purity: you can use mutation in the implementation of a pure function, as long as it's transitory and private. You can then put pure in that function's signature and the compiler will compile it without a hitch:

```
pure uint fib(uint n)
{
    ... iterative implementation ...
}
```

The way D relaxes purity is quite useful because you're getting the best of both worlds: iron-clad functional purity guarantees, and comfortable implementation when iteration is the preferred method. If that's not cool, I don't know what is.

Last but not least, functional languages have another way of defining a Fibonacci sequence: as a so-called infinite list. Instead of a function, you define a lazy infinite list that gives you more Fibonacci numbers the more you read from it. D's standard library offers a pretty cool way of defining such lazy lists. For example, the code below outputs the first 50 Fibonacci numbers (you'd need to *import std.range*) :

```
foreach (f; take(50, recurrence!("a[n-1] + a[n-2]")(0, 1)))
{
    writeln(f);
}
```

That's not a one-liner, it's a half-liner! The invocation of recurrence means, create an infinite list with the recurrence formula *a[n] = a[n-1] + a[n-2]* starting with numbers 0 and 1. In all this there is no dynamic memory allocation, no indirect function invocation, and no nonrenewable resources used. The code is pretty much equivalent to the loop in the iterative implementation. To see how that can be done, you may want to read through the next section.

Generic Programming

(You know the kind of caution you feel when you want to describe to a friend a movie, a book, or some music you really like but don't want to spoil by overselling? That's the kind of caution I feel as I approach the subject of generic programming in D.) Generic programming has several definitions — even the neutrality of the Wikipedia article on it is being disputed at the time of this writing. Some people refer to generic programming as "programming with parameterized types, aka templates or generics," whereas others mean "expressing algorithms in the most generic form that preserves their complexity guarantees." I'll discuss a bit of the former in this section, and a bit of the latter in the next section.

D offers parameterized structs, classes, and functions with a simple syntax, for example here's a *min* function:

```
auto min(T)(T a, T b) { return b < a ? b : a; }
...
auto x = min(4, 5);
```

T would be a type parameter and *a* and *b* are regular function parameters. The *auto* return type leaves it to the compiler to figure out what type *min* returns. Here's the embryo of a list:

```
lass List(T)
{
   T value;
   List next;
   ...
}
...
List!int numbers;
```

The fun only starts here. There's too much to tell in a short article to do the subject justice, so the next few paragraphs offer "deltas" —

differences from the languages with generics that you might already know.

Parameter Kinds. Not only types are acceptable as generic parameters, but also numbers (integral and floating-point), strings, compile-time values of *struct* type, and aliases. An *alias* parameter is any symbolic entity in a program, which can in turn refer to a value, a type, a function, or even a template. (That's how D elegantly sidesteps the infinite regression of template template template — parameters; just pass it as an alias.) Alias parameters are also instrumental in defining lambda functions. Variable-length parameter lists are also allowed.

String Manipulation. Passing strings to templates would be next to useless if there was no meaningful compile-time manipulation of strings. D offers full string manipulation capabilities during compilation (concatenation, indexing, selecting a substring, iterating, comparison....)

Code Generation: The Assembler of Generic Programming. Manipulating strings during compilation may be interesting, but is confined to the data flatland. What takes things into space is the ability to convert strings into code (by use of the *mixin* expression). Remember the recurrence example? It passed the *recurrence* formula for Fibonacci sequences into a library facility by means of a string. That facility in turn converted the string into code and provided arguments to it. As another example, here's how you sort ranges in D:

```
// define an array of integers
auto arr = [ 1, 3, 5, 2 ];
// sort increasingly (default)
sort(arr);
// decreasingly, using a lambda
sort!((x, y) { return x > y; })(arr);
// decreasingly, using code generation; comparison is
// a string with conventional parameters a and b
sort!("a >  b")(arr);
```

Code generation is very powerful because it allows implementing entire features without a need for language-level support. For example, D lacks bitfields, but the standard module *std.bitmanip* defines a facility implementing them fully and efficiently.

Introspection. In a way, introspection (i.e., the ability to inspect a code entity) is the complement of code generation because it looks at code instead of generating it. It also offers support for code generation — for example, introspection provides the information for generating a parsing function for some enumerated value. At this time, introspection is only partially supported. A better design has been blueprinted and the implementation is "on the list," so please stay tuned for more about that.

is and static if. Anyone who's written a nontrivial C++ template knows both the necessity and the encumbrances of (a) figuring out whether some code "would compile" and deciding what to do if yes vs. if not, and (b) checking for Boolean conditions statically and compiling in different code on each branch. In D, the Boolean compile-time expression *is(typeof(expr))* yields *true* if *expr* is a valid expression, and *false* otherwise (without aborting compilation). Also, *static if* looks pretty much like *if*, except it operates during compilation on any valid D compile-time Boolean expression (i.e., *#if* done right). I can easily say these two features alone slash the complexity of generic code in half, and it filled me with chagrin that C++0x includes neither.

But Wait, There's. . .Well, You Know. Generic programming is a vast play field, and although D covers it with a surprisingly compact conceptual package, it would be hard to discuss matters further without giving more involved information. D has more to offer, such as customized error messages, constrained templates a la C++0x concepts (just a tad simpler — what's a couple of orders of magnitude between friends?), tuples, a unique feature called "local instantiation" (crucial for flexible and efficient lambdas), and, if you call within the next five minutes, a knife that can cut through a frozen can of soda.

A Word on the Standard Library

This subject is a bit sensitive politically because, as mentioned, there are two full-fledged libraries that can be used with D, Phobos and Tango. I only worked on the former so I will limit my comments to it. For my money, ever since the STL appeared, the landscape of containers+algorithms libraries has forever changed. It's changed so much, in fact, that every similar library developed after the STL but in ignorance of it runs serious risks of looking goofy. (Consequently, a bit of advice I'd give a programmer in any language is to understand the STL.) This is not because STL is a perfect library — it isn't. It is inextricably tied to the strengths and weaknesses of C++; for example, it's efficient but it has poor support for higher order programming. Its symbiosis with C++ also makes it difficult for non-C++ programmers to understand the STL in abstract, because it's hard to see its essence through all the nuts and bolts. Furthermore, STL has its own faults; for example, its conceptual framework fails to properly capture a host of containers and ways of iterating them.

STL's main merit was to reframe the entire question of what it means to write a library of fundamental containers and algorithms, and to redefine the process of writing one in wake of the answer. The question STL asked was: "What's the minimum an algorithm could ever ask from the topology of the data it's operating on?" Surprisingly, most library implementers and even some algorithm pundits were treating the topic without due rigor. STL's angle put it in stark contrast to a unifying interface view in which, for example, it's okay to unify indexed access in arrays and linked lists because the topological aspect of performing it can be written off as just an implementation detail. STL revealed the demerit of such an approach because, for example, it's disingenuous to implement as little as a linear search by using an unifying interface (unless you enjoy waiting for quadratic algorithms to terminate). These are well-known truths to anyone serious in the least about algorithms, but somehow there was a disconnect between understanding of algorithms and their most general implementations in a programming language. Although I was conversant with algorithm fundamentals, I can now say I had never really thought of what the pure, quintessential, Platonic linear search is about until I first saw it in the STL 15 years ago.

That's a roundabout way of saying that Phobos (places to look at in the online documentation: *std.algorithm* and *std.range*) is a lot about the STL. If you ask me, Phobos' offering of algorithms is considerably better than STL's, and for two reasons. One, Phobos has the obvious advantage of climbing on the shoulders of giants (not to mention the toes of dwarfs). Two, it uses a superior language to its fullest.

Ranges are Hot, Iterators are Not. Probably the most visible departure from the STL is that there are no iterators in Phobos. The iterator abstraction is replaced with a range abstraction that is just as efficient but offers vastly better encapsulation, verifiability, and abstraction power. (If you think about it, none of an iterator's primitive operations are naturally checkable. That's just bizarre.) Code using ranges is as fast, safer, and terser than code using iterators — no more for loop that's too long to contain in one line. In fact, thinking and coding with ranges is so much terser, new idioms emerge that may be thinkable but are way too cumbersome to carry through with iterators. For example, you might have thought of a *chain* function that iterates two sequences one after another. Ve ry useful. But *chain* with iterators takes four iterators and returns two, which makes it too ham-fisted to be of any use. In contrast, *chain* with ranges takes two ranges and returns one. Furthermore, you can use variadic arguments to have *chain* accept any number of ranges — and all of a sudden, we can avail ourselves of a very useful function. chain is actually implemented in the standard module std.range. As an example, here's how you can iterate through three arrays:

```
int[] a, b, c;
...
foreach (e; chain(a, b, c))
{
    ... use e ...
}
```

Note that the arrays are not concatenated! chain leaves them in place and only crawls them in sequence. This means that you might think you could change elements in the original arrays by means of chain, which is entirely true. Guess what this code does:

```
sort(chain(a, b, c));
```

You're right — the collective contents of the three arrays has been sorted and, without modifying the size of the arrays, the elements have been efficiently arranged such that the smallest come in *a* and so on. This is just a small example of the possibilities offe red by ranges and range combinators in conjunction with algorithms.

Laziness to Infinity and Beyond. STL algorithms (and many others) are eager: By the time they return, they've finished their job. In contrast, Phobos uses lazy evaluation whenever it makes sense. By doing so, Phobos acquires better composition capabilities and the ability to operate with infinite ranges. For example, consider the prototypical higher order function *map* (popular in functional programming circles, not to be confused with the homonym STL data stru c-ture) that applies a given function to each element in a range. If *map* were insisting to be eager, there'd be two problems.

- First, it would have to allocate new space to store the result (e.g., a list or an array).
- Second, it would have to consume the range in its entirety before returning control to the caller.

The first is an efficiency problem: Memory allocation could and should be avoided in many cases (for example, the caller wants to just look at each result of map in turn). The second is a problem of principles: Eager map simply can't deal with infinite ranges because it would loop forever.

That's why Phobos defines map to return a lazy range — it incrementally makes progress as you consume elements from it. In contrast, the reduce function (in a way, a converse of *map*) is eager. Some functions need both lazy and eager versions. For example, *retro(r)* returns a range that iterates the given range *r* backwards, whereas *reverse(r)* reverses *r* in place.

Conclusion

There would be more things to talk about even in an overview, such as unit tests, UTF strings, compile-time function evaluation (a sort of D interpreter running during compilation of a D program), dynamic closures, and many others. But with any luck, your curiosity has been piqued. If you are looking for a system-level language without the agonizing pain, an application language without the boredom, a principled language without the attitude, or — most importantly — a weighted combination thereof, then D may be for you.

If you feel like asking further questions, write the author, or better yet, tune to the Usenet server news.digitalmars.com and post to the digitalmars.d newsgroup — the hub of a vibrant community.

Acknowledgments

Many thanks to Scott Meyers who pointed out the necessity of such an article and suggested its title. I have gotten excellent reviews, feedback, and suggestions from Bill Baxter, Jason House, John "Eljay" Love-Jensen, Denis Koroskin, leonardo maffi (sic), Petru Marginean, Ba rtosz Milewski, Derek Parnell, Brad Roberts, Joel Salomon, Benjamin Shropshire, David Simcha, Florin Trofin, Cristian Vlasceanu, and Walter Bright.

— Andrei Alexandrescu is the author of Modern C++ Design *and* The D Programming Language. *He can be contacted at erdani.org.*

Return to Table of Contents

Debugging MySQL Stored Procedures

Being able to do low- or no-cost logging with Stored Procedures is an extremely useful technique

by Brian J. Tarbox

Stored Procedures are programs that execute within a database server. They are usually written in a database language such as PL/SQL or ANSI SQL:2003 SQL/PSM. (Granted, some database servers do support Java stored procedures, but I don't examine them here.) There are any number of books for learning to write Stored Procedures — *MySQL Stored Procedure Programming*, by Guy Harrison, and *Teach Yourself PL/SQL in 21 Days*, by Jonathan Gennick and Tom Luers, come to mind), but there are a handful of general reasons to write code in a stored procedure:

- The logic being implemented might be database logic and so a database language is closer to the problem domain than a general-purpose language like Java.
- A stored procedure can be significantly faster than a Java program, which might make multiple calls to the database.
- A stored procedure can be more secure.

Regardless of the reasons for choosing to write a stored procedure, the problem of how to debug one remains. What if you could debug in both development and production at little to no cost to the performance of the Stored Procedures? Traditional debuggers do not generally work with stored procedures, which can leave a developer with a fast and broken procedure executing within their database server.

Approaches That Don't Work

Debug the SQL in your Stored Procedure. This approach works on the assumption the main logic of your Stored Procedures is the actual DDL and DML within the procedure, in other words, the queries, inserts, and so on. It assumes that the rest of the Stored Procedures is largely scaffolding to support the database operations. In many cases this is a valid assumption, after all if the Stored Procedure wasn't manipulating the database you probably wouldn't have written it as a Stored Procedure.

It goes without saying that regardless of how much non-SQL code you have in your Stored Procedures, you need to validate the SQL itself, especially since this level of testing can be relatively straightforward. It can be as simple as starting your database command-line tool (or query browser for the GUI-inclined) and pasting in the guts of your SQL statements to verify correctness. This of course goes beyond simple syntactic correctness, you must validate the semantic correctness as well.

In some cases, however, it's not quite that simple, for a couple of classes of reasons. First, your SQL code can (and usually will) rely on variables and parameters that have been defined and/or manipulated by the Stored Procedures. If you have a select statement that stored its results into a variable, and then a later SQL statement that uses that variable, then your "paste the sql into the command line" approach of testing gets a bit harder. You have to insert one or more statements, execute them, perhaps creating temporary variables along the way, and possibly modify the SQL you are actually trying to test. This happens by degree but you can certainly reach a point where it's clear that you're no longer testing the SQL you started with.

The second class of problem with this approach is that often the logic of the Stored Procedures lives in the procedural code of the procedure and not in the SQL statements themselves. SPs are commonly used to instantiate business logic — and this is usually embodied in the flow of the code through the procedure or procedures. In this kind of situation simply bench testing the SQL statements does not really test the procedure.

Insert *print* statements in your Stored Procedure. Another common approach is to sprinkle *print* statements throughout your procedure.

This has also been described as "Sherman set the way back machine to 1980" when *print* statements were about the only game in town. This approach can actually be very useful, especially during the early stages of development. Each database server tends to have its own way of doing *print* statements and each has their own idiosyncrasies. For example, when using MySQL *concat()* calls to build up a string to output, you have to guard against null values, which turn your entire string to null. The following code can be danagerous:

```
select someColumn from someTable into myVar where.
concat( 'better hope myVar is not null ', myVar);
```

If the *where* condition results in no rows being selected then *myVar* might be null and the output of the *concat* will also be null. It's better to use *concat_ws("delimiter", "text to store")* which handles null values appropriately.

There are two main drawbacks to using *print* statements in this way. First, the *print* statements are live during production (unless you guard each one with a conditional flag), meaning that you pay the significant perf o rmance penalty for logging all the time.

Second and more serious is that if your stored procedures are invoked from a Java application, the *print* statements don't go anywhere. The *print* statements can only be seen if you execute your Stored Procedures from the command line. What's the point of log messages that you can't see?

Develop a rigorous set of return codes. In this approach, you define a detailed set of return codes to cover all interesting cases. The implied contract here is that a given specific retum code tells you eve rything you need to know about the execution of the procedure. Theoretically this is a fine approach, but in the real production world it tends to fall apart. A return code might tell you what finally went wrong with a procedure but it's just too easy to imagine needing to know more about how the procedure got to that failure condition.

Put another way, if you get a support call from your most important customer at 3:00 AM do you want to have to a grand total of one return code to tell you what went wrong?

The Pro posed Approach

The current approach uses several of MySQL's special features to c reate a logging scheme that is both robust and practically free. We create two tables: a temporary log table using the Memory engine and a permanent log table using the MyISAM engine. MySQL supports several storage engines that act as handlers for diffe rent table types. MySQL storage engines include both those that handle transaction-safe tables and those that handle non-transaction-safe tables. The memory engine perf o rms all operations in memory and never writes to disk. As such it is very fast, though transient. The MyISAM engine is a nontransactional engine; transactions can include both transactional and non-transactional tables (the nontransactional table simply ignore the transactional commands).

Log messages are inserted into the tmplog table, which is a practically free operation since that table lives in memory. The extremely low cost of this operation means that a developer can be verbose in their use of logging. Rather than agonize over whether to log something or not, you can simply log at will.

In the (hopefully) usual case where nothing goes wrong, the Stored Procedures does not have to do anything. Temporary tables only exist for the duration of a connection. In the typical J2EE usage pattem, an external request arrives at the system, a connection is retrieved from the connection pool, and is used and then returned to the pool. When the connection is returned to the pool the temporary table is effectively dropped — the code does not have to do anything to remove the log messages that were inserted during the successful invocation of the Stored Procedures. In this way, the system suffers little to no perf o rmance cost for a successful operation.

When the Stored Procedure detects a failure condition, it does an INSERT SELECT from the temporary memory table into the permanent MyISAM table. Thus it preserves all log messages that were written to the memory table into the MyISAM table. In this way the system records all of the information it needs but only in the cases where it needs it.

Log4J users can imagine being able to run your production system at DEBUG level for all the failure cases but only pay the overhead of running at WARN level for all of the successful cases — without having to know which cases were which ahead of time.

An important thing to note is the choice of the MyISAM engine for the permanent log table. Remember that rows are typically only written to that table when things are going badly. That would usually result in the current transaction being rolled back — except that we really want the inserts into the log table to succeed! The MyISAM engine is not a transactional engine. This means that even if a transaction is rolled back, the inserts into the log table are prese rved — which is exactly the desired behavior.

The Code

There are four SPs defined in the debugLogging.sql package. Two SPs help setup the tables to be used, one perf o rms temporary logging, and the last copies messages to the permanent table.

This first procedure creates both the temporary and permanant tables. Note the use of the engine specifier to distinguish the two tables. The temporary log contains a single interesting column, which was called *msg*. The permanent table adds an automatic timestamp and a *thingID*. The assumption is that the logs are written during an operation that creates, destroys, or modifies some object, and that that object has a unique identifier. In the video-on-demand space that I work in, this can be the identifier of a movie being streamed to a customer's set-top box.

```
Create Procedure setupLogging()
 BEGIN
      create temporary table if not exists tmplog (msg varchar(512))
           engine = memory;
      create table if not exists log (ts timestamp default
           current_timestamp, thingID bigint,
         msg varchar(512)) engine = myisam;
  END;
```

This next procedure creates just the temporary log table. We've seen cases in the field where the temporary table does not exist when we need to insert into it. In that case, we have a procedure to recreate the table.

```
Create Procedure setupTmpLog()
 BEGIN
     create temporary table if not exists tmplog
       (msg varchar(512)) engine = memory;
  END;
```

The next procedure is the one that is called the most and performs the actual logging. A message parameter is written to the temporary table. There is a *continue* handler that creates the temporary table if it does not already exist.

```
Create Procedure doLog(in logMsg varchar(512))
BEGIN
 Declare continue handler for 1146 — Table not found
 BEGIN
   call setupTmpLog();
   insert into tmplog values( 'resetup tmp table ');
   insert into tmplog values(logMsg);
 END;

  insert into tmplog values(logMsg);
 END;
```

The last procedure is the one to call when an error is detected within the user's SP and all of the logging needed to be saved for later analysis. This procedure takes a final message, presumably one saying what the final error condition is. It then inserts all of the rows in the temporary table into the permanent table, setting the timestamp along the way. To handle the situation where the user of the logging system chooses to continue after an error, and then runs into another error, we make sure to delete all rows in the temporary table after inserting them into the permenant table.

```
Create Procedure saveAndLog(in thingId int, in lastMsg varchar(512))
BEGIN
   call dolog(lastMsg);
   insert into log(thingId, msg) (select thingId, msg from tmplog);
   truncate table tmplog;
END;
```

Sample Use

The following code sample shows a possible usage of the logging procedures. The example is based on the notion of a Stored Procedure that would be called from within another stored procedure. The mission of the inner procedure is to parse a comma-separated list of values and insert each value into a cache table for use in a query by the calling procedure. It illustrates the value of being able to log not just the final results but the steps followed along the way since it might be very valuable to know which iteration of the loop contained the failure.

```
Create Procedure parseAndStoreList(in thingId int,
                 in i_list varchar (128),
                 out returnCode smallInt)
 BEGIN
 DECLARE v_loopIndex default 0;
 DECLARE Exit Handler for SQLEXCEPTION
 BEGIN
   call saveAndLog(thingId,  'got exception parsing list '); —
        save the logs if things go badly
   set returnCode = -1;
 END;

call dolog(concat_ws( 'got list: ', i_list)); — say we got to the start
pase_loop: LOOP
  set v_loopIndex = v_loopIndex + 1;
  call dolog(concat_wc( ', ',  'at loop iteration  ', v_loopIndex);
  — say we got to nth iteration — actually do the parsing, or whatever
 END LOOP parse_loop;
 set returnCode = 0;
END;
```

Conclusion

The ability to do low- or no-cost logging with Stored Procedures has proven to be an extremely useful technique. It lets you instrument your code with extensive information, which is available at production runtime, but only when needed.

—Brian Tarbox is a Principal Staff Engineer in Motorola's Home and Network Mobility group.

Return to Table of Contents

Experiences with Kanban

Somewhere between the structure afforded by Scrum and the fluidity of Extreme Programming, Kanban is a very lean Agile development technique

by Charles Suscheck

In 2007, the company I work for purchased the rights to a medical practice management and billing system from a third-party software development firm. Technically, the software is an n-tiered product written in .NET that can be fully hosted at a client site or on our servers. A project was launched to rebrand the product, interface the product into existing business lines, test and debug the product, and add market differentiators.

Because the product was new, the work was highly exploratory with rapidly evolving requirements that could only come to light by learning the product. We decided to use Agile development supported by Scrum and elements of Extreme Programming. The entire team went through a tremendous learning curve in both learning Agile development and the product itself. Without Agile's benefit of small iterations and the chance to adjust our progress, the project most likely would have had significant difficulties. After 18 months, the project reached a high level of agile maturity, even incorporating numerous lean principles.

Our Scrum process was solid: Two-week sprints with a demo at the end, seven people consistently on the sprint teams, burndown charts, sprint planning for each sprint with whole team commitment, estimation by story points, velocity measures, and a story point estimation session with the entire team one week before each sprint.

The Problem

At first, the business side of the company was not accustomed to the responsiveness of Agile development. The norm was a formal waterfall approach to software development that could require several quarters to develop a software release. But it didn't take long for the business to develop such a hunger for change that requirements (user stories) were barely stable enough for a two-week sprint. It was not unusual for the product owner group to give top priority to stories that were not well thought out, forcing development to cobble together a sprint full of unknowns. Changes also worked their way directly into the sprints — in mid-sprint new stories were swapped with stories that hadn't been started. It seemed as though a two-week sprint was too long as the business side of the project swung from rigid change control to demanding immediate responsiveness. Something had to be done.

Our Solution

The management team brainstormed around ways to make Scrum as flexible as possible on intake. We considered ways to shorten the sprint length, the estimation sessions, sprint planning, and sprint demo length even further, all with the goal of keeping people working on stories that were of highest value to our product owner and increasing productivity. (See the sidebar for ideas on Scrum.)

Where Can Scrum Be Leaned Out?

Our thought process was to take lean principles from Mary Poppendieck's *Implementing Lean Software Development: From Concept to Cash* and apply them to the meta processes of Scrum, posing questions such as:

- Why do we have two-week sprints? Isn't that an artificial and therefore wasteful limit that batches up work?
- Why do we have seven people on the team? Can we have fewer (or more) team members?
- Why do we demo at the end of a sprint and not when the story is complete? Doesn't this sound like batching work?

- Why do we estimate story points in an estimation session when some of those stories may not be played because of reprioritization?
- Shouldn't estimates be done ONLY by those working on a story? Having people that will not work on the story estimate seems like a handoff situation.
- Why do we work on several stories during a sprint? Can we work on just one and reduce inventories of work?

Our conclusion was that the basics of software Kanban, as described by Corey Ladas (leansoftwareengineering.com/ksse/scrum-ban/), would fit the bill.

How Kanban Worked for Us

While Kanban means "visual card" or "board", using Kanban in software development is much more than a card display board. It involves reducing waste and emphasizing the ability to change, partly through limiting availability of inventory (the story cards of Kanban). In our process, the work is divided into a series of stories. The stories then end up on a series of lists: the Most Wanted list, the Pre-ready list, the Kanban board (Ready, In Process, and Done). The only status the Kanban team saw were the Ready, In Process, and Done lists. Figure 1 shows the story progression.

The product owner group creates rough cut stories and sequences them in a Most Wanted list. Analysts groom the top stories in the list and, once they are detailed, put them in the Pre-ready list in a tool called "Jira." The scrum master moves stories from the Pre-ready list to the Ready list on the Kanban board only when the development team pulls a story from the Ready list. The number of stories in the Ready list is kept small — four to eight — for two reasons: Only the highest priority stories are worked, and stories are flexible until the Kanban team pulls the story. We also limit work by maintaining a strict rule that a teamlet (one or two people) can only work on one story at a time unless a story is blocked; then, and only then, can a second story be pulled in.

While the analysis work may seem like a mini-waterfall, it is certainly not. The analysts learn the details of the story and collect information for the development group. When a story is pulled into the Kanban, the analyst is a first-class member of the teamlet, just like on a sprint team.

When a teamlet picks a story, the story is estimated in ideal hours. If the story proves to be more than a week's worth of work, we meet with the analysts and break the story into smaller segments, if at all possible.

As stories are completed, the teamlet demonstrates them to the analysts and product owner group. A short retrospective follows with the teamlet, very much like Scrum, but at a story level rather than sprint level.

Figure 2 is our Kanban board. The In Process list is further subdivided into Started (stories that are being analyzed and digested by the team), Code/Test (stories that are being coded via test driven development), Awaiting Test (stories that are awaiting acceptance testing), and Pending Release (stories that are acceptance tested but are awaiting a patch build). Each story is written on a green card with an orange tag indicating patch number, and a blue tag indicating who is working on the story.

Every day we have standup meetings where the commitments for the day are written on the purple tags next to the stories. There is also an issue list at the bottom of the board for technical debt or other items that need to be addressed by either the teams or the scrum master. A typical story will progress through each list on the Kanban board in just a few days.

As you can see in Figure 2, acceptance testing and patch builds are currently batched. This is due to staffing restrictions and other circumstances beyond our control. While this situation is not perfect, our Kanban board makes the batching under Awaiting Test and Pending Release highly visible.

Table 1 shows a number of points where our Scrum practices were modified to work with Kanban. We continue to use a number of Scrum practices where it adds value such as daily standups, story cards, and information radiators.

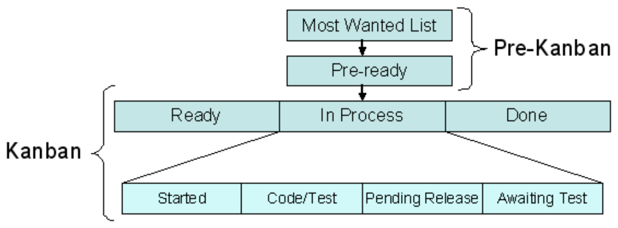


Figure 1

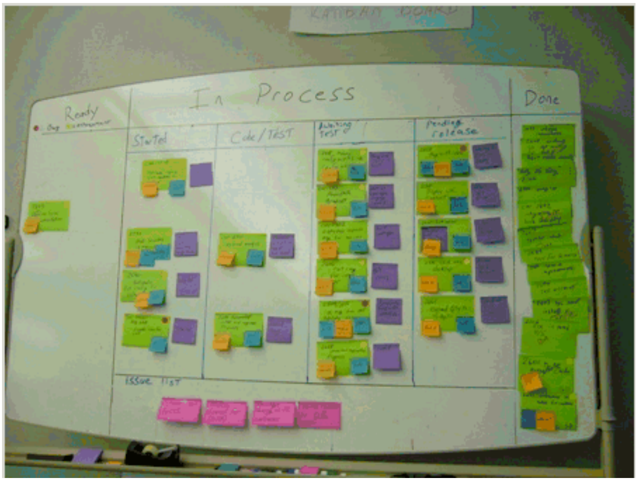
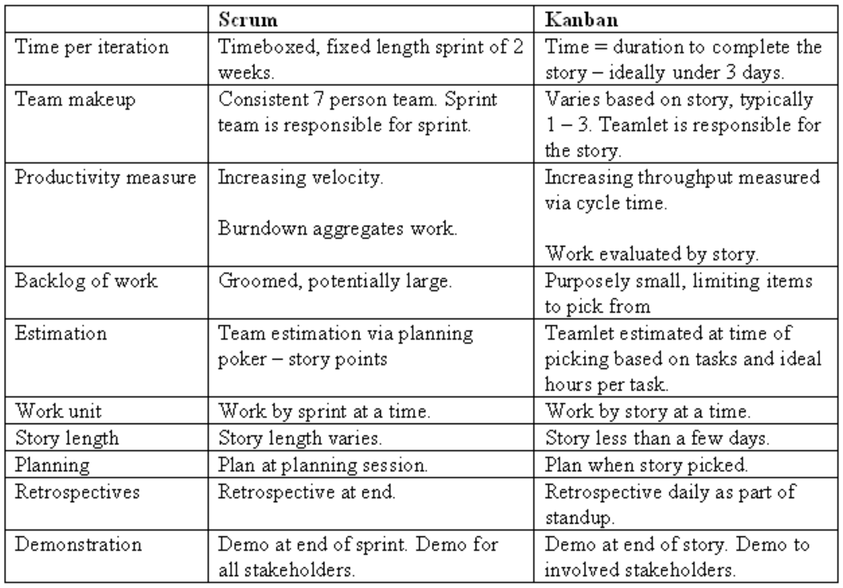


Figure 2



	Scrum	Kanban
Time per iteration	Timeboxed, fixed length sprint of 2 weeks.	Time = duration to complete the story – ideally under 3 days.
Team makeup	Consistent 7 person team. Sprint team is responsible for sprint.	Varies based on story, typically 1 – 3. Teamlet is responsible for the story.
Productivity measure	Increasing velocity. Burndown aggregates work.	Increasing throughput measured via cycle time. Work evaluated by story.
Backlog of work	Groomed, potentially large.	Purposely small, limiting items to pick from
Estimation	Team estimation via planning poker – story points	Teamlet estimated at time of picking based on tasks and ideal hours per task.
Work unit	Work by sprint at a time.	Work by story at a time.
Story length	Story length varies.	Story less than a few days.
Planning	Plan at planning session.	Plan when story picked.
Retrospectives	Retrospective at end.	Retrospective daily as part of standup.
Demonstration	Demo at end of sprint. Demo for all stakeholders.	Demo at end of story. Demo to involved stakeholders.

Table 1: Modifications of Scrum to move to Kanban

What We Lost and Gained

Kanban has a number of clear advantages. In particular, there is a large degree of flexibility in story prioritization. As stories are not cast in stone until they are pulled into the Kanban, there is very little waste developing stories that aren't played and significant flexibility in reprioritizing the backlog. Stories are typically a day's worth of work, so emergency fixes can be pulled in almost immediately without disrupting the work in progress. Stories are autonomous units of work so there is no need to group stories based on a sprint goal, or release goal for that matter — there is less emphasis (and time spent) on planning sessions.

Kanban is highly productive. The amount of work completed in just a few months rivaled that of six months of Scrum because there is little wasted time. A number of things contribute to the productivity. I found that team members were not distracted with the end of the sprint push like they were in Scrum. Because high-priority stories can be pulled from the backlog without waiting until the end of a two-week sprint, there is less temptation for management to change priorities in the middle of work, resulting in reduced context switching for the developers. The developers have something they can produce quickly, leading to a good sense of accomplishment and a chance to react quickly to lessons learned from a story. Altogether, the team and the business are pleased with the application of Kanban.

However, Kanban does not come without a number of challenges. With teamlets forming and reforming on a story by story basis, there is a danger of team cohesion being lost. Fortunately, we did not experience cohesion problems because most team members have had together for a long time. Kanban simply became a new way to pull work. Several Scrum measurements are lost. Predictability by sprint is lost (no sprint) and velocity is no longer measurable. The only metric is cycle time though the Kanban board — which only works if stories are all the same size. Also, it is hard to tell when a certain story in the backlog will be completed because new stories can shuffle the priority frequently.

Conclusion and Recommendations

Kanban seems like a logical next evolutionary step to Scrum, but only in the right circumstances — projects where the work doesn't need to be completed in groups of stories. Kanban has a number of advantages such as allowing extensive flexibility of the backlog, more flexibility in staffing, and the ability to manage work in the face of uncertainty.

Kanban is successful in our environment, although it should be applied carefully. Kanban has increased productivity and given the business a chance to react quickly to lessons learned and, because of our team experience, cohesion has not been a problem.

I can see difficulties with using Kanban in a product development project; a pure Scrum-based approach may be better. The benefits of Scrum, such as iteration 0, sprint planning, release planning, and team protection are lost with Kanban. Collaborative product design and technical design are part of the sprint planning that is not part of Kanban. Even the teamlet concept, rather than Scrum's whole team effort, can lead to a a certain level of lost collaborative design. If you decide to use Kanban for product development, you must have strong leadership from a product owner.

Kanban can be effective, I know from experience, but it should not be applied without understanding the trade-offs. Keep up with the literature on the Internet; while it's a new idea, there is a wealth of information already available.

—Dr. Charles Suscheck is a consultant specializing in software process optimization. He specializes in software development methodologies and project management and has over 20 years of professional experience in information technology. Dr. Suscheck has held positions of Process Architect, Director of Research, Principle Consultant, and Professional Trainer at some of the most recognized companies in America. He has spoken at national and international conferences such as OOPSLA, ECOOP, and Borcon on topics related to project management.

Return to Table of Contents

Monitoring Remote Tasks in AJAX

The Progress Indicator pattern lets the JavaScript client and the server application share information

by Dino Esposito

When users start a potentially lengthy operation, the user interface should be updated to reflect that work is in progress and that results may not be available for a while. Implementing this pattern is relatively easy in Windows applications, but not in web applications.

In web applications, displaying a static text such as *"Please, wait"* just before the operation begins is easy, but what if you want to display the percentage of work done?

In web applications, lengthy tasks occur on the server. No server environment provides facilities to push state information to the client browser. At the same time, there's no easy way either for a client to grab status information and update a progress bar.

As a result, it is up to the developer devising and implementing a solution. The Progress Indicator pattern offers guidance on how to structure the JavaScript client and the server application so that they can share information about server progress and report that information timely to the user.

The idea behind the pattern is that you design the lengthy task to expose some information about its progress. In other words, the code that implements the task doesn't include only functional steps. Instead, functional steps are intertwined with calls to an API that exposes progress information. The pattern also suggests you employ a second component that from the client monitors any exposed information by essentially polling the server.

The progress API is made of two distinct set of functions — one for the client and one for the server. The server API offers a class through which the task can save its progress. The class gets status information and saved it somewhere — be it a database table, a disk file, or perhaps an in-memory buffer such as the ASP.NET Cache. The server API also must expose an HTTP endpoint for the client monitor to poll to know about the status of the ongoing task.

In ASP.NET, the task can take the following form:

```
void ExecuteTask(int taskID, /* params */ ...)
{
    ProgressMonitor progMonitor =
      new ProgressMonitor();
    progMonitor.SetStatus(taskID, "5%");
    DoFirstStep(...);
    :
    progMonitor.SetStatus(taskID, "100%");
    DoFinalStep(...);
}
```

The *ProgressMonitor* class writes to a known location any information it receives from the task. On the server, you also need a service that can be called from JavaScript. The purpose of this service is reading the status of the task and return that to the client for UI updates. The simplest way to do this is creating a web service that uses the *ProgressMonitor* class to read what a given task has saved.

For the whole machinery to work, the task must be uniquely identified with an ID. The ID must be passed to the task when it is first started. Hence, the task ID must be generated on the client. The JavaScript's *Math.random* function is the tool to use.

Right after starting the remote task, the client activates a monitoring channel which makes periodical calls to previously created Web service. In this way, the user interface knows in real time what's going on in the server. A piece of user interface is then updated to reflect the information imported from the server. The layout of this piece of user interface can be defined at will. It can be a progress bar (that is, an HTML table) or a plain label. It is up to you how you represent the information; the trickiest part is bringing that the right server information down to the client.

— Dino is the author of Programming Microsoft ASP.NET 2.0 (*Microsoft Press, 2005*).

Return to Table of Contents

Of Interest

Intel has announced updates to a number of its compilers, libraries, and cluster tools. In general, the upgrades focus on parallel programming support and optimizations. The tools that are part of the upgrades and the new version numbers include: **Intel Compilers 11.1** (Fortran and C/C++, for Windows, Linux, Mac OS X); **Intel Integrated Performance Primitives** (IPP) 6.1 (for Windows, Linux, Mac OS X); **Intel Math Kernel Library (MKL) 10.2** (for Windows, Linux, Mac OS X); and **Intel Cluster Toolkit, Compiler Edition 3.2.1** (for Windows, Linux). Support is included for the new AVX and AES instructions. AVX, short for "Advanced Vector Extensions," is a 256-bit instruction set extension to SSE and is designed for applications that are floating-point intensive. AVX can improve performance on existing and new applications that lend themselves to largely vectorizable data sets. Wider vector data sets can process up to twice the throughput of 128-bit data sets. The AES (short for "Advanced Encryption Standard") instructions set is the U.S. Government standard for symmetric encryption. It includes four instructions to facilitate high-performance AES encryption and decryption, and two instructions support the AES key expansion procedure. **http://www.intel.com**

ComponentOne has released **XapOptimizer**, a standalone utility that reduces the size of XAP files by removing unused classes and XAML resources. This reduces the size of Silverlight RIA applications, thereby improving download time and consumption of network resources, as well as securing the code. According to ComponentOne, the size of a Silverlight application can be reduced by up to 70% without any loss in functionality. **http://www.componentone.com**

Wind River has announced the availability of its Wind River **Hypervisor**, a high-performance Type-1 hypervisor that supports virtualization on single and multicore processors. Wind River Hypervisor provides integration with Wind River's VxWorks and Wind River Linux operating systems and supports other operating systems. The Wind River Workbench development tools suite has been extended to support developing software that runs on the Wind River Hypervisor. **http://www.windriver.com**

Blueprint has announced **Blueprint Requirements Center 2010**, a tool that provides a traceable and integrated platform that empowers requirements authors to define requirements assets including functional and nonfunctional requirements, business processes, use-cases, user interfaces, glossaries, data, and rules. **http://www.blueprintsys.com**

The Portland Group has announced an agreement with NVIDIA under which the two companies plan to develop new **Fortran language support for CUDA GPUs**. The pair released the Fortran language specification for CUDA GPUs at the *International Conference on Supercomputing* in Hamburg, Germany. The CUDA Fortran compiler will be added to a production release of the PGI Fortran compilers scheduled for availability in November 2009. **http://www.pgroup.com** and **http://www.nvidia.com**

Return to Table of Contents

Q&A: Twitter And Clouds

Twitter presents a perfect vector for malicious code and phishing

by Jonathan Erickson

Gary McGraw is CTO of Cigital, a software security and quality consulting firm. He recently spoke with *Dr. Dobb's* editor-in-chief Jonathan Erickson about security in the age of Twitter and cloud computing.

Dr. Dobb's: Does Twitter pose security-related problems?
McGraw: Twitter presents a perfect vector for malicious code and phishing, especially since most users use bit.ly or tinyurl to fit clickable URLs into their messages. Twitter allows dingbats to cash in their last remaining privacy chit with a coolness factor that often overrides common sense.

In fact, the last point applies equally well to Facebook and MySpace. The big problem is many users of these systems seem to have little understanding that postings, tweets, tequila drinking photos, and everything they post in the Web 2.0 world is public. Before Tweeting whatever occurs to you, think about whether you would want your mom to read it. Also note that the Tweet will be around basically forever! Will your future potential employers search Twitter? Why wouldn't they?

Dr. Dobb's: And virtualization?
McGraw: Some easy questions turn out to open various cans of worms. How can I tell if I am running on a VM? Can I figure out what chip I'm actually on? These questions get particularly hairy when it comes to mobile computing. There is an important class of problems in security called "interposition" attacks. Virtualization opens up all new places to get these classic old dinosaur attacks all gussied up for the future.

Dr. Dobb's: Does security have a role in cloud computing?
McGraw: There are many different types of clouds — public cloud computing is a world away from private cloud computing. Who owns what cycles and what runs where? Equally important for security are infrastructure as service clouds versus software apps as service clouds. Most effort seems to be based around securing data, both in transit and at rest. The different cloud models imply different application architectures, and different architectures (as we all know) imply different security solutions.

Dr. Dobb's: Distributed systems are the norm these days. Has security kept pace with technology implementation in this regard?
McGraw: There are some real challenges with securing massively distributed systems. If you want a good example of what we can expect when a majority of apps are distributed, just take a look at MMORPGs (or "massively multiplayer online role playing games"). Greg Hoglund and I wrote a book called *Exploiting Online Games* that is really a case study for the future of software security.

Probably the most important issue developers and architects need to understand when it comes to distributed systems is the notion of trust boundaries. As an example, it is a really bad idea to include code running on a user's PC or phone or whatever (that is, client code) on the "trusted" side of the trust boundary. Instead, think about that code being completely and utterly exposed, rewired, hacked, etc. In *Exploiting Online Games*, we do plenty of work disassembling the client code for *World of Warcraft* with amusing but scary security results.

Don't disregard trust boundaries.

Return to Table of Contents

Land the Tech Job You Love

Reviewed by Mike Riley

Land the Tech Job You Love
by Andy Lester
The Pragmatic Bookshelf
280 pages; $23.95

It goes without saying that times are tough and finding the perfect job is even tougher. Chicago-based technologist and author Andy Lester offers no-nonsense job search and interview advice in his recent Pragmatic Bookshelf title, *Land the Tech Job You Love*. Read on to find out if the tips he shares are worth the book's cover price.

Unlike the pile of job search and interview tip books on the market, *Land the Tech Job You Love* is a book exclusively optimized for devoted IT professionals looking to improve their chances of landing the ideal job (or, at the very least, "a" job) in the highly competitive and job-constrained technology sector. Keenly written especially for the highly intelligent yet often introverted technical crowd, Andy Lester has distilled a number of excellent reminders for readers seeking out what jobs they desire and what they qualify for, as well as customizing their resume for the targeted employer, ace'ing the interview and negotiating the job offer.

Having been on both sides of the hiring fence, I can attest to many of Andy's recommendations for interviewees. I can also relate to some of the horror stories he relates in the book, since I have interviewed some of the very character types Andy uses as worst-case scenario examples.

The book itself can be read in an hour or two, but putting into practice all the suggestions will take days or even weeks to refine. Like most Pragmatic Bookshelf titles, this book takes a very pragmatic approach to the resume and interview process. Strongly opinionated dos and don'ts are punctuated with real world examples and continued emphasis on what it will take to honestly impress hiring managers and interviewers. I appreciated Andy's honest, take-it-or-leave-it approach to getting serious about the job search, coupled with the real-world realities of the current economic and job availability climate. The book is devoid of rah-rah feel good pep talks. Instead, the author tackles the reality of the current tech job market head-on, dealing with tough interview questions, avoiding clichés and generally telling it like it is — very refreshing.

While most tech managers and executives will probably not discover any new insights shared in the book, *Land the Tech Job You Love* supplies an excellent distillation of all the 'best of' job search tips amassed over a successful IT professional's career. For those seeking a change in their current tech employment situation, this book is definitely worth its asking cover price. Given the author's welcome knack for hard truths combined with real-world examples, I'd like to see him follow up this book with a sequel, possibly titled *Keep the Tech Job You* Love to carry forward the successful practices of people who make themselves invaluable participants to their organizations.

Return to Table of Contents

The Power of "In Progress"

Hold no hostages: Know when to design for "partially updated" as the normal state

By Herb Sutter

Don't let a long-running operation take hostages. When some work that takes a long time to complete holds exclusive access to one or more popular shared resources, such as a thread or a mutex that controls access to some popular shared data, it can block other work that needs the same resource. Forcing that other work to wait its turn hurts both throughput and responsiveness.

To solve this problem, a key general strategy is to "design out" such long-running operations by splitting them up in to shorter operations that can be run a piece at a time. Last month [1], we considered the special case when the hostage is a thread, and we want to prevent one long operation from commandeering the thread (and its associated data) for a long time all at once. Two techniques that accomplish the strategy of splitting up the work are continuations and reentrancy; both let us perform the long work a chunk at a time, and in between those pieces our thread can interleave other waiting work to stay responsive.

Unfortunately, there's a downside: We also saw that both techniques require some extra care because the interleaved work can have side effects on the state the long operation is using, and we needed to make sure any interleaved work wouldn't cause trouble for the next chunk of the longer operation already in progress. That can be hard to remember, and sometimes downright complicated and messy.

Is there another, more general way?

Let It "Partly" Be: Embrace Incremental Change

Let's look at the question from another angle, suggested by my colleague Te rry Crowley: Instead of viewing partially-updated state with in-progress work as an 'unfortunate' special case to remember and recover from, what if we simply embrace it as the normal case?

The idea is to treat the state of the shared data as stable, long-lived and valid even while there is some work pending. That is, the "work pending" that results from splitting long operations into smaller pieces isn't tacked on later via a black-box continuation object or encoded in a stack frame on a reentrant call; rather, it's promoted to a full-fledged, first-class, designed-in-up-front part of the data structure itself.

Looking at the problem this way has several benefits. Perhaps the most important one is correctness: We're making it clear that each "chunk" of processing is starting fresh and not relying on system state being unchanged since a previous call. In [1], we had to remember not to make that implicit assumption when we resumed a continuation or returned from a yield; this way, were are explicit about the "data plus work pending" state as the normal and expected state of the system.

This approach also enables several performance and concurrency benefits, including that we have a range of options for when and how to do the pending work. We'll look at those in more detail once we've seen a few examples, but for now note that they include that we can choose to do the pending work:

- with finer granularity, so that we hold exclusion (e.g., locks) for shorter times as we do smaller chunks of the work;
- asynchronously, so that it can be more easily performed by another helper thread; and/or
- lazily, instead of all up front.

Note the "and/or"—one or more of these may apply in any given situation. Some of these techniques, notably enabling lazy evaluation, are well-known optimizations for ordinary performance, but they also directly help with concurrency and responsiveness.

The rest of this article will consider two mostly orthogonal issues: First, we'll consider different ways to represent the pending work in the data structure, with real-world examples. Then, we'll separately consider what major options we have for actually executing the work, how to use them singly or in combination, and what their tradeoffs are and where each one applies.

Option 1: Data + Pending Changes

Perhaps the simplest option is to represent the data along with a (possibly empty) queue of pending updates or other work to be performed against it, as illustrated in Figure 1. This configuration is the most similar to the continuation style in [1]. However, unlike that style where we explicitly created a continuation object and appended it to a queue that was logically separate from the data, here the pending work queue is an intentional first-class part of the data structure integrated into its design and operation.

One potential drawback to Option 1 is that it can be less flexible if we may need to interrupt and restart an operation we've split into pieces and enqueued. For example, if recalculations are affected by further user input, and multiple chunks of the recalculation work are already in the queue, we may need to traverse the queue to remove specific work items. One way to minimize this concern is to only enqueue one "next step" at a time for each long-running operation, and have the end of each chunk of work enqueue its own continuation (see sample code in [1]).

Option 2: Data + Cookies

Figure 2a shows a different way to encode pending work: Attach reminders (or "cookies") that go with the data, each of which remembers the current state of a given operation in progress.

Consider the example of a word processing application, as suggested in Figure 2a: Pagination, or formatting the content of a document as a series of pages for display, is one of dozens of operations that can take far longer than would be acceptable to perform synchronously; there's no way we want to make the user wait that long while typing. In Microsoft Word, the internal data structure that represents a document makes it well-defined to have a document that is only partially paginated, specifically so Word can break the pagination work up into pieces. By executing the operation a piece at a time, it can stay responsive to any pending user messages and provide intermediate feedback before continuing on to the next chunk.

At any given time, we only need to immediately and synchronously perform pagination up to the currently displayed page, and then only if the program is in a mode that displays page breaks. Later pages in the document can be paginated lazily during idle time, on a background thread, or using some other strategy (see "When and How To Do the Work" later in this article). However, if the user jumps ahead to a later page, pagination must be completed to at least that page, and any that is not yet performed must be done immediately.

This approach can be more flexible than Option 1 in the case when we may need to interrupt and restart the operation, such as if pagination is impacted by some further user input such as inserting a paragraph. Instead of walking a work queue, we update the (single) state of pagination for this document; in this case, it's probably sufficient to just resume pagination from a specific earlier point where the new paragraph was inserted.

Figure 2b shows another example of how this technique can be used in a typical modern word processor, in this case OpenOffice Writer. Like many word processors, Writer offers a synchronous "spell check my document now" function that pops up a dialog and lets you step through all the potential spelling mistakes in your document. But it also offers something even more helpful: "red squigglies," or immediate visual feedback as you type to highlight words that may be misspelled. In Figure 2b, the document contains what looks more like C++ code than English words, and so a number of words get the red-squiggly I-can't-find-that-in-my-dictionary

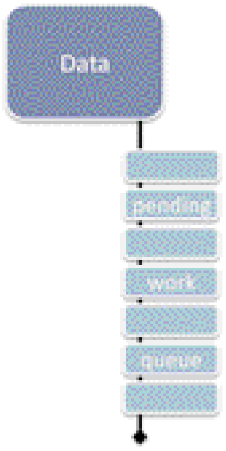


Figure 1: Deferring work via "data + pending changes."

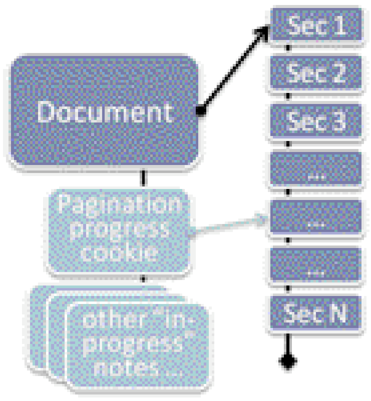
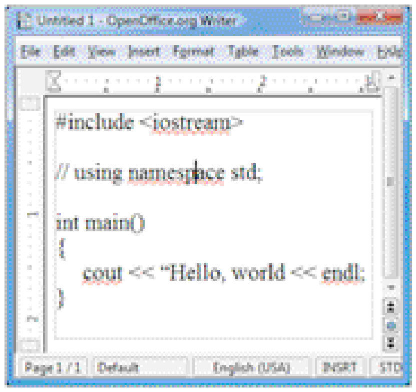


Figure 2a: Deferring work via "data + cookies."


Untitled 1 - OpenOffice.org Writer
File Edit View Insert Format Table Tools Window Help
Page 1 / 1
Default
English (USA)
INSRT
STD

```
#include <iostream>

// using namespace std;

int main()
{
    cout << "Hello, world << endl;
}
```

Figure 2b: Red squigglies in a word processor—without "spell check document" (courtesy OpenOffice.org Writer).

treatment. (Aside: Yes, the typo is intentional. See Figure 3b.)

Consider: Would you add red squigglies synchronously as the user types, or would you do it asynchronously? Well, if all the user has done is type a new word, it may be okay to do it synchronously for that word because each dictionary lookup is probably fast. But what about when the user pastes several paragraphs at once? Or loads a large document for the first time? With a lot of work left to do, we may well want to do the spell checking in the background and let the red squigglies appear asynchronously. We can optimize this technique further in a number of ways, such as giving priority to checking first the visible part of the document, but in general we can get better overall responsiveness by doing all spell checking in the background by default to build up a list of spelling errors in the document, so that the information is already available and the user doesn't have to wait as he navigates around the document or enters the dedicated spell-check mode.

Option 3: Data + Lazy Evaluation

Lazy evaluation is a traditional optimization technique that happens to also help concurrency and responsiveness. The basic idea is simple: We want to optimize performance by not doing work until that work is actually needed. Traditional applications include demand-loading persistent objects from a slow database store and creating expensive Singleton objects on demand. In Excel, for example, it is well-defined to have worksheets with pending recalculations to perform, and each cell may be "completely evaluated" or "pending evaluation" at any given time, so that we can immediately evaluate those that are visible on-screen and lazily evaluate those that are off-screen or hidden.

Figure 3a illustrates how we can use lazy evaluation as a natural way to encode pending work. A compiler typically stores the abstract representation of your program code as a tree. To fully compile your code to generate a standalone executable, clearly the compiler has to process the entire tree so that it can generate all the required code. But do we always need to process the whole tree immediately?

One common example where we do not is compilation in managed environments like Java and .NET, where it's typical to use a just-in-time (JIT) compiler that compiles classes and methods on demand as they get invoked. In Figure 3a, for example, we may have called and compiled *Class2::MethodA()*, but if we haven't yet called *Class2::MethodB* or anything in *Class1* or *Class3*, those entities can be compiled later on demand (or asynchronously in the background during idle time or some other strategy; again, see "When and How To Do the Work").

But lazy compilation is useful for much more than just JIT. Now consider Figure 3b: Let's say we want to dynamically provide red-squiggly feedback, not on English misspellings, but rather on code warnings and errors. Just as we wanted dynamic spell-checking feedback without going into a special spell-check-everything-now mode (see Figure 2b), we might want dynamic compilation warnings and errors without going into a separate compile-everything-now mode.

In Figure 3b, the IDE is helpfully informing us that the code has several problems, and even provides the helpful message "missing closing quote" in a tooltip balloon as the mouse hovers over the second error—all dynamically as we write the code, before we try to explicitly recompile anything. Clearly, we have to compile something in order to diagnose those errors, but we don't have to compile the whole program. As with the word processing squigglies, we can prioritize the visible part of the code, and lazily compile just enough of the context to make sense of the code the user sees; in this example, we can avoiding compiling pretty much the entire *<iostream>* header because nothing in it is relevant to diagnosing these particular errors.

When and How To Do the Work

This brings us to the key question: So, when and how is the pending work performed?

One option is to do the pending work interleaved with other work, such as in response to timer messages or using explicit continuations as in [1]. This approach is especially useful when the updates must be performed by a single thread, either for historical reasons (e.g., on systems that require a single GUI thread) or to avoid complex locking and synchronization of internal data structures by making the data isolated to a particular thread.

A second approach is to do the work when idle and there is no other work to do.

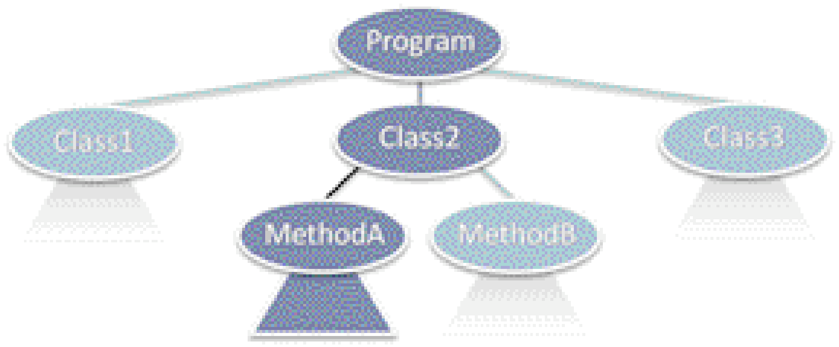
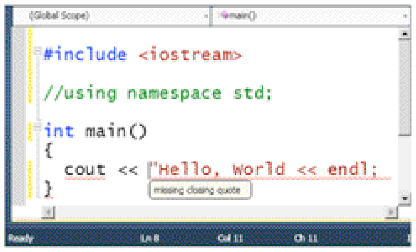


Figure 3a: Deferring work via traditional lazy evaluation.



```
#include <iostream>

//using namespace std;

int main()
{
   cout << "Hello, World << endl;
}
```

Figure 3b: Red squigglies in a C++ IDE — without "rebuild" (courtesy Visual Studio 2010).

For example, Word normally performs pagination and similar operations in incremental chunks at idle time. This approach is usually used in combination with a fallback to one of the other approaches if we discover that some part of the work needs to happen more immediately, for example if the user jumps to not-yet-paginated part of the document.

The internal data structure makes it well-defined to have a document that is only partially paginated, specifically so Word can break the pagination work up into pieces

There are multiple ways to implement "when idle":

- If all of the updates must be performed by a single thread, we can register callbacks that the system will invoke when idle (e.g., using a Windows *WM_IDLE* message); this is a traditional approach for GUI applications that have legacy requirements to do their work on the GUI thread.
- If the updates can be performed by a different thread, we can perform the work on one or more low-priority background threads, each of which pauses every time it completes a chunk of work. To minimize the potential for priority inversion, we want to avoid being in the middle of processing an item (and holding a lock on the shared data) when the background thread is preempted by the operating system, so each chunk of work should fit into an OS scheduling quantum and the pause between work items should include yielding the remainder of the quantum (e.g., using *Sleep(0)* on Windows).

Third, we can do the work asynchronously and concurrently with other work, such as on one or more normal background worker threads, each of which locks the structure long enough to perform a single piece of pending work and then pauses to let other threads make progress. For example, in Excel 2007 and later, cell recalculation uses a lock-free algorithm that executes in parallel in the background while the user is interacting with the spreadsheet; it may run on several worker threads whose number is scaled to match the amount of hardware parallelism available on the user's machine.

Fourth, in some cases it can be appropriate to do the work lazily on use, where each use of the data structure also performs some pending work to contribute to overall progress; or similarly we may do it on demand specifically in the case of traditional lazy evaluation. With these approaches, note that if the data structure is unused for a time then no progress will be made; that might be desirable, or it might not. Also, if the accesses can come from different threads, it must be safe and appropriate to run different pieces of pending work on whatever threads happen to access the data.

Summary

Embrace change: For high-contention data that may be the target of long-running operations, consider designing for "partially updated" as a normal case by making pending work a first-class part of the shared data structure. Doing so enables greater concurrency and better responsiveness. It lets us shorten the length of time we need to hold exclusion on a given piece of shared data at any time, while still allowing for operations that take a long time to complete—but can now run to completion without taking the data hostage the whole time.

We can express the pending work in a number of ways, including as a queue of work, as cookies representing the state of operations still in progress, or using lazy evaluation for its concurrency and responsiveness benefits as well as for its traditional optimization value. Then we can execute the work using one or more strategies that make sense; common ones include executing it interleaved with other work, during idle processing, asynchronously on one or more other threads, on use, or on demand.

It's true that we'll typically incur extra overhead to store and "rediscover" how to resume the longer operation at the appropriate point, but the benefits to overall system maintainability and understandability will often far outweigh the cost. Especially when the interleaved work may need to be canceled or restarted in response to other actions, as in the pagination and recalculation examples, it's easier to write the code correctly when the work still in progress is a well-defined part of the overall state of the system.

Acknowledgments

Thanks to Terry Crowley, director of development for Microsoft Office, for providing the motivation to write about this topic and several of the points and examples. Other examples were drawn from the development of Visual C++ 2010.

Notes

[1] H. Sutter. "Break Up and Interleave Work to Keep Threads Responsive" (*Dr. Dobb's Digest*, June 2009). Available online at www.ddj.com/go-parallel/article/showArticle.jhtml?articleID=217801299.

—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