United Business Media

Dr. Dobb's DIGEST

The Art and Business of Software Development July 2010

Shareware: Thanks for the Memories

By Jonathan Erickson,
Editor In Chief

Shareware. Now there's a word you don't hear much these days. In a nutshell, shareware is proprietary software provided to users without payment, usually on a trial basis. By "trial basis" I mean that either the software is crippled by missing functionality or by a limited number of uses or by a set period of time. By whatever mechanism, shareware is built on the try-before-you-buy model.

Shareware became wildly popular in the late 1970s and on into the early '90s, spurred along by the legendary (at least, in the world of shareware) Bob Wallace with his PC-Write wordprocessor, Jim Button with his PC-File database, and Andrew Fluegelman with his PC-Talk communication program.

But then the Internet and Web came along, challenging a lot of things — including the shareware model — in the process. Still, shareware hung in there, thanks in part to the Association of Shareware Professionals, an organization founded in 1987 provide independent software developer support and guidance in marketing software.

"Back then, the Association of Shareware Professionals was a small band of independent underdogs who defied the traditional software distribution channel, and created the try-before-you-buy model," says Mike Dulin, the organization's current president. Amazingly, shareware developers persisted, enabling the organization to evolve into an international group of 1,000 software developers who are pioneering cloud computing, software as a service (SaaS), smartphone development, and desktop/laptop development on all popular computing platforms.

As time passed, however, even the die-hards acknowledged that the "shareware" moniker was becoming a bit of an albatross, so the organization began adopting the short version of "Association of Shareware Professionals" — "ASP" in other words.

But "shareware" has proven hard to shake, so the organization has stopped kidding around and officially changed its name to the Association of Software Professionals (http://www.asp-software.org).

"With fewer and fewer developers calling their applications 'shareware,' software buyers have become confused about the meaning of the term," explains Rich Holler, ASP's Executive Director. "And since nearly all software developers today, from the smallest micro independent software vendor (mISV) to the largest software publisher, offer trial versions of their programs, the term 'shareware' has become less useful."

This makes sense to me. Shareware is a marketing method. ASP's members develop and market software. And the ASP will be called the Association of Software Professionals as it moves into the future.

Nevertheless, it is the passing of another era, which always gives pause. Now remind me: Is there shareware for the iPad?

Jonathan Erickson

Return to Table of Contents

An Internet 100x as Fast

A new network design that avoids the need to convert optical signals into electrical ones could boost capacity while reducing power consumption

By Larry Hardesty *MIT News Office*

The heart of the Internet is a network of high-capacity optical fibers that spans continents. But while optical signals transmit information much more efficiently than electrical signals, they're harder to control. The routers that direct traffic on the Internet typically convert optical signals to electrical ones for processing, then convert them back for transmission, a process that consumes time and energy.

In recent years, however, a group of MIT researchers led by professor Vincent Chan (http://web.mit.edu/chan/www/) has demonstrated a new way of organizing optical networks that, in most cases, would eliminate this inefficient conversion process. As a result, it could make the Internet 100 or even 1,000 times faster while actually reducing the amount of energy it consumes.

One of the reasons that optical data transmission is so efficient is that different wavelengths of light loaded with different information can travel over the same fiber. But problems arise when optical signals coming from different directions reach a router at the same time. Converting them to electrical signals allows the router to store them in memory until it can get to them. The wait may be a matter of milliseconds, but there's no cost-effective way to hold an optical signal still for even that short a time.

Chan's approach, called flow switching, solves this problem in a different way. Between locations that exchange large volumes of data — say, Los Angeles and New York City — flow switching (http://www.mit.edu/~medard/papersnew/WOBS%20Final.pdf) would establish a dedicated path across the network. For certain wavelengths of light, routers along that path would accept signals coming in from only one direction and send them off in only one direction. Since there's no possibility of signals arriving from multiple directions, there's never a need to store them in memory.

Reaction Time

To some extent, something like this already happens in today's Internet. A large Web company like Facebook or Google, for instance, might maintain huge banks of web servers at a few different locations in the United States. The servers might exchange so much data that the company will simply lease a particular wavelength of light from one of the telecommunications companies that maintains the country's fiber-optic networks. Across a designated pathway, no other Internet traffic can use that wavelength.

In this case, however, the allotment of bandwidth between the two endpoints is fixed. If for some reason the company's servers aren't exchanging much data, the bandwidth of the dedicated wavelength is being wasted. If the servers are exchanging a lot of data, they might exceed the capacity of the link.

In a flow-switching network, the allotment of bandwidth would change constantly. As traffic between New York and Los Angeles increased, new, dedicated wavelengths would be recruited to handle it; as the traffic tailed off, the wavelengths would be relinquished. Chan and his colleagues

Dr.Dobb's

EDITOR-IN-CHIEF
Jonathan Erickson

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

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

AUDIENCE DEVELOPMENT
CIRCULATION DIRECTOR
Karen McAleer
MANAGER
John Slesinski

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

UBM LLC

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

TechWeb

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

techweb™

have developed network management protocols that can perform these reallocations in a matter of seconds.

In a series of papers published over a span of 20 years, they've also performed mathematical analyses of flow-switched networks' capacity and reported the results of extensive computer simulations. They've even tried out their ideas on a small experimental optical network that runs along the Eastern Seaboard.

Their conclusion is that flow switching can easily increase the data rates of optical networks 100-fold and possibly 1,000-fold, with further improvements of the network management scheme. Their recent work has focused on the power savings that flow switching offers: In most applications of information technology, power can be traded for speed and vice versa, but the researchers are trying to quantify that relationship. Among other things, they've shown that even with a 100-fold increase in data rates, flow switching could still reduce the Internet's power consumption.

Growing Appetite

Ori Gerstel, a principal engineer at Cisco Systems, says that several other techniques for increasing the data rate of optical networks, with names like burst switching and optical packet switching, have been proposed, but that flow switching is "much more practical." The chief obstacle to its adoption, he says, isn't technical but economic. Implementing Chan's scheme would mean replacing existing Internet routers with new ones that don't have to convert optical signals to electrical signals. But, Gerstel says, it's not clear that there's currently enough demand for a faster Internet to warrant that expense. "Flow switching works fairly well for fairly large demand — if you have users who need a lot of bandwidth and want low delay through the network," Gerstel says. "But most customers are not in that niche today."

But Chan points to the explosion of the popularity of both Internet video and high-definition television in recent years. If those two trends converge — if people begin hungering for high-definition video feeds directly to their computers — flow switching may make financial sense. Chan points at the 30-inch computer monitor atop his desk in MIT's Research Lab of Electronics (http://www.rle.mit.edu/). "High resolution at 120 frames per second," he says: "That's a lot of data."

Return to Table of Contents

Get Software Quality Right

We've known the basics for years but need to apply them

By Capers Jones

Software quality suffers as application size increases. We've known that since the 1970s but still haven't solved the problem. The National Institute of Standards and Technology found in a 2002 study that more than 60% of manufacturing companies reported major defects in software they bought, with just under 80% reporting minor defects.

Part of the problem involves how and what you measure. A common approach is the lines-of-code metric (referred to as KLOC, for thousands of lines of code), but it ignores important stages of the software development lifecycle such as requirements and design. IBM dealt with this shortcoming back in the '70s by developing two metrics to gauge software quality: defect potentials and defect removal efficiency. Both are still highly relevant today and have had the greatest impact on software quality, costs, and schedules of any measures.

Defect potentials are the probable numbers of defects that will be found in various stages of development, including requirements, design, coding, documentation, and "bad fixes" (new bugs introduced when repairing older ones). Defect removal efficiency is the percentage of the defect potentials that will be removed before an application is delivered to users.

Defect potentials can be measured with function points, units of measurement that express the amount of business functionality an information system provides to users. Function points don't measure the number of lines of code because most serious defects aren't found in the code but instead occur in the application's requirements and design (see the sidebar "A Better Gauge of Quality").

The range of defect potentials typically scales from just less than two per function point to about 10. Defect potential correlates to application size: As size increases, defect potential rises. It also varies with the type of software, CMMI levels, development methodology, and other factors.

Comparing the quality from different software methodologies is complicated. However, if the applications being compared are of similar size and if they use the same programming languages, then it's possible to compare quality, productivity, schedules, and other areas. Table 1 compares the defect potentials and removal efficiencies of several software development methodologies.

The examples shown here are based on a combination of the C and C++ programming languages. Although the actual sizes varied, the sizes were converted mathematically to exactly 1,000 function points or 75,000 logical code statements.

METHOD	DEFECT POTENTIALS	DEFECT REMOVAL	EFFICIENCY DELIVERED DEFECTS
CMMI level 1	6,000	78.69%	1,079
CMMI level 3	4,500	91.18%	302
Agile	4,800	92.30%	276
Extreme Programming	4,500	93.36%	209
Rational Unified Process	3,900	95.07%	145
CMMI level 5	3,000	95.95%	83
Team Software Process	2,700	96.79%	67

Table 1: Defect Potentials vs. Removal Efficiency.

The "defect potentials" are the total numbers of defects found in five sources:

1. Requirements
2. Design
3. Source code
4. User documents
5. Bad fixes or bugs in defect repairs

The "high severity" defects are those ranked as #1 and #2 using the classic IBM Defect Severity Scale:

- Severity 1 indicates total stoppage of the application; software does not operate at all.
- Severity 2 indicates that major features cannot be used, are disabled or incorrect.
- Severity 3 indicates minor features are disabled or incorrect.
- Severity 4 indicates a cosmetic error that does not affect operation in significant ways.

.

The "defect removal efficiency" was the total number of defects found and eliminated before the software applications reached clients. After 90 days of usage, client-reported defects were added to internal defects in order to calculate the percentage of defects eliminated prior to release. (There will of course be more defects found after 90 days, but a standard time interval is necessary to have consistent calculations.)

Compared to the poor results associated with CMMI level 1, all of the other methods listed in Table 1 demonstrate significant improvements. With larger samples there would be somewhat different results, but the basic concept of measuring defect potentials and defect removal efficiency levels would remain the same.

Based on studies of about 13,000 applications, the approximate average for defect removal efficiency in the U.S. is only about 85%. Therefore all of the methods that top 85% can be viewed as being better than average.

The upper limit of measured defect removal efficiency is only about 99%. Projects that top 95% tend to use sophisticated combinations of inspections, static analysis, and multiple test stages. Projects that achieve only 85% or less in defect removal efficiency typically use only testing, and are also low in test coverage.

If you have a scientific calculator handy, take the size of the application in function points and raise it to the 1.25 power. The result is the approximate number of defects that will occur. Try this for 10, 100, 1000, and 10,000 function points, and you can see that as code gets bigger, you get dramatically more functional defects. (While you have your calculator out, raise the size to the 1.2 power to see how many test cases you'll need. If you raise size to the 0.4 power, you get the number of months it will take. Divide size by 150, and you get how many people are needed on the development team.)

Function Points: A Better Gauge Of Quality

Function points express the amount of business functionality an information system provides to users. The advantage of this approach over the defects per thousands-of-lines-of-code metric, or KLOC, is that it can measure defects in requirements and design as well as in code. More important, KLOC penalizes modern high-level programming languages when measuring quality and productivity because they can use fewer lines of code to achieve the same results as older languages.

To illustrate, assume an application written in Java requires 1,000 Java code statements and has 10 coding bugs, equal to 10 bugs per KLOC. If the same app written in C requires 3,000 C statements and contains 30 coding bugs, it also has 10 bugs per KLOC. The Java version has better quality since it delivers the app with only a third as many bugs as the C version. But when measured using lines of code, the Java and C versions appear to be identical in terms of quality at 10 per KLOC.

When these apps are measured with function points,we assume that they're both 10 function points in size. The Java version contains one coding bug per function point, while the C version has three. In other words, function points match standard economic definitions for measuring productivity, and they measure quality without omitting requirements and design bugs, and without penalizing modern high-level languages.

Defect Removal Efficiency In Depth

The U.S. average for defect removal efficiency is only 85%, based on my research on about 13,000 development projects. The primary cause for projects running late or having cost overruns is excessive numbers of defects that aren't discovered or removed until testing starts. Such projects appear to be on schedule and within budget — until testing begins. Then, hundreds or even thousands of latent defects are discovered causing delays and cost overruns, and the test schedule ends up far exceeding the original plans.

Most software testing averages about 35% to 50% in defect removal efficiency levels. As application size increases, test coverage and test removal efficiency drop. This suggests that additional quality control methods such as inspections are needed.

Formal inspections of requirements, design, and source code have been in use since IBM began looking for better quality control methods in the '70s. With inspections, defect removal efficiency levels spike higher than 85% and testing defect removal efficiency goes up by about 5% per test stage. More recently, static analysis tools used prior to testing were found to contribute to high levels of defect removal efficiency (in the 85% range), although not against dynamic problems such as performance. (See Table 2 for defect removal efficiency levels for various stages.)

While defect potentials and defect removal efficiency are the most effective ways of evaluating software quality controls, actually improving software quality requires two process improvements: defect prevention and defect removal. Defect prevention refers to technologies and methodologies that lower defect potentials or reduce the numbers of bugs that must be eliminated. Examples of defect prevention methods include joint application design, quality function deployment, Six Sigma, structured design, and participation in formal inspections.

For its part, defect removal refers to methods that can either raise the efficiency levels of specific forms of testing or raise the overall cumulative removal efficiency by adding other reviews or tests. The two approaches can be implemented at the same time.

To achieve a cumulative defect removal efficiency of 95%, it's necessary to apply at least nine defect removal activities in sequence:

1. Design inspections
2. Code inspections
3. Automated static analysis
4. Unit test
5. New function test
6. Regression test
7. Performance test
8. System test
9. External beta test

Requirements inspections, test case inspections, and specialized forms of testing (such as human factors, performance, and security testing) add to defect removal efficiency levels.

Who's Using These Metrics?

Since defect potentials and defect removal efficiency metrics are among the easiest to use and most effective techniques for improving software quality, you have to wonder why everyone isn't using them. My research has shown that companies most likely to use these metrics are ones that make computers and other complex hardware devices, such as telecommunication, aerospace, embedded equipment, and defense contracting companies. Many of them are topping 95% in defect removal efficiency, compared with an industry average of 82%.

Dovel Technologies, a software developer and system integrator that builds IT systems for government and private industry, reported in 2009 a 96% defect removal efficiency, which it credits to the adoption of defect potentials and defect removal efficiency metrics, along with close monitoring throughout the development lifecycle using formal and informal reviews, among other approaches.

ACTIVITY	RANGE
Formal requirement inspections	50% to 90%
Formal design inspections	45% to 85%
Formal code inspections	45% to 85%
Automated static analysis	55% to 90%
Automated unit test	20% to 60%
Regression test	15% to 30%
Integration test	25% to 40%
Performance test	20% to 40%
System test	25% to 55%

Table 2: Software Defect Removal Efficiency Ranges.

Companies that have adopted these metrics have cut their development and maintenance costs as well. When you have to rework defective requirements, design, and code, it can consume as much as 50% of the total cost of software and development.

What It All Means

Combining inspections, static analysis, and testing is cheaper than testing by itself and leads to much better defect removal efficiency levels. In concert, these approaches also shorten development schedules by more than 45% because, when testing starts after inspections, almost 85% of the defects already will have been addressed.

To measure defect potentials, it's necessary to keep good records of all defects found during development. When IBM applied formal inspections to a large database project, delivered defects were reduced by more than 50% compared with previous releases, according to my research, and the schedule was shortened by about 15%. Testing was reduced from three shifts over 60 days to one shift over 40 days. Most importantly, customer satisfaction improved to "good" compared with prior releases in which customers rated "very poor" satisfaction levels.

Cumulative defect removal efficiency was raised from about 80% to just over 95% as a result of using formal design and code inspections, and maintenance costs came down by more than 45% for the first year of deployment. Those are the kind of results that speak for themselves.

— Capers Jones is the founder and former chairman of Software Productivity Research, a software development consultancy.

Return to Table of Contents

C++ and *format_iterator*

Enhancing C++'s flexibility and expressiveness

By Matthew Wilson

One of the subjects covered in my book *Extended STL, Volume 1* (http://synesis.com.au/publishing/xstl/) was a discussion about the lack of flexibility and expressiveness in the C++ standard library's *std::ostream_iterator*, along with a (partial) resolution in the form STLSoft's *stlsoft::ostream_iterator*. Chapter 34 addressed these issues in part and was adapted as a *Dr. Dobb's* article entitled "An Enhanced *ostream_iterator*" (http://www.drdobbs.com/cpp/201200278).

The deficiencies of *std::ostream_iterator* for which *stlsoft::ostream_iterator* provides improvements are:

- It allows for the specification of a prefix to be emitted with each element in the streamed sequence; *std::ostream_iterator* provides only for a suffix.
- It allows the prefix (and the suffix) to be of arbitrary string type; *std::ostream_iterator's* suffix must be a C-style string.

But issues of flexibility and expressiveness remain, for which neither component provides solutions:

- Output must still be to an *IOStreams* output stream (or a structurally conformant class providing insertion operators).
- The type of the element must be explicitly specified, which can be cumbersome when dealing with template specializations and/or deep namespaces.
- The only "formatting" supported is to specify prefix and postfix; anything more sophisticated requires hand-written loops.

In the last few years my research into flexibility has progressed, leading to the application of shims and type-tunneling (http://www.drdobbs.com/cpp/184401689) into areas of C++ programming such as diagnostic logging and string formatting. It is the latter case I want to draw on for this article, specifically the FastFormat C++ formatting library (http://accu.org/index.php/journals/1539).

FastFormat offers two APIs, Format and Write, supporting replacement-based and concatenation-based formatting of arguments of arbitrary type to output "sinks" of arbitrary type, as in:

```
int          i =   13;
CComVariant v(L”abcd”);
std::string salutation = “Hello”;
Person      person(“John”, “Smith”);

// gives: “i=13, v=        abcd        .”
ff::fmtln(std::cout, “i={1}, v={0,20,,^}.”, v, i);

// gives: “Hello, John Smith”
std::string result;
ff::write(result, salutation, “, “, person);
```

The high flexibility afforded by the library got me to thinking once again about formatting output iterators: Could it be possible to implement an iterator in terms of FastFormat that would substantially improve on *stlsoft::ostream_iterator*?

fastformat::format_iterator

Fairly obviously (since I'm writing this article), the answer is "Yes!" The resulting component — *fastformat::format_iterator* — allows algorithmic formatting:

- Of sequences of arbitrary type; all types that are compatible with FastFormat's Format and Write APIs (http://accu.org/index.php/journals/1553) will be automatically compatible with the iterators
- Without having to explicitly write the type in the expression
- To sinks of arbitrary type
- Using formats of arbitrary complexity, including use of other parameters from the expression context.

Let's look at a scenario to illustrate. We'll consider again (a chopped down version of) the example used in *Extended STL, Volume 1*, but this time using the recently released version 1.9 of the recls library (http://synesis.com.au/software/recls/).

Consider that we want to list all the C/C++ header files under the current directory, and place them within XML tags. In each case we will instantiate an instance of the STL Collection sequence type, *recls::search_sequence*, and write to an *std::stringstream* instance, as follows:

```
std::string const prefix("<file>");
char const* const suffix = "</file>";

std::stringstream      stm;
recls::search_sequence headers(".", "*.h|*.hpp|*.hxx",
recls::RECURSIVE);
```

To output this list using *std::ostream_iterator* requires something like the following:

```
// via std::ostream_iterator

if(headers.begin() != headers.end())
{
  stm << prefix;
}
std::copy(
  headers.begin(), headers.end()
, std::ostream_iterator<recls::search_sequence::value_type>(
    stm
  , (suffix + prefix).c_str()));
if(headers.begin() != headers.end())
{
  stm << suffix;
}
```

Running this gives output like:

```
<file>H:\freelibs\b64\1.4\include\b64\b64.h</file><file>H:\freeli
bs\b64\1.4\include\b64\implicit_link.h</file><file>H:\freelibs\b6
4\1.4\include\b64\b64.hpp</file><file>H:\freelibs\b64\1.4\include
\shwild\implicit_link.h</file><file>H:\freelibs\b64\1.4\include\s
hwild\shwild.h</file><file>H:\freelibs\b64\1.4\include\shwild\shw
ild.hpp</file><file>H:\freelibs\b64\1.4\include\xcontract\implici
t_link.h</file><file>H:\freelibs\b64\1.4\include\xcontract\xcontr
act.h</file>. . . // more output
```

Contrast this with the (nearly) equivalent implementation using *stlsoft::ostream_iterator*:

```
// via stlsoft::ostream_iterator

std::copy(
  headers.begin(), headers.end()
, stlsoft::ostream_iterator<recls::search_sequence::value_type>(
    stm
  , prefix
  , suffix));
```

What in the former case required two conditionals and three statements is now encapsulated in a single statement. There's no longer any need to concatenate prefix and postfix in order to obtain a C-style string: *stlsoft::ostream_iterator* is implemented in terms of String Access Shims and so works with arbitrary string types. Furthermore, there's no longer any need to (conditionally) insert a leading prefix or to add a final suffix.

In fact, the version using *std::ostream_iterator* is actually defective. If we look at the end of the output, it's clear how:

```
. . .
<file>H:\freelibs\b64\1.4\src\shwild\shwild_vector.hpp</file><file></file>
```

For any non-empty sequence of files, there's always an extra, empty *<file></file>* pair. It's possible a consumer may be able to ignore empty items, but this is not the way to write good software.

With *stlsoft::ostream_iterator* this does not occur, because prefix and postfix are applied to each element directly (see "An Enhanced *ostream_iterator*" (http://www.drdobbs.com/cpp/201200278).

It's also quite clearly more expressive and flexible. But it still has issues:

- The (ungainly) name of the sequence's value type must be explicitly specified
- The sequence can be written only to an *IOStream*'s output stream (or an instance of any type that has conformance with it)
- The "formatting" can consist only of a prefix and suffix

Let's look at examples that address the first two of these; the third we'll cover later as we look at the evolution of the component. Here's how we might use *fastformat::format_iterator*:

```
std::copy(
  headers.begin(), headers.end()
, fastformat::format_iterator(
    stm
  , prefix + "{0}" + suffix));
```

We do not need to specify the sequence value type: FastFormat's type inference mechanisms handle this for us. And the flexibility does not end there: We can skip the string stream and write directly to a string.

```
std::string result;
std::copy(
  headers.begin(), headers.end()
, fastformat::format_iterator(
    result
  , prefix + "{0}" + suffix));
```

With equal ease, we can output to Windows' *OutputDebugString()* API function, illustrating the capability to write to arbitrary sink types:

```
std::copy(
  headers.begin(), headers.end()
, fastformat::format_iterator(
    fastformat::to_sink(::OutputDebugString)
  , prefix + "{0}" + suffix));
```

In both these last two cases, we're doing that for which we earlier criticized the version using *std::ostream_iterator*: explicitly preparing the format string. We'll see later how we can do better, after looking at the implementation.

Version 1 Implementation: Test-Driven Development

I implemented the first version of *format_iterator* using test-driven development principles, in test-first mode. I confess that I anticipated that I would be able to write an article about its development, and implementing in this way would allow me to explain output iterator implementation by example, relying largely on the code to explain. So, that's what I'll do.

Assume any one of the uses of the iterator shown above. Compiling the code gives lots of errors about *format_iterator* not existing. Listing 1 shows the first iteration, the class skeleton that defines the appropriate member types for an output iterator. As explained in detail in *Extended STL, Volume 1*, an output iterator should have an *iterator_category* of *std::output_iterator_tag*, and all other members (*difference_type*, *pointer*, *reference*, and *value_type*) should be *void*; the type generator *std::iterator* is used to define them.

Listing 1

```
#include <stlsoft/util/std/iterator_helper.hpp>
namespace fastformat
{
  class format_iterator
    : public std::iterator<std::output_iterator_tag, void, void, void, void>
  {
  };

} /* namespace fastformat */
```

Listing 2 shows the next iteration, which changes the class name to *format_output_iterator*; *format_iterator* is now a creator function (à la *std::make_pair*()). This is because the iterator class will need to be a template, and using a creator function means that the compiler will automatically deduce the specializing types, saving us from having to explicitly specify them.

Listing 2

```
template <typename S, typename F>
class format_output_iterator
  : public std::iterator<std::output_iterator_tag, void, void, void, void>
{
public:
  typedef format_output_iterator<S, F>  class_type;
public:
  format_output_iterator(S& sink, F const& format);
};
template <typename S, typename F>
inline format_output_iterator<S, F> format_iterator(S& sink, F const& format)
{
  return format_output_iterator<S, F>(sink, format);
}
```

Next, the compiler will complain that *format_output_iterator<>* does not support the dereference and increment operators. Listing 3 shows the addition of these in the next iteration. Note the use of the Dereference Proxy pattern (http://www.xstl.org/doc-1.9/group__group____pattern____dereference__proxy.html), which proscribes fatuous (and non-portable) syntactic misuse of output iterator instances.

Listing 3

```
template <typename S, typename F>
class format_output_iterator
  : public std::iterator<std::output_iterator_tag, void, void, void, void>
{
public:
  typedef format_output_iterator<S, F>  class_type;
private:
  class deref_proxy;
  friend class deref_proxy;
public:
  format_output_iterator(S& sink, F const& format);
private:
  class deref_proxy
  {
  public:
    deref_proxy(format_output_iterator* it)
      : m_it(it)
    {}
  private:
    format_output_iterator* const m_it;
  };
public:
  deref_proxy operator *()
  {
    return deref_proxy(this);
  }
  class_type& operator ++()
  {
    return *this;
  }
  class_type operator ++(int)
  {
    return *this;
  }
};
```

Now the compiler will complain that the *format_output_iterator<>::deref_proxy* type does not support the assignment operator, leading us to Listing 4.

Listing 4

```
template <typename S, typename F>
class format_output_iterator
  : . . .
{
  . . .
private:
  class deref_proxy
  {
  public:
    deref_proxy(format_output_iterator* it);
  public:
    template <typename A>
    void operator =(A const& value);
  private:
    void operator =(deref_proxy const&);
    . . .
  };
public:
  deref_proxy operator *()
  {
    return deref_proxy(this);
  }
  . . .
};
```

Now we get a linker problem, because the dereference proxy type's assignment operator template is not defined. We implement it in terms of the new *invoke_*() method of the iterator class template (see Listing 5). Hopefully, in this step you can see how the format iterator works: I t's implemented in terms of FastFormat's Format API function *fmt*().

Listing 5

```
template <typename S, typename F>
class format_output_iterator
  : . . .
{
  . . .
private:
  class deref_proxy
  {
  public:
    deref_proxy(format_output_iterator* it);
  public:
    template <typename A>
    void operator =(A const& value)
    {
      m_it->invoke_(value);
    }
  private:
    void operator =(deref_proxy const&);
  private:
    format_output_iterator* const m_it;
  };
  template <typename A>
  void invoke_(A const& value)
  {
      fmt(m_sink, m_format, value);
    }
public:
    deref_proxy operator *()
    {
      return deref_proxy(this);
    }
    . . .
};
```

Naturally, the compiler now complains that it does not know what *m_sink* and *m_format* are. These members are defined in Listing 6.

Listing 6

```
template <typename S, typename F>
class format_output_iterator
  : . . .
{
public:
  typedef format_output_iterator<S, F>     class_type;
  typedef S                                   sink_type;
  typedef F                                   format_type;
private:
  typedef std::basic_string<ff_char_t>     string_type_;
private:
  class deref_proxy;
  friend class deref_proxy;
public:
  format_output_iterator(sink_type& sink, format_type const&
format)
    : m_sink(sink)
    , m_format(::stlsoft::c_str_data(format),
::stlsoft::c_str_len(format))
  {}
private:
  class deref_proxy
  {
    . . .
  };
  template <typename A>
  void invoke_(A const& value);
public:
  deref_proxy operator *();
  class_type& operator ++();
  class_type& operator ++(int);
private: // Fields
  sink_type&           m_sink;
  string_type_ const m_format;
};
```

Believe it or not, this series of steps was carried out in just a couple of hours. I've found this to be one of the effects of a test-driven approach. In suitable cases it can result in very fast development times. (This may be because, being guided by the objective tests, I have much more confidence, as well as making fewer missteps.) Of course, I had the guidance of the relevant chapters of *Extended STL* to help. (Not to mention the obvious advantage of having written it, of course!)

Version 2 Implementation: Multiple Format Parameters

The version shown in Listing 6 is a fully fledged output iterator. But we still have the issue of having manually concatenated the prefix and suffix to the format replacement parameter to form the format string in application code. It would be nice to be able instead to write the format statements along the lines of:

```
std::copy(
  headers.begin(), headers.end()
, fastformat::format_iterator(
    stm
  , "{1}{0}{2}"
  , prefix
  , suffix));
```

The zero'th replacement parameter always represents the current sequence element; the others are 1-based indexes of the additional arguments passed to *format_iterator()*.

Of course, once we can do two additional arguments, why not allow more values from the calling context be included in the logging statement, as in:

```
std::string const prefix("\t");
char const* const suffix = "\n";

void f(std::string const& dir)
{
  recls::search_sequence headers(dir, "*.h|*.hpp|*.hxx",
                                         recls::RECURSIVE);

  std::copy(
    headers.begin(), headers.end()
  , fastformat::format_iterator(
      std::cout
    , "{1}{0} found in {3}{2}"
    , prefix
    , suffix
    , dir));
```

This can give output such as the following:

```
H:\freelibs\b64\1.4\include\b64\b64.h found in
H:\freelibs\b64\1.4
H:\freelibs\b64\1.4\include\b64\implicit_link.h found in
H:\freelibs\b64\1.4
H:\freelibs\b64\1.4\include\b64\b64.hpp found in
H:\freelibs\b64\1.4
. . .
```

To support multiple replacement parameters requires quite a bit more cunning (and a bit of template meta-programming). I'm not going to show the full implementation, as it is available in the FastFormat distribution (via http://www.fastformat.org), but I will illustrate the technique with the relevant sections of the class template, creator function template(s), and worker types, shown in Listing 7.

Listing 7

```
namespace impl
{
  struct no_arg_t
  {};
  template <typename T>
  struct ref_helper
  {
  public:
    static T const& get_ref(T const& t)
    {
      return t;
    }
  };
  template <>
  struct ref_helper<no_arg_t>
  {
    static ff_char_t const* get_ref(no_arg_t const&)
    {
      static ff_char_t s_empty[] = { '\0' };
      return s_empty;
    }
  };
} /* namespace impl */

template< typename S
        , typename F
        , typename A1
        , typename A2
        , typename A3
        , typename A4
```

```
         , typename A5
         , typename A6
         , typename A7
         , typename A8
         >
class format_output_iterator
  : . . .
{
  . . .
private: // Construction
  static impl::no_arg_t const& no_arg_ref_()
  {
    // NOTE: this is a race-condition, but it's entirely benign
    static impl::no_arg_t r;
    return r;
  }
public:
  format_output_iterator(sink_type& sink, format_type const& format)
    : m_sink(sink)
    , m_format(::stlsoft::c_str_data(format),
::stlsoft::c_str_len(format))
    , m_n(0)
    , m_arg1(no_arg_ref_())
    , m_arg2(no_arg_ref_())
    , m_arg3(no_arg_ref_())
    , m_arg4(no_arg_ref_())
    , m_arg5(no_arg_ref_())
    , m_arg6(no_arg_ref_())
    , m_arg7(no_arg_ref_())
    , m_arg8(no_arg_ref_())
  {}
  format_output_iterator(
    sink_type&  sink
  , format_type const&  format
  , unsigned     n
  , A1 const&    arg1
  , A2 const&    arg2 = no_arg_ref_()
  , A3 const&    arg3 = no_arg_ref_()
  , A4 const&    arg4 = no_arg_ref_()
  , A5 const&    arg5 = no_arg_ref_()
  , A6 const&    arg6 = no_arg_ref_()
  , A7 const&    arg7 = no_arg_ref_()
  , A8 const&    arg8 = no_arg_ref_()
  )
    : m_sink(sink)
    , m_format(::stlsoft::c_str_data(format),
::stlsoft::c_str_len(format))
    , m_n(n)
    , m_arg1(arg1)
    , m_arg2(arg2)
    , m_arg3(arg3)
    , m_arg4(arg4)
    , m_arg5(arg5)
    , m_arg6(arg6)
    , m_arg7(arg7)
    , m_arg8(arg8)
  {}
  . . .
  template <typename A>
  void invoke_(A const& value)
  {
    switch(m_n)
    {
      case    0:
        fmt(m_sink, m_format, value);
        break;
      case    1:
        fmt(m_sink, m_format, value
          , impl::ref_helper<A1>::get_ref(m_arg1));
        break;
      case    2:
        fmt(m_sink, m_format, value
          , impl::ref_helper<A1>::get_ref(m_arg1)
          , impl::ref_helper<A2>::get_ref(m_arg2));
        break;
      case    3:
        fmt(m_sink, m_format, value
          , impl::ref_helper<A1>::get_ref(m_arg1)
          , impl::ref_helper<A2>::get_ref(m_arg2)
          , impl::ref_helper<A3>::get_ref(m_arg3));
          break;
        . . . 4 - 7:
      }
    }
  . . .
private: // Fields
  sink_type&         m_sink;
  string_type_ const m_format;
  unsigned const       m_n;
  A1 const&            m_arg1;
  A2 const&            m_arg2;
  A3 const&            m_arg3;
  A4 const&            m_arg4;
  A5 const&            m_arg5;
  A6 const&            m_arg6;
  A7 const&            m_arg7;
  A8 const&            m_arg8;
};
. . .
template<   typename S
        ,   typename F
        ,   typename A1
        ,   typename A2
        ,   typename A3
        ,   typename A4
        >
inline format_output_iterator<S, F, A1, A2, A3, A4, impl::no_arg_t,
impl::no_arg_t, impl::no_arg_t, impl::no_arg_t> format_iterator(
  S&         sink
, F const&  format
, A1 const& arg1
, A2 const& arg2
, A3 const& arg3
, A4 const& arg4
)
{
  return format_output_iterator<S, F, A1, A2, A3, A4,
impl::no_arg_t, impl::no_arg_t, impl::no_arg_t,
impl::no_arg_t>(sink, format, 4u, arg1, arg2, arg3, arg4);
}
. . .
```

The iterator class template provides for up to eight additional arguments. (Need for any more than eight seemed unlikely.) There are eight additional creator function templates, taking 1, 2, 3 … 8 additional arguments. Each creator function tells the iterator instance how many parameters are expected, and this is used to select (at runtime) the requisite *fmt()* statement.

The type *fastformat::iterators::impl::no_arg_t* is used as a placeholder where a parameter, and therefore a type, is not required, and together with the traits type *fastformat::iterators::impl::ref_helper<>* is used to ensure that the iterator's *invoke_()* method compiles: even where the switch cases don't match the *n* parameter, and therefore will never be executed, they are still compiled. Note the manner in which the *ref_helper* specialization creates a thread-safe and linkage-safe empty string, independent of character types; this technique is described in detail in *Imperfect C++* (http://www.synesis.com.au/publishing/imperfect/cpp/).

I could have gone further, and used meta-programming techniques to select the requisite *fmt()* statement at compile-time; I leave that as an exercise for the reader. (If anyone provides an implementation that works with all supported compilers, I'll be happy to consider it for inclusion in the library.)

Copy-Assignment and Concept Checking

Eagle-eyed STL extenders may have spotted one remaining problem with the implementation: It is not copy-assignable. To all practical extents, this does not matter, since it's hard to conceive of any practical use case where one would need to take a copy of what is effectively a stateless output iterator. The non-copyability arises from the use of *const* and reference members (a practice I advocate strongly in *Imperfect C++* as it makes the compiler a proactive aid in enforcing design).

Notwithstanding, the C++ standard requires copy-assignment of output iterators, and some implementations of the standard library incorporate active concept checks, preventing compilation of statements involving standard algorithms (such as the examples shown using *std::copy()*). So, we need to make iterator instances copyable, which is achieved by making all reference members be pointers, and by making all members mutable. Thankfully, all the necessary changes are within the *format_output_iterator* class template implementation (see Listing 8), and the class interface and the creator functions are unchanged.

Listing 8

```
template< typename S
          . . .
        >
class format_output_iterator
  : . . .
{
  . . .
  format_output_iterator(sink_type& sink, format_type const& format)
    : m_sink(&sink)
    , m_format(::stlsoft::c_str_data(format),
::stlsoft::c_str_len(format))
    , m_n(0)
    , m_arg1(&no_arg_ref_())
    , m_arg2(&no_arg_ref_())
    , m_arg3(&no_arg_ref_())
    , m_arg4(&no_arg_ref_())
    , m_arg5(&no_arg_ref_())
    , m_arg6(&no_arg_ref_())
    , m_arg7(&no_arg_ref_())
    , m_arg8(&no_arg_ref_())
  {}
  format_output_iterator(
    sink_type&   sink
  , format_type const&  format
  , unsigned     n
  , A1 const&    arg1
  , A2 const&    arg2 = no_arg_ref_()
  , A3 const&    arg3 = no_arg_ref_()
  , A4 const&    arg4 = no_arg_ref_()
  , A5 const&    arg5 = no_arg_ref_()
  , A6 const&    arg6 = no_arg_ref_()
  , A7 const&    arg7 = no_arg_ref_()
  , A8 const&    arg8 = no_arg_ref_()
  )
    : m_sink(&sink)
    , m_format(::stlsoft::c_str_data(format),
::stlsoft::c_str_len(format))
    , m_n(n)
    , m_arg1(&arg1)
    , m_arg2(&arg2)
    , m_arg3(&arg3)
    , m_arg4(&arg4)
    , m_arg5(&arg5)
    , m_arg6(&arg6)
    , m_arg7(&arg7)
    , m_arg8(&arg8)
  {}
private:
    class_type& operator =(class_type const&);
  . . .
  template <typename A>
  void invoke_(A const& value)
  {
    switch(m_n)
    {
      case    0:
        fmt(*m_sink, m_format, value);
        break;
      case    1:
        fmt(*m_sink, m_format, value
          , impl::ref_helper<A1>::get_ref(*m_arg1));
        break;
      case    2:
        fmt(*m_sink, m_format, value
          , impl::ref_helper<A1>::get_ref(*m_arg1)
          , impl::ref_helper<A2>::get_ref(*m_arg2));
        break;
      case    3:
        fmt(*m_sink, m_format, value
          , impl::ref_helper<A1>::get_ref(*m_arg1)
          , impl::ref_helper<A2>::get_ref(*m_arg2)
          , impl::ref_helper<A3>::get_ref(*m_arg3));
          break;
        . . . 4 - 7:
    }
  }
  . . .
private: // Fields
  sink_type*    m_sink;
  string_type_  m_format;
  unsigned      m_n;
  A1 const*     m_arg1;
  A2 const*     m_arg2;
  A3 const*     m_arg3;
  A4 const*     m_arg4;
  A5 const*     m_arg5;
  A6 const*     m_arg6;
  A7 const*     m_arg7;
  A8 const*     m_arg8;
};
```

Robustness

One final note: Although use of *format_iterator* is, like the FastFormat library as a whole, 100% type-safe, being implemented in terms of a replacement-based API means that its use is always subject to format specification defects (http://accu.org/index.php/journals/1561), to which FastFormat responds by throwing an appropriate exception, as illustrated by the following code.

```
std::copy(
  headers.begin(), headers.end()
, fastformat::format_iterator(
    stm
  , "{1}{0}{2}{5}"
  , prefix
  , suffix)); // throws fastformat::missing_argument_exception
```

In this respect, then, it is less robust than either *std::ostream_iterator* and *stlsoft::ostream_iterator*. You should balance that against its substantially increased expressiveness and flexibility when choosing the tools for your particular requirements.

Note: If you have a good reason to ignore format specification defects, this can be done using the scoping classes — *ignore_missing_arguments_scope* and *ignore_unreferenced_arguments_scope* — supplied with the library.

Acknowledgments

Many thanks to Chris Oldwood and Garth Lancaster for their efforts in helping to increase the readability of the article and to reduce its size. Any failures in either regard are entirely my own.

— *Matthew Wilson is a software development consultant and trainer for Synesis Software who helps clients to build high-performance software that does not break, and an author of articles and books that attempt to do the same. He can be contacted at matthew@synesis.com.au.*

Return to Table of Contents

The Software Package Data Exchange (SPDX) Format

A common software package data exchange format — who needs it?

By Phil Odence

The debate about the increasing role of open source in the software community is over. A large and growing pool of applications is available under open source licenses, but open source code is most pervasive in components embedded in almost any application developed today. Superimpose that on the overall ubiquity of software in products from cars to handsets to power plants and it becomes clear that open source code is being channeled through countless supply chains in almost every industry.

Companies at all points in the supply chain are becoming conscious of the need to treat open source just like any other third-party code. They need to know and document the components in the products and software they are consuming and distributing for a variety of reasons, not the least of which is to make sure they understand their legal obligations. Thus the need for a common approach to sharing information about software packages and their related content has never been greater. Breaking down information silos is still a work in progress. Fortunately a new working group is tackling one of the toughest obstacles to sharing information about software packages — collaborating on discovering and sharing information about software packages and their related content, including licenses.

Do the Right Thing

Software license proliferation — there are nearly 2000 software licenses for software freely available on the Internet — is a major headache for software development organizations that want to speed development with software component reuse as well as for companies redistributing software packages as part of their products. Scope is one problem: From the Free Beer license to the GPL family of licenses to platform-specific licenses such as Apache and Eclipse, the sheer number and variety of licenses makes it difficult for companies to "do the right thing" with respect to the software components in their products and applications.

Each license carries within it the author's definition of how the software can be used and reused. Permissive licenses like BSD and MIT make it easy; software can be redistributed and developers can modify code without the requirement of making changes publicly available. Reciprocal licenses, on the other hand, place varying restrictions on reuse and redistribution. Woe to the developer who snags a bit of code after a simple web search without understanding the ramifications of license restrictions.

License Compliance: A First Step To Doing the Right Thing

While most companies want to do the right thing with regard to license compliance when reusing code components, the lack of a clear set of software package data exchange standards complicates matters. Many approaches to ensuring license compliance exist — from hand-crafted spreadsheets to free software options like FOSSology, to enterprise-class applications such as the Black Duck Suite — yet an overarching standard for software package data exchange has been elusive. Suppliers, if they are cataloging the data at all, have their own formats and conventions. Corporate consumers are increasingly asking for this information, but again, there seem to be as many formats as askers.

That situation is changing, however, thanks to the efforts of the Linux Foundation's FOSSBazaar Software Package Data Exchange (SPDX) Working Group (http://www.spdx.org/). The grass-roots effort includes representatives from more than 20 organizations — software, systems and tool vendors, foundations and systems integrators — all committed to creating a standard for software package data exchange formats.

Really, Though: Who Cares About Software Package Data Exchange Formats?

It's not just software development managers and lawyers who care about having a standardized approach to software license compliance. Any corporation that uses and/or distributes software packages has a stake in the outcome. IT managers care; software development managers who want to know what's in the code their developers are writing care, and executives at companies buying software packages care. And software development organizations care — especially distributed, global development teams who collaborate and need visibility into licenses and their obligations. Some of this interest is being driven by more and more companies demanding that suppliers provide them with a Bill of Materials that states clearly which software components are in a specific package, and which licenses are represented. Simply saying your company is doing the right thing is not enough: Savvy users want proof to limit the risk of non-compliance with licenses.

Because there are so many stakeholders, the SPDX Working Group has formulated a straightforward charter:

Create a set of data exchange standards to enable companies and organizations to share license and component information (metadata) for software packages and related content with the aim of facilitating license and other policy compliance.

The goal — to create a common software package data exchange format to simplify the discovery, collection, and sharing of information about software packages and related content — promises to save time, improve the accuracy of license data collection, and simplify compliance with software licenses.

The Scope of the Problem

Most companies have well-established practices that govern the release and distribution of software. But software reuse has created additional wrinkles. Because most software products developed today are composed of mixed code acquired from many different sources — in many cases, without the knowledge of product and development managers and executives — the software supply chain has become more complex.

Breaking the problem down into its component pieces gives a sense of its scope:

- Prior to distributing a collection of software, the contents of each package to be included need to be reviewed to ensure compliance with all the licenses in the code being redistributed.
- Therefore, the supply chain for products requires developers to create a "software pedigree" that includes information necessary to avoid misuse and mitigate risk.
- A software package's declared license may not always match the licenses of individual files inside the package.
- In fact, a typical software package may consist of thousands of files with different licenses.
- Code reuse may have introduced code fragments and components covered by a range of incompatible licenses.

Therefore, the industry needs a standard way of referring to the legal compliance "bill-of-materials" of a software package. It's necessary to standardize a way to exchange information about the licenses contained in a software package efficiently and accurately.

Adding to the urgency of this problem, software packages with more than one version have complex interdependencies. As software evolves over time, new code components may be included that have different licenses, conceivably at any level of the software. Code reuse is a great way to speed up development, but it can introduce license conflicts over time. After all, with almost 2,000 licenses out there, it's clear that not all licenses will be compatible.

Just the Facts, Please

Although most software licenses convey intent, the SPDX effort is focusing on getting at facts. By describing the solution to the problem as a "defined format of file to accompany any software package," the SPDX effort eases the exchange of license information between companies by looking at three areas: facts that deal with identification, facts that provide overview information, and facts that provide file-specific information about the software package. The SPDX Working Group does not attempt to apply legal judgment, for example, by classifying a license as "BSD-like."

Version 1 of the SPDX standard provides a format for representing facts first identifying the package, then about the package content, and finally about the files composing the package.

Facts that deal with a software package's identification (metadata) included in the SPDX specification are:

- Which version of the SPDX specification is in use
- Unique identifier
  - o The cryptographic hash algorithm representing a unique identifier that correlates with a specific software package

- How the information was generated
  - o The SPDX spec defines a way to specify manual/visual review of code (who, when), or
  - o Tools used (ID, version, when)

- Independent audit
  - o SPDX includes the possibility of a multi-person "signoff/reviewed by"" process

Facts that provide overview information about a software package's content also are included in the SPDX specification, e.g.:

- Formal Name
- Package Name
- Download Location
- Declared License(s)
- Copyrights and Dates

Finally, facts that are specific to a software package's file-specific properties included in the SPDX spec cover standardized fields, e.g.:

- File Name (including subdirectory)
- File Type (source or binary)
- Declared license(s) governing file (from file)
- Copyright owners (if listed)
- Copyright dates (if listed)

Because of the license orientation of the specification, the Working Group is committed to providing standardized license references. It's more complex than one might think to reference exactly the right revision of the right license. The spec includes:

- License names
- Unique identifiers for common open source licenses
- Mechanisms for handling non-standard licenses.

So Where Is SPDX Now?

Clearly, there's a need to create a set of software package data exchange standards that will eliminate ambiguity for software development organizations, systems and tool vendors, and open source projects — one that is supported by best practices, use cases, and prototype tools — that can be used by a broad range of constituents.

The SPDX standard Working Group has set an ambitious goal — to have a defined format for a file of license fact information — in place by Q4 2010. Work is underway via in-person meetings and a project Wiki, and a website is under construction. Testing with use cases and prototype tools is next up, with a group review planned with the Linux Foundation legal working group in July 2010. From there a V1 draft standard should be published — the target is August — with V2 to follow.

Participate!

If you're interested in participating in the SPDX Working Group, send an email to one of the chairpersons below, or check out http://spdx.org.

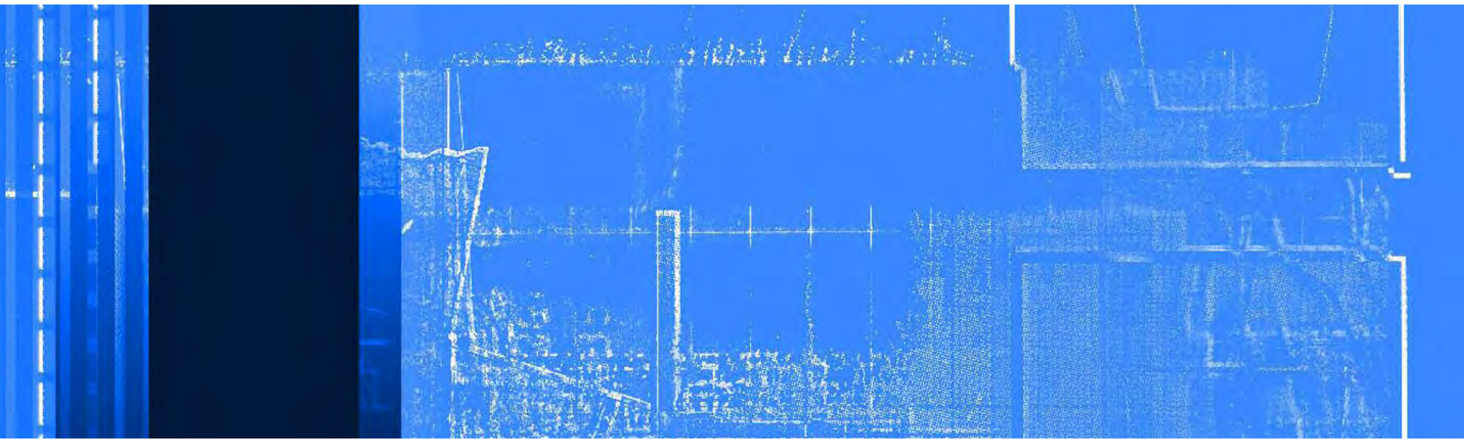
Kate Stewart (k.stewart@freescale.com)
Phil Odence (podence@blackducksoftware.com)

Why Are We Participating?

As an open source management vendor, Black Duck has helped hundreds of companies develop better approaches for consuming and contributing to open source projects. Customers typically consume data about open source usage from Black Duck (or from each other as part of a supply chain) in formats ill-suited to the purpose, generally open or proprietary formats that suit their existing office applications (ODF, Microsoft Excel and Word files, Adobe PDF files, etc.). These formats, being general-purpose text-oriented ones, are neither easily shared nor "machine readable" and therefore do not facilitate creation of an ecosystem of tools customers can use collaboratively to meet their needs. Much in the way other standards help entire industries to grow, Black Duck hopes by our participation to invest in a "lingua franca" to enable increased usage of open source throughout the world and to make it easier for everyone to do the right thing.

— Phil Odence is Co-chair of the FOSSBazaar SPDX Working Group and vice president of Black Duck Software.

Return to Table of Contents



Pointers in Objective-C

Mastering Objective-C is the key to unlocking the iPhone's potential

By Christopher K. Fairbairn and Collin Ruffenac

Within Objective-C, the use of object-oriented programming is, in fact, optional. Since Objective-C is based upon C foundation, it is possible to use C-style functions. However, Objective-C's full power is only unlocked if you make full use of its object-oriented extensions.

In this article, we will discover some of the benefits of object-oriented development. Let's take a look at one of the most basic concepts of object-oriented programming. A variable within an application can be considered to consist of four components. These are:

- Name
- Location — where it is stored in memory
- Type — what kind of data it can store
- Current value

Understanding where variables are located and if it is possible to access them by means other than their name is closely related to the concept of pointers.

Memory Maps

You can consider the memory of the iPhone as being made up of a large pile of bytes, each stacked one on top of another. Each byte has a number, called an address, associated with it, just like houses have an associated street number. Figure 1 represents several bytes of the iPhone's memory, starting at address 924 and extending through address 940.

When you allocate a variable within your application, the compiler reserves an explicit amount of memory for it. For example, a statement such as *int x = 45* will cause the compiler to reserve 4 bytes of memory to store the current value of *x*. This is represented in Figure 1 by the 4 bytes starting at address 928.

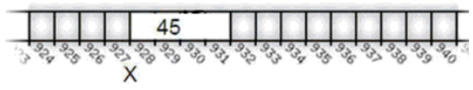


Figure 1: A sample snapshot of a region in the iPhone's memory showing the location of variable x.

Obtaining the Address of a Variable

Referring to the name of a variable within an expression will access or update its current value. By placing the *address-of* (&) operator in front of a variable name we are able to learn the address at which the variable is currently stored.

A variable that is able to store the address of another variable is called a "pointer". This is because the variable is said to point to the location of another value. The following code snippet demonstrates how we can use the *address-of* operator:

```
int x = 45;
int* y = &x;
```

This code snippet declares an integer variable *x* that is initialized to the value 45. It also declares variable *y* with a data type of *int**. The * at the end of the data type indicates a pointer and means we do not want to store an actual integer value but rather the memory address at which one can be found. This pointer is then initialized with the address of variable *x* using the *address-of* operator. If variable *x* had been stored at address 928 (as previously mentioned), we could graphically represent the result of executing this code snippet by updating the memory map to be similar to that in Figure 2.

Notice how the 4 bytes allocated to store variable *y* now store the number 928. When interpreted as an address, this indicates the location of variable *x*, as indicated by the arrow. The expression *b>y = &x* can be read as "place the address of variable *x* into variable *y*." Once you have an address stored within a pointer variable, it is only natural to want to determine the value of whatever it points at. This operation is called dereferencing the pointer and is also achieved using the * symbol, as demonstrated below.

```
int x = 45;
int *y = &x;
NSLog(@"The value was %d", *y);
```

The statement on the last line will print out the message "The value was 45" because the * in

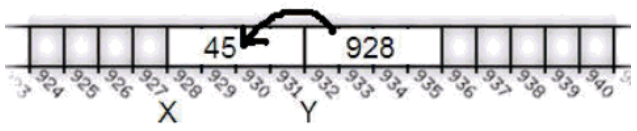


Figure 2: An updated memory map showing how variable y stores the address of variable x.

front of variable *y* causes the compiler to follow the pointer and access the value it currently points at. Instead of reading the value it is also possible to replace it, as demonstrated below. Confusingly, this also makes use of the * operator:

```
int x = 45;
int *y = &x;
*y = 92;
```

The statement on the last line stores the value 92 at the address located within variable *y*. Referring to Figure 2, you will see that variable *y* stores (or points to) address 928; hence, executing this statement actually updates the value of variable *x*, even though *x* is never explicitly referred to within the statement.

Arrays Are Pointers In Disguise

A variable identifier for a C-style array can at some level be thought of as a simple pointer. This pointer always points to the first element within the array. As an example, the following is perfectly valid Objective-C source code.

```
int ages[50];
int* p = ages;
NSLog(@"Age of 10th person is %d", p[9]);
```

Notice that we can assign the array variable to a pointer variable directly without the use of the *&* operator, and we can use the familiar *[]* syntax to calculate an offset from the pointer. A statement such as *p[9]* is simply another way to express **(p + 9)*. In other words, *p[9]* is a shorthand way to say, "Add 9 to the pointer's current value and then dereference it."

When working with pointers to structure-based data types, a special dereferencing syntax allows you to deference the pointer and access a specific field within the structure in a single step. To do this we use the -> operator, as demonstrated below:

```
struct box * p = ...;
p->width = 20;
```

The -> operator demonstrated on the second line dereferences the pointer *p* and then accesses the width field within the structure. While following a pointer to read or alter the value it points at, it is also helpful at times to compare two pointers to check if they point to identical values.

Comparing the Value of Pointers

When comparing the value of two pointers, it is important to ensure you are actually performing the intended comparison. As an example, consider the following code snippet:

```
int data[2] = { 99, 99 };
int* x = &data[0];
 int* y = &data[1];
if (x == y) { NSLog(@"The two values are the same"); }
```

If you expect this to emit the message "The two values are the same" you would be wrong. The statement *x == y* compares the address of each pointer and, since *x* and *y* both point to different elements within the data array, the statement returns *NO*.

If, instead, we wanted to determine whether the values pointed to by each pointer were identical, we would need to dereference both pointers as demonstrated below:

```
 if (*x == *y) { NSLog(@"The two values are the same"); }
```

Indicating the Absence of a Value

Sometimes you will want to detect if a pointer variable is currently pointing at anything of relevance. For this purpose, you will most likely initialize the pointer to one of the special constants *NULL* or *nil*, as demonstrated below:

```
 int* x = NULL;
NSString* y = nil;
```

Both constants are equivalent to the value 0 and are used to indicate that the pointer is not currently pointing at anything. The Objective-C convention is to use *nil* when referring to an object while relegating *NULL* for use with older C-style data types. Initializing the pointer to one of these special values enables a simple *if (y != nil)* check to determine if the pointer is currently pointing at anything. Since *nil* is equivalent to the value 0, you may also see this condition simply written as *if (!y)*.

Also be careful not to dereference a *NULL* pointer. Trying to read or write from a pointer that points to nothing (*NULL*) will cause an access violation error, which immediately exits your application.

Now that we understand the concept of pointers and how multiple pointers can reference the same object, we are ready to communicate with the object. Communicating with an object enables us to interrogate it for details it stores or request the object perform a specific task on our behalf using the information and resources at its disposal.

Summary

Enhancing the procedural C language to have object-oriented features is essentially what brought Objective-C to life. The benefits of developing applications in an object-oriented manor generally far out weight the extra effort required to learn the various amounts of additional terminology and techniques that object-orientation entails. Chief among the advantages of object-oriented programming is an improved ability to separate a complex application up into a number of smaller and discrete building blocks called classes. Rather than considering a large complex system, the developer's task becomes one of developing multiple smaller systems that combine and build on top of each other to perform tasks far more complex than any one part.

— Christopher Fairbairn and Collin Ruffenach are the authors of Objective-C for the iPhone *(http://www.manning.com/fairbairn/).*

Return to Table of Contents

The JavaFX JDBC Data Source

JavaFX has built-in support for JDBC and various data formats

By Eric J. Bruno

In the article "JavaFX Database Programming with Java DB" (www.drdobbs.com/java/224202518), I discussed ways to integrate JavaFX with databases using JDBC, thanks to JavaFX-Java integration. One topic I didn't cover was JavaFX's built-in support for JDBC and various data formats through a set of Data Source components. This includes relational databases, files, the file system itself, and web-based data (XML, JSON, and so on). In this article, I examine how to use them to write very little data handling code. To do so, I use the JavaFX Composer tool (a NetBeans plug-in) to do the heavy lifting.

The JavaFX Composer

If you plan on doing a lot of JavaFX GUI work and don't know about the JavaFX Composer, prepare to get acquainted. Initially released as a beta component with JavaFX 1.2, it was only recently officially released along with JavaFX 1. You can download the NetBeans 6.9 / JavaFX 1.3 bundle from http://javafx.com if you haven't already. With the Composer, you can visually lay out components from the NetBeans palette that's located in a pane in the upper right of the screen (see Figure 1). Refer to JavaFX Database Programming with Java DB for more in-depth information on using the Composer plug-in.

All of the JavaFX layout containers, controls, shapes, effects, charts, data sources, and so on, are available in the palette to be dragged and dropped onto your application's Scene. To begin, drag a List View control onto the Scene, and place it on the left side. Next, drag a second List View control and place it to the right of the first one. Next, locate the JDBC Data Source component (listed under Data Sources in the Palette), and drag one onto the Scene as well. Once you do, a window titled Data Source Customizer will appear. This is where you fill in the JDBC source information.

The JavaFX JDBC Data Source

Let's fill in the information required by the JDBC Data Source. To do this, use the sample "programmer's library" database I presented in JavaFX Database Programming with Java DB. Follow the directions in that article to create a Java DB entry within the NetBeans Services tab, and then choose it as the Connection URL in this window when you hit the Browse button. Next, enter the user name and password for the database in the appropriate fields (use "dbuser" for both with the sample library database).

You need to specify a query for the data source. You can either type the query manually in the SQL Query field, or you can have the Composer help you generate it. To do that, click the Create button and select *theBOOK* table in the Tables list of the smaller Browse Tables window that appears (see Figure 2). Click OK to set the query, and the window will close. To test your SQL, click the Fetch Data button, and the Query Result list should fill with the results of your query.

Since we'll be adding a second JDBC data source later, rename this one to *jdbcBooks*. To do that, locate and select the component under the Data Sources entry in the NetBeans Navigator pane. Then, type the new name in the Identifier property within the Properties pane.

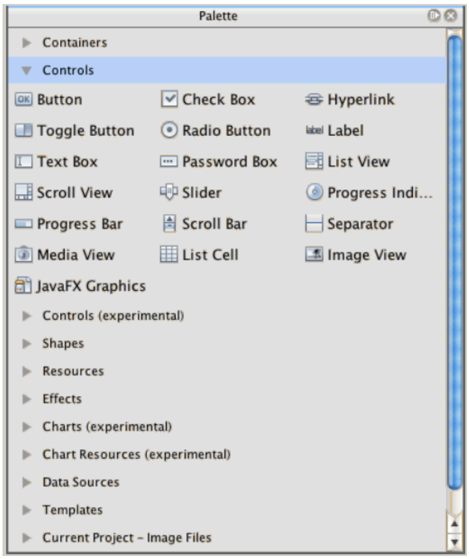


Figure 1: JavaFX Composer Palette.

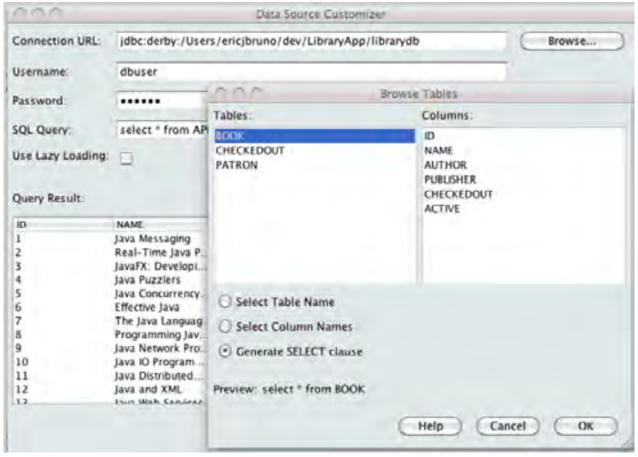
Next, you need to bind the list view's data to the results of the query. To do this, select the first list view control you added (the one on the left), then locate the Items property in the Properties pane. To the far right of the property, there is a small rectangle (see Figure 3). Click this rectangle and a window will open where you can set this property. I've highlighted both the property and the rectangle in Figure 3 to help you locate these.

Next, you need to bind the list view to the data from the JDBC data source, so click the Bind button at the top of the pop-up window for the Items property. In this window, you'll find three lists: in the Components list, select *jdbcBooks*; in the Property list, select *All Records*; and in the Converters list, select *Record[] -> String[]* (see Figure 4). Before you click the Close button, you need to make a slight modification to the *bind* statement. Simply change the text "name" in the call to *record.getString("name")* to all uppercase; for example, *record.getString("NAME")*.

Next, you want to populate the second list view with the list of library patrons from the database. To do this, add a second JDBC Data Source component and connect it to the same library database as the previous data source, but this time use *select * from APP.PATRON* for the SQL statement. Additionally, you should name this instance *jdbcPatrons* to avoid confusion. To bind the second list view's items to this data source, follow the same steps as you did above, with these exceptions:

1. Select the second list view on the Scene.
2. In the Bind pop-up window, choose *jdbcPatrons* in the Component list, *All Records* in the Property list, and *Record[] -> String[]* in the Converters list.
3. At the end of the *bind* statement, replace *record.getString("name")* with "*{record.getString("FIRSTNAME")} {record.getString("LASTNAME")}*". Be sure to include the surrounding quotes as shown here.

You're almost ready to run the application. First, be sure everything compiles in case you missed a step or made a typo along the way. Next, since you're using a Java DB (Apache Derby) database, you need to include the client JAR file in your project. To do this, right-mouse click on the project in NetBeans and select Properties from the pop-up menu. Next, select Libraries in the list, and then click the Add JAR/Folder button on the right. Locate and select the derby.jar file, and click OK to close the Project Properties window. Finally, execute the application from within NetBeans so that if any errors or exceptions occur, you'll see them in the Output window. If all goes well, the result will be as in Figure 5.

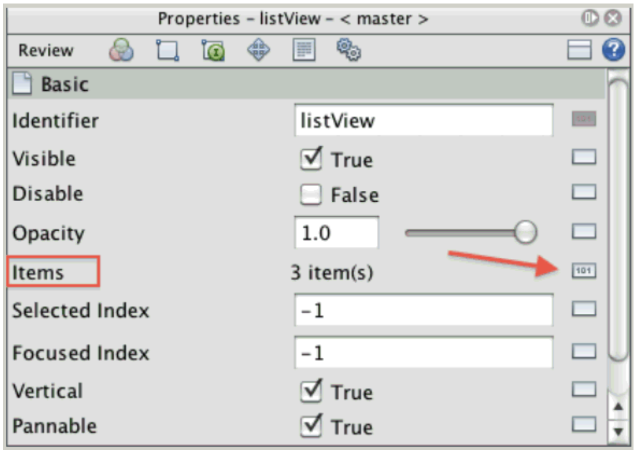


Connection URL: jdbc:derby:/Users/ericjbruno/dev/LibraryApp/librarydb
Username: dbuser
Password: ••••••
SQL Query: select * from AP
Use Lazy Loading: ☐
Query Result:

ID	NAME
1	Java Messaging
2	Real-Time Java P...
3	JavaFX: Developi...
4	Java Puzzlers
5	Java Concurrency...
6	Effective Java
7	The Java Languag...
8	Programming Jav...
9	Java Network Pro...
10	Java IO Program...
11	Java Distributed...
12	Java and XML

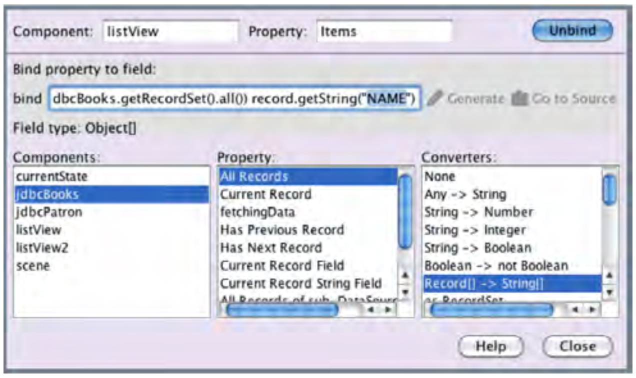
Tables: BOOK, CHECKEDOUT, PATRON
Columns: ID, NAME, AUTHOR, PUBLISHER, CHECKEDOUT, ACTIVE
○ Select Table Name
○ Select Column Names
● Generate SELECT clause
Preview: select * from BOOK

Figure 2: The Data Source Customizer window.



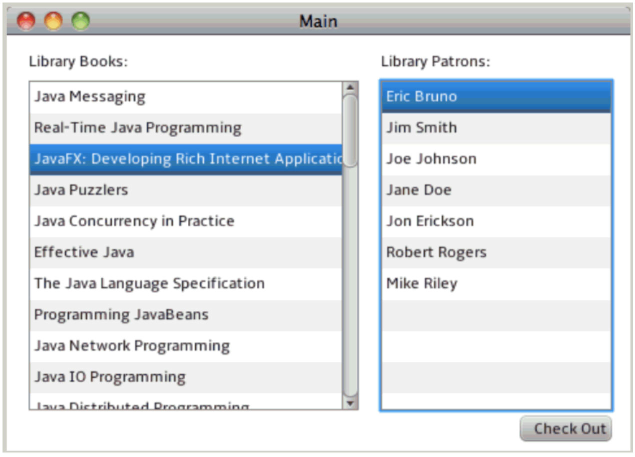
Identifier	listView	
Visible	✓ True	
Disable	☐ False	
Opacity	1.0	
Items	3 item(s)	
Selected Index	-1	
Focused Index	-1	
Vertical	✓ True	
Pannable	✓ True	

Figure 3: The Data Source Customizer window.



Component: listView Property: Items
Bind property to field:
bind dbcBooks.getRecordSet().all()) record.getString("NAME")
Field type: Object[]
Components: currentState, jdbcBooks, jdbcPatron, listView, listView2, scene
Property: All Records, Current Record, fetchingData, Has Previous Record, Has Next Record, Current Record Field, Current Record String Field
Converters: None, Any -> String, String -> Number, String -> Integer, String -> Boolean, Boolean -> not Boolean, Record[] -> String[]

Figure 4: Binding the list view's items to the JDBC data source.



Library Books: Java Messaging, Real-Time Java Programming, JavaFX: Developing Rich Internet Applicatio, Java Puzzlers, Java Concurrency in Practice, Effective Java, The Java Language Specification, Programming JavaBeans, Java Network Programming, Java IO Programming

Library Patrons: Eric Bruno, Jim Smith, Joe Johnson, Jane Doe, Jon Erickson, Robert Rogers, Mike Riley

Figure 5: The running JavaFX database application.

Next, let's take a peek at what's behind all of the JavaFX magic you've explored so far.

Inside the Code

When you work with the JavaFX Data Source component, as well as the JavaFX Composer, you'll notice that you don't need to manually write any code. However, there may be times you'd like to go further than this example or what the Composer allows you to do. Fortunately, you can take a look at the raw JavaFX Script that's generated, which you can use as a starting point in the future. To do this, switch the file to Source mode in NetBeans, and expand the "Generated Code" section in the listing.

As you scroll through the code, you'll find the object declarations for the components you dragged onto the Scene. For instance, the code below is for the two list view controls you added, and includes the *bind* statements that were generated by the Composer:

```
public-read def listView: ListView = ListView {
   layoutX: 17.0
   layoutY: 35.0
   layoutInfo: __layoutInfo_listView
   items: bind for(record in jdbcBooks.getRecordSet().all()) /
          record.getString("NAME")
}

public-read def listView2: ListView = ListView {
   layoutX: 288.0
   layoutY: 35.0
   layoutInfo: __layoutInfo_listView2
   items: bind for(record in jdbcPatron.getRecordSet().all()) /
          "{record.getString("FIRSTNAME")} {record.getString("LASTNAME")}"
}
```

Connecting the two JDBC data source components involves only two straightforward object declarations:

```
public-read def jdbcBooks: DbDataSource = DbDataSource {
   connectionString: "jdbc:derby:/db/LibraryApp/librarydb"
   user: "dbuser"
   password: "dbuser"
   query: "select * from APP.BOOK"
}

public-read def jdbcPatron: DbDataSource = DbDataSource {
   connectionString: "jdbc:derby:/db/LibraryApp/librarydb"
   user: "dbuser"
   password: "dbuser"
   query: "select * from APP.PATRON"
}
```

Since JavaFX Script is a declarative language (as opposed to an imperative language), you simply state what you want to do, and the runtime takes care of how to do it. The rest of the code includes labels and the button, along with some layout, but the JavaFX Script object declarations and the *bind* statements that connect them do all of the work. Listing 1 contains the the source for the entire application.

Listing 1: The JavaFX JDBC Data Source sample application (Main.fx)

```
   package desktopapplication;

/**
* @author ericjbruno
*/

public class Main {
    def __layoutInfo_listView: javafx.scene.layout.LayoutInfo = javafx.scene.layout.LayoutInfo {

        width: 254.0
    }
    public-read def listView: javafx.scene.control.ListView = javafx.scene.control.ListView {
        layoutX: 17.0
        layoutY: 35.0
        layoutInfo: __layoutInfo_listView
        items: bind for(record in jdbcBooks.getRecordSet().all()) record.getString("NAME")
    }
    def __layoutInfo_listView2: javafx.scene.layout.LayoutInfo = javafx.scene.layout.LayoutInfo {
        width: 178.0
    }
    public-read def listView2: javafx.scene.control.ListView = javafx.scene.control.ListView {
        layoutX: 288.0
        layoutY: 35.0
        layoutInfo: __layoutInfo_listView2
        items: bind for(record in jdbcPatron.getRecordSet().all()) "{record.getString("FIRSTNAME")} {record.getString("LASTNAME")}"
    }
    public-read def label: javafx.scene.control.Label = javafx.scene.control.Label {
        layoutX: 17.0
        layoutY: 13.0
        text: "Library Books:"
    }
    public-read def label2: javafx.scene.control.Label = javafx.scene.control.Label {
        layoutX: 288.0
        layoutY: 13.0
        text: "Library Patrons:"
    }
    public-read def btnCheckOut: javafx.scene.control.Button = javafx.scene.control.Button {
        layoutX: 395.0
        layoutY: 293.0
        text: "Check Out"
    }
    public-read def scene: javafx.scene.Scene = javafx.scene.Scene {
        width: 480.0
        height: 320.0
        content: getDesignRootNodes ()
    }
    public-read def jdbcBooks: org.netbeans.javafx.datasrc.DbDataSource = org.netbeans.javafx.datasrc.DbDataSource {
        connectionString: "jdbc:derby:/db/LibraryApp/librarydb"
        user: "dbuser"
        password: "dbuser"
        query: "select * from APP.BOOK"
    }
    public-read def jdbcPatron: org.netbeans.javafx.datasrc.DbDataSource = org.netbeans.javafx.datasrc.DbDataSource {
        connectionString: "jdbc:derby:/db/LibraryApp/librarydb"
        user: "dbuser"
        password: "dbuser"
        query: "select * from APP.PATRON"
    }
    init {
    }
    public-read def currentState: org.netbeans.javafx.design.DesignState = org.netbeans.javafx.design.DesignState {
        names: [ ]
        timelines: [
        ]
    }
    public function getDesignRootNodes (): javafx.scene.Node[] {
        [ listView, listView2, label, label2, btnCheckOut, ]
    }
    public function getDesignScene (): javafx.scene.Scene {
       scene
    }
}
function run (): Void {
    var design = Main {};
    javafx.stage.Stage {
        title: "Main"
        scene: design.getDesignScene ()
    }
}
```

Conclusion

When working with JavaFX Script alone, performing tasks such as UI layout and database connectivity are very straightforward, and far less complicated compared with Java and Swing. Additionally, NetBeans and the JavaFX Composer take it a step further, and make building applications — even database applications — as simple as drag-and-drop. As with most frameworks, there's always some coding involved, but in this case it amounted to modifying two lines of code (the two *bind* statements). Working with files and web-based data in comma-delimited, line-deliminted, XML, or JSON formats are equally as straightforward. I plan to extend this article to cover those two formats in the near future.

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

Return to Table of Contents

Combinatorial Testing

Complex problems sometimes require clever testing

By Dennis Shasha

Long-time *Dr. Dobb's* readers surely remember *Dr. Ecco's Omniheuristic Corner*, where Dennis Shasha posed puzzles that required computer solutions. Dennis is returning to *Dr. Dobb's*, but with a new twist on puzzles — one that takes inspiration from the challenging apps that readers have faced and to turn those into puzzles.

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

In the 1990s, four researchers at Telcordia found a good way to test software. Instead of trying to achieve high code coverage (which might not be possible when one doesn't control the source), they proposed to test the external behavior of the software in a clever way. By "external behavior" I mean testing the interface and user-settable configuration parameters. For example, if testing the implementation of a language, one would test language constructs and optimization parameters.

Why does one need to be clever about such testing? The short answer is — combinatorial explosion. A language will often allow a countably infinite number of statements. It's impossible to test them all. The good news is that they found that many software bugs occur because of an interaction between two factors (e.g., two language features or perhaps a language feature and a configuration parameter), so if they could test every combination of values of every pair of factors, they would often catch most if not all bugs. Because even a single factor may have an infinite number (or at least billions) of values (e.g., all floats), one has to discretize the test space. This may mean extensive sampling on boundary conditions and a few samples of large intermediate sets of values.

For example, suppose you are testing a new SQL engine. Optimization parameters include buffer space, query caching, replacement algorithm, and so on. Language features include the formation of the *SELECT* clause, number of tables, the *WHERE* clause, grouping, aggregates, and subqueries.

In my work, I like to lay out each factor in a line along with its values, as you can see below:

buffer space: 20 megabytes, 200 megabytes, 2 gigabytes, 20 gigabytes, 200 gigabytes
query caching: off, on
replacement algorithm: least recently used, most recently used
select clause: asterisk, named columns, aggregates
number of tables: 1, 2, 10, 50
grouping: unused, used one attribute, used many attributes
aggregates: none, single, five
subqueries: none, one, five

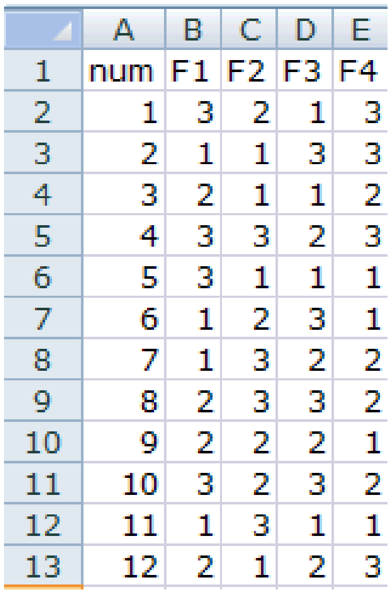
Testing all these combinations would require 5 * 2 * 2 * 3 * 4 * 3 * 3 * 3 = 6,480 experiments. You may consider this to be too many, so you'd like to do far fewer but still satisfy the two factor condition: For every pair of values in every pair of factors, some experiment tests this pair. The puzzle is to find such a small set of experiments.

Warm-Up

Suppose there were only four factors each having three values:

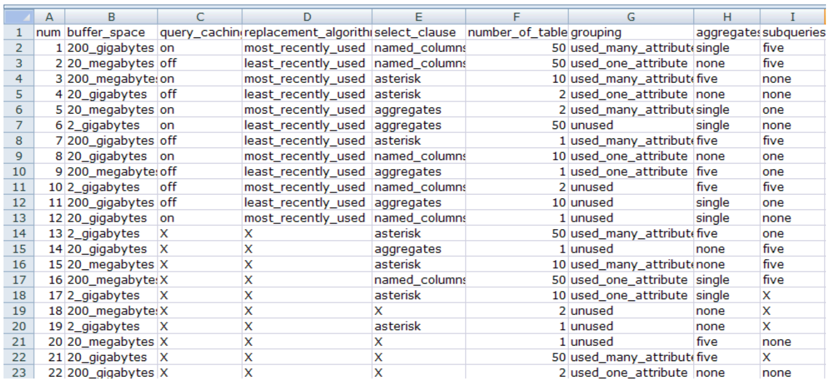
F1: 1 2 3
F2: 1 2 3
F3: 1 2 3
F4: 1 2 3

Testing every combination requires 3*3*3*3 = 81 experiments, but achieving the two factor condition requires only 12 experiments: see Figure 1.



	A	B	C	D	E
1	num	F1	F2	F3	F4
2	1	3	2	1	3
3	2	1	1	3	3
4	3	2	1	1	2
5	4	3	3	2	3
6	5	3	1	1	1
7	6	1	2	3	1
8	7	1	3	2	2
9	8	2	3	3	2
10	9	2	2	2	1
11	10	3	2	3	2
12	11	1	3	1	1
13	12	2	1	2	3

Figure 1



	A	B	C	D	E	F	G	H	I
1	num	buffer_space	query_cachin	replacement_algorith	select_clause	number_of_table	grouping	aggregate	subqueries
2	1	200_gigabytes	on	most_recently_used	named_column	50	used_many_attribut	single	five
3	2	20_megabytes	off	least_recently_used	named_column	50	used_one_attribute	none	five
4	3	200_megabyte	on	most_recently_used	asterisk	10	used_many_attribut	five	none
5	4	20_gigabytes	off	least_recently_used	asterisk	2	used_one_attribute	none	none
6	5	20_megabytes	on	most_recently_used	aggregates	2	used_many_attribut	single	one
7	6	2_gigabytes	on	least_recently_used	aggregates	50	unused	single	none
8	7	200_gigabytes	off	least_recently_used	asterisk	1	used_many_attribut	five	five
9	8	20_gigabytes	on	most_recently_used	named_column	10	used_one_attribute	none	one
10	9	200_megabyte	off	least_recently_used	aggregates	1	used_one_attribute	five	one
11	10	2_gigabytes	off	most_recently_used	named_column	2	unused	five	five
12	11	200_gigabytes	off	least_recently_used	aggregates	10	unused	single	one
13	12	20_gigabytes	on	most_recently_used	named_column	1	unused	single	none
14	13	2_gigabytes	X	X	asterisk	50	used_many_attribut	five	one
15	14	20_gigabytes	X	X	aggregates	1	unused	none	five
16	15	20_megabytes	X	X	asterisk	10	used_many_attribut	none	five
17	16	200_megabyte	X	X	named_column	50	used_one_attribute	single	five
18	17	2_gigabytes	X	X	asterisk	10	used_one_attribute	single	X
19	18	200_megabyte	X	X	X	2	unused	none	X
20	19	2_gigabytes	X	X	asterisk	1	unused	none	X
21	20	20_megabytes	X	X	X	1	unused	five	none
22	21	20_gigabytes	X	X	X	50	used_many_attribut	five	X
23	22	200_gigabytes	X	X	X	2	used_one_attribute	none	none

Figure 2

On the left you see the experiment number followed by the values of each factor in each experiment. For example, experiment 5 has value 3 for factor *F1*, and 1 for all other factors. The two factor condition is achieved because for any pair of factors, e.g. *F1* and *F4*, every pair of values is found in at least one experiment. Experiment 1 has *F1* = 3 and *F4* = 3, experiment 2 has *F1* = 1 and *F4* = 3, experiment 3 has *F1* = 2 and *F4* = 2, experiment 4 repeats *F1* = 3 and *F4* = 3, and so on. A covering set for *F1* and *F4* would be experiments 11, 7, 2, 9, 3, 12, 5, 10, and 1.

End of Warm-Up

Here is the challenge for you: Can you build a program that will give you a set of experiments that satisfies the two factor condition for the SQL case?

Hint: You need fewer than 25 experiments.

The original paper by the Telcordia people is: "The Combinatorial Design Approach to Automatic Test Generation," by David M. Cohen, Siddhartha R. Dalal, Jesse Parelius, Gardner C. Patton (http://www.argreenhouse.com/papers/gcp/AETGissre96.shtml).

Solution

In this solution, the X represents a "don't care," meaning any value for that factor would be okay; see Figure 1.

Reader Response

Since this puzzle was originally posted on www.drdobbs.com, Michael Schneider of Saarbruecken, Germany suggested a simple nine-experiment solution to the warm-up problem.

This one is clearly optimal since even just two factors demand nine experiments as can be seen in the Excel spreadsheet posted at http://cs.nyu.edu/cs/faculty/shasha/papers/schneiderCombinatorialTesting.xls.

— Dennis Shasha is a professor of Computer Science at New York University. His latest puzzle book Puzzles for Programmers and Pros *(http://www.wrox.com/WileyCDA/WroxTitle/Puzzles-for-Programmers-and-Pros.productCd-0470121688.html) outlined some general approaches to programmatic puzzle solving. This column challenges you to find the solutions to the puzzles that lie at the core of some cool tough applications. He solicits your help in identifying such cool apps.*

ClICK HERE TO COMMENT ON THIS PUZZLE (http://www.drdobbs.com/architecture-and-design/225701535)

Return to Table of Contents

Enterprise Development with Flex

Reviewed by Mike Riley

Enterprise Development with Fle:
Best Practices for RIA Developers
Yakov Fain, Victor Rasputnis and Anatole Tartakovsky
O'Reilly Media
$54.99

O'Reilly has been on a Flex fixation, with Enterprise Development with Flex being their second Flex book publishing in the last few months. Two weeks ago, I reviewed O'Reilly's *Flex 4 Cookbook* (http://www.drdobbs.com/blog/archives/2010/06/flex_4_cookbook.html), a "must-have" title for Flex developers. Does Enterprise Development with Flex match this categorization?

This multi-authored book begins with a comparison of several popular Flex frameworks such as Cairngorm (http://opensource.adobe.com/wiki/display/cairngorm/Cairngorm), Mate (http://mate.asfusion.com/), and puremvc (http://puremvc.org/) ActionScript framework, and ultimately decide that the Clear Toolkit (http://cleartoolkit.sourceforge.net/) is their framework of choice. Considering the book's authors created the open source MIT-licensed toolkit, this should come as no surprise.

Next, various design patterns ranging from Singleton, Proxy, and Mediator to Data Transfer Object, Asynchronous Token, and Class Factory are examined. Chapter 3 discusses the components required to build a Flex-based enterprise framework. Chapter 4 covers the external factors such as the type of staff talent required, their workstation configurations, web page interactions, testing, logging, and documenting the project at hand. Chapter 5 on Flex Messaging offers an excellent dive into one of Flex's more challenging aspects. Adobe's Java-based BlazeDS (http://opensource.adobe.com/wiki/display/blazeds/BlazeDS/) and LiveCycle Data Services (LCDS) are reviewed and demonstrated. Modules and Library options are detailed in Chapter 7, followed by a 50+ page chapter on performance optimization suggestions. Adobe Air and LiveCycle Enterprise Suite are also explored. Chapter 11 on Printing with Flex is another one of the book's gems, as this seemingly simple task often confounds Flex developers seeking the 'flexibility' they need in dynamically generating various PDF-based reports.

The book concludes with an introduction to Model-Driven development courtesy of Adobe's LCDS ES2 (http://www.adobe.com/products/livecycle/dataservices/). The authors show how the process begins using Flash Builder 4, walking through what is generated and how master/detail/search views can be created.

Numerous screenshots and copious quantities of code listings accompany the descriptive text, and the authors clearly have the Flex expertise necessary to be the authoritative subject matter experts on Flex enterprise development. I earned a deeper level of respect for the Flex framework reading this book, though I still don't see myself using it for internal projects any time soon. However, knowing the capabilities this expansive environment offers has better prepared me for the time when the request arises to consider if Flex has a role in the enterprise I support.

I have only a few criticisms. So much can be said about Flex that the book could have easily been divided into two or more volumes. More chapters like those on the subjects of printing and messaging would address these very basic yet very challenging enterprise application feature expectations. I would have also preferred to see chapters on integrating Flex with other languages besides Java, such as PHP (beyond the very brief mention of AMFPHP; http://www.amfphp.org/) and ASP.NET.

The authors conclude the book by stating their intended objective: "We tried to discuss the most important subjects that Flex/AIR practitioners face while working on an enterprise RIA. We tried not to just give you better Flex components, but to explain how you can build similar or better ones for your enterprise-wide framework." They succeeded.

Return to Table of Contents

Q&A: Bilski Follow-Up

What the U.S. Supreme Court did — and didn't — decide

by Jonathan Erickson

The U.S. Supreme Court recently had its say regarding the case of Bilski v. Kappos, a case that important ramifications for software developers. To get a clear understanding of the Supreme Court's decision, we recently spoke with Craig Hemenway, a partner in Dorsey Whitney Intellectual Property practice group.

Dr. Dobb's: Can you give us a snapshot of Bilski? A quick historical account of the case.

Hemenway: In Bilski, the Federal Circuit was asked to rule on the patentability of a method for hedging risks in trades. The Circuit took the opportunity to set out a test for the patentability of business methods. Specifically, the Federal Circuit endorsed the so-called "machine or transformation" test, in which a method or process is only patentable subject matter if it is tied to a particular machine or transforms an article in some fashion.

Dr. Dobb's: What is the core legal issue at stake?

Hemenway: The Supreme Court granted certiorari and issued its decision in Bilski v. Kappos on Monday. In the decision, the Court unanimously upheld the Federal Circuit's ruling that Bilski's claims were not patentable, stating that the claims covered nothing more than an abstract idea. The Court likewise held that the "machine or transformation" test is not an exclusive test, but indicated it could still provide a good clue as to the patentability of a method.

Dr. Dobb's: What did the Supreme Court decide?

Hemenway: The Supreme Court split on the issue of whether or not business methods should be patentable at all. Four Justices agreed that certain business methods could be patentable but declined to provide guidance or examples. In this portion of the opinion, Justice Kennedy indicated that "new technologies may call for new inquiries." In short, Kennedy acknowledged that inventions today may be different than inventions seen in the so-called "Industrial Age" and that an industrial-era test may not be appropriate now. Given the similarities between business method claims and software claims, much of this discussion is germane to software patents as well.

Four other Justices filed a second opinion that business method patents should never be patentable. This opinion suggests that business methods were not considered patentable historically, and so should not be considered patentable subject matter now.

In short, business methods and software still may be eligible for patent protection and the "machine or transformation test," which could exclude many such inventions, is not the sole law of the land for determining patentability. However, the Supreme Court left ambiguity in place by refusing to articulate any particular patentability test. This, it appears, will be left up to the Federal Circuit and the Patent Office. Given the Federal Circuit's long history with patent law, perhaps this was the wisest possible choice.

Dr. Dobb's: How will this affect software developers?

Hemenway: Software developers should continue to seek patent protection where appropriate and possible. However, developers and companies need to realize that the law is again in a state of flux and a new test may come out that could — again — change the playing field. Thus, although software patent applications are legitimate and valid, one should prepare for the possibility that future rulings could impact existing patents. I would suggest continuing to include at least some claims in software and business method patents that satisfy the "machine or transformation" test to ensure that future rulings don't strip issued patents in these areas of all enforceability. After all, if the "machine or transformation" test is too restrictive in view of Bilski, then any claims satisfying its requirements should likewise meet the (presumably) relaxed standards of the future.

Dr. Dobb's: How will this decision affect software users?

Hemenway: Software users shouldn't see much of an impact from this ruling. In most cases users have legal copies of software in the first place and software is also protected by copyright. To the extent that patent coverage for software is expanded by Bilski, it may lead to an uptick in patent infringement lawsuits against developers, distributors, and manufacturers.

Dr. Dobb's: Thanks for your time.

Return to Table of Contents

Standards, Stacks, and Mergers: The SQL Backstory

By Ken North

In a movie plot, a backstory can help an actor understand the role he is playing and the audience to understand a charcter. Likewise, the SQL backstory helps us understand why some corporations spent billions to acquire companies with a flagship SQL database platform. It also provides perspective as we look at the software eco-system and technology stack for building successive generations of services and applications.

The SAP-Sybase and Oracle-Sun mergers provide an entree to discuss various topics about databases. One is a retrospective on SQL databases and how SQL vendor acquisitions provide a yardstick for measuring database technology. What's the reach of the SQL database today? Will it be included in the technology stack for the next generation of applications and services?

IBM, Oracle, Microsoft, and SAP are betting the answer is 'Yes'. For example, recent acquisitions enabled SAP and Oracle to add a major SQL platform to their product portfolio. Both companies spent billions to broaden their product lines and the value of database platforms was undoubtedly a factor in the acquisition price. Oracle, IBM and Microsoft rank 1-2-3 in the database market, with each company having billions in revenue from its database products. Ninety percent of enterprises have a mission-critical application running on a database server from IBM, Oracle, or Microsoft.

Database Industry In the Wayback Machine

The history of SQL since its development by Boyce and Chamberlin, the IBM System R incubator and the founding of Oracle spans several decades. SQL was developed to provide a declarative query language in an era when we were still working on expensive mainframe computers. But as computing and software have evolved, SQL database technology has proven to be resilient and adaptable.

During the SQL lifespan of almost 30 years, we've seen the arrival of networks, proliferation of personal computers, emergence of open source software, and software industry expansion and consolidation.

One reason for SQL's resilience is a commitment by database companies to standards. The first SQL standard arrived in 1986 and there have been several major updates. Nonetheless, periodic technology shifts usually bring out predictions the SQL database won't be with us much longer. But SQL vendors continue to evolve their platform, including Sybase. SAP would not invest billions to acquire a company whose primary revenue stream comes from technology that was soon to be obsolete.

When products such as DB2 and Oracle arrived on the database scene, the SQL database was a mainframe or minicomputer proposition. Proponents of hierarchical and CODASYL databases argued SQL performance was unacceptable, but SQL technology offered some advantages. It avoided painful, long-running database reorganizations required by schema changes and it offered a superior capability for ad hoc queries.

What ramped up SQL database sales was adoption of networks and client-server architecture, the infrastructure for distributed processing. The attraction of low-cost, decentralized computing produced an explosion of PCs and networks in institutional settings. This resulted in a proliferation of ISAM software for desktop databases and SQL products for server databases. The database wars kicked into high gear. SQL became a buyer's market with more than 150 products and there was an update to the SQL standard in 1992.

Amidst growing adoption of SQL, object-oriented programming (OOP) became the rage. Some believed the new programming paradigm would prove to be an SQL killer.

After object-oriented programming (OOP) became the fashion, there arose an argument that an impedance mismatch between SQL and objects called for a new type of database. Some believed the Object Database Management Group (ODMG) standard and object databases would wipe out the franchise of database vendors such as Oracle, IBM, and Sybase. Instead, object databases remained a niche product. Some even added a SQL interface as a concession to the number of 'shrink-wrapped' applications that used SQL queries.

By 1997, it was suggested the World Wide Web would be the catalyst for abandonment of the SQL database. Subsequently it was argued that XML, and then open source databases, would undermine the maintream SQL vendors. Most recently some have argued that NoSQL databases will displace both commercial and open source SQL products. At times it seems the successive wave of 'SQL-killer' pronouncements is driven less by problems with the technology and more by a desire to displace vendors and open up markets.

Is Database Dead?

So how accurate was the 'database is dead' mantra? Not very. The database market grew every year from 2000 to 2009, with SQL products holding the lion's share.

To better understand the importance of the SQL database to today's technology stack, we can look at the acquisition of several companies with a mainstream SQL DBMS in their product portfolio.

Forrester Research recently estimated the total database market (licenses, support, consulting) would grow from $27 billion in 2009 to $32 billion by 2012. SQL technology is entrenched in many organizations and across millions of web sites. Perhaps that explains why, during the past decade, IBM, Oracle, Sun, and SAP made billion-dollar investments in a 'dead' technology.

During the Dot Com Boom, investors were in love with 'new tech' companies. Some argued the Web was going to change everything and make existing software technology obsolete. Some industry pundits suggested the database industry was going into a death spiral. But they based that conclusion on the wrong indicators. What they were observing was the thinning of the herd, not the demise of the entire database industry. There was a trend towards industry consolidation and the bigger vendors gained market share while evolving their SQL products along the Swiss Army Knife model. Today IBM, Oracle and Microsoft hold more than 80% of the database market.

Informix and IBM

The "database is dead" rumors were an enabler for a major IBM acquisition of database technology and customers. IBM acquired Informix, a company that was a leader in object-relational technology. Informix had an early solution to integrating XML data into SQL databases and was the only company with a solution for embedding both Java classes and OLE components in SQL databases. IBM paid $1 billion for a company with $840 million in annual revenue, a price that was 1.2 times annual revenue.

MySQL, Sun, and Oracle

The Internet and XML databases did not eradicate commercial SQL products, but some believed the open source DBMS would do the job. Enthusiasm for open source software and widespread adoption of the LAMP stack produced a perfect storm in which Sun Microsystems acquired MySQL AB for $1 billion. For the same price IBM paid for Informix, Sun picked up a company approaching $100 million in annual revenue. The price tag to Sun for forging an identity as a champion of open source databases was roughly 10 times annual revenue.

Sun CEO Jonathan Schwartz said:

> *"MySQL's employees and culture, along with its near ubiquity across the Web, make it an ideal fit with Sun's open approach to network innovation. And most importantly, this announcement boosts our investments into the communities at the heart of innovation on the Internet and of enterprises that rely on technology as a competitive weapon."*

At the time of the MySQL acquisition, I wrote of one favorable outcome. Sun could produce a unique new generation of database servers by tight integration of Java and MySQL with the transactional memory of the UltraSPARC (Rock) processor. But Sun did not put the Rock processor into production and Oracle has no plan to do so.

During the era when Digital Equipment Corporation (DEC) was failing, Oracle acquired DEC's relational and CODASYL model database products. It still offers the CODASYL product. Oracle has done some serious shopping for companies, buying 67 of them since 2005. It acquired Sleepycat Software, maker of the Berkeley DB product, and TimesTen, maker of a leading in-memory SQL database product. With the acquisition of Sun Microsystems for $7.3 billion, Oracle has added hardware and yet another DBMS to its product portfolio.

Oracle has niche database products and a major presence in applications suites, analytics and enterprise computing. With the acquisition of Sun, Oracle expands its portfolio with the Java platform, server hardware and the most popular SQL database for powering web sites. The Sun acquisition contributed $1.23 billion in hardware sales and $400 million in profits to Oracle's fourth-quarter earnings report.

SAP and Sybase

SAP offered $5.6 billion for a company that recently experienced double-digit growth in database licensing revenue. For 2009, Sybase reported annual revenue of $1.17 billion. For the first quarter of 2010, it reported 10% year-over-year revenue growth to $294 million. SAP paid approximately 4.78 times annual revenue for Sybase. The Informix and MySQL deals were for database companies. But in Sybase, SAP sees the growth potential of other products in addition to the database platforms.

Technology Stacks, APIs

Over the lifetime of the SQL database, we've seen the emergence of various software and computing paradigms. Early SQL developers were programming mainframe computers and creating monolithic executables, using embedded SQL (ESQL) and subroutine libraries. SQL developers were migrating to smaller computers (minicomputers, servers, PCs), client-server architecture programming, Windows and dynamic linking. ESQL became an industry standard in 1989. The C language gained a following and there were soon C call-level interfaces for accessing SQL databases. The Open Database Connectivity (ODBC) API was refined to produce the SQL Call-Level Interface (SQL CLI) standard in 1995. Java followed, along with the JDBC™ API.

The popularity of the Internet technology stack encouraged development of web database interfaces. XML-based protocols encouraged the development of web services and collaborative applications, followed soon by emphasis on service-oriented architecture (SOA). The interest in Web-Oriented Architecture (WOA) put a heavy emphasis on solutions to enable JavaScript, PHP, Perl, and scripting languages to access SQL data.

Regardless of the evolution of client, middle-tier and web development technologies, SQL at the back end has remained a constant. Today ODBC and JDBC are still the predominant APIs for accessing databases, although service provider interfaces and XML-based data services are often part of a system architecture. The SQL language standard and standard APIs are undoubtedly reasons companies have decided having an SQL DBMS in the product portfolio is important. Proponents of some newer technologies, such as NoSQL databases and cloud computing, are still sorting out issues of not having standard APIs. In a recent webinar, the CTO of a leading cloud computing company indicated that proliferation of APIs for the cloud could slow the growth of cloud computing. The SQL database industry learned that lesson long ago and subsequently saw that growth followed adoption of standards.

ClICK HERE TO COMMENT ON THIS POST (http://www.drdobbs.com/blog/archives/2010/06/database_industhtml)

Return to Table of Contents