SPECIAL ISSUE

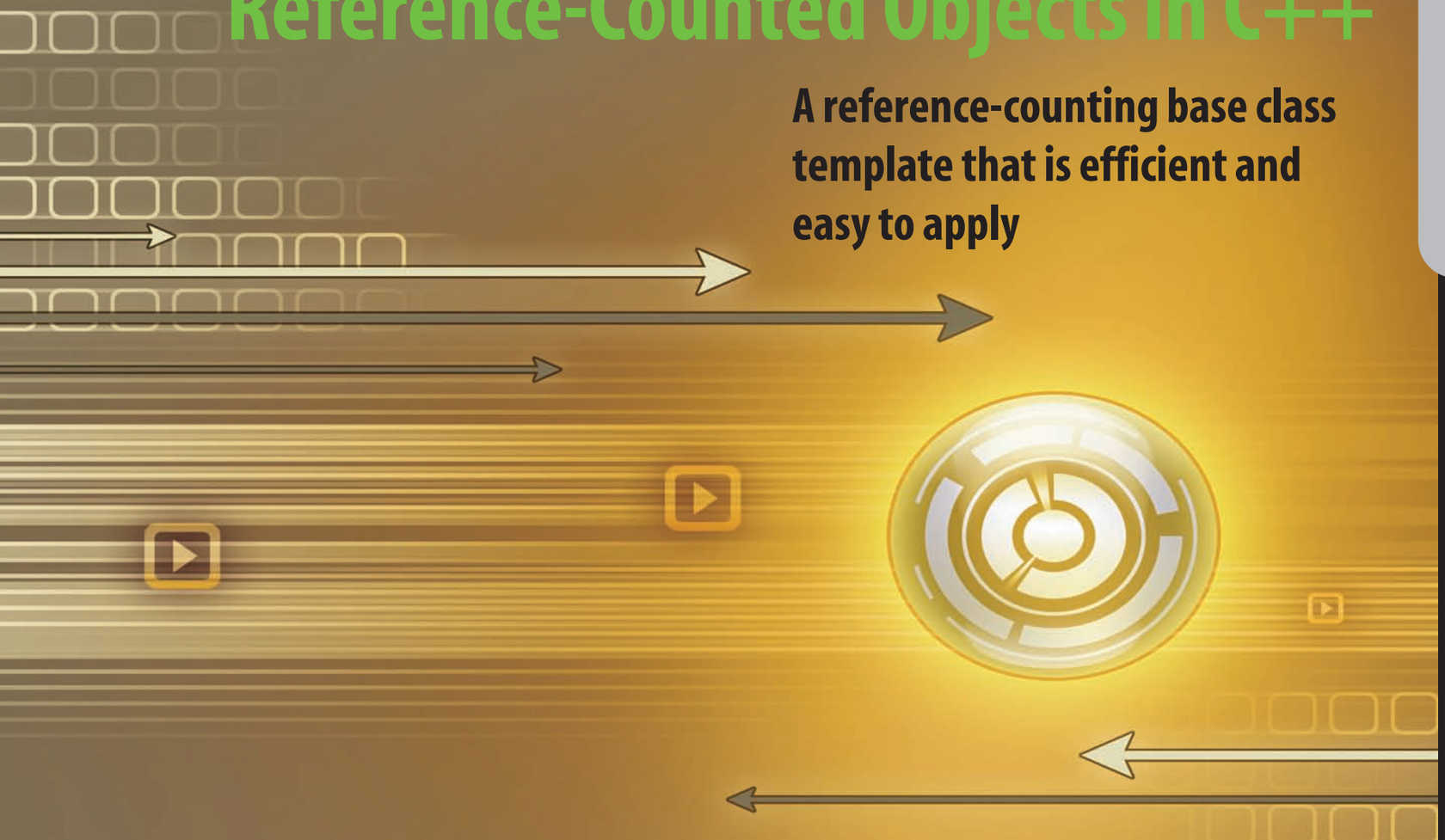
Dr. Dobb's Journal

MARCH 2011

Next

A Base Class for Intrusively Reference-Counted Objects in C++

A reference-counting base class template that is efficient and easy to apply



ALSO INSIDE

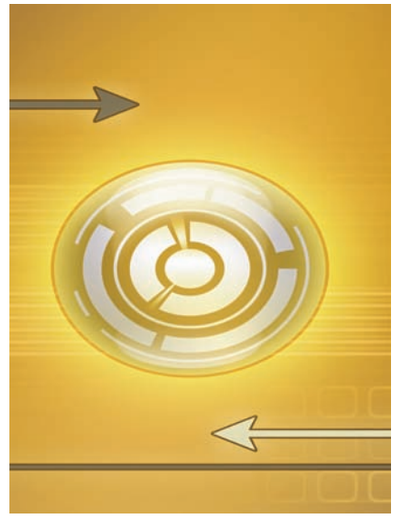
UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

SPECIAL ISSUE March 2011



COVER STORY

More on DrDobbs.com

Agile at 10: Evolving Again

Little by little, the core practices that underlay Agile principles have become part of how most developers approach their work

www.drdobbs.com/architecture-and-design/229204128

The Current Windows Ecosystem

For developers, new niches in Microsoft's new platforms

www.drdobbs.com/windows/229000855

Silverlight Developers, Take Note

Beta tool checks how an app runs across multiple threads

www.drdobbs.com/windows/229000838

Java EE is Dead

Or is it? Did I get your attention? Well, all of this talk about simplification has gotten mine, especially since it's been going on for years regarding Enterprise Java. For instance, Java EE has introduced features such as annotations and dependency injection to help with complex tasks such as transactions and database connectivity.

www.drdobbs.com/blog/archives/2011/02/java_ee_is_dead.html

Present Day Software is Unimaginative and Staid

I have been an independent consultant for more than 20 years now, which means I see a lot of projects and work with a lot of teams and companies. The teams, the companies, and the projects have all been great for the most part, but what I see most commonly are teams comprised of managers and programmers. The other thing I see is a lot of software that lacks design imagination, creativity, and looks a lot like avocado kitchen appliances from the 1970s.

www.drdobbs.com/blog/archives/2011/02/present_day_sof.html

Jolt Awards: Utilities Category

Balsamiq Mockups version 2.0.10, Camtasia, and VMware Workstation shine

http://www.drdobbs.com/joltawards/229200035

WebMatrix

Take the Blue Pill

Last week, Microsoft released its WebMatrix package http://www.microsoft.com/web/webmatrix/), a single download that provides most everything an SMB would need to build a small web site. Wizards, tools, database, web server, plus various open source packages all in one convenient download. The Redmond site avers: "It's all-inclusive, simple and best of all free. Developing websites has never been easier."

Other than the notion that it's free, I can find little to support these claims. I could do a long, detailed review of the product, but there is already excellent analysis (see http://www.infoworld.com/d/developer-world/microsoft-webmatrix-mixed-bag-170?source=IFWNLE_nlt_stradev_2011-01-25), and my intention is not to assess WebMatrix, but to examine the goal behind it.

The overarching aim of the product is to make it so simple to create websites that people with little technical knowledge can do it. And there's the rub. The intended goal is drawn across actual reality: Web development is hard enough that tools alone cannot make it easy. Microsoft's aim to simplify it to the point where non-professionals can do it follows in a long tradition of trying to push apparently simple tasks from the developer to the layperson.

The earliest instance of this quixotic pursuit was probably COBOL, which aimed to enable businessmen to write their own reports. It did this by having a built-in database (ISAM) and using a syntax that read like spoken English. ("add 3 to total. read next customer-record.") However, while some intrepid business people could manage to write a simple program, they frequently went over the cliff when logic became more than trivial. Reports consisting of simple lists were within their purview, but once multi-level control breaks were thrown in or first and last record issues appeared, the users discovered that the task was harder than it appeared and more effort than it was worth. If the business user did

not capitulate at this point, he certainly did when it came time to debug. Debugging is the rough shore on which the layman's programming aspirations invariably run aground.

What was true of COBOL in the '60s was true of report writers (in the '70s) and 4GLs (in the '80s and '90s). There then followed the Web revolution and the implausible idea that non-technologists could design their own websites. Sites in those days were largely static HTML so, on the surface, the claim seems reasonable. However, the proliferation of horrid websites demonstrated that even if writing HTML were possible for lay users, basic design was beyond their grasp. But in fact, most users could not write HTML, and relied on so-called "design tools." These worked until the user had to go into the code to fix something. Then, boom!, the layout would suddenly go haywire, the user would revert to the previous code, and never touch the site again. Thousands of such marooned sites live on today — a testament to the failed ambition of bringing users into the fold — even at its very lowest level.

"Whoever it is had better be really good — that is, demonstrably superior to all his forebears who failed at crossing this chasm — because WebMatrix, inexplicably, has virtually no debugging support"

Comes now WebMatrix with the same aim, but in a far more complex universe: Web sites today rely on HTML, CSS, scripting languages, databases, and the like. WebMatrix supports all these activities. So, given that users' skills are unlikely to have expanded to encompass all these technologies, who is the prospective user? Whoever it is had better be really good — that is, demonstrably superior to all his forebears who failed at crossing this chasm — because WebMatrix, inexplicably, has virtually no debugging support. A user will either get it right the first time or give up.

I was just online looking at the product's forums and sure enough, the majority of WebMatrix users are Web developers. To wit, "I am using Win7 64 Ultimate, IE9, Intel i7-870, ATI HD 5450, 8GB." Not exactly a guy in the backroom of an SMB. In the same thread, other respondents suggest workarounds that are based on running in VS 2010.

So, Web Matrix has already migrated from its intended audience back to the users who least need it — developers.

— Andrew Binstock is Executive Editor for Dr. Dobb's and can be contacted at alb@drdobbs.com.

A Base Class for Intrusively Reference-Counted Objects in C++

If you want to use Boost's *intrusive_ptr*, but have no reference counter at hand, here is a starting point

By Peter Weinert

This article presents a reference-counting base class template that is efficient and easy to apply for the smart pointer **intrusive_ptr**. Smart pointers encapsulate pointers to dynamically allocated resources such as heap memory or handles. They relieve you of explicit resource deallocation, therefore simplifying resource management, exception-safe programming, and the "Resource Acquisition Is Initialization" (RAII) idiom. Your tool box provides different smart pointer templates: **auto_ptr**, **unique_ptr**, **shared_ptr**, **weak_ptr**, or **intrusive_ptr**. This section sketches these template classes.

An Introduction to Common Smart Pointers

The **auto_ptr** class template provides a semantics of strict ownership of objects obtained via new ([C++03 §20.4.5]). Strict ownership means that no more than one **auto_ptr** object shall own the same object. The owned object is disposed using delete when the **auto_ptr** object itself is destroyed. Even though strict ownership precludes the **CopyConstructible** and the **CopyAssignable** requirements ([C++0x §20.2.1]), the copy constructor and the copy assignment operators are accessible. If an **auto_ptr** object is copied, ownership is transferred, thereby resetting the source. The copy and the source are not equivalent. This limits usability and may lead to incorrect and dangerous use. C++0x therefore deprecates **auto_ptr** and offers **unique_ptr** as an improved class template.

The **unique_ptr** class template provides a semantics of strict ownership of objects ([C++0x §20.9.9]). Contrary to **auto_ptr**, strict ownership is constrained by syntax. The copy constructor and the copy assignment operator are inaccessible. Instead, **unique_ptr** meets the **MoveConstructible** and **MoveAssignable** requirements ([C++0x §20.2.1]) and move semantics are enabled in order to transfer ownership. Therefore, a function can explicitly receive or return ownership. A template parameter accepts a client-supplied deleter that becomes part of the type. Disposal is indirected to this deleter. **Unique_ptr** can manage objects obtained using allocators other than **new**. Because of this additional flexibility, the memory footprint may be larger compared to **auto_ptr**.

The **shared_ptr** class template implements semantics of shared ownership ([C++0x §20.9.10.2], [**BoostSmartPtr**]). More than one **shared_ptr** object can own the same object, satisfying the **CopyConstructible** and the **CopyAssignable** requirements. A **shared_ptr** object typically maintains an internal reference counter. The **make_shared** and **allocate_shared** helper template functions may efficiently combine the allocation of the reference counter and the owned object, as most implementations allocate the reference counter dynamically. Overloaded constructors accept a client-supplied deleter as well, if disposal using delete is not appropriate. Thanks to clever type elimination, this deleter is not part of the type (contrary to **unique_ptr**). Once the reference counter drops to zero, the owned object is disposed.

Semantics of strict and shared ownership can lead to cycles (e.g. objects owning each other) where the owned objects will never be disposed. A **weak_ptr** object does not own the object and is used to break cycles of **shared_ptr** objects ([C++0x §20.9.10.3], [**BoostSmartPtr**]). As opposed to a raw pointer, **weak_ptr** observes its **shared_ptr** and therefore avoids dereferencing a dangling pointer.

The **intrusive_ptr** class template provides a semantics of shared ownership [**BoostIntrusivePtr**]. As this smart pointer uses intrusive reference counting, the owned object must provide the reference counter. **Intrusive_ptr** is not directly aware of this counter. You supply free functions, **intrusive_ptr_add_ref** and **intrusive_ptr_release**, which solely manipulate the counter. The constructors and the destructor of **intrusive_ptr** invoke these functions through unqualified calls with a pointer to the reference-counted object as an argument. The object is not directly disposed by **intrusive_ptr** but by **intrusive_ptr_release**. To sum up, some requirements must be addressed when using **intrusive_ptr**:

- Semantics of **intrusive_ptr_add_ref** and **intrusive_ptr_release**
- Disposal of the object
- Visibility of **intrusive_ptr_add_ref** and **intrusive_ptr_release**

A Naive Approach

I develop a base class template, **ref_counter**, that is efficient and easy to apply with **intrusive_ptr**, and guarantees disposal using delete. As often, the devil is in the details.

Let's start with a test case:

```
// test.cpp: A simple class foo using the base class ref_counter

#include <boost/intrusive_ptr.hpp>
#include "ref_counter.h"

namespace foo {
class foo: public intrusive::ref_counter {};
}

int main( )
{
     boost::intrusive_ptr<foo::foo> p1(new foo::foo());
}
and get it running:
// ref_counter.h: A naive definition
#ifndef INTRUSIVE_REF_COUNTER_H_INCLUDED
#define INTRUSIVE_REF_COUNTER_H_INCLUDED
#include <boost/assert.hpp>

namespace intrusive {
class ref_counter
{
public:
     friend void intrusive_ptr_add_ref(ref_counter* p)
     {
          BOOST_ASSERT(p);
          ++p->counter_;
     }
     friend void intrusive_ptr_release(ref_counter* p)
     {
```

```
            BOOST_ASSERT(p);
            if (--p->counter_ == 0)
                  delete p;
      }
protected:
      ref_counter(): counter_(0) {}
      virtual ~ref_counter() = 0 {}
private:
      unsigned long counter_;
};
}
#endif
```

The class is fully exception safe, meeting the strongest guarantee, the no-throw guarantee (**noexcept** in C++0x). But have the requirements I mentioned above been met?

- Semantics: The counter (counter always refers to **ref_counter::counter_**) is initialized with zero, **intrusive_ptr_add_ref** increments the counter, and **intrusive_ptr_release** decrements the counter. Both free functions are friends, as they have to manipulate the encapsulated counter. In addition to the compiler-declared copy constructor and copy assignment operator, they provide the public interface of **ref_counter**. As the class is intended to be used as a base class, it is abstract and the constructor and destructor are placed in the protected section.
- Disposal: The object is disposed if the counter drops to zero. As it is deleted using a pointer to the base class, the destructor is virtual.
- Visibility: The **intrusive_ptr_add_ref** and **intrusive_ptr_release** functions reside in the same namespace scope as the base class **ref_counter**. They do not pollute the global namespace. In fact, their names are not even visible during an ordinary lookup [C++03 §14.6.5-2], because they are in the lexical scope of **ref_counter**. Their names are found through argument-dependent lookup (ADL, Koenig lookup [C++03 §3.4.2-2]).

Getting It Right

Did you spot the bug? Do the two free functions solely manipulate the counter? Think about compiler-generated definitions in **ref_counter**:

```
p1 = new foo::foo(*p1); // copy construction of foo
```

This code allocates a new **foo**-object using the copy constructor of **foo**, which uses the implicitly defined copy constructor of its **ref_counter** base class ([C++03 §12.8-8]), which copies the non-zero counter. But as a new **foo**-object is constructed, the counter should start over with zero. Remember, only the two free functions should manipulate the counter. This happens in the assignment operator of **intrusive_ptr**, where the old foo object is detached and the new one is attached (indirectly calling **intrusive_ptr_release** and **intrusive_ptr_add_ref**) using the "copy construct a temporary and swap" idiom.

The copy assignment operator of **ref_counter** is another suspicious candidate:

```
*p1 = foo::foo(); // copy assignment of foo
```

The implicitly defined copy assignment operator performs memberwise assignment of its subobjects ([C++03] §12.8-13). Therefore, the implicitly defined copy assignment of **ref_counter** is called. It overwrites the left counter with the right (overwritten with zero in this case). Do not get confused: The **intrusive_ptr** object is not assigned to or changed and so is its pointer member, but the object pointed to is assigned to. The number of referencing pointers did not change. Therefore, the assignment operator of **ref_counter** must not alter the counter, but do nothing. Because **ref_counter** is an abstract base class, the missing explicit definitions are placed in the protected section:

```
// ref_counter.h: Adding copy constructor and assignment operator
class ref_counter
{
// …
protected:
     ref_counter(const ref_counter&) : counter_(0) {}
     ref_counter& operator=(const ref_counter&) { return *this; }
};
```

Always keep assignment and copy construction in mind, especially if the class is managing a resource. In addition, the free **std::swap** function works correctly with classes derived from **ref_counter**, such as in **std::swap(*p1, foo::foo())**. Swapping **ref_counter** does nothing, and a swap-member function looks like:

```
void ref_counter::swap(ref_counter&) {} // noexcept
```

Cv-qualifier

A smart pointer should operate on both constant and volatile objects. But the compiler complains about a line as innocent as:

```
boost::intrusive_ptr<const foo::foo> p1(new foo::foo());
```

The problem is that the two free functions expect a pointer to a non-**const ref_counter** object, because the counter has to be manipulated. The solution is straightforward:

```
// ref_counter.h: Adding support for constant objects
class ref_counter
{
// …
     friend void intrusive_ptr_add_ref(const ref_counter* p);
     friend void intrusive_ptr_release(const ref_counter* p);
private:
     mutable unsigned long counter_;
};
```

The two free functions accept a pointer to a **const ref_counter** object while the counter is mutable, so it can yet be manipulated. I leave the support for the volatile qualifier as an exercise.

Polymorphism

The **ref_counter** class has a virtual destructor. A virtual destructor forces objects to carry a pointer to a virtual method table along with them. If **intrusive_ptr_release** did not polymorphically delete the object using a pointer to the base class, but using a pointer to the derived class, the virtual destructor could be abandoned. The **intrusive_ptr** class template calls the two free functions with a pointer, whose type is parameterized. This type is the derived class, not the **ref_counter** base class. Perfect. This means that **intrusive_ptr_release** can take a pointer to the derived class, but therefore, it must know the type of the derived class. Coplien's "Curiously Recurring Template Pattern" (CRTP) [C++Templates 16.3] provides the solution to this dilemma: A class, **X**, is derived from a class template instantiation, base, and uses itself as the template argument, **class X: base<X>**. Here is the definition of the class template announcing the type of the derived class using CRTP, the free functions taking pointers to the derived class and therefore obviating deletion using a base class pointer:

```
// ref_counter.h: disposal using a pointer to the derived class
template <class Derived>
class ref_counter
{
     friend void intrusive_ptr_add_ref(const Derived* p);
     friend void intrusive_ptr_release(const Derived* p);
// …
protected:
     ~ref_counter() {};       // non-virtual
//~ref_counter() = default; // in C++0x
};
```

The class definition of foo has to pass its type to the base class template **ref_counter**:

```
class foo: public intrusive::ref_counter<foo> {};
```

Meyers uses CRTP to provide a static counter for different derived types [Counting]. Here, I use CRTP in combination with the "friend name injection" (a component of the Barton-Nackman Trick) [C++Templates 9.2.2 and 11.7] to inject the two free non-template functions into namespace scope, accepting pointers to the derived class instead of pointers to the base class. The **ref_counter** class template does not need a virtual destructor anymore. Of course the derived class may need one.

Enabling Private Inheritance

A derived **foo** class has no conceptual relationship to **ref_counter**, there is no reason for **ref_counter** to add to **foo**'s interface. Instead, **foo** uses the generic reference-counting implementation of **ref_counter**, it is implemented in terms of **ref_counter**. This is expressed with private inheritance, while public inheritance supplies an interface, not an implementation [EffectiveC++ item 42].

A client could gather a pointer to the base class with an implicit upcast. Is that sensible? Or is it even a violation of the Liskov Substitution Principle [LSP]? Clearly, **foo** should not be used as a **ref_counter**, **foo** IS-NOT-A **ref_counter**. So, it should not be possible to provide a **foo** where a **ref_counter** is expected. Class design should prevent the user from misuse. Prohibiting the implicit conversion from the derived class, **foo**, to the base class, **ref_counter**, can be done by private inheritance:

```
class foo: intrusive::ref_counter<foo> {};
```

Now the compiler complains of an inaccessable conversion from a pointer to the derived class to a pointer to the base class. The problem is that the two free functions have to access the counter in the base class through a pointer to the derived class. An implicit conversion and a **static_cast** from a pointer to the derived class to a pointer to the inaccessible base class both are ill-formed. Obviously, **dynamic_cast** (**downcast**) or **const_cast** (**cv-qualifier** conversion) won't do the trick. A **reinterpret_cast** is not only dangerous and implementation dependent, but it will fail in multiple inheritance. This leaves us with the dangerous explicit C-style **cast**. It will work even if the base class type is not accessible [C++03 §5.4-7 and §11.2-3], but the class type must be complete — otherwise, the explicit C-style **cast** may be interpreted as a **reinterpret_cast** instead of a **static_cast**. By the way, the **delete** expression also expects a complete class type if the class has a non-trivial destructor [C++03 §5.3.5-5]. **ref_counter** is complete, as within the class member specification, the class is regarded as complete within function bodies [C++03 §9.2-2]. Declarations inside the base class template are instantiated when the derived class is declared, but their bodies are instantiated after the full declaration or instantiation of the derived class, so **Derived** is complete [TMP 9.8]. Therefore, inside the friend function bodies, both object-types are completely defined. The C-style **cast** and the **delete** expression are safe and well defined:

```
// ref_counter.h: enabling private inheritance
template <class Derived>
class ref_counter
{
     friend void intrusive_ptr_add_ref(const Derived* p)
     {
          BOOST_ASSERT(p);
          ++((const ref_counter*) p)->counter_;
     }
     friend void intrusive_ptr_release(const Derived* p)
     {
```

```
            BOOST_ASSERT(p);
            if (--((const ref_counter*) p)->counter_ == 0)
                delete p;
        }
//…
};
```

Private inheritance is enabled and so is the **foo** IS-NOT-A **ref_counter** property.

Concurrency

Thread-safety is a complex mission. In C++0x, it is easy to make the counter atomic. Just include the **<atomic>** header and have the counter a type of **std::atomic_size_t**. The increment and decrement operations are atomic now. Now, the **ref_counter** class template offers the same level of thread safety as built-in types. Simultaneous reads are fine, but beware of other access patterns, for example:

```
// shared global
intrusive_ptr<foo> ps(new foo());

// Thread 1
intrusive_ptr<foo> p1(ps);      // read

// Thread 2
ps.reset();  // write
```

This is a data race that breaks invariances, even though the counter is atomic. The problem is that the **intrusive_ptr** operations are not atomic as a whole. Here is the copy constructor of **intrusive_ptr**:

```
intrusive_ptr( intrusive_ptr const & rhs ): px( rhs.px )
{
    if( px != 0 ) intrusive_ptr_add_ref( px );
}
```

Note that the pointer and the counter are not updated atomically, so, an invalid state can be observed. Just assume the following interleave:

- Thread 1 calls the copy constructor and performs the copy of the pointer. The invariants are broken, as the pointer was copied and the counter is not yet incremented.
- Thread 2 resets ps. This operation decrements the counter, which drops to zero, and thus the owned object is disposed.
- Thread 1 completes the copy construction, uses the dangling pointer, and accesses the counter. Bang! undefined behavior based on a data race.

The race cannot be avoided in **ref_counter**, but rather in **intrusive_ptr**. [AtomicRCP] provides a lock-free non-portable non-C++0x implementation for PowerPC. Boost's **shared_ptr** uses a spin-lock pool, **intrusive_ptr** can easily use a similar implementation. **shared_ptr** provides operations like **atomic_load**, **atomic_store**, **atomic_exchange**, or **atomic_compare_exchange** ([C++0x 20.9.10.5, N2674]). The interface of **intrusive_ptr** should offer these operations. A compatible interface has the advantage of easy refactoring between **intrusive_ptr** and **shared_ptr**.

Customized Disposal

The disposal of the object can be hard coded or customizable. The customization can be carried in the type (like in **unique_ptr**) or not (using type elimination like in **shared_ptr**). When it comes to intrusive reference counting, disposal gets intrusive as well. The interface of the class must expose customizable disposal or should include creation (and loosing flexibility), otherwise creation and disposal are not adhered. A clear architectural flaw. It is your choice to leave hard-coded disposal, add custom disposal in the type (maybe as in **unique_ptr** [C++0x 20.9.9.1] or use CRTP and empty-base-optimization for disposal), or use type elimination (using the function class template).

Alternative Implementations

A base class is not the only possibility to develop a reusable reference counter. If a reference-counting class is added as a class member, it would necessitate some member functions to reach the counter and the two free functions had to be delivered separately. This is too demanding. Another possibility is to add reference counting by deriving from the **foo** class:

```
template <class Base>
class add_ref_counter : public Base {/*…*/};

and to use it like this:
intrusive_ptr<add_ref_counter<foo> >
  p(new add_ref_counter<foo>());
```

A generic implementation of **add_ref_counter** would make use of C++0x variadic templates and perfect forwarding. The downside of such a technique would be a broken upcast in a class hierarchy, because it adds **add_ref_counter-leaves**. The **intrusive_ptr** object holds a pointer to a **add_ref_counter<foo>** object, but not to a **foo** object. Although a **foo_child** may be used as a **foo**, an **add_ref_counter<foo_child>** cannot be used as an **add_ref_counter<foo>**. On the other hand, this technique makes it possible to add reference counting and customized disposal to classes afterwards.

Results

Intrusive reference counting may be an efficient way to manage shared resources. I presented a way to implement a reusable and easy to apply base class. However, intrusive reference counting has its downsides. Its inflexible requirements force the managed object to hold the reference counter and to know something about disposal. This is a clear flaw, as it separates creation and disposal. Intrusive reference counting can be used only with classes designed this way. Finally, the interface of **intrusive_ptr** must be extended to provide thread safety.

I prefer **shared_ptr<X> px(CreateX(), DestroyX)**, adhering creation with disposal. **X** does not have to know anything about **shared_ptr**, creation or disposal. Additionally, the interface of **shared_ptr** features thread-safe functions. Finally, **shared_ptr** is part of the C++0x standard library. I recommend using **shared_ptr** for shared ownership and **unique_ptr** for strict ownership. Only if you can prove that **intrusive_ptr** solves problems that you have with **shared_ptr** should you revert to **intrusive_ptr**. Here is the definition of the **ref_counter** class template for single-threaded in C++03:

```
template <class Derived>
class ref_counter
{
public:
     friend void intrusive_ptr_add_ref(const Derived* p)
     {
          ++((const ref_counter*) p)->counter_;
     }
     friend void intrusive_ptr_release(const Derived* p)
     {
          if (--((const ref_counter*) p)->counter_ == 0)
               delete p;
     }
protected:
     ref_counter(): counter_(0) {}
     ref_counter(const ref_counter&) : counter_(0) {}
     ref_counter& operator=(const ref_counter&) { return *this;
}
     ~ref_counter() {};
     void swap(ref_counter&) {};
private:
     mutable unsigned long counter_;
};
```

References

[C++03] ISO/IEC 14882:2003: "Programming Language C++," 2003, Wiley.

[C++0x] ISO/IEC Working Draft: "Standard for Programming Language C++," 2010, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3225.pdf.

[**BoostSmartPtr**] Colvin, Dawes, Adler: "Boost smart pointers", http://www.boost.org/doc/libs/1_45_0/libs/smart_ptr/smart_ptr.htm.

[**BoostIntrusive**] Dimov: "Boost **intrusive_ptr** class template", http://www.boost.org/doc/libs/1_45_0/libs/smart_ptr/intrusive_ptr.html.

[C++Templates] Vandevoorde, Josuttis: "C++ Templates – The Complete Guide," 2003, Addison-Wesley Professional.

[Counting] Meyers: "Counting Objects in C++," 1998, http://www.drdobbs.com/184403484.

[EffectiveC++] Meyers: "Effective C++," 2nd ed., 1997, Addison-Wesley Professional.

[LSP] Martin: "The Liskov Substitution Principle," 1996, http://www.objectmentor.com/resources/articles/lsp.pdf.

[TMP] Abrahams, Gurtovoy: "C++ Template Metaprogramming," 2005, Addison-Wesley Professional.

[AtomicRCP] Reinholtz: "Atomic Reference Counting Pointers," 2004, http://www.drdobbs.com/184401888.

[N2674] Dimov, Dawes: "**Shared_ptr** atomic access," revision 1, 2008, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2674.htm.

— Peter Weinert is a scientific staff member of the Leibniz Supercomputing Centre (www.lrz.de) in Munich, Germany.

Programming Python, 4th Edition Review

Python has grown into a language capable of running sophisticated and flexible applications

By Mike Riley

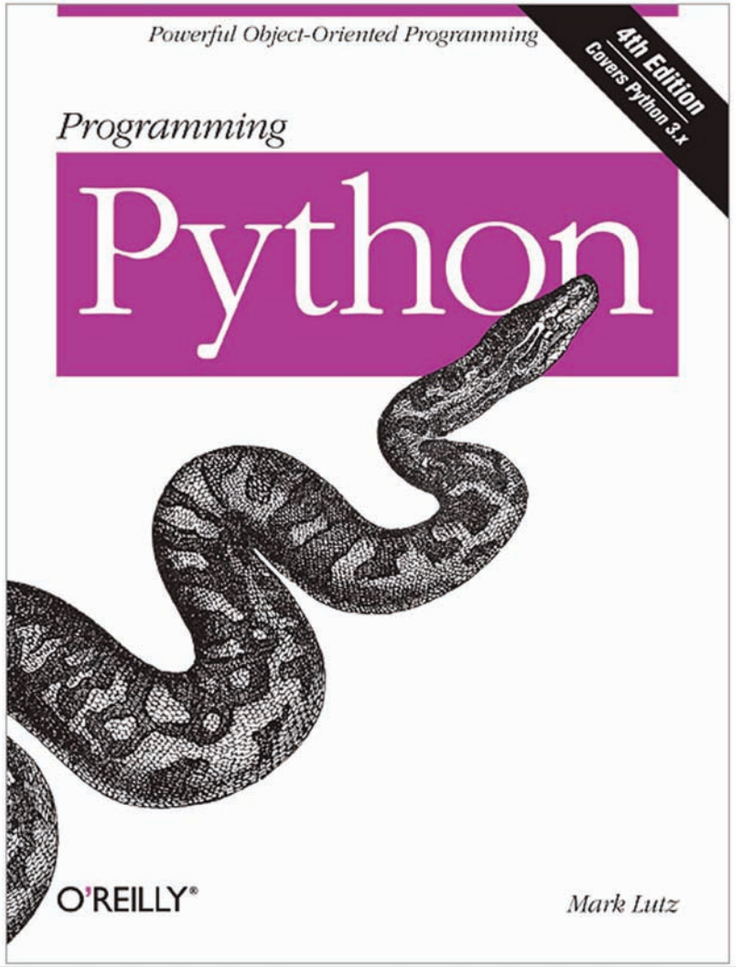
I have had the pleasure of reading author Mark Lutz's work for nearly 15 years, starting with his first edition of *Programming Python* way back in 1996. Since then, Python has grown from a Perl alternative to a language capable of running some of the most sophisticated and flexible applications for global companies like Google. How does this 4th release hold up to the previous books it was built upon?

This latest edition begins with a quick summary of what it has been built upon as well as the bold assertion that it only focuses on Python 3 (specifically 3.1, though the code examples work with the latest 3.2 release as well), leaving the Python 1.x and 2.x legacies behind. For those developers not yet ready to make a break for the 3.x world, stick with the 3rd edition.

The book is divided into six parts, starting with the aptly named "Part 1. The Beginning." It's also the shortest part of the book, just over 65 pages long, giving a grand overview of the book's contents. "Part 2, System Programming," explores system tools (the os and sys modules), script execution context, file tools, and threads. The third part focuses a large portion of the book on GUI programming via the Tk/tkinter framework. Part 4 covers Internet programming from the socket level and various libraries (FTP, email, HTTP and NNTP) to server-side scripting via raw CGI as opposed to spending much time with purpose-built web frameworks like Turbogears and Django. Part 5 delves into tools and techniques, covering databases via pickled objects, ZODB OODB, SQL and ORMs as well as data structures (sets, search trees, sorting, etc.), text and language via string methods, regular expressions, and HTML/XML parsing.

A chapter covering the Python/C and C++ connection via the SWIG integration code generator also includes a section on embedding Python in C and the issues that need to be considered when doing so.

The book concludes with its sixth and final section called, appropriately enough, "The End." And for those who have spent the time reading the preceding 1,500 pages of the book, Mark shares the meaning of life (keeping in character of paying homage to Python's fixation with all things *Monty Python*).



I also appreciated his brief statement that "something's wrong with the way we program computers," and the wonderful "Gilligan Factor." Indeed, reading the book's final chapter was a nice reward coupled with some especially insightful commentaries on the current state of application development in general.

Even with the emphasis on Python 3, the book continues to clock in at over 1,600 information-packed pages, leaving no standard Python library or related technology unturned. Considering that the previous edition was roughly the same page count, the exclusion of older, deprecated Python explorations is significant. Also making its appearance for the first time in the book is the recognition of Django and Jython; however, Mark smartly abdicates any deep discussion of these technologies to other books. They merely make a limited appearance as a way to recognize their contributions to the Python ecosystem.

For those interested in having a solid foundation Python and are untethered by the past, this edition is a must-have title on your technical bookshelf. For those experienced Pythonistas who own previous editions and are ready to either move their legacy code forward into the new Python 3 practices, Mark shows the way.

Given the significant differences between the 3.x and older versions of Python, there remain enough similarities that a large majority of Mark's code has made the jump to this latest edition. And while he continues to rely on Tk for his Python GUI foundation of choice (a good one considering that many others have yet to fully support Python 3.x), the examples are timeless and do the job of advertising the powerful punch that Python provides even after all these years.

Title: Programming Python, 4th Edition
Author: Mark Lutz
Publisher: O'Reilly Media
ISBN: 978-1-4493-0183-5
Pages: 1,632
Price: $51.99 (Ebook), $64.99 (Print)
URL: http://oreilly.com/catalog/9780596158118/

—Mike Riley is a Dr. Dobb's Contributing Editor.

Improving OLTP Database Throughput via Batching

How to improve throughput for On-line Transaction Processing (OLTP) database applications without introducing the overhead of contention between threads

by John Lane

Parallel threads and JDBC connection pools can be useful, but should not be used blindly. When handling numerous transactions with small overhead on heavily contended systems, batch processing often provides a better solution.

Order Management System

The case history is simple enough. The system consisted of several Java-based order management servers that held details of orders for financial instruments. The orders themselves were made up of more than 100 attributes, including product type, order size, bid and ask prices, etc., and they would frequently be updated after they were placed.

To allow for the database being unavailable, the servers had their own local persistent storage, to which they would initially save order details. Periodically, they would send new or updated order details to the database in bulk. Because the order management system was Java based, the updates were implemented as stored procedure calls via JConnect (JDBC) to a Sybase Adaptive Server Enterprise (ASE) database server.

The system developers decided that to maximize throughput, they would use 20 parallel threads, with a JDBC connection pool to access the database. The database server utilized three of the available single-core processors, but the system suffered from a high degree of contention and deadlocking. In fact, the system was designed with a backing-off algorithm such that the time between deadlock retries would be increased with each failure, until it eventually gave up and waited for an hour before repeating the process.

Inevitably, the order management servers would end up deadlocking with themselves over different pool connections, and thus updates would often fail 100 times. As a result, orders frequently took a considerable length of time to be persisted to the database. What went wrong?

Note that while this article uses this particular system as a case study, it could equally be applied to any On-line Transaction Processing (OLTP) database application.

Query Lifecycle

Several steps occur during the life of a single database query. These will vary, depending on your particular RDBMS, but will generally include:

1. Initialization of the client library (e.g. JDBC) structures
2. Sending the query to the server, using a vendor-specific protocol over some communication mechanism, e.g. Tabular Data Stream (TDS) over TCP/IP
3. Parsing and compilation of the query
4. Executing the query
5. Returning the query results to the client

If we were to construct a rough time line of the query, it would look something like that shown in Figure 2.

When a query executes on a server, it continues to run until either it has used up its allotted time slice (and is swapped out for another query), or a condition occurs that prevents it from continuing. There are many such conditions, but some of the more significant ones are as follows:

- A lock cannot be obtained because another connection is already holding it
- A page is not in the cache and must be read from disc
- A write to disk is required

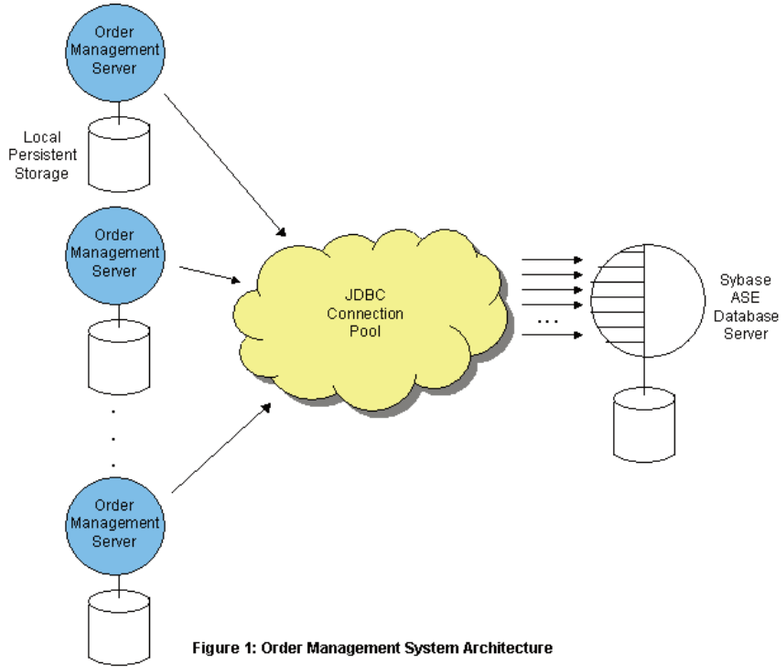


Figure 1: Order Management System Architecture

At the point where one of these conditions occurs, the query will be put to sleep until the condition has been resolved. For example, if a query has been put to sleep because it was waiting for a lock, it will resume once the lock has been obtained.

From the perspective of a single thread, the time spent sleeping — waiting for a lock or I/O — is "dead time." It follows that if we want the query to execute faster, we should reduce the length of time that each step takes to complete or minimize the amount of dead time.

In reality, of course, ours may not be the only query running on the server. So dead time for our query is time that the server can use to do something else, such as executing another query. Indeed,

for many different execution threads, and systems as a whole, we tend to think more about overall throughput of the system rather than response time for individual queries. It would seem that the next logical step, therefore, is to make use of the multithreaded nature of servers by using multiple connections — in effect, "filling in" the dead time with other queries.

We must remember, however, that running multiple threads is only useful if there is indeed some dead time. Even with a low ratio of threads to processor cores, there can be sufficient work for the server to fill in all the available time. Further connections will only result in the queries being queued up, and the server experiencing increased contention for resources. This is important to bear in mind with systems that are heavily CPU bound, because there will be little natural dead time for other queries to make use of. This is what happened on our system: The developers of the order management system failed to take into consideration that the ratio of 20 threads to 3 processor cores was far too high on a CPU-bound database server, and it contributed to their high contention and deadlocking problems. So how can we improve throughput without introducing the overhead of contention between threads?

Batching

The answer is to use batching. I'm sure that most of you will have heard of batching, but it may not be immediately apparent exactly why batching works, nor how much of an improvement can be obtained from using it. In database terms, a batch is a single string or statement sent to the server. If you are using ISQL with a Sybase or SQL Server database, the text before you type "go" is a single batch. If you are writing application code, multiple queries can be concatenated together into a single string (separated by white space) and sent as a single batch.

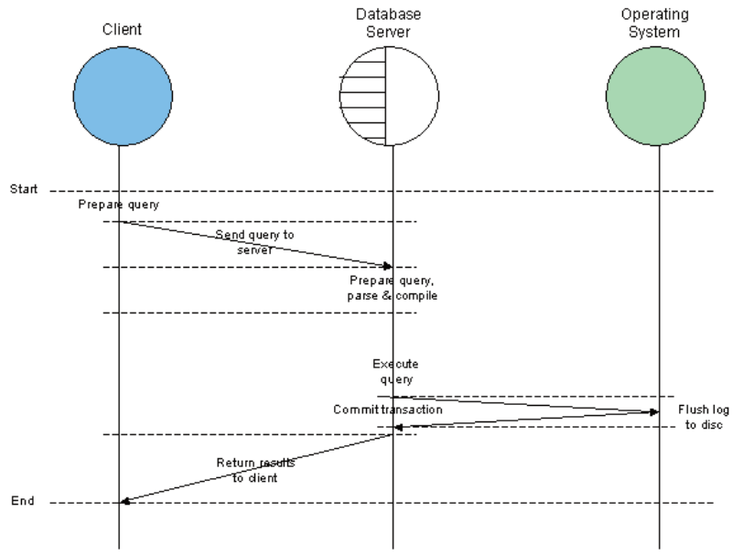


Figure 2: Order Update Timeline

On an abstract level, a batch of work usually consists of a fixed cost or overhead and a marginal cost. The overhead is the fixed cost of performing the batch no matter how large the batch, whilst the marginal cost is the additional cost per extra element of work.

The various costs include these factors:

- The overhead in sending network packets can be large compared to the marginal cost of increasing the size of a packet
- An acknowledgement packet is probably the same size no matter how large the original packet was

- The time taken to set up and start parsing or compilation can be great compared to the time taken to parse or compile each additional query

For OLTP systems, when the overheads are comparable to the execution times for individual queries, they can become bottlenecks themselves.

So the reason that database batching works is simple: When the fixed cost of a batch is significant compared with the marginal cost of a query, combining many queries into a single batch effectively shares the overhead between them. The more queries in a batch, the smaller the overhead per query.

Results in Practice

To illustrate the savings, I created a test system consisting of 500,000 rows of pseudo-orders, each having a unique identifier and 10 further attributes. I wrote a simple stored procedure that updated orders and called it multiple times, using random values. To remove the effects of caching, the data cache was sized to be larger than the order database, and was "warmed" beforehand by running a set of queries to ensure the whole database was held in memory.

The results of running the test with different numbers of updates in the same batch are shown in Table 1.

As you can see from the table, the throughput increases dramatically when we move from a batch size of 1 to a batch size of 5. This improvement continues as we increase the batch size, although by a smaller amount each time.

I should point out that this behavior is only useful for true OLTP queries — those queries that require a small amount of work (network communication, CPU cycles, I/O etc.) to process. Only in these types of queries are the overheads a significant proportion of the overall cost. If we are running queries that take many seconds to complete, batching them will yield little noticeable gain.

Batch Size	Throughput (updates / second)
1	361
5	545
10	597
20	622
50	635
100	644

Table 1: Throughput for various batch sizes

In contrast to batching, if we use a purely multithreaded approach, the fixed costs are multiplied: Every query carries the same overhead.

Practical Considerations

The actual implementation of batching will depend on your development platform. Using Java and with JDBC to connect to the database, you can use the batching facility provided by JDBC to implement this easily. There is one very important caveat, however: You must verify that the batching functionality is properly implemented in your JDBC driver/database server. Early versions of some JDBC drivers, for instance, did not actually batch queries into a single string — they were functionally equivalent to sending individual queries.

If your version of JDBC, or particular JDBC driver, does not batch correctly, or you are using connection libraries that do not support batching, then you will need to implement this yourself. The way to do this is to simply to build up a single string, concatenating all the queries together.

One important point to mention is that, as yet, we have not changed the transaction granularity — each query is still in a separate transaction. If we look back at the timeline shown in Figure 2, we can see that the commit portion of the update itself adds an overhead to the execution time because we must wait for the log to be flushed to non-volatile storage. We can take our batching one step further, and share this overhead between queries by combining them into single transactions.

Indeed, in the absence of contention with other connections, this will have a further dramatic effect on throughput. Care should be taken, however, to balance the transaction granularity with the increased contention that will result: Locks will be held for longer, and there will be an increased risk of deadlocks.

If we use query batching but maintain single-update transaction granularity, we get the best of both worlds: Efficiency is improved without a corresponding increase in contention.

Conclusion

The lead developer of the order management system was quite horrified when I suggested dropping connection pools and moving to a single-threaded model using batching. He wouldn't even consider reducing the number of threads in an attempt to reduce the contention issues. The horrendous number of deadlocks wasn't enough, it would seem, to convince him.

Had he done so, he would have seen his database contention all but disappear, and may have been quite surprised exactly how much of an improvement he obtained. While the results on my test system are very modest, I have often observed quite a dramatic increase in throughput with real systems: One particular application improved its throughput by over 36 times, simply by implementing batch sizes of 500 — and there wasn't a single connection pool in sight!

Your results will (of course) vary, depending on where the bottlenecks lie in your particular system, but my suggestion is to use a two-pronged approach to increasing throughput:

- Batching queries is efficient because it shares (and thus minimizes) a number of overheads; creating parallel threads multiplies them. Always try batching your OLTP queries before you implement multiple threads, and tune the batch, transaction, and network packet sizes according to your application requirements.
- Use multiple threads of execution only after you have exhausted the improvement that you get from batching queries. Make sure your throughput is actually increasing by means of testing. Remember: Less is more. Start with a low ratio of threads to processor cores, and increase the number gradually — monitoring the throughput until you reach a peak.

Combining the two approaches can yield significant performance improvements.

— John Lane is a freelance consultant specializing in high-performance systems, developing application code, optimized databases, algorithms, and protocols. He can be contacted through his website at www.sturnus.co.uk/performance/.

IN THIS ISSUE

Dr.Dobb's

Jon Erickson Editor in Chief, Dr. Dobb's
jerickson@drdobbs.com

Andrew Binstock Executive Editor, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

CONTRIBUTING EDITORS

Mike Riley

Herb Sutter

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

Dr.Dobb's Business Contacts

DR. DOBB'S

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600-3106, sguttman@techweb.com

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

CMO, Scott Vaughan
(949) 223-3662, svaughan@techweb.com

VP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Sales Assistant, Group Sales, Kelly Glass
(212) 600-3327, kglass@techweb.com

Publisher's Assistant, Esther Rodriguez
(949) 223-3656, erodriguez@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Western Regional Director, Matt Stovall
(415) 947-6245, mstovall@techweb.com

District Sales Manager, Rachel Calderon
(516) 562-5338, rcalderon@techweb.com

Account Manager, Alison Rubino
(415) 947-6248, arubino@techweb.com

Inside Sales Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Sales Assistant, Ian Doyle
(415) 947-6105, idoyle@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Account Manager, Shoshana Freisinger
(415) 947-6349, sfreisinger@techweb.com

Sales Assistant, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Jenny Hanna
(516) 562-5116, jhanna@techweb.com

District Manager, Michael Greenhut
(516) 562-5044, mgreenhut@techweb.com

Account Manager, Cori Gordon
(516) 562-5181, cgordon@techweb.com

Inside Sales Manager East, Ray Capitelli
(212) 600-3045, rcapitelli@techweb.com

Sales Assistant, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

SALES CONTACTS—NATIONAL

Global CIO, InformationWeek 500 Conference

National Account Manager, Amy Neidlinger
(212) 600-3163, aneidlinger@techweb.com

Dr. Dobb's

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600-3106, sguttman@techweb.com

SALES CONTACTS—EVENTS

Director, Event Operations, Jennifer Russo
(516) 562-5094, jrusso@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Director of Marketing, Sherbrooke Balser
(949) 223-3605, sbalser@techweb.com

Marketing Manager, Monique Luttrell
(949) 223-3609, mluttrell@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000

UBM TECHWEB

Tony L. Uphoff CEO

John Dennehy CFO

David Michael CIO

Bob Evans Sr. VP and Global CIO Director

Joseph Braue Sr. VP, Light Reading Communications Network

Scott Vaughan CMO

Ed Grossman Executive Vice President, InformationWeek Business Technology Network

John Ecke VP and Group Publisher, Financial Technology Network, InformationWeek Government, InformationWeek Healthcare

Martha Schwartz VP, Group Sales, InformationWeek Business Technology Network

Beth Rivera Senior VP, Human Resources

David Berlind Chief Content Officer, TechWeb, and Editor in Chief, TechWeb.com

Fritz Nelson VP, Editorial Director, InformationWeek Business Technology Network, and Executive Producer, TechWeb TV

UNITED BUSINESS MEDIA LLC

Pat Nohilly Sr. VP, Strategic Development and Business Admin.

Marie Myers Sr. VP, Manufacturing