Stephen C. Dewhurst is the president of Semantics Consulting, Inc., located in southeastern Massachusetts. He specializes in C++ and object-oriented consulting and mentoring, and training in advanced C++ programming, STL, and design patterns. He may be contacted at www.semantics.org.

COMMON KNOWLEDGE

Don't Ask, Don't Tell

MY "BEST FRIEND and severest critic" has long suspected that we software types are socially inept and in basic need of more excitement. I think this started more than a dozen years ago while I was involved in a long explanation of Unix processes. When I got to the part about parent processes killing the children and becoming zombies, the seed of the suspicion was planted. She's too careful a scientist to draw conclusions from a small sample, but due to the restrictions she's placed on after-hours socialization with other software types, it's unlikely the sample size is going to get much larger.

Lately, she's also begun to suspect that we're blithering idiots as well. I'm frequently involved in client phone consultations in which my half of the conversation goes something like: "No, an employee is not a person, an employee is a asset, and has a role. A person has a address." I've often turned around in mid-conversation to find my critic in the office doorway with an eloquent "you get paid for this?" raised eyebrow.

Well, we do actually get paid for this kind of analysis, if we're doing our jobs correctly. One of the advantages of object-oriented design is that it permits clear, bidirectional translation between the problem space and the solution space (in our case, C++ source code). If the problem is simple, the solution should be as simple, or simpler. One indicator of a good design then, is that it is usually possible to describe a use case in the language of the OO analysis and have the description correspond to correct, rational action in the problem domain. We'll make use of this approach here.

This column treats an issue of current importance in OO design: runtime type information. The issue is of current importance because the C++ standard has standardized the form of runtime type queries, effectively legitimizing their use with an implicit seal of approval. While there are legitimate uses of runtime type queries in C++ programming, we'll see that these uses should be rare, and should almost never form the basis for a design. Tragically, much of the tried and true wisdom that the C++ community has accumulated over the past decade and a half about proper and effective communication with hierarchies of types is often jettisoned in favor of underdesigned, overgeneral, complex, unmaintainable, and error-prone approaches using runtime type queries.

NO PERSONNEL QUESTIONS Consider the venerable employee hierarchy in Figure 1. It's often the case that features must be added after a significantly large subsystem has been developed and tested. For instance, there is a glaring omission from the Employee base class interface.

```
class Employee {
  public:
    Employee( const Name &name,
        const Address &address );
    ~Employee();
    void adoptRole( Role *newRole );
    const Role *getRole( int ) const;
    //...
};
```

That's right. We have to be able to right-size these assets. (We also have to pay these assets, but that can wait until a future release.) Our management tells us to add the capability to fire an employee given only a pointer to the employee base class, and without recompiling or otherwise changing the Employee hierarchy. Clearly, salaried employees must be fired differently from hourly employees.

```
void terminate( Employee * );
void terminate( SalaryEmployee * );
void terminate( HourlyEmployee * );
```

The most straightforward way to accomplish this is to hack:

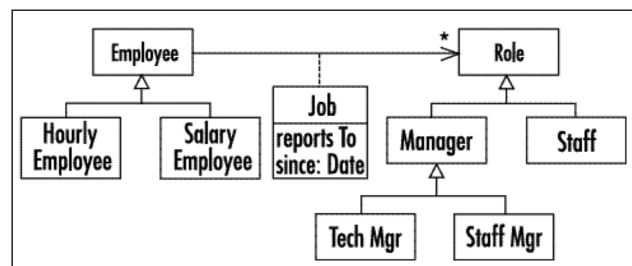

Figure 1. Employee hierarchy.

```
void terminate( Employee *e ) {
    if( HourlyEmployee *h =
      dynamic_cast<HourlyEmployee *>(e) )
        terminate( h );
    else if( SalaryEmployee *s =
      dynamic_cast<SalaryEmployee *>(e) )
        terminate( s );
    else
        throw UnknownEmployeeType( e );
}
```

The syntactic elegance of C with the efficiency of Smalltalk. There are clear problems with this approach in terms of efficiency and the potential for runtime error in the case of an unknown employee type. Generally, because C++ is a statically typed language, and because C++'s dynamic binding mechanism (the virtual function) is statically checked, we should be able to avoid this class of runtime errors entirely. This is reason enough to recognize the terminate function as a temporary hack, rather than the basis of an extensible design.

The poverty of the design is perhaps even more obvious if the code is back-translated into the problem domain it is supposedly modeling:

The vice president of widgets storms into her office in a terrible rage. Her parking space has been occupied for the third time this month by the junk heap of that itinerant developer she hired the month before. "Get Dewhurst in here!" she roars into her intercom. Seconds later, she fixes the hapless developer with a gimlet eye and intones, "If you are an hourly employee, you are fired as an hourly employee. Otherwise, if you are a salaried employee, you are fired as a salaried employee. Otherwise, get out of my office and become someone else's problem."

I'm a consultant, and I've never lost a contract to a manager who used runtime type information to solve her problems. The correct solution, of course, is to put the appropriate operations in the Employee base class, and use standard, type-safe, dynamic binding to resolve type-based questions at runtime:

```
class Employee {
  public:
    Employee( const Name &name,
        const Address &address );
    ~Employee();
    void adoptRole( Role *newRole );
    const Role *getRole( int ) const;
    virtual bool isPayday() const = 0;
    virtual void pay() = 0;
    virtual void terminate() = 0;
    //...
};
```

… she fixes the hapless developer with a gimlet eye and intones, "You're fired!"

SECURITIES AT ANY COST In fact, abuse of runtime type information as obvious as that in the terminate function is usually the result of compounded hacks and poor project management rather than bad design. However, some "advanced" uses of dynamic casting with multiple inheritance are often pressed into service to form the basis of an architecture.

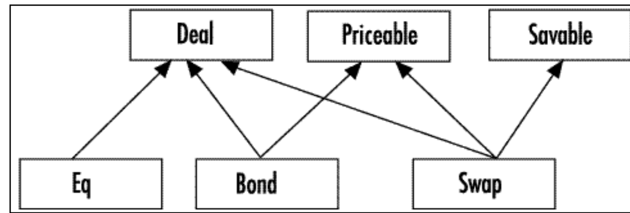

Figure 2. Multiple inheritance.

Suppose we're trading derivative securities (that is, financial instruments that allow you to leverage your potential losses enough to endanger all of civilization). We have at our disposal a pricing and a persistence subsystem whose code we would like to leverage in the implementation of our financial security hierarchy. The requirements of each subsystem are clearly stated in an interface class from which the user of the subsystem must derive.

```
class Savable { // persistent interface
  public:
    virtual ~Savable();
    virtual void save() = 0;
    // ...
};
class Priceable { // pricing interface
  public:
    virtual ~Priceable();
    virtual void price() = 0;
    // ...
};
```

Some concrete classes of the Deal hierarchy fulfill the subsystem contracts and leverage the subsystem code (see Fig. 2). This is a standard and effective use of multiple inheritance.

```
class Deal {
  public:
    virtual void validate() = 0;
    //...
};
class Bond
  : public Deal, public Priceable
    {/*...*/};
class Swap
  : public Deal, public Priceable,
    public Savable
    {/*...*/};
```

Now we have to add the ability to "process" a deal, given just a pointer to the Deal base class. A naive approach would simply ask straightforward questions about the object's type, which is no better than our first attempt to terminate employees.

```
void processDeal( Deal *d ) {
    d->validate();
    if( Bond *b =
        dynamic_cast<Bond *>(d) )
        b->price();
    else if( Swap *s =
        dynamic_cast<Swap *>(d) ) {
        s->price();
        s->save();
    }
    else
        throw UnknownDealType( d );
}
```

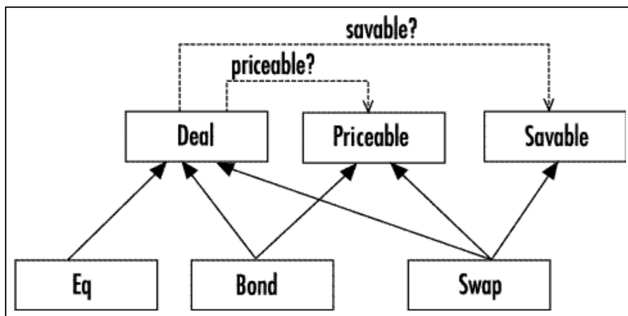

Figure 3. Second version of processDeal.

Another distressingly popular approach is not to ask the object what it is, but rather what it can do. This is often called a "capability query":

```
void processDeal( Deal *d ) {
    d->validate();
    if( Priceable *p =
        dynamic_cast<Priceable *>(d) )
        p->price();
    if( Savable *s =
        dynamic_cast<Savable *>(d) )
        s->save();
}
```

Each base class represents a set of capabilities. A dynamic cast across the hierarchy, or "cross cast," is equivalent to asking whether an object can perform a particular function or set of functions (see Fig. 3). The second version of processDeal essentially says: *Deal, validate yourself. If you can be priced, price yourself. If you can be saved, save yourself.*

This approach is a bit more sophisticated than the previous implementation of processDeal. It may also be somewhat less fragile, since it can handle new types of deals without throwing an exception. However, it still suffers from efficiency and maintenance problems. Consider what would happen if a new interface class should appear in the Deal hierarchy (see Fig. 4).

The appearance of a new capability in the hierarchy is not detected. Essentially, the code never thinks to ask if the deal is legal (which, if you think about it, is pretty accurate domain analysis). As with our earlier solution to the problem of terminating an employee, this capability query–based approach to processing a deal is an ad hoc solution, not a basis for an architecture.

The root problem with both type-based and capability-based queries in OO design is that some of the essential behavior of an object is determined externally to the object itself. This approach runs counter to the principle of data abstraction, perhaps the most basic of thefoundations of object-oriented programming. With these approaches, the meaning of an abstract data type is no longer encapsulated within the class used to implement it, but is distributed throughout the source code.

As with the Employee hierarchy, the safest and most efficient way to add a capability to the Deal hierarchy is also the simplest*:

```
class Deal {
  public:
    virtual void validate() = 0;
    virtual void process() = 0;
    //...
};
class Bond
  : public Deal, public Priceable {
  public:
    void validate();
    void price();
    void process() {
        validate();
        price();
    }
};
class Swap
  : public Deal, public Priceable,
    public Savable {
  public:
    void validate();
    void price();
    void save();
    void process() {
        validate();
        price();
        save();
    }
};
// etc...
```

O TEMPORA, O MORES More than a decade ago, the C++ community decided that "cosmic" hierarchies (architectures in which every object type is derived from a root class, usually called Object) were not an effective design approach in C++. There were a number of reasons for rejecting this approach, both on the design and implementation level. From a design standpoint, cosmic hierarchies often give rise to generic containers of "objects." The contents of the containers are often unpredictable and lead to unexpected runtime behavior. Stroustrup's classic counterexample considered the possibility of putting a battleship in a pencil holder—something that a cosmic hierarchy would allow, but that would probably surprise a user of the pencil holder.

A pervasive and dangerous assumption among inexperienced designers is that an architecture should be as flexible as possible. Error. Rather, an architecture should be as close to the problem domain as possible while retaining sufficient flexibility to permit reasonable future extension. When "software entropy" sets in, and new requirements are difficult to add within the existing structure, then the code should be refactored into a new design. Attempts to create maximally flexible architectures a priori are similar to attempts to create maximally efficient code without profiling; there will be no useful architecture, and there will be a loss of efficiency.

This misapprehension of the goal of an architecture coupled with an unwillingness to do the hard work of abstracting a complex problem domain often results in the reintroduction of a particularly noxious form of cosmic hierarchy:

```
class Object {
  public:
```

* Other techniques can be used to improve on the capability query without modifying the hierarchy, provided that the original design makes provision for them. The Visitor pattern allows new capabilities to be added to a hierarchy, but is fragile when the hierarchy is maintained. The Acyclic Visitor pattern is less fragile than Visitor, but requires a runtime type query that may fail. Either of these approaches, however, is an improvement over systematic use of capability queries.

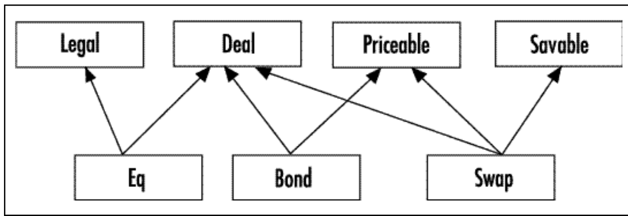

Figure 4. New interfaceclass in the Deal hierarchy.

```
    Object( void *, const Type * );
    virtual ~Object();
    const Type &type();
    void *object();
    //...
};
```

Here, the designer has abdicated all responsibility for understanding and properly abstracting the problem domain, and has instead created a wrapper that can be used to effectively "cosmicize" unrelated types. An object of any type can be wrapped in an Object, and we can create containers of Objects into which we can (and frequently do) put anything at all.

The designer may also provide the means to perform a type-safe conversion of an Object wrapper to the object it wraps†:

```
template <class T>
T *dynamicCast( Object *o ) {
    if( o && o->type() == typeid(T) )
        return reinterpret_cast<T *>
            (o->object());
    return 0;
}
```

Now consider the problem of extracting and using the contents of a container that can contain anything at all:

```
void process( list<Object *> &c ) {
    for( list<Object *>::iterator i(c.begin());
            i != c.end(); ++i ) {
        if( Pencil *p =
            dynamicCast<Pencil *>(*i) )
            p->write();
        else if( Battleship *b =
            dynamicCast<Battleship *>(*i) )
            b->anchorsAweigh();
        else
            throw InTheTowel();
}
```

The repercussions of this abdication of design responsibility are extensive and serious. Any use of a container of Objects is a potential source of an unbounded number of type-related errors. Any change to the set of object types that may be wrapped as Objects will require maintenance to an arbitrary amount of code, and that code may not be available for modification. Finally, because no effective architecture has been provided, every user of the container is faced with the problem of how to extract information about the anonymous objects. Each of these acts of design will result in different and incompatible ways of detecting and reporting errors. For example, one user of the container may feel just a bit silly asking questions like "Are you a pencil? No? A battleship? No? A bond? . . . " and opt for a capability query approach. The results are not much better.

The employee reports to the HR department on his first day of work and is told to "get in line with the other assets." He is directed to a long line of other employees and, strangely, a variety of office equipment, vehicles, furniture, and legal agreements. Finally reaching the head of the line, he is assaulted by a sequence of odd questions: "Do you consume gasoline?" "Can you program?" "Can I make copies with you?" Answering "no" to all the questions, he is eventually sent home, wondering why no one thought to ask him to mop floors, since that was what he was hired to do.

Obviously, if an hourly employee or a bond somehow find its way into our pencil holder, it won't be terminated or priced. That is just the point: How did they get there in the first place? Clearly, there exists, somewhere, a piece of code that thinks it's a good idea to put a battleship in a pencil holder. It's unlikely that this corresponds to anything in the application domain, and this is not the type of design we should encourage or submit to. A local requirement for a cosmic hierarchy generally indicates a design flaw elsewhere.

The employee reports to the HR department on his first day of work. He is directed to a long line of other employees. Finally reaching the head of the line, he is told, "Get to work!" Since he was hired as a janitor, he grabs a mop and spends the rest of the day washing floors.

CONCLUSION Runtime type queries are sometimes necessary or preferable to other design choices. As we've seen, they can be used as a convenient and temporary hack when one is faced with poorly designed third-party software, or with an otherwise impossible requirement to modify existing code without recompilation, when that code was not designed to accommodate such modification. Runtime type queries are also handy in debugging code and have rare, scattered uses in specific problem domains (debuggers, browsers . . .).

Since the standardization of runtime typing mechanisms in C++, however, many designers have employed runtime typing in preference to simpler, more efficient, and more maintainable design approaches. Typically, runtime type queries are used to compensate for bad architectures, which typically arise from compounded hacks, poor domain analysis, or the mistaken notion that an architecture should be maximally flexible.

In practice, it should rarely be necessary to ask an object personal questions about its type or capabilities.

† Note that this template version of dynamic cast is more restricted in its capabilities than the dynamic_cast operator. We could probably do better, but that might have the effect of encouraging its use!