



semantics

C++ Hierarchy Design Idioms

Stephen C. Dewhurst
Semantics Consulting, Inc.

www.semantics.org

C++ Hierarchy Design Idioms

- **Hierarchy Design Idioms**
 - **Data abstraction**
 - **Base class member roles**
 - **Overloading, overriding, and hiding**
 - **Hierarchies and polymorphism**
 - **Conditional code**
 - **Substitutability**
 - **Totalitarianism, tough love, and reuse**
 - **Degenerate hierarchies**
 - **Design for repair**
 - **Composition of simple hierarchies**
 - **Abstract bases, slicing, and copying**

The Advantage of Abstract Data Types

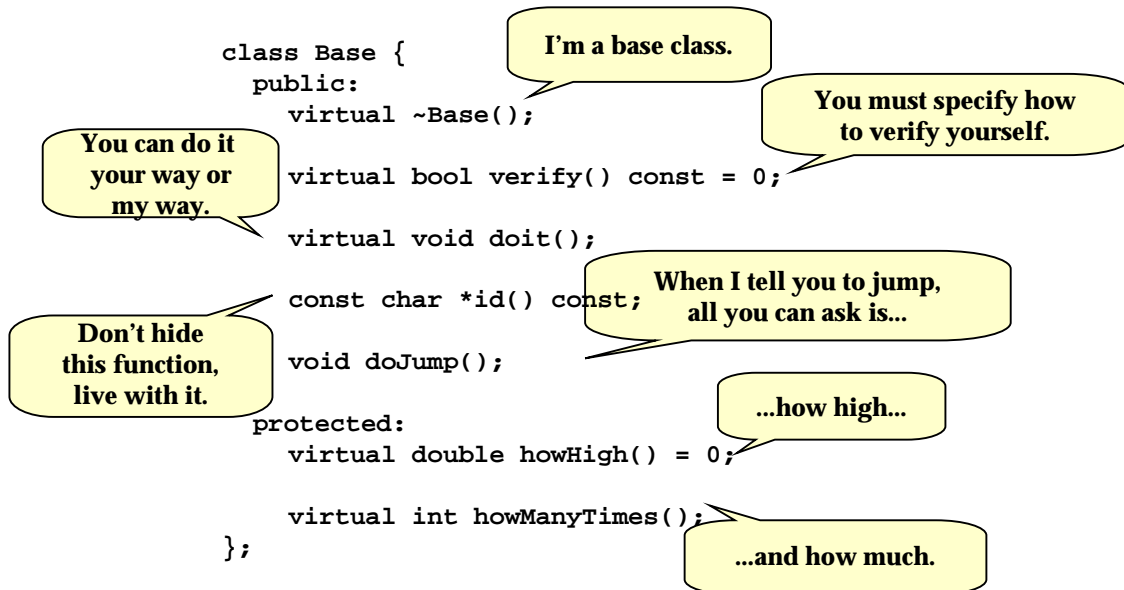
- **A type is a set of operations.**
- **An abstract data type is set of operations with an implementation.**
- **When we identify objects in a problem domain, the first question we ask about them is “What can I do with this object?” not “How is this object implemented?”**
- **If a natural description of a problem involves employees, contracts, and payroll records, then the programming language used to solve the problem should contain `Employee`, `Contract`, and `PayrollRecord` types.**
- **This allows an efficient, two-way translation between the problem domain and the solution domain.**
- **Software written this way has less “translation noise,” and is simpler and more correct.**
- ***The purpose of an abstract data type is to extend the programming language into the problem domain.***

Creating an Abstract Data Type

- **Choose a descriptive name for the type.**
 - If you have trouble choosing a name for the type, you don't know enough about what you want to implement to design an abstract data type.
 - An abstract data type should represent a single, well-defined concept.
- **List the operations that the type can perform.**
 - An abstract data type is defined by what you can do with it.
 - Remember initialization, cleanup, copying, and conversions.
- **Design an interface for the type.**
 - The type should be “easy to use correctly, hard to use incorrectly.”
 - Look out for language-specific pitfalls.
 - An abstract data type extends the language; do proper language design.
 - Put yourself in the place of the user of your type.
- **Implement the type.**
 - Don't let the implementation affect the interface of the type.
 - Implement the contract promised by the type's interface.

Bossy Bases

- Well-designed base classes tell derived classes how they may customize or extend the base class.



- *Design totalitarian base classes.*
- *When deriving, bend to the will of the base class.*

- Note the use of the template method pattern to give the base class finer control over the derived class. It puts the invariant parts of a member function's algorithm into a non-virtual base class member function, and allows derived classes limited customization through the hooks the template method provides.

The Template Method Pattern

- **A template method partitions an algorithm into invariant and variant parts.**
 - The invariant part is defined in a base class.
 - The variant parts of the algorithm are provided as virtual functions that may be overridden in derived classes.

```
class Base {
public:
    //...
    void algorithm();
protected:
    virtual bool hook1() const;
    virtual void hook2() = 0;
};
void Base::algorithm() {
    //...
    if( hook1() ) {
        //...
        hook2();
    }
    //...
}
```

- **A template method gives the base class designer a level of control somewhere between a non-virtual and virtual function.**

- Typically, the invariant part is declared as a non-virtual base class member function. This tells the derived class author that the basic algorithm is an invariant over the hierarchy.
- **Gotcha:** Making the “invariant” algorithm a virtual function provides even more flexibility. However, this is not a good idea, because it defeats the intent of providing an invariant algorithm over the hierarchy.
- Typically, the variant parts of the algorithm are declared as protected virtual members. This tells the derived class author precisely how the invariant algorithm may be customized.
- In another variation on the pattern’s implementation, callbacks or functor callbacks could be used to provide the variant parts of the algorithm.
- The base class may or may not provide a default implementation of the variant parts.
- Use of a template method gives the base class designer fine-grain control over how the base class may be customized by derived classes (but only if the derived class author listens to what the base class is saying!)

Common Sense

- **A base class doesn't *always* have to have a virtual destructor.**

```
namespace std {
    template <class Arg, class Res>
    struct unary_function {
        typedef Arg argument_type;
        typedef Res result_type;
    };
}

namespace Loki {
    struct OpNewCreator {
        template <class T>
        static T *Create() { return new T; }
    protected:
        ~OpNewCreator() {}
    };
}
```

- **Just make sure that your base class really is one of these exceptional cases.**

- **In the first case, the standard library `unary_function` template is just defining a set of typedefs that should be accessible from the derived class. This implies (without requiring) that the inheritance should be public. However, the class must be lightweight because it will be used primarily to implement STL function objects for use as predicates.**
- **In the second case, we're looking at an example of one of Andrei Alexandrescu's policy classes (p. 13), which also has the requirement that it be lightweight. Some protection is provided by making the destructor protected.**

Review: Overloading, Overriding, and Hiding

- **Overloading and overriding are two entirely separate concepts.**
- **Function overloading refers to a set of functions in the same scope that have the same name and different signatures.**

```
class Base {
    void f();
    virtual void f( int );
};
```

- **Overriding refers to a derived class function that has the same name and signature as a base class virtual function.**

```
class Derived : public Base {
    void f( int );
};
```

- **A name in an inner scope hides the same name in outer scopes.**

```
class Derived2 : public Derived {
    int f;
};
Derived2 *d2p = something;
d2p->f( 12 ); // error! f is not a function
d2p->Derived::f( 12 ); // OK
```

- ***Careless combination of overloading, overriding, and hiding can make code difficult to understand and maintain.***

- **Note: A function's signature consists of the number, type, and order of its formal arguments, and its type-qualifiers (const and/or volatile). It does not include the return type or any default initializers.**
- **Note that a name declared in an inner scope hides all names with the same spelling in outer scopes. This implies that a derived class function hides all overloaded base class functions with the same name.**
- **Let's be clear: A derived class function hides base class functions with the same identifier. It does not overload them. That's Java, not C++.**

Hiding Base Class Non-Virtuals

- **A base class non-virtual function specifies an invariant that applies to all derived classes.**
- **Hiding a base class non-virtual raises the complexity of the hierarchy, and will lead to misunderstanding and error.**

```
class B {
    public:
        void f();
        void f( int );
};
class D : public B {
    public:
        void f();
};

B *bp = new D;
bp->f(); // oops! called B::f() for D object
D *dp = new D;
dp->f( 123 ); // error! B::f(int) hidden
```

- **Hiding non-virtuals defeats polymorphism; different interfaces to the same object give different behavior.**
- ***Do not hide base class non-virtuals.***

- **The problem is that the derived object has polymorphic type. It has two equally valid interfaces; that of its base class and its own class. It should produce the same behavior when accessed through either interface.**

Overloading Virtual Functions

- **Be careful about overloading virtual functions.**
- **A set of overloaded virtual functions in the base class will be hidden by a single overriding function in a derived class.**
- **This will result in different functions being called depending on the static type used to call the function.**

```
class Base {
public:
    virtual void f( double );
    virtual void f( int );
};
class Derived : public Base {
public:
    void f( int );
};
// ...
Derived *dp = new Derived;
Base *bp = dp;
bp->f( 12.3 ); // calls Base::f( double )
dp->f( 12.3 ); // calls Derived::f( int )!!!
```

- ***Don't overload virtual functions.***

- **Like most design heuristics, this one has its exceptions. Probably the most common example of an exception is to be found in a common implementation of the Visitor pattern, in which there is some benefit to overloading the virtual visit operations.**

Overloading Virtual Functions

- **If you need an overloaded member function name, overload non-virtual member functions that “kick down” to differently-named virtual functions.**

```
class Base {
public:
    void f( double );
    void f( int );
protected:
    virtual void f_double( double );
    virtual void f_int( int );
};
inline void f( int i ) { f_int( i ); }
inline void f( double d ) { f_double( d ); }
```

- **Derived classes may then override a single base class function without hiding all the others.**

Virtual Functions and Default Initializers

- **A function's signature does not include default argument initializers.**
- **A base class virtual with a default initializer can be overridden by a derived class function without a default initializer, or with a different default initializer. This can lead to confusion.**

```
class Base {
public:
    virtual void f( int = 12 );
    virtual void g( int = 10 );
};
class Derived : public Base {
public:
    void f( int );
    void g( int = 5 );
};
// ...
Derived *dp = new Derived;
Base *bp = dp;
bp->f(); // calls Derived::f( 12 )
dp->f(); // error!
bp->g(); // calls Derived::g( 10 )!
dp->g(); // calls Derived::g( 5 )
```

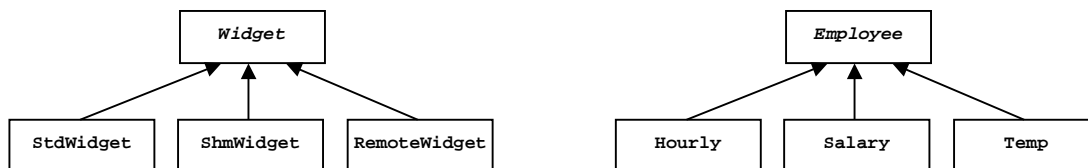
- ***Avoid default argument initializers for virtual functions.***

Abstract and Concrete Classes

- **An abstract class can not be used to instantiate an object.**
- **Base classes represent abstract concepts, and should therefore be abstract.**
 - there are no “employees” in the problem domain, so there shouldn’t be any `Employee` objects
 - there are no “symbol tables” in the compiler, only specific types of symbol table
- **Generic code should be written to a base class’s interface, without making the assumption that it is dealing precisely with a base class.**
- **Concrete base classes may give rise to low-level problems.**
 - Slicing!
 - Hard to implement copy operations.
- ***Class hierarchies should be designed with abstract base classes and concrete leaves.***

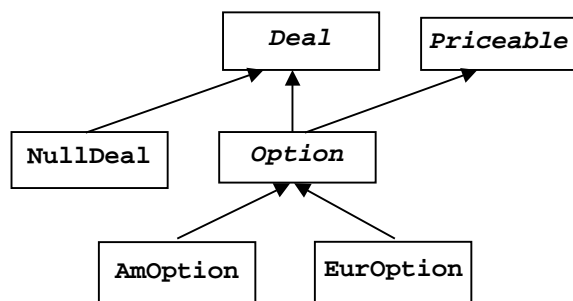
Where Do Hierarchies Come From?

- We may recognize a hierarchy from the top, through specialization.
 - “Our application deals with employees.” “What kind of employees are there?” “The usual: hourly, salaried, and probably some others in the future.”
- We may recognize a hierarchy from the bottom, through abstraction.
 - “I’ve got a class table, a function table, and a global table, and they all have different implementations.” “Have they got anything in common?” “Well, they all behave like symbol tables.”
- We may recognize a hierarchy late in development, from implementation issues.
 - “I’ve got a Widget object that may be in my local memory, in shared memory, or on another node in the network. I’m getting pretty tired of special casing every time I want to access a Widget.” “Don’t.”



The Meaning of Polymorphism

- Consider a type of financial option, `AmOption`.
- It is simultaneously an `AmOption`, an `Option`, a `Deal`, and a `Priceable`.
- This means it can respond to messages sent to any of its four interfaces.



- This means that an `AmOption` can leverage generic code written to any of its base classes' interfaces.
- Our hierarchy design heuristics tell us how to craft class hierarchies to make this possible.

Static and Dynamic Binding

- **Polymorphism depends on dynamic binding.**
- **Static binding determines what function is going to be called at compile time, based on the declared type of the object.**

```
Employee *ep = getNextEmployee();  
cout << ep->getName() << endl; // calls Employee::getName
```

- **Dynamic binding waits until runtime to determine what function to call.**

```
Deal *dp = getNextDeal();  
dp->validate(); // calls some sort of validate...
```

- **Typically, the function called depends on the type of a single object (single dispatch).**
- **It's also possible to implement dynamic binding based on the types of multiple objects (multiple dispatch).**
- ***An object should exhibit the same behavior no matter which of its interfaces is used to manipulate it.***

```
AmOption *d = new AmOption;  
Option *b = d;  
d->price();  
b->price(); // should be same behavior!
```

Type-Based Conditionals

- **We don't switch on type codes in object-oriented programs.**

```
void process( Employee *e ) {  
    switch( e->type() ) { // evil code!  
        case SALARY: fireSalary( e ); break;  
        case HOURLY: fireHourly( e ); break;  
        case TEMP: fireTemp( e ); break;  
        default: throw UnknownEmployeeType();  
    }  
}
```

- **The polymorphic approach is more appropriate.**

```
void process( Employee *e )  
    { e->fire(); }
```

- **The advantages are enormous:**

- It's simpler.
- It doesn't have to be to be recompiled as new employee types are added.
- It is impossible to have type-based runtime errors.
- It's probably faster and smaller!

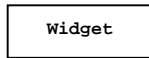
- ***Implement type-based decisions with dynamic binding, not with conditional control structures.***

- **Note that there are some situations in which one has to deal with type codes. For example, consider an application that reads a message that is just a sequence of bytes, preceded by a integer type code. In this case we inevitably have to deal with type codes, but the goal would be to transform the data into a well-formed object as soon as possible, and allow the rest of the program to be written without reference to type codes. Note that it is still not necessary (although it may be desirable) to explicitly refer to the type codes in the program.**

Avoiding Control Structures with Dynamic Binding

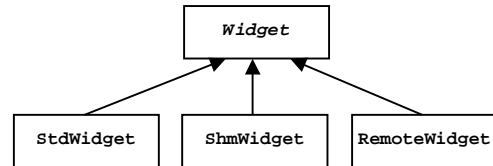
- One way to avoid making an incorrect decision is not to make a decision.
- Many conditional constructs can be “encoded” in a class hierarchy.

Before



```
if( Widget is in local memory )
    w->process();
else if( Widget is in shared memory )
    do horrible things to process it
else if( Widget is remote )
    do even worse things to process it
else
    error();
```

After

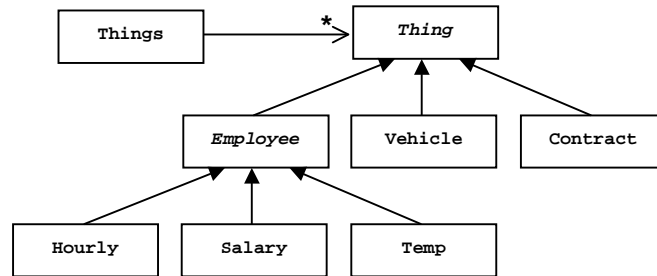


```
w->process();
```

- We effectively convert conditional code into type-based code.
- *Convert conditional control structures into type-based decisions where appropriate.*

Cosmic Hierarchies

- **Overly-inclusive hierarchies are generally bad design.**



- **Such hierarchies tend to give rise to “containers of anything.”**
- **Type information is lost, and must be recovered through conditional code.**
 - “Ok, thing, I’m going to process you. Are you a vehicle?” **“No.”** “All right, are you a contract?” **“Nope.”** “Well, perhaps you’re an employee?” **“Wrong again.”** “I give up!”
- **This kind of conditional code is particularly inefficient, hard to maintain, and prone to error.**
- **Such hierarchies may also be inefficient.**
- ***Avoid cosmic hierarchies.***

Some Bad Code

```
void process( Thing *a ) {
    if( Vehicle *v = dynamic_cast<Vehicle *>(a) )
        v->drive();
    else if( Contract *c = dynamic_cast<Contract *>(a) )
        c->enforce();
    else if( Employee *e = dynamic_cast<Employee *>(a) )
        e->fire();
    else
        throw UnknownAssetType( a );
}

void doThings( list<Thing *> things ) {
    for( list<Thing *>::iterator i(things.begin); i != things.end(); ++i )
        try {
            process( *i );
        }
        catch( UnknownThing &ut ) {
            // ???
        }
}
```

Casting for Flexibility and Disaster

- **In order to add functionality, users of the Thing hierarchy may have to resort to type-based conditional code.**

```
void process( Thing *a ) {
    if( Vehicle *v = dynamic_cast<Vehicle *>(a) )
        v->drive();
    else if( Contract *c = dynamic_cast<Contract *>(a) )
        c->enforce();
    else if( Employee *e = dynamic_cast<Employee *>(a) )
        e->fire();
    else
        throw UnknownThing( a );
}
```

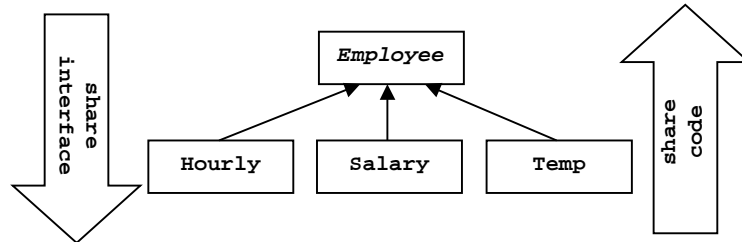
- **This approach is both slow and difficult to maintain.**
- **Alternatively, the Thing hierarchy can provide a hook for a Visitor pattern implementation.**

```
class ThingVisitor;
class Thing {
public:
    void accept( ThingVisitor & ) = 0;
    //...
};
```

- **Note that there is no reason to move existing, general purpose operations out of the hierarchy. They're more efficiently implemented as single-dispatch operations.**

Hierarchies and Reuse

- **Class hierarchies promote reuse in two ways.**
 - code sharing
 - interface sharing
- **We get code sharing by putting common derived class implementations in base classes. This is good.**



- **We get interface sharing by writing substitutable derived classes. This is better.**
- ***Interface sharing is more important than code sharing. Don't sacrifice the base class interface in order to share code.***

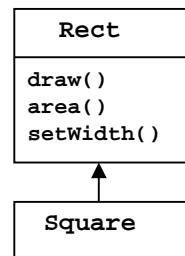
Is-A Relationships

- **Public inheritance is used to model the is-a relationship.**
- **Fine. What does the is-a relationship model?**
- **Is-a does not *necessarily* mean specialization.**
 - a `Stack` is not a `List`
- **Is-a does not *necessarily* mean subset.**
 - a `Circle` is not necessarily an `Ellipse`
- **On the other hand, an is-a relationship may hold between two types that are not logically related, or whose relationship emerged after analysis, during design.**
 - a `Model` is not a `DisplayProtocol`, but a `Model` is-a `DisplayProtocol` if it responds to the display protocol
 - a `Person` is not a `Persistent`, but a `Person` is-a `Persistent` if it can be saved to disk
- **Is-a means substitutable.**
- ***Public inheritance should imply substitutability.***

- **Note: The concepts of specialization and subsetting are very valuable tools for thinking about and discovering class hierarchies, but substitutability is considered the most important property for proper implementation of inheritance.**
- **A `Stack` is not a `List` because `Lists` have operations that do not apply to `Stacks`.**
- **A `Circle` may not be an `Ellipse` if an `Ellipse` may be deformed asymmetrically..**
- **Public inheritance is often used improperly to implement a has-a relationship.**
- **Public inheritance is often used improperly to implement an “is-implemented-in-terms-of” relationship. A `Stack` may be implemented in terms of a `List`. Implemented-in-terms-of is implemented with layering (membership) or, if absolutely necessary, with private inheritance.**

Isa Relationships and Substitutability

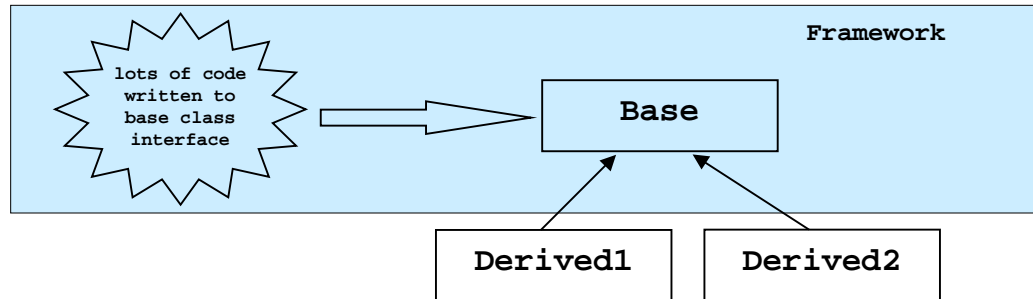
- **When we analyze a domain, we may recognize isa relationships, and enshrine them in hierarchies.**
 - a salaried employee isa employee
 - a function symbol table isa symbol table
 - a square isa rectangle
- **However, substitutability is a low-level property of a hierarchy.**
 - it depends on the set of operations promised by the base class
 - it depends on the current and future uses of the base class interface by generic code
- **For instance, a Square may not be a Rectangle.**
- **Our base class design heuristics tell us how to communicate the requirements for substitutability to derived class designers.**



- ***Substitutability is the most important property of a class hierarchy.***

The Contract

- A base class establishes a contract between generic code written to the contract and derived classes that implement the contract.

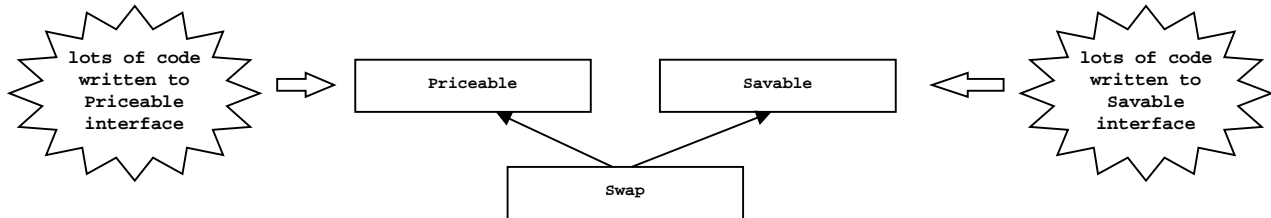


- The generic code knows nothing about the derived classes.
- The generic code may have been compiled long before the derived classes existed.
- The authors of the generic code and base class may have no knowledge of or control over the derived classes.
- *The contract provided by the base class is what allows the derived classes and generic code to work together.*

- A derived class object must be substitutable for an object of its base classes.
- This is what enables us to write generic, extensible code.
 - we provide a generic framework defined in terms of base classes.
 - others extend the utility of our framework by defining substitutable derived classes.
- Some of the more interesting object-oriented design techniques deal with the problem of how generic code that knows only about the base class interface can create objects of concrete derived classes.
- Two quotes from Cline and Lomow (*C++ FAQs*) that succinctly express the motivation behind substitutability:
 - “A derived class shouldn’t shock users of the base class.” If someone asks an object of a class derived from `Bird` to fly, the derived class object shouldn’t refuse!
 - A derived class should “require no more, promise no less” than its base class.
- Barton and Nackman (*Scientific and Engineering C++*) avoid to a large extent the use of `is-a`, instead employing the term “is-usable-as,” which more closely reflects the goal of substitutability.

Totalitarianism, Substitutability, and Tough Love

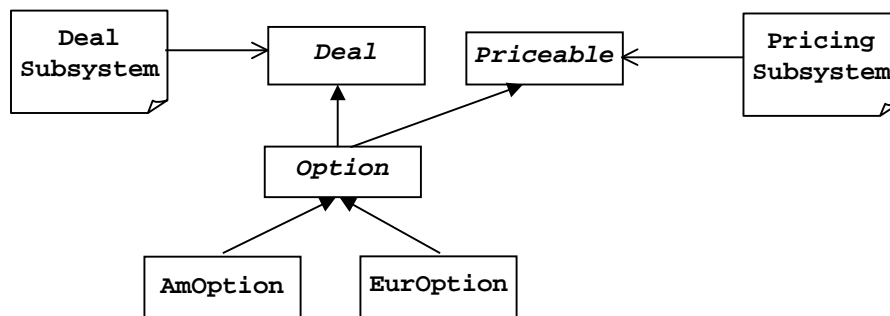
- **Most code is written in terms of abstract base class interfaces.**



- **The only way this can work is if *every* derived class is *always* substitutable for *each* of its public base classes.**
- **The only way this can work is if base classes are hard on the derived classes, and if derived classes listen to and obey the base classes.**

Contracts and Leveraging Generic Code

- **A base class specifies a contract.**
 - generic code is written to the base class interface
 - derived classes customize the generic code by being substitutable for the base class



- **The greatest reuse is achieved by leveraging entire subsystems with substitutable derived classes.**
- *Base class design is about writing clear contracts.*
- *Derived class design is about fulfilling base class contracts.*
- *The base class is ignorant of its derived classes.*

Attaching Interfaces

- **Consider the following interface classes.**

```
class Saveable { // persistent interface
    virtual void save() = 0;
    // ...
};
class Priceable { // pricing interface
    virtual void price() = 0;
    // ...
};
```

- **These interface classes would commonly be used to attach attributes to classes in a single inheritance hierarchy.**

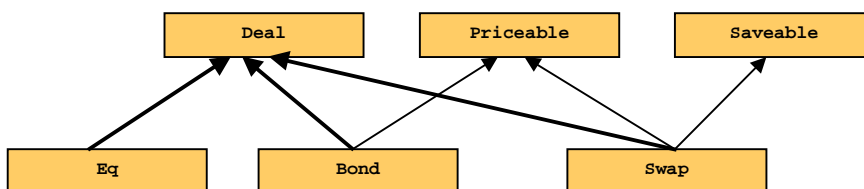
```
class Deal {
public:
    virtual void validate() = 0;
    //...
};
class Bond : public Deal, public Priceable {
public:
    void validate();
    void price();
    //...
};
class Swap : public Deal, public Priceable, public Saveable
    { /*...*/};
```

dynamic_cast as the Downcast of Doom

- **Consider adding a new capability without changing or recompiling the hierarchy.**
- **Naive code might simply ask the obvious questions.**

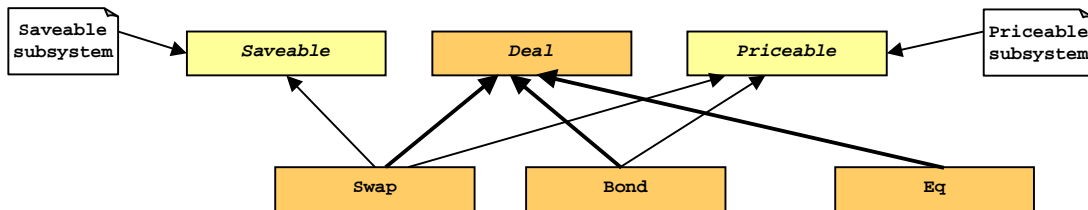
```
void processDeal( Deal *d ) {  
    d->validate();  
    if( Bond *b = dynamic_cast<Bond *>(d) )  
        b->price();  
    else if( Swap *s = dynamic_cast<Swap *>(d) ) {  
        s->price();  
        s->save();  
    }  
}
```

- **This code is very fragile, slow, and hard to maintain.**



Capability Queries

- A common organization in OOD is to attach “capabilities” to classes in a hierarchy through multiple inheritance.



- This is fine.
- The interface classes represent a set of operations; a capability. A successful cross-cast to the interface type indicates that the unknown concrete class has that capability.

```
Deal *dp = getNextDeal();
Priceable *pp = dynamic_cast<Priceable *>(dp); //can you be priced?
if( pp ) {
    // yup.
}
else {
    // nope.
}
```

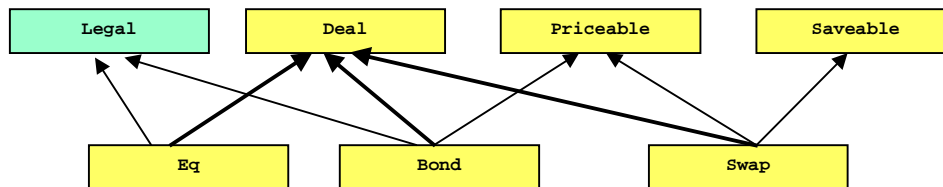
- This is usually a bad idea.

Capability Queries as a Stopgap

- A `dynamic_cast` can be used to ask if a particular `Deal` object, referred to through a base class pointer, may be priced and/or is persistent.

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

- This code is somewhat less fragile.
- It's still slow and dangerous.
- Capability queries are not a good base for a design; they are a hack.



A Better Design

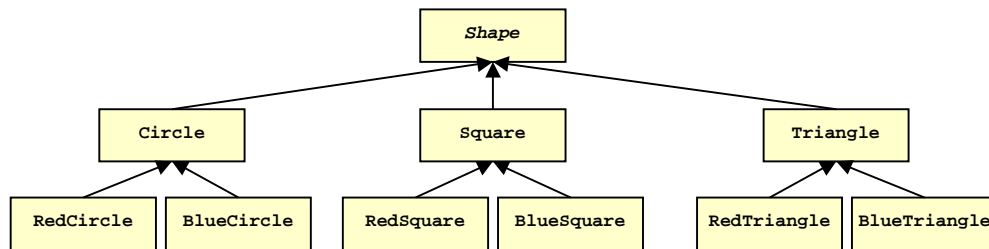
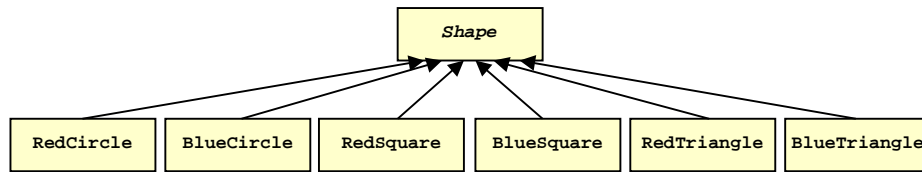
- **Remember OOD 101?**

```
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 Saveable {
public:
    void validate();
    void price();
    void save();
    void process()
        { validate(); price(); save(); }
};
```

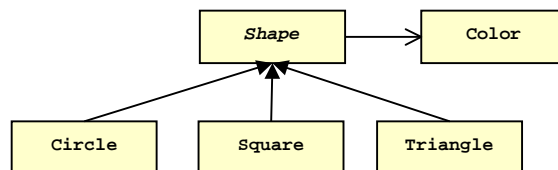
- **This code is (much) faster and simpler, but we had to modify the hierarchy.**

Exponentially Expanding Hierarchies

- A common error among new OO designers is to overuse inheritance.

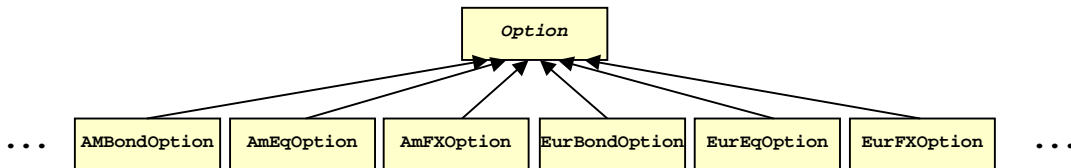


- Composition, or composition of simpler hierarchies, is usually a better choice.

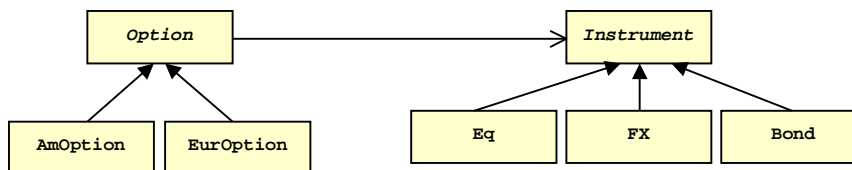


Wide or Deep Hierarchies

- A very wide or very deep inheritance hierarchy usually indicates a design flaw.
- A hierarchy that exhibits “exponential expansion” during maintenance usually indicates a design flaw.



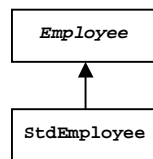
- Generally, the correct design involves composition of simpler hierarchies.



- *Avoid overusing inheritance.*
- *Use composition of simple hierarchies rather than monolithic hierarchies.*

Base Classes and Standalone Classes are Different

- **There are three types of classes.**
 - abstract base classes
 - concrete derived classes
 - standalone classes
- **The design heuristics for each of these three types of class are very, very different. Decide what you're trying to design before you design it.**
- **Client code treats base classes very differently from standalone classes.**
- **Standalone classes that later become base classes wreak havoc on using code. Start a potential base class off as an abstract base class.**

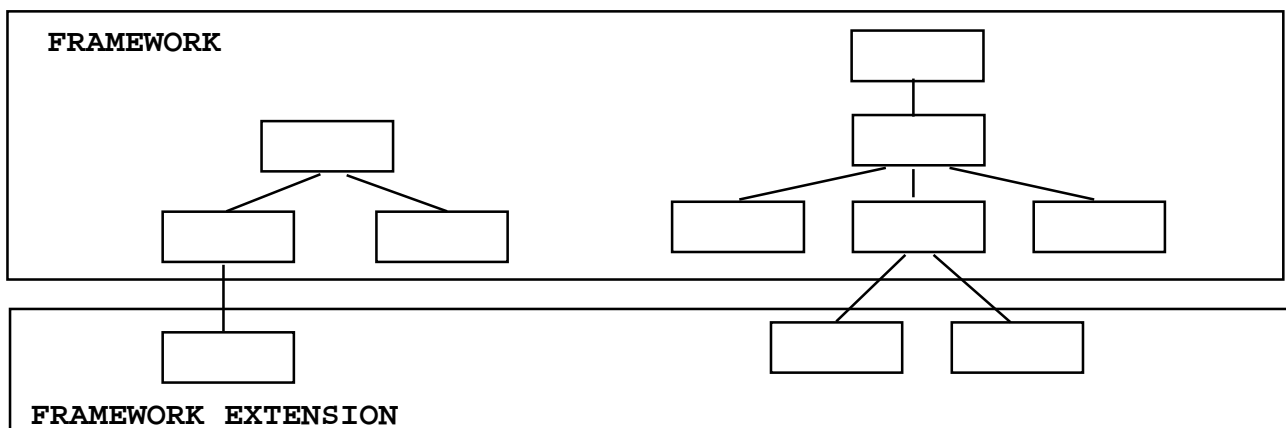


- ***Classes that are part of a hierarchy and standalone classes come from different planets.***
- ***Degenerate hierarchies are your friends.***

Framework-Oriented Design

- **Every significant application has variations, either in “space” or in time.**
- **Framework-oriented design deals well with these issues.**
 - supports the open/closed principle (Meyer)
 - “build for today, design for tomorrow” (Goldfedder)
- **It is a mistake to write an application.**
 - every significant application should be designed as a framework.
 - patterns help a lot here, but remember to be wary of the hype.

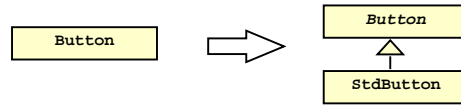
- **It helps to think of a framework as a fully-functional but “vanilla” application. The framework is extended through inheritance and callbacks to produce a family of related applications.**



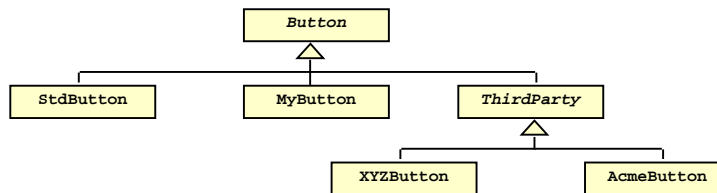
- **Framework design typically employs the Hollywood Principle: “Don’t call us, we’ll call you.” Major decisions about application structure and control flow are determined by the framework. The framework determines when an operation is to occur; customization of the framework determines precisely what will happen.**
- **Even if your application is not being designed as a framework, the benefits of framework-style design are significant. Applications designed as if they were frameworks are typically more correct, reuse more code, and are easier to maintain than other applications.**

Degenerate Hierarchies

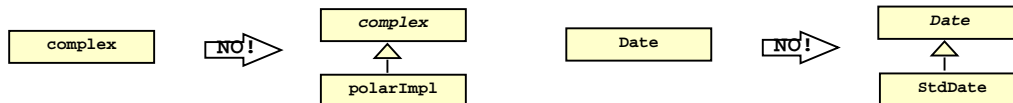
- Recognizing that a class *may* become a base class in the future, and transforming it into a simple two-class hierarchy, is an example of “designing for the future.”



- The alternative of initially employing a concrete class, and introducing derived types later, would force us, or our users, to rewrite existing framework code.



- Obviously, this transformation does not make sense for every concrete class. As designers, we must use our judgement.



- In general, it is a good idea to make all base classes abstract. Equivalently, avoid deriving from concrete classes.
- Making all base classes abstract eliminates slicing problems, and makes the coding of copy operations easier and more likely to be correct.
- Deriving from abstract base classes only also has the effect of rendering code more maintainable. Many “design patterns” can be retrofitted into a class hierarchy if all bases are abstract.
- Unfortunately, we do not always have control over our base classes (for instance, if they are part of a third-party library). In that case, derive from concrete classes when necessary, but do so with extreme caution!
- Note, however, that some of the standard C++ library uses concrete base classes. However, in each case there is a good reason for abandoning this advice.

Inheritance for Code Reuse

- **Inheritance is primarily about interface reuse, not code reuse.**
- **Use of inheritance *solely* for the purpose of reusing base class implementations in derived classes often results in unnatural, unmaintainable, and ultimately more inefficient designs.**
- **A priori use of inheritance for code reuse will result in less code sharing than use of inheritance for interface reuse.**
- ***Concentrate on inheritance of interface. Proper and efficient code reuse will follow automatically.***

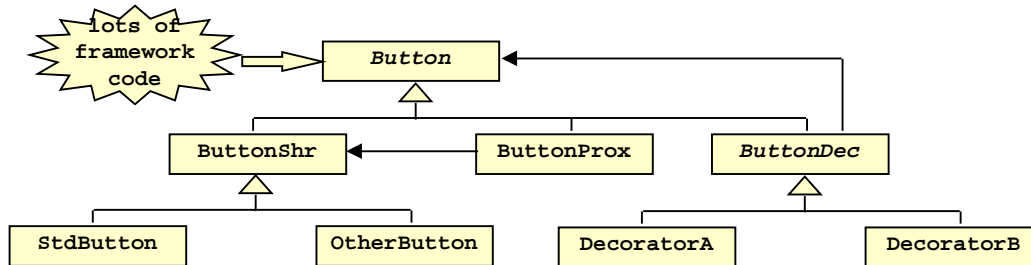
- **Note that private inheritance may sometimes be used in place of composition. Typically, we prefer to use composition rather than private inheritance. However, there are three common reasons to use private inheritance instead:**
 - **We need to access protected members.**
 - **We need to override virtual functions.**
 - **We have a “zero-sized” class and we don’t want to cause storage to be allocated for it in our class.**
- **An example of a class with zero size might be a class that contains only typedefs, enumerators, and static members. However, any C++ object must have non-zero size, and a data member of a zero-sized class probably will increase the size of the class that contains it. (This may not be the case if the data member occupies an offset within the object that would otherwise contain unused “slack bytes” inserted for alignment of the following data member.)**

```
template <class T>
struct AllAboutMe {
    typedef T MyType;
    typedef T *PtrToMe;
    typedef T &RefToMe;
    static T *makeMe() { return new T; }
};
class SomeClass : private AllAboutMe<SomeClass> {
    //...
};
```

- **Most C++ compilers are capable of optimizing away the space for a zero-sized base class, and so private inheritance is sometimes used in preference to composition in order to permit this optimization.**

Contradictory Design Forces

- **Sharing common code in base classes is good.**
- **Leveraging the base class interface is better.**
- **Can we have both?**



- **This introduces an “artificial” class into the hierarchy, but may *sometimes* be justified.**

Observations

- **Design patterns are often selected for “force resolution,” but knowledge of their existence is also a force on the structure of a hierarchy.**
- **Composition of simple parts is simpler than a monolithic design, but can represent a more complex structure.**
- **Designs that promote ignorance and a single point of change are good.**
- **Code the minimum, but design toward the future.**
- **Maximum flexibility is not a goal, reasonable flexibility is.**
- **Idioms are useful only if they are both generally used and sometimes disregarded.**
- **There is no substitute for thoughtful abstraction and careful design. There are no cookbooks for OOD.**

More Information

- **This talk is available in a version that contains accompanying notes. See <http://www.semantics.org/presentations.html#notes> for downloadable pdf.**
- **Steve also has a mailing list to which he posts periodic and asynchronous announcements that are likely to be of interest to his clients, readers, and technical adversaries. See the “Mailing List” section of www.semantics.org. The message volume is low, and is used to announce upcoming conference talks, magazine articles, courses, web casts, books, and web articles, including Steve’s “Once, Weakly” C++ topic of the week.**