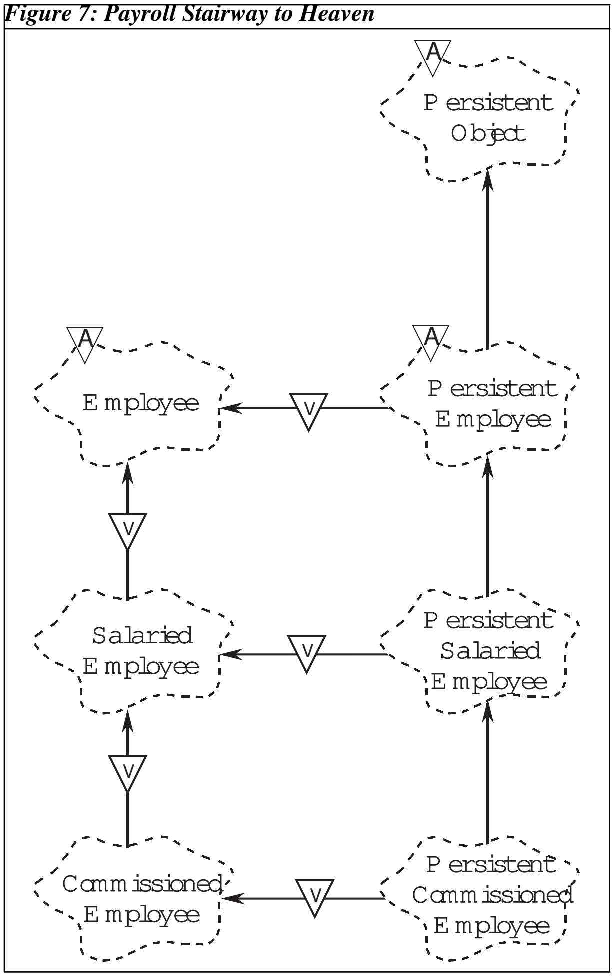
One thing that I found especially interesting while writing this paper, was the number of other patterns that I used in my examples. When determining the best way to present the problems of dual hierarchies to a group of pattern literate readers, the answer in each case was to relate the problems to a different pattern. This shows that patterns are effective ways for engineers to reason about software problems. It also demonstrates that there are deeper relationships between patterns than we can see by examining them in isolation.

objects separate and reusable.

Structure

See Figure 7

Figure 7: Payroll Stairway to Heaven



Notes

Figure 7 shows the use of virtual inheritance within the structure of this pattern. This is a C++ -ism, but has correspondents in other languages. The use of virtual inheritance is necessary in this pattern to prevent the repeated inheritance of the base classes. The desire is that there be only one copy of all the base objects in any of the derived objects.

Applicability

This pattern is best used to insure the isolation of concepts so that reusable class hierarchies do not become polluted with concepts that are application specific.

CONCLUSION

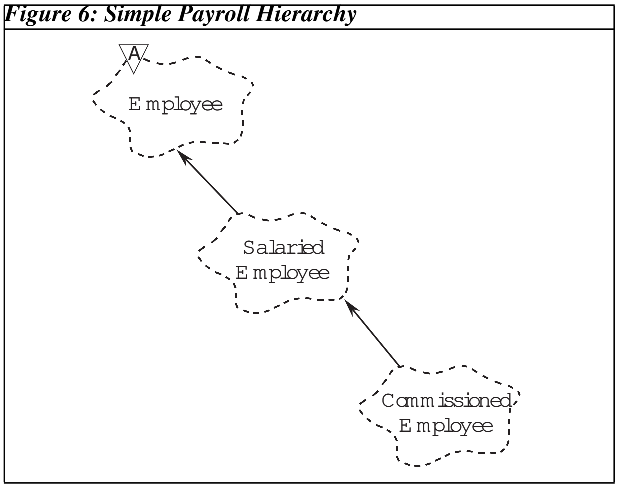
This paper has discussed three patterns that can be used to manage dual hierarchies. We have noted that dual hierarchies arise because of the desire to separate orthogonal concepts from each other. Most typically this allows reusable classes to remain independent of application specific details.

1. The implementation language does not directly support type checked downcasting
2. The type checked downcasting supported by the language is too expensive to use where needed.

DUAL HIERARCHIES OF INTERFACE AND IMPLEMENTATION

Consider a use of the ADAPTER[1] pattern. We have a hierarchy of classes that model part of the payroll problem (See Figure 6) and we would like to ADAPT these classes to a class named **`PersistentObject`** which provides the methods and facilities for writing objects out onto some persistent storage device.

Figure 6: Simple Payroll Hierarchy



The **`Employee`** class and the **`PersistentObject`** class can be ADAPTED by inheriting both of them into a **`PersistentEmployee`** class. It is clear that **`PersistentSalariedEmployee`** should inherit from **`SalariedEmployee`**, but it must also inherit from **`PersistentEmployee`**. Likewise, **`PersistentCommissionedEmployee`** must inherit from **`CommissionedEmployee`** as well as **`PersistentSalariedEmployee`**. This, is the STAIRWAY TO HEAVEN pattern (See Figure 7)

STAIRWAY TO HEAVEN

Intent

This pattern describes the network of inheritance relationships that is needed when a given hierarchy must be adapted, in its entirety, to another class.

Motivation

If a class hierarchy is to be reusable, it cannot depend upon detailed implementations. For example, one might have a class hierarchy of payroll objects such as: Employee, SalariedEmployee, HourlyEmployee, etc. If this these classes focus only upon the algorithms necessary to implement their particular abstractions, then they are highly reusable. However, if they were to incorporate the methods for reading and writing such objects to a particular database engine, then they would not be reusable in applications that did not have access to, or need that particular engine. Thus, we would like to keep any knowledge of database engines out of these objects.

To do this, a new set of classes needs to be created that inherits the ability to read and write themselves using the particular database engine required by the application, and the methods that model the payroll abstraction. This keeps the payroll

---

[1] *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994 p. 139

Listing 4: RTTI Visitor (Continued)

```
    : itsD1(0)
    , itsD2(0)
    {}

    void Visit(D1& d1) {itsD1 = &d1;}
    void Visit(D2& d2) {itsD2 = &d2;}
    D1* GetD1() const {return itsD1;}
    D2* GetD2() const {return itsD2;}

  private:
    D1* itsD1;
    D2* itsD2;
};

class D1 : public B
{
  public:
    virtual void Visit(BVisitor& bv)
    {bv.Visit(*this);} // class Visit(D1&)
};

class D2 : public B
{
  public:
    virtual void Visit(BVisitor& bv)
    {bv.Visit(*this);} // class Visit(D2&)
};

// Global functions
D1* dynamic_cast_D1(B* b)
{
    BVisitor v;
    b->Visit(v);
    return v.GetD1();
}

D2* dynamic_cast_D2(B* b)
{
    BVisitor v;
    b->Visit(v);
    return v.GetD2();
}
```

Notes

The dependency structure of the VISITOR pattern is cyclic. That is, the base class depends upon the visitor class. The visitor class depends upon the derived classes. The derived classes depend upon the base class through the inheritance relationship, thus completing the cycle.

What this means is that users of any of the derived classes must be recompiled (and possibly retested) whenever *any* other derivative changes. Thus, this pattern must be used with care.

Applicability

This pattern is most applicable in situations where type checked downcasting is required and either:

rectly support it. The technique mapped out here is also very fast.

Motivation

Although type checked downcasting is often considered inappropriate, and sometimes even dangerous, there are still some situations that require its use. Not all languages provide a mechanisms for type checked downcasting.

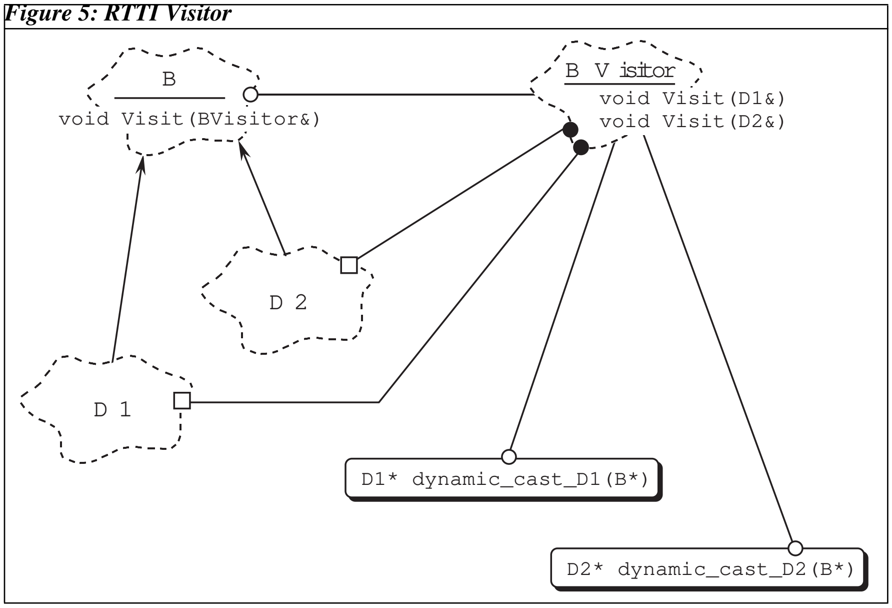
Solution

Use the VISITOR[1] pattern to provide classes that can be invoked by global downcast functions.

Structure

See Figure 5.

Figure 5: RTTI Visitor



Sample Code

See Listing 4

Listing 4: RTTI Visitor

```
class BVisitor;
class D1;
class D2;

class B
{
  public:
    virtual void Visit(BVisitor&) = 0;
};

class BVisitor
{
  public:
    BVisitor()
```

---

[1] *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994 p. 331

acceptable mechanisms for determining if the association has been built correctly. Moreover, in statically typed languages the type checked downcast is the only way to safely gain access to the specific methods of the peers.

Sample Code

```
class B1
{
  public:
    B2* GetB2() const; // cannot be overridden
};

class B2 {};
class D2 : public B2
{
  public:
    void D2Operation();
};

class D1 : public B1
{
  public:
    void DoOperation();
};

void D1::DoOperation()
{
    B2* b2 = GetB2();
    D2* d2 = dynamic_cast<D2*>(b2);
    if (d2)
    {
        d2->D2Operation();
    }
}
```

Notes

There is much debate over whether type checked downcasts are appropriate tools for use in object oriented software. This controversy stems from the fact that there are many uses to which type checked downcasts *could* be put that are better addressed with polymorphic message dispatch; as in the INTELLIGENT CHILDREN pattern. However, there are cases, such as those described in this pattern, where polymorphic dispatch can not resolve the issue. In those cases, type checked downcasts are appropriate.

The type checked downcast can be implemented in many different ways. Many languages have some form of query to check if an object conforms to a certain type, or responds to a certain interface. In languages that do not support this directly, or in which the type check operation is too expensive for the algorithm that uses it, unique type values can be added to all the derived objects.

Another option is to use the RTTI VISITOR pattern described below.

Applicability

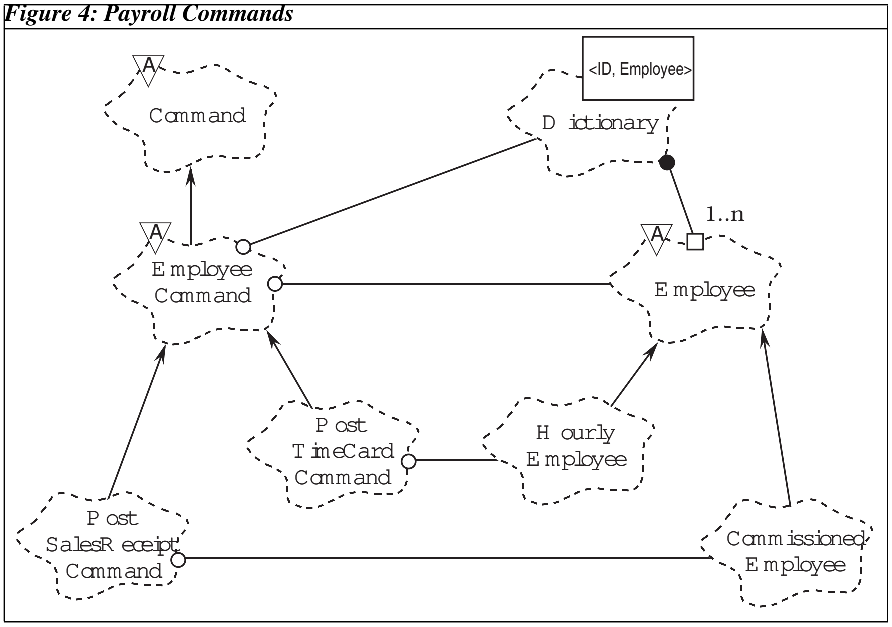
Use this pattern whenever you are programming a dual hierarchy and there is no way to retain the type identity of the derived peers in the peers themselves.

RTTI VISITOR

Intent

This pattern provides a mechanism by which type checked downcasting can be implemented in languages that do not di-

Figure 4: Payroll Commands



`ProcessEmployee` which it expects the derivatives to implement.

Here we face the same downcasting problem that we faced with the OBSERVER pattern, when a **`PostTimeCardCommand`** object gets a **`ProcessEmployee`** message it is passed an **`Employee`** object which it must downcast to a **`HourlyEmployee`** object. However, this time we cannot employ INTELLIGENT CHILDREN to avoid the downcast because the peers are not permanently associated, and the temporary associations are being built by the base class **`EmployeeCommand`** which does not know about the derivatives.

Although this is primarily a problem associated with statically typed languages, this situation cannot be ignored by programmers of dynamically typed languages. The risk is that a **`PostTimeCardCommand`** will one day be asked to operate on the ID of a **`CommissionedEmployee`** object.

The RUNGS OF A DUAL HIERARCHY pattern addresses this situation by specifically permitting a type checked downcast.

RUNGS OF A DUAL HIERARCHY

Intent

To provide a context that justifies the use of type checked dynamic downcasting in dual hierarchy situations.

Motivation

Downcasting, even type checked downcasting, is often considered a poor engineering practice[1]. However, there are situations in which there is no alternative to using a downcast. This pattern is motivated by the need to identify one of those situations.

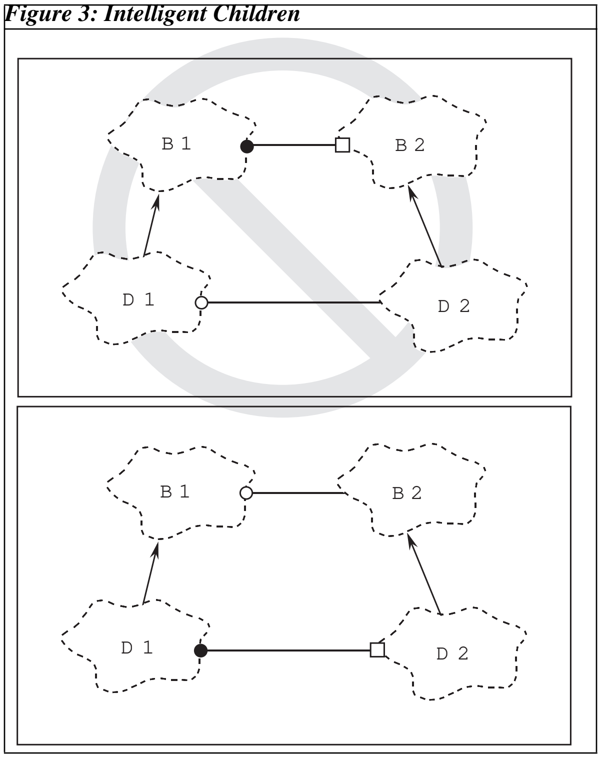
Solution

When programming a dual hierarchy in which peer to peer associations are built dynamically by objects that are not aware of the derived peers (i.e. objects at a level of abstraction that is higher than the peers) then type checked downcasting is an

---

[1] The reason for this is beyond the scope of this article, but has to do with potential violations of the open/closed principle of object oriented design. See: *Designing Object Oriented C++ Applications using the Booch Method*, Robert C. Martin, Prehtice Hall, 1995, p. 286 and p. 360

Figure 3: Intelligent Children



DYNAMIC DUAL HIERARCHIES

A dual hierarchy is like a ladder. The two inheritance hierarchies are the supports of the ladder, and in the Intelligent Children pattern above, the contains relationships in the peers are the rungs of the ladder. However there are dual hierarchies in which the rungs cannot be supported with simple contains relationships. In such hierarchies the peers do not retain their association indefinitely. Instead, peer objects are associated for a period of time and then the association is broken.

If the association of the two peers takes place at the base level, then there is no way that Intelligent Children can be employed to prevent the required downcast. As an example, consider the Command[1] pattern. A **Command** object is generally transient. It is usually associated with other objects that it operates upon. And it is often found in a hierarchy that parallels the objects that it operates on.

I will draw upon the Payroll example from my recent book[1]. See Figure 4. Here we see that there is a hierarchy of **Command** classes. We also see that there is a hierarchy of **Employee** classes. The dual nature of the hierarchy is clearly evident. **EmployeeCommand** is associated with **Employee**, **PostTimeCardCommand** it associated with **HourlyEmployee** and **PostSalesReceiptCommand** is associated with **CommissionedEmployee**.

Note that all the **Employee** objects are contained in a **Dictionary** that associates the objects with an ID. This ID is used by the **EmployeeCommand** object to locate the appropriate **Employee** object to operate upon. Thus the **EmployeeCommand** class is invoking the Template Method[1] pattern by factoring the access of **Employee** objects out of the derived commands. Having fetched the **Employee** object the **EmployeeCommand** class calls its own pure interface

---

[1] *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994 p. 233

[1] *Designing Object Oriented C++ Applications using the Booch Method*, Robert C. Martin, Prentice Hall, 1995.

[1] *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994 p. 325

Solution

Move the contains relationship from the bases to the derivatives. That is, have the derivative D1 contain a reference to D2. Demote the relationships between the bases to a using relationship. Provide a pure interface in the base classes that acts as an accessor function. The derived classes can implement that pure interface.

Structure

See Figure 3.

Applicability

Not all dual hierarchies can yield to this simple approach. Many must procure derived objects from sources that only the base peers have access to. In such cases, INTELLIGENT CHILDREN is not a viable option. However, when it is possible to create the derived peers such that they know about each other, or when the knowledge of a peer can be passed directly to the other peer without going through a base, then INTELLIGENT CHILDREN is a good option.

Sample Code

See Listing 3

Listing 3: Intelligent Children

```
class B1
{
  public:
    virtual B2& GetB2() const = 0;
};

class B2 {};
class D2 : public B2 {};

class D1 : public B1
{
  public:
    D1(D2& d2) : itsD2(d2) {};
    virtual D2& GetB2() const {return itsD2;}

  private:
    mutable D2& itsD2;
};
```

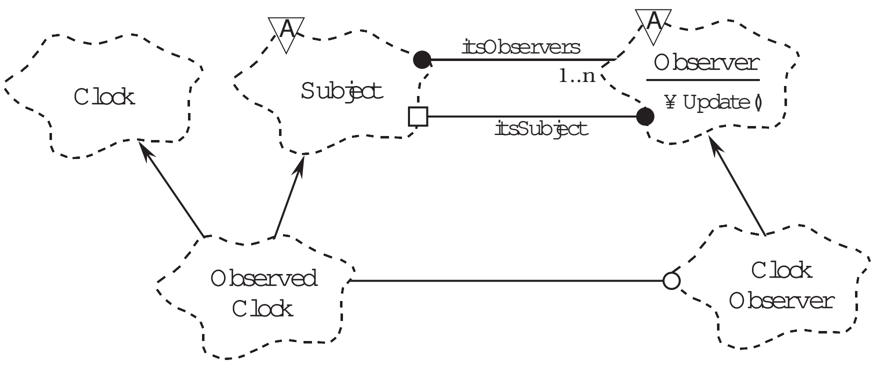
Consequences

The most serious consequence of this pattern is a minor warping of the object model. Consider Figure 3 again. Note that the upper diagram is a better representation of the abstract concept. All B1 objects contain a reference to a B2 object. The presence of the containment at this level makes it impossible for a derivative of B1 *not* to contain a derivative of B2. So by moving the containment to the derived peers we are creating an opening for a B1 derivative that does not contain a B2 derivative.

This is mitigated to some extent by the presence of the pure accessor interface. If B1 has a pure `GetB2()` interface, then it is likely that derivatives of B1 will have some way to procure a B2 instance, even if it is not through containment.

There is a burden placed upon the programmer to remember to include this containment in every derivative of B1.

Figure 2: Clock Observer



Listing 2: ClockObserver

```
class ClockObserver : public Observer
{
  public:
    ClockObserver(ClockSubject& s);
    virtual void Notify();
};

ClockObserver::ClockObserver(ClockSubject& s)
: Observer(s)
{}

void ClockObserver::Notify()
{
    Clock& c = (Clock&) GetSubject(); // ugly downcast
    Display(c.GetTime());
}
```

The typing problem does not exist in dynamically typed languages since the language system does not perform any static checks upon the type. In a dynamically typed language the derived **`Observer`** would simply send the messages that it expects the derived **`Subject`** to be able to respond to.

In statically typed language the typing problem stems from the fact that corresponding elements of the two hierarchies know about each other, but their base classes do not know about the derivatives. Since **`Observer`** contains a reference to a **`Subject`**, derivatives of **`Observer`** must downcast the **`Subject`** to the corresponding kind.

The INTELLIGENT CHILDREN pattern addresses this issue.

INTELLIGENT CHILDREN

Intent

When using a statically typed language, this pattern provides a way to avoid downcasting in some dual inheritance hierarchies.

Motivation

In a dual hierarchy rooted at the base classes B1 and B2, if B1 contains a reference to B2, and if derivatives of B1 gain access to derivatives of B2 through this pointer, then those objects must be downcast before they can be used as their true type. It is best to avoid downcasting where possible.

Typically, the `Observer` derivative must communicate back to the changed object, so it holds a pointer or reference to it.

The code for these classes is shown in Listing 1.

Listing 1: Observer Classes

```
class Observer;

class Subject
{
  public:
    void Register(Observer& o);
    void Notify() const;

  private:
    Set<Observer*> itsObservers;
};

void Subject::Register(Observer& o)
{itsObservers.Add(&o);}

void Subject::Notify() const
{
    for (Iterator<Observer*> i(itsObservers); i; i++)
        (*i)->Notify();
}

class Observer
{
  public:
    Observer(Subject& s) : itsSubject(s) {};
    Subject& GetSubject() const {return itsSubject;};
    virtual void Notify() = 0;
  private:
    mutable Subject& itsSubject;
};
```

Now consider how this pattern might be employed with a `Clock` object (Figure 2) to display the time of day on a screen. `Clock` is a simple concrete class whose task is to keep track of the time. `Clock` is a highly reusable object, so we don't want it to have to know about `Observer`. So we derived `ObservedClock` from both `Clock` and `Subject`.

The methods of `Clock` that change its state are overridden in `ObservedClock` so that they will delegate to `Clock`, and also inform the `Subject` of the state change. The `ClockObserver` is notified when the `ObservedClock` is changed. It then procures the current time from the `Clock` and displays it on a screen.

Figure 2 is a typical dual hierarchy. The `Subject` hierarchy and the `Observer` hierarchy are extremely similar. For every derivative of `Subject` there will be a corresponding derivative of `Observer`. Notice that the reason this dual hierarchy was created was to separate, from `Clock`, any concern about the type of actions, or number of actions that are precipitated by a change in the state of the `Clock` object. It is this separation of concerns that typically causes dual hierarchies to be created.

The problem of static typing.

Consider the code for `ClockObserver` in Listing 2. Note that the `Notify` function of this class must downcast the `Subject` object that it gets from its base class `Observer`. This is the traditional typing problem caused by dual hierarchies in languages that are statically typed.

Design Patterns for Dealing with Dual Inheritance Hierarchies in C++

Robert C. Martin
Object Mentor
rmartin@oma.com

INTRODUCTION

Dual hierarchies are a common element of significant Object-Oriented applications, They arise out of the need to separate concerns. Despite their prevalence, they present problems to the designer that are often solvable only by using techniques that are generally considered unsafe.

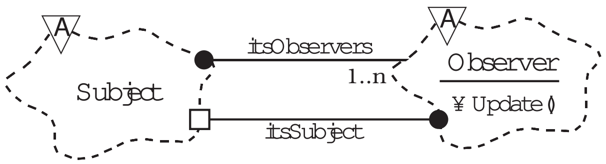
This paper presents three patterns that can be employed to deal with the problems of dual hierarchies. The patterns may be used in isolation, to solve specific application related problems, or they can be used together as a small pattern language in order to more comprehensively address the issues in a larger application.

The patterns are called: RUNGS OF A DUAL HIERARCHY, INTELLIGENT CHILDREN, and STAIRWAY TO HEAVEN. In addition, another pattern, RTTI VISITOR, is presented as an ancillary pattern that supports the others.

WHAT IS A DUAL HIERARCHY?

Consider the OBSERVER[1] pattern (Figure 1). This pattern is often employed when the actions precipitated by the change of an object's state must be disassociated from that object. The changed object derives from the **`Subject`** class which simply sends a **`Notify`** message to the abstract **`Observer`** interface. Derivatives of **`Observer`** implement the appropriate actions.

Figure 1: The Observer Pattern



---

[1] *Design Patterns Elements of Reusable Object Oriented Software*, Gamma, et. al. Addison Wesley, 1994, p. 293

The following paper was originally published in the
Proceedings of the USENIX 1996
Conference on Object-Oriented Technologies
Toronto, Ontario, Canada, June 1996.

Design Patterns for Dealing with Dual Inheritance Hierarchies in C++

Robert Martin
Object Mentor