```
    }
    private String _number;
    private String _areaCode;
```

I begin by declaring all the visible methods on telephone number on person:

```
  class Person...
    String getAreaCode() {
        return _officeTelephone.getAreaCode();
    }
    void setAreaCode(String arg) {
        _officeTelephone.setAreaCode(arg);
    }
    String getNumber() {
        return _officeTelephone.getNumber();
    }
    void setNumber(String arg) {
        _officeTelephone.setNumber(arg);
    }
```

Now I find clients of telephone number and switch them to use the person's interface. So

```
        Person martin = new Person();
        martin.getOfficeTelephone().setAreaCode ("781");
```

becomes

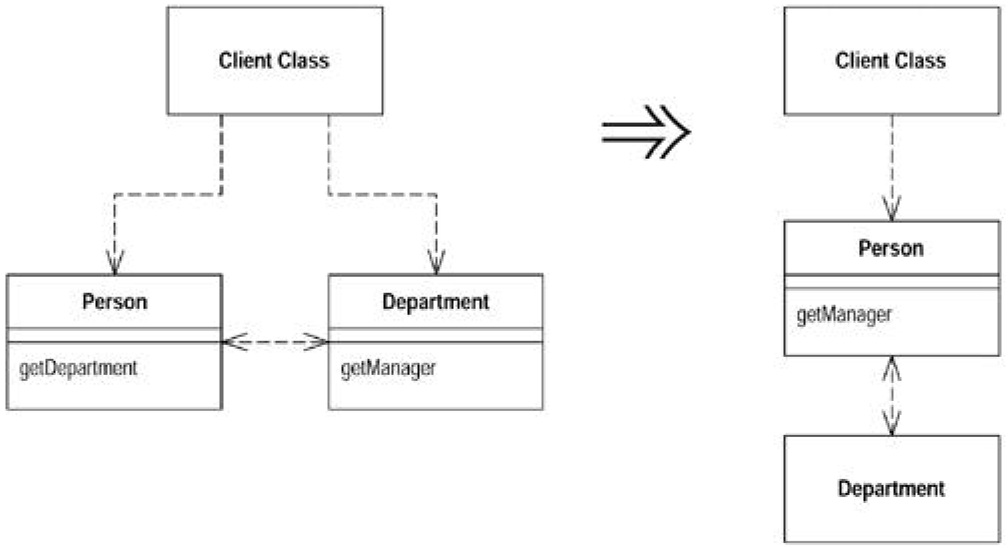
```
        Person martin = new Person();
        martin.setAreaCode ("781");
```

Now I can use Move Method and Move Field until the telephone class is no more.

Hide Delegate

A client is calling a delegate class of an object.

Create methods on the server to hide the delegate.



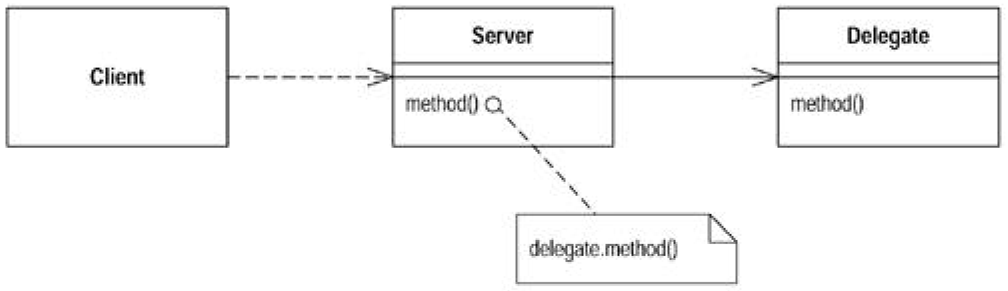
Motivation

One of the keys, if not *the* key, to objects is encapsulation. Encapsulation means that objects need to know less about other parts of the system. Then when things change, fewer objects need to be told about the change—which makes the change easier to make.

Anyone involved in objects knows that you should hide your fields, despite the fact that Java allows fields to be public. As you become more sophisticated, you realize there is more you can encapsulate.

If a client calls a method defined on one of the fields of the server object, the client needs to know about this delegate object. If the delegate changes, the client also may have to change. You can remove this dependency by placing a simple delegating method on the server, which hides the delegate (Figure 7.1). Changes become limited to the server and don't propagate to the client.

Figure 7.1. Simple delegation



You may find it is worthwhile to use Extract Class for some clients of the server or all clients. If you hide from all clients, you can remove all mention of the delegate from the interface of the server.

Mechanics

- For each method on the delegate, create a simple delegating method on the server.

- Adjust the client to call the server.

  ?rarr; *If the client is not in the same package as the server, consider changing the delegate method's access to package visibility.*

- Compile and test after adjusting each method.
- If no client needs to access the delegate anymore, remove the server's accessor for the delegate.
- Compile and test.

Example

I start with a person and a department:

```
class Person {
  Department _department;

  public Department getDepartment() {
      return _department;
  }
  public void setDepartment(Department arg) {
      _department = arg;
  }
}

class Department {
  private String _chargeCode;
  private Person _manager;

  public Department (Person manager) {
      _manager = manager;
  }

  public Person getManager() {
      return _manager;
  }
...
```

If a client wants to know a person's manager, it needs to get the department first:

```
manager = john.getDepartment().getManager();
```

This reveals to the client how the department class works and that the department is responsible to tracking the manager. I can reduce this coupling by hiding the department class from the client. I do this by creating a simple delegating method on person:

```
  public Person getManager() {
      return _department.getManager();
  }
```

I now need to change all clients of person to use this new method:

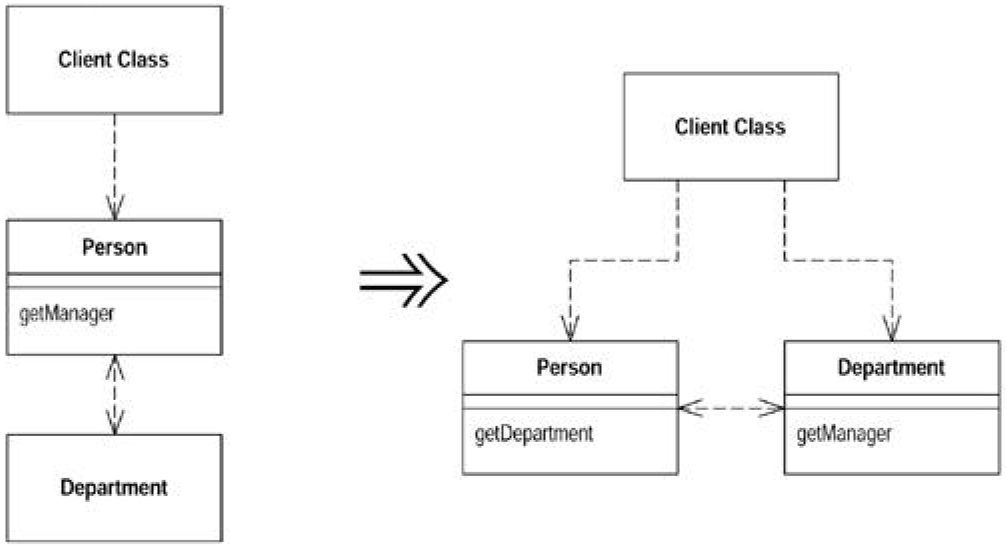
```
  manager = john.getManager();
```

Once I've made the change for all methods of department and for all the clients of person, I can remove the `getDepartment` accessor on person.

Remove Middle Man

A class is doing too much simple delegation.

Get the client to call the delegate directly.



Motivation

In the motivation for *Hide Delegate,* I talked about the advantages of encapsulating the use of a delegated object. There is a price for this. The price is that every time the client wants to use a new feature of the delegate, you have to add a simple delegating method to the server. After adding features for a while, it becomes painful. The server class is just a middle man, and perhaps it's time for the client to call the delegate directly.

It's hard to figure out what the right amount of hiding is. Fortunately, with *Hide Delegate* and Remove Middle Man it does not matter so much. You can adjust your system as time goes on. As the system changes, the basis for how much you hide also changes. A good encapsulation six months ago may be awkward now. Refactoring means you never have to say you're sorry—you just fix it.

Mechanics

- Create an accessor for the delegate.
- For each client use of a delegate method, remove the method from the server and replace the call in the client to call method on the delegate.
- Compile and test after each method.

Example

For an example I use person and department flipped the other way. I start with person hiding the department:

```
class Person...
  Department _department;
  public Person getManager() {
      return _department.getManager();

class Department...
  private Person _manager;
  public Department (Person manager) {
      _manager = manager;
  }
```

To find a person's manager, clients ask:

```
manager = john.getManager();
```

This is simple to use and encapsulates the department. However, if lots of methods are doing this, I end up with too many of these simple delegations on the person. That's when it is good to remove the middle man. First I make an accessor for the delegate:

```
class Person...
  public Department getDepartment() {
      return _department;
  }
```

Then I take each method at a time. I find clients that use the method on person and change it to first get the delegate. Then I use it:

```
manager = john.getDepartment().getManager();
```

I can then remove `getManager` from person. A compile shows whether I missed anything.

I may want to keep some of these delegations for convenience. I also may want to hide the delegate from some clients but show it to others. That also will leave some of the simple delegations in place.

Introduce Foreign Method

A server class you are using needs an additional method, but you can't modify the class.

Create a method in the client class with an instance of the server class as its first argument.

```
Date newStart = new Date (previousEnd.getYear(),
                  previousEnd.getMonth(), previousEnd.getDate() + 1);
```

⇓

I begin the delegation by creating a field for the delegated vector. I link this field to `this` so that I can mix delegation and inheritance while I carry out the refactoring:

```
  private Vector _vector = this;
```

Now I start replacing methods to get them to use the delegation. I begin with push:

```
      public void push(Object element) {
        _vector.insertElementAt(element,0);
    }
```

I can compile and test here, and everything will still work. Now pop:

```
      public Object pop() {
        Object result = _vector.firstElement();
        _vector.removeElementAt(0);
        return result;
    }
```

Once I've completed these subclass methods, I need to break the link to the superclass:

```
 class MyStack extends Vector

   private Vector _vector = new Vector();
```

I then add simple delegating methods for superclass methods used by clients:

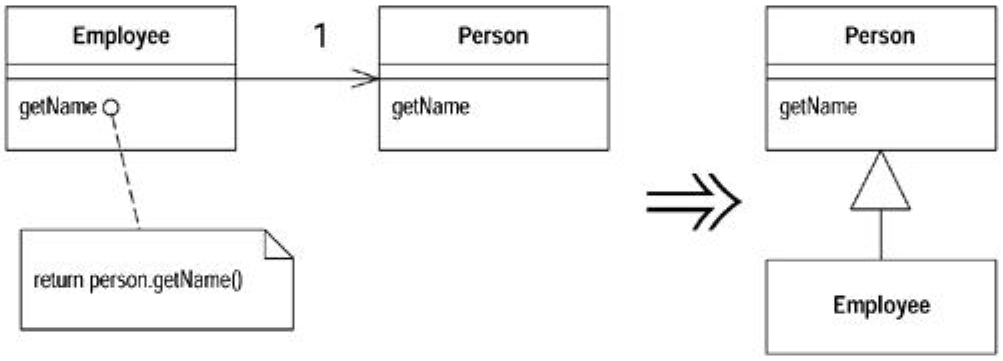
```
   public int size() {
       return _vector.size();
   }
   public boolean isEmpty() {
       return _vector.isEmpty();
   }
```

Now I can compile and test. If I forgot to add a delegating method, the compilation will tell me.

Replace Delegation with Inheritance

You're using delegation and are often writing many simple delegations for the entire interface.

Make the delegating class a subclass of the delegate.



Motivation

This is the flip side of Replace Delegation with Inheritance. If you find yourself using all the methods of the delegate and are sick of writing all those simple delegating methods, you can switch back to inheritance pretty easily.

There are a couple of caveats to bear in mind. If you aren't using all the methods of the class to which you are delegating, you shouldn't use *Replace Delegation with Inheritance,* because a subclass should always follow the interface of the superclass. If the delegating methods are tiresome, you have other options. You can let the clients call the delegate themselves with Remove Middle Man. You can use Extract Superclass to separate the common interface and then inherit from the new class. You can use Extract Interface in a similar way.

Another situation to beware of is that in which the delegate is shared by more than one object and is mutable. In this case you can't replace the delegate with inheritance because you'll no longer share the data. Data sharing is a responsibility that cannot be transferred back to inheritance. When the object is immutable, data sharing is not a problem, because you can just copy and nobody can tell.

Mechanics

- Make the delegating object a subclass of the delegate.
- Compile.

> ?rarr; *You may get some method clashes at this point; methods may have the same name but vary in return type, exceptions, or visibility. Use* Rename Method *to fix these.*

- Set the delegate field to be the object itself.
- Remove the simple delegation methods.
- Compile and test.
- Replace all other delegations with calls to the object itself.
- Remove the delegate field.

Example

A simple employee delegates to a simple person:

```
class Employee {
  Person _person = new Person();

  public String getName() {
      return _person.getName();
  }
  public void setName(String arg) {
       _person.setName(arg);
  }
  public String toString () {
      return "Emp: " + _person.getLastName();
  }
}

class Person {
  String _name;

  public String getName() {
      return _name;
  }
  public void setName(String arg) {
      _name = arg;
  }
  public String getLastName() {
      return _name.substring(_name.lastIndexOf(' ')+1);
  }
}
```

The first step is just to declare the subclass:

```
class Employee extends Person
```

Compiling at this point alerts me to any method clashes. These occur if methods with the name have different return types or throw different exceptions. Any such problems need to be fixed with Rename Method. This simple example is free of such encumbrances.

The next step is to make the delegate field refer to the object itself. I must remove all simple delegation methods such as `getName` and `setName.` If I leave any in, I will get a stack overflow error caused by infinite recursion. In this case this means removing `getName` and `setName` from `Employee.`

Once I've got the class working, I can change the methods that use the delegate methods. I switch them to use calls directly:

```
  public String toString () {
       return "Emp: " + getLastName();
```

```
    }
```

Once I've got rid of all methods that use delegate methods, I can get rid of the `_person` field.

The exact code in the controlling modifier varies with the multiplicity of the association. If the customer is not allowed to be null, I can forgo the null checks, but I need to check for a null argument. The basic pattern is always the same, however: first tell the other object to remove its pointer to you, set your pointer to the new object, and then tell the new object to add a pointer to you.

If you want to modify the link through the customer, let it call the controlling method:

```
  class Customer...
    void addOrder(Order arg) {
        arg.setCustomer(this);
    }
```

If an order can have many customers, you have a many-to-many case, and the methods look like this:

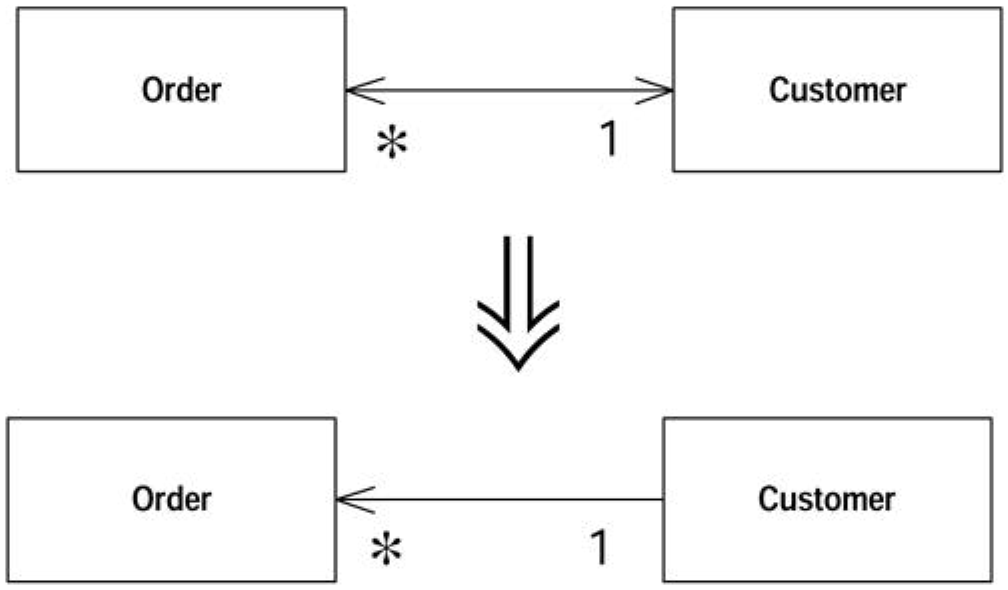
```
  class Order... //controlling methods
    void addCustomer (Customer arg) {
        arg.friendOrders().add(this);
        _customers.add(arg);
    }
    void removeCustomer (Customer arg) {
        arg.friendOrders().remove(this);
        _customers.remove(arg);
    }

class Customer...
    void addOrder(Order arg) {
        arg.addCustomer(this);
    }
    void removeOrder(Order arg) {
        arg.removeCustomer(this);
    }
```

Change Bidirectional Association to Unidirectional

You have a two-way association but one class no longer needs features from the other.

Drop the unneeded end of the association.



Motivation

Bidirectional associations are useful, but they carry a price. The price is the added complexity of maintaining the two-way links and ensuring that objects are properly created and removed. Bidirectional associations are not natural for many programmers, so they often are a source of errors.

Lots of two-way links also make it easy for mistakes to lead to zombies: objects that should be dead but still hang around because of a reference that was not cleared.

Bidirectional associations force an interdependency between the two classes. Any change to one class may cause a change to another. If the classes are in separate packages, you get an interdependency between the packages. Many interdependencies lead to a highly coupled system, in which any little change leads to lots of unpredictable ramifications.

You should use bidirectional associations when you need to but not when you don't. As soon as you see a bidirectional association is no longer pulling its weight, drop the unnecessary end.

Mechanics

- Examine all the readers of the field that holds the pointer that you wish to remove to see whether the removal is feasible.

  ?rarr; *Look at direct readers and further methods that call the methods.*

  ?rarr; *Consider whether it is possible to determine the other object without using the pointer. If so you will be able to use* Substitute Algorithm *on the getter to allow clients to use the getting method even if there is no pointer.*

  ?rarr; *Consider adding the object as an argument to all methods that use the field.*

- If clients need to use the getter, use Self Encapsulate Field, carry out Substitute Algorithm on the getter, compile, and test.
- If clients don't need the getter, change each user of the field so that it gets the object in the field another way. Compile and test after each change.
- When no reader is left in the field, remove all updates to the field, and remove the field.

> ?rarr; *If there are many places that assign the field, use* Self Encapsulate Field *so that they all use a single setter. Compile and test. Change the setter to have an empty body. Compile and test. If that works, remove the field, the setter, and all calls to the setter.*

- Compile and test.

Example

I start from where I ended up from the example in Change Unidirectional Association to Bidirectional. I have a customer and order with a bidirectional link:

```
class Order...
  Customer getCustomer() {
      return _customer;
  }
  void setCustomer (Customer arg) {
      if (_customer != null) _customer.friendOrders().remove(this);
      _customer = arg;
      if (_customer != null) _customer.friendOrders().add(this);
  }
  private Customer _customer;

class Customer...
  void addOrder(Order arg) {
      arg.setCustomer(this);
  }
  private Set _orders = new HashSet();
  Set friendOrders() {
      /** should only be used by Order */
      return _orders;
  }
```

I've found that in my application I don't have orders unless I already have a customer, so I want to break the link from order to customer.

The most difficult part of this refactoring is checking that I can do it. Once I know it's safe to do, it's easy. The issue is whether code relies on the customer field's being there. To remove the field, I need to provide an alternative.

My first move is to study all the readers of the field and the methods that use those readers. Can I find another way to provide the customer object? Often this means passing in the customer as an argument for an operation. Here's a simplistic example of this:

```
class Order...
      double getDiscountedPrice() {
      return getGrossPrice() * (1 - _customer.getDiscount());
  }
```

changes to

```
  class Order...
    double getDiscountedPrice(Customer customer) {
        return getGrossPrice() * (1 - customer.getDiscount());
    }
```

This works particularly well when the behavior is being called by the customer, because then it's easy to pass itself in as an argument. So

```
  class Customer...
    double getPriceFor(Order order) {
        Assert.isTrue(_orders.contains(order)); // see Introduce
Assertion (267)
        return order.getDiscountedPrice();
```

becomes

```
  class Customer...
    double getPriceFor(Order order) {
        Assert.isTrue(_orders.contains(order));
        return order.getDiscountedPrice(this);
    }
```

Another alternative I consider is changing the getter so that it gets the customer without using the field. If it does, I can use Substitute Algorithm on the body of `Order.getCustomer.` I might do something like this:

```
    Customer getCustomer() {
        Iterator iter = Customer.getInstances().iterator();
        while (iter.hasNext()) {
            Customer each = (Customer)iter.next();
            if (each.containsOrder(this)) return each;
        }
        return null;
    }
```

Slow, but it works. In a database context it may not even be that slow if I use a database query. If the order class contains methods that use the customer field, I can change them to use `getCustomer` by using Self Encapsulate Field.

If I retain the accessor, the association is still bidirectional in interface but is unidirectional in implementation. I remove the backpointer but retain the interdependencies between the two classes.

If I substitute the getting method, I substitute that and leave the rest till later. Otherwise I change the callers one at a time to use the customer from another source. I compile and test after each change. In practice, this process usually is pretty rapid. If it were complicated, I would give up on this refactoring.

Once I've eliminated the readers of the field, I can work on the writers of the field. This is as simple as removing any assignments to the field and then removing the field. Because nobody is reading it any more, that shouldn't matter.

Replace Magic Number with Symbolic Constant

You have a literal number with a particular meaning.

Create a constant, name it after the meaning, and replace the number with it.

```
double potentialEnergy(double mass, double height) {
    return mass * 9.81 * height;
}
```

⇓

```
double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

Motivation

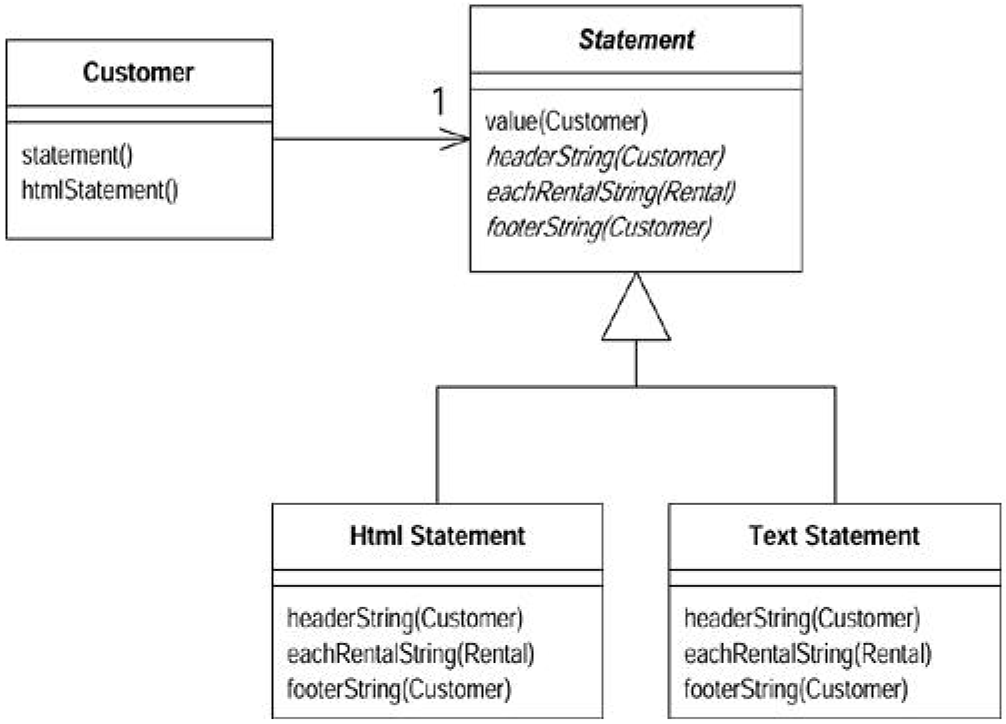
Magic numbers are one of oldest ills in computing. They are numbers with special values that usually are not obvious. Magic numbers are really nasty when you need to reference the same logical number in more than one place. If the numbers might ever change, making the change is a nightmare. Even if you don't make a change, you have the difficulty of figuring out what is going on.

Many languages allow you to declare a constant. There is no cost in performance and there is a great improvement in readability.

Before you do this refactoring, you should always look for an alternative. Look at how the magic number is used. Often you can find a better way to use it. If the magic number is a type code, consider Replace Type Code with Class. If the magic number is the length of an array, use `anArray.length` instead when you are looping through the array.

Mechanics

- Declare a constant and set it to the value of the magic number.
- Find all occurrences of the magic number.
- See whether the magic number matches the usage of the constant; if it does, change the magic number to use the constant.
- Compile.
- When all magic numbers are changed, compile and test. At this point all should work as if nothing has been changed.

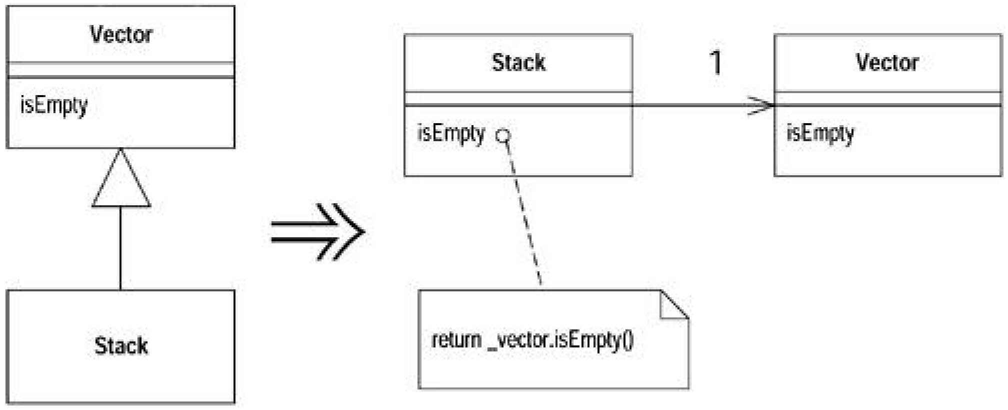


After this refactoring, it is easy to add new kinds of statements. All you have to do is create a subclass of statement that overrides the three abstract methods.

Replace Inheritance with Delegation

A subclass uses only part of a superclasses interface or does not want to inherit data.

Create a field for the superclass, adjust methods to delegate to the superclass, and remove the subclassing.



Motivation

Inheritance is a wonderful thing, but sometimes it isn't what you want. Often you start inheriting from a class but then find that many of the superclass operations aren't really true of the subclass. In this case you have an interface that's not a true reflection of what the class does. Or you may find that you are inheriting a whole load of data that is not appropriate for the subclass. Or you may find that there are protected superclass methods that don't make much sense with the subclass.

You can live with the situation and use convention to say that although it is a subclass, it's using only part of the superclass function. But that results in code that says one thing when your intention is something else—a confusion you should remove.

By using delegation instead, you make it clear that you are making only partial use of the delegated class. You control which aspects of the interface to take and which to ignore. The cost is extra delegating methods that are boring to write but are too simple to go wrong.

Mechanics

- Create a field in the subclass that refers to an instance of the superclass. Initialize it to `this.`
- Change each method defined in the subclass to use the delegate field. Compile and test after changing each method.

  > ?rarr; *You won't be able to replace any methods that invoke a method on* `super` *that is defined on the subclass, or they may get into an infinite recurse. These methods can be replaced only after you have broken the inheritance.*

- Remove the subclass declaration and replace the delegate assignment with an assignment to a new object.
- For each superclass method used by a client, add a simple delegating method.
- Compile and test.

Example

One of the classic examples of inappropriate inheritance is making a stack a subclass of vector. Java 1.1 does this in its utilities (naughty boys!), but in this case I use a simplified form of stack:

```
class MyStack extends Vector {

   public void push(Object element) {
       insertElementAt(element,0);
   }

   public Object pop() {
       Object result = firstElement();
       removeElementAt(0);
       return result;
   }
}
```

Looking at the users of the class, I realize that clients do only four things with stack: push, pop, size, and `isEmpty.` The latter two are inherited from Vector.

I begin the delegation by creating a field for the delegated vector. I link this field to `this` so that I can mix delegation and inheritance while I carry out the refactoring:

```
  private Vector _vector = this;
```

Now I start replacing methods to get them to use the delegation. I begin with push:

```
      public void push(Object element) {
        _vector.insertElementAt(element,0);
    }
```

I can compile and test here, and everything will still work. Now pop:

```
      public Object pop() {
        Object result = _vector.firstElement();
        _vector.removeElementAt(0);
        return result;
    }
```

Once I've completed these subclass methods, I need to break the link to the superclass:

```
 class MyStack extends Vector

   private Vector _vector = new Vector();
```

I then add simple delegating methods for superclass methods used by clients:

```
   public int size() {
       return _vector.size();
   }
   public boolean isEmpty() {
       return _vector.isEmpty();
   }
```

Now I can compile and test. If I forgot to add a delegating method, the compilation will tell me.

Replace Delegation with Inheritance