Chapter 9: Organizing Data

In this chapter I'll discuss several refactorings that make working with data easier. For many people *Self Encapsulate Field (185)* will seem unnecessary. It's long been a matter of good-natured debate about whether an object should access its own data directly or through accessors. Sometimes you do need the accessors and then you can get them with *Self Encapsulate Field (185)*. I generally use direct access as I find it simple to do this refactoring when I need it.

One of the useful things about object languages is that they allow you to define new types that go beyond what can be done with the simple data types of traditional languages. It takes a while to get used to how to do this, however, and often you start with a simple data value and then realize that an object would be more useful. *Replace Data Value with Object (189)* allows you to turn dumb data into articulate objects. When you realize that these objects are instances that will be needed in many parts of the program then you can use *Change Value to Reference (193)* to make them into reference objects. If you see an array acting as a data structure, you can make the data structure clearer with *Replace Array with Object (197)*. In all these cases the object is but the first step - the real advantage comes as you use *Move Method (160)* to add behavior to the new objects.

Magic numbers, numbers with special meaning, have long been a problem. I remember being told not to use them in my earliest programming days. They do keep appearing, however, and I use *Replace Magic Number with Symbolic Constant (217)* to get rid of them whenever I figure out what they are doing.

Links between objects can be one-way or two-way. One-way links are easier, but sometimes you need to *Change Unidirectional Association to Bidirectional (209)* to support new function. *Change Bidirectional Associa-*

tion to Unidirectional (213) removes unnecessary complexity should you find you don't need this any more.

I've often run into cases where gui classes are doing business logic that they shouldn't. To move the behavior into proper domain classes you need to have the data in the domain class as well as supporting the gui by using *Duplicate Data From Presentation to Domain (201).* Normally I don't like duplicating data, but this is an exception that is usually impossible to avoid.

One of the key tenets of object-oriented programming is encapsulation. So if there's any public data streaking around, you can use *Encapsulate Field (219)* to decorously cover it up. If that data is a collection then use *Encapsulate Collection (221)* instead as that has special protocol. If you have a whole naked record then use *Replace Record with Data Class (228)*

One form of data that requires particular treatment is the type code: some special value that indicates something particular about what type of instance we are dealing with. These often show up as enumerations, often implemented as static final integers. If the codes are for information and do not alter the behavior of the class, then you can use *Replace Type Code with Class (229),* which gives you better type checking and a platform for moving behavior later. If the behavior of a class is affected by a type code then use *Replace Type Code with Subclasses (235)* if possible, or if you can't do that use the more complicated (but more flexible) *Replace Type Code with State/Strategy (239).*

Self Encapsulate Field

You are using accessing a field directly, but the coupling to the field is becoming awkward.

Create Getting and Setting Methods for the field and use only those to access the field

```
private int _low, _high;
boolean includes (int arg) {
    return arg >= _low && arg <= _high;
}
```

```
private int _low, _high;
boolean includes (int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() {return _low;}
int getHigh() {return _high;}
```

Motivation

When it comes to accessing fields there are two schools of thought. One is that within the class where the variable is defined, you should access it the variable freely (direct variable access). The other says that even within the class you should always use accessors (indirect variable access). Debates between the two can get heated. (see also the discussion in [Beck].

Essentially the advantages of indirect variable access are

- it allows a subclass to override how to get that information with a method

- it supports more flexibility in managing the data, such as lazy initialization which only initializes the value when you need to use it.

The advantage of direct variable access is

- the code is easier to read - you don't need to stop and say "this is just a getting method"

I'm always in two minds with this choice, so I'm usually happy to do what the rest of the team wants to do. Left to myself though I like to use direct variable access as a first resort, until it gets in the way. Once things start becoming awkward I switch to indirect variable access. Refactoring gives you the freedom to change your mind.

The most important time to do this is when you are accessing a field in a superclass, but you want to override this variable access with a computed value in the subclass. Self-encapsulating the field is the first step, after that you can override the Getting and Setting methods as you need to.

Mechanics

- Create a Getting and Setting Method for the field
- Find all references to the field and replace them with a getting or setting method
  - For accesses to the field, replace with a call to the getting method, for assignments replace it with a call to the setting method.
  - You can't entirely rely on the compiler in a strongly typed language here, as it is not an error to refer to the field in its own class.
- Make the field private.
  - Smalltalk cannot do this (all subclasses can see a superclass variable). Making a field private will allow the compiler to catch any subclass using the field, but the compiler will still not catch references with the field's class
- Double check you have caught all references
- Compile and Test

Example

This seems almost too simple for an example, but hey at least it'll be quick to write.

```
class IntRange {

  private int _low, _high;
```

```
boolean includes (int arg) {
    return arg >= _low && arg <= _high;
}

void grow(int factor) {
    _high = _high * factor;
}

IntRange (int low, int high) {
    _low = low;
    _high = high;
}
```

To self encapsulate I define getting and setting methods (if they don't already exist) and use those.

```
class IntRange {

  boolean includes (int arg) {
      return arg >= getLow() && arg <= getHigh();
  }

  void grow(int factor) {
      setHigh (getHigh() * factor);
  }

  private int _low, _high;

  int getLow() {
      return _low;
  }

  int getHigh() {
      return _high;
  }

  void setLow(int arg) {
      _low = arg;
  }

  void setHigh(int arg) {
      _high = arg;
  }
```

In cases like this you have to be more careful about using the setting method in the constructor. Often it is assumed that you use the setting method for changes after the object is created, so the setting method has different semantics. So in cases like this I prefer a separate initialization method.

```
  IntRange (int low, int high) {
      initialize (low, high);
  }

  private void initialize (int low, int high) {
      _low = low;
      _high = high;
  }
```

The value in doing all this comes when you have a subclass.

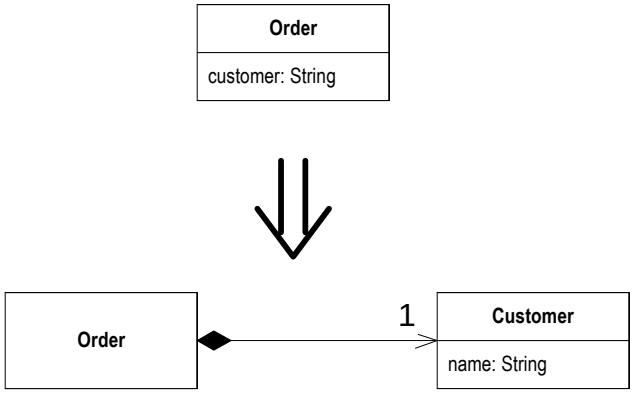
```
class CappedRange extends IntRange {

  CappedRange (int low, int high, int cap) {
      super (low, high);
      _cap = cap;
  }

  private int _cap;

  int getCap() {
      return _cap;
  }

  int getHigh() {
      return Math.min(super.getHigh(), getCap());
  }
}
```

I can override all of IntRange's behavior to take into account the cap, without changing any of that behavior.

Replace Data Value with Object

You have a data item that needs additional data or behavior

Turn the data item into an object



Motivation

Often in early stages of development you make decisions about representing simple facts as simple data items. As development proceeds you realize that those simple items aren't so simple any more. A telephone number may be represented as a string for a while, but later on you realize that the telephone needs special behavior for formatting, extracting the area code and the like. For one or two items you may just put the methods in the owning object, but quickly the code smells of duplication and feature envy.

When the smell begins turn the data value into an object.

Mechanics

- Create the class for the value. Give it a final field of the same type as the value in the source class. Add an getter and a constructor that

takes the field as an argument.

- ❍ Compile
- ❍ Change the type of the field in the source class to the new class
- ❍ Change the getter in the source class to call the getter in the new class
- ❍ If the field is mentioned in the source class constructor, assign the field using the new class's constructor
- ❍ Change the getting method to create a new instance of the new class.
- ❍ Compile and test.
- ❍ You may now need to use *Change Value to Reference (193)* on the new object.

Example

I'll start with an order class that's stored the customer of the order as a string and would like to turn the customer into an object. This way we have somewhere to store data such as an address, credit rating and the like (as well as useful behavior that uses this information).

```
class Order...
  public Order (String customer) {
      _customer = customer;
  }
  public String getCustomer() {
      return _customer;
  }
  public void setCustomer(String arg) {
      _customer = arg;
  }
  private String _customer;
```

Some client code that uses this looks like

```
 int numberOfOrdersFor(Vector listOfOrders, String customer) {
    int result = 0;
    Enumeration e = listOfOrders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        if (each.getCustomer().equals(customer)) result++;
    }
    return result;
}
```

First I create the new customer class. I give it a final field for a string attribute, as that is that the order currently uses. I call it name, since that seems to be what the string is used for. I also add a getting method and provide a constructor that uses the attribute.

```
class Customer {
  public Customer (String name) {
      _name = name;
  }
  public String getName() {
      return _name;
  }
  private final String _name;
}
```

Now I change the type of the customer field and change methods that reference it to use the appropriate references on the customer class. The getter and constructor are obvious. For the setter I create a new customer.

```
class Order...
  public Order (String customer) {
      _customer = new Customer(customer);
  }
  public String getCustomer() {
      return _customer.getName();
  }
  private Customer _customer;

  public void setCustomer(String arg) {
      _customer = new Customer(customer);
  }
```

The setter creates a new customer because the old string attribute was a value objects, and thus the customer currently is also a value object. This means that each order has its own customer objects. As a rule value objects should be immutable — this avoids some nasty aliasing bugs. Later on we will want customer to be a reference object, but that's another refactoring.

At this point we can compile and test.

Now we should look at the methods on order that manipulate customer and make some changes to make the new state of affairs clearer.

With the getter we use *Rename Method (235)*

```
  public String getCustomerName() {
      return _customer.getName();
```

```
}
```

On the constructor and setter, we don't need to change the signature, but the name of the arguments should change.

```
public Order (String customerName) {
    _customer = new Customer(customerName);
}
public void setCustomer(String customerName) {
    _customer = new Customer(customerName);
}
```

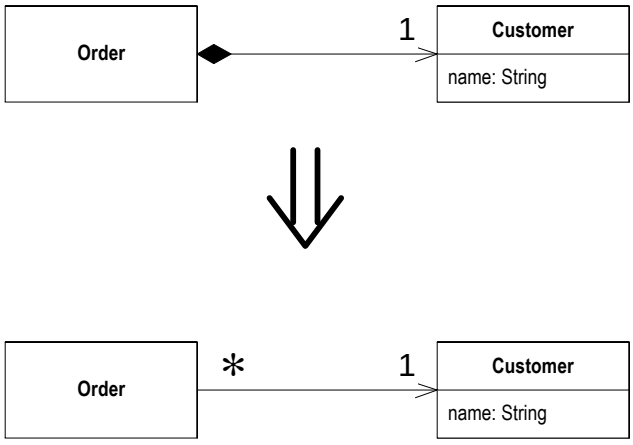
Further refactoring may well cause us to add a new constructor and setter that takes an existing customer.

This finishes this refactoring, but in this case, as in many others, there is another step. If we wish to add such things as credit ratings and addresses to our customer, we cannot do so now. This is because the customer is treated as a value object. Each customer has it's own customer object. To give a customer these attributes we need to apply *Change Value to Reference (193)* to the customer. You'll find this example continued there....

Change Value to Reference

You have a class with many equal instances that you want to replace with a single object

Turn the object into a reference object



Motivation

There is a useful classification you can make of objects in many systems: reference objects and value objects. A reference object is something like customer, or account. Each object stands for one object in the real world. You don't usually copy reference objects, and you use the object identity to test if they are equal. Value objects are things like date, or money. They are entirely defined through their data values. You don't mind copying them, you may have hundreds of "1/1/2000" objects around your system. You do need to tell if two of them are equal, so you need to override the equals method (and the hashCode method too.)

The decision between reference and value is not always clear cut. Sometimes you start with a value and want to give it some data that really makes it a reference object. You can do this by turning it into a reference object.

Mechanics

- Use *Replace Constructor with Factory Method (263).*
- Compile and test
- Decide what object is responsible for providing access to the objects
  - This may be a static dictionary or a registry object.
  - You may have more than one object that acts as an access point for the new object.
- Decide whether the objects are pre-created or created on the fly.
  - If they are pre-created and you are retrieving them from memory you need to ensure they are loaded before they are needed.
- Alter the factory method to return the reference object.
  - If the objects are pre-computed, you need to decide how to handle errors if someone asks for one that does not exist.
  - You may wish to *Rename Method (235)* on the factory to convey that it returns an existing object.
- Compile and test

Example

I'll start with where I left off in the example for *Replace Data Value with Object (189).* I have the following customer class.

```
class Customer {
  public Customer (String name) {
      _name = name;
  }
  public String getName() {
      return _name;
  }
  private final String _name;
}
```

It is used by an order class.

```
class Order...
  public Order (String customerName) {
      _customer = new Customer(customerName);
  }
  public void setCustomer(String customerName) {
      _customer = new Customer(customerName);
  }
  public String getCustomerName() {
```

```
        return _customer.getName();
    }
    private Customer _customer;
```

and some client code

```
    int numberOfOrdersFor(Vector listOfOrders, String customerName) {
        int result = 0;
        Enumeration e = listOfOrders.elements();
        while (e.hasMoreElements()) {
            Order each = (Order) e.nextElement();
            if (each.getCustomer().equals(customerName)) result++;
        }
        return result;
    }
```

At the moment it is a value. Each order has it's own customer object even if they are for the same concpeptual customer. I want to change this so that if we have several orders for the same conceptual customer, they share a single customer object. For this case this means that there should only be one customer object for each customer name.

I begin by using *Replace Constructor with Factory Method (263)*. I define the factory method on customer.

```
class Customer {
  public static Customer create (String name) {
      return new Customer(name);
  }
```

Then replace the calls to the constructor with calls to the factory.

```
    class Order {
    public Order (String customer) {
        _customer = Customer.create(customer);
    }
```

Then I make the constructor private.

```
class Customer {
  private Customer (String name) {
      _name = name;
  }
```

Now I have to decide how I access the customers. My preference is to use another object. Such a situation works well with something like the line items on an order. The order is responsible for providing access to the line items. However here there isn't such an obvious object. In this situation I usually create a registry object to be the access point. For

simplicity in this example however, I will store them using a static field on customer, making the customer class the access point.

```
private static Dictionary _instances = new Hashtable();
```

Then I decide whether I create customers on the fly when asked, or whether they are pre-created. I'll use the latter. In my application start-up code I'll load up the customers that are in use. These could come from a database or from a file, but again for simplicity I'll use explicit code. I can always use *Substitute Algorithm (143)* to change it later.

```
class Customer...
  static void loadCustomers() {
      new Customer ("Lemon Car Hire").store();
      new Customer ("Associated Coffee Machines").store();
      new Customer ("Bilston Gasworks").store();
  }
  private void store() {
      _instances.put(this.getName(), this);
  }
```

Now I'll alter the factory method to return the pre-created customer.

```
public static Customer create (String name) {
    return (Customer) _instances.get(name);
}
```

Since the create method always returns an existing customer, I should make this clear by using *Rename Method (235)*.

```
class Customer...
  public static Customer getNamed (String name) {
      return (Customer) _instances.get(name);
  }
```

Replace Array with Object

You have an array where certain elements mean different things

Replace the array with an object, with a field for each element

```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

Motivation

Array's are a common structure for organizing data. However they should only be used to contain a collection of similar objects in some order. Sometimes, however you see them used to contain a number of different things. Conventions such as "the first element on the array is the person's name" are hard to remember. With an object you can use names of fields and methods to convey this information so you don't have remember it, or hope the comments are up to date. You can also encapsulate the information, and use *Move Method (160)* to add behavior to it.

Mechanics

- Create a new class to represent the information in the array. Give it a public field for the array
- Change all users of the array to use the new class
- Compile and test
- One by one, add getters and setters for each element of the array.

Name the accessors after the purpose of the array element. Change the clients to use the accessors. Compile and test after each change

- When all array accesses are replaced by methods, make the array private
- Compile
- For each element of the array, create a field in the class and change the accessors to use the field.
- Compile and test after each element is changed
- When all elements have been replaced with fields, delete the array.

Example

I'll start with an array that's used to hold the name, wins, and losses of a sports team. It would be declared as

```
String[] row = new String[3];
```

It would be used with code like

```
row [0] = "Liverpool";
row [1] = "15";

String name = row[0];
int wins = Integer.parseInt(row[1]);
```

To turn this into an object, we begin by creating a class.

```
class Performance {}
```

For our first step we give the new class a public data member. (I know this is evil and wicked, but I'll reform in due course.)

```
    public String[] _data = new String[3];
```

Now I find the spots that create and access the array. Where the array is created I use

```
Performance row = new Performance();
```

Where it is used, I change to

```
row._data [0] = "Liverpool";
row._data  [1] = "15";

String name = row._data[0];
int wins = Integer.parseInt(row._data[1]);
```

Now, one by one, I add more meaningful getters and setters. I start with the name.

```
class Performance...
  public String getName() {
      return _data[0];
  }
  public void setName(String arg) {
      _data[0] = arg;
  }
```

I alter the users of that row to use the getters and setters instead,

```
row.setName("Liverpool");
row._data  [1] = "15";

String name = row.getName();
int wins = Integer.parseInt(row._data[1]);
```

I can do the same with the second element. To make matters easier I can encapsulate the data type conversion as well.

```
class Performance...
  public int getWins() {
      return Integer.parseInt(_data[1]);
  }
  public void setWins(String arg) {
      _data[1] = arg;
  }

....
client code...
  row.setName("Liverpool");
  row.setWins("15");

  String name = row.getName();
  int wins = row.getWins();
```

Once I've done this for each element, I can make the array private.

```
private String[] _data = new String[3];
```

The most important part of this refactoring, changing the interface, is now done. However it is also useful to replace the array internally. I can do this by adding a field for each array element, and changing the accessors to use it.

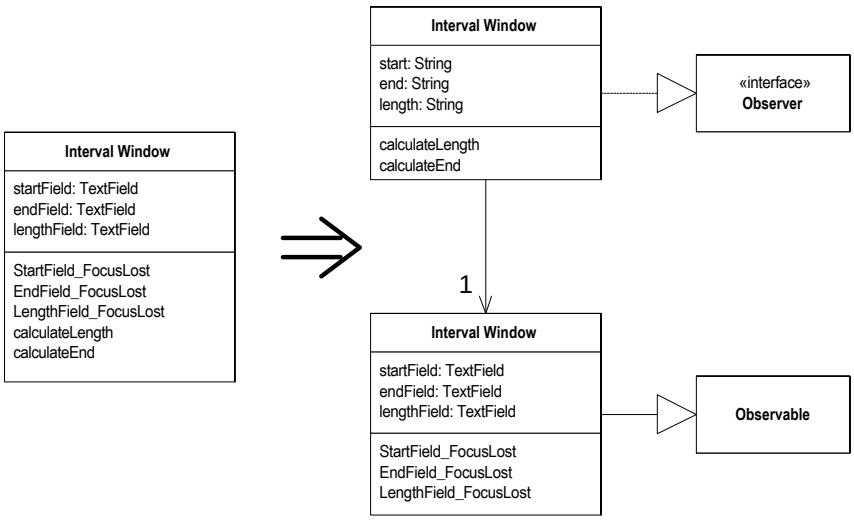
```
class Peformance...
  public String getName() {
      return _name;
  }
  public void setName(String arg) {
      _name = arg;
  }
  private String _name;
```

I do this for each element in the array. When I've done them all I delete the array.

Duplicate Data From Presentation to Domain

You have domain data embedded in a gui control

Copy the data to a domain object. Set up a mechanism to synchronize the two pieces of data.



Motivation

A well layered system will separate the code that handles the user interface from code that handles the business logic. It does this for several reasons: you may want several interfaces for similar business logic, the user-interface gets too complicated if it does both, it is easier to maintain and evolve domain objects separate from the gui, or you may have different developers handling these different pieces.

Although the behavior can be separated easily, the data often cannot. Data needs to be embedded in gui controls which has the same meaning as data that lives in the domain model. UI frameworks, from MVC onwards, that are used to a multi-tiered system provide mechanisms to allow you to provide this data and keep everything in sync.

If you come across that has been developed within a two tier approach, where business logic is embedded into the UI, you will need to separate out this behavior. Much of this is about decomposing and moving methods. For the data however, you cannot just move the data, you have to duplicate it and provide the synchronization mechanism.

Mechanics

- Make the presentation class an observer [Gang of Four] of the domain class
  - If there is no domain class yet, create one.
  - If there is no link from the presentation class to the domain class, put the domain class in a field of the presentation class.
- Use *Self Encapsulate Field (185)* on the domain data within the gui class.
- Compile and test
- Add a call to the setting method in the event handler, to update the component with its current value using direct access.
  - That is put a method in the event handler that updates the value of the component based on its current value. Of course this is completely unnecessary, you are just setting the value to its current value, but by using the setting method you allow any behavior there to execute.
  - When you do this change, don't use the getting method for the component, use direct access to the component. Later on the getting method will pull the value from the domain, which does not change until the setting method executes.
  - Make sure the event handling mechanism is triggered by the test code
- Compile and test
- Define the data and accessor methods in the domain class
  - Make sure the setting method on the domain triggers the notify mechanism in the observer pattern.
  - Use the same data type in the domain as was on the presentation (usually a string). Convert the data type in a later refactoring.
- Redirect the accessors to write to the domain field.
- Modify the observer's update method to copy the data from the domain field to the gui control.
- Compile and test

Example

I'll start with the window in Figure 9.1. The behavior is very simple. Whenever you change the value in one of the text fields, the other ones update. If you change the start or end fields, it calculates the length, if you change the length field it calculates the end.

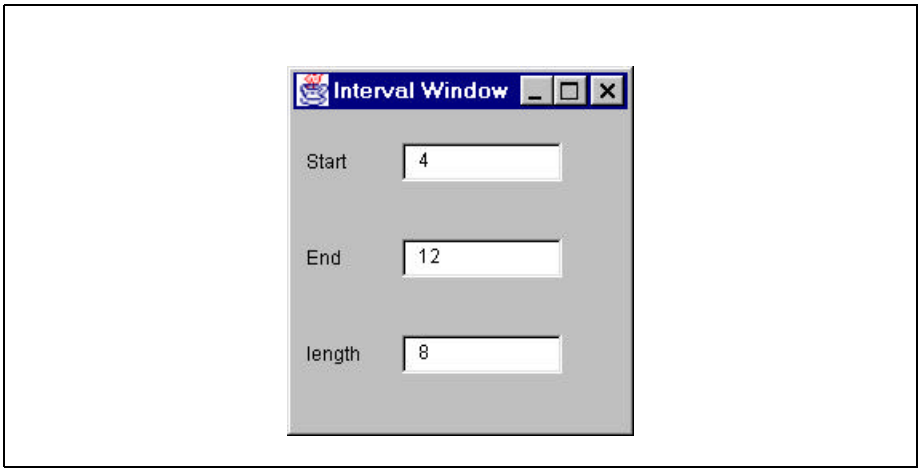


Figure 9.1: A simple gui window

All the methods are on a single `IntervalWindow` class. The fields are set to respond to the loss of focus from the field.

```
public class IntervalWindow extends Frame...
  java.awt.TextField _startField;
  java.awt.TextField _endField;
  java.awt.TextField _lengthField;

  class SymFocus extends java.awt.event.FocusAdapter
  {
      public void focusLost(java.awt.event.FocusEvent event)
      {
          Object object = event.getSource();
          if (object == _startField)
             StartField_FocusLost(event);
          else if (object == _endField)
             EndField_FocusLost(event);
          else if (object == _lengthField)
             LengthField_FocusLost(event);
      }
  }
```

The listener reacts by calling StartField_FocusLost when focus is lost on the start field, and EndField_FocusLost and LengthField_FocusLost for the other fields. These event handling methods look like this:

```
void StartField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_startField.getText()))
        _startField.setText("0");
    calculateLength();
}

void EndField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_endField.getText()))
        _endField.setText("0");
    calculateLength();
}

void LengthField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_lengthField.getText()))
        _lengthField.setText("0");
    calculateEnd();
}
```

(If you are wondering why I did the window this way, I just did it the easiest way my IDE (Cafe) encouraged me to.)

All of them insert a zero if any non integer characters appear and call the relevant calculation routine.

```
void calculateLength(){
    try {
      int start = Integer.parseInt(_startField.getText());
      int end = Integer.parseInt(_endField.getText());
      int length = end - start;
      _lengthField.setText(String.valueOf(length));
    } catch (NumberFormatException e) {
      throw new RuntimeException ("Unexpected Number Format Error");
    }
}
void calculateEnd() {
    try {
      int start = Integer.parseInt(_startField.getText());
      int length = Integer.parseInt(_lengthField.getText());
      int end = start + length;
      _endField.setText(String.valueOf(end));
    } catch (NumberFormatException e) {
      throw new RuntimeException ("Unexpected Number Format Error");
    }
}
```

My task, should I choose to accept it, is to separate out the non-visual logic from the gui. Essentially this means moving calculateLength and calculateEnd to a separate domain class. However to do this they need to refer to the start, end, and length data *without* referring to the window class. The only way I can do this is to duplicate this data in the domain class and synchronize the data with the gui.

I don't currently have a domain class, so I create an (empty) one.

```
class Interval extends Observable {}
```

The interval window needs a link to this new domain class.

```
    private Interval _subject;
```

I then need to properly initialize this field, and make interval window an observer of the interval. I can do this by putting the following code in interval window's constructor.

```
    _subject = new Interval();
    _subject.addObserver(this);
    update(_subject, null);
```

I like to put this code at the end of construction process. The call to update will ensure that as I duplicate the data in the domain class the gui is initialized from the domain class.

Of course to do this I need to declare that interval window implements observable.

```
public class IntervalWindow extends Frame implements Observer
```

In order to implement Observer I need to create an update method. For the moment this can be blank.

```
    public void update(Observable observed, Object arg) {
    }
```

I can compile and test at this point. I haven't made any real changes yet, but I can make mistakes in the simplest places.

Now I can turn my attention to moving fields. As usual I make the changes one field at a time. To demonstrate my command of the English language I'll start with the end field. The first task is to apply *Self Encapsulate Field (185)*. Text fields are updated with getText and setText methods. I create accessors that call these

```
String getEnd() {
    return _endField.getText();
```

```
}

void setEnd (String arg) {
    _endField.setText(arg);
}
```

I find every reference to `_endField` and replace them the appropriate accessors.

```
void calculateLength(){
  try {
    int start = Integer.parseInt(_startField.getText());
    int end = Integer.parseInt(getEnd());
    int length = end - start;
    _lengthField.setText(String.valueOf(length));
  } catch (NumberFormatException e) {
    throw new RuntimeException ("Unexpected Number Format Error");
  }
}

void calculateEnd() {
  try {
    int start = Integer.parseInt(_startField.getText());
    int length = Integer.parseInt(_lengthField.getText());
    int end = start + length;
    setEnd(String.valueOf(end));
  } catch (NumberFormatException e) {
    throw new RuntimeException ("Unexpected Number Format Error");
  }
}

void EndField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(getEnd()))
        setEnd("0");
    calculateLength();
}
```

That's the normal process for *Self Encapsulate Field (185)*. However when you are working with a GUI, there is a complication. The user can change the field value directly without calling `setEnd`. So I need to put a call to `setEnd` into the event handler for the GUI. This call will change value of the end field to the current value of the end field. Of course this does nothing at the moment, but it does ensure the user input goes through the setting method.

```
void EndField_FocusLost(java.awt.event.FocusEvent event) {
    setEnd(_endField.getText());
    if (isNotInteger(getEnd()))
        setEnd("0");
```

```
        calculateLength();
    }
```

You should notice that in this call I don't use getEnd, instead I access the field directly. I do this because later on in the refactoring getEnd will get a value from the domain object, not from the field. At that point using it would mean that every time the user changed the value of the field this code would just change it back again, so here I must use direct access.

At this point I can compile and test the encapsulated behavior.

Now I add the end field to the domain class.

```
class Interval...
    private String _end = "0";
```

I initialize it to the same value that it is initialized to in the gui. I now add getting and setting methods.

```
class Interval...

  String getEnd() {
      return _end;
  }
  void setEnd (String arg) {
      _end = arg;
      setChanged();
      notifyObservers();
  }
```

Since I'm using the observer pattern I have to add the notification code into the setting method. You'll also notice that I use string, not a (more logical) number. This is because I want to make the smallest possible change. Once I've successfully duplicated the data I can look to change the internal data type to an integer.

I can now do one more compile and test before I perform the duplication. By getting all this preparatory work done I've minimized the risk in this tricky step.

The first change is updating the accessors on interval window to use interval.

```
class IntervalWindow...
  String getEnd() {
      return _subject.getEnd();
  }
  void setEnd (String arg) {
```

```
        _subject.setEnd(arg);
    }
```

I also need to update `update` to ensure the gui reacts to the notification.

```
class IntervalWindow...
    public void update(Observable observed, Object arg) {
        _endField.setText(_subject.getEnd());
    }
```

This is the other place where I have to use direct access. If I called the setting method I would get into an infinite recurse.

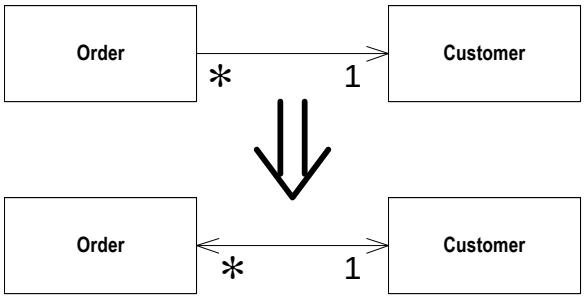
I can now compile and test and the data is properly duplicated.

I can repeat for the other two fields. Once this is done I can apply *Move Method (160)* to move `calculateEnd` and `calculateLength` over to the interval class.

Change Unidirectional Association to Bidirectional

You have classes that need to have a two way reference

Add back-pointers, change modifiers to update both sets



Motivation

You may find that you have initially set up two classes so that one class refers to the other. Over time you may find that a client of the referred class needs to get to the objects that refer to it. Effectively this means navigating backwards along the pointer. Pointers are one-way links, so you can't do this. Often you can get around this by find another route. This may cost in computation, but is still reasonable, and you can have a method on the referred class that uses this behavior.

However sometimes this is not easy, and you need to set up a two-way reference (sometimes referred to as back pointers). If you aren't used to these, it's easy to get tangled up doing this, but once you get used to the idiom it is not too complicated.

The idiom is awkward enough that you should have tests, at least until you are comfortable with the idiom. Since I usually don't bother test-

ing accessors (the risk is not high enough) this is the rare case of a refactoring that adds a test.

This refactoring uses back-pointers to implement the bidirectionality. There are other techniques, such as link objects, that would require other refactorings.

Mechanics

- Add a field for the back--pointer
- Decide which class will control the association
- Create a helper method on the non-controlling side of the association. Name this to clearly indicate its restricted use.
- If the existing modifier is on the controlling side, modify it to update the back pointers
- If the existing modifier is on the controlled side, create a controlling method on the controlling side and call it from the existing modifier.

Example

I'll start with a simple program that has an order which refers to a customer

```
class Order...
  Customer getCustomer() {
      return _customer;
  }
  void setCustomer (Customer arg) {
      _customer = arg;
  }
  Customer _customer;
```

The customer class has no reference back to the order.

I start the refactoring by adding a field to the customer. As a customer can have several orders this field will be a collection. Since we don't want a customer to have the same order more than once in it's collection, the correct collection would be a set. However since I'm using vanilla Java 1.1 I'll have to make do with a vector.

```
class Customer  {
  private Vector _orders = new Vector();
```

Now I need to decide which class will take charge of the association. My decision process runs like this

- If both objects are reference objects, and it is a one to many association, then the object which has the one reference is the controller. (That is if one customer has many orders, then the order controls the association)
- If one object is a component of the other, then the composite should control the association.
- If both objects are reference objects, and it is a many to many association, then it doesn't matter.

As the order will take charge I need to add a helper method to the customer that the order's modifier will use. I use name friendOrders to signal that this method is only to be used in special cases. I also minimize its visibility by making it package visibility if at all possible. I do have to make it public if the other class is in another package.

```
class Customer ...
  Vector friendOrders() {
  /** should only be used by Order */
      return _orders;
  }
```

Now I update the modifier to update the back pointers.

```
  void setCustomer (Customer arg) ...
      if (_customer != null) _customer.friendOrders().removeElement(this);
      _customer = arg;
      if (_customer != null) _customer.friendOrders().addElement(this);
  }
```

The exact code in the controlling modifier will be different depending on the multiplicity of the association. If the customer is not allowed to be null, I can forego the null checks (but I need to check for a null argument). But the basic pattern is always the same, first tell the target object to disconnect, second reconnect at your end, then tell the new target to reconnect.

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

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

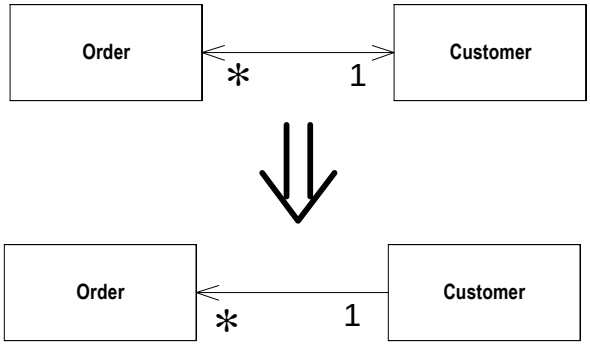
If an order could have many customers, then we would have a many to many case, and the methods would look like this.

```
class Order... //controlling methods
  void addCustomer (Customer arg) {
      arg.friendOrders().addElement(this);
      _customers.addElement(arg);
  }
  void removeCustomer (Customer arg) {
      arg.friendOrders().removeElement(this);
      _customers.removeElement(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 that no longer needs to be two way

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, and so they often are a source of errors.

Lots of two way links also make it too easy for mistake to lead to zombies: objects that should be dead but still hang around due to some reference that never got cleared.

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

As such 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 needs to be eliminated to see if it is feasible to eliminate the field
  - Look at direct readers and further methods that call the methods
  - Consider the possibility of using *Substitute Algorithm (143)* on the getter for the field
  - Consider adding the object as an argument to all methods that use the field
- If the change seems feasible then decide whether you need to substitute the getting method.
- If you are substituting the getting method, use *Self Encapsulate Field (185)*, carry out *Substitute Algorithm (143)* on the getter, compile and test.
- If you aren't substituting the getter then change each user of the field so that it gets the object in the field another way. Compile and test after each change.
- When there is no reader left of the field, remove all updates to the field, and remove the field.
  - If there are many places that assign the field then use *Self Encapsulate Field (185)* so that they all use a single setter. Compile and test. Then change the setter to have an empty body. Compile and test. If that works then remove the field, the setter, and all calls to the setter.
- Compile and test.

Example

I'll start from where I ended up from the example in *Change Unidirectional Association to Bidirectional (209)*. I have a customer and order with a bi-directional link.

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

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

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

The hardest part of this refactoring is checking that I can do it. Once I know it's safe to do, then it's easy. The issue is whether code is relying on the customer field being there. Then in order to remove it, I need to provide an alternative.

So 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, since it's then easy to pass itself in as an argument. So

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

becomes

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

Another alternative I might have is to give order an operation to get a customer, that does not use the field. Here my aim is to use *Substitute Algorithm (143)* on the body of `Order.getCustomer`. I might do something like this

```
Customer getCustomer() {
    Enumeration e = Customer.getInstances();
    while (e.hasMoreElements()) {
        Customer each = (Customer)e.nextElement();
        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 I can change them to use `getCustomer` by using *Self Encapsulate Field (185)*.

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

When I'm doing this I first study all the uses to see if it seems feasible. If I substitute the getting method, then 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 usually is pretty rapid, because 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. Since nobody is reading it any more, that shouldn't matter.

Replace Magic Number with Symbolic Constant

You have a quoted 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 * height * 9.81;
}
```

```
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 are usually not obvious. They are really nasty when you need to reference the same logical number in more than one place. If the numbers might ever change, then making the change is a nightmare. Even if you don't make a change, you have the difficulty of figuring out what to do.

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

However 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 then consider *Replace Type Code with Class (229)*. If it is the length of an array then 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 if it matches the usage of the constant, if so change it to use the constant
- Compile
- When all are changed compile and test, at this point all should work as nothing has been changed.
  - ☞ A good test for this to see if you can change the constant easily. This may mean altering some expected results to match the new value. This isn't always possible, but it is a good trick when it works.

Encapsulate Field

There is a public field

Make it private and provide accessors

```
public String _name
```

```
private String _name;
public String getName() {return _name;}
public void setName(String arg) {_name = arg;}
```

Motivation

One of the principal tenets of object-orientation is encapsulation, or data hiding. This says that you should never make your data public. When you make data public it allows other objects to change and access data values without the owning object knowing about it. This separates data from behavior.

This is seen as a bad thing because it reduces the modularity of the program. When the data and behavior that uses it are clustered together, then it is easier to change, because the changed code is in one place rather than scattered all over the program.

This refactoring begins the process by hiding the data and adding accessors. But this is only the first step. A class with only accessors is a dumb class that doesn't really take advantage of the opportunities of objects, and an object is terrible thing to waste. Once I've done *Encapsulate Field (219)* I look for those methods that use the new methods to see if they fancy packing their bags and moving to the new object with a quick *Move Method (160)*.

Mechanics

- Create getting and setting methods for the field
- Find all places outside the class that reference the field. If it uses the value then replace the reference with a call to the getting method. If it changes the value then replace with a call to the setting method.
  - ☞ If the field is an object and the client invokes a modifier on the object, then that is a use. Only use the setting method to replace assignment.
- Compile and test after each change
- Once all references are changed, declare the field as private.
- Compile and test

Encapsulate Collection

A method returns a collection

Make it return a read only view and provide add/ remove methods

Motivation

Often a class will contain a collection of instances. This collection might be an array, vector, dictionary, or one of the more sophisticated collections available in Java 1.2. In such cases there is often the usual getter and setter for that collection.

However collections should use a slightly different protocol to other kinds of data. The getter should not return the collection object itself, for that allows clients to manipulate the contents of the collection without the owning class knowing what is going on. Also it reveals too much to clients about the object's internal data structures. So a getter for a multi-valued attribute should return an iterator, which in Java means an Enumeration.

In addition there should not be a setter for collection, rather there should be operations to add and remove elements. This gives the owning object control over adding and removing elements from the collection.

With this protocol the collection is properly encapsulated, which reduces the coupling of the owning class to its clients.

Mechanics

- Add an add and remove method for the collection
- Initialize the field to an empty collection
- Compile
- Find callers of the setting method. Either modify the setting method to use the add/remove operations or have the clients call those operations instead.
  - ☞ Two cases are when the setter only is used when the collection is empty, and when the setter is used to replace a full collection.

☞ You may wish to rename the setting method to better communicate the intention using *Rename Method (235)*

- ❍ Compile and test
- ❍ Find all users of the getter that are modifying the collection. Change them to use the modifier. Compile and test after each change
- ❍ When all uses of the getter to modify have been changed, modify the getter to return a copy of the collection
- ❍ Compile and test
- ❍ Change the name of the current getter and add a getter to return an enumeration. Find users of the getter and change them to use one of the new methods.

☞ If this is too big a jump use *Rename Method (235)* on the getter, create a new method that returns an enumeration, and change callers to use the new method.

- ❍ Compile and test
- ❍ Find the users of the getter. Look for code that should be on the host object. Use *Extract Method (114)* and *Move Method (160)* to get it there.
- ❍

Example

As an example we will have a person taking courses. Our course is pretty simple

```
class Course ...
  public Course (String name, boolean isAdvanced) {...};
  public boolean isAdvanced() {...};
```

We will not bother with anything further on the course. The interesting class is the person.

```
class Person...
  public Vector getCourses() {
      return _courses;
  }
  public void setCourses(Vector arg) {
      _courses = arg;
  }
  private Vector _courses;
```

With this interface, clients adds courses with code like

```
Person kent = new Person();
Vector v = new Vector();
v.addElement(new Course ("Smalltalk Programming", false));
v.addElement(new Course ("Appreciating Single Malts", true));
kent.setCourses(v);
Assert.equals ("courses", 2, kent.getCourses().size());
```

```
Course refact = new Course ("Refactoring", true);
kent.getCourses().addElement(refact);
kent.getCourses().addElement(new Course ("Brutal Sarcasm", false));
Assert.equals ("2nd courses", 4, kent.getCourses().size());
kent.getCourses().removeElement(refact);
Assert.equals ("3rd courses", 3, kent.getCourses().size());
```

A client that wants to know about advanced courses might do it this way

```
Enumeration e = person.getCourses().elements();
int count = 0;
while (e.hasMoreElements()) {
    Course each = (Course) e.nextElement();
    if (each.isAdvanced()) count ++;
}
```

The first thing I want to do is to create the proper modifiers for the collection and compile

```
class Person
  public void addCourse(Course arg) {
      _courses.addElement(arg);
  }
  public void removeCourse(Course arg) {
      _courses.removeElement(arg);
  }
```

Life will be easier if I initialize the field as well

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

I then look at the users of the setter. If there are many clients, and the vector is used heavily then I will need to replace the body of the setter to use the add / remove operations. The complexity of this depends on how the setter is used. There are two cases. In the simplest case the client uses the setter to initialize the values, i.e. there are no courses before the setter is applied. In this case I replace the body of the setter to use the add method.

```
class Person...
  public void setCourses(Vector arg) {
      Assert.isTrue(_courses.isEmpty());
      Enumeration e = arg.elements();
      while (e.hasMoreElements()) {
          addCourse((Course) e.nextElement());
      }
  }
```

After changing the body like this it is wise to use *Rename Method (235)* to make the intention clearer

```
public void initializeCourses(Vector arg) {
    Assert.isTrue(_courses.isEmpty());
    Enumeration e = arg.elements();
    while (e.hasMoreElements()) {
        addCourse((Course) e.nextElement());
    }
}
```

In the more general case I have to use the remove method to remove every element first and then add the elements. But I find that occurs rarely (general cases often are).

However if the clients simply set up a vector and use it, I can get them to use the add and remove methods directly, and remove the setter completely. So code like

```
Person kent = new Person();
Vector v = new Vector();
v.addElement(new Course ("Smalltalk Programming", false));
v.addElement(new Course ("Appreciating Single Malts", true));
kent.setCourses(v);
```

becomes

```
Person kent = new Person();
kent.addCourse (new Course ("Smalltalk Programming", false));
kent.addCourse (new Course ("Appreciating Single Malts", true));
```

Now I start looking at users of the getter. My first concern are those cases where somebody uses the getter to modify the underlying collection, cases like

```
kent.getCourses().addElement(new Course ("Brutal Sarcasm", false));
```

I need to replace this with a call to the new modifier.

```
kent.addCourse(new Course ("Brutal Sarcasm", false));
```

Once I've done this for everyone I can check that nobody is modifying through the getter by changing the getter body to return a copy.

```
class Person...
  Vector getCourses() {
      return (Vector) _courses.clone();
  }
```

At this point I've encapsulated the collection. Nobody can change the elements of collection except through the person. However returning a

collection is still poor style, and most of the time not that helpful to its clients. Often an enumeration is more helpful since they will usually be enumerating through the collection. So in this case change the name of the existing getter and create a new getter that returns an enumeration

```
public Vector getCoursesVector() {
    return (Vector) _courses.clone();
}
public Enumeration getCourses() {
    return _courses.elements();
}
```

Now I go back to all the users of the getter. I either change them to use the enumeration, or let them call the new method. In many cases callers will get the vector in order to get the enumeration

```
Enumeration e = person.getCourses().elements();
```

The new interface makes that simpler.

```
Enumeration e = person.getCourses();
```

When I've changed all these I can compile and test. If that is too big a change I can use *Rename Method (235)* to change `getCourses` to `getCoursesVector`, and then alter callers one by one to use the enumeration directly. Most of the time I find it easier to do the two together as I am working through the caller anyway.

Now I have the right interface. Once I've done that, however, I like to look at the users of the getter, looking for code that ought to be on Person. Code like

```
Enumeration e = person.getCourses();
int count = 0;
while (e.hasMoreElements()) {
    Course each = (Course) e.nextElement();
    if (each.isAdvanced()) count ++;
}
```

Is probably better moved to person. First I use *Extract Method (114)* on the code

```
int numberOfAdvancedCourses(Person person) {
    Enumeration e = person.getCourses();
    int count = 0;
    while (e.hasMoreElements()) {
        Course each = (Course) e.nextElement();
```

```
            if (each.isAdvanced()) count ++;
        }
        return count;
    }
```

And then I use *Move Method (160)* to move it to person.

```
class Person...
    public int numberOfAdvancedCourses() {
        Enumeration e = getCourses();
        int count = 0;
        while (e.hasMoreElements()) {
            Course each = (Course) e.nextElement();
            if (each.isAdvanced()) count ++;
        }
        return count;
    }
```

A common case is

```
kent.getCoursesVector().size()
```

which can be changed to

```
kent.numberOfCourses()

class Person...
    public int numberOfCourses() {
        return _courses.size();
    }
```

A few years ago I was concerned that moving this kind of behavior over to person would lead to a bloated person class. In practice, I've found that usually isn't a problem.

If your collection is an array, it is a little harder. You can provide an enumeration with an inner class

```
    private String[] _skills = new String[10];

    public Enumeration getSkills() {
        return new Enumeration () {
            private int _position = 0;
            public boolean hasMoreElements() {
               return (_position < _skills.length);
            }
            public Object nextElement() throws NoSuchElementException {
               if (hasMoreElements())
                  return  _skills[_position++];
               else throw new NoSuchElementException();
            }
        };
```

```
}
```

Similarly you can provide methods to update the array. Usually, however, I find it easier to just change the field to be a vector instead of an array. That way I get all of the behavior of vector and I don't have to worry about sizing the array. You can use the `copyInto` method of vector to provide an array for clients who need the array.

```
private Vector _skills = new Vector();
public String[] getSkillsArray() {
    Object[] result = new Object[_skills.size()];
    _skills.copyInto(result);
    return (String[]) result;
}
```

Replace Record with Data Class

You have a record structure in a traditional programming environment. How do you begin to make this object oriented?

Make a dumb data object for the record.

Motivation

Record structures are a common feature of programming environments. There are various reasons you may want to bring them into an object-oriented program. You could be copying a legacy program as in *Replace Program With Class (305)*, you could be communicating a structured record with a traditional programming API, or a data base record. In these cases it is useful to create an interfacing class to deal with this external element. It is simplest to make the class look just like the external record. You then move other fields and methods into the class later.

A less obvious, but very compelling case of this is an array where the element in each index has some special meaning. In this case you use *Replace Array with Object (197)*.

Mechanics

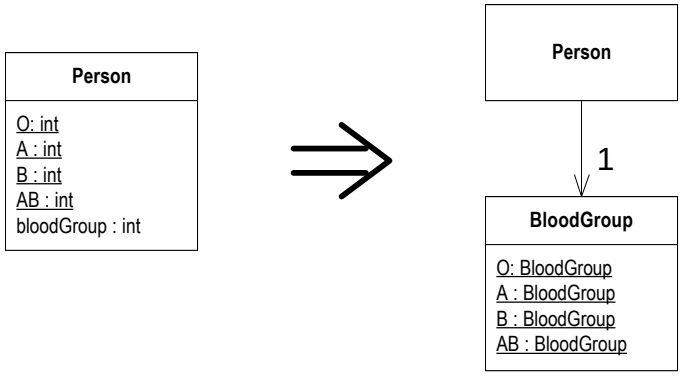
- Create a class to represent the record
- Give the class a private field with a getting method and a setting method for each data item.

You now have a dumb data object, it has no behavior yet, but further refactoring will explore that issue.

Replace Type Code with Class

You have a numeric type code that does not affect behavior

Replace the number with a new class



Motivation

Numeric type codes, or enumerations, are a common feature of C-based languages. With symbolic names they can be quite readable. The problem is that the symbolic name is only an alias, the compiler still sees the underlying number. Thus the compiler type checks using the number not the symbolic name. Any method that takes the type code as an argument will expect a number, and there is nothing to force a symbolic name to be used. This can reduce readability and be a source of bugs.

If you replace the number with a class, the compiler can now type check on the class. By providing factory methods for the class you can statically check that only valid instances are created, and that those instances are passed on to the correct objects.

Before you do this, however, you need to consider the other type code replacements. Only replace the type code with a class if the type code

is pure data, that is it does not cause different behavior inside a switch statement somewhere. For a start Java can only switch on an integer, not an arbitrary class, so the replacement will fail. But more importantly than that, any switch needs to be removed by *Replace Conditional with Polymorphism (154)*. In order to do that the type code has to be dealt with by *Replace Type Code with Subclasses (235)* or *Replace Type Code with State/Strategy (239)* first.

Even if a type code does not cause different behavior depending on its value, there might be behavior that is better placed in the type code class, so be alert to the value of a *Move Method (160)* or two.

Mechanics

- Create a new class for the type code
  - The class needs a code field which matches the type code, and a getting method for this value. It should have static variables for the allowable instances of the class, and a static method which returns the appropriate instance from an argument based on the original code.
- Modify the implementation of the source class to use the new class
  - Maintain the old code-based interface, but change the static fields to use new class to generate the codes, and alter the other code based methods to get the code numbers from the new class
- Compile and Test
  - At this point the new class can do run time checking of the codes.
- For each method on the source class that uses the code, create a new method that uses the new class instead.
  - Methods that use the code as an argument need new methods using an instance of the new class as an argument. Methods that return a code need a new method returning the code. It is often wise to use *Rename Method (235)* on an old accessor before creating a new one, to make the program clearer when it is using an old code
- One by one, change the clients of the source class to use the new interface
- Compile and test after each client is updated
  - You may need to alter several methods before you have enough consistency to compile and test.
- Remove the old interface that uses the codes, and the static declarations of the codes
- Compile and test.

Example

I'll start with a person who has a blood group modeled with a type code

```
class Person {

  public static final int O = 0;
  public static final int A = 1;
  public static final int B = 2;
  public static final int AB = 3;

  private int _bloodGroup;

  public Person (int bloodGroup) {
      _bloodGroup = bloodGroup;
  }

  public void setBloodGroup(int arg) {
      _bloodGroup = arg;
  }

  public int getBloodGroup() {
      return _bloodGroup;
  }
}
```

I start by creating a new blood group class with instances that contain the type code number.

```
class BloodGroup {
  public static final BloodGroup O = new BloodGroup(0);
  public static final BloodGroup A = new BloodGroup(1);
  public static final BloodGroup B = new BloodGroup(2);
  public static final BloodGroup AB = new BloodGroup(3);
  private static final BloodGroup[] _values = {O, A, B, AB};

  private final int _code;

  private BloodGroup (int code ) {
      _code = code;
  }

  public int getCode() {
      return _code;
  }

  public static BloodGroup code(int arg) {
      return _values[arg];
  }
```

```
}
```

I then replace the code in Person with code that uses the new class.

```
class Person {

  public static final int O = BloodGroup.O.getCode();
  public static final int A = BloodGroup.A.getCode();
  public static final int B = BloodGroup.B.getCode();
  public static final int AB = BloodGroup.AB.getCode();

  private BloodGroup _bloodGroup;

  public Person (int bloodGroup) {
      _bloodGroup = BloodGroup.code(bloodGroup);
  }

  public int getBloodGroup() {
      return _bloodGroup.getCode();
  }

  public void setBloodGroup(int arg) {
      _bloodGroup = BloodGroup.code (arg);
  }
}
```

At this point I now have run-time checking within the blood group class. But to really gain from the change I have to alter the users of the person class to use blood group instead of integers.

To begin this I use *Rename Method (235)* on the accessor for the person's blood group to clarify the new state of affairs and add a new getting method that uses the new class

```
class Person...
  public int getBloodGroupCode() {
      return _bloodGroup.getCode();
  }
  public int getBloodGroup() {
      return _bloodGroup;
  }
```

I also create a new constructor and setting method that uses the class.

```
public Person (BloodGroup bloodGroup ) {
  _bloodGroup = bloodGroup;
}
```

```
public void setBloodGroup(BloodGroup arg) {
  _bloodGroup = arg;
}
```

Now I go to work on the clients of Person. The art is to do one client at a time so that you can take smaller steps. Each client may need various changes, however, that makes it more tricky. Any reference to the static variables needs to be changed. So

```
Person thePerson = new Person(Person.A)
```

becomes

```
Person thePerson = new Person(BloodGroup.A);
```

References to the getting method need to use the new one, so

```
thePerson.getBloodGroupCode()
```

becomes

```
thePerson.getBloodGroup()
```

The same is true for setting methods, so

```
thePerson.setBloodGroup(Person.AB)
```

becomes

```
thePerson.setBloodGroup(BloodGroup.AB)
```

Once this is done for all clients of person, I can remove the getting method, constructor, static definitions, and setting methods that use the integer.

```
class Person ...
  public static final int O = BloodGroup.O.getCode();
  public static final int A = BloodGroup.A.getCode();
  public static final int B = BloodGroup.B.getCode();
  public static final int AB = BloodGroup.AB.getCode();
  public Person (int bloodGroup) {
      _bloodGroup = BloodGroup.code(bloodGroup);
  }
  public int getBloodGroup() {
      return _bloodGroup.getCode();
  }
  public void setBloodGroup(int arg) {
      _bloodGroup = BloodGroup.code (arg);
  }
```

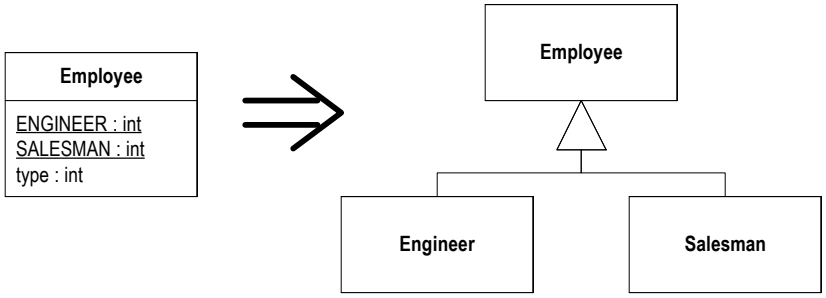
I can also privatize the methods on blood group that use the code.

```
class BloodGroup...
  private int getCode() {
      return _code;
  }

  private static BloodGroup code(int arg) {
      return _values[arg];
  }
```

Replace Type Code with Subclasses

You have a type code which affects the behavior of a class

Replace the type code with subclasses



Motivation

If you have a type code that does not affect behavior, then you can use *Replace Type Code with Class (229)*. However if the type code affects behavior, then the best thing to do is to use polymorphism to handle the variant behavior.

This situation is usually indicated by the presence of case-like conditional statements. These may be switches or if-then-else constructs. In either case they test the value of the type code and then execute different code depending on the type code's value. Such conditionals need to be refactored with *Replace Conditional with Polymorphism (154)*. However to do this the type code needs to be replaced with an inheritance structure which will host the polymorphic behavior. Such an inheritance structure will have a class with subclasses for each type code

The simplest way to do this is this refactoring. Here you take the class that has the type code and create a subclass for each type code. However there are cases when you can't do this.

- The value of the type code changes after the object is created
- The class with the type code is already subclassed for another reason.

In either of these cases you need to use *Replace Type Code with State/ Strategy (239).*

This refactoring is primarily a scaffolding move that enables *Replace Conditional with Polymorphism (154).* So the trigger to do this is the presence of such conditional statements. If there are no such conditional statements then *Replace Type Code with Class (229)* is the better (and less critical move).

Another reason to do this refactoring is if there are features that are only relevant to objects with certain type codes. Once you've done this refactoring you can use *Push Down Method (279)* and *Push Down Field (281)* to clarify that these features are only relevant in certain cases.

The advantage of this refactoring is that it moves the knowledge of the variant behavior from clients of the class to the class itself. If we add new variants, all we need to do is add a subclass. Without polymorphism we have to find all the conditionals and change those. So this is particularly valuable when these variants keep changing.

Mechanics

- ❍ *Self encapsulate* the type code.
  - ☞ If the type code is passed into the constructor, you will need to *replace the constructor with a factory method.*
- ❍ For each value of the type code create a subclass. Override the getting method of the type code in the subclass to return the relevant value.
  - ☞ This value will be hard coded into the return (e.g. "return 1). This looks messy but it is a temporary measure until all case statements have been replaced.
- ❍ Compile and test after replacing each type code value with a subclass.
- ❍ Remove the type code field from the superclass. Declare the accessors for the type code as abstract.
- ❍ Compile and test

Example

I will use the boring and unrealistic employee payment example.

```
class Employee...
  private int _type;
  static final int ENGINEER = 0;
  static final int SALESMAN = 1;
  static final int MANAGER = 2;

  Employee (int type) {
      _type = type;
  }
```

The first step is to use *Self Encapsulate Field (185)* on the type code.

```
  int getType() {
      return _type;
  }
```

Since the employee's constructor uses a type code as a parameter, I need to replace it with a factory method.

```
  Employee create(int type) {
      return new Employee(type);
  }

  private Employee (int type) {
      _type = type;
  }
```

I can now start with the engineer as a subclass. First I create the subclass and the overriding method for the type code.

```
class Engineer extends Employee {

  int getType() {
      return Employee.ENGINEER;
  }

}
```

I also need to alter the factory method to create the appropriate object.

```
class Employee
  static Employee create(int type) {
      if (type == ENGINEER) return new Engineer();
      else return new Employee(type);
  }
```

I continue, one by one, until all the codes are replaced by subclasses. At this point I can get rid of the type code field on employee and make

getType an abstract method. At this point the factory method looks like this.

```
abstract int getType();

static Employee create(int type) {
    switch (type) {
        case ENGINEER:
           return new Engineer();
        case SALESMAN:
           return new Salesman();
        case MANAGER:
           return new Manager();
        default:
           throw new IllegalArgumentException("Incorrect type code value");
    }
}
```

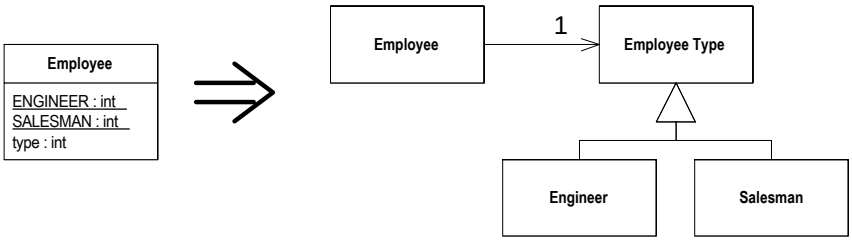
Of course this is the kind of switch statement I would prefer to avoid. But there is only one of them, and it is only used at creation. There are ways to deal with this too, see the discussion in *Replace Constructor with Factory Method (263)*.

Naturally once you have created the subclasses you should use *Push Down Method (279)* and *Push Down Field (281)* on any methods and fields that are only relevant for particular types of employee.

Replace Type Code with State/Strategy

You have a type code which affects the behavior of a class. The class is already subclassed or the type code is mutable.

Replace it with a state object.



Motivation

This is similar to *Replace Type Code with Subclasses (235)*, but can be used if the type code changes during the life of the object, or if some other reason prevents subclassing. It uses either the state or strategy pattern [Gang of Four].

State and strategy are very similar, so the refactoring is the same whichever you use, and it doesn't really matter. Choose the pattern based on what fits the specific circumstances better. If you are moving for a single algorithm, then strategy is the better term. If you are going to move state specific data and you think of the object as changing state use the state pattern.

Mechanics

- *Self encapsulate* the type code.
- Create a new class, name it after the purpose of the type code. This is the state object.
- Add subclasses of the state object, one for each type code.
  - ☞ It is easier to do them all at once, rather than one at a time.

- Create an abstract query in the state object to return the type code. Create overriding queries of each state object subclass to return the correct type code.
- Compile
- Create a field in the old class for the new state object.
- Adjust the type code query on the original class to delegate to the state object.
- Adjust the type code setting methods on the original class to assign an instance of the appropriate state object subclass.
- Compile and test.

Example

I'll use the tiresome and brainless employee example.

```
class Employee {

  private int _type;
  static final int ENGINEER = 0;
  static final int SALESMAN = 1;
  static final int MANAGER = 2;

  Employee (int type) {
      _type = type;
  }

  int payAmount() {
      switch (_type) {
          case ENGINEER:
             return _monthlySalary;
          case SALESMAN:
             return _monthlySalary + _commission;
          case MANAGER:
             return _monthlySalary + _bonus;
          default:
             throw new RuntimeException("Incorrect Employee");
      }
  }
```

I assume this is an exciting and go-ahead company that allows promotion of managers to engineers. Thus the type code is mutable and I can't use subclassing. My first step, as ever, is to self-encapsulate the type code.

```
  Employee (int type) {
      setType (type);
  }
```

```
int getType() {
    return _type;
}

void setType(int arg) {
    _type = arg;
}

int payAmount() {
    switch (getType()) {
        case ENGINEER:
           return _monthlySalary;
        case SALESMAN:
           return _monthlySalary + _commission;
        case MANAGER:
           return _monthlySalary + _bonus;
        default:
           throw new RuntimeException("Incorrect Employee");
    }
}
```

Now I declare the state class. I declare this as an abstract class and provide an abstract method for returning the type code.

```
abstract class EmployeeType {
  abstract int getTypeCode();
}
```

I'll now create the subclasses.

```
class Engineer extends EmployeeType {

  int getTypeCode () {
      return Employee.ENGINEER;
  }
}

class Manager extends EmployeeType {

  int getTypeCode () {
      return Employee.MANAGER;
  }
}

class Salesman extends EmployeeType {

  int getTypeCode () {
      return Employee.SALESMAN;
  }
}
```

I compile so far, and it is all so trivial that, even for me, it compiles easily. Now I actually hook the subclasses into the employee by modifying the accessors for the type code.

```
class Employee…
  private EmployeeType _type;

  int getType() {
      return _type.getTypeCode();
  }

  void setType(int arg) {
      switch (arg) {
          case ENGINEER:
             _type = new Engineer();
             break;
          case SALESMAN:
             _type = new Salesman();
             break;
          case MANAGER:
             _type = new Manager();
             break;
          default:
             throw new IllegalArgumentException("Incorrect Employee Code");
      }
  }
```

Of course this means I now have a switch statement here. But once I'm done refactoring it will be the only one anywhere in the code, and it will only be executed when the type is changed. All the other case statements can now be eliminated by *Replace Conditional with Polymorphism (154).*

Still I like to finish the job by moving all knowledge of the type codes and subclasses over to the new class. First I copy the type code definitions into the employee type, create a factory method for employee types, and adjust the setting method on employee

```
class Employee…
  void setType(int arg) {
      _type = EmployeeType.newType(arg);
  }

class EmployeeType…
  static EmployeeType newType(int code) {
      switch (code) {
          case ENGINEER:
             return new Engineer();
          case SALESMAN:
```

```
            return new Salesman();
        case MANAGER:
            return new Manager();
        default:
            throw new IllegalArgumentException("Incorrect Employee Code");
    }
}
static final int ENGINEER = 0;
static final int SALESMAN = 1;
static final int MANAGER = 2;
```

Then I remove the type code definitions from the employee and replace them with references to the employee type.

```
class Employee…
  int payAmount() {
      switch (getType()) {
          case EmployeeType.ENGINEER:
              return _monthlySalary;
          case EmployeeType.SALESMAN:
              return _monthlySalary + _commission;
          case EmployeeType.MANAGER:
              return _monthlySalary + _bonus;
          default:
              throw new RuntimeException("Incorrect Employee");
      }
  }
```

I'm now ready to use *Replace Conditional with Polymorphism (154)* on `payAmount`.