```
    }
```

When you are using self-encapsulation you have to be 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 you may have different behavior in the setter than you have when initializing. In cases like this I prefer using either direct access from the constructor or 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, as follows:

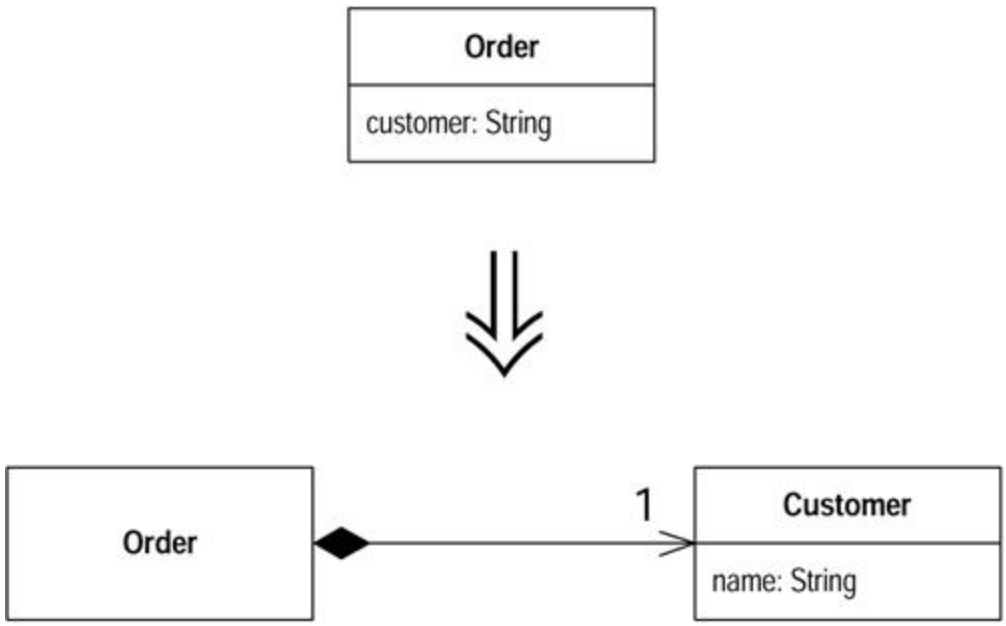
```
  class CappedRange extends IntRange {

    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 the behavior of `IntRange` 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 anymore. A telephone number may be represented as a string for a while, but later you realize that the telephone needs special behavior for formatting, extracting the area code, and the like. For one or two items you may 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 a 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 constructor of the new class.
- 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 on the new object.

Example

I start with an order class that has stored the customer of the order as a string and wants to turn the customer into an object. This way I have somewhere to store data, such as an address or credit rating, and 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

```
    private static int numberOfOrdersFor(Collection orders, String
customer) {
        int result = 0;
        Iterator iter = orders.iterator();
        while (iter.hasNext()) {
            Order each = (Order) iter.next();
            if (each.getCustomerName().equals(customer)) result++;
        }
        return result;
    }
```

First I create the new customer class. I give it a final field for a string attribute, because that is what the order currently uses. I call it *name,* because 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 object, and thus the customer currently also is a value object. This means that each order has its own customer object. As a rule value objects should be immutable; this avoids some nasty aliasing bugs. Later on I will want customer to be a reference object, but that's another refactoring. At this point I can compile and test.

Now I look at the methods on order that manipulate customer and make some changes to make the new state of affairs clearer. With the getter I use Rename Method to make it clear that it is the name not the object that is returned:

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

On the constructor and setter, I 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 me 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 I want to add such things as credit ratings and addresses to our customer, I cannot do so now. This is because the customer is treated as a value object. Each order has its own customer object. To give a customer these attributes I need to apply Change Value to Reference to the customer so that all orders for the same customer share the same customer object. 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.

Make a dumb data object for the record.

Motivation

Record structures are a common feature of programming environments. There are various reasons for bringing them into an object-oriented program. You could be copying a legacy program, or you could be communicating a structured record with a traditional programming API, or a database 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 like the external record. You move other fields and methods into the class later. A less obvious but very compelling case is an array in which the element in each index has a special meaning. In this case you use Replace Array with Object.

Mechanics

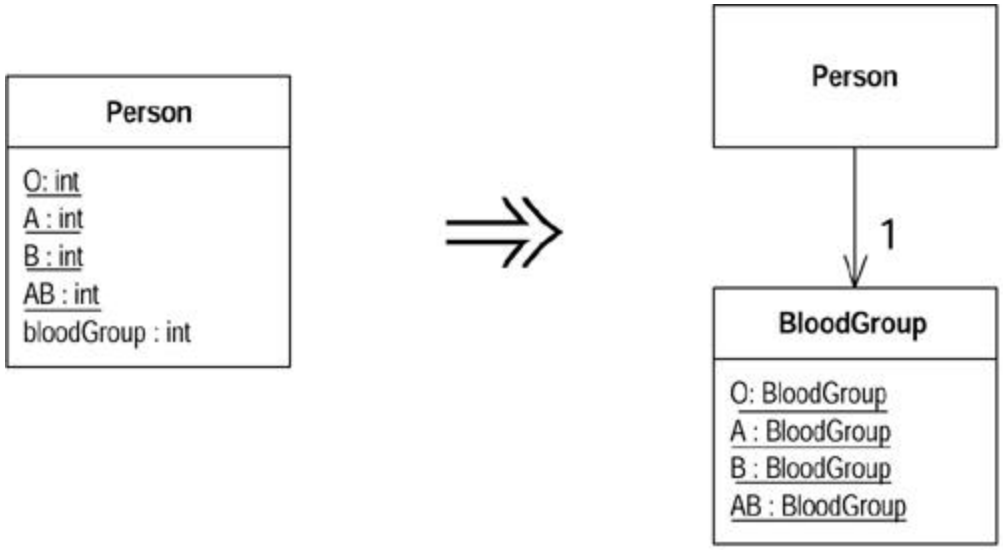
- Create a class to represent the record.
- Give the class a private field with a getting method and a setting method for each data item.

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

Replace Type Code with Class

A class has a numeric type code that does not affect its 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. The compiler type checks using the number

not the symbolic name. Any method that takes the type code as an argument expects 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 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 *Replace Type Code with Class,* however, you need to consider the other type code replacements. Replace the type code with a class only if the type code is pure data, that is, it does not cause different behavior inside a switch statement. For a start Java can only switch on an integer, not an arbitrary class, so the replacement will fail. More important than that, any switch has to be removed with Replace Conditional with Polymorphism. In order for that refactoring, the type code first has to be handled with Replace Type Code with Subclasses or Replace Type Code with State/Strategy.

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 or two.

Mechanics

- Create a new class for the type code.

  ?rarr; *The class needs a code field that 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 that returns the appropriate instance from an argument based on the original code.*

- Modify the implementation of the source class to use the new class.

  ?rarr; *Maintain the old code-based interface, but change the static fields to use new class to generate the codes. Alter the other code-based methods to get the code numbers from the new class.*

- Compile and test.

  ?rarr; *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.

  ?rarr; *Methods that use the code as an argument need new methods that use an instance of the new class as an argument. Methods that return a code need a new method that returns the code. It is often wise to use* Rename Method *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 so that they use the new interface.
- Compile and test after each client is updated.

?rarr; *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 remove the static declarations of the codes.
- Compile and test.

Example

A person 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 have run-time checking within the blood group class. 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 I use Rename Method on the accessor for the person's blood group to clarify the new state of affairs:

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

I then add a new getting method that uses the new class:

```
  public BloodGroup 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 work on one client at a time so that you can take small steps. Each client may need various changes, and 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().getCode()
```

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);
    }
```

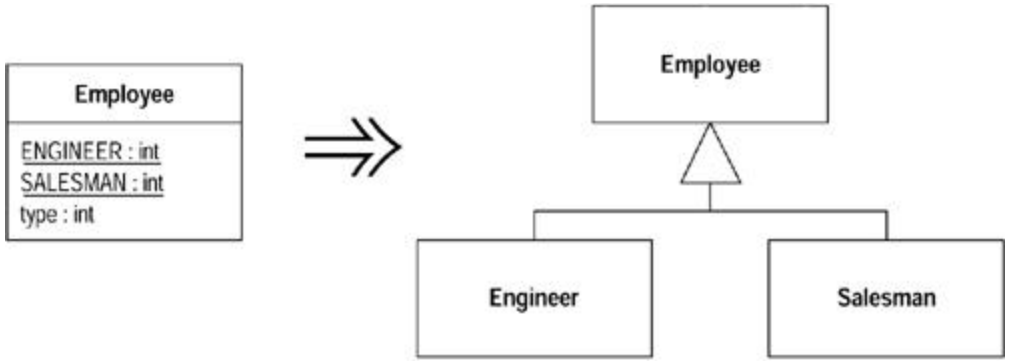
I can also privatize the methods on blood group that use the code:

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

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

Replace Type Code with Subclasses

You have an immutable type code that affects the behavior of a class.

Replace the type code with subclasses.



Motivation

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

This situation usually is 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 value of the type code. Such conditionals need to be refactored with Replace Conditional with Polymorphism. For this refactoring to work, the type code has to be replaced with an inheritance structure that will host the polymorphic behavior. Such an inheritance structure has a class with subclasses for each type code.

The simplest way to establish this structure is *Replace Type Code with Subclasses.* You take the class that has the type code and create a subclass for each type code. However, there are cases in which you can't do this. In the first the value of the type code changes after the object is

created. In the second 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.*

Replace Type Code with Subclasses is primarily a scaffolding move that enables Replace Conditional with Polymorphism. The trigger to use *Replace Type Code with Subclasses* is the presence of conditional statements. If there are no conditional statements, Replace Type Code with Class is the better and less critical move.

Another reason to *Replace Type Code with Subclasses* is the presence of features that are relevant only to objects with certain type codes. Once you've done this refactoring, you can use Push Down Method and Push Down Field to clarify that these features are relevant only in certain cases.

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

Mechanics

- Self-encapsulate the type code.

  ?rarr; *If the type code is passed into the constructor, you 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.

  ?rarr; *This value is 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 use the boring and unrealistic example of employee payment:

```
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 on the type code:

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

Because 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 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 with 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");
        }
    }
```

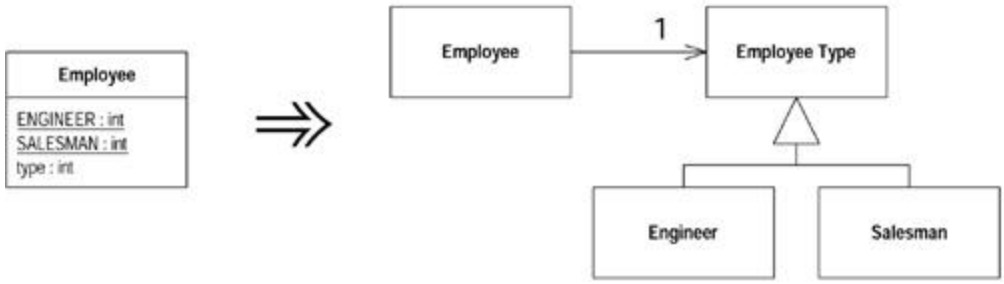
Of course this is the kind of switch statement I would prefer to avoid. But there is only one, and it is only used at creation.

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

Replace Type Code with State/Strategy

You have a type code that affects the behavior of a class, but you cannot use subclassing.

Replace the type code with a state object.



Motivation

This is similar to Replace Type Code with Subclasses, but can be used if the type code changes during the life of the object or if another 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 that better fits the specific circumstances. If you are trying to simplify a single algorithm with Replace Conditional with Polymorphism, 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, and 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.

    ?rarr; *It is easier to add the subclasses 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 again use the tiresome and brainless example of employee payment:

```
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;
  }
```

Here's an example of the kind of conditional behavior that would use these codes:

```
  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 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");
        }
    }
```

This means I now have a switch statement here. Once I'm finished refactoring, it will be the only one anywhere in the code, and it will be executed only when the type is changed. I can also use Replace Constructor with Factory Method to create factory methods for different cases. I can eliminate all the other case statements with a swift thrust of Replace Conditional with Polymorphism.

I like to finish Replace Type Code with State/Strategy 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:

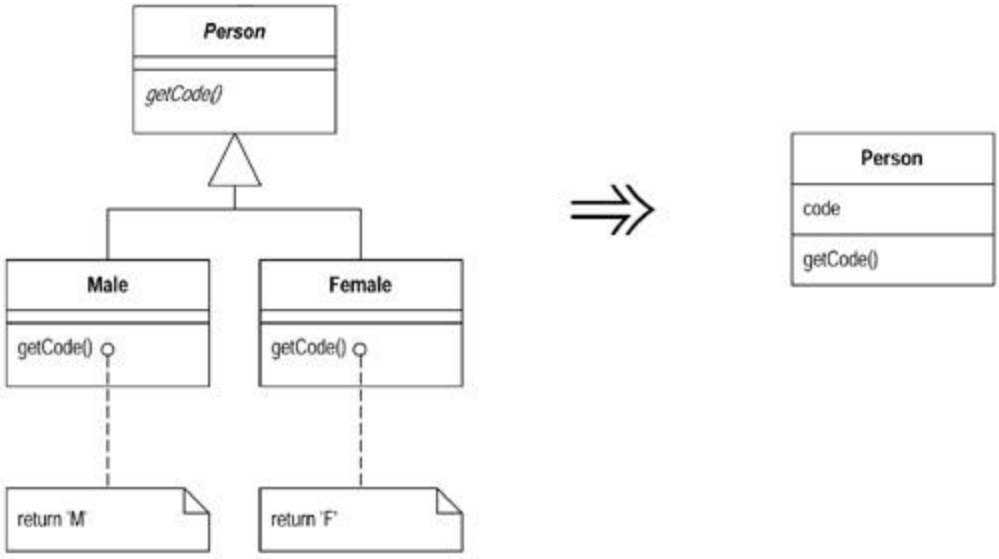
```
class Employee...
  int payAmount() {
      switch (getType()) {
          case EmployeeType.ENGINEER:
             return _monthlySalary;
          case EmployeeType.SALESMAN:
             return _monthlySalary + _commission;
          case EmployeeType.MANAGER:
             return _monthlySalary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

I'm now ready to use Replace Conditional with Polymorphism on `payAmount`.

Replace Subclass with Fields

You have subclasses that vary only in methods that return constant data.

Change the methods to superclass fields and eliminate the subclasses.



Motivation

You create subclasses to add features or allow behavior to vary. One form of variant behavior is the constant method [Beck]. A constant method is one that returns a hard-coded value. This can be very useful on subclasses that return different values for an accessor. You define the accessor in the superclass and implement it with different values on the subclass.

Although constant methods are useful, a subclass that consists only of constant methods is not doing enough to be worth existing. You can remove such subclasses completely by putting fields in the superclass. By doing that you remove the extra complexity of the subclasses.

Mechanics

In this case the object is immutable, so the next step is to define an equals method:

```
public boolean equals(Object arg) {
    if (! (arg instanceof Currency)) return false;
    Currency other = (Currency) arg;
    return (_code.equals(other._code));
}
```

If I define equals, I also need to define hashCode. The simple way to do this is to take the hash codes of all the fields used in the equals method and do a biwise xor (^) on them. Here it's easy because there's only one:

```
public int hashCode() {
    return _code.hashCode();
}
```

With both methods replaced, I can compile and test. I need to do both; otherwise any collection that relies on hashing, such as Hashtable, HashSet or HashMap, may act strangely.

Now I can create as many equal currencies as I like. I can get rid of all the controller behavior on the class and the factory method and just use the constructor, which I can now make public.

```
new Currency("USD").equals(new Currency("USD")) // now returns true
```

Replace Array with Object

You have an array in which certain elements mean different things.

Replace the array with an object that has 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

Arrays are a common structure for organizing data. However, they should be used only 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 to remember it or hope the comments are up to date. You can also encapsulate the information and use Move Method 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 start with an array that's used to hold the name, wins, and losses of a sports team. It would be declared as follows:

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

It would be used with code such as the following:

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

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

To turn this into an object, I begin by creating a class:

```
class Performance {}
```

For my first step I 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. When the array is created I use

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

When 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]);
```

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:

```
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. It is also useful, however, to replace the array internally. I can do this by adding a field for each array element and changing the accessors to use it:

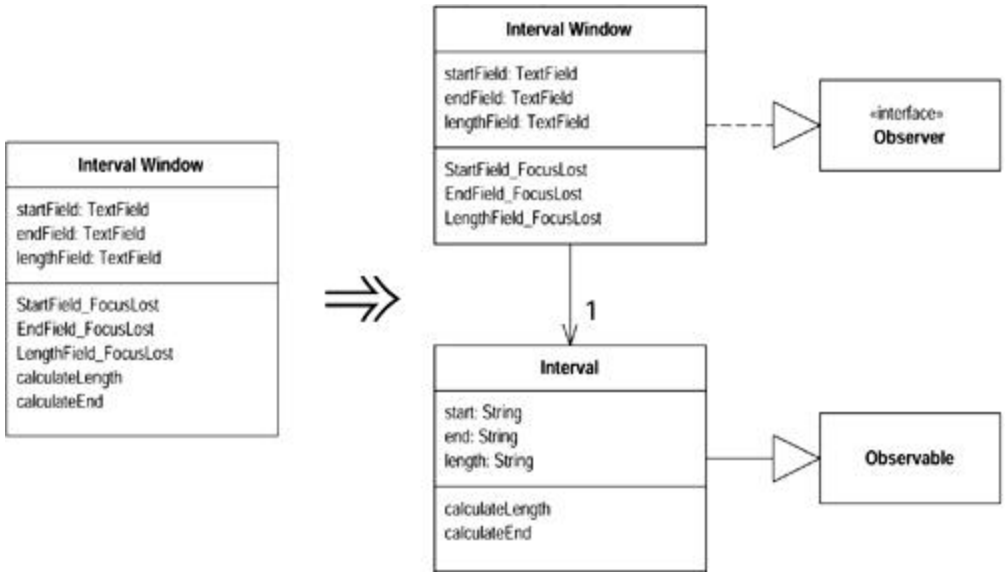
```
class Performance...
  public String getName() {
      return _name;
  }
  public void setName(String arg) {
      _name = arg;
  }
  private String _name;
```

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

Duplicate Observed Data

You have domain data available only in a GUI control, and domain methods need access.

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



Motivation

A well-layered system separates 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 becomes 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 the different pieces.

Although the behavior can be separated easily, the data often cannot. Data needs to be embedded in GUI control that has the same meaning as data that lives in the domain model. User interface frameworks, from model-view-controller (MVC) onward, used a multitiered system to provide mechanisms to allow you to provide this data and keep everything in sync.

```
        if (_intRate <= 0.0 || _duration <= 0.0) return result;
        result = (_income / _duration) * ADJ_FACTOR;
        return result;
    }
```

In these situations I prefer to put an explicit value on the returns from the guards. That way you can easily see the result of the guard's failing (I would also consider Replace Magic Number with Symbolic Constant here).

```
    public double getAdjustedCapital() {
        double result = 0.0;
        if (_capital <= 0.0) return 0.0;
        if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
        result = (_income / _duration) * ADJ_FACTOR;
        return result;
    }
```

With that done I can also remove the temp:

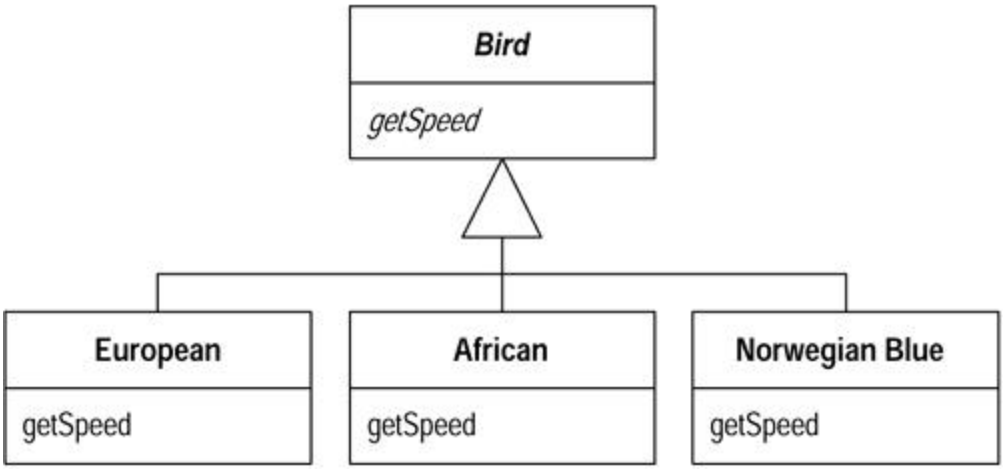
```
    public double getAdjustedCapital() {
        if (_capital <= 0.0) return 0.0;
        if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
        return (_income / _duration) * ADJ_FACTOR;
    }
```

Replace Conditional with Polymorphism

You have a conditional that chooses different behavior depending on the type of an object.

Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

```
    double getSpeed() {
        switch (_type) {
            case EUROPEAN:
               return getBaseSpeed();
            case AFRICAN:
               return getBaseSpeed() - getLoadFactor() *
_numberOfCoconuts;
            case NORWEGIAN_BLUE:
               return (_isNailed) ? 0 : getBaseSpeed(_voltage);
        }
        throw new RuntimeException ("Should be unreachable");
    }
```



Motivation

One of the grandest sounding words in object jargon is *polymorphism.* The essence of polymorphsim is that it allows you to avoid writing an explicit conditional when you have objects whose behavior varies depending on their types.

As a result you find that switch statements that switch on type codes or if-then-else statements that switch on type strings are much less common in an object-oriented program.

Polymorphism gives you many advantages. The biggest gain occurs when this same set of conditions appears in many places in the program. If you want to add a new type, you have to find and update all the conditionals. But with subclasses you just create a new subclass and provide the appropriate methods. Clients of the class don't need to know about the subclasses, which reduces the dependencies in your system and makes it easier to update.

Mechanics

Before you can begin with *Replace Conditional with Polymorphism* you need to have the necessary inheritance structure. You may already have this structure from previous refactorings. If you don't have the structure, you need to create it.

To create the inheritance structure you have two options: Replace Type Code with Subclasses and Replace Type Code with State/Strategy. Subclasses are the simplest option, so you should use them if you can. If you update the type code after the object is created, however, you cannot use subclassing and have to use the state/strategy pattern. You also need to use the state/strategy pattern if you are already subclassing this class for another reason. Remember that if several case statements are switching on the same type code, you only need to create one inheritance structure for that type code.

You can now attack the conditional. The code you target may be a switch (case) statement or an if statement.

- If the conditional statement is one part of a larger method, take apart the conditional statement and use Extract Method.
- If necessary use Move Method to place the conditional at the top of the inheritance structure.

- Pick one of the subclasses. Create a subclass method that overrides the conditional statement method. Copy the body of that leg of the conditional statement into the subclass method and adjust it to fit.

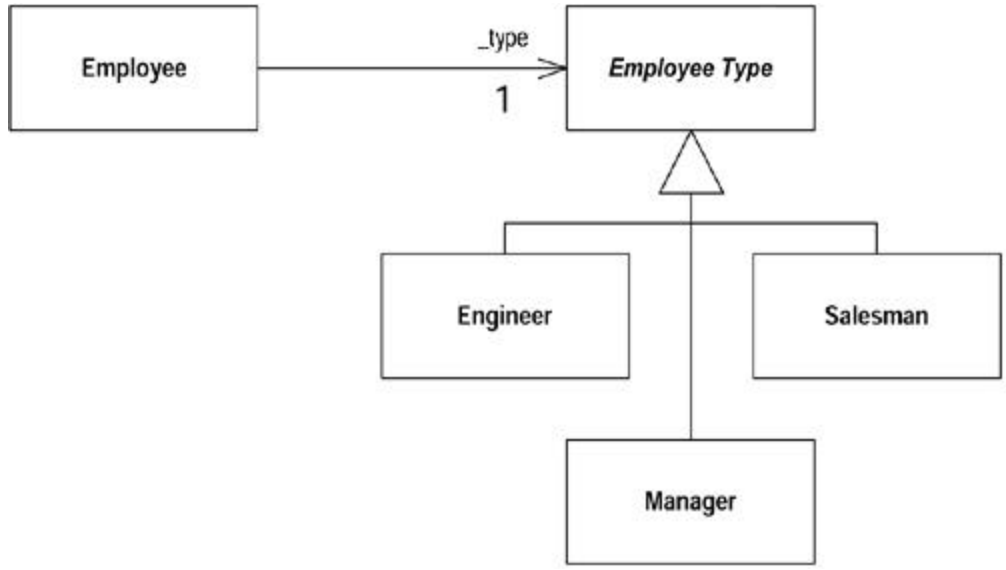
    ?rarr; *You may need to make some private members of the superclass protected in order to do this.*

- Compile and test.
- Remove the copied leg of the conditional statement.
- Compile and test.
- Repeat with each leg of the conditional statement until all legs are turned into subclass methods.
- Make the superclass method abstract.

Example

I use the tedious and simplistic example of employee payment. I'm using the classes after using Replace Type Code with State/Strategy so the objects look like Figure 9.1 (see the example in Chapter 8 for how we got here).

Figure 9.1. _The inheritance structure




```
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");
      }
  }
```

```
  int getType() {
      return _type.getTypeCode();
  }
  private EmployeeType _type;

abstract class EmployeeType...
  abstract int getTypeCode();

class Engineer extends EmployeeType...
  int getTypeCode() {
      return Employee.ENGINEER;
  }

... and other subclasses
```

The case statement is already nicely extracted, so there is nothing to do there. I do need to move it into the employee type, because that is the class that is being subclassed.

```
class EmployeeType...
  int payAmount(Employee emp) {
      switch (getTypeCode()) {
          case ENGINEER:
             return emp.getMonthlySalary();
          case SALESMAN:
             return emp.getMonthlySalary() + emp.getCommission();
          case MANAGER:
             return emp.getMonthlySalary() + emp.getBonus();
          default:
             throw new RuntimeException("Incorrect Employee");
      }
  }
```

Because I need data from the employee, I need to pass in the employee as an argument. Some of this data might be moved to the employee type object, but that is an issue for another refactoring.

When this compiles, I change the `payAmount` method in Employee to delegate to the new class:

```
class Employee...
  int payAmount() {
      return _type.payAmount(this);
  }
```

Now I can go to work on the case statement. It's rather like the way small boys kill insects—I remove one leg at a time. First I copy the Engineer leg of the case statement onto the Engineer class.

```
class Engineer...
  int payAmount(Employee emp) {
      return emp.getMonthlySalary();
  }
```

This new method overrides the whole case statement for engineers. Because I'm paranoid, I sometimes put a trap in the case statement:

```
  class EmployeeType...
    int payAmount(Employee emp) {
        switch (getTypeCode()) {
            case ENGINEER:
               throw new RuntimeException ("Should be being
overridden");
            case SALESMAN:
               return emp.getMonthlySalary() + emp.getCommission();
            case MANAGER:
               return emp.getMonthlySalary() + emp.getBonus();
            default:
               throw new RuntimeException("Incorrect Employee");
        }
    }
```

carry on until all the legs are removed:

```
  class Salesman...
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() + emp.getCommission();
    }

  class Manager...
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() + emp.getBonus();
    }
```

and then declare the superclass method abstract:

```
  class EmployeeType...
    abstract int payAmount(Employee emp);
```

Introduce Null Object

You have repeated checks for a null value.

Replace the null value with a null object.

```
          if (customer == null) plan = BillingPlan.basic();
          else plan = customer.getPlan();
```

```
        return new Dollars (result);
    }

    protected int usageInRange(int start, int end) {
        if (lastUsage() > start) return Math.min(lastUsage(),end) -
start;
        else return 0;
    }
```

The trick is to spot code that is repetitive on the basis of a few values that can be passed in as parameters.

Replace Parameter with Explicit Methods

You have a method that runs different code depending on the values of an enumerated parameter.

Create a separate method for each value of the parameter.

```
void setValue (String name, int value) {
    if (name.equals("height"))
        _height = value;
    if (name.equals("width"))
        _width = value;
    Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg) {
    _height = arg;
}
void setWidth (int arg) {
    _width = arg;
}
```

Motivation

Replace Parameter with Explicit Methods is the reverse of Parameterize Method. The usual case for the former is that you have discrete values of a parameter, test for those values in a conditional, and do different things. The caller has to decide what it wants to do by setting the parameter, so you might as well provide different methods and avoid the conditional. You not only avoid the conditional behavior but also gain compile time checking. Furthermore your interface also is clearer. With the parameter, any programmer using the method needs not only to look at the methods on the class but also to determine a valid parameter value. The latter is often poorly documented.

The clarity of the explicit interface can be worthwhile even when the compile time checking isn't an advantage. `Switch.beOn()` is a lot clearer than `Switch.setState(true),` even when all you are doing is setting an internal boolean field.

You shouldn't use *Replace Parameter with Explicit Methods* when the parameter values are likely to change a lot. If this happens and you are just setting a field to the passed in parameter, use a simple setter. If you need conditional behavior, you need Replace Conditional with Polymorphism.

Mechanics

- Create an explicit method for each value of the parameter.
- For each leg of the conditional, call the appropriate new method.
- Compile and test after changing each leg.
- Replace each caller of the conditional method with a call to the appropriate new method.
- Compile and test.
- When all callers are changed, remove the conditional method.

Example

I want to create a subclass of employee on the basis of a passed in parameter, often the result of Replace Constructor with Factory Method:

```
   static final int ENGINEER = 0;
   static final int SALESMAN = 1;
   static final int MANAGER = 2;

   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");
       }
   }
```

Because this is a factory method, I can't use Replace Conditional with Polymorphism, because I haven't created the object yet. I don't expect too many new subclasses, so an explicit interface makes sense. First I create the new methods:

```
   static Employee createEngineer() {
       return new Engineer();
   }
   static Employee createSalesman() {
       return new Salesman();
   }
   static Employee createManager() {
       return new Manager();
   }
```

One by one I replace the cases in the switch statements with calls to the explicit methods:

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

I compile and test after changing each leg, until I've replaced them all:

```
   static Employee create(int type) {
       switch (type) {
           case ENGINEER:
              return Employee.createEngineer();
           case SALESMAN:
              return Employee.createSalesman();
           case MANAGER:
              return Employee.createManager();
           default:
              throw new IllegalArgumentException("Incorrect type code
value");
       }
   }
```

Now I move on to the callers of the old create method. I change code such as

```
  Employee kent = Employee.create(ENGINEER)
```

to

```
  Employee kent = Employee.createEngineer()
```

Once I've done that for all the callers of `create`, I can remove the `create` method. I may also be able to get rid of the constants.

Preserve Whole Object

You are getting several values from an object and passing these values as parameters in a method call.

Send the whole object instead.

```
      int low = daysTempRange().getLow();
```

This new method overrides the whole case statement for engineers. Because I'm paranoid, I sometimes put a trap in the case statement:

```
  class EmployeeType...
    int payAmount(Employee emp) {
        switch (getTypeCode()) {
            case ENGINEER:
               throw new RuntimeException ("Should be being
overridden");
            case SALESMAN:
               return emp.getMonthlySalary() + emp.getCommission();
            case MANAGER:
               return emp.getMonthlySalary() + emp.getBonus();
            default:
               throw new RuntimeException("Incorrect Employee");
        }
    }
```

carry on until all the legs are removed:

```
  class Salesman...
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() + emp.getCommission();
    }

  class Manager...
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() + emp.getBonus();
    }
```

and then declare the superclass method abstract:

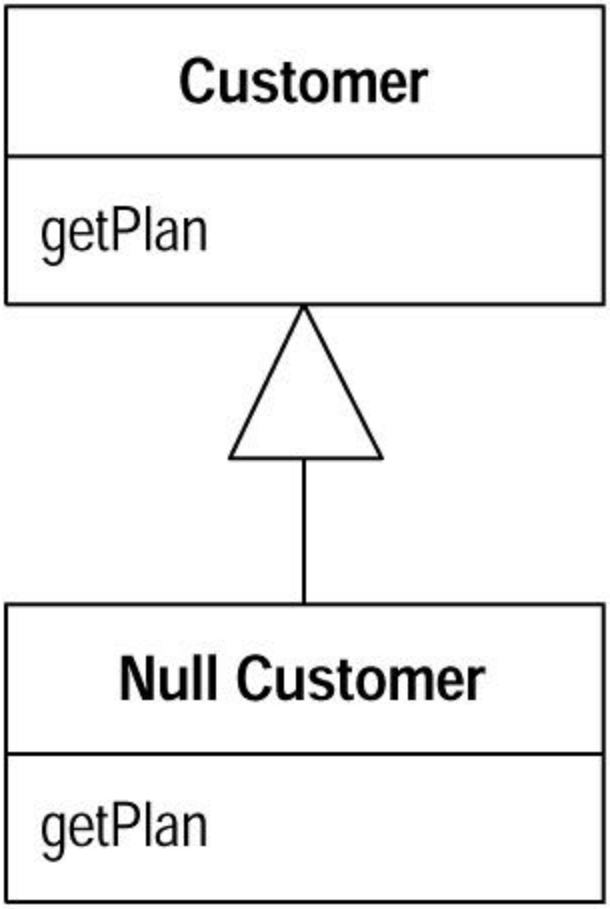
```
  class EmployeeType...
    abstract int payAmount(Employee emp);
```

Introduce Null Object

You have repeated checks for a null value.

Replace the null value with a null object.

```
          if (customer == null) plan = BillingPlan.basic();
          else plan = customer.getPlan();
```



Motivation

The essence of polymorphism is that instead of asking an object what type it is and then invoking some behavior based on the answer, you just invoke the behavior. The object, depending on its type, does the right thing. One of the less intuitive places to do this is where you have a null value in a field. I'll let Ron Jeffries tell the story:

> *Ron Jeffries*
>
> We first started using the null object pattern when Rich Garzaniti found that lots of code in the system would check objects for presence before sending a message to the object. We might ask an object for its person, then ask the result whether it was null. If the object was present, we would ask it for its rate. We were doing this in several places, and the resulting duplicate code was getting annoying.
>
> So we implemented a missing-person object that answered a zero rate (we call our null objects missing objects). Soon missing person knew a lot of methods, such as rate. Now we have more than 80 null-object classes.

Our most common use of null objects is in the display of information. When we display, for example, a person, the object may or may not have any of perhaps 20 instance variables. If these were allowed to be null, the printing of a person would be very complex. Instead we plug in various null objects, all of which know how to display themselves in an orderly way. This got rid of huge amounts of procedural code.

Our most clever use of null object is the missing Gemstone session. We use the Gemstone database for production, but we prefer to develop without it and push the new code to Gemstone every week or so. There are various points in the code where we have to log in to a Gemstone session. When we are running without Gemstone, we simply plug in a missing Gemstone session. It looks the same as the real thing but allows us to develop and test without realizing the database isn't there.

Another helpful use of null object is the missing bin. A bin is a collection of payroll values that often have to be summed or looped over. If a particular bin doesn't exist, we answer a missing bin, which acts just like an empty bin. The missing bin knows it has zero balance and no values. By using this approach, we eliminate the creation of tens of empty bins for each of our thousands of employees.

An interesting characteristic of using null objects is that things almost never blow up. Because the null object responds to all the same messages as a real one, the system generally behaves normally. This can sometimes make it difficult to detect or find a problem, because nothing ever breaks. Of course, as soon as you begin inspecting the objects, you'll find the null object somewhere where it shouldn't be.

Remember, null objects are always constant: nothing about them ever changes. Accordingly, we implement them using the Singleton pattern [Gang of Four]. Whenever you ask, for example, for a missing person, you always get the single instance of that class.

You can find more details about the null object pattern in Woolf [Woolf].

Mechanics

- Create a subclass of the source class to act as a null version of the class. Create an `isNull` operation on the source class and the null class. For the source class it should return false, for the null class it should return true.

  ?rarr; *You may find it useful to create an explicitly nullable interface for the* `isNull` *method.*

  ?rarr; *As an alternative you can use a testing interface to test for nullness.*

- Compile.
- Find all places that can give out a null when asked for a source object. Replace them to give out a null object instead.

- Find all places that compare a variable of the source type with null and replace them with a call `isNull`.

    ?rarr; *You may be able to do this by replacing one source and its clients at a time and compiling and testing between working on sources.*

    ?rarr; *A few assertions that check for null in places where you should no longer see it can be useful.*

- Compile and test.
- Look for cases in which clients invoke an operation if not null and do some alternative behavior if null.
- For each of these cases override the operation in the null class with the alternative behavior.
- Remove the condition check for those that use the overriden behavior, compile, and test.

Example

A utility company knows about sites: the houses and apartments that use the utility's services. At any time a site has a customer.

```
class Site...
  Customer getCustomer() {
      return _customer;
  }
  Customer _customer;
```

There are various features of a customer. I look at three of them.

```
class Customer...
  public String getName() {...}
  public BillingPlan getPlan() {...}
  public PaymentHistory getHistory() {...}
```

The payment history has its own features:

```
public class PaymentHistory...
  int getWeeksDelinquentInLastYear()
```

The getters I show allow clients to get at this data. However, sometimes I don't have a customer for a site. Someone may have moved out and I don't yet know who has moved in. Because this can happen we have to ensure that any code that uses the customer can handle nulls. Here are a few example fragments:

```
        Customer customer = site.getCustomer();
```

```
          BillingPlan plan;
          if (customer == null) plan = BillingPlan.basic();
          else plan = customer.getPlan();
 ...
          String customerName;
          if (customer == null) customerName = "occupant";
          else customerName = customer.getName();
 ...
          int weeksDelinquent;
          if (customer == null) weeksDelinquent = 0;
          else weeksDelinquent =
customer.getHistory().getWeeksDelinquentInLastYear();
```

In these situations I may have many clients of site and customer, all of which have to check for nulls and all of which do the same thing when they find one. Sounds like it's time for a null object.

The first step is to create the null customer class and modify the customer class to support a query for a null test:

```
  class NullCustomer extends Customer {
    public boolean isNull() {
        return true;
    }
  }

  class Customer...
    public boolean isNull() {
        return false;
    }

    protected Customer() {} //needed by the NullCustomer
```

If you aren't able to modify the Customer class you can use a testing interface.

If you like, you can signal the use of null object by means of an interface:

```
  interface Nullable {
    boolean isNull();
  }

  class Customer implements Nullable
```

I like to add a factory method to create null customers. That way clients don't have to know about the null class:

```
  class Customer...
    static Customer newNull() {
        return new NullCustomer();
    }
```

Now comes the difficult bit. Now I have to return this new null object whenever I expect a null and replace the tests of the form `foo == null` with tests of the form `foo.isNull()`. I find it useful to look for all the places where I ask for a customer and modify them so that they return a null customer rather than null.

```
  class Site...
    Customer getCustomer() {
        return (_customer == null) ?
            Customer.newNull():
            _customer;
    }
```

I also have to alter all uses of this value so that they test with `isNull()` rather than `== null.`

```
        Customer customer = site.getCustomer();
        BillingPlan plan;
        if (customer.isNull()) plan = BillingPlan.basic();
        else plan = customer.getPlan();
  ...
        String customerName;
        if (customer.isNull()) customerName = "occupant";
        else customerName = customer.getName();
  ...
        int weeksDelinquent;
        if (customer.isNull()) weeksDelinquent = 0;
        else weeksDelinquent =
customer.getHistory().getWeeksDelinquentInLastYear();
```

There's no doubt that this is the trickiest part of this refactoring. For each source of a null I replace, I have to find all the times it is tested for nullness and replace them. If the object is widely passed around, these can be hard to track. I have to find every variable of type customer and find everywhere it is used. It is hard to break this process into small steps. Sometimes I find one source that is used in only a few places, and I can replace that source only. But most of the time, however, I have to make many widespread changes. The changes aren't too difficult to back out of, because I can find calls of `isNull` without too much difficulty, but this is still a messy step.

Once this step is done, and I've compiled and tested, I can smile. Now the fun begins. As it stands I gain nothing from using `isNull` rather than `== null.` The gain comes as I move behavior to the null customer and remove conditionals. I can make these moves one at a time. I begin with the name. Currently I have client code that says

```
        String customerName;
        if (customer.isNull()) customerName = "occupant";
        else customerName = customer.getName();
```

I add a suitable name method to the null customer:

```
  class NullCustomer...
    public String getName(){
        return "occupant";
```

```
    }
```

Now I can make the conditional code go away:

```
        String customerName = customer.getName();
```

I can do the same for any other method in which there is a sensible general response to a query. I can also do appropriate actions for modifiers. So client code such as

```
        if (! customer.isNull())
            customer.setPlan(BillingPlan.special());
```

can be replaced with

```
  customer.setPlan(BillingPlan.special());

  class NullCustomer...
    public void setPlan (BillingPlan arg) {}
```

Remember that this movement of behavior makes sense only when most clients want the same response. Notice that I said *most* not *all.* Any clients who want a different response to the standard one can still test using `isNull.` You benefit when many clients want to do the same thing; they can simply rely on the default null behavior.

The example contains a slightly different case—client code that uses the result of a call to customer:

```
        if (customer.isNull()) weeksDelinquent = 0;
        else weeksDelinquent =
customer.getHistory().getWeeksDelinquentInLastYear();
```

I can handle this by creating a null payment history:

```
  class NullPaymentHistory extends PaymentHistory...
    int getWeeksDelinquentInLastYear() {
        return 0;
    }
```

I modify the null customer to return it when asked:

```
  class NullCustomer...
    public PaymentHistory getHistory() {
        return PaymentHistory.newNull();
    }
```

Again I can remove the conditional code:

```
        int weeksDelinquent =
customer.getHistory().getWeeksDelinquentInLastYear();
```

You often find that null objects return other null objects.

Example: Testing Interface

The testing interface is an alternative to defining an isNull method. In this approach I create a null interface with no methods defined:

```
  interface Null {}
```

I then implement null in my null objects:

```
  class NullCustomer extends Customer implements Null...
```

I then test for nullness with the instanceof operator:

```
  aCustomer instanceof Null
```

I normally run away screaming from the `instanceof` operator, but in this case it is okay to use it. It has the particular advantage that I don't need to change the customer class. This allows me to use the null object even when I don't have access to customer's source code.

Other Special Cases

When carrying out this refactoring, you can have several kinds of null. Often there is a difference between there is no customer (new building and not yet moved in) and there is an unknown customer (we think there is someone there, but we don't know who it is). If that is the case, you can build separate classes for the different null cases. Sometimes null objects actually can carry data, such as usage records for the unknown customer, so that we can bill the customers when we find out who they are.

In essence there is a bigger pattern here, called *special case.* A special case class is a particular instance of a class with special behavior. So `UnknownCustomer` and `NoCustomer` would both be special cases of `Customer.` You often see special cases with numbers. Floating points in Java have special cases for positive and negative infinity and for not a number (NaN). The value of special cases is that they help reduce dealing with errors. Floating point operations don't throw exceptions. Doing any operation with NaN yields another NaN in the same way that accessors on null objects usually result in other null objects.

Introduce Assertion

A section of code assumes something about the state of the program.

Make the assumption explicit with an assertion.

Chapter 7. Moving Features Between Objects

One of the most fundamental, if not the fundamental, decision in object design is deciding where to put responsibilities. I've been working with objects for more than a decade, but I still never get it right the first time. That used to bother me, but now I realize that I can use refactoring to change my mind in these cases.

Often I can resolve these problems simply by using Move Method and Move Field to move the behavior around. If I need to use both, I prefer to use Move Field first and then Move Method.

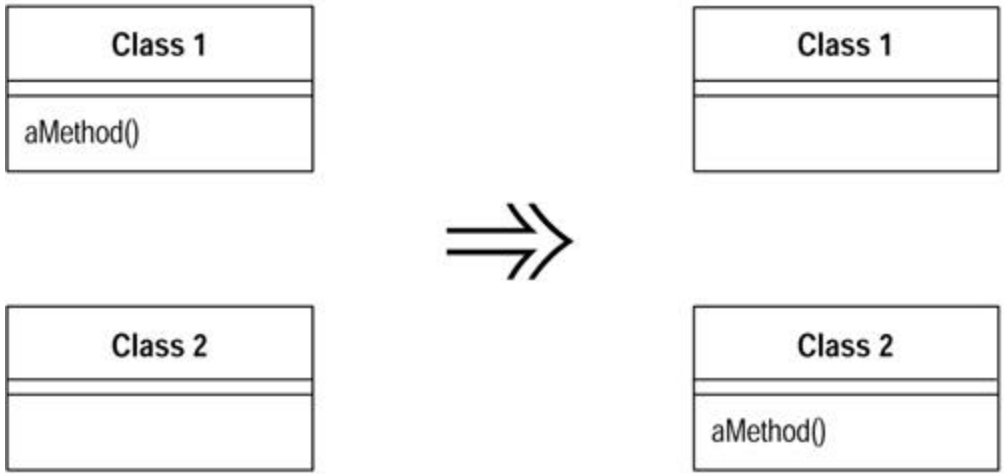
Often classes become bloated with too many responsibilities. In this case I use Extract Class to separate some of these responsibilities. If a class becomes too irresponsible, I use Inline Class to merge it into another class. If another class is being used, it often is helpful to hide this fact with Hide Delegate. Sometimes hiding the delegate class results in constantly changing the owner's interface, in which case you need to use Remove Middle Man

The last two refactorings in this chapter, Introduce Foreign Method and Introduce Local Extension are special cases. I use these only when I'm not able to access the source code of a class, yet I want to move responsibilities to this unchangeable class. If it is only one or two methods, I use Introduce Foreign Method; for more than one or two methods, I use Introduce Local Extension.

Move Method

A method is, or will be, using or used by more features of another class than the class on which it is defined.

Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.



Motivation

Moving methods is the bread and butter of refactoring. I move methods when classes have too much behavior or when classes are collaborating too much and are too highly coupled. By

moving methods around, I can make the classes simpler and they end up being a more crisp implementation of a set of responsibilities.

I usually look through the methods on a class to find a method that seems to reference another object more than the object it lives on. A good time to do this is after I have moved some fields. Once I see a likely method to move, I take a look at the methods that call it, the methods it calls, and any redefining methods in the hierarchy. I assess whether to go ahead on the basis of the object with which the method seems to have more interaction.

It's not always an easy decision to make. If I am not sure whether to move a method, I go on to look at other methods. Moving other methods often makes the decision easier. Sometimes the decision still is hard to make. Actually it is then no big deal. If it is difficult to make the decision, it probably does not matter that much. Then I choose according to instinct; after all, I can always change it again later.

Mechanics

- Examine all features used by the source method that are defined on the source class. Consider whether they also should be moved.

  ?rarr; *If a feature is used only by the method you are about to move, you might as well move it, too. If the feature is used by other methods, consider moving them as well. Sometimes it is easier to move a clutch of methods than to move them one at a time.*

- Check the sub- and superclasses of the source class for other declarations of the method.

  ?rarr; *If there are any other declarations, you may not be able to make the move, unless the polymorphism can also be expressed on the target.*

- Declare the method in the target class.

  ?rarr; *You may choose to use a different name, one that makes more sense in the target class.*

- Copy the code from the source method to the target. Adjust the method to make it work in its new home.

  ?rarr; *If the method uses its source, you need to determine how to reference the source object from the target method. If there is no mechanism in the target class, pass the source object reference to the new method as a parameter.*

  ?rarr; *If the method includes exception handlers, decide which class should logically handle the exception. If the source class should be responsible, leave the handlers behind.*

- Compile the target class.
- Determine how to reference the correct target object from the source.

?rarr; *There may be an existing field or method that will give you the target. If not, see whether you can easily create a method that will do so. Failing that, you need to create a new field in the source that can store the target. This may be a permanent change, but you can also make it temporarily until you have refactored enough to remove it.*

- Turn the source method into a delegating method.
- Compile and test.
- Decide whether to remove the source method or retain it as a delegating method.

?rarr; *Leaving the source as a delegating method is easier if you have many references.*

- If you remove the source method, replace all the references with references to the target method.

?rarr; *You can compile and test after changing each reference, although it is usually easier to change all references with one search and replace.*

- Compile and test.

Example

An account class illustrates this refactoring:

```
 class Account...
   double overdraftCharge() {
       if (_type.isPremium()) {
           double result = 10;
           if (_daysOverdrawn > 7) result += (_daysOverdrawn - 7) *
0.85;
           return result;
       }
       else return _daysOverdrawn * 1.75;
   }

   double bankCharge() {
       double result = 4.5;
       if (_daysOverdrawn > 0) result += overdraftCharge();
       return result;
   }
   private AccountType _type;
   private int _daysOverdrawn;
```

Let's imagine that there are going to be several new account types, each of which has its own rule for calculating the overdraft charge. So I want to move the overdraft charge method over to the account type.

The first step is to look at the features that the overdraftCharge method uses and consider whether it is worth moving a batch of methods together. In this case I need the `_daysOverdrawn` field to remain on the account class, because that will vary with individual accounts.

Next I copy the method body over to the account type and get it to fit.

```
class AccountType...
  double overdraftCharge(int daysOverdrawn) {
      if (isPremium()) {
          double result = 10;
          if (daysOverdrawn > 7) result += (daysOverdrawn - 7) * 0.85;
          return result;
      }
      else return daysOverdrawn * 1.75;
  }
```

In this case fitting means removing the `_type` from uses of features of the account type, and doing something about the features of account that I still need. When I need to use a feature of the source class I can do one of four things: (1) move this feature to the target class as well, (2) create or use a reference from the target class to the source, (3) pass the 0source object as a parameter to the method, (4) if the feature is a variable, pass it in as a parameter.

In this case I passed the variable as a parameter.

Once the method fits and compiles in the target class, I can replace the source method body with a simple delegation:

```
class Account...
  double overdraftCharge() {
      return _type.overdraftCharge(_daysOverdrawn);
  }
```

At this point I can compile and test.

I can leave things like this, or I can remove the method in the source class. To remove the method I need to find all callers of the method and redirect them to call the method in account type:

```
class Account...
  double bankCharge() {
      double result = 4.5;
      if (_daysOverdrawn > 0) result +=
_type.overdraftCharge(_daysOverdrawn);
      return result;
  }
```

Once I've replaced all the callers, I can remove the method declaration in account. I can compile and test after each removal, or do them in a batch. If the method isn't private, I need to look for other classes that use this method. In a strongly typed language, the compilation after removal of the source declaration finds anything I missed.

In this case the method referred only to a single field, so I could just pass this field in as a variable. If the method called another method on the account, I wouldn't have been able to do that. In those cases I need to pass in the source object:

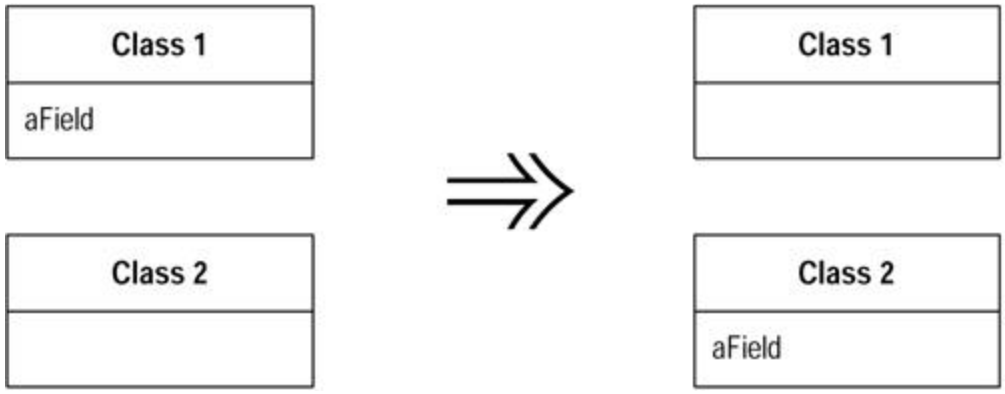
```
class AccountType...
  double overdraftCharge(Account account) {
      if (isPremium()) {
          double result = 10;
          if (account.getDaysOverdrawn() > 7)
             result += (account.getDaysOverdrawn() - 7) * 0.85;
          return result;
      }
      else return account.getDaysOverdrawn() * 1.75;
  }
```

I also pass in the source object if I need several features of the class, although if there are too many, further refactoring is needed. Typically I need to decompose and move some pieces back.

Move Field

A field is, or will be, used by another class more than the class on which it is defined.

Create a new field in the target class, and change all its users.



Motivation

Moving state and behavior between classes is the very essence of refactoring. As the system develops, you find the need for new classes and the need to shuffle responsibilities around. A design decision that is reasonable and correct one week can become incorrect in another. That is not a problem; the only problem is not to do something about it.

I consider moving a field if I see more methods on another class using the field than the class itself. This usage may be indirect, through getting and setting methods. I may choose to move the methods; this decision based on interface. But if the methods seem sensible where they are, I move the field.

Another reason for field moving is when doing Extract Class. In that case the fields go first and then the methods.

Mechanics

- If the field is public, use Encapsulate Field.

?rarr; *If you are likely to be moving the methods that access it frequently or if a lot of methods access the field, you may find it useful to use* Self Encapsulate Field

- Compile and test.
- Create a field in the target class with getting and setting methods.
- Compile the target class.
- Determine how to reference the target object from the source.

?rarr; *An existing field or method may give you the target. If not, see whether you can easily create a method that will do so. Failing that, you need to create a new field in the source that can store the target. This may be a permanent change, but you can also do it temporarily until you have refactored enough to remove it.*

- Remove the field on the source class.
- Replace all references to the source field with references to the appropriate method on the target.

?rarr; *For accesses to the variable, replace the reference with a call to the target object's getting method; for assignments, replace the reference with a call to the setting method.*

?rarr; *If the field is not private, look in all the subclasses of the source for references.*

- Compile and test.

Example

Here is part of an account class:

```
class Account...
  private AccountType _type;
  private double _interestRate;

  double interestForAmount_days (double amount, int days) {
      return _interestRate * amount * days / 365;
  }
```

I want to move the interest rate field to the account type. There are several methods with that reference, of which `interestForAmount_days` is one example.I next create the field and accessors in the account type:

```
class AccountType...
  private double _interestRate;


  void setInterestRate (double arg) {
      _interestRate = arg;
  }
```

```
double getInterestRate () {
    return _interestRate;
}
```

I can compile the new class at this point.

Now I redirect the methods from the account class to use the account type and remove the interest rate field in the account. I must remove the field to be sure that the redirection is actually happening. This way the compiler helps spot any method I failed to redirect.

```
private double _interestRate;

double interestForAmount_days (double amount, int days) {
    return _type.getInterestRate() * amount * days / 365;
}
```

Example: Using Self-Encapsulation

If a lot of methods use the interest rate field, I might start by using Self Encapsulate Field:

```
class Account...
  private AccountType _type;
  private double _interestRate;

  double interestForAmount_days (double amount, int days) {
      return getInterestRate() * amount * days / 365;
  }

  private void setInterestRate (double arg) {
      _interestRate = arg;
  }

  private double getInterestRate () {
      return _interestRate;
  }
```

That way I only need to do the redirection for the accessors:

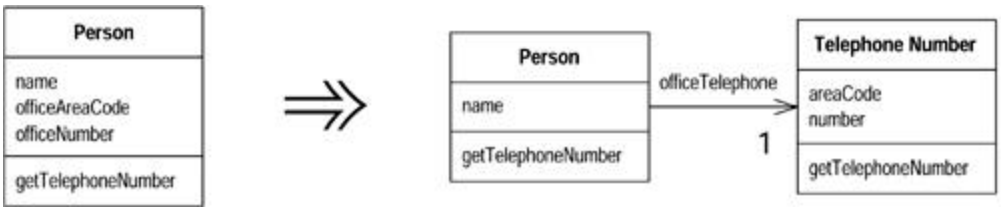
```
  double interestForAmountAndDays (double amount, int days) {
      return getInterestRate() * amount * days / 365;
  }

  private void setInterestRate (double arg) {
      _type.setInterestRate(arg);
  }

  private double getInterestRate () {
      return _type.getInterestRate();
  }
```

I can redirect the clients of the accessors to use the new object later if I want. Using self-encapsulation allows me to take a smaller step. This is useful if I'm doing a lot of things with the class. In particular, it simplifies use Move Method to move methods to the target class. If they refer to the accessor, such references don't need to change.

Extract Class

You have one class doing work that should be done by two.

Create a new class and move the relevant fields and methods from the old class into the new class.



Motivation

You've probably heard that a class should be a crisp abstraction, handle a few clear responsibilities, or some similar guideline. In practice, classes grow. You add some operations here, a bit of data there. You add a responsibility to a class feeling that it's not worth a separate class, but as that responsibility grows and breeds, the class becomes too complicated. Soon your class is as crisp as a microwaved duck.

Such a class is one with many methods and quite a lot of data. A class that is too big to understand easily. You need to consider where it can be split, and you split it. A good sign is that a subset of the data and a subset of the methods seem to go together. Other good signs are subsets of data that usually change together or are particularly dependent on each other. A useful test is to ask yourself what would happen if you removed a piece of data or a method. What other fields and methods would become nonsense?

One sign that often crops up later in development is the way the class is subtyped. You may find that subtyping affects only a few features or that some features need to be subtyped one way and other features a different way.

Mechanics

- Decide how to split the responsibilities of the class.
- Create a new class to express the split-off responsibilities.

  ?rarr; *If the responsibilities of the old class no longer match its name, rename the old class.*

- Make a link from the old to the new class.

  ?rarr; *You may need a two-way link. But don't make the back link until you find you need it.*