Refactoring

Refactoring

- Martin Fowler's definition :
  - (noun) a change made to the internal structure of software to make it easier to understand and cheaper to modify **without changing its observable behaviour**
  - (verb) to reconstruct software by applying a series of refactorings **without changing its observable behaviour**
- improves maintainability :
  - understandability
  - adaptability
  - extensibility
  - complexity : cohesion and coupling

Refactoring

- art of safely improving the design of existing code
    - does not include any changes in the system
    - not rewriting from scratch
    - not just any restructuring intended to improve code
    - changes the balance point between up-front design and emergent design
    - can be small or large

Refactoring : Simple Example

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
} else {
    total = price * 0.98;
    send();
}
```

duplicate code

Refactoring

```
if (isSpecialDeal()) {
    total = price * 0.95;
} else {
    total = price * 0.98;
}
send();
```

Refactoring

- Where to start?
  - codes that needs to be refactored
- Bad smell
  - possible indication of deeper problem or design flaw
  - not technically a bug, but if ‘left untreated’ may lead to bugs later
  - identified through
    - experience
    - code review
    - code metrics
    - tools

Refactoring

- basic pattern or cycle
  - while smell remains :
    - choose the worst smell
    - select a refactoring that will address the smell
    - apply the refactoring

- to find simplest design
  - Only and Only Once (OOO)
    - runs all the tests
    - has no duplicated logic

      (beware of hidden duplication such as parallel class hierarchies)
    - state every intention importance
    - Has the fewest possible classes and methods

Flow of Refactoring

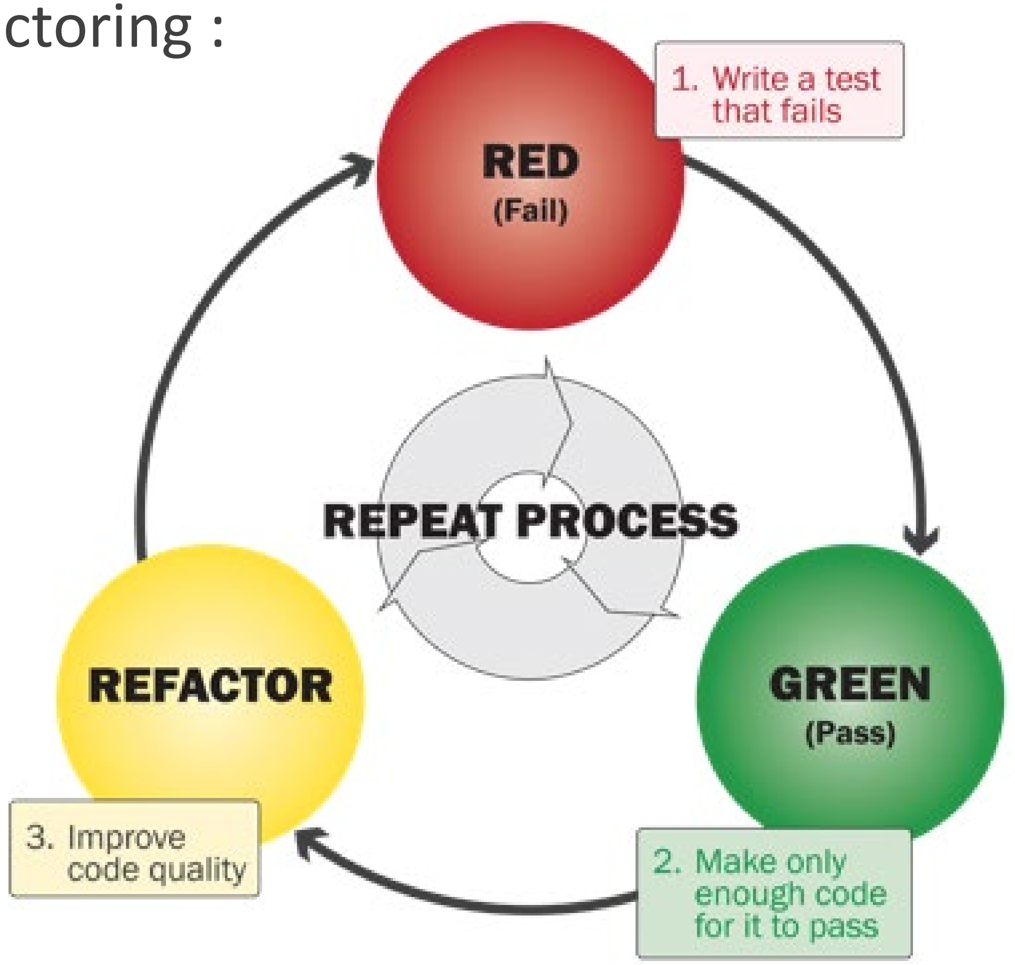
- with automated tests :
  1. make smallest discrete change that will compile, run, and function
  2. run existing tests
  3. make next smallest discrete change
  4. run existing tests again

  … repeat change + tests

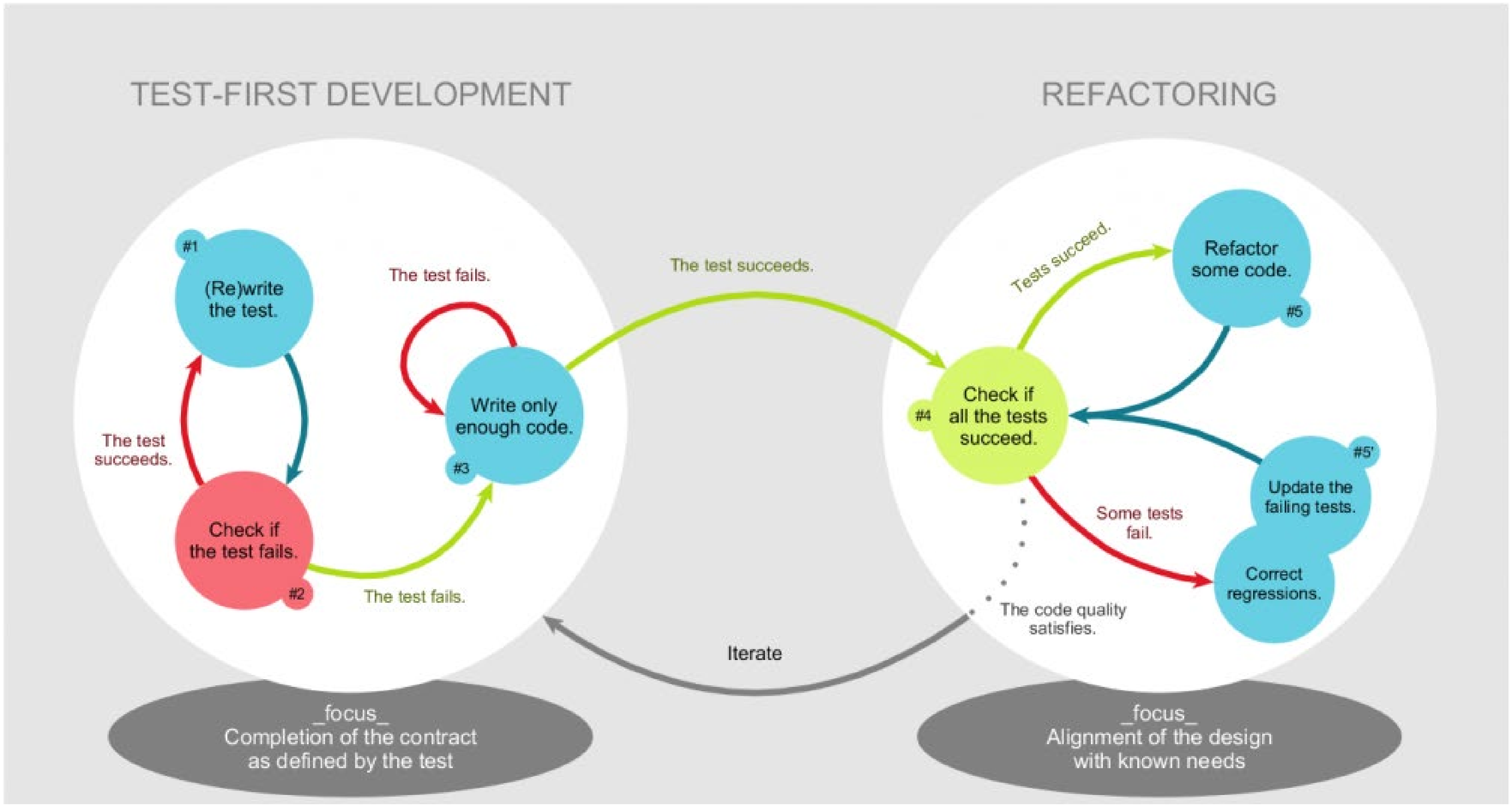
  6. when refactoring is in place and all tests run clean, remove the old smelly parallel code
  7. once tests run clean after that, done!

Flow of Refactoring

- TDD Refactoring :


RED
(Fail)
1. Write a test that fails
GREEN
(Pass)
2. Make only enough code for it to pass
REFACTOR
3. Improve code quality
REPEAT PROCESS

Flow of Refactoring

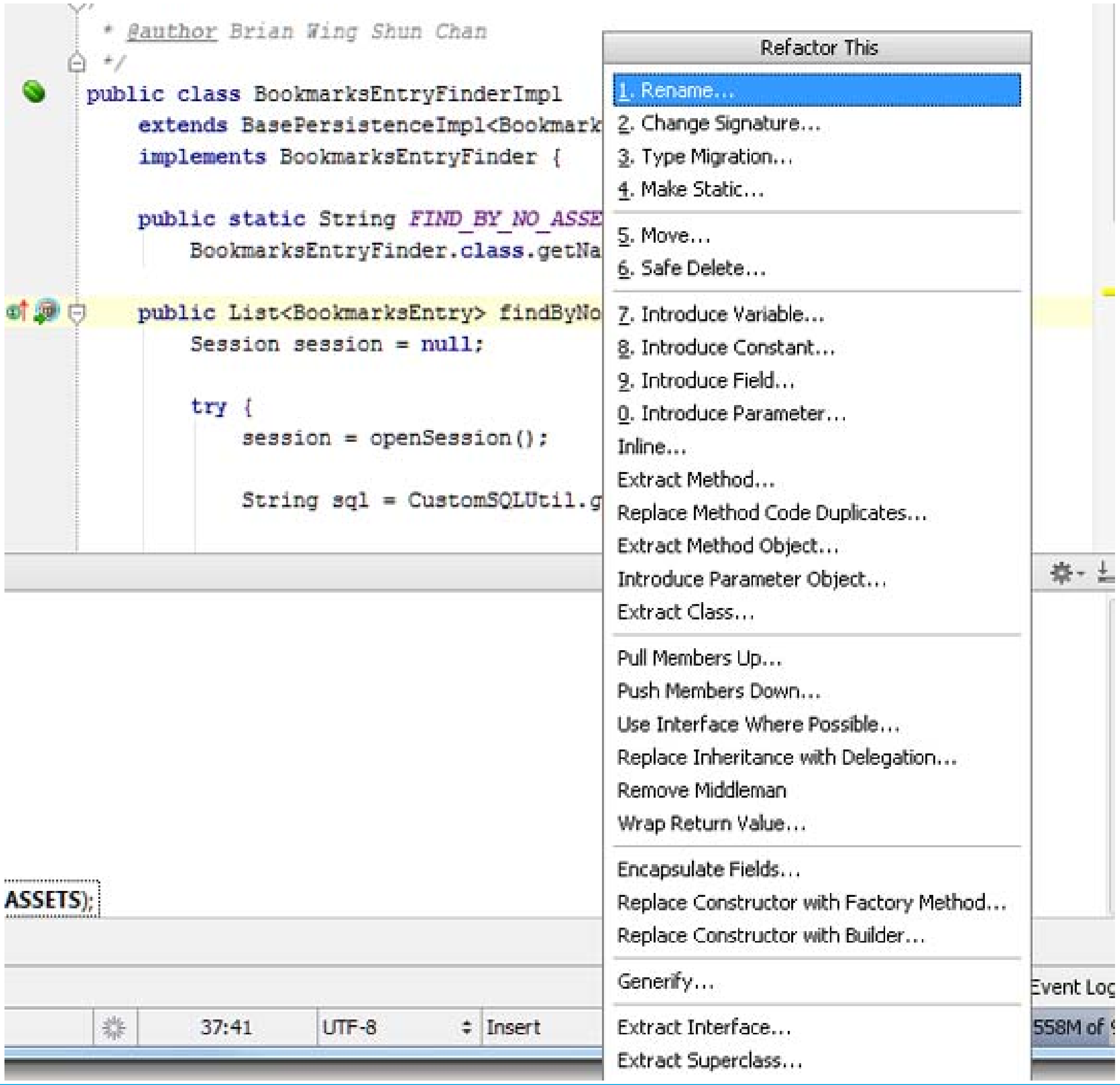


Refactoring Tool Support

- IDE's automated refactoring support
- Java
  - intelliJ IDEA
  - Eclipse's Java Development Toolkit (JDT)
  - NetBeans
- .Net
  - Visual Studio
  - ReShaper (Visual Studio Plug-in)
  - Refactor Pro (Visual Studio Plug-in)
  - Visual Assist (Visual Studio Plug-in)
- DMS Software Reengineering Toolkit
  - Supports large scale refactoring for C, C++. C#, COBOL, Java, PHP and others

Refactoring Tool Support

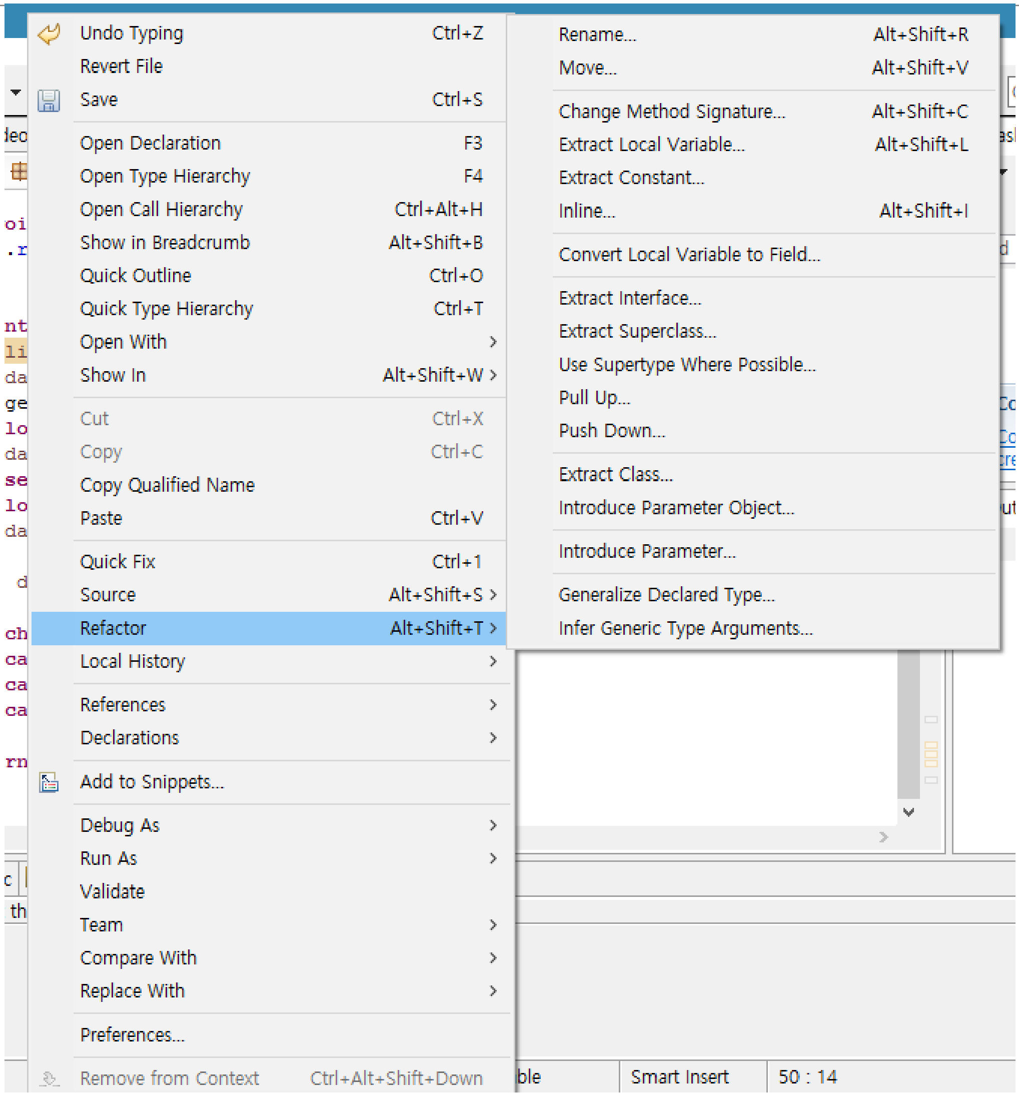
- intelliJ



```
 * @author Brian Wing Shun Chan
 */
public class BookmarksEntryFinderImpl
    extends BasePersistenceImpl<Bookmark
    implements BookmarksEntryFinder {

    public static String FIND_BY_NO_ASSE
        BookmarksEntryFinder.class.getNa

    public List<BookmarksEntry> findByNo
        Session session = null;

        try {
            session = openSession();

            String sql = CustomSQLUtil.g
```

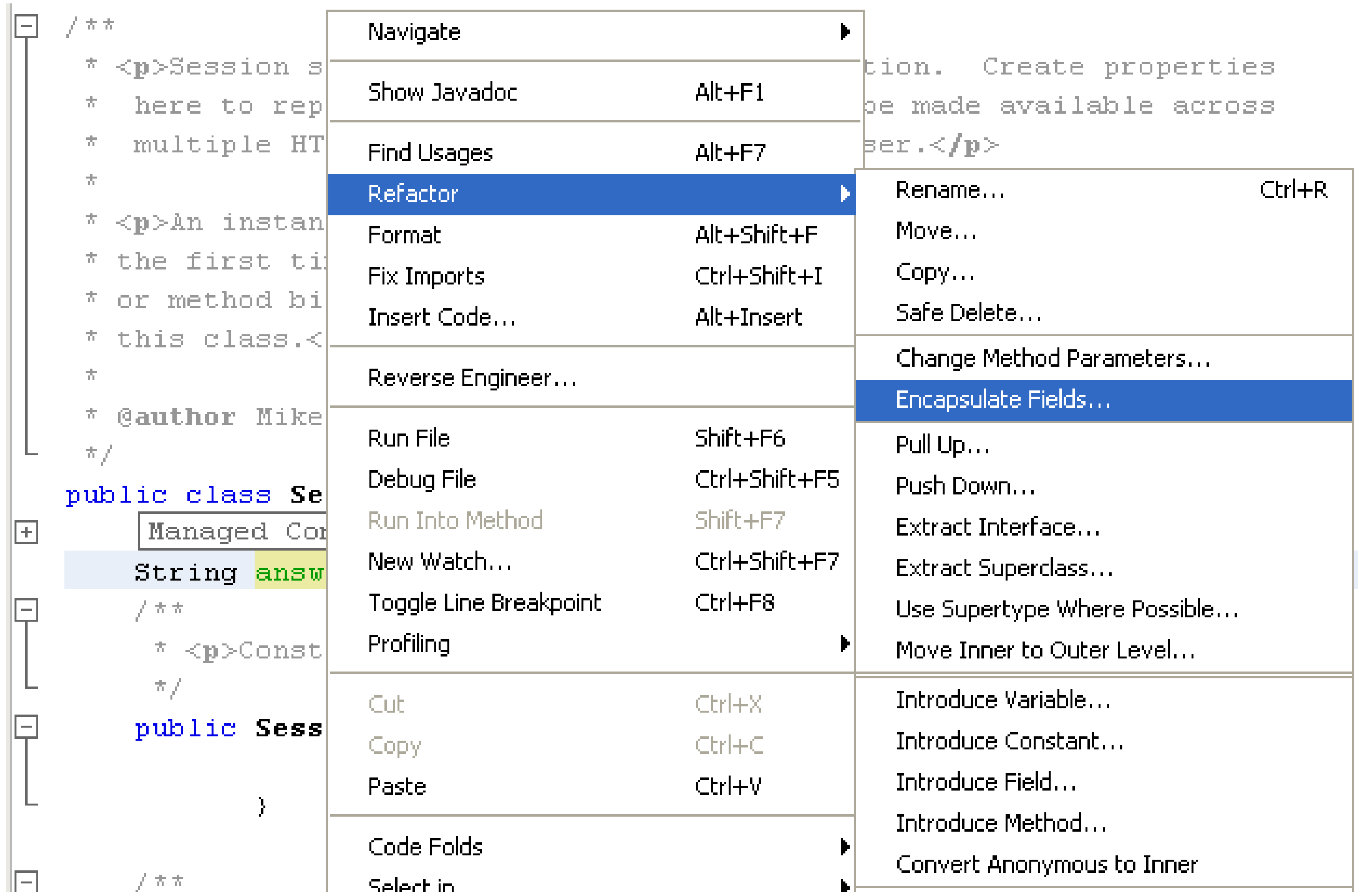
Refactoring Tool Support

- Eclipse



Refactoring Tool Support

- Netbeans



```
/**
 * <p>Session s                                      tion.  Create properties
 *  here to rep                                      be made available across
 *  multiple HT                                      ser.</p>
 *
 * <p>An instan
 * the first ti
 * or method bi
 * this class.<
 *
 * @author Mike
 */
public class Se
    Managed Cor
    String answ
    /**
     * <p>Const
     */
    public Sess
            }

    /**
```

Bad Smells

CODE SMELLS

Bad Smells

- Bloaters
  - **gargantuan** code, methods and classes that are difficult to handle
    - Long Methods
    - Large Class
    - Primitive Obsession
    - Long Parameter List
    - Data Clumps

Bloaters

- **Long Method**
  - method containing too many lines of code
- **Larger Class**
  - class containing many fields / methods / lines of code
- **Primitive Obsession**
  - using primitives instead of small objects for simple tasks (currency, ranges, special string for phone no.)
  - using constants for coding information (`ADMIN = 1`)
  - using string constants as field names for use in data arrays

Bloaters

- **Long Parameter List**
  - more than three or four parameters for a method

- **Data Clumps**
  - different parts of the code containing identical groups of variables - should be turned into their own classes

Bad Smells

- Object-Oriented Abusers
  - **incomplete** or **incorrect** application of OO principles
    - Switch Statement
    - Temporary Field
    - Refused Bequest
    - Alternative Classes With Different Interface

Object-Oriented Abusers

- **Switch Statements**
  - having complex switch operator or sequence of if statements

- **Temporary Field**
  - fields getting their values only under certain circumstances and being empty outside of these circumstances

Object-Oriented Abusers

- **Refused Bequest**
    - subclass using only some of the methods and properties inherited from its parents
    - the unneeded / unwanted methods may simply go unused or be redefined and give off exceptions

- **Alternative Classes with Different Interfaces**
    - two classes performing identical functions but having different method names

Bad Smells

- **Change Preventers**
  - **change** is one place **resulting** in many **changes in other** places
    - Divergent Change
    - Shotgun Surgery
    - Parallel Inheritance Hierarchies

Change Preventers

- **Divergent Change**
  - changing a class causes having to change many unrelated methods
- **Shotgun Surgery**
  - making any modifications requires making many small changes to many different classes
- **Parallel Inheritance Hierarchies**
  - creating a subclass for a class causes a need to create a subclass for another class

Bad Smells

- Dispensables
  - something **pointless** and unnecessary whose absence would make code cleaner
    - Comments
    - Duplicate Code
    - Lazy Class
    - Data Class
    - Dead Code
    - Speculative Generality

Dispensables

- **Comments**
  - method filled with explanatory comments

- **Duplicate Code**
  - two code fragments looking almost identical

- **Lazy Class**
  - understanding and maintaining classes always costs time and money
  - class which doesn't do enough to earn attention should be deleted

Dispensables

- **Data Class**
  - class containing only fields and crude methods for accessing them (getters and setters)
  - these are simply containers for data used by other classes, not containing any additional functionality and cannot independently operate on the owned data
- **Dead Code**
  - no longer used variable, parameter, field, method or class (obsolete).
- **Speculative Generality**
  - having an unused class, method, field or parameter

Bad Smells

- Couplers
  - contributing to **excessive coupling** between classes
    - Feature Envy
    - Inappropriate Intimacy
    - Message Chains
    - Middle Man

Couplers

- **Feature Envy**
  - method accessing the data of another object more than its own data
- **Inappropriate Intimacy**
  - one class using the internal fields and methods of another class

Couplers

- **Message Chains**
  - series of calls resembling $a->b()->c()->d()

- **Middle Man**
  - class performing only one action which is delegating work to another class
  - why does this class exist at all?

Other Smell

- **Incomplete Library Class**
  - provided methods from library is incomplete

Refactoring Techniques

Refactoring Techniques

- Composing Methods
  - Extract Method
  - Inline Method
  - Extract Variable
  - Inline Temp
  - Replace Temp with Query
  - Split Temporary Variable
  - Remove Assignments To Parameters
  - Replace Method with Method Object
  - Substitute Algorithm

Extract Method

- look for codes that can be grouped together
- extract them into a separate method with a meaningful name representing what the purpose of those code was

```
void printOwing(double amount) {
    printBanner();

    // print details
    System.out.println("name:" + name);
    System.out.println("amount" + amount);
}
```

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}
void printDetails(double amount){
    System.out.println("name:" + name);
    System.out.println("amount" + amount);
}
```

Inline Method

- inverse of Extract Method
- method body is more obvious than the method itself
- replace calls to the method with the method's content and delete the method itself

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
}
```

↓

```
int getRating() {
    return (numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Extract Variable

- expression that is hard to understand (complicated)
- put the result of expression or its parts in separate self-explanatory variables

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
     (browser.toUpperCase().indexOf("IE") > -1) &&
      wasInitialized() && resize > 0 ) {
    // do something
}
```

```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
    // do something
}
```

Inline Temp

- temporary variable assigned with the result of a simple expression and nothing more (get in way of refactoring)
- replace all references to the temporary variable with the expression itself

```
double basePrice = anOrder.basePrice();
   return (basePrice > 1000)
```

```
return (anOrder.BasePrice() > 1000)
```

Replace Temp with Query

- temporary variable are used to hold the result of an expression in a local variable for later use
- extract expression into a method that return the result and query the method instead of using a variable
- new method can be used in other methods, if necessary

```
double basePrice = quantity * itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice;
```

```
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;
...
double basePrice() {
    return quantity * itemPrice;
}
```

Split Temporary Variable

- temporary variable used to store various intermediate values inside a method (except for cycle variables)
- different variable for different values where each variable is responsible for only one particular thing

```
double temp = 2 * (height + width);
System.out.println(temp);
temp = height * width;
System.out.println(temp);
```

```
final double perimeter = 2 * (height + width);
System.out.println(perimeter);
final double area = height * width;
System.out.println(area);
```

Remove Assignments to Parameters

- value is assigned to a parameter inside methods' body
- temporary variable instead of parameter

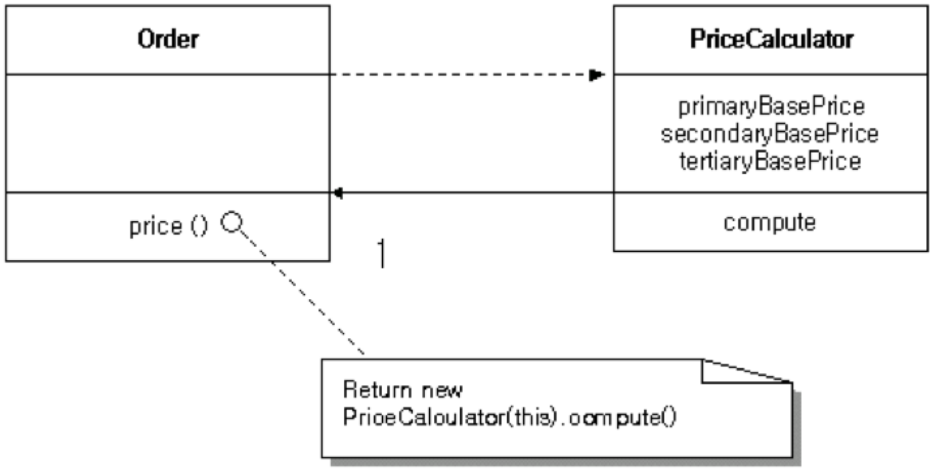
```
int descount (int inputVal, int Quantity, int yearToDate) {
    if (inputVal > 50)
        inputVal -= 2;
}
```

```
int descount ( int inputVal, int Quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50)
        result -= 2;
}
```

Replace Method with Method Object

- long method where local variables are so intertwined preventing Extract Method technique cannot be applied
- replace method into a separate class where local variable becomes fields of the class → split method into several methods within the same class

```
class Order ...
double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double teriaryBasePrice;
    // long computation;
    ...
}
...
```



Replace Method with Method Object

```
class Order ...
    public double price() {
        return new PriceCalculator(this).compute();
    }
}

class PriceCalculator {
    private double primaryBasePrice;
    private double secondaryBasePrice;
    private double tertiaryBasePrice;

    public PriceCalculator(Order order) {
        // copy relevant information from order object.
        ...
    }

    public double compute() {
        // long computation.
        ...
    }
}
```

Substituted Algorithm

- wanting to replace existing algorithm with a clearer one
- replace body of the method implementing the algorithm with a new one

```
String foundPerson(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            return "Don";
        }
        if (people[i].equals("John")) {
            return "John";
        }
        if (people[i].equals("Kent")) {
            return "Kent";
        }
    }
    return "";
}
```

Substituted Algorithm

```
String foundPerson(String[] people) {
    ListCandidates = Arrays.asList(new String[] {"Don", John", "Kent"});
    for (int i = 0; i < people.length; i++) {
        if(candidates.contains(people[i])) {
            return people[i];
        }
    }
    return "";
}
```

Refactoring Exercises

- Composing Methods
  - Extract Method
  - Replace Temp with Query
  - Replace Method with Method Object

Extract Method

◦ method to print customer owing amount

```
Public void printOwing() {
    Enumeration elements = orders.elements();
    double outstanding = 0.0;
    // print banner
    System.out.println("**************************");
    System.out.println("***** Customer Owes ******");
    System.out.println("**************************");
    // calculate outstanding
    while (elements.hasMoreElements()) {
        Order each = (Order) elements.nextElement();
        outstanding += each.getAmount();
    }
    //print details
    System.out.println("name:" + name);
    System.out.println("amount" + outstanding);
}
```

Extract Method

- extract method that prints banner

Extract Method

- extract method that prints details

Extract Method

- extract method that calculate outstanding

Extract Method

Replace Temp with Query

◦ method to calculate and return price

```
Public double getPrice() {
    int basePrice = quantity * itemPrice;
    double discountFactor;

    if (basePrice > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;

    return basePrice * discountFactor;
}
```

Replace Temp with Query

- extract expression that calculates basePrice into method and replace local variable with query

Replace Temp with Query

- extract expression that calculates discountFactor into method and replace local variable with query

Replace Method with Method Object

- complex code with no special meaning (just for demo)
- gamma is a method with sophisticated calculations and entangled local variables
- extract method of underlined code segment from gamma

```
class Account {
  …
  int gamma(int inputVal, int quantity, int yearToDate){
    int importantValue1 = (inputVal * quantity) + delta();
    int importantValue2 = (inputVal * yearToDate) + 100;
    if ((yearToDate - importantValue1) > 100) {
      importantValue2 -= 20;
    }
    importantValue3 = importantValue2 * 7;
    …
    return importantValue3 - 2 * importantValue1;
  }
}
```

Replace Method with Method Object

- replace gamma method to a separate class
- store local variables / parameters as a fields of the class

Replace Method with Method Object

- create constructor that accepts methods parameters and stores them in class fields

Replace Method with Method Object

- move original method into the class appropriately

Replace Method with Method Object

- replace body of original method with a call to a method in a new class

Replace Method with Method Object

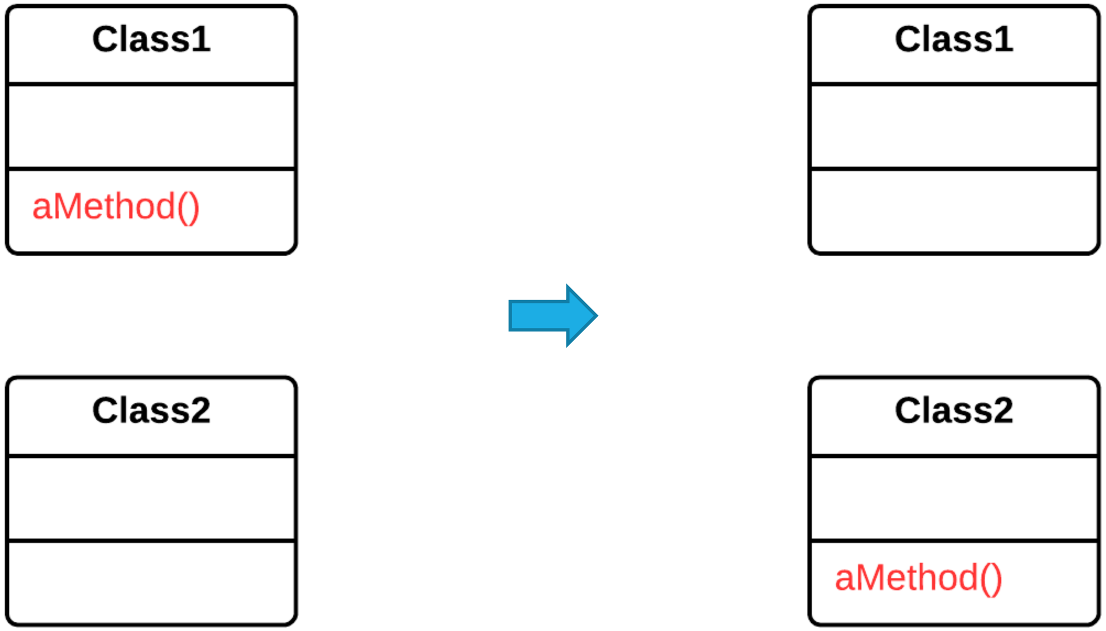
- now extract method

Refactoring Techniques

- Moving Feature between Objects
  - Move Method
  - Move Field
  - Extract Class
  - Inline Class
  - Hide Delegate
  - Remove Middle Man
  - Introduce Foreign Method
  - Introduce Local Extension

Move Method

- method using / used by more features in another class than the class on which it is defined
- create a new method in the class that uses the method the most then turn old method into delegation or remove it entirely



Move Method

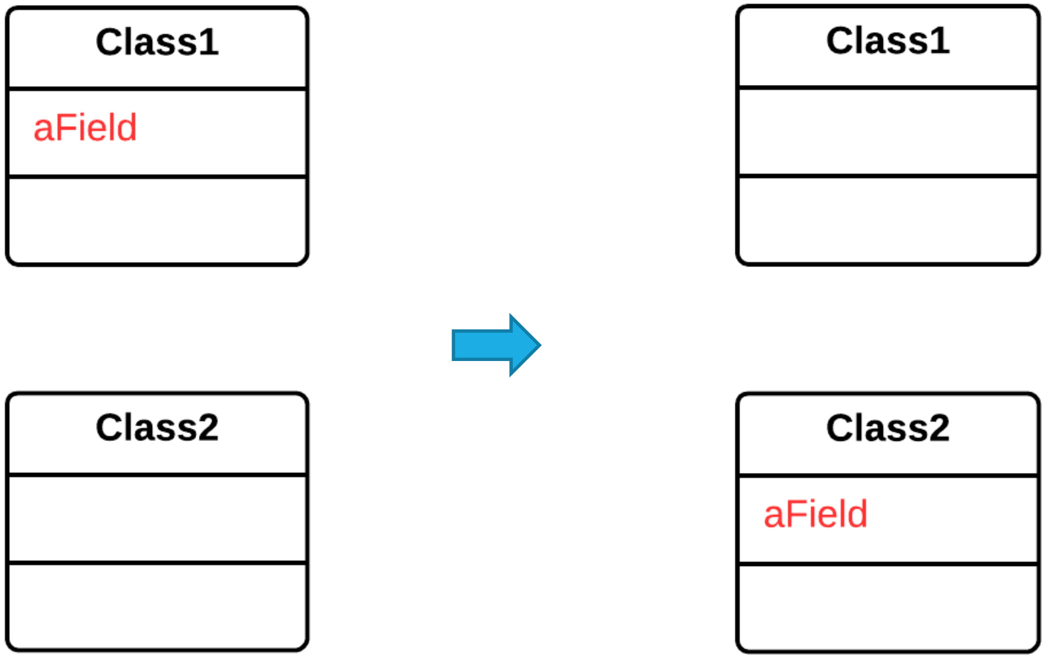
```
class Project {
    Person[] participants;
}

class Person {
    int id;
    boolean participate(Project p) {
        for(int i=0; i<p.participants.length; i++) {
            if (p.participants[i].id == id)
                return(true);
        }
        return(false);
    }
}

... if (x.participate(p)) ...
```

Move Method

```
class Project {
    Person[] participants;
    boolean participate(Person x) {
        for(int i=0; i<participants.length; i++) {
            if (participants[i].id == x.id)
                return(true);
        }
        return(false);
    }
}

class Person {
    int id;
}

... if (p.participate(x)) ...
```

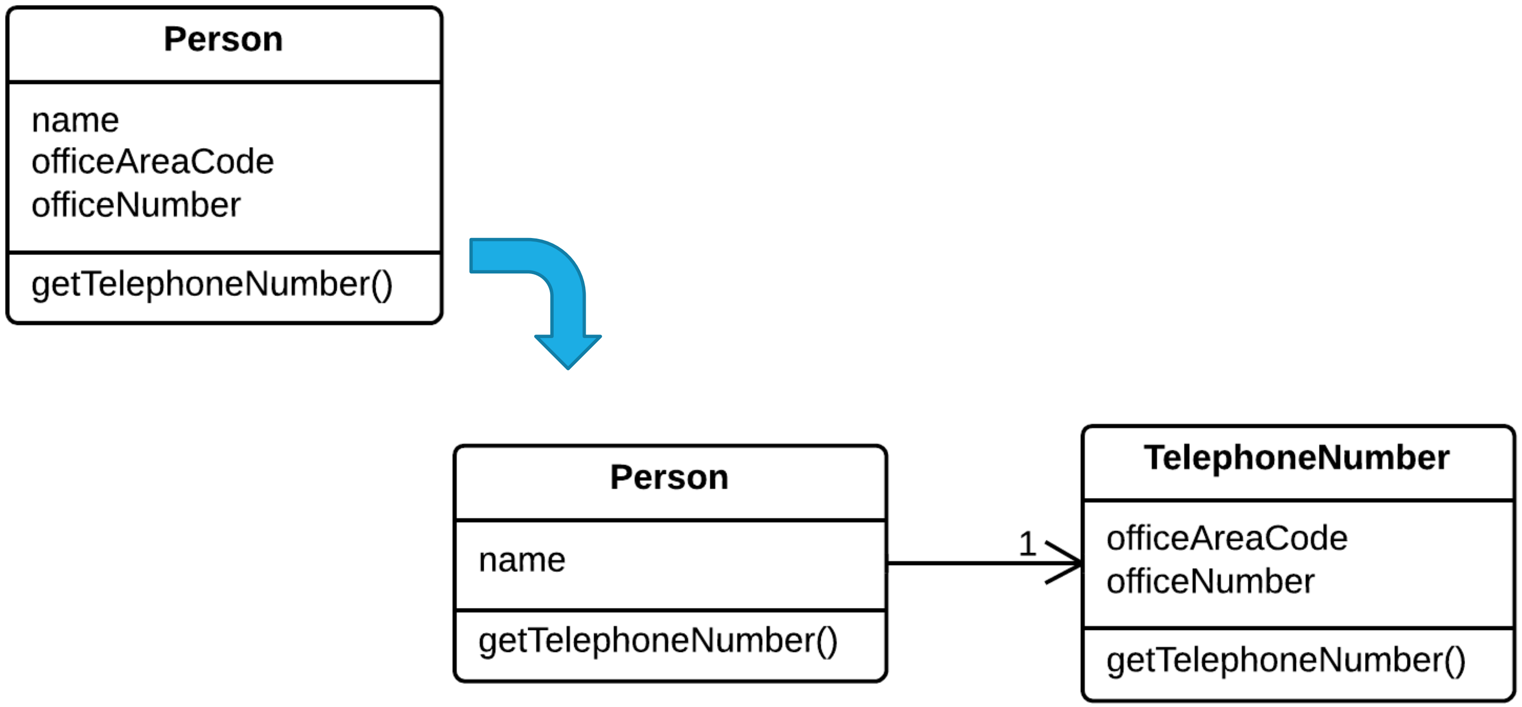
Move Field

- field used by another class more than the class on which it is defined
- create a new field in the target class and redirect all users of the old field to it



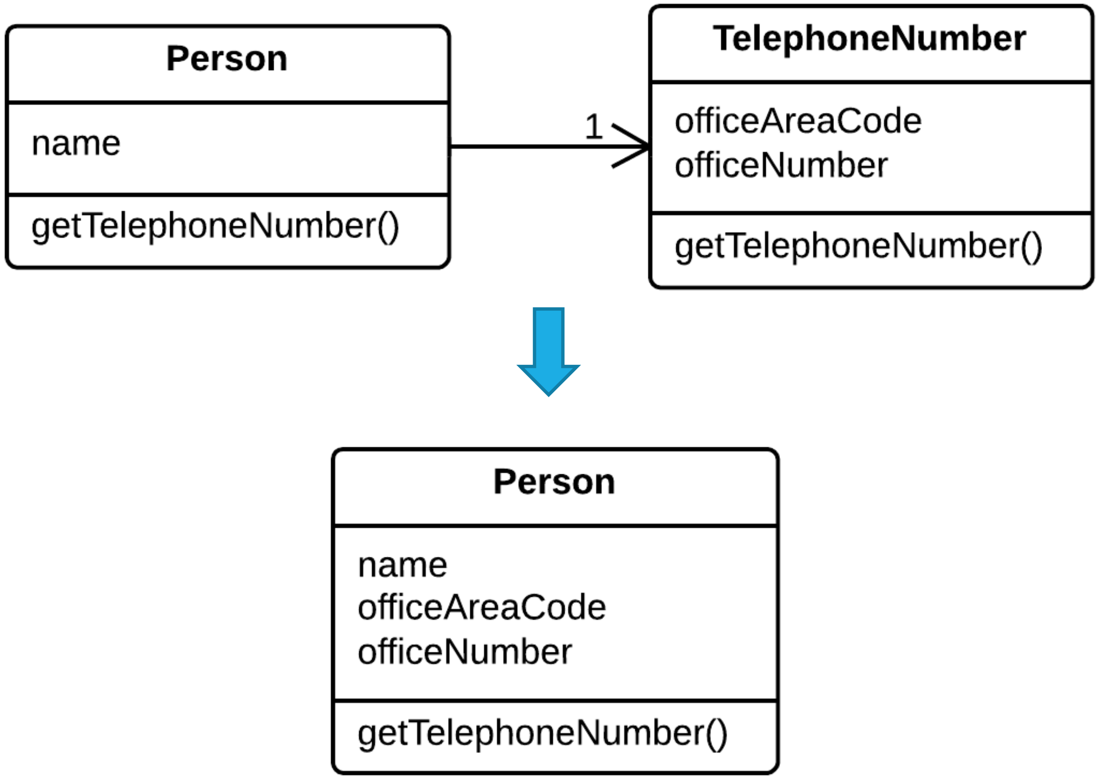
Extract Class

- when one class does the work of two
- create a new class and place the fields and methods responsible for the relevant functionality in it



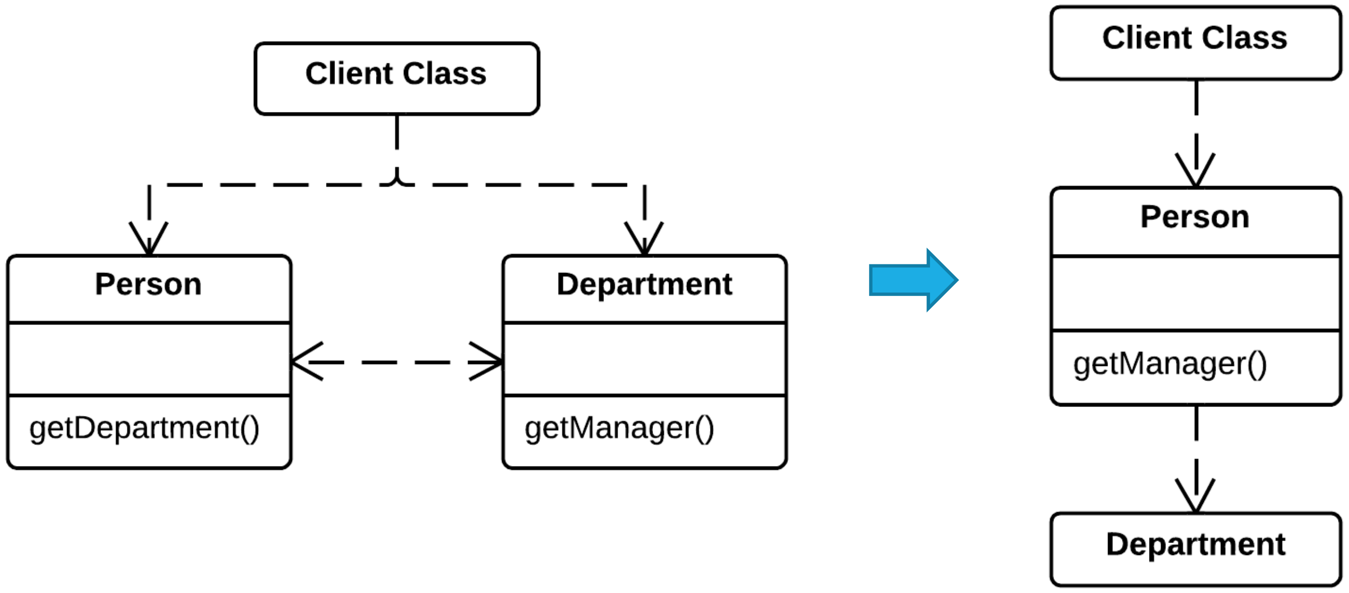
Inline Class

- class does almost nothing and it not responsible for anything and no additional responsibility is planned
- move all features from the class to another



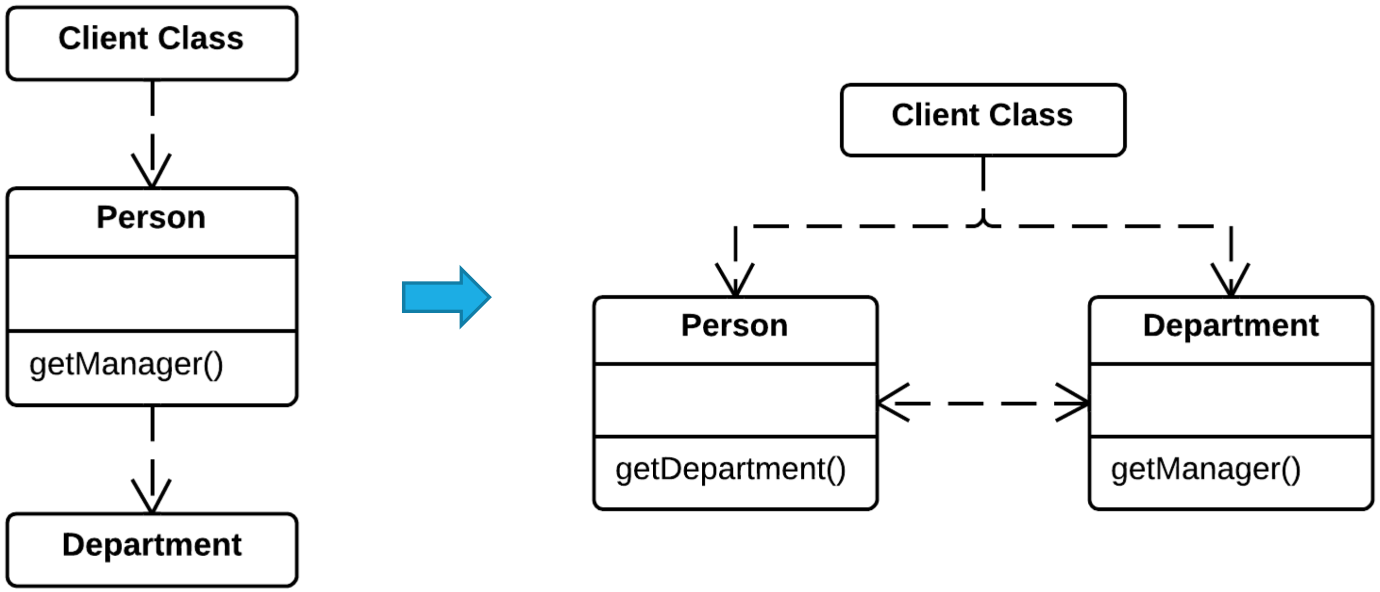
Hide Delegate

- client gets object B from a field or method of object A and calls a method of object B
- create a new method in a class A that delegates the call to object B (client does not know / depend on class B)



Remove Middle Man

- class has too many methods that simply delegate to other objects
- delete these methods and force the client to call the end methods directly



Introduce Foreign Method

- server class does not contain the needed method and the method cannot be added to the class
- add a method in the client class and pass an instance of the server class as an argument

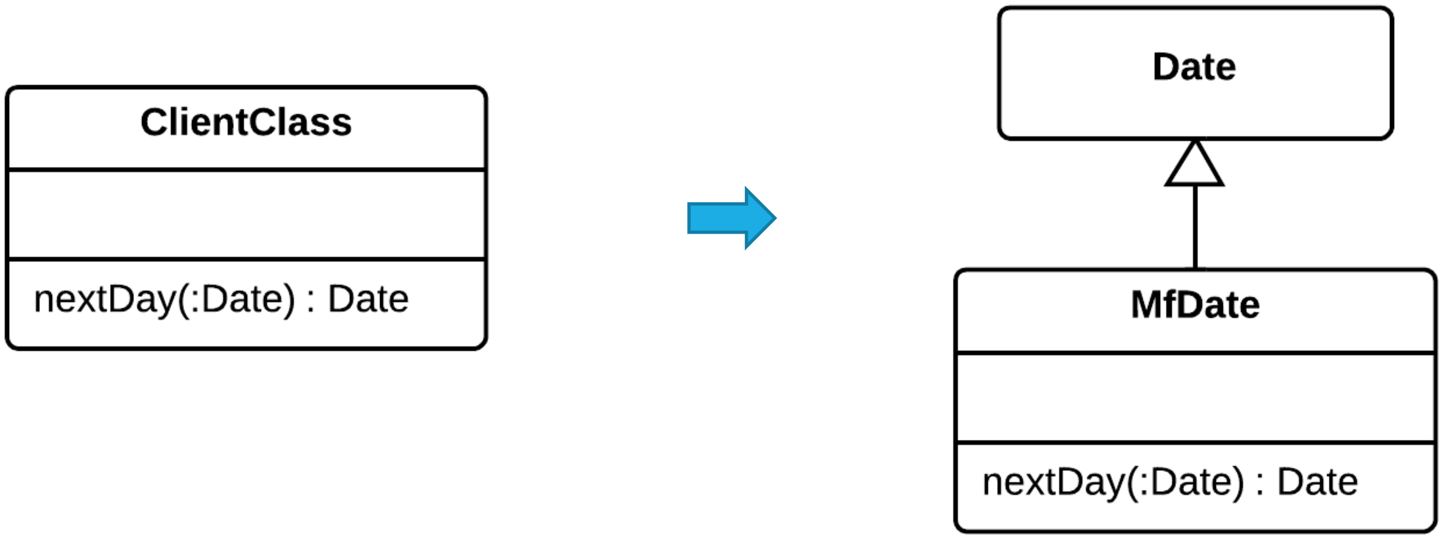
```
Date newStart = new Date(previousEnd.getYear(),
previousEnd.getMonth(), previousEnd.getDate() + 1);
```

⇩

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
    return new Date(arg.getYear(), arg.getMonth(),
    arg.getDate() + 1);
}
```

Introduce Local Extension

- server class does not contain some methods that you need and the methods cannot be added to the class
- create a new class containing the methods and make it either the subclass or wrapper of the original



Refactoring Exercises

- Moving Feature between Objects
  - Move Method
  - Extract Class
  - Hide Delegate

Move Method

- moving `overdraftCharge()` to AccountType class

```
public class Account...
    private AccountType type;
    private int daysOverdrawn;

    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;
    }
…
```

Move Method

Extract Class

- simple Person class doing work of two

```
public class Person...
  private String name;
  private String officeAreaCode;
  private String officeNumber;

  public String getName() { return name; }

  public String getTelephoneNumber() {
    return ("(" + officeAreaCode + ") " + officeNumber);
  }

  String getOfficeAreaCode() { return officeAreaCode; }

  void setOfficeAreaCode(String arg) { officeAreaCode = arg; }

  String getOfficeNumber() { return officeNumber; }

  void setOfficeNumber(String arg) { officeNumber = arg; }
```

Extract Class

Extract Class

Hide Delegate

- simple person and department

```
class Person {
    Department department;
    public Department getDepartment() { return department; }
    public void setDepartment(Department arg) {
        department = arg;
    }
}

class Department {
    private String chargeCode;
    private Person manager;
    public Department (Person arg) { manager = arg; }
    public Person getManager() {
        return manager;
    }
...


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

Hide Delegate

Refactoring Techniques

- Organizing Data
  - Self Encapsulate Field
  - Replace Data Value with Object
  - Change Value to Reference
  - Change Reference to Value
  - Replace Array with Object
  - Duplicate Observed Data
  - Change Unidirectional Association to Bidirectional
  - Change Bidirectional Association to Unidirectional
  - Replace Magic Number with Symbolic Constant
  - Encapsulate Field
  - Encapsulate Collection
  - Replace Type Code with Class
  - Replace Type Code with Subclasses
  - Replace Type Code with State/Strategy
  - Replace Subclass with Fields

Self Encapsulate Field

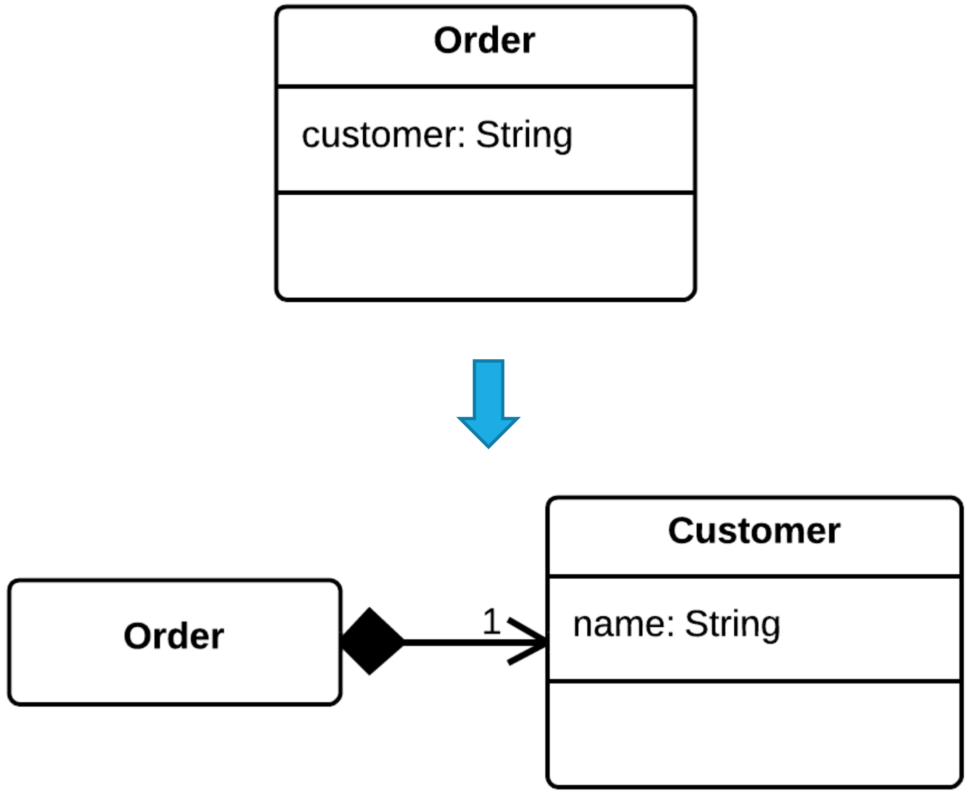
- direct access to a field inside a class but the coupling to the filed is becoming awkward
- create a getter and setter for the field, and use only them for accessing the field

```
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;
}
```

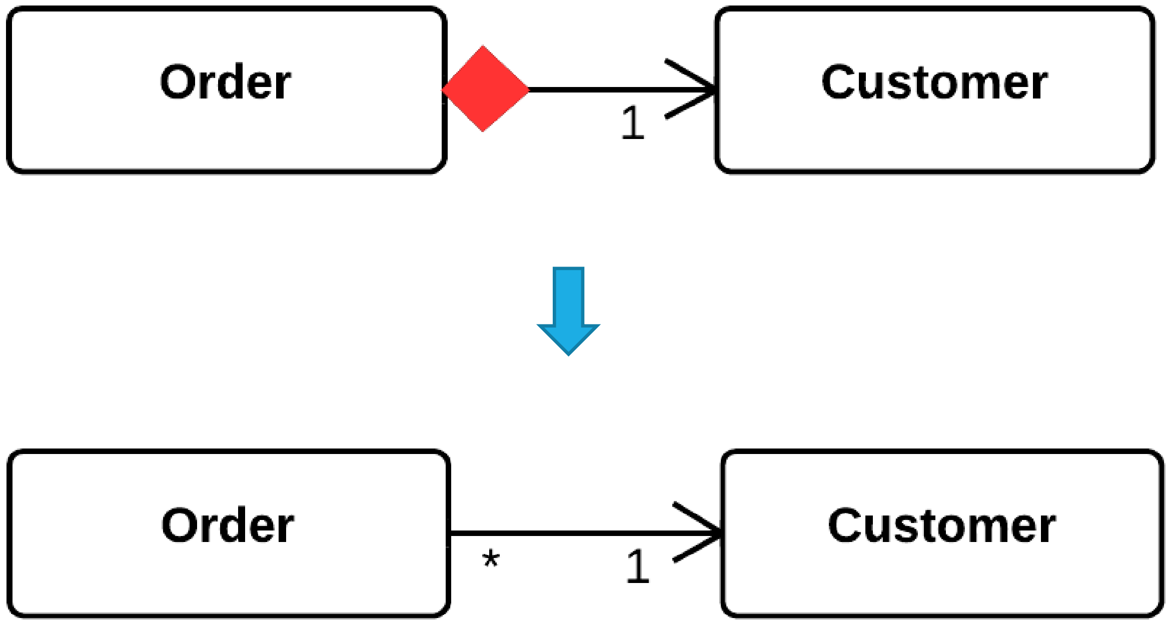
Replace Data Value with Object

- class or classes contain a data field with its own behaviour and associated data
- change the data item into an object



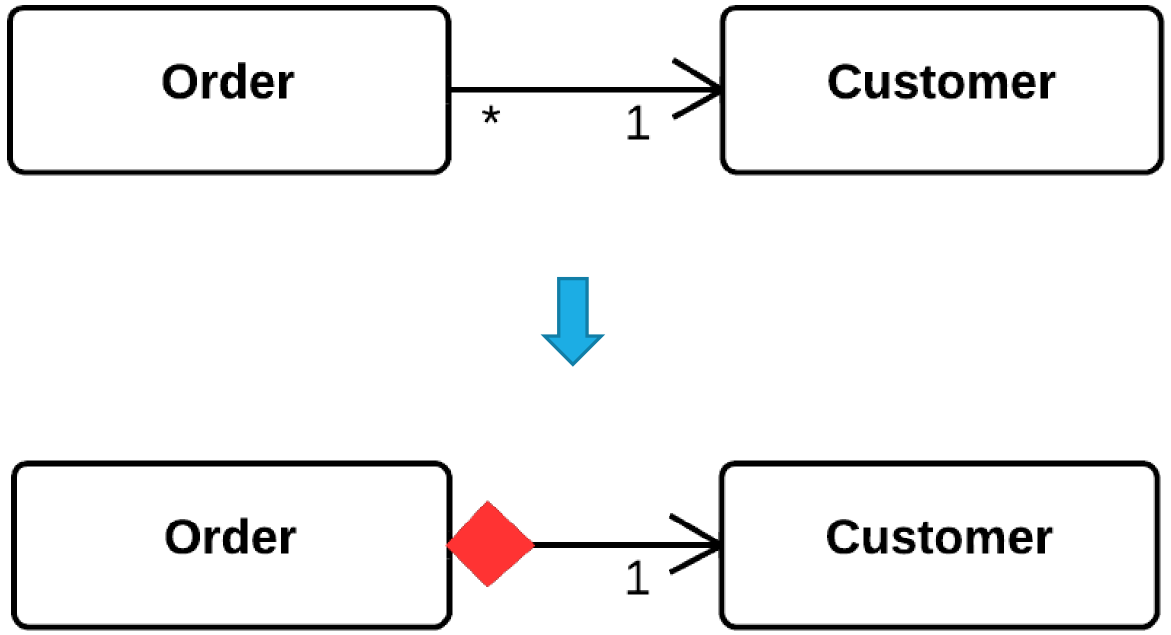
Replace Value to Reference

- many identical instances of a single class that needs to be replaced with a single object
- convert the identical objects to a single reference object



Change Reference to Value

- reference object is too small and infrequently changed to justify managing its lifecycle
- convert it into a value object



Replace Array with Object

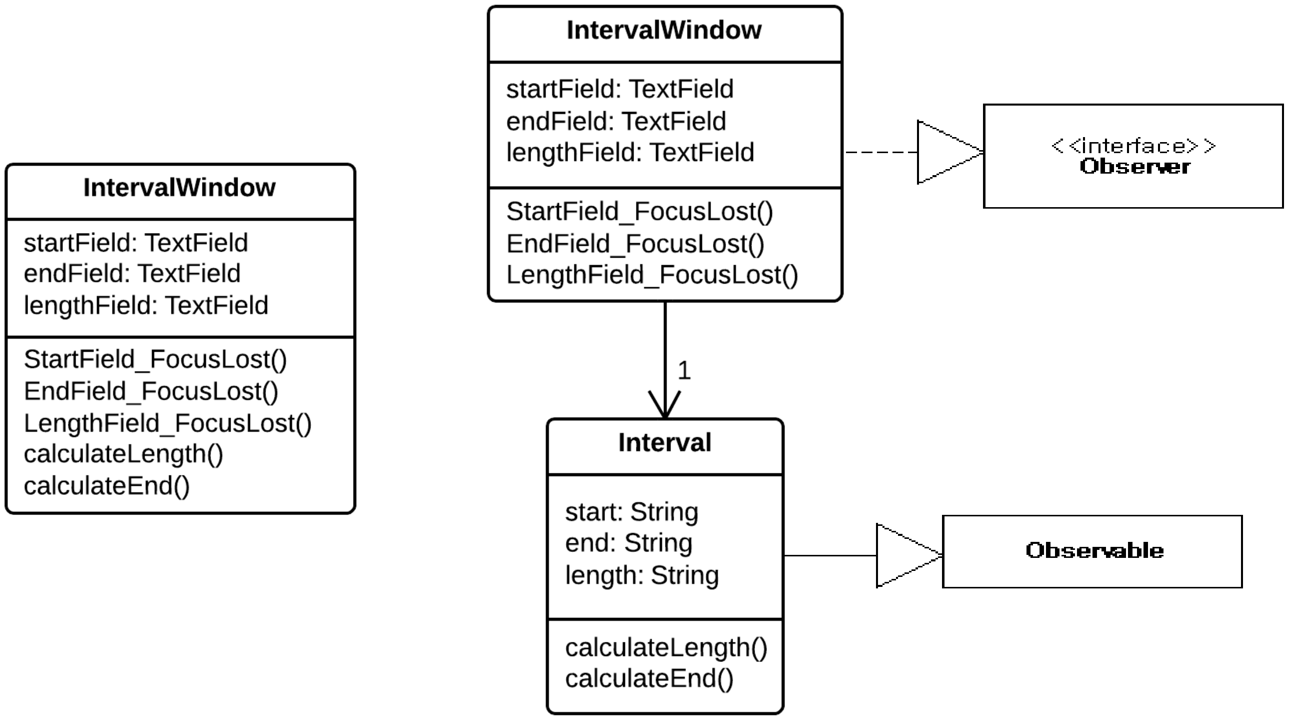
- array that contains various types of data
- replace array with an object with separate field for each element

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

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

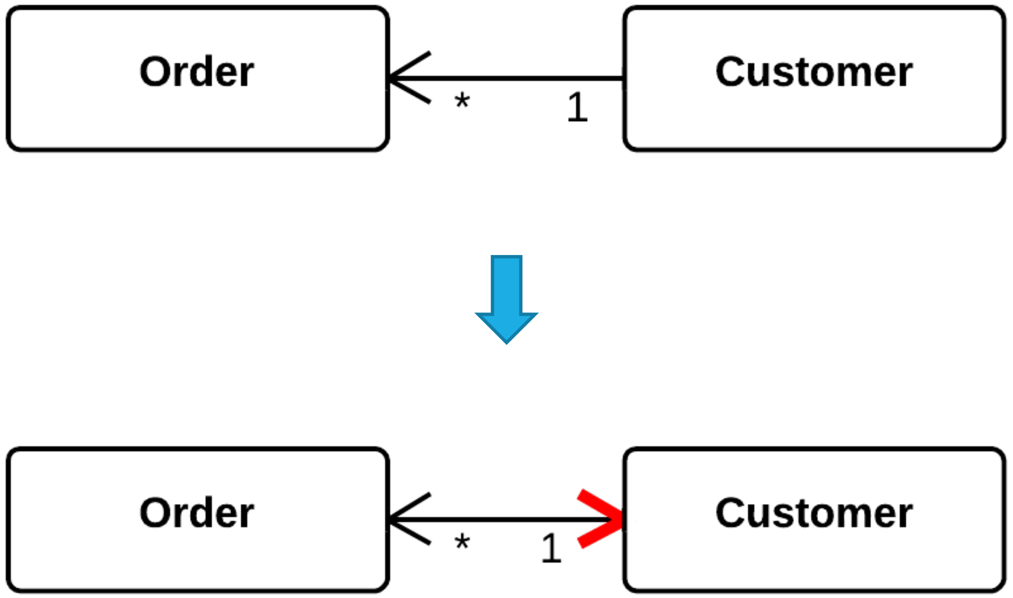
Duplicate Observed Data

- domain data available only in a GUI control and domain methods need access
- copy data to a domain object and set up an observer to synchronize the two pieces of data



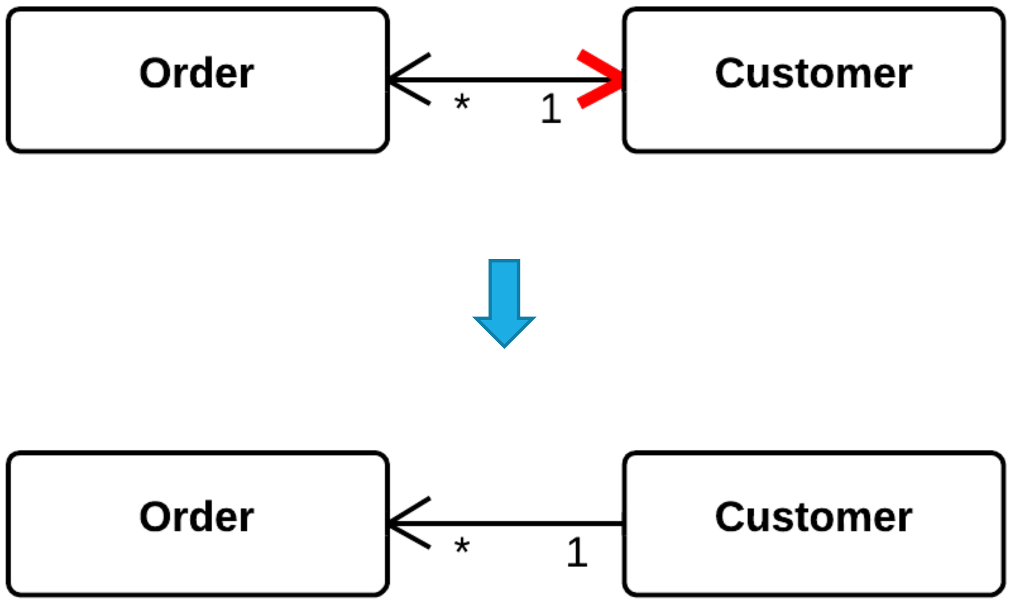
Change Unidirectional Association to Bidirectional

- two classes that each need to use the feature of the other, but the association between them is only unidirectional
- add the missing association to the class that needs it and change modifier to update both set



Change Bidirectional Association to Unidirectional

- bidirectional association where one class does not use the other's features
- remove unused association



Replace Magic Number with Symbolic Constant

- code using number that has a certain meaning
- replace this number with constant that well represents the meaning of the number

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

```
double potentialEnergy(double mass, double height) {
    return mass * GRAVITATION_CONSTNAT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9,81;
```

Encapsulate Field

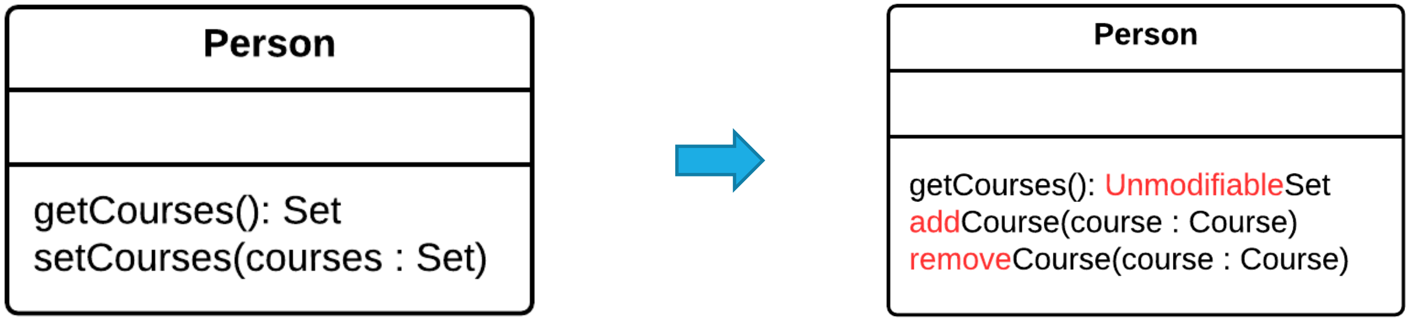
- there is a public field
- make it private and provide accessors

```
public String name;
```

```
private String name;

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

Encapsulate Collection

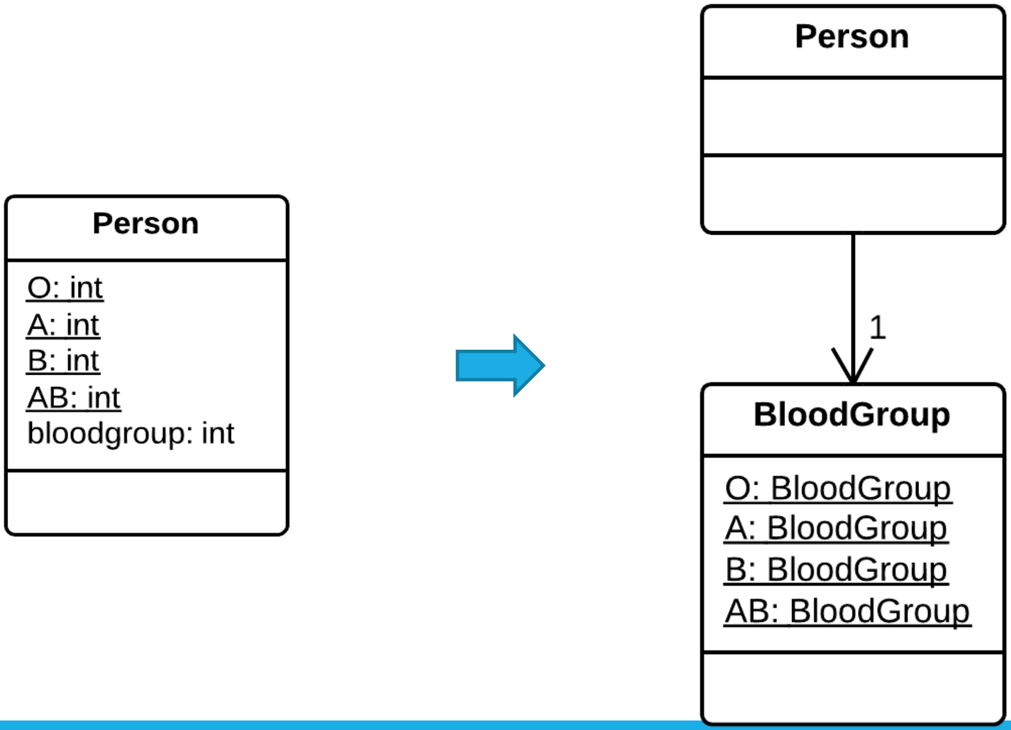
- class contains a collection field and a simple getter / setter for working with the collection
- make the getter return read-only value and create methods for adding and deleting elements of the collection



Person
getCourses(): Set
setCourses(courses : Set)
Person
getCourses(): UnmodifiableSet
addCourse(course : Course)
removeCourse(course : Course)

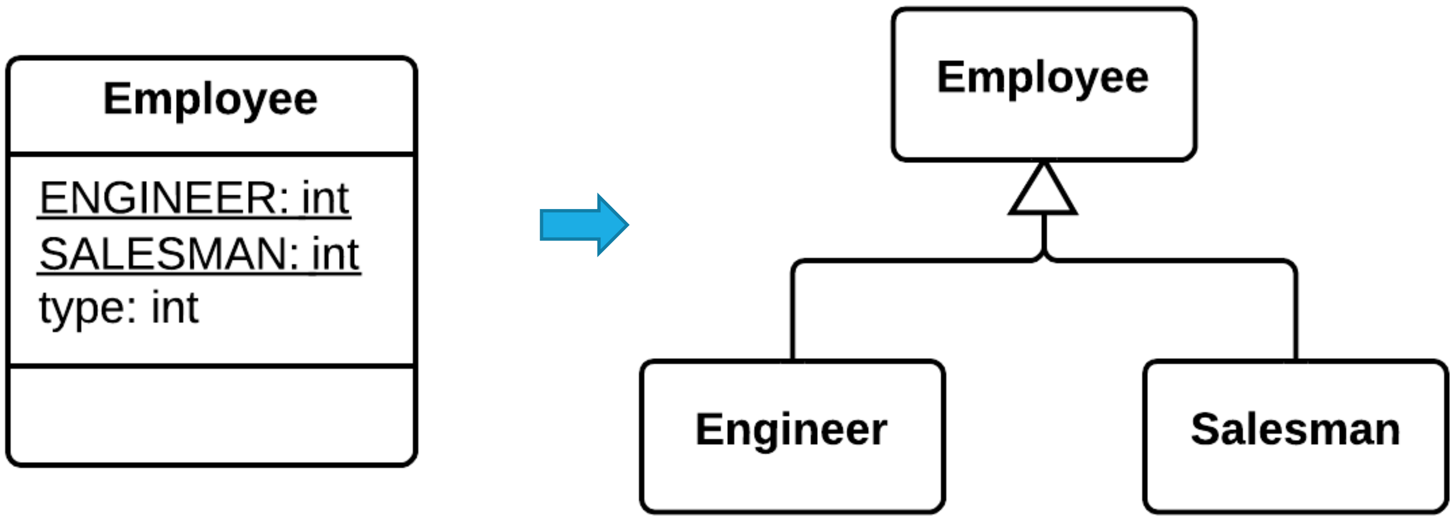
Replace Type Code with Class

- class has a field that contains type code where values of these type are not used in operator conditions and do not affect the behaviour of the program
- create a new class and use its objects instead of the type code values



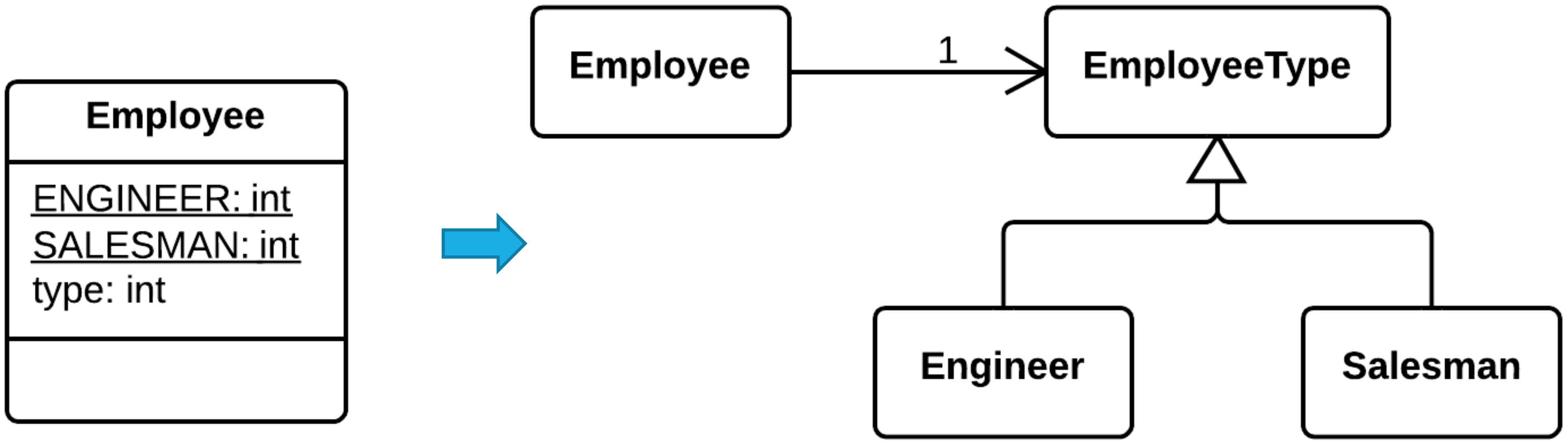
Replace Type Code with Subclasses

- immutable type code affecting the behaviour of a class
- create subclasses for each value of the coded type



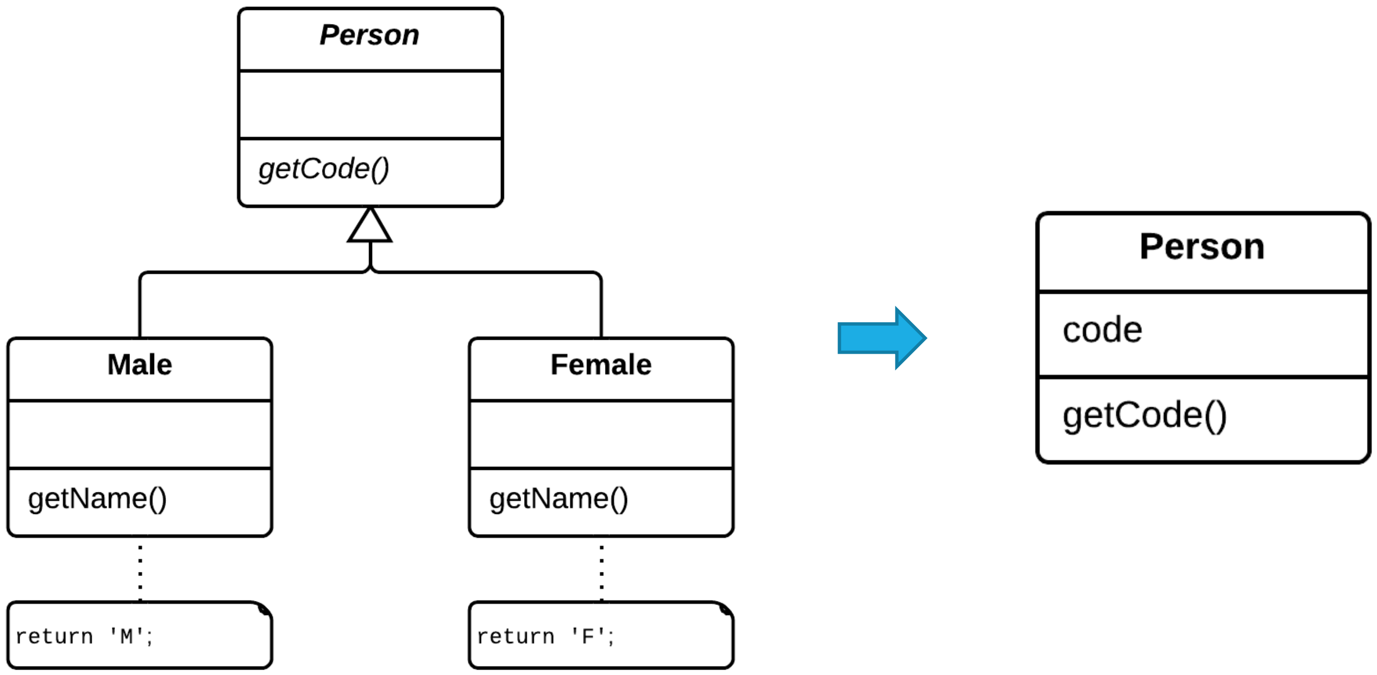
Replace Type code with State/Strategy

- type code that affects the behaviour of a class, but cannot use subclassing
- replace type code with a state object



Replace Subclass with Fields

- subclass differing only in their (constant-returning) methods
- replace the methods with fields in the superclass and delete the subclasses



Refactoring Exercises

- Organizing Data
  - Replace Data Value with Object
  - Change Value to Reference
  - Encapsulate Collection
  - Replace Type Code with Class
  - Replace Type Code with Subclasses

Replace Data Value with Object

- Order class with customer of order as a string
- replace customer data value with customer object

```
public class Order...
    private String customer;

    public Order (String customer) {
        customer = customer;
    }

    public String getCustomer() {
        return customer;
    }

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

Replace Data Value with Object

Change Value to Reference

- customer in order class being used as a data value

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

class Order {
  …
  private Customer customer;
  public String getCustomerName() {
    return customer.getName();
  }
  public void setCustomer(String customerName) {
    customer = new Customer(customerName);
  }
  public Order (String customerName) {
    customer = new Customer(customerName);
  }
  …
}
```

Change Value to Reference

- customer in order class being used as a data value

```
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;
  }
}
```

Change Value to Reference

Change Value to Reference

Encapsulate Collection

- person is taking a course

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

class Person...
  private Set courses;

  public Set getCourses() {
    return courses;
  }
  public void setCourses(Set arg) {
    courses = arg;
  }
```

Encapsulate Collection

- Kent add courses and retrieves all his advanced courses

```
Person kent = new Person();
Set s = new HashSet();
s.add(new Course ("Smalltalk Programming", false));
s.add(new Course ("Appreciating Single Malts", true));
kent.setCourses(s);
Assert.equals (2, kent.getCourses().size());
Course refact = new Course ("Refactoring", true);
kent.getCourses().add(refact);
kent.getCourses().add(new Course ("Brutal Sarcasm", false));
Assert.equals (4, kent.getCourses().size());
kent.getCourses().remove(refact);
Assert.equals (3, kent.getCourses().size());

Iterator iter = person.getCourses().iterator();
int count = 0;
while (iter.hasNext()) {
    Course each = (Course) iter.next();
    if (each.isAdvanced()) count ++;
}
System.out.print("Advanced courses: " + count);
```

Encapsulate Collection

Encapsulate Collection

- Kent add courses and retrieves all his advanced courses

Replace Type Code with Class

- Person class blood group modeled with a type code

```
public 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) {
        this.bloodGroup = bloodGroup;
    }
    public void setBloodGroup(int arg) {
        bloodGroup = arg;
    }
    public int getBloodGroup() {
        return bloodGroup;
    }
}
```

Replace Type Code with Class

- create new Bloodgroup class

Replace Type Code with Class

- update Person class to use created BloodGroup class

Replace Type Code with Subclasses

- salary calculation for different types of employee

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

  public int type;

  public Employee(int arg) { type = arg; }

  public int monthlySalary;
  public int commission;
  public int bonus;
  public int payAmount() {
    switch (type) {
      case ENGINEER: return monthlySalary;
      case SALESMAN: return monthlySalary + commission;
      case MANAGER: return monthlySalary + bonus;
      default: throw new RuntimeException("Incorrect Code");
    }
  }
}
```

Replace Type Code with Subclasses

- encapsulate field (type)
- replace constructor with factory method

Replace Type Code with Subclasses

- convert ENGINEER into a subclass

Replace Type Code with Subclasses

- convert all type code into a subclass & make class abstract

Replace Type Code with Subclasses

- push down methods and fields into appropriate subclass

Replace Type Code with Subclasses

- push down methods and fields into appropriate subclass

Refactoring Techniques

- Simplifying Conditional Expression
  - Decompose Conditional
  - Consolidate Conditional Expression
  - Consolidate Duplicate Conditional Fragments
  - Remove Control Flag
  - Replace Nested Conditional with Guard Clauses
  - Replace Conditional With Polymorphism
  - Introduce Null Object
  - Introduce Assertion

Decompose Conditional

- complex conditional (if-then-else)
- decompose conditional part into separate methods, the condition, then part, and else parts

```
if (data.before(SUMMER_START) ||
     data.after(SUMMER_END))
    charge = quantity * winterRate +  winterServeceCharge;
else
    charge = quantity * summerRate;
```

```
if (notSummer(date))
    charge = winterCharge(quantity);
else
    charge = summerCharge(quatity);
```

Consolidate Conditional Expression

- multiple conditionals that lead to the same results or action
- consolidate all these conditionals in a single expression

```
double disabilityAmount() {
    if (seniority < 2)
        return 0;
    if (monthsDisabled > 12)
        return 0;
    if (isPartTime)
        return 0;
    // compute the disability amount
```

```
double disabilityAmount() {
    if(isNotEligableForDisability())
        return 0;
    // compute the disability amount;
```

Consolidate Duplicate Conditional Fragments

- identical code in all branches of a conditional
- move the code outside of the conditional

```
if (isSpecialDeal()){
    total = price * 0.95;
    send();
}
else {
    total = price * 0.98;
    send();
}
```

```
if (isSpecialDeal())
    total = price * 0.95
else
    total = price * 0.98;
send();
```

Replace Nested Conditional with Guard Clauses

- group of nested conditionals and it is hard to determine the normal flow of code execution
- use guard clauses for all the special cases

```
double getPayAmount() {
    double result;
    if(isDead)
        result = deadAmount();
    else {
        if (isSeparated)
            result = separatedAmount();
        else {
            if (isRetried)
                result = retiredAmount();
            else result = normalPayAmount();
         };
     }
     return result;
}
```

Replace Nested Conditional with Guard Clauses

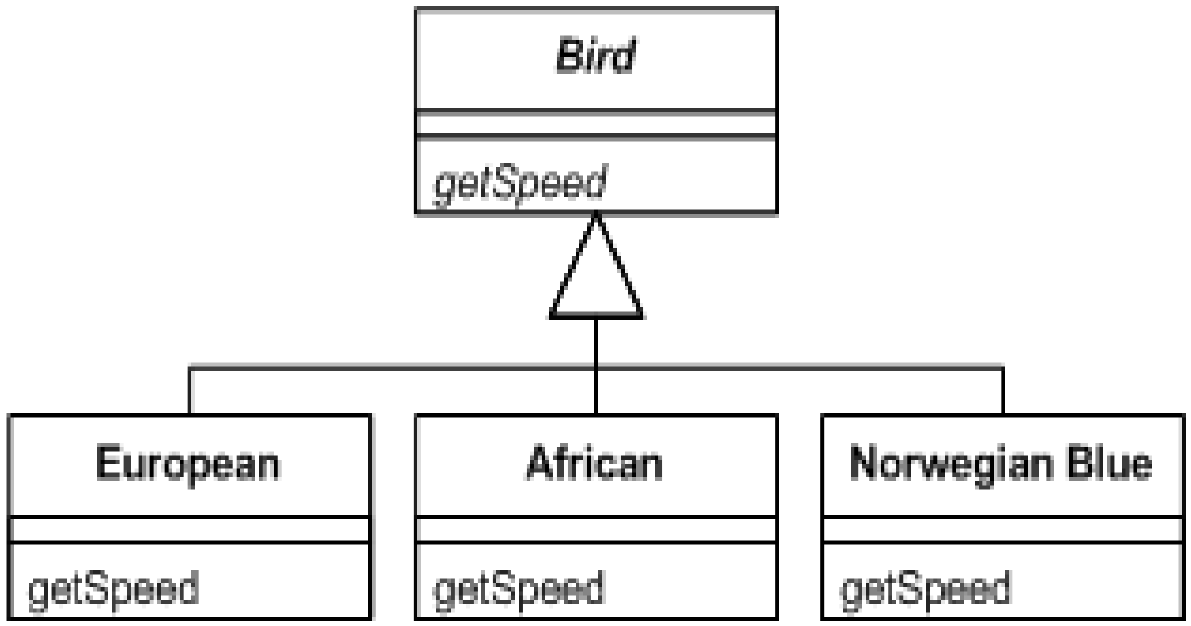
```
double getPayAmount() {
    if (isDead)
        return deadAmount();
    if (isSeparated)
        return separatedAmount();
    if (isRetried)
        return retiredAmount();
    return normalPayAmount();
}
```

Replace Conditional with Polymorphism

- a conditional that performs different actions depending on the object type
- move each leg of the conditional to an overriding method in a subclasses and make the original 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");
}
```

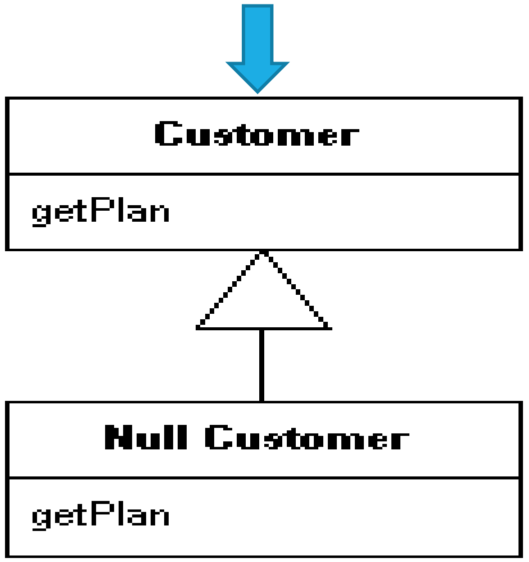
Replace Conditional with Polymorphism



Introduce Null Object

- methods return `null` instead of real objects causing repeating checks for `null` in code
- return a null object that exhibits the default behaviour

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



Introduce Assertion

- a portion of code to work correctly, certain conditions or values must be true
- replace these assumptions with specific assertion checks

```
double getExpenseLimit() {
    //should have either expense limit or a primary project
    return (expenseLimit != NULL_EXPENSE)? expenseLimit:
      primaryProject.getMemberExpenseLimit();
}
```

⇩

```
double getExpenseLimit() {
    Assert.isTrue(expenseLimit != NULL_EXPENSE ||
                                        primaryProject != null);
    return (expenseLimit != NULL_EXPENSE)? expenseLimit:
      primaryProject.getMemberExpenseLimit();
```

Refactoring Exercises

- Simplifying Conditional Expression
  - Decompose Conditional
  - Replace Nested Conditional with Guard Clauses

Decompose Conditional

- code to calculate the charge for something that has separate rates for winter and summer

```
Class Stadium {
  …
  public double summerRate;
  public double winterRate;
  public double winterServiceCharge;

  public double getTicketPrice(Date date, int quantity) {
    double charge;
    if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
      charge = quantity * winterRate + winterServiceCharge;
    } else {
      charge = quantity * summerRate;
    }
    return charge;
  }
}
```

Decompose Conditional

- decompose conditional

Decompose Conditional

- extract methods for price calculation expressions

Replace Nested Conditional with Guard Clauses

- method to calculate adjusted capital

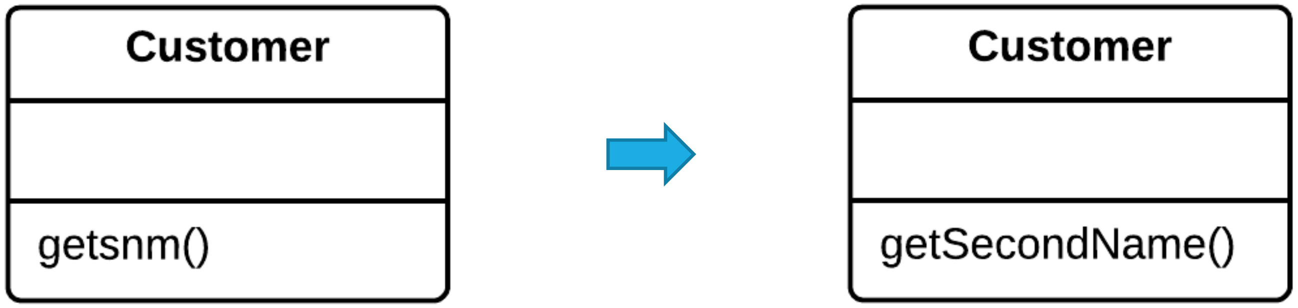
```
public double getAdjustedCapital() {
    double result = 0.0;
    if (capital > 0.0) {
        if (intRate > 0.0 && duration > 0.0) {
            result = (income / duration) * ADJ_FACTOR;
        }
    }
    return result;
}
```

Refactoring Techniques

- Simplifying Method Calls
  - Rename Method
  - Add Parameter
  - Remove Parameter
  - Separate Query from Modifier
  - Parameterize Method
  - Replace Parameter With Explicit Methods
  - Preserve Whole Object
  - Replace Parameter With Method Call
  - Introduce Parameter Object
  - Remove Setting Method
  - Hide Method
  - Replace Constructor With Factory Method
  - Replace Error Code With Exception
  - Replace Exception With Test

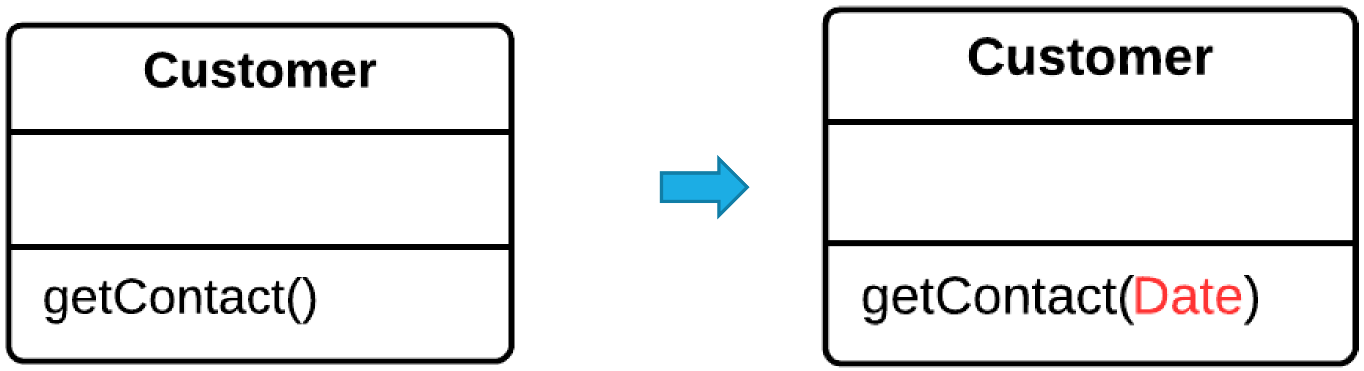
Rename Method

- name of a method does not explain method's purpose
- rename the method



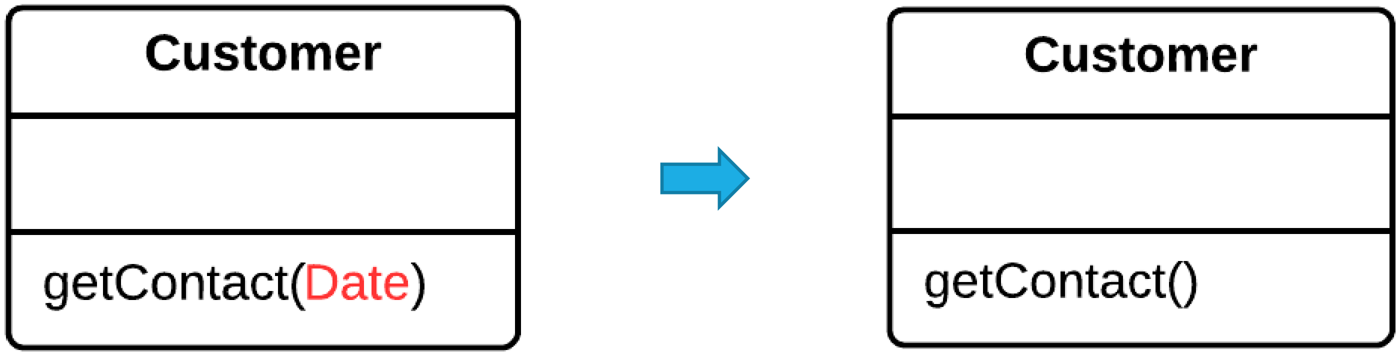
Add Parameter

- method does not have enough data to perform certain actions
- create a new parameter to pass the necessary data



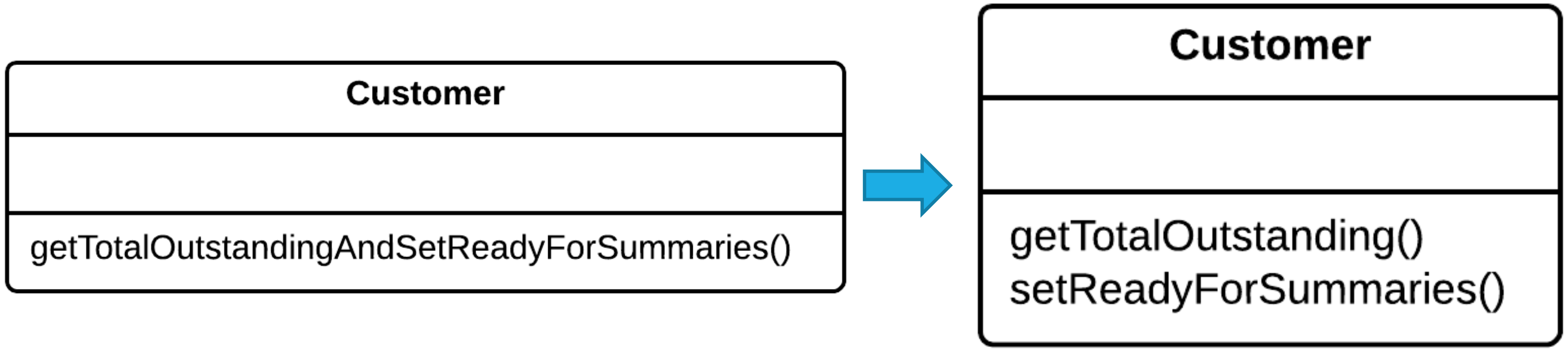
Remove Parameter

- a parameter is not used in the body of a method
- remove the unused parameter



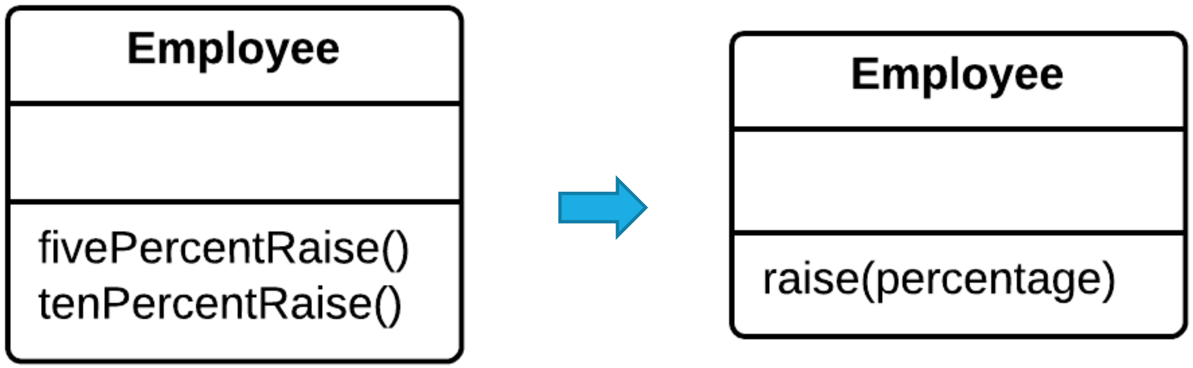
Separate Query from Modifier

- method that returns a value but also changes something inside an object
- split the method into two separate methods where one of them should return the value and the other one modifies the object



Parameterize Method

- multiple methods perform similar actions that are different only in their internal values
- combine these methods by using a parameter that will pass the necessary special value



Replace Parameter with Explicit Method

- method is split into parts, each of which is run depending on the value of a parameter
- extract the individual parts of the method into their own methods and call them

```
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;
}
```

Preserve Whole Object

- get several values from an object and then pass them as parameters to a method
- pass the whole object instead

```
int low = daysTempRange().getLow();
int high = daysTempRange().getHight();
withinPlan = plan.withinRange (low, high);
```

```
withinPlan = plan.withinRange (daysTempRange());
```

Replace Parameter with Method Call

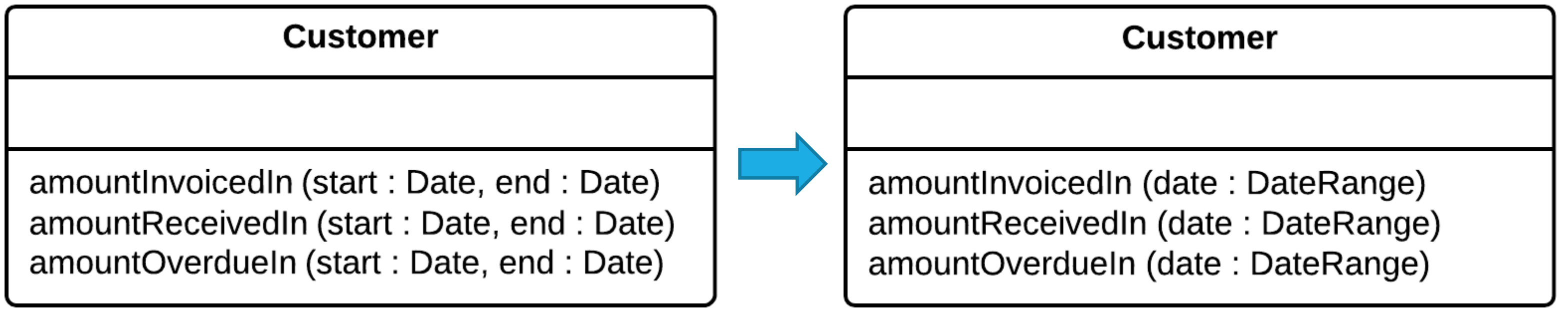
- calling a query method and passing its results as the parameters of another method
- instead of passing the value through a parameter, try placing a query call inside the method body

```
int basePrice = quantity * itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice (basePrice,
                                     discountLevel);
```

```
int basePrice = quantity * itemPrice;
double finalPrice = discountedPrice (basePrice);
```

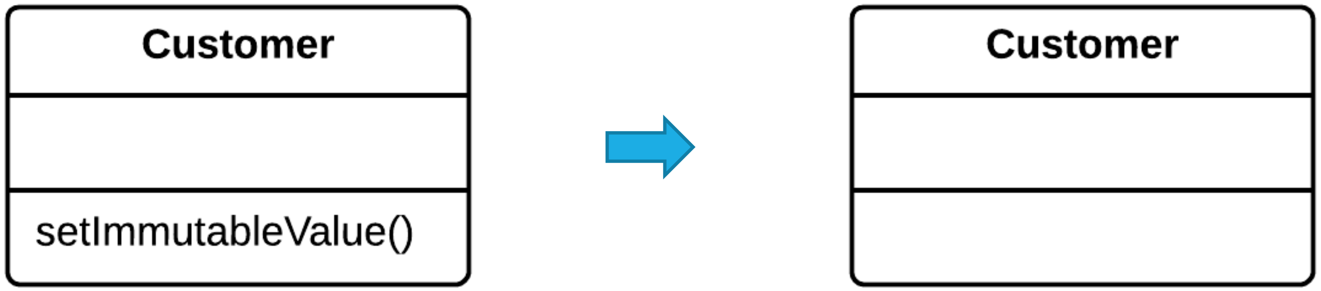
Introduce Parameter Object

- methods contain a repeating group of parameters
- replace these parameters with an object


Customer
amountInvoicedIn (start : Date, end : Date)
amountReceivedIn (start : Date, end : Date)
amountOverdueIn (start : Date, end : Date)
Customer
amountInvoicedIn (date : DateRange)
amountReceivedIn (date : DateRange)
amountOverdueIn (date : DateRange)

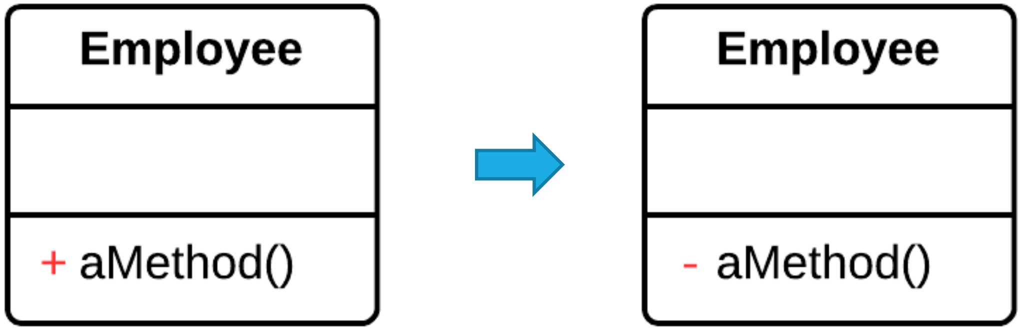
Remove Setting Method

- value of a field should be set only when it is created, and not change at any time after that
- remove methods that set the field's value



Hide Method

- method not used by other classes or is used only inside its own class hierarchy
- make the method private or protected



Replace Constructor with Factory Method

- complex constructor that does something more than just setting parameter values in object fields
- create a factory method and use it to replace constructor calls

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

⬇

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

Replace Error Code with Exception

- method returns a special value that indicates an error
- throw an exception instead

```
int withdraw(int amount) {
    if (amount > balance)
        return -1;
    else {
        balance -= amount;
        return 0;
    }
}
```

⇩

```
void withdraw(int amount) throws BalanceException {
    if(amount > balance) throw new BalanceException();
        balance -= amount;
}
```

Replace Exception with Test

- throw an exception in a place where a simple test would do the job
- replace the exception with a condition test

```
double getValueForPeriod(int periodNumber){
    try{
        return values[periodNumber];
    }catch(ArrayIndexOutOfBoundsException e){
        return 0;
    }
}
```

⇩

```
double getValueForPeriod(int periodNumber){
    if (periodNumber >= values.length)
        return 0;
    return values[periodNumber];
}
```

Refactoring Exercises

- Simplifying Method Calls
  - Separate Query from Modifier
  - Replace Parameter With Explicit Methods
  - Replace Parameter With Method Call
  - Introduce Parameter Object

Separate Query from Modifier

- method to return names of miscreant in security system and sends an alert

```
public void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}

public string foundMiscreant(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
            sendAlert();
            return "Don";
        }
        if (people[i].equals ("John")){
            sendAlert();
            return "John";
        }
    }
    return "";
}
```

Separate Query from Modifier

- create query method that returns appropriate value
- change original method to modifier

Replace Parameter with Explicit Method

- creating different subclass of employee

```
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");
    }
}
```

Replace Parameter with Explicit Method

- replace type code parameter to create Employee with explicit create method

- client code

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

Employee kent =
```

Replace Parameter with Method Call

- calculating discounted price

```
public double getPrice() {
  int basePrice = quantity * itemPrice;
  int discountLevel;
  if (quantity > 100)
    discountLevel = 2;
  else
    discountLevel = 1;
  double finalPrice = discountedPrice(basePrice, discountLevel);
  return finalPrice;
}

private double discountedPrice(int basePrice, int discountLevel){
  if (discountLevel == 2)
    return basePrice * 0.1;
  else
    return basePrice * 0.05;
}
```

Replace Parameter with Method Call

- extract methods for the parameters

Replace Parameter with Method Call

- any further refactoring ?

Introduce Parameter Object

- pair of values showing range in Account class

```
class Account...
  private Vector entries = new Vector();

  double getFlowBetween (Date start, Date end) {
    double result = 0;
    Enumeration e = entries.elements();
    while (e.hasMoreElements()) {
      Entry each = (Entry) e.nextElement();
      if (each.getDate().equals(start) ||
          each.getDate().equals(end) ||
          (each.getDate().after(start) &&
           each.getDate().before(end))) {
              result += each.getValue();
      }
    }
    return result;
  }
```

Introduce Parameter Object

- introduce DataRange parameter object

Introduce Parameter Object

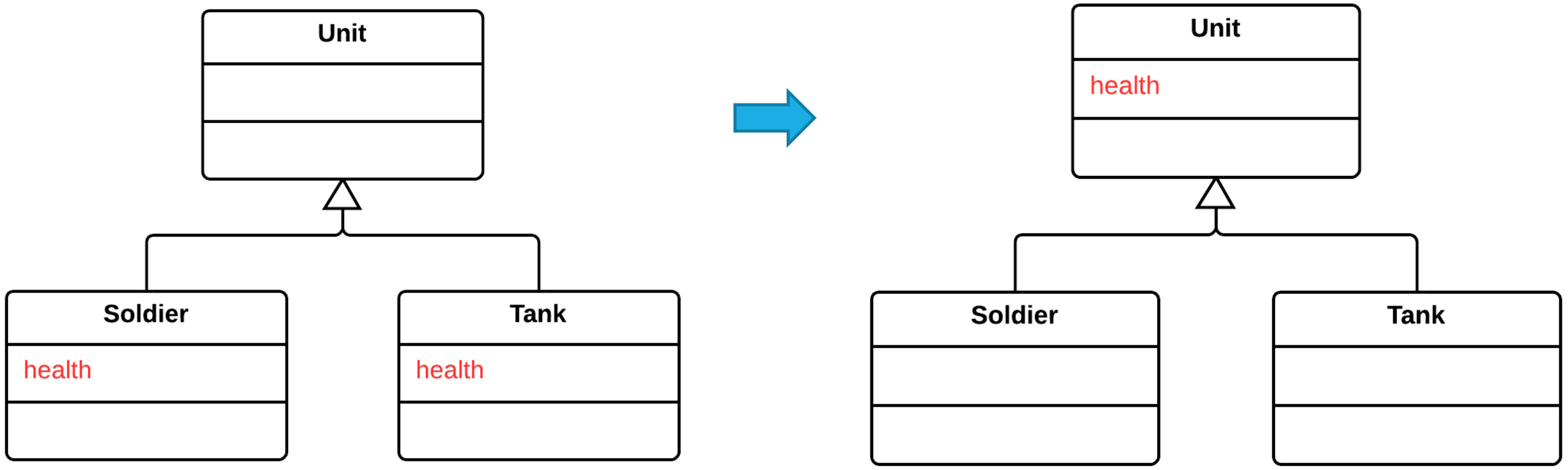
- update Account class accordingly

Refactoring Techniques

- Dealing with Generalization
  - Pull Up Field
  - Pull Up Method
  - Pull Up Constructor Body
  - Push Down Method
  - Push Down Field
  - Extract Subclass
  - Extract Superclass
  - Extract Interface
  - Collapse Hierarchy
  - Form Template Method
  - Replace Inheritance with Delegation
  - Replace Delegation With Inheritance

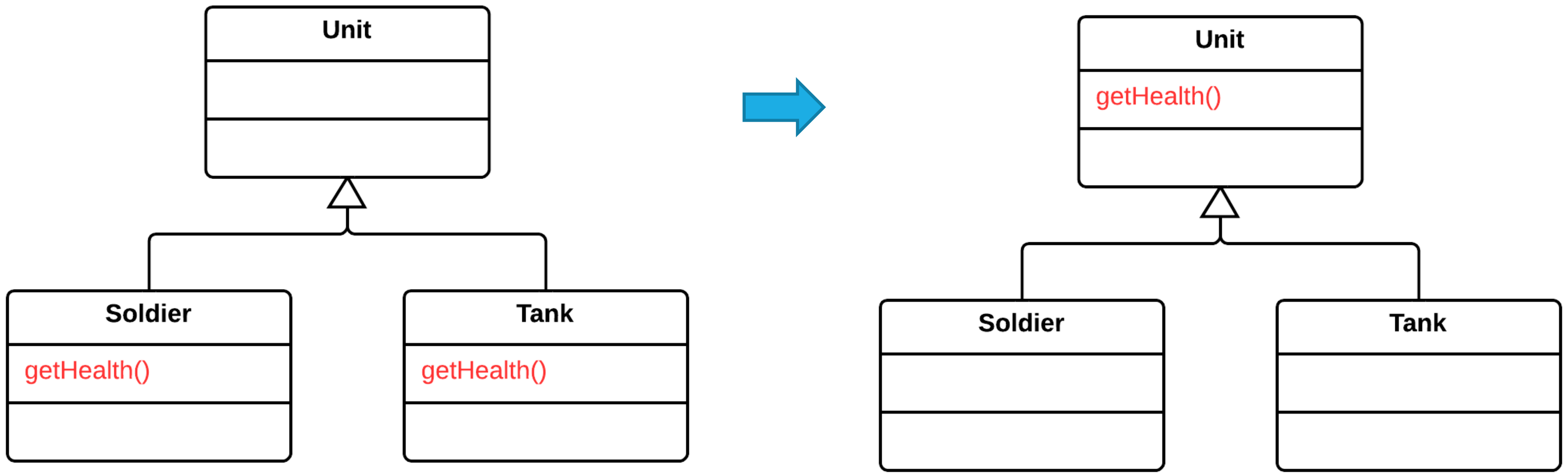
Pull Up Field

- two subclasses having the same field
- move it to the superclass



Pull Up Method

- subclasses have methods that perform similar work
- move them to appropriate superclasses



Pull Up Constructor Body

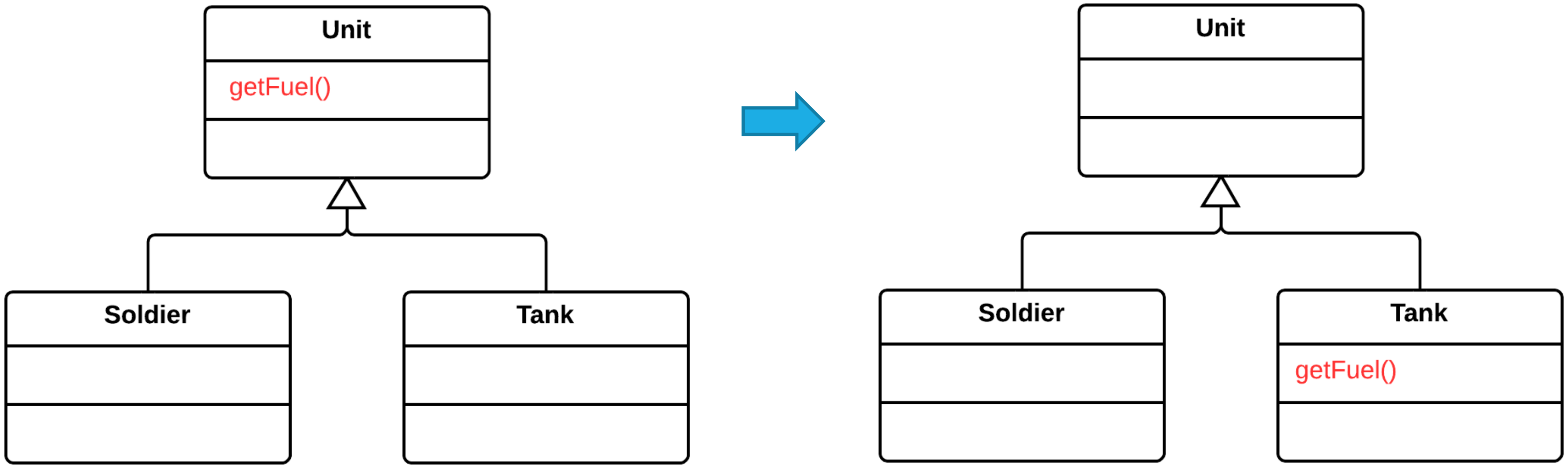
- subclasses have constructors with code that is mostly identical
- create a superclass constructor and move the code that is the same in the subclasses to it and call the superclass constructor in the subclass constructors

```
class Manager extends Employee...
    public Manager (String name, String id, int grade) {
        name = name;
        id = id;
        grade = grade;
    }
```

```
public Manager (String name, String id, int grade) {
    super (name, id);
    grade = grade;
}
```

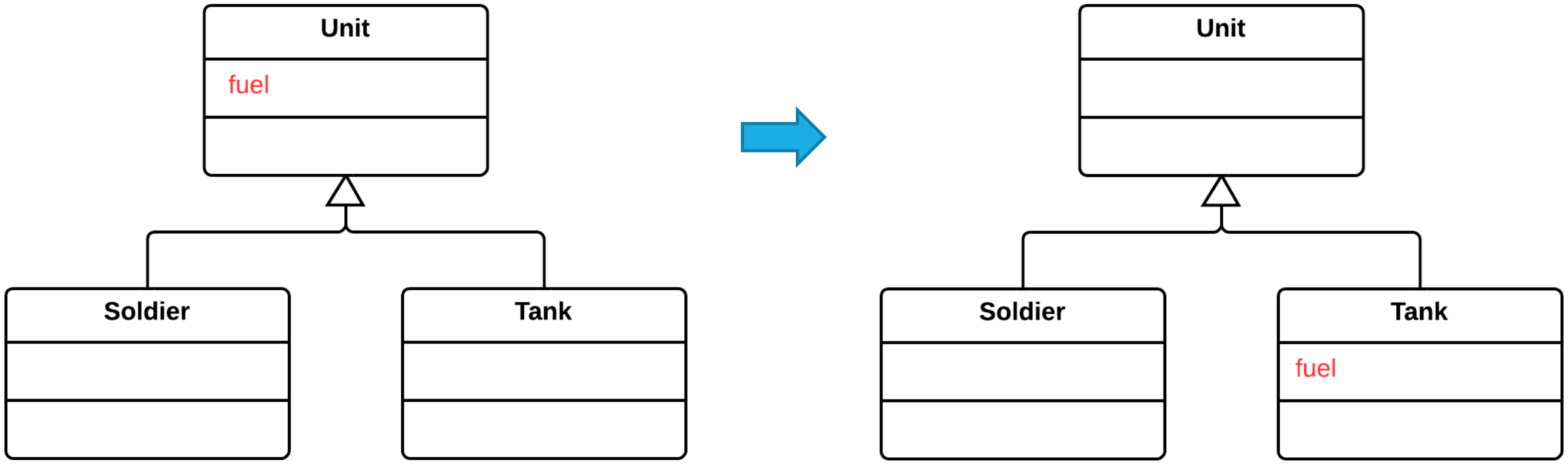
Push Down Method

- behaviour implemented in a superclass used by only one (or a few) subclasses
- move this behaviour to the subclasses



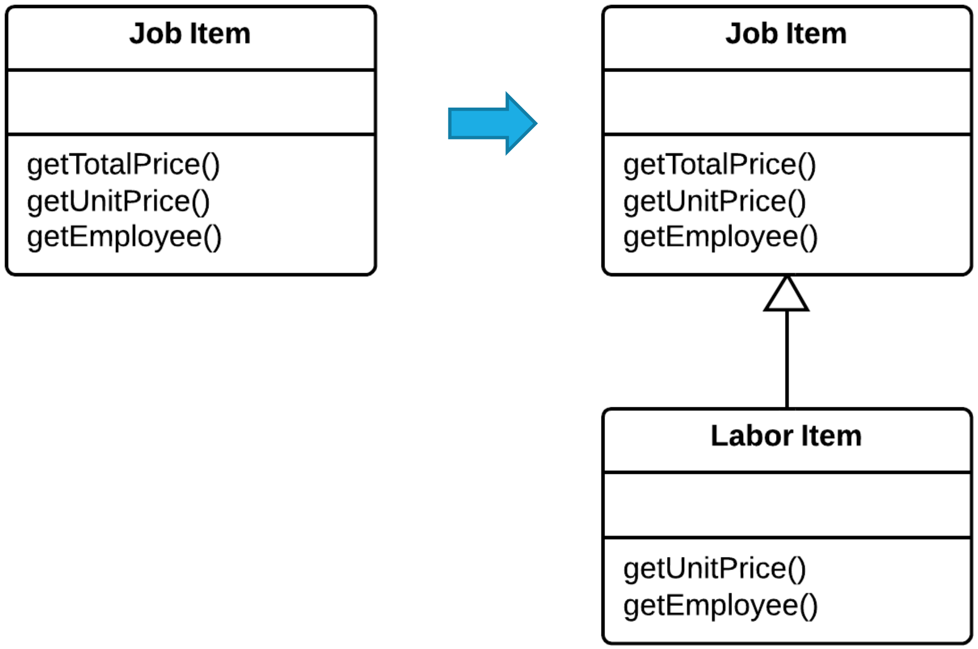
Push Down Field

- field used only in a few subclasses
- move the field to these subclasses



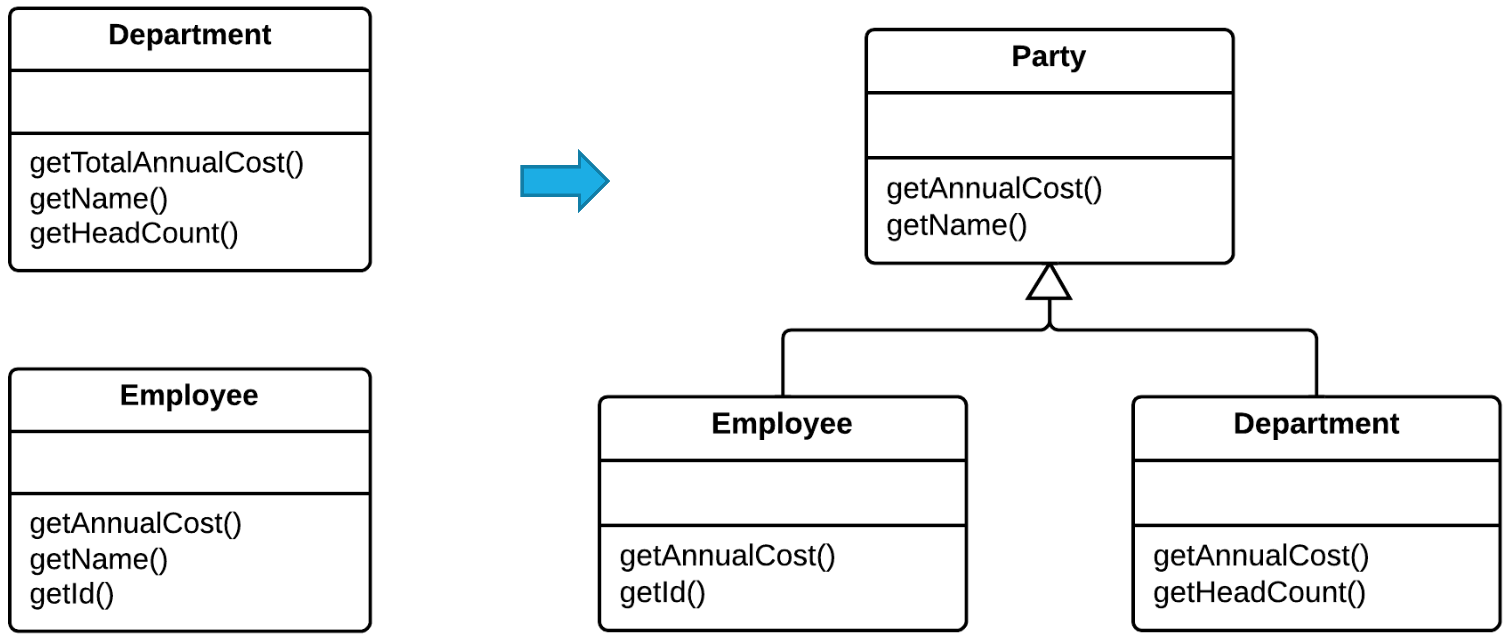
Extract Subclass

- class has features that are used only in certain cases
- create a subclass and use it in these cases



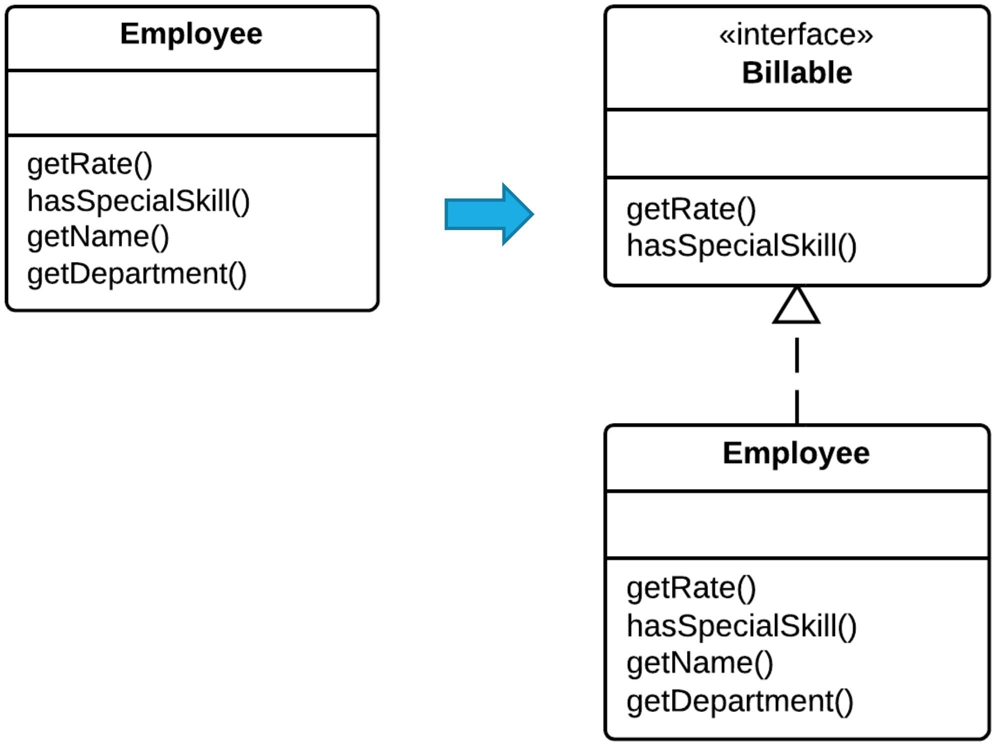
Extract Superclass

- two classes with common fields and methods
- create a shared superclass and move all the identical fields and methods to it



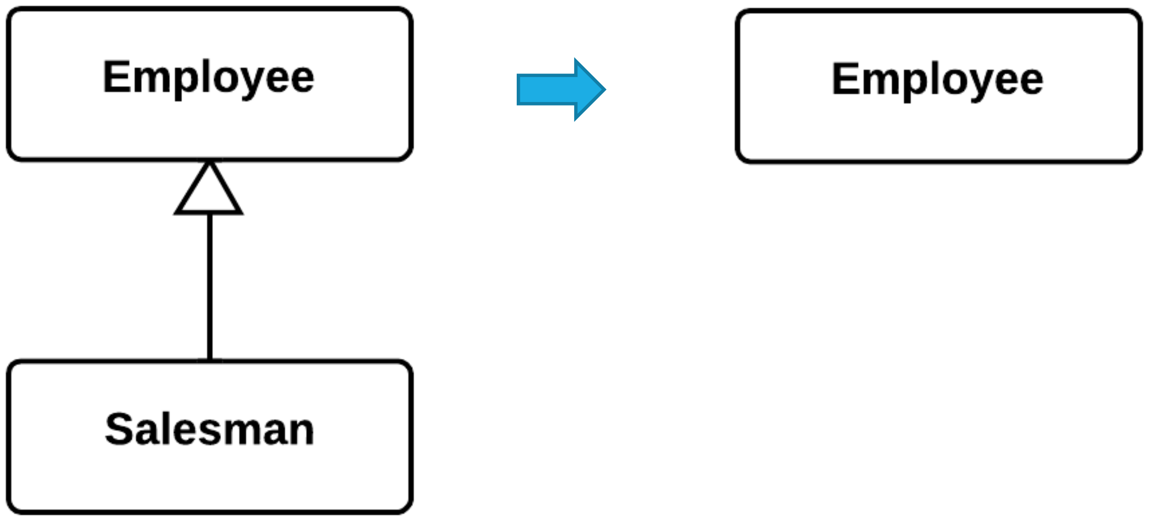
Extract Interface

- multiple clients are using the same part of a class interface or part of the interface in two classes is the same
- move this identical portion to its own interface



Collapse Hierarchy

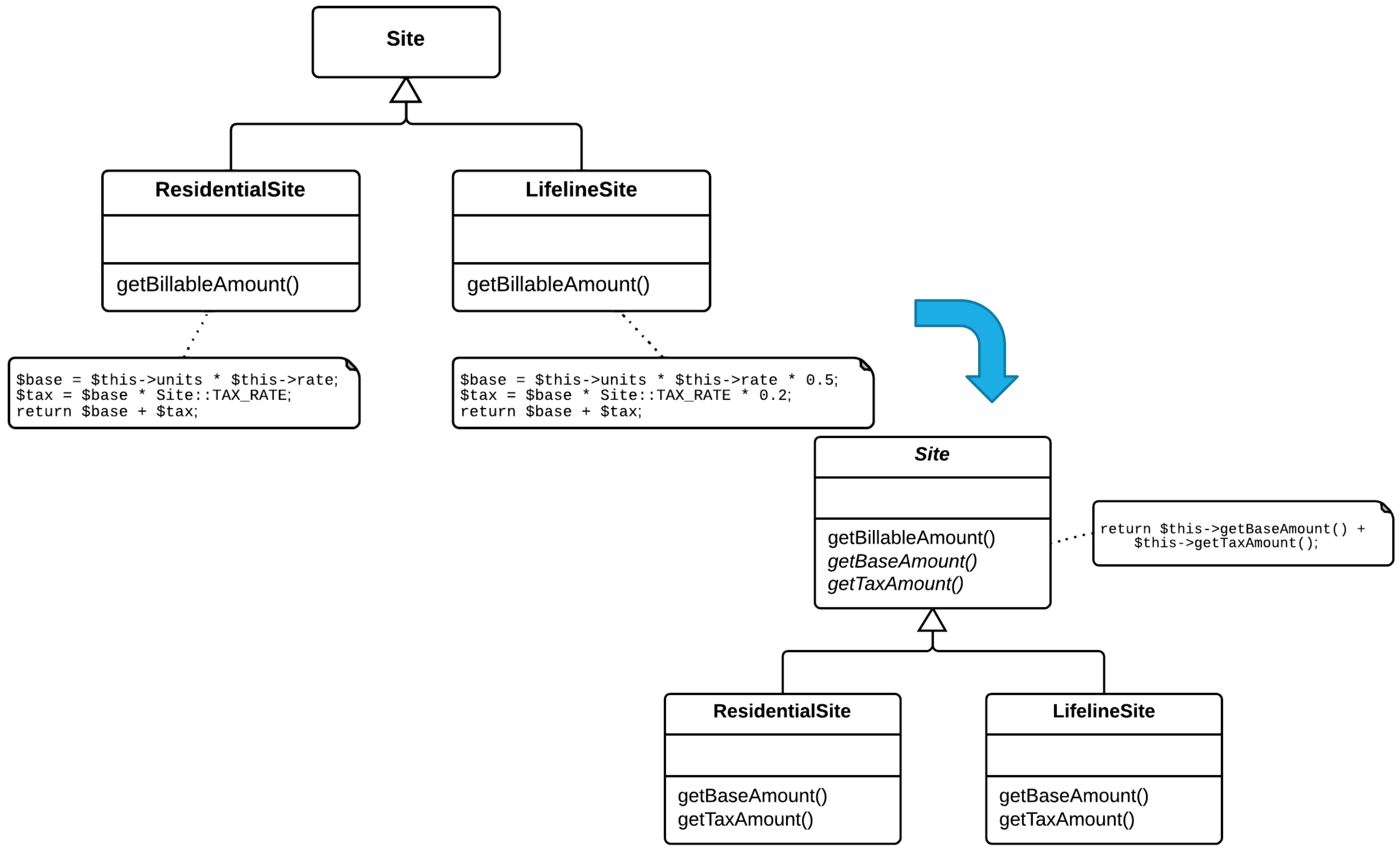
- class hierarchy in which a subclass is practically the same as its superclass
- merge the subclass and superclass



Form Template Method

- subclasses implement algorithms that contain **similar** steps in the same order
- move the algorithm structure and identical steps to a superclass, and leave implementation of the different steps in the subclasses

Form Template Method



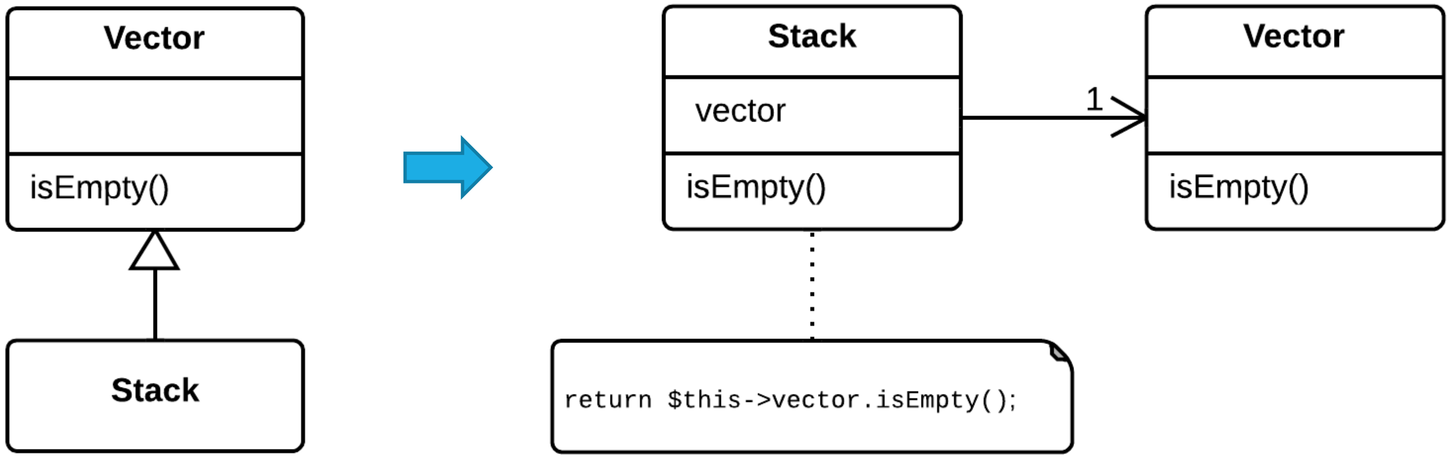
```
$base = $this->units * $this->rate;
$tax = $base * Site::TAX_RATE;
return $base + $tax;
```

```
$base = $this->units * $this->rate * 0.5;
$tax = $base * Site::TAX_RATE * 0.2;
return $base + $tax;
```

```
return $this->getBaseAmount() +
    $this->getTaxAmount();
```

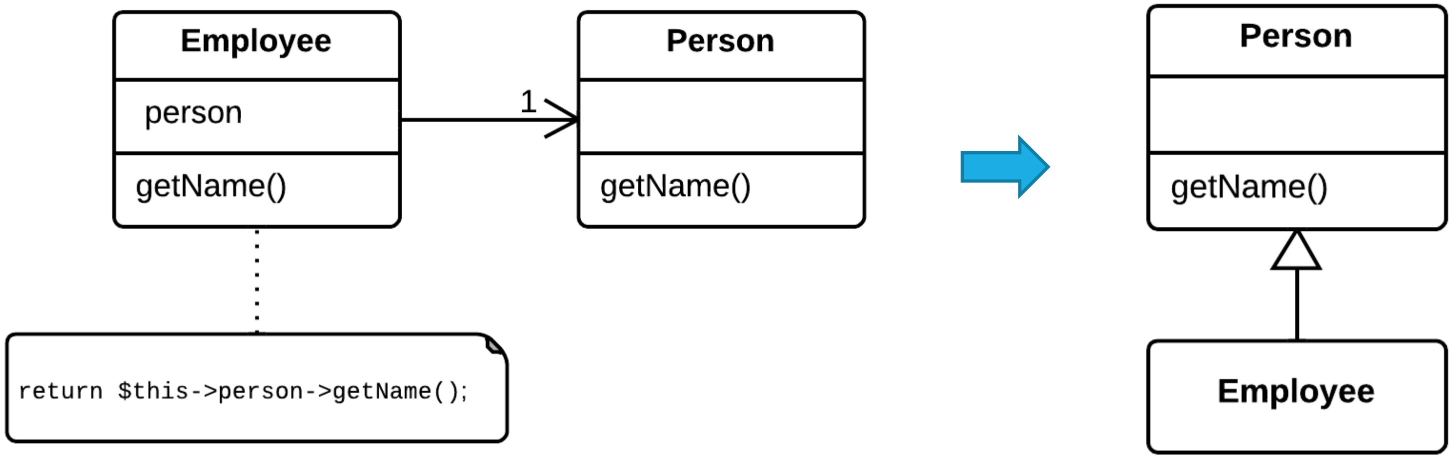
Replace Inheritance with Delegation

- subclass using only a portion of the methods of its superclass (or it does not want to inherit superclass data)
- create a field and put a superclass object in it, delegate methods to the superclass object, and get rid of inheritance



Replace Delegation with Inheritance

- class containing many simple methods that delegate to all methods of another class
- make the class a delegate inheritor, which makes the delegating methods unnecessary

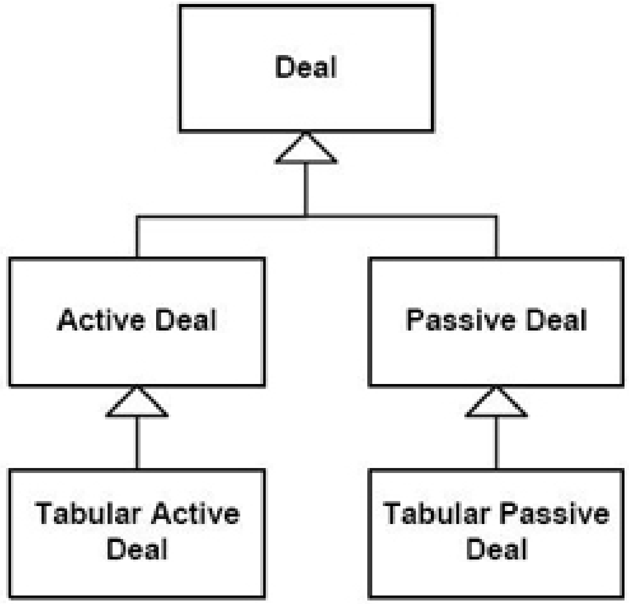


```
return $this->person->getName();
```

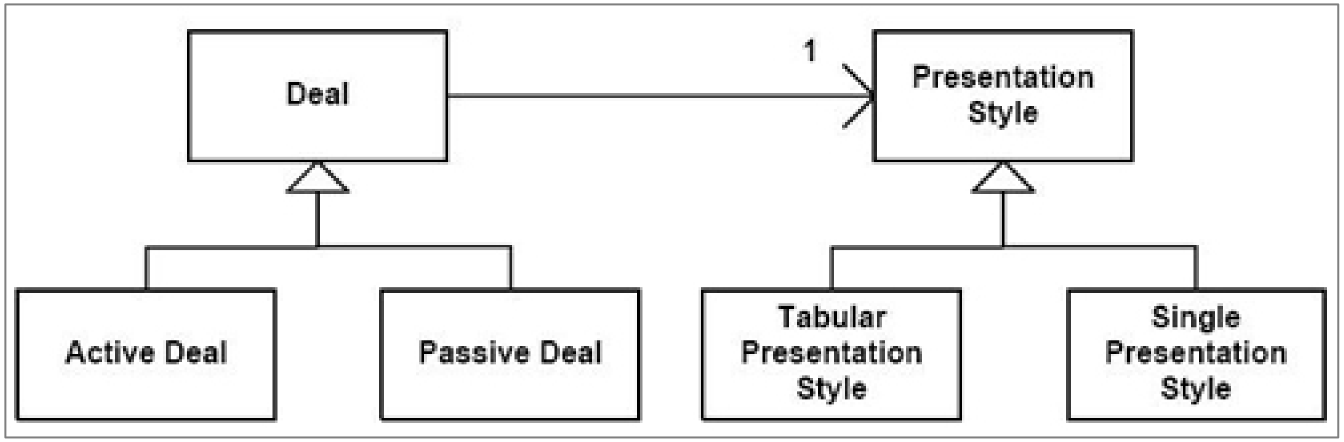
Refactoring Techniques

- Big Refactorings
  - Tease Apart Inheritance
  - Convert Procedural Design to Objects
  - Separate Domain from Presentation
  - Extract Hierarchy

Tease Apart Inheritance

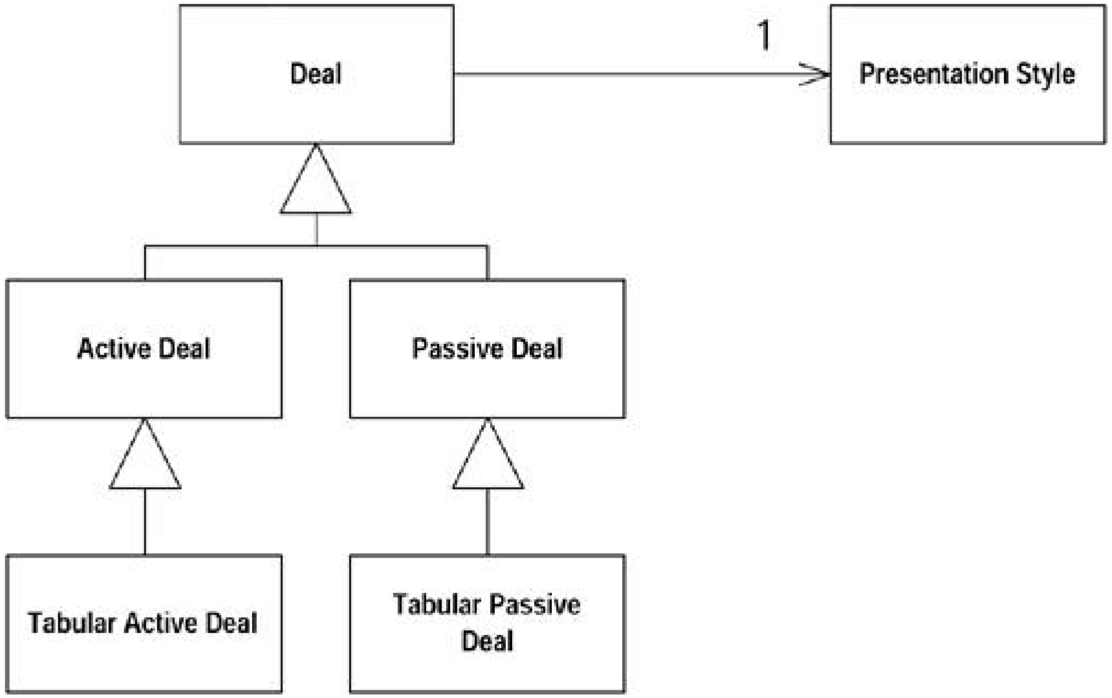


- an inheritance hierarchy that is doing two jobs at once
- create two hierarchies and use delegation to invoke one from the other



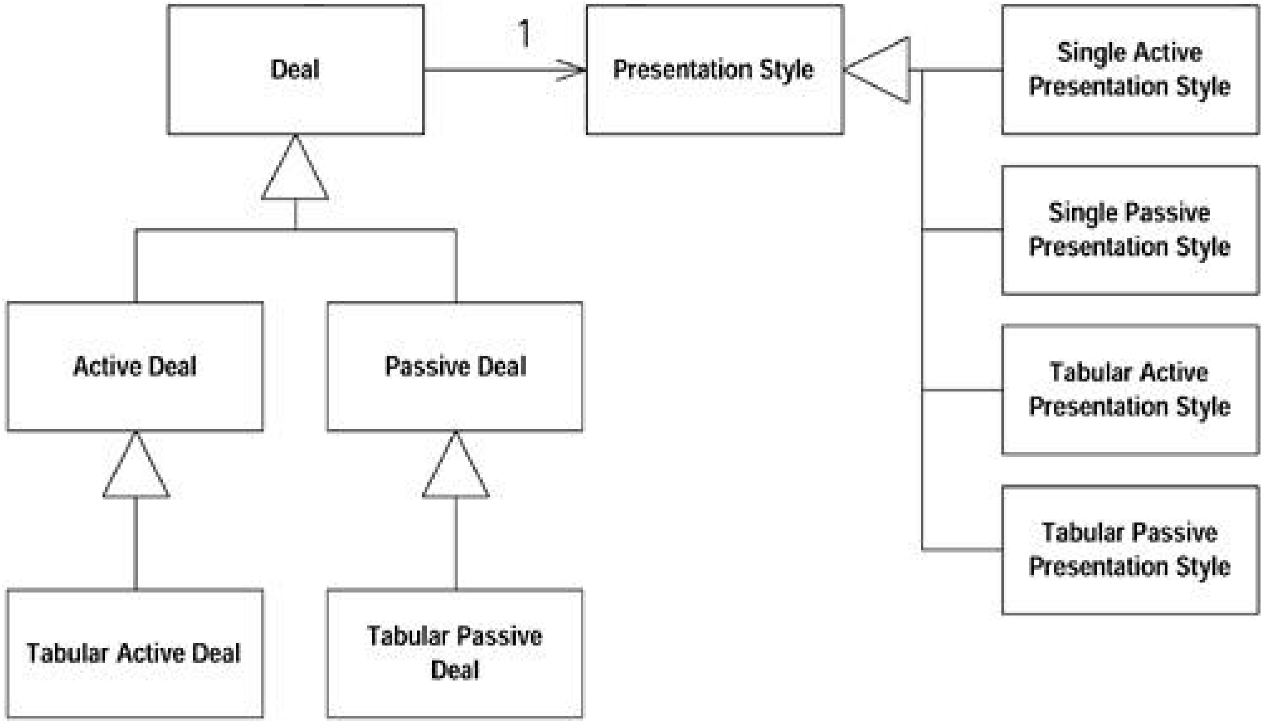
Tease Apart Inheritance

1. decide job to be retained in the current hierarchy (more important) and which to be moved to another hierarchy
2. create an object at common superclass with Extract Class for the subsidiary job and add an instance variable to hold this object



Tease Apart Inheritance

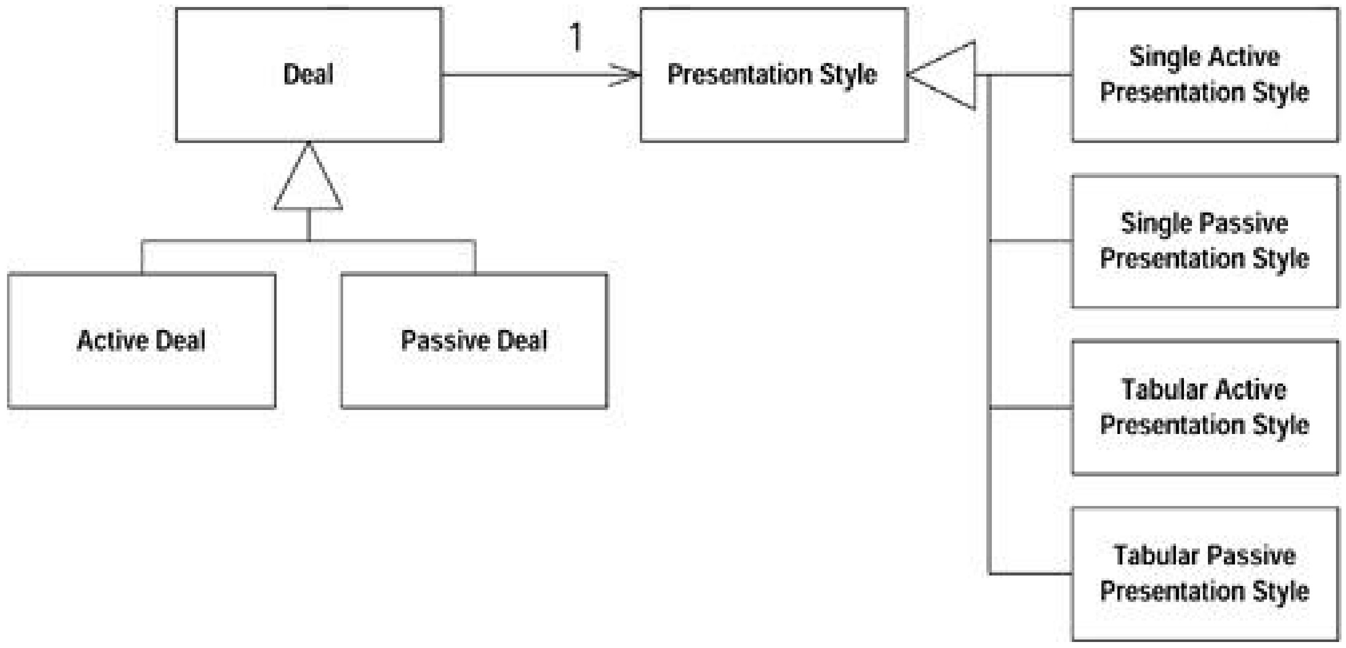
3. create subclasses of the extracted object for each of the subclasses in the original hierarchy and initialize the instance variable created in the previous step to an instance of this subclass
4. for each subclasses, move the behaviour in the subclass to the extracted object - Move Method



Deal
1
Presentation Style
Single Active Presentation Style
Single Passive Presentation Style
Tabular Active Presentation Style
Tabular Passive Presentation Style
Active Deal
Passive Deal
Tabular Active Deal
Tabular Passive Deal

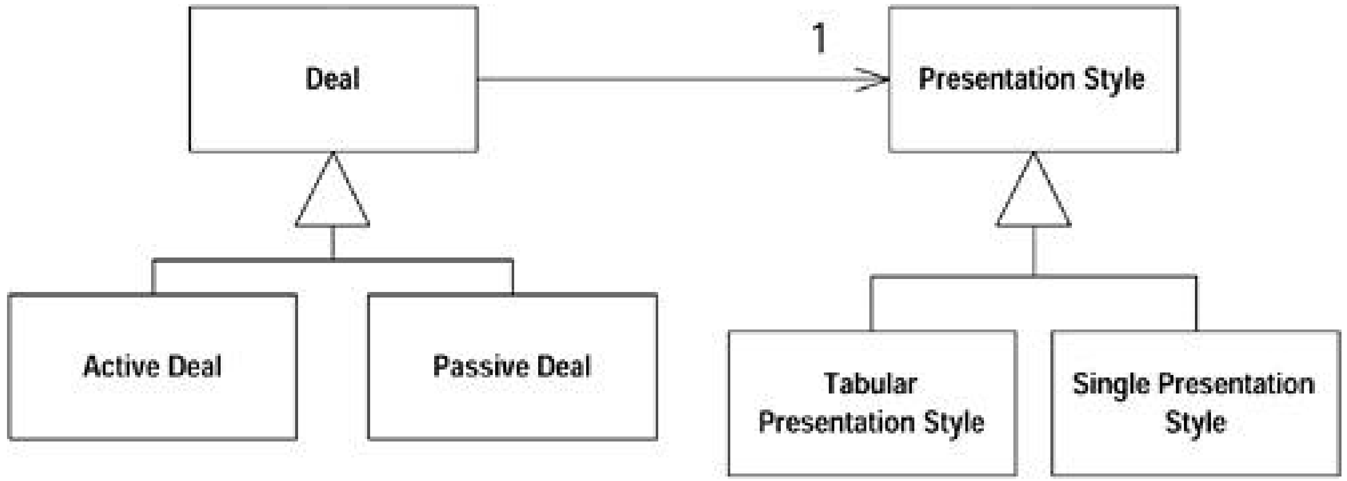
Tease Apart Inheritance

5. delete the subclass when it has no more code


Deal
1
Presentation Style
Single Active Presentation Style
Single Passive Presentation Style
Tabular Active Presentation Style
Tabular Passive Presentation Style
Active Deal
Passive Deal

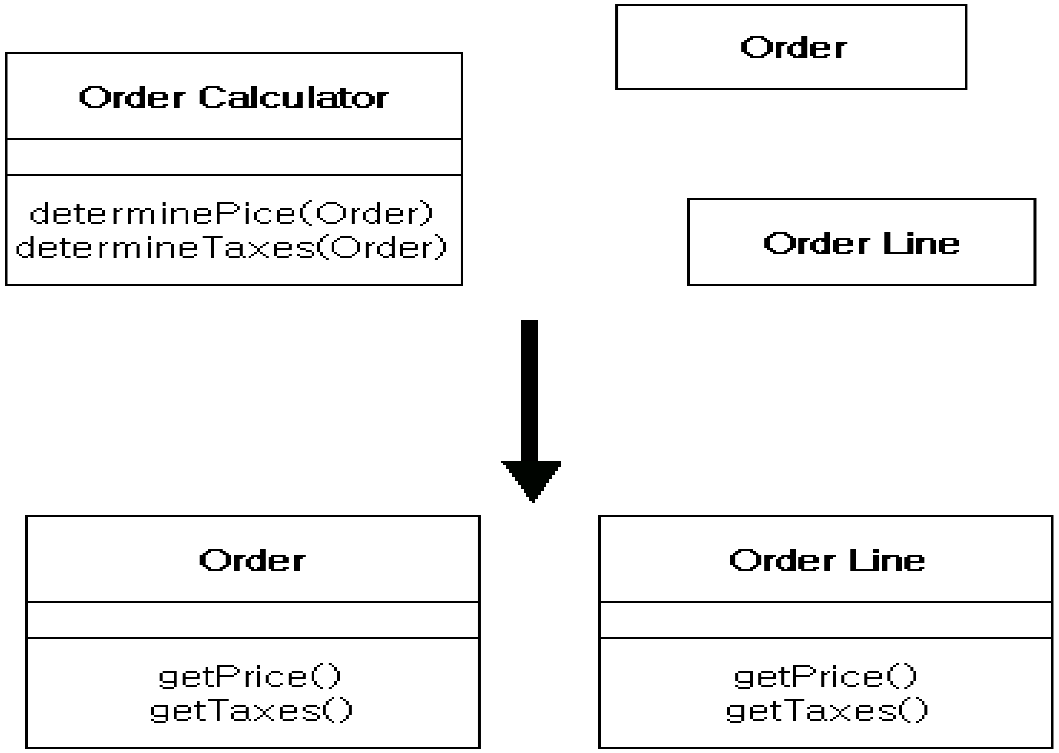
Tease Apart Inheritance

6. continue until all the subclasses are gone and examine the new hierarchy for possible further refactoring such as Pull Up or Pull Up Field



Convert Procedural Design to Objects

- code written in a procedural style
- turn data records into objects, break up the behaviour, and move the behaviour to the objects



Convert Procedural Design to Objects

1. take each record type and turn it into a dumb data object with accessors

2. take all the procedural code and put it into a single class

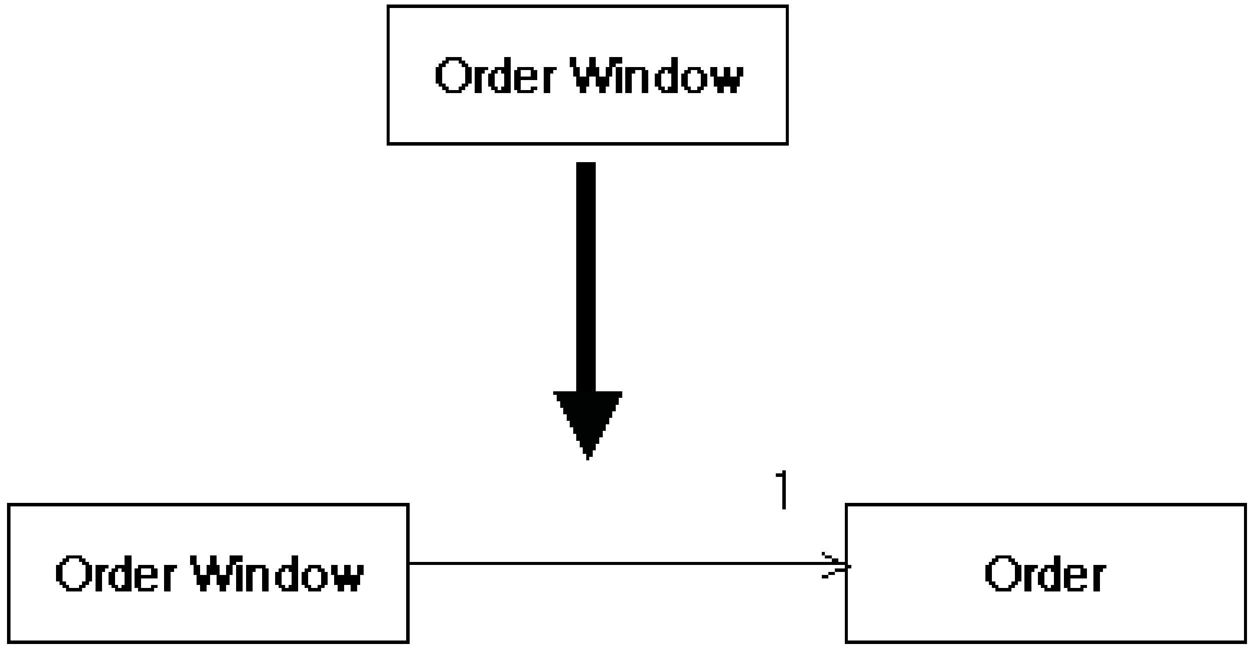
3. Extract Method for each long procedural and the related refactoring to break it down

4. during the procedure break down, move each one to the appropriate dumb data class - Move Method

5. continue until all the behaviour is removed from the original class. (delete the original class if it was purely procedural)

Separate Domain from Presentation

- GUI classes that contain domain logic
- separate the domain logic into separate domain class

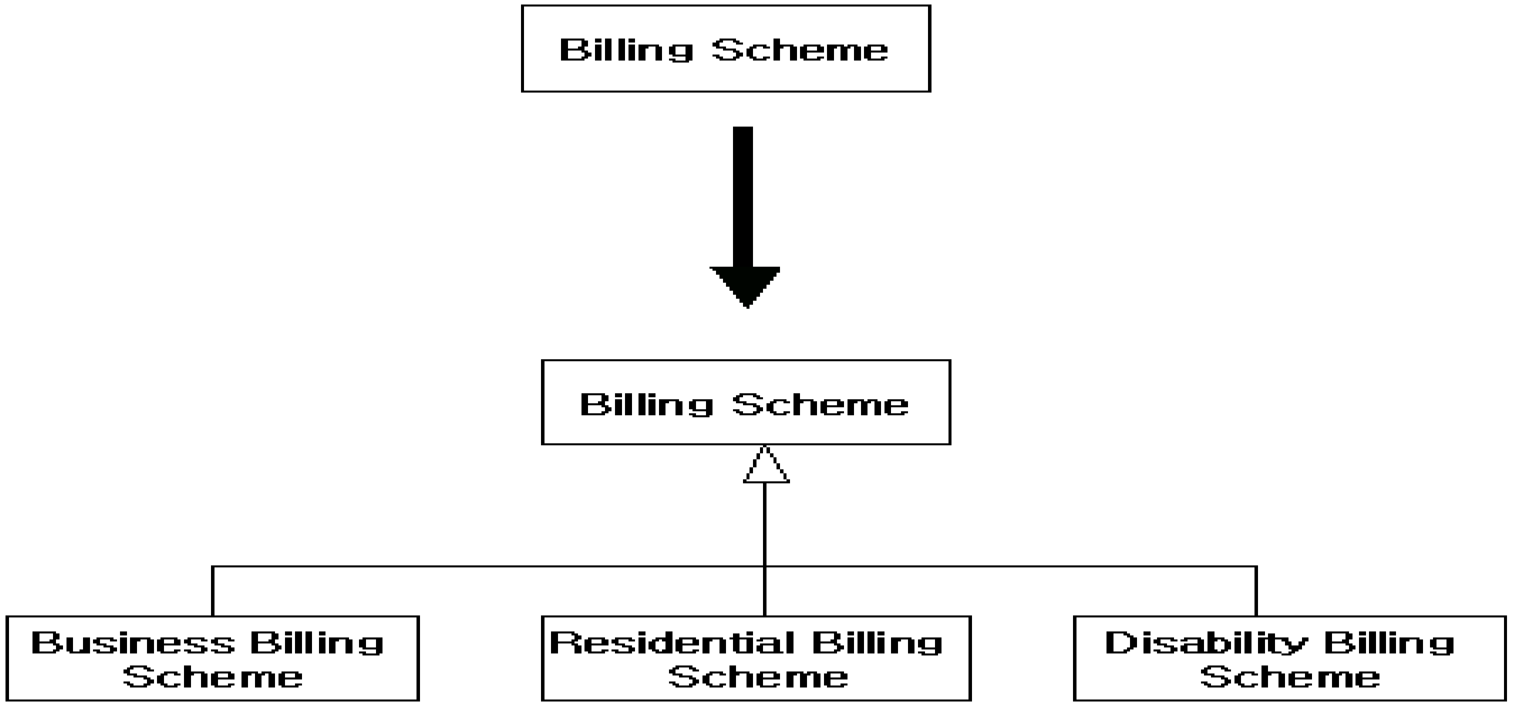


Separate Domain from Presentation

1. create a domain class for each window
2. if there is a table, create a class representing the table rows and use a collection on the domain class for the window to hold the row domain objects
3. examine the data on the window
   - if used only for user interface purpose, leave it on the window.
   - if used within the domain logic but is not actually displayed on the window, Move Method to move it to the domain object
   - if used by both, it should be in both place with kept sync, Duplicate Observed Data
4. examine the logic in the presentation class :
   - separate logic about the presentation from domain logic - Extract Method and move it to the domain model - Move Method
5. further refactor domain objects

Extract Hierarchy

- a class that is doing too much work, at least in part through many conditional statement
- create a hierarchy of classes in which each subclass represents a special case



Extract Hierarchy

- variation is unknown :
  1. identify variation
  2. create a subclass for that special case and use Replace Constructor with Factory Model on the original
  3. one at a time, copy methods that contain conditional logic to the subclass and simplify the methods
  4. continue isolating special cases until you can declare the superclass abstract
  5. delete the bodies of the method in the superclass that are overridden in all subclasses and make the superclass declarations abstract
- variation is clear :
  1. create a subclass for each variation
  2. use Replace Constructor with Factory Model to return appropriate subclass for each variation
  3. take methods that have conditional logic and apply Replace Conditional with Polymorphism

Treatments for Bad Smells

- Bloaters
  - **gargantuan** code, methods and classes that are difficult to handle
    - Long Methods
    - Large Class
    - Primitive Obsession
    - Long Parameter List
    - Data Clumps

Treatments for Bloaters

- Long Method
  - method containing too many lines of code

- Treatment
  - if comment is needed inside a method, move the code to a new method with descriptive name (even a single line)
    - to reduce the length of a method body, use Extract Method.
    - if local variables and parameters interfere, use Replace Temp with Query, Introduce Parameter Object or Preserve Whole Object
    - if none of above works try moving the entire method to a separate object via Replace Method with Method Object
    - conditional operators and loops are a good clue that code can be moved to a separate method : Decompose Conditional for conditionals and Extract Method for loops

Treatments for Bloaters

- Larger Class
  - class containing many fields / methods / lines of code

- Treatment
  - split up class :
    - if part of the behaviour in the large class can be split up into a separate component, Extract Class
    - if part of the behaviour in the large class can be implemented in different ways or used in rare cases, Extract Subclass
    - if a list of operations and behaviour that client can use is needed, Extract Interface
    - if a large class is for the graphical interface, move some of its data and behaviour to a separate domain object. This may need storing copies of some data in two places and keeping the data consistent. Then, Duplicate Observed Data

Treatments for Bloaters

- Primitive Obsession
  - using primitives instead of small objects for simple tasks
  - using constants for coding information
  - using string constants as field names for use in data arrays

- Treatment
  - for a large variety of primitive fields, logically group some of them with associated behaviour into their own class Replace Data Value with Object.
  - if the values of primitive fields are used in method parameters, Introduce Parameter Object or Preserve Whole Object.
  - when complicated data is coded in variables, Replace Type Code with Class, Replace Type Code with Subclasses or Replace Type Code with State/Strategy
  - if there are arrays among the variables Replace Array with Object

Treatments for Bloaters

- Long Parameter List
  - more than three or four parameters for a method

- Treatment
  - if some of the arguments are just results of method calls of another object, Replace Parameter with Method Call and place the object in the field of its own class or pass as a method parameter.
  - instead of passing a group of data received from another object as parameters, pass the object itself to the method, Preserve Whole Object
  - for several unrelated data elements, sometimes it is possible to merge them into a single parameter object, Introduce Parameter Object

Treatments for Bloaters

- Data Clumps
  - different parts of the code containing identical groups of variables - should be turned into their own classes

- Treatment
  - if repeating data comprises the fields of a class, Extract Class to move the fields to their own class
  - if the same data clumps are passed in the parameters of methods, Introduce Parameter Object to replace them as a class
  - if some of the data is passed to other methods, consider passing the entire data object, Preserve Whole Object
  - examine the code used by these fields and consider moving this code to a data class if appropriate

Treatments for Bad Smells

- Object-Oriented Abusers
  - **incomplete** or **incorrect** application of OO principles
    - Switch Statement
    - Temporary Field
    - Refused Bequest
    - Alternative Classes With Different Interface

Treatments for OO Abusers

- Switch Statements
  - having complex switch operator or sequence of if statements

- Treatment
  - to isolate `switch` and put it in the right class, Extract Method and then Move Method
  - if a `switch` is based on type code (e.g. when the program's runtime mode is switched), Replace Type Code with Subclasses or Replace Type Code with State/Strategy
  - after specifying the inheritance structure, Replace Conditional with Polymorphism
  - if operator has not too many conditions and they all call same method with different parameters, polymorphism will be superfluous → break the method into multiple smaller methods with Replace Parameter with Explicit Methods and change the `switch` accordingly
  - if one of the conditional options is null, Introduce Null Object

Treatments for OO Abusers

- Temporary Field
  - fields getting their values only under certain circumstances and being empty outside of these circumstances

- Treatment
  - put temporary fields and all code operating on them in a separate class, Extract Class - in other words, creating a method object Replace Method with Method Object
  - Introduce Null Object and integrate it in place of the conditional code which checks the temporary field values for existence

Treatments for OO Abusers

- Refused Bequest
  - subclass using only some of the methods and properties inherited from its parents
  - the unneeded / unwanted methods may simply go unused or be redefined and give off exceptions

- Treatment
  - if inheritance makes no sense and the subclass really does have nothing in common with the superclass, eliminate inheritance - Replace Inheritance with Delegation
  - if inheritance is appropriate, get rid of unneeded fields and methods in the subclass: extract all fields and methods needed by the subclass from the parent class, put them in a new subclass, and set both classes to inherit from it - Extract Superclass

Treatments for OO Abusers

- Alternative Classes with Different Interfaces
  - two classes performing identical functions but having different method names

- Treatment
  - put interface of classes in terms of a common denominator:
    - make method name identical in all the classes - Rename Method
    - make the signature and implementation of methods the same - Move Method, Add Parameter, Parameterize Method
    - if only part of the functionality of the classes is duplicated - Extract Superclass (existing classes become subclasses)
    - after determining which treatment to use and implementing it, one of the classes can be deleted

Treatments for Bad Smells

- Change Preventers
  - **change** is one place **resulting** in many **changes in other** places
    - Divergent Change
    - Shotgun Surgery
    - Parallel Inheritance Hierarchies

Treatments for Change Preventers

- Divergent Change
  - changing a class causes having to change many unrelated methods
- Treatment
  - split up the behaviour of the class Extract Class
  - if different classes have the same behaviour, combine the classes through inheritance - Extract Superclass, Extract Subclass

Treatments for Change Preventers

- Shotgun Surgery
  - making any modifications requires making many small changes to many different classes

- Treatment
  - move existing class behaviours into a single class - Move Method, Move Field (if no appropriate class for this exists, create a new one)
  - if moving code to the same class leaves the original classes almost empty, delete these now-redundant classes - Inline Class

Treatments for Change Preventers

- Parallel Inheritance Hierarchies
  - creating a subclass for a class causes a need to create a subclass for another class

- Treatment
  - two step de-duplication of parallel class hierarchies:
    1. make instances of one hierarchy refer to instances of another hierarchy
    2. remove the hierarchy in the referred class, Move Method, Move Field

Treatments for Bad Smells

- Dispensables
  - something **pointless** and unnecessary whose absence would make code cleaner
    - Comments
    - Duplicate Code
    - Lazy Class
    - Data Class
    - Dead Code
    - Speculative Generality

Treatments for Dispensables

- Comments
  - method filled with explanatory comments

- Treatment
  - if a comment is intended to explain a complex expression, split it into understandable subexpressions - Extract Variable
  - if a comment explains a section of code, turn it into a separate method - Extract Method
  - if an already extracted method still needs explanation, give it a self-explanatory name - Rename Method
  - if assert rules are needed about a state necessary for the system to work - Introduce Assertion

Treatments for Dispensables

- Duplicate Code
  - two code fragments looking almost identical
- Treatment
  - if the same code is in multiple methods of the same class, Extract Method and place calls for the new method in both places
  - if the same code is in two subclasses of the same level :
    - Extract Method for both classes and Pull Up Field for the fields used in the method that are being pulled up
    - if the duplicate code is inside a constructor - Pull Up Constructor Body
    - if the duplicate code is similar but not completely identical - Form Template Method
    - if two methods do the same thing but use different algorithms, select the best algorithm and substitute - Substitute Algorithm

Treatments for Dispensables

- Duplicate Code
  - two code fragments looking almost identical

- Treatment
  - if duplicate code is found in two different classes :
    - if the classes are not part of a hierarchy, Extract Superclass to create a single superclass for these classes maintaining all the previous functionality
    - if it is difficult or impossible to create a superclass, Extract Class for one class and use the new component in the other
  - if many conditional expressions exists performing the same code with different conditions, merge these operators into a single condition Consolidate Conditional Expression and Extract Method to place the condition in a separate method with self explanatory name
  - if the same code is performed in all branches of a conditional expression: place the identical code outside of the condition tree - Consolidate Duplicate Conditional Fragments

Treatments for Dispensables

- Lazy Class
  - understanding and maintaining classes always costs time and money
  - class which doesn't do enough to earn attention should be deleted

- Treatment
  - components that are near-useless should be given - Inline Class
  - for subclasses with few functions - Collapse Hierarchy

Treatments for Dispensables

- Data Class
  - class containing only fields and crude methods for accessing them (getters and setters)
  - these are simply containers for data used by other classes, not containing any additional functionality and cannot independently operate on the owned data

- **Treatment**
  - if a class contains public fields, Encapsulate Field to hide them from direct access where access be performed via getters and setters only
  - Encapsulate Collection for data stored in collections (such as arrays)
  - migrate functionality in client code (used by the class) that would be better located in the data class to the data class - Move Method, Extract Method
  - remove methods for data access that give overly broad access to the class data - Remove Setting Method, Hide Method

Treatments for Dispensables

- Dead Code
  - no longer used variable, parameter, field, method or class (obsolete).

- Treatment
  - find dead code using a good IDE
  - delete unused code and files
  - an unnecessary class where a subclass or superclass is used, Inline Class or Collapse Hierarchy
  - remove unneeded parameters - Remove Parameter

Treatments for Dispensables

- Speculative Generality
  - having an unused class, method, field or parameter

- Treatment
  - remove unused abstract classes - Collapse Hierarchy
  - eliminate unnecessary delegation of functionality to another class - Inline Class
  - remove unused methods - Inline Method
  - methods with unused parameters should be examined - Remove Parameter
  - delete unused fields

Treatments for Bad Smells

- Couplers
  - contributing to **excessive coupling** between classes
    - Feature Envy
    - Inappropriate Intimacy
    - Message Chains
    - Middle Man
    - Incomplete Library Class

Treatments for Couplers

- Feature Envy
  - method accessing the data of another object more than its own data

- Treatment
  - if things change at the same time, keep them in the same place since data and functions that use this data are usually changed together (exceptions possible)
  - if a method should be moved clearly to another place, Move Method
  - if only part of a method accesses the data of another object, Extract Method
  - if a method uses functions from several other classes,
    1. determine which class contains most of the data used
    2. place the method in this class along with the other data
    - alternatively, split the method into several parts which can be placed in different places in different classes - Extract Method

Treatments for Couplers

- Inappropriate Intimacy
  - one class using the internal fields and methods of another class

- Treatment
  - move parts of one class to the used class - Move Method and Move Field (only works if the parts are unneeded in the first class)
  - Extract Class and Hide Delegate on the class to make the code relations "official"
  - if the classes are mutually interdependent, Change Bidirectional Association to Unidirectional
  - if this is between a subclass and the superclass, Replace Delegation with Inheritance

Treatments for Couplers

- Message Chains
  - series of calls resembling $a->b()->c()->d()

- Treatment
  - delete a message chain - Hide Delegate
  - think of why the end object is being used and Extract Method for this functionality and move it to the beginning of the chain Move Method

Treatments for Couplers

- Middle Man
  - class performing only one action which is delegating work to another class
  - why does this class exist at all?

- Treatment
  - if most of a method's classes delegate to another class - Remove Middle Man