ISLAMIC UNIVERSITY OF TECHNOLOGY

SWE 4301

LECTURE 18

Topic: Refactoring

March 15, 2025

PREPARED BY

Maliha Noushin Raida

Lecturer, Department of CSE

Contents

1 REFACTORING

Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
Refactor (verb): to restructure software by applying a series of refactorings without changing its observable behavior.

Refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code yet improves its internal structure. It is a disciplined way to clean up code that the chances of introducing bugs. In essence, when you refactor, you improving the design of the code after it has been written.

With refactoring, you can take a bad design, chaos even, and rework it into well-designed code. Each step is simple, even simplistic. Yet the cumulative effect of these small changes can radically improve the design. It is the exact reverse of the normal notion of software decay.

1.1 Composing Method

Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand—and even harder to change. The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.

1.1.1 Extract Method

Problem

You have a code fragment that can be grouped together.

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

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

Solution

Move this code to a separate new method (or function) and replace the old code with a call to the method.

```
void printOwing() {
  printBanner();
  printDetails(getOutstanding());
}

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

Why Refactor

The more lines found in a method, the harder it's to figure out what the method does. This is the main reason for this refactoring.

Most developers prefer short, well-named methods for several reasons. First, it increases the chances that other methods can use a method when the method is finely grained. Second, it allows the higher-level methods to read more like a series of comments. Overriding also is easier when the methods are finely grained.

Besides eliminating rough edges in your code, extracting methods is also a step in many other refactoring approaches.

Benefits

- More readable code! Be sure to give the new method a name that describes the method's purpose: createOrder(), renderCustomerInfo(), etc.
- Less code duplication. Often the code that's found in a method can be reused in other places in your program. So you can replace duplicates with calls to your new method.
- Isolates independent parts of code, meaning that errors are less likely (such as if the wrong variable is modified).

How to Refactor

- Create a new method and name it in a way that makes its purpose self-evident.
- Copy the relevant code fragment to your new method. Delete the fragment from its old location and put a call for the new method there instead.
- Find all variables used in this code fragment. If they're declared inside the fragment and not used outside of it, simply leave them unchanged—they'll become local variables for the new method.

- If the variables are declared prior to the code that you're extracting, you will need to pass these variables to the parameters of your new method in order to use the values previously contained in them.
- If you see that a local variable changes in your extracted code in some way, this may mean that this changed value will be needed later in your main method. Double-check! And if this is indeed the case, return the value of this variable to the main method to keep everything functioning.

1.1.2 Inline Method

Problem

When a method body is more obvious than the method itself, use this technique.

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

Solution

Replace calls to the method with the method's content and delete the method itself.

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

Why Refactor

A method simply delegates to another method. In itself, this delegation is no problem. But when there are many such methods, they become a confusing tangle that's hard to sort through.

Often methods aren't too short originally, but become that way as changes are made to the program. So don't be shy about getting rid of methods that have outlived their use.

Benefits

By minimizing the number of unneeded methods, you make the code more straightforward.

How to Refactor

- Make sure that the method isn't redefined in subclasses. If the method is redefined, refrain from this technique.
- Find all calls to the method. Replace these calls with the content of the method.
- Delete the method.

1.1.3 Extract Variable

Problem

You have an expression that's hard to understand.

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

Solution

Place the result of the expression or its parts in separate variables that are self-explanatory.

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

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

Why Refactor

The main reason for extracting variables is to make a complex expression more understandable, by dividing it into its intermediate parts. These could be:

- Condition of the if() operator or a part of the ?: operator in C-based languages
- A long arithmetic expression without intermediate results
- Long multipart lines

Benefits
More readable code! Try to give the extracted variables good names that announce the variable's purpose loud and clear. More readability, fewer long-winded comments. Go for names like customerTaxValue, cityUnemploymentRate, clientSalutationString, etc.
Drawbacks

- More variables are present in your code. But this is counterbalanced by the ease of reading your code.
- When refactoring conditional expressions, remember that the compiler will most likely optimize it to minimize the amount of calculations needed to establish the resulting value.

How to Refactor

- Insert a new line before the relevant expression and declare a new variable there. Assign part of the complex expression to this variable.
- Replace that part of the expression with the new variable.
- Repeat the process for all complex parts of the expression.

1.2 Moving Features between Objects

Even if you have distributed functionality among different classes in a less-than-perfect way, there's still hope.

1.2.1 Move Method

Problem
A method is used more in another class than in its own class.
Solution
Create a new method in the class that uses the method the most, then move code from the old method to there. Turn the code of the original method into a reference to the new method in the other class or else remove it entirely.
Why Refactor

- You want to move a method to a class that contains most of the data used by the method. This makes classes more internally coherent.

- You want to move a method in order to reduce or eliminate the dependency of the class calling the method on the class in which it's located. This can be useful if the calling class is already dependent on the class to which you're planning to move the method. This reduces dependency between classes.

How to Refactor

- Verify all features used by the old method in its class. It may be a good idea to move them as well. As a rule, if a feature is used only by the method under consideration, you should certainly move the feature to it. If the feature is used by other methods too, you should move these methods as well. Sometimes it's much easier to move a large number of methods than to set up relationships between them in different classes.
- Make sure that the method isn't declared in superclasses and subclasses. If this is the case, you will either have to refrain from moving or else implement a kind of polymorphism in the recipient class in order to ensure varying functionality of a method split up among donor classes.
- Declare the new method in the recipient class. You may want to give a new name for the method that's more appropriate for it in the new class.
- Take a look: can you delete the old method entirely? If so, place a reference to the new method in all places that use the old one.

1.2.2 Move Field

Problem
A field is used more in another class than in its own class.
Solution
Create a field in a new class and redirect all users of the old field to it.
Why Refactor
Deciding which class to leave the field in can be tough. Here is our rule of thumb: put a field in the same place as the methods that use it (or else where most of these methods are). This rule will help in other cases when a field is simply located in the wrong place.
How to Refactor

- If the field is public, refactoring will be much easier if you make the field private and provide public access methods.

- Create the same field with access methods in the recipient class.
- Decide how you will refer to the recipient class. You may already have a field or method that returns the appropriate object; if not, you will need to write a new method or field to store the object of the recipient class.
- Replace all references to the old field with appropriate calls to methods in the recipient class. If .
- the field isn't private, take care of this in the superclass and subclasses.
- Delete the field in the original class.

1.2.3 Extract Class

Problem
When one class does the work of two, awkwardness results.
Solution
Instead, create a new class and place the fields and methods responsible for the relevant functionality in it.
Why Refactor
Classes always start out clear and easy to understand. They do their job and mind their own business as it were, without butting into the work of other classes. But as the program expands, a method is added and then a field... and eventually, some classes are performing more responsibilities than ever envisioned.
Benefits
This refactoring method will help maintain adherence to the Single Responsibility Principle. The code of your classes will be more obvious and understandable. Single-responsibility classes are more reliable and tolerant of changes. For example, say that you have a class responsible for ten different things. When you change this class to make it better for one thing, you risk breaking it for the nine others.
How to Refactor

- Create a new class to contain the relevant functionality.
- Create a relationship between the old class and the new one. Optimally, this relationship is unidirectional; this allows reusing the second class without any issues.

- Use Move Field and Move Method for each field and method that you have decided to move to the new class. For methods, start with private ones in order to reduce the risk of making a large number of errors. Try to relocate just a little bit at a time and test the results after each move, in order to avoid a pileup of error-fixing at the very end.
- Also give thought to accessibility to the new class from the outside. You can hide the class from the client entirely by making it private, managing it via the fields from the old class. Alternatively, you can make it a public one by allowing the client to change values directly.

1.3 Organizing Data

1.3.1 Replace Data Value with Object

Problem
A class (or group of classes) contains a data field. The field has its own behavior and associated data.
Solution
Create a new class, place the old field and its behavior in the class, and store the object of the class in the original class.
Why Refactor
This refactoring is basically a special case of Extract Class. What makes it different is the cause of the refactoring.

In Extract Class, we have a single class that's responsible for different things and we want to split up its responsibilities.

With the replacement of a data value with an object, we have a primitive field (number, string, etc.) that's no longer so simple due to the growth of the program and now has associated data and behaviors. On the one hand, there's nothing scary about these fields in and of themselves. However, this field-and-behavior family can be present in several classes simultaneously, creating duplicate code.
Benefits
Improves relatedness inside classes. Data and the relevant behaviors are inside a single class.
How to Refactor

- Create a new class and copy your field and relevant getter to it. In addition, create a constructor that accepts the simple value of the field. This class won't have a setter since each new field value that's sent to the original class will create a new value object.

- In the original class, change the field type to the new class.
- In the getter in the original class, invoke the getter of the associated object.
- In the setter, create a new value object. You may need to also create a new object in the constructor if initial values had been set there for the field previously.

1.3.2 Self Encapsulate Field

Problem You use direct access to private fields inside a class.

```
class Range {
  private int low, high;
  boolean includes(int arg) {
    return arg >= low && arg <= high;
  }
}
```

Solution

Create a getter and setter for the field, and use only them for accessing the field.

```
class Range {
  private int low, high;
  boolean includes(int arg) {
    return arg >= getLow() && arg <= getHigh();
  }
  int getLow() {
    return low;
  }
  int getHigh() {
    return high;
  }
}
```

Why Refactor

Sometimes directly accessing a private field inside a class just isn't flexible enough. You want to be able to initiate a field value when the first query is made or perform certain operations on new values of the field when they're assigned, or maybe do all this in various ways in subclasses.

Benefits

- Indirect access to fields is when a field is acted on via access methods (getters and setters). This approach is much more flexible than direct access to fields.

  - First, you can perform complex operations when data in the field is set or received. Lazy initialization and validation of field values are easily implemented inside field getters and setters.
  - Second and more crucially, you can redefine getters and setters in subclasses.

- You have the option of not implementing a setter for a field at all. The field value will be specified only in the constructor, thus making the field unchangeable throughout the entire object lifespan.

Drawbacks

When direct access to fields is used, code looks simpler and more presentable, although flexibility is diminished.

How to Refactor

- Create a getting and setting method for the field.
- Find all references to the field and replace them with a getting or setting method.
- Make the field private.
- Double check that you have caught all references

1.3.3 Encapsulate Collection

Problem

A class contains a collection field and a simple getter and setter for working with the collection.

```
class Team {
    public List<String> members = new ArrayList<>(); // Direct access
   to the collection
}
public class Main {
    public static void main(String[] args) {
        Team team = new Team();
        team.members.add("Alice"); // Direct modification
        team.members.add("Bob");
        team.members.remove("Alice"); // Can modify freely
        System.out.println(team.members); // No control over how it's
   accessed
    }
}
```

Solution

Make the getter-returned value read-only and create methods for adding/deleting elements of the collection.

```
class Team {
    private List<String> members = new ArrayList<>();
    public List<String> getMembers() {
        return Collections.unmodifiableList(members);
    }
    public void addMember(String member) {
        if (member != null && !member.isEmpty()) {
            members.add(member);
        }
    }
    public void removeMember(String member) {
        members.remove(member);
    }
}
```

Why Refactor

Often a class contains a collection of instances. This collection might be an array, list, set, or vector. Such cases often have the usual getter and setter for the collection. However, collections should use a protocol slightly different from that for other kinds of data. The getter should not return the collection object itself, because that allows clients to manipulate the contents of the collection without the owning class's knowing what is going on. It also reveals too much to clients about the object's internal data structures. A getter for a multivalued attribute should return something that prevents manipulation of the collection and hides unnecessary details about its structure. How you do this varies depending on the version of Java you are using. In addition, there should not be a setter for collection; rather, there should be operations to add and remove elements. This gives the owning object control over adding and removing elements from the collection. With this protocol, the collection is properly encapsulated, which reduces the coupling of the owning class to its clients.

How to Refactor

- Create methods for adding and deleting collection elements. They must accept collection elements in their parameters.
- Assign an empty collection to the field as the initial value if this isn't done in the class constructor.

- Find the calls of the collection field setter. Change the setter so that it uses operations for adding and deleting elements, or make these operations call client code.
- Find all calls of the collection getter after which the collection is changed. Change the code so that it uses your new methods for adding and deleting elements from the collection.
- Change the getter so that it returns a read-only representation of the collection.
- Inspect the client code that uses the collection for code that would look better inside of the collection class itself.

1.3.4 Replace Magic Literal

Problem

Your code uses a number that has a certain meaning to it.

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

Solution

Replace this number with a constant that has a human-readable name explaining the meaning of the number.

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

1.4 Simplifying Conditional Expressions

Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.

1.4.1 Decompose Conditional

Problem

You have a complex conditional (if-then/else or switch).

```
if (date.before(SUMMER_START) || date.after(SUMMER_END)) {
  charge = quantity * winterRate + winterServiceCharge;
```

```
}
else {
  charge = quantity * summerRate;
}
```

Solution

Decompose the complicated parts of the conditional into separate methods: the condition, then and else.

```
if (isSummer(date)) {
  charge = summerCharge(quantity);
}
else {
  charge = winterCharge(quantity);
}
```

Why Refactor

The longer a piece of code is, the harder it's to understand. Things become even more hard to understand when the code is filled with conditions:

- While you're busy figuring out what the code in the then block does, you forget what the relevant condition was.
- While you're busy parsing else, you forget what the code in then does.

Benefits

By extracting conditional code to clearly named methods, you make life easier for the person who'll be maintaining the code later.

This refactoring technique is also applicable for short expressions in conditions. The string isSalaryDay() is much prettier and more descriptive than code for comparing dates.

Mechanics

- Extract the condition into its own method.
- Extract the then part and the else part into their own methods.

1.4.2 Consolidate Conditional Expression

Problem

You have multiple conditionals that lead to the same result or action.

```
double disabilityAmount() {
  if (seniority < 2) {
```

```
    return 0;
  }
  if (monthsDisabled > 12) {
    return 0;
  }
  if (isPartTime) {
    return 0;
  }
  // Compute the disability amount.
}
```

Solution

Consolidate all these conditionals in a single expression.

```
double disabilityAmount() {
  if (isNotEligibleForDisability()) {
    return 0;
  }
  // Compute the disability amount.
}
```

Why Refactor

Your code contains many alternating operators that perform identical actions. It isn't clear why the operators are split up. The main purpose of consolidation is to extract the conditional to a separate method for greater clarity.

Benefits

Eliminates duplicate control flow code. Combining multiple conditionals that have the same "destination" helps to show that you're doing only one complicated check leading to one action. By consolidating all operators, you can now isolate this complex expression in a new method with a name that explains the conditional's purpose.

How to Refactor

- Consolidate the conditionals in a single expression by using and and or. As a general rule when consolidating:
  - Nested conditionals are joined using and.
  - Consecutive conditionals are joined with or.
- Perform Extract Method on the operator conditions and give the method a name that reflects the expression's purpose.

1.4.3 Replace Nested Conditional with Guard Clauses

Problem

You have a group of nested conditionals and it's hard to determine the normal flow of code execution.

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

Solution

Isolate all special checks and edge cases into separate clauses and place them before the main checks. Ideally, you should have a "flat" list of conditionals, one after the other.

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

Why Refactor

Spotting the "conditional from hell" is fairly easy. The indentations of each level of nestedness form an arrow, pointing to the right in the direction of pain and woe:

```
if () {
    if () {
        do {
            if () {
                if () {
                    if () {
                        ...
                    }
                }
                ...
            }
            ...
        }
        while ();
        ...
    }
    else {
        ...
    }
}
```

It's difficult to figure out what each conditional does and how, since the "normal" flow of code execution isn't immediately obvious. These conditionals indicate helter-skelter evolution, with each condition added as a stopgap measure without any thought paid to optimizing the overall structure.

To simplify the situation, isolate the special cases into separate conditions that immediately end execution and return a null value if the guard clauses are true. In effect, your mission here is to make the structure flat.

How to Refactor

- For each check put in the guard clause.
  - The guard clause either returns, or throws an exception.
- Compile and test after each check is replaced with a guard clause.
  - If all guard clauses yield the same result, use Consolidate Conditional Expressions.

1.4.4 Replace Conditional with Polymorphism

Problem

You have a conditional that performs various actions depending on object type or properties.

```
class Bird {
  // ...
  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");
  }
}
```

Solution

Create subclasses matching the branches of the conditional. In them, create a shared method and move code from the corresponding branch of the conditional to it. Then replace the conditional with the relevant method call. The result is that the proper implementation will be attained via polymorphism depending on the object class.

```
abstract class Bird {
  // ...
  abstract double getSpeed();
}
class European extends Bird {
  double getSpeed() {
    return getBaseSpeed();
  }
}
class African extends Bird {
  double getSpeed() {
    return getBaseSpeed() - getLoadFactor() * numberOfCoconuts;
  }
}
class NorwegianBlue extends Bird {
  double getSpeed() {
    return (isNailed) ? 0 : getBaseSpeed(voltage);
```

```
18    }
19 }
20 // Somewhere in client code
21 speed = bird.getSpeed();
```

Benefits

- This technique adheres to the Tell-Don't-Ask principle: instead of asking an object about its state and then performing actions based on this, it's much easier to simply tell the object what it needs to do and let it decide for itself how to do that.
- Removes duplicate code. You get rid of many almost identical conditionals.
- If you need to add a new execution variant, all you need to do is add a new subclass without touching the existing code (Open/Closed Principle).

Refactoring Steps

- If the conditional is in a method that performs other actions as well, perform Extract Method.
- For each hierarchy subclass, redefine the method that contains the conditional and copy the code of the corresponding conditional branch to that location.
- Delete this branch from the conditional.
- Repeat replacement until the conditional is empty. Then delete the conditional and declare the method abstract.

1.5 Simplifying Method Calls

These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes.

1.5.1 Rename Method

Problem
The name of a method doesn't explain what the method does.
Solution
Rename the method.
Why Refactor

Perhaps a method was poorly named from the very beginning—for example, someone created the method in a rush and didn't give proper care to naming it well.

Or perhaps the method was well named at first but as its functionality grew, the method name stopped being a good descriptor.

How to Refactor

- See whether the method is defined in a superclass or subclass. If so, you must repeat all steps in these classes too.
- The next method is important for maintaining the functionality of the program during the refactoring process. Create a new method with a new name. Copy the code of the old method to it. Delete all the code in the old method and, instead of it, insert a call for the new method.
- Find all references to the old method and replace them with references to the new one.
- Delete the old method. If the old method is part of a public interface, don't perform this step. Instead, mark the old method as deprecated.

1.5.2 Introduce Parameter Object

Problem

Your methods contain a repeating group of parameters.

Solution

Replace these parameters with an object.

Why Refactor

Identical groups of parameters are often encountered in multiple methods. This causes code duplication of both the parameters themselves and of related operations. By consolidating parameters in a single class, you can also move the methods for handling this data there as well, freeing the other methods from this code.

Benefits

More readable code. Instead of a hodgepodge of parameters, you see a single object with a comprehensible name. Identical groups of parameters scattered here and there create their own kind of code duplication: while identical code isn't being called, identical groups of parameters and arguments are constantly encountered.

How to Refactor

- Create a new class that will represent your group of parameters. Make the class immutable.

- In the method that you want to refactor, use Add Parameter, which is where your parameter object will be passed. In all method calls, pass the object created from old method parameters to this parameter.
- Now start deleting old parameters from the method one by one, replacing them in the code with fields of the parameter object. Test the program after each parameter replacement.
- When done, see whether there's any point in moving a part of the method (or sometimes even the whole method) to a parameter object class. If so, use Move Method or Extract Method.

1.5.3 Remove Setting Method

Problem
The value of a field should be set only when it's created, and not change at any time after that.
Solution
So remove methods that set the field's value.
Why Refactor You want to prevent any changes to the value of a field.
How to Refactor

- The value of a field should be changeable only in the constructor. If the constructor doesn't contain a parameter for setting the value, add one.
- Find all setter calls.
  - If a setter call is located right after a call for the constructor of the current class, move its argument to the constructor call and remove the setter.
  - Replace setter calls in the constructor with direct access to the field.
- Delete the setter.

1.6 Dealing with Generalization

Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.

1.6.1 Pull Up Constructor Body

Problem

Your subclasses have constructors with code that's mostly identical.

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

Solution

Create a superclass constructor and move the code that's the same in the subclasses to it. Call the superclass constructor in the subclass constructors.

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

Motivation

If you see subclass methods with common behavior, your first thought is to extract the common behavior into a method and pull it up into the superclass. With a constructor, however, the common behavior is often the construction. In this case you need a superclass constructor that is called by subclasses. In many cases this is the whole body of the constructor.

How to refactor

- Create a constructor in a superclass.
- Extract the common code from the beginning of the constructor of each subclass to the superclass constructor. Before doing so, try to move as much common code as possible to the beginning of the constructor.
- Place the call for the superclass constructor in the first line in the subclass constructors.

1.6.2 Push Down Method/Field

Problem
Is behavior/attribute implemented/used in a superclass used by only one (or a few) subclasses?
Solution
Move this method/Field to the subclasses.
Why Refactor
At first a certain method/field was meant to be universal for all classes but in reality is used in only one subclass. This situation can occur when planned features fail to materialize.

Such situations can also occur after partial extraction (or removal) of functionality from a class hierarchy, leaving a method/field that's used in only one subclass.

If you see that a method/field is needed by more than one subclass, but not all of them, it may be useful to create an intermediate subclass and move the method/field to it. This allows avoiding the code duplication that would result from pushing a method/field down to all subclasses.
Benefits
Improves class coherence. A method/field is located where you expect to see it.
How to Refactor

- Declare the method/field in a subclass and copy its code from the superclass.
- Remove the method/field from the superclass.
- Find all places where the method/field is used and verify that it's called from the necessary subclass.

1.6.3 Pull Up Method/Field

Problem
Two classes have the same field/method perform similar work.
Solution
Remove the field/method from subclasses and move it to the superclass.
Why Refactor
Subclasses grew and developed separately, causing identical (or nearly identical) fields and methods to appear.
Benefits

- Eliminates duplication of fields in subclasses.
- Eases subsequent relocation of duplicate methods, if they exist, from subclasses to a superclass.

How to Refactor: Pull up Field

- Inspect all uses of the candidate fields to ensure they are used in the same way.
- If the fields do not have the same name, rename the fields so that they have the name you want to use for the superclass field.
- Create a new field in the superclass.
  - If the fields are private, you will need to protect the superclass field so that the subclasses can refer to it.
- Delete the subclass fields.

How to Refactor: Pull up Method

- Inspect the methods to ensure they are identical.
  - If the methods look like they do the same thing but are not identical, use algorithm substitution on one of them to make them identical.
- If the methods have different signatures, change the signatures to the one you want to use in the superclass.
- Create a new method in the superclass, copy the body of one of the methods to it, adjust, and compile.
  - If you are in a strongly typed language and the method calls another method that is present on both subclasses but not the superclass, declare an abstract method on the superclass.
  - If the method uses a subclass field, use Pull Up Field Encapsulate Field or Self and declare and use an abstract getting method.
- Delete one subclass method.