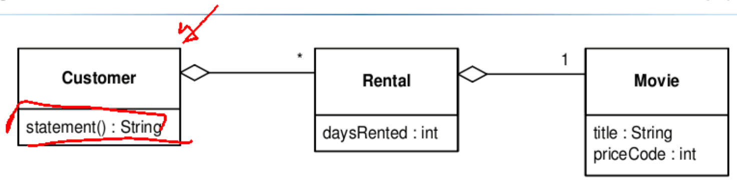
Script **generated by TTT**

Title:	Matthes: Soft-Arch (17.01.2012)
Date:	Tue Jan 17 18:15:49 CET 2012
Duration:	57:51 min
Pages:	35

Demo…



- The program is told which movies a customer rented and for how long.
- There are three kinds of movies: regular, children’s, and new releases.
- The statement method of Customer calculates and prints a statement of a customer’s charges at a video store.
- In addition to calculating charges, the statement also computes frequent renter points, which vary depending on whether the film is a new release.

The Initial statement() Method (1)

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        String result = "Rental Record for " + getName() + "\n";
        for (Rental each : rentals) {
            double thisAmount = 0;
            //determine amounts for each line
            switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2) {
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                }
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDREN:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3) {
                    thisAmount += (each.getDaysRented() - 3) * 1.5;
                }
                break;
            }

…
```

The Initial statement() Method (2)

```
…

            // add frequent renter points
            frequentRenterPoints++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) && each.getDaysRented() > 1) {
                frequentRenterPoints++;
            }
            // show figures for this rental
            result += "\t" + each.getMovie().getTitle() + "\t" + String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;

        }
        // add footer lines
        result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
        return result;
    }
```

Basic Rules for Refactoring

sebis

There are two changes to be made on the example:

- The statement should be formatted in HMTL
- The way the movies are classified should be changed.

(1) When you find you have to add a feature to a program, and the program's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.

(2) Before you start refactoring, check that you have a solid suite of tests. These tests must be self-checking.

(3) Refactoring changes the programs in small steps. If you make a mistake, it is easy to find the bug.

(4) Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Defining Refactoring

sebis

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.

The Two Hats

sebis

Using refactoring to develop software leads to a division of time into two distinct activities: **adding function** and **refactoring**.

- When you add function, you shouldn't be changing existing code. You add tests and get the tests to work.
- When you refactor, you make a point of not adding function; you only restructure the code. You don't add any tests.

As you develop software, you probably find yourself swapping hats frequently.

Why Should You Refactor? Refactoring Improves the Design of Software

sebis

Without refactoring, the design of software will decay: as people change code, the code loses its structure.

Loss of the structure of code has a cumulative effect:

- The harder it is to see the design in the code, the harder it is to preserve it, and the more rapidly it decays

An important aspect of improving design is to **eliminate duplicate code.**

- Reducing the amount of code does make a big difference in modification of the code. The more code there is, the harder it is to modify correctly. There's more code to understand.

Why Should You Refactor? Refactoring Makes Software Easier to Understand

sebis

- If the code isn't clear, it's an odor that needs to be removed by refactoring, not by deodorizing the code with a comment.
- Refactoring makes code less annoying.
- Refactor only what you truly understand!

Why Should You Refactor? Refactoring Helps You Program Faster

sebis

Refactoring helps you develop more code more quickly.

Without a good design, you can progress quickly for a while, but soon the poor design starts to slow you down.

- You spend time finding and fixing bugs instead of adding new function.
- Changes take longer as you try to understand the system and find the duplicated code.
- New features need more coding as you patch over a patch that patches a patch on the original code base.

Good design is essential for rapid software development.

When Should You Refactor?

sebis

The rule of three

- The first time you do something, you just do it.
- The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway.
- The third time you do something similar, you refactor.

Refactor when you add function

Refactor when you need to fix a bug

Refactor as you do a code review

Indirection and Refactoring

sebis

Most refactoring introduces more indirection into a program.

Drawback of indirection:

- More things to manage
- Can make a program harder to read as an object delegates to an object delegating to an object

But, indirection can pay for itself:

- To enable sharing of logic
- To explain intention and implementation separately

WHY → HOW

- WHAT

WHY?

Problems with Refactoring

Databases

- Most business applications are tightly coupled to the database schema that supports them
- Data migration can be a long and fraught task

Changing interfaces

- There is a problem if the interface is being used by code that you cannot find and change (this happens in the case of published interfaces)
- Once you publish an interface, you can no longer safely change it and just edit the callers → what do you do about refactorings that change published interfaces?
- Don't publish interfaces prematurely. Modify your code ownership policies to allow people to change other people's code in order to support an interface change.

When shouldn't you refactor?

- When you should rewrite from scratch instead
  - The code does not work correctly and cannot be stabilized
- When you are close to a deadline

Refactoring and Performance

To make the software easier to understand, you often make changes that will cause the program to run more slowly.

Refactoring certainly will make software go more slowly, but it also makes the software more amenable to performance tuning.

The secret to fast software, is to write tunable software first and then to tune it for sufficient speed.

Observation: most programs waste most of their time in a small fraction of the code.

Build your program in a well-factored manner without paying attention to performance until you begin a performance optimization stage, usually fairly late in development.

Code Smells

The most common design problems result from code that

- Is duplicated
- Is unclear
- Is complicated

Many programmers find this list to be too vague; they don't know how to spot duplication in code that isn't outwardly the same, they aren't sure how to tell when code is clearly communicating its intent, and they don't know how to distinguish simple code from complicated code. A

"Bad smells" provide additional guidance for identifying design problems.

Code smells target problems that occur everywhere: in methods, classes, hierarchies, packages (namespaces, modules), and entire systems.

The names of the smells provide a rich and colorful vocabulary with which programmers may rapidly communicate about design problems.

Code smells are no precise criteria for when a refactoring is overdue; no set of metrics rivals informed human intuition.

A Catalog of Code Smells

Duplicated Code

- The same structure in more than one place
- The simplest duplicated code problem is when you have the same expression in two methods of the same class

Long Method

Large Class

Long Parameter List

Divergent Change

- A single functional change requires multiple changes at not obviously related code locations

Format of the Refactorings

Each refactoring has five parts, as follows:

- The **name** is important to building a vocabulary of refactorings.
- The name is followed by a short **summary** of the situation in which you need the refactoring and a summary of what the refactoring does.
- The **motivation** describes why the refactoring should be done and describes circumstances in which it shouldn't be done
- The **mechanics** are a concise, step-by-step description of how to carry out the refactoring
- The **examples** show a very simple use of the refactoring to illustrate how it works

Format of the Refactorings

Each refactoring has five parts, as follows:

- The **name** is important to building a vocabulary of refactorings.
- The name is followed by a short **summary** of the situation in which you need the refactoring and a summary of what the refactoring does.
- The **motivation** describes why the refactoring should be done and describes circumstances in which it shouldn't be done
- The **mechanics** are a concise, step-by-step description of how to carry out the refactoring
- The **examples** show a very simple use of the refactoring to illustrate how it works

Extract Method (1)

You have a code fragment that can be grouped together.
Turn the fragment into a method whose name explains the purpose of the method.

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

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

$\longrightarrow$

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

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

Motivation

- Is one of the most common refactorings
- Leads to short, well-named methods
  - Increases the chance that other methods can use a method
  - Allows the higher-level methods to read more like a series of comments
  - Overriding is easier when the methods are finely grained
- The key is the semantic distance between the method name and the method body

Extract Method (2)

sebis

Mechanics

- Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it)
  - If the code you want to extract is very simple, you should extract it if the name of the new method will reveal the intention of the code in a better way. If you can't come up with a more meaningful name, don't extract the code.
- Copy the extracted code from the source method into the new target method.
- References that are local in scope to the source method will become parameters of the target method

Replace Temp With Query (1)

sebis

You are using a temporary variable to hold the result of an expression.

Extract the expression into a method. Replace all references to the temp with the new method. The new method can then be used in other methods.

```
  double basePrice = _quantity *
_itemPrice;
  if(basePrice > 1000) {
    return basePrice * 0.95;
  } else {
    return basePrice * 0.98;
  }
```

→

```
  if(basePrice() > 1000) {
    return basePrice() * 0.95;
  } else {
    return basePrice() * 0.98;
  }
…
double basePrice() {
  return _quantity * _itemPrice;
}
```

Motivation

- The problem with temps is that they are temporary and local.
- By replacing the temp with a query method, any method in the class can get at the information
- Replace Temp With Query if often a vital step before Extract Method

Replace Temp With Query (2)

sebis

Mechanics

- Look for a temporary variable that is assigned to once
- Declare the temp as final
- Compile
- Extract the right-hand side of the assignment into a method
  - Initially mark the method as private. You may find more use for it later, but you can easily relax the protection later
  - Ensure the extracted method is free of side effects, that is, it does not modify any object
- Compile and test
- *Inline temp* on the temp

Replace Temp With Query (1)

sebis

You are using a temporary variable to hold the result of an expression.

Extract the expression into a method. Replace all references to the temp with the new method. The new method can then be used in other methods.

```
  double basePrice = _quantity *
_itemPrice;
  if(basePrice > 1000) {
    return basePrice * 0.95;
  } else {
    return basePrice * 0.98;
  }
```

→

```
  if(basePrice() > 1000) {
    return basePrice() * 0.95;
  } else {
    return basePrice() * 0.98;
  }
…          private
double basePrice() {
  return _quantity * _itemPrice;
}
```

Motivation

- The problem with temps is that they are temporary and local.
- By replacing the temp with a query method, any method in the class can get at the information
- Replace Temp With Query if often a vital step before Extract Method

Replace Temp With Query (2)

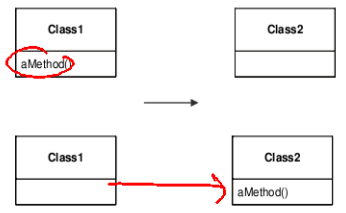
Mechanics

- Look for a temporary variable that is assigned to once
- Declare the temp as final
- Compile
- Extract the right-hand side of the assignment into a method
  - Initially mark the method as private. You may find more use for it later, but you can easily relax the protection later
  - Ensure the extracted method is free of side effects, that is, it does not modify any object
- Compile and test
- *Inline temp* on the temp

Move Method (1)

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

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



Motivation

- Move methods, when classes have too much behavior or when classes are collaborating too much and are too highly coupled.
- Moving methods can make the classes simpler and they end up being a more crisp implementation of a set of responsibilities

Replace Temp With Query (2)

Mechanics

- Look for a temporary variable that is assigned to once
- Declare the temp as final
- Compile
- Extract the right-hand side of the assignment into a method
  - Initially mark the method as private. You may find more use for it later, but you can easily relax the protection later
  - Ensure the extracted method is free of side effects, that is, it does not modify any object
- Compile and test
- *Inline temp* on the temp

Software Architectures: 5. Evolution © sebis 31

The Initial statement() Method (1)

```
public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        String result = "Rental Record for " + getName() + "\n";
        for (Rental each : rentals) {
            double thisAmount = 0;
            //determine amounts for each line
            switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2) {
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                }
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDREN:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3) {
                    thisAmount += (each.getDaysRented() - 3) * 1.5;
                }
                break;
            }

...
```

Move Method (2)

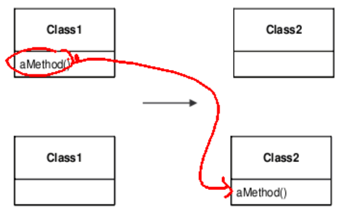
Mechanics

- Examine all features used by the source method that are defined on the source class. Consider whether they also should be moved.
  - If a feature is used only by the method you are about to move, you might as well move it, too. If the feature is used by other methods, consider moving them as well. Sometimes it is easier to move a clutch of methods than to move them one at a time.
- Check the sub- and superclasses of the source class for other declarations of the method
- Declare the method in the target class
- Copy the code from the source method to the target. Adjust the method to make it work in its new home.
- Compile the target class
- Determine how to reference the correct target object from the source.
- Turn the source method into a delegating method
- Compile and test
- Decide whether to remove the source method or retain it as a delegating method

Move Method (1)

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

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



Motivation

- Move methods, when classes have too much behavior or when classes are collaborating too much and are too highly coupled.
- Moving methods can make the classes simpler and they end up being a more crisp implementation of a set of responsibilities

Move Method (2)

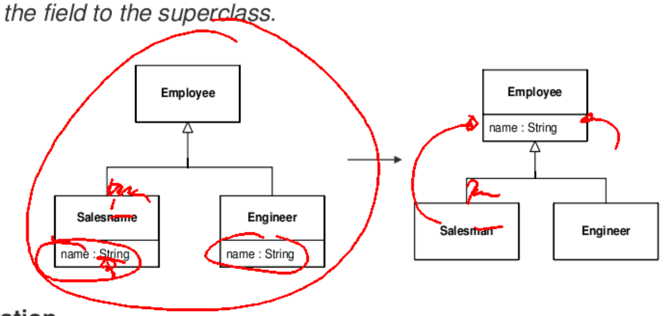
Mechanics

- Examine all features used by the source method that are defined on the source class. Consider whether they also should be moved.
  - If a feature is used only by the method you are about to move, you might as well move it, too. If the feature is used by other methods, consider moving them as well. Sometimes it is easier to move a clutch of methods than to move them one at a time.
- Check the sub- and superclasses of the source class for other declarations of the method
- Declare the method in the target class
- Copy the code from the source method to the target. Adjust the method to make it work in its new home.
- Compile the target class
- Determine how to reference the correct target object from the source.
- Turn the source method into a delegating method
- Compile and test
- Decide whether to remove the source method or retain it as a delegating method

Pull Up Field (1)

Two subclasses have the same field.

Move the field to the superclass.



Motivation

- If subclasses are developed independently, or combined through refactoring, you often find that they duplicate features. If they are used in a similar way, you can generalize them.
- Pulling up a field reduces duplication of the data declaration and behavior.

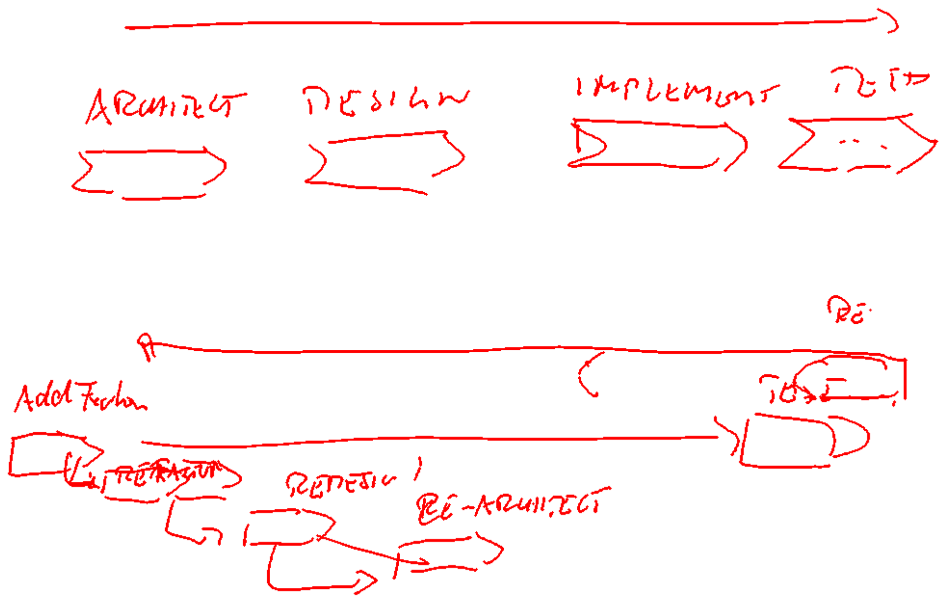
Pull Up Field (2)

sebis

Mechanics

- 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
- Compile and test
- 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
- Compile and test

End of presentation. Click to exit.


ARCHITECT
DESIGN
IMPLEMENT
TEST
RE-
TEST
Add Feature
REFACTOR
REDESIGN
RE-ARCHITECT