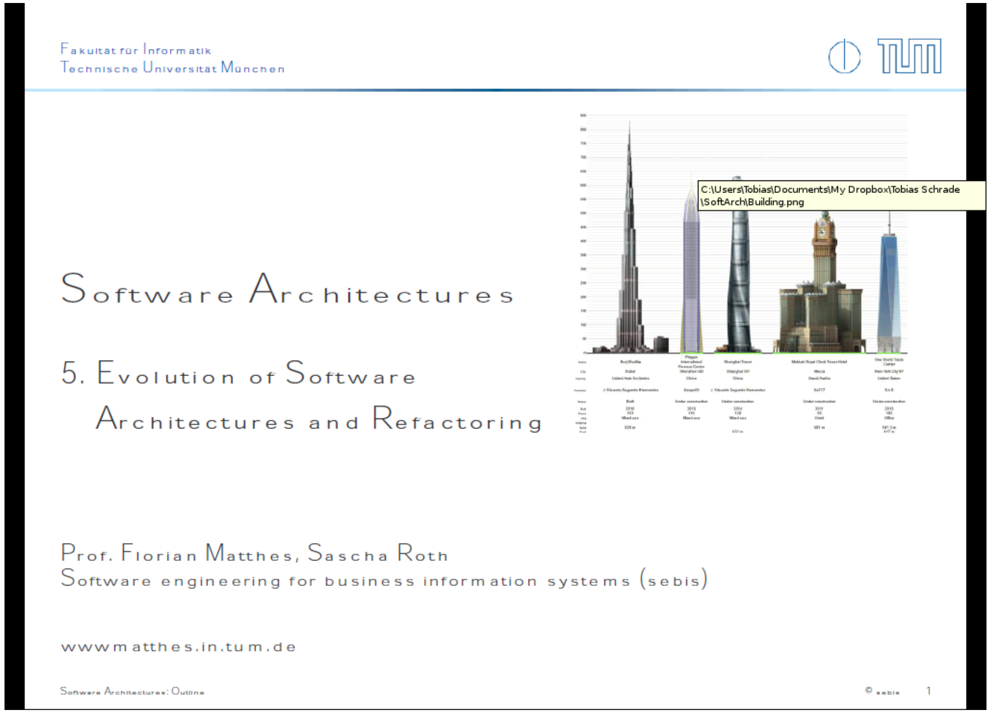
Script **generated by TTT**

Title: Matthes: Soft-Arch (10.01.2012)

Date: Tue Jan 10 19:19:38 CET 2012

Duration: 16:17 min

Pages: 10



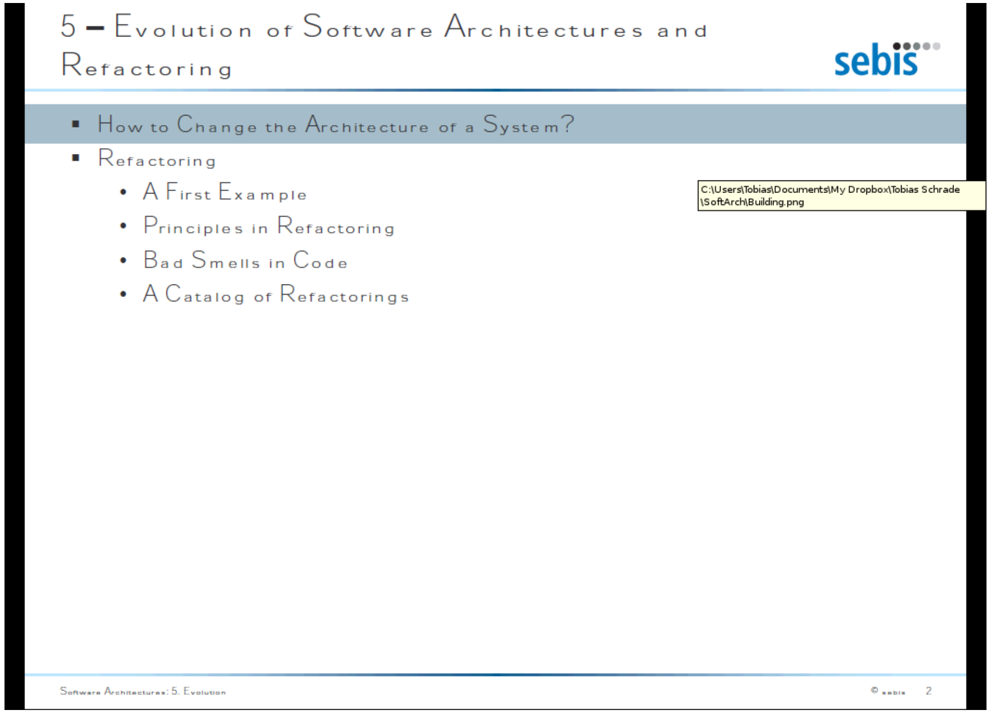
C:\Users\Tobias\Documents\My Dropbox\Tobias Schrade \SoftArch\Building.png

Software Architectures

5. Evolution of Software Architectures and Refactoring

Prof. Florian Matthes, Sascha Roth
Software engineering for business information systems (sebis)

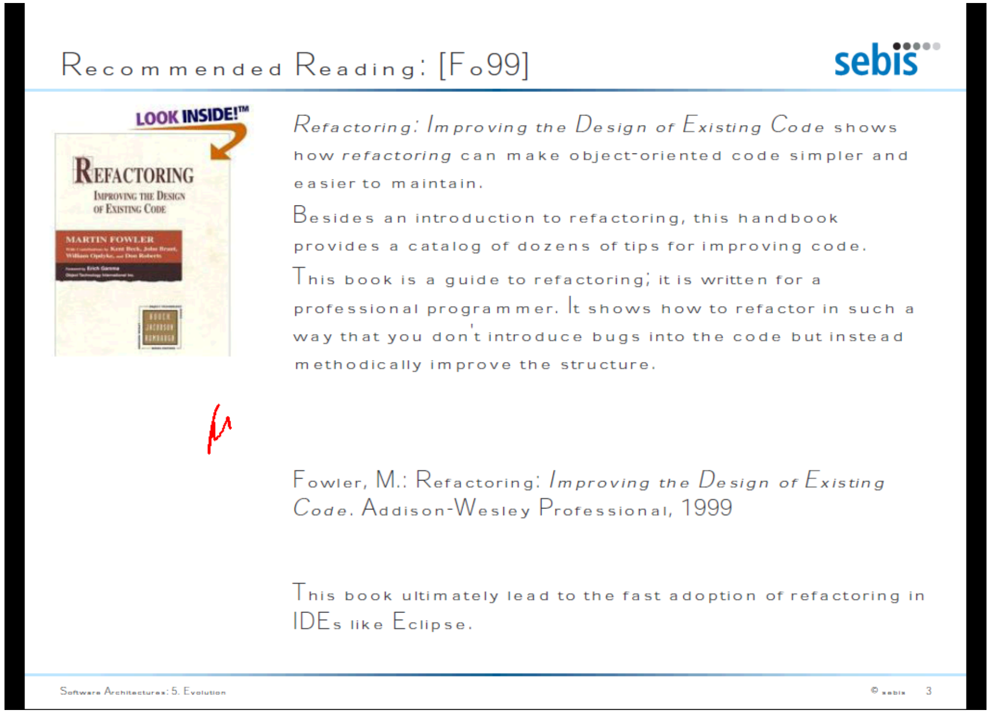
www.matthes.in.tum.de



5 – Evolution of Software Architectures and Refactoring

- How to Change the Architecture of a System?
- Refactoring
  - A First Example
  - Principles in Refactoring
  - Bad Smells in Code
  - A Catalog of Refactorings

C:\Users\Tobias\Documents\My Dropbox\Tobias Schrade \SoftArch\Building.png



Recommended Reading: [Fo99]

Refactoring: Improving the Design of Existing Code shows how *refactoring* can make object-oriented code simpler and easier to maintain.

Besides an introduction to refactoring, this handbook provides a catalog of dozens of tips for improving code.

This book is a guide to refactoring; it is written for a professional programmer. It shows how to refactor in such a way that you don't introduce bugs into the code but instead methodically improve the structure.

Fowler, M.: Refactoring: *Improving the Design of Existing Code*. Addison-Wesley Professional, 1999

This book ultimately lead to the fast adoption of refactoring in IDEs like Eclipse.

Evolution of a Software System without Architectural Changes

Architectural changes

- are difficult and expensive – even in the design phase
- initially decrease the quality of the system
- may require retraining of developers and updates of neighboring systems

This often leads to the “Piggyback” syndrome which tries to avoid architectural changes by quietly violating architectural rules:

- Functionalities are introduced into the system, often by bypassing the interfaces which should be used, but are not totally adequate.
- Encapsulation is violated.
- Parts of the code are not used anymore or are duplicated.
- Code is not written as compact as it could be possible.

→ Changes become more and more expensive and risky.

The maintainability and adaptability of a “piggyback” system at some point “hits a wall of complexity”.

In the worst case, these systems have to be replaced by completely new systems.

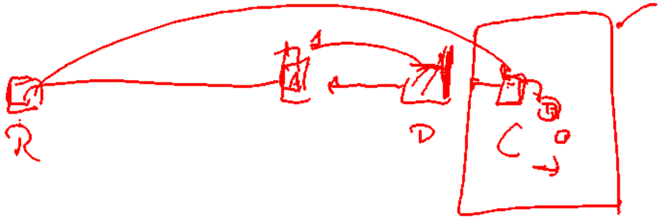
[RH06]

Keeping a Software System Healthy and Alive

Goal: keeping requirements, architecture, design, and implementation aligned through continuous stepwise (managed) evolution

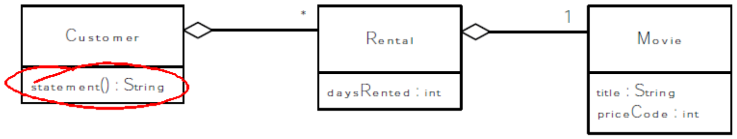
Principles

- Unused or dysfunctional code is replaced immediately.
- Refactor the code and the interfaces as soon as the program becomes difficult to change.
- If the refactorings do not match the existing architecture, consider architectural changes which have to be consistent with strategic business requirements.
- The architecture is simplified where this is possible.



[RH06]

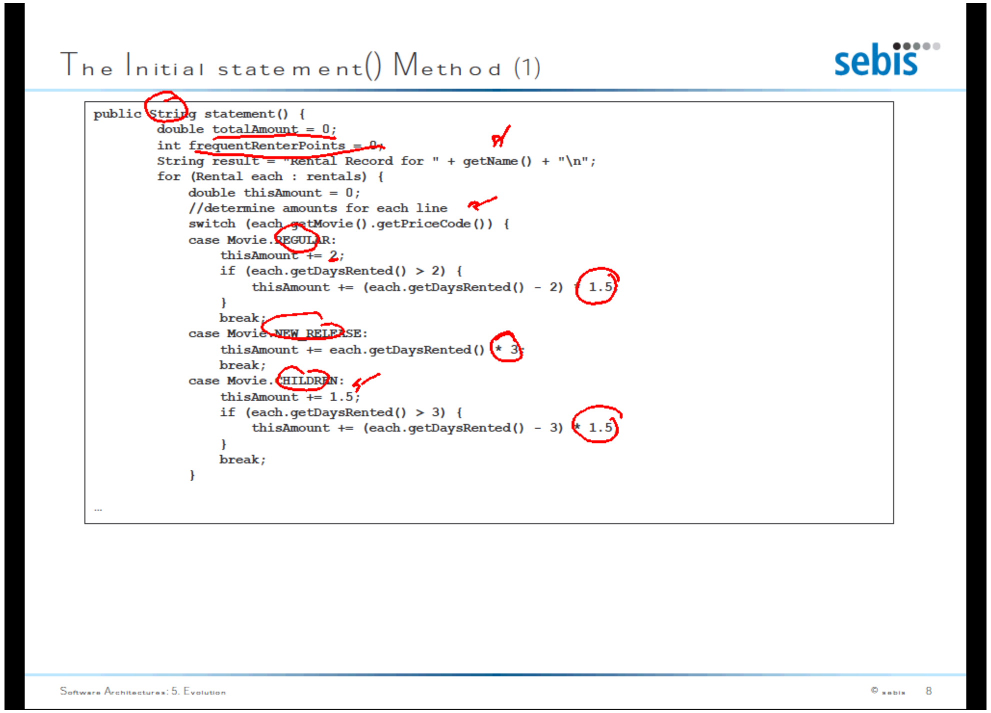
Demo…



- 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.

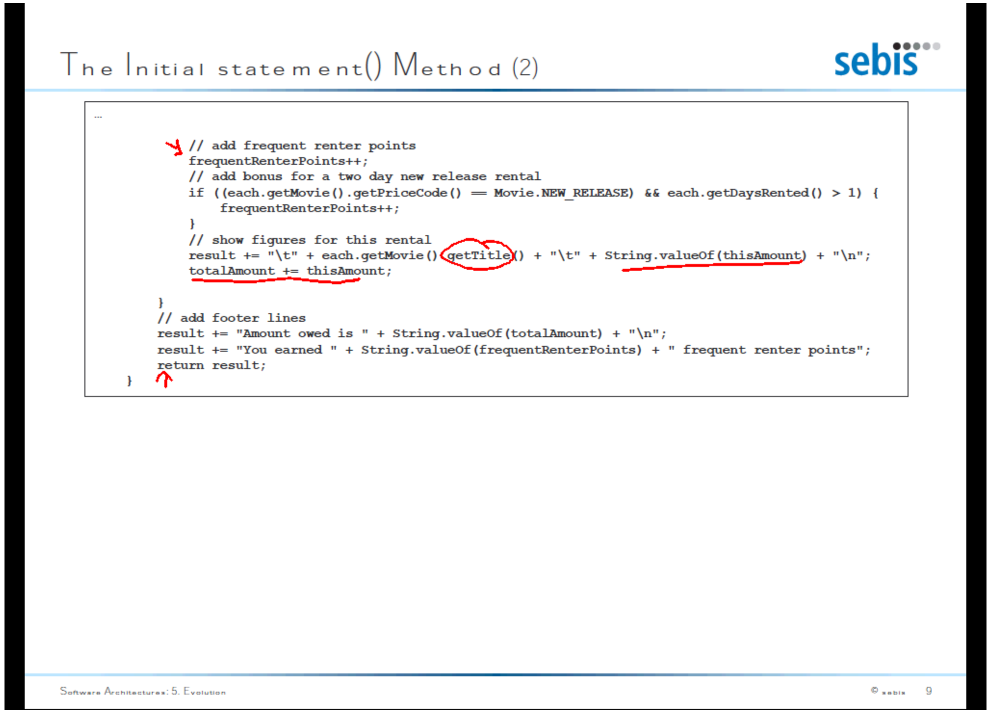
The Initial statement() Method (1)

```
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;
            }

…
```



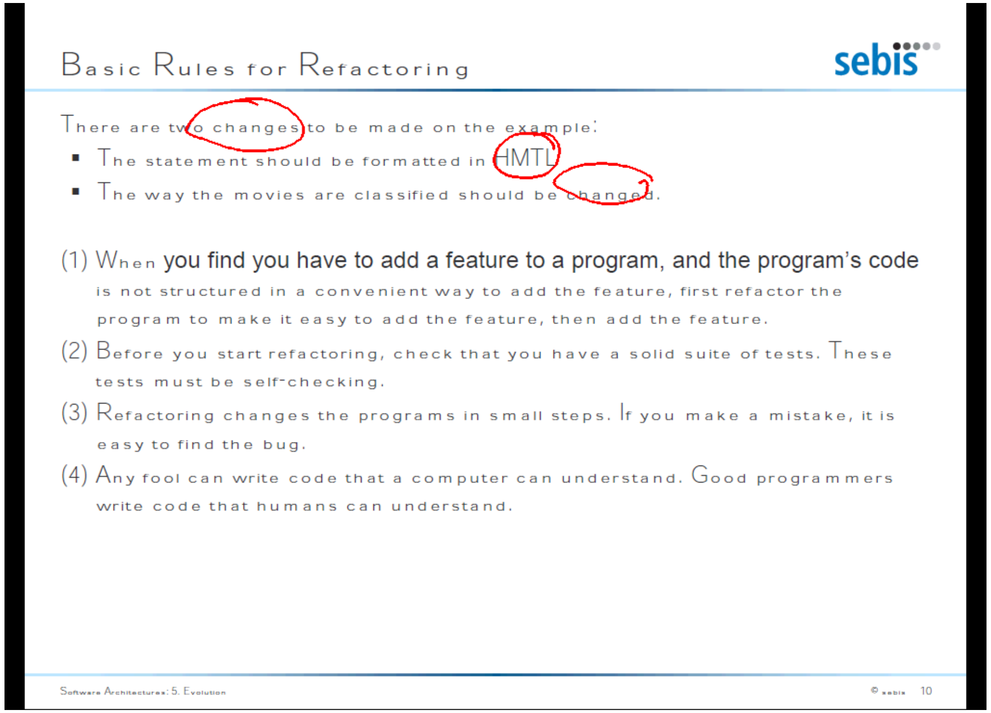
The Initial statement() Method (1)

```
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

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.