nagarro

04

Refactoring

Costs of Bad and Messy Code

- Slows down the team productivity
- Takes out the fun of the work
  - Coding becomes boring
- Difficult to understand
  - High learning curve
- Increases development costs
- Very hard / impossible to estimate

Why Is It Important to Take Care of Your Code?

- Most of the time developers are reading code
  - Adapting code to fix bugs and new requirements
- Your code will be read by people
  - The compiler doesn‘t care
- It slows down one person, but speeds up a team
  - ...and sometimes yourself, too
- Quality becomes a shared asset of the team

Refactoring – Definition

„Refactoring is a change
made to the internal structure of software
to make it easier to understand and
cheaper to modify
without changing its observable behaviour”
Martin Fowler

What is Refactoring?

Modifying software to improve its internal structure/design

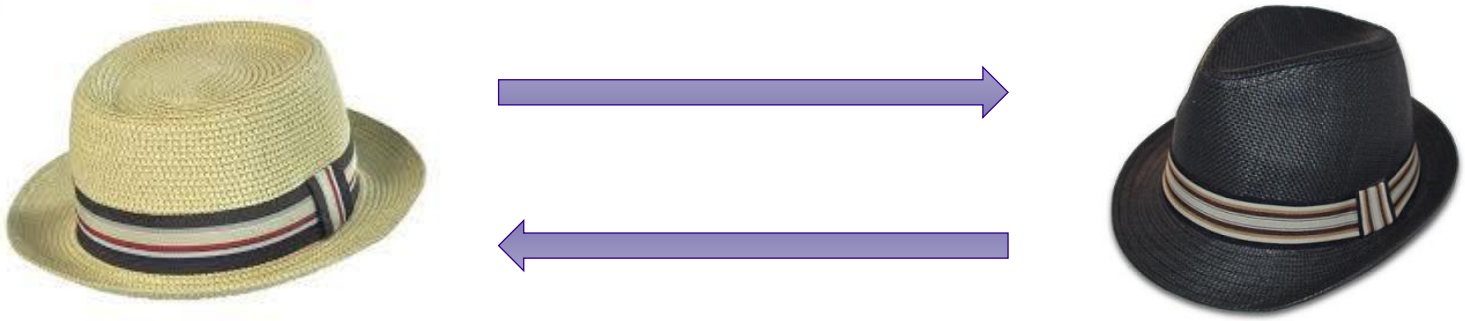
- Makes software easier to understand and maintain
- Helps finding bugs
- Helps program faster

Does not change the external behaviour

When to Refactor?

- Before and/or after you
  - add new functionality,
  - find bad code (code smells) or
  - need to fix a bug
- As part of the
  - routine (e.g. TDD) or
  - code review

A Programmer Wears Two Hats...



Adding functionality hat

- Extend functionality
- Write tests
- Execute tests

Refactoring hat

- Restructure your code
- Don‘t add new functionality
- Don‘t implement new tests

Change between the two hats regularly!

How to Refactor? - Safety First!

- Before you start, make sure you have a set of solid tests
- Test continously!
  - Automated tests & TDD wherever possible
- Refactor in **small** steps
  - Larger refactoring patterns can often be applied by using smaller refactoring techniques in the right order
- Use known refactoring methods
  - Martin Fowler: http://www.refactoring.com/catalog/index.html
- Use tools!
  - IDEs support basic refactoring methods
  - Use tools as far as you can go safe changes!
  - Only do manual refactoring when the limits of the tool have been reached

Exercise – Common Refactorings?

Which refactoring techniques do you know?

The trainer will guide you through this brainstorming session. He will stick your notes on the board and presents ask you to explain the technique.

Common Refactorings

- Extract Method / Class / Local Variable / Constant
- Rename Method / ...
- Move Method / Variable
- Introduce Parameter Object
- Form Template Method
- Replace Data with Value Object
- Replace Nested Conditional with Guard Clauses
- ... and many more: http://www.refactoring.com/catalog/index.html

Refactoring Method: Extract Method

- You have a code fragment that can be grouped together

- Extract that fragment into a method whose name explains the purpose of the method

Extract Method - Example

Before refactoring

```
public void PrintDebts (double amount)
{
    PrintBanner();

    // print debitor details
    Console.WriteLine("name: " + this.name);
    Console.WriteLine("amount: " + amount);
}
```

After refactoring

```
public void PrintDebts (double amount)
{
    PrintBanner();
    PrintDebitorDetails(amount);
}

private void PrintDebitorDetails (double amount)
{
    Console.WriteLine("name: " + this.name);
    Console.WriteLine("amount: " + amount);
}
```

Refactoring Method: Introduce Parameter Object

- You have a group of parameters that naturally belong together

- You have a group of parameters that naturally belong together

Introduce Parameter Object - Example

Before refactoring

```
public bool IsCandidateRejected (int englishScore, int scienceScore,
                                 int socialStudiesScore, int mathScore)
{
    // ...
}
```

After refactoring

```
public bool IsCandidateRejected (ExamResult result)
{
    // ...
}
```

Refactoring Method: Replace Nested Condition with Guard Clauses

- A method has conditional behavior that does not make clear what the normal path of execution is

- Use Guard Clauses for all the special cases

Replace Nested Condition with Guard Clauses - Example

```
public double GetTotalAmount (int orderId)
{
    if (orderId > 0 )
    {
      Order order = OrderRepo.FindById(orderId);
      if (order != null)
      {
        double totalAmount = 0.0;
        foreach (OrderItem item in order.getItems())
        {
            totalAmount += item.GetPrice() *
                           item.GetQuantity();
        }
        return totalAmount;
      }
      else
      {
         throw new OrderNotFoundException(...);
      }
    }
    else
    {
      throw new IllegalArgumentException(...);
    }
}
```

Replace Nested Condition with Guard Clauses - Example

Before refactoring

```
public double GetTotalAmount (int orderId)
{
    if (orderId > 0 )
    {
      Order order = OrderRepo.FindById(orderId);
      if (order != null
      {
        double totalAmount = 0.0;
        foreach (OrderItem item in order.getItems())
        {
            totalAmount += item.GetPrice() *
                           item.GetQuantity();
        }
        return totalAmount;
      }
      else
      {
         throw new OrderNotFoundException(...);
      }
    }
    else
    {
      throw new IllegalArgumentException(...);
    }
}
```

After refactoring

```
public double GetTotalAmount (int orderId)
{
    if (orderId <= 0)
    {
       throw new IllegalArgumentException(...);
    }

    Order order = OrderRepo.FindById(orderId);
    if (order == null)
    {
        throw new OrderNotFoundException(...);
    }

    double totalAmount = 0.0;
    foreach (OrderItem item in order.GetItems())
    {
       totalAmount += item.GetPrice() *
                      item.GetQuantity();
    }
    return totalAmount;
}
```

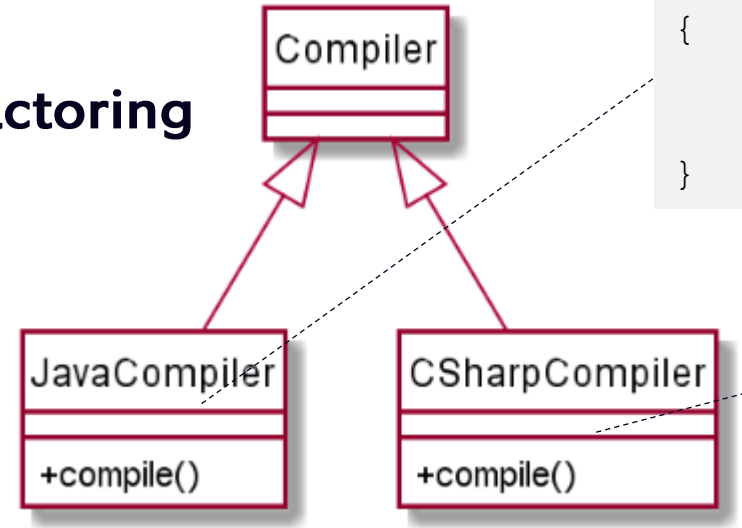
Refactoring Method: Form Template Method

- You have methods in subclasses that perform similar steps in the same order, yet the steps are different

- Generalize the methods by extracting their steps into methods with identical signatures
- Pull the identical methods into the superclass

Form Template Method - Example

Before refactoring

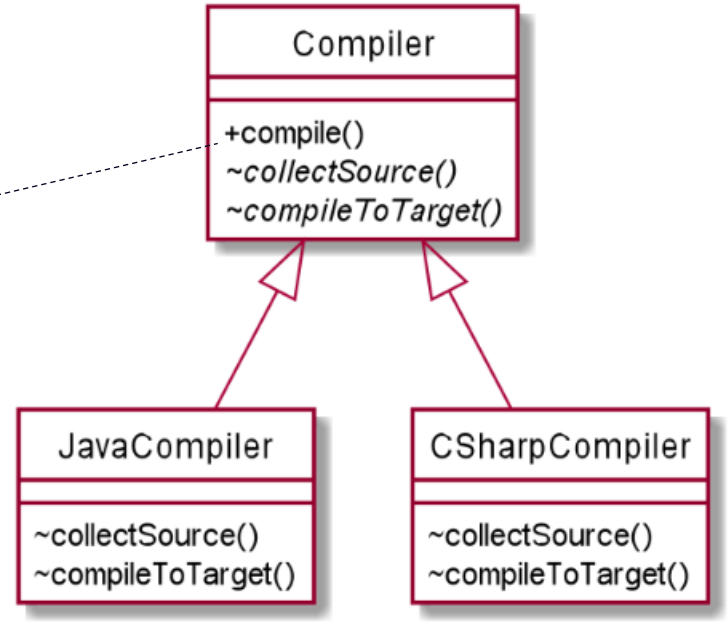



```
public void Compile()
{
    // code to collect source (java specific)
    // code to compile to target (java specific)
}
```

```
public void Compile()
{
    // code to collect source (C# specific)
    // code to compile to target (C# specific)
}
```

After refactoring

```
public void Compile()
{
    CollectSource();
    CompileToTarget();
}
```



Refactoring Is Not Just for Production Code

- Apply refactorings also in tests
- Tests should be as clear as production code

Applying Sequence of Refactoring Methods

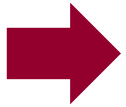
- For example Composed Method Pattern

- *“Divide your program into methods that perform one identifiable task. Keep all of the operations in a method at the same level of abstraction. This will naturally result in programs with many small methods, each a few lines long.”*

Composed Method Pattern - Example

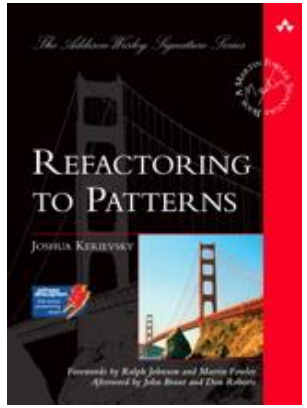
```
public void Add (Object element)
{
   if(!readOnly) {
      int newSize = size + 1;
      if (newSize >= elements.length) {
         Object[] newElements =
            new Object[elements.length + 10];
         for (int i = 0; i < newElements.length; i++) {
            newElements[i] = elements[i];
         }
         elements = newElements;
      }

      elements[size++] = element;
   }
}
```



```
public void Add (Object element)
{
    if (readOnly)
    {
        return;
    }

    if (ExceedsCapacity())
    {
        Grow();
    }
    AddElement(element);
}
```



nagarro

04a

Hands-On: Refactoring

The Guilded Rose

The Gilded Rose – The System

Hi and welcome to team Gilded Rose.

As you know, we are a small inn with a prime location in a prominent city ran by a friendly innkeeper named Allison. We also buy and sell only the finest goods. Unfortunately, our goods are constantly degrading in quality as they approach their sell by date. We have a system in place that updates our inventory for us. It was developed by a no-nonsense type named Leeroy, who has moved on to new adventures.

First an introduction to our system:

- All items have a SellIn value which denotes the number of days we have to sell the item
- All items have a Quality value which denotes how valuable the item is
- At the end of each day our system lowers both values for every item

Pretty simple, right? Well this is where it gets interesting:

- Once the sell by date has passed, Quality degrades twice as fast
- The Quality of an item is never negative
- "Aged Brie" actually increases in Quality the older it gets
- The Quality of an item is never more than 50
- "Sulfuras", being a legendary item, never has to be sold or decreases in Quality
- "Backstage passes", like aged brie, increases in Quality as it's SellIn value approaches: Quality increases by 2 when there are 10 days or less and by 3 when there are 5 days or less but Quality drops to 0 after the concert

Just for clarification, an item can never have its Quality increase above 50, however "Sulfuras" is a legendary item and as such its Quality is 80 and it never alters.

The Gilded Rose – Your Task

- Clone the repository “04-Refactoring”
- Apply refactorings until you think the code is readable and maintainable so that you are ready to add new features without any pain.
- Feel free to make any changes to the *UpdateQuality* method and add any new code as long as everything still works correctly.
- Use the refactoring tools of your IDE whenever possible.
- Execute the approval tests (see ApprovalTest) after each refactoring step which serve as golden master and make sure the behaviour of the system has not changed

- Constraints
  - Feel free to make any changes to the UpdateQuality method and add any new code as long as everything still works correctly. However, do not alter the Item class or Items property as those belong to the goblin in the corner who will insta-rage and one-shot you as he doesn’t believe in shared code ownership (you can make the UpdateQuality method and Items property static if you like, we’ll cover for you).

The Gilded Rose – Additional Task

- We have recently signed a supplier of conjured items. This requires an update to our system:
  - "Conjured" items degrade in quality twice as fast as normal items

Appendix: References

- Refactoring: Improving the Design of Existing Code; Martin Fowler and Kent Beck; Addison Wesley; 1st edition; 28th June 1999

- Refactoring Workbook; William C. Wake; Pearson Education; 1st edition, 4th September 2003

- Refactoring to Patterns; Joshua Kerievsky; Addison Wesley; 1st edition, 5th August 2004