Refactoring Fundamentals

Code Smells: Dispensables

Steve Smith
Ardalis.com
@ardalis

pluralsight
hardcore developer training

In This Course

- ~~What is Refactoring?~~
- ~~Why do it?~~
- ~~What's the process?~~
- ~~What are some tools that can assist with it?~~
- ~~What is a *Code Smell*?~~
- What are some examples of Code Smells?
- What are some common refactorings?
- How does one apply them correctly?

Organizing Code Smells

- **Taxonomy proposed by Mäntylä, M. V. and Lassenius, C.**
  - http://www.soberit.hut.fi/~mmantyla/ESE_2006.pdf

- **Organization of Code Smells into 5 Groups**
  - ~~The Bloaters~~
  - ~~The Object-Orientation Abusers~~
  - ~~The Change Preventers~~
  - The Dispensables
  - The Couplers

- **I've added three more:**
  - ~~The Obfuscators~~
  - Environment Smells
  - Test Smells

Code Smells: The Dispensables

- Provide little or no value
- Dispensable Classes
- Dispensable Code

Delete, Erase, or Remove the Redundant, Unnecessary, or Dispensable

Dispensables: Lazy Class

- Every class costs something to maintain and understand

- Classes that don't do enough to justify their existence should be removed

Rules of Simple Design

1. Runs all tests
2. No duplication
3. Expresses intent
4. Minimizes number of classes and methods

Refactor

- Collapse Hierarchy
- Inline Class

http://www.fotopedia.com/items/flickr-386529128

Dispensables: Data Class

- **Contain fields and/or properties, but nothing else**
- **Likely to be manipulated far too much by other classes**
  - OK as ViewModels, DTOs, etc.

```
public class Account
{
    public int Id;
    public string AccountType;
    public decimal Balance;
}
```

Refactor

- **Move / Extract Method**
- **Hide Method / Remove Setting Method**
- **Encapsulate Field / Collection**

Data classes are like children… okay as a starting point, but to [be a] grownup object, need some responsibility.

Refactoring

Data Class

```
public class Account
{
    public int Id;
    public string AccountType;
    public decimal Balance;
}
```

Data Class Usage

```
public class InterestCalculator
{
    /* constructor omitted */
    public void CalculateInterest(Account account)
    {
        if (account.AccountType == "Checking")
        {
            return;
        }
        if (account.AccountType == "Savings")
        {
            decimal interest = account.Balance * this._interestRate;
            account.Balance += interest;
            return;
        }
        throw new InvalidOperationException(string.Format("Unknown Account
    Type: {0}", account.AccountType));
    }
}
```

```
public class RefactoredAccount
{
    public RefactoredAccount (int id, string accountType, decimal balance)
    {
        this.Balance = balance;
        this.AccountType = accountType;
        this.Id = id;
    }

    public int Id { get; private set; }
    public string AccountType { get; private set; }
    public decimal Balance { get; private set; }

    public void CalculateAndApplyInterest()
    {
        var calculator = CalculatorFactory.GetFactory(AccountType);

        var interest = calculator.CalculateInterestForBalance(Balance);

        AdjustBalance(interest);
    }

    private void AdjustBalance(decimal amount)
    {
        // log access to balance
        Balance += amount;
    }
}

public class CalculatorFactory
{
    public static ICalculateInterest GetFactory(string accountType)
    {
        // TODO: Implement this method
        throw new NotImplementedException();
    }
}

public interface ICalculateInterest
{
    decimal CalculateInterestForBalance(decimal balance);
}
```

Dispensables: Duplicate Code

- Obviously, this violates the Don't Repeat Yourself principle
- Frequently a result of copy-paste-coding

Refactor

- Extract Method
- Pull Up Method
- Extract Class
- Form Template Method

```
public class ArticleService
{
    public void Edit(Article article, User user)
    {
        if (!user.IsInRole("Editors") && !article.Author.Equals(user))
        {
            // throw exception
        }
        // do other work
    }

    public void Publish(Article article, User user)
    {
        if (!user.IsInRole("Editors") && !article.Author.Equals(user))
        {
            // throw exception
        }
        // do other work
    }

    public void Delete(Article article, User user)
    {
        if (!user.IsInRole("Editors") && !article.Author.Equals(user))
        {
            // throw exception
        }
        // do other work
    }
}
```

```
public class RefactoredArticleService
{
    public void Edit(Article article, User user)
    {
        VerifyUserCanPerformAction(user, article);
        // do other work
    }

    public void Publish(Article article, User user)
    {
        VerifyUserCanPerformAction(user, article);
        // do other work
    }

    public void Delete(Article article, User user)
    {
        VerifyUserCanPerformAction(user, article);
        // do other work
    }

    private void VerifyUserCanPerformAction(User user, Article article)
    {
        if (!user.IsInRole("Editors") && !article.Author.Equals(user))
        {
            // throw exception
        }
    }
}
```

Dispensables: Dead Code

- **Code that is never used**
  - Classes
  - Methods
  - Condition blocks
- **Often easily detected by tools**

Refactor

- **Delete**

Tip

When you find dead code, do the right thing. Give it a decent burial. Delete it from the system.

Robert C. Martin, *Clean Code*

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;
using System.Dynamic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Globalization;
using System.Linq;
using System.Reflection;
using System.Security;

namespace CodeSmells.Dispensables
{
    public class DeadCode : ISeeDeadCode
    {
        public void DoStuff()
        {
            int upperBound = 100;
            if (upperBound > 50)
            {
                throw new NotImplementedException();
            }
            var fibNumbers = new List<long>();
            if (fibNumbers.Count == 0)
            {
                fibNumbers.Add(1);
                fibNumbers.Add(2);
            }
            int index = 2;
            long term = 0;
            while (term <= upperBound)
            {
                term = fibNumbers[index - 2] + fibNumbers[index - 1];
                fibNumbers.Add(term);
                index++;
            }
        }

        private void DoOtherStuff()
        {
            // does other stuff
        }
    }
```

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Data;
using System.Dynamic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Globalization;
using System.Linq;
using System.Reflection;
using System.Security;

namespace CodeSmells.Dispensables
{
    public class DeadCode : ISeeDeadCode
    {
        public void DoStuff()
        {
            int upperBound = 100;
            if (upperBound > 50)
            {
                throw new NotImplementedException();
            }
            var fibNumbers = new List<long>();
            if (fibNumbers.Count == 0)
            {
                fibNumbers.Add(1);
                fibNumbers.Add(2);
            }
            int index = 2;
            long term = 0;
            while (term <= upperBound)
            {
                term = fibNumbers[index - 2] + fibNumbers[index - 1];
                fibNumbers.Add(term);
                index++;
            }
        }

        private void DoOtherStuff()
        {
            // does other stuff
        }
    }
```

Dispensables: Speculative Generality

- "Oh, someday we might need to… "
- If you're using it, keep it. If not…
- Remember YAGNI; forget "someday"

Refactor

- Collapse Hierarchy
- Inline Class
- Remove Parameter

Tip

Programmers are notoriously bad at guessing what functionality might be needed someday.

Steve McConnell, *Code Complete*

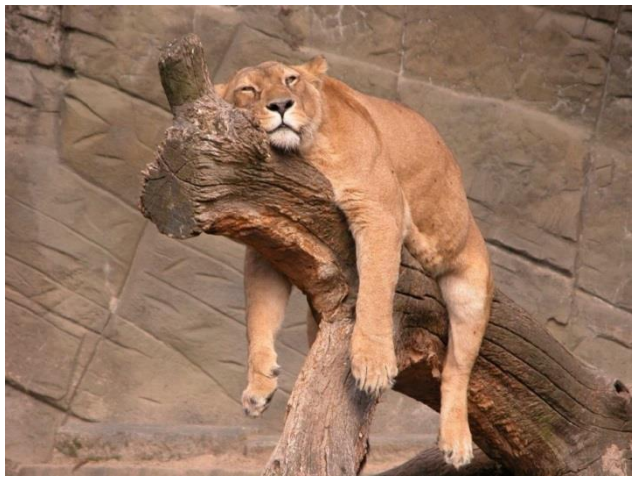
Problems with Speculative Generality

- **Requirements aren't known, so programmer must guess**
  - Wrong guesses will mean the code must be thrown away
- **Even a close guess will likely be wrong about the details**
  - These intricacies will undermine the programmer's assumptions – the code must be (or should be) thrown away
- **Other/future programmers may assume the speculative code works better or is more necessary than it is**
  - They build code on the foundation of speculative code, adding to the cost when the speculative code must be removed or changed
- **The speculative generality adds complexity and requires more testing and maintenance**
  - This adds to the cost and slows down the entire project

Summary

```
public class Account
{
    public int Id;
    public string AccountType;
    public decimal Balance;
}
```

Delete, Erase, or Remove the Redundant, Unnecessary, or Dispensable



DO NOT DUPLICATE

References

Related Pluralsight Courses

SOLID Principles of Object Oriented Design http://bit.ly/rKbR9a

Design Patterns Library http://bit.ly/SJmAX1

Books

Code Complete http://amzn.to/Vq5YLv

Clean Code http://amzn.to/YjUDI0

Refactoring http://amzn.to/110tscA

Web

Rules of Simple Design http://bit.ly/btTomW

Thanks!

Steve Smith

Ardalis.com

Twitter: @ardalis

To Teach Is To Learn Twice

pluralsight
hardcore developer training

Guest Opinion: Scott Hanselman