Code Analysis

Reflect on Your Code
Abstract

Most of the time developers produce code, but they rarely manage to adequately review their code to a high level of quality. I aim to introduce simple analysis insights (such as code metrics, complexity, etc...) and present some crucial tools which really pay-off for medium-large scale code-bases. Simple concepts such as code-reuse and re-factoring, although much discussed in the community, are still not thoroughly understood/employed by developers. This is especially obvious when analyzing common open-source projects (.NET). I'll illustrate how a very rigorous process of code review and continuous refactoring have a huge impact.
Disclaimer/Delimitation

• The author does not have significant experience to provide personal judgements over specific matter

• Introductory, no in-depth worked example
Plan

● Introduction
● Code Metrics
● Refactoring
● Tools
  o VS Ultimate
  o NDepend
● Conclusion
Introduction
Complexity - Accidental

Remember the evolution:
- Assembly
- High level/order
- Garbage collection
- Domain specific
bubbleSort( A : list of items )

n = length of A
set swapped false
repeat
    for i = 2 to n-1 inclusive do
        if A[i-1] > A[i] then
            swap A[i-1] with A[i]
            set swapped to true
        end if
    end for
until not swapped
end
Interrelation

Analysis ⇔ Refactoring ⇔ Testing
Code Metrics
Problems

- Technical Debt

- Code Smells
  - Large classes
  - Long names
  - 5 indentation levels...

- Copy-paste code reuse
Software

output = function X(input) {
    // Local work
    // Global work
}
Example - from OOP

No global => functional => fail

How would a functional method look like?

class X {
    output = Method(input)
}

Example - to functional

1st step

output = Method(this, input)

2nd step

output = Method(global, this, input)
“Software development is and always will be somewhat experimental.”
Code Metrics

- Lines of code
- Cyclomatic Complexity
- Maintainability Index

+ etc…
Code Metrics - LOC + extensions

Example
for (i = 0; i < 100; i++) printf("hello");

/* Versus */
for (i = 0; i < 100; i++)
{
    printf("hello");
}
RefactorExample 1

```java
public enum DanishMonths {
    JANUAR, FEBRUAR, MARTS, APRIL, MAJ, JUNI,
    JULI, AUGUST, SEPTEMBER, OKTOBER, NOVEMBER, DECEMBER
}
```
RefactorExample 1 - Refactored

```csharp
var culture = CultureInfo.GetCultureInfo("da-DK");

var dateTimeInfo = DateTimeFormatInfo.GetInstance(culture);

var months = dateTimeInfo.CurrentInfo.MonthNames;
```
Code Metrics - Halstead Volume

\[ V = N \times \log_2 \eta \]

\[ N = \text{operators} + \text{operands} \]
\[ \eta = \text{distinct} \ (\text{operators} + \text{operands}) \]
Code Metrics - Halstead Volume

Example

```javascript
var x, y
var z = f(x, y)
z = (x+y/2)/3
f2(z)
```

\[ N = (2+1+1+2+1+1+1+1)+(3+3+3) = 18 \]

\[ \eta = 7 + 3 = 10; \{((), +, /, =, var, f, f2\}, \{x, y, z\} \]

\[ \Rightarrow V = 10 \times \log(10) = 59.7 \]
Code Metrics - Cyclomatic Complexity

\[ M = E - N + 2P \]

\[ E = \text{edges}. \]
\[ N = \text{nodes}. \]
\[ P = \text{connected components(cycles)}. \]

\[ \Rightarrow 9 - 8 + 2 \times 1 = 3 \]
Code Metrics - Cyclomatic Complexity

Example:

```c
while( c1() ) f1();

if( c2() ) f3();
else f4();
```
Code Metrics - Cyclomatic Complexity

Deceiving
● non-disjoint Ifs
● not accounting for libraries

Testing
● will complexity += 1 => tests += 1? (hint: no!)
● code/branch/path coverage...
**Code Metrics - Cyclomatic Complexity**

Example

```javascript
var c
if( c1() ) x = f1();
else x = f2();
if( c2() ) y = f3();
else y = f4();
if( c3(x, y) ) f5();
else f6();
```
Code Metrics - Cyclomatic Complexity

Further useful for improving
- Time to fix bugs
- Regressing bug
RefactorExample 2

```csharp
private string MapBathRooms(string value) {
    double retValue = 0;
    if (value == "1" || value == "One")
        retValue = 1;
    if (value == "OneAndHalf" || value == "1.5" || value == "1 1/2")
        retValue = 1.5;

    //... Up to 10

    return retValue.ToString();
}
```
Dictionary<string, string> BathRoomMap = new Dictionary<double, List<string>>
{
    { 1, new List<string>() {"1", "One" } }
    { 1.5, new List<string>() {"1 1/2", "OneAndHalf" } },
    // etc
};
private string MapBathRooms(string value) {
    var retKeyValue = BathRoomMap.GetKeyValues()
        .SingleOrDefault(x=>x.Contains(value))
        .FirstOrDefault(x=>x.Contains(value))
    if(retKeyIdValue==0) return 0;
    return retKeyIdValue.Key;
}
Code Metrics - Maintainability Index

\[171 - 5.2 \ln(HV) - 0.23CC - 16.2 \ln(LOC) + 50.0 \sin \sqrt{2.46 \times COM}\]

Problems?

- 1 - magic numbers
- 2 - averages
- ..
- n
Code Metrics - Empirical Research

“Empirical Analysis of CK Metrics for Object-Oriented Design Complexity”
=> some correlation, interdependence

“Empirical Analysis of Object-Oriented Design Metrics for Predicting High and Low Severity Faults”
=> some correlation, most with SLoC
Code Metrics - Empirical Research

“Questioning Software Maintenance Metrics: A Comparative Case Study”

=> Only system size and low cohesion were strongly associated with increased maintenance effort”

=> quote more research...
Software Architecture - Ideal
Software Architecture - Cycles
Coupling

Any Methods, Types, Namespaces that have a direct reference to
• Fields, Methods, Types, Namespaces

Depending on direction: afferent or efferent
Metrics

● Stability
  ○ Couplings (dependencies) – afferent/efferent

● Abstractness
  ○ Types - abstract/concrete

\[ I = \frac{C_e}{C_a + C_e} \]

\[ A = \frac{N_a}{N_c} \]
Principles

• Stable Abstractions Principle – stability should match abstractness as close as possible

• Stable Dependencies Principle – fewer dependencies on fast-changing types
Software Architecture – done right
Refactoring
Part 2
Refactoring Methods

rename element
create getter setter
extract local variable
override methods
move element
extract method
generate constructor using fields
surround with try catch
modify method parameters
inline
add unimplemented constructors
extract constant
sort members
generate hashcode equals
generate tostring
self encapsulate field
promote local variable
extract interface
Refactoring - Empirical Research

“A Field Study of Refactoring Challenges and Benefits” by Microsoft, Windows 7 =>

"The difficulty of merging and integration after refactoring often discourages people from doing refactoring"

"If there is insufficient documentation for scenarios, refactoring should not be done."
Refactoring - Empirical Research

... =>
"The primary risk is regression, mostly from misunderstanding subtle corner cases in the original code and not accounting for them in the refactored code."

- dev.

"top 25% of refactored binaries have 12 percent more reduction in post-release defects compared to all modified binaries"

- author
Refactoring - Empirical Research

“An Empirical Investigation into the Impact of Refactoring on Regression Testing” by Texas University =>

"The results on three open source projects, JMeter, XMLSecurity, and ANT, show that only 22% of refactored methods and fields are tested by existing regression tests."
Refactoring - Empirical Research

... =>

"The study found that test coverage of refactoring is insufficient and that regression tests are significantly impacted by refactorings edits..."
Demos
Tools

- Visual Studio Ultimate
  - code cloning
  - metrics
  - dependency graph
- FxCop
  - command line, rules...
- NDepend
  - all above + more
Tools - Choices

- Visual Studio Ultimate
  - code cloning
  - metrics
  - dependency graph
- FxCop
  - command line, rules...
- NDepend
  - all above + more
Conclusion
Incentives

Would incentivizing compliance lead to a better development process?

Maybe... No

Why?
Validity

Code analysis

- Fails to capture true complexity
- Is heavily correlated
- Helps enforce qualitative constraints
... in the end

Fundamentally, there is
● Breadth
● Depth

For a given requirements set
  \[ F(\text{Breadth}, \text{Depth}) = \text{CONSTANT} \]
Thanks

for patiently listening