Refactoring as a Lifeline: Lessons Learned from Refactoring

Amr Noaman Abdel-Hamid

Agile2013 Conference – Nashville, Tennessee
Wednesday August 7th, 2013

About Amr:
– Senior Consultant, SECC
– Agile coach, trainer, and practitioner
– Sun Certified Enterprise Architect, Certified Scrum Professional, Six-sigma Black Belt
– Co-Author of the Process Increments method, an Agile method for process improvement
– Initiator of Egypt’s GoAgile program at 2011, an initiative intended to boost lean & agile software development in Egypt

amr.noaman@gmail.com
amr-noaman.blogspot.com
www.secc.org.eg
Agenda

- Egypt GoAgile Program
- First (Unsuccessful) Attempts to Refactoring
- A New Approach!
- More Attempts to Refactoring
- Lessons Learned and Keys to Successful Refactoring

Agenda

- Egypt GoAgile Program
- First (Unsuccessful) Attempts to Refactoring
- A New Approach!
- More Attempts to Refactoring
- Lessons Learned and Keys to Successful Refactoring
Egypt GoAgile Program

- Agile adoption program funded by the Government
- The program comprises of:
  - 4-day Agile Fundamentals training
  - Onsite workshops about some selected Agile practices
  - Working hand-in-hand with the team for one complete release (2-4 months)
- Accomplishments in 3 years
  - 30+ companies
  - 400+ trainees
  - 50+ teams

Impact of Poor Code Quality

- Poor code quality and accumulated technical debt
- Very slow velocity
- Less restrictive done definition
- The ‘no-demo-because-of-low-quality’ syndrome
- Very similar to the traditional lifecycle

For very large code base with lots of bugs and change requests, it becomes a nightmare!
Our First Approach

- Write system/functional automated tests to enable safe refactoring
- Enhance the code:
  - Identify and apply design patterns
  - Split into architectural layers
  - or any other best practices
Our First Refactoring Attempts

<table>
<thead>
<tr>
<th>Team</th>
<th>Product</th>
<th>Code quality</th>
<th>Management style</th>
</tr>
</thead>
<tbody>
<tr>
<td>SKM</td>
<td>.NET, business intelligence and planning tool</td>
<td>Very poor. It was based on a prototype and was not intended to be shipped to customers</td>
<td>Managers are technical people but senior managers are overly pragmatic business men</td>
</tr>
<tr>
<td>ACD</td>
<td>Java, distribution management for a large telecom provider</td>
<td>Very poor spaghetti code. All business logic was concentrated in three classes and logic coded directly in database procedures</td>
<td>Managers are very demanding and are very distant from the technical solution. They deal with only one SPOC from the development team</td>
</tr>
<tr>
<td>ORB</td>
<td>.NET, government management and online services</td>
<td>Originally followed MVC, then deteriorated until it became very poor.</td>
<td>Technical manager is overwhelmed with details, and senior managers very distant from development</td>
</tr>
</tbody>
</table>

Our First Refactoring Results

• 2 Projects stopped refactoring and started a complete re-write

• 1 Project stopped refactoring and kept a lengthy round of regression testing for every release
What Went Wrong?

1. Objectives of refactoring were vague
2. Failure to code automated tests
3. It’s none of the Managers’ Business!
4. Technical Glut Trap
5. Unsustainable Development Pace

1. Objectives of Refactoring were Vague

- No quantified objectives
- Some hazy impressions about what to be done
- Next steps were determined by the technical team’s “gut feeling”!
2. Failure to Code Automated Tests

- Automated tests are safety net
- Perceived as a pre-requisite for refactoring
- Challenges:
  - No clear business interfaces
  - Scattered business logic in all layers
  - Very high cost (4 month dedicated work)
- Automated tests were viewed as ‘The first impediment to refactoring!’

3. It’s None of the Managers’ Business!

We are busy refactoring the product code for the benefit of all of us

It is a highly technical stuff which you will not grasp, even if you tried hard

We need to concentrate. Please do not keep nagging for status and end dates

When we finish, we’ll let you know

😊 Thank you for your understanding😊
3. It’s None of the Managers’ Business!

The First Rule of Managing Refactoring Effort:

Managers will not sponsor any activity unless they can track and control it

4. Technical Glut Trap

It is a situation when the team indulges in deep technical reviews and code refactoring with no limits to their technical imagination and creativity
4. Technical Glut Trap

One team kept trying large refactorings. Every time it resulted in overly complicated code, so they stop and start over.

Second team stuck to their first set of (very large) refactorings. Code complexity kept rising, and finally time elapsed while not yet finished.

5. Unsustainable Development Pace

- Extra cost for no visible ROI (at least in the short run)
- Delayed fixes and prolonged plans for new developed features
- Two code bases to maintain:
  - One for refactoring
  - Another one for maintaining existing old code
Let’s Review

1. Even with unclear objectives, effort should be planned and tracked
2. Start from another place other than automated tests
3. Find a way to involve busy managers
4. Find a way to control the team’s technical appetite
5. Ensure sustainable refactoring

Agenda

Egypt GoAgile Program

First (Unsuccessful) Attempts to Refactoring

A New Approach!

More Attempts to Refactoring

Lessons Learned and Keys to Successful Refactoring
Sustainable and Stepwise Refactoring Roadmap

- Only 10% of team effort for code refactoring
- Team should abide by the roadmap
- Team should prepare monthly status reports presented to managers with indicators of progress
- Refactoring is done on the mainline; no dedicated code branch is allowed!
- Only “safe” refactoring are allowed

Agenda

- Egypt GoAgile Program
- First (Unsuccessful) Attempts to Refactoring
- A New Approach!
- More Attempts to Refactoring
- Lessons Learned and Keys to Successful Refactoring
# Teams who Applied the Roadmap

<table>
<thead>
<tr>
<th>Team</th>
<th>Product</th>
<th>Tech.</th>
<th>Size</th>
<th>Code quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>SDY</td>
<td>6 developers, 2 testers</td>
<td>Electrometer management and control systems</td>
<td>.NET</td>
<td>204 kloc</td>
</tr>
<tr>
<td>DS</td>
<td>5 developers, 2 testers</td>
<td>Cardiac MRI analysis software</td>
<td>C++</td>
<td>541 kloc</td>
</tr>
</tbody>
</table>
Remove Dead Code

- Defined as:
  - Unused classes
  - Uncalled methods
  - Unused variables
- Would it make any difference?
  - Less time in debugging and locating defects
- Why would the team keep dead code?
  - Team dreads changing the code
  - Team thinks that old code may guide them later on

Remove Dead Code

- SWD:
  - Instantly identified dead code using Visual Studio Code Analysis Tools
  - Reduced the code size by 6.4%
- DS:
  - CPPDepend (www.cppdepend.com) used to identify dead code
  - Reduced the code size by 4%

VS code analysis rules to detect dead code:

- CA1804: Remove unused locals
- CA1811: Avoid uncalled private code
- CA1812: Avoid uninstatiated internal classes
- CA1801: Review unused parameters
Remove Code Duplicates

- AKA code clones
- Has high impact on code quality
  - Bugs are solved once and only once
- Two types
  - Level 1: exact Clones
  - Level 2: similar Clones

Level 1 Clones (Exact)

```java
private static Properties serverProperties = null;
/** Properties file that saves the properties */
private static File propsFile = null;

// Initialize and load the properties of the database server.
static {
    serverProperties = new Properties();
    propsFile = new File(\"./server.properties\");
    try {
        if (propsFile.exists()) {
            serverProperties.load(new FileInputStream(propsFile));
        } else {
            propsFile.createNewFile();
        }
    } catch (IOException e) {
        e.printStackTrace(System.err);
    }
}
```
Level 2 Clones (Similar)

```
switch (request.getType()) {
    case RequestTypes.MODIFY:
        try {
            modifiedRecord = (DataInfo) request.get;
        }
        catch (DatabaseException e) {
            System.err.println(new java.util.Date().
                e.printStackTrace(System.err));
            return e;
        }
    case RequestTypes.FIND:
        try {
            return (DataInfo) dataHandler.find();
        }
        catch (DatabaseException e) {
            System.err.println(new java.util.Date().
                e.printStackTrace(System.err));
            return e;
        }
    case RequestTypes.READ:
        try {
            return (DataInfo) dataHandler.get();
        }
        catch (DatabaseException e) {
            System.err.println(new java.util.Date().
                e.printStackTrace(System.err));
            return e;
        }
    case RequestTypes.CANCEL:
        case RequestTypes.EXCEPTION:
            break;
    case RequestTypes.EXCEPTION:
        return e;
}
break;
```

Simple and “Safe” Refactorings at this Stage

- Extract/move method
- Extract superclass, or any refactoring which deals with adjusting the type hierarchy or polymorphic operations
Quick-wins for the DS Team

“Burn charts have become a favorite way to give visibility into a project’s progress. They are extremely simple and astonishingly powerful”

-- Alistair Cockburn

Code Size Reduction Target (CSR Target) =

• 100% of Dead Code
• 90% of Exact Clones
• 60% of Similar Clones
Planning & Tracking
Quick-wins for the DS Team

Code Size Reduction Target (CSR Target) =

- 100% of Dead Code
- 90% of Exact Clones
- 60% of Similar Clones

Quick-wins Tracking Using Run Chart for the DS Team

- CSR Speed:
  - Number of LOC removed per hour of work
  - Calculated every iteration
Quick-wins Tracking Using Run Chart for the SWD Team

Lessons Learned from Refactoring

DIVIDE & CONQUER
Divide & Conquer

• Before this stage, code had no structure:

Before quick wins

After quick wins

• After this stage, code is like:
How?

- Group “similar” code together

- Similar code may be:
  - Functional
  - Utility
  - Architectural

More discussion about types of components at:

‘Divide & Conquer’ Using ConQAT

More discussion about types of components at:
‘Divide & Conquer’ Using ConQAT

This should be the architecture

This is the really happens!

Initial Architectural Analysis for the SWD Team
Iterating Through ‘Divide & Conquer’ Stage

Re/assign classes into components

Detect and resolve violations

Enhance Component Interfaces

No

Yes

Divide & Conquer Metrics

- Number of public methods per component
  - Good indicator of the complexity of the component (Ideally <= 30)
  - Smaller interface reduces work in the next stage
- Number of violations
  - The number of illegal calls from one component to another:
    - High coupling between components
    - “Backdoor” access to components
Important Note

- Only simple (and safe) refactorings are allowed, like:
  - move class, extract/move method, encapsulate field, move field, introduce parameter object, preserve whole object

- Refactorings like extracting subclass, superclass, or interface; or introducing factory method or classes are considered risky and less safe

BUILD QUALITY IN
Wrap Components with Automated Tests

- Now, we have components with clear interfaces
- For every component interface method, write one or more component tests

‘Build Quality-In’ Metrics

- Target Test Coverage
  \[ \text{target test coverage} = \frac{\text{included LOC}}{\text{total LOC}} \%
  \]

Where excluded LOC is non-business logic code like:
- GUI code
- Auto-generated code
- Third party code
Tracking the ‘Build Quality-In’ Phase using Burnup Chart

• For the SWD team:
  – target test coverage = 58% (103k / 176k)

Agenda

Egypt GoAgile Program

First (Unsuccessful) Attempts to Refactoring → A New Approach! → More Attempts to Refactoring

Lessons Learned and Keys to Successful Refactoring
Impact on Quality – DS Team

<table>
<thead>
<tr>
<th>Metric</th>
<th>Release 5.5 (Before refactoring)</th>
<th>Release 5.6 (During Refactoring)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total bugs detected</td>
<td>128</td>
<td>176</td>
</tr>
<tr>
<td>% of Regression bugs</td>
<td>29.7%</td>
<td>25.1%</td>
</tr>
<tr>
<td>Average bug fixing cost (hours)</td>
<td>1.97</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Key Lessons Learned

- **Management Support**
  - Management and Technical Teams are Collaborating
- **Sustainable Refactoring**
  - Slow but sure steps in the right direction
- **Way out of the Dilemma of Un-testable Code**
  - Code is still of poor quality, but it is now covered by Tests!
- **More Mature Development Practices**
  - After suffering from code duplicates removal, they will never do it again!
- **Tools Support**
Keys to Successful Refactoring

- Management buy-in through active participation
- Limit refactoring efforts (don't burn out the team)
- Start with simple wins, then build on them
- Uncover/polish the architecture (components)
- Measure and stop when gains in improvement are diminishing
- Use tools to help find/carry out refactoring opportunities
Questions?

References

- CanQAT tool: [https://www.conqat.org/](https://www.conqat.org/)

- Blogs:
  - Other refactoring topics: [http://amr-noaman.blogspot.com/p/refactoring-legacy-applications.html](http://amr-noaman.blogspot.com/p/refactoring-legacy-applications.html)
Thank you