Geometric

Building Testability into Legacy Code

Version 1.0
October, 2010

WHITE PAPER

Geometric

Contents

Introduction

Industry studies say that about a quarter of software development cost is spent on Testing. Moreover the investment in terms of time and money increases exponentially with the age and size of the software product or solution.

As the code is continually stretched to satisfy changing requirements; it develops resistance to stretch it further. This leads to a lot of quality related problems like Fragility[1], where a small legitimate change in one area causes some other area to fail in a manner that cannot be logically explained. Effectively, stakeholders have a poor confidence on the quality of software product.

To address this problem project managers have been investing heavily on Test Automation. There is a wealth of tools available in the market that helps testers to chop down the testing turnaround time to a great extent. Though valuable, these tools have got some inherent limitations:

- These tests (or test scripts) are not readable enough to convey the intent of what is being tested.
- They do not provide immediate feedback that developers really need.
- They are not light-weight enough.

The limitations are inherent because they are designed to mainly cater to the needs of testers. These tests are written as well as maintained by testers. There are other important stakeholders like developers and product managers, who also need automated tests for different reasons.

Experienced software architects recommend the best practice of building testability right into the software design/ code. 'Design for Testability' has gained a lot of attention, and over the years practitioners have built good wisdom on it – in the form of various design principles, patterns, etc.

However, things are not so straight forward when it comes to building testability into the code base that has been around for quite a few years. Design choices are often limited by how the existing structure has evolved over the years. Implicit assumptions and lack of documentation, which is not uncommon, makes understanding of existing design even more challenging.

Focus of this paper is to share challenges faced and techniques used in building testability into legacy code in order to have continual quality improvements in the software system.

Background

This paper builds on the existing knowledge base available in the public domain, which has been built over the years. This has been contributed to not only by software professionals, but also by practitioners from other streams.

Techniques discussed here are inspired from and is built upon the following principle sources:

- The book "Working Effectively With Legacy Code" by Michael C. Feathers.
- Theory of Constraints conceived by Dr. Eliyahu M. Goldratt.
- The book THE GOAL by Dr. Eliyahu M. Goldratt
- The book "Refactoring: Improving The Design Of Existing Code" by Martin Fowler.

What is Legacy Code?

Different people may have different perceptions about Legacy Code. However this paper uses the definition provided by Michael Feathers in his book Working Effectively with Legacy Code. The definition goes as follows:

Looking at the code base, let's ask ourselves the following questions:

- Is the code easy to change?
- Do I get nearly instantaneous feedback when I change it?
- Do I understand it?

If answer to the any of the above questions is NO, then we have a legacy code.

Legacy code has got an interesting characteristic:

It works … as long as we don't touch it!

Nevertheless, we need to modify code in order to change its behavior in the form of fixing bugs or implementing change requests. Quite often developers do not have an adequate understanding of the design; hence, the modifications made in the code are sub-optimal. This leads to piling of Technical Debt[2], which leads to increase in the entropy[3] of software systems. Software systems degrade over a period of time. This is not because of lack of competency of developers, but it is an inherent property of any system. The challenge is - how can we reduce the rate of degradation? Or can we even reverse the entropy? Building testability is all about reversing the entropy of software systems.

Fundamental Challenges in Building Testability into Legacy Code

Development team can deliver high quality software that meets customers' needs and expectation in given time frame only when they get an early feedback. This is a fundamental assumption.

We are building testability so that the system can provide immediate feedback to the developers on what code they are writing. It is advisable to cover a safety net in form of automated tests before making serious changes in the code.

There is a challenge in bringing the code under suite of tests. Very often legacy code doesn't have well defined module boundaries. Code that handles GUI (Graphical User Interface) is often mixed with business logic. Algorithms are often coupled with code that talks to database. 'Coupling' is a well-known major challenge in software design. In many cases initial design of the system is modular, but the boundaries get blurred as requirements change, experienced developers leave and knowledge is not effectively transferred.

Given that legacy code is not typically test friendly, one has to modify it to bring it under test harness. Now here is the real dilemma. As discussed in previous sections, legacy code is typically fragile. An innocent, legitimate change in module can make something else fail in a manner that cannot be logically explained. Refactoring is often more aggressive, and hence, risky. So developers are often advised not to modify the code unless there is no workaround!

So we have a deadlock as shown in Figure 1. The real challenge is how to

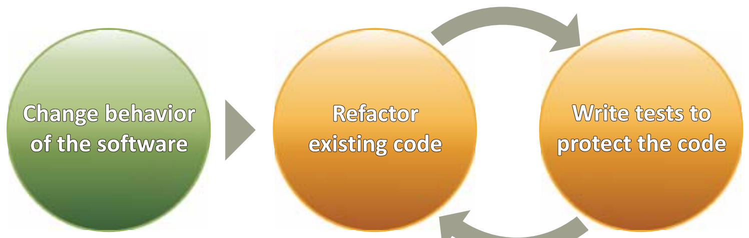
Change behavior of the software
Refactor existing code
Write tests to protect the code

Fig 1: The deadlock faced in working with legacy code

The Seam Concept

'Seam' [4] is a concept as described by Michael Feathers in his book, which helps us in breaking the deadlock. Definition of a Seam is as follows:

"A seam is a place where you can alter behavior in your program without editing in that place."

In the code (irrespective of it being a legacy code or not) there are always some points that can act as hooks or can be converted into hooks, where we can write additional code to change behavior of the program. Effectively these are the seams.

Every seam has an enabling point, a place where we can make the decision to use one behavior or another.

The book describes following types of seams, which we could successfully exploit in our projects:

- **Object Seams:** Exploit the polymorphism
- **Pre-Processor Seams:** This is available in languages like C/C++ where a pre-processing step is involved before compiling the code.

- **Link Seam:** Statically typed languages involve a linking step after compiling. At this step we can decide which code one needs to link to: original production code or the code added for testing.

In our project, we could identify one more seam type i.e. DLL Hooks. This is a technique, where we write a piece of code which modifies memory addresses contained in import section of a loaded program (executable) with new addresses that contain the testing code.

For example, consider a CAD program which has calls to a graphic library dispersed all over the places. At runtime, we can intercept certain calls to the graphic library that are of our interest, run some testing code and then, if required, route those calls again to the original graphic library.

We could successfully exploit the Seam concept to perform automated testing at GUI (Graphical User Interface) level, without modifying the code in that place. This can be done without the use of external automation tools because testability is built right inside the code. Another main advantage of this technique is that the tests (or test scripts) are highly readable compared to test scripts used with external automation tools. They clearly convey the intent of the test, what it is doing. Tests, rather than depending on Windows controls, work at more semantic level, hence they are more stable. Since they are readable and stable, they are easy to maintain as well.

Further Challenges in Building Testability

The Seam concept works very well, but that is not enough. If we can identify the seams, then we can exploit them to cover existing code with tests. However, this is a challenging task when the code base is huge. It is not uncommon to have multi million lines of code for a legacy code base. Covering the entire code with tests is a mammoth task. This is the place where the Theory of Constraints[5] conceived by Dr. Goldratt helps to manage the challenge.

The Theory of Constraints and its philosophy have the following fundamental belief:

"Every complex system is based on inherent simplicity. And the key to improve the system is to exploit this inherent simplicity."

Have a look at the following figure, which shows two systems – System A and System B. The task is to find out which system is more complex.

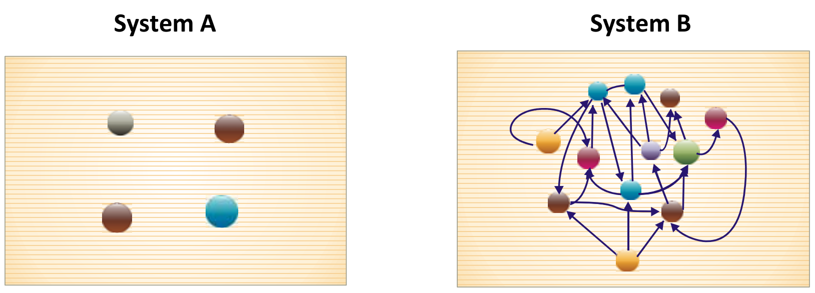

Fig 2: Which system is more complex? Taken from http://www.toc-lean.com/The_Choice.htm

Apparently System B is more complex than System A. However a closer look reveals something different. System A has got four degrees of freedom. One has to manage these four points in order to improve System A. Structure of System B is such that there is one node that is heavily depended upon by all the other nodes, either directly or indirectly. Any improvement made in this node directly translates to the improvement in the overall system.

Software systems too exhibit a similar behavior. Irrespective of how huge the code base is or how complex the system is, there exists a simplicity which can be exploited. In fact the Seam concept discussed above is an example of simplicity, while the concept of ‘enabling point’ is about how to exploit the same.

Five Focusing Steps of TOC (Theory of Constraints)

Theory of Constraints further generalizes this as dealing with constraints. In the above example of System B, there is one node in the system which limits or rather determines performance of the overall system. Any effort spent on improving this node directly translates to the overall system efficiency. This node is called as constraint.

TOC advocates following five focusing steps to achieve continual improvement in the system:

1. Identify the CONSTRAINT
2. EXPLOIT the CONSTRAINT
3. SUBORDINATE everything else to the tune of above decision(s)
4. ELEVATE the constraint
5. If the constraint is broken then go back to step #1.

   IMPORTANT: Never allow INERTIA to become a constraint.

These five focusing steps effectively work with a wide range of systems including our software systems. The steps are general and their sequence must be enforced. However, implementation of these steps can vary depending on the context.

Now let us look at how these steps can be implemented in context of a software system to achieve continual improvement.

STEP I: Identify the Constraint

Usually an experienced developer, QA or project manager has some gut feeling regarding where the constraint is. In many cases this gut feeling is correct, but it can go wrong as well. Identifying the constraint is the critical step. If this is identified incorrectly, then efforts spent on improvement may not directly translate into system improvements.

Hence, it is desirable to have some analytical/ evidence based approach. Following are possible (but not limited to) ways to identify the constraint, which can actually be measured:

- Identify which parts of the system (e.g. module, package, function) are modified quite often in order to fix bugs.
- Identify which part of the system is depended upon heavily by other parts in the system.
- Identify which parts of the system contain code that is difficult to understand, rigid and fragile.

This will provide us the modules which are in heavy demand to change and at the same time are quite difficult to change. These are the hot spots in the code base and the candidates to be called CONSTRAINTS.

Following are some mechanism to do the same:

- Bug/ Regression Analysis: Analysis of recently introduced bugs/ regressions may reveal the modules in the system they originate from. Many bug tracking systems provide tight integration with Software Configuration Management tools. This may reveal which part of the code is modified quite often to fix bugs.
- Version Control Analysis: There are some open source as well as proprietary tools available that run on version control systems (or configuration management system) to reveal the trend in which part of the code base is modified quite often and the manner in which it is getting modified.
- Static Dependency Analysis: There are tools available in the market that can perform static analysis of the code and infer the dependency structure of the system. There are some well-defined metrics used in software engineering that tell whether the given dependency is desirable or not. Some examples are Stability, Abstractness, and Distance from Main Sequence [6].
- Dynamic Dependency Analysis: As against static analysis, these tools actually execute the code in order to see which parts of the code are executed quite often.
- Complexity Analysis: Static analysis tools can also reveal which parts of the code (in terms of modules, classes, functions etc) contain highly complex code. This can be measured in terms of Cyclomatic Code Complexity [7].

  Good thing about these mechanisms are:

  - They work on well-defined metrics, and
  - They are non-destructive.

At the end of this step we should be able to arrive at the CONSTRAINT.

STEP II: Exploit the Constraint

Since constraint is something that limits or determines performance of the overall system, it needs to be utilized to its full potential. Following are possible (but not limited to) ways to achieve the same:

- Assign best/ experienced developers to work on the module which is identified as CONSTRAINT. Since they are supposed to know the code very well, they will make changes that are more likely to be appropriate.
- Code Reviews: Every change made in the constraint module needs to be reviewed with a magnifying glass.
- Identify SEAMs in this module so that it can be brought under tests. This requires some minor refactoring which is supposed to be less-risky. This kind of refactoring mainly works on dependency breaking technique described in the book "Working Effectively With Legacy Code".
- Start writing automated tests such that all the code in the constraint module will get covered. It does not really matter whether these tests are written at Unit level or at System level. The objective is to cover the code by automated tests that would provide an immediate feedback.

STEP III: Subordinate everything else to above decision(s)

One important thing that TOC advocates is getting rid of 'local optimization'. All parts of the system don't have to be 100% efficient.

This effectively translates to focus. Staying focused is important. That does not mean other areas are to be ignored, but the efforts must be focused on breaking the constraint.

Other areas need efforts that are just good enough.

This can be achieved by the following possible (but not limited to) mechanisms:

- See if the change requests can be routed such that they do not need to modify the code in the constraint modules.
- Exploit mechanisms such as polymorphism in Object Oriented language (provided the code follows object oriented paradigm). Typically one can think of sub-classing and put new code in the sub-classes rather than actually modifying the code in constraint module. This confirms to the well known Open Closed Principle in software engineering.
- Automate code reviews by using static code analysis tools. These tools relieve a lot of efforts from manual code reviews and the saved efforts can be used to stay focused on constraint modules.

Subordination may need to have a change in policy as well. For example, many project teams have a policy of 100% code review, i.e. each and every line of the code change must be manually reviewed. It is good to have 100% code reviews, but in many cases it may not be effective, or rather it may prove to be counter-productive. Use of tools for automated code reviews helps to a great extent.

STEP IV: Elevate the Constraint

This is the real objective that works towards breaking the constraint. This effectively needs to modify design of the code such that it is easy to understand and easy to change. Essentially this should be done without modifying external behavior of the code. This way of improving design of existing code without changing its behavior is called Refactoring [8]

Over the years, Refactoring has become a disciplined area in software engineering.

There are well defined techniques available for refactoring. A book by Martin Fowler on Refactoring is an excellent valuable resource for the same.

Refactoring effectively reverses the entropy of the software system.

STEP V: If the constraint is broken, then go back to step I

Step V is not infinite; it has a definite end point. In order to know where to stop, one needs to have tab on various metrics and how the code base, especially the constraint module is improving against the metrics.

As the code is refactored and becomes simpler, at some point of time it no more remains a constraint.

However, the fundamental belief of TOC is that the system performance is always limited by a constraint. If the above constraint is broken, effectively it means the constraint is now shifted to somewhere else, i.e. to some other module in the system. This once again means, any further efforts spent on the current module will no more translate into the overall system improvement and will only result in local optimization.

Hence one has to go back to Step I, i.e. IDENTIFY the NEW CONSTRAINT. This is an ever running process meaning **CONTINUAL IMPROVEMENT.**

About the Author

Manoj Phatak works in the field of Software Engineering with Geometric Limited. He has been associated with software product development in Geometric for about nine years, and has more than 15 years of professional experience spanning manufacturing engineering and software development. He can be reached at manoj.phatak@geometricglobal.com

Disclaimer

Author and Geometric Ltd respects the Intellectual Property Rights for every source it refers to. Care has been taken to ensure that credits are mentioned. However, it is possible for some of the things to be overlooked. If any such thing is observed, it is purely incidental and it is a sincere request to bring it the notice of the author.

References

1. Fragility: http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
2. Technical Debt: http://www.martinfowler.com/bliki/TechnicalDebt.html

   http://www.c2.com/cgi/wiki?TechnicalDebt
3. Software Entropy: http://en.wikipedia.org/wiki/Software_entropy
4. The SEAM Model:

   http://ptgmedia.pearsoncmg.com/images/0131177052/samplechapter/0131177052_ch04.pdf
5. “Theory of Constraints” conceived by Dr. Eliyahu M. Goldratt.

   http://en.wikipedia.org/wiki/Theory_of_Constraints
6. OOAD Metrics: http://www.objectmentor.com/resources/articles/oodmetrc.pdf
7. Cyclomatic Code Complexity: http://en.wikipedia.org/wiki/Cyclomatic_complexity
8. Refactoring: http://en.wikipedia.org/wiki/Code_refactoring

About Geometric