# Breakdown Model:

## A Disruptive Software Development Lifecycle for Fault Tolerant Software Systems

**Vaibhav Prakash, University of Texas**
**Danny Sunderesan, University of Texas**

**Abstract.** The software development lifecycle is the most important part of Software Engineering. It determines the outcome of an idea into a tangible software. Here we present a variant of the Harmony process, the breakdown model which focuses on not only developing software but deleting all possible scenarios for failures in each phase of the development process. This framework is adaptable with existing software development lifecycles.

### Introduction

Traditional software development lifecycles follow 7 core activities. They are requirements, design, construction, testing, debugging, deployment and maintenance. Naturally, apart from the requirements and testing phase, all other phases concentrate on building the software. In the requirements phase, some teams calculate the risk management which deals with the possible failure scenarios and in testing which deals with finding the loop holes based on a multitude of input values and boundary value working environments. The core idea of all software development lifecycles is to build software and not actually break it down. We believe that this is the main reason for the declining quality of software. None of the models build and destroy the software in parallel. It is quintessential to factor into our equations of how our software can fail in each phase while we are building the same. The breakdown model does exactly this—build and destroy software in parallel. Destroying software is as important as building it. Only when we understand all possible failure scenarios can we truly understand how to build software which is resistant to failure in each phase of the development lifecycle.

### Methodology - Breakdown model

The normal software lifecycle architecture involves the four core parts of a software project lifecycle:

- Analysis (Requirements definition, Iterative prototypes, Object Analysis)
- Design (Architectural Design, Detailed Design)
- Implementation (Translation, Unit Testing)
- Testing (Integration testing, Validation testing, Increment Review)

The breakdown model goes a step further and adds the following addition to the process

- Analysis and Anti-Analysis
- Design and Anti-Design
- Implementation and Anti-Implementation
- Testing and Anti-Testing

### What is Anti-Analysis?

In order to understand what anti-analysis is, we will first see what analysis means. Normally, the software team goes through the requirements phase and risk management is a part of it. But, in the breakdown model, a part of the team known as the anti-team (20%-25% of the team) works in breaking down the requirement documents and tries to find flaws it in. The sole purpose of the anti-team is to find ways in which the requirements definition can be proved false. The anti-analysis team can also make the requirement definition resilient to change as "changing requirements" are the number one cause for software failure

What is Anti-Design?

The same concept applies here too. A part of the team (20% - 25% of the team) acts as the anti-team here. But, the people involved in the anti-team in the anti-analysis phase cannot be duplicated here. It has to be picked from the remaining 75% of the team. The anti-design phase works in breaking down the architecture and detailed design concepts which the team have built. The anti-design team works carefully to weed out all possible scenarios where the design will fail.

### What is Anti-Implementation?

A part of the team (20% - 25% of the team) acts as the anti-team here. But, the people involved in the above two phases cannot be duplicated and have to be picked from the remaining 50% of the team. The anti-implementation phase works in breaking down the implementation (such as test-driven development) while the software is being built. The anti-implementation team works in tandem with the implementation team to wipe out all possible failures in the code.

### What is Anti-Testing?

The remaining team members (20% - 25%), who have not participated in the above three phases come into picture in this phase. The anti-testing team does not break the software but shows if the software works for the intended purpose. Testing and re-testing only for positive values (or) working values. They can work with the customer or simulate the intended customer who will use the software.

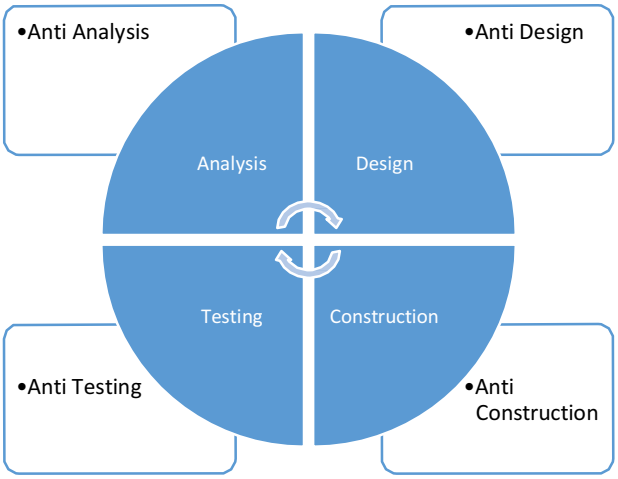


Figure 1

*Number of iterations needed – N, Number of phases in life cycle – P (4 in this case)*

*Number of anti-team members per cycle – 100/N*

*Number of anti-team members in each phase – N/P*

*Higher the value of N, higher the probability of a software system that is fault tolerant*

$N \propto$ ***Fault tolerance***

*Ideally, N-* $\infty$

*Practically, N=5 should result in a fairly fault tolerant system*

Figure 2

The breakdown model can be used in conjunction with the Spiral model to develop better fault tolerant systems. In order to determine the number of iterations needed for a complete fault tolerant system, we divide the number of iterations by 100, which gives the percentage of team members needed for the anti-team.

Let us take an example to better understand the above concept. If we want our software to be completed in 3 iterations, then we divide 100/3 which gives 33.3% (recurring). This means that in each iteration of the spiral, 33.3% of members act as the anti-team. Since there are primarily four phases of development, we divide it by 4, which gives 8.25% of the team to participate in anti-analysis, anti-design, anti-implementation and anti-testing separately.

Therefore, in three iterations, the entire team, in effect would have contributed to build and destroy software from end to end which gives a substantially higher probability of a fault tolerant system as all the team members would have contributed to it. The more you can involve people in the anti-teams, the better your chances are of building software which has fault tolerance.

Even when N=1(i.e. the most basic software development lifecycle incorporating the waterfall model with 4 phases viz. analysis, design, implementation and testing), the breakdown model results in a system which is 4 times more fault tolerant. This is because the system is tested only during the testing phase in the above traditional methodology. In the breakdown model, the system is broken down and tested for loop holes in each phase resulting in a better fault tolerant system.

## Highlights

Weeds out errors through multiple iterations and different perspectives

We found out that with N=5. A relatively high fault tolerant system can be developed

This framework can be adapted into any of the existing software development lifecycles

## Case study (Application)

We applied this to 15 software projects at the Erik Jonsson School of Engineering, The University of Texas at Dallas. All the projects were part of the coursework for graduate students. All the teams who used this framework had better fault tolerance in their software code. Although they used variants of this and incorporated the thinking into their lifecycles, it made a significant change to the product at the end compared to other teams who followed traditional lifecycles.

## Conclusion

The breakdown model is best utilized when used in conjunction with the spiral or other iterative models where repeated phases are inserted into the development lifecycle. They key aspect here is using every team member's capability to see as many ways in which the system might fail in the analysis and design phase itself. The breakdown model produces better systems when used with the simplistic waterfall model too. Lastly, from the case study it is evident that the model works as intended.

## ABOUT THE AUTHORS

**Vaibhav Prakash** is currently a Site Reliability Engineer at Microsoft Corporation in Redmond, Washington, United States of America. He holds a bachelor's degree in Computer Science and Engineering from S J College of Engineering, Mysore, India. He has completed his Master's degree in Software Engineering and Computer Networks from The University of Texas at Dallas, USA. He has published 2 papers and has done internships at The Indian Institute of Science, Bangalore, India; IBM Research and Development, Bangalore, India; Research Assistant, Multi Agent and Visualization Lab, The University of Texas at Dallas and at Microsoft Corporation, USA.

**Email: vaibhav.prakash@utdallas.edu**

**Danny Matthew Sundaresan** received his bachelor degree in Electronics and Communication from Anna University, India and is currently a Master's student in The University of Texas at Dallas, USA graduating in the field of Software Engineering. He has an experience of 6 years working as a web developer in the corporate world. He was a co founder of a freelancing web firm which brought innovative solutions to its clients during his time as an undergraduate student. His main interest involves developing user friendly tools and methods to increase the performance of the web.

**Email: danny.sunderesan@utdallas.edu**

## REFERENCES

1. Benington, Herbert D. (1 October 1983). "Production of Large Computer Programs". IEEE Annals of the History of Computing (IEEE Educational Activities Department) 5 (4): 350–361. doi:10.1109/MAHC.1983.10102. Retrieved 2011-03-21.
2. Smith MF Software Prototyping: Adoption, Practice and Management. McGraw-Hill, London (1991).
3. Dr. Alistair Cockburn (May 2008). "Using Both Incremental and Iterative Development". STSC CrossTalk (USAF Software Technology Support Center) 21 (5): 27–30. ISSN 2160-1593. Retrieved 2011-07-20.
4. Boehm, B, "Spiral Development: Experience, , Special Report CMU/SEI-2000-SR-008, July 2000
5. Boehm, Barry (May 1988). "A Spiral Model of Software Development". IEEE Computer. Retrieved 1 July 2014.
6. Abrahamsson, P., Salo, O., Ronkainen, J., & Warsta, J. (2002). Agile Software Development Methods: Review and Analysis. VTT Publications 478.
7. Hughes, Bob and Cotterell, Mike (2006). Software Project Management, pp.283-289. McGraw Hill Education, Berkshire. ISBN 0-07-710989-9
8. <http://en.wikipedia.org/wiki/Lightweight_methodology>
9. "Crystal Methods Methodology | Infolific". Mariosalexandrou.com. Retrieved 2013-07-25.
10. "Manifesto for Agile Software Development", Agile Alliance, 2001, webpage: Manifesto-for-Agile-Software-Dev
11. Coad, P., Lefebvre, E. & De Luca, J. (1999). Java Modeling In Color With UML: Enterprise Components and Process. Prentice Hall International. (ISBN 0-13-011510-X)
12. Rosenberg, D. & Stephens, M. (2007). Use Case Driven Object Modeling with UML: Theory and Practice. Apress. (ISBN 1590597745)
13. ACM Digital Library, The chaos model and the chaos cycle, ACM SIGSOFT Software Engineering Notes, Volume 20 Issue 1, Jan. 1995
14. <http://en.wikipedia.org/wiki/Incremental_funding_methodolog y>
15. Mike Goodland; Karel Riha (20 January 1999). "History SSADM – an Introduction. Retrieved 2010-12- 17.
16. <http://www.martinfowler.com/bliki/TechnicalDebt.html>
17. Jacobson, Sten (2002-07-19). "The Rational Objectory Process - A UML-based Software Engineering Process". Rational Software Scandinavia AB. Retrieved 2014-12-17.
18. Clarus Concept of Operations. Publication No. FHWA- JPO-05-072, Federal Highway Administration (FHWA), 2005