IJCSE International Journal of Computer Sciences and Engineering Open Access
Research Paper Vol.-7, Issue-5, May 2019 E-ISSN: 2347-2693

Tree Structure of Requirements and HKEH Prioritization

Saurabh Dhupkar

Mumbai, India

DOI: https://doi.org/10.26438/ijcse/v7i5.12431247 | Available online at: www.ijcseonline.org

Accepted: 14/May/2019, Published: 31/May/2019

***Abstract*—** Requirement gathering is one of the most crucial phases in the software development life cycle and the change life cycle. Any discrepancy in requirements leads to Technical Debt , has comparatively more impact than other phases of the software development life cycle and has an adverse impact on the software life expectancy. Most of the times, requirements discussed are very superficial that leads to architects and developers assuming so many details. These assumptions mostly end up being discrepancies. Therefore, having precise complete and unambiguous requirements in the initial phase makes the design and development less error prone due to reduced or minimized surprises. Thus, adequate and unambiguous requirements are the key elements of software success.

Along with un-ambiguity and completeness, correct prioritization is vital. Incorrect or misleading prioritization results into inaccurate estimation and unmanageable scope. Therefore having a common vocabulary for prioritization along with precise and detailed requirements can help keeping Technical Debt minimized and longer life expectancy of the software.

There are many techniques of the requirement gathering. In this paper, this author proposes a method for requirement structuring and prioritization.

***Keywords*—** Requirement Gathering, Requirements Prioritization, Software Engineering, Software Architecture, Software Quality, Software Life Expectancy

I. INTRODUCTION

Most of the times requirements are written down in description or story format. However, due to general human tendency this format tends become too much lengthy containing unnecessary details for some requirements, while missing out some crucial parts of requirements at other end.

Some of these missed out parts may come to limelight during design or development, which lead to confusion, misunderstanding and loss of trust on the documented requirements. However, if the missed out parts are not identified in the design phase, they can result into major deficiency in software quality and lead to Technical Debt.

On the other hand, the decreasing trend in human attention[1] span is making the people tend not to read the lengthy documents, which is in turn causing discrepancies between requirements and software.

To overcome these issues, this paper suggests a method for requirement structuring and prioritization.

II. REQUIREMENT STRUCTURING

In current methods, people tend to create documents describing the requirements. These requirements are categorized in intuitive way. However, this author suggests to structure it in a way, each requirement can be tracked individually and no requirement gets shadowed by other more important requirement(s).

A. Tree Structure of Requirements

In this author's opinion, the requirements should be represented in a tree structure, where each node represents one requirement. Each requirement node can have one or more child requirements. These child requirements should represent 'drilled down' requirements.

For example -

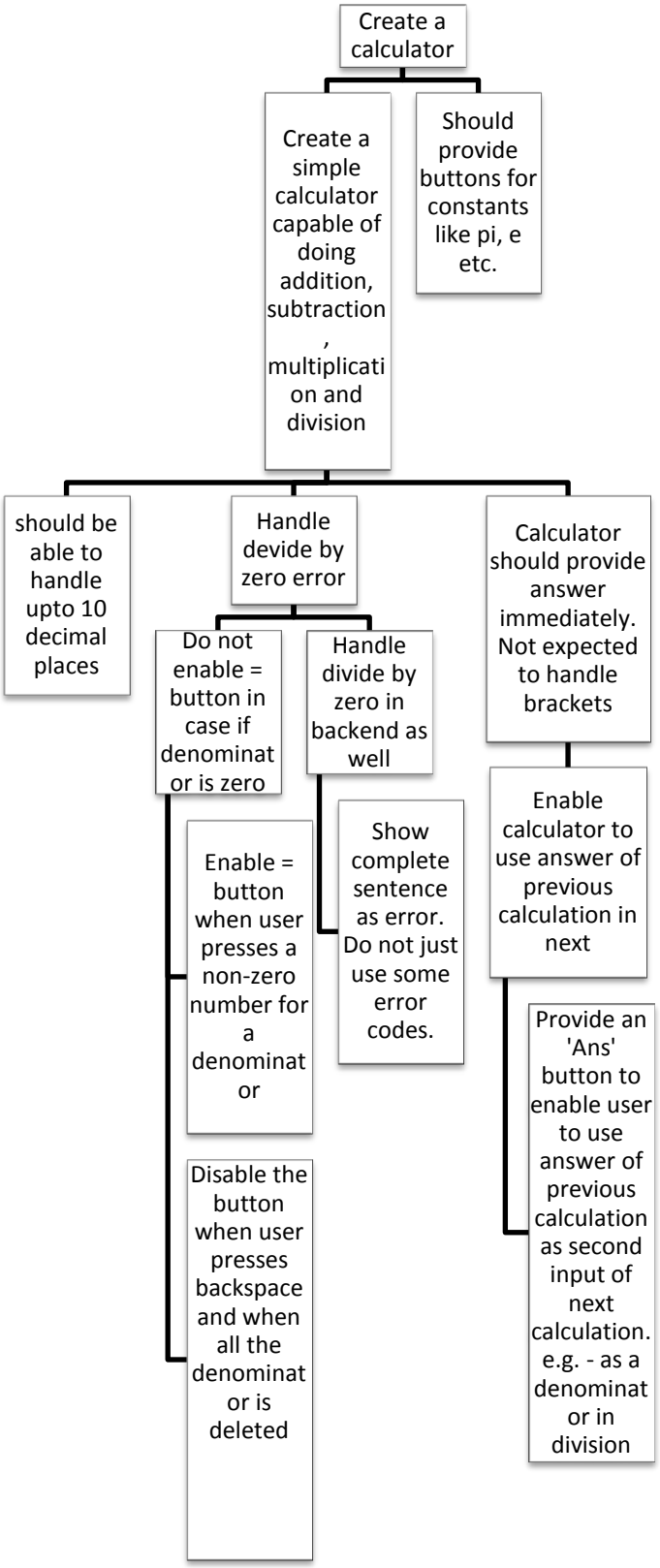

Figure 1 : Tree Structure of Requirements for a Simple Calculator

The idea is to have a very brief description of the requirement at each node level. This method doesn't expect any lengthy description using paragraphs.

In case of very large systems, the tree is supposed to be broken into multiple components based on functionality and each component is to be described using different trees. This mechanism will help to divide large systems into smaller 'neurons' like component who can talk to each other and work together to provide a set of functionalities.

This method inherently compels the stakeholders to use divide and conquer and also mandates to follow SOLID design principles. This method inherently creates low coupling high cohesion components at the requirement gathering phase itself. In case of enhancements or changes, this nature of the method keeps on showing need and places of code refactoring and in turn keep the Technical Debt minimized and maintains longer life expectancy.

B. Benefits

1) In this example, the architects and the developers can easily go through the requirements and track each requirement individually.
2) It is easier for testers to come up with better coverage as they can link a series of test cases with each requirement individually.
3) Each requirement can be prioritized separately which helps scrum master for better planning
4) It is easier to identify and settle contradictory requirements in requirement gathering itself, which helps to avoid any confusion in later stages
5) Requirement dependency is identified in the requirement gathering stage itself.
6) Each requirement gets equal 'Human Eye Time' and reduces the possibility of being shadowed by other influential (more important / more complex) requirements
7) More the stakeholder dig down in the requirements, less the probability of surprises, leads to better software quality and easier change management.
8) Pushes architects and developers for low coupling high cohesion components
9) Highlights the places that require code refactoring
10) Minimized Technical Debt and longer life expectancy

C. Limitations

1) Has a human factor. Success of method depends upon the detailing level. However, this limitation can be overcome by specifying minimum depth as it can be quantified and controlled

III. REQUIREMENT PRIORITIZATION

Along with requirement gathering, requirement prioritization is a key element. Un-prioritized or inconsistently prioritized requirements cause confusion and leads to incorrect planning.

A. Challenges with Requirement Prioritization

Many times, different stakeholder use different references for requirements for prioritization. However, when combined together, their comparative view doesn't make any sense because of their different frame of references.
It always creates confusion, whenever a child element is prioritized above its parent. It becomes hard for stakeholders to specify the case of 'whenever you are developing this, it is mandatory to develop this one too ...'
A common understanding and sense of priorities matter a lot when prioritizing requirements.

B. HKEH Prioritization Model for Requirements

This author suggests a HKEH prioritization model for all requirements irrespective of level. In this model HKEH represent human organs, so that a common understanding, vocabulary and sense can be achieved without any need of defining them individually.
HKEH stand for Heart, Kidney, Eyes and Hair.

1) Heart

Represents the highest importance of the requirement
Without this, the parent requirement / module / entire software will not be able to survive even a minute in real world.

2) Kidney

Without this, the parent requirement / module / entire software will not be able to function at all. Even though it will be able to survive for some it, it will be extremely difficult to make the system work and eventually the system will cease to exist.

3) Eyes

Without this, the parent requirement / module / entire software will be able to perform almost all the activities that it is supposed to but with some difficulty. Application will never be able to perform at its utmost performance level if this requirement is not fulfilled. However, this will not cause any issue in survivability of the application.

4) Hair

These represent pure decoration kind of requirements. Without these, the application will be able to work at their optimum performance level without any issue. It will be 'cool' to have these requirements fulfilled.

C. Tree Structure with HKEH Prioritization

If we consider same example as above -

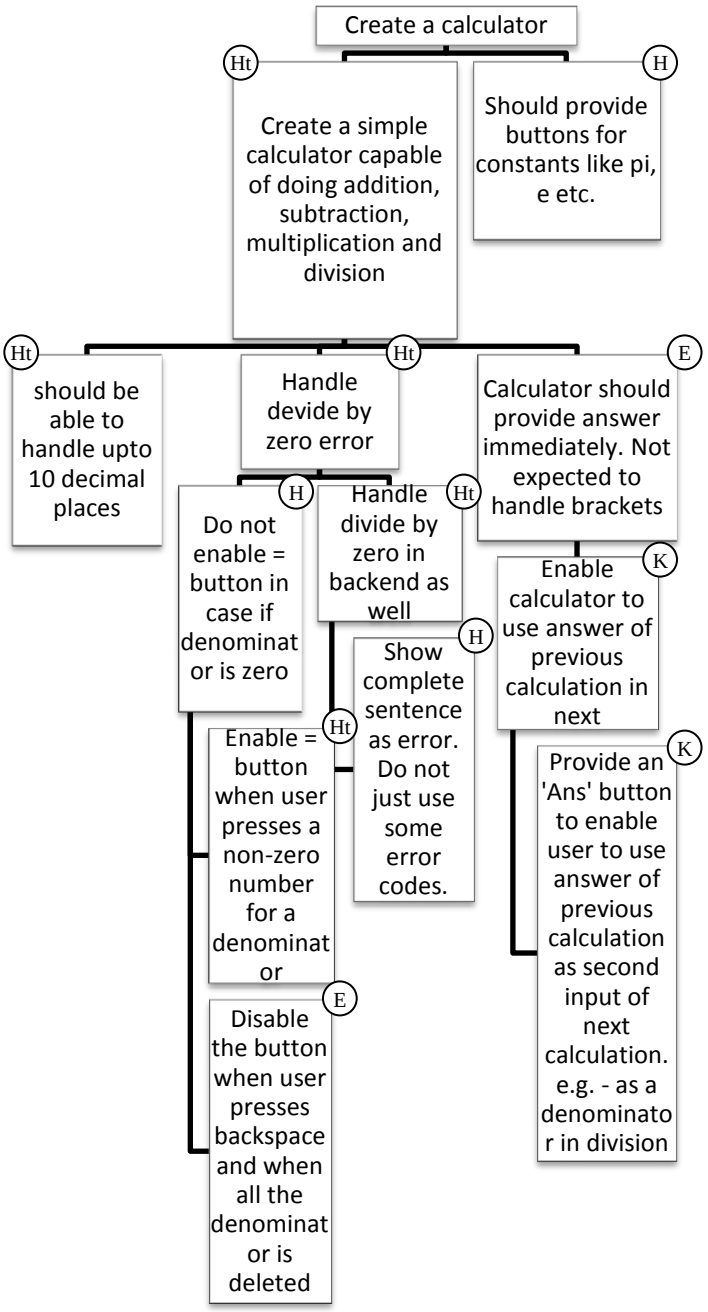

Figure 2 : Tree Structure of Requirements for a Simple Calculator with HKEH Prioritization

In the example shown above, there is a 'Hair' requirement about disabling '=' button immediately when user presses '/' button. However, it has a 'Heart' requirement about enabling it back when user presses a non-zero number. That indicates that implement disabling logic only if it can be enabled when it should be. Otherwise, no user will ever be able to use division functionality.
However, disabling it back if user presses backspace is marked as 'Eye' indicating that even if it doesnt happen, there is a 'Heart' requirement about even handling 'divide by zero' error after user presses '='. This indicates, that user might observe this as bug, but will still be able to use it. It is marked as 'Eye' and not 'Hair' because, a bug observed by user results into questioning the 'Reliability' of software which is the top of the Common Critical Software Quality Attributes[2].

1) Benefits

1) Provides clear and easy understanding without any need to define the priority levels.
2) Shares a common understanding and sense about each requirement among all the stakeholders
3) Helps to gain consensus about prioritization of all requirements and highlights any discrepancies in early stage.
4) Provides a quantifiable method for managing each development iteration.
5) Helps in efficient and effective scope definition.

2) Limitations

1) In case of huge systems, it highly depends on how the trees are being represented and how they are functionally grouped.

IV. ESTIMATION

Tree structure not only helps in understanding and prioritizing requirements but also helps to identify complexity of change, estimation and solution.

Whenever application needs to undergo some change, the probable solution should fit in current tree structure of application. If it doesn't fit, developers and architects can plan to mould current tree structure in such a way that it can absorb the new change. This can help in avoiding ECTD[3].

Management can include this moulding activity in estimation and can schedule it accordingly.

Before starting the development all the stakeholders get to have a common understanding of amount of work, complexity and estimate.

V. METHOD

This author proposes following method for effective requirement gathering, structuring and prioritization.

A. Meetings

1) Requirement Gathering Meetings

a) Requirement Identification Meeting

This is the first meeting where all business users are expected to come together and discuss the requirements among themselves. They are expected to brainstorm internally and come up with a list of requirements in a common human language.

In this meeting, all business users finalize a subset of themselves as representatives who will discuss these requirements with Business Analysts and Technical Architects.

b) Requirement Introduction Meeting

In this meeting the representatives of business users share their requirements with Business Analysts and Technical Architects and answer the queries of BA and Technical Architects.

2) Requirement Structuring and Prioritization Meeting(s)

This is a series of meetings in which BA and Technical Architects share the Requirement Tree along with HKEH prioritization with the business representatives, where they discuss each node of requirement tree for not more than two minutes.

The requirement tree and nodes are displayed using cards pinned on a board and relations are shown using threads.

In case of very huge systems, BA and Technical Architects can discuss one tree per meeting.

These meetings are to be held until a common understanding is achieved

3) Scope Meeting(s)

Similar to previous, these are also series of meetings where based on previous discussions, the scope of iterations or phases of release are discussed.

BA and Technical Architects share the estimation and proposed schedule along with budget.

These meetings are to be held until a consensus is achieved on plan.

B. Display

This author proposes to have a open or private wall, based on sensitivity of requirements, for each technical team. On these walls, these requirement trees are to be displayed through out the project.

The requirement tree and nodes are displayed using cards pinned on a board and relations are shown using threads, where the colour of requirement card indicates its priority and a mark showing release iteration / phase / version number.

This kind of displays are to be used in periodic status calls like scrum

VI. RESULT

With this requirement structuring, the Bas, architects, developers and testers minimize a possibility of missing out any requirement at any phase in software life cycle.

Requirement structuring helps testers to come up with higher functional coverage than traditional method. Instead of reading lengthy documents with long explanations, all the stakeholders get an opportunity to have a precise and complete set of requirements along with complete and common understanding of their dependencies.

Requirement prioritization using HKEH method gives everyone a common vocabulary and understanding about criticality of requirements and their dependencies. It also eases the scope definition and versioning.

VII. CONCLUSION

'Divide and Conquer' is not only for problem solving, it can also be used to understand the requirements. Unless the requirements are clearly understood, irrespective of testing quality and testing efforts, application will always end up with huge technical debt and higher impact.

This method for requirement structuring provides a method to avoid missing out 'too obvious' requirements which later on turn out to be 'not so obvious'.

Life expectancy of software basically depends upon how good it is understood by architect(s), developers and testers. This method will help to understand the application easily and priorities of functionalities, which will lead to better quality software with higher life expectancy.

As success of a software is not a Boolean but ratio of length of its earning phase by total life span.

REFERENCES

[1] Microsoft Corporation, Canada, "Microsoft Attention Spans Research Report", *www.scribd.com.* [Online] Spring 2015

[2] S. S. Dhupkar, "*Measuring Software Life Expectancy*", International Journal of modern Trends in Engineering and Research, Vol. 3, Issue 10, pp. 178-184, 2016

[3] S. S. Dhupkar, "*Technical Debts, Impact and Settlement*", International Journal for Research in Applied Science and Engineering Technology, Vol. 5, Issue 11, 2017

Authors Profile

Mr. S S Dhupkar pursed Bachelor of Engineering from University of Pune, India in 2008 and Master of Technology from Birla Institute of Technology in year 2016. He is a freelance research enthusiast and has published 5 research papers in reputed international journals and these are also available online. His main research work focuses on Software Architecture and Software Quality.

