International Journal of Pure and Applied Mathematics
Volume 119 No. 10 2018, 643-656
ISSN: 1311-8080 (printed version); ISSN: 1314-3395 (on-line version)
url: http://www.ijpam.eu
Special Issue

Effective Management of Various Forms of Creeping Featurism -"A Little More", But Not Anymore!!

[1]Keerthana G Raman, [2]Meenakshi and [3]V.R. Rajalakshmi
[1]Department of Computer Science and IT,
Amrita School of Arts and Sciences, Kochi,
Amrita Vishwa Vidyapeetham, India. gramskeerthy@gmail.com
[2]Department of Computer Science and IT,
Amrita School of Arts and Sciences, Kochi,
Amrita Vishwa Vidyapeetham, India. meenu061295@gmail.com
[3]Department of Computer Science and IT,
Amrita School of Arts and Sciences, Kochi,
Amrita Vishwa Vidyapeetham, India.rajiprithviraj@gmail.com

Abstract

It is very prevalent in IT sectors that the programmers want to do the best of the possible job to deliver the most desirable product to their customers, thus winning fame, fortune and more expanded offers. Once the programmers have configured the result that can grant all that it is expected to do far before the project timeframe, the customers and developers envisage additional capability that can be obtained from the solution which result needing additional engaging for the development. This will lead to an overly complex software product that caters for various requirements but are not simple and easy on the clients on usage of the solution. Unnecessary "little mores" that are added to make the software perfect actually brings about characteristic crawl, "Feature Creep. Uncontrolled featuritis might prompt growth creep known as "Scope creep"- extension for a task growth. This paper presents an approach to reduce feature creep to a certain extent as complete avoidance may not be reasonably expected. To ensure that those necessities which are highly required, we might select them by applying some selection strategies rather than by applying the conventional requirements freezing and code freezing approaches or any other follow up feature reduction methods. We have proposed a proactive technique that can be implemented along with the conventional SDLC processes of software development. This system deals with all those forms of creep that can occur in each of different phases of development cycle.

Key Words: Black swans, feature creep, ratchet effect, scope creep, software bloat.

1. Introduction

The vast majority of the companies have right away acted with the agile programming or the lean production for a past couple of decades. The primary concern of the agile methodology is that the software results are conveyed in a shorter duration of the time and adjustment with the evolving condition is reasonably expected. Even though the agile methods are implemented, the developers have not completely overcome those hidden risks such as the requirements or scope creeps and most of the time they are gone unnoticed. Thus the major issue faced by the developers in developing the software is creeping featurism.

Feature Creep is Brought on by the Following Causes

1. Poor definition of project goal.[4][Gurlen,2003]
2. Projects are driven by the client's endless wish lists which may go beyond boundaries of the scope of the project.
3. They concentrate more on the capability rather than on its usability proposed by [7][Thompson, 2005].
4. Customers are pulled in by lots of functionalities.
5. The level of uncertainty of a project being developed is very high.
6. The developers focus more on perfectionism.

Uncontrolled feature creep leads to kludge, software bloats [5][Marciuska, 2005]. They often prompt overwhelming costs and schedule overruns. Such software requires higher hardware capacities for their execution. Thus the product cannot be improved in a core market over some period of time thereby a non-extendable software product would become obsolete which then points to its decline as additional features added during the software life cycle will lose its value by time. The more the features are incorporated the more it influences the usability of the software. It will be ideal to concentrate on the usability of the software as opposed acknowledging its capabilities.

Currently, we have software development techniques which try to solve the feature creep problem by centring ahead the features that have the highest noteworthy feature value[5] [Marciuska, 2005]. This methodology is followed by the start-up companies, but not all of them use this method from scratch. The main shortcoming of this strategy is that the exact value of the feature is difficult to be discovered and calculated. It is very much necessary that we require some fixed set of steps that can be applied along with software development life cycle processes to forestall the occurrence of the featuristis. A feature creep could happen on three possible phases of the software programming life-cycle namely, the requirements elicitation, software designing and the software development and implementation. Many researchers have proposed methodologies for feature reduction, i.e., eliminating all the extra features which have the lowest feature

value.

We have proposed a strategy that prevents these extra features to a certain extent from creeping in during the processes of development.

2. Related Work

THOMPSON, HAMILTON, and RUST[2005]. Presented a paper mainly that focuses on the emphasize given to the complexity of the deliverable product rather than its usability. It says, as the customers wish to choose an overly complex product it results in an imbalance in the functionality of the product and its usage is focused. They have inferred results on the implications that have emerged from users perspective in choosing the number of features. This paper mainly concentrates on the hardware products.

MARCIUSKA, GENCEL, WANG, and ABRAHAMSSON[2005], their paper presents a notation to mitigate the unnecessary features in the system. The proposed notation, known as feature usage diagram visualizes the features with its value, location in the system along with their usage interpretations and their interdependency with other features. This paper has proposed a feature reduction technique in a software rather than a management technique.

Gurlen [2003] has concentrated on the creep that occurs in the later part of SDLC, scope creep. He has brought out the reasons why a scope creep emerge in systems, their consequences, how to prevent them. He also points when a scope creep can do good to a project.

3. Proposed Work

Feature creep in general is one of the standouts amongst the number of reasons projects ship late. The main reason is that people want more and more great features in their projects. These little additions creep into your product until they devour the entire part of the project time frame. The vicinity of a single creep will lead to greater amount of its sorts during the rest of the development time. The scope creep or mission creep defined as the never-ending expansion of the project scope [4][Gurlen,2005], which is a consequence of feature creep, in turn, introduces the ratchet effect, the inability of a system to reduce its scope once it expands. It is ideal for software development to implement a proactive measure to avoid these creeps than implementing a reactive technique to reduce those extended features.

A conventional software development life cycle figure.(1) includes Requirements Elicitation, Software Design, Software Development and Implementation, Software Testing, Software Release, Operation, and Maintenance phases. We have designated the creeps in different phases in accordance with their occurrence during a project's development. The creep that occurs in the earlier stages, i.e., Requirement Elicitation phase of the development cycle is known as "Feature Creep or Requirement Creep". While the ideas that creep in during the later stages, the design and development

phases are recognized as the "Scope creep" or “mission creep”.

These three creeps are handled during their corresponding phase's execution. Each of these three phases is explained in detail.

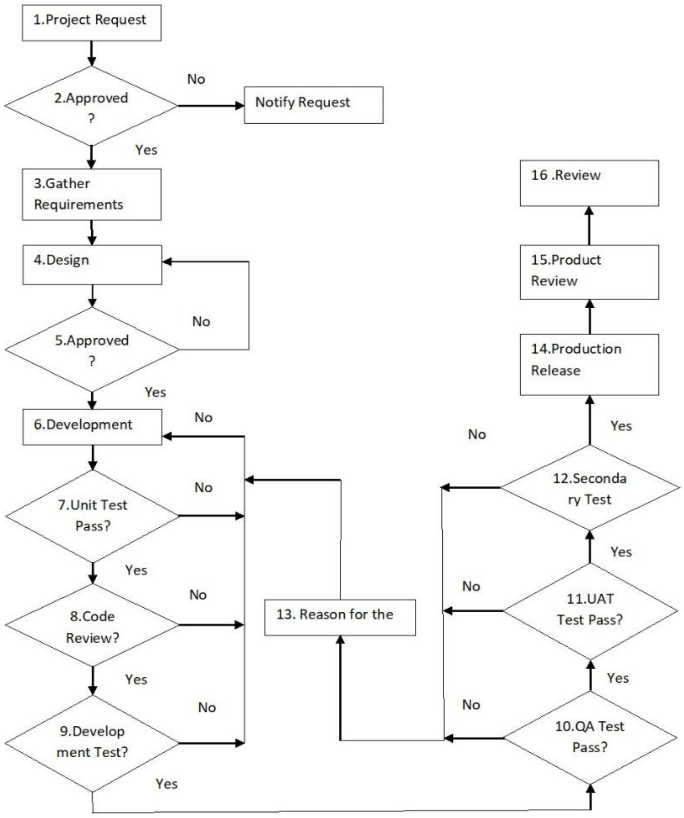

Figure 1*:* Conventional SDLC

Phase 1–Requirements Elicitation

The Requirement Elicitation is the first and foremost phase of SDLC. The development team and the clients sit together and discuss every requirement they think is necessary and gather all such possible requirements regarding what the expected system must do and must not do. Usually, after gathering the requirements, they are analyzed by the team and a requirements review is conducted. Finally, they are recorded as a document known as Software Requirements Specification (SRS) that acts as an agreement between the clients and the developers. There are tendencies that the customers may raise with a new requirement any time after the review or the developers may bring out new necessities during documentation. The developer is left with the choice of either implementing these ideas or ignore them. In such situations, it is a common practice to implement all these new requirements along with the existing requirement; this may prompt a feature creep. Complete rejection of the new

requirements cannot be done in a go. We will then perform a requirement selection to select the best requirement from the new set.

So, these requirements can be subjected to a two-layered iterative filtering technique which has the following layers.

1. The Prerequisite usage grid. figure (2)
2. The Effort matrix. figure (3)

Layer 1-Prerequisite Usage Grid

This is the first layer of requirement filtration for visualizing the features. The prerequisite usage grid aka feature usage diagram.[5][Marciuska,2005] is used to select those requirements that are used most frequently by most of the users. This is a grid that locates each requirement to each of its corresponding cells depending upon its usage and preference.

The grid is mainly associated with two parameters, namely

1. The number of users: describes the total count of the users who use that particular requirement.
2. The frequency of time of use: Describes how often a feature is used.

Most preference is given to those requirements that fall in the top rightmost-top cells ” all of the people-all of the time” i.e., the requirements that is used by all the users always. While least significance is give to the leftmost-bottom cells i.e.,” few of the people-few of the time”, those requirements that are hardly or seldom used by least number of people.

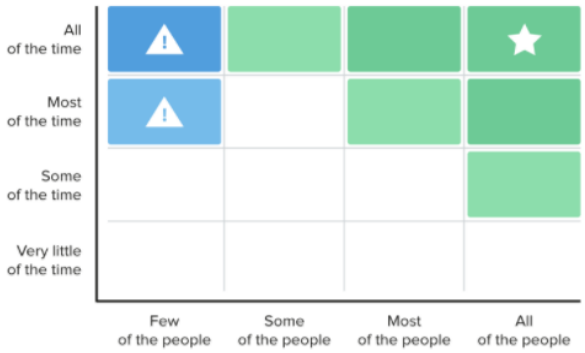

Figure 2: Feature Usage Diagram

Layer 2-Effort Matrix

The selected features are then put into the Effort Matrix (figure 3). The effort matrix is a second layer for screening the requirements. The effort matrix has two parameters. They are

1. Value: Describes the value of a requirement.
2. Effort: The effort required to implement the requirement.

The requirements with high value are worth including in the software. So we will consider the requirements that have high value and are effortless in implementing. The requirements designated in the low-value grids are considered less. The effort-value matrix along with its rank matrix is shown as follows.

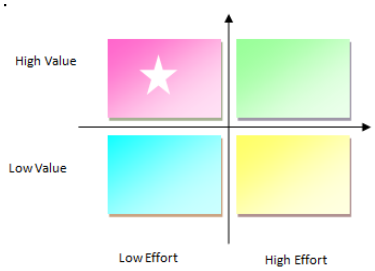

Figure 3: Effort Matrix

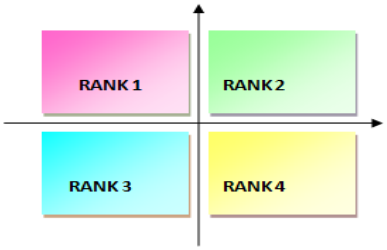

Figure 4: Rank Matrix

The requirements with the highest ranks are considered most from the rank matrix (figure 4).

Once the requirements are screened from both these layers, we get those best requirements which are used the most, have high value and require comparatively less effort to implement. The following flowchart (Figure5) elucidates the entire process of the requirement elicitation along with the activities that are prone to the occurrence of the requirement creep and the solution to avoid them. The yellow bubble indicates the probability of occurrence of requirement creep during a particular activity to which it points to. Here, the bubble points to SRS, meaning new features can be documented without analysing them.

Phase 2-Design Prototype

The transformations of all the previously gathered requirements into a prototype with which the developers implement the software are done in this phase. The applications, user and system interfaces, database and networks are implemented in the prototype. This phase takes the deliverable of its previous phase, the SRS, and transforms it into a logical structure which describes a detailed set of specifications that satisfy those requirements and can be

implemented in a programming language. As a result of the design phase process, the design specifications are then documented as Software Design Document (SDD) by the developing team.

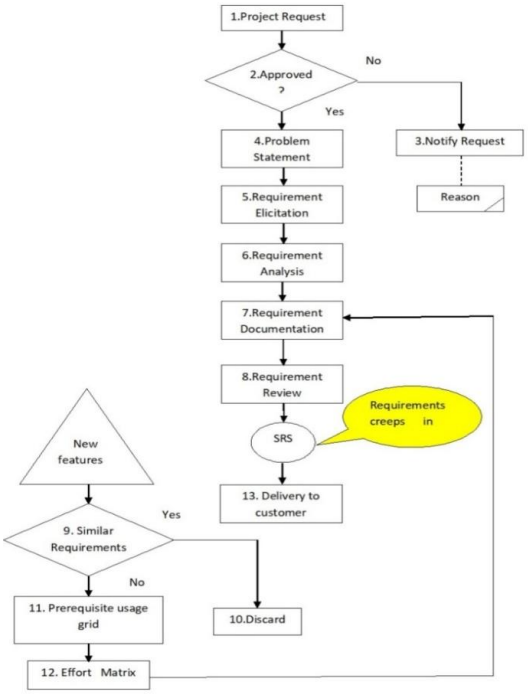

Figure 5: At the Requirements Elicitation Phase

Extra features can creep into the document at any point in design documentation thus deviating from the scope of the project. A creep at this point of software development is more relevant if we address it as feature creep. To deal with these features, it is appropriate to consider the complexity of the solution. The complexity matrix (Figure6) thus contains two parameters namely,

1. Level of improvement: Describes how much is the scope for improvement in the project.
2. Complexity of the solution: Describes how complex the solution is.

The following figure is the complexity matrix.

Figure 6: Complexity Matrix

Each feature is visualized and calibrated in appropriate cell in the complexity matrix and the best ones are chosen. Initially the level of improvement on implementing a particular feature is envisaged and the complexity of that solution when that feature is implemented is calculated. The most preference is given to the features that if included yields high level of improvement in the design and has the least level of complexity. Such features are considered and implemented first. While the features that increases the complexity on implementation are not considered.

Perfectionism, "adding more and more features to simplest software which can possibly give desired result just" noted by [3][Fitsilis, 2006] to make it perfect is one way of increasing its complexity, thus deviating from its scope. Hence we have managed the complexity of the software during its design to keep its scope intact.

Below flowchart(Figure7) shows the complete avoidance from scope creep in design phase. Here, the bubble points to the building prototype activity i.e., new features might be incorporated in the design by the developer without even actualizing them.

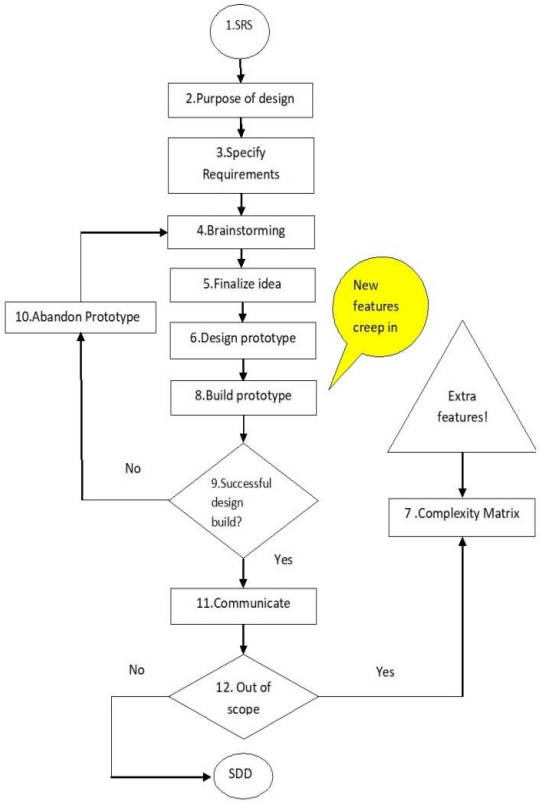

Figure 7: At the Design Phase

Phase 3-Development and Integration

Once the SDD is received by the development team, they divide the work into smaller modules and assign them to different programmers, which known as task allocation. Each programmer then codes his module in a suitable programming language. Thus this phase involves a large number of developers working to achieve a common goal. To make the module better and better, the programmer may contribute many extra ideas. Or while coding a piece, the developer may infer from the current state of module development, some new features should be added to the module to sustain the module that is being implemented. These new ideas may seem to be easy and cost-effective implementing alone, but they ruin the entire program when these coded modules are integrated and pointing to overwhelming cost. Moreover, this simply leads to the "black swan"-events that come as a surprise and has major consequences, in future. Implementing all the ideas that creep in during development phase my not yield a positive result. This too may lead to scope creep. So at this point of SDLC, we will associate the features/ideas with their implementation risk. Therefore we will use a risk matrix and emphasize on how crucial the features at that point of development. Again the features are subjected to a two layered screening process.

1. Risk Matrix (figure 8): Determines the likelihood of occurrence of a risk and its degree of intensity of consequences.
2. Eisenhower's Matrix(figure 9): Determines how important a feature is at that point of development and how fast it has to be implemented.

Layer 1-Risk Matrix

The risk matrix concentrate on the frequency of occurrence of a risk and its significance. A risk of implementing a feature along with its consequence is calculated and is delegated in appropriate cell of the risk matrix. A feature that has the least likelihood of occurrence of risk with the least consequence "unlikely-insignificant" is considered for implementation. While features with highest likelihood of occurrence of risk and the great consequences "almost certain-catastrophic" are not considered.

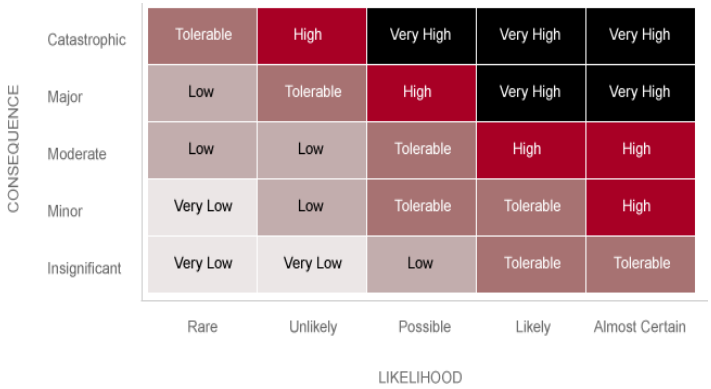

Figure 8: Risk Matrix

Layer 2-Eisen Hower's Matrix

Each new features/ideas obtained as a result from the risk matrix is exposed to the Eisenhower's matrix, which emphasizes on how important a feature is to the software module at a particular point of time frame and how fast it has to be implemented. Here, initial emphasis is given to the risk analysis, as implementing all the ideas that creep in to the software piece will make it more prone to risks or unexpected events to pop up in the future, i.e., black swans as referred earlier and will ruin the entire work. Once the risks are analyzed the results are the put into the Eisenhower's matrix to find which among the features (with low risk) have more priority and should be implemented faster. The Eisenhower's matrix is shown below.

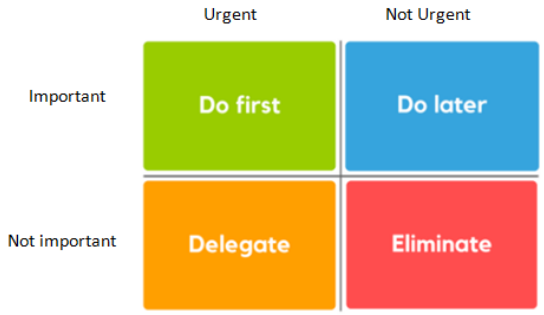

Figure 9: Eisenhower's Matrix

The following flow chart (Figure 10) describes the entire Development phase in dealing with the creep, we can see the bubble points to the code generation activity, the developers who generate code for their module mad add new ideas to make their code even better that the one before, they do this without visualizing their final product which will be highly sophisticated, complex and not user-friendly.

4. Conclusion

One of the hidden challenges on agile programming is the gold plating in other words the creeping featurism. Most of the developers fail to foresee these small features that creep in, which eventually turns the software into a big failure. Even if the development of the creep chunk seems to be easy and is within schedule, they certainly tend to lag at a point when they have to be integrated to the main module or software piece, just like the tip of an iceberg. The creep can arise in software in multiple forms at different phases. We have dealt with three basic forms of creep at three main phases. While the others emerge from the former. Thus, this paper extends the scope of dealing with mitigating the occurrence of other forms of creeps such as hope creep, effort creep, bug creep, size creep(-an extension of scope creep as proposed by Einen,2011) as all of these are interrelated. Once these creeps are reduced, they lower the chance of

further risks such as software bloat, black swans. Not always feature creeps are bad, there are many projects to which they have done good too. Above all, everything rely on our hands to judge if a creep is to be used as a utility or to be scored out.

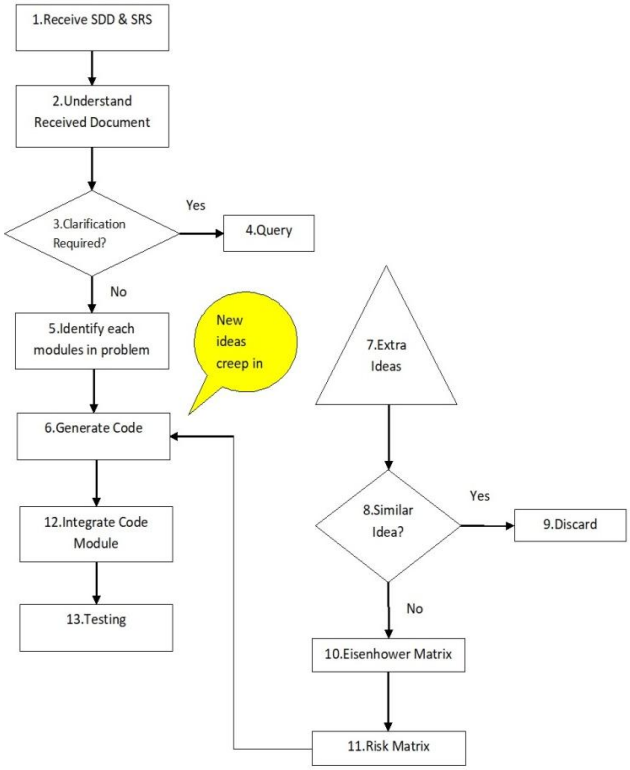

Figure 10: At the Development Phase

References

[1] Cohen S., Dori D., de Haan U., A software system development life cycle model for improved stakeholders' communication and collaboration,International Journal of Computers Communications & Control 5(1) (2010), 20-41.

[2] Einen R.D., Managing "size creep" in software development projects, Global Congress 2011—North America, Dallas, TX. Newtown Square, PA: Project Management Institute (2011).

[3] Panos Fitsilis, Vyron Damasiotis, James F. O' Kane, Scope management complexity in software projects, Conference British Academy of Management-BAM, At Belfast, North Ireland (2014).

[4] Stephanie Gurlen, Scope Creep (2003).

[5] Marciuska S., Gencel C., Wang X., Abrahamsson P., Feature usage diagram for feature reduction, International Conference on Agile Software Development (2013), 223-237.

[6] Christos Svitis, Lean Software Development, Theory validation in terms of cost-reduction and quality-improvement, Bachelor of Science Thesis in Software Engineering and Management (2015).

[7] Thompson D.V., Hamilton R.W., Rust R.T., Feature fatigue: When product capabilities become too much of a good thing, Journal of marketing research 42(4) (2005), 431-442.

[8] Tushar Sharma, Process Improvement: A Process Owner's Dilemma, ISixSigma (2009).

[9] Wikipedia- feature creep:

https://en.wikipedia.org/wiki/Feature_creep

[10] Wikipedia- scope creep:

https://en.wikipedia.org/wiki/scope_creep