CHAPTER 2

LITERATURE REVIEW

This chapter presents a comprehensive review of the available literature related to defect removal and software quality assurance of software. The literature is classified under two heads. The first one is related to the software defect models and the second one deals with software quality assurance models.

2.1 STUDIES ON DEFECT REMOVAL AND PREDICTION OF SOFTWARE PROJECTS

In this section various models related to software project defect removal are presented.

2.1.1 Software Development Life Cycle Model

Mercedes et al (2001) highlighted the dynamic models that use a number of parameters and functions to characterize the organization environment. Having the initial values for all these parameters and functions is a requirement to run simulations of it. Unfortunately, the absence of historical databases and the uncertainty of software make it difficult to set these initial values. So a simplified model, with a smaller number of parameters and functions, but still producing useful results, is necessary. To obtain this reduced model, a theory of the simplification of dynamic models has been applied.
Nguyen (2006) examined some issues that hinder the effective management of, and decision-making on, quality software development process and products delivery by practitioners. The study generated a decision model for managing software developments. The model used four concepts: mappability, accountability, interoperability, and controllability in decision-making which is assumed to be based on a set of indicators that link task status of the development process and its quality assessment to the responsible authorities. The quality of the tasks, and hence, of the deliverables is measured using four attributes: completeness, correctness, consistency, and compliance.

Arilo (2009) discussed the software organizations already exploring model-based testing in their software, as it can provide information regarding different MBT approaches and their impact on the software testing process. This support is related to the goals set in the CMMI process areas: Risk Management and Decision Analysis and Resolution.

Xiaojian (2011) explained the model-driven development of complex software-intensive systems; it was the first class issue of how to capture the software models in the very beginning of development and the focus was on the domain of automotive software and on exploring how to capture software models from requirement documents. To this end, the researcher mainly investigated these three closely related problems: what requirement information should be elicited from requirement documents, how to organize this information as aspectual models, and how to integrate the aspectual models together to form a complete specification of requirements. This work will help software analyzers to decompose complicated requirement documents into separated concerns, and organize essential requirement information as rigorous models, which will both facilitate
simulation and verification of requirements and set up the starting point for the model-driven development (Masood Uzzafer 2013).

Rakesh Rana(2014) stated that SRGMs (software reliability growth models) provided empirical basis for evaluating and predicting reliability of software systems. When using SRGMs for the purpose of optimizing testing resource allocation, the model’s ability to accurately predict the expected defect inflow profile was useful. Using defect inflow profiles from large software’s from Ericsson, Volvo Car Corporation and Saab, he evaluate commonly used SRGMs for their ability to provide empirical basis for making these decisions.

Mohsen Asadi(2014) presented a novel approach for handling inconsistencies resulting from the customization of business process models, which were originally derived from references process through configuration procedures. By automatically identifying inconsistencies, he was able to find if customized process models were consistent with the behavioral and configuration knowledge encoded in reference process models.

Thomas (2014) found topics to be effective tools for structuring various software artifacts, such as source code, requirements documents, and bug reports. This research also hypothesized that using topics to describe the evolution of software repositories could be useful for maintenance and understanding tasks.

MirkoPerkusich(2015) described several software process models and methodologies such as waterfall, spiral, and agile. Even so, the rate of successful software development is low. Since software is the major output of software processes, increasing software process management quality should increase the chances of success. Organizations have invested to adapt
software processes to their environments and the characteristics to improve the productivity and quality of the products.

### 2.1.2 Software Testing Models

The following literature describes various software testing models:

Ytzjak(1990) has explained that defect removal is ruled by the “laws of the physics” of defect behavior that control the defect removal process. The time to defect detection, the defect repair time and the factor of introduction of new defects due to imperfect defect repair are some of the “constants” in the laws governing defect removal. Test coverage is a measure of defect removal effectiveness. A birth-death mathematical model based upon these constants was developed and used to model field failure report data.

Keiller(1991) highlighted the number of failures occurring during a finite future time interval being predicted from the number of failures observed during an initial period of usage by using software reliability growth models. Two different methods for using the models were considered: straightforward use of individual models (simple models) and dynamic selection among models based on goodness-of-fit and quality-of-prediction criteria (super models). Performance was judged by the relative error of the predicted number of failures over future finite time intervals relative to the number of failures eventually observed during the intervals.

Lennselius(1987) emphasized on the fault content and reliability estimations, but software metrics were needed for all aspects concerning quality, resources, economy etc. Fault content estimations are very important when developing software products because time schedules, quality and the
overall economic result of the products are often highly dependent on the fault content.

George (1995) discussed errors to be made early in the development and testing cycle of a product are of extreme importance to software developers because it can help to

(i) Manage the various phases of the development Cycle.

(ii) Determine the amount of time that needs to be devoted to testing.

(iii) Determine the reliability of the product when it is released

(iv) Estimate the cost of the product, etc.

Pasquini (1995) presented the activity undertaken to validate the model proposed in (Pasquini & Di Marco 1992). The results showed that the model was conservative (that is, the actual reliability is always higher than estimated) during the first phases of testing, but fitted the actual data after the execution of accurate and extensive testing. The information collected during testing was used to refine and improve the estimate. In addition, the validation showed that a significant effort was needed for a complete application of the model.

Ioana Rus et al (1999) explained the concept of process modeling and simulation was first applied to the software development process by Abdel-Hamid (Abdel-Hamid et al 1991). Others (Madachy 1996; Tvedt 1996) produced models as well, but there was little evidence that they had been successfully applied to real software developments. The main goal for software developers and an important marketing consideration for organizations is to meet the requirements of the time and cost. However,
quality by itself is not a strategy that will ensure a competitive advantage. There are other drivers like budget and delivery time that must be considered in relation to quality. Achieving the optimal balance among these three factors is a real challenge for any manager (Boehm 1996). The modeling approach here focused on software reliability, but was just as applicable to other software quality factors, as well as to cost and schedule factors. The process simulator was developed as a part of a decision support system for assisting managers in planning or tailoring the software development process, in a quality driven manner.

Fenton & Neil (1999) stated that many organizations wanted to predict the number of defects (faults) in software systems, before they were deployed, to gauge the likely delivered quality and maintenance effort. To help in this numerous software metrics statistical models had been developed, with a correspondingly large literature. They provided a critical review of this literature and the state-of-the-art. Most of the wide ranges of prediction models used size and complexity metrics to predict defects. Others were based on testing data, the “quality” of the development process, or took a multivariate approach. More significantly many prediction models tended to model only part of the underlying problem and seriously misspecify it.

Sun Sup So et al (2002) inspected metrics to identify characteristics of effective, questionable, or marginal inspection sessions. Classification was based primarily on preparation effort and inspection rate. Such information was used to improve inspection process by issuing guidelines on the amount of preparation effort needed prior to formal inspection meetings and inspection rate as goals. Furthermore, they used inspection data to estimate density of errors remaining in the code to help managers decide whether or not re-inspection was warranted.
Bahador Ghahramani (2003) has highlighted the implementation and utilization of the software reliability analysis model (SRAM) that provides analysts, software engineers, and systems analysts and developers (SA&D) the means to predict, estimate, and measure rate-of-failure occurrences in software (including firmware). Rate-of-failure measures are understandable to system users. By implementing the SRAM in the content of software engineering, SA&D could: (1) analyze, manage, and improve the reliability of their systems; (2) balance users’ needs for competitive price, product reliability, and timely delivery; (3) determine when the software is ready to be released to the users, thereby minimizing the risks of releasing software with serious problems; and (4) avoid excessive time-to-market of the system due to system over testing.

Noore & Liang (2005) classified into two types of discrete defect removal models that consider the dynamics of code churn behavior during software testing phases under distributed software development environment being proposed. The first model was based on a sequential debugging process, while the second model was based on an iterative debugging process during each testing phase. A mathematical relationship was formulated between the number of defects detected during a testing phase and the total estimated remaining defects at the end of the same testing phase for both models.

Gunes Koru & Hongfang Liu (2005) described the effective software modules causing software failures, increase development and maintenance costs, and decrease customer satisfaction. Effective defect prediction models could help developers focus quality assurance activities on defect-prone modules and thus improve software quality by using resources more efficiently. These models often used static measures obtained from source code—mainly size, coupling, cohesion, inheritance, and complexity.
measures—which were associated with risk factors, such as defects and changes.

Luciano Baresi (2006) states that the development of large software systems is a complex and error prone process. Faults might occur at any development stage and they must be identified and removed as early as possible to stop their propagation and reduce verification costs.

Pankaj Jalote et al (2007) has explained that many software projects involve large teams that work over many months or years for building the project. During the development process, data about defects found is generally logged in a central repository. This defect data are important for management purposes as hundreds of people may detect defects over many months or years, and without logging of defects proper tracking is almost impossible. Accurate defect data are also important for making product release decisions. Consequently, defects are logged regularly and usually in a timely manner and a considerable amount of information is recorded to facilitate tracking or resolution. It is considered one of the best practices in software engineering (Brown 1996). Predicting defect prone components using past defect data has been, and still is, an active area of research and many approaches have been tried with varying success (for a survey of these models the reader is referred to (Fenton & Neil 1999; Ostrand et al 2005)). Defect data are also the primary source for many software reliability growth models which use the failure times for predicting reliability of the delivered software (for a survey of some such models, the reader is referred to (Goel, 1985; Yamada & Osaki 1985; Farr, 1996)). Defect data also form the starting point of the root cause analysis for defect prevention (Card 1998; Lezzak et al 2000; Jalote & Agrawal 2005).

Pankaj Jalote’s (2007) analyzing defect data for process understanding and improvement, however, has attracted limited attention. The
general approach for analyzing defect data for process analysis is to categorize defects in different categories and then use the distribution for making deductions about the process. A classical approach is to identify each defect the phase in which it is detected as well as the phase in which it was introduced, and then use the data to determine the defect removal efficiency of the various defect removal tasks (Kan 1995; Jones 1996). The orthogonal defect classification, besides the defect type, also uses defect trigger (which specifies what is needed for a defect to manifest itself) for categorizing defects (Chillarege 1992). The distribution of different types of defects and triggers and how it changes across design, unit test, etc., can be used to deduce improvement opportunities in different process phases. For process analysis, most of these approaches implicitly assume a waterfall type development model with process phases like specification, design, coding, testing, etc. In a complex product, the development is highly distributed with different teams often following different processes and approaches.

Ching-Pao Chang & Chih-Ping Chu (2007) dealt with, in addition to degrading the quality of software projects, software defects also require additional efforts in rewriting software and expose the success of software s. Software defects should be prevented to reduce the variance and increase the stability of the software process. Factors causing defects vary according to the different attributes, and including the experience of the developers, the product complexity, the development tools and the schedule. The most significant challenge for a manager is to identify actions that may incur defects before the action is performed. Actions performed in different software may yield different results, which are hard to predict in advance. To alleviate this problem, this study proposed an Action-Based Defect Prevention (ABDP) approach, which applied the classification and Feature Subset Selection (FSS) technologies to data during execution.
Tomaszewski(2007) has highlighted that a lot of work has been done in the area of fault detection improvement. A large portion of this research focuses on building fault prediction models. Depending on the output (the dependant variable), these fault prediction models belong to one of the following groups (Khoshgoftaar & Seliya 2003):

(i) Quality prediction models – these models attempt to quantify the quality of the code unit, e.g. by predicting the number of faults in the code unit. Examples of such models can be found in (Cartwright & Shepperd 2000; Chidamber et al 1998; Nikora & Munson 2003; Ping et al 2002; Zhao et al 1998).

(ii) Classification models – these models classify code units as fault-prone or not, i.e., they predict if the code unit contains faults. Examples of such models can be found in (Briand et al 2000; El Emam et al 2001; Fioravantiand Nesi 2001; Khoshgoftaar et al 2000a,b). The models often operate at different levels of the logical structure of the code. There are models that predict fault-proneness of classes (Basili & Briand 1996; Briandet al 1999; Cartwright & Shepperd 2000; El Emamet al 2001; Li et al 2001; Zhao et al 1998), modules (Fenton & Ohlsson 2000; Khoshgoftaar et al 2002; Khoshgoftaar et al 2000a,b; Ohlsson et al 1998), components (Ohlsson et al 2001), or files (Ostrand et al 2005).

The prediction models are usually based on different characteristics of the code units. These characteristics are commonly presented in the form of different code metrics (e.g., Khoshgoftaar et al 2000a,b; Pighin & Marzona 2005; Zhao et al 1998) or, for classes, variations of (CK Chidamber & Kemerer 1994) object oriented metrics (e.g., Briand et al 2000; El Emam et al 2001; Zhaoet al 1998). There are also studies that take historical information
about fault-proneness of code units into account (e.g., Pighin & Marzona, 2003; Pighin & Marzona 2005).

Karim (2008) highlighted the fact that the effective prediction of defect-prone software modules could enable software developers to focus on quality assurance activities and allocate effort and resources more efficiently. Support vector machines (SVM) had been successfully applied for solving both classification and regression problems in many applications. Identification of defect-prone software modules was commonly achieved through binary prediction models that classified a module into either defective or not-defective category. These prediction models almost always utilize static product metrics, which have been associated with defects, as independent variables (Emam et al 2001).

Staron (2008) stated that defects discovered during the testing phase in software needed to be removed before the software was shipped to the customers. The removal of defects could constitute a significant amount of effort to be made by managers who face the problem of taking a decision whether to continue development or shift some resources to cope with defect removal and they developed three kinds of models, based on: (i) high level progress data, (ii) using test status data, and (iii) using a subset of test status data. Models (i) and (ii) were developed using principal component analysis and multivariate linear regression, while model (iii) was developed by manual selection of variables and multivariate linear regression. The problem with using these prediction models may occur if the processes changes once or more during the lifetime.

Parastoo Mohagheghi et al (2009) performed a systematic review of studies discussing model quality published since 2000 to identify what model quality meant and how it could be improved. From forty studies covered in the review, six model quality goals were identified; i.e., correctness,
completeness, consistency, comprehensibility, confinement, and changeability. They further presented six practices proposed for developing high-quality models together with examples of empirical evidence. The contributions of the article were identifying and classifying definitions of model quality and identifying gaps for future research.

Kai-Yuan Cai et al (2010) has discussed and proposed two models for the software testing process. In the first model it is assumed that the probability of a test case revealing a failure is proportional to the number of defects remaining in the software under test, and upon a failure being revealed, debugging may make the number of remaining defects reduce by one, or remain intact, or increase by one. In the second model it is still assumed that the probability of a test case revealing a failure is proportional to the number of defects remaining in the software under test. However, there is an upper bound for the number of remaining defects. When the number of remaining defects reaches the upper bound, the probability that the number of remaining defects is increased by one by debugging is zero.

The expected behaviors of the cumulative number of observed failures and the number of remaining defects in the first model show that the software testing process may induce a linear or nonlinear dynamic system, depending on the relationship between the probability of debugging introducing a new defect and that of debugging removing a detected defect. The second-order behaviors of the first model also show that in the case of imperfect debugging, although there may be an unbiased estimator for the initial number of defects remaining in the software under test, the cumulative number of observed failures and the current numbers of remaining defects are not sufficient for precisely estimating the initial number of remaining defects.

Kai Petersen (2011) identified studies for software productivity prediction and measurement. Based on the identified studies they first
created a classification scheme and map the studies into the scheme (systematic map). Thereafter, a detailed analysis and synthesis of the studies was conducted. As a research method for systematically identifying and aggregating the evidence of productivity measurement and prediction approaches systematic mapping and systematic review were used. Results: In total 38 studies were identified, resulting in a classification scheme for empirical research on software productivity. The mapping allowed identifying the rigor of the evidence with respect to the different productivity approaches. Overall, further evidence was needed to make stronger claims and recommendations. In particular, the discussion of validity threats should become standard, and models need to be compared with each other.

Phan, Dien et al (1995) used IBM’s OS/400 R.1 to address key quality management and control issues in large developments. The major task was improving the software quality during and after development. During the development of OS/400 R.1 at IBM corporation, thousands of programmers were involved in writing and refining millions of lines of code. Such an effort would fail without good software quality management. Hence, software developers cannot make good software quality products.

Kai –Yuan Cai (1998) applied a new static model for estimating the number of remaining defects, and used a set of real data to test the new model. The new model resembled the Mills model in a particular case, and was attractive in its applicability to a broader scope of circumstances. A practical example showed that the new model could offer good estimates for a number of software defects. It was also applicable to statistical problems other than software reliability modeling. The researcher did not give a systematic review, and hence it could not be applied for estimating the number of remaining defects.
Biff (2003) compared and investigated the performance of objective and subjective defect content estimation techniques. For the validation of these techniques they conducted a controlled experiment with 31 inspection teams, each of which consisted of 4-6 persons. They reported the data from an experiment with a number of software engineering defects in a requirement document. The researcher used the relative error, a confidence level, and the correctness for deciding reinspection as the main evaluation criterion, but did not provide the major defects in the requirement document.

Chun Young (2006) applied a new method (capture-recapture Model) that estimated the number of undetected errors in complex software design documents. This method used the correlation matrix of multiple inspectors to formulate the estimation problem as a goal program. The Capture-recapture model initially used by biologists to estimate the size of wildlife population, has been widely used to estimate the number of software design errors. It shows that undetected errors are present and this leads to Software fault or failure.

Nalini Ravishanker (2008) applied the nonhomogeneous Poisson process model and the multivariate model that applied Markov switching to characterize the software defect discovery process. However, the process remained complex and hence there was an increase in failure rate.

Jef Jacobs et al (2007) studied the Defect Injection (DI) and Defect Detection (DD) influencing factors and their grouping resulting in their use in developments. To decrease the number of injected defects in a development, the DI factors could be used as areas of attention while the quality of documentation was poor leading to a lack of product quality.

Tong-Seng Quah et al (2009) studied defect tracking as a proxy method to predict software readiness. They developed the defect predictive
model that was divided into three parts: (1) Prediction model for presenting the logic tier (2) Prediction model for the business tier and (3) prediction model for the data access tier. Evaluating the software readiness was found to be very complex.

Sunita Chulani (1999) has applied the COQUALMO model to predict the defect density of the software under development, where defects conceptually flow into a holding tank through various defect introduction pipes, and are removed through various defect removal pipes. In this model, it was difficult to increase the quality without increasing the cost and time. They injected the defects and removed them, but it involved more computation time, cost and man power to predict the residual defects.

Christopher Westland (2004) found that the short software development life cycle appear to be in favor of higher rates of detection, but for some reasonable development cycle, most of the errors were found uncorrected. Short life cycles are likely to force constrained software development cycles, and are likely to exacerbate the risk from post-release defects. Defining uncorrected defects become exponentially costlier in each phase.

Turakhia Ritesh et al (2006) used statistical testing to isolate the embedded outlier population, test conditions and test application support for the statistical-testing framework and the data modeling for identifying the outliers. The identification of outliers that correlated to latent defects critically depended on the choice of the test response and the statistical model’s effectiveness in estimating the healthy-die response, but it provided low efficiency, less reliability, and the cost was very high.

Zuo Xiaode et al (2009) studied the quality prediction model and found that the number of faults was negatively correlated with the workload
fluctuation, which indicated that the quality was decreasing due to the heavy workload. Owing to the problems that occurred in the testing phase for overheads, there was a decrease in the quality of the product.

Cagatay Catal (2011) has studied the software engineering discipline, that contains several prediction approaches such as test effort prediction, correction cost prediction, fault prediction, reusability prediction, security prediction, effort prediction, and quality prediction. They investigated 90 software fault prediction papers published between 1990 and 2009. They gave a road map for research scholars in the area of software fault prediction.

2.2 STUDIES RELATED TO SOFTWARE QUALITY ASSURANCE MODELS

In this section literature related to two important models of software quality assurance of software is presented. The two models covered are Soft Quality Function Deployment and Software Failure Mode and Effects Analysis.

2.2.1 Software Quality Function Deployment (SoftQFD)

Originally all the models of quality assurance were developed for software applications. Later the same models were employed with suitable modification to most IT service sector applications including software’s.

The manufacturing QFD was developed in Japan in the mid 60s by Yoji Akao and Shigeru Mizuno. QFD is a method to transfer customer needs into products and process requirements. First research in this field took place in the late 1970s under the guidance of professor Yoshizawa at the Research Committee of the Japanese Society of Quality Control (JSQC). In the early
1980s, the method was pushed particularly by professor A. Kanno, head of the Software Production Control (SPC) board of the Japanese Union of Scientists and Engineers (JUSE), leading to QFD being widely accepted for the first time in the Japanese software industry. Today Software-QFD is considered one of the most important techniques of product engineering in information technology in Japan.

Software quality has traditionally been defined in terms of fitness for use (Georg Herzwurm et al 1995b &Mekki& Joseph 1997). A software product is deemed fit for use if it performs to some level of satisfaction, in terms of functionality and continuous operation (William & Raja 1995). Discovery of quality problems by waiting for system failures is no longer acceptable given the critical role software plays in embedded systems and mission critical applications (Tan et al 1998). QFD is a useful quality analysis and improvement tool with many advantages. This is also the case when designing Information Technology-related products, although it has been used mainly in the manufacturing industry (Richard 2003). When using QFD, it is important to understand the needs of the customers, and then we can design or modify the product to meet their needs (Ronald 1996).

QFD is a system for translating customer requirements into appropriate technical requirements at each stage of the product-development process (Eureka 1988). QFD, the Japanese approach to new product development has been widely used to make quality and customer satisfaction more important (Glenn 1997).

Soft QFD is aimed at improving the quality of software development process by using QFD requirement refinement techniques. These techniques lead to increased analysis effort and the reduction in design changes and requirement errors passed from requirement phase to the next. Soft QFD is mainly used in the requirement phase in a software project. The
process is quite similar to that in the other industries in terms of building the House of Quality and customer requirements refinements. Several researchers have discussed the use of QFD for software products. But only a few (Georg &Sixten 2003 & William & Raja 1995) have applied QFD for software products.

Georg Herzwurm et al (1995a) explained how manufacturing QFD could be adopted to the software products and coined the terminology of “user” or “customer” or “client” with respect to SoftQFD. The application of QFD to the planning of a CASE-tool and adaptation of QFD method for software products was demonstrated by (Georg Herzwurm et al 1995b). (Glenn 1995) explored other process intensive fields such as software engineering for more systematic techniques. The major difference between manufacturing and software development with respect to QFD was discussed by William &Raja (1995).

The use of SQFD tools, namely the Affinity Diagram, the Matrix Diagram, the Matrix Data Analysis, the Analytic Hierarchy Process and Brainstorming to software products was discussed by He et al (1996).

Ronald(1996) &Yoji Akao(1997a) discussed the basics of QFD such as the voice of the Customer, Customer Importance Rating, Hows, Target Values, Relationships, Relationship Matrix, Correlation “Roof” Matrix, Organizational Difficulty, Weightage Factor, Subsystem, or Module Deployment, and Technology Deployment applicable to software products.

Georg Herzwurm et al (1997) described the results of a computer oriented evaluation of QFD Software tools. (Ita Richardson 1997) presented a model based on QFD which could be used as a tool to aid the implementation of a software process improvement action plan and discussed the use of this model in a small software development in Ireland. Advantages of using QFD
are: better focus on customers and users; an effective means of prioritizing and communicating software requirements; managing non-functional requirements and QFD as a framework for managing software requirements (Karlsson 1997).

Dave Menks et al (2000) discussed how to use QFD, provided an exercise to gain more experience with the technique, identified a SQFD case study, compared QFD with other requirements techniques, and finished with a Software QFD analysis and CASE tools summary.

According to (Zultner Richard 2000) the application of QFD to software development requires a combination of understanding users, management and software development tools. Attention must be paid to product policy and cross-functional approach to make QFD a valuable technological and organizational aid for innovations (Cor 2001).

Lai-Kow Chan & Ming-Lu Wu (2002) presented a literature review on QFD and gave a reference bank of about 650 QFD publications established through searching various sources. The origination and historical development of QFD, especially in Japan and the US, were briefly accounted first, followed by a partial list of QFD organizations, software and on-line resources. Ten informative QFD publications were also suggested, particularly for those who are not yet familiar with QFD.

Modern software development methods must cope with issues not relevant to manufacturing. Some of the issues are the use of code libraries and reuse, as well as distributed processing (Georg Herzwurm & Sixten Schockert 2003). (San Myint 2003) described the background of QFD and need for development of intelligent QFD due to uncertainty on the available human experts in product development and expansion. Table 2.5 shows the
drawbacks about the existing Software QFD models. Some of the drawbacks of the existing Software QFD models are presented in Table 2.2.

Aleksandr Kharchenko et al (2011) included the development of quality requirements based on ISO 25010, and implementation procedures of communication requirements. This allowed to realize the monitoring process as an intermediate product life-cycle and thus ensure compliance with software quality requirements. Communication requirements of the procedure developed based on the method SQFD (Software Quality Funktion Deployment) and the modified Analytic Hierarchy Process.

Lujing Yang (2011) developed ‘Four Layers Structure Model’ and ‘Life-cycle Oriented Design Model’ for PSS (product service system) design. This model provided a lifecycle oriented design methodology for PSS development and the fuzzy theory was introduced to develop HoQ, because it was difficult to evaluate the relationships between customer requirements and system engineering characteristics accurately in design stage.

Tuli Bakshi & Subir Kumar Sanyal (2011) had three phases. In the approach of first phase (i.e., pre-qualification selection), a set of alternatives was selected by the proposed fuzzy method. This method could handle qualitative and quantitative criteria. In the second phase (i.e., final selection) quality function deployment (QFD) was utilized to select the best option. QFD is a unique tool considering the relationship between customer requirement and technical requirement criteria. Lastly in the third phase (i.e., the post final testing) sensitivity analysis was derived. It ensured the robustness of the selection methodologies and the viability of the selection from the economic point of view. It was worth to apply this integrated method to compare the efficiency between project selection criteria and software engineering development characteristics.
Liberatore Matthew & Bruce Pollack-Johnson (2013) focused on the relationship between time and cost in report of software projects and considered it as an important component of project management. Its joint relationship with time and cost previously was not modeled. Using real data from two case studies, a translation agency, and a software development company, the quality function was specified and incorporated into a mathematical programming model that allowed quality to be explicitly considered in project planning and scheduling. An alternative model formulation led to the creation of quality level curves that enabled managers to evaluate the nonlinear trade-offs between quality, time, and cost for each of the example projects. The results of these analyses led to specific decisions about the planned values for these three fundamental dimensions at the task level and provided insights for project planning and scheduling that could be gained through improved understanding of the choices and trade-offs.

Divya Kashyap & Misra(2013) used PSO (Particle Swarm Optimization) to build a suitable model for software cost estimation by tuning the COCOMO parameters and also used QFD technique to establish a high correlation between customer requirements and design specification. This not only guides the software development process but helps to find the cost driver’s values also. This gives a better cost estimate and an integration of QFD technique and PSO method to develop a more precise software cost estimation model.

Eduardo Kazumi (2013) used a QFD with fuzzy logic results demonstrated to be a useful tool to evaluate the design specifications compared to customer requirements even if the relationships were verbalized on a vague linguistic way. From this work, it was possible to achieve a prioritization of energy efficiency indicators using Fuzzy QFD through the Matlab software where the relationship between the customer requirements
and energy efficiency indicators were described by linguistic variables. This is the highlight of this work concerning a data mining process based on Fuzzy-QFD to pre-process consumer’s perceptions of products and ranking the most valuable ones helping to design a decision support process.

From the literature review it was found that very few researchers had developed and applied Software QFD models to practical cases. Also the existing models have several drawbacks as listed in Table 2.5. In this study it was proposed to develop Soft QFD model for a new embedded software product and implement it in a leading software company.

2.2.2 Software Failure Mode and Effects Analysis (SWFMEA)

A review of literature on general FMEA applicable to manufacturing products was made first followed by the literature related to software products. Since the basic FMEA concepts are adapted to software products, literature related to both manufacturing and software products are presented in this section.

“The FMEA discipline was originally developed in the United States military. Military procedure MIL-P-1629, titled “Procedures for performing a failure mode, effects and criticality analysis dated Nov 9th 1949 was released. The method was used as reliability evaluation technique to determine the effect of system and equipment failures. Failures were classified according to their impact on the military mission success and personnel/equipment safety.

The definition of FMEA is “a systematic way to recognize and evaluate the potential failures of a product or process. It provides a formal mental discipline for eliminating or reducing the risks of product failure. It also serves as a living document, providing a method of organizing and tracking concerns and changes through product development and launch”.

39
The formal application of FMEA was first adopted by the aerospace industry, where FMEA had already been used during the Apollo missions in the 1960’s. Academic discussion on FMEA dates back to the 1960’s when studies of component failures were broadened to include the effects of component failure on the system of which they were a part. The generic nature of the method facilitated the rapid broadening of FMEA to different application areas and various practices.

### Table 2.1 Drawbacks about the existing SQFD models

<table>
<thead>
<tr>
<th>Sl.No.</th>
<th>The Model</th>
<th>Drawbacks</th>
</tr>
</thead>
</table>
| 1.     | The Erikkson and McFadden’s Software QFD Model (William & Raja 1995) | * Does not employ the House of Quality (HOQ)  
* Developers are not able to explicitly examine the relationships between implementation deployments or the hows listed along the x-axis of the matrix  
* There is no statement of how the customer needs will be met |
| 2.     | Zultner’s Software QFD Model (William & Raja 1995)   | * Does not use the conventional HOQ for its QFD matrices  
* Does not draw the connection between customer segments and the organizational processes |
| 3.     | Shindo’s Software QFD Model (Georg & Sixten et al 2003) | * Decomposes the customer requirements directly into functions and data and then into modules. Does not have any formal mechanism for determining the importance of the function |
| 4.     | Ohmori’s matrix-of-matrices Software QFD Model (Georg & Sixten 2003) | * A total of 14 matrices are to be implemented which is very tedious and complex |
| 5.     | The Herzwurm & Schockert’sPriFo Software QFD Model (Georg and Sixten 2003) | * Covers only the first phase i.e. product planning in the classic QFD and designing part is not dealt with |
In the early 1980s, United States automotive companies began formally to incorporate FMEA into their product development process. A task force representing Chrysler Corporation, Ford Motor Company and General Motors Corporation developed QS 9000 standard in an effort to standardize supplier quality systems. QS 9000 is the automotive analogy to better known standard ISO 9000. QS 9000 complaint automotive suppliers must utilize FMEA in the advanced quality planning process and in the development of their quality control plans. The effort made by the task force led to an industry-wide FMEA standard SAE J-1739 issued by the Society of Automotive Engineers in 1994 (Chrysler 1995).

Software FMEA is of recent origin and emerged from manufacturing FMEA. Only a few researchers have discussed the use of FMEA for Software Products.

The FMEA was applied in the analysis of software-based systems and one of the first articles regarding software failure mode and effects analysis (SWFMEA) was given by (Reifer 1979).

According to (Hall et al 1983) has to be taken into consideration. There have been attempts to generate a procedure for software FMEA. Even though there is no explicit standard for Software FMEA, the standard IEC 60812 published in 1985 is often referred when carrying out FMEA for software-based systems.

Goddard (1993) described the use of SWFMEA at Hughes Aircraft. He noted that performing the SWFMEA at early stages allowed early identification of potential failure modes. He pointed out that a static technique like FMEA could not fully assess the dynamics of control loops. (Mazur 1994) proposed the use of bi-directional analysis, an integrated extension of
Software FMEA and Software Fault Tree Analysis (FTA), as a core assessment technique by which safety critical software could be certified.

Banerjee (1995) highlighted the fact that almost all FMEA research has been directed toward manufactured products rather than information products. There is very little information published on the use of FMEA for information systems. He provided an insightful look at how teams should use FMEA in software development. He presented lessons learned in using FMEA at Isardata, a small German software company. Many potential failure modes are common to a class of software. He also pointed out that the corresponding recommended actions are also common. FMEA can improve software quality by identifying potential failure modes. He stated that FMEA could improve productivity through its prioritization of recommended actions.

Luke (1995) discussed the use of FMECA for software. He pointed out that early identification of potential failure modes could be an excellent practice in software development. It helped in the design of tests to check for the presence of the failure modes. He noted that software requirements not being met were failure modes. In FMECA, a software failure might have effects in the current module, in higher-level modules and the system as a whole. (Stamatics 1995) presented the use of FMEA with information systems. He noted that computer industry failures might result from software development process problems, coding system analysis, systems integration, software errors, and typing errors. Stamatics pointed out that failures might arise from the work of testers, developers, and managers. He also noted that a detailed FMEA analysis might examine the source code for errors in logic and loops, parameters and linkage, declarations and initializations, and syntax.

Becker & Flick (1996) gave the following classes of failure modes: Hardware or software stop, hardware or software crash, slow response, startup
failure, faulty message, checkpoint file failure, internal capacity exceeded or loss of service.

Loftus et al (1996) have highlighted the definition of Software Failure Modes, Effects and Criticality Analysis (SWFMECA), explained how it related to the engineering of software-based systems and presented an open set of pertinent issues. The entries in the FMEA worksheet were voluminous and, as a result, very brief.

SWFMEA helps to understand the early requirements, communication, and error removal. Lutz and Woodhouse noted that SWFMEA was a time consuming, tedious, manual task and depended on the domain knowledge of the analyst. Also a complete list of software failure modes could not be developed (Lutz and Woodhouse 1996). FMEA applications in the aerospace and nuclear industries have seen an exponential increase in product software content and complexity (Perkins 1996).

Ammar et al (1997) used severity measures with FMEA for risk assessment of a large-scale spacecraft software system. They noted that severity considered the worst potential consequence of a failure, whether degree of injuries or system damages. They used four severity classifications. Catastrophic failures were those that cause death or system loss. Critical failures were failures that might cause severe injury or major system damage that resulted in mission loss. Marginal failures were failures that might cause minor injury or minor system damage that resulted in delay or loss of availability of mission degradation. Minor failures were not serious enough to cause injuries or system damage but resulted in unscheduled maintenance or repair.

The integration of a forward search (SWFMEA) for consequences of reaching these forbidden modes with a backward search (FTA) for
contributing causes is necessary (Lutz & Woodhouse 1997). The use of FMEA during the development of robot control system software for a fusion reactor was studied by (Maier 1997). Maier used FMEA to examine software requirements for all possible failure modes. He stated that unforeseen external events might be eliminated by protective measures or changing the design. He recommended that the methodology he presented be applied at any early stage of the software development process to focus development and testing efforts.

Pfleeger (1998) briefly discussed the use of FMEA in software engineering. He noted that FMEA worked from unknown failure modes to unknown system effects.

Carol Smidts et al (1998) presented an approach to predict software reliability based on a systematic identification of Software Process Failure Modes and their likelihood’s. (Pries 1998) outlined a procedure for software design FMEA. Pries stated that software design FMEA should start with system or subsystem outputs listed in the item and function columns of the FMEA along with potential failure modes, effects of failure and potential causes. He noted that current design controls could include design reviews, walkthroughs, inspections, complexity analysis, and coding standards. He argued that because reliable empirical numbers for occurrence values were difficult or impossible to establish, FMEA teams could set all occurrences to a value of 5 or 10. He noted that detection numbers were highly subjective and heavily dependent on the experience of the FMEA team.

Lutz & Woodhouse (1999b) considered from failure modes namely missing data, in-correct data, timing of data and extra data for each input and each output of the software component.
Goddard (2000) has suggested two types of software FMEA for embedded control systems. The first one is system SWFMEA that can be used to evaluate the effectiveness of the software architecture and the second one detailed SWFMEA, which validates the safety requirements of the software. Goddard has argued that detailed software FMEA is often cost effective only for systems with limited hardware integrity. (Signor 2000) has conducted a very comprehensive survey of FMEA in the software industry.

The distinguishing characteristics between FMEA for hardware and software are presented in Table 2.2 (Ristord & Esmenjaud 2001).

**Table 2.2 Characteristics differences between Hardware and Software FMEA**

<table>
<thead>
<tr>
<th>Sl.No.</th>
<th>Characteristics</th>
<th>Hardware FMEA</th>
<th>Software FMEA</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Performance</td>
<td>Functional or Part Level</td>
<td>Functional Level</td>
</tr>
<tr>
<td>2.</td>
<td>Application</td>
<td>Considers system to be free from failed components</td>
<td>Considers system to be containing software faults, which may lead to failure under triggering conditions</td>
</tr>
<tr>
<td>3.</td>
<td>Failure Modes</td>
<td>Ageing, wearing or stress</td>
<td>Functional failure modes due to potential software faults</td>
</tr>
<tr>
<td>4.</td>
<td>Analysis of failure</td>
<td>At system level</td>
<td>At system level</td>
</tr>
</tbody>
</table>

They have also listed five general purpose failure modes at processing level:

(a) Operating program stops
(b) Program stops with clear message
(c) Program stops without clear message
(d) The program runs, producing obviously wrong results

(e) The program runs, producing apparently correct but in fact wrong results.

Fundamentally there are two approaches for partitioning software for a system FMEA, namely functional or by output variables (Bowles & chi 2001). (Paul Palady 2001) has outlined the software FMEA concepts in terms of model, life-cycle trend, failure mode, process, product and systems.

Herbert Hecht et al (2003) discussed the generation of the software FMEA and system design. The adoption of computer aided software FMEA provides large economic benefits because verification and validation frequently consumes the majority of the development resources for embedded software. Software diversity, Off-the-self software, Failure Mode and Effect Analysis, rigorous and formal software development are well defined (Janusz Zalewski et al 2003).

Jianyum Zhou & Tor Stalhane(2004a) proposed a general framework for conducting early robustness analysis for web based systems based on FMEA.

Hamid Reza Feili(2013) discussed a key role in the transition toward a low carbon economy and the provision of a secure supply of energy. Geothermal energy is a versatile source as a form of renewable energy that meets popular demand. Since some Geothermal Power Plants (GPPs) face various failures, the requirement of a technique for team engineering to eliminate or decrease potential failures is considerable. Because no specific published record of considering an FMEA applied to GPPs with common failure modes has been found already, the utilization of Failure Modes and Effects Analysis (FMEA) as a convenient technique for determining, classifying and analyzing common failures in typical GPPs is considered.
As a result, an appropriate risk scoring of occurrence, detection and severity of failure modes and computing the Risk Priority Number (RPN) for detecting high potential failures was achieved. In order to expedite accuracy and ability to analyze the process, XFMEA software is utilized. Moreover, 5 major parts of a GPP were studied to propose a suitable approach for developing GPPs and increasing reliability by recommending corrective actions for each failure mode.

Gee-Yong Park, Dong Hoon Kim& Dong Young Lee (2014) discussed safety-related application software. The target software system is a software code installed at an Automatic Test and Interface Processor (ATIP) in a digital reactor protection system (DRPS). For the ATIP software safety analysis, at first, an overall safety or hazard analysis is performed over the software architecture and modules, and then a detailed safety analysis based on the software FMEA (Failure Modes and Effect Analysis) method is applied to the ATIP program. For an efficient analysis, the software FMEA analysis is carried out based on the so-called failure-mode template extracted from the function blocks used in the function block diagram (FBD) for the ATIP software. The software safety analysis by the software FMEA analysis, being applied to the ATIP software code, which has been integrated and passed through a very rigorous system test procedure, is proved to be able to provide very valuable results (i.e., software defects) that could not be identified during various system tests.

Chin-Feng Fan (2013) described the operation that provided important information for constructing safer systems: To assist anomaly analysis, and develop an integrated Failure Mode and Effect Analysis (FMEA) model to analyze causal scenarios and a Three-Frame Mode model to analyze the working mode inconsistencies of failure cases. The models were used to analyze 180 digital Instrumentation and Control (I&C) failure
events from the operation of nuclear power plants. The results confirmed software engineering principles and showed that software faults and human errors were inevitable in complex systems; therefore, recovery should be emphasized and planned.

Qing-Lian Lin (2014) have highlighted failure modes and effects analysis being a risk assessment tool that mitigates potential failure in system, design, process or service. It is widely used to define, identify and eliminate known and potential failures, problems, and errors (Liu, 2011). FMEA has been used as a powerful tool for safety and reliability analysis of products and processes in a wide range of aerospace, nuclear and automotive industries (Sankar & Prabhu 2001; Wang et al 2009). In recent years, there has been more research about FMEA application in the healthcare system (Nichols, 2004). (Liu et al 2012) proposed a fuzzy FMEA based on fuzzy set theory and VIKOR to assess the risk of the general anesthesia process. (Reichert 2004) used FMEA in healthcare and described the FMEA project process. (Wetterneck 2004) described the method and challenges of performing a process and design FMEA to prepare for the implementation of a new intravenous infusion pump, and they made recommendations for the performance of a process and design FMEA for new technology implementation in healthcare organizations. (Reiling et al 2003) used FMEA to create a replacement facility aimed at reducing errors and promoting patient safety and satisfaction. FMEA is a valuable tool in designing a healthcare facility that focuses on patient safety, and it will also result in increased architect, owner and contractor awareness.

From the available literature on SWFMEA it is observed that the researchers have either discussed or described the concepts of FMEA for software projects.
2.3 SUMMARY

From the overall review of the literature related to defect removal and Software quality assurance of software projects, the following conclusions are drawn:

1. Software defect removal models clearly indicate that the usage or application of the models is very limited. The suitability of these models is not appropriately highlighted. The existing models only show the procedure and the steps involved.

   (i) The waterfall model and COQUALMO do not emphasize more on correctiveness of software testing.

   (ii) The strengths and weaknesses of the other techniques are complementary.

2. Software industry is yet to catch on to apply the standard off-line quality assurance models (QFD and FMEA) for software testing. Only a little research has been carried out in software quality improvements. All the existing research work is only in the primitive stage.

3. The literature reveals that the existing Software Quality Function Deployment models need refinement to tackle all the aspects of project development process.

4. Only concepts are presented in respect of Software Failure Mode Effects and Analysis (SWFMEA) Model, but a comprehensive SWFMEA model for software projects and its practical applications is badly needed.